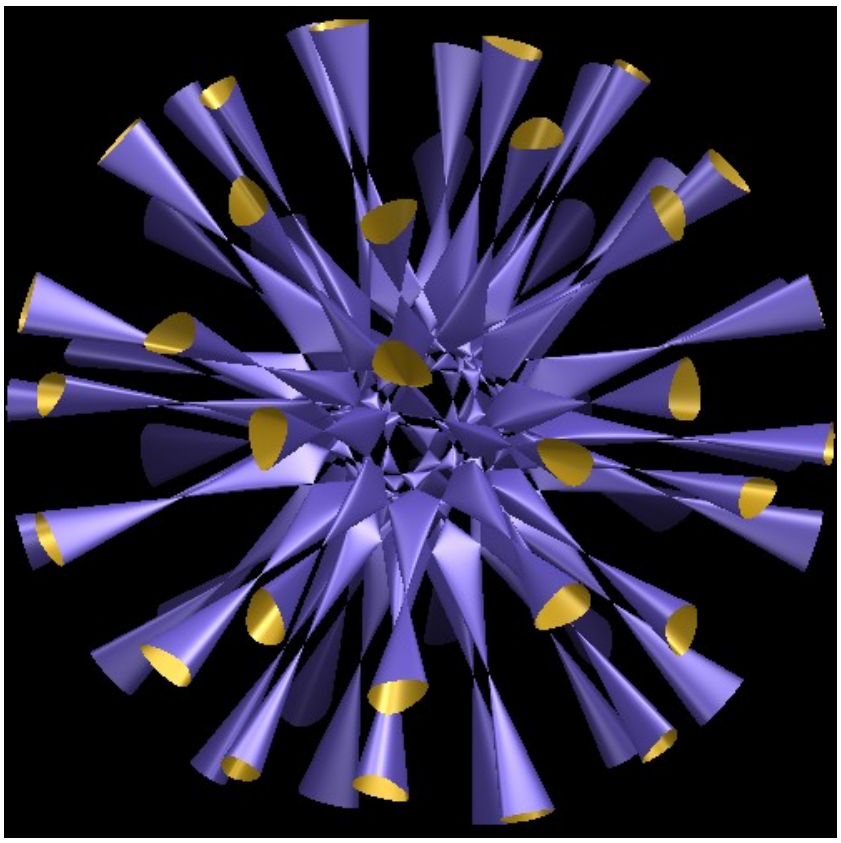
はじめての Maxima (改訂 α_{30}(今年もよろしく) 版)

横田博史

平成 25 年 1 月 5 日 (土)



MATLAB は The MathWorks の登録商標です.
Maple は Waterloo Maple Inc. の登録商標です.
Mathematica は Wolfram Research Inc. の登録商標です.
VMWare は WMWare Inc. の登録商標です.
はじめての Maxima(改訂 α 版)©(2012) 横田 博史著
この文書の内容の誤りなどによって起こった損害に対し, 工学社, KNOPPIX/Math-Project のメンバー, および, 著者の私は一切の責任を負いません. なお, 質問や意見があれば直接, 著者に連絡して下さい.
この文書の二次配布を行う場合は著者に連絡した上てお願いします.
連絡先:ponpoko@cap.bekkoame.ne.jp

まえがき

数式処理システム Maxima は 1960 年代から 70 年代にかけて MIT で開発された数式処理システム MACSYMA をテキサス大学の Schelter 氏が Common Lisp 上に移植し, GPL 下で配布したソフトウエアです. 商用の *Mathematica* や Maple 等と比較して古色蒼然とした面も否定出来ませんが, 強力で魅力的な汎用の数式処理ソフトウエアです.

この本は Maxima の入門編, マニュアル編, 応用編と Octave による数値行列処理, そして, Maxima のための環境構築と Maxima のインストールに分かれています. さらに Maxima を理解する上で最低限必要と思われる LISP のことと数学上の概念に関しても簡単な説明を加えています.

マニュアルは Schelter 氏か記述した Maxima-5.6 に附属のマニュアルを参考にして, Maxima-5.15 以降に対応するように修正を加えています. ただし, Maxima のパッケージ全てを網羅するものではないために, この点は今後改善してゆく予定です.

さて, この文書は書籍の「はじめての Maxima」を基に修正と調査の結果から新たに判ったことや理解したことを色々と追加していますが, 文書自体は正規のものではない α 版です. そして, 未完成のまま, あるいは間違ったままの個所も数多く存在しますが, それらの問題点を除外しても公開すべき多くの長所を持っていると自負しています.

ここで「はじめての Maxima」の改訂版が実際に出版されるかどうかは既刊の本の売行きと, この文書の出来に大きく依存するために全く将来は不明瞭で, 仮に出版されるにしても, この文書のままということは本文書の性格上から有り得ないでしょう. 実際, A4 版で千頁を軽く越えており, 物理的にも一冊の本として出版されることは難しいでしょう. しかし, 確かなことは正規の改訂版の原稿を目指し, α 版から β 版, そして正規版へと更新作業が今後も続くことです. それはひょっとすると永久的なことかもしれませんし, どうやら最近はそれを楽しみながら, ながながと駄文を書き散らかしている有様です.

何か人質にしているようで申し分けないことですが, 何れにせよ, この文書をより良くする時間は十分にある訳で, 皆様のご協力を謹んで請う次第です.

平成 25 年 1 月 5 日 (土)　　狸穴主人 横田博史

目 次

第 1 章　この本の趣向について **1**
1.1 想定読者について 2
1.2 数式処理って何？ 5
1.2.1 数式処理？ 5
1.2.2 Prolog を使った機械的処理の例 5
1.2.3 安易な機械的処理の陥穽 6
1.2.4 数式処理を記述する言語 7

第 2 章　ちょっとした計算例 **9**
2.1 Maxima のユーザーインターフェイス 10
2.2 入力 13
2.3 演算子 16
2.4 式の評価 20
2.5 数値計算 23
2.6 式の微分・積分 25
2.7 方程式の解 27
2.8 行列 28
2.9 FORTRAN や TEX への出力 31
2.10 グラフ表示 32
2.10.1 gnuplot による Klein の壷 35
2.10.2 openmath による Klein の壷 35
2.10.3 Geomview による Klein の壷 36
2.10.4 番外編 36
2.11 ファイル 37

第 3 章　LISP について **41**
3.1 背景 42
3.2 数値, 文字列 43
3.2.1 LISP の代表的な数値函数 45
3.3 リスト 47
3.4 t と nil 48
3.5 配列 49
3.6 ハッシュ表 50

3.7 割当と評価 50
3.8 構造体 51
3.9 写像函数 52
3.10 lambda 式 52
3.11 函数の定義 53
3.12 制御文 53
3.13 属性 54
3.14 入出力 55

第 4 章 数学のいろいろなこと **59**
4.1 集合について 60
4.2 同値関係について 63
4.2.1 数式処理システムとオブジェクト指向 63
4.2.2 等しいということの考察 63
4.2.3 分数と同値関係 64
4.2.4 きちんと定義できていることの検証 65
4.2.5 $\mathbb{R}/(x^2+1)$ 67
4.2.6 ちょっとしたまとめ 69
4.3 群について 69
4.3.1 群の例 71
4.4 環について 73
4.4.1 イデアル 74
4.4.2 環の例 75
4.5 体について 77
4.6 準同型写像について 78
4.7 数式の表現 79
4.8 順序について 82
4.8.1 色々な順序 83
4.9 多項式の表現 85
4.10 Gröbner 基底の紹介 87
4.11 代数的数と超越数 89
4.12 濃度/基数 90
4.12.1 等数性 90
4.12.2 可附番集合 92
4.12.3 自然数の濃度 92
4.12.4 実数の濃度 93
4.13 自然数 94
4.13.1 自然数の基礎付けについて 94
4.13.2 Leibniz による $2+2=4$ の証明 94
4.13.3 自然数の後者 95

4.13.4 Peano の公理系 . . . 95
4.13.5 原始帰納的函数 . . . 96
4.13.6 原始帰納的函数の例 . . . 98
4.13.7 一般帰納的函数 . . . 99
4.13.8 自然数の性質 . . . 99
4.14 整数 . . . 100
4.15 実数 . . . 101
4.15.1 古代ギリシャ . . . 101
4.15.2 基本列を用いた実数の創造 . . . 105
4.15.3 Dedekind の切断 . . . 106
4.15.4 順序集合の連続性 . . . 107
4.15.5 切断による実数の創造 . . . 107
4.15.6 実数の公理系 . . . 109
4.16 超限順序数 . . . 110
4.16.1 超限順序数 . . . 112
4.17 数学の厳密化 . . . 113
4.17.1 逆理の分類 . . . 115
4.17.2 19 世紀の数学の算術化との関連 . . . 116
4.17.3 Poincaré による逆理の分析 . . . 117
4.17.4 三つの道 . . . 118
4.17.5 Hilbert 計画 . . . 122
4.18 命題と述語 . . . 123
4.18.1 小史 . . . 123
4.18.2 伝統的論理学について . . . 128
4.18.3 Leibniz . . . 131
4.18.4 19 世紀の論理学の進展 . . . 132
4.19 Frege の概念記法 . . . 135
4.19.1 概念記法 (Begriffsschrift) の概要 . . . 135
4.19.2 論理学の刷新 . . . 140
4.19.3 Frege の自然数の構成 . . . 153
4.19.4 破綻 . . . 163
4.20 Russell の階型理論 . . . 166
4.20.1 項, 概念, 個体 . . . 167
4.20.2 変項と函数 . . . 167
4.20.3 PM の論理式 . . . 167
4.20.4 基本命題と明瞭な変項 . . . 168
4.20.5 命題と函数の階層 . . . 169
4.20.6 悪循環原理による非可述的述語の排除 . . . 171
4.20.7 還元可能性公理 . . . 172
4.20.8 クラスについて . . . 172

4.20.9 PM の公理系 . . . 174
4.21 Hilbert による形式化 . . . 175
4.21.1 Hilbert の立場 . . . 175
4.21.2 項 . . . 175
4.21.3 論理記号 . . . 176
4.21.4 論理式 . . . 178
4.21.5 恒真な論理式 . . . 179
4.21.6 導出 . . . 179
4.21.7 公理系 . . . 181
4.21.8 Russell との違い . . . 184
4.22 集合論の公理化 . . . 185
4.22.1 ZFC-公理系 . . . 185
4.22.2 選択公理について . . . 187
4.23 論理式の代表的な操作 . . . 188
4.23.1 Skolem の標準形について . . . 188
4.23.2 Davis-Putnam の手続 . . . 189
4.23.3 Herbrand の定理 . . . 190
4.23.4 λ 計算 . . . 190
4.24 Gödel の不完全性定理 . . . 190
4.24.1 背景 . . . 190
4.24.2 体系 P . . . 191
4.24.3 Gödel 数 . . . 193
4.24.4 述語の算術化 . . . 193
4.24.5 決定不能な命題 . . . 194
4.24.6 数学基礎論の勝者は？ . . . 194
4.25 そして計算機 . . . 194
4.25.1 数学の現代化 . . . 194
4.25.2 LISP と MACSYMA へ . . . 195

第 5 章 Maxima の処理原理について **197**
5.1 Maxima の基礎概念 . . . 198
5.1.1 Maxima の原子 . . . 198
5.1.2 Maxima の記号 (symbol) . . . 199
5.1.3 Maxima の文字列 . . . 201
5.1.4 整数の表現 . . . 201
5.1.5 実数の表現 . . . 202
5.1.6 真理値の表現 . . . 202
5.1.7 有理数と複素数の表現 . . . 203
5.1.8 Maxima の変数 . . . 203
5.1.9 Maxima の函数と演算子 . . . 204

5.1.10 マクロ . . . 206
5.1.11 配列 . . . 206
5.1.12 リスト . . . 206
5.1.13 属性 . . . 207
5.1.14 Maxima の式 . . . 208
5.1.15 大域変数 . . . 209
5.1.16 Maxima の論理式 . . . 209
5.1.17 文脈 . . . 210
5.1.18 規則 . . . 211
5.1.19 式の自動簡易化 . . . 211
5.1.20 まとめ . . . 212
5.2 順序 . . . 213
5.2.1 Maxima の変数順序 . . . 213
5.2.2 項式項に対する順序 . . . 214
5.2.3 局所的な順序の変更 . . . 215
5.2.4 順序に関連する函数 . . . 216
5.2.5 函数を含めた順序 . . . 217
5.3 演算子 . . . 219
5.3.1 Maxima の演算子の属性 . . . 219
5.3.2 演算子の束縛力 . . . 220
5.3.3 演算子と被演算子の型 . . . 222
5.3.4 演算子の属性を宣言する函数 . . . 224
5.3.5 演算子属性の削除 . . . 228
5.3.6 算術演算子 . . . 229
5.3.7 論理演算子 . . . 232
5.3.8 割当の演算子 . . . 233
5.3.9 その他の演算子 . . . 234
5.3.10 演算子に関連する函数 . . . 235
5.4 属性 . . . 237
5.4.1 Maxima の属性 . . . 237
5.4.2 属性と設定函数 . . . 237
5.4.3 put 函数と qput 函数による属性指定 . . . 237
5.4.4 declare 函数 について . . . 239
5.4.5 declare 函数で付与可能な属性 . . . 240
5.4.6 利用者による属性の追加 . . . 241
5.4.7 属性の表現函数 . . . 242
5.4.8 declare 函数に用意された属性 . . . 243
5.4.9 declare 函数以外の函数による函数属性の付加 . . . 253
5.4.10 depends 函数と gradef 函数 . . . 256
5.4.11 属性を削除する函数 . . . 258

5.4.12 属性の表示 261
5.5 論理式 264
5.5.1 Maxima の論理式について 264
5.5.2 論理式の判断 264
5.5.3 量化詞を表現する函数 266
5.5.4 同値性と非同値性の表現 267
5.5.5 論理式を評価する函数 269
5.5.6 Maxima の真理函数 271
5.5.7 引数が一つの真理函数 274
5.5.8 その他の函数 275
5.6 文脈 277
5.6.1 文脈の概要 277
5.6.2 文脈に登録可能な論理式 279
5.6.3 論理式の文脈への登録 279
5.6.4 文脈内部での属性と論理式の表現 281
5.6.5 文脈を用いた推論 283
5.6.6 文脈の階層 286
5.6.7 文脈の指定に関連する大域変数 290
5.6.8 変数の正値性に関連する大域変数 291
5.7 規則と式の並びについて 293
5.7.1 規則の概要 293
5.7.2 述語と変換函数の定義 293
5.7.3 述語と変数の指定 295
5.7.4 並びの指定に関連する函数 297
5.7.5 defrule 函数による規則 298
5.7.6 defrule を用いた微分作用素 302
5.7.7 tellsimp 函数と tellsimpafter 函数による規則の定義 303
5.7.8 let 函数による規則 305
5.7.9 規則の削除 308
5.8 式の評価 311
5.8.1 Maxima での式の評価について 311
5.8.2 式の自動簡易化 311
5.8.3 ev 函数 312
5.8.4 ev 函数の引数について 314
5.8.5 評価に関連する函数 321
5.8.6 函数や演算子に影響を与える大域変数を表示する函数 322
5.9 LISP に関係する函数 324
5.9.1 Maxima と LISP 324
5.9.2 Maxima から LISP の利用 324
5.9.3 LISP から Maxima の函数を利用 326

第 6 章　Maxima の対象とその操作　327
6.1　数値 . . . 328
6.1.1　Maxima で扱える数値について . . . 328
6.1.2　四則演算について . . . 330
6.1.3　数値に関連する大域変数 . . . 331
6.1.4　Maxima の数学定数 . . . 332
6.1.5　数に関連する真理函数 . . . 333
6.1.6　整数値函数 . . . 333
6.1.7　一般の数値函数 . . . 335
6.1.8　疑似乱数に関する函数 . . . 336
6.1.9　複素数に関連する函数 . . . 337
6.1.10　LISP 由来の数値函数 . . . 337
6.2　多項式 . . . 339
6.2.1　多項式の一般表現 . . . 339
6.2.2　多項式の CRE 表現 . . . 341
6.2.3　係数体について . . . 342
6.2.4　多項式に関する函数 . . . 343
6.2.5　有理式に関連する函数 . . . 367
6.2.6　その他の函数 . . . 369
6.3　級数の扱い . . . 371
6.3.1　Maxima に於ける級数の表現 . . . 371
6.3.2　Taylor 級数の内部表現 . . . 371
6.3.3　taylor 函数 . . . 372
6.3.4　Taylor 級数に関連する函数 . . . 373
6.3.5　Taylor 級数に関連する大域変数 . . . 373
6.4　式について . . . 374
6.4.1　変数や文字列の内部表現 . . . 374
6.4.2　二項演算の内部表現 . . . 374
6.4.3　割当の演算子の内部表現 . . . 376
6.4.4　Maxima の函数の内部表現 . . . 376
6.4.5　配列とリストの内部表現 . . . 378
6.4.6　Maxima の制御文の内部表現 . . . 378
6.4.7　表示式と内部表現 . . . 379
6.4.8　変数と変数項 . . . 381
6.4.9　部分式に分解する函数 . . . 383
6.4.10　部分式を扱う函数 . . . 386
6.4.11　総和と積 . . . 388
6.4.12　式の様々な操作を行う函数 . . . 394
6.4.13　TeX や FORTRAN の書式に式の変換を行う函数 . . . 395
6.4.14　FORTRAN の書式に変換 . . . 397

6.5 リスト 400
6.5.1 Maxima のリスト 400
6.5.2 リストの生成を行う函数 400
6.5.3 リスト処理に関連する大域変数 401
6.5.4 リスト処理に関連する主な函数 402
6.5.5 map 函数族 407
6.5.6 map 函数族に関連する大域変数 408
6.5.7 map 函数いろいろ 408
6.5.8 apply 函数 411
6.5.9 リストを使った四則演算 411
6.6 集合について 413
6.6.1 概要 413
6.6.2 集合の生成に関連する函数 413
6.6.3 リスト操作の函数 416
6.6.4 集合演算の函数 416
6.6.5 集合操作の函数 417
6.6.6 集合に関連する函数 419
6.6.7 分割に関連する函数 421
6.6.8 集合に関連する真理値函数 423
6.7 配列 424
6.7.1 Maxima の配列について 424
6.7.2 配列操作に関連する函数 427
6.8 行列 429
6.8.1 行列の内部表現 429
6.8.2 行列を生成する函数 430
6.8.3 行列の操作函数 435
6.8.4 行, 列, 及び成分の操作函数 435
6.8.5 転置, 上三角, 共役行列を計算する函数 437
6.8.6 行列式に関連する函数 439
6.8.7 行列の四則演算 442
6.8.8 行列演算に関連する大域変数 443
6.8.9 eigen パッケージ 445
6.9 LAPACK パッケージの利用 449
6.9.1 BLAS と LAPACK について 449
6.9.2 BLAS について 449
6.9.3 ベクトルと行列の表記について 450
6.9.4 配列への格納方法 451
6.9.5 ルーチンの命名規則 453
6.9.6 ルーチンの引数について 455
6.9.7 BLAS のルーチン 456

6.9.8 BLAS の第 0 水準のルーチン . . . 457
6.9.9 BLAS の第一水準のルーチン . . . 458
6.9.10 BLAS の第二水準のルーチン . . . 465
6.9.11 LAPACK の構成 . . . 475
6.9.12 Maxima の函数 . . . 477
6.9.13 LAPACK パッケージの利用方法 . . . 478
6.9.14 橋渡しを行う函数 . . . 479
6.10 文字列 . . . 481
6.10.1 Maxima の文字列 . . . 481
6.10.2 ストリーム処理に関連する函数 . . . 483
6.10.3 stringproc パッケージの真理函数 . . . 487
6.10.4 文字列変換の函数 . . . 489
6.10.5 文字列操作の函数 . . . 490
6.10.6 真理函数を利用する文字列操作の函数 . . . 494
6.10.7 関連する大域変数 . . . 498
6.11 構造体 . . . 500
6.11.1 関連する函数と大域変数 . . . 500
6.11.2 構造体の例 . . . 500
6.12 ラベル . . . 502
6.12.1 ラベルの概要 . . . 502
6.12.2 ラベルに関連する大域変数 . . . 503
6.12.3 ラベル処理の函数 . . . 505
6.13 Maxima の対象 . . . 507
6.13.1 Maxima の対象とその実体 . . . 507
6.13.2 対象の削除 . . . 509

第 7 章 式の操作 511
7.1 代入操作 . . . 512
7.1.1 通常の代入函数 . . . 512
7.1.2 式の内部構造を考慮した代入函数 . . . 514
7.2 式の展開と簡易化 . . . 517
7.2.1 自動展開を行う大域変数 . . . 517
7.2.2 指数函数の展開に関連する函数 . . . 520
7.2.3 式の展開に関連する函数 . . . 520
7.2.4 演算子の分配に関連する函数 . . . 522
7.2.5 distrib 函数,multthru 函数,expand 函数の比較 . . . 523
7.2.6 sum 函数の簡易化に関連する函数 . . . 523
7.2.7 簡易化を行う函数 . . . 524
7.2.8 簡易化に関する補助的函数 . . . 524
7.2.9 共通の項で纏める函数 . . . 524

7.3 代数方程式 . . . 526
7.3.1 Maxima での方程式とその解法について . . . 526
7.3.2 1 変数多項式方程式の場合 . . . 530
7.3.3 一般の多項式方程式の場合 . . . 531
7.3.4 漸化式の場合 . . . 538
7.4 極限 . . . 539
7.4.1 極限について . . . 539
7.4.2 limit 函数 . . . 541
7.4.3 tlimit 函数: . . . 541
7.4.4 極限に関連する大域変数 . . . 542
7.5 微分 . . . 543
7.5.1 微分に関係する函数 . . . 543
7.5.2 vect パッケージ . . . 545
7.6 積分 . . . 549
7.6.1 記号積分について . . . 549
7.6.2 integrate 函数と risch 函数 . . . 549
7.6.3 integrate 函数と risch 函数に関連する大域変数 . . . 551
7.6.4 changevar 函数による変数変換 . . . 551
7.6.5 有理式の記号積分 . . . 553
7.6.6 記号積分の検証について . . . 554
7.6.7 defint 函数 . . . 556
7.6.8 Laplace 変換に関連する函数 . . . 557
7.6.9 その他の積分に関連する函数 . . . 559
7.6.10 定積分を行う函数 . . . 559
7.6.11 数値積分について . . . 561
7.6.12 antid パッケージ . . . 564
7.7 常微分方程式 . . . 565
7.7.1 常微分方程式の書式 . . . 565
7.7.2 常微分方程式の解法 . . . 565
7.7.3 常微分方程式の一般解を求める函数 . . . 567

第 8 章 プログラム **571**
8.1 Maxima でプログラム . . . 572
8.1.1 block 文 . . . 572
8.1.2 block 文内部で利用可能な函数 . . . 573
8.1.3 if 文 . . . 573
8.1.4 do 文による反復処理 . . . 574
8.1.5 エラー処理 . . . 576
8.1.6 プログラムに関連する大域変数 . . . 578
8.2 函数とマクロの定義 . . . 579

8.2.1 函数とマクロについて . . . 579
8.2.2 函数の定義 . . . 579
8.2.3 函数定義に関連する大域変数 . . . 582
8.2.4 函数定義に関連する函数 . . . 583
8.2.5 マクロの定義 . . . 584
8.2.6 マクロの展開に関連する函数 . . . 587
8.2.7 マクロに関連する大域変数 . . . 588
8.2.8 利用者定義函数とマクロの確認 . . . 589
8.2.9 利用者定義函数とマクロの削除 . . . 589
8.3 自動的に読込まれる函数 . . . 590
8.4 式と函数の最適化 . . . 592
8.4.1 最適化について . . . 592
8.4.2 式の最適化 . . . 592
8.4.3 LISP の函数に変換する函数 . . . 593
8.4.4 変数型指定に関連する函数 . . . 599
8.4.5 型の検証に関連する大域変数 . . . 602

第 9 章 Maxima で扱う数学的対象 605
9.1 数論に関連する函数 . . . 606
9.1.1 階乗 . . . 606
9.1.2 剰余 . . . 607
9.1.3 Bell 数 . . . 608
9.1.4 Bernoulli 数 . . . 608
9.1.5 *B*(beta) 函数 . . . 610
9.1.6 二項係数 . . . 612
9.1.7 Euler 数 . . . 613
9.1.8 Fibonacci 数 . . . 613
9.1.9 Γ 函数 . . . 614
9.1.10 多重対数函数 . . . 616
9.1.11 Möbius の函数 μ . . . 617
9.1.12 numfactor 函数 . . . 618
9.1.13 digamma(polygamma) 函数 ψ . . . 618
9.1.14 ζ 函数 . . . 619
9.1.15 連分数に関連する函数 . . . 621
9.1.16 二次体に関連する函数 . . . 622
9.1.17 ifactor パッケージに含まれる函数 . . . 623
9.1.18 ifactor パッケージに含まれる大域変数 . . . 626
9.1.19 numth パッケージ . . . 628
9.1.20 Kronecker の δ と Stiring 数 . . . 630
9.2 三角函数 . . . 632

9.2.1 三角函数一覧 . . . 632
9.2.2 三角函数に関連する函数 . . . 635
9.2.3 atrig1 パッケージ . . . 637
9.2.4 trgsmp パッケージ . . . 637
9.3 指数函数と対数函数 . . . 639
9.3.1 指数函数と対数函数の概要 . . . 639
9.3.2 対数函数に関連する函数 . . . 641
9.4 超幾何微分方程式 . . . 645
9.4.1 Airy 函数 . . . 646
9.4.2 Bessel 函数 . . . 648
9.4.3 Hankel 函数 . . . 649
9.5 hypgeo パッケージ . . . 650
9.6 orthopoly パッケージ . . . 650
9.6.1 Chebyshev 多項式 . . . 650
9.6.2 Hermite 多項式 . . . 650
9.6.3 超球多項式 . . . 651
9.6.4 Jacobi 多項式 . . . 651
9.6.5 Laguerre の多項式 . . . 651
9.6.6 Legendre の多項式 . . . 651
9.7 楕円函数 . . . 652
9.7.1 楕円積分の概要 . . . 652
9.7.2 楕円積分が満す関係式 . . . 653
9.7.3 Maxima での楕円積分 . . . 654
9.7.4 Jacobi の楕円函数 . . . 656
9.7.5 Maxima での Jacobi の楕円函数 . . . 658
9.7.6 ζ 函数に関連する函数 . . . 661
9.8 力学系と dynamics パッケージ . . . 663
9.8.1 力学系について . . . 663
9.8.2 fractal を生成する代表的な函数系について . . . 669
9.8.3 dynamics パッケージの概要 . . . 672
9.8.4 dynamics.mac に含まれる函数 . . . 672
9.8.5 complex_dynamics.lisp ファイルに含まれる函数 . . . 676
9.8.6 二次の力学系を対話的に描画する函数 . . . 678
9.8.7 幾つかの例題 . . . 682

第 10 章 Maxima のシステム関連の函数 687
10.1 計算結果の初期化 . . . 688
10.2 処理の中断 . . . 689
10.2.1 制御文字による中断 . . . 689
10.2.2 函数による意図的な中断 . . . 689

10.3 結果の表示 690
10.3.1 表示に関連する大域変数 691
10.3.2 式の表示を行う函数 693
10.3.3 エラー表示 698
10.4 ヘルプに関連する函数 699
10.5 システムの状態を調べる 703
10.5.1 status 函数と sstatus 函数 703
10.5.2 room 函数 705
10.6 時間に関連する函数 705
10.6.1 処理時間に関連する函数 705
10.6.2 システムの時間を返す函数 706
10.6.3 timer 函数,untimer 函数と timer_info 函数 707
10.7 便利な函数 707
10.8 外部プログラムの起動 708
10.9 Maxima の終了 708
10.10ファイル操作について 710
10.10.1 ファイルを使った入出力 710
10.10.2 Maxima のファイル検出方法 710
10.10.3 ファイル検出に関連する函数 712
10.10.4 バッチ処理に関連する函数 713
10.10.5 ファイルの読込を行う函数 714
10.10.6 ファイルに書込みを行う函数 716
10.10.7 その他のファイルに関連する函数 719
10.10.8 maxima-init.mac ファイル 720
10.10.9 maxima-init.mac ファイルの設置場所 721
10.11虫取りに関連する函数 722
10.11.1 動作追跡に関連する函数 722
10.11.2 bug_report 函数と build_info 函数 724
10.11.3 関連する大域変数 725
10.11.4 LISP の trace 函数との併用 726
10.11.5 システムの検証計算を行う函数 727

第 11 章 Maxima でグラフ表示 **729**
11.1 Maxima のグラフ表示 730
11.1.1 はじめに 730
11.1.2 plot2d と plot3d による描画の概要 730
11.1.3 plot2d と plot3d で利用可能な外部アプリケーションについて 730
11.1.4 与件ファイルについて 731
11.1.5 plot2d 函数 733
11.1.6 plot3d 函数 736

11.2 大域変数 plot_options 737
11.2.1 大域変数 plot_options 概要 737
11.2.2 大域変数 plot_options の設定に関連する函数 737
11.2.3 外部アプリケーションの設定に関連する項目 738
11.2.4 表示領域に関する項目 739
11.2.5 描画に直接関連するフラグ 740
11.2.6 gnuplot に関連する項目 741
11.2.7 gnuplot の描画に直接関連する項目 745
11.2.8 gnuplot との連動に関連する函数 747
11.3 その他の描画函数 747
11.3.1 openplot_curves 747
11.3.2 contour_plot 750
11.3.3 Postscript に関連する函数 750
11.4 plot_option 以外の描画に関連する大域変数 751
11.5 gnuplot による描画について 753
11.5.1 maxout.gnuplot の内容について 753
11.5.2 set 命令 754
11.5.3 plot 命令による曲線の表示 754
11.5.4 splot 命令による曲面の表示 757
11.5.5 pm3d 757
11.5.6 ticslevel による射影平面の調整 760
11.5.7 等高線の階調の変更 760
11.5.8 等高線の表示 762
11.5.9 contour と pm3d の共存 764
11.5.10 等高線の間隔調整 765
11.5.11 視点の変更 766
11.5.12 cbrange と cblabel 767
11.5.13 陰線処理 769
11.5.14 Maxima のグラフと gnuplot のグラフの比較 770
11.5.15 曲線と曲面の細かさの指定 772
11.5.16 描画の領域設定 773
11.5.17 ラベル表示と注釈に関連する事項 775
11.5.18 gnuplot の式 780
11.5.19 電卓としての gnuplot 784
11.5.20 プログラム言語としての gnuplot 785
11.6 plot2d 函数と plot3d 函数の活用事例 788
11.6.1 gnuplot_preamble の使い方 788
11.6.2 Maxima のバッチ処理 790
11.7 draw パッケージ 795
11.7.1 draw パッケージの概要 795

11.7.2 外部アプリケーションの切替について 795
11.7.3 対象とその属性について 796
11.7.4 draw 函数と draw2d 函数,draw3d 函数との関係 799
11.7.5 draw 函数 800
11.7.6 属性と draw 函数 801
11.7.7 大域的属性について 804
11.7.8 対象について 809
11.7.9 局所的属性について 810
11.7.10 対象について 840
11.7.11 その他の函数 871
11.7.12 大域変数について 871
11.8 drawutils パッケージ 873
11.8.1 picture パッケージ 874
11.9 世界地図 (worldmap パッケージ) 879
11.9.1 世界地図の概要 879
11.9.2 worldmap パッケージの函数 879

第 12 章 積分函数の動きを観察しよう **885**
12.1 積分函数ツアー募集要項 886
12.2 Maxima のソースファイル 886
12.3 integrate 函数 886
12.4 integrate_use_rootsof を用いた積分 892

第 13 章 Maxima の簡単な改造 **895**
13.1 使い勝手の向上を目指して 895
13.2 describe 函数は何処にある？ 895
13.3 describe 函数の動作 896
13.4 函数 ponpoko の仕様 897
13.5 大域変数の作り方 898
13.6 判別について 899
13.7 外部アプリケーションの立ち上げ方 900
13.8 完成 900
13.8.1 MS-Windows 環境の場合 902
13.9 大域変数の変更について 903

第 14 章 結び目の Alexander 多項式 **905**
14.1 結び目の概要 905
14.2 結び目の射影図 909
14.3 結び目群の Wirtinger 表示 910
14.4 群環と Fox の微分作用素 913
14.5 Maxima で遊ぶ Fox の微分子 914

14.5.1 函数 t の表現 . . . 914
14.5.2 群環 $\mathbb{Z}F$ の表現 . . . 915
14.5.3 真理函数 wordp の構成 . . . 915
14.5.4 Fox の微分子の構成 . . . 917
14.6 プログラムファイルの構成 . . . 922
14.7 Alexander 多項式 . . . 923
14.7.1 Alexander 行列の計算 . . . 923
14.7.2 Alexander 多項式の計算 . . . 924
14.8 Alexander 多項式で結び目を分類しよう . . . 927
14.9 結び目の連結和と Alexander 多項式 . . . 930
14.10結び目の鏡像と Alexander 多項式 . . . 932
14.11Alexander 多項式の意味 . . . 933
14.12おまけ：スケイン多項式 . . . 947

第 15 章 surf を使う話 **951**
15.1 代数曲面 . . . 951
15.2 surf/surfer の概要 . . . 951
15.3 Maxima から surf/surfer を使う方法 . . . 952
15.4 surf の設定と Maxima への取込み方法 . . . 953
15.4.1 共通設定 . . . 953
15.4.2 曲面固有の設定 . . . 953
15.4.3 助変数の Maxima での表現方法 . . . 954
15.5 多項式の処理 . . . 956
15.6 曲線と曲面の描画の違いについて . . . 956
15.7 surfplot.mc . . . 959
15.7.1 surfplot.mc の使い方 . . . 962
15.8 簡単な例 . . . 962
15.9 Barth Diec . . . 964
15.10Steiner のローマ曲面 . . . 965
15.10.1 曲面の概要 . . . 965
15.10.2 Steiner のローマ曲面の描画 . . . 967
15.10.3 頂点の計算 . . . 970
15.10.4 surf による断面の描画 . . . 970
15.11SINGULAR を使ってみよう . . . 972
15.11.1 SINGULAR 初歩 . . . 973
15.11.2 SINGULAR で surf を使ってみよう . . . 975
15.12Maxima で終結式を使ってみよう . . . 976
15.13デモファイルでも書いてみよう . . . 980
15.14例題ファイルもついでに… . . . 983

第 16 章 MATLAB 風言語で遊ぶ話 **987**
16.1 数値計算を主体にした環境 . . . 987
16.2 MATLAB と MATLAB 風言語 . . . 987
16.3 オンラインヘルプ . . . 988
16.4 基本的な対象 . . . 989
16.4.1 数値 . . . 989
16.4.2 論理値 . . . 990
16.4.3 ベクトルと行列 . . . 990
16.4.4 文字列 . . . 990
16.5 基本的な計算式の入力と値の代入 . . . 992
16.5.1 数学定数 . . . 992
16.6 行列処理 . . . 994
16.6.1 ベクトルと行列の書式 . . . 994
16.6.2 行列の大きさを返す命令 . . . 995
16.6.3 ベクトルと行列の成分の取出し . . . 996
16.7 MATLAB 系言語での演算 . . . 999
16.7.1 四則演算を含む基本的な演算 . . . 999
16.7.2 演算の処理速度の比較 . . . 1002
16.8 並びの照合 . . . 1003
16.8.1 for 文と並びの照合の処理速度の比較 . . . 1006
16.8.2 any と all . . . 1007
16.9 便利な行列の定義方法 . . . 1007
16.9.1 等間隔のベクトルの生成 . . . 1007
16.9.2 対角行列の生成 . . . 1008
16.10多項式の扱い . . . 1009
16.11M-file . . . 1010
16.12外部アプリケーションの起動命令 . . . 1012
16.13グラフ表示機能 . . . 1014
16.14Octave で file を利用する話 . . . 1017
16.14.1 load 命令によるデータファイルの処理 . . . 1017
16.14.2 save 命令による行列の保存 . . . 1018
16.14.3 ファイルの Open と Close . . . 1021
16.14.4 データの読み込み . . . 1021
16.14.5 ファイルの更新とデータの追加の例 . . . 1027
16.15Maxima との簡単なインターフェイス作製 . . . 1028

第 17 章 Maxima を動作させる環境について **1031**
17.1 道標 . . . 1031
17.2 Maxima の初期設定 . . . 1032
17.3 我慢強い方 . . . 1032

17.3.1 Common Lisp の選択 1032
17.3.2 コンパイルの手順 1033
17.4 MS-Windows 環境への Maxima のインストール 1034
17.4.1 Maxima のインストール 1034
17.4.2 起動時の注意 1040
17.4.3 環境変数 Path の設定 1041

第 18 章 KNOPPIX/Math 2010 の活用 **1043**
18.1 はじめに 1043
18.2 仮想計算機環境について 1043
18.3 VirtualBox で KNOPPIX を利用する場合 1044
18.3.1 VirtualBox の概要 1044
18.4 設定方法 1044
18.5 VMware Player で KNOPPIX を利用する場合 1051
18.5.1 VMware Player について 1051
18.5.2 設定方法 1051
18.6 仮想計算機と既存環境との共存 1055
18.7 KNOPPIX/Math2010 の使い方 1056
18.7.1 Flash memory へのインストール 1056
18.7.2 KNOPPIX-Math-Start 1057
18.7.3 JDML 1058
18.7.4 KNOPPIX/Math 上での全文検索 1059

第 19 章 最後に **1061**

第1章　この本の趣向について

Maxima と聞いて一体何を意味するのか全く判らない方. 知ってはいても何なのかまるで見当のつかない方. そして, この本を書店で見つけてひやかし半分で眺めている貴方向けに, ちょっとした性格判断と, その傾向と対策について.

1.1　想定読者について

はじめに FAQ の形式で Maxima がどのようなもので, この本がどのような癖を持っているか簡単に説明しておきましょう.

Maximaって何？ 美味しいもの？

Maxima は汎用の数式処理システムです. つまり数式そのものを計算機に入力すれば, あとは計算機が貴方の代わりに計算や, おまけに曲線や曲面さえも綺麗に表示してくれるという, とても美味しいシステムです.

Maxima は 1960 年代の MIT の Project MAC で開発された MACSYMA(MAC's SYmbolic MAnipulation system) [1] の DOE(エネルギー省) 版を Texas 大学の Schelter 氏 [2]が「Common Lisp: The language 第 1 版」(cltl1 と略記) に対応した GCL に移植したものです[3]. 当初は Schelter 氏が Maxima の開発と管理を行っていましたが, Schelter 氏の死後はメイリングリストを中心に Maxima の保守・管理と開発が進められています. 詳細は sourceforge の Maxima のサイト (http://maxima.sourceforge.net/) を参照して下さい.

Maxima は幾らで買えますか？

Maxima は GPL ver. 2 というライセンス[4] の下で配布される無課金で利用可能なソフトウエアです. このライセンスには Maxima の自由な改変や配布を妨げること以外の制限はありません. この Maxima のソースファイルや実行ファイルは http://maxima.sourceforge.net/download.shtml から入手できます.

他の入手方法はやや大掛かりなものですが, Maxima を動かすための環境も一度に整備してしまうものです. この手法として KNOPPIX/Math と KNOPPIX/Edu を挙げておきましょう. ここで KNOPPIX は一枚の CD/DVD で起動可能な Linux 環境で, Maxima と他のアプリケーションとの連携をさせることが容易になります.

もう一つの異端的な手法は, SAGE という数式処理システムを利用するというものです. これは SAGE が Maxima を数式処理エンジンの一つとして内包し, さらに Maxima を直接呼び出して利用できるからです. なお, MS-Windows 版は Ubuntu という Linux 上に構築された VMWare や VirtualBox 向けの仮想計算機です. この方法は大規模なソフトウェアの開発や配布を考える上で非常に興味深い一例となっています.

---

[1] Macsyma の歴史については Petti 氏によるまとめがあります: http://www.math.utexas.edu/pipermail/maxima/2003/005861.html

[2] http://en.wikipedia.org/wiki/Bill_Schelter,
http://www.utexas.edu/faculty/council/2004-2005/memorials/schelter/schelter.html

[3] http://www.ma.utexas.edu/users/wfs/maxima-doe-auth.gif 参照

[4] GNU GENERAL PUBLIC LICENSE Version 2 の日本語訳は次のサイトを参照:
http://sourceforge.jp/projects/opensource/wiki/ licenses%2FGNU_General_Public_License
なお, GPL の下で配布されるソフトは俗に言う "無料(タダ)" のソフトではありません.

Maxima が動作可能な環境は？

「Common Lisp」という言語が動作する環境でなければなりません. Common Lisp はさまざまな計算機環境に移植されているので, 標準的な環境, たとえば, Solaris , Linux, FreeBSD, MacOS X や MS-Windows であれば動作します. また, 一般的な計算機に限らず, 携帯電話やタブレットで用いられる Android 上でも動作します (Maxima on Android). なお, この本では Common Lisp の中でも多くの環境に移植されている CLISP, 高速な処理の SBCL, SAGE に内包されている Maxima で用いられている ECL で動作の確認を行っています.

どのような読者層を想定していますか？

この本では以下の読者を想定しています：

- 我慢強い方 (ソースの修正, configure, make は当たり前, やはり vi が好き)
- 軟派な方 (楽をしたい. 興味があるのでちょっと使ってみたい)
- MS-Windows 派 (MS-Windows 上で全てを済ませたい. その他は知りたくもありません)

我慢強い方は UNIX 環境, ここでは特に Linux 環境で使うことを想定しています. なお, Linux にはツールやアプリケーション等のパッケージを纏めた「**ディストリビューション**」と呼ばれる OS のパッケージがあり, RedHat , Debian , Slackware の 3 種類に大きく分類できます. この分類はパッケージ管理の違いが根底にあり, パッケージファイルの修飾子からも容易に判別できます. たとえば, RedHat は ".rpm", Debian なら ".deb", Slackware であれば ".tar.gz" や ".tgz" です. ここで NOPPIX/Math は Debian 系になります. これらのディストリビューションの違いは, システムの設定ファイルの置き方や, パッケージ管理で表に出ますが, 単に Maxima を利用したり, 自力でコンパイルしてインストールするときにはあまり関係しません. むしろ, コンパイルで利用する Common Lisp と Maxima の側で問題が出易いでしょう.

軟派な方には KNOPPIX/Math という素晴しい玩具箱を用意しました. KNOPPIX の全般的な事項に関しては産総研のサイト (http://unit.aist.go.jp/itri/knoppix/) を参照して下さい. なお, KNOPPIX は一枚の CD や DVD から立ち上げ可能な Linux 環境で, KNOPPIX/Math は数学関係のソフトウエアを集積した KNOPPIX の分派です. 手軽に Linux 上で動作する数学ソフトを手軽に幅広く利用したい方には特におすすめです. 莫大な数学アプリケーションが簡単に利用できるからです. この KNOPPIX/Math の情報は http://www.knoppix-math.org/ から入手できます. **KNOPPIX/Math を使う気が全くなくても, この CD/DVD-ROM には色々な数学ソフトに関する文書が収録されているので内容を確認されることをお勧めします.**.

最後の MS-Windows 派の方は, インストールも簡単なので何も特別に書かなくても良い…としたいところですが, Maxima のインストールが簡単でも自分で工夫をしはじめると一番苦労しなければならない実に不幸な環境です.
でも, 福音があります. 「VMware Player」や「VirtualBox」を使えば MS-Windows 環境上でも Linux を動かすことが出来ます. そうすれば両方の楽しい所だけを堪能できますし, 他のソフトの連動や動作の確認といったこともできるでしょう. 仮想計算機が利用できる環境であれば, この際

に VMware Player や VitrualBox を使って KNOPPIX で遊ぶのはいかがでしょうか？

そして, Maxima は数学のソフトウェアです. そのために数学の知識があることに越したことはありません. この本ではささやかですが Maxima の考え方に必要なことや知っていて楽しい数学の話題を §4 に書いています.

Maxima も車と同様の道具です. 使い方だけを知ってさえいれば利用に困ることはないでしょう. しかし, その原理を知っていれば応用が効くだけではなく, 何よりも, より楽しく使えます. 個人的には, 群, 環, イデアルと聞いて, 心ときめかせられる高校生, 大学生, 一般社会人が増えれば良いなぁと思っています.

初心者なのでよくわかりません

この本は小説とは違い頭から読む必要はありません. Maxima を使う上で必要となる予備知識や雑多なことを色々と詰め込んでいるので厚くなっていますが, そこから, 貴方が必要と思えるものだけを読めば良いのです.
ただし, Maxima をどのように使えばよいのか見当がつかないのであれば, 最初に §2 の「ちょっとした計算例」を読んで雰囲気を把握し, それから Maxima の基本的な考え方を纏めている §5.1 を眺め, あとは必要や興味に沿って読むことはいかがでしょうか. さらに索引に「逆引き」を多少入れているので, 貴方のやりたいことに最も近そうな項目を選んでマニュアルや関連項目を読むこともできます.

また, Maxima の活用方法を手短に知る文書として, 中川義行氏による「Maxima 入門ノート」[5]を挙げておきます. この文書も KNOPPIX/Math に収録されています.
ちなみに, 私の本が世間一般で言われる初心者向けでない理由の一つが, この優れた「Maxima 入門ノート」があるお陰です[6].
そして, 些細なことでも Maxima を使って色々と試してみましょう. とにかく, 日常的に利用して試行錯誤しながら遊ぶことこそが一番の上達の方法です.

私は Mac ユーザです !!

Mac OS X を UNIX として使えるように環境設定していれば, FINK を用いたインストールができるようです. さらに最近の Intel Mac では KNOPPIX の立ち上げも可能とのことです. それ以外の古典的 Mac OS の利用者の方々は Mac 版の Linux(Debian , openSUSE や Vine 等) をインストールし, CLISP 等の MacOS X 上でも動作し, Maxima に対応した Common Lisp を使ってコンパイルすれば普通に使えると思います. なお, この本では Mac ユーザーは我慢強い方に分類しています. 日常的に我慢を強いられることが多いでしょうから…[7].

---

[5] http://www.wakaba.jp/moriarty/works/index.html からも入手可能です

[6] 「初心者向け」の本を作ることはとても難しいことです. 単なるマニュアルの翻訳だけ, 高校数学の例題を幾つかという代物が果して「初心者」にとって本当に良いものとは思えません. 初心者だからこそ「歯応えのある本」, あるいは「胡桃の様に堅い本」が必要だと私は考えています.

[7] 幸いなるかな汝等 MacOS X の利用者は！強者達の MachTen, LinuxPPC, MkLinux での苦戦・苦闘を思わば.

1.2 数式処理って何？

1.2.1 数式処理？

さて前節の FAQ では:

- Maxima は汎用の数式処理である
- Common Lisp が動作する環境があれば Maxima を動かせるかも

と申しました. とは言え「数式処理」？「Common Lisp」？となる方も多いことでしょう. そこで, もう少し詳しく解説しましょう.

まず,「**数式処理**」は数式の直接処理を目的としています. こう言うと「数式の直接処理」は C や BASIC でもできると思う方も多いと思いますが, 標準的なライブラリを用いて C や BASIC で扱う数式は, その式から得られる数値を扱っているのであり, 数式それ自体を扱っている訳ではありません.

具体的に考えましょう. 1 + 1 の計算は C や BASIC でも簡単に処理出来ます. 128 の因数分解はどうですか？ プログラムを組めばなんとかなるでしょう. それから x が 1 のときの x^2+2x+1 の値を計算することも簡単ですね. それでは x^2+2x+1 の因子分解はどうでしょう？ こちらは公式集を使えば何とかなるかもしれませんが, x^2-2y+y^2 の因子分解はどうでしょうか？答は $(x-y^2)$ ？ それとも $(y-x)^2$ とすべきでしょうか？ さらに $(x+1)(x-1)^{10}+(4x-2)^4$ の展開や $2\cos^2\theta+\cos 2\theta$ を簡単な式にするとか sin や cos といった初等函数の微分や積分は？これらの問題では式を単なる文字列として見るのではなく, 式の意味, すなわち何が変数で, どのような性質を持った演算や函数が含まれているかといったことをちゃんと解釈していなければなりませんね.

このように数式に含まれる記号をそのまま扱い, 与えられた式のさまざまな計算を行うことを数式処理は目的としています. この処理では代数学の考えが色々用いられており, そんなこともあって数式処理は「記号代数処理」とも呼ばれています.

さて, 数式処理自体はシステムに公式集を持たせ, 与式に公式を機械的に適用するだけで十分なように見えますが, それだけで本当によいのでしょうか？

1.2.2 Prolog を使った機械的処理の例

そこで「**Prolog**」を使った簡単な微分操作の例を見てみましょう.

まず, 1 変数 'x' による微分の処理をファイル "diff.pl" に次のように記述しておきます:

```
1 dx(x,1).
2 dx(Y,0) :- atomic(Y),Y \== x.
```

```
3 dx(Y+Z,DY+DZ) :− dx(Y,DY),dx(Z,DZ).
4 dx(Y−Z,DY−DZ) :− dx(Y,DY),dx(Z,DZ).
5 dx(Y∗Z,DY∗Z+Y∗DZ) :− dx(Y,DY),dx(Z,DZ).
6 dx(Y^N,N∗Y^N1∗DY) :− integer(N),N1 is N−1,dx(Y,DY).
```

Prolog では「**Horn 節**」と呼ばれる論理式を使ってプログラムを作成します. たとえば, 'dx(x, 1)' は変数 'x' による式 'x' の微分が '1' であるということを意味し, 'dx(Y, 0)' は 'Y' が原子で, 'Y' が 'x' と等しくなければ '0' であることを意味します. 以降, 和と差の場合の微分, 積の場合と定数の冪の場合の微分について記述しているのです.

このプログラムを Prolog 処理系の一つの「**SWI-Prolog**」[117] で実行してみましょう:

```
?- consult(diff).
% diff compiled 0.00 sec, 2,560 bytes

Yes
?- dx(x^2+2*x*y+y^2,Y).

Y = 2*x^1*1+ ((0*x+2*1)*y+2*x*0)+2*y^1*0

Yes
```

この例では最初に consult 命令でファイル "diff.pl" を読込み, 'dx(x^2+2*x*y^2, Y).' で 変数 x による $x^2 + 2xy + y^2$ の微分を計算しています. このプログラムでは 1 倍や 0 倍の処理, 式の展開やまとめといった処理が欠落していますが, 計算機は変数 x, y の意味とは無縁に機械的に処理を遂行しており, 計算の補佐役としては十分実用的です. しかし, 公式に当て嵌めるだけの処理は非常に大きな危険性を持っているのです.

1.2.3　安易な機械的処理の陥穽

ここでは函数 $1/x^2$ を区間 $[-1, 1]$ 上で定積分することを考えましょう. ちなみに $f(x)$ の原始函数が $F(x)$ のとき, 区間 $[a, b]$ での函数 $f(x)$ の定積分は公式 1.1 で与えられます:

$$\int_a^b f(x)dx = F(b) - F(a) \tag{1.1}$$

また, $n \in \mathbb{N}^+$($\mathbb{N}^+$は自然数の集合から 0 を抜いたもの) であれば $1/x^n$ の不定積分は公式 1.2 で与えられます:

$$\int \frac{1}{x^n}dx = -\frac{1}{(n-1)x^{n-1}} + c \tag{1.2}$$

さて, これらの公式を $1/x^2$ の積分に式 1.3 に示すように機械的に適用すると結果として -2 が得られます:

$$-\frac{1}{(2-1)(1)^{(2-1)}} - \left(-\frac{1}{(2-1)(-1)^{(2-1)}}\right) = -2 \tag{1.3}$$

これにて一件落着としたいところですが, 残念なことに, この結果は間違いです. 最初は直感的に考えてみましょう. まず, 被積分函数 $1/x^2$ の値は常に正です. そして, $1/(-x)^2 = 1/x^2$ となるので Y 軸に対して対称になりますね. さて, 1 変数函数の定積分は曲線と X 軸で囲まれた領域の (符号付きの) 面積となります. すると常に 0 より大きな値を持つ函数 $1/x^2$ の積分が -2 と負の値になることは非常に奇妙な話です！ このことから何かが間違っていると判断できますね.

では何処が怪しいのか考えてみましょう. そこで, ε を十分小さな正の実数とし, 区間 $[\varepsilon, 1]$ で積分することにします. ここで, 公式 1.1 が成立するためには, 何が必要でしょうか？ 単純なことですが, **函数 $F(x)$ が区間 $[a, b]$ 内で存在していなければなりません**. ところで, 函数 $1/x^2$ の形式的な積分は公式 1.2 から $-1/x$ です. この $1/x$ は区間 $[\varepsilon, 1]$ 上であればちゃんと計算できるので, 区間 $[\varepsilon, 1]$ での積分は公式 1.1 から $1/\varepsilon - 1$ になります.
さて, ε が 0 に近づくとどうなるでしょうか？ この値は一方的に増大し, それを抑える値, つまり, 上限がないことが判ります. 実際, 実数 $M > 0$ で $1/\varepsilon - 1$ が抑えられているとしましょう. ここで, ε を $1/M + 1$ よりも小さい値, たとえば, $\varepsilon = 1/2(M+1)$ にしてみましょう. すると $1/\varepsilon - 1 = 2M + 1$ となり, M よりも大きくなって最初の仮定に矛盾します, この上限を持たない性質を「**無限大**」と呼び, 函数 $1/x^2$ が無限大となる点 0 の様に, 函数が無限大となる点のことを「**極**」と呼びます. ここで極には二種類の極があります. 1 つは $1/x$ での点 0 の様に, 一定の値に収まらない性質を持った極です. それに対し, 函数 x^3/x を考えてみましょう. この場合, 点 0 で x^3/x は 0/0 という不定形になるので, x^3/x の極になります. しかし, $x \neq 0$ に対しては $x^3/x = x^2$ となり, $\varepsilon \to 0$ で $x^3/x \to 0$ となって, 一定の値で抑えることができない性質を持つのではなく, ある有限の一定値を持ち, 極にはなっていません. このように函数の極の近傍で, 極を持たない函数で置換できる極のことを「**除去可能な極**」と呼びます. この除去可能な極ならば, ここで示した問題は生じません.

このように数式の処理をちゃんと遂行するためには, 与式の計算で公式が適用可能な範囲に収まっているかどうかも検証する必要があります. そのことを忘れると, 先程の結果のように間違った結果を得ることになります. そのため, より本格的な数式処理は単に公式に当て嵌めるだけではなく, 与式に含まれる函数等の性質や特徴も把握するシステムであるべきです.

1.2.4 数式処理を記述する言語

では, 数式処理はどのような計算機言語で記述されているのでしょうか？現在は C や C++で記述することが多くなっていますが, 古いものの多くが LISP で記述されています. 先程, Prolog の例を出しましたが, LISP は 1980 年代に日本の第 5 世代計算機の研究が刺激となって「**人工知能 (AI)**」が流行ったときに Prolog と共に脚光を浴びた言語です. この LISP という言語は FORTRAN とほぼ同時期に生れた古い言語の一つですが, その一方で非常に柔軟な言語です. ただし, 規格化の進んだ FORTRAN と比べて方言の多い言語ですが, Common Lisp は Scheme と共に, 主要な LISP 言

語の一つです.

LISP の特徴は色々ありますが, LISP はリストと呼ばれる記号や数値等で構成されたデータ形式が容易に扱える特徴があります. さらに, LISP は言語的に函数型と呼ばれ, 定義した函数を組合せてプログラムを組む様式になります[8]. ちなみに C や FORTRAN は手続型と呼ばれる言語となり, Prolog は論理型と呼ばれ, プログラミングと処理の間に区別がない独特の言語です.

この本で解説する Maxima は Common Lisp で記述された「**汎用の数式処理システム**」です. ここで「**汎用の数式処理**」とは, さまざまな問題への対処を目指したシステムであることを意味します. また「**システム**」とは Maxima 言語で小さくまとまったものではなく, Maxima を核としてグラフ処理に GNU Plot 等の外部アプリケーションを組合せて使うことを意味します.

では, 次の章で Maxima について実例を交えて簡単な説明をおこないます.

コラム：数学ソフト色々

本格的な数式処理で有名なものに AXIOM [102], *Mathematica* [107], Maple [108], MuPAD [110], REDUCE [112] や SAGE [115], それに, TI のグラフ電卓でも使われている DERIVE [103], ワープロと合体した mathcad [106] やカルキング J[118] 等が挙げられます. これらは高校生や大学の教養程度の数式の処理が可能なものです.
これに対して, 数学の特定分野に特化したシステムもあります. これらは GPL や BSDL, あるいはこれらに類似したライセンスを持っています.
ツールボックスで MuPAD が使える MATLAB [119] は数式処理とはまた別の種族で, 数値行列データを効率的に扱うことを目的とした言語仕様となっています. この MATLAB は (小中規模の) 数値行列の処理で広く使われており, 標準的なソフトウエアになっています. そのために, Octave, Scilab, FreeMat, RLab 等の MATLAB 風のシステムが多くあります.
統計処理が必要な場合は S 言語の系統 (S-PLUS [124] や GNU R [123]) が広く用いられています.

[8] より徹底した函数型言語としては Ocaml や Haskell が挙げられます)

第2章　ちょっとした計算例

この章では Maxima で何ができるかを具体的に実例を使って紹介しましょう.
ここでの実例で紹介した函数等の気になった事項は Maxima のオンラインマニュアルや, この本に収録したマニュアルを参照して下さい.

2.1　Maxima のユーザーインターフェイス

さて, Maxima を使おうとするとき, 式や命令を Maxima に伝えてやらなければなりません. 「Hello Computer!」と呼びかけると応答し, 命じるままに処理してくれれば良いのですが, 残念ながら現状では「キーボード+マウス」やそれに類似した手法に敵うものはまだありません. そして, それらを駆使して, 受付となるアプリケーション (フロントエンド) が Maxima に情報を伝え, 処理結果を表示することになります.
ここで Maxima にはフロントエンドとして仮想端末を用いる **maxima**, GUI を駆使する **wxMaxima** や **xMaxima** といったフロントエンドがあります. 他にも **GNU Emacs** を利用する **imaxima**, KDE 環境で利用可能な **Cantor**, さらにはエディタの **TeXmacs** を Maxima のフロントエンドとして利用する方法もあります. なお, ここで紹介するフロントエンドのほとんどが KNOPPIX/Math に収録されています.

wxMaxima:　MS-Windows 版の Maxima に付属し, 標準のフロントエントになっており, 図 2.1 に示すように数式が美しく出力されます:

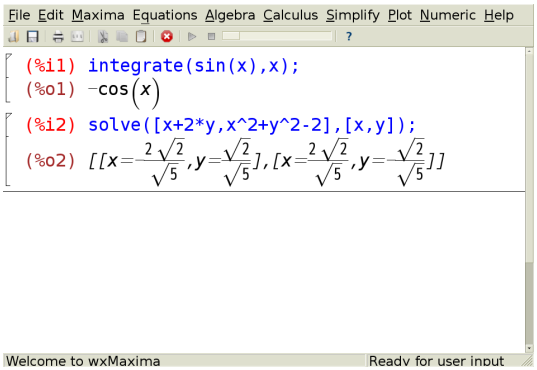


図 2.1: wxMaxima

この wxMaxima のことを Maxima と思っている方も多いようですが, wxMaxima は入力と出力の表示を受け持つアプリケーションで Maxima そのものではありません.
wxMaxima の新版は *Mathematica* のフロントエンドに類似したものになっており, 縮約・展開可能なセルに式を入力する方式になっています. ただし, 立ち上げた時点では古い wxMaxima の様な入力欄やボタンを持たず, のっぺらぼうなワークシートが表示されているので面食らうかもしれません. 使い方は, まず, そののっぺらぼうなワークシートに数式を入力します. 実は, よくよく見れば黒い線がワークシート上にあり, 入力を行うと, その黒線の直上に入力式を含むセルが生成されます. そして, 式の評価は *Mathematica* 風に Shift+Enter と入力すれば良いのです.
wxMaxima は入力の受付と出力表示を担当し, Maxima と TCP/IP を使って通信しています. そのためにファイヤーウオールの設定次第で, この通信が遮断されることに注意して下さい. もし Maxima を立ち上げても変なメッセージしか出ない, ファイヤーウォールソフトから何らかの警告が出る場合, wxMaxima-Maxima 間の通信を許可するようにファイヤーウォールの設定を変更して下さい.

xMaxima: Maxima のソースファイルに標準で附属し, 以前は標準的なフロントエンドでした:

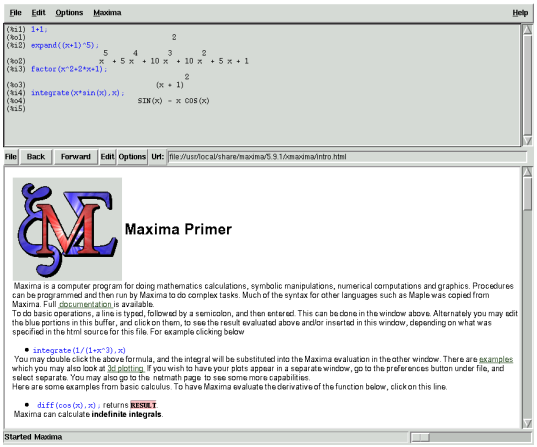


図 2.2: xMaxima

xMaxima は図 2.2 に示すように上下の副ウィンドウに分割されており, ウィンドウの上段側に通常の入力と結果の表示が行われます. また, 結果は ASCII-art 風に数式が表示される古風なものです. また, ウィンドウの下段は例題の表示に用いられ, この例題ではハイパーリンクが支えます. この xMaxima は tcl/tk を使って記述されているために tcl/tk で記述された openmath とも相性が良い長所があります. 実際, plot_format に openmath を指定していれば, 表示グラフは他の出力と同様に表示されます.
この xMaxima も wxMaxima と同様に Maxima 本体と通信を行います. そのため, ファイアーウォールの設定で Maxima - xMaxima 間の通信を遮断しないように注意しなければなりません.

Cantor : Cantor は Maxima だけではなく, さまざまな数学アプリケーションのフロントエンドとなるように開発されており, Maxima の他には SAGE, KAlgebra や GNU R のフロントエンドとしても利用できます. Cantor では Tab キー による入力式の補完機能もあります. そして入出力式を LaTeX を使ってレンダリングを行ない, その結果, 美しい数式が表示されます. その上, グラフ出力も商用の *Mathematica* 同様にワークシート内に表示され, 三次元グラフであればワークシート内のグラフを直接マウスで把持して回転や拡大・縮小が容易に行えます.
なお, Cantor は基本的に KDE4 の環境のみで動作します.

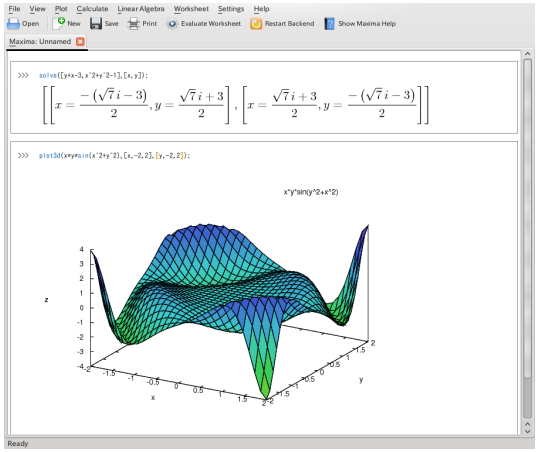


図 2.3: Cantor

TeXmacs :　WYSWYG なエディタで数式が綺麗に表示できるだけではなく, Maxima 以外のアプリケーションのフロントエンドとしても使えます．対応しているアプリケーションは gnuplot , Macaulay2 , Octave , PARI/GP や Risa/Asir で, Maxima のフロントエンドとして利用する場合, **挿入 (Insert)** → **セッション (Session)** → **Maxima** と指定します.

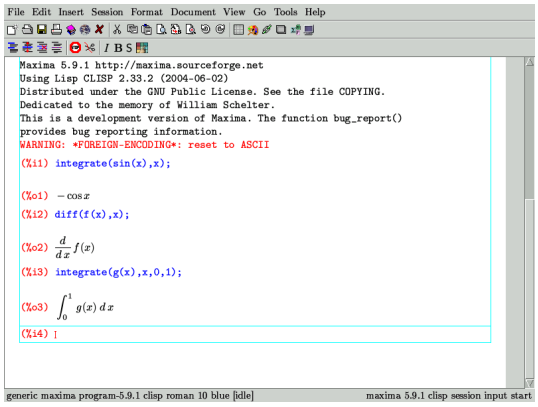


図 2.4: TeXmacs

なお, KNOPPIX/Math に収録された TeXmacs は日本語にも対応したものになっています.

さて, この本では仮想端末上で起動した Maxima の入力と出力を表示します. この理由ですが, どのような環境でも利用できることに加え, こちらの方が慣れてしまえば手軽だからです.

2.2 入力

Maxima を立ち上げると入力行のプロンプト “(%i1)” が出ています. 処理させたい式はこのプロンプトに続けて入力します. ここで入力行末尾には必ずセミコロン “;” , あるいは, 記号 “$” といった入力行の末尾を示す文字を付けなければなりません. Maxima はこれらの行末を示す文字が入力されるまで入力が継続していると判断し, 入力の完遂を待つからです. このときにはプロンプトが現われずに, Enter キー を押しても改行されるだけになります.

では簡単な数値計算を幾つか実行させてみましょう. ここで和 “+”, 差 “−”, 積 “∗”, 商 “/” を含む数式は C や FORTRAN と同じ書式です:

```
(%i1) 1+2;
(%o1)                                  3
(%i2) 2/3+3/5
;
                                      19
(%o2)                                 --
                                      15
(%i3) display2d : false ;
(%o3) false
(%i4) 2/3+3/5;
(%o4) 19/15
(%i5) (%o1)+(%o2)*15;
(%o5)                                 22
(%i6) quit ();
```

通常, Maxima に数式を入力すれば直ちに計算結果を返します. この対話を行うように処理が進められる言語のことを「**対話処理言語**」と呼びます. ここで Maxima の入力と出力で重要なことを幾つか纏めておきましょう:

プロンプトとラベル: 入力行のプロンプトの書式は ‘(%i⟨ 番号 ⟩)’ で, ⟨ 番号 ⟩ が入力行番号, ‘%i⟨ 番号 ⟩’ が番号に対応する入力行のラベルになっています. この入力に対応する出力行の先頭には ‘(%o⟨ 番号 ⟩)’ というプロンプトがあり, ⟨ 番号 ⟩ が出力行の行番号で ‘%o⟨ 番号 ⟩’ がそのラベルになります. これらのラベルには対応する入出力が保存され, 利用者はラベルを指定することで入力式や結果の参照や再利用が容易に行えます. この例では, 入力ラベル “%i1” には入力式の ‘1+2’, 対応する出力ラベル “%oi” には結果の ‘3’ が保存されています.

行末を示す文字: 分数の計算例として 2/3 + 3/5 を計算をさせていますが, この例ではセミコロン “;” を行末に付けなかったため, Maxima は入力待ちになっています. 実際, (%i2) 行で ‘2/3+3/5’ と入力していても結果が何も表示されていませんね. そこでセミコロン “;” を入力すると直ちに計算結果を返し, その結果は出力ラベル ‘%o2’ の行に表示されます. このように Maxima は入力行に入力行末を示す記号 “;” や記号 “$” を入力するまで式が入力中であると判断して計算処理を行いません. そして, 記号 “;” が行末であれば, その入力行の処理結果を表示し, 記号 “$” が行末であればその処理結果を表示しません.

結果の表示:　Maxima は wxMaxima, Cantor や TeXmacs を利用している場合, 本に印刷されたような綺麗な式を返しますが, 通常の Maxima や xmaxima では計算結果を ASCII 形式で通常の数式のように二次元的に表示しようとします. そこでウインドウが小さなとき, あるいは複雑な式や長い式であれば表示領域が足らずにまともに読めないことがあります. さらに計算結果を他のプログラムに書き写そうとしても, この表記ではそのまま使えません. この事態を避けるため, Maxima の大域変数 display2d に 'false' を割当てて, この表示を止めさせることができます. この例では大域変数 display2d に 'false' を割当てるために演算子 ":" を用います. ここで, 演算子とは Maxima の対象を結合することで新しい Maxima の対象を生成したり, 既存の対象に何らかの作用を及ぼす記号です. 割当の演算子として C は演算子 ":=" を用いますが, Maxima では演算子 ":" を用います. なお, Maxima で演算子 ":=" は函数定義で用いる演算子, 演算子 "=" は等号の演算子です. C や FORTRAN の演算子と混同しないように注意が必要です.

さて, 大域変数 display2d を 'false' に変更すると Maxima の式 '2/3+3/5;' の結果は '19/15' と表示され, 一行に収まるように表示されていますね. ここで, 初期状態の二次元表示に戻したければ今度は 'display2d:true;' と入力すれば良いのです.

このように Maxima の大域変数には, 一つの函数のみに影響を与えるものから, 複数の函数に影響を与えるものと色々あります.

履歴の利用:　ここで Maxima の式 '(%o1)+(%o2)*15' の意味は, ラベル '(%o1)' に表示された計算結果にラベル '(%o2)' で表示された結果を 15 倍したものとの和を計算した結果です. Maxima では計算結果を入出力の値を番号に対応するラベルに対応させて保存しているので, ラベルを履歴として用いた計算が行えます. この履歴は Maxima の終了, 初期化やラベルの消去で消去できます.

この Maxima にはその履歴を参照する幾つかの大域変数や函数が用意されています. まず大域変数 %に直前の結果が割当てられています. ここで整数を引数とする函数%th は '%th(n)' で n 回前の計算結果を参照します. だから, '%th(1)' の返す値は大域変数 '%' に割当てられた値と同じです.

履歴は「**バッチ処理**」でも利用できます. ここで「**バッチ処理**」は, 処理する内容を予めファイルに記述して, それをアプリケーションに逐次実行させる処理方法です. この処理内容を記述したファイルのことを「**バッチファイル**」と呼びます.

この手法は昔の大型計算機では一般的でしたが, PC でもバッチ処理を上手く使えば定型処理が容易に行える利点があります. ただし, ラベルを直接 '(%o10)' のように直接指定して利用する場合は注意が必要です. 実際, 初期化の処理が計算機の環境毎に異なっているだけでラベルの位置が異なる可能性があり, その上, ファイルが大きくなるに従って確認が難しくなる欠点もあります. そのために相対的ラベルを指定する方法が無難です.

履歴の消去:　履歴の消去は 'kill(labels)' でラベルに割当てた値を全て破棄することで行います. この処理は計算機の記憶容量が足りなくなったときに行います. この処理を行うと履歴が全て消去さ

れて以前の履歴が参照できなくなります. さらに 'kill(labels)' を実行すると, ラベルの番号は全て 1 に戻されます. そのために 'kill(labels)' を頭に入れて, その後に実際の処理を行うバッチファイルであれば '%o1+%o2*15' のようにラベル番号を直接指定する処理でも問題は生じ得ません.

Maxima の終了: 'quit();' で Maxima を終了します. この quit 函数は引数が必要ありませんが, 'quit' のうしろに小括弧 "()" を忘れないようにしましょう. Maxima では引数を必要としない函数に対しても, その末尾の小括弧 "()" が省略できません. 省略すれば Maxima は入力式を変数と判断し, その変数が束縛変数であれば割当てられた値を返却し, 自由変数であればその変数名を返却します.

オンラインマニュアル: Maxima をまた立ち上げ, 今度は help; と入力して下さい:

```
(%i1) help;
(%o2)         type ' describe ( topic );' or 'example(topic );' or '? topic '
(%i2)
```

すると describe 函数や example 函数を使えと返事がありますね. describe 函数は texinfo を用いたオンラインマニュアルを表示し, example 函数は Maxima の例題ファイル (*.dem) を捜して実行する函数です.

この describe 函数はちょっとした字引代りにも使えます. つまり Maxima の函数や大域変数の名前が分らなくても関連する事項を指定することで Maxima が一覧を表示してくれます. たとえば, 方程式を解きたければ "solve" をキーワードにして describe(solve); と入力します:

```
(%i36) describe ( solve );
 1: fast_linsolve : definitions for affine .
 2: funcsolve : definitions for equations .
 3: globalsolve : definitions for equations .
 4: linsolve : definitions for equations .
 5: linsolve_params : definitions for equations .
 6: linsolvewarn : definitions for equations .
 7: solve : definitions for equations .
 8: solve_inconsistent_error : definitions for equations .
 9: solvedecomposes : definitions for equations .
 10: solveexplicit : definitions for equations .
 11: solvefactors : definitions for equations .
 12: solvenullwarn : definitions for equations .
 13: solveradcan : definitions for equations .
 14: solvetrigwarn : definitions for equations .
 15: zsolve : definitions for equations .
enter space-separated numbers, all or none:
```

この状態から番号 7 を入力すると「番号 7:」の「solve 函数のオンラインマニュアル」が表示されます. ただし describe 函数は単純に info ファイルを表示するだけなので, 使い勝手はあまり良いものではありません. その上, マニュアルの内容が詳細になると仮想端末のバッファに収まりきらないために先頭から読めないこともあります. オンラインマニュアルを使って色々と調べながら Maxima

を利用するのであれば, wxMaxima, xMaxima, Cantor や TeXmacs, あるいは GNU Emacs をフロントエンドを利用すべきです.

この describe 函数と同じ働きをする特殊な記号 (演算子) に “?” があります. ‘? 事項’ は ‘describe(事項)’ と同じ結果になります. ここで “?” のうしろには必ず「空白文字」(SPACE) か「タブ」(Tab) を入れましょう. もし空白文字を入れなければ, Maxima は調べたい事項を LISP の函数と看做しますが, ここで引数がないためにエラーにはならずに単純に ‘false’ が返却されるだけです.

また函数や大域変数の綴を忘れた場合, 与えられた文字列から適合する函数名や大域変数等のリストを返す apropos 函数 が使えます. たとえば, 積分函数の名前が int か integrate のどちらかか判らなくなったときに apropos(int) と入力すれば, 名前が “int” から開始する函数や大域変数のリストが返されます:

```
(%i11)  apropos( int );
(%o11)  [ int , intanalysis , inte , integer , integerp , integer_partitions ,
 integfactor , integrate , integrate_use_rootsof , integrationconstant ,
 integration_constant_counter , interaction , interpolate , interpolate_subr ,
 intersect , intersection , intfaclim , intfactor , intopois , intosum , intp ,
 intpolabs , intpolerror , intpolrel , int_partitions ]
```

この apropos 函数は函数と大域変数等を一つのリストに纏めて返すために不必要に長いリストが返却されることもあります. 実際, この “int” の例では整数 “integer” に関連するもののといった積分 “integrate” 以外の名前が含まれており, その分, 返却されたリストが長くなっています.

2.3　演算子

Maxima で扱う “数式” は C や FORTRAN で用いられる数式とほぼ同じ書式です. 実際, 和, 差, 積, 商といった四則演算 (+, −, *, /) は C, FORTRAN だけでなく, 一般のプログラム言語で用いられる記号と同名で同じ使い方になります. これらの四則演算の記号のように対象を繋ぎ合せることで新たな対象を生成する働きを持つ記号を「**演算子**」, 演算子が作用する対象を「**被演算子**」と呼びます. ここで挙げた四則演算の演算子は二つの演算子を取ることができますが, このように二つ取る演算子を「**二項演算子**」と呼びます. さらに被演算子を左右に配置する二項演算子のことを「**内挿表現の演算子**」, あるいは「**中置表現の演算子**」と呼びます.

また演算子 “+” や演算子 “*” は両側の被演算子を入れ替えても同じ結果を得ますが, 演算子 “−” と演算子 “/” は違います. 実際, 1 + 2 と 2 + 1 は同じ結果になる一方で, 1/2 と 2/1 は違います. 和 “+” のように左右の被演算子を交換することができる性質を「**可換**」, 演算子 “−” や演算子 “/” のように可換でない性質を「**非可換**」と呼びます. 通常, 二項演算子は非可換です. 何故なら, 可換となるためには左右の被演算子が入れ換えられるという性質, すなわち「**属性**」を別途与えなければならないからです.

一方で演算子 “+” と演算子 “−” は中置表現の演算子ですが, $+a$ や $-a$ のように被演算子が一つの場合も許容し, 被演算子の前に置かれます. このように被演算子を一つだけ取り, 被演算子の前に配置される演算子を「**前置表現の演算子**」と呼びます. もちろん, 「**前**」があれば「**後**」もあります. この類の演算子には階乗の演算子 “!” があります. Maxima には他に演算子 “!!” があります:

```
(%i14) 2*3+5/(2*(4+1));
                                    13
(%o14)                              --
                                    2
(%i15) (x+1+y^2)/(x*(x+y^2+1))-1/x;
(%o15)                               0
(%i16) 11!;
(%o16)                          39916800
(%i17) 11!!;
(%o17)                            10395
(%i18) factor(%o17);
                                   3
(%o18)                          3  5 7 11
(%i3) stardisp:true$
(%i20) factor(11!!);
                                   3
(%o21)                          3 *5*7*11
```

この例では最初に整数の演算, 次に多項式の演算を行い, 最後に階乗 “!” と “!!” の違いを示しています. ここで factor 函数は整数や式の因子分解を行う函数で, ‘11!!’ の計算結果が割当てられたラベル %o17 を使って, その値を因数分解しています. Maxima の出力では可換積演算子 “*” は空白文字に置換えられて表示されます. ここで演算子 “*” の表示は大域変数 stardisp で制御されており, 既定値の ‘false’ であれば演算子 “*” を表示しませんが, ‘true’ であれば演算子 “*” を表示します.

Maxima の面白い点の一つは利用者が必要に応じて演算子の定義が行えることです. ここで演算子の定義手順には二種類あり, 第 1 の方法は, 前置表現, 内挿表現, 後置表現, あるいは無引数といった演算子の属性を先に指定し, 具体的な演算子本体を Maxima の函数として定義する方法, 第 2 の方法は, Maxima の函数を定義してから, その函数を演算子として宣言する方法があります. 前者の方法は, 形式的な演算子を容易に定義できる手法です.

ここでは infix 型の内挿表現演算子 “mike” を定義してみましょう:

```
(%i1) infix("mike");
(%o1)                                "mike"
(%i2) x mike y:=x*y*sin(sqrt(x^2+y^2));
                                                    2     2
(%o2)                  x mike y := x y sin(sqrt(x  + y ))
```

```
(%i3) 10 mike 5;
                                          3/2
(%o3)                                50 sin(5    )
```

では, 上記の例の内容を吟味しましょう. 最初の 'infix("mike")' で対象 "mike" が infix 型の内挿表現の演算子であると宣言しています. このように演算子としての宣言だけを行って, 演算子本体を定義をあとに回しても構いません. ただし, 演算子本体の定義は演算子としての宣言の有無で書式が異なります. この例では infix 演算子として属性を与えたために通常の函数の書式ではなく内挿表現の演算子の書式で定義することになります. そのために函数定義の演算子 ":=" の左辺に内挿表現 "x mike y" を置き, 右辺に, 被演算子に対応する処理の内容を記述していますね.

参考までに函数を定義し, それから函数に演算子の属性を与える方法で演算子 "mike" を定義しておきます:

```
(%i1) mike(x,y):=x*y*sin(sqrt(x^2+y^2));
                                                    2     2
(%o1)                 mike(x, y) := x y sin(sqrt(x  + y ))
(%i2) infix("mike");
(%o2)                                mike
(%i3) 10 mike 5;
                                          3/2
(%o3)                                50 sin(5    )
```

§5.3 で詳細を述べますが, Maxima の演算子には「**infix 型**」の内挿表現の演算子に加え,「**nary 型**」の内挿表現の演算子,「**前置 (prefix)**」,「**後置 (postfix)**」,「**matchfix 型**」と「**引数無し (nofix)**」の演算子があります. 被演算子の個数は前置表現と後置表現の演算子で 1 個, 内挿表現の演算子で 2 個, matchfix 型は任意, 引数なしは 0 個です. 次に前置表現と後置表現の演算子を実際に定義してみましょう:

```
(%i7) prefix("tama");
(%o7)                               "tama"
(%i8) tama x:=2^x;
                                          x
(%o8)                             tama x := 2
(%i9) tama 3;
(%o9)                                 8
(%i10) postfix("pochi");
(%o10)                              "pochi"
```

```
(%i11) x pochi := x/10;
                                          x
(%o11)                          x pochi := --
                                          10
(%i12) 2 pochi;
                                        1
(%o12)                                  -
                                        5
```

ここで数式 $(2\times 3^5/4)-1$ が与えられたとき, この数式をどのように解釈しているでしょうか？ 通常は記号 “( )” を使って $((2\times(3^5))/4)-1$ と演算子の処理に順序を入れて解釈しています. Maxima でも数式を処理する上で ‘2*3^ 5/4-1’ と入力されると内部では ‘((2 *(3^5))/4)-1’ で処理します[1]. ではどのような仕組で「**演算子の優先度**」を定めているのでしょうか？ 少し考えてみましょう. 数式 $2^3\times 4+1$ を $((2^3)\times 4)+1$ と解釈し, 決して $2^{3\times(4+1)}$ とは解釈しません. ここで演算子に被演算子を引きつける力を考えてみましょう. すると通常の数式では「**冪**」が最も強く, その次に「**商**」, 「**積**」, 「**差**」, そして最後に「**和**」になります. この「**演算子が被演算子を引き付ける力**」のことを「**束縛力**」と呼び, 0 から 200 以下の整数値で表現します. そして, 二つの演算子が一つの被演算子を共有する場合, この束縛力の大きな演算子が優先される仕組みになっているのです. Maxima では演算子の宣言で束縛力の大きさを設定しなければ, 既定値として 180 が設定されます. この束縛力は左右別々に設定できます. たとえば, 和 “+” は左右の束縛力が 100 と左右が同じですが, 積 “*” は左束縛力が 120 で右束縛力が未設定, 冪 “^” は, 左が 140 で右が 139 となっています.

ここで具体的な例で束縛力を確認してみましょう:

```
(%i29) prefix("test:")$
(%i30) test:2+3-test:(2+3);
(%o30)                      - test: 5 + test: 2 + 3
(%i31) prefix("test:",90)$
(%i32) test:2+3-test:(2+3);
(%o32)                        test: 5 - test: 5
(%i33) %-test:(5-test:5);
(%o33)                                0
```

この例では前置表現の演算子 “test:” を定義していますが, 被演算子が左側のみに配置するために左束縛力のみを設定しています. 最初の定義では既定値として 180 が自動的に設定され, この束縛力が和や差と比べて格段に大きいために ‘test:2+3’ を ‘(test:2)+3’ と Maxima は解釈します. ここで演算子の束縛力の変更は演算子を宣言する函数で束縛力を設定し直せばよいので, 今度は prefix 函数を使って束縛力を和や差よりも小さな 90 に設定します. すると ‘test:2+3-test:(2+3)’ の最初の演算子 “test:” の直後にある 2 は和の演算子 “+” に引き寄せられるので, 今度は ‘2+3’ が優先して処理されます. そして, 次の差の演算子 “-” よりも演算子 “test:” の方が束縛力が今度は弱いので ‘2+3’ は差の演算子に引き寄せられます. 以上から ‘test:2+3-test:(2+3)’ は ‘test:((2+3)-test:(2+3))’ として解釈されることが分かります.

[1] 内部表現を噛み砕いた表記で, 内部表現そのものではありません.

このような方法で必要とされる演算子が簡単に定義できます．また演算子の定義に加え, Maxima の規則を上手く使えば, より高度な処理が行えるのです.

2.4　式の評価

Maxima の数式は C や FORTRAN の数式と同様の書式になります:

```
(%i10) (1+2+3*x)^2*(1+y)/(z-1);
                                        2
                                  (3 x + 3)  (y + 1)
(%o10)                            ------------------
                                        z - 1
(%i11) a:sin(x)*cos(y)-x^2+2-(1+u^2);
                                                2     2
(%o11)                        sin(x) cos(y) - x  - u  + 1
(%i12) a^2;
                                                2     2      2
(%o12)                      (sin(x) cos(y) - x  - u  + 1)
```

通常の数式と同様に, 式を括る必要があるときは小括弧 “()” を用い, 変数への値の割当や代入には演算子 “:” を用います. なお, 現在の多くの数式処理で演算子 “:=” が割当や代入で利用されていますが, Maxima では, この演算子を函数定義で用いてます. また演算子 “=” は通常の数式と同様の等値性を表現する「**比較の演算子**」で, C の演算子 “==” に対応します．これらのことは間違い易い点なので注意して下さい.

さて, Maxima は入力した式を評価し, その結果を返却します．ここでの評価は, 入力式中の項を Maxima の項順序に従って並び換えること, それに伴う項のまとめといった簡単な簡易化です．より徹底した簡易化処理が必要であれば, expand 函数や ev 函数等に加えて文脈や規則といった仕組を適宜利用しなければなりません.

この実例を幾つか示しておきましょう．ます, 式の展開は expand 函数を使います:

```
(%i13) (x+1)*(x-1)+(y+1)^2;
                                  2
(%o13)                      (y + 1)  + (x - 1) (x + 1)
(%i14) expand((x+1)*(x-1)+(y+1)^2);
                                   2          2
(%o14)                            y  + 2 y + x
```

それから, 与えられた式を簡単な式で纏めたければ, factor 函数, 有理式を整理したければ ratsimp 函数を使います:

```
(%i8) 2*x^2+4*x^3+x^4-2*x^5-x^6;
p                                6      5    4      3      2
(%o8)                        - x  - 2 x  + x  + 4 x  + 2 x
(%i9) factor(%);
```

```
                                    2          2    2
(%o9)                         - x   (x + 1)   (x  - 2)
(%i10) 1/(x+1)+x^2/(x^2-1);
                                      2
                                     x          1
(%o10)                             ------ + -----
                                    2         x + 1
                                   x  - 1
(%i11) ratsimp(%);
                                      2
                                     x  + x - 1
(%o11)                               ----------
                                        2
                                       x  - 1
```

通常の手計算で一番苦労することが, 与えられた式を見通しの良い式に変換することです. この点は Maxima を使っても同様で, 目的に適した式を得るためには Maxima に色々な指示を与える必要があります.

そこで一般的な方法を幾つか説明しておきましょう. 最初に括弧で括られた式を展開する方法では expand 函数を用います. これで式が括弧を持たない平坦な式になります. このときに式を構成する項の並び替えが Maxima の項順序に従って実行され, 項のまとめという自律的な簡易化が行われます. ここで式を簡単にする函数の多くが項順序を用いた簡易化を行うために, expand 函数による処理が前提となっています. 次に, 式を共通の因子で纏めて簡単にする操作, すなわち因子分解を factor 函数で実行します. また与えられた式に有理数係数の多項式や有理式があればさらに ratsimp 函数を, 式に三角函数や指数函数が含まれていれば, trigsimp 函数を使います.

通常, 言葉の意味を考える場合, その言葉が含まれている文も含めて考慮しなければなりません. 実際, 「赤」という言葉にしても, 会社経営では「赤字」, 出来の悪い学生なら「落第点」の意味だったりします. これは「**文脈原理**」と呼ばれている原理です[2]. このことは数学でも同様です.

たとえば数式 $\sqrt{x^2}$ で考えてみましょう. この数式だけであれば $\sqrt{x^2}$ 以上のことは言えませんが, ここで x に関する言及があれば違ってきます. つまり, 「x **が** 0 **よりも大である**」という言及があれば $\sqrt{x^2}$ は x, 「x **が** 0 **以下である**」という言及があれば $\sqrt{x^2}$ は $-x$ と簡単にできますね. このように数式を構成する式や変数に関する言及の蓄積, すなわち, 情報によって数式の値, すなわち意味が決定されます. この情報のことを「**文脈 (context)**」と呼び, Maxima では assume 函数を使って構築できます:

```
(%i1) assume(x>0);
(%o1)                                [x > 0]
(%i2) sqrt(x^2);
(%o2)                                  x
(%i3) sqrt(y^2);
(%o3)                                abs(y)
```

[2]「語の意味は文での関連において問われるべきであって孤立して問われるべきではない」, §4.19 参照

この例で示すように変数 x に仮定 ‘x>0’ が設定されたために Maxima の式 ‘sqrt(x^2)’ が ‘x’ に簡易化されましたが, ‘sqrt(y^2)’ の変数 y には何等の条件もないので ‘abs(y)’ が返却されています. このように Maxima の式の処理は文脈上で実行されます. 利用者が文脈を指定しなければ文脈 ‘initial’ が用いられます. Maxima の文脈は木構造であり, 利用者が自分専用の文脈を生成し, 必要に応じて文脈を切り換えることもできます.

式の簡易化では文脈だけではなく簡易化を行う函数の動作を制御するさまざまな Maxima の大域変数を調整する必要が生じることもあります. このような処理に長じているのが ev 函数です.

この函数は非常に機能が高いので, 詳細は §5.8.3 を参照して下さい. ここでは ev 函数の代表的な使い方を幾つか示しておきましょう:

```
(%i9) A:sin(3*x)-cos(3*y)*z;
(%o9)                          sin(3 x) - cos(3 y) z
(%i10) ev(A,trigexpand=true,trigexpandtimes=true,eval,z=2);
                 3                  2          3          2
(%o10)   - 2 (cos (y) - 3 cos(y) sin (y)) - sin (x) + 3 cos (x) sin(x)
(%i11) B:sin(2*x)-cos(3*y)*(z+1)^3;
                                                    3
(%o11)                    sin(2 x) - cos(3 y) (z + 1)
(%i12) B,trigexpand=true,trigexpandtimes=true,eval,z=2;
                                          3                  2
(%o12)          2 cos(x) sin(x) - 27 (cos (y) - 3 cos(y) sin (y))
```

ここでは式 ‘A’ を三角函数の倍角公式を使って ev 函数で展開して変数 z に 2 を代入する例と, ev 函数を表に出さずに同様の処理を行う例を示しています. ev 函数の構文は ‘ev(⟨ 与式 ⟩,⟨ 条件$_1$⟩,$\cdots$,⟨ 条件$_n$⟩) ’で, 与式のうしろに続く条件を逐次適用します. Maxima の最上層 (プロンプトが (%i 番号) となっている階層) では最後の例のように ev 函数の中身だけを記述しても構いません.

さらに変数に属性を与えたり, 定義した函数の性質を付加する必要が出ることもあります. Maxima には対象を定めて規則を定義し, その規則を式に対して適用することもできます:

```
(%i19) matchdeclare(_a,true);
(%o19)                                done
(%i20) let(tama(_a)^2,tama(2*_a)+1);
                                2
(%o20)                    tama (_a) --> tama(2 _a) + 1
(%i22) letsimp(expand((1+tama(x))^2));
(%o22)                      tama(2 x) + 2 tama(x) + 2
```

この例では tama という函数名のみを持つ実態のない函数に対し, ‘tama(x)^ 2’ を ‘tama(2*x)+1’ で置換する規則を let 函数を使って定義し, この規則を letsimp 函数によって与えられた式に適用させています. Maxima の「**規則**」の詳細については §5.7 を参照して下さい.

2.5 数値計算

Maxima で扱える数には,「**整数**」,「**有理数**」,「**浮動小数点数**」(倍精度と多倍長の二種類) と「**複素数**」があります. まず整数は '123' のような ASCII 文字の "0" から "9" までの数字で構成された対象, 有理数は '2/3' のように二つの整数の商 "/" の式として表現されます.

浮動小数点数は実数を表現するために用いられる数ですが, 本質的に実数の近似値です. この浮動小数点数は符号部, 仮数部と指数部の 3 個のブロックで構成された二進数で表現された対象で, 倍精度が 64 ビット長の二進数, 多倍長は利用者が指定した長さ (¿64 ビット) の二進数で表現されています. Maxima の倍精度の浮動小数点数の表記は "0" から "9" までの数字とただ一つの小数点記号 "." を含む対象ですが, たとえば, '123.4' の他に, '12.34e+1' や '1.234e2' といった表記もあります. これに対して多倍長の浮動小数点数は '12.34b+1' や '1.234b+2' といった表記のみが許容されます. そして, 複素数はこれらの数 a, b と純虚数 '%i' を用いて 'a+b*%i' と表記される Maxima の式として表現されます.

ここで多倍長浮動小数点数では必要に応じて利用者が浮動小数点数の精度を変更できるといいましたが, この精度は Maxima では大域変数 fpprec と大域変数 fpprintprec の設定に依存します. 精度の設定そのものは大域変数 fpprec で桁数を指定することで行い, 大域変数 fpprintprec には表示桁数を指定します. もし大域変数 fpprintprec が 0 であれば表示桁数は大域変数 fpprec の値と一致します. これらの大域変数の働きを有理数 1/3 を多倍長浮動小数点数に変換する例で見てみましょう:

```
(%i3) fpprec;fpprintprec;
(%o3)                                16
(%o4)                                 0
(%i5) bfloat(1/3);
(%o5)                          3.333333333333333b-1
(%i6) fpprec:100$bfloat(1/3);
(%o7) 3.333333333333333333333333333333333333333333333333333333333333333333333333333333333333\
33333333333333333333333333333333b-1
(%i8) fpprintprec:16$bfloat(1/3);
(%o9)                          3.333333333333333b-1
```

大域変数 fpprec の既定値は 16 と倍精度の浮動小数点数の精度と一致し, 大域変数 fpprintprec の値は既定値の 0 なので表示桁数も大域変数 fpprec で指定した桁数の 16 と一致します. 計算桁数の変更は大域変数 fpprec に直接桁数を割当て, 表示桁数は別途, 大域変数 fpprintprec で指定できるので, 内部的に 100 桁で計算しても表示だけを 16 桁にすることができます. Maxima では通常の数値から多倍長浮動小数点数への変換は bfloat 函数で行います.

多倍長浮動小数点数で任意桁の浮動小数点数の計算ができるとはいえ, 扱う対象は浮動小数点数です. したがって, 整数, 有理数や代数的数を浮動小数点数に変換する時点で「**切捨**」や「**丸め**」が生じ, 近似値としての性格を持っていることを忘れてはいけません.

同時に多倍長浮動小数点数の大きな欠点は, 通常の浮動小数点数と比較しても処理に時間がかかる

点です. 何故なら, Maxima の多倍長浮動小数点数は, LISP の多倍長浮動小数点数を使わずに, 自前で定義した独特の書式の多倍長浮動小数点数を利用するからです.

そのために C や FORTRAN, あるいは MATLAB といった数値計算に適したアプリケーションの方がよりよく処理を行うこともあります. ただし, Maxima を組合せて使えば, Maxima が得意とする数式処理によって式を簡易化し, それによって高速な数値計算が可能となります. その一例として horner 函数を使った例を挙げておきましょう. horner 函数は与式を「**Horner 則**」によって式中の積演算の回数を減らした式に変換する函数で, 数値計算の高速化や精度の面で大きな効果があります:

```
(%i11) expr1:(x+2*y)^5,expand;
                   5          4        2  3        3  2        4       5
(%o11)          32 y  + 80 x y  + 80 x  y  + 40 x  y  + 10 x  y + x
(%i12) horner(expr1);
                                                 2         3         4       5
(%o12)       y (y (y (y (32 y + 80 x) + 80 x ) + 40 x ) + 10 x ) + x
(%i13) fortran(%);
      y*(y*(y*(y*(32*y+80*x)+80*x**2)+40*x**3)+10*x**4)+x**5
(%o13)
```

この例では多項式 '(x+2y)^5' を ev 函数で展開した式 'expr1' に対して horner 函数を適用し, 適用した式に fortran 函数を適用することで FORTRAN の書式に変換しています. ここでは KNOPPIX/Math に収録されている Octave を使って効果を確認しましょう:

```
octave:1>  x=10.1;y=12.034;
octave:2>  a1=time();32*y**5+80*x*y**4+80*x**2*y**3+40*x**3*y**2+10*x**4*y+x**5;time()-a1
ans =  1.2207e-04
octave:3>  a1=time();y*(y*(y*(y*(32*y+80*x)+80*x**2)+40*x**3)+10*x**4)+x**5;time()-a1
ans =  1.0896e-04
```

この例では先程の展開した式と Horner 則を使った式の二つに 'x=10.1' と 'y=12.034' を代入した計算を行い, この計算に要した時間を time 函数を使って計測しています. ここで用いた式は 5 次の多項式ですが, それでも Horner 則を適用した式の方が処理が速いことが判ります. また演算回数を減らすことは速度の向上だけではなく, 演算のたびの生じる累積誤差を減らすことにも効果的です. このことを Maxima の結果と比較することで確認しましょう:

```
(%i2) expr1:expand((x+2*y)^5);
                   5          4        2  3        3  2        4       5
(%o2)           32 y  + 80 x y  + 80 x  y  + 40 x  y  + 10 x  y + x
(%i3) expr1,x=101/10,y=12034/1000;
                                  1421175975029930351
(%o3)                             -------------------
                                      30517578125
(%i4) float(%);
(%o4)                          4.6569094349780754e+7
```

ここでは変数 x, y に有理数を与え, ev 函数で評価した結果を float 函数を使って倍精度の浮動小数

点数に変換しています. もしも, 誤差が生じるとすれば, 最後の有理数から浮動小数点数に変換する時点だけです. だから精度が高いことが判りますね. この結果と Octave の結果を比べてみましょう:

```
octave:6>  eror1=32*y**5+80*x*y**4+80*x**2*y**3+\
>40*x**3*y**2+10*x**4*y+x**5-4.6569094349780754e+7
eror1  =   1.4901e-08
octave:7>  error2=y*(y*(y*(y*(32*y+80*x)+80*x**2)+\
>40*x**3)+10*x**4)+x**5-4.6569094349780754e+7
error2  =   7.4506e-09
```

この結果から Horner 則を用いた式の方がより良好な精度であることが判ります.

一般的に数式処理システムに数値計算をさせるよりも, 数値計算に適したツールで数値計算を行った方が処理速度に関しは有利です. しかし, 数式処理を利用して与式を効率良く計算できる式に変換すれば, 結果として全体の処理の高速化に結びつけることができるのです. その意味で, 数値計算ツールと数式処理は互いに補完し合うことかできるのです.

2.6　式の微分・積分

Maxima で数式の微分は diff 函数で行います. 一般的に, 微分は機械的な操作のため, その結果は積分よりも信頼できます:

```
(%i5) diff(x*y*sin(sqrt(x^2+y^2)),x);
                                          2              2    2
                          2    2       x  y cos(sqrt(y  + x ))
(%o5)          y sin(sqrt(y  + x )) + ------------------------
                                                  2    2
                                            sqrt(y  + x )
(%i6) diff(x*y*sin(sqrt(x^2+y^2)),x,n);
                        n
                       d                    2    2
(%o6)                  --- (x y sin(sqrt(y  + x )))
                         n
                       dx

(%i7) 'diff(x*y*sin(sqrt(x^2+y^2)),x,n);
                        n
                       d                    2    2
(%o8)                  --- (x y sin(sqrt(y  + x )))
                         n
                       dx
```

ここでは函数 $x\,y\sin\sqrt{x^2+y^2}$ の 1 階微分と n 階微分を計算しています. ただし, n 階微分で n の値が不明なために, そのままの形で返却されています. 最後の例は, 重要な演算子 “ ’ ” の例です. この演算子 “ ’ ” が付けられた項を「**名詞型項**」と呼び, Maxima では評価が行われない項になります. この様に, 項の先頭に演算子 “ ’ ” を付けると評価しない作法は Maple や *Mathematica* でも同様で

す. ちなみに, この演算子 “ ’ ” は LISP に由来します.

次に積分の例を示しましょう. 機械的な処理が行える微分と比較して, 積分は一筋縄では行かない難しさがあります.

Maxima では不定積分, 定積分の両方が integrate 函数を使って計算できます. この integrate 函数で物足りない場合, 「**Risch の積分アルゴリズム**」に基いた risch 函数が使えますが, この Risch の積分手法は完全に実装されたものではありません. ここでは integrate 函数を用いて不定積分と定積分を計算してみましょう:

```
(%i18) integrate((sqrt(x^2+1)),x);
                                                  2
                                asinh(x)    x sqrt(x  + 1)
(%o18)                          -------- + --------------
                                   2              2
(%i19) diff((%o18),x);
                        2                2
                 sqrt(x  + 1)          x                 1
(%o19)           ------------ + -------------- + --------------
                      2                 2                2
                                2 sqrt(x  + 1)   2 sqrt(x  + 1)
(%i20) ratsimp(%);
                                                 2
(%o20)                                   sqrt(x  + 1)

(%i21) integrate((sqrt(x^2+1)),x,1,2);

                   asinh(2) + 2 sqrt(5)    asinh(1) + sqrt(2)
(%o21)             -------------------- - ------------------
                            2                      2
(%i22) defint((sqrt(x^2+1)),x,1,2);
                   asinh(2) + 2 sqrt(5)    asinh(1) + sqrt(2)
(%o22)             -------------------- - ------------------
                            2                      2
```

全般的に, 積分は微分と比べて注意が必要です. 何故なら, 積分という処理は非常に難しい問題だからです. 基本的に, 多項式や初等函数の項の和で構成された式は大きな問題はないでしょう. しかし, 有理式や, 初等函数が入り組んだ式の積分になると, 式の変形の仕方次第で答が得られるかどうかが決まったり, もっともらしい誤った解を返却することさえあります. だから, 少しでも計算が面倒そうな式であれば, 積分した結果を微分して積分する前と一致するかを確認したり, 函数のグラフを描いて視覚的に確認することを薦めます. 特にグラフを使った可視化は, たとえば, 本来なら連続な函数なのに積分すると何故か非連続な函数になるといった誤った計算が視覚的に理解できます. この点についてはのちの章でも解説します.

2.7 方程式の解

Maxima の方程式は数式と同様に演算子 “=” で等値性を表現した式として与えられます. つまり, 方程式 $x^2+2x+1=0$ は Maxima では ‘x^2+2*x+1=0’ で表現されます. ここで, 方程式 $x^2+2x+1=0$ のように, 0 に等しいという方程式は, “= 0” を省略した式のみで表現することが許容されています. 従って, 方程式 $x^2+2x+1=0$ を表現する Maxima の式としては, ‘x^2+2*x+1=0’ に加え, ‘x^2+2*x+1’ という表記も認められています. また Maxima では連立方程式も扱えます. この場合は ‘[eq1, eq2]’ のように方程式を成分とする平坦なリストで表現します.

では, 実際に適当な解いてみましょう. そこで, 方程式の厳密解を計算してみましょう. Maxima では代数方程式の厳密解を求めるときは solve 函数が使えます:

```
(%i25) solve(x^2+2*x+1,x);
(%o25)                                [x = - 1]
(%i26) solve([x^2+2*x*y+y^2+1,y+2*x-1],[x,y]);
(%o26)[[x = 1 - %i,y = 2 %i - 1],[x = %i + 1,y = - 2 %i - 1]]
```

もし, 複素数の解が不要で, 実数解のみが必要であれば, realroots 函数を使えば, 実数解のみの計算を行います:

```
(%i3) realroots(x^8+x^3+x-5);
                                 43873333       37579631
(%o3)                      [x = - --------, x = --------]
                                 33554432       33554432
```

この例で示す様に, realroots 函数が返すのは, 実数に属する代数的数です. 浮動小数点数を返す函数ではありません.

連立方程式を解く場合, linsolve 函数 や algsys 函数が使えます. 特に, algsys 函数は厳密解が計算可能なら厳密解, 困難であれば浮動小数点数の数値近似解を計算する函数です:

```
(%i31) eq1:[x^2+y*x-1,x*y^2-x+y];
                                    2          2
(%o31)                       [x y + x  - 1, x y  + y - x]
(%i32) ans:algsys(eq1,[x,y]);
                                          sqrt(sqrt(2) + 2)
(%o32) [[x = sqrt(sqrt(2) + 2), y = - ------------------],
                                              sqrt(2)
                                  sqrt(sqrt(2) + 2)
[x = - sqrt(sqrt(2) + 2), y = ------------------],
                                      sqrt(2)
                                   sqrt(2 - sqrt(2))
[x = - sqrt(2 - sqrt(2)), y = - ------------------],
                                      sqrt(2)
                                sqrt(2 - sqrt(2))
[x = sqrt(2 - sqrt(2)), y = ------------------]]
                                      sqrt(2)
(%i33) eq1,ans[1]$
```

```
(%i34) trigsimp(%);
(%o34)                                [0, 0]
```

この例では連立方程式 $x^2 + yx - 1 = 0, xy^2 - x + y = 0$ を algsys 函数を用いて解き, 4 組の解の中の最初の組を使って検算を実行しています. ここで, 式 'eq1,ans[1]' は 'ev(eq1,ans[1])' の別表記で, ev 函数は引数から式の評価を行う函数です. ここでは, 実際に解を求めたのだから, その解の一つを使って, 方程式として与えた式が 0 になることを確認しているのです. 具体的には, 解のリスト ans の第 1 成分の値を, 連立方程式 eq1 を構成する方程式の変数 x, y に代入して, 各式が 0 になることを確認するものです. このような芸当が行えるのは, リスト ans の書式が, '[x=a, y=b]' のように演算子 "=" を用いたものになっているためです.
Maxima が解くことのできる方程式は, 通常の代数的方程式に限らず, 線形常微分方程式も解くことができます. たとえばバネの微分方程式

$$\frac{d^2x(t)}{dx^2} + Kx(t) = 0 \tag{2.1}$$

を Maxima で解いてみましょう:

```
(%i48) ode2('diff(x(t),t,2)+K*x(t)=0,x(t),t);
Is  K  positive, negative, or zero?

pos;
(%o48) x(t) = %k1 sin(t sqrt(K)) + %k2 cos(t sqrt(K))
(%i49) ic2(%,t=0,x(t)=0,'diff(x(t),t)=10);
                          10 sin(t sqrt(K))
(%o49)           x(t) = ------------------ + x(0) cos(t sqrt(K))
                              sqrt(K)
```

ここでは常微分方程式を解くために ode2 函数 を使っています. この ode2 函数は必要に応じて式の正負を尋ねてきます. ode2 函数の出力は微分方程式の一般解で, この一般解の中に "%k1" と "%k2" がありますが, これらは ode2 函数が定めた定数です. もちろん, 常微分方程式の初期条件を予め与えて特殊解を求めることもできます. 2 階の常微分方程式であれば, 初期条件を ic2 函数や bc2 函数で与えられます. ここの例では 'ic2(%,t=0,x(t)=0,'diff(x(t),t)=10)', すなわち, $x(0) = 0, x'(0) = 10$ という初期条件を与えています.

2.8　行列

Maxima で扱える行列の大きさと成分に制限はありません. 成分が数値でも多項式でも扱えますが, 行列処理は MATLAB のような数値行列処理ソフトと比較して速いものではありません. そのために数値行列の計算で多倍長浮動小数点数 (bigfloat 型の行列) で計算する必要がなければ MATLAB のような数値行列ソフトの利用を強く薦めます. この本では §16 で GNU の MATLAB と言える Octave の利用に触れているので Octave に興味があれば参照して下さい.
行列の定義方法には幾つかの方法があります. 一般的な方法は, matrix 函数で定義する方法です:

```
(%i13) A: matrix([1,2,3],[4,3,2],[5,1,3]);
                                  [ 1  2  3 ]
                                  [         ]
(%o13)                            [ 4  3  2 ]
                                  [         ]
                                  [ 5  1  3 ]
```

同じ大きさの行列の和や差は演算子 “+” と演算子 “-” で行えます. ただし, 可換積 “*” とその冪 “^” は行列に対する通常の行列の積や冪ではなく, 各成分同士の可換積や成分単位の冪となるので注意しましょう:

```
(%i15) A+B;
                                 [ 11  11  11 ]
                                 [            ]
(%o15)                           [ 5   5   5  ]
                                 [            ]
                                 [ 7   5   9  ]
(%i16) A*B;
                                 [ 10  18  24 ]
                                 [            ]
(%o16)                           [ 4   6   6  ]
                                 [            ]
                                 [ 10  4   18 ]
(%i17) A^2;
                                  [ 1   4  9 ]
                                  [          ]
(%o17)                            [ 16  9  4 ]
                                  [          ]
                                  [ 25  1  9 ]
```

ここで行列の積は非可換です. そこで Maxima では行列の積に非可換演算子, その冪に非可換演算子の冪を用います:

```
(%i18) A . B;
                                 [ 18  25  32 ]
                                 [            ]
(%o18)                           [ 47  50  53 ]
                                 [            ]
                                 [ 57  59  61 ]
(%i19) A^^2;
                                 [ 24  11  16 ]
                                 [            ]
(%o19)                           [ 26  19  24 ]
                                 [            ]
                                 [ 24  16  26 ]
```

```
(%i20) A .A^^(-1);
                                  [ 1  0  0 ]
                                  [         ]
(%o20)                            [ 0  1  0 ]
                                  [         ]
                                  [ 0  0  1 ]
```

逆行列の計算は invert 函数や演算子^^が使えます:

```
(%i21) invert(A);
                              [    7     1      1   ]
                              [ - --    --     -   ]
                              [    30    10     6   ]
                              [                     ]
                              [  1       2       1  ]
(%o21)                        [  --      -     - -  ]
                              [  15      5       3  ]
                              [                     ]
                              [  11       3     1   ]
                              [  --    - --    -   ]
                              [  30       10    6   ]
(%i22) A^^(-1);
                              [    7     1      1   ]
                              [ - --    --     -   ]
                              [    30    10     6   ]
                              [                     ]
                              [  1       2       1  ]
(%o22)                        [  --      -     - -  ]
                              [  15      5       3  ]
                              [                     ]
                              [  11       3     1   ]
                              [  --    - --    -   ]
                              [  30       10    6   ]
```

ここで Maxima で行列の固有値の計算では eigen パッケージを用います:

```
(%i5) load(eigen);
(%o5)          /usr/local/share/maxima/5.9.2/share/matrix/eigen.mac
(%i6) A:matrix([1,2,0],[2,1,1],[0,1,1]);
                                  [ 1  2  0 ]
                                  [         ]
(%o6)                             [ 2  1  1 ]
                                  [         ]
                                  [ 0  1  1 ]
(%i7) eigenvalues(A);
(%o7)                [[1 - sqrt(5), sqrt(5) + 1, 1], [1, 1, 1]]
(%i8) eigenvectors(A);
                                                          sqrt(5)  1
(%o8) [[[1 - sqrt(5), sqrt(5) + 1, 1], [1, 1, 1]], [1, - -------, -],
                                                            2      2
                                              sqrt(5)  1
                                        [1, -------, -], [1, 0, - 2]]
                                            2      2
```

```
(%i9) charpoly(A,t);
                                    2
(%o9)                  ((1 - t)  - 1) (1 - t) - 4 (1 - t)
```

固有値の計算は eigenvalues 函数, 固有ベクトルの計算は eigenvectors 函数で行えます.
また特性多項式の計算は charpoly 函数 で行えます.
ここで Maxima には eigen 以外にもさまざまなパッケージがあり, Maxima の立ち上げ時に読込まれるものも幾つかありますが, 立ち上げ時に読込まれないパッケージを利用する場合, この例の eigen パッケージのように load 函数を用いて読込む必要があります.

2.9 FORTRAN や TEX への出力

Maxima の式等の対象は FORTRAN の書式や TEX の書式に変換できます. ここで FORTRAN の書式に式を変換する函数は fortran 函数, TEX の書式に変換を行う函数は tex 函数です. これらの函数は変換すべき式のみを引数に取ります:

```
(%i12) expr1:expand((x+2*y)^5);
                   5          4        2  3        3  2        4       5
(%o12)        32 y  + 80 x y  + 80 x  y  + 40 x  y  + 10 x  y + x
(%i13) fortran(expr1);
      32*y**5+80*x*y**4+80*x**2*y**3+40*x**3*y**2+10*x**4*y+x**5
(%o13)                                 done
(%i14) tex(expr1);
$$32\,y^5+80\,x\,y^4+80\,x^2\,y^3+40\,x^3\,y^2+10\,x^4\,y+x^5$$
(%o14)                                 false
```

最初の fortran 函数の結果から判るように冪は記号 “**” で置換えられます. このように, FORTRAN で扱えない Maxima の演算や表記が与式に含まれていれば, fortran 函数は, その変換すべき対象が予め持っている属性値を使って置換を行います. ここで, 変換に用いる属性値がなければ, Maxima の対象のままで返却します. 従って, 自前の演算子を定義して, fortran 函数で変換させる表現を属性で定義していなければ, その自前の演算子がそのまま返却されます. この処理方法は tex 函数でも同様で, 与式に含まれる対象の属性値を見ながら与式を TEX のソースコードに変換し, 全体を記号 “$$” で括ったものを返却します:

tex 函数で変換した結果を利用

$$32\,y^5+80\,x\,y^4+80\,x^2\,y^3+40\,x^3\,y^2+10\,x^4\,y+x^5$$

これらの函数の他に Maxima には利用者定義の函数を LISP の函数に変換する translate 函数, この translate 函数で変換した函数を LISP コンパイラでコンパイルして函数の最適化を図る compile 函数もあります. 詳細は §8.4 を参照して下さい.

2.10　グラフ表示

Maxima は 2 次元グラフや 3 次元グラフの表示が可能です. グラフ表示を行う函数の中で最も多機能なものに plot2d 函数と plot3d 函数, そして, draw パッケージに含まれる draw2d や draw3d 函数が挙げられます. ここでは幅広い環境で直感的に使える plot2d 函数と plot3d 函数について簡単に解説しておきましょう.

最初に plot2d 函数を使った 2 次元グラフの簡単な例を示します. この plot2d 函数では, X 座標と Y 座標の関係が $y = f(x)$ で定まる函数 $f(x)$, X, Y 座標が媒介変数 t を使って $x = f_1(t), y = f_2(t)$ の関係を持つときの軌跡を描くことができます. 基本的な例として正弦函数を区間 $[-2\pi, 2\pi]$ で描いてみましょう.

この場合は `plot2d(sin(x),[x,-2*%pi,2*%pi]);` と入力すると xmaxima+openmath 以外であれば別ウィンドウが開かれてグラフが表示されます:

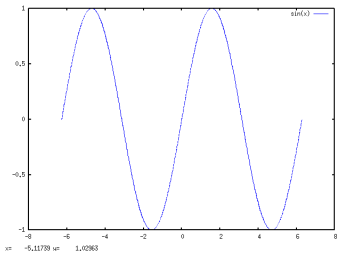


図 2.5: gnuplot による正弦函数の表示

次に $[\cos(t), \sin(t)\cos(t)]$ のグラフを媒介変数表示で描いてみましょう. ここで曲線の点数が少ないと綺麗な絵が描けないことがあるので set_plot_option 函数の nticks を適切な値, ここでは 100 にしましょう:

```
(%i5) set_plot_option([nticks,100]);
(%o5) [[x, - 1.75555970201398e+305, 1.75555970201398e+305],
[y, - 1.75555970201398e+305, 1.75555970201398e+305], [t, - 3, 3],
[grid, 30, 30], [transform_xy, false], [run_viewer, true], [axes, true],
[plot_format, gnuplot_pipes], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 100], [adapt_depth, 5],
[gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]], [gnuplot_curve_styles,
[with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,
with lines 6, with lines 7]], [gnuplot_default_term_command,
set term x11 font "Helvetica,16"], [gnuplot_dumb_term_command,
set term dumb 79 22], [gnuplot_ps_term_command,
set size 1.5, 1.5;set term postscript eps enhanced color solid 24],
[plot_realpart, false]]
```

この設定で `plot2d([parametric,cos(t),cos(t)*sin(t),[t,-5,5]],[x,-3,3]);` を実行した結果が図 2.6 になります:

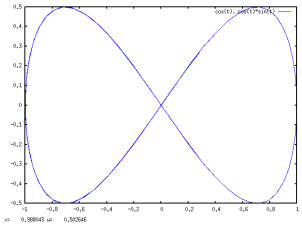


図 2.6: gnuplot によるリサージュ曲線の表示

3 次元グラフ表示では plot3d 函数を用います. ここではマニュアルにもある「Klein の壷」を代表的な外部アプリケーションの gnuplot, openmath と Geomview で描いてみましょう. この Klein の壷を図 2.7 に示しておきます:

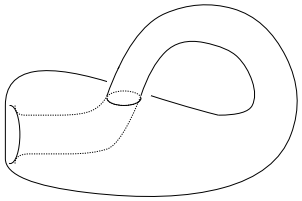


図 2.7: Klein の壷

この Klein の壷で自分自身が交わっているように見える個所は 4 次元空間であれば, その個所だけを違う時間に移動させることができるので, 4 次元空間内では自己交差はありません. しかし, 3 次元空間内ではどうしても自己交差が生じます. この Klein の壷の作り方を図 2.8 に示しています. この作り方は矢印 A に沿って長方形を貼り合せて円筒を作り, それから円筒の両端を矢印 B に沿って貼り合せるというものです. もしこの矢印 B の一方の向きが逆であればドーナッツの表面, すなわちトーラス:$S^1 \times S^1$ になりますが, 互いに逆向きなので表からではなく図 2.7 のように裏側から貼り合せます:

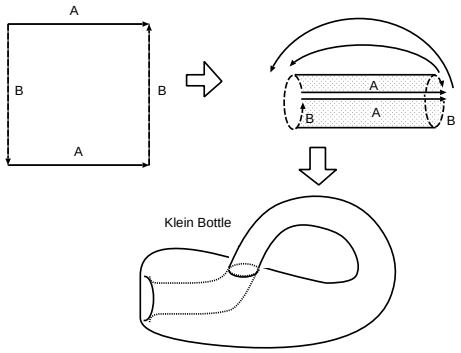


図 2.8: Klein の壷の構成方法 (その 1)

この図 2.8 とは別の構成方法もあります. その構成方法を図 2.9 に示します:

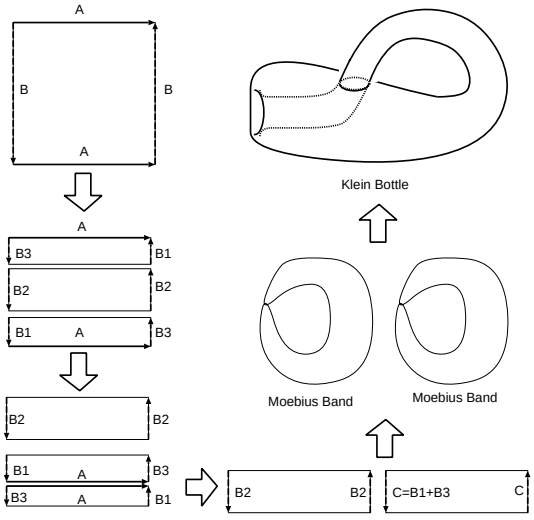


図 2.9: Klein の壷の構成方法 (その 2)

この方法は長方形を矢印 A に平行な 3 個の短冊に分割します. それから上と下の短冊は矢印 A に沿って貼り合せると二つの短冊ができます. この短冊の両端の矢印 B2 や C に沿って短冊を捻じって貼り合せると二つの「Möbius の輪」と呼ばれる曲面が得られ, これらの Möbius の輪を境界で貼合せることで Klein の壷が得られます. Klein の壷と Möbius の輪は向付けができない曲面, すなわち裏表がない曲面の著名な一例です.

では Klein の壷を図 2.9 の方法で gnuplot, openmath と Geomview を使って次の入力式で描いてみます:

Klein の壷を描く Maxima の式

```
plot3d([5*cos(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3.0)-10.0,
 -5*sin(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3.0),
   5*(-sin(x/2)*cos(y)+cos(x/2)*sin(2*y))],
[x,-%pi,%pi],[y,-%pi,%pi],['grid,40,40])
```

ここで描画アプリケーションの切替は set_plot_option 函数で行います:

描画アプリケーションの切替

アプリケーション	切替のための入力式
gnuplot	set_plot_option([plot_format,gnuplot])
openmath	set_plot_option([plot_format,openmath])
Geomview	set_plot_option([plot_format,geomview])

2.10.1 gnuplot による Klein の壷

gnuplot は Maxima で利用可能な外部表示アプリケーションの中では最も高機能で良好な画が表示可能なものの一つです. さらに gnuplot は一寸したグラフ計算機として利用することさえもできます:

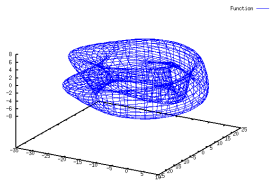


図 2.10: gnuplot による Klein の壷

ここで面を付けてもう少し格好良くすることもできます:

gnuplot で面付き Klein の壷を描く

```
plot3d([5*cos(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3.0)-10.0,
 -5*sin(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3.0),
   5*(-sin(x/2)*cos(y)+cos(x/2)*sin(2*y))],
[x,-%pi,%pi],[y,-%pi,%pi],['grid,40,40],
[gnuplot_pm3d,true],[gnuplot_preamble,"unset surf"]);
```

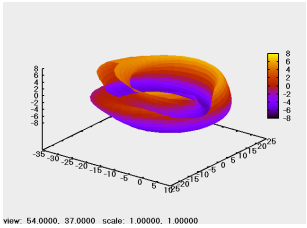


図 2.11: gnuplot による面付き Klein の壷

図 2.11 の一部の面が消えていますが, これは表面が見えるように自動的に裏面を除去しているためです. ここでのオプションの意味と使い方は §11.2 や §11.5 を参照して下さい.

2.10.2 openmath による Klein の壷

openmath は標準で Maxima に附属し, 割と高機能で綺麗な描画が描けるだけでなく, 視点の変更もマウスで直接操作ができます:

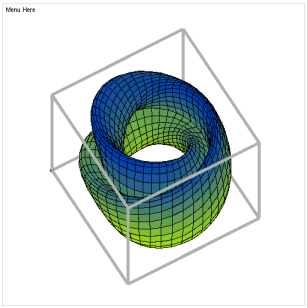


図 2.12: openmath による Klein の壷

ウィンドウ左上 (立ち上げ時には Menu Here と表示されており, そののちは座標値が表示されている個所) にマウスを移動させてマウスの左ボタンをクリックすればメニューが現われます. 描画を止めたければメニューの最上段の Dismiss を選択します.

2.10.3　Geomview による Klein の壷

Geomview はミネソタ大学の Geometry Center で開発されたツールで, plot3d が扱える外部アプリケーションの中では最も大規模で描画も美しいものです:

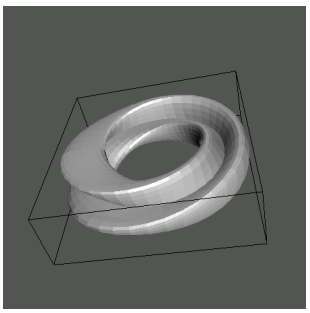


図 2.13: Gemoview による Klein の壷

描画を開始するとグラフと 2 個の制御パネルが現われ, 視点の変更は長細い制御ウィンドウを用います. この Geomview はさまざまな幾何学的対象の可視化ツールであり, モジュールで機能の拡張が行えます. 公開されているモジュールには面白いものが幾つかあります. たとえば図 2.14 に示す太陽系シミュレータ Orrey もその一つです:

2.10.4　番外編

グラフ表示に関して簡単に解説しましたが, Maxima はグラフデータを生成して外部のアプリケーションに引渡すことだけです. だから, あるグラフ表示アプリケーションの入力データ書式が判って

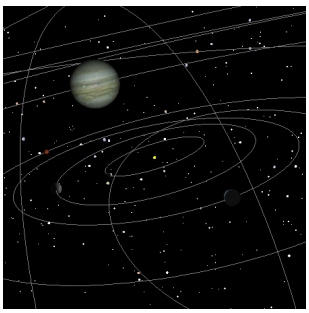


図 2.14: 太陽系シミュレーター Orrey

いれば, Maxima でその書式のデータファイルを生成したのちに system 函数を使ってアプリケーションにデータファイルを読込ませてしまえば良いのです. この手法の簡単な例として §15 に surfer を用いた代数曲線や曲面の可視化について解説しています.

2.11 ファイル

Maxima は Common Lisp で記述され, 入出力は LISP の入出力函数を利用しています. そのために LISP の入出力函数の縮小版の傾向があります. ファイルに画面と同じ出力を行いたければ writefile 函数を用いますが, この writefile 函数は LISP の dribble 函数を用いた函数で, 入力と出力をそのまま指定したファイルに保存します. この writefile 函数は指定したファイルを新規に生成するために単純に既存のファイルに記録したければ, appendfile 函数を用います. writefile 函数や appendfile 函数で開いたファイルを閉じる場合は closefile() で開いたファイルを閉じます.
実際に試してみましょう:

```
(%i1) writefile("maxima.tmp");
(%o1)      #<OUTPUT BUFFERED FILE-STREAM CHARACTER maxima.tmp>
(%i2) integrate(sqrt(x^2+1),x);
                                                  2
                                asinh(x)    x sqrt(x  + 1)
(%o2)                           -------- + --------------
                                   2              2
(%i3) closefile();
(%o3)     #<CLOSED OUTPUT BUFFERED FILE-STREAM CHARACTER maxima.tmp>
(%i4) ratsimp(diff(%o2,x));
                                                2
(%o4)                                   sqrt(x  + 1)
```

次に writefile 函数で指定した maxima.tmp ファイルの中身を確認してみます:

```
;; Dribble of #<IO TERMINAL-STREAM> started 2005-12-21 06:41:28
(%o1)          #<OUTPUT BUFFERED FILE-STREAM CHARACTER maxima.tmp>
(%i2) integrate(sqrt(x^2+1),x);
```

```
                                                  2
                                asinh(x)    x sqrt(x  + 1)
(%o2)                          -------- + --------------
                                   2             2
(%i3) closefile();
;; Dribble of #<IO TERMINAL-STREAM> finished 2005-12-21 06:41:44
```

このように writefile 函数を実行した時点から closefile 函数を実行した時点までの入出力がそのままファイルに書込まれています. ただし, これらのファイルは実質的に記録ファイルであって, Maxima でそのまま再利用できません. 再利用可能なファイルを生成するのは, save 函数と stringout 函数, そして grind 函数です.
ここで save 函数は Maxima の内部表現を保存する函数で, load 函数や loadfile 函数を用いて Maxima に読込めます.
これに対して stringout 函数や grind 函数は式の内部表現ではなく, Maxima の入力にそのまま対応する書式のデータ保存を行います.
ここで内部表現と呼んでいますが save 函数で生成したファイルの中身はどのようなものでしょうか？ そこで最初に以下の式を入力して最後に save 函数でファイル test に Maxima の内部データ全てを保存します:

```
(%i1) 1+2+3;
(%o1)                                  6
(%i2) a1:x^2+y^2+1;
                                    2    2
(%o2)                              y  + x  + 1
(%i3) resultant(x-t,y-t^2,t);
                                          2
(%o3)                                y - x
(%i4) save("test",all);
(%o4)                                 test
```

以下に test ファイルの内容を先頭の部分だけを示しておきましょう:

```
;;; -*- Mode: LISP; package:maxima; syntax:common-lisp; -*-
(in-package "MAXIMA")
(DSKSETQ $%I1 '((MPLUS) 1 2 3))
(ADDLABEL '$%I1)
(DSKSETQ $%O1 6)
(ADDLABEL '$%O1)
(DSKSETQ $%I2 '((MSETQ) $A1 ((MPLUS) ((MEXPT) $X 2)
                ((MEXPT) $Y 2) 1)))
(ADDLABEL '$%I2)
(DSKSETQ $%O2 '((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2)
                ((MEXPT SIMP) $Y 2)))
(ADDLABEL '$%O2)
(DSKSETQ $%I3
 '(($RESULTANT) ((MPLUS) $X ((MMINUS) $T))
   ((MPLUS) $Y ((MMINUS) ((MEXPT) $T 2))) $T))
(ADDLABEL '$%I3)
(DSKSETQ $%O3
 '((MPLUS SIMP) ((MTIMES SIMP) -1 ((MEXPT SIMP RATSIMP) $X 2))
                                  $Y))
(ADDLABEL '$%O3)
(DSKSETQ $%I4 '(($SAVE) &TEST $ALL))
```

```
22 (ADDLABEL '$%I4)
23 (DSKSETQ $A1 '((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2)
24                                ((MEXPT SIMP) $Y 2)))
25 (ADD2LNC '$A1 $VALUES)
26 以下略
```

如何ですか？ Maxima で入力したもの以外のさまざまな設定で膨れ上っていますね. それに '(MSETQ)' とか何か怪しいものがいろいろありますが, この括弧だらけのデータは何でしょう？ これは LISP の S 式と呼ばれるデータで, プログラムでもありますが, このファイルの内容を知るためには LISP の大雑把な知識が少しあればよいのです.

以上で Maxima の簡単な入門を終えます. 次の章では Maxima の根底にある計算機言語 LISP について簡単に紹介しましょう.

第3章　LISPについて

Maxima は LISP で記述されています. そのために単純にマニュアルを読むときでさえも LISP の知識の有無によって理解の度合が異なるのが実情です. この章では LISP について簡単に解説します.

3.1 背景

この本は数式処理システム Maxima の本です. だから Maxima の使い方だけを解説すれば良い筈で, 貴重な紙面を括弧だらけの変な言語の解説でただでさえ厚い本を膨らませる必要がなく, 紙面の無駄だと怒る方もいるかもしれませんが, ここは我慢して暫く私につきあって下さい.
実際, LISP を知っていると Maxima をより深く理解できるだけではなく, ちょっとした修正さえもできます. これは修正したソースファイルを load 函数で読込ませるだけでコンパイルは不要なのです. この辺が商用の *Mathematica* や Maple とは違った大きな特徴であり, 大きな長所なのです.
ここでは最初に LISP の概要をその歴史から簡単に解説しましょう.
LISP の歴史は非常に古く, FORTRAN よりも僅かに新しい言語です. FORTRAN が数値計算向けであるのに対し, LISP(LIStProcessor) はリスト (list) と呼ばれるデータを扱う事を目的としています. LISP には “car” とか “cdr” といった風変りな名前の函数があります. これらの函数名は LISP の歴史を語っているものなのです. 最初の LISP は FORTRAN で記述されて IBM の計算機 704 で動作するもので, “car” や “cdr” はその計算機の CAR 命令 (Contents of Address Register) と CDR 命令 (Contents of Decrement Register) に由来します. John McCarthy が λ-計算で用いる言語として発表した論文「Recursive Functions of Symbolic Expression and their Computation by Machine (Part I)」にて導入された時点では car と cdr の他に cons, atom と eq の五つの基本函数がある程度です ([87] 参照). そこから LISP は成長したのです.
LISP のプログラムの明瞭な特徴は括弧 “( )” ばかりが目立つ言語でしょう. ところが, 原子とリストから再帰的に構築される S 式[1] を用いることて非常に柔軟に問題に対処できるだけではなく, プログラム自体も S 式であるという特徴, さらに λ-記法が使えるといった数学基礎論の考え方も実装し易くて美しい言語です. そんなこともあって古い言語でありながら, いつになっても新しく, 魅力的な言語になっています. LISP は実装にも依存しますが, SBCL のようにインタプリタ言語としても非常に高速なものもあります. その上, 問題によってはC よりも効率良く, その上, 高速に処理が行えることもあります. また 1980 年代に人工知能が注目を集めていた時期に LISP は人工知能[2]で幅広く用いられており, Symbolic Inc. からは LISP 専用の計算機さえも出ていた程です.
この LISP は言語的に非常に柔軟性が高いので, GNU Emacs や AutoCAD のように独自の LISP を処理言語として用いることもあります[3]. その一方で方言の多い言語です.
現在のように LISP の標準化が行われる以前の 1970 年代には MIT の MAC 計画で開発された MACLisp[4] と Bolt Beranek and Newman Inc. と Xerox Palo Alto 研究所で開発された InterLisp が主要な方言でした. ちなみに 1975 年には LISP の主要な方言の一つである Scheme が開発されています.
やがて 1980 年代には「Common Lisp: The Language 第 1 版 (1984)」を基に京都大学で「KCL(Kyoto Common Lisp)」が作成されます. この KCL がのちに GNU に寄贈されて「**GCL**」の原型になります. 現在の LISP の方言として主要なものは Common Lisp と Scheme の二系統です.

---

[1] FORTRAN 風の M 式というものもあります. M 式を使った本には シャイテンの本があります.
[2] 当時は人工知能記述のためのアセンブラとも呼ばれていました. 昔は計算機の記憶容量が小さかったために, C よりもアセンブラ言語の方が広く用いられていました.
[3] Emacs の場合は寧ろ LISP で記述されていると言った方が正確でしょう
[4] MACLisp は Macintosh とは無関係の LISP です. MACLisp は MACSYMA の記述に用いられただけではなく MACSYMA からのフィードバックもありました

以下の節では Common Lisp の中では非常に多くの計算機環境で利用可能な CLISP を中心に LISP の解説をしますが初歩的な水準の解説でしかないので, GCL その他でも大きな違いは出てこないでしょう.

3.2 数値, 文字列

では, 早速 Common Lisp で遊んでみましょう. CLISP 等の Common Lisp が使える環境であれば LISP を立ち上げてみて遊んで下さい. ここでは CLISP の結果を主に載せています. また Maxima がインストールされている環境であれば Maxima を起動して Maxima に `to_lisp();` と入力して下さい. すると裏に隠れていた Common Lisp が表に出てきます[5]:

```
(%i1)  to_lisp  ();
Type (to-maxima) to  restart , ($quit) to  quit  Maxima.

MAXIMA> (+ 1 2)
3
MAXIMA> (append '(1 2 3) '(4  5  6))
(1  2  3  4  5  6)
MAXIMA> (to-maxima)
Returning  to  Maxima
(%o1)                                   true
```

この例で示すように Maxima の to_lisp 函数で Maxima の裏で動作する LISP を表に出すとプロンプトが `MAXIMA>`に切替わり, `(to-maxima)` と入力するまで LISP が表に出ています. この LISP が表に出た状態で `($quit)` と入力すれば LISP を含めて全体を終えます. この '($quit)' は Maxima 上の 'quit();' と同じ意味になりすが, このことは Maxima の式の内部表現に深く関連します (§6.4 参照). それでは LISP の環境で数値や文字列を入力してみましょう:

```
[1]>  1
1
[2]>  1234
1234
[3]>  12/984
1/82
[4]>  0.01
0.01
[5]>  "abc"
"abc"
[6]>  "abc  def/234"
"abc  def/234"
[7]>  #\a
#\a
```

この例では整数, 分数, 浮動点小数と文字列と文字の入力を行っています. LISP の分数では分母と分子は共に整数でなければなりません. また分数は自動的に約分されます. 文字列は単純に二重引

[5]Sourceforge から Maxima のバイナリ版を入手されている方ならば GCL になります.

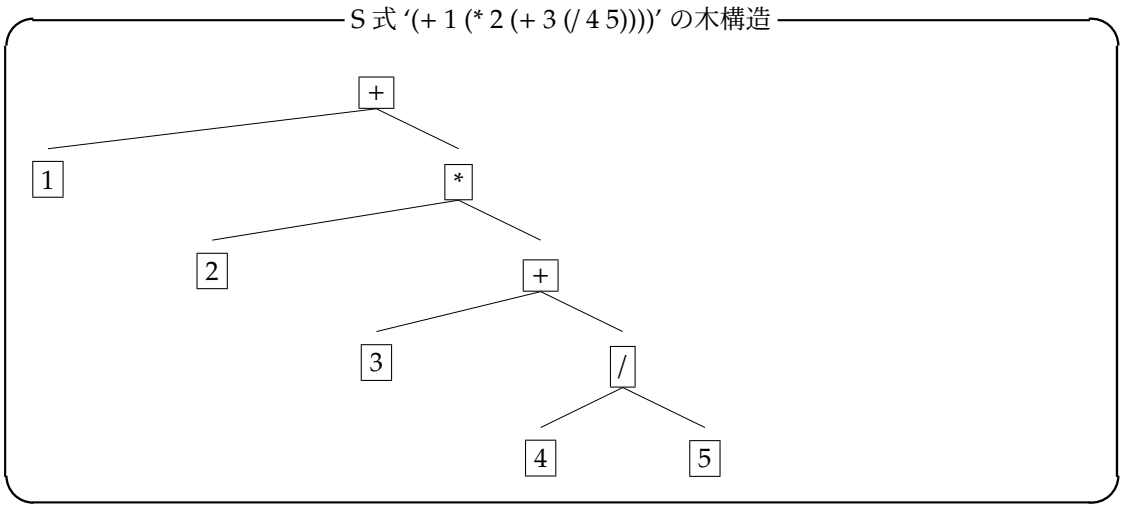
用符 “’” で括った対象になりますが, 文字は ‘#\’ のように記号 “#\” を文字の先頭に付けます. LISP では整数等の数値や文字や文字列の事を原子と呼びます. この原子が文字通り全てのデータを構成する原子となり

> 原子はリストを生成し, リストは S 式を生成し, S 式は万物を生成する

といった有様になります.
さて, 複数の数値の和, 積, 商は LISP ではどう表現するのでしょうか. C や FORTRAN では共に ‘1+2+3+4’, ‘1*2*3*4’ や ‘1/2/3/4’ と記述しますが, LISP では次の表記となります:

```
[7]>  (+  1  2  3  4)
10
[8]>  (*  1  2  3  4)
24
[9]>  (/  1  2  3  4)
1/24
```

LISP ではこのように演算子や函数名を先頭に置きます. 演算子や函数名を前に置く表現方法を「前置表現」と呼びます. 通常の $1+2$ のような演算子の置き方を「内挿表現」, あるいは「中置表現」と呼びます. 参考までに「前」・「内」とあれば勿論「後」の「後置表現」もあります. この後置表現で有名な計算機言語に Abobe の PostScript(略して,PS) があります. PostScript は画像データの印象が強いかもしれませんが, 実際はれっきとしたプログラム言語で, スタック型言語と呼ばれる言語に属します. このスタック型言語は手続型言語の一種類で, スタックマシンを実行モデルとする言語です. PostScript の他には Forth や KNOPPIX/Math に収録されている kan/sm1 もそうです.
閑話休題, 上の例では非常に簡単な例を示しましたが, より一般的な数式は括弧 “( )” を使って $1+2*(3+4/5)$ のように括られています. この表記を Lisp では ‘(+ 1 (* 2 (+ 3 (/ 4 5))))’ と括弧 “( )” を入れ子にして記述します.
この考え方は演算子 “+” や “*” が二項演算子であることから小括弧 “( )” を使って分けて行けば判り易くなります. 最初に $1+2*(3+4/5)$ は $1+(2*(3+(4/5)))$ へと括弧 “( )” を使って纏めることができます. この式に対して括弧の部分を外側, あるいは内側から前置表現に置換えることで S 式 ‘(+ 1 (* 2 (+ 3 (/ 4 5))))’ が得られます. この前置表現を階層的な木構造として表現したものを次に示しておきます:



このリストの階層構造は非常に重要です. 内部では Maxima の数式も前置表現のリストで表現され, この内部表現に沿って式の簡易化や値の代入等の処理が実行されるからです.
ここで LISP では演算子だけではなく函数も前に置きます. つまり, 函数 f の作用 $f(x_1, \ldots, x_n)$ を LISP では $(f\ x_1 \ldots\ x_n)$ と表記します.

3.2.1 LISP の代表的な数値函数

Common Lisp で利用可能な数値函数に次のものがあります:

LISP の標準的な数値函数

演算	演算子および函数
四則演算:	"+", "-", "*", "/"
剰余:	mod
三角函数:	cos, sin, tan, acos, asin, atan
指数及び対数函数:	exp, log

これらの函数の書式は全て先頭に函数名があり, その後に数値が続きます.
ここで数値函数の例として LISP の組込函数の sin 函数を試してみましょう:

```
[9]> (sin pi)
-5.016557613684336024l-20
[10]> (sin (/ pi 2))
1.0 l0
```

この例では $\sin(\pi)$ と $\sin(\pi/2)$ の値を計算しています. $\pi/2$ の個所で函数表記が入れ子になっていることに注意して下さい. ここで "(/ pi 2)" の代りに "pi/2" と入力するとエラーになります. なぜなら分数の分母と分子は両方が整数でなければならないことに加え, 演算子 "/" が前置表現の演算子だからです. このように LISP では式は括弧 "( )" だらけになります. そのために LISP は Lots of Irritating Superfluous Parentheses の略記であるという冗談さえもあります.

この括弧 “( )” で括られた対象を LISP でリストと呼び, LISP の名前の由来はこのリスト処理 (LISt Processing) から来ています. リストにはリストを含んだり, 後述の配列やハッシュ表を混在させることができます. そして原子やリストを併せて S 式と呼びます.
他に Common Lisp で使える数に複素数もあり, 複素数 $a + ib$ を LISP では ‘#c(a b)’ で表現します:

```
[26]> (* #c(1 2) #c(2 -1))
#c(4 3)
[27]> (* #c(2 1) #c(2 -1))
5
[28]> (* 2 #c(2 1))
#c(4 2)
[29]> (* #c(0 1) #c(2 1))
#c(-1 2)
[30]> (- #c(1 0) 1)
0
[31]> (= #c(1 0) 1)
t
```

ここで S 式 ‘#c(a 0)’ は実部 a と同じです. 最後の二つの例では一つは 1 との差を取ることで, もう一つは演算子 “=” を用いてることで等値性を確認しています. 最後の演算子 “=” で調べた結果でが返却されていますが, この ‘t’ は LISP では特別な意味を持ち, Boole 代数の「真」を意味します. 一方の偽は ‘nil’ と記述されますが, この ‘nil’ は真理値の「偽」の他に「空リスト」, 「EOF」(=end of file) を表現したりと様々な場面で, どちらかと言えば否定的な意味で用いられています.
LISP の文字データは文字列とは別のデータ型です. たとえば, 文字 “a” は ‘#\a’ のように先頭に記号 “#\” を付けて表現されますが, 文字列は ‘”abcd”’ のように文字の羅列を二重引用符 “”” で括ります. このように文字列の表現自体は C や FORTRAN と同じです. また文字型も string 函数を用いれば文字列に変換できます.
文字列の同士の結合は concatenate 函数を用います. concatenate 函数の実例を示しておきましょう:

```
[23]> (concatenate 'string (string #\a))
"a"
[24]> #\a
#\a
[25]> (string #\a)
"a"
[26]> (code-char 65)
#\a
[27]> (char-code #\a)
65
[28]> (concatenate 'string (string #\a) (string #\b))
"ab"
[29]> (concatenate 'string "abc" "123")
"abc123"
```

この例では最初に string 函数を用いて文字 ‘#\a’ を文字列 ‘”a”’ に変換しています. それから code-char 函数で整数値から文字への変換を, char-code 函数はその逆操作になります. 最後の例では concatenate 函数を用いた文字列の結合を行っています.

3.3 リスト

LISP はリストと呼ばれる構造を持った与件の処理を主な目的としています. リストの基本的な形は '(1 2 34 'abcd)' のように空行で区切られた原子を括弧で括った対象ですが, リストのリストといった複合リストもリストです. リストは C の配列と似てなくもありませんが, C の配列のように最初に宣言した型に縛られない, 非常に柔軟な構造を持っています.
LISP では一般の函数も '(+ 1 2 34)' のようにリストの先頭に配置することが大きな特徴です. ちなみにこの式の意味は第一成分の "+" が第 2 成分以降を引数とする函数で, 具体的には 1 + 2 + 34 を意味します.
LISP のプログラムは簡単に言ってしまえば函数の集合になります. 函数がリストとなっているので LISP もリストの集合, 正確に言えば S 式で構成されています. そのために函数を定義域に取る函数, すなわち, 「汎函数」が簡単に定義できます.
たとえば, 1 から指定された自然数 n までのリストを生成し, そのリストに函数 f を作用させる函数 example1 を次で定義します:

```
1 (defun example1 (f n)
2   (let ((a  nil ))
3     (dotimes(i n)
4       (setq  a  (append a (list  (+ i   1)))))
5     (apply f  a )))
```

函数定義では defun 函数が使えます. 詳細はあとの節で解説します. この函数による処理例を以下に示します:

```
[36]>  (example1 '+  10)
55
[37]>  (example1 '*  10)
3628800
[38]>  (example1 '/  10)
1/3628800
[39]>  (example1 '-  10)
-53
[40]>  (example1 ' list   10)
(1  2  3  4  5  6  7  8  9  10)
```

このように C と随分と雰囲気が違うことが判るかと思います. ここで C や FORTRAN は手続型言語と呼ばれる範疇に入ります. 実際, プログラムは計算機に実行させる手続を順番に記述する形になっています. これに対して LISP は函数型と呼ばれる言語で, 構築した函数を組合せてプログラムを組立てる形になります.
以降で基本的な LISP の命令を幾つか紹介しましょう. ここで紹介する命令は他のリストを扱える数式処理でも実装されている事が多いので覚えておいても損はないと思います.

- car
  リストの先頭の要素を取り出します:

```
[1]> (car '(1 2 3 4 54))
1
```

- cdr
  リストの先頭の要素を除いたリストを返します. なお, 空のリストの値は nil になります:

```
[2]> (cdr '(1 2 3 4 5))
(2 3 4 5)
[3]> (cdr (cdr (cdr '(1 2 3 ))))
nil
```

- append
  複数のリストを結合して一つのリストにします. 新しく生成されたリストは, 基となる各リストの成分から構築されます. 要するに各リストの一番外側の括弧を外して, リストを繋いで行く感じになります:

```
[3]> (append '(1 2 3 4) '(2 3 4))
(1 2 3 4 2 3 4)
[4]> (append '(1 2 3 4) '(5 6 7 ) '(' a 'b 'c 'd))
(1 2 3 4 5 6 7 'a 'b 'c 'd)
[5]> (append '(1 2 3 4) '(5 6 7 ) '(1 2 3 ('a 'b 'c 'd)))
(1 2 3 4 5 6 7 1 2 3 ('a 'b 'c 'd))
```

- cons
  二つのリストを結合して新しいリストを生成します. たとえば, 二つのリスト a と b から cons で生成したリストに対し, car を作用させると a, cdr を作用させると b が戻るように結合したリストを返却します:

```
[6]> (cons '(1 2 3 4) '(5 6 7 ))
((1 2 3 4) 5 6 7)
[7]> (car (cons '(1 2 3 4) '(5 6 7)))
(1 2 3 4)
```

他に caar, cadr や cddr のように car と cdr の組み合わせた函数があります. 最初の caar は二度引数に car を作用させる函数と同値です. 次の cadr は最初に cdr を作用させて, その結果に car を作用させます. 最後の cddr は何でしょうか？ 答は cdr を二度作用させる事と同値な函数になります.

3.4　t と nil

LISP での真偽値は真であれば 't', 真でなければ 'nil' を用います. この 'nil' は LISP では何かと良く使う便利な値です. たとえば, 空のリストも 'nil' と表現します:

```
[2]> (> 1 3)
NIL
[3]> (> 3 2)
T
```

評価によって真理値を返す S 式を述語と呼びます. 述語は論理和 or や論理積 and で結合することができます:
論理和 “or” は (or ⟨ 述語$_1$⟩ ⟨ 述語$_2$⟩ ⋯) のように用い, 述語の内の何れか一つが ‘t’ であれば ‘t’ を返し, それ以外は ‘nil’ を返します.
論理積 “and” は (and ⟨ 述語$_1$⟩ ⟨ 述語$_2$⟩ ⋯) のように用い, 全ての述語が ‘t’ の場合のみ ‘t’ を返し, それ以外は ‘ni” を返します.
LISP には述語を用いた条件分岐として if 式と cond 式があります. 次の例では不等号を用いて大小関係を評価した結果を示しています：

```
[10]> ( if (< 3 2)( print 'neko)( print 'mike))

mike
mike
[11]> ( if (> 3 2)( print 'neko)( print 'mike))

neko
neko
[12]> (setq a 128)
128
[13]> (cond ((> a 10)( print 'mike))
((a <=10)(print 'tama )))

mike
mike
```

3.5 配列

LISP は配列が扱えます. 配列の生成は make-array 函数を用い, 生成した配列に既定値を与えたければ, オプションキーワード “:initial-element” を用いて既定値の設定が行えます. これはリストと同様です.
配列で成分の参照を行う場合, aref 函数を用います. (aref ⟨ 配列 ⟩ n) で与えられた ⟨ 配列 ⟩ の n 番目の成分を参照します. 配列の添字は C と同様に 0 から開始します:

```
[6]> (setq a1 (make-array 5 : initial -element 1))
#(1 1 1 1 1)
[7]> ( aref a1 0)
1
[8]> ( setf ( aref a1 2) 2)
2
[9]> a1
#(1 1 2 1 1)
```

setq 函数は原子に値を割当てる函数です. これに対して, setf 函数は setq 函数に似ていますが, setq 函数と違って函数の返却値を割当てることができます. この例では texttt(setf (aref a1 2) 2) で配列 a1 の第 3 成分に 2 を割当てていますが, これは配列 a1 の第 3 成分を参照すると 2 が返るといった方式で割当を行っています.

3.6　ハッシュ表

LISP はリストと配列の他にハッシュ表が扱えます. ハッシュ表の生成は make-hash-table 函数を用います. `(make-hash-tabel a)` でハッシュ表 a が生成されます. このハッシュ表は配列とは少し違った使い方ができます. 配列であれは値と整数列で指示される配列内部の位置が対応しますが, ハッシュ表であれば「キー」と呼ばれる記号と表の値が対応します. ハッシュ表の値の参照では gethash 函数を用います:

```
[3]> (setq a (make-hash-table))
#s(hash-table eql)
[4]> (setf (gethash :one a) 1)
1
[5]> (setf (gethash :two a) '(1 2))
(1 2)
[6]> (setf (gethash :three a) "1 2 3")
"1 2 3"
[7]> a
#s(hash-table eql (:three . "1 2 3") (:two 1 2) (:one . 1))
[8]> (gethash :three a)
"1 2 3" ;
t
```

この例では変数 a にハッシュ表を割当てています. ハッシュ表に値を割当てるために setf 函数を用いますが, その方法は gethash 函数でキーが指定されたときの値として行います. ここでキーは ":one", ":two", ":three" といった札を用いています.

3.7　割当と評価

setf 函数と setq 函数は変数に値 (原子, 或いはリスト) を割当てる函数です. setf 函数と setq 函数は通常の変数に対しては同じ作用を行いますが, setf 函数は変数以外の対象への割当も行えます. この性質を上手く使うことで setf 函数だけで済ませられます.
eval 函数は変数に割当てられた対象を評価します. let 函数は指定した対象で変数を一時的に束縛する函数です:

```
[26]> (setf a 1)
1
[27]> (* 2 a (+ a a ))
4
[28]> (setq a '1)
1
[29]> (setq b '128)
```

```
128
[30]> (setq c '(/ a b))
(/ a b)
[31]> (eval c)
1/128
[32]> (let ((x '2)) (* x 2))
4
[33]> x

*** - eval: variable x has no value
```

eval は先頭に評価をさせないための前置詞 “’”(逆単引用符) が付いた S 式を評価する際に使います. この方式は Maxima,Maple や *Mathematica* でも `'diff(a,x)` といった名詞型の表記を評価する際に同名の eval 函数を用います.
let は局所変数を作り出す函数です. 上の例で示すように一時的に記号 x に 2 を束縛させ, let 文が終了すると記号 x は最初の値が割当てられていない状態に戻ります. 値が設定されていれば let 函数を実行する前の値に戻されます. 次に値が設定されていた場合を観察してみましょう:

```
[35]> (setf y '3)
3
[36]> (let ((x '2)(y '10)) (* x y))
20
[37]> y
3
```

let 函数によって一時的に変数 x に 10 が束縛されていますが, let 函数を終えると変数 x に束縛された値は元の 3 に戻っていますね.
ここで let 函数では局所変数に与える初期値として別の局所変数の値が使えません. 別の局所変数の値を利用したければ let*函数を用います. この let*函数の使い方は let 函数とほぼ同じです:

```
[12]> (let ((x 2)(y (sin x)))(+ x y))

*** - eval: variable x has no value
the following restarts are available:
store-value :r1 you may input a new value for x.
use-value :r2 you may input a value to be used instead of x.

break 1 [13]> :q

[14]> (let* ((x 2)(y (sin x)))(+ x y))
2.9092975
```

3.8 構造体

LISP でも C のような構造体を持っています. C と違う点は構造体を定義すると, その構造体を扱うための函数が自動的に生成されることです. ここで構造体の定義は defstruct 函数を用います. ここでは force という構造体を定義してみましょう. 名前の通り力を表現しようと意図したもので, こ

こでは 3 次元の Decarte 座標系で力を X 軸方向の fx, Y 軸方向の fy, Z 軸方向の fz で表現してみましょう:

```
[1]> ( defstruct  force  fx  fy  fz )
FORCE
[2]> ( setf  F_1  (make-force : fx  0  : fy  0  : fz  -9.80665))
#S(FORCE :FX 0 :FY 0  :FZ  -9.80665)
[3]> ( force -fz  F_1 )
-9.80665
[4]> ( force -p F_1 )
T
```

構造体 force を定義すると, この構造 force を生成するための函数 make-force と各成分を取出す函数 force-fx, force-fy, force-fz, そして定義した構造体であるかどうかを判定する述語函数 force-p が自動的に生成されり, 実際に構造体の定義と値の取り出しと検証が行えていますね.

3.9　写像函数

LISP には写像函数と呼ばれ, 函数をリストの各成分に作用させる函数があります. このような函数はリストが扱えるように設計された言語にも実装されていることが多いので覚えていても損はありません (たとえば Maple の map 函数).

mapcar

mapcar 函数は函数をリストの各成分に作用させる際に用いる函数です:

```
[1]> (mapcar ' sin  '(1  2  3  4  5))
(0.84147096  0.9092974  0.14112  -0.7568025  -0.9589243)
```

この例では sin 函数をリスト '(1 2 3 4 5)' の各成分に作用させたリストを返しています.

apply

apply 函数も函数をリストに作用させる函数です. 簡単に言えばリストの先頭に函数を挿入した S 式を評価するものと言えるでしょう:

```
[154]> (defun pab( x y ) (+ x y))
pab
[155]> (apply 'pab '(1  2))
3
```

3.10　lambda 式

lambda 式は Church の λ – 計算 を LISP に実装したものです. LISP の大きな特徴の一つが lambda 式の存在です. λ 式の考え方は古くは Frege にも見られますが, $f(x)$ という表記が x という変数を持った函数 f なのか, 函数 f の x における値を指しているのかが不明瞭であるために, 函数を明記

することを目的に Church が始めた表記法です. たとえば $f(x) = x^2$ が与えられたときに函数 f を $\lambda\, x\, x^2$ と表記することで, x における値 $f(x)$ と函数 f を明瞭に区別できるわけです.
これを LISP の lambda 式に書き写すと '(lambda (x) (* x x))' となります. この lambda 式だけでは計算手順を記しただけのものですが, LISP では, この lambda 式に apply 函数, mapcar 函数等の函数を用いて柔軟な処理が可能になります.
この lambda 函数は Maxima のソースファイルを眺めるとそこらじゅうに出てきますし, 実際に使えると便利なので使い方を覚えておくと良いでしょう.

3.11 函数の定義

函数の定義は defun 函数を用います. たとえば与えられた数値を 2 倍にする函数は `(defun x2 (x) (* 2 x))` と定義できます.
もし局所変数を利用したければ let 函数や let 函数が使えます. let 函数は配下の局所変数に初期値として対象を割当てることに使えますが, 他の局所変数の初期値を用いて別の局所変数への割当は行えません. そのようなことを行いたければ let*函数を用います.

3.12 制御文

条件分岐には if 式や cond 式が使えます.

if 式: 1 つの述語に対する分岐処理が表現できます:

if 式の構文
(if ⟨ 述語 ⟩)(⟨S 式$_1$⟩) (⟨S 式$_2$⟩)

if 式は ⟨ 述語 ⟩ が 't' であれば ⟨S 式$_1$⟩ を実行し, 述語が 'nil' であれば ⟨S 式$_2$⟩ を実行します.
なお ⟨S 式$_1$⟩ には論理積 "and" や論理和 "or" で結合した S 式が使えます. さらに ⟨S 式$_2$⟩ も論理積 "and" で結合した S 式が使えます:

```
1 (defun my_matrixp (x)
2         ( if (and(typep x 'hash-table)
3                  (numberp (gethash :row x))
4                  (numberp (gethash :col x)))t
5             nil ))
```

cond 式: 複数の述語に応じて処理を実行することができます:

cond 式の構文

(cond	((⟨ 述語$_1$⟩)	(⟨S 式$_1$⟩))
	...	
	((⟨ 述語$_n$⟩)	(⟨S 式$_n$⟩))
	(*t*	(⟨S 式$_{n+1}$⟩))))

cond 式では *i* 版目の ⟨ 述語$_i$⟩ で始めて 't' になると ⟨S 式$_i$⟩ を実行して cond 式を抜けます. もし全ての述語が 'nil' であれば, 最後の ⟨S 式$_{n+1}$⟩ を実行して終了します:

```
1 (defun set−matrix−element (a x e)
2       ( if (my_matrixp x)
3          (cond((> (car  a)(gethash :row x))nil)
4                ((>  (cadr a)(gethash :col  x)) nil)
5                (t(setf  (gethash a x)  e )))))
```

3.13　属性

LISP の任意の記号に対して属性を付与することができます. この属性は属性 (キー) と属性値で構成された属性リスト (P リスト) で表現されます:

属性リストの構造

(⟨ 属性$_1$⟩⟨ 属性値$_1$⟩ ··· ⟨ 属性$_n$⟩⟨ 属性値$_n$⟩)

属性の設定と取出に関連する函数を次に纏めておきます:

属性の設定と取出を行う函数の構文

函数名	構文
setf	(setf (get ⟨ 記号 ⟩⟨ 属性 ⟩)⟨ 属性値 ⟩)
push	(push ⟨ 属性値 ⟩ (get ⟨ 記号 ⟩⟨ 属性 ⟩))
get	(get ⟨ 記号 ⟩⟨ 属性 ⟩⟨ 属性値 ⟩)
symbol-plist	(symbol-plist ⟨ 記号 ⟩)

最初の setf と push 函数 で記号に対する属性リストの設定が行えます. ここで pop 函数, inof 函数や deof 函数も属性リストの設定で利用できます. また属性値の取出は get 函数, 記号に設定した属性リストの表示は symbol-plist 函数を用います:

```
[8]> (push 'みけ (get 'my_cat '名前))
(みけ)
[9]> (push 'たま (get 'my_cat '名前))
(たま みけ)
[10]> (push 'とら (get 'my_cat '名前))
(とら たま みけ)
[11]> (push 'カトリーヌ (get 'my_cat '名前))
(カトリーヌ とら たま みけ)
```

```
[12]> (symbol−plist 'my_cat)
(名前 (カトリーヌ とら たま みけ))
[13]> (get 'my_cat '名前)
(カトリーヌ とら たま みけ)
```

属性リストの基本的な考え方は Russell の「還元可能性公理」や集合論の「内包の公理」を LISP に適合させたものと言えるでしょう. ここで「還元可能性公理」は,「任意の階の述語に対して同値な 1 階の可述的函数が存在する」というものです. この還元可能性公理や内包公理の解説は §4.20 を参照して下さい.
属性を用いると該当する属性に対応する集合 (クラス, あるいは外延) が一つ定まります. 属性を用いる利点は集合が外延的定義ではなく内包的定義で行えること, つまり, ある対象が何等かの集合の成分であることを述べる際に集合を構成する成分を列記したリストを必要しないことです. たとえば「××高校 3 年 A 組のクラス」とすれば人名を列記したリストは必要ありませんね. そして「実数の集合」のような抽象的な概念に対し, 内包的定義は威力を発揮します.

3.14 入出力

この節では簡単に LISP の入出力について述べます.

load 函数: LISP のプログラムファイルの読込は load 函数を用います:

load 函数の構文

構文	**概要**
(load ⟨ ファイル名 ⟩)	指定したファイルの読込を行う

ここで ⟨ ファイル名 ⟩ は LISP の文字列で, たとえば “tama.lisp” という名前のファイルに記述した函数等の読込を行う場合, `(load "tama.lisp")` で読込みを行います.
Common Lisp はストリームを用いてファイルへの入出力を行います.

open 函数： ストリームを開くときに用いる函数です:

open 函数

構文	概要
(open ファイル名 :direction :input)	指定したファイルを読込用に開く.
(open ファイル名 :direction :output)	指定したファイルを書込用に開く.

close 函数: 開いたストリームを閉じるために使われます:

close 函数

構文	概要
(close ファイル名)	指定したファイルを閉じる

では実際に簡単なファイル操作の実例を示しましょう. 予め準備した “test1” ファイルを LISP 側から内容を読取って “test2” ファイルに書出してみましょう. まず “test1” の内容を次に示しておきます:

ファイル test1 の内容

```
1 2 3 4
"TEST"
```

以下に LISP での処理の様子を示します:

```
[1]> (setf test1 (open "test1" :direction :input))
#<INPUT BUFFERED FILE-STREAM CHARACTER #P"test1" @1>
[2]> (setf x1 (read test1) x2 (read test1) x3 (read test1) x4 (read test1))
4
[3]> (setf str1 (read test1))
"TEST"
[4]> (close test1)
T
[5]> (setf test2 (open "test2" :direction :output))
#<OUTPUT BUFFERED FILE-STREAM CHARACTER #P"test2">
[6]> (print str1 test2)
"TEST"
[7]> (print x1 test2)
1
[8]> (print x2 test2)
2
[9]> (print x3 test2)
3
[10]> (print x4 test2)
4
[11]> (close test2)
T
```

この例では最初に open 函数で読込用にファイル “test1” を開きます. ここで “test1” をストリーム名にして read 函数でファイルの読込を行います. それから close 函数でストリーム test1 を閉じることでファイル “test1” を閉じます. そして今度はファイル “test2” を書込用に開いて, 今度は print 函数で各原子に割当てられた値をファイル “test2” に書込み, close 函数でストリーム test2 を閉じます. ここで print 函数では改行を行うためにファイル “test2” の内容は次のようになります:

ファイル “test2” の内容

```
"TEST"
1
2
3
4
```

このファイルでの入出力で出て来た read 函数は, ストリームが指定してあれば指定したストリームからの読込みを行い, ストリームが指定されていなければキーボード (正確には*standard-input* ストリーム) からの読込を行う函数です.

原子を評価して書式なしで出力する函数に prin1 函数, princ 函数と print 函数があります. これらの函数は引数は一つですが, オプションとして出力ストリームが指定できます.
書式付きの出力函数には format 函数があります. format 函数は C の書式付きの fprint 函数と似た機能を持ってるものの書式は fprint 函数とやや異なっています:

format 函数

(format ⟨ 出力ストリーム ⟩ ⟨ 書式文字列 ⟩ ⟨S 式 ⟩)
(format t ⟨ 書式文字列 ⟩ ⟨S 式 ⟩)

ここで書式文字列に書式指示子を幾つ書いても構いませんが, 書式指示子の総数と S 式の総数は合せておく必要があります. また出力ストリームの代りに “t” を置くと通常の表示となります. ここで指定可能な書式指示子の主なものを列記しておきます:

主な書式指示子

概要	対応する指示子
エスケープ無し出力	a
エスケープ付き出力	s
改行	%
固定少数点表示	w, dF
固定少数点表示	w, dE
一般小数点表示	w, dG

```
[75]> (setq a (sin (/ pi 4)))
0.70710678118654752444L0
[76]> (format t "a= a" a)
a=0.70710678118654752444L0
NIL
[77]> (format t "a= s" a)
a=0.70710678118654752444L0
NIL
[78]> (format t "a= 5,3f" a)
a=0.707
NIL
[79]> (format t "a= 5,3e" a)
a=7.071L-1
NIL
[80]> (format t "a= 5,3g" a)
a=.707
NIL
```

大雑把ですが LISP の概要を終えます. なお §12 や §13 に Maxima の動作や Maxima の改造に関連した話を載せているので参考にされると良いでしょう.

コラム:Common Lisp 色々

Maxima のコンパイルで利用可能な Common Lisp としては, 無課金で利用可能な GCL, CLISP, CMUCL, SBCL ,OpenMCL , 商用の Allegro Common Lisp(ACL) があります. sourceforge から入手できる Maxima のバイナリ版は GCL を用いていますが, LISP の選択自体は個々人の環境や目的に大きく依存します. 現時点で様々な環境で使いたければ CLISP, 速度重視であれば CMUCL や SBCL, 速度重視で 64bit 環境への対応なら SBCL, 過去の遺産を使うのであれば GCL が良いでしょう.

なお, Maxima でどの Common Lisp が正式に対応しているかどうかは, sourceforge の Maxima の頁を参照するか, ソースファイルに附属の configure で, `./configure --help` を実行させ, Optional Features に現われる LISP の解説を読めば判ります. たとえば Maxima-5.15.0 のソースファイルを展開して, そのディレクトリ上で configure –help を実行すると Optional Features の欄は以下のものとなります:

−−enable−clisp	Use clisp
−−enable−cmucl	Use CMUCL
−−enable−scl	Use SCL
−−enable−sbcl	Use SBCL
−−enable−acl	Use ACL
−−enable−gcl	Use GCL
−−enable−openmcl	Use OpenMCL

このことから, Maxima-5.15.0 では CLISP, CMUCL, SCL, SBCL, ACL, GCL と OpenMCL が正式に対応している事が判ります. とはいえ, 簡単にコンパイルができるかどうかは実際にやってみないと判りません. 私の経験では CLISP が最も問題がないようです.

第4章　数学のいろいろなこと

原初に言葉 (*λογος*) があった.
言葉は神と共にあった.
言葉は神であった.
これは原初に神と共にあった.
万物はこれによって成り, これに因らぬ物は何一つとして無かった.

ヨハネの福音書, ロゴス讃歌より

この章では Maxima を使う上で知っておくと便利な数学の言葉と考え方について簡単に解説します. Maxima には代数学の考え方が色々と取り入れられています. この考え方は今では計算機代数と呼ばれる分野に相当しますが, ともすれば人工知能の萌芽も含まれており, 歴史的にも興味深いものとなっています.
そのために, この本では代数的な側面だけではなく, 哲学的な側面も解説することになるでしょう. ここで代数的な側面を理解するために**群**, **環**, **イデアル**, そして, **多項式環**や**順序**といった考え方を紹介します. そして, 哲学的側面については数理論理学に関連した話を紹介します. しかし, これらは簡単な紹介の域を出るものではないので, 興味を持たれた場合は参考文献を参照されると良いでしょう.
なお, この本は教科書のように真面目な本ではなく, 寧ろ, 鬼火のようにふらふらと飛び回っては面白そうな所を摘み食いするとても不健全, かつ不真面目な本です. それ故にあちらこちらへと脱線するかと思いますが, その際は御容赦を.

4.1　集合について

概念

さて, ここでは数学の色々なことを解説する前に「**概念**」について簡単に述べておきましょう. そもそも, 考察や議論を行うにあたって, その対象が何であるか不明なままでは考察はできません. では, 考察すべき対象はどのようにして定めればよいのでしょうか？ 自分の持物を他人に説明するときを考えると, その物の特徴を列記したり, 相手が知っている類似物を挙げて説明するでしょう. つまり, 対象の特徴を列記することで他と区別を行っているのです.
この対象を特徴付けて他と区別することとなる事柄, すなわち, **対象の特徴や性質**のことを「**徴表**」, あるいは「**属性**」と呼びます. そうして, 対象に共通する徴表で対象をひとまとめにすることができます. このように対象の徴表の共通性を取出すこと, すなわち抽出によって得られるものが「**概念**」なのです.

この概念の表現に関しては,「**内包**」と「**外延**」の 2 つの言葉があり,「**内包**」は概念が持つ徴表で構成され,「**外延**」は概念が適用される対象の範囲を示します. たとえば,「**人間**」という概念の場合, その内包は「**動物である**」,「**理性的である**」,「**二本足で歩く**」等の特徴, すなわち, 徴表から構成されるでしょう[1]. 一方で「**人間**」という概念の外延として,「**人種**」で列記すれば,「**モンゴロイド**」,「**ネグロイド**」,「**コーカソイド**」,「**オーストライド**」,... となるでしょう.

外延を用いて表現された概念は内包で表現することができますが, その逆の, 内包で表現された概念は外延で表現できるとは限りません. たとえば, “ミケ, タマ” は “我が家のペット” と言い換えられますが, “明日の新人歓迎会の欠席者” を正確に列記することは流石に無理ですね.

そして, 内包と外延には「**内包外延反比例増減の法則**」と呼ばれる関係があります. この関係は, 内包が増大するに従って外延が減少し, 外延が増加すれば内包が減少するという関係です. たとえば,「昆虫」という概念に対し,「羽がない」,「集団で生活する」といった内包を追加すると,「カブトムシ」,「セミ」といった外延が消えて行きますね. では,「アリ」を考えてみましょう.「アリ」だけなら「羽がない」,「集団で生活する」,「足が六本」といったアリの性質, つまり, 内包を列記できます. ここで「セミ」を外延として追加すると,「羽がない」や「集団で生活する」といった内包が消えます.「**内包外延反比例増減の法則**」とはこの事実を示しています.

それから「**明瞭な概念**」とは概念の内包を構成する徴表が明確な場合,「**明晰な概念**」とは概念の外延の範囲が明確な場合を言い,「**明確な概念**」とは概念が明瞭で明晰な場合です.
二つの概念の外延を比較したときに, より大きな外延を持つ概念を「**上位概念**」, あるいは「**類概念**」, 外延がより小さな概念を「**下位概念**」, あるいは「**種概念**」と呼びます. 先程の「人類」で解説するならば,「モンゴロイドの類概念」が「人類」,「モンゴロイド」が「人類の種概念」となります. そして,「**種**」の違いを示す徴表を「**種差**」と呼びます. それから, 最上位の上位概念を「**範疇 (Cathegory)**」, 最下位の下位概念を「**単独概念**」, あるいは「**個体概念**」と呼びます. この個体概念は「**個体 (individual)**」を指定する概念となります.

[1] 人間が「**二本足で歩く, 羽のない動物**」だとすれば,「**羽を毟られた鶏**」という Diogenes の有名な反例があります.

集合とは

概念について解説したので, 今度は「**集合**」の話を始めましょう. ここで集合という代物が出てくる理由ですが, 先ず, 考察や議論を行う上で対象を明瞭にしておくことは非常に重要です. そして, 現代の数学はその対象を明確にするために集合を用いており, その集合に関連する幾つかの用語や記号を知っておくと何かと便利だからです.

さて, **集合**とは具体的なもの (対象) のあつまりのことです. ここでの「具体的なもの」とは「明確に言及可能な性質を持つもの」です. たとえば,「X 家の家族」,「偶数の集合」,「1 よりも大きな実数の集合」のように, その対象の持つ性質, 属性によって明確に構成要素が指定できるもの, あるいは,「太郎, 花子, みけ, ポチで構成された集合」のように集合の構成要素が明示的に列記できることです. ここで前者の言い分は「明瞭な概念であること」, つまり,「明確な内包を持つこと」, 後者が「明晰な概念であること」, つまり,「明確な外延を持つこと」です. したがって, 集合は1明瞭な概念, あるいは 明晰な概念 でなければなりません, ここでの集合の説明には大きな問題があるのですが暫く頬被りしたままにしておきます[2].

なお, ここでの集合の説明は論理学の「**クラス/類**」と呼ばれるものです.

集合の表記

集合は通常, 記号 “{ }” を用いて, その元や対象が充すべき述語を表記します. ここで集合の表記には二種類の方法があります. まず,「偶数の集合」は ‘{ 偶数の集合 }’ とそのままの表記の他に, ‘{x|x は偶数 }’ のように記号 “{ }” の中を記号 “|” で二つに分けて記号 “|” の左側に「**変項**」と呼ばれる表象を配置し, 記号 “|” の右側に変項が充すべき属性を指示する「**述語**」を記述する方法です. なお, この表記を簡単に ‘{x は偶数 }’ と記述することもあります. もう 1 つは,「X 家の家族」の構成員が, 太郎, 花子, みけ, ポチのときに, ‘{ 太郎, 花子, みけ, ポチ }’ と, 集合の成員, すなわち, 元を列記する方法です. 最初の集合を構成する成分の属性 (ここでの例では ‘x は偶数’ という命題), すなわち, 徴表から構成する定義方法を「**内包的定義**」と呼び, 後者の集合を構成する要素を列記することで集合の範囲を明確に定める定義を「**外延的定義**」と呼びます. そして, 集合を構成する対象を「**元**」, あるいは「**成員**」と呼び, 対象 a_i が集合 A の元を ‘$a_i \in A$’ と表記します. ここで集合の元を列記する場合に記号 “{ }” の内部での順序が異なっていても集合としての違いはありません. したがって, 集合 ‘{1, 2}’ と集合 ‘{2, 1}’ は同じ集合になります.

集合 A と集合 B の全ての元が一致するときに「**集合 A と B は等しい**」と呼び, ‘$A = B$’ と表記します. また, 集合の元の個数を「**基数 (cardinal number)**」, あるいは「**濃度**」と呼び, 集合 M の基数を $\mathrm{card}(M)$ と表記します.

[2] この素朴とも言える集合の定義から, 20 世紀初頭に §4.17 の「数学の危機」と呼ばれる状況を招いています. 結論としては「**概念は必ずしもその集合を持つとは限らない**」ということです. この詳細は §4.22 を参照.

集合の関係

集合 A と B について, 集合 A の元が全て集合 B の元である場合, 集合 A を「**部分集合**」と呼び, 両者の関係を '$A \subseteq B$' と表記します. さらに集合 A が集合 B の部分集合であり, 集合 A に含まれない集合 B の元 b が存在するときに集合 A のことを集合 B の「**真部分集合**」と呼び, 両者の関係を '$A \subset B$' と表記します. また, '$A \subseteq B$' かつ '$A \supseteq B$' のときに集合 A と集合 B は等しいと呼び, 両者の関係を '$A = B$' と表記します. そして, これらの記号 "$\subseteq$", "$\subset$", "$=$" で表現される二つの集合の間で成立する関係を「**包含関係**」と呼びます.

特殊な集合

特殊な集合の一つに**一切の元を持たない集合**があります. この集合を「**空集合**」と呼び, 記号 "$\emptyset$" で表記します[3]. ここで集合を表現する記号 "$\{\,\}$" を使って '$\{\,\}$' で空集合を表記することもあります. この空集合 $\emptyset$ は任意の集合の部分集合で, その基数 $\mathrm{card}(\emptyset)$ は 0 になります.

そして, 集合 S に含まれる元から構成される全ての部分集合を元とする集合を集合 S の「**冪集合**」と呼び $\mathfrak{P}(S)$, あるいは 2^S と記述します. たとえば, 集合 $S = \{1, 2, 3\}$ の冪集合は $\mathfrak{P}(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1, 3\}, \{2,3\}, \{1, 2, 3\}\}$ となり, 冪集合の元の個数, すなわち, 基数 $\mathrm{card}(\mathfrak{P})$ は $8(= 2^3)$ です. そして, 濃度が自然数で与えられる集合のことを「**有限集合**」と呼びます. なお, 集合 S とその冪集合 $\mathfrak{P}(S)$ に対して '$\mathrm{card}(\mathfrak{P}(S)) = 2^{\mathrm{card}(S)}$' が成立します. このことが冪集合を 2^S とも表記する理由になっています.

集合の演算

集合 S の冪集合 $\mathfrak{P}(S)$ に対しては "$\cup$", "$\cap$", "^{c}" や "$-$" といった演算があります. ちなみに, ここでの「**演算**」とは二つの集合を使って新しい集合を生成する操作と解釈しておいて下さい.

まず, '$A, B \in \mathfrak{P}(S)$' に対し, 集合 A と集合 B の全ての元で構成された集合を集合 A と集合 B の「**和集合**」と呼び '$A \cup B$' と記述します. 同様に集合 A と集合 B の共通成分で構成された集合を「**共通集合**」と呼び $A \cap B$ と記述します. そして, 集合 B から集合 A の元を取り除いた集合を $A - B$ と記述し, 「**差集合**」と呼びます. 特に, $S - A$ を A の「**補集合**」と呼び A^c と記述します. そして, 演算子 "$-$" と演算子 "^{c}" については定義から '$A - B = A \cap B^c$' が成立します.

[3]空集合の表記は André Weil によるものです. 「ヴェイユ自伝」([7]) を参照. なお, 空集合 $\emptyset$ はギリシャ文字の ϕ とは違います.

4.2 同値関係について

4.2.1 数式処理システムとオブジェクト指向

数式処理で扱う対象は変数や函数を含んだ数式です. この数式処理は式を人間が扱うように処理することから人工知能の一分野のように見えますが, 実際は式の代入, 置換等の代数的処理を中心に行うものです. すなわち, 有限個の「**語 (word)**」と呼ばれる文字列に対し, 数学上の性質やさまざまな規則に対応する有限個の操作を行って, 与えられた語を変換するシステムです.
この数式処理システムには, Singular や Macaulay2 のように最初に環と呼ばれる数学的対象が存在する考える世界を指定し, その中で数式や写像を処理してゆくものや, *Mathematica*, Maple や Maxima のように, 処理する数式が何処に存在しているかを予め指定する必要なしに, 対象の処理が行えるものがあります.
前者の予め世界を指定するシステムは Java のようなオブジェクト指向の言語の影響を強く受けた数式処理システムで, どちらかと言えば専門用途向けのものが多くあります. このオブジェクト指向の数式処理システムの利点は, たとえば環を定義することで, 環に予め用意されたメソッドを用いてさまざまな操作が可能であり, その環を使って新たな数学的対象を定義すると, 環のメソッドが継承されるといった非常に大きな利点があります. しかし, 環を定義しなければ整数の四則演算といった基本的な処理しかできないという弱点もあります.
後者は, 数式処理でも適用範囲が広い汎用の数式処理に多く, 数式の処理を行うための環を定義するといった準備が不要なので手軽に使えますが, 逆に, 群や環を処理したければ, そのような数学的な構造やメソッドを自分で定義する必要があります. 大規模なプログラムの作成や過去の計算の成果を継承する場合, オブジェクト指向的なシステムは非常に効率が良く, そのために Python で既存の OSS のアプリケーションを繋ぎ合せることで構築された汎用の数式処理 SAGE はオブジェクト指向の数式処理システム (というよりも, 数式処理も含む数学環境) でありながら, Singular や Macaulay2 のように多項式の計算で環を定義する必要もなく, *Mathematica* や Maxima のような操作性を実現させています.

4.2.2 等しいということの考察

ところで計算機にとって, システムに入力された式は, 人間が何らかの指定をしない限りは単なる文字の羅列に過ぎません. では, 与えられた二つの数式が等しいかそうでないかを計算機に判断させるためには何が必要でしょうか？ より一般的に二つの与えられた式が等しいとはどのようなことでしょうか？ この問に対し, 初期の数式処理の開発で人工知能が深く関係しました. しかし, 人工知能の話まで行かなくても, より機械的な操作にまでもって行くことが可能です.
その機械的な操作の一つに, 式の変数に数値を入れて同じ値となるかを確認する方法があるでしょう. この方法を視覚的に行えばグラフの比較となりますが, これらの方法は数値誤差の問題に加えて, 実際の計算機で扱える数値の範囲の問題, 計算の手間といった問題もあって, 領域を限定して上で, 近似的に正しいとは言えても, 本当に等しいとは結論付けることはできません. では, 等しいと結論付けるにはどうすれば良いのでしょうか.

そこで, ここでは古来から厳密で完成していると考えられていた Euclid の幾何学を参考に考えてみましょう. まず, 二つの図形が与えられたとき, Euclid の幾何学でこれらが等しい図形 (合同) であるとは, 二つの図形を重ね合せられるかどうかに帰着されます. この重ね合せる操作は「平行移動と回転の有限回の組み合わせによる操作」, つまり, 「**一次変換**」と呼ばれる操作によって一方の図形を片方の図形に変換できることです[4].
さて, 幾何学の場合は一次変換で重ね合せられる図形を同じ図形としましたが, これと同様に「**与えられた式が等しい**」ということを「双方に有限回のある操作を加えて互いに移り合える場合」と定義すればどうでしょうか? このときに「ある操作」が何であるかを明確にしておく必要がありますが, たとえば, 二つの数式 $x+y+1$ と $y+x+1$ が与えられたとき, 両者を繋ぐ演算が和 "+" なので変数 x と変数 y の順序の交換が可能で, 一度だけ交換操作を行えば互いに移り合え, 変数 x と変数 y の取り得る値の領域とは無関係に同値性が確認できていますね. つまり, 和 "+" や積 "×" といった演算に対しては左右の項 (被演算子と呼びます) の交換という操作が一つ挙げられます. それから式の展開, たとえば, $(x+1)^2$ を x^2+2x+1 で置き換える操作と, その逆の $xy+x$ を $x(y+1)$ のように式を纏める操作が挙げられますね. それから式の変数や項を公式等を使って別の式で置き換える操作もあります. たとえば, 三角函数だけでも '$\cos^2(x)+\sin^2(x)=1$', '$\cos(2\theta)=2\cos^2(\theta)-1$', '$\sin(2\theta)=2\sin(\theta)\cos(\theta)$' や'$e^{a+i\theta}=e^a(\cos(\theta)+i\cdot\sin(\theta))$' といった公式が存在し, これらの公式を適用することで式の変形が可能なのです. このように式の変形で互いに移りあえることで合同性を判断する方法が変数に具体的な数値を当て嵌めて判断するよりもより適切な方法であると言えます.

では, この置き換えという操作と, この置き換えで同じ式とみなすという同値性とは具体的にどのようなものでしょうか. また, 対象を扱う上で「同値」であるからといって, 計算で全く同じ扱いをしても構わないものなのでしょうか. これらの点について身近な数である分数を例に考え直してみましょう.

4.2.3　分数と同値関係

分数は小学生にとって鬼門の一つですが考え直して見ると面白い数です. 分数は分母と分子と呼ばれる二つの数から構成されています. より正確には分母が零と異なる自然数 $\mathbb{N}-\{0\}$, 分子が整数 $\mathbb{Z}$ の二つの数で構成された数です.
では, 有理数 1/2 と有理数 2/4 は同じものでしょうか？ これは有理数 2/4 を「約分」すれば直ちに有理数 1/2 が得られるので, これらの有理数は同じ数だと結論付けることができます. ところが, これらの有理数を単なる文字列 "1/2" と "2/4", あるいは分子と分母を並べた自然数の対 '(1, 2)' と '(2, 4)' で考えると全くの別物です. しかし, これらの対を有理数に対応させるときには, 自然数の対 '(1, 2)' と '(2, 4)' は等しいものであるべきです.
このことは, これらの対が等しいか等しくないかを判断する明確な手続や規則が要求されていることを意味します. この点を曖昧にしていると, 有理数を計算機のプログラムで上記のように分子と分母の二成分の配列で表現し, 「分数 a と分数 b が等しい」ことの判定が要求されたときに, 安易に「a[1] = b[1] かつ a[2] = b[2]」という「**規則**」だけで処理する羽目になって間違った判定を行うこ

[4] この変換を「**Euclid 変換**」とも呼びます. Euclid 変換には「**平行移動**」と「**回転**」に加えて「**鏡像**」も含まれますが, ここでは図形の裏と表を分けて, つまり, 図形の向きも考えたいので「**鏡像**」を除外しています.

とになります. 具体的には, 分母と分子から共通の約数を削除する「**約分**」に対応する配列操作の「**手続**」を明確に定め, 対として等しいことにだけではなく, 約分に対応する手続によって対が等しくなるものを同じものと見なす「**規則**」を計算機に教えておく必要があります.

この約分という手続で同じ有理数が得られたときに同じ有理数とみなすという規則についてもう一歩踏み込んで考えてみましょう. まず, 分数は有理数 $\mathbb{Q}$ と呼ばれる数で, 分母と分子の二つの整数で構成されています. そこで, 有理数 a/b を対 (a,b) のように整数と零と異なる自然数の対で表現します. このとき二つの整数と自然数の対 (a,b) と (c,d) が等しくなるのは約分でどちらかに移れることですが, 約分という操作で分母と分子の共通の因子を削除することなので, 零と異なる整数 q が存在して '$a = cq$ かつ $b = dq$', あるいは '$c = aq$ かつ $d = bq$' となるときです. これらの条件は「'$ad - bc = 0$' を充すとき」という条件に集約することが可能です. そこで二つの整数の対 $A = (a,b)$ と $B = (c,d)$ が '$ad - bc = 0$' を充すときに '$A \sim B$' と表記しましょう. この例のように二つの対象 A, B がある条件を充すことを「**関係**」と呼び, A と B が充す関係を $A \sim B$ で表記するときに記号 "~" を簡単に関係 "~" と呼びます. そして, 関係 "~" が次に示す性質を充すときに「**同値関係**」と呼びます:

同値関係

1. 反射律:$A \sim A$
2. 対称律:$A \sim B$ ならば, $B \sim A$
3. 推移律:$A \sim B$, かつ, $B \sim C$ ならば, $A \sim C$

ここで同値関係 "~" に関して同値になるものから構成される対象から構成される集合のことを「**同値類**」と呼びます. そして同値類から取出した一つの元, ここでの有理数の例で自然数対 $(1,2)$ のことを「**同値類の代表**」と呼びます. ここでの有理数では互いに素となる整数 a, b に対して構成される集合 $\{(aq)/(bq) \mid q$ は 0 と異なる自然数 $\}$ が有理数 a/b の同値類であり, 有理数 a/b はこの同値類の代表の一つとなります. そして, 全体集合を X とするときに関係 "~" による同値類の集合を $X/\sim$ と表記します. この対集合の例では '有理数 ≡ 整数 × 自然数 − $\{0\}/\sim$' が成立することになります.

ここで「**同値類**」と呼ばれる何やら凄いものが出てきましたが, お馴染の処理方法や考え方をより一般的なものとして言い直したものに過ぎません. この分数の話では「約分」という「**手続**」と, 約分によって一致する有理数を同一視するという「**規則**」によって整数と自然数の対に「**同値関係**」を導入し, その「**同値類**」という「**概念**」を抽出していることに注目してください. このように「**抽象化**」によって「**事象**」は「**概念**」へと辿り着くことで最終的に「**普遍性**」を得るのです.
小学校で分数の計算で苦労された記憶を持つ方も多いかもしれませんが, ここで言い直したような高度な概念を含んでいるので苦労しても仕方がないと言えるでしょう.

4.2.4 きちんと定義できていることの検証

分数 a/b は整数と自然数の対 (a,b) の同値類として表現できると述べました. ここでの自然数の対で有理数を表現するときに, 整数 n は有理数の形では $n/1$ として表現できるので整数 n を対集合の

元 $(n,1)$ で代表させることが判ります.
ここで整数には四則演算や大小関係といったものがあります. これらの演算や関係を自然数の対に自然な形で拡大することができるのでしょうか？ もしも, これらの演算や関係が自然な形で拡大することができないのであれば, このような対を考える操作は「面倒な手間を増やす処理」でしかないかもしれません.
そこで, ここでは整数に与えられた四則演算を使って有理数の四則演算を定義し, それがきちんとしたものであるかを確認してみましょう. なお, 有理数を自然数の対として表現しているので, 有理数で普通に計算している「和」“+”, 「差」“−”, 「積」“×” や「商」“÷” といった「**四則演算**」が整数と自然数の対でも行えるかどうかは判らず, 判っているのは整数の四則演算は使えるということから開始します. ここで行おうとすることは, オブジェクト指向言語で, 既存のクラスから新しいクラスを構成し, その新しいクラスのメソッドを構成する手法に似ています. その際に重要なことは, 新しい対象で従来のメソッドが自然に拡張できるかどうかということがあります. そこで, 最初に分数の四則演算について思い出してみましょう

最初に「**分数の和**」 $a/b + c/d$ は $(ad+bc)/(bd)$, 「**分数の積**」 $a/b \times c/d$ は $(ac)/(bd)$, そして「**分数の商**」$a/b \div c/d$ は $(ad)/(bc)$ でそれぞれ計算します. なお, 「**分数の差**」は $a/b - c/d = a/b + (-c)/d$ とできるので和の特殊な場合と考えられるので, 「和」“+”, 「積」“∗”, 「商」“/” について考察すれば十分です. すると「和」“+”, 「積」“∗”, 「商」“/” が自然数の対に対して定義できることが判ります:

演算 “+”, “∗”, “/” の定義

$$
\begin{aligned}
\text{演算 “+” の定義：}\quad & (a,b)+(c,d) \stackrel{def}{=} (ad+bc,bd)\\
\text{演算 “∗” の定義：}\quad & (a,b)*(c,d) \stackrel{def}{=} (ac,bd)\\
\text{演算 “/” の定義：}\quad & (a,b)/(c,d) \stackrel{def}{=} (ad,bc)
\end{aligned}
$$

ここでの「演算の定義」では, 記号 “$\stackrel{def}{=}$”f の右辺の式が新しく導入する左辺の式の内容を示します. この記号はこの本で通じて使いますが, この定義式の演算子 “$\stackrel{def}{=}$” の右側の式で, 通常の整数の演算が用いられていることに注意して下さい. この手法は整数という対象とそれに付随する演算というメソッドを使って有理数という対象と演算というメソッドを定めること, すなわち有理数の演算で整数の演算が継承されることを意味します.

上記の定義で整数と自然数の対に対して演算を定義しました. では, これで十分でしょうか？ 残念ながらこれでは不十分なのです. 何故なら演算がきちんと定義できていること, つまり, 二つの整数と自然数の対で演算を行う場合, 同値関係 “∼” の代表の取り方が違っても結果が異ならないことがまだ保証されていないからです. このことは同値類で演算を行う際に非常に重要です. 実際, $2/4\times2$ か $1/2\times2$ のどちらかを使うかで計算で結果が違うと困りますね[5]. これらの演算は有理数に対しては何気に使っているので経験的に保証されていますが, このことを実際に確認してみましょう. そこで整数と自然数の対 (a,b) と (a',b') に対して ‘$(a,b) \sim (a',b')$’, すなわち, ‘$ab' - ba' = 0$’ を仮定し, (a,b) と (c,d) の演算と (a',b') と (c,d) の演算が「**きちんと定義できている**」(=矛盾が無く定義できている)

[5]出来の悪いプログラムでは良くあることです

ことを確認しましょう:

“+” がきちんと定義できていることの確認: ‘$(a,b)+(c,d) \sim (a',b')+(c,d)$’ となること, すなわち, ‘$(ad-bc)(b'd)-(bd)(a'd-b'c)=0$’ を示さなければなりません. ここで $(ad-bc)(b'd)-(bd)(a'd-b'c)$ を展開すると $(ab'-a'b)d^2$ を得ますが, 仮定 ‘$a'b=ab'$’ より与式は 0 となるので演算 “+” が同値関係を保つことが判ります.

“∗” がきちんと定義できていることの確認: この場合は ‘$(a,b)*(c,d) \sim (a',b')*(c,d)$’ となること, つまり, ‘$(ac)(b'd)-(bd)(a'c)=0$’ を示します. ここで項の変数を並び換えると $(ac)(b'd)-(bd)(a'c)$ から $ab'cd-a'bcd$, すなわち, $(ab'-a'b)cd$ を得ますが, ここで仮定 ‘$ab'-a'b=0$’ から 0 となり, 同値関係を保つことが判ります.

“/” がきちんと定義できていることの確認: 最後に ‘$(a,b)/(c,d) \sim (a',b')/(c,d)$’ となること, つまり, ‘$(ad)(b'c)-(bc)(a'd)=0$’ を確認すれば済みます. ここで $(ad)(b'c)-(bc)(a'd)$ を展開すると $ab'dc-a'bcd$, すなわち, $(ab'-a'b)(cd)$ を得るので, 仮定 ‘$ab'-a'b=0$’ より与式が 0 となって同値関係を保つことが判ります.

結論: これらの検証から言えることは, 整数の和, 積, 商といった演算を用いて, 整数と自然数の対に対して和, 積, 商といった演算が行え, これらの演算は対に対する同値類の代表のとりかたに因らず, 同値類として一意に定まること, つまり, 演算が「きちんと定義できている」ことです.
この確認作業は, 分数の演算をややこしく言い換えただけのように見えますが, 既存の対象を使って新しい対象を定義するときに, どのように演算を定義するか, そして, 定義された演算がきちんと定義されているかを確認することは非常に重要なことです. その確認ができなければ, 勝手気儘に無意味な対象を生成していると非難されても仕方がないのです.

4.2.5 $\mathbb{R}/(x^2+1)$

さて今度はもう少し変ったものを考えてみましょう. そこで, 係数が実数 $\mathbb{R}$, 変数が x の「**多項式の集合**」を考えてみましょう. この集合を $\mathbb{R}[x]$ と記述します. 後の節で詳しく説明しますが, これは数学では「**(一変数の) 多項式環**」と呼ばれます. この世界では, 多項式同士の演算として, 和, 差, 積が挙げられます.

では多項式 $f,g \in \mathbb{R}[x]$ に対し, 差 $f-g$ が多項式 x^2+1 で割切れるときに ‘$f \sim g$’ とすることで $\mathbb{R}[x]$ に関係 “~” を入れます. すると, この関係 ‘~’ は同値関係になります. 実際, 反射律と対称律は自明ですね. そして推移律は ‘$f \sim g$’, ‘$g \sim h$’ であれば ‘$f-h=(f-g)+(g-h)=q_1(x^2+1)+q_2(x^2+1)$’ となる多項式 q_1 と q_2 が存在するので多項式 $f-h$ も多項式 x^2+1 で割切れますね. このように関係 “~” は同値関係になることが判ります. では同値類 $\mathbb{R}[x]/\sim$ は何になるのでしょうか? まず 1 や x はそのままですが, x^2+1 は 0 と同値になるので 0 で置換えることができます. では $n \geq 2$ に対し, 多項式 x^n はどうなるでしょうか? ここで ‘$x^n=x^{n-2}(x^2+1)-x^{n-2} \sim -x^{n-2}$’ となるので, 与式 x^n の冪 n を $n-2$ へと 2 づつ減らせられ, その際に符号が反転します. さらに ‘$x^2=(x^2+1)-1 \sim -1$’ な

ので, 結局, 冪 x^n の項は n が奇数の場合のみ, しかも x しか残りません. そのために同値類 $\mathbb{R}[x]/\sim$ の代表は $a+bx$ の形に限定され, その上, 変数 x は方程式 '$x^2=-1$' を充す数です. さて, この方程式を充す数 x は何処かで見たことのある数ですね. すなわち 「**純虚数** i」です. つまり, 1 変数多項式環の関係 "$\sim$" による同値類 $\mathbb{R}[x]/\sim$ は「**複素数全体** $\mathbb{C}$」なのです!

ザミャーチン (Zamyatin) の SF 小説「われら」の一節で, 純虚数の存在が整数の調和を乱すものとして主人公は純虚数を嫌悪していますが, 事実, 純虚数は相当人工的な匂のする数です. この数は整数係数の方程式 '$x^2+1=0$' に縛り付けられています. この純虚数 i のように整数係数の方程式の根として現れる数の仲間のことを「**代数的整数**」と呼びます. この代数的整数は整数係数の多項式で表現されるので正体が判り易い数です. 実際, 代数的整数が変数 x のままでも多項式として処理することが可能なので自然に計算が行えるのです.

次に, この 1 変数多項式環の関係 "$\sim$" による同値類 $\mathbb{R}[x]/\sim$ の代表 $a+bx$ を実数の対として, (a,b) と表記してみましょう. すると, 有理数のときのように和, 積, 商といった演算が定義できます:

演算 "+", "*", "/" の定義

$$\begin{array}{lll} \text{和 "+" の定義} & (a,b)+(c,d) & \stackrel{def}{=} \quad (a+c,b+d) \\ \text{積 "*" の定義} & (a,b)*(c,d) & \stackrel{def}{=} \quad (ac-bd,ad+bc) \\ \text{商 "/" の定義} & (a,b)/(c,d) & \stackrel{def}{=} \quad \left(\frac{ac+bd}{c^2+d^2},\frac{bc-ad}{c^2+d^2}\right) \end{array}$$

ここで商の計算ですが, $(a+bx)/(c-dx)=((a+bx)(c-dx))/((c+dx)(c-dx))$ より $(ac-bdx^2+(-ad+bc)x)/(c^2-d^2x^2)=(ac+bd)/(c^2+d^2)+(-ad+bc)/(c^2+d^2)x$ となることから判ります.
また, この 2 対の実数は平面上の点に対応させることも可能です. このときの平面のことを Gauss 平面と呼びます. この Gauss 平面で考えた場合, 複素数同士の和はベクトルの和に対応します. 面白いのが積で, 二点 (a,b) と (c,d) が極座標で $\sqrt{a^2+b^2}(\cos\alpha,\sin\alpha)$, $\sqrt{c^2+d^2}(\cos\beta,\sin\beta)$ と表現されるとき, その二点の積は $\sqrt{a^2+b^2}\sqrt{c^2+d^2}(\cos(\alpha+\beta),\sin(\alpha+\beta))$ と回転と拡大によって得られるものであることが判ります. このことは, 三角函数の和の公式 $\cos(\alpha+\beta)=\cos\alpha\cdot\cos\beta-\sin\alpha\cdot\sin\beta$ と $\sin(\alpha+\beta)=\sin\alpha\cdot\cos\beta+\cos\alpha\cdot\sin\beta$, また $\cos\alpha=a/\sqrt{a^2+b^2}$, $\sin\alpha=b/\sqrt{a^2+b^2}$, $\cos\beta=c/\sqrt{c^2+d^2}$, $\sin\beta=d/\sqrt{c^2+d^2}$ から確かめることができます. 実際に Maxima で検証した様子を示しておきます:

```
(%i11) sqrt(a^2+b^2)*sqrt(c^2+d^2)*cos(alpha+beta),trigexpand,
       cos(alpha)=a/sqrt(a^2+b^2),sin(alpha)=b/sqrt(a^2+b^2),
       cos(beta) =c/sqrt(c^2+d^2),sin(beta) =d/sqrt(c^2+d^2),ratsimp;
(%o11)                                a c - b d
(%i13) sqrt(a^2+b^2)*sqrt(c^2+d^2)*sin(alpha+beta),trigexpand,
       cos(alpha)=a/sqrt(a^2+b^2),sin(alpha)=b/sqrt(a^2+b^2),
       cos(beta) =c/sqrt(c^2+d^2),sin(beta) =d/sqrt(c^2+d^2),ratsimp;
(%o14)                                a d + b c
```

このように複素数は 「1 変数多項式環の同値類」, 「実数の対」や「平面上の点」として見ることが可能です. そして, どちらの見方にせよ, 演算はその大本となる 1 変数多項式, 実数やベクトルの演算を継承し, それらを拡張したものとなっています.

ところが, 複素数を構成する上で用いた実数には大小関係がありますが, 複素数全体には大小関係を入れることはできません. このように既存の対象から自然に拡張のできる演算や関係がある一方で, 全体に拡張できない演算や関係も存在します.

4.2.6 ちょっとしたまとめ

ここまで同値関係を導入することで, 整数から有理数を表現したり, 1 変数の多項式を使って純虚数 i を表現しました. このことは整数が表現された世界なら有理数が構成可能で, その上, 整数上で定められた演算も有理数に自然に延長可能なこと, さらに 1 変数の実数多項式が表現可能であれば, 多項式を使って純虚数も表現できることをが判りました. そして, 整数上の演算は有理数上に自然に拡張され, 同様に, 多項式上の和, 差, 積といった演算も複素数環に自然に拡張されることも判りました.
この既存の対象を用いて新たな対象を生成する能力は数学では至って普通に見られることです. その際に, 既存の対象上の演算や関係は新しい対象上に拡張されたり, 逆このように同値関係を導入することで既存の対象から構成されたものを用いて新しい数学的対象が表現され, 同時に, 既存の対象の演算も導入できています. そのために数式処理システムは同値類を処理する系としての側面を持っています. たとえば, Maxima では演算子の属性 (性質)[6] や式に対する規則の定義が行え, それらによって Maxima に同値関係を処理するための仕組みがあります. そして, 現実問題として, 与えられた問題を数式処理システムで効率的に解くことは, 関係を如何に上手く設定できるかどうかに大きく依存します. ここで Maxima で関係を定めて処理を行う方法については, この本の §5.4 や §5.7 を参照して下さい.

さて, 整数から有理数を表現しましたが, 実数の表現には至っていません. そして, 同値類を用いて 1 変数多項式から純虚数 i を構築しましたが, そもそも, この 1 変数多項式をどのように導入すべきでしょうか？そして, 実数や 1 変数多項式が持つ性質にはどのようなものがあるのでしょうか？ ここでは, それらの疑問に答える前に, はじめに一般的な代数学の話をしておこうと思います. それから, 数学をより厳密なものへと変貌させようとした歴史の話もしておこうと思います.
そこで, この章の全般的な参考文献として, 丸山 ([57]), 寺坂 ([35]), Cox([77]) と Greuel-Pfister([80]) を挙げておきます. ここで, 寺坂 ([35]) は中学生にも拾い読みができる楽しい本ですが, 数学基礎論の章は流石に古いと思えるので, この辺の話題に関してはウリグト ([9]) や林 ([30]) の付録, 総合的な文献集として Heijenoort の本 ([81]) を挙げておきます. さらに数に興味がある方は高木貞治の名著「数の概念」([32]), 比較的最近の本では「数」([10]) を参考にされると良いでしょう.

4.3 群について

ここでは非常に重要な概念である群について簡単に解説しておきましょう. この「**群**」を天下り的に言うならば,「**群は性質の良い二項演算を持った集合**」となります. そこで, どのように性質が良いかを具体的に説明しましょう.

[6] 属性は property で個体の性質, attribute で類 (クラス) の二種類に分けられます.

ここで集合を A, 二項演算を “$*$” と表記します. このときに二項演算 “$*$” は集合 A の二つの元 a_1, a_2 に対して集合 A の元を対応させる函数です. この二項演算 “$*$” による a_1 と a_2 の像を $a_1 * a_2$ と表記します. なお, 一般的に $a_1 * a_2$ と $a_2 * a_1$ は異なることに注意して下さい. 何故でしょうか？ ‘$a_1 * a_2 = a_2 * a_1$’ という言明がありますか？ ここではまだ演算が存在する以上のことや具体的な性質については何も言及されていないからです. さて, 二項演算を誤解する恐れがないときには単に「**演算**」, 演算記号 “$*$” を「**演算子**」と呼びます. そして, 群を集合と演算の対 ‘$(A, *)$’ として表記しますが, 演算を省略しても問題のないときには群 A と簡単に記述します. さて, 表記 $(A, *)$ が群と呼ばれるためには演算 “$*$” が「**良い性質を持っている**」と述べました. そこで, 「**良い性質**」を列記しておきましょう.

まず, 集合 A の任意の二つの元 a, b に対して $a * b$ は必ず集合 A に含まれていなければなりません. この性質を持つ演算 “$*$” を持つ集合 A を「**集合 A は演算 “$*$” に関して閉じている**」と言います.
演算 “$*$” が閉じていれば, 任意の集合 A の元 a ,b, c に対して $a * b$,$b * c$,$(a * b) * c$ や $a * (b * c)$ も集合 A に含まれます. では任意の 3 個の元 a, b, c に対して $(a * b) * c$ と $a * (b * c)$ は集合 A の同じ元になるでしょうか？この保証は実は全くありません. この ‘$(a * b) * c = a * (b * c)$’ となる性質は「**結合律**」と呼ばれる性質で, $(A, *)$ が群になるためには, 結合律をまず満さなければなりません.
この結合律を満せば何が良いかと言えば, 最初に $a * b$ を計算して, その結果を使って c との演算を計算する $(a * b) * c$ と, $b * c$ を計算した結果と a との演算を計算する $a * (b * c)$ の両者に違いがないことが保証されるので $a * b * c$ と括弧を外して表記して良いことになります. そうでなければ, 演算の順番で結果が異なることを意味し, それを避けるために式は括弧 “()” だらけになって非常に見通しの悪いものになるでしょう[7]. この結合律の話はのちの式の表現でまた出てきます.
それから単位元という特別な元が集合 A に存在します. ここで集合 A の元 u が「**単位元**」であるとは, 任意の集合 A の元 a に対して ‘$a * u = u * a = a$’ を充す場合です. この単位元 u を 1 と表記することもあります. この単位元は存在すれば一つだけです. 何故なら, 単位元として u の他に v も存在したとして, $u * v$ を考えると, u が単位元なので ‘$u * v = v$’ となります. その一方で v も単位元なので ‘$u * v = u$’ となって両方を併せれば ‘$u = v$’ が得られるからです.
演算について閉じており, 演算が結合律を満して単位元が存在する $(A, *)$ を「**半群**, あるいは, 「**準群**」と呼びます. この半群が本当に群になるために必要な条件が逆元と呼ばれる元の存在です. 集合 A の元 b が集合 A の元 a の逆元と呼ばれるのは ‘$a * b = b * a = u$’ を充すときです. この元 b を a^{-1} と記述し, 逆元を持つ元のことを「**正則元**」と呼びます. $(A, *)$ が群になるためには集合 A の全ての元が演算 “$*$” に対して逆元を持つこと, すなわち, 集合 A の元が全て正則元でなければなりません.

では, ここで群となる条件を纏めておきましょう:

[7] とは言え, 式の構造, あるいは式の成り立ちは明瞭になります.

群の条件

- 演算 $*$ に対して閉じている:
  $a, b \in A \rightarrow a * b \in A$
- 結合律が成立:
  $(a * b) * c = a * (b * c)$ となる
- 単位元 1 の存在:
  $a * 1 = 1 * a = a$ となる $1 \in A$ が存在する.
- 逆元の存在:
  任意の $a \in A$ に対し, $a * b = b * a = 1$ を充す $b \in A$ が存在する.

演算 "$*$" が常に '$a * b = b * a$' を充す場合, 演算 "$*$" を「**可換 (commutative)**」, 群 $(A, *)$ のことを「**可換群**」と呼びます. 可換でなければ「**非可換 (noncommutative)**」, 非可換な演算を持つ群を「**非可換群**」と呼びます. ちなみに Maxima では記号 "$*$" を可換積に用い, 記号 "." を非可換積で用います. なお, Maxima で和 "+" は常に可換です.

4.3.1 群の例

☆ $(\mathbb{Z}, +)$**:整数全体, 演算が和**

群 $(\mathbb{Z}, +)$ の単位元は 0 で, $a \in \mathbb{Z}$ の逆元は $-a$ になります. 和 "+" は可換なので, この群は可換群になります.
整数 $\mathbb{Z}$ には積 "$*$" もあり, 積 "$*$" の単位元は 1 ですが, 積 "$*$" については 1 以外は逆元を持ちません. しかし, 結合律を充すので $(\mathbb{Z}, *)$ は半群になります.

☆ $(\mathbb{Q} - \{0\}, *)$**:0 を除く有理数全体, 演算が積**

有理数全体の集合 $\mathbb{Q}$ から 0 を抜いた集合 $\mathbb{Q} - \{0\}$ の全ての元は整数 a, b を使って a/b と記述できます. 演算 "$*$" の単位元は 1 で, その逆元は b/a です.
なお, $(\mathbb{Q}, *)$ は 0 が逆元を持たないことから, 積 "$*$" に関して半群になりますが群にはなりません. その一方で $(\mathbb{Q}, +)$ は可換群になります. このように演算によって群になったりならなかったりする集合もあります.

☆ $(SL(m), *)$

行列式の値が 1 になる m 次の正方行列の集合 $SL(m)$ は行列の積 $*$ に対して群になります. ただし, 任意の $a, b \in \mathrm{SL}(m)$ に対し, $a * b$ が $b * a$ となるとは限らないので, $(\mathrm{SL}(m), *)$ は可換群になりません. m 次の正方行列全体の集合 $M(m)$ の場合, 全ての元が逆元を持つとは限らないために $(M(m), *)$ は群になりません. ただし, m 次の単位行列が単位元とし, 結合律も成立するので, $(M(m), *)$ は半群になります.

なお, Maxima では行列の積を非可換積 "." で, その冪を演算子 "^^" で表記しています.

☆語

集合 A を a から z までのローマ小文字を並べた全ての文字列 (「**語**」と呼びます) と空白 " "(ここでは 1 と表記します) を元とする集合とします. この集合に対して演算 "$*$" を単純に二つの語を繋ぐ操作とします.
たとえば, $abc*def$ の結果は $abcdef$ のように演算子 "$*$" の右の語を演算子 $*$ の左の語の末尾に繋げます. なお, 空白を語の左右に繋いでも元の語となるので, 空白が演算 $*$ の単位元になることが判ります. また, $WWWWW\ldots$ のように同じ語 W が n 回続く場合を W^n と表記します.
語 W_1, W_2, W_3 に対して $(W_1 * W_2) * W_3$ と $W_1 * (W_2 * W_3)$ は同じ語 $W_1 W_2 W_3$ になりますね. このことから演算子 "$*$" は結合律を充すことが判ります. このように単位元が存在し, しかも, 結合律を充すので, $(A, *)$ には半群の構造が入ります.
次に, 語 W に対して W^{-1} を考えます. この W^{-1} は $W * W^{-1}$ と $W^{-1} * W$ を空白で置換する操作とします. ここで, 語 w が 2 個以上のアルファベットで構成された語であれば, そのアルファベットの並びを逆にし, 各文字を対応する文字の除去操作に対応します.
たとえば, 語 W が $neko$ であれば W^{-1} は $o^{-1}k^{-1}e^{-1}n^{-1}$ とします. こうすることで語 W^{-1} は語 W の逆元となります. それから, 最後に空白の逆元を空白とすると, 全ての A の元に逆元が定まるので $(A, *)$ は群になります.
この群を $\langle a, \ldots, z\rangle$ と記述し, "$\langle\rangle$" の中の $a,\ \ldots, z$ を「**群の生成元**」と呼びます. そして, このように記述される群を「**自由群**」と呼びます. ちなみに, この群は非可換群になります. 何故なら, 演算 $*$ の逆元はあっても, $a*b = b*a$ という可換性が成立する保証が何処にもないからです.
Maxima では, このような群の積は非可換積 "." で表現できます. さらに, 非可換積の冪乗は通常の冪 "^" ではなく非可換積の冪 "^^" を用います.

☆ $\langle x, y | x\,y\,x^{-1}\,y^{-1}\rangle$

自由群 $\langle x, y\rangle$ に対して「**規則**」を入れた群を定義してみましょう. 生成元の x と y に対し, 式 $x\,y\,x^{-1}\,y^{-1}$ を単位元 1 で置換する規則とします. さて, この規則を入れた群 G を $\langle x, y | x\,y\,x^{-1}\,y^{-1}\rangle$ と表記し. 規則 $x\,y\,x^{-1}\,y^{-1}$ を「**関係子 (relator)**」と呼びます.
では, 具体的な規則の適用を見てみましょう. たとえば, 語 $x\,x\,y\,x^{-1}\,y^{-1}\,x$ の中には関係子 $x\,y\,x^{-1}\,y^{-1}$ がありますね. この関係子の個所を分かり易く括弧で括ると $x\,(x\,y\,x^{-1}\,y^{-1})\,x$ になります. それから規則にしたがって, この括弧の個所を 1 で置換えると x^2 が得られます.
ここで注目して頂きたいことは, 語の並びを検出して関係子で規定される規則をあてはめる操作を上の処理で行ったことです. 実際, 計算機にとって与えられた式は最初は文字の羅列でしかありません. この羅列に意味を持たせる処理を Maxima では行っており, さらに, 利用者がこのような操作を Maxima にさせることもできます. この処理は Maxima では非常に重要な処理になります. このことは §5.7 でより詳しく述べます.

さて, 群 $\langle x, y | x\,y\,x^{-1}\,y^{-1}\rangle$ はどのような群でしょうか？ 語 $y\,x$ は $1\,y\,x = x\,y\,x^{-1}\,y^{-1}\,y\,x$ なので, 式の右辺を計算すれば, $x\,y = y\,x$ という関係が得られます. すなわち, この群は二つの元 x と y で生成される可換群になります.
すると, こんな写像 f が見えませんか？

$$\begin{array}{ccccc} f & : & \langle x, y\rangle & \to & \langle x, y | x\,y\,x^{-1}\,y^{-1}\rangle \\ & & \cup & & \cup \\ & & (x, y) & \mapsto & (x, y) \end{array}$$

この写像 f は自由群 $\langle x, y\rangle$ から $\langle x, y | x\,y\,x^{-1}\,y^{-1}\rangle$ への写像で, 単純に自由群の x と y を群 G の x と y にそのまま対応させている写像です. この写像は「**準同型写像**」と呼ばれます.
次に, 集合 $H = \{r | r = h\,x\,y\,x^{-1}\,y^{-1}\,h^{-1}, r = h\,y\,x\,y^{-1}\,x^{-1}\,h^{-1}, h \in \langle x, y\rangle\}$ を考えると, この集合 H は群 G の部分群になることが判ります. この群は面白い性質があり, 任意の $h \in G$ に対して $hHh^{-1} = H$ を満します. この性質を充す群 G の部分群を「**正規部分群**」と呼びます.
さらに, 準同型写像 f による像 $f(H)$ は 群 G の単位元 1 になります. この部分群 H を写像 f の「**核 (kernel)**」と呼びます.
この部分群 H を用いて群 G を別の見方をしてみましょう. まず, 自由加群 $A = \langle x, y\rangle$ に次の同値関係 "$\sim$" を入れます:

$$a \sim b \Leftrightarrow a\,b^{-1} \in H \qquad \text{ここで } a, b \in \langle x, y\rangle$$

この関係 "$\sim$" による同値類の集合 $A/\sim$ も群になります. これを群 A を群 H による「**剰余群**」, あるいは簡単に, 「群 A を群 H で割ったもの」と呼び, A/H と表記します. すると, 群 G は A/H と一致します.
より一般的には群 $G = \langle x_1, \ldots, x_m | r_1, \ldots, r_n\rangle$ は自由群 $A = \langle x_1, \ldots, x_m\rangle$ をその正規部分群 $H = \langle hr_1h^{-1}, hr_1^{-1}h^{-1}, \ldots, hr_nh^{-1}, hr_n^{-1}h^{-1} | h \in G\rangle$ で割ったもの $G = A/H$ と同じ群になります.

4.4 環について

群に続いて重要な概念である環について解説しましょう. ここで整数の集合を思い出して下さい. この整数の集合 $\mathbb{Z}$ には二つの演算: 和 "+" と積 "$*$" があります. そして $(\mathbb{Z}, +)$ は可換群になる一方で, $(\mathbb{Z}, *)$ には逆元が 1 以外に存在しないために群にならずに可換半群になります.

この整数の集合 $\mathbb{Z}$ のように二つの閉じた演算を持ち, 次に列記した性質を持つ $(A, +, *)$ を「**環**」と呼びます:

環の条件

- 集合 A には二つの演算の積 “$*$” と和 “$+$” があり, これらの演算に対して閉じている.
- $(A, +)$ は可換群になる. この時, 0 が演算 “$+$” の単位元となる
- $(A, *)$ は半群になる. 演算 “$*$” の単位元は 1 である
- 演算 “$+$” と演算 “$*$” は以下の分配律を持つ.
  左分配律:　$a * (b + c) \quad = \quad a * b + a * c$
  右分配律:　$(a + b) * c \quad = \quad a * c + b * c$

ここで積 “$*$” が可換であれば「**可換環**」と呼びます. このときは, 左右の分配律に区別はありません.

零因子

環によっては「**零因子**」と呼ばれる元を持つことがあります. これは環 $(A, +, *)$ の 0 と異なる元 a, b で, ‘$a,\ b \in A$ で ‘$a * b = 0$’ を充すものです.

この零因子を持つ環の例として 2×2 の実数行列で構成された一般行列環 $(\mathrm{M}(\mathbb{R}, 2),\ +,\ *)$ を挙げておきます. まず, この行列環での 0 は全ての成分が 0 の元 $\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$ です.

ところが $a = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ は零行列と異なるものの, ‘$a * a = 0$’ となることが容易に確認できるので, 零因子であることが判ります.

整域

零因子が存在しない可換環を「**整域**」と呼びます. 整域の例として, 整数環 $(\mathbb{Z},\ +,\ *)$ を挙げておきます.

4.4.1　イデアル

環には「**イデアル (ideal)**」[8]と呼ばれる環 $(A,\ +,\ *)$ の部分集合 I があります:

[8] Kummer の理想数 (Ideal numbers) を Dedekind が改良したものです.

イデアルの定義

- 集合 I は和に関して群になります.
- 環 $(A, +, *)$ の任意の元 a に対して $a * I \in I$ を充す場合, I を「**左イデアル**」と呼びます.
- 環 A の任意の元 a に対して $I * a \in I$ を充す場合, I を「**右イデアル**」と呼びます.
- I が左イデアル, かつ, 右イデアルのときに I を「**両側イデアル**」, あるいは単に「**イデアル**」と呼びます.

可換群であれば右左のイデアルの区別はありません. (左右) イデアルは和 "+" の単位元 0 を含みますが, イデアルに積 "$*$" の単位元 1 を含む場合は環そのものと一致します. そのため, イデアルが環 $(A, +, *)$ の真部分集合であれば積 "$*$" の単位元 1 を含みません.
イデアルにはいろいろな性質を持つものがあります. 代表的なイデアルを示しておきましょう:

- **単項イデアル**
  イデアル I が一つの元 a だけで生成される場合, イデアル I を「**単項イデアル**」と呼びます. なお, 環 R の イデアルが全て単項イデアルとなる環を「**主イデアル整域** (PID(Principal Ideal Domain)」と呼びます.
- **素イデアル**
  $a\ b \in I$ であれば $a \in I$ か $b \in I$ の何れかが成立するイデアル I を「**素イデアル (prime ideal)**」と呼びます. 丁度, 整数環 $\mathbb{Z}$ の素数のようなものです.
- **極大イデアル**
  $R \supset J \supset I$ を充す環 R のイデアル J が存在しない環 R のイデアル I を「**極大イデアル (maximal ideal)**」と呼びます. 一般的に極大イデアルは一つだけとは限りません. 極大イデアルが一つだけしか存在しない環を「**局所環 (local ring)**」と呼びます.

4.4.2 環の例

☆多項式環

有理数 $\mathbb{Q}$ を係数とする n 変数多項式 $\sum_{i=0}^{n} a_i x^i$ の集合を $\mathbb{Q}[x_1, \ldots, x_n]$ と記述します. すると, $(\mathbb{Q}[x_1, \ldots, x_n], +, *)$ は和 "+" に関して群, 積 "$*$" に対して半群になります. そして, これらの演算に対して分配則が成立し, 積 "$*$" が可換なので, $(\mathbb{Q}[x_1, \ldots, x_n], +, *)$ は可換環になります.

☆群環

一般的な群 G と可換環 R が与えられたときに環 R を係数とする G の元の形式的な有限個の和を考えることで, 「群環」(「**多元環**」とも呼びます) RG が定義できます. この群環は Maxima の動作原理を考える際に非常に重要なものです.

そこで, 可換環を整数環 $(\mathbb{Z}, +, *)$, 群 G を $\langle x_1, x_2, \ldots, x_n | r_1, \ldots, r_m \rangle$ とするとき, 群環 $\mathbb{Z}G$ の構成方法を具体的に解説しましょう.

まず, 群 G の元 g と環 $\mathbb{Z}$ の元 a に対して形式的な積 $a\,g$ を定めます. この際, '$a\,g = g\,a$' とします. なお, $0 \in \mathbb{Z}$ に対しては '$0\,g = 0$' とします. 次に, 群 G の有限個の元 $g_1, \ldots, g_m$ と環 $\mathbb{Z}$ の元 $a_1, \ldots, a_m$ との積の形式的な和 $a_1\,g_1 + \cdots + a_m\,g_m$ の集合を $\mathbb{Z}G$ とします. この形式的な和に関して $f, g, h \in G$ とすると, 「和の可換性」'$f + g = g + f$', 「和の結合律」'$(f + g) + h = f + (g + h)$' を入れます. こうすることで $\mathbb{Z}G$ は「和」"+" に関して可換群になります. 次に, '$f\,(g + h) = f\,g + f\,h$' と '$(g + h)\,f = g\,f + h\,f$' で左右の分配律を定義すると集合 $\mathbb{Z}G$ は環になります.

このとき, 群環 $\mathbb{Z}G$ の積に可換性を入れてしまえば整数係数の n 変数多項式環 $\mathbb{Z}[x_1, \ldots, x_n]$ の剰余環になります.

☆整数環 $\mathbb{Z}$ のイデアル

2 の倍数の数の集合 (2) , 3 の倍数の数の集合 (3) と 6 の倍数の数の集合 (6) を考えましょう. これらは全て整数環 $\mathbb{Z}$ のイデアルとなります. 実際, 2 の倍数の元の和は 2 の倍数の数の集合に属し, 任意の $a \in \mathbb{Z}$ に対し, $b \in (2)$ との積 $a\,b$ は当然 2 の倍数となります. 他の (3), (6) も同様です.

さらに, $a, b \in \mathbb{Z}$ に対し, $a\,b \in (2)$ であれば, a か b の何れかが必ず, (2) に含まれます. これは (3) も同様です. このように整数環 $\mathbb{Z}$ の素数 p で生成されるイデアル (p) は素イデアルになります.

その一方で (6) に 2 と 3 は含まれませんが, '$2 * 3 = 6$' となるので, (6) は素イデアルではありません. また, 6 は 2 と 3 の倍数のため, '$(2) \supset (6)$' かつ '$(3) \supset (6)$', さらに, '$(2) \cap (3) = (6)$' を満します. すなわち, '$2 * 3 = 6 \Leftrightarrow (2) \cap (3) = (6)$' という対応関係があります.

整数環 $\mathbb{Z}$ のイデアルは全て単項イデアルになります. 何故なら, I を $\mathbb{Z}$ のイデアルとし, $a \in I$ をイデアル I に含まれる最小の正整数としましょう, すると, I が a で生成されることが判ります. 実際, $b \in I$ で '$b = a * q + r$' となる $r < a$ が存在すると, '$r = b - a * q$' から $r \in I$ となり, a が I に含まれる最小の正整数であることに矛盾するので, $r = 0$, すなわち, $b = a\,q$ となるので I は最小の正整数 a で生成されることが判ります. 以上から, 整数環 $\mathbb{Z}$ は主イデアル整域 (PID) になります.

☆有理数環 $\mathbb{Q}$

$(\mathbb{Q}, +, *)$ は和 "+" に対して可換群, 積 "*" に対しては半群になり, 積 "*" は可換であり, 和 "+" と積 "*" が分配律を充すことから可換環になります. さらに, $\mathbb{Q}$ の 0 を除く元 a/b は全て逆元 b/a を持ちます. そのため, $\mathbb{Q}$ のイデアルは 0 で生成される (0) 以外のイデアルはありません. このことから (0) は有理数環 $\mathbb{Q}$ の唯一の極大イデアルとなるので $\mathbb{Q}$ は局所環になります.

☆可換環 R の局所化

R を可換環とし, p を R の素イデアルとしましょう, ここで R から p の元を除去した集合 $R-p$ は積 "$*$" に関して閉じており, 和 "+" の単位元 0 を含まない集合になります. この積演算で閉じた性質を持った集合を「**積閉集合**」と呼びます.

次に, $(R-p)^{-1}R \overset{def}{=} \{(a,b)|a \in R, b \in b\}/\sim$ を考えます. そして, ここでの同値関係 "~" を '$(a,b) \sim (c,d) \Leftrightarrow ad-bc=0$' で定めます. この関係は自然数対から有理数を構成する際に用いましたね. そして同値類の代表元を a/b と記述しましょう. この操作で生成された環 $R/(R-p)$ は極大イデアルとして $p \cdot (R-s)^{-1}R$ のみを持つ局所環になります. そして, この操作を「**局所化**」と呼びます.

なお, 有理数環 $\mathbb{Q}$ は整数環 $\mathbb{Z}$ の局所化で得られた環, すなわち '$\mathbb{Q} \equiv (\mathbb{Z}-(0))^{-1}\mathbb{Z}$' として解釈できます. このように整数から有理数を構成する手法は可換環の局所化として, より一般的な局所環の構成方法として昇華されたのです.

☆集合

$\mathfrak{P}(S)$ を集合 S の冪集合とし, 和を "$\cup$", 積を "$\cap$" とします. このときに $(\mathfrak{P}(S), \cup, \cap)$ は $A, B \in \mathfrak{P}(S)$ に対し '$A \cup B = B \cup A$' と '$A \cap B = B \cap A$' が成立するため, 演算 "$\cup$" と "$\cap$" は可換演算になります. さらに演算 "$\cup$" と "$\cap$" はそれぞれ結合律を満します. 空集合 $\emptyset$ に対しては, '$A \cup \emptyset = A$' となるので, 空集合 $\emptyset$ は演算 "$\cup$" に対する単位元になります.
また, 演算 "$\cup$" と演算 "$\cap$" に関しては次の分配律が成立します:

- $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

補集合を取る演算 "^{c}" に対しては「**De Morgan の法則**」と呼ばれる性質があります:

- $(A \cup B)^c = A^c \cap B^c$
- $(A \cap B)^c = A^c \cup B^c$

ただし, 演算 "$\cup$" に対しては集合 A の逆元が存在しないため, $(\mathfrak{P}(S), \cup, \cap)$ は可換環にはなりません.

4.5 体について

「**体**」は可換環で, さらに, 零元 0 を除いた任意の元が積 "$*$" に関して逆元を持つ環のことです. たとえば, 有理数環 $(\mathbb{Q}, +, *)$ は可換環であり, その零元 0 を除く全ての元 a/b は逆元 b/a を持つことから体になります. なお, 非可換環で零元 0 を除いた任意の元が積 "$*$" に対して逆元を持つ場合は「**斜体**」と呼びます.

☆**剰余環** R/I

環 R とそのイデアル I が与えられたとき, $x,\ y \in R$ に対し, $x-y \in I$ であれば, $x \sim y$ とします. この関係 "$\sim$" は同値関係になります. そこで, 同値関係 "$\sim$" の同値類の集合 $R/\sim$ を R/I と記述します. このとき R/I は環になり, 環 R/I を「**剰余環**」, あるいは, 紛らわしいのですが「**商環**」とも呼びます.

可換環 R のイデアル I が素イデアルであれば R/I は整域になります. 実際, $\bar{x} * \bar{y} = 0 \Leftrightarrow x * y \in I$ より, $x \in I$ か $y \in I$, すなわち, $\bar{x} = 0$ か $\bar{x} = 0$ の何れかが成り立ちます. さらに, イデアル I が極大イデアルの場合, R/I は体になります.

4.6　準同型写像について

既に「**準同型写像**」という言葉が出ていますが, この準同型写像は, 群や環の性質を調べる上で非常に性質の良い性質を持った函数のことです. ここでは, どのように性質が良いか説明しましょう.

群の準同型写像:　群 G_1, G_2 が与えられ, それらの演算を "$*_1$" と "$*_2$" とします. このとき, 写像 $f : G_1 \to G_2$ が群準同型写像となるためには, 第一に演算を保つ, すなわち, '$f(a_1 *_1 b_1) = f(a_1) *_2 f(b_1)$' を満します. それから, 単位元を保ちます. これは, 群 G_1 の単位元を 1_1 とすると '$f(a_1) = f(1_1 *_1 a_1) = f(1_1) *_2 f(a_1)$' を充すことから判ります. これらを次に纏めておきましょう:

$$f(a_1 *_1 a_2) = f(a_1) *_2 f(a_2) \tag{4.1}$$
$$f(1) = 1' \tag{4.2}$$

環の準同型写像:　環 R_1 と R_2 が与えられ, 各環の積と和を $(*_1, +_1)$ と $(*_2, +_2)$ としまます. このとき, 環準同型写像は次の性質を満します:

$$f(a_1 +_1 a_2) = f(a_1) +_2 f(a_2) \tag{4.3}$$
$$f(a_1 *_1 a_2) = f(a_1) *_2 f(a_2) \tag{4.4}$$
$$f(1) = 1' \tag{4.5}$$

なお, 準同型写像 $f : A \to B$ が写像として 1 対 1 の対応の写像であれば「**単射**」, また, '$f(A) = B$' であれば「**全射**」, そして, 単射かつ全射であれば「**全単射**」, あるいは「**同型**」と呼びます.
群や環の準同型写像 $f : A \to A$ の集合を考えると, 写像の合成 "$\circ$" を積とする半群となります. 実際, f と g を A から A への準同型写像とすると, $f \circ g$ は A から A の準同型で, '$(f \circ g) \circ h = f \circ (g \circ h)$' を満し, 単位元 1 は恒等写像 $1 : A \to A$ が対応します.

良く使われる用語に, 準同型写像 $f : A \to B$ の「**核**」 $ker(f)$ があります. これは B の単位元を 1_B とするとき, 準同型 f による 1_B の逆像 $f^{-1}(1_B)$, つまり, 準同型写像 f によって B の単位元 1_B に写さ

れる元で構成される集合です. この $ker(f)$ は A, B が環の場合は環 A のイデアルになり, A, B が群の場合は群になります.

ここで数式の処理に戻ると, 数式は多項式環に幾つかの函数を追加した環の元として考えられます. 二つの異った書式の数式が等しいとは, 式の変形, 函数や変数に対して与えた規則によって共通の式に変形可能であるという同値関係 “~” をこの環に入れたものとして考えられます. したがって, 規則は自然な写像から誘導される準同型の核を構成するものとして考えられます.

4.7 数式の表現

重要な概念について概要を述べたので, それらの概念を活用して数式の処理について具体的に考えてみましょう. ここで最初に問題となることに, 与えられた数式が等しいかどうかをどうやって判断するかということです. この点について, 与えられた二つの数式が等しいとは「**式の展開や代入といった処理に加えて同値関係を利用することで互いに共通の式に変形できること**」としています. では, このことを計算機でどのように処理すべきでしょうか？ また, どうすれば効率良く処理が行えるでしょうか？そのためには式そのものの表現についてにも考慮する必要があります.

たとえば, 数式 $x+y$ と $y+x$ を表現する文字列 “x+y” と “y+x” が与えられたとき, これらの入力された文字列が示す数式が同じものであるかどうかは計算機にとって自明ではありません. 第一, これらの式は ASCII 文字の列で表現され, 文字列そのものは文字通り計算機にとって単なる文字の羅列でしかありません. さらに, これらの文字列は文字 “x” と文字 “y” の順序が互いに逆なので文字列としても一致していません. これらの文字列が等しい数式であると計算機に認識させるためには, **どのような規則が存在しているのかを, その「いろは」から計算機に教えてやる必要**があります. さて, これらの数式 $x+y$ と $y+x$ は被演算子の順序を変更した式であることは式を一目見れば分かることですが, ここで被演算子の入換えが可能 (演算子の可換性) という演算子の属性がなければ入れ換え操作が行えなければなりませんが, 幸いにして和という演算では被演算子の交換を行っても結果に違いはないので入れ換えが可能です. したがって, これらの数式に対応する文字列 “x+y” と “y+x” については文字列中の記号 “+” の左右の文字の置換が許容されるので互いに移りあえることが判ります. この入れ換えは無差別に行う訳にはゆきません. 数式 x/y と数式 y/x を表現する文字列 “x/y” と “y/x” であればどうでしょうか. この場合, “x” と “y” の入れ換えを行ってしまえば互いに移りあえますが, 演算子 “/” が非可換であるため, この被演算子の交換は許容されません. だから “x/y” と “y/x” は等しくないと判断できます.

このように数式とその数式を表現する文字列との間には, 数式の同値性を保つ式の変形に対応する文字列の変換の存在が必要であり, さらに同値性を保つ文字列操作によって得られた文字列に対応する数式も変換前の文字列に対応する数式と同値でなければなりません. すなわち, 計算機は文字列として与えられた与件に含まれる演算子 (の表現) を抽出して, 演算子の属性から規定される規則を用いて式を表現する文字列を操作した結果, 二つの文字列が互いに移り合えば等しい式と判断させることを意味します. そこで, 数式とその数式を表現する文字列を区別せずに, 以後, 簡単に数式,

あるいは式と呼ぶことにしましょう.

さて, 二つの式 A と式 B が式に含まれる演算子の性質を利用して, 互いに移り合える場合に '$A \sim B$' と記述します. すると, この二項関係 "~" は同値関係になります. 実際, 最初の反射律については問題はないでしょう. それから, 対称律は式 A から B に変形可能なら, その逆操作を行なえば良いのでこれも大丈夫です. 推移律は式 A から B への操作に続けて B から C への変換操作を行えば良いので, この一連の操作によって式 A から C に変換できるので, これも問題ありません. したがって, 二項関係 "~" は同値関係になります.

さて, 変形処理の関係 "~" による同値類の代表の選び方を考えましょう. 数式 $x + y$ の場合, 演算子を鍵として演算子の属性 (可換性) から被演算子を入換える規則で式の変形を行い, 同じ式が得られれば等しいと判断します. そこで非常に安易ですが, 鍵となる演算子を先頭に置いて,"+ x y" のような表記で計算機に処理させればどうでしょうか？もちろん "+ x y" のような表記をでは, 複雑な式に対しては人間にとっても判り難くなるで "(+ x y)" と括弧で括ってしまいましょう. この表記は何処かで見たことがありますね. 実際, LISP の S 式と同じ格好になっています. この表記 "(+ x y)" のように演算子を前に置いた数式の表現を「**前置表現**」, あるいは「**ポーランド記法**」と呼びます. また, $x + y$ のように演算子が二つの被演算子の中に置かれた数式の表現を「**内挿表現**」, あるいは「**中置表現**」と呼びます.

では引数を追加して $x + y + z$ はどうでしょう？ この式では最初の演算 "+" の前の x が第 1 引数で $y + z$ が第 2 引数となり, 先程の表記では "(+ x (y + z))" になります. ここで部分式の $y + z$ も同じ考え方で処理すれば "(+ y z)" となり, 最終的には "(+ x (+ y z))" が得られ, 演算子を節, 被演算子を葉と看做せば, 図 4.1 に示す「**木構造**」と呼ばれる構造が数式に入ります:

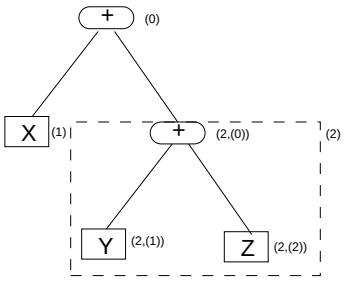


図 4.1: $X + (Y + Z)$ の木構造図

この演算子 “+” はどんな性質を持っているでしょうか？ 最初に「**可換性**」‘$a+b=b+a$’ があります. 次に「**結合律**」‘$(a+b)+c=a+(b+c)$’ も成立します. この結合律がある御陰で括弧を外して $a+b+c$ と記述できます. したがって, 我々の式の表現 “(+ (+ a b) c)” と “(+ a (+ b c))” も共通の形式で表現されるべきです. この $a+b+c$ は演算の括り方は問わず, その部分式は a, b, c のみのため, “(+ a b c)” と平坦に表記する方が妥当です. これと似た演算子に可換積 “$*$” かあります. 可換積 “$*$” も同様に $x_1 * x_2 * \cdots * x_n$ を $(* \; x_1 \; x_2 \; \cdots \; x_n)$ と表記します. この考え方は Maxima の「**nary 型**」演算子の扱いに反映されています.

次に, 差と商に関しては, $a-b$ を $a+(-b)$, a/b も $a*(1/b)$ で置換します. 最後の冪に関しては a^b を $(\hat{} \; a \; b)$ で表現すれば良いでしょう. すると, 数式は数, 変数や函数に, それらで構成された部分式を演算子 “+” や演算子 “$*$” 等の演算子で結合したものとして表現できます.

たとえば, $1+x+(y-1)/(z^3+2xy)$ を LISP の S 式で表現すると,
“(+ 1 x (/ (+ y -1) (+ (^ z 3) (* 2 x y)))” となり, その木構造は図 4.2 に示すものとなります:

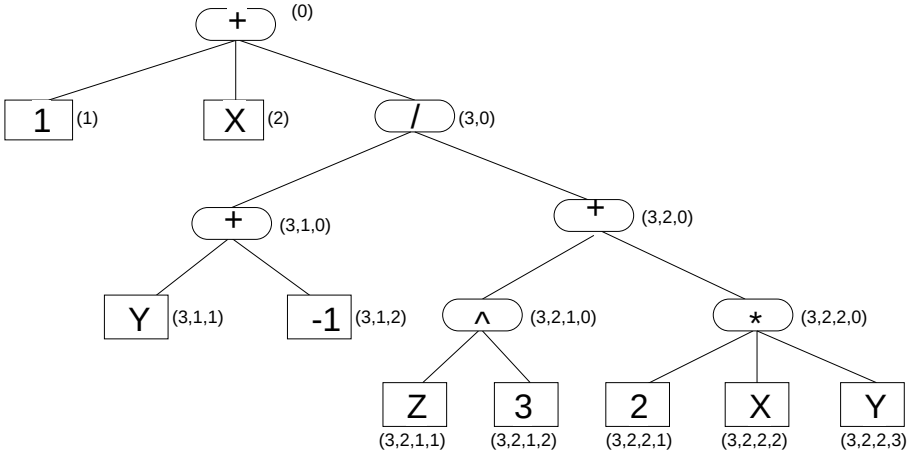


図 4.2: $1+x+(y-1)/(z^3+2*x*y)$ の木構造図

木構造と S 式は 1 対 1 に対応が付きますが, 式には項の順番の問題があって一意に定まりません. たとえば, 数式 $x+y+z$ は項の並び換えにより, 数式 $y+x+z$ や数式 $x+z+y$ 等の 6 通の並び方があります. 機械的に処理する際に式が一意に決らないことは困りますね. 実際, 数式 $x+y+z$ の第 1 項 x に 1 を代入するといった処理を行う場合でも, 並びが一意に決っていなければ, 式の項を頭から一々確認しなければなりません. ところが, ある規則で式の並びが一意に決定するようにしておけば, あとは計算機にその規則を教えてしまえば安心して処理をさせられますね. ではとうすれば良いでしょうか？ そこで項に順番, つまり, 「**順序関係**」を入れましょう. 具体的には, 変数をアルファベット順に並べるようにすれば如何でしょうか？ すると, 数式 $y+z+x$ は $x+y+z$ になって式の表示が一通りに決ります.

このように, 変数に順序を入れることで式の表示が一意に定まりそうです. ここで「**順序**」と簡単

に述べていますが, 順序は数学ではどのように定義されるべき概念なのでしょうか？ 次の節では順序について考えましょう.

4.8　順序について

数学では「**順序 (order)**」という考えがあります. この順序は集合 S の二つの元に対して成立する関係 (二項関係と呼びます) の一つです. たとえば, 集合を実数とすると大小関係 "≥" は順序関係の一例で, 逆に言えば, 順序関係は, この大小関係を一般化したものです.

それでは順序について定義を述べることにしましょう. 集合 S と, その集合の任意の二つの元の関係 "$\geq_{\text{order}}$" が以下の性質を充すときに「**順序**」, 順序の入った集合 S のことを「**順序集合**」と呼びます:

順序の定義

反射律:	$x \geq_{\text{order}} x$				
対称律:	$x \geq_{\text{order}} y$	かつ	$y \geq_{\text{order}} x$	$\Rightarrow$	$x = y$
推移律:	$x \geq_{\text{order}} y$	かつ	$y \geq_{\text{order}} z$	$\Rightarrow$	$x \geq_{\text{order}} z$

集合 S の順序 "$\geq_{\text{order}}$" で集合の任意の元 x, y に対し, '$x \geq_{\text{order}} y$', あるいは, '$y \geq_{\text{order}} x$' の何れかが成立する場合, 順序 "$\geq_{\text{order}}$" を「**全順序**」, 集合 S を「**全順序集合**」と呼びます.

たとえば, 集合を実数 $\mathbb{R}$ とすると, 二項関係 "≥" は全順序, そして, 集合 $\mathbb{R}$ が全順序集合になります. 次に, 二項関係を包含関係 "⊇" とすると, この包含関係 "⊇" は集合 $\mathbb{R}$ の順序になりますが, 今度は全順序にはなりません. 何故なら, 偶数全体の集合 A と奇数全体の集合 B を考えるとどうでしょうか？集合 A と集合 B の間には包含関係がありませんが, 全順序となるためには任意の二つの元に対して包含関係がなければなりません. ただし, この包含関係のように, 任意の元 a, b に対して二項関係が成立するとは限らない順序を「**半順序**」と呼びます.

この半順序関係を使えば構造がきちんと定義できます. まず, 関係 $<$ を持つ集合 T が「**木構造**」を持つとは, 関係 $\leq$ が集合 T にて半順序, この半順序 $\leq$ について最小限 $x_0 \in T$ が存在して, 任意の $x \in T$ に対して定まる部分集合 $T_x \stackrel{\text{def}}{=} \{y | x \leq y\}$ にて関係 $>$ が全順序となるときです. ここで最小限 x_0 のことを「**根**」と呼びます. また, $x \leq y$ を充す $x, y \in T$ について, $x \leq z \leq y$ を充す $y \in T$ が存在しないときに y を x の「**子**」, 逆に x を y の「**親**」と呼びます. そして木構造を持つ $(T, \leq)$ の集合 T の成員を「**節**」. $y \leq x$ となる $y \in T$ が存在しない $x \in T$ のことを「**葉**」と呼びます.
さて, 数式処理が扱う対象を考えてみましょう. ここで扱う対象の集合は, 整数, 有理数といった一般の数, 変数や函数で構築された式から構成されるでしょう. このとき, 式は数, 変数や函数を演算子で結合した対象となりますが, 数については大小関係で順序関係を入れ, 変数や函数 (名) に対しては, 辞書で採用されている方式で順序が入れられます. そして, 数と変数に対しては, 変数の方がより大きく, 変数と函数に対しては, 変数よりも函数が大きいと順序付けると, 変数の積で構成された項の順序が残ります.

ここで考えている世界が 1 変数の世界であれば, 単項式 x^m と x^n との順序を冪の次数 m と n の大小関係 “$\geq$” で決めれば良いでしょう. では, 多変数の場合はどうすれば良いのでしょうか？ この場合, 変数に入れた順序で項を構成する変数の次数リストで比較する方法があります.

たとえば, x, y, z の 3 変数多項式環 $K[x, y, z]$ で, 変数間の順番をアルファベット順で並べます. ここで yz のように変数が抜けていれば, 抜けている変数の次数を 0 とします. すると, 項は $x^{i_1} y^{i_2} z^{i_3}$ の形式になります. これから長さ 3 の次数のリスト (i_1, i_2, i_3) が得られ, このリストと項は一対一に対応します. したがって, n 変数 $x_1, \ldots, x_n$ の単項式 A と B が与えられたとき, n 個の変数 $x_1, \ldots, x_n$ の並びを固定します. 次に, 単項式 A と B の中の変数 x_i が項に存在しない場合, x_i^0 を挿入します. こうすることで単項式 A と B を表現する長さが n の次数リスト $(\alpha_1, \ldots, \alpha_n)$ と $(\beta_1, \ldots, \beta_n)$ が一意に決ります. 後はこれらの次数リストに対して順序を入れれば良いのです. ここで, 項の順序としては二つの項 x^α , x^β に対して, ‘$x^\alpha > x^\beta$’ であれば, x^γ を両辺にかけた場合でも, ‘$x^{\alpha+\gamma} > x^{\beta+\gamma}$’ を充すものの方が都合が良いですね. この項に対する性質 ‘$x > y \Leftrightarrow x \cdot a > y \cdot a$’ を充す順序を「**項順序**」と呼びます.

では, 先程の例の二つの次数リスト $x^2 y^2 z (= (2, 2, 1))$ と $x y^2 z^3 (= (1, 2, 3))$ が得られたとき, 順序はどのように入れれば良いでしょうか. 単純に x の多項式と見れば, $x^2 y^2 z$ の方が大きく, 次に Z の多項式と見れば, $x y^2 z^3$ の方が大きくなります. ところが ‘$x = y = z$’ として x の多項式として変形すると, $x y^2 z^3$ の方が次数が大きくなります. この場合は次数の大きさも含めて考えた方が良さそうです. このように多項式であれば, 項の順序には色々な考え方があります. 次に代表的な項の順序について説明しましょう.

4.8.1 色々な順序

項順序として代表的なものに「**辞書式順序**」, 「**斉次辞書式順序**」, 「**逆辞書式順序**」, 「逆斉次辞書式順序」, そして, これらの順序に変数の重みを加味したものといろいろあります. ここでは基本的な辞書式順序, 斉次辞書式順序, 逆辞書式順序と斉次逆辞書式順序について簡単に説明しましょう. なお, 変数の並び順を $x_1, \ldots x_n$ とし, $x_1^{a_1}, \ldots x_n^{a_n}$ の次数リストを $a = (a_1, \ldots, a_n)$ $x_1^{b_1}, \ldots x_n^{b_n}$ の次数リストを $b = (b_1, \ldots, b_n)$ とします. さらに, $|a| = a_1 + \cdots + a_n$ で次数リスト a に含まれる次数の総和を表記します. また, 順序の定義に含まれる “$>$” は通常の整数環 $\mathbb{Z}$ の元に対する大小関係とします.

☆辞書式順序

辞書式順序 “$>_{\text{lex}}$”

$a >_{\text{lex}} b \quad \Leftrightarrow \quad a_1 = b_1 \quad \ldots \quad a_i = b_i$ かつ $\quad a_{i+1} > b_{i+1}$

この関係で定められる順序を「**辞書式順序**」と呼びます. この順序を示す記号として “$>_{\text{lex}}$” を使います.

辞書式順序では二つの項を比較したときに左側の変数の次数が大きなものが大きな項となります. たとえば, $x^2\,y^2\,z$ と $x\,y^2\,z^3$ を各々表現する整数リスト $(2,2,1)$ と $(1,2,3)$ に対しては, 先頭の 2 と 1 を比較した段階で $2 > 1$ から $(2,2,1) >_{\mathrm{lex}} (1,2,3)$, すなわち, $x^2\,y^2\,z >_{\mathrm{lex}} x\,y^2\,z^3$ となります.

☆斉次辞書式順序

斉次辞書式順序 "$>_{\mathrm{glex}}$"

$$a >_{\mathrm{glex}} b \quad \Leftrightarrow \begin{cases} |a| > |b| \text{ または} \\ a_1 = b_1, \ldots, a_i = b_i, a_{i+1} > b_{i+1} \end{cases}$$

この関係で定められる順序を「斉次辞書式順序」と呼びます. この順序を示す記号として "$>_{\mathrm{glex}}$" を使います.
この斉次辞書式順序の場合, 総次数で最初に項を比較し, 総次数が一致すれば, 今度は項を辞書式順序で比較する二段式となっています.
たとえば, $x^2\,y^2\,z$ と $x\,y^2\,z^3$ を各々表現する整数リスト $(2,2,1)$ と $(1,2,3)$ に対し, '$|x^2y^2z| = 5$' かつ '$|xy^2z^3| = 6$' となるので '$(1,2,3) >_{\mathrm{glex}} (2,2,1)$', すなわち, '$x\,y^2\,z^3 >_{\mathrm{glex}} x^2\,y^2\,z$' となり, 辞書式順序とは逆の結果になります.

☆逆辞書式順序

逆辞書式順序 "$>_{\mathrm{revlex}}$"

$$a >_{\mathrm{revlex}} b \quad \Leftrightarrow \quad a_n = b_n \ldots a_i = b_i \quad \text{かつ} \quad a_{i-1} < b_{i-1}$$

この関係で定められる順序を「**逆辞書式順序**」と呼びます. この順序を示す記号として "$>_{\mathrm{revlex}}$" を使います. 辞書式との違いは調べる方向が逆で, その上, 次数の小さい方を取る点で逆になっています.
具体的には, 二つの項 $x^3y^2z^3$ と xy^2z^3 が与えられた場合、各々の項を表現する整数列 $(3,2,3)$ と $(1,2,3)$ が得られます. 逆辞書式順序では, この列の後から比較を行います. すると, うしろから先に移動して, リストの先頭の 3 と 1 に到達すると定義から '$(1,2,3) >_{\mathrm{revlex}} (3,2,3)$', すなわち, '$xy^2z^3 >_{\mathrm{revlex}} x^3y^2z^3$' を得ます.

☆斉次逆辞書式順序

斉次逆辞書式順序 "$>_{\mathrm{grevlex}}$"

$$a >_{\mathrm{grevlex}} b \quad \Leftrightarrow \quad \begin{cases} |a| > |b| \text{ または} \\ a_n = b_n, \ldots, a_n = b_n, a_{i-1} < b_{i-1} \end{cases}$$

この関係で定められる順序を「**斉次逆辞書式順序**」と呼びます. この順序を示す記号として “$>_{\text{grevlex}}$” を使います.
この順序は項の総次数で最初に項を比較して 総次数が等しければ逆辞書式順序で項の順序を決める二段方式のものです.
x^2y^2z と xy^2z^3 の場合, 総次数がそれぞれ 5 と 6 になるので, ‘$xy^2z^3 >_{\text{grevlex}} x^2y^2z$’ となります. ちなみに, 逆辞書式の場合, 次数リストの右端の z の次数から見るので, ‘$x^2y^2z >_{\text{revlex}} xy^2z^3$’ となって斉次逆辞書式順序と逆の結果になります.

☆順序の重要性

今迄見てきたように, 項順序は多項式の計算で非常に重要です. 順序を入れることで式の表記が一意に定まるからですが, 実用の上でも大きな意味を持ちます. たとえば, Gröbner 基底の計算では項順序を用いた計算を行います. この Gröbner 基底の計算は, 選択した項順序に対して基底が一意に決まりますが, この順序を替えると普通は別の基底が得られます. つまり, 順序毎に異なった Gröbner 基底が得られることになるのです. その結果, Gröbner 基底を用いた処理の能率も大きく異なることが生じるのです.

また, 代数学の定理の中には対象に順序を入れることで証明が容易にできるものが数多くあり, その意味でも目的に応じて用いる順序を考えることは非常に重要です. そのために新しい数式処理システムでは必要に応じてさまざまな順序が扱えるようになっています.

4.9 多項式の表現

ここでは多項式の表現について再度, 考えてみましょう.

ここまでに, 変数に順序を入れ, その順序で並べた項に対しても順序を入れました. これで項を順番に並べてしまえば, 与えられた数式の表記に対応する前置表現で式が一意に決定します. ただし, ここでの前置表現は与えられた式そのものを単純に置換えるだけで, 関係 “~” で同値な式が全て同じ前置表現で置換えられる訳ではありません. そのため, 多項式は予め展開し, 項毎に簡易化を行っておく必要があります. この簡易化を行えば, 少なくとも式の変形操作による同値関係 “~” に関しては前置表現による式と本来の式が一対一に対応します.

ところで, この内挿表現を計算機で利用することを考えると, いささか冗長な表現です. そこでよりコンパクトな「**正準表現**」を用いる方法があります. この正準表現を解説するために, 多項式 $3x^2-1$ を使って正準表現を構成を説明しましょう.

まず, 与式の変数が何であるかが重要ですね. そこで, 内挿表現から前置表現に変換するときと同様に, その先頭に変数 x を置いて, $(x, 3x^2-1)$ と表記しましょう. ここで与式 $3x^2-1$ は $3x^2+(-1)x^0$ と同値です. つまり, この式は x の多項式としては 2 次の項と 0 次の項があり, 2 次の項の係数は 3, 0 次の項の係数は -1 です. そこで, 変数 x を先頭に置いたリスト $(x, 3x^2+(-1)x^0)$ で与式を表現し

ます. 次に, 各項を変数, 次数, 係数のリストで置き換えてみましょう. すると, 最終的には表記 '(x, (x, 2, 3), (x, 0, -1))' を得ます.

ここで表記 '(x, (x, 2, 3), (x, 0, -1))' をよくよく眺めてみましょう. まず, 項のリストの変数 x は全体のリストの第 1 成分と一致するので, ここでは不要です. つまり '(x (2 3) (0 -1))' でも十分なのです. この与式は前置表現であれば '(+ (* 3 (^ x 2)) -1)' となるので, 随分とすっきりとした表現になることが判りますね. さらに, 内部の括弧は冪と係数の対で表現されると決まっているので, 平坦なリスト '(x 2 3 0 -1)' でも十分です. 文字数を数えても, 前置表現で 7 個, 新方式で 5 個と小さくなっており, 新方式の表現 '(x 2 3 0 -1)' がコンパクトなことが判ります. そして, この新表現は元の多項式と, その構成方法から判るように一対一に対応します. このように式と表現の関係が一対一になる表現を「**正準表現**」と呼びます.

この方法を単変数多項式に適用すれば次の正準表現が得られます:

単変数多項式の正準表現

(⟨ 変数 ⟩⟨ 次数$_1$⟩⟨ 係数$_1$⟩⟨ 次数$_2$⟩⟨ 係数$_2$⟩ ...)

これで一変数の場合は片付きました. では一般の多変数多項式はどうでしょうか？実際, 多変数多項式 $K[x_1, \ldots, x_n]$ に対しても同様の手法で正準表現が構成できます. ただし, 多変数多項式の場合に重要なことは項に順序を入れることです. そこで, 辞書順序 "$>_{\mathrm{lex}}$" を多項式環 $K[x_1, \ldots, x_n]$ に入れてみましょう.

まず, 変数の並びを $x_1, \ldots, x_n$ で固定し, 項 $x_1^{\alpha_1} \ldots x_n^{\alpha_n}$ を次数リスト $(\alpha_1, \ldots, \alpha_n)$ で表現します. 二つの項 $x_1^{\alpha_1} \ldots x_n^{\alpha_n}$ と $x_1^{\beta_1} \ldots x_n^{\beta_n}$ の比較は次数リストで行います. この際, リストの左端から順番に比較しますが, '$\alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}$' で '$\alpha_i > \beta_i$' の場合, '$x_1^{\alpha_1} \ldots x_n^{\alpha_n} >_{\mathrm{lex}} x_1^{\beta_1} \ldots x_n^{\beta_n}$' となります.

次に, 多項式 $\sum a_{1_{i_1}, \ldots, n_{i_n}} x_1^{i_1} \ldots x_n^{i_n}$ が与えられると最初に x_1 の多項式と看做し, 1 変数の場合と同様の考え方で正準表現を構成します. まず, 式に含まれる変数が x_1 だけの場合, 1 変数の方法で正準表現が得られます. もし, x_1 以外の変数が存在するときは最初に, x_1 を変数とする多項式と看做して式を纏めます. すると x_1 の各項の係数は高々$n-1$ 変数の多項式 $(\in K[x_1, \ldots, x_{n-1}])$ となります. そこで今度は各係数に対して x_2 の多項式表現を構成します. 以降, 係数に対して帰納的に処理を行うと次の多項式の正準表現が得られます:

多変数多項式の正準表現

(⟨ 変数 ⟩⟨ 次数$_1$⟩⟨ 係数多項式の正準表現$_1$⟩⟨ 次数$_2$⟩⟨ 係数多項式の正準表現$_2$⟩ ...)

今度は具体的に多項式環 $\mathbb{Z}[x, y]$ の元 $yx + 2xy^3 - 3$ に対し, 順序を辞書式順序 "$>_{\mathrm{lex}}$" として考えてみましょう. なお, 変数の並びは x, y とします. それから最初に与式を x の多項式と看做して変数 x で式を纏めれば $(2y^3 + y)x - 3$, したがって, 第 1 段として, '(x 1 $2y^3 + y$ 0 -3)' を得ます. それから係数の処理に移り, リストの係数各成分の処理を行います. そこで, 係数を x の次の変数 y の多項式として書き直します. すると, 第 2 成分の $2y^3 + y$ が '(y 3 2 1 1)' で置換えられるので最終的に '(x 1 (y 3 2 1 1) 0 -3)' が $yx + 2xy^3 - 3$ の正準表現として得られます.

このように順序を決めていれば正準表現が得られますが, 変数の並びや順序を変更することで同じ多項式でも表現が異なります.

4.10 Gröbner 基底の紹介

Gröbner 基底/標準基底の考え方そのものは広中先生の「**標準基底 (standard basis)**」として知られていました. この Gröbner 基底を計算するアルゴリズムは Buchberger [9]によるものです. そして, この計算手法は, 多項式に対して Euclid の互除法を適用して生成元の最大公約数を計算するユークリッドの互除法に似たアルゴリズムとなっています.

まず, 体係数の多項式環 $K[x_1, \ldots, x_n]$ の任意のイデアルは有限生成となることが知られています (Hilbert の基底定理). そのために体係数の多項式環の任意のイデアルの生成元は必ず有限個となり, 特に 1 変数多項式環 $K[x]$ の任意のイデアルは単項イデアルとなるために環 $K[x]$ は単項イデアル整域 (PID) となります. このとき, イデアルの生成元は, そのイデアルに含まれる最も低い次数の多項式となります. 実際, 写像 $\deg : K[x] \to \mathbb{N}$ を多項式の最高次数を返す函数とするとき, この函数から得られる自然数 $\mathbb{N}$ の部分集合 $\deg(K[x])$ は下に有界な空でない集合になります. そして, 自然数 $\mathbb{N}$ は, その任意の部分集合が必ず最小値を含むという性質を持つために写像 deg によるイデアルの像は必ず最小値を持ちます. ここで, 最小値を与える多項式が f と g の二つが存在したとします. このとき f と g を簡単のために最大次数の項の係数を 1 とすると, $f - g$ はより小さな次数の多項式となりますが, この $f - g$ が 0 でなければ f や g よりも小さな次数の元がイデアルに存在することとなって矛盾が生じます. したがって, 最低次数の多項式はその最大次数の項の係数を 1 とするとただ一つ定まります. そこで今度は h をイデアルに含まれる元とします. すると, h は f で割切れなければなりません. もしも割り切れないのであれば '$h = fk + r$' を充す 多項式 f よりも次数が低い多項式 r が存在することとなって, 多項式 f がイデアルの最小次数の元であることに反するからです.

このように体 K を係数とする 1 変数多項式環 $K[x]$ が単項イデアル整域となることが分りましたが, 多変数多項式環となると事情が異なります. 多変数の多項式環ではイデアルを生成する多項式系が一つであるとは限りません. たとえば, 連立一次方程式 $[2x + y = 4, x - 3y = -5, 3x - 2y = -1]$ を考えてみましょう. この方程式の解は計算してみれば容易に判ることですが $[x = 1, y = 2]$ となります. ここで頭を切り替えて, 三つの多項式 $p_1 = 2x + y - 4$, $p_2 = x - 3y + 5$, $3x - 2y + 1$ で生成されるイデアル Eq を考えましょう. ここで連立方程式の Gauß の消去法を思い出して下さい. ここでは p_2 の変数 x の係数が 1 なので他の多項式の x の項を消去しましょう. まず, p_1 に対しては $p_1 - 2p_2$ より $7y - 14$ が得られ, さらに係数環が体なので全体を 7 で割った $y - 2$ はイデアル Eq に含まれます. p_3 に対しては $p_3 - p_1 - p_2 = 0$ となるので $p_3 = p_1 + p_2$ であることが判ります. それから最後に p_2 から変数 y の除去を行いますが, これは $p_2 + 3(y - 2)$ を計算すればよく, 直ちに $x - 1$ が得られます. そして, この式もイデアル Eq の元になります. 次に $Eq_2 = \langle x - 1, y - 2 \rangle$ でイデアル Eq_2 を

[9]Gröbner は Buchberger に指導教官の名前です.

定義すると, Eq_2 の生成元は多項式 p_1, p_2 の線形和から生成されるので '$Eq_2 \subset Eq$', また p_1 と p_2 はイデアル Eq_2 の元の線形結合で記述されるので '$Eq_2 \supset Eq$' も充し, '$Eq = Eq_2$' となります. さて, Eq は三つの多項式で生成されるイデアルとして定めましたが, 結局, Gaußの消去法によって得られた二つの $x-1$ と $y-2$ の多項式が生成元となります. ここで $x-1$ と $y-2$ は一次独立となるので, 一つの多項式でイデアル Eq が生成されるということにはなりません. このように多変数多項式環ではイデアルの生成元が一つでになるとは限りません.

さて, この例では単純な連立一次方程式の求解をイデアルの生成元を求める話に置き換えました. このように連立一次方程式の求解とは, 方程式を構成する多項式を整理して無駄な多項式を削除し, 本当に必要な個数の, より計算のし易い多項式で置き換える処理という側面があるのです. そして, この処理では変数を順番に消去する操作を行っています. つまり, 最初に変数 x を他の式から消去し, それから変数 y の整理を行っていますが, 3 変数以上の連立方程式であれば, $x, y, z \ldots$ と変数を消去するでしょう.
では, どのようなイデアルの生成元がより計算し易いと言えるでしょうか? 要するに, $(x_1 - a_1, \ldots, x_n - a_n)$ といった多項式を最終的に得たいのですが, これをより単純に考えてしまえば, 新しい生成元は前の生成元と比べて最高次の次数が低い項を持つ式で構成されて, 生成元が他の生成元の線形結合で生成されないもの, すなわち, 線形独立であれば良いでしょう.

そこで多項式の項の次数に着目して多項式を見直し, イデアルの生成元を整理する必要がでてきます. この際に重要なのが「**筆頭項 (Leading Term)**」と「**筆頭単項式**」しょう. これは多項式環の順序による多項式 f で最大の次数を持つ項で, LT(f) (Leading Term) と表記します. また, 多項式 f の筆頭項が構成する単項式を LM(f) と表記し, 「**筆頭単項式**」(Leading Monomial) と呼びます. さて, 多項式環の部分集合 S に対して LM(S) を S に属する多項式 f の LM(f) から生成されるイデアルとして定義します.

準備ができたので Gröbner 基底の定義を述べることとしましょう:

Gröbner 基底の定義

多項式環 $R[x_1, \ldots, x_n]$ のイデアル I に対し, イデアル I に含まれる多項式の集合 $S = \{g_1, \ldots, g_m\}$ が次の条件を充すときに集合 S をイデアル I の Gröbner 基底と呼ぶ:

- $I = \langle g_1, \ldots, g_m \rangle$ ($g_1, \ldots, g_m$ が I を生成)
- LM(I) = LM(S)

この定義からも判るように Gröbner 基底は多項式環に入れた順序に依存するために順序が異なれば, イデアルを生成する多項式系も異なる性質を持っています.

4.11 代数的数と超越数

さて, 複素数を多項式環 $\mathbb{R}/\langle x^2+1\rangle$ で表現する話で, 純虚数は「**代数的整数**」と呼ばれる集合に属すると説明しました. 復習をすると, 代数的整数は最高次数項の係数が 1 となる整数係数多項式 (このような多項式を「**monic な多項式**」とも呼びます) の解となる数です. たとえば, 任意の整数 n は方程式 '$x-n=0$' の解となるので代数的整数になります. さらに '$x^2-2=0$' の解となる $\sqrt{2}$ も代数的整数になります. このように代数的整数は整数の集合 $\mathbb{Z}$ を含む集合ですが, 純虚数も含むので複素数 $\mathbb{C}$ の部分集合になります.

この代数的整数に加え, 最高次数項の係数が 1 とは限らない整数係数多項式の解となる数を「**代数的数**」と呼び, 整数係数多項式を「**代数方程式**」と呼びます. たとえば, 有理数は代数的整数になります. 実際, 有理数 n/m は代数方程式 $mx-n=0$ が対応するからです. また, 方程式 $2x^2+1=0$ の解 $i/\sqrt{2}$ は対応する多項式の筆頭項の係数が 1 ではなく 2 なので, 代数的整数ではなく, 代数的数になります.

ここで a を代数的数とすると, 定義から, この a を解に持つ代数方程式が存在します. ここで代表を一つ選んでおくと後々便利です. ここでの代表は a を解に持つ整係数多項式から多項式の次数が最小のものとします. この代数的数 a を解に持つ最小次数の整数係数多項式を, 代数的数 a の「**最小多項式**」と呼びます. たとえば, 純虚数 i は方程式 '$x^4+x^2=0$' や '$x^2+1=0$' の解ですが, 純虚数 i の最小多項式は最小次数の多項式になるので, x^2+1 になります. この選び方の理由ですが, 同じ根を持つ多項式は, その中の最小次数の多項式で割切れる性質があるからです. 実際, 多項式 x^4+x^2 は多項式 x^2+1 で割切れますね. そして, x^2+1 は整数係数多項式環 $\mathbb{Z}[x]$ では他の多項式で割切れない「素」の多項式でもあります. このように, 最小多項式は整数の素数に対応しています. また, 多項式環 $\mathbb{Z}[x]$ の多項式 x^4+x^2 で生成されるイデアルと x^2+1 で生成されるイデアルを考えてみましょう. すると, x^4+x^2 で生成されるイデアルは x^2+1 で生成されるイデアルに包含されます. つまり, 最小多項式を考えることは, その代数的数を解に持つ多項式で生成されるイデアルで最大のものを考えることに対応します.

さて, この代数的数の生成から新しい環が構成できます. この新しい環は 1 変数の多項式環に対し, 最小多項式を 0 と同値とする同値関係を入れることで得られます. たとえば, 整数係数の多項式環 $\mathbb{Z}[x]$ に対し, 最小多項式 x^2+1 を 0 とする同値関係を入れると, この環は複素数の部分環で, その元は二つの整数 a,b を使って $a+bi$ の形で表現されます. この数は「**Gauß整数**」と呼ばれます. この環を G と表記しましょう. それから, G を係数とする多項式環 $G[y]$ を考え, y^2-2 を 0 と看做す同値関係を入れましょう. このことによって, 新しく得られた環 H, すなわち, $G[y]/(y^2-2)$ は Gauß 整数に $\sqrt{2}$ を追加した環になります. このように環を続々と生成することで, 「新しい数」を追加して行けるのです.

この調子で行けば, 有理数として表現できない実数, すなわち, 無理数も全て包含しそうに思えますが, 残念なことに多項式の解として表現できない実数が存在します. この代数的数にならない数を「**超越数**」と呼びます. 超越数で有名なものに「円周率 $\pi=3.14\ldots$」や「Napia 数 $e=2.71\ldots$」と

いった数があります. そして, これらの超越数を計算するためには級数の計算が必要になりますが, 結局のところ, 近似計算しかできません.
このように, 実数は整数と整数環から出発することのできる代数的数, そして, 一筋縄ではゆかない超越数から構成されているのです. 19 世紀で確実な数は自然数と有理数だけで, 無理数を疑いの目で見ていた人も居る程です[10].

整数, 有理数, 無理数, 複素数, Gauß整数や代数的数と色々出て来ました. では, これらの数を集合として見た場合, その大きさにはどのような関係があるのでしょうか？

4.12　濃度/基数

4.12.1　等数性

実数 $\mathbb{R}$ は有理数 $\mathbb{Q}$ と無理数に分類されます. ここで有理数は整数と 0 を除く自然数の対として表現され, 無理数は整数と自然数との対として表現できない数, たとえば $\sqrt{2}$ のような代数的数や円周率 π のような超越数から構成されます. そこで, 前節の素朴な疑問ですが, 整数, 代数的数, そして超越数の集合の中で, どの集合の個数が一番多いのでしょうか？ どれも沢山ありそうです！その上, これらの集合は有限ではなさそうです. 実際, 整数は 1 を加え続けることで幾らでも大きな整数が作られて限度がありませんね. このように限度がないという性質から整数が無限集合であることが判ります (§4.13 を参照). その一方で $\sqrt{2}$ に整数をかけた数は全て代数的数になるので代数的数は整数と同程度かそれ以上ありそうです. 同様に円周率 π を整数をかけたものも超越数になるので, 超越数も整数よりは沢山ありそうです. ではこれらの数の個数をどうやって比較すれば良いのでしょうか？

そこで, 手始めに有限集合同士の比較をどのよう行っているかを思い出してみましょう. この本を手に取って (あるいは枕元で) 読んでいる皆さんにとって非常に基礎的過ぎて面白くないかもしれませんが, 幼児の数のお勉強から始めてみます. ちなみに幼児に数を理解させるのは結構大変なことです. 呪文のように数を「いち, にい, さん, … 」と数えさせても, 数を理解をしている訳ではありません. どうも一種の歌 (呪文？) として覚えているようです. そこで, 幼児向けの教材で試してみることになりますが, 最近, 面白いと思った方法は図 4.3 に示すもので, 図の左側には二つの林檎を盛った皿, 右側には表があります. そして, 子供には表の白丸と林檎を対応させて表の白丸を塗り潰させることで, 子供に数を理解させようとするものです:
この図で示すように, 林檎の皿に対応する表の白丸 “○” を林檎と対応させて塗り潰すことで “●” の列ができます. すると黒丸 “●” の列の長さで林檎の多寡が把握できるだけではなく, 数字との対応がつくという方法です. 以降, この調子で林檎の他にある蜜柑やケーキを使って子供に数を理解させ, ものの多寡を判断させようとしています.

[10] 自然数, 全部創ったのは神様, その他全部は人様の創作 (Die ganzen Zahlen hat der liebe Gott gemacht, alles andrere ist Meschenwerk.) - Leopold Kronecker

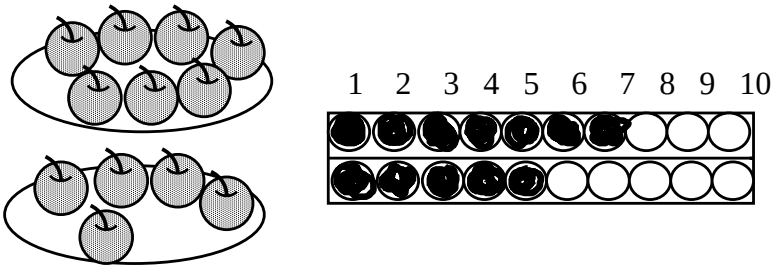


図 4.3: 林檎の個数の比較

この手法は非常に意味深いものです. まず第一に林檎でも蜜柑でも何でも対象を一度, 記号 “●” に対応させて, 対応させたあとは記号 “●” の列のみで考察しています. この操作は林檎や蜜柑を記号 “●” で代表させるという「**同値類**」の考えに繋がり, こうすることで「A の皿に盛った林檎」と「B の皿に盛った蜜柑」が共に { ●, ●, ● } となったときに「**両者の個数は同じ**」, すなわち「**等数的**」ということの認識が行えるのです. そして, 「**“●” の列 { ●, ●, ● } が基数 3 という概念に対応する**」ということを子供に理解させようとしているのです. つまり, この方法は二つの有限集合 A, B が与えられたときに, 各集合の一つ一つの元に記号 “●” を間に挟んで 1 から順番に自然数を記号 “●” に対応させ, そうして得られた自然数の列の最後尾を集合の個数としているのです. こうすることで自然数の大きさは記号 “●” の列の長さとしても直接把握できるというものです. このように “●” の列を基数に対応させるという考え方は Frege や Russell の等数性による基数の定義そのものに繋がります[11].

この方法の特徴を纏めると, 集合 A と B に対して一対一の関係がある自然数 $\mathbb{N}$ の部分集合 S_A と S_B を考えることで S_A と S_B の比較に置換えていること, 集合 A と集合 B が同じ個数であるとは「**集合 A から B への一対一の写像が存在すること**」で, そうでない場合, たとえば, 集合 A と集合 B の元を対応させて行くときに途中で集合 B の元が足りなくなったのであれば, 「**集合 A の個数が集合 B よりも個数が多い**」ことを意味します. また, このときに集合 B から集合 A への自然な単射が存在する一方で逆向きの単射が存在しないことも分ります. さらに集合 A, B に対応する自然数の集合 S_A, S_B に対しては, その集合の包含関係 “$\subset$” を使って自然数の大小関係 “$<$” が自然に導入できます.

この方法を無限集合にも適用してみましょう. そこで, 自然数 1, 2, 3,... と平方数 1, 4, 9,... を比較してみましょう. 平方数の集合は明らかに自然数全体の集合に含まれており, さらに, 数 5 は平方数の集合に含まれないので, 平方数の集合は自然数全体の集合の真部分集合になります. ところが自然数 n と平方数 n^2 には次の対応関係が存在します:

$$\begin{array}{ccccccc} 1 & 2 & 3 & \dots & n & \dots \\ \updownarrow & \updownarrow & \updownarrow & \dots & \updownarrow & \dots \\ 1 & 4 & 9 & \dots & n^2 & \dots \end{array}$$

[11] Frege に関しては §4.19 参照. 公理的な自然数の扱いに関しては §4.13 参照

このように自然数の集合から平方数の集合の一対一写像が存在することから, 自然数の集合とその真部分集合である平方数の集合は, 先程の有限個の集合で考えると個数が同じ (等数的) でなければなりません!

この自然数と平方数の比較は Galileo の「新科学対話」([14]) の中に出ている例ですが, Euclid の幾何学原論で述べられている命題「**部分は全体よりも小である**」に明らかに反することです！ ここで Galileo は「**'等しい', '大きい', '小さい' といったことは有限個のもの同士の比較でのみ言えることであって, 無限個のものには使えない**」と述べ, それ以上の言及をしていません ([14],p.59-61). このように無限は一種の魔境で,「**無限大 (∞)**」は数ではなく,「**制約を越えようとする性質**」として見做していたのです. この無限により深く入った人が 19 世紀末から 20 世紀初頭に活躍した Cantor です.

4.12.2　可附番集合

Cantor がはじめた集合論では, 自然数の集合 $\mathbb{N}$ との間に一対一写像が定義できる集合を「**可附番集合**」, あるいは「**可算無限集合**」と呼びます. 先程の自然数と平方数の例では, 写像 $f : x \to x^2$ によって一対一対応となるので平方数の集合は可附番集合になります.
そして, 集合の元の個数を「**濃度**」, あるいは「**基数**」と呼びます. ここで濃度と基数の違いですが,「**濃度**」は Cantor の集合論で集合の元の個数として用いたものです. 一方の「**基数**」は論理主義の Frege や Russell が「**命題の外延 (類/クラス) と等数性を持つ命題の外延で構成される外延**」として用いていましたが, 現在は同じ意味で用います, なお, この本では無限の対象については**濃度**を, 有限個の対象や自然数と一対一対応のある対象に対しては**基数**を使うことになるでしょう.

4.12.3　自然数の濃度

さて, 有限個の集合であれば濃度は個々の自然数に対応しますが, 可算無限集合の濃度はどうなるのでしょうか？ 可算無限集合の濃度は「**可附番濃度**」,「**可算濃度**」や「**可算個**」と呼ばれ, ヘブライ文字の $\aleph$(アーレフ) を用いて $\aleph_0$(アーレフ・ゼロ) でその濃度を表記します. この $\aleph$ はヘブライ文字の第一番目の無音の文字で, 数字の 1 や牛, さらに, カバラでは空気を意味します[12].

この自然数の濃度 $\aleph_0$ と等しい濃度の集合としては, 先程の平方数の集合に加え, 整数 $\mathbb{Z}$ の集合, 偶数の集合や奇数の集合が挙げられます.
整数 $\mathbb{Z}$ と自然 数 $\mathbb{N}$ の対応は, 正整数を奇数, 負の整数を偶数, 0 を 0 に対応させれば良いのです. 具体的には, $n \in \mathbb{N}$ が 0 であれば 0, $n > 0$ ならば $2n + 1$, $n < 0$ ならば $-2n$ を対応させれば一対一対応となります. そして, 偶数と整数は $n \to 2n$, 奇数と整数は $n \to 2n + 1$ で一対一対応, このことから自然数とも一対一であることが判ります. さらに $\mathbb{N} \times \ldots \mathbb{N}$ の濃度も $\aleph_0$ になります.
すると有理数の濃度は, 有理数が $\mathbb{Z} \times (\mathbb{N} - \{0\})$ として考えられることから $\aleph_0$ になります. また, 代数的数も同様に $\aleph_0$ になります. 何故なら, 代数的数に対応する整数係数の代数方程式が有限個の整数係数で構成されたリスト, つまり, $\mathbb{N} \times \ldots \mathbb{N}$ の元で表現されるからです.

[12]「創造の書」では $\aleph$, ב (mem), ש (sheen) の三つが母なる文字で, $\aleph$ が空気, ב が水, そして, ש が火を示します ([40]).

4.12.4 実数の濃度

自然数の濃度は $\aleph_0$ と表記されますが, 実数の濃度はどうなるのでしょうか. ここでいきなり実数本体にとりかかるのは流石に恐しいので, 実数と開区間 $(0,1)$ の関係から攻めてみましょう. 実は, 次に示す写像 f によって区間 $(0,1)$ から $\mathbb{R}$ は一対一対応となるので両者の濃度は一致します:

$$f(x) = \log x - \log(1-x) \qquad x \in (0,1)$$

この $(0,1)$ と $\mathbb{R}$ のように全体と等しい濃度を持つ真部分集合が存在する集合のことを「**Dedekind 無限集合**」[13]と呼びます.

それでは開区間 $(0,1)$ と自然数 $\mathbb{N}$ の関係はどのようになるでしょうか. 実は開区間 $(0,1)$ の方が自然数の集合 $\mathbb{N}$ よりも圧倒的に大きな集合なのです. このことは「**Cantor の対角線論法**」と呼ばれる非常に有名な手法を用いて示すことができます:

対角線論法を用いた証明: 開区間 $(0,1)$ が可符番であったと仮定します. すると, 次の表に示すように左側に番号, 右に番号に対応する開区間 $(0,1)$ の数が並べられます:

$$\begin{array}{ll} 1 & 0.x_{11}x_{12}x_{13}\ldots x_{1n}\ldots \\ 2 & 0.x_{21}x_{22}x_{23}\ldots x_{2n}\ldots \\ \vdots & \qquad\ddots \\ n & 0.x_{n1}x_{n2}x_{n3}\ldots x_{nn}\ldots \\ \vdots & \qquad\ddots \end{array}$$

そして, この表から新しい数列 $0.y_1y_2\ldots$ を構築します. この構成方法は小数点下 m 桁の数 y_m を m 番目の数の小数点下第 m 桁と異なるように設定します. したがって, 新しく生成した数の小数点下 k 番目の成分は表の k 番目の対角成分と異なる数になります. すると, この数は表には現われません. 何故なら, この数が表の k 番目に出るとすれば, 小数点下第 k 桁目の数 y_k は定義の方法からこの表の x_{kk} と異なっていなければならないからです. しかし, $(0,1)$ が可符番ならば, $(0,1)$ に含まれる数は先程の表に含まれなければなりません. この矛盾が得られたことから開区間 $(0,1)$ は可算ではなく, 自然数よりも大きな濃度を持つ集合であることが分ります. したがって, 開区間 $(0,1)$ と同じ濃度になる実数 $\mathbb{R}$ の濃度を 「$\aleph$」 と記述します. なお, 集合 $\mathbb{R}^n$ の濃度も $\aleph$ になることが知られています.

ここで任意の 1 以上の自然数 n に対して, $\mathbb{R}^n$ の濃度が $\aleph$ になるということは, 次元の考え方が無意味なもののように思えます. しかし, この写像は「**連続性**」という性質を持ちません. 逆に言えば, 写像を「連続写像」に限定すると次元は意味を持つことになります. なお, このような写像による像の次元は「**フラクタル次元**」というものがあります. この次元は通常の整数値の次元ではなく実数値の次元を持ちます. たとえば平面を埋め尽す曲線として, Peano 曲線が挙げられますが, この Peano 曲線のフラクタル次元は 1 と 2 の中間の実数値になります.

[13]「数とは何か, 何であるべきか」([36] の §64(p.80-81)) に本来の定義があります.

4.13　自然数

4.13.1　自然数の基礎付けについて

では, 今迄, 何気なく使っている自然数 $\mathbb{N}$ はどのような数なのでしょうか？とにかく, 自然数は代数的整数の一部ですねぇ…. そして, この自然数を料理することで, いろいろな数ができるだけではなく, 可附番集合の定義では自然数の集合 $\mathbb{N}$ との一対一対応となる写像が存在するとか, 何かと物事の基礎にある数です. その割には 1, 2, 3… と無数に続く数として漠然としか考えておらず, 既知のものとして安心して使っている数ですが, この状況は 19 世紀末でも同様で, 悪く言えば非常に曖昧な存在であったのにも関わらず, その上に壮麗な数学が構築されているという砂上の楼閣の如き危い状況だったのです.

この自然数の厳密な定義は 19 世紀末から本格的に進められますが, それ以前に, Leibniz も自然数の基礎付けを行っているそうです. ここでは, 19 世紀末の Frege の著作「算術の基礎」([49]) に記載されている, Leibniz による ‘$2+2=4$’ の証明の補完を紹介しておきましょう.

4.13.2　Leibniz による $2+2=4$ の証明

Leibniz は自然数 2, 3 と 4 を次で定義しています:

Leibniz による自然数 2, 3, 4 の定義

- 2 を $1+1$ と定義する
- 3 を $2+1$ と定義する
- 4 を $3+1$ と定義する

それから公理として「**同じもので置き換えを行っても式は同じである**」と述べて, それから ‘$2+2=4$’ の証明を行っていますが, この証明の概要は, 式 $2+2$ の演算子 “+” の右辺の 2 は上の自然数 2 の定義から $1+1$ になります. したがって, ‘$2+2=2+1+1$’ が得られます. すると自然数 3 の定義から ‘$2+1+1=3+1$’ となるので, 自然数 4 の定義から 4 が得られるというものです.

さて, この証明は正しいでしょうか？ まず, 群の定義を思い出して下さい．群の定義には結合律: ‘$(a+b)+c=a+(b+c)$’ がありました. Leibniz の証明では, ‘$2+2=2+1+1$’ だから ‘$2+1+1=3+1$’ になると主張していますが、これらの式が成立することを主張するためには演算 “+’ が結合律を充すことが必要です．そのことを念頭に置いて Frege による修正を解説しましょう．与式 $2+2$ の右側の 2 に対し, 公理の「**同じもので置き換えを行っても式は同じである**」から, 2 を定義する $1+1$ で 与式を置換えることで ‘$2+2=2+(1+1)$’ が得られます．それから, 結合律によって ‘$2+(1+1)=(2+1)+1$’ を得ます, ここで, ふたたび, 公理の「**同じもので置き換えを行っても式は同じである**」を使って, $2+1$ で定義される 3 で置換えることで ‘$(2+1)+1=3+1$’ が得られ, 最後に 4 の定義から $2+2=4$ が得られるのです.

さて, Leibniz は $2+2$ を素直に計算せずに, しかも $2+2$ を $(2+1)+1$ と置換えていますね. この理由は何でしょうか？この方法を見ると, $(m+1)+1$ という式が見えます. つまり, Leibniz は自然数を $n+1$ で与えられる数として $n=1$ から「帰納的」[14]に定義しようとしたことが見えてきます

4.13.3 自然数の後者

ここで自然数を 0, 1, 2, 3, ... と左から右へと大きくなるように並べた状態を考えて下さい. このときに自然数 n に対して自然数 $n+1$ は自然数 n の直後 (右側) に並びます. この n に続く自然数 $n+1$ を「**自然数 n の後者 (successor)**」と呼ばれ, 記号 $s(n)$ で表記します.

では Leibniz の '$2+2=4$' の証明に「後者」を入れて見直してみましょう. $2+2$ の右側の 2 は 1 の後者 $s(1)$ なので, $1+1$ で置換えられます. したがって, '$2+2=2+(1+1)$' となります. それから, 加法 "+" の持つ結合律によって $(2+1)+1$ が得られます. すると, $2+1$ は 2 の後者の 3 なので '$2+2=3+1$' が得られます. このとき $3+1$ は 3 の後者 4 なので '$2+2=4$ が証明できます. さらに $2+2$ は $s(s(2))$ なので 2 の 2 番目の後者は 4 と等しいことが分ります. この方法を一般化すると, 自然数 m と n の和 $m+n$ は, 自然数 m に対して函数 s を n 回の作用させた $\underbrace{s\circ\cdots\circ s}_{n}(m)$, すなわち, m の n 番目の後者となります.

4.13.4 Peano の公理系

この考えで Peano は「**自然数の公理**」を構築しています. この Peano の 1889 年の論文「Arithmetices principia nova methodo exposita」[15]に現在の数理論理学の源流とも言える一連の記号を用いて Peano はこの公理系を記述しています. なお, Peano の本来の公理系では自然数を 1 から開始する数とし, 現在では自然数を 0 から開始する数として公理化していますが, 両者に本質的な違いはありません.

Peano の公理系

自然数全体の集合 $\mathbb{N}$ は次の 5 条件を充す:

1. $0\in\mathbb{N}$
2. $x\in\mathbb{N}$ ならば $s(x)\in\mathbb{N}$
3. $x\in\mathbb{N}$ ならば $s(x)\neq 0$
4. $s(x)=s(y)$ ならば $x=y$
5. $0\in M$ かつ $x\in M$ かつ $s(x)\in M$ ならば $M\supset\mathbb{N}$

[14]後述の Peano の公理系の帰納法の原理に基く方法ですが, ここでは自然数を使って足場を固めながら前進する手法 (つまり, 1 を使って $1+1$ から 2 を定義, 2 を使って $2+1$ から 3 を定義..) と理解して下さい.

[15][81] に英訳 (The principles of arithmetic,presented by a new method) が収録されています.

ここで自然数を 1 から開始して 0 を含めない場合, 公理 1. と公理 2. の 1 を 0 で置換えることになります. 公理 2. は自然数 x に対して必ず後者 $s(x)$ が存在することを保証し, その一方で, 公理 3. で 0 を後者とする自然数は存在せず, 公理 4. で一対一の写像であることを保証します. そして, 最後の公理 5. が「**帰納法の原理**」と呼ばれる公理です. 実際, 自然数 n に対する命題を $P(n)$ とすれば, 数学的帰納法は以下になります:

数学的帰納法

1. 命題 $P(0)$ が真である.
2. 任意の自然数 k に対して命題 $P(k)$ が真であれば命題 $P(k+1)$ も真である.
3. 1. と 2. が証明されていれば, 全ての自然数 n に対して命題 $P(n)$ が成立する.

ここで, Peano の公理系の公理 5.「**数学的帰納法の原理**」と「**数学的帰納法**」を照合させてみましょう. まず, 公理 5. の '$0 \in M$' が「$P(0)$ が真」に, 公理 5. の '$x \in M$ かつ $x+1 \in M$' が「$P(x)$ と $P(x+1)$ が真である」に, そして公理 5. の '$M \supset \mathbb{N}$' が「全ての自然数 n に対して命題 $P(n)$ が成立する」に対応します.

この Peano の公理系では数 0 と数 1 が一体何者であるかは何も述べず, 既知のものとして公理を構築しています. この点を逆手に取ることで, Russell は「数理哲学入門」[93][94] で面白い例を挙げています.
たとえば, 0 が "100" を意味し, その後者の 1 が "101" としても Peano の公理系を充します. この場合, "99" は自然数ではありません. また, $0, 2, 4, \ldots$ の偶数の列はどうでしょうか？ この列も Peano の公理系を充します. そして, 0 を "1", その後者の 1 を "1/2", 1 の後者を "1/4", n の後者を "$1/n^2$" で定めた列 $1/2, 1/4, 1/8, 1/16, \ldots, 1/n^2, \ldots$ も Peano の公理系を充します. さらに, 0 を "x_0", 0 の後者を "x_1", n の後者を x_{n+1} とする列 $x_0, x_1, x_2, \ldots, x_n, \ldots$ も Preano の公理系を充します.

では, 数 0 と 1 は何であり, また, 何であるべきなのでしょうか？先程の Russell の "100" や "クレオパトラの針" というオベリスクかもしれません. この問題に厳密に答えようとして, より徹底して自然数の定義を行った先駆者として, 論理学から数学を導出しようとした「**論理主義**」と呼ばれる立場の Frege と Russell が挙げられます. Frege と Russell に関しては §4.19 や §4.20 で詳細を述べることとしましょう. それから, 集合論でも自然数の構築を行っています. こちらは §4.16 で解説します. この両者に共通して言えることですが, 自然数の導出はとてつもなく難しいことが実感されるでしょう.

4.13.5　原始帰納的函数

"0" と "1" を既知のものとして定義した自然数に和 "+" と積 "$*$" といった演算をどのように入れれば良いのでしょうか？ 後者を対応させる函数 s を使えば, $x+1 \stackrel{def}{=} s(x)$ と定義することで演算子 "+" を $\mathbb{N}$ に導入できます, では, より一般の自然数の和, たとえば, $3+2$ に対してはどうすれば良いでしょうか？ そこで Leibniz や Frege の方法:'$3+2 = 3+1+1 = s(3+1) = 5$' を思い出しましょう. このことから判るように後者を与える函数 s には '$a + s(b) = s(a+b)$' という (再帰的な) 性質があります. この手法で $a+b$ から 函数 s を b 回作用させることで $s \circ \cdots \circ s(a)$ に到着し, そこから

$a+b$ に対応する一つの自然数が得られることになります.
そこで和 “+” を次で定義しましょう:

和 “+” の定義

- $a+0=a$
- $a+s(b)=s(a+b)$

そして, 積 “$*$” も同様に定義できます:

積 “$*$” の定義

- $a*0=0$
- $a*s(b)=a*b+a$

ここでの定義では, 最初に a と 0 との結果を述べ, 次に a と b の後者に対しては, a と b 間の演算が入った式で定義が行われています. つまり, $a+s(b)$ の計算を知りたければ $a+b$ が判っていれば, その後者として求められ, 以降, $a+0=a$ から全てが求まるという形式になっています. ここで, 表記上, $a+s(b)=s(a+b)$ と左辺は a と $s(b)$ に対する「和」, その右辺は a と b の「和」の後者と, 右辺でも「和」について言及されています. これは積 “$*$” でも同様で, このように定義の中に, 定義すべきものの参照が行われている状態を「**再帰的**」と呼びます. と呼びます. そして, この再帰的な方法で定義することを, 「**帰納法による定義**」, あるいは「**帰納的定義**」と呼びます. そして, 帰納的定義で定義された函数達は「**原始帰納的函数**」と呼ばれる函数として纏められます.

原始帰納的函数: 次の (I), (II), (III) に示す「**始函数**」と呼ばれる函数を基に (IV) と (V) の合成操作を繰り返して適用することで構成された函数です:

始函数と合成操作

(I) $f(x)=s(x)$

(II) $f(x_1,\ldots,x_n)=q$ (q は定数)

(III) $f(x_1,\ldots,x_n)=x_i$ ($1\leq i\leq n$)

(IV) $f(x_1,\ldots,x_n)=g(h_1(x_1,\ldots,x_n),\ldots,h_m(x_1,\ldots,x_n))$

(V)
$$\begin{cases} f(0,x_2,\ldots,x_n)=g(x_2,\ldots,x_n) \\ f(s(y),x_2,\ldots,x_n)=h(y,f(y,x_2,\ldots,x_n),x_2,\ldots,x_n) \end{cases}$$

まず, (I) の函数は後者を返す函数, (II) は定数函数です. それから, (III) の函数は $1\leq i\leq n$ を充す i に対し, $x_1,\ldots,x_n$ から x_i を返す射影函数になります. そして, (IV) は既存の原始帰納的函数 $g,h_1,\ldots,h_n$ の合成で, (V) は既存の帰納的函数を使って帰納的に定義される函数であることを示しています.
ここで重要なこととして「**原始帰納的函数**」は「**計算可能性**」, つまり, 函数を計算するための手続の存在を意味します. 始函数が計算可能なことは問題がないでしょう. そして, 計算可能な始函数を (IV) や (V) の組合せて計算する原始帰納的函数が計算可能であることも問題はないでしょう.

さて, 和 “+” や積 “∗” は原始帰納的函数となることを確認してみましょう. そこで $a+b$ を $f(b,a)$ と記述します. このとき ‘$a+0=a$’ から ‘$f(0,a)=a$’ となるので (V) の函数 g として恒等写像 Id を指定すれば良いことになります. 次に, b の後者に対する定義式 ‘$a+s(b)=s(a+b)$’ は ‘$f(s(b),a)=s(f(b,a))$’ となりますが, ここで写像 h を ‘$h(x_1,x_2)=x_2$’ となる (III) の始函数とすれば, ‘$s(f(b,a)=s(h(b,f(b,a)))$, 即ち, ‘$s(f(b,a)=s\circ h(b,f(b,a))$’ を満して (V) の定義式にあてはめられるので, 和 “+” が原始帰納的函数となることが判ります.

積 “∗” に関しても同様に $k(b,a) \stackrel{def}{=} a*b$ で定義します. まず, ‘$k(0,a)=0$’ となるので写像 g に (II) の定数写像 ‘$g(x_1)=0$’ を対応させます. 次に b の後者に対する定義では, 先程の原始帰納的函数 f を用いて, ‘$h'(x_1,x_2,x_3,x_4)=f(p_4(x_1,x_2,x_3,x_4),p_3(x_1,x_2,x_3,x_4))$’ と ‘$h(x_1,x_2,x_3,x_4)=f(p_3(x_1,x_2,x_3,x_4),h'(x_1,x_2,x_3,x_4))$’ を定めます. ここで p_i は (II) の第 i 成分の射影函数です. これによって ‘$k(s(b),a)=h(b,k(b,a),a,b)$’ を得るので, 積 “∗” も原始帰納的函数になります.

4.13.6　原始帰納的函数の例

先程の和 “+” と積 “∗” を含めて, ここでは代表的な 16 個の原始帰納的函数を示しておきす. なお, 函数番号は「ゲーデルの世界」([42],p.91-94) の函数番号に合せています:

原始帰納的函数の例

1.	$a+b$	2.	$a*b$	3.	a^b
4.	$a!$	5.	$pd(a)$	6,	$a\dot{-}b$
7.	$\min(a,b)$	8.	$\min(a_1,a_2,\ldots,a_n)$	9.	$\max(a,b)$
10.	$\max(a_1,\ldots,a_n)$	11.	$\mathrm{sg}(a)$	12.	$\overline{\mathrm{sg}}(a)$
13.	$\lvert a-b\rvert$	14.	$\mathrm{rem}(a,b)$	15.	$[a/b]$
16.	$\sum_{y<z} f(x_1,\ldots,x_n,y)$ $\prod_{y<z} f(x_1,\ldots,x_n,y)$				

☆ a^b　函数 a^b は a の b による冪乗を返す函数です.

☆ $a!$　$a!$ は階乗を返す函数です. この函数は Lisp 等の再帰的な処理の行える言語の再帰的処理の例題でお馴染でしょう.

☆ $\mathrm{pd}(a)$　函数 $\mathrm{pd}(a)$ は自然数 a の後者を返す函数です. すなわち, $a\geq 1$ の場合に $a-1$ を返し, それ以外は 0 を返す函数です.

☆ $a\dot{-}b$　演算 $a\dot{-}b$ は自然数に対する減算で $a-b\geq 0$ の場合に $a-b$ を返し, それ以外は 0 を返します.

☆ sg **と**☆ $\overline{\mathrm{sg}}$　函数 $\mathrm{sg}(a)$ は $a>0$ がの場合に 1 を返し, それ以外は 0 を返す函数です. そして, 函数 $\overline{\mathrm{sg}}(a)$ は $\mathrm{sg}(a,b)$ とは逆に $a>0$ のときに 0 を返し, それ以外は 1 を返す函数です.

☆ $|a-b|$　この函数は $a-b$ の絶対値を返す函数で, $|a-b| \stackrel{def}{=} (a \dot{-} b) + (b \dot{-} a)$ で定義可能です.

☆ $\mathrm{rem}(a,b)$　函数 $\mathrm{rem}(a,b)$ は a の b による剰余, つまり, $a = bq + r$ の場合に r を返す函数です.

☆ $[a/b]$　函数 $[a/b]$ は $b = 0$ の場合には 0 を返し, それ以外は a を b で割った商, すなわち, $a = bq + r$ の場合に q を返す函数です.

☆ **16. の函数**　n+1 変項の函数 $f(x_1, \ldots, x_{n+1})$ が原始帰納的函数の場合, その和 $\sum_{y<z} f(x_1, \ldots, x_n, y) \stackrel{def}{=} f(x_1, \ldots, x_n, 0) + f(x_1, \ldots, x_n, 1) + \cdots + f(x_1, \ldots, x_n, z \dot{-} 1)$, および, その積 $\prod_{y<z} f(x_1, \ldots, x_n, y) \stackrel{def}{=} f(x_1, \ldots, x_n, 0) * f(x_1, \ldots, x_n, 1) * \cdots * f(x_1, \ldots, x_n, z \dot{-} 1)$ も原始帰納的函数になるというものです.

4.13.7 一般帰納的函数

原始帰納的函数を拡張したものに「**一般帰納的函数**」があります. この一般帰納的函数は §4.23.4 で紹介する「**λ-表記可能な函数**」と同値なことが知られています. さらに一般的帰納的函数と計算可能な函数に関して Church は次の重要な「**Church の提唱**」を行っています:

Church の提唱

計算可能な函数は一般帰納的函数とみなす.

この Church の提唱は定理ではありませんが, これに反する事例はまだ発見されていません.

4.13.8 自然数の性質

自然数の整列性

自然数の集合 $\mathbb{N}$ で, 順序 “>” を次の性質を充す関係として定めることが可能です:

順序 “>” の定義

$a > b \stackrel{def}{=}$ 「零と異なる自然数 x が存在して $a = b + x$ を充す場合」

この定義から ‘$s(a) > a$’ であること, そして, 任意の $n \in \mathbb{N} - \{0\}$ に対して ‘$n > 0$’ であることが判ります. さらに, 任意の自然数 $m, n \in \mathbb{N}$ に対して ‘$m = n$’, ‘$m > n$’, ‘$m < n$’ の何れかが成立するので, $(\mathbb{N}, >)$ は全順序集合になります. さらに, 自然数 $\mathbb{N}$ の任意の部分集合は順序 “>” に対して最小値 (下限) を持ちます. このように任意の部分集合に下限が存在するという性質を持つ集合のことを「**整列集合**」と呼び, この順序のことを「**整列順序**」と呼びます.

Euclid の互除法

二つの自然数の最大公約数を取り出す手法で広く用いられている手法に,「**Euclid の互除法**」, あるいは簡単に「**互除法**」と呼ばれる手法があります. この手法は Euclid の原論に出ている由緒ただしいもので, 自然数が整列集合となることを利用しています.
この Euclid の互除法は二つの自然数 m, n(ただし, $m > n$) に対して次の手順を踏みます:

Euclic の互除法

0. 変数 a と b に自然数 m と n をそれぞれ代入する.
1. a を b で割り, 剰余の r を計算する.
2. もし, '$r = 0$' であれば b が m と n の最大公約数として返す.
   '$r = 0$' でなければ a に b を代入し, それから b に r を代入して
   1. に戻る.

ここで, この計算の各段で得られる剰余 r は順序 ">" に対して単調な自然数の減少列となります. そして,「**自然数の整列性**」により, 剰余 r に下限 0 が存在するので有限回数で処理が終了することが保証されます.

4.14　整数

0 と 1 が何者であるかは不問として, 自然数 $\mathbb{N}$ の公理化/形式化が Peano の公理系によってできました. 次に目標になるのが整数 $\mathbb{Z}$ です. そこで整数 $\mathbb{Z}$ を代数的に構築してみましょう.
まず, 自然数の対集合 $\mathbb{N} \times \mathbb{N}$ を考え, 自然数対に次の同値関係 "~" を導入します:

$$(a, b) \sim (c, d) \Leftrightarrow a + d = b + c$$

それから, この同値関係 "~" による自然数対の集合 $\mathbb{N} \times \mathbb{N}$ の同値類を整数として定義します:

$$\mathbb{Z} \stackrel{def}{=} \mathbb{N} \times \mathbb{N} / \sim$$

この関係 "~" による $\mathbb{N} \times \mathbb{N}$ の同値類に新しい表記を導入しましょう. 最初に $(a + k, k) \in \mathbb{Z}$ を a, $(k, a + k) \in \mathbb{Z}$ を $-a$, それから $(k, k) \in \mathbb{Z}$ を 0 と表記します. これで自然数 $n \in \mathbb{N}$ に対して $n \in \mathbb{Z}$ を充し, 負の数 $-n$ も自然に整数 $\mathbb{Z}$ 内に構築できました.

また, 整数 $\mathbb{Z}$ の順序 ">" も自然数の順序 ">" から構築できます. 実際, 二つの整数 $(a, b), (c, d) \in \mathbb{Z}$ に対し, $(a, b) > (c, d) \stackrel{def}{=} a + d > b + c$ によって整数の順序関係を定義します. 実際, 自然数 a, b は $(a, 0), (b, 0)$ と表現され, $a > b$ であれば $a + 0 > b + 0$ より $(a, 0) > (b, 0)$ となるので, この定義で問題ありません. さらに負の自然数 '$-a > -b$' であることと '$a < b$' は同値なので, きちんと拡張できていることが判ります.

では, 整数 $\mathbb{Z}$ の演算はどうでしょうか？ 和 "+" は単純に $(a, b), (c, d) \in \mathbb{Z}$ に対して $(a, b) + (c, d) \stackrel{def}{=} (a + c, b + d)$ で定めます. 差 "−" は $(a, b) \in \mathbb{Z}$ に対し $a - b$ で定義します. すると, 先程の和

“+” の定義から ‘$(a,b) = (a,0) + (0,b)$’ なので, ‘$a - b = a + (-b)$’ も得られます. さらに $-(-a)$ は ‘$-(-a) = -(k_1, a + k_1) = (a + k_1 + k_2, k_1 + k_2)$’ を充すために a と等しくなります. 最後に積 “$*$” は $(a,b) * (c,d) \stackrel{def}{=} (ac + bd, ad + bc)$ で定めます. このように整数 $\mathbb{Z}$ には演算 “+”,“−”,“$*$” が導入可能で, これらの演算に対して閉じています.

整数の大きな特徴として, 次に説明する最小性を持つことが挙げられます. この性質は自然数 $\mathbb{N}$ を含む集合 A で, 恒等写像と異なる一対一写像 $\phi : A \to A$ が存在する集合とし, このような集合 A で構成される集合 $\mathfrak{A}$ を考えるとき, 整数の集合 $\mathbb{Z}$ は $\mathfrak{A}$ の成分に対する包含関係 “$\subset$” で最小の集合になります. この写像 ϕ の具体例として, $\phi : (a,b) \mapsto (b,a)$, すなわち, $a \mapsto -a$ や後者を与える写像 $s : a \mapsto a + 1$ があります.
ここで自然数 a に対応する負の数 $-a$ は $(0,a)$ で表現されますね. このようにして負の数が自然数を用いて表現できています. De Morgan の法則で有名な 19 世紀の数学者 De Morgan は負の数の存在を認めなかったそうですが, このように定義することで $-a$ という数も具体的に表現することができます. 勿論, 負の数の別の表現として多項式環 $\mathbb{Z}[x]$ に対して ‘$x + a = 0$’ による剰余環を考える方法もありますね. こちらは Kronecker の構成方法になります.

さて, ここでの整数の構成法で面白い点は自然数の対 (配列) だけで同値類として負の数を表現していることです. そして, これらのことから自然数さえ扱える計算機があれば, 整数や有理数は配列を使えば表現できるということも判ります.

4.15 実数

自然数, 整数, 有理数と来れば, 残りに実数があります. ここでは実数について一寸, 歴史を辿ってみましょう.

4.15.1 古代ギリシャ

古代ギリシャでは基本的に数とは零以外の自然数 (基数) で, 有理数は自然数で表現された量の比として現われ, 無理数は比で表現できない量として認識されています. ちなみに Frege は基数以外の数を量として捉えおり, その点では一種の先祖帰りといえるでしょう.

この自然数の比で表現できるかどうかということは, ある単位となる長さの辺を使って整数倍で表現できること, すなわち, 「**計測可能な量**」であることを意味します. たとえば, 辺の長さが 1 の正方形の対角線の長さとして現われる $\sqrt{2}$ のように正方形の辺で計ることができない量は「**非共測 (alogos=$\alpha\lambda o\gamma o\varsigma$)**」, すなわち, 通約できないと呼ばれます[16]. ちなみに非共測の語源 “$\alpha\lambda o\gamma o\varsigma$” は否定を意味する “$\alpha$” を比や割合を意味する “$\lambda o\gamma o\varsigma$” につけたものです. ところが “$\lambda o\gamma o\varsigma$” を所謂「**ロゴス**」の意味と捉えたところから, 日本語の「**無理数**」の「**無理 (ir+ration)**」が生じています.

[16] 非共測の意義については [24] を参照

さて, 古代ギリシャの数学で忘れてはならないのが Pythagoras 学派です. Pythagoras 本人は直角三角形の「**Pythagoras の定理 (三平方の定理)**」でその名を残していますが[17], この Pythagoras には合理的な話よりも神秘的な話[18]が多く, 「Pythagoras の徒」(Pythagoraioi) と呼ばれる一種の秘教的集団もあって, 正五角形から得られる五芒星をその象徴としていたと言われています[19]. この五芒星はそののちも永く魔除としてヨーロッパで用いられています. たとえば Göthe の「Faust」では Faust が床に描いた五芒星の先が欠けていたために Mephistopheles が Faust の部屋に入り込めていますが, その魔除としての機能が不完全ながらもあるために出られなくなったり, 民衆本 (人形劇) の Faust では, Faust が五芒星の外に出たために Mephistopheles に弱味を握られるというあんばいです. そして, この五芒星は日本でも陰陽師の安倍晴明との関連で有名ですね.

この「Pythagoras の徒」は Aristotle の「形而上学」によると, 「**数の構成要素を全ての存在の構成要素で X あると判断し, 天界全体をも音階 (調和) であり数であると考えた**」とあります. こので Aristotle が述べている数は勿論, 零を除く自然数です. Pythagoras 学派が古代ギリシャの初期の数学を独占していたと考えられていたこともあって, 無理数の発見に関する Pythagoras 学派の Hippasus of Metapontum の話が何かと有名です. この広く流布されている Hippasus の物語では, Pythagoras 学派の象徴である正五角形の辺と対角線の比の研究から無理数を発見しますが, この数の存在は Pythagoras 学派の基本理念と矛盾するために学派に大きな衝撃を与えます. その上, 無理数の存在を外部に漏らしたため, Hippasus は溺死させられたというものです. この説は von Fritz の 1945 年の論文によるものですが, 現在, このスキャンダル説は否定されています [24]. とは言え, その面白さから, Hippasus の再構成と呼ばれる構成法について解説しましょう.

von Fritz による再構成:　この手法は, 無理数が Pythagoras 学派の象徴である五芒星から黄金数として得られることに加え, Hippasus の悲劇的な運命という与太話のために広く紹介されているのでしょう ([10] 等).
まず, 図 4.4 に示す正五角形を考えます:

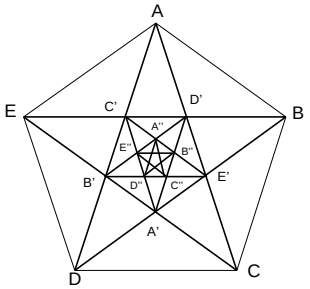


図 4.4: von Fritz による再構成

[17]「ピタゴラスィッチ」, あるいは「ピタゴラ装置」は残念ながら違います.
[18]輪廻転生を信じ, 肉や豆を食べることを忌むといったことが挙げられます. Lucianos の「にわとり」([70] に収録) には Pythagoras の生まれ変わりの「にわとり」が出て来てますが, この「にわとり」によると肉と豆のことは人を驚かせて畏れ敬われるためだそうです. 真偽は何如に.
[19]実はこれも明確なことではないようです.

ここで対角線 AC, AD, BD, BE と CE によって正五角形 $ABCDE$ 内部に相似な正五角形 $A'B'C'D'E'$ が構成されます. ここで辺 AE と対角線 AD の比を考えましょう. 三角形 ADE と三角形 $BE'C$ は対角線 AD と辺 BC, 対角線 BD と辺 AE, 対角線 AC と辺 ED が平行なので相似形になります. したがって, $AD : AE = BC : BE'$ を満します. ここで AE と BC は辺なので $BC = AE$ となります. 辺 AE と対角線 BD, 辺 ED と対角線 AC が平行のために四辺形 $AE'DE$ は平行四辺形になって $AE = DE'$ がえられます. したがって, $BE' = BD - DE' = AD - AE$ となるので $AD : AE = AE : AD - AE$ を得ます. すなわち, $\boxed{\text{対角線} : \text{辺} = \text{辺} : \text{対角線} - \text{辺}}$ の関係式が正五角形で成立しますが, ここで $AE = 1$ とすると AD は方程式 $x(x-1) = 1$ の解 $x = (1+\sqrt{5})/2$ として得られます. そして, この数を黄金数と呼びます.

さて, 正五角形 $ABCDE$ の対角線 AD を a_0, 辺 AE を a_1, 線分 BE' を a_2, 線分 $A'E'$ を a_3 と置いてみましょう. すると, 上の議論から直ちに $a_0 : a_1 = a_1 : a_2, a_1 = a_2 + a_3, (a_1 > a_2 > a3)$ が得られます. ここで BE' は $B'D'$ に等しく, $A'E'$ と $B'D'$ は正五角形 $ABCDE$ と相似な小正五角形 $A'B'C'D'E'$ の辺と対角線になることが判ります. このことから, これらの対角線と辺の等式は際限なく継続できることが判ります. ここで式を次の様に変形してみましょう:

$$
\begin{aligned}
a_0 = a_1 + a_2 \quad &\Leftrightarrow \quad \frac{a_0}{a_1} = 1 + \frac{1}{\frac{a_1}{a_2}} \\
a_1 = a_2 + a_3 \quad &\Leftrightarrow \quad \frac{a_1}{a_2} = 1 + \frac{1}{\frac{a_2}{a_3}} \\
a_2 = a_3 + a_4 \quad &\Leftrightarrow \quad \frac{a_2}{a_3} = 1 + \frac{1}{\frac{a_3}{a_4}} \\
&\cdots
\end{aligned}
$$

それから上記の式を組合せると下記の連分数が得られます:

$$
\frac{a_0}{a_1} = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\cdots}}}}}
$$

ここで上の表の左辺の式 $a_n = a_{n+1} + a_{n+2}$ $(n \in \mathbb{N})$ から最初の a_0, a_1 が自然数であれば, その構成方法から任意の a_n も自然数になります. さらに構成方法から $a_{n+1} > a_{n+2}$ なので, 式 $a_n = a_{n+1} + a_{n+2}$ $(n \in \mathbb{N})$ は a_0 と a_1 の Euclid の互除法から得られる式になります. ここで a_0 と a_1 の比が整数比であれば, この処理は有限回で終了しなければなりませんが, これには際限がありませんね. このことから黄金比は有理数で表現できない数, すなわち, 無理数であることが判ります.

さて, Hippasus がこの方法で無理数を発見したかどうか, さらには Hippasus 本人が無理数の発見者であったかどうかは明確でありません. Hippasus が正五角形を使って球面を構築したこと, そして, そのようなことが不敬とされ, やがて彼が海で溺死したことが神罰であるかのように述べた断片が存在するだけです. その上, Aristotle の「形而上学」では「正方形の対角線が辺で計り得ないこ

と」についての言及が幾つかありますが, 五芒星や正五角形に関して同様の言及はありません. そのことから五芒星よりも正方形の辺と対角線の関係から発見したことの方が自然に思えます.

正方形で Euclid の互除法を用いた再構成:　正方形で Euclid の互除法を用いる再構成として図 4.5 に示す方法があります:

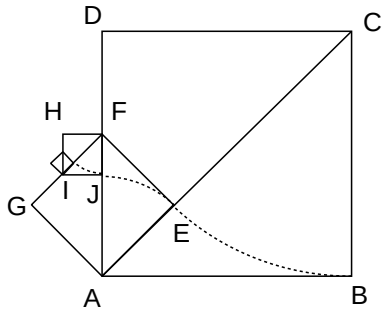


図 4.5: 正方形の対角線と辺に対する互除法

この手法は正方形 $ABCD$ の対角線 AC と辺 AB に対して互除法を用いるものです. 最初に対角線 AC 上の点 E を CE が辺 $BC(= AB)$ と同じ長さとなるように取ります. それから対角線 AC と点 E で直交する線分を引き, 辺 AD と交差する点を F とします. ここで三角形 CDF と CEF は斜辺を共有する直角三角形なので $EF = DF$, さらに三角形 AEF は二等辺直角三角形なので $AE = EF$ より $AC : AB = AF : AE$ が成立します. それから対角線 AC と線分 EF の垂線の交点を点 G とすると図形 $AEFG$ は正方形となります. ところで, この正方形 $AEFG$ に対しても同様の処理が行えるので, この操作は終了しません. このことから対角線は辺に対して非共測であることが判ります. この手法は Borelli(1608-1679) 等で提案されている手法ですが, この手法の発見者も特定することはできていません ([24] 参照).

Theodoros の授業:　これは Plato の「テアイテトス (Theaithetos)」([47]) の中に出てくるものです. 最初に紹介した二つの構築は後世の推測ですが, こちらはほぼ同時代の著作です. さて, この Theaithetos の舞台は紀元前 399 年とされ, そこで紹介されている Theodoros の授業では, 面積が 3 の正方形の辺の長さが 1 の正方形の辺で計り切れないことを示し, 以降, 面積が 17 となる正方形まで示し, 何故か Theodoros 先生は 17 で止めたというものです ([47],p.26 を参照). ここで面積が 2 となる正方形を挙げていないことから, Theodoros の授業の頃には $\sqrt{2}$ が無理数であることは既知の事柄であったと思われます. その一方で Theodoros の授業の水準が今一つであることから, 発見から然程時間が経っていなかったとも考えられています. この Theodoros の授業で用いられた手法の再構成に関しては [24] を参照して下さい.

偶数奇数論による証明:　これは現在の教科書等でも見られる背理法を用いた証明方法です. この証明方法を簡単に解説しておきましょう.

まず, 正方形の辺と対角線が整数の比 $a : b$ で表現されているとします. このときに三平方の定理から '$b^2 = 2a^2$' を充さなければなりません. このことから b は偶数で a は b と互いに素でなければならないので a は奇数でなければなりません. ここで b は偶数なので, '$b = 2n$' を充す整数 n が存在しなければなりませんが, このときに '$a^2 = 2n^2$' も充さなければなりません. したがって, 奇数の筈の a も偶数でなければならなくなりますが, れは流石に矛盾なので正方形の辺と対角線は非共測でなければならないというものです.

なお, この証明方法は Aristotle の著作「トピカ」の一節に, 「**対角線が共測であれば, 偶数は奇数と等しくなる**」とあるため, Aristotle が活躍していた時点では, $\sqrt{2}$ が無理数である証明が偶数奇数論による方法で知られていたことが伺えます.

無理数の発見の年代: 無理数の発見に関係する再構成には色々なものがありますが, 無理数の最初の発見者が誰であるか, そしてどのようにして発見したかは一切不明です. この無理数の発見は紀元前 430 年頃, 偶数奇数論で見付けたという Knorr の結論が妥当そうです ([24],p.118-121).

ところで, この黄金数の例は実数の一つの表現方法を示唆します. そのために Cantor による基本列 (Cauchy 列) と呼ばれる非常に良い性質を持った数列を使った実数の創造を紹介しましょう.

4.15.2 基本列を用いた実数の創造

数列 $\{a_i\}$ は任意の自然数 $i \in \mathbb{N}$ に対応する数 $b(= a_i)$ を与える対象です. ここで, 与えられた数列 $\{a_i\}$ が「**基本列**」, あるいは「**Cauchy 列**」であるとは, 任意の $\varepsilon > 0$ に対して十分大きな $n \in \mathbb{N}$ が存在して $m, n > N$ を充す自然数 m, n に対し, $|a_m - a_n| < \varepsilon$ を充す場合です. この基本列の性質として, 数列の絶対値がある一定の数値で抑えられること, すなわち, 「**上界を持つ**」という性質があります. そして, 任意の基本列 $\{a_i\}$ と $\{b_i\}$ の項単位の和, 差, 積から定められる数列も基本列となることが判ります:

基本列の和が基本列になること 任意の $\varepsilon > 0$ に対して十分大きな $N \in \mathbb{N}$ を取ると, m, $n > N$ に対して '$|a_m - a_n| < \varepsilon/2$ かつ $|b_m - b_n| < \varepsilon/2$' を充すようにできます. すると三角不等式によって '$|(a_m + b_m) - (a_n + b_n)| < |a_m - a_n| + |b_m - b_n| < \varepsilon$' となるので, 数列 $\{a_i + b_i\}$ も基本列になります.

基本列の積が基本列になること 任意の $\varepsilon > 0$ に対して十分大きな $N \in \mathbb{N}$ を取ると, m, $n > N$ に対して '$|a_m| < M/2$ かつ $|b_n| < M/2$' となる $M > 0$ が存在し, さらに '$|a_m - a_n| < \sqrt{\varepsilon}/M$ かつ $|b_m - b_n| < \sqrt{\varepsilon}/M$' を充すようにできます. すると '$|a_m b_m - a_n b_n| < |a_m b_m - a_m b_n + a_m b_n - a_n b_n| < |a_m||b_m - b_n| + |b_n||a_m - a_n| < \varepsilon$' となるので, 数列 $\{a_i b_i\}$ も基本列になります.

基本列を用いた実数の定義 有理数の基本列 $\{a_n\}$ の集合を F, その部分集合 N を零に収束する基本列 $\{r_n\}$ の集合とします. それから基本列の集合に入れる関係 "~" を有理数の列 $\{a_n\}$ と $\{b_n\}$ の極限が等しい場合に同値とすることで定めます:

$$\{a_n\} \sim \{b_n\} \stackrel{def}{=} \lim_{i\to\infty} a_i = \lim_{i\to\infty} b_i$$

この関係 "~" は同値関係となり, 実数 $\mathbb{R}$ を $\mathbb{R} \stackrel{def}{=} F/\sim$ で定義します. このときに有理数 $\mathbb{Q}$ の元 q を成分が全て q の数列 $\{q, q, q, \ldots\}$ に対応させることで '$\mathbb{Q} \subset \mathbb{R}$' が成立します. この同値関係による定義の長所は四則演算が有理数の四則演算から自然に拡張できる点です. たとえば $\alpha, \beta \in \mathbb{R}$ の代表を $\{a_n\}, \{b_n\}$ とするとき, 基本列の項毎の和 $a_i + b_i (i \in \mathbb{N})$ で構成された数列 $\{a_n + b_n\}$ も基本列で, その極限も一意に定まることから, $\mathbb{R}$ の元 $\alpha + \beta$ が定義できます. 乗法も同様にして得られた有理数の積の列 $\{a_n b_n\}$ が基本列となり, その極限が一意に定まるので $\mathbb{R}$ の元の積も自然に定義できます.

4.15.3　Dedekind の切断

基本列を用いた実数の創造の他に「**Dedekind の切断**」による実数の創造もあります. この切断は Dedekind の著書「連続性と無限数 (1872)」([36]) で初めて導入されたもので, 実数論の基礎付けを与えるものの一つです[20].

最初に幾つかの用語を導入しておきましょう. そこで, 順序集合 $(M, >)$ を考えます. ここで, '$a > b$' を充す任意の M の元に対し, '$a > c > b$' を充す $c \in M$ のことを「**中間元**」と呼びます. それから, $a, b \in M$ の間に中間元が存在する場合に順序集合 $(M, >)$ を「**稠密**」と呼びます. たとえば, 整数 $\mathbb{N}$ は稠密になりません. 何故なら, 1 と 2 の間に整数 $\mathbb{N}$ の元が何も存在しないからです. しかし, 有理数 $\mathbb{Q}$ は稠密集合となります. 実際, $a > b \in \mathbb{Q}$ とすると, 自然数 $n > 0$ に対して各 $b + (a - b)/(2^{n+1})$ が a, b の中間元となるからです.
次に, 順序集合 $(M, >)$ の集合 M を二つの部分 A, A' に分けますが, その分け方は任意の A の元 a が A' の任意の元　a'　に対して '$a' > a$' となるように分けます. このとき A を「**下組**」, A' を「**上組**」と呼ます. そして, 下組と上組の対 (A, A') を集合 M の「**Dedekind の切断**」, あるいは簡単に「**切断**」と呼びます.

Dedekind の切断には次の組合せが考えられます:

切断

D1.　下組　A　に最大元が有り, 上組　A'　にも最小元がある (「**跳躍**」)
D2.　下組　A　に最大元が無く, 上組　A'　に最小限がない (「**隙間**」)
D3.　下組　A　に最大元があり, 上組　A'　に最小限がない. または下組
　　A　に最大元がなく, 上組　A'　に最小限がある (「**正常の場合**」)

まず, 整数　$\mathbb{Z}$　の切断は常に D1. の跳躍だけです. そして, 有理数 $\mathbb{Q}$ では D2. の隙間と D3. の正常の場合の双方があります. たとえば, 下組　A を $\{x^2 < 2$ を充す有理数, あるいは, $x^2 > 2$ を充す負の有理数 $\}$, 上組　A' を $\{x^2 \geq 2$ を充す正の有理数 $\}$ とする切断 (A, A') は $\sqrt{2}$ が有理数でないために

[20] なお,「連続性と無限数」の序文によると,Dedekind は 1858 年の秋に切断を導き出しています. この切断の詳細は [36] や [32] を参照.

下組にも上組にも属さずに隙間になります. さらに, 有理数 $\mathbb{Q}$ の切断 $(\{x \in \mathbb{Q} | x < 1\}, \{x \in \mathbb{Q} | x \geq 1\})$ は上組に最小値 1 が存在し, 下組には最大値が存在しないので正常の切断になります.
なお, 有理数には D1. の跳躍となる切断は存在しません. 何故なら, 有理数 $\mathbb{Q}$ は稠密集合であり, 稠密集合については, その異なる任意の元 a, b に対して必ず中間元 c が存在するからです.

4.15.4 順序集合の連続性

さて, この切断を用いて順序集合 $(M, >)$ の「**連続性**」が定義できます[21]:

Dedekind の意味で連続

順序集合 $(M, >)$ が稠密であり, その全ての切断が正常の場合に「**Dedekind の意味で連続**」, あるいは簡単に「**連続**」と呼ぶ.

この Dedekind の連続の定義に対し, 古代ギリシャの Aristotle は「形而上学」において次の連続の定義を行っています: 「**.... 或る二つ以上の事物が連続的であるというのは, これらの事物がその各々の限界に於て接触し連続して, その各々の限界が同じになり一つになっている場合にである...**」([3](下巻,p.132-133) 参照). ちなみに, これに似た定義を Leibniz も行っています[22]. Dedekind の「**正常な切断**」は正にこの状況を表現したものになっていますね.

4.15.5 切断による実数の創造

Dedekind の切断の面白いところは新しい数を生成する能力を持つことで, Dedekind の著作「連続性と無限数」([36]) の §4 で無理数を創造しています. しかし, その前に Frege の主張にも留意しておくべきでしょう. それは「**数学者が創造を行うにあたって, それがまず可能であるかどうかを, 次に, それが既存のものと無矛盾であるかどうかに注意を払うべきである**」ということです (「算術の基本法則第 II 巻」([50],§138-142)). この点について Dedekind はやや安易で, 「連続性と無限数」([36]) の §4 にて単純に隙間が無理数と見做せると主張していますが, 切断は集合の対なので数と同列に論じることに躊躇いを感じます. そこで少し工夫をします. まず, D3. の正常の場合について下組 A の最大元が切断にある場合, その下組の最大元を上組 A' に移すことで下組に最大元がなく, 上組に最大元を持つ切断が得られます. またこの逆も可能なので, 下組に最大元がなく上組に最小元がある切断で正常な切断を代表させます. これによって切断は下組 A だけで定めることができます. すると集合 M の元 m を下組 $\{x \in M | x < m\}$ に一対一に対応させられるので, この下組を $\underline{m}$, M の切断の下組の集合を$\underline{M}$ と表記します. 次に集合 M の順序関係 “$>$” と互換な $\underline{M}$ の順序関係として$\underline{M}$ の包含関係 “$\supset$” を使います. 実際, $a, b \in M$ に対して ‘$a > b$’ を仮定すると数 b に対応する切断の下組 $\underline{b}$ に属する全ての元は数 a に対応する切断の下組 $\underline{a}$ に包含されます. このように順序集合 $(M, >)$ を順序集合 $(\underline{M}, \supset)$ で置換えて考察することができます.

[21] Dedekind による連続の説明は [36] の「連続性と無限数」,§3 の「直線の連続性」, 連続の定義は同 §3,p.20 を参照.
[22] [32] 上の第 2 章「実数」の前置き (p.29) を参照.

順序集合 (M, >) を上述のように M の正常な切断の集合と同一視した場合, m, $n \in$ M に対して和, 差, 積や商は次に示す切断にそれぞれ対応します:

切断と演算の対応

$$
\begin{array}{lcll}
m+n & \Leftrightarrow & \underline{m+n} & = \{x \in \mathrm{M} | x < m+n\} \\
m-n & \Leftrightarrow & \underline{m-n} & = \{x \in \mathrm{M} | x < m-n\} \\
m*n & \Leftrightarrow & \underline{m*n} & = \{x \in \mathrm{M} | x < m*n\} \\
m/n & \Leftrightarrow & \underline{m/n} & = \{x \in \mathrm{M} | x < m/n\}
\end{array}
$$

有理数の集合 M から「**実数の創造**」に移りましょう. まず, 切断 (A, A') を順序集合 (M, >) の隙間とします. ここで, この切断を α と名付け, この切断の下組を $\underline{\alpha}$ と表記します. そして, この「新成分」α を集合 M に取込みますが, M の元に対応する切断の下組の全体集合 $\underline{\mathrm{M}}$ を考え, この集合に対して新たに $\underline{\alpha}$ を追加して得られる全順序集合 $\underline{\mathrm{M}} \cup \{\underline{\alpha}\}$ を M′ と記述します. このとき, 新成分 $\underline{\alpha}$ と従来の $\underline{\mathrm{M}}$ の成分に対する順序を全順序集合 (M′, ⊃) の包含関係 "⊃" を用いて定めます. すると, この包含関係 "⊃" は$\underline{\mathrm{M}}$ の元同士の順序を保ち, $\underline{\alpha}$ との順序も自然に入り, その結果, (M′, ⊃) は全順序集合になります. ここで $a \in$ M と $\underline{a} \in \underline{\mathrm{M}}$ は同一視が行え, 順序 "⊃" も順序 ">" で置換えられます. この同一視によって全順序集合 ($\mathrm{M} \cup \{\alpha\}$, >) が得られますが, この全順序集合を ($\bar{\mathrm{M}}$, >) と表記します. この構成方法から α に対応する集合 $\bar{\mathrm{M}}$ の切断は正常な切断となることが分ります.
この全順序集合 ($\bar{\mathrm{M}}$, >) は稠密です. 実際, 任意の異なる $\bar{\mathrm{M}}$ の元 a, b を考えます. ここで $a > b$ を仮定しても一般性を失わないので, $a > b$ とします. もし, a, b の双方が M の元であれば集合 M の稠密性から $a > c > b$ を充す $c \in$ M が存在します. したがって, a か b のどちらかが α の場合を考察すればよいことになります. ここで $a = \alpha$ と仮定しましょう. このとき a と b の切断に対応する下組 $\underline{a}$ と $\underline{b}$ については $a > b$ より $\underline{\mathrm{a}} \supset \underline{\mathrm{b}}$ が成立しするので, M の切断の下組 $\underline{\mathrm{a}}$ に含まれて M の切断の下組 $\underline{\mathrm{b}}$ に含まれない $c \in$ M が存在しなければなりません. 何故なら, もし, 存在しなければ $\underline{\mathrm{a}} = \underline{\mathrm{b}}$ となるからです. これと同様の議論は $b = \alpha$ の場合でも行えるので, 以上から a と b の間に $\bar{\mathrm{M}}$ の中間元が存在すること, すなわち $\bar{\mathrm{M}}$ も稠密であることが判ります.

このように $\bar{\mathrm{M}}$ が稠密であることが判りました. では稠密な順序集合 (M, >) の全ての隙間を次々と新成分として加えて最終的に得られる順序集合 ($\mathfrak{M}$, >) は「**Dedekind の意味で連続**」になるでしょうか？ そのためには $\mathfrak{M}$ の切断が全て正常の場合であることを示さなければなりません.

そこで $\mathfrak{M}$ の切断 ($\mathfrak{A}$, $\mathfrak{A}'$) を考えます. この切断の下組 $\mathfrak{A}$ と上組 $\mathfrak{A}'$ の M の元の集合をそれぞれ A, A' とすると集合対 (A, A') は M の切断になります. ここで切断 (A, A') が表現する元を c とします. この元 c は $\mathfrak{M}$ の元でもあるので, 切断 $\mathfrak{A}$ が $\mathfrak{A}'$ の何れかに属します. そこで, $c \in \mathfrak{A}$ としましょう. もし, c が $\mathfrak{A}$ の最大元でなければ '$a > c$' を充す $a \in \mathfrak{A}$ が存在します. ところが M の稠密性より '$a > a_1 > c$' を充す $a_1 \in A$ が存在しなければなりませんが, このことは A が c の下組であることに矛盾します. したがって, $c \in \mathfrak{M}$ であれば c は $\mathfrak{A}$ の最大元でなければなりません. 同様に, $c \in \mathfrak{A}'$ とした場合に c が $\mathfrak{A}'$ の最小元でなければ, '$c > a$' を充す $\mathfrak{A}'$ の元 a が存在します. ここで, M の稠密性から '$c > a_1 > a$' を充す元 $a_1 \in A$ が存在することになって再び矛盾します.

このように, 稠密集合 M の全ての隙間を潰して得られた集合 $\mathfrak{M}$ は「Dedekind の意味で連続」であることが判りました. そして, 有理数 $\mathbb{Q}$ の隙間を潰して得られ順序集合こそが実数 $\mathbb{R}$ になります.

4.15.6 実数の公理系

Dedekind の切断を用いて「実数」を創造してみましたが, 前述のように Cantor の基本列を用いた方法でも実数を構成できました. では, これらの構成で造られた「実数」はすべて同じもののでしょうか？ さらに, どうして同じ「実数」であるといえるのでしょうか？実際は, このことは証明すべきことなのですが, ここでは実数が持つべき性質を公理として次に纏めておきましょう:

実数の公理系

R1. $(K, +, *)$ は体である
R2. 順序 "≥" は演算 "+" と演算 "$*$" と両立する K 上の全順序である
R3. 下界を持つ任意の空でない部分集合 $M \subset K$ は K において下限を持つ

まず, R1 で k は和 "+" と積 "*" を持つ体であることを主張しています. そして R2 の意味することは次のことです:

1. K は全順序集合である. すなわち, $x, y \in K$ に対して関係 '$x < y$', '$x > y$', '$x = y$' の何れか一つだけが成立する.
2. '$x \leq y$' を充す K の元 x, y と任意の $z \in K$ に対して '$x + z \leq y + z$' を充す
3. '$x \leq y$' を充す K の元 x, y と任意の $z \geq 0 \in K$ に対して '$x * z \leq y * z$' を充す

最後の R3 では「下界」や「下限」といった言葉が出ています. ここでは「**下界**」と「**下限**」, あるいは「**上界**」と「上限」という言葉の意味を確認しておきましょう.
まず, 集合 $M \in \mathbb{R}$ に対して $a \in \mathbb{R}$ が集合 M の「**下界**」であるとは, 集合 M が '$\forall x \in M x > a$' を充す場合です. 同様に a が「**上界**」であるとは '$\forall x \in M x < a$' を充す場合です. そして, 全ての集合 M の下限の集合を考えたとき, その最大の元を集合 M の「**下限**」と呼び, 全ての集合 M の上界の集合を考えたとき, その最小の元を集合 M の上限と呼びます.
この R3 を充す集合 K では切断は正常のものになり, 集合 K は Dedekind の意味で連続であることが分ります.

では, 今迄出てきた自然数, 整数や有理数は, どの公理を充し, あるいは充さないのでしょうか？ まず, 自然数 $\mathbb{N}$ は群にも環にもならないために公理 R1 を充しませんが, 大小関係によって全順序集合であり, 和 "+" と積 "$*$" と両立するので, 公理 R2 は充します. そして, 整数 $\mathbb{Z}$ は環になるものの体にはならないために公理 R1 を充しませんが, 公理 R2 は自然数と同様に充します. そして, 有理数 $\mathbb{Q}$ は体となるために公理 R1 を充し, さらに公理 R2 も充します. ところが, 有理数 $\mathbb{Q}$ については, 集合 $\{x \in \mathbb{Q} | x \geq \sqrt{2}\}$ を考えるとその下限の $\sqrt{2}$ は $\mathbb{Q}$ には含まれないために R3 を充さず, この $\sqrt{2}$ が切断の隙間となるために Dedekind の意味で連続になりません.
そして実数はこの公理系を充す対象としては包含関係 "⊃" に対して最大のものになります. つまり, 実数を包含する集合は上記の公理系の何れか一つを充さなくなります. たとえば, 複素数 $\mathbb{C}$ は順

序集合にはなりません. 実際, 純虚数 i と整数 1 の間には大小関係 “>” が入りませんね. もし, 複素数の絶対値で大きさを入れても, 集合 $\{e^{ia}, a \in \mathbb{R}\}$ は Gauß平面上の半径 1 の円周となるので, 同じ大きさの複素数が多数存在することが判ります.

さて, 集合 K が実数の公理系を充すとします. すると, この集合 K は次の命題も充さなければなりません:

1. K の下方有界な全ての部分集合は下限を持つ
2. K の上方有界な全ての部分集合は上限を持つ
3. (α, β) を K の切断とするときに β は最小限を持つ
4. 全ての下方有界な単調減少列は収束し, その極限は K に含まれる
5. 全ての上方有界な単調増加列は収束し, その極限は K に含まれる
6. 体 K は Archimedes 的に順序付けられており, K の元で構成された基本列は収束する
7. 体 K は Archimedes 的に順序付けられており, 自然数 n が増大するときに, 区間 I_n の長さが零に収束する K の全ての区間縮小列 $I_0 \supset I_1 \supset \cdots$ に対し, 全ての区間 I_n に含まれる K の元 s が唯一存在する.

ここで「**Archimedes 的に順序付けられている**」とは, 任意の正の数 $a, b \in K$ に対し, ある自然数 c が存在して ‘$a < c < b$’ となる性質です. この性質は「**Archimedes の公理**」とも呼ばれています. そして「**区間縮小法の原理**」は二組の数列 $\{a_n\}, \{b_n\}$ に対して ‘$a_1 \leq a_2 \leq \cdots \leq a_n \leq b_n \leq \cdots \leq b_2 \leq b_1$ かつ $\lim_{n \leftarrow \infty}(b_n - a_n) = 0$’ であれば ‘$\lim a_n = \lim b_n = c$’ となる数 $c \in K$ が存在するというものです.

さて, この実数の公理系では「無限」についての言及はありません. そして, 今迄の「**無限**」は際限なくどこまでも成長してゆく性質であって, 単独で存在するものではありません. ところが論理主義や集合論では「**無限**」を実在のものとして捉え, その上, 集合論では「**無限**」に対しても順序を入れることで「**超限順序数**」と呼ばれる順序数を構築しているのです.

4.16　超限順序数

自然数には色々な意味を持たせることができます. たとえば, 袋の中の林檎の個数は, 「ひとつ」, 「ふたつ」,…と数られますが, ここでの林檎の個数に対応するものが「基数」, あるいは「濃度」です. そして, 競争順位であらわれる順序のように「いちばん」,「にばん」,…と順序に対応するものがあり, こちらは「順序数」と呼ばれます. すると個数としての「2」と順序としての「2」は字面が同じなので区別がつきません. この辺をすっきりさせることを目的として話を進めましょう.
この自然数は Cantor の集合を使って構築することができます. 次の集合の列を考えてみましょう:

$$\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}, \ldots,$$

この列は帰納的に生成して得られた列ですが, 各集合の列の濃度 (基数) が自然数に対応します:

$$
\begin{aligned}
0 &\Leftrightarrow \emptyset \\
1 &\Leftrightarrow \{\emptyset\} \\
2 &\Leftrightarrow \{\emptyset, \{\emptyset\}\} \\
3 &\Leftrightarrow \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} \\
&\cdots
\end{aligned}
$$

そこで, $\emptyset$ を 0 とします. 次の $\{\emptyset\}$ は $0 = \emptyset$ としたために $\{0\}$ となり, $\{0\}$ を 1 とします. 次の, $\{\emptyset, \{\emptyset\}\}$ は $\{0, 1\}$ で, これを 2 としましょう. 以降, この作業を繰返すと上記の集合は $\{1, 2, \ldots, n-1\}$ の形の集合の列になり, 集合 $\{1, 2, \ldots, n-1\}$ が自然数 n に対応します[23].

この集合の列を構成する任意の集合には, その構成方法から包含関係 "$\subset$" による順序が自然に入ります. このことから, これらの集合で構成された集合 C は順序 "$\subset$" で順序が入った全順序集合になります. さらに, この集合の列から任意の部分列を取出すと, 包含関係による最小元が, ここでの集合の構成方法から必ず存在します. この性質を「整列性」と呼び, 整列性を持つ集合を「整列集合」と呼びます.

一般的に全順序集合 $(A, >_A)$ と $(B, >_B)$ 与えられ, 写像 $f : B \to A$ が一対一の写像であり, $a, b \in B$ に対して $a >_B b$ であって, $f(a) >_A f(b)$ を充す場合に, $(B, >_B)$ は $(A, >_A)$ と「**同じ順序型を持つ**」と呼びます. そして, ここでの同型写像 f を「**順序同型**」と呼び, 集合 $(A, >_A)$ が整列集合の場合を特に「**順序数**」と呼びます.

さて, 集合 C の元と自然数を集合 C の元の濃度を返す函数 card で対応関係を付けましたが, この函数は集合 C の元 a, b に対し, $a \subset b$ であれば,$\mathrm{card}(a) < \mathrm{card}(b)$ が成立します. すなわち, この函数 card は順序を保つ函数です. このことから自然数 $\mathbb{N}$ は整列集合 C と同じ順序型を持つために順序数としての特徴も持つことが判ります.

LISP を使った実験

ここで解説した集合の定義方法を使うと, 個数を表現する基数と順番を表現する順序数が厳密に区別されていますが, ちょっと判り難いかもしれません. そこで, LISP を使って, この考え方を実験してみましょう:
最初の空集合 $\emptyset$ は空リスト "()" を用いて表現します. すると, 上記の集合を作る操作は与えられたリストを成分に持つ単リストを生成して, その単リストを与えられたリストに加える操作になります. ここで, この操作を行う函数を s と名付けましょう. この函数 s は `(defun s (x) (append x (list x))` で定義できますが, より LISP らしく次の函数で定義します:

函数 s の LISP による表現

```
(defun s (x) (cons x x))
```

[23] 色不異空, 空不異色, 色即是空, 空即是色, 受想行色, 亦不如是.
実体は空に他ならず, 空は実体に他ならない. 実体はすなわち空であり, 空はすなわち実体であり, 受, 想, 行, 識もまた, これと同様である (般若心経より)

この函数の実行例を次に示しておきます. 先程の集合と順序が逆になっていますが同じものが得られていますね:

```
[2]> (s '())
(NIL)
[3]> (s (s '()))
((NIL) NIL)
[4]> (s (s (s '())))
(((NIL) NIL) (NIL) NIL)
[5]> (s (s (s (s '()))))
((((NIL) NIL) (NIL) NIL) ((NIL) NIL) (NIL) NIL)
[6]> (s (s (s (s (s '())))))
(((((NIL) NIL) (NIL) NIL) ((NIL) NIL) (NIL) NIL) (((NIL) NIL) (NIL) NIL)
 ((NIL) NIL) (NIL) NIL)
```

LISP ではリストの長さは length 函数で求められます. このリストの長さが集合の基数に対応します. 実際に確認してみましょう:

```
[7]> (length '())
0
[8]> (length (s '()))
1
[9]> (length (s (s '())))
2
[10]> (length (s (s (s '()))))
3
[11]> (length (s (s (s (s '())))))
4
[12]> (length (s (s (s (s (s '()))))))
5
```

このように基数と順序数が対応しています. なお, この構成による順序数の基数が有限であれば「**有限順序数**」と呼びます.

4.16.1　超限順序数

この方法の面白いことは, より大きな数を続々作られることです. そこで, この作業を延々と続けて得られる $\{\emptyset, \{\emptyset\}, \{\{\emptyset, \{\emptyset\}\}, \ldots\}$ を「**超限順序数**」と呼んで, ω と記述します. このとき, ω の濃度は $\aleph_0$ になります. それから, この ω に対して $\omega + 1$ を $\{\emptyset, \{\omega\}\}$ で定義します. あとは, この操作を延々と続けて行くことで次の順序数列が得られます:

$$1, 2, \ldots, \omega, \omega + 1, \omega + 2, \ldots, 2\omega, \ldots, 3\omega, \ldots, \omega^{\omega}, \ldots, \omega^{\omega^{\omega}}, \ldots, \omega^{\omega^{\omega^{\omega}}}, \ldots$$

なお, これらの超限順序数の基数は全て $\aleph_0$ となるので, 順序数と基数は異なった性質を持つことが判ります.
ここで注意することに, この超限順序数で現われる「無限」は Aristotle の著作「形而上学」で「**無限とは限界が無く, それ自体のみで存在できるものではない…**」([3] 下巻,p.120-125) とは随分異っていることです. ここで Aristotle の考えは Poincaré の著書「科学と方法」の第三章の「数学と論理」([52],p.151) にも見られるものです. 実際,「その無限とは哲学者によって生成と呼ばれるところのものであった. 数学的無限とは, あらゆる量を越えて増大する可能性を持つ変量に過ぎなかった」とあります. ここでの超限順序数での「無限」は, こういった考えとは異なっており, 現に存在する実体なのです. この点が Cantor の集合論の受容の障害の一つとなったように思えます[24].
さらに Cantor の集合論では集合を際限なく生成できることから, このような「無限」さえも際限なく生成できます. この際限もなく集合を生成することのできる能力から, いろいろ厄介な問題も出て来たのです. そして, このことは, ほぼ同時期に始まった「数学の厳密化」とも関係してゆきます.

4.17 数学の厳密化

クレタ人の逆理

最初に昔からの有名な逆理を一つ:

クレタ人の逆理

クレタ人はうそつきである.

これを当時のエジプト人やギリシャ人が言ったのであれば問題がないのですが, Epimenides というクレタ人[25]が言ったがために有名になった主張です. この主張が本当であれば主張した本人がうそつきとなるので, この主張は嘘だということになります. したがって,「クレタ人がうそつきではない」が本当となりますが, すると, Epimenides が「クレタ人はうそつきである」という嘘をついていることになるので, 彼はうそつきでなければなりません. と, 堂々巡りとなって何ともややこしいですね. なお, この逆理は「**うそつきの逆理**」としても知られており, この逆理の派生版として「**俺はうそつきだ**」というものもあります.

Russell の逆理

「**クレタ人の逆理**」と似た逆理に「**Russell の逆理**」[26] があります. この逆理は式で書くと明瞭なのですが, 普通の人にも判りやすく Russell 本人が言い換えたものが「**床屋の逆理**」として知られています:

[24][52],p.151 ではさらにこう述べています:「カントルは数学に実無限, いいかえれば, あらゆる限界を越えようとする可能性あるのみならず, 実際それを越えてしまったと見做される如き量を導入しようと企てた」
[25]紀元前 6 世紀頃のクレタのクノッソスの哲学者だそうです
[26]1901 年の春に Russell が発見していますが, Peano の助手であった Cesare Burali-Forti も 1897 年に発見しています. さらに Zermelo も独立して発見しており, 1903 年以前に Hilbert 達も知っています ([51], p.90 の Hilbert から Frege への 1903 年 11 月 7 日付けの書簡参照)

床屋の逆理

とある村には床屋が一軒だけあります. そこの床屋の主人は日頃, 自分で髭を剃らない村人の髭しか剃らないと言っています. では, その床屋の主人の髭は誰が剃るのでしょうか？

床屋の主人は女だったという話は抜きにして[27], この床屋の主張が正しいとしましょう. すると, 床屋が自分の髭を剃れば, 床屋本人は自分で髭を剃る人になってしまうので, 床屋の日頃のモットーから自分の髭を剃るわけにはいきません. だからといって髭を剃らなれけば「自分で髭を剃らない人」になってしまうので, 自分の髭を剃らなければならないことになります.

この Russell の逆理は本質的に $\{x|x \notin x\}$ という類を考えたときに生じる問題です. 実際, $v \stackrel{def}{=} \{x|x \notin x\}$ と置いて '$v \in v$' とすると, 類 v の定義から '$v \notin v$' でなければなりません. ところが, '$v \notin v$' とすれば類 v の定義より '$v \in v$' でなければなりません. この類は「**めろめろな猫**」のように単に不明瞭な命題による定義ではなく, Cantor 流の「**無限**」が関係する話でもなく, '$x \notin x$' という単純で明快な論理式[28]で発生している事態が非常に重要です. そのため, Russell の立場に批判的な Poincarè もある意味で賞賛した程です. なお, この Russell の逆理の派生版として, ここで解説した「**床屋の逆理**」の他に「**図書館の目録の逆理**」,「**オランダの市長の逆理**」があります[29].

Richard の逆理

「**Richard の逆理 (1905)**」[30] を紹介しておきましょう. この Richard はフランスの Lycée de Dijon の数学教師だったそうです:

Richard の逆理

最初に有限個の語で定義された全ての自然数を考えます. たとえば, E をそのような自然数の集合としましょう. すると, 集合 E の基数は $\aleph_0$ となるので, 集合 E の元を順番に並べられます. そこで自然数 N を次の☆の手順で定めます:
☆:「n 番目の自然数の n 桁目が p ならば自然数 N の n 桁目を $p+1$ とします.
もしも $p=9$ であれば n 桁目を 0 とします.」
すると, この自然数 N は集合 E に含まれるどの数とも異なる数になりますが, 自然数 N は上記のように有限個の語で定義されている数なので, 集合 E に含まれていなければなりません.

この逆理では新しい自然数を構成するために「**Cantor の対角線論法**」を用いています. そして, 最初に全体集合を定め, それから, 全体集合を使って集合に所属する元を対角線論法を用いて構築している点が大きな特徴です. そして, この問題となる元は, その元が帰属する全体集合を定めなければ定義できない性質を持っています.
これに似たものとして「数学雑談」([31]) に出ている「**Finsler の考案の逆理**」を挙げておきましょう:

---

[27] 女でも髭の濃い人は居ます.
[28] Poincaré の「科学と方法」[52] での分析や Russell の「悪循環原理」(§4.20) で排除される迄のことですが
[29] これらの逆理の紹介は「ゲーデルの世界」[42], p.36-38 を参照.
[30] Jules Richard が Reveu générale des Sciences に投稿した書簡
「Les Principles des mathématiques et le probléme des ensembles」にある逆理です.
なお, 書簡は 1905 年 6 月 30 日の Reveu générale des Sciences に掲載され ([52],p.204 参照), 現在は
`http://visualiseur.bnf.fr/CadresFenetre?O=NUMM-17080&I=545&M=tdm` で読めます.

1, 2, 3, 4, 5,
この枠の中で指示されていない最小の自然数.

ここで上の黒枠の中に直接書かれていない最小の自然数は 6 です. ところが, この 6 という自然数は「**この黒枠の中で指示されていない最小の自然数**」として指定されています. とすれば, 6 ではなく次の自然数の 7 がここでの最小の自然数となりますが, 今度は 7 が指示されることになります. このように際限がなくなってしまいます[31].

Cantor の逆理

最後に Cantor が見付けた「**Cantor の逆理**」として知られる逆理を紹介しましょう:

Cantor の逆理

1. 集合 S に対して $\mathrm{card}(S) < \mathrm{card}(\mathfrak{P}(S))$ が成立:「**Cantor の定理**」.
2.「全ての集合の集合」M に対しては $\mathrm{card}(M) \geq \mathrm{card}(\mathfrak{P}(M))$ も成立.

「**Cantor の定理**」として知られている定理が 1. に示す定理で, その意味は「集合 S の基数 $\mathrm{card}(S)$ は集合 S の冪集合 $\mathfrak{P}(S)$ の基数 $\mathrm{card}(\mathfrak{P}(S))$ よりも小である」というものです.
では, 今度は集合 M を「全ての集合の集合」としましょう. それから集合 M の冪集合 $\mathfrak{P}(M)$ の基数 $\mathrm{card}(\mathfrak{P}(M))$ を考えます. すると「Cantor の定理」により直ちに '$\mathrm{card}(M) < \mathrm{card}(\mathfrak{P}(M))$' を得ますが, その一方で「全ての集合を元とする集合」である集合 M を考えているので, 当然 '$\mathfrak{P}(M) \in M$' が成立します. このことから '$\mathrm{card}(M) \geq \mathrm{card}(\mathfrak{P}(M))$' も同時に得られるので矛盾が生じます.

この逆理を発見した Cantor 自身はこれを逆理とは思わず, むしろ, 積極的に超限数の一つ特徴と考えていたようです ([30] 参照). その一方で Russell は「Cantor の逆理」を研究することで, Cantor の逆理を純化した逆理として前に紹介した「Russell の逆理」を得ています[32].

4.17.1 逆理の分類

Ramsey[33]や Peano による逆理の分析から, 現在は「**論理的逆理 (logical paradox)**」と「**意味論的逆理 (semantic paradox)**」の二種類に逆理は分類されます. ここで「Russell の逆理」と「Cantor の逆理」のように意味論的な項を全く含まない, 純粋に論理的な逆理は論理的逆理に分類され,「嘘つきの逆理」と「Richard の逆理」のように命題の真偽や指示といった命題の意味を参照する逆理は意味論的逆理に分類されます. 後述の型の理論は論理的逆理に対しては非常に有効ですが, 意味論的な逆理に対してはまだ不十分のようです.

[31]「死刑囚の逆理」で囚人が行う推論に似ていますね. ただし, こちらの結末は最悪ですが.
[32]1902 年 6 月 24 日付けの Russell から Frege への書簡にその経緯が記されています ([51],p.122 参照).
[33]Ramsey は短い生涯でしたが, 色々なことをしている人です. 数理論理学だけではなく経済学でも名前を残しています.

4.17.2　19 世紀の数学の算術化との関連

さて, 逆理は昔から上述の「クレタ人の逆理」, 先行する亀に俊足のアキレスが追い付けないという「アキレスの逆理」, 「全能の神は自身で持ち上げられない巨石を創り出せるか？」[34]といった神学上の逆理と古来から実にいろいろあります. ここで, 「Russell の逆理」や「Cantor の逆理」が真面目に研究された時代背景としては 19 世紀の数学の厳密化と算術化の進行が挙げられます.

まず, 数学の厳密化は Cauchy の解析学の教科書「解析学教程」(1821 年) からはじまったと言われています. この「解析学教程」以前の微分積分学では「**限りなく小さくなるが 0 に決してならない数**」としての「**無限小**」を用いているために「**無限小解析**」と呼ばれています. この無限小の扱いはともすれば雑なもので, 「無限小」が「0 ではない」ためにそれで式を割るといった操作を許容する一方で, 「限りなく小さい」ために 0 で置換えるといっ都合の良い利用方法で 17, 18 世紀の解析学の重要な定理が導かれています.

Cauchy の「解析学教程」は Euclid 幾何学をモデルとして解析学を展開することを目的としたもので, 最初に極限を定義し, その極限が 0 となる変量として「無限小」の定義を行います. 函数の連続性については, この無限小を用いた定義を用いています. すなわ ち a を無限小量としたときに $f(x+a)-f(x)$ が a の減少と共に減少する場合を「x で連続」としています[35]. この「解析学教程」を経て, Weierstrass の「$\varepsilon-\delta$ 論法 (1861 年)」によって函数の連続性は不等式を用いた代数的な処理に帰着されています.

すると次に問題になるのが「**数とは何か？**」という疑問です. 自然数が確固としたものであれば前節で解説したように, 自然数から整数, 有理数, 実数が構成できます. その上, 無限を扱う論証には数学的帰納法が使えます. とすれば, この自然数とは一体何でしょうか？
この疑問について, 19 世紀後半では非常に素朴で形式的な立場, 心理的なものとする立場, あるいは経験的なものとする立場がありました. なお, ここでの素朴な形式的な立場は Hilbert の形式主義とは異なる非常に純朴なもので, 単純に数を記号とみなして式の操作を機械的に行うという立場です. この立場ではあまりにも純朴過ぎて, たとえば, 印刷された '1' と '*1*' が同値であるかどうかということさえ満足に説明できません.

この状況を垣間見るために, Dedekind の「数とは何か, 何であるべきか (1884)」([36]) を眺めてみましょう. この著作は自然数を公理的な立場で明確にしようとした立場にあり, 興味深いことに, その表紙には「**いつでも人間は算術をする (**$\grave{\alpha}\epsilon\iota\ o\ \grave{\alpha}\nu\theta\rho\omega\pi o\varsigma\ \grave{\alpha}\rho\iota\theta\mu\tau\iota\zeta\epsilon\iota$**)**」と掲げられています. これは Plato の言葉と伝えられる「**いつでも神は幾何学をする**」に対応する言葉で, このことから数を算術的に定義付けしようとする立場であることが伺えます. それから序文を見てみましょう. そこには「**私が数論 (代数学, 解析学) が論理学の一部分であると言ったこと…**」([36],p.41) と述べている一節が目を引きます. このことからも Dedekind は論理主義的な立場にあると言えるでしょう. ところが §5 の 66. 無限集合の存在証明 ([36],p.81) で Dedekind は「**私の思考の世界, すなわち, 私**

[34] イエス:「神を試してはならない」(マタイによる福音書).
[35] これだけでは $\epsilon-\delta$ 論法の創始とは言えませんが,Cauchy の「微分積分学要論」の 7 講で $\epsilon-\delta$ 論法による証明があるとのことです. なお, $\epsilon-\delta$ 論法の形成過程については [39] を参照

の思考の対象となり得るあらゆる事物の集合 S は無限である」と述べ, さらに, 「**もし, s が S の要素とすると, s が私の思考の対象である得るという考え s' はそれ自身 S の要素である…**」と Russell や Cantor の逆理を思わせる陳述, 「**…したがって S' のうちには含まれていないような要素 (たとえば, 私本来の「我」) が存在しているからである…**」といった心理主義的な記述があります. Frege ならずとも, そもそも数が心理的なものであれば, 「**私の数**」と「**貴方の数**」が同じ概念, あるいは同じ対象であるかどうかは疑わしく, 数の議論は満足にはできないと判断しても仕方がないでしょう. このように, Dedekind の本では数の導出を公理的に進めようとしているものの, 心理学を持ち出している論証をみる限り, 純粋に論理学から算術を導出してはいません. [36]

この状況に対して Frege は Euclid 幾何学をモデルに論理から算術を導出することを目指した「概念記法」([48]), 「算術の基礎」([49]), 「算術の基本法則」([50]) で代表される著作の中で, この Dedekind の例に散見される心理主義, 形式主義, そして経験主義による数の考え方を批判しています. さらに Frege は厳密に論理主義を遂行するために「概念記法」と呼ぶ新しい論理式の表記, さらには函数や「**全て**」や「**ある**」といった「**量化詞**」を論理学に導入して Aristotle 以来の論理学を一新しています. ただし, 「概念記法」自体は一般には受入れられず, 同時期に現われた Peano 流儀の論理式の表記が用いられています.

この Frege に少し遅れて論理学から自然数だけではなく数学全体を体系的に導出して見せたのが Russell の「数学の諸原理 (The Principles of Mathematics,1903,[90])」です. しかし, この「数学の諸原理」の出版間際に著者本人によって「Russell の逆理」が発見されたために, その本来の目論見は破綻してしまい, 1908 年の Poincaré の著書「科学と方法」では,「数学の諸原理」で展開した「論理主義」の死亡宣告が行われている有様です[37].

ところで, この「Russell の逆理」は非常に純粋に論理学的な逆理であるため, 数学の根幹に関わる問題と考えられました. この論理主義に懐疑的であった Poincaré も, この逆理に対しては「**数学的論理学はもはや不毛ではなく, 実に二律背反を産むのである**」([52],p.209) と述べ, これらの逆理の分析を行っています.

4.17.3 Poincaré による逆理の分析

Poincaré は著書「科学と方法」([52]) で幾つかの逆理を分析しています. そして「Richard の逆理」の分析から「**非可述的 (non-predicative)**」[38]な定義から逆理が生じていると「非確定的な定義と見做さなければならぬ定義は循環論法を含む定義である」と述べています ([52],p.204). たとえば, 「偶数の集合」や「身長 170cm 以下の人の集合」といった集合の定義では「自然数の集合」や「人間の集合」といった集合の概念全体に触れずに集合がきちんと定義ができているので, これらの集合の定義を「**可述的**」と呼びます. ところが, ここで挙げた逆理に関連する集合はどれも, その集合の

[36]西田幾多郎の「善の研究」の最初の頁から矢鱈に心理学的な記述があることからも, 心理学的な考えは 19 世紀末の思潮だったのかもしれません.

[37][52],p.211,「古い数理論理学は死した」. ただし, これに続けて Russell が当時研究中であった無クラス理論とジグザグ理論を挙げ,「新しいものを判ずるためには, その生れ出ずるのを待とうと思う」で章を結んでいます.

[38]「科学と方法」([52]) では「非確定的」と訳されています.

概念に直接触れなければ定義ができないものや, それも循環論法に訴えなければ定義できない集合です. 実際,「クレタ人の逆理」では「私の全ての主張 は嘘であるという主張」,「Russell の逆理」では「自分自身 を含まない集合」, Richard の逆理では「全体を用いて定義可能な元」,「Cantor の逆理」では「全ての集合 の集合」と見事に循環論法になっていることが判りますね. ただし, 全ての循環的な定義が逆理となるとは限らず, たとえば解析学で循環的な定義が所々にある点が難しいことです.
なお, Poincaré は Cantor の集合論には批判的でした. たとえば Cantor が実無限を実在のものとして扱うことについて,「科学と方法」では「数学的無限はあらゆる限界を越えて増大しようとする量」なのに Cantor の実無限は「実際に超てしまったと見做さる如き量」であると述べ,「実無限は存在しない」と言い切っています. そして,「Cantor の徒はこれを忘れて矛盾に陥ったのである」と述べています. それから「論理主義者もこれを忘れて同様の困難に遭遇した」と述べ, Russell の「The Principles of Mathematics」([90]) の実在論的傾向[39] を指摘しています ([52],p.151-153,p.210-211 を参照)

4.17.4　三つの道

さて,「Russell の逆理」に代表される逆理は単純に Russell や Frege の数学の厳密化の遂行を危機に陥れただけではなく, 数学全般に対して「**数学の危機**」と呼ばれる状況を招きます. 実際,「Cantor の逆理」であれば, Cantor の集合論の怪しさで片付けられますが,「Russell の逆理」のように純粋に論理学的な逆理は, 数学の厳密化・算術化を遂行する上で大きな障害となるのです.
この逆理が契機となり, 数学をより強固なものにする立場として, Russell に代表される「**論理主義**」, Brouwer の「**直観主義**」, Hilbert の「**形式主義**」といった三つの立場が現れます.

ここで各立場の概要を説明しておきましょう:

論理主義:　論理主義の立場は, 論理学からの数学の導出を目指すというもので, これは Boole が論理学の代数化を図ったこととちょうど逆の方向になります. この論理主義は Euclid の幾何学をモデルとし, 数学を論理学の一部として捉え, 少数の公理で構成された公理系から数学を再構築することを目的としており,「論理学は数学の青年時代であり, 数学は論理学の壮年時代である」という Russell の主張がその本質を語っています.
この論理主義は Boole, Frege や Peano の仕事に大きな源を持っています. まず, Boole は現在では Boole 代数で知られていますが, 命題を数式で表現することで代数的に扱っていますが, どちらかと言えば冒険者的な立場です. この Boole の論理学から Schöder は論理学を構築しています. そして, Peano は現在の数理論理学で用いられている表記の大本を作った人で, 前述のように算術の公理化を行っています.

Frege は Peano と異なる独自の二次元的な表記の「概念記法」[40]を用い, さらには変項・函数による命題の表記と量化記号の導入を行って, Aristotle から続く論理学を一新しています. そして Frege

[39] [90] の実在論的傾向は [4] の第三章を参照.
[40] 詳細は [48],[50] を参照. なお, [81] には [48] の英訳. [33] には Frege の著作の解説があります. 簡単な紹介は §4.19 を参照

は「算術の基礎」([49]) や「算術の基本法則」([50]) で論理学から算術を導出しようとしています. まず, Frege によれば, 算術はインドに起源を持つことから曖昧な面が残っているために, より厳密な Euclid の原論をモデルに算術を再構築しようとしたものです. これは Aristotle の方針にしたがって幾何学を構築した Euclid に倣ったものといえるでしょう. この方針は「算術の基本法則」で頂点に達します. 「算術の基本法則」では「概念の外延」, すなわち, クラスを根底に自然数を構築し, Cantor の $\aleph_0$ に相当する $\backsim$ を定義しています. ところが「外延 (クラス)」という集合論的な対象を導入したため, 最初から厄介な問題 (Russell の逆理という時限爆弾) を包含することになります. この「算術の基本法則」は出版社に出版を渋られたために I 巻,II 巻と分けて出版しますが, 結局, II 巻は I 巻の出版の 10 年後に自費出版しています. 更に不幸なことに, この II 巻の出版直前に Russell からの手紙で彼の基本法則の公理系から「Russell の逆理」が導出可能なことを報らされます. そこで, Frege は問題となる公理 V の修正を行っていますが, 晩年には論理学から数学を導出することを諦めていたようです[41].

この Frege と入れ替わるように Russell が論理主義を強力に推し進めます. 「数学の諸原理」(The principles of Mathematics,1903) で Frege と独立して自然数の定式化を行い, さらに数学を論理学から導出しています. また, Appendix で Frege の本格的な紹介も行っています. ところが, Russell は「数学の諸原理」の著作中に「Russell の逆理」を発見して大きく落胆することになります. この「数学の諸原理」で代表される「論理主義」に対して Poincaré は「科学と方法」([52]) にて「体系の無矛盾を証明するためには最終的に数学的帰納法に訴えるしかないが, その数学的帰納法自体が論理に還元できない性質を持っている」と述べて疑問を投げ掛けています.
Russell は逆理対策のために無クラス理論等を考察しましたが, Poincaé の分析に答える形 ([92],p.37 の注を参照) で Russell は分岐的階型理論を論文「The Mathematical logic as based on the theory of types」(1908,[91] 参照) で導入し, Whitehead との共著「数学原論」(Principia Mathematica,Vol.1 1910,Vol.2 1912,Vol.3 1913) で一つの体系として完成させました.

この Principia Mathematica では「悪循環原理 (Vicious cycle principle)」により, Russell の逆理のように自分自身に言及しなければ定義ができない「非可述的命題」の排除を行います. そして, 論理学で扱う対象には階層を入れます. まず, 命題や函数にもならない対象を「個体 (individual)」と呼び, この個体が 0 階の論理式の型を構成します. そして, 個体のみを変項として持つ述語を第 1 階述語と呼びます. それから, 第 1 階述語を変項の値域とする述語を第 2 階述語と呼んで, 以降, 同様に $n+1$ 階述語は n 階述語を変項の値域とする述語になります. この Russell の体系は分岐的階型理論と呼ばれる複雑怪奇な構造となります.
ところで循環論法の禁止と分岐的階型理論によって循環論的な定義が使えなくなります. その結果, 幾つかの解析学の重要な結果が無効となります. 副作用はそれだけに留まらずに数学的帰納法も適用不能となります. そこで, Russell は「還元可能性公理 (Axiom of Reducibility)」を導入します. この還元可能性公理は「任意の階の命題函数には, それと同値な可述的函数が存在する」というものです[42]. この還元可能性公理は命題函数に導入した階差を本質的なくせることを主張しており, この性質から, わざわざ階を命題函数に導入した必要性がないのではないかと非難されたり, この公

[41] [33] 参照. なお, その修正案でも逆理は防げていないことが没後に証明されています.
[42] Prinpicia Mathematica の表記を用いると,$(\exists\varphi).\psi\, x. \equiv_x .\varphi! x.$ と記述されます

理の性格が他の公理と比較して論理学的なものとは言い難いことが問題視されています.
なお, Gödel は「還元可能性公理」を「集合の内包性公理」で置換えた Principia Mathematica の体系を用いて不完全性定理を証明しています. この集合の内包性公理は非可述的な性格を持った公理で, このことから伺えるように Principia Mathematica には排除した筈の非可述的な命題が裏口から入っていたのが実情です.
このように Principia Mathematica の体系で問題のない論理式を充す集合の存在を保証する公理として捉えられるものであり, Principia Mathematica の体系は実質的に集合論といえる側面を持ちます. そのこともあって, Russell 自身もこの還元公理には満足していませんでしたが, 結局, この公理を別の妥当な公理で置換えられませんでした. そのこともあって, Russell の論理主義も成功したとは言い難く, Principia Mathematica も本来の目的を達成することのできなかった失敗作と考える向きもあります[43].

しかし, Russell の Principia Mathematica の体系によって, Hilbert 達の形式主義に必要とされる道具が揃ったことになり, 一時期の中断から数学の基礎付けの問題に戻った Hilbert は, この Principia Mathematica の体系を用いて彼の形式主義を進めることになります[44].

直観主義:　その名前のとおり, 直観主義は数学的知識の基礎を「直観」に置くものです. 代表的な人物としては Brouwer が有名ですが, 古くは Kronecker や Poincaré [45]もこの考えに繋がります.

Brouwer は数学的知識の基礎を「論理」に置く論理主義, 数学の公理化で得られた超数学を研究対象とする Hilbert の形式主義に反対しています. この直観主義では厳密さを追求したために数学は非常に窮屈なものになり, 重要な数学の原理の幾つかが無効なものとなります. Hilbert が Brouwer の直観主義に反発した大きな点の一つに, Brouwer の「排中律 (law of excluded middle)」への攻撃が挙げられます. ここで排中律は「命題 P が真であるか題命 P が偽であるか何れかが成り立つ」というもので[46], これは Aristotle の形而上学にもある原理の一つです. この排中律を利用した証明の手法が「背理法 (帰謬法)」 と呼ばれる証明法です. 背理法は「命題 P が成立しない場合」に矛盾が生じることを示し, これによって「命題 P が成立する」ことを証明する手法で, 背理法を用いた証明は古くからあります. この背理法を用いた例として, [素数が無限個存在すること」を証明してみましょう:

[43]たとえば,[11] 等.
[44][44],p.20 に Principia Mathematica の体系を採用することへの言及があります.
[45]彼の著書「科学と方法」[52] に「直観」に関することが Poincaré 自身の体験も含めていろいろと書かれており, 非常に興味深く, 面白い本です.
[46]命題 P かその否定 $\neg P$ の何れかが成立し, その中間を排除することから排中律 (principle of excluded Middle) の名があります.

素数が無限個あることの証明

1. 証明すべき命題の否定「素数が無限個存在しない」を仮定します
2. 素数が無限個存在しなければ素数の最大値 M が存在します
3. 最大の素数 M 以下の素数 a, b, $c\ldots$, M の積に 1 を加えた数 $X = a\cdot b\cdot c\cdots M+1$ を作ります
4. X は M 以下の素数で割切れないので, X は最大の素数 M よりも大きな素数になり, M が最大の素数であることに矛盾します
5. 「素数が有限個存在する」が否定されたため, 「素数は無限個存在する」が真となります

このように最初に証明すべき命題を否定し (「素数が有限個」), そのことから矛盾を導いて最初の命題が偽であると結論付け, 命題 (「素数は有限個ではない」) が証明されたと主張する方法です. なお, Euclid の原論で素数が無限個存在することを, 既知の素数 $p_1,\ldots,p_n$ を使って構成した数 $p_1\times\cdots\times p_n+1$ が素数となることを示すことで, 際限なく素数が生成できるという, 上の背理法とは異なった構成的な方法で証明しています.

さて, Brouwer によると, 証明する対象が有限個であれば命題が成立するかどうかを一つ一つ検証することは可能です. ところが, 相手が無限個になると全ての対象に対して命題が成立するかどうかを検証できるとは限らないことを挙げています[47]. たとえば, Brouwer は「円周率で 0123456789 と連続して出現する個所があるか」[48]という命題を挙げています[49]. この場合, その箇所を計算して見付けるか, そのような数列が出現しないことを厳密に証明する必要があり, この命題が成立するか間違っているかどうか簡単に片付けられません. このように排中律を検証が十分に行えない無限集合に対して無制限に利用することを問題視しています.
ここで, もし排中律を排除すれば, 排中律を前提とした背理法が無効になって, 背理法を用いた証明は全て効力を失うので, 排中律に基かない別の証明方法で置換えなければなりません. もし, それができなければ直観主義に立脚した数学では使えない命題となり, 排除されますす. この直観主義の有名な犠牲者は,「有界な実数の部分集合は上界を持つ」という Weierstrass の定理です. そのため, Hilbert は排中律の攻撃を「数学から排中律を奪うことは天文学者から望遠鏡, ボクサーから拳を奪うようなものだ」[50]と大いに反発しています.
排中律の話から伺えるように直観主義は厳密であるものの, その生産性の低さもあって数学の主流にはなりませんでした. さらに Brouwer は Russell のように Principia Mathematica のような体系を作った訳ではなく, どちらかと言えば問題提起が主であり, 実際のモデルは Brouwer の弟子の Heyting が構築しています[51].
ただし, 排中律への疑問等と非常に重要な主張をしており, それらは部分的に形式主義にも取り込

[47] これに似たことを Poincaré も「科学と方法」([52]) の中で述べており, 無限の物に対する検証方法を数学的帰納法に限定しています.
[48] 近年の π の計算によると実際にその個所が存在するそうです.[56] で探してみるのはいかがでしょうか？
[49] これと似た命題として, ウィーン学派の話に出てくる「月の裏側には標高 1 万メートルの山がある」という命題もありますね.
[50] 1927 年 6 月の Hamburg におけける数学セミナーでの講演 (The foundationsof mathematics[83], p.476) での発言
[51] たとえば Heyting 代数

まれています. さらに, この考えは数理論理学の直観的論理学等に引き継がれています.

形式主義:　ここでの「形式主義」は Frege が批判した初期の安易な形式主義ではなく, Hilbert による (後期の) 形式主義を指示します. この Hilbert の形式主義は, 数学を徹底して公理化することで数学自体を形式化し, 有限回の機械的な処理で問題を解く (有限の立場) という考えが根底にあります. 戦前の日本では「**公理主義**」として紹介されています. Hilbert の形式主義は一見すると論理主義に似ています. 実際, Hilbert の最初の立場は論理主義と混同されていたようです[52]. ただし, その初期の段階でも Hilbert の形式主義は論理学から数学を導出することが目的ではなく, 数学の形式化を行なうことで数学をより強固なものにすることを目的としています. Hilbert の数学の基礎付けは 1904 年から 1914 年の 9 年程の中断を挟んで前期と後期に区分できます. そして, 後期の形式主義では Russell の Principia Mathematica の体系をその手段として採用しています.
この形式主義では論理主義や直観主義と比較して数学に加わる制約も緩いものであったために他と比較して失われる成果も少なく, その上, 形式化による作業効率の向上が望めるという, 実利的で生産性が非常に高いものであったことから主流になりました. そして, 現在の数学もこの形式主義の延長線上にあります.
さて, この形式主義を推し進める方針として, Hilbert 計画と呼ばれるものがありました. 次に, この Hilbert 計画について軽く触れておきましょう.

4.17.5　Hilbert 計画

Hilbert の構想では数学は従来の素朴な「**非形式的数学**」, 公理を使って「**形式化された数学**」と「**超数学 (Metamathematics)**」の三つで構成されます. ここで形式的数学は素朴な従来の数学に公理を導入することによって形式化して得られるもので, 逆に非形式的数学は形式的数学の解釈から得られ, 両者は密接に連環します. なお, これらの二つの数学は日常語で表記されます. これらの数学に対して超数学 (Metamathematics) は Principia Mathamatica で体系化された論理式を用いて記述され, 形式的数学の妥当性 (無矛盾性) を考察するものとなり, この超数学は所謂, 有限の立場を堅持したものになります.

超数学や形式的数学では数理論理学の論理演算のように, もはや命題の内容は問わずに機械的な処理が可能となります. このことは問題の本質を理解していない石頭の計算機でも手続をきちんと与えさえすれば処理が可能であることを意味します. これはタイプライターを猿に与えて, 猿が打ち出す出鱈目な文字の羅列の中から, やがて文学的作品が得られるかもしれないという確率的な話よりも現実的な話です. この考えが計算機の基礎にも繋っているのです[53].

実際, Hilbert は形式化を推し進めることで数学を確固たる基盤の上に載せ, 数学自体を機械的な計算処理で済ますことを目的としていました. これが「**Hilbert 計画**」と呼ばれるもので, 具体的には

[52] Poincaré の「科学と方法」には論理主義者から Hilbert 氏は破門されたとの記述があります.
[53] 計算機は第二次世界大戦中に発展していますが, 理論的にはそれよりも遥かに先行していたわけです. むしろ, 戦争がその発展を歪め, 遅らせたのかもしれません. ドイツ気触れした戦前の日本の軍部は「たたかいは創造の父, 文化の母」といった国民向けの小冊子を作っています (1934 年 10 月 2 日, 所謂, 「陸軍パンフレット事件) が実体はその逆です.

以下の三段階を実施する研究プロジェクトです. この歴史的経緯とその詳細は林 ([30]) の解説に非常に詳しく出ています.

Hilbert 計画

1. 現実の数学を形式化し, 数学の実体ではなく形式化したものを数学の実体とみなす
2. 形式系の無矛盾性を示す. これによって数学の絶対的な安全性が示される.
3. 形式系の完全性. すなわち, 任意の命題が決定可能であることを示す. これによって数学の任意の問題は常に解決ができることになる.

この計画が完了すれば数学は安全であると同時に完全なものとなる筈です. それに加えて機械的な処理も安心して行えることになるのです. この Hilbert 計画のことは同時代の高木の著書「数学雑談」([31],p.269) に「ロボット式数学」として簡単に触れており, 「**ロボットができて円滑 (無矛盾) に動くであろうかが興味のある問題だ**」と書かれています.

次の節では, この Hilbert の構想を眺めるため, 命題や述語の話をしましょう.

4.18 命題と述語

4.18.1 小史

論理学の発生

歴史上, 論理学には二つの大きな流れがあり, 一つはギリシャに源を発する西洋論理学, もう一つはインドのインド論理学です. これらの他に戦国時代の中国にも墨家による論理学の萌芽がありましたが, 残念ながら, こちらは漢代には途絶え,「墨子」[53] の経編等に断片的に残っているものの, その全体像を窺うには不十分な状態です[54].

古因明

日本への論理学の伝来は比較的早く, インド論理学の「**因明**」が仏教と共に伝来しており南都六宗 (奈良仏教) で研究されていたとのことです.

[54]中国哲学書電子化計画 (http://chinese.dsturgeon.net/index_gb.html) の「墨子」 (http://chinese.dsturgeon.net/text.pl?node=101&if=en) にて英文対訳付きで読めます. なお, この「墨子」には論理学, 幾何学, その他のいろいろな技術的な断片を含んでいるために清朝末の西洋科学の導入で大きく取り上げられています. ちなみに中国・朝鮮のような伝統的社会で新規なものを導入するにあたって, 自らの古典で似たものを「再発見」し, それから導入するという手続を踏む傾向があります. そのため, 実質的な革新運動が復古運動の衣を被ることもあります. 古くは宋明理学, 最近では清朝末期の戊戌の変法の指導者である康有為は董仲舒の公羊学を思想的な柱としています. そういえば, 明治維新も最初は王政復古でありながらも実体は徹底した文化革命であり, 北一輝の昭和維新も表向きは復古的でありながら実体は象徴的な天皇を頂く国家社会主義とハイパーモダンなものでした. もっとも 2.26 事件の青年将校達がどれだけ理解していたかは別問題です. これと比べると, 大東亜戦争開戦後の座談会「近代の超克」[12] は名前の仰々しさの割に, 中身は第一次世界大戦中の同盟国側のプロパガンダ (商人根性 v.s. 軍国主義精神) の焼き直し, 転向者の「真面目に教えてくれなかった大人が悪いからこうなった」式の反省文, そして, 周回遅れのドイツ浪漫主義の詰合せと羊頭狗肉の観があります. この毒にあたった場合, 解毒剤として夏目漱石の「我輩は猫である」(日露戦争中の著作！台所での鼠退治で「旅順椀」といったパロディも出る程) を強く薦めておきます.

ここでは古因明による推論の例を挙げておきましょう：

古因明の例

主張 (宗)　あの山は火を有する。
理由 (因)　煙を有するが故に。
実例 (喩)　煙を有するものは火を有する。竈のように。
適合 (合)　竈のように、あの山もそうである。
結論 (結)　あの山は火を有する。

古因明では「宗」,「因」,「喩」,「合」,「結」の「五段 (五分作法)」で構成されており, 宗が命題, 因が命題が成り立つ理由, 喩が具体的な事例, 合が因との適合を示し, 結が結論となります. この古因明をさらに洗練したものが「**新因明**」と呼ばれるもので, この新因明では宗 , 因, 喩, 合, 結が整理され, 宗, 因, 喩で構成される三枝作法になっています. この三枝作法と後述の Aristotle の三段論法を比較すると, 宗が結論, 因が小前提, 喩が大前提に対応しそうです. ただし, インドの論理学は真なる結論を導き出す論証の学 ([60] 参照) であり, 喩の存在から帰納的な傾向が強い論証を行っています[55].

ヘレニズム世界

一方のギリシャでは, Aristotle が名辞論理によって論理学を整理しました. のちのメガラ派 (Megaric school)[56], ストア派 (Stoic school) といった哲学諸学派の論理学には命題論理の萌芽があります. たとえば, ストア派では哲学は論理的部門, 倫理的部門と自然学的部門の三部から構成されると主張し, 哲学を生き物に譬えて, その骨や腱に論理的部門, 肉の類いを倫理的部門, 魂に自然学的部門をあてがっています[57]. そして, 論理学に関しては基本的な類 (クラス) を基体, 性質, 状態と関係の四つに切り分けています. ストア派の哲学者の Chrysippos [58]は判断について「**それ自体完結していて, それだけで否定, もしくは肯定され得るものである**」と述べ, 論証についても「**第一に何よりも言明であり, 第二に推論的, 第三に真であり, 第四にその結論が自明でないもの, 第五に自明でない結論を前提の力で見出しているものである**」としています. そして証明を要さない推論として次の五つを挙げています:

[55]桂 [13] では帰納的な推論であると述べていますが, 異論も多いようです. また, インドの論理学は弁論と討論の術としての側面もあります. この点はヘレニズムのソフィストと同様で, また, 中国の戦国時代の諸子百家も同様であったといえるでしょう. さて, インドの話に戻すと, たとえば,「屍鬼二十五話」[29] では「判っていて質問に答えないと頭が破裂するよ」と屍鬼が主人公を脅します. これと同様に, 討論でも論敵の質問に答えられなければ「頭が破裂する」程で, おまけに頭の破裂した哲学者の骨がどのように再利用されるかという解説もあるのが如何にもインド的です.

[56]メガラ派の Eubilides は「角の論」で有名です:「もし何かをなくしたのでなければ, 君はそれを持っている. だが, 君は角をなくしたことはない. したがって, 君は角を持っている」([63],p.2). なお, 五賢帝末期の頃の風刺作家 Lucianos の「にわとり」には「二つの否定は一つの肯定である」,「君には角が生えている」等の哲学談義で主人公をうんざりさせる哲学者の話があります ([70],p.121).

[57][63],p.70-71

[58]Chrysippos は 705 巻という莫大な著作を行ったとのことですが, この当時の書籍は比較的薄いものなので量的には意外と少ないかもしれません. とは言え, 現在に残った書籍はなく断片のみが残されています. 有名な与太話に, 無花果を食べさせた驢馬にワインを飲まそうとして笑い死にしたというものがあります. この Chrysippos は面白いことを主張しています. それはあらゆる種類の蛎, 紫貝, 磯巾着や多くの鳥を神が用意したのは, 我々がスープや山海の珍味を味わえるようにするためだとか. そんな訳で,「孔雀はその尾が美しいことから尾のために生じた」と述べているのも不思議はありませんね ([63],p.80)

Chrysippos による無証明推論

1.	「A ならば B」と「A である」から「B である」を推論	⇔	$A \to B, A \vdash B$
2.	「A ならば B」と「B でない」から「A でない」を推論	⇔	$A \to B, \neg B \vdash \neg A$
3.	「(A かつ B) ではない」と「A である」から「B ではない」を推論	⇔	$\neg(A \wedge B), A \vdash \neg B$
4.	「A か B のどちらか一方である」と「A である」から「B でない」を推論	⇔	$A \underline{\vee} B, A \vdash B$
5.	「A または B である」と「B でない」から「A である」を推論	⇔	$A \vee B, \neg A \vdash B$

ここで最初の 1. は「**前提肯定 (Modus Ponens, MP とも略記)**」と呼ばれる推論規則で, Frege が彼の論理学の推論規則として取り入れた唯一のものです.

このようにメガラ・ストア派の論理学は, のちの Frege を思い起させるものを多く持っていますが, 断片的にしか伝わっていないのが残念な所です[59].

ローマ帝国の最盛期には, ストア派, 新プラトン主義 (Neoplatonism)[60]の哲学, グノーシス (Gnosis) 等が盛んになります. このグノーシスの特徴の一つに星辰崇拝の否定, 創造主 (デミウルゴス) の悪しき意図による世界の創造という悲観的世界観があります. 帝政ローマでは死んだ皇帝は天に登り神となります[61]. また, ギリシャやローマの神話では結末で英雄等が天上に上げられて星座になる例が多々ありますが, それと比べても非常に違和感があります. この創造主については過酷なローマ支配を受けた属州の事情に加え, 高圧的な神々を信仰するバビロニア等の所謂オリエントの宗教の影響もあるでしょう. 実際, 帝政末期はキリスト教以外のオリエント由来の宗教としてキリスト教と国教の座を争ったイラン由来のミトラス教 (Mithraism), マリア崇拝等でその痕跡を地中海周辺で未だに残すイシス (Isis) 信仰が流布しています. たとえば, 「黄金のろば」[1] のイシスやオシリス (Osiris) の秘教的な箇所はオリエント神話そのものです. たとえば, ギルガメッシュの冥界行きの話には太陽が通る地下道の話がありますが, 「黄金のロバ」にもその地下道のことを仄めかす記述があります. また, グノーシスの特異なものにヘルメス主義 (Hermetism) があります. こちらは Poimandres が有名ですが, ここでは神の子としての人間が地上に向う際に天上の星の支配を如何に受けるようになって, 最後にフュシス (本性, $\phi\upsilon\sigma\iota\varsigma$) との愛欲によってフュシスに捉えられ, 堕落したかといったことが書かれています ([26]). この Hermetism は, のちのヨーロッパの錬金術や占星術にも影響を与えています. ヘルメス文書と呼ばれる一連の著者とされた Hermes は Hermes trismegistus(三重に偉大なヘルメス) として知られ, 「賢者の石」を実際にえた人物として錬金術で重要な位置を占めます.

[59] ここで引用した断片は [63],p.88-92, メガラ・ストア派の論理学の概要については [62] 参照

[60] 新プラトン主義は Plotinus によるプラトン主義の発展を区分するために 19 世紀初頭のヨーロッパにて導入された言葉です. 基本的には Plotinus 以降を指します.

[61] そのためには元老院での議決が必要で, そのときに神として天上に登って行くさまを見たとの目撃談も必要だったようです (Seneca の Apocolocyntosis(神君クラウディウスのひょうたん化)[28][97] を参照)

この古代ギリシャの合理主義から帝政末期の神秘主義への傾倒については, まず, 古代ギリシャの宗教が都市国家宗教であり, Alexander 大王とその後継者による国家の出現によって都市国家が存在意義を失ったこと, 個人主義の時代の到来と共に宗教的な空白が生じたこと, 理知的で合理的な哲学は一般大衆にとっては捉え難いものであったこと, さらに理知的なギリシャ哲学は死後の世界について明確な答が得られないのに対し, ゾロアスター教やエジプトの宗教, さらにはキリスト教等のオリエントの宗教は逆に積極的に答えるものであったことが挙げられるでしょう. やがて哲学も, シリアの占星術, バビロニア, エジプトやイランの宗教の影響を受けて神秘的傾向を強めて行くことになり, その宗教を補完するものとなってゆきます ([71], 「嘘つき, または懐疑者」の解説を参照). 功利的, 実利的な明治に構築された国家神道が崩壊した戦後期に, その間隙を突くようにいろいろな新興宗教が流行り, やがてはスピリチュアルと称するオカルトが半ば公認される現在の日本に似ていなくもないですね[62].

殊に初期の (正統派) キリスト教は Plotinus の新プラトン主義の影響を受けて, その神学を構成して行きます. これも新プラトン主義の絶対的な一者からの万物の流出を説く流出論による解釈があってはじめてキリスト教やイスラーム教といったセム的一神教への受容が容易になったと言われています. その一方で Plato や Aristotle の思想も新プラトン主義的な夾雑物を多分に含んで伝播する大きな要因となります. なお, キリスト教は Hypatia の虐殺に見られるように古代文明を破壊したイメージが強くありますが, その一方で古代ローマの宇宙観をそのまま継承した側面も持っています ([26]).

イスラーム世界

ローマ帝国の衰退以降にヘレニズム文明を伝承したのがイスラーム世界です. このイスラームにも独特の論理学がありますが, これは必要によって生じたものです. まず, イスラーム法ではクルアーン (コーラン) とハディース (預言者 Muḥammad の言行録) を基にしています. ところでクルアーン (コーラン) は体系的な経典では決してなく, それどころか矛盾が多いことでも知られています[63]. それに加え, イスラーム世界の拡大, 文化の発達等の要因でクルアーンやハディースに言及されていない事物に対して, それが「**宗教的に禁止されたもの**」, すなわち「**ハラーム (harām)**」であるかどうかを判断しなければならないことが数多く生じました.

ここでは一例を挙げておきましょう. まず, 葡萄酒は預言者によって宗教的に禁止されていますが, 他の酒, たとえば, ラム酒に関しては一切言及がありません. したがって, この酒がハラームであるかどうかはクルアーンやハディースだけでは判断できません. そこで「**キャース (qyās)**」と呼ばれる三段論法が用いられます:

キャースの例 ([5])

(大前提)　発酵させて造り、人を酩酊させる飲料は harām である。
(小前提)　ラム酒は発酵させて造り、人を酩酊させる飲料である。
(結論)　　故に、ラム酒は harām である。

[62] この時期の風刺作家 Lucianos の短編「逃亡者」 [69] に襤褸を纏ってストア派等の哲学者の真似をして俗物の金持に集る輩の話があります. ちなみに, 五賢帝の最後の皇帝 Marcus Aurelius Antoninus もストア派の哲学者としても知られています.

[63] これはクルアーン (コーラン) の成立事情も関係することです.

キャースの大前提は何でも良いものではありません. 大前提を求めるためには帰納法を用いなければならず, さらに一般的に根拠があるものでなければ大前提としては使えないのです. たとえば, イスラーム神学の超合理主義派の「**ムアタズィラ (Mu'tahzilah)**」では三段論法を駆使して, ともすれば異端的な結論を導き出していたと言われます.
ちなみにイスラム思想では真理を「**ハック (Ḥaqq)**」と呼びます. そして, 哲学者 al-Kindi 以降, 神のこともハックとも呼びます. そんな訳で日本語で「ハックする」と言えば「真理を探求すること」, 「ハッカー」とは「真理の探求者」であると駄洒落にできますね[64].

閑話休題(さて), ヘレニズム文明のイスラームへの導入ではシリアのキリスト教徒が大いに活躍しています. まず最初にシリアのキリスト教徒の医者達が医学書や自然科学の文献をアラビア語に翻訳しています. やがてアッバース朝[65]で非常に合理主義的なイスラム神学の一派であるムアタズィラ (Mu'tahzilah) が公認され, 学問好きのカリフ (教皇)[66]の保護もあって, 「知恵の館」(Bayt Ul-Hikma) で組織的にギリシャ哲学, 自然科学, 医学等の文献をアラビア語へと非常に精密な翻訳が行なわれます[67].
ここでアッバース朝公認のムアタズィラは自らを「正義と神の唯一性の提唱者」と自称し, その合理性も徹底したものでしたが, このムアタズィラの徹底した合理主義は al-Ghazali に非難され, のちの「正統派」によってその著作が根絶させられるという憂き目にあっています. このムアタズィラには次の特徴があるそうです ([5],p.54-65. 参照):

- 予定論を認めない.
- ムハマンドによる執り成しを認めない.
- 神を人格化して表現することを認めない等

最初の「**予定論**」は, 神によって人間の運命は完全に定められていると主張するものです. この予定論を認めない理由は, 正義の神が人間に不正義を行わせる筈がなく, だから人間の行為は全て人間に帰着するというものです. さらに人間は神と並んで第二の創造主となるという主張もあり, Göthe の Faust の思想に似てなくもありません. そして, ムアタズィラによると, 善悪は理性に照して判断されるものであるために, 神ですら理性の規準に従わなければならないと主張しています. なお, 予定論も受け身であれば何かと問題が出ますが, これを積極的に捉えたのがキリスト教のカルバン派等のプロテスタントで, 彼等が商業等で社会的な成功を収める原動力の一つであったと言われています.
次の「**執り成し**」は, イスラム教徒であれば最後の審判でムハマンドが信徒のために神に執り成しを行って罪の軽減が計られるというものです. 人間が第二の創造主であるとすれば, 全ては自分に責任があり, 悪行故に地獄の業火で焼かれるのは当然であるということです.

---

[64]だからと言って, 過労で「即身成仏」, 乃至, 「神と合一」することは神秘思想とは無関係です.
[65]千夜一夜物語の舞台となる王朝です.
[66]たとえば, マームーン (al-Ma'mun)
[67]学問的なこと以外でローマの都市文明を受け継ぎ, 発展させたのもイスラーム世界です. たとえばローマの公衆浴場はイスラーム世界でもハンマーム/ハマムとして引き継がれ, 江戸の浴場のように都市に居住する市民にとってなくてはならないものとなっています. 一方で, ヨーロッパでは中世以降, 悪徳の巣として消えてしまいますが, これが 19 世紀に入って導入されています. その理由に富国強兵と国民衛生の問題 [15] といった見方もあります.

最後の神の人格表現の否定は, クルアーン (コーラン) に描かれた神の人間的表現を比喩として捉えるものです. 知識や理性の神は哲学者にとっては望ましいことであるにしても, より具体的なものを望む一般の信者にとっては非常に迷惑なことでしょう. この点はストア派の哲学で「**知識 = 神**」と見做したことと非常に近いものです. ただし, この考えを推し進めると「**哲学こそが全て, 宗教は一般大衆向けの幼稚な哲学**」となり, これに反対する側は原理主義的なものになりかねません. 実際, ムアタズィラはその合理性故に非難され, 著作が根絶されてしまいます[68].
さて, イスラーム世界の哲学者で最も著名な人物は, Ibn Sina と Ibn Rushd です. Ibn Sina はヨーロッパでは Avicenna と呼ばれ, 彼がイスラーム哲学の頂点とも言われています. Avicenna は新プラトニズム主義に立脚した哲学者でしたが同時に医学者としても著名です. なお, 同時代のイスラーム世界の学者からは彼の死後に「自らの哲学で自らの魂を救えず, 自らの医術で自らの肉体を救えず」とも評されています.
Rushd はヨーロッパでは Averroës と呼ばれています. 彼は Aristotle 注釈で著名であり, Aristotle の原典に戻れと主張したように従来の Aristotle 解釈が新プラトン主義に汚染されたものであることを知っており, そのような夾雑物の排除を目指していました. ただし, 新プラトン主義的なものの排除は非常に困難でした[69].
一方のヨーロッパでは東西交易の活性化になったこともあり, イスラーム哲学を経由することで Aristotle 等の古代ギリシャ哲学が再導入されています. さらにイスラーム経由の新プラトン主義やヘルメス主義は文芸復興期のヨーロッパでは強い影響を持ちます.

4.18.2　伝統的論理学について

命題

論理学で扱う命題とは「人間は死すべき生き物である」,「ソクラテスは人間である」,「4 は 2^2 に等しい」や「1 は 10 よりも大である」のように, ある事柄について真か偽か判定されうる文のことです. そして,Aristotle の論理学では, この命題を「**主語 (subject)**」,「**述語 (predicate)**」の二項, そして, これらの二項を繋ぐ「**繋辞 (copula)**」に分けて考え, 論証の形式的な正しさに重点が置かれていました.

たとえば, 命題を「ソクラテスは人間である」とすると, 主語が「ソクラテス」, 述語が「人間」, 繋辞が主語と述語を繋ぐ,「…は…である」になります. ここで命題の解釈として内包的と外延的の二通りの解釈があります. これは「全ての烏は黒い鳥である」という命題を解釈する場合,「烏」の属性として「黒い鳥」という概念が付属すると解釈するのが「**内包的解釈**」,「烏」の外延 (類, クラス) が「黒い鳥」という外延に含まれるという解釈が「**外延的解釈**」になります. ちなみに, Aristotle は内包的解釈で, そのこともあって伝統的西洋哲学では内包的解釈が主流でした. 現在, 集合の定義

[68] 徹底した合理主義のムアタズィラがある一方で, 神との合一 (我は汝, 汝は我) を目指すスーフィズムに代表される神秘主義もあります. このスーフィズムに関しては仏教の影響も指摘されています. 特に戦前 満州に住んでいた人で回教徒のことを「フィフイ」と呼んでいる人もいますが, この呼び名もスーフィに由来します. なお, ダルビッシュはスーフィの修行僧 (乞食僧) に由来します.
[69] Dante の「神曲」では, Sina, Rushd, Plato, さらには Aristotle 達の未洗礼の偉人が送られる Limbo(リンボ) にいます. このことからも著名さの度合いが判るかと思います.

で述語を用いて表記する方法が「**内包的定義**」[70], 要素を列挙する方法が「**外延的定義**」[71]と呼ばれます.

命題の判断

「**命題の判断**」とは, 命題を構成する主語と述語の持つ概念が一致するかどうかを断定することです. この判断には「肯定判断」,「否定判断」,「全称判断」と「特称判断」といった代表的な判断があります:

命題の判断

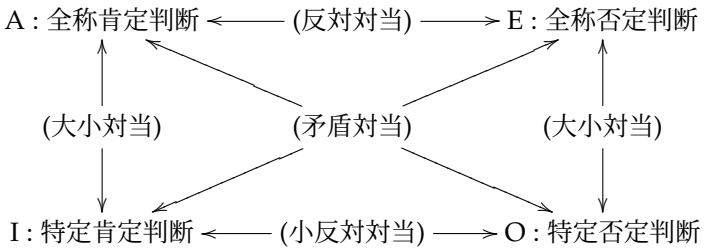
肯定判断：	命題「A は B である」のように, あるもの (A) に対してあること (B) を認める判断
否定判断：	命題「A は B ではない」のように, あるもの (A) に対してあること (B) を否定する判断
全称判断：	命題「全ての A は B である」のような判断
特称判断：	命題「全ての A は B である」のような判断

そして, 肯定的判断, 否定的判断に対して全称判断, 特称判断を組合せて, 次の「A:全称肯定判断 (Universal Affirmative)」,「E:全称否定判断 (Universal Negative)」,「I:特称肯定判断 (Particular Affirmative)」,「O:特称否定判断 (Particular Negative)」が得られます:

A,E,I,O

(A)	全称肯定判断：	「すべての A は B である」
(E)	全称否定判断：	「いかなる A も B ではない」
(I)	特称肯定判断：	「ある A は B である」
(O)	特称否定判断：	「ある A は B ではない」

ここで「A」,「E」,「I」,「O」はラテン語の動詞 AFFIRMO(我は肯定する) と NEGO(我は否定する) に由来するものです. さらに, これらの「AEIO」に対しては, 次の「対当関係表 (SQUARE)」が得られます:



この対当関係から命題の真偽について次の推論が可能です:

[70][内包的定義の例:$\{x|x$ は骰子の目の数 $\}$
[71]外延的定義の例:$\{1,2,3,4,5,6\}$

- 大小対当 (A と I,E と O)
    1. 全称が真の場合, 必ず特称も真
    2. 特称が偽の場合, 必ず全称も偽
    3. 全称が偽の場合, 特称の真偽は不定
    4. 特称が真の場合, 全称の真偽は不定
- 反対対当 (A と E)
    5. 何れか一方が真の場合, 必ず他方は偽
    6. 何れか一方が偽の場合, 他方の真偽は不定
- 小反対対当 (I と O)
    7. 何れか一方が偽の場合, 他方は必ず偽
    8. 何れか一方が真の場合, 他方の真偽は不定
- 矛盾対当 (A と O,E と I)
    9. 何れか一方が真の場合, 他方は必ず偽
    10. 何れか一方が偽の場合, 他方は必ず真

ここで伝統的論理学で扱う命題には「存在含意 (external import)」 が含まれています. 伝統的論理学では命題を主語と述語と繋辞に分解しますが, ここで主語となる対象の存在を前提にして命題を扱っています. たとえば「A is B」であれば, 主語が「A」, 述語が「B」, 連辞が「is」になりますが, それに加え, 伝統的論理学では主語の「A」には「A exists」をあらかじめ含めて考えています[72]. そのため,「あるドラゴンは火を吹く」といった命令を扱おうにもドラゴンの存在が肯定されない限り, この命題は扱えないために真偽が不定になります. これは本当に正しいかどうか判らない仮説を基に推論を行うことができないことも同時に意味し, このために Aristotle の論理学では「仮説」が立てられず, 科学の発展では Idea を背景に持つを Plato を哲学的な背景 (所謂, プラトニズム) とすることに繋がります. なお, Aristotle が師 Plato のイデア論に批判的で, このように現実的であったのは, 彼が医者の息子であったことも大きかったといわれています.

推論

推論の形式で代表的なものとして「**三段論法 (Syllogism)**」があります. 三段論法は,「大前提」,「小前提」と「結論」の「三つの命題」から成る「推論の規則」です. さらに三段論法には「定言三段論法」,「**仮言三段論法**」,「**選言三段論法**」と「**両刀論法 (ディレンマ (dilemma))**」の四種類があります. これらの非常に安易な例を次に示しておきましょう:

[72] be 動詞には「存在」の意味も含まれていますね

定言三段論法の例

大前提: 人間は死すべきものである
小前提: ソクラテスは人間である
結論: 故にソクラテスは死すべきものである

仮言三段論法の例

大前提: ソクラテスは人間である
小前提: 人間は死すべきものである
結論: 故にソクラテスは死すべきものである

選言三段論法の例

大前提: 禿であるかフサフサであるか
小前提: フサフサではない
結論: 故に禿である

両刀論法の例

大前提: 雨なら傘が要る, 且つ, 雪なら傘が要る
小前提: 雨ないし雪だ
結論: 傘が要る

Aristotle は「排中律」と「矛盾律」を「形而上学」で述べています (「形而上学」[3],p.120-152). ここで「排中律」は「命題 P あるいは, 命題 P の否定の何れかが成り立つ」というもので, 「矛盾律」は「命題 P と, 命題 P の否定が同時に成立することはない」というものです.

この Aristotle の影響は絶大で, のちに Kant が「Aristotle 以来進歩もなければ後退もない, いわば完成された学問」と (伝統的) 論理学のことを呼んだとのことですが, 実際は伝統的論理学で 19 世紀末に至るまで満足な解決ができなかった問題, たとえば, 多重量化詞を含む推論もあったのです ([4] 参照).

4.18.3 Leibniz

Aristotle 以降で興味深い人は Leibniz です. Leibniz は最後の万能の天才と言える人で「普遍言語」と「普遍記号」を考えています. まず, 「普遍言語」は日常言語の曖昧さを防ぐために命題に自然数を対応させ, 「A かつ B」をそれらの命題に対応する「素数の積」として表現することを考えています[73]. たとえば, 命題 A を「動物である」, 命題 B を「理性的」であるとすれば, 命題「理性的な動物」である人間は「理性的, かつ, 動物」なので「$2 \cdot 3 = 6$」で表現されます. 同様の考察で Leibniz は猿が 10 であると述べ, このことから「人間と猿が異なる」と結論付けています. 逆に, 6 と 10 に

[73] Gödel 数みたいですね

は共通因子として 2 がありますが, これは人間も猿も共に数 2 で表現される「動物」だからです[74]. なお, 命題と数値を結び付けることは Pythagoras 学派でもやっています. たとえば, 結婚を表現する数 5 は女を表現する数 2 と男を表現する数 3 との和というものですが, こちらは数秘術的なものであって, Leibniz の明快さとは異なる代物です. さらに Leibniz は命題 A に対して '$A + A = A$' (Leibniz の表記では '$A + A \infty A$') と後述の Boole に 150 年も先立って述べています ([96],[6]).

Leibniz といえば現在では微分記号 "dx" や積分記号 "$\int$" で知られていますが, Newton の表記 (たとえば, 函数 f の微分は "$\dot{f}$") と比較して明瞭なことは, 彼が普遍言語やそれを表現し得る記号に関する考察があったからできたこととも言われています. なお, イギリスでは Newton 流の表記が長く用いられており, Leibniz 流儀の表記が導入されたのは, Herschel, Babbage と Peacock の解析協会 (Analtical Society)[75]の労力によるものですが, この Newton 流儀の微分積分を使い続けたことが 19 世紀初頭にイギリスが大陸側の数学と比較して大きく遅れる原因となったと言われています.

その他に Leibniz は歯車式の計算機を作成したり, 二進法による数値の表現を考案しています. その点では現在の計算機の先駆者としての側面も持っています. ただし, 二進法に関しては宗教的神秘主義めいた考えさえ持っていました. たとえば, 数字の 7 については天地創造が 7 日で行われたことと, 二進法で 7 が 111 と表記されることを三位一体と結び付けています ([45],p.207). その点では Pythagoras の輩とも似てなくもありませんね.

ところで Leibniz は自らの学派を構成することもなく, さまざまな思索を膨大なメモに残したため, 彼の再評価は 19 世紀末から始まります[76].

4.18.4　19 世紀の論理学の進展

19 世紀に入ると, イギリスで集合を用いて論理式の定式化を行った De Morgan, 0 と 1 を使って定式化を行った Boole([74],[75]), Venn 図で知られる Venn が現われます.

De Morgan の表記

まず, De Morgan [77]は論理式に集合を導入し, その際に全体集合を導入しています. それから, 彼独自の記号を導入しますが, その記号を使って三段論法の「AEIO」がどのように表現されるかを見てみましょう [96]:

---

[74]Swift の「ガリバー旅行記」にはラピュタ国の学者の言語の研究が紹介されています. これは当時の普遍言語に対する皮肉なのでしょう. さらに, このラピュタ国では Newton も風刺されています. なお, 計算機内部の番地の処理で, Intel の「little-endian」と Motrola の「big-endian」という名称は小人国の戦争の原因 (卵を細い方から先に食べるか, 太い方から食べるか) に由来してつけられたものです.

[75]現在の Cambridge Philosophical Society. なお, 学生の冗談で協会の目的は '大学の dot-世代への純正 d-主義原理'('the principle of pure d-sim as opposed to the dot-age of the university') の宣伝だと言われていたそうです. ちなみに Herschel は天文学者, Babbage は解析機関で有名ですね.

[76]Voltaire の哲学的コント「カンディード」[8] では, Voltaire の誤解も含めて徹底して Leibniz の哲学 (楽観主義や予定調和, 「この最善なる可能世界においては, あらゆる物事はみな最善である」) を茶化しています.

[77]この De Morgan は最初の女流プログラマと呼ばれる Ada の数学の家庭教師だったそうです.

De Morgan による AEIO の表記

(A)	「全ての A は B である」	$\Leftrightarrow$	$A)B$
(E)	「全ての A は B ではない」	$\Leftrightarrow$	A, B
(I)	「ある A は B である」	$\Leftrightarrow$	AB
(O)	「ある A は B ではない」	$\Leftrightarrow$	$A : B$

De Morgan の記号に否定記号はありませんが, 命題 A の否定を小文字を用いて a と表記します. そして,「PQ の否定は q, p」,「P, Q の否定は pq」であると述べています. これが現在,「De Morgan の法則」と呼ばれているものです. この意味を現代の集合の記号を用いて書けば次のものになります:

De Morgan の法則

1. $(A \cap B)^c = A^c \cup B^c$
2. $(A \cup B)^c = A^c \cap B^c$

このように (E) 全称肯定判断と (I) 特称否定判断は否定によって互いに移り会えることが判ります[78].

Boole による論理学の代数化

Boole[79]は初めて論理式の代数的処理の体系を創った人ですが, どちらかと云えば, 論理学の冒険者としての側面が多分にあります. この Boole は論理式に対し, その論理式を充す集合を考え, この集合に対して演算を行っています. この演算は集合の和 “+” と積 “×”, 補集合 “−” の三種類になります. また, 論理式を ‘X’ とするときに ‘X’ を充す対象を「elective symbol」と呼ぶ小文字の x で表現し, この小文字に対して演算を行います. そして, 特別な記号として 1 と 0 を導入します. ここで 1 は全体集合を表現し, Boole の代数処理では積の単位元としての働きをします. これに対して 0 はそのような命題が成立しないことを示し, 和の単位元としての働きをします. なお, $1 \neq 0$ は明確に述べていないものの, 実際の計算では $1 \neq 0$ として処理をしています.
次に Boole による基本的な論理式の基本的な代数化を示しておきましょう:

Boole による命題の代数化

- $x1$, すなわち x は X を充すこと
- $y1$, すなわち y は Y を充すこと
- $xy1$ すなわち,xy は X を充し, 且つ,Y も充すこと
- $1 - x$ は X でないこと
- $x(1 - y)$ は X を充し, 且つ, Y を満さないこと
- $(1 - x)(1 - y)$ は X を充さず, 且つ, Y を充さないこと

[78] 14 世紀には既に知られていたことだそうです (Willians of Ockham)([96], [84]).
[79] Boolean でお馴染の Boole です. 何故か Haskell で真理値は “Boole” 型ではなく “Bool” 型なのです.

さらに $x+y$ を集合 X と集合 Y の和集合とし, この和と積に対して次の性質があると Boole は述べています:

Boole による演算子の性質

可換性:	$x+y=y+x$	$xy=yx$
結合律:	$(x+y)+z=x+(y+z)$	$(xy)z=x(yz)$
単位元:	$x+0=x$	$x1=x$
分配律:	$x(y+z)=xy+xz$	
冪の性質:	$x^n=x$, ただし,n は 1 以上の正整数	

このように和と積に対しては分配律が成立し, 積に対しては可換であると述べています. この二番目の可換性の説明で, Boole は同じ意味を持つ日常語の「善良で賢い男」と「賢くて善良な男」を挙げています. すなわち, [善良である」という命題を X , 「賢い」という命題を Y とすれば「善良で賢い」は xy となり, 「賢くて善良」は yx となります. それから, これらの命題の意味を考えると '$xy=yx$', すなわち, 積は可換となるのです. そして, '$x^2=x$' から通常の式の変形で '$x(1-x)=0$' が得られます. この式は「X かつ X でないことはありえない」と読めます. このことから Boole による体系が 1 を真, 0 を偽としていることが理解できるでしょう.

さて, 定言三段論法で現われた次の命題はどのように表記されるでしょうか？

Boole による AEIO の表記

(A)	「全ての Y の元は X の元である」	$\Leftrightarrow$	$y=vx$
(E)	「全ての Y の元は X の元ではない」	$\Leftrightarrow$	$y=v(1-x)$
(I)	「ある Y の元は X の元である」	$\Leftrightarrow$	$vy=v'x$
(O)	「ある Y の元は X の元ではない」	$\Leftrightarrow$	$vy=v'(1-x)$

最初の全称肯定判断は「Y ならば X」と読み替えられます. Boole の考えでは, ある時点で y が, ある x となることを意味しており, そのために '$Y\subset X$' という包含関係が生じます. この「ある x 」に対しては集合記号 V を導入して vx と表現し, '$y=vx$' が求める代数式になります. この集合記号 V は命題を表現する集合 X や Y と同様の性質 '$v^2=v$' や '$v(1-v)=0$' を充すものです. このように正体不明な集合が突如出ているのが気持ち悪いことと, その考え方に必然的に「時間」が入っています. ただし, 命題は真か偽の何れかに判断可能なものとして考えています.

二番目の全称否定判断は「X ではない」を $1-x$ と表現し, 全称肯定判断の定式化から '$y=v(1-x)$' となることが判ります. 三番目の特称肯定判断は「ある Y 」が「ある X 」に一致すると言い換えられるために '$vy=v'x$' と表現されます. 最後の特称否定判断は「ある非 X の元である」と言い換えることで '$vy=v'(1-x)$' となることが判ります.

このように命題の代数的な処理はある程度できますが, この体系で三段論法が上手く表現できるかと言えば, Frege が「ブールの論理計算と概念記法」([48]) で論じているように代数式と日常語が入り交じった式になってしまいます. さらに '$x^n=x$' や '$x+\cdots+x=x$' となる点, さらには 0/0 や

1/0 といった表記が出ることとや通常の演算とは全く別の演算となるのが厄介な点です. たとえば, Boole の挙げた例ですが, ユダヤ教では清浄な動物を「蹄が割れていて, 反芻する動物」としています. ここで命題 'X' を「清浄な動物」, 命題 'Y' を「蹄が割れている動物」, 命題 'Z' を「反芻する動物」とした場合, 清浄な動物の定義は '$x = yz$' と記述されます. ここで普通の式ならば '$z = x/y$' とできますね. ではこの '$z = x/y$' はどのような意味を持つのでしょうか？ とは言え, この Boole による命題の代数的処理は利用価値の非常に高いもので, Boole の成果を基に現在の Boole 代数と呼ばれる体系が構築され, 計算機で盛んに活用されていることは言うまでもないでしょう.

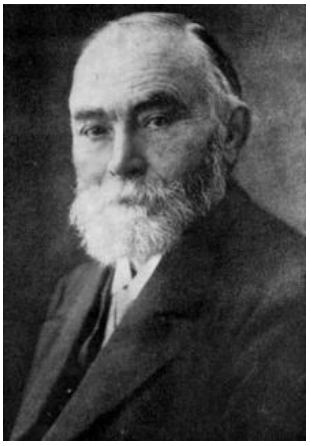
論理主義へ

1878 年になると Frege が「概念記法」([48]) にて「述語 (函数)」という概念を導入し, 述語論理を彼独自の表記で構築します. Frege は従来の命題を主語と述語といった関係ではなく, その命題の判断に重きを置いて命題を函数と変項で表現したのです. たとえば「水素ガスは炭酸ガスよりも軽い」という命題を Frege は主語を「水素ガス」, 述語を「炭素ガスよりも軽い」といった分析ではなく '軽い (水素ガス, 炭素ガス)' のように命題を変項を持つ函数として表現したのです. 次の §4.19 で Frege による命題の形式化を眺めてみましょう.

4.19 Frege の概念記法

4.19.1 概念記法 (Begriffsschrift) の概要

ここでは Frege の概念記法 (Begriffsschrift) の解説をします. Frege の著作「概念記法」([48]),「算術の基礎」([49]) と「算術の基本法則」([50]) は非常に面白く, 論理主義を素材から一つづつ造り上げ, それから, 壮麗な大聖堂を構築する凄さ, そして, 明解さがあります[80].
Frege の命題の表記は非常に独特のものです. 現在, 用いられている Peano 流儀の線的な論理式の表記方法とは異なり, 平面的な図式で命題を表現しています.


Gottlob Frege(1848-1925)

概念記法を構成する要素

概念記法では, 命題 A, あるいは「A という命題がある」ということを '$\_\_\_\_A$' と表記します[81]. この図式の '$\_\_\_\_$' は「概念記法」([48]) では「**内容線**」と呼ぶ複数の命題を繋ぐ構成要素です. ただし,「算術の基本法則」([50]) では「**水平線**」と呼ばれて函数を構成する作用素としての性質を強めていま

[80] ありがたいことに, 現在, Frege の全集が読めるのは日本だけです！
[81] Frege の概念記法の表記に begriff.sty[133] を利用しています.TeXlib にも含まれていますが, http://www.ctan.org/tex-archive/macros/latex/contrib/begriff/からも入手できます.

す. たとえば ‘ξ’ や ‘1’ は表示と呼ばれる対象で, 論理式では
ありませんが, 水平線を追加した ‘——ξ’ は変項 ξ の値が真
の場合に真を返し, その他の場合は偽を返す函数として扱います. このように, 水平線 ‘——’ は S 式を新たに構築する LISP の括弧 “()” に似た作用素になります.
また, この水平線の重要な性質として「**融合**」と呼ばれる性質があります. これは, ——(—A) と ——A が同値であるというもので, これによって, 論理式を枝で分けたり, 函数を論理式で置換することが自由自在に行えるのです[82].
次に,「命題 A が判断できる」ということを水平線線の左側に「**判断線**」と呼ぶ記号 ‘⊢’ を導入し, ‘⊢——A’ と表記します. この判断線によって, その右側に置かれた命題の真偽が判断できることが明瞭になります. なお, 現在の数理論理学で論理式が証明可能であることを表す記号 “⊢” は, この判断線に由来します. ちなみに, この記号 “⊢” に似た記号 “⊨” もありますが, この記号は A を仮定することで常に B となるときに ‘A ⊨ B’ と表記します.
「**命題の否定**」は内容線の中程に短い縦棒を追加します. たとえば, ‘——A’ の否定は ‘—┬—A’ になります. この表記で短い縦棒 ‘|’ が命題の否定を表現し, 縦棒の右側の横棒 ‘(——)’ が命題 A の内容線, 縦棒の左側の横棒 ‘(——)’ が命題 A を否定した命題 ‘—┬—A’ の内容線となります. また, ‘—┬—(—A’ や ‘——(—┬—A)’ は融合によって ‘—┬—A’ となります. なお, この否定の表記から現在の論理学で用いられている「**否定記号**」“¬” が出ています.

次に, Frege は概念の従属関係を含む諸関係を表記するために記号を導入します. まず, 命題 A と命題 B が与えられたときに次の四つの可能性があります:

1. A が肯定され, かつ, B が肯定される
2. A が肯定され, かつ, B が否定される
3. A が否定され, かつ, B が肯定される
4. A が否定され, かつ, B が否定される

さて, Frege は三番目のことを排除し, 残りの三つ内の一つが生じるという判断を意味する表記として ‘——B’ を導入しています.

この図式の左側最初の横棒 ‘(——)’ が命題全体の「**水平線**」, そして, 縦棒 ‘(|)’ を「**条件線**」と呼び, 各命題の左にある横棒 ‘(——)’ が各命題の水平線となります.

[82] Frege は LISP 出現前の Lisper ではないかと思ってしまいます. なお, 私は LISP の S 式や概念記法の式を見ていると作曲家の Mahler の次の発言を思い出します:「植物において一枚の葉という基本形態から花や幾千という枝や葉を茂らせた樹木が育つように, また, 人間の頭部が脊椎の発達した形であるのと同様, 声楽曲の純粋な書法は大管弦楽曲の複雑化した声部のからみに至るまで貫かれていなければなりません」([18],p.278)

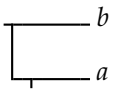
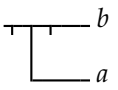
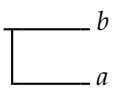
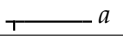
この図式 ‘ ’ は「A ならば B」を意味し, 現代の表記では, ‘$A \supset B$’ あるいは ‘$A \to B$’ となります. この ‘$A \to B$’ は ‘$\neg A \vee B$’ と同値になります. このように論理学の「A ならば B」は日常で用いる「A ならば B」とは意味合いが少々異なることに注意して下さい.

さて, この Frege の表記を用いると「A または B」は ‘ ’,「A かつ B」は ‘ ’ と表記されます. なお, 現在の論理学の表記では,「A または B」は ‘$A \vee B$’,「A かつ B」は ‘$A \wedge B$’ となります.

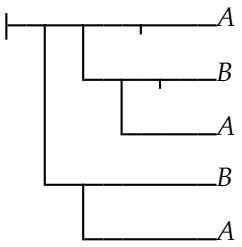
現在の表記との比較

ここで比較のために現在の表記, Peano の表記, そして Frege の概念記法による表記を表に纏めておきましょう:

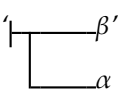
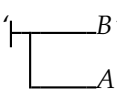
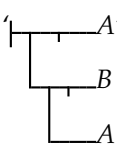
現在の表記,Peano の表記と概念記法

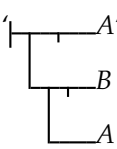
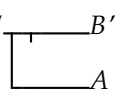
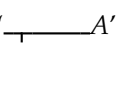
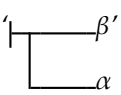
	現在の表記	Peano の表記	Frege の概念記法
選言:	$a \vee b$	$a \cup b$	
連言:	$a \wedge b$	$a \cap b$ 又は ab	
含意:	$a \supset b$ 又は $a \to b$	$a \supset b$	
否定:	$\neg\, a$ 又は $\bar{a}$ 又は $\sim a$	$-a$	

現在の表記が Peano 流儀の系統であることが判りますね. そして, 概念記法の方が見通しが悪いように見えませんか？ そこで, もう少し複雑な命題になるとどうなるか見てみましょう. たとえば, ‘$(A \to B) \to ((A \to \neg B) \to \neg A)$’ は概念記法では



となりますが, この図式からは, この論理式の構造が明瞭に判りますね.

つまり, 基本となる図式 ‘  ’ を基に, その α に ‘  ’, β に ‘  ’ を代入することで

得られること. そして, ‘  ’ 自体がまた, ‘  ’ と ‘  ’ を基本的な図式 ‘  ’

に代入することで得られるという事実が 2 次元的に表示されているのです.

一方で, この論理式を Peano の表記で記述すると:

$$a \supset b.\supset : a \supset - b.\supset - a$$

となります. Peano の本来の表記では括弧を使わずに区切記号 “.”, “:”, “∴”, “::” を使います これらの区切記号の使い分けは最初の括弧の区切に記号 “.”, 次の括弧に記号 “:” と点を増やして行きます. この方法は漢文の返り点の要領, たとえば,「在二伯楽一」で「伯楽在レ」と読ませる方法に似ているのが面白いことでしょう. このように Peano の表記は Frege の概念記法と比較すると印刷は楽でしょうが, 長い式になると, それなりに面倒で, 明瞭さがなくなります. 少なくとも, 現代の表記 ‘$(A \to B) \to ((A \to \neg B) \to \neg A)$’ と比べても, この表記では
論理式の構造が判り難いことは理解されるでしょう.

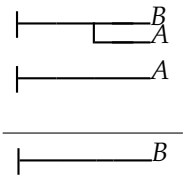
Peano の表記と比較して, 概念記法にはそれなりの領域を必要としますが, 基本的に「判断」‘’,「水平線」‘’,「条件」‘ ’,「否定」‘’ と後述の「全称記号」‘’ の組合せであり, 論理式の

構造を把握することに関しては意外に便利な図式です. この点では漢字と同様に, この表記は現在の GUI を用いる計算機にとって Peano 流儀の表記以上に親和性が高いのではないかと私は思っています. またプログラム言語でも, C や FORTRAN は基本的に文を一直線上線に並べても問題のない線的な言語ですが, Python ではインデントが構文の 1 つとなった二次元的な言語になっています. こうした点からも再評価されても良いのではないかと思っています.

推論

Frege は命題を独自の表記による図式にしただけではなく, 表現しています. ここで Frege が採用している推論は「**前提肯定 (Modus Ponens, MP とも略記)**」のみで, これは次で表記されます:

Frege による前提肯定の表記



この表記で上の二つの論理式が仮定の命題 '$A \to B$' と命題 'A' で, これらの仮定から命題 'B' が判断できるということを明瞭にするために仮定と結論の間に線分, すなわち, Frege が「**移行記号**」と呼ぶ記号を入れています. この表記方法も現在の論理学の証明図式で用いられています.

この前提肯定は三段論法の一種で, 次のような推論を行います:

前提肯定の例

(大前提)　雨であれば運動会は中止である。
(小前提)　雨である。
(結論)　　だから、運動会は中止である。

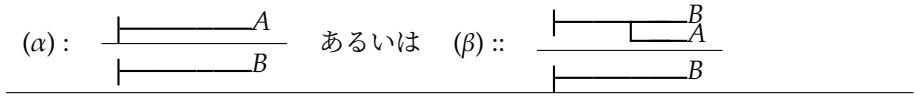
ここでの大前提の「P ならば Q である」に対し, 小前提が「P である」と大前提の前件 (前提である命題 P) を肯定的に主張しているために「前提肯定」と呼ばれます. さらに「前提肯定」は「前件肯定」, あるいは「分離規則」とも呼ばれます.
なお,「前件肯定」があれば,「後件肯定」もちゃんとあります. 後件肯定は前提肯定に似た推論で次に示す推論を行います:

後件肯定の例

(大前提)　雨であれば運動会は中止である。
(小前提)　運動会は中止である。
(結論)　　だから、雨である。

このように大前提の後件「運動会が中止である」を小前提が肯定していることから「後件肯定」と呼びます. ただし, こちらは正しくない推論です. 実際, 雨でなくても, 火山の噴火でポンペイ同様になっていれば運動会をやりたくても到底できませんから….
この Frege の概念記法は 2 次元的な広がりを持つため, 仮定が多くなると証明図式の表記が流石に難しくなります. そこで, Frege は幾つかの略記方法を導入しています.
たとえば, 前提肯定の図式で大前提の命題 ' ' を ' ' とすることで, 命題にラベル α を割当て, 小前提の命題 '' にも同様に '' でラベル β を割当てます. そうすると次のようにで略記できます:

Frege による前提肯定の略式表記



この最初の図式では左側に (α) : のように移行記号の左側に式番号とコロン ":" が置かれています. そして, このコロンで 'α' が前提肯定の大前提であることを示します. 次の図式では移行記号の左側に '(β) ::' と表記されています. ここでの二重のコロン "::" は 'β' が小前提であることを示します. さらに, 二つの小前提を用いた推論を行う場合は移行記号の左側に '(α, β) ::' と記述して二重コロン "::" の右側にある移行記号の横棒二本引く表記になります.
さらに図式を明瞭に表現するために図式の代入が行われることを明示する表記も考えています. この表記は公理の解説で説明します.
Frege の概念記法では推論規則としては MP だけですが, 伝統的論理学では数多くの推論式を列挙しなければなりません. 実際, 三段論法では命題の主語・述語のくみあわせを考慮するだけで推論規則は 256 種類にも上り, そのため, 無効な推論に対しては替え歌や Barbara(AAA),Celarent(EAE), Darii(AII),Ferio(EIO) 等の人名で憶えていた程です.

4.19.2　論理学の刷新

意味と意義

Frege は命題の「**意味 (Bedeutung)**」と「**意義 (Sinn)**」を区別しています. たとえば, 命題 '$2^2 = 4$' と命題 '$2 + 2 = 4$' は何れも真となる命題で, 「**真**」という真理値が, これらの命題の意味になります. しかし, これらの命題が主張していることは, 前者が「2 を二乗する と 4 になる」, 後者は「2 に 2 を加える と 4 になる」ことで, 全く別の主張なのです.
このように Frege は論理式が持つ真理値が論理式の「**意味**」であり, 論理式自体の主張を「**意義**」, あるいは「**思想**] とし, 意義と意味を分けて考察しています. このような命題に関する思索から Frege は分析哲学の始祖とされている所以です. さらに「**語の意味は文での関連において問われるべきであって孤立して問われるべきではない**」という「**文脈原理**」は非常に有名です. Maxima では文脈 (Context) を用いて数式の評価を実際に行いますが, この手法の根源はこの文脈原理にあります.

函数, 概念, 値域

Frege の功績で最も重要なことは「変項・函数方式」をはじめて論理学に取り入れたことです. これによって「xl は 0 よりも大きい」という命題を「x」が主語で述語が「0 よりも大」と命題を分析するのではなく, 'Greater$(x, 0)$' のように函数を用いて記述しています.

函数:　Frege は「**函数の表現は補完を要するものである**」と述べています. たとえば, '$(2+3\cdot x^2)\cdot x$' といった x の函数式が与えられたときに文字 "x" は数値を代入すること, すなわち, 補完によって式の意味が確定されるための場所を空けることに役立っていると主張しています. その意味で '$(2+3\cdot(\,)^2)\cdot(\,)$' のように補完を必要とする位置を示す括弧でも構わないことになります. この補完

すべき対象が置かれる場所を「**項場所**」, 項場所を占有する対象を「**項**」と呼びます. そして, 函数の項場所を示すためにギリシャ文字の母音以外の小文字 “ξ” を用います. そして, “ξ” が現われる箇所を「ξ-**項場所**」と呼びます. この表記方法によって, 函数と函数値は厳密に区別されます. たとえば, 数学的函数 $x^2 + x$ は ‘$\xi^2 + \xi$’ とメタ変項 ξ を用いた函数の表現となります. これは Church の λ 記法による表記: $\lambda\xi.\, \xi^2 + \xi$ の先駆と言えるものです[83].

概念: 函数 $F(\xi)$ の値が真理値となる場合に函数 $F(\xi)$ を「**概念 (Begriff(独),concept(英))**」と名付けています. それから $F(\Delta)$ が真となる場合, 対象 Δ を「**概念 F に属する対象**」と呼びます. たとえば, 概念 ‘$\xi^2 - 4 = 0$’ が真になるためには “ξ” は -2 か 2 でなければなりません. したがって, この場合は -2 と 2 が概念 ‘$\xi^2 - 4 = 0$’ に属する対象となります.

値域, 外延 函数 $F(\xi)$ の ξ が取り得る対象で構成されるもののことを「**値域**」と呼びます. さらに函数 $F(\xi)$ が真理値を取る場合, 函数 F の値域を「**概念の外延**」, あるいは簡単に「**クラス**」と呼びます. そして, Γ を外延とする概念のことを「Γ-**概念**」と呼びます.

値域の表現: Frege は函数の値域を表現するために, 変項記号と古代ギリシャ語の無気息記号 “$\dot{}$” [84]の組合せで表現します. たとえば, ‘$\sin\xi$’ の値域は $\dot{\varepsilon}(\sin\varepsilon)$ となり Church の λ-式風になります. 次に ‘$\xi^2 - 4 = 0$’ の値域は $\dot{\varepsilon}(\varepsilon^2 - 4 = 0)$ となりますが, この値域に帰属する対象は -2 と 2 です. これらの例で示すように記号 “$\dot{\varepsilon}$” を函数の左側に置きます. そして, 右側の函数の変項は無気息記号が置かれたギリシャ母音小文字で充足されていなければなりません. なお, この無気息記号が影響を及ぼす範囲を「**作用域**」と呼びます. この無気息記号の作用域は複数の無気息記号が置かれたギリシャ母音小文字がある式で無気息記号を伴うギリシャ母音小文字が現われた位置から開始して無気息記号を伴う同名のギリシャ母音小文字が現われた個所で途切れます. つまり, この表記は LISP の λ 式に似た表記になります. 実際,$\dot{\varepsilon}(\varepsilon = \dot{\varepsilon}(\varepsilon^2 - \varepsilon))$ は函数 $\xi = \dot{\varepsilon}(\varepsilon^2 - \varepsilon)$ の値域であり, $\dot{\varepsilon}(\varepsilon = \dot{\alpha}(\alpha^2 - \varepsilon))$ は函数 $\xi = \dot{\alpha}(\alpha^2 - \xi)$ の値域となります.

次に概念 ‘$\xi + 2 = 5$’ のように帰属する対象が一つのみの場合, Frege は記号 “$\backslash$” を用いて表現します. たとえば, $\backslash\dot{\varepsilon}F(\varepsilon)$ で概念 F に帰属する唯一の対象を表現します. この記号 “$\backslash$” は印欧諸言語での定冠詞 (英語なら THE) の代用を意図したものです. たとえば, 概念 ‘$\xi + 2 = 5$’ であれば, $\backslash\dot{\alpha}(\alpha + 2 = 5)$ でその対象を表現し, ‘$\backslash\dot{\alpha}(\alpha + 2 = 5) = 3$’ は真になります. この記号 “$\backslash$” は現在の数理論理学の “$\iota$” 記号 [85]に相当するものです. しかし, 概念 $F(\xi)$ に帰属する対象が複数存在する場合, 概念 $F(\xi)$ が偽の場合は $\backslash\dot{\varepsilon}F(\varepsilon)$ の値は値域 $\dot{\varepsilon}F(\varepsilon)$ になります. したがって, ‘$\backslash\dot{\varepsilon}(\varepsilon^2 = 1) = \dot{\varepsilon}(\varepsilon^2 = 1)$’ は値域 $\dot{\varepsilon}(\varepsilon^2 = 1)$ が唯一の対象を含まないために真となり, 同様に ‘$\backslash\dot{\varepsilon}(\overline{\quad\quad}\varepsilon = \varepsilon) = \dot{\varepsilon}(\overline{\quad\quad}\varepsilon = \varepsilon)$’ も値域 $\dot{\varepsilon}(\overline{\quad\quad}\varepsilon = \varepsilon)$ が空となるために真となります.

---

[83]Church の λ-記法は Frege の表記の影響を受けていない独自のものだそうです ([24],p.132-134).

[84]古代ギリシャ語では H を現わす文字がないため, 先頭に H の音が来る場合に母音の上に気息記号を入れます. 無気息記号は逆に母音が先頭の場合に H の音が入らないことを示す記号です.

[85]ι 記号の詳細は前原 [55], p.59-60 を参照. 又,Russell の Principia Mathematica では定冠詞 “THE” を表現するものとして “$\imath$” が用いられています ([92],p.68 を参照)

真理値:　Frege は真理値に関しても面白いことを述べています. これは函数の値域を一層厳密に規定するための考察との関連で「$\dot{\varepsilon}(\_\_\_\_\_\varepsilon)$ **が真であり,** $\dot{\varepsilon}(\_\_\_\_\_\overset{\mathfrak{a}}{\smile}\_\_\_\mathfrak{a} = \mathfrak{a})$ **を偽であるべしと約定しよう**」と述べています ([50]§10). これは「**Church の真理値**」TRUE $\overset{def}{=} \lambda xy.x$,FALSE $\overset{def}{=} \lambda xy.y$ と比較しても興味深いことですが, このことは「**うそつきの逆理**」に対処できなくなることも意味します.

二項函数:　Frege は変項が二つの函数, すなわち, 二項函数についても議論しています. 「**二項函数**」とは「**二重の補完を必要とする函数**」, すなわち,「**項場所を二つ持つ函数**」です. この二項函数の表記では二つの項を示すためにギリシア文字 ξ と ζ を用います. たとえば, 数学的函数 $(x+y)^2+y$ は $(\xi+\zeta)^2+\zeta$ となります. そして函数の場合と同様に,ξ が置かれている場所を「ξ**-項場所**」, ζ が置かれている場所を「ζ**-項場所**」と呼びます.

関係:　二項函数で, '$\xi = \zeta$' や '$\xi > \zeta$' のように値として真理値を持つものを「**関係**」と呼びます. そして, 関係 $\Phi(\Gamma, \Delta)$ が真の場合に「**対象 Γ は対象 Δ に対して関係 $\Phi(\xi, \zeta)$ にある**」といいます. そして, 二項函数 $\Phi(\xi, \zeta)$ の値域を「**重積値域**」と呼び, $\dot{\alpha}\dot{\varepsilon}\Phi(\alpha, \varepsilon)$ で重積値域を表現します. たとえば, $\xi + \zeta$ の重積値域は $\dot{\alpha}\dot{\varepsilon}(\varepsilon + \alpha)$ となります. ここで, α が二項函数の ξ 項, ε が二項函数の ζ 項に対応します.

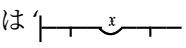
命題を函数として表現した結果, 函数 $F(\xi)$ に対し, この函数記号 "F" 自体を函数記号 "G" で置換えることも可能になります. 現在の言葉では, Frege の概念記法は一階の述語論理ではなく, 高階の述語論理です.

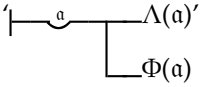
量化詞の導入

Frege のもう一つの大きな功績は量化詞を導入し, 多重量化を取り入れたことです.
たとえば, 「全ての x に対して $\Phi(x)$ が成立する」は現在, '$\forall x\Phi(x)$' と記述します. ここで記号 "$\forall$" を「**全称記号**」と呼び, 記号 "$\forall$" の直後にある変項 x を「**束縛変項**」と呼びます. 概念記法に於ける全称記号は '$\_\overset{\smile}{\_}\_$' で, この水平線の窪みに束縛変項を記述し, 概念記法による式の左側に置きます. したがって, '$\forall x\Phi(x)$' を概念記法で記述すると '$\vdash\!\_\overset{\mathfrak{x}}{\smile}\_\_\ \Phi(\mathfrak{x})$' となります.

ここで全称記号によって束縛された変項の影響が及ぶ範囲, すなわち,「**作用範囲 (scope)**」は, その束縛変項が置かれた窪みから右側の式に対して同じ文字の全称記号が出現するまでの範囲になります. そして, 全称記号の束縛変項を他の項と区別するためにドイツ文字を用い, 全称記号に束縛されない項, すなわち,「**自由変項**」を通常のラテン文字のローマン体を利用します. なお, '$\_\overset{\mathfrak{x}}{\smile}\_(\_\Phi(\mathfrak{x}))$' や '$\_(\_\overset{\mathfrak{x}}{\smile}\_\_\Phi(\mathfrak{x}))$' は融合によって '$\_\overset{\mathfrak{x}}{\smile}\_\_\Phi(\mathfrak{x})$' になります.

次に「$\Phi(x)$ を充す x が存在する」といった命題は現在,「**存在記号**」"$\exists$" を用いて '$\exists x\Phi(x)$' と記述されます. 全称記号と存在記号には '$\neg(\forall x\Phi(x)) = \exists x\neg\Phi(x)$' の関係があります. そのため, Frege は存在記号 "$\exists$" に対応する表記を創らずに, 単純に否定と全称記号の組合で表現しています. たと

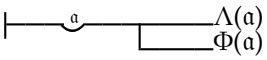
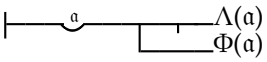
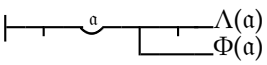
えば, ‘∃xΦ(x)’ は ‘  ’ となります.

この全称記号を用いると全称肯定の「全ての Φ は Λ である」は「全ての ɑ に対し, Φ(ɑ) は Λ(ɑ) である」と言い換えられるので, ‘  ’ で表現できます.

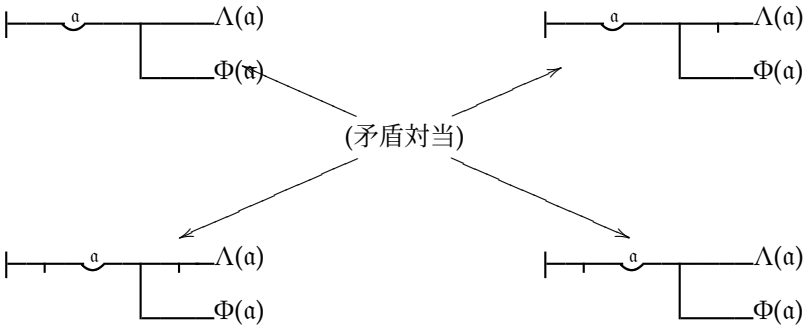
概念記法による AEIO

そこで概念記法を用いて AEIO を表記してみましょう:

Frege による AEIO の表記

(A)	「全ての Φ(*a*) は Λ(*a*) である」	
(E)	「全ての Φ(*a*) は Λ(*a*) ではない」	
(I)	「ある Φ(*a*) は Λ(*a*) である」	
(O)	「ある Φ(*a*) は Λ(*a*) ではない」	

概念記法を用て次の対当関係表 (SQUARE) を得ます:

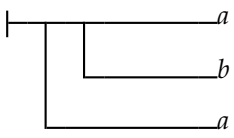


伝統的論理学と違って反対関係にある全称肯定と全称否定, そして, 特称肯定と特称否定は共に真, あるいは偽でありえます. これは Frege の肯定判断の表記に存在含意 (Existence Import) がないためです ([50],§13 の注 1). 現代の数理論理学でも Frege と同様に存在含意を含みません.

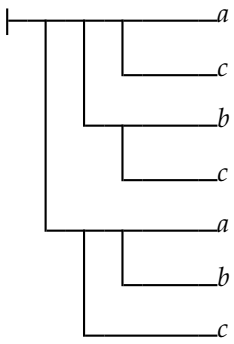
「概念記法」の公理系

「概念記法」で掲げられている公理を次にまとめおきます:

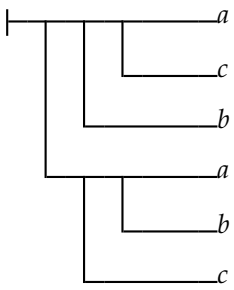
概念記法の公理系

1. 

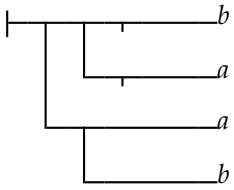
$a \to (b \to a)$

2. 

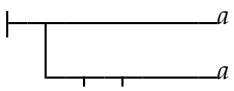
$(c \to (b \to a)) \to ((c \to b) \to (c \to a))$

8. 

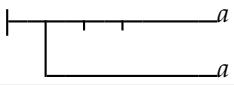
$(c \to (b \to a)) \to (b \to (c \to a))$

28. 

$(b \to a) \to (\neg a \to \neg b)$

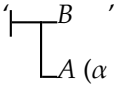
31. 

$\neg\neg a \to a$

41. 

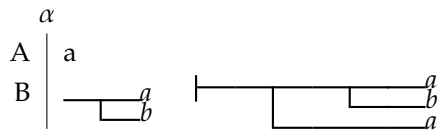
$a \to \neg\neg a$

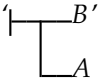
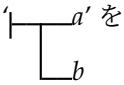
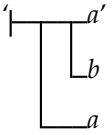
公理の番号は「概念記法」([48]) の式の番号です. ここで番号 8. の式は一つは他の公理から導出可能なため, 独立な公理は 5 個です ([33]). ではこれらの公理の内容を吟味してみましょう:

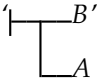
番号 1. の公理　最初の番号 1. の公理で ‘ 

’ のように命題にラベル α をつけて次の表記が可能です:

番号 1 の公理の考え方



この図式の意味は左側の縦棒 "|" の上のラベル α で示される式 '  ' の A に a を代入, B に '  ' を代入して右側の式 '  ' が得られるというものです.

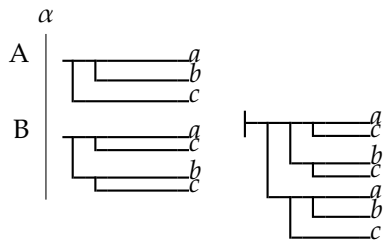
ここで '  ' は, 「B が否定され, かつ A が肯定される事態を除外する」と読め, 番号 1. の公理は「

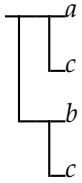
が否定され, a が肯定される事態を除外する」と読めます.

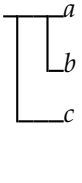
さらに, この ' ' が否定されるのは「a が否定されて b が肯定される」場合です. 以上から, この公理は「a が否定され, b が肯定され, さらに a が肯定される事態が生じない」ことになりますが, a が同時に否定や肯定されることがありえないために番号 1. の式が恒等的に真の論理式, すなわち, 「恒真式 (tautology)」 であることが判ります.

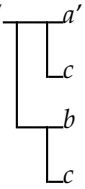
番号 2. の公理　番号 1. の公理と同様に番号 2. の公理も式 α から生成できます:

番号 2 の公理の考え方



この場合も「 

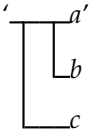
が否定され 

が肯定される場合を除外する」と読めます. ここで ' 

'

が否定されるのは「

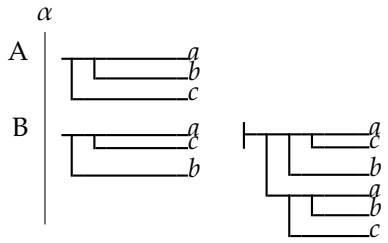
が否定され

が肯定される」場合です.
すなわち,「c と b が肯定されて a が否定される」場合が相当します.
次の ‘ 

’ が肯定されるのは「b と c が否定されて a が肯定される」場合となり, これらの結果は互いに矛盾しているために番号 2. の式が恒真式であることが判ります.

番号 8. の公理　番号 8. の公理は「条件の可換性」の公理と呼ばれるものです.

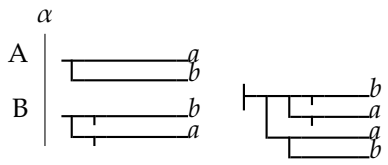
番号 8 の公理の考え方



この式は「B が肯定されて A が否定されることはない」と読めますが, ここで B が肯定されるのは「a が肯定される」場合, A が否定されるのは「a が否定されて b と c が肯定される」場合ですが, a が同時に肯定・否定されることはありえないことなので, 番号 8. は恒真式であることが判ります.

番号 28. の公理　番号 28. の公理は「命題の対偶」と呼ばれる公理です.

番号 8 の公理の考え方



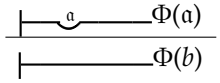
この命題は「B が肯定され A が否定されることはない」と読めます. A が否定されるのは「a が否定されて b が肯定される」場合, B が肯定されるのは「b が否定されるか, a が肯定される」場合です. これらは同時に生じないため, 番号 28. も恒真式であることが判ります.

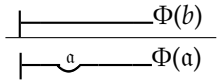
番号 31. と番号 41. の公理　番号 31. と番号 41. の公理が「二重否定の除去」と呼ばれる公理になります. これらの式は ‘a’ が真でも, ’a が偽でも常に真になることが確認できます. このことから, これらの式が恒真式であることが判ります.

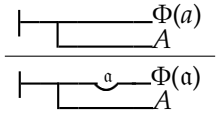
正しい推論

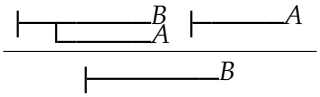
Frege は正しい推論として次の図式を挙げています:

概念記法の推論規則

1. 

2. 

3. 

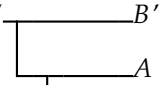
4. 

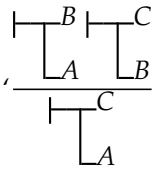
これらの推論を今風に記述すれば次のようになります:

推論規則の現在の表記

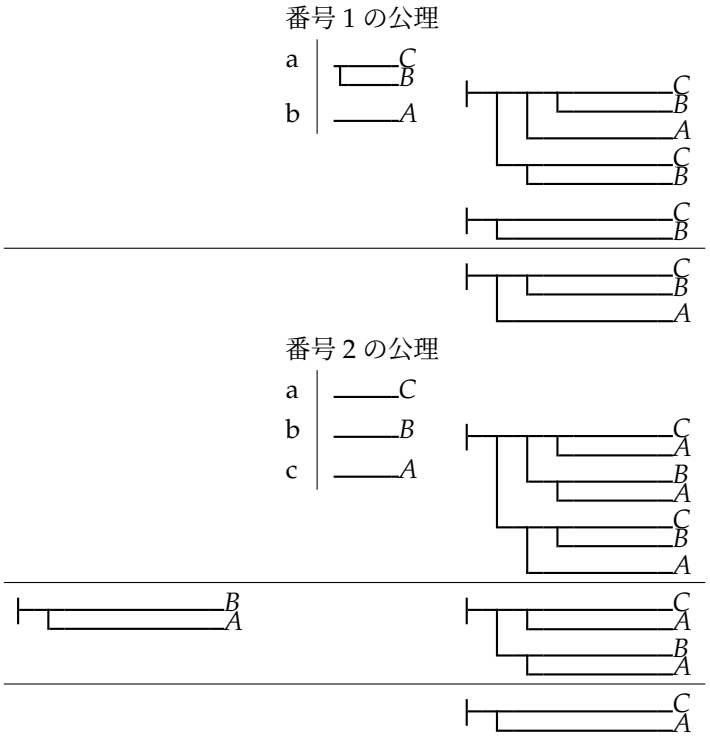
1. $\dfrac{\forall x \Phi(x)}{\Phi(a)}$
2. $\dfrac{\Phi(a)}{\forall x \Phi(x)}$
3. $\dfrac{A \to \Phi(b)}{A \to \forall x \Phi(x)}$
4. $\dfrac{a \; a \to b}{b}$

推論規則 1., 2., 3. は全称記号 “∀” に関連した規則で, 最後の推論規則 4. は「前提肯定 (MP)」です.

☆ **“∨” の可換性**　ここでは具体的な例として離接記号 “∨” の可換性を示してみましょう.
まず, ‘ 

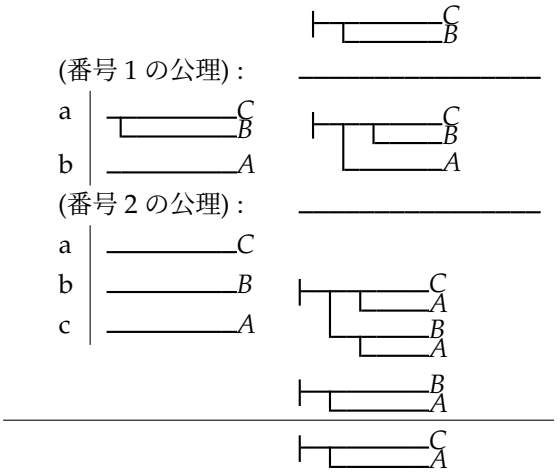
’ によって ‘$A \vee B$’ を定義します.
それから次の推論規則 ‘ 

’ を証明しておきます. この証明で用いる公理は番号 1 の公理と番号 2 の公理, それから推論規則は MP のみです.
この証明図を次に示しておきます:



この図式によって新しい推論規則 ‘$\dfrac{\begin{array}{cc}B & C\\ | & |\\ A & B\end{array}}{\begin{array}{c}C\\ |\\ A\end{array}}$’ がえられます.

この証明図式で (α) : や (β) :: を用いてより簡易に表示することができます.
では, この証明図を簡易にしてみましょう:

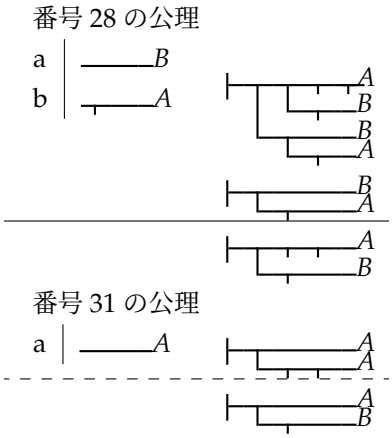


この図では最初の MP で番号 1 の公理への代入から得られた式を大前提とするために,‘(番号 1 の

公理) :__' のようにコロンが一つ付きます. また, 第二の MP でも番号 2 の公理への代入から得られた式を大前提とするので同様にコロンが一つ付きます.
さて, ここで証明をしてた推論規則は「仮言三段論法」に相当するもので, Frege は「算術の基本法則」の番号 1,2 の公理のような考え方で証明しています. さらに Frege は '⊢B ', '⊢C ' のとき, 仮言三段論法による推論を次のように表記します[86]:

仮言三段論法による推論

この推論と番号 28 と番号 31 の公理を組合せると, 次の図式が得られます:

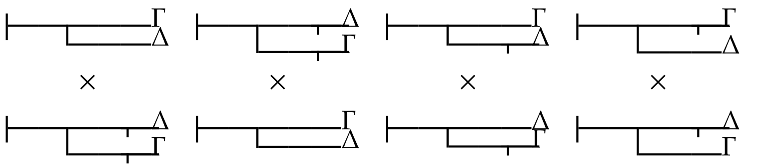


ここで, この図式の最初の段 (実線の箇所) の推論では MP, 最終段 (破線の箇所) の推論では先程証明した推論規則 (仮言三段論法) を用いています.
この図式から '⊢B ∨ A' と判断でき, '⊢A' と '⊢B' を交換して, '⊢A ∨ B'も同様に示せるため, 'A ∨ B = B ∨ A', すなわち, 演算子 "∨" が可換演算子であると判断できます.
同様に '⊢A ∧ B' を '⊢B' で定義した場合, この図式は '⊢' と分解されます. したがって, '⊢A ∧ B = ⊢ ((⊢A) ∨ (⊢B))' となり, ここで括弧の中の '(⊢A) ∨ (⊢B)' に注目すると, "∨" の可換性から目的の "∧" の可換性も言えます. なお, 現在の表記による証明は前原 ([55],p.13-37) にもあります. このように論理主義は基本となる少数の公理から数学を構築するものだということが理解されたでしょうか？
次に Frege は記号 "×" を用いて命題の対偶を取る操作を表現しています. この変換によって前者か

[86]「算術の基本法則」[50],p.93

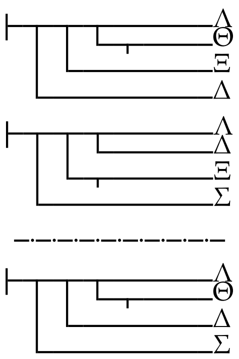
ら後者 (記号 “×” を挟んで上から下) にも, 後者から前者者 (記号 “×” を挟んで下から上) にも移行が可能です:

対偶変換

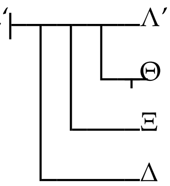


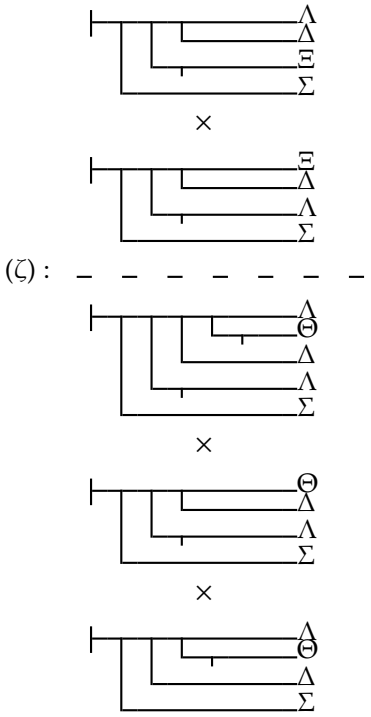
そして, 推論規則として「MP」と「対偶変換」を用いて「構成的両刀論法」も証明しています ([50],p.105). さらに「構成的両刀論法」を用いた場合には, 移行記号として “——.——” を用いています:

構成的両刀論法による推論

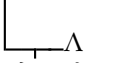


この推論の証明図は「対偶変換」と「前提肯定」を用いたものです.

実際,‘  Λ’ をζと置いた「構成的両刀論法」の証明図を次に示しておきます:



この図式で示すように最初に Λ と Ξ に対して対偶変換を行った命題に対して ζ の命題を合せた前提肯定から破線下の命題が得られます. そして, ‘——Λ’ が ‘——Λ’ と

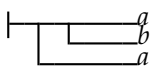
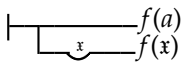
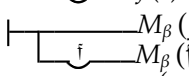
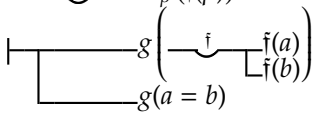
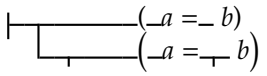
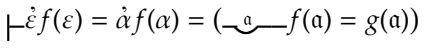


真理値が等しいことから, 一つの Λ で纏め, それから Λ と Θ に対偶変換を行って中間の式をえて, 最後に Λ と Θ に対して対偶変換を行なって目的の命題をえることが判るかと思います.

「算術の基本法則」の公理系

さて,Frege の論理主義は「算術の基本法則」([50]) で頂点を迎えることににになります. そこで, この「算術の基本法則」の公理系 ([50],p.209-210) から代表的なものを次の表に纏め, 現在の表記との対照を付けておきましょう:

「算術の基本法則」の公理系

	Frege の表記	現在の表記
I.		$\neg a \to a$
IIa.		$\forall x f(x) \to f(a)$
IIb.		$\forall f\ M_\beta(f(\beta)) \to M_\beta(f(\beta))$
III.		$g[a = b] \to$ $g[\forall f(f(a) \to f(b))]$
IV.		$\neg(a = \neg b) \to (a = b)$
V.		$\dot{\varepsilon}.f(\varepsilon) = \dot{\alpha}.g(\alpha) \equiv$ $\forall x f(x) = g(x)$
VI.	$a = \backslash\dot{\varepsilon}(a = \varepsilon)$	$a \equiv \iota\lambda\varepsilon.(a = \varepsilon)$

公理 I:　公理 I は「概念記法」の公理に含まれている公理です. この公理は Hilbert の体系にも導入されいます.

公理 IIa と公理 IIb:　公理 IIa は 1 階の函数に対する「**普遍例化**」, 次の公理 IIb は単変項の 2 階函数に対する普遍例化になります. ここで M_β は一つの変項 (β) を持つ 2 階の函数を意味しています. このように Frege の体系は第 1 階の論理式に限定されない高階の論理式です.

公理 III:　公理 III は「**代入則**」になります.

公理 IV:　公理 IV を現在に記号で書換えると '$\neg(a = \neg b) \to (a = b)$' となり, 「命題 '$a = \neg b$' と 命題 '$a = b$' が共に否定されることはない」となります.
この公理は「**排中律**」に似てなくもありません. なお, 排中律はここで明記されていませんが前提としてあります[87].

公理 V:　公理 V を現在の数理論理学の記号を先ぜて書くと, '$\dot{\varepsilon}f(\varepsilon) = \dot{\varepsilon}g(\varepsilon) \equiv \forall a(f(a) = g(a))$' になります. この意味は「値域が同じものに対して, その値も等しく, その逆も成立する」というものです. これを集合論風に言えば,「論理式 f と g が定める集合が等しければ, その集合の任意の元 a について $f(a) = g(a)$ を充し, その逆も成立する」という意味です. この公理は「算術の基本法則」で数の定義そのものに関わる重要な公理です.

[87]第三の途はない (teritum non datur) 「算術の基本法則 第 II 巻」§56([50],p.256)

公理 VI: 公理 VI の記号 "\" は現在の数理論理学の ι-記号に相当し, その値域に帰属する唯一の対象を示すものです. この公理 VI の '$\grave{\varepsilon}(a = \varepsilon)$' で概念 $a = \varepsilon$ を充す対象が a だけであることを保証します.

4.19.3 Frege の自然数の構成

個数言明

Frege は「算術の基本法則」で, 算術が論理学から導出可能であることを示すための議論を進めていますが, この目的遂行のために自然数の定義を行わなければなりません. この自然数に関しては「算術の基礎」にて彼独自の概念記法等の記号を極力用いずに議論を進めていますが, この基本法則では厳密さと明確さを保持して自然数の定義を行うためにさまざまな記号を用いています. この記号や函数の導入を行う前に Frege の基数を簡単に解説しておきましょう.

日常では, $1, 2, 3, \ldots$ と何気に使っている数ですが, 幼児にとっては難しい概念です. さらに幼児は非常に実在論的です. 物がないこと (**ないない…**) と物が存在すること (**ばぁ!**) への理解は早いもので, 触れられる, 或いは味わえるということも彼等にとっては非常に重要です. そこで, 顔の鼻や口に数 1 を対応付けたり, 両目に数 2 を対応付けて理解させようと苦労するわけですが, それで判ってくれるかどうかはいささか？です. この Frege の基数も物と対応させる思考に連なるものと言えるでしょうか.

たとえば数 6 を説明する場合, うかつに「**6 は 5 たす 1 で得られる数**」だと言うと, 「**5**」は何か, 「**1**」は何か, さらに「**たす**」とは何かといった問題が出ます. それに数が何であるかを説明する際に, 下手に数自身に訴えると悪循環に陥ってしまいます[88]. その上, 実在論を信条とする幼児には判らないでしょう. そこで, 彼等にも理解可能な林檎を使って §4.12 で解説した手法を採用するのが妥当でしょう.

この話をもう少し進めてみましょう. そこで今度は「**骰子の目**」のように命題が言明する数としてみましょう. すると, この定義に数字は何処にも出ていないので妙な循環は生じません. それに幼児には骰子の模様を見せて, 描かせれば良いのです. こうすることで数 6 は幼児にとっても実体を伴う物として捉えられます. 同様に「甲虫の足」はどうでしょうか. 甲虫は昆虫なので足は 6 本あります. したがって, この命題は「骰子の目」と同じ数に関する言明となっています. そして, 重要なことですが「骰子の目」に属する対象, 要するに面の模様と「甲虫の足」に属する対象, すなわち, 虫の足の間に一対一の対応関係を入れられます. これも非常に現実的に, 甲虫の足に骰子の目の模様を付けてやれば良いので, 幼児にもこの方法で等数性が理解できそうです.

それでは,「骰子の目」と「蛸の足」はどうでしょうか？「骰子の目」に属する対象から「蛸の足」に属する対象への一意的な写像はできますが, その逆写像はできません. このように $1, 2, 3, \ldots$ と対象を数え上げなくても, 同じ数の言明に対し, そして, その場合に限って概念に属する対象の間に一

[88] 男を調べると「女ではない人間」, 女を調べると「男ではない人間」のように循環したり, 男のことを「人間の雄」, 女のことを「人間の雌」のような言い換えをする辞書では困りますね.

対一対応の関係が存在することが判ります. ここで,「**概念 F に族する諸対象を概念 G に属する諸対象に一対一で対応付ける関係 φ が存在する**」場合に「**概念 F が概念 G と等数的である**」と呼びましょう. このことは「**Hume の原理**」(HP とも略記) と呼ばれ, Frege の算術の導出で核となる重要な原理です. さて, この Hume の原理によれば,「骰子の目」と「等数的な概念」の間には一対一対応の函数が存在するので同値関係が入れられます. だから,「骰子の目と等数的な概念」で数 6 が定義できそうです. そして, 個数言明と Hume の原理を組合せれば基数の定義さえもできそうです.

Julius Caesar 問題

ところが個数言明と Hume の原理で数を定義する方法には大きな問題, それも大変難しい問題があります. たとえば「**数 Julius Caesar**」といった命題はどうでしょうか？ これは「**Julius Caesar 問題**」と呼ばれる重大な問題で, この命題の中の「Julius Caesar」が歴史的な人物 (ガリアの征服者) なのか, それとも, 何かの数なのか, この命題だけでは全く判断ができません. Hume の原理によると, 与えられた数に対しては, それが等しいかどうかの判断ができますが, 与えられた対象が数であるかどうかの判断はできません. つまり, 個数言明と Hume の原理だけでは, 数を定義するための能力がまだ足りないのです.

概念の外延

この難題に対して Frege は個数言明ではなく「**概念の外延**」, すなわち,「**クラス (類)**」という集合論的な対象を導入します. そして, クラスによって Frege は Julius Caesar 問題は取り敢えず解決したものと考え[89], 基数を「**あるクラスと等数的なクラス**」として定義します. つまり, 先程の数 6 を例に解説すると, 数 6 は「**骰子の目のクラスと等数的なクラスで構成されたクラス**」[90]として定義されるのです.

このように, 数の定義を行うためには, 対象がある概念に帰属すことと, ある概念に属する対象を別の概念に属する対象に一対一対応付ける写像が存在することを示し, その上で, あるクラスと等数的なクラスを定義しなければなりません.

幾つかの式の定義

ここでは基数の定義で利用される幾つかの基本的な式の定義を述べます[91]. なお, ここでの表記の定義では記号 "⊩" を用いています. この記号 "⊩" は記号 "$\stackrel{def}{=}$" に相当し, 新しい表現を定義する記号です. たとえば, Δ を既存の表現を用いて構築した式とし, Γ を新しく導入する表現とすると $\Vdash \Gamma = \Delta$ は $\Delta \stackrel{def}{=} \Gamma'$ の意味になります:

[89] 詳細は [50],§10(p.69-77) とその注釈を参照.
[90] このように基数はクラスのクラスになります
[91] 特殊な記号を用いるため,fge パッケージ [134] を利用しています.

基本法則の重要な記号の定義

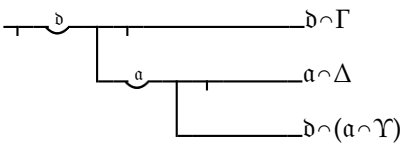
記号	定義式
$a \frown u$	$\vdash \backslash\dot{\alpha}\left(\begin{array}{l}\mathfrak{g}(a) = \alpha \\ u = \dot{\varepsilon}g(\varepsilon)\end{array}\right) = a \frown u$
$\mathrm{I}p$	$\vdash \left(\begin{array}{l}\mathfrak{d} = \mathfrak{a} \\ \mathfrak{e} \frown (\mathfrak{a} \frown p) \\ \mathfrak{e} \frown (\mathfrak{d} \frown p)\end{array}\right) = \mathrm{I}p$
$\rangle p$	$\vdash \dot{\alpha}\dot{\varepsilon}\left[\begin{array}{l}\mathfrak{d} \frown \varepsilon \\ \mathfrak{a} \frown \alpha \\ \mathfrak{d} \frown (\mathfrak{a} \frown \xi) \\ \mathrm{I}p\end{array}\right] = \rangle p$
$\mathfrak{H}p$	$\vdash \dot{\alpha}\dot{\varepsilon}(\alpha \frown (\varepsilon \frown p)) = \mathfrak{H}p$
$p \backsmile \mathfrak{q}$	$\vdash \dot{\alpha}\dot{\varepsilon}\left(\begin{array}{l}\varepsilon \frown (\mathfrak{r} \frown p) \\ \mathfrak{r} \frown (\alpha \frown q)\end{array}\right) = p \backsmile q$

☆ $a \frown u$　この式は対象の概念への帰属を表現するために用います ([50]§34). たとえば, '$\Delta \frown \dot{\varepsilon}\Phi(\varepsilon)$' が真であるとは '$\Phi(\Delta)$' が真になることと同値で, このことは対象 Δ が概念 Φ に帰属することを意味します. なお, 現在の表記を併用して $a \frown u$ を表現すると, u が概念の外延であれば,$a \frown u \overset{def}{=} a \in u$ が対応し, u が概念の外延でなければ真理値の偽 (false) が対応します.

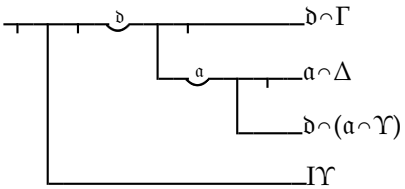
☆ $\mathrm{I}p$　この式は関係 P の「多対一性」を表現する式になります ([50]§37). この表記の定義式の二つの前提条件は関係 P の外延 p に対して演算子 "$\frown$" を作用させたものになっています.
ここで関係 $\Phi(\xi, \zeta)$ の外延は $\dot{\alpha}\dot{\varepsilon}\Phi(\varepsilon, \alpha)$ になります. そして, $\Delta \frown (\dot{\alpha}\dot{\varepsilon}\Phi(\varepsilon, \alpha))$ は函数 $\Phi(\xi, \Delta)$ の値域 $\dot{\varepsilon}\Phi(\varepsilon, \Delta)$ に対応し, '$\Gamma \frown (\Delta \frown (\dot{\alpha}\dot{\varepsilon}\Phi(\varepsilon, \alpha)))$' に '$\Phi(\Gamma, \Delta)$' が対応します. したがって, $\Gamma \frown (\Delta \frown (\dot{\alpha}\dot{\varepsilon}\Phi(\varepsilon, \alpha)))$ は $\Phi(\Gamma, \Delta)$ になります. そして, このとき, Γ と Δ は P-関係にあると呼びます.
さて, この式の定義式は「e と a が P-関係,e と b が P-関係であれば $b = a$ となる」と読めます. つまり,「e と P-関係になるものは a 一つに限定される」という意味で, これが「多対一」という関係なのです. この関係を現在の論理学の式で表現すれば, '$\forall x \forall y \forall z[p(y, x) = p(y, z) \rightarrow x = z]$' となるでしょう.

☆ $\rangle\xi$　この式は概念間の写像の単射性を表現するために用います ([50]§ 38). まず, Frege は「関係 φ が概念 F に属する対象を概念 G に属する対象に対応付けることの意味」を「a が概念 F に属するという命題と a が概念 G に属する対象に対して関係 φ にないという命題は全ての a に対して両立しない」と言い換えます ([49]§ 71, [50]§ 38,p.151). これは次のように理解することができるでしょう. まず, 概念 F に属する元 a が概念 G に属する元 b と対応が付くということは, 概念 F から概念 G の函数 f が存在して '$f(a) = b$' となることです. ここで '$f(a) = b$' となることを '$\varphi(a, b)$' と記述しましょう. このとき, a と b の関係は φ-関係になります. ところで, このような対応関係がないということは概念 F に属する任意の元 a に対して,φ-関係となる概念 G に帰属する対象 b が存在しないことなのです. そのことから, 先程の Frege の言い換えが得られるのです.

次に概念 F を '$\_\_\_\_\_\xi \frown \Gamma$', 概念 G を同様に '$\_\_\_\_\_\zeta \frown \Delta$', 関係 φ を '$\_\_\_\_\_\xi \frown (\zeta \frown \Upsilon)$' とすると, 先程の「関係 Υ が概念 F と概念 G に帰属する対象を対応付けること」は次の式で表記されます:



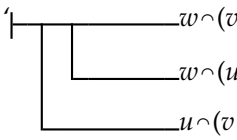
次に, この対応関係は「一意 (多対一)」の関係であるべきです. そのためには ‘Iϒ’ が真でなければなりません.
以上の考察から次の式をえます:



この式の外延を取出して束縛変項で置換えて項 ϒ の函数とみなしたものが ‘⟩ϒ’ になります. そして, ‘Γ⌢(Δ⌢⟩ϒ)’ が真のときに「ϒ-関係は Γ– 概念を Δ– 概念に写像する」と言います. これで概念 Γ から概念 Δ への単射の存在が表現できました. すると, 今度は概念 Δ から概念 Γ への単射の存在が必要になります. そのために逆関係 “⅄” を導入します.

☆ ⅄p　記号 “⅄” は「逆関係」を表現する記号です. この記号の定義式から判るように重積値域の変項を入れ替えたものとして表現されます.
もし, この定義式を現在の表記で一部を置換えると, 関係 $p(\xi,\zeta)$ は $⅄p \overset{def}{=} p(\zeta,\xi)$ で置換えられるでしょう.

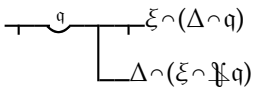
☆ $p \llcorner q$　この式は二つの関係 p と q の合成を意味します. すなわち, 関係 p, 関係 q に対して



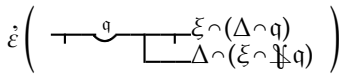
‘$w \overset{p}{\twoheadrightarrow} u \overset{q}{\twoheadrightarrow} v$’ が成立つというもので, この場合, 関係 p と関係 q が「$p \llcorner q$ 関係にある」と呼びます. なお, Frege はこの「$p \llcorner q$ 関係」を直観的に

基数

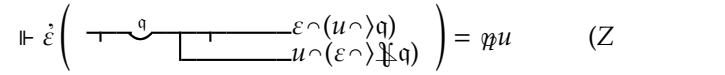
Frege は基数を概念 F と概念 G の等数性に訴えています. そのためには概念 F に属する諸対象が概念 G に属する諸対象に一意的に対応し, さらに, その逆も成立しなければなりません. このことから概念 F を ‘＿＿＿ξ⌢Γ’, 概念 G を ‘＿＿＿ζ⌢Δ’ とし, 概念 F と概念 G の一意な関係を p としましょう. すると, 等数性を充すためには「Γ-概念」を「Δ-概念」に写像する関係 p があれば, その逆関係 ⅄p によって「Δ-概念」は「Γ-概念」に写像されなければなりません. このことを纏めると, ある関係 $\mathfrak{q}$ が存在して ‘＿＿＿ξ⌢(ζ⌢p)’ と ‘＿＿＿ζ⌢(ξ⌢⅄p)’ の双方を充すので, 結局, 次の式が成立しなければなりません:

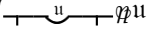


そして, この概念の外延は次の式で与えられます:

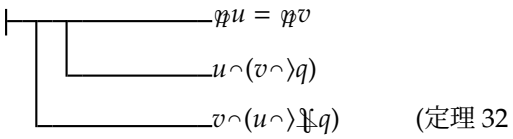


この概念の外延を「＿＿＿ξ∩Δ に帰属する基数」と呼びます. また, 簡単に「Δ-概念に帰属する基数」, あるいは「Δ-概念の基数」と呼びます. そして, 概念の外延 u に帰属する基数を $\wp u$ で定義します:



ここで ‘  ’ は「基数 Γ が帰属する概念 u が存在する」, つまり, 「Γ は基数である」と言えます. そして, この式から得られる函数 を「基数の概念」と呼びます.
さて, 基数の概念が出たところで今度は基数概念に対応する「Hume の原理」を Frege の式を使って記述しておきましょう:

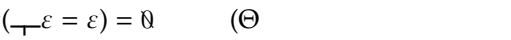
Hume の原理 ([50]§ 32 参照)



この式の意義は逆関係 $\backslash q$ によって概念 v は概念 u に写され, 関係 q によって概念 u は概念 v に写されるのであれば概念 u の基数 $\wp u$ と概念 v の基数 $\wp v$ が等しいということを主張しています. この「Hume の原理」を用いて Frege は自然数を構築してゆくのです.

自然数の構築

さて, Frege はどのようにして自然数を構築したのでしょうか, 実際にその構築を追ってみましょう. まず, 数 0 は「それに帰属する対象が存在しない概念に属する数」として定義し, 次のように表記します:



ここでの $\dot{\varepsilon}(\_\_ \varepsilon = \varepsilon)$ は「自分自身が異なる対象という概念の外延」ですが, このようなクラスは対象を何も含まないので, 集合論の空集合 $\emptyset$ に相当します. ここで集合論の自然数の構成でも 0 に $\emptyset$ が対応していましたが, Frege の定義もこれと似たものとなっています. そして, Frege は基数 $\mathfrak{0}$ と独特の表記を用いていますが, これは実際の数 0 と基数 $\mathfrak{0}$ が異なる数だからです. 何故なら, Frege

の基数は個数という量に関係する数であるのに対し, 実数は単位量の比として現われる数のために基数をそのまま実数に拡張することができないからです.
さて, 次の数 1 はどのような数になるでしょうか？ 数 1 は基数 $\mathfrak{0}$ と異なり,「帰属する対象を一つだけ持つ概念の外延」となります. そこで Frege はその対象が基数 $\mathfrak{0}$ に等しいものと定義しています:

$$\Vdash \mathfrak{p}\dot{\varepsilon}(\varepsilon = \mathfrak{0}) = \mathfrak{1} \qquad (I$$

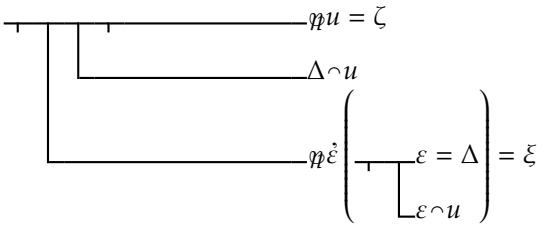
集合論に於ける自然数の構成は 1 に $\{\emptyset\}$ が対応していましたが, Frege の $\mathfrak{1}$ も同様に「$\mathfrak{0}$ のみを成員とするクラス」として定義されます. 以上から基数 $\mathfrak{0}$ と $\mathfrak{1}$ が定義されました.
残りの自然数を構築するために今度は「直続」という概念を導入しましょう:

自然数 n が自然数 m の直続であることの定義

ある概念 F と, その下に属するある対象 x が存在し, 自然数 n が概念 F に帰属する基数であり, かつ, 概念 F に帰属するが x と等しくないという概念 G に帰属する基数が m であれば, 自然数 n は自然数 m に直続すると呼ぶ.

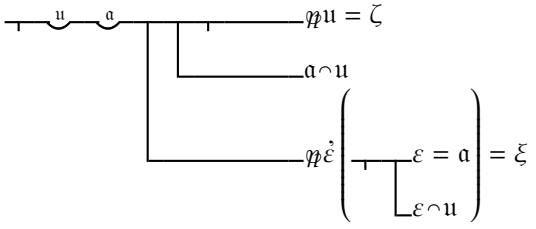
「自然数 n が自然数 m に直属する」とは, 簡単に言えば $n = m + 1$ となることです. このことは自然数 n が属する概念 F は自然数 m が属する概念 G よりも一つ対象が多いことを意味するので, したがって, 概念 G の対象を概念 F の対象と対応付けれてゆくと, 概念 F に帰属する対象 x が一つ余ってしまいます. 「直続」は, この性質を述べたものです.
さて, この直続関係を概念記法で表記してみましょう. そこで概念 u を「基数 n が帰属する概念」, 対象 Δ を「基数 m が帰属する概念に属さない対象であり, 概念 u に属する対象」とします. すると, $\mathfrak{p}\dot{\varepsilon}\left(\begin{array}{l} \varepsilon = \Delta \\ \varepsilon \frown u \end{array}\right)$ で基数 m が表記できます. そして,「Δ が概念 u に属する」は '$\Delta \frown u$' で表記できます.
次に基数 m が帰属する概念を ξ, 基数 n が帰属する概念を ζ とすると, 概念 ξ と概念 ζ は次の関係を充します:

$$\begin{array}{l} \mathfrak{p}u = \zeta \\ \Delta \frown u \\ \mathfrak{p}\dot{\varepsilon}\left(\begin{array}{l} \varepsilon = \Delta \\ \varepsilon \frown u \end{array}\right) = \xi \end{array}$$

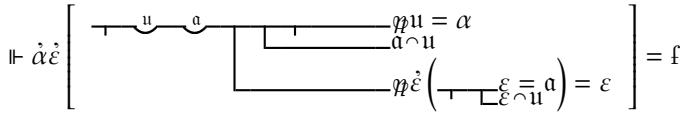


ここでは基数 n が帰属する概念について述べているので, 概念 u を, ある概念 $\mathfrak{u}$ で置換え, Δ もある対象 $\mathfrak{a}$ で置換えると次の関係をえます:

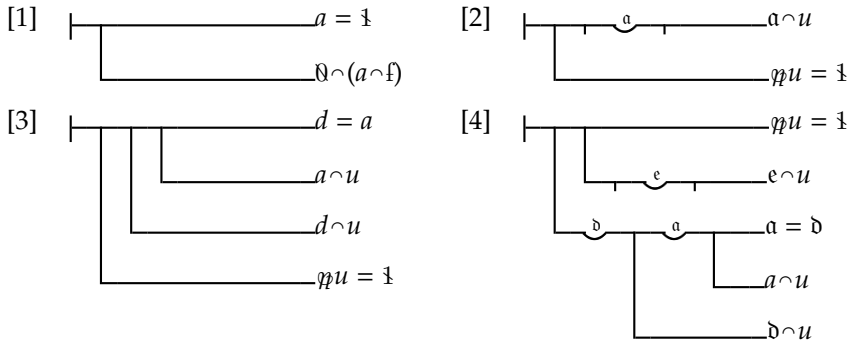
$$\begin{array}{l} \mathfrak{p}\mathfrak{u} = \zeta \\ \mathfrak{a} \frown \mathfrak{u} \\ \mathfrak{p}\dot{\varepsilon}\left(\begin{array}{l} \varepsilon = \mathfrak{a} \\ \varepsilon \frown \mathfrak{u} \end{array}\right) = \xi \end{array}$$



この関係の外延を「直続関係」𝔣 として定義します:



ここで 𝔒 と 𝔦 の関係を考えてみましょう. 基数 𝔒 に対応する概念は帰属する対象を何も持ちませんが, 基数 𝔦 に対応する概念は基数 𝔒 のみです. したがって, 基数 𝔦 は基数 𝔒 の直続関係, すなわち, '𝔒∩(𝔦∩𝔣)' が成立することが判ります.
この他にも次が成立します ([50]§ 44 参照):



[1] は「𝔒 と直属する基数は 𝔦 に限られる」ことを示しています. 次の [2] は「基数が 𝔦 に等しい概念 u に関しては, その概念に属する対象が存在する」ことを示しています. そして, [3] は「a と d が基数 𝔦 となる概念 u の下に属する対象であれば, $d = a$ である」ことを示します. [4] は「概念 u に属する対象が全て一致し, 帰属する対象が存在する場合, その概念が基数 𝔦 である」ことを示すものです.
さて, Frege の記号を用いずに基数を $\langle n \rangle$ と平易に表記てみましょう[92]. そして直続関係を用いると, 基数 $\langle 0 \rangle$ から $\langle 1 \rangle$ が定義され, 以下同様に基数 $\langle 2 \rangle$,$\langle 3 \rangle$,... とつぎつぎと定義されます.
では, Frege の方法による自然数の定義を次に纏めておきましょう:

Frege による自然数の定義

$\langle 0 \rangle$ =「概念$\xi \neq \xi$と同数である」という概念の外延
$\langle 1 \rangle$ =「0 と同一」という概念に属する数
$\langle 2 \rangle$ =「0 又は 1 と同一」という概念に属する数
$\langle 3 \rangle$ =「0 又は 1 又は 2 と同一」という概念に属する数
⋮
$\langle n \rangle$ =「0 又は 1 又は ... 又は $(n-1)$ と同一」という概念に属する数

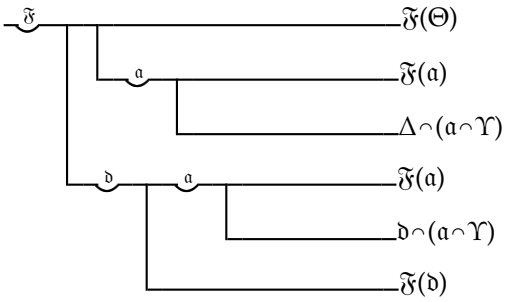
勿論,「直続」があれば「後続」もあります. これは一種の性質の伝播を示す関係になります. この後続の定義を次に述べておきましょう:

[92] 基数 𝔒, 基数 *fgestruckone* があっても基数 2 以降のフォントが無いための苦肉の策

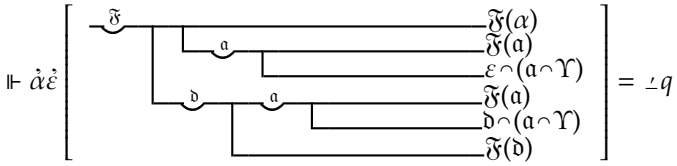
後続の定義

対象 Δ と対象 Θ に対し,Δ と ϒ-関係にある各対象が, 概念 ＿＿＿$F(\xi)$ に属し,
この概念 ＿＿＿$F(\xi)$ に属する対象と ϒ-関係となる対象も概念 ＿＿＿$F(\xi)$ に属し,
対象 Θ も概念 ＿＿＿$F(\xi)$ に属する場合,「対象 Θ は ϒ-系列に於て対象 Δ に後続する」と呼ぶ.

このことを概念記法で記述すると次の式を得ます:



さて, ここで対象 Δ と対象 Θ を変項 ξ と ζ で置換し, さらに関係 ϒ を関係 q で置換えると二項函数が得られます. ここで, この函数の重積値域を $\perp q$ と表記します.
すなわち, 次の定義式になります:

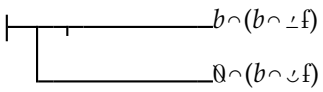


ここで Δ⌒(Θ⌒⊥ϒ) が真であれば「Θ が ϒ-系列に於て Δ に後続する」, あるいは「Δ は ϒ-系列に於て Θ に先行する」と呼びます. そして, この関係を強先祖関係と Frege は名付けています.

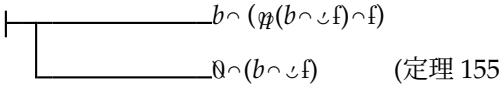
次に強先祖関係を用いて弱先祖関係と Frege が呼ぶ関係を定めます. まず, ＿＿＿Θ = Δ　　は Θ が
　　Δ⌒(Θ⌒⊥ϒ)
ϒ-系列に於て Δ に後続するか Δ と一致することの真理値となります. これを Frege は「Θ が Δ で開始する ϒ-系列に所属する」, あるいは,「Δ は Θ で終了する ϒ-系列に所属する」と呼びます. それから, この関係の外延を考えて弱先祖関係の記号 ⌣ を次で導入します:

$$\Vdash \dot{\alpha}\dot{\varepsilon}\left(\begin{array}{l} \alpha = \varepsilon \\ \quad \varepsilon \frown (\alpha \frown \perp q) \end{array} \right) = \smile q$$

ここで 𝔘⌒(Θ⌒⌣𝔣) は基数 Θ が基数 𝔘 から開始する基数列に所属することを意味し, この性質を持つ基数 Θ を有限基数と呼びます. そして, Θ⌒⊥ϒ で「Θ で終わる基数列に所属する」ことを意味し,「有限基数 n で終わる基数列に所属する基数」は ＿＿＿n⌒(𝔓(n⌒⌣𝔣)⌒𝔣) と記述できます.
　　𝔘⌒(n⌒⌣𝔣)
そして, 基数 𝔘 から開始する基数列に所属する対象は自分自身に後続しません. このことは概念記法で次のように表記できます:



さらに有限基数に対して Frege は次の定理 155 を述べています:



この式は基数 $\mathfrak{0}$ から開始し, 基数 b で終了する基数列に所属する基数は b が有限基数であれば b に直続するという命題です.

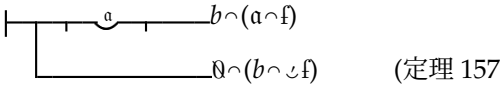
Peano の公理系との対照

では Frege の自然数の公理と Peano の自然数の公理の対比を行ってみましょう. なお, ここでは Peano のラテン語の論文の英訳「The principle of arithmetics,presented by a new method」([88]) からの本来の表記を併せて示しておきましょう:

☆第 1 公理 (0 の存在) 自然数の第 1 公理は数 0 の存在に関する公理です. Peano の本来の記述では $1\ \varepsilon\ \mathbb{N}$, 現在の表記では $1 \in \mathbb{N}$ と 0 ではなく 1 を用いています. なお, Peano 自身は 0 や 1 の定義を行わず, 無定義述語として用いています.
Frege の概念記法では基数 $\mathfrak{0}$ の定義 (Θ) が相当するでしょう:

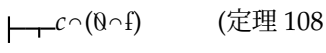
$$\Vdash \mathfrak{N}\dot{\varepsilon}(\neg\varepsilon = \varepsilon) = \mathfrak{0} \qquad (\Theta$$

☆第 2 公理 (後者の存在性) 自然数の第 2 公理は後者の存在性です. 本来表記で $a\ \varepsilon\ \mathbb{N}.\supset.a+1\ \varepsilon\ \mathbb{N}$, 現在の表記では $a \in \mathbb{N} \to a+1 \in \mathbb{N}$ となります.
これは Frege の次の定理 157 が対応するでしょう:

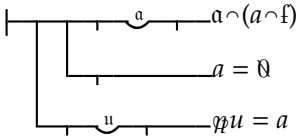


この定理は基数 $\mathfrak{0}$ から開始する基数列に所属する任意の対象 b に直続する対象が存在すというものです. これによって基数を次々と構築して行くことが可能になります.

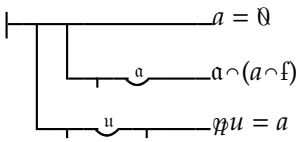
☆第 3 公理 (0 の先行者の非存在性) 本来の Peano の定義では公理 1 と同様に 1 に対するもので $a\ \varepsilon\ \mathbb{N}.\supset.a+1-=1.$, 現在の表記では $a \in \mathbb{N} \to \neg(a+1=1))$ と, 1 から逆に進めないことを示す公理です.
この公理には次の定理 108 が対応します:



次の定理 ([50]§ 44 の [6]) も挙げておきます:

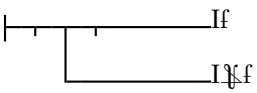


この定理の意味は基数 𝔘 を例外として, 各基数は基本列において直接の先行者が一つ存在するというものです. この定理の対偶として次が得られます:



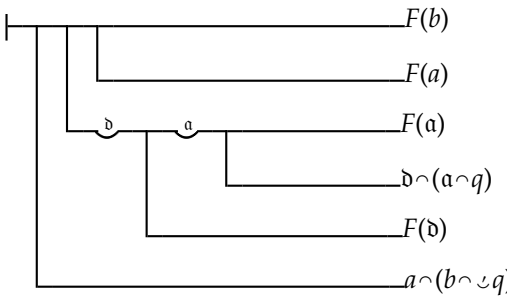
すなわち, 基数 a が先行する基数を持たなければ a は 𝔘 と結論付けられます.

☆第 4 公理 (後者関係の一意性)　第 4 公理は後者関係の一意性を示すもので
$a, b\ \varepsilon\ \mathbb{N}.\supset :\ a = b. = .a + 1 = b + 1.$, 現在の表記ならば $a, b \in \mathbb{N} \to a = b \equiv a + 1 = b + 1$ と表記されます.
この公理は ⊢ I 𝔣 (定理 78, と ⊢ I 𝔊 𝔣 (公理 89, の連言として表現されます:



ここでの I𝔣 は 𝔣-関係が一意的, すなわち, 基数列において直続する基数は一つ以上存在しないことを意味し, そして, I 𝔊 𝔣 も同様に後続する基数が一つしか存在しないことを意味しています.

☆第 5 公理 (数学的帰納法)　数学的帰納法は Peano の本来の表記では
$k\ \varepsilon\ K.\therefore\ 1\ \varepsilon\ k\ .\therefore\ x\ \varepsilon\ \mathbb{N}.\ x\ \varepsilon\ k : \supset_x.\ x + 1\ \varepsilon\ k :: \supset.\ \mathbb{N} \supset k.$, 現代の表記では
$((k \in K \wedge 1 \in k \wedge x \in \mathbb{N} \wedge x \in k) \to x + 1 \in k) \to \mathbb{N} \subset k$ となります.
Frege は後続関係を用いて表現しています:



(定理 144

この公理の意味は,a が b で終わる q-系列に属する基数で, 任意の 𝔰 に対して $F(\mathfrak{s})$ が成立し, 𝔰 と q 関係にある任意の基数 𝔞 に対して $F(\mathfrak{a})$ が成立するときに, $F(a)$ であれば $F(b)$ が成立するというもので

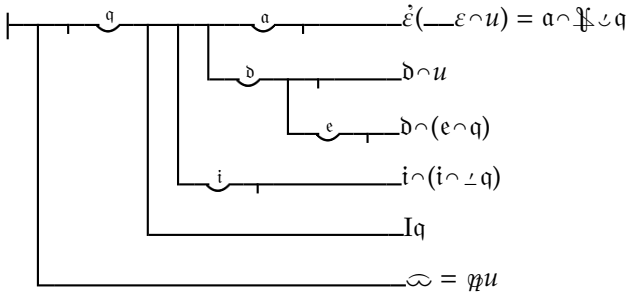
す. ここで q を $\mathfrak{f}$ で置換えると, $\mathfrak{d}\frown(\mathfrak{a}\frown q)$ が直続関係 $\mathfrak{d}\frown(\mathfrak{a}\frown\mathfrak{f})$ になるので, 数学的帰納法そのものになります.
このように, これらの命題から得られる Frege の算術は Peano の算術と同型となります.

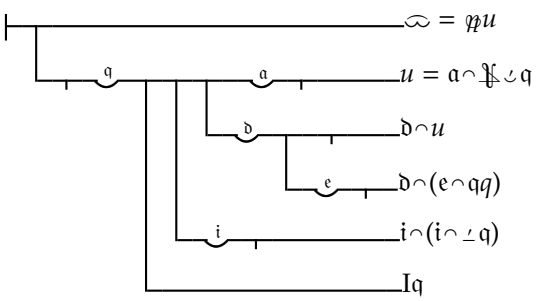
無限基数 $\frown\!\!\!\!\smile$

Frege の基数からも自然数が構築可能ですが, Frege はさらに進んで無限基数 $\frown\!\!\!\!\smile$ を次の式で定義しています:

$$\Vdash \mathfrak{p}\!\!\!\!\mathfrak{p}(\mathfrak{0}\frown\mathfrak{f}\!\!\!\!\smile\mathfrak{f}) = \frown\!\!\!\!\smile$$

ここで $\mathfrak{0}\frown(\mathfrak{f}\!\!\!\!\smile\mathfrak{f})$ は「有限基数という概念の外延」, すなわち, 「有限基数のクラス」[93] となります. そして, $\frown\!\!\!\!\smile$ は自分自身と $\mathfrak{f}$ 関係にある, すなわち, $\vdash\frown\!\!\!\!\smile\frown(\frown\!\!\!\!\smile\frown\mathfrak{f})$ を充し, さらに $\vdash\!\!\!\!\dashv\mathfrak{0}\frown(\frown\!\!\!\!\smile\frown\smile\mathfrak{f})$ が成立します. このことから, $\frown\!\!\!\!\smile$ は $\mathfrak{0}$ から開始する基数列に所属しないことを意味します. そして, この $\frown\!\!\!\!\smile$ が集合論の $\aleph_0$ に相当するのです.
この $\frown\!\!\!\!\smile$ に対しては次の二つの命題を証明しています:





このように Frege は一見して無限とは無関係そうな「Hume の原理」だけで無限列を構成し, 最終的に $\aleph_0$ に到達しているのです.

4.19.4 破綻

ざっと「算術の基本法則」を眺めてみました. この Frege の体系は非常に強固なものに思えますが, その強固さには思わぬ脆弱性が潜んでいました.

[93] 集合論的には $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \ldots\}$

この「算術の基本法則」は表記の複雑さ故に出版を渋られたため,「算術の基本法則」を二巻に分け, 第 I 巻の売上によって第 II 巻を刊行するという方針で, やっと 1893 年に第 I 巻を出版します. しかし, 第 II 巻は結局, その 10 年後の 1903 年に自費で出版する羽目になっています. その上, 第 II 巻の校正中の 1902 年に,Russell から「Russell の逆理」に関する手紙を受け取ります[94]. この逆理は Frege の論理主義の公理 V から導き出せるもので, 付録として対処法をなんとか第 II 巻に載せたものの, 彼の没後 10 年後に, この対処法でも逆理が導出可能なことが示されています.
この「Russell の逆理」の解説は II 巻の付録にあります. ここは Frege による説明を追ってみましょう. なお,Frege は真理性が疑わしいために判断線 ‘$\vdash$’ を落した式を用いています.
まず, Δ が「自分自身に属さないクラス」であるということを, 次の式で表現しています:

$$\vdash \overset{\mathfrak{g}}{\smile} \begin{array}{l} \mathfrak{g}(\Delta) \\ \dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \Delta \end{array}$$

どうして, この式で表現できているのか, その理由を解説しておきましょう.
まず, ‘$\dot{\varepsilon}(\_\_\_\mathfrak{g}(\varepsilon)) = \Delta$’ は $\mathfrak{g}$ の外延が Δ に等しいことを意味します. そして, $\vdash\_\_\mathfrak{g}(\Delta)$ により, Δ は $\mathfrak{g}$ の外延に属さないことを意味します. だから, これらを組合せると Δ がある概念の外延 (クラス) であるが, そのクラスには属さない, すなわち, 自分自身に属さないクラスとなるわけです.
この式の Δ を ξ に置換えると, この式は一変数の函数になります. こうすることで自分自身に属さないクラスのクラスは次の函数の値域として表現されます:

$$\dot{\varepsilon}\left(\vdash \overset{\mathfrak{g}}{\smile} \begin{array}{l} \mathfrak{g}(\varepsilon) \\ \dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \varepsilon \end{array} \right)$$

このクラスを $\forall$ と名付けて, このクラスに対し公理 V の片割れの公理 Vb

$$\vdash \begin{array}{l} f(a) = g(a) \\ \dot{\varepsilon} f(\varepsilon) = \dot{\alpha} g(\alpha) \end{array}$$

を適用すると次の式が得られます:

$$\begin{array}{l} (\_\_f(\forall)) = \vdash \overset{\mathfrak{g}}{\smile} \begin{array}{l} \mathfrak{g}(\forall) \\ \dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \forall \end{array} \\ \dot{\varepsilon}\,(\_\_f(\varepsilon)) = \dot{\varepsilon} \left(\vdash \overset{\mathfrak{g}}{\smile} \begin{array}{l} (\varepsilon) \\ \dot{\varepsilon}\,(\_\_\mathfrak{g}(\varepsilon)) = \varepsilon \end{array} \right) \end{array}$$

この式に対して省略式 $\forall$ を用いると次の式が得られます:

$$\begin{array}{l} (\_\_f(\forall)) = \vdash \overset{\mathfrak{g}}{\smile} \begin{array}{l} \mathfrak{g}(\forall) \\ \dot{\varepsilon}\,(\_\_\mathfrak{g}(\varepsilon)) = \forall \end{array} \\ \dot{\varepsilon}\,(\_\_f(\varepsilon)) = \forall \end{array}$$

[94] その手紙の内容は [33],p.204 や書簡集 [51] にあります

そして, 公理 (IIIa) $\vdash f(a)$ から得られる式に対し, 推論規則 MP を適用します:

$f(b)$

$a = b$

公理 IIIa

a | $\_\_f(\forall)$

b | $\mathfrak{g}$ $\mathfrak{g}(\forall)$ $\dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \forall$

f | $\_\_\xi$

$(\_\_f(\forall))$ $\mathfrak{g}$ $\mathfrak{g}(\forall)$ $\dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \forall$

$(\_\_f(\forall)) =$ $\mathfrak{g}$ $\mathfrak{g}(\forall)$ $\dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \forall$

$(\_\_f(\forall)) =$ $\mathfrak{g}$ $\mathfrak{g}(\forall)$ $\dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \forall$

$\dot{\varepsilon}(\_\_f(\varepsilon)) = \forall$

$(\_\_f(\forall))$ $\mathfrak{g}$ $\mathfrak{g}(\forall)$ $\dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \forall$

$\dot{\varepsilon}(\_\_f(\varepsilon)) = \forall$

そして, 結果の条件を入れ替えると次の式が得られます:

$f(\forall)$

$\dot{\varepsilon}(\_\_f(\varepsilon)) = \forall$

$\mathfrak{g}$ $\mathfrak{g}(\forall)$

$\dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \forall$

さらに, f は任意の概念なので, この f に対して全称化を行ってみましょう.
すると, 次に示すように見事に矛盾が生じています!

$\mathfrak{g}$ $\mathfrak{g}(\forall)$

$\dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \forall$

$\mathfrak{g}$ $\mathfrak{g}(\forall)$

$\dot{\varepsilon}(\_\_\mathfrak{g}(\varepsilon)) = \forall$

ここで問題となるのは排中律と公理 V ですが, Frege は分析を進めることで公理 V が偽であると最終的に判断しています.
この公理 V は簡単に言えば,「函数 $\Phi(\xi)$ は函数 $\Psi(\xi)$ と同じ値域を持つ」ということと「函数 $\Phi(\xi)$ 函数 $\Psi(\xi)$ とは同じ項に対して常に同じ値を持つ」ということが同値であることを保証する公理です. 基数の定義で判るように, 外延 (値域) を根底に置いて自然数の定義を行っているために, 公理 V は体系全体に影響を与えてしまいます.
なお, この公理 V に対しては公理的集合論では制限を加え, Russell の論理主義では分岐的階型理論を導入することで「Russell の逆理」を排除しています. Frege は値域を持たない概念の存在 (公理的集合論の方法) や型の理論を用いた解決を示唆しますが, 最終的に Frege の取った手段はそのどちらでもありませんでした.

それは公理 (V): $\vdash(\dot{\varepsilon}f(\varepsilon) = \dot{\alpha}f(\alpha)) = (\overset{\frown}{\mathfrak{a}}f(\mathfrak{a}) = g(\mathfrak{a}))$ に制限を入れて次の公理 (V′) で置換えるものです:

$$\vdash(\dot{\varepsilon}f(\varepsilon) = \dot{\alpha}f(\alpha)) = \overset{\frown}{\mathfrak{a}}\begin{array}{l} f(\mathfrak{a}) = g(\mathfrak{a}) \\ \quad \mathfrak{a} = \dot{\varepsilon}f(\varepsilon) \\ \quad \mathfrak{a} = \dot{\alpha}g(\alpha) \end{array} \qquad (V'$$

これに伴い, 公理 Vb も次の公理 V′b で置換えられます:

$$\begin{array}{l} \vdash f(a) = g(a) \\ \quad a = \dot{\varepsilon}f(\varepsilon) \\ \quad (\dot{\varepsilon}f(\varepsilon) = \dot{\alpha}f(\alpha)) \end{array} \qquad (V'b$$

ただし, この解決方法も Frege の没後 10 年後に矛盾が導出されています.
結局, この逆理に対する根本的な対処ができずに Frege の論理主義は破綻してしまいます. ただし, 近年の研究では外延を用いずに「算術の基礎」で最初に取り上げられた数の言明と所謂「Hume の原理」を用いることで Peano 算術が遂行可能なことが示されています[95]. そのこともあって再評価が進んでいるそうです (田畑 [33]).

4.20　Russell の階型理論

Frege の論理主義は「Russell の逆理」で大きく頓挫しましたが, Frege と入れ替わるように今度は Peano 流儀の表記を導入した Russell が強力に論理主義を押し進めます.
Russell は非常に長生きした哲学者です. ただし, 数理論理学に大きな影響を与えた期間は 20 世紀の四半世紀の間です. 最初に Russell は 1900 年の Paris で開催された第 1 回国際哲学者会議で Peano の記号論理学に興味を持って研究を開始し, その成果は「Principles of Mathematics」([90]) に反映されます. この「Principles of Mathematics」では論理学から数学全体の導出を行っていますが, この「Principles of Mathematics」の完成間際に「Cantor の逆理」の研究から「Russell の逆理」を発見し, 大きな挫折感を味わうことになります.

Bertland Russell(1872-1970)

そののち, 逆理に対処するために「型の理論」(「Mathematical logic as based on the theory of types(1908)」([81],p.150-182 に収録)」(TT と略記) を考案し, 「Principia Mathematica([92],PM と略記) でその実験を行います. この Russell の論理主義に関してはいろいろな批判もあります. ここでは TT と PM のさわりの部分を簡単に眺めてみることにしましょう.

[95] なお, Russell も分岐的階型理論を導入する前に, 無クラス理論で Hume の原理に似た原理を用いて算術の構築を試みたことがありますが, この試みは失敗しています. 詳細は [51] の Russell から Frege への書簡を参照.

4.20.1 項, 概念, 個体

項 (term) は命題の主語となりうる対象です. それから概念 (concept) は命題の述語や関係です (TT.p.164 参照). そして, 後述の基本命題は一つ, あるいはそれ以上の概念 (Concept) によって一つ, あるいはそれ以上の項 (term) に分類されます.
個体 (individual) は TT では基本命題の項として定義され, PM では函数にも命題にもならない対象です. そして個体は階型理論ではその最下層を構成する対象になります.

4.20.2 変項と函数

Russell の函数の考え方は Frege の函数を特徴付ける「不飽和」といった考え方とやや異っています. Russell によると, 命題函数 φx は, それ自身を構成する不定値 (曖昧値)x に対して意味のある値を確定することで命題となる陳述です. ここで不定値 x を (実際の) 変項と呼びます. ここで Russell によると変項は幾つかの意味のある値を予め持っており, それらの値の一つを曖昧に表現する対象ですとしての性格があります. そのため, 変数を不定値, あるいは曖昧値と呼んでいます. そして, 曖昧値) を持つ変項を $x, y, z, \ldots$ で表現し, 逆に確定した意味のある値を持っている場合に $a, b, c, \ldots$ を用いて記述します. これは函数も同様で, ギリシャ文字 $\varphi, \psi, \chi, \ldots$ を用いて函数変項を表現し, 定項函数や具体的な値を持った函数は $f, g, \ldots$ のようにラテン小文字を用います.
ここで Russell の命題函数 φx は $\varphi a, \varphi b, \varphi c, \ldots$ といった具体的な値の何れかを曖昧に表記します. たとえば, 「x は人間である」は「Socrates は人間である」,「Aristotle は人間である」等々を前提とし, さらに, これらの何れかを指示するものです. このように命題函数はその全ての値がきちんと定義されていなければ, きちんと定義された函数とは言えません.
したがって, φx は函数自体ではなく, 曖昧値 x による値を示す記号です. そこで, Russell は函数本体を表現する記号として $\varphi\hat{x}$ を導入して, この $\varphi\hat{x}$ を命題函数と呼び, φx を命題函数 $\varphi\hat{x}$ の曖昧値と呼びます. ここで曖昧値 φx に現われる変項 x を「実際の変項 (real variable)」と呼びます.

4.20.3 PM の論理式

Russell は Frege 風の概念記法ではなく, Peano 流儀の線的な表記を用いています. 次に幾つかの表記上の約束を示しておきましょう:

1. 命題函数:命題函数は変項が x の場合は φx, 変項が $x, y, z, \ldots$ であれば $\varphi(x, y, z, \ldots)$ と表記
2. 命題 p の否定は $\sim p$ と表記
3. 二つの命題の論理和は $p \vee q$ と表記
4. 任意の変項に対して命題函数 φ が真の場合, 不定値の x を用いて φx と表記
5. 全ての変項に対して命題函数 φ が真の場合, $(x).\varphi x$ と表記
6. 証明可能なことは $\vdash$ で表記
7. 定義は論理式の末尾に “Df.” を置く

では, 論理式で用いる幾つかの記号の定義を現在の表記と比較してみましょう:

基本的な記号の定義

Russell			現在の表記
$p \supset q \,.=.\, \sim p \vee q$	Df.	$\Leftrightarrow$	$(p \to q) \stackrel{def}{=} (\neg p \vee q)$
$p \,.\, q \,.=.\, \sim(\sim p \vee \sim q)$	Df.	$\Leftrightarrow$	$p \wedge q \stackrel{def}{=} f\neg(\neg p \vee \neg q)$
$p \equiv q \,.=.\, p \supset q \,.\, q \supset p$	Df.	$\Leftrightarrow$	$p \equiv q \stackrel{def}{=} (p \to q) \wedge (q \to q)$
$(\exists x).\varphi x \,.=.\, \sim\{(x) \,.\sim \varphi x\}$	Df.	$\Leftrightarrow$	$\exists x[\varphi x] \stackrel{def}{=} \neg[\forall x(\neg(\varphi x))]$
$x = y \,.=:\, (\varphi) \,:\, \varphi!x. \supset .\varphi!y$	Df.	$\Leftrightarrow$	$x = y \stackrel{def}{=} \forall\varphi(\varphi!x \supset \varphi!y)$

このように Russell の表記は現在のものとほぼ同じ表記になっています. たとえば Peano の含意 $\supset$ は $\supset$, Frege の判断 $\vdash$ からは $\vdash$ を導入しています. ただし, 論理式の区切に括弧を用いずに区切記号として “.”, “:”, “:.”, “::” 等を用いている点が多少異なります.

PM で定義式を示す Df の使い方は $p \supset q.=.\sim p \vee q$ Df. のように式の右端に置いて, 記号 = の左側に新しい表記, ここでの例では $p \supset q$ を記述し, 左側の式の意味を既に定義された記号を用いて記述したもの, この例では $\sim p \vee q$ を = の右側に記述します.

PM では「全ての x」“(x)” や「x が存在する」“$(\exists x)$” といった量化詞を導入しています. そして, これらの量化詞に対しては次の略記を認めています:

量化詞に関連する略記

$\varphi x. \supset_x .\phi x :=: (x) : \varphi x. \supset .\phi x$	Df.
$\varphi x. \equiv_x .\phi x :=: (x) : \varphi x. \equiv .\phi x$	Df.
$(x, y).\varphi(x, y). =: (x) : (y).\varphi(x, y)$	Df.
$\varphi(x, y). \supset_{x,y} .\phi(x, y) :=: (x, y) : \varphi(x, y). \supset .\phi(x, y)$	Df.

4.20.4　基本命題と明瞭な変項

基本命題 (Elementary proposition) は変項, あるいは「明瞭な変項 (apparent variable)」を持たない命題です. ここで「明瞭な変項」とは, 命題を分析することで現われる変項で, 「全ての××」や「ある××」といった形で命題に現われます. たとえば「全ての人間は死すべき存在である」といった命題を考えましょう. この命題には「全ての人間」という文言がありますね. この命題を「全ての x に対し,x は人間であり, かつ, x には寿命がある」と言い換えると, 今度は「寿命」という変項も見えてきます. この命題を現在の表記で書き換えると「$\forall x$[x は人間 $\wedge$ $\exists t$(x は寿命 t を持つ)]」となります. このように分析することで表に現われた二つの変項 “x” と “t” が「明瞭な変項」になります. ここで $\forall x\, \phi(x)$ や $\exists y\, \phi(y)$ を PM の体系で表記すると, $(x).\phi x$ や $(\exists y).\phi y$ となります. なお, 変数 x, y を現在は (量化詞の) 束縛変項と呼びます.

最後に基本命題を構成する項を変項で置換えて造られる函数を, 基本命題函数 (Elementary propositional function) と呼びます.

4.20.5 命題と函数の階層

ここでは最初に Russell の分岐的階型理論で見られる階と型について解説しましょう. なお, 型理論は後の計算機言語にも影響を与えています.
型は函数の変項の取り得る値 (意味の値域,range of significance) として現れます. 函数はこの値域内部で意味を持ち, この値域の外部では函数は無意味 (nonsense) になります.
ここで個体が型の最下層を構成します. 本来の個体は命題でも函数でもない対象を特にさします. ところが, この型の理論ではより広い意味を持たせて, それ単独で存在し得る命題の構成要素のことで, 相対的な型の最下層を構成する対象 (個体, あるいは論理式や函数) も意味します.
次に基本命題に含まれる個体に対して量化を行って新しい命題が得られます. たとえば「人間である Socrates は死すべき存在である」から「人間である全ての x は死すべき存在である」が得られますが, この命題が成立するためには「人間である Socrates は死すべき存在である」, 「人間である Aristotle は死すべき存在である」等が前提となり, これらの真実の上に量化した命題が成立します. したがって,「人間である Socrates は死すべき存在である」が真であることと, それを前提とする「全ての人間は死すべき存在である」が真であることは異なったものになります. すなわち, 量化された命題は元の命題よりも一つ上の階層の命題になります. つまり, 階数が n の命題から $n+1$ の階の命題が構成されます[96]. また, この議論から判るように真偽値も階を持ち, 異なった階の真偽値は異なった対象になります. 実際,「人間である Socrates は死すべき存在である」が取る値の真理値の「真」が第 1 階の「真 $true_1$」であれば,「全ての人間は死すべき存在である」はこれら第 1 階の「真」の上に成立つために第 2 階の「真 $true_2$」となるのです.
ここで, 二つの対象 u,v の型が等しくなるのは次の場合です[97]:

1. u, v が共に個体である場合
2. u, v が共に同じ型の変項を持つ基本命題の場合
3. u を函数とするときに v がその否定である場合
4. u を基本命題 φx か ψx の何れかで, v が $\varphi x \vee \psi x$ の場合
5. $\varphi(x,y)$ と $\psi(x,y)$ を同じ型の函数とするときに u を $(x).\varphi(\hat{x},y)$,v を $(x).\psi(\hat{x},y)$ とする場合
6. u, v の双方が基本命題の場合
7. φx と ψx を同じ型の函数とするときに u を $(x).\varphi x$,v を $(x).\psi x$ とする場合

さて, 基本命題 $\varphi\hat{x}$ に含まれる項 a に対する代入操作で命題函数が構築されると述べました. この代入操作で得られる命題函数を $\varphi/a : x$ と記述し, 命題 φ を雛形, そして, φ/a を母体 (matrix) と呼びます. それから基本命題 φ に含まれる個体 a の他の個体 $b, c \ldots$ を用いて, $p/(a,b), p/(a,b,c), \ldots$ と母体が構築されます (TT 参照). そして一つの変項を個体とする母体 φx を特に $\varphi!x$ と記述して「可述的函数」とも呼びます.
ここで可述的函数は, 可述的函数の否定, 可述的函数同士の選言や含意で構成することが可能です:

[96] Poincaré:「…序数にてもあれ, 基数にてもあれ, 数形容詞や, また「数個の」という如き不定形容詞がいくつ位含まれているかを戯れにかぞえて見ることは止めておこう」([52],p.175)
[97] Russel2,*9.131,p.133

可述的函数の構成例 ([92],p.163)

$\sim \varphi!a$, $\sim \varphi!x$, $\varphi!x \vee \varphi!y$, $\varphi!x \vee \psi!x$
$\varphi!x \vee \psi!y$, $\varphi!x. \supset .\psi!x$, $\varphi!x.\psi!x$, $\varphi!x. \vee .\psi!y \vee \chi!z$ 等々

これらの可述的函数 (母体) は第 1 階函数と呼ばれるものになります. そして母体を使って函数と命題に対して Russell は次の階層構造を導入します ([92],p.163-164):

- 第 1 階母体, 函数と命題:
  (a) 第 1 階母体: 個体を変項として持つ命題
  (b) 第 1 階函数: 第 1 階母体から構成される函数. 変項は全てではなく, 一部が量化されていても構いません. ここで一変項の第 1 階の函数を $\varphi!x$ と表記します
  (c) 第 1 階命題: 第 1 階母体の変項を全て量化することで得られる命題
- 第 2 階母体, 函数と命題:
  (a) 第 2 階母体: 少なくとも一つの第 1 階母体を変項として含み, 第 1 階函数や個体変項以外を含まない命題
  (b) 第 2 階函数: 第 2 階母体から構成される函数. 第 2 階母体の変項は全てではなく, 一部が量化されていても構いません. 具体的には次の形が挙げられます:
    1. 函数 $\varphi!\hat{z}$ を変項の値域とする函数:
       $(x).\varphi!x$,$(\exists x).\varphi!x$, $\varphi!a. \supset .\varphi!b$, $\varphi!x. \supset_x .g!x$ 等. ここで $g!x$ は定項函数とします.
    2. 函数 $\varphi!\hat{z}$ と $\psi!\hat{z}$ を変項の値域とする函数:
       $\varphi!x. \supset_x \psi!x$, $\varphi!x. \equiv_x .\psi!x$, $(\exists x).\varphi x.\psi x$, $\varphi!a. \supset .\psi!b$ 等. ここで a と b は定項とします.
    3. 個体 x を変項の値域とする函数:
       $(\varphi).\varphi!x$, $(\exists\varphi).\varphi!x$, $\varphi!x. \supset_\varphi .\varphi!a$ 等. ここで a は定項とします.
    4. 個体 x と函数 $\varphi!\hat{z}$ を変項の値域とする函数:
       $\varphi!x$, $\varphi!x. \supset .\varphi!a$ 等. ここで a は定項とします.
  (c) 第 2 階命題: 第 2 階母体の変項を全て量化することで得られる命題.

以降, 同様にして第 n 階母体を変項の値域とする第 $n+1$ 階母体が得られます:

- 第 n+1 階母体, 函数と命題:
  (a) 第 n+1 階母体: 第 n 階母体を変項として含み, 第 n 階以下の函数と個体変数以外を含まない命題.
  (b) 第 n+1 階函数: 第 n+1 階母体から構成される函数. 一部が量化されていても構いません.
  (c) 第 n+1 階命題: 第 n+1 階函数の実際の変項の全てを量化して得られる命題.

さて, 個体を値域とする母体を可述的函数と呼びましたが, より一般的に定義することが可能です. すなわち, 命題が可述的 (predicative) であるとは, いかなる明瞭な変項も含まず, 変項の階数が n で

あれば命題の階数が $n+1$ となる命題のことです. この可述的な命題 φ を PM では $\varphi!x$ と表記します. なお, PM 体系独特なことに, $\varphi!x$ には変項 x と函数 $\varphi\hat{x}$ の二つの変項があることです.
なお, Frege の 1 階の函数は変項が取りうる値が個体, 2 階の函数は 1 階の函数を変項として取りうる函数で, 一見すると Russell の函数の階と似たものですが, その中身はやや異なったものです[98].

4.20.6 悪循環原理による非可述的述語の排除

さて, Poincaré の言う非可述的な言明による逆理に対しては, 次の「悪循環原理」で, その発生を封じ込めようとしています. この悪循環原理は Gödel によると次の三種類があります:

悪循環原理

1. ある集まりの全てを含む物は, その集まりの一つの元であってはならない.
2. ある全体を有している集まりが, その全体でのみ定義される元を持つのであれば, その集まりは何如なる全体も持たない.
3. いかなる全体も, この全体によってのみ定義可能, あるいは, この全体を含む, または, この全体を前提とする元を持たない.

Russell はこれらの原理を使い分けていますが, その中で最初の 1. が Poincaré の分析を受けたものになります.
さて, この悪循環原理によって最初に挙げた逆理はどのようになるでしょうか？まず, Russell の逆理: 「自分自身を含まない全ての対象」は悪循環原理 1. によって自分自身を含むことができなくなるため, Russell の逆理は見事に PM の体系から排除されます.
次に Epimenides の嘘つきの逆理: 「自分が主張する命題は全て嘘である」に対しては, まず, この嘘つきの逆理を φ とします. ここで命題 φ の中の個体の「命題」を第 1 階命題とすれば, φ を雛形として得られた述語 φ/命題 : x は変項として第 1 階命題を取る函数なので第 2 階命題函数となります. ここで悪循環原理によって, 変項 x が取りうる値域から「自分が主張する全ての命題」は除外されます. そして,「自分が主張する第 1 階命題 x は嘘である」という第 2 階命題函数は, 彼が主張する第 1 階の命題が嘘であろうがなかろうが, 第 2 階の真理値とは階が違うために逆理が回避されます.
これで Russell の逆理の排除や嘘つきの逆理もめでたく解決できたように見えますが, この悪循環原理で排除される命題には無害なものや, 重要な成果さえも失われてしまう副作用があります. たとえば,「3 年 2 組で一番背の高い人」という命題は, 個人をその個人が所属する全体「3 年 2 組」という類 (クラス) に言及することで指定しているので, これは悪循環原理では排除されるべき命題になります. さらに解析学の定義では循環論法を用いたものがあるため, これらも無効になってしまいます. さらには分枝的階型理論によって数学的帰納法さえも利用できなくなります.
たとえば,「0 で性質 F を持ち,n で性質 F を持つのであれば n の後者も性質 F を持つのであれば, 全ての自然数は性質 F を持つ」は「0 が性質 F を持つ」,「n が性質 F を持つ」と「n の後継者が性質 F を持つ」は全て 1 階の命題ですが,「全ての自然数が性質 F を持つ」の「全ての自然数」は「悪

[98] [51], Frege = Jourdain 往復書簡での PM に関する Frege の感想等

循環原理」によって命題「自然数 0」と同じ階には置けずに 1 階の対象となります. そのため,「全ての自然数が性質 F を持つ」は 2 階の命題となって階が異なるために結論付けることができなくなり, その結果, 数学的帰納法が使えなくなるのです.

4.20.7　還元可能性公理

この対処方法として Russell は還元可能性公理 (Axiom of reducibility) を導入して, この難点を打開しようとしています. この公理を Russell の記号を用いて表記しておきましょう:

還元可能性公理 (還元公理)

$$\exists\varphi : .(x) : \phi x. \equiv_x .\varphi!x.$$

この還元可能性公理の意味は「任意の命題函数 ϕ に対して同値な述語 φ が存在する」であり, 現在の表記では $\exists\varphi[\forall x(\phi(x) = \varphi!x)]$ となるでしょうか. なお, ":." や "." は Peano 流の区切記号で式の区切を表現しています.
この公理によって任意の命題函数には, その対象よりも 1 階上の集まり, つまり, クラス (類) の存在が保証されます. ただし, この公理の難点は, その述語が存在すると主張するだけで, その構成方法を述べたものではないこと, さらに, この還元可能性公理自体が実は非可述的な性質を持っていることです. また, 還元可能性公理は命題函数を充すクラスを, その命題と同値な述語で一時的に定義し, その述語を暫く使って元の命題函数に戻すといったことを都合良く使っており, 幾分,「機械仕掛けの神 (Deus ex Machina)」的でさえあります. さらに, この「還元可能性公理」は「クラス」を否定する性質を持っています ([11],p.203-206). 実際, PM の体系でクラスは便宜的に用いられ, 必要であれば消すこともできる対象となっているからです[99].
なお, Gödel の不完全性定理の証明では,Principia Mathematica の公理系の中で還元可能性公理を用いずに, 次の「集合の内包公理」を採用しています[100]:

集合の内包公理

$$\exists u[\forall v(u(v) \equiv a)]$$

ここで v は n 階の変数,u は $n+1$ 階の述語, a は u を含まない $n+1$ 階の命題になります.

4.20.8　クラスについて

Russell は函数 $\varphi\hat{z}$ を充すクラス (類) を $\hat{z}(\varphi z)$ で表記します (*20.01). この表記は Frege の $\grave{\alpha}\varphi(\alpha)$ に似たものですね. そして, クラスへの帰属は Peano 流儀で記号 $\in$ を用います. たとえば $x \in A$ を「x は A である (x is A)」と読みます. また, クラスはギリシャ文字で "$\phi, \psi, \kappa, \theta$" のように函数で用いられるものや "$\iota, \epsilon$" のように記号で用いられる文字を除いた小文字 "$\alpha, \beta \ldots$" を用います.
このクラスの定義は「還元可能性公理」に訴えて次で定めています.

---

[99] [51] に収録された Russell からの書簡を見る限り, できるだけクラスを用いずに体系化しようとする様子が伺えます.
[100] [55] でも「還元可能性公理」ではなく, この集合の内包公理を用いています. これは [55] が Gödel の不完全性定理の証明に繋がるように工夫されているためです.

命題函数と関係のクラス

$f\{\hat{z}\phi(z)\}. =: (\exists\varphi) : \varphi!x. \equiv_x .\phi x : f\{\varphi!\hat{z}\}$ Df.
$f\{\hat{x}\hat{y}\phi(x,y)\}. =: (\exists\varphi) : \varphi!(x,y). \equiv_{x,y} .\phi(x,y) : f\{\varphi!(\hat{x},\hat{y})\}$ Df.

このクラスの定義では, クラス $\hat{z}(\phi z)$ を直接定義するのではなく, クラスの属性を意味する函数 f を含めて $f\{\hat{z}(\phi z)\}$ で定義しています. 何故なら $\hat{z}(\phi z)$ 単体だけで定義することは無意味で, 函数 f を含めて定義することではじめてクラスが意味を持つからです. さらに命題函数 ϕ のクラス $\hat{z}(\phi z)$ は還元可能性公理がその存在を保証する命題函数 ϕ と同値な可述的函数 φ を用いて定義します. そのために Russell のクラスは非常に便宜的なものとなります. と言うのも, 必要に応じて可述的函数を切り換えれば良いからです.
さて, 対象 x がクラス $\hat{z}(\phi z)$ に帰属することも同様に定めます:

対象の帰属

$x \varepsilon \hat{z}(\phi!z). = .\varphi!x$ Df.

この定義の意味は「個体 a が命題函数 ϕ のクラスに所属することは ϕ と同値な述語函数 φ に対し,$\varphi!a$ が真となることである」になります. これは Frege の $\Delta\smile p$ の定義と似たものですが, ここでも還元可能性公理が介在しています.
実際, 一般の命題函数 ϕ のクラス $\hat{z}\phi(z)$ に対象 x が帰属することは,

$$x \varepsilon \hat{z}(\phi z). =: (\exists\varphi) : \varphi!y. \equiv_y .\phi y : \varphi!x$$

と還元可能性公理を用いて $\vdash: x \varepsilon \hat{z}(\phi z). \equiv .\phi x.$ をえます.
ここでクラスに関する記号の定義をしておきましょう:

クラスに関する定義

$\mathrm{Cls} = \hat{\alpha}\{(\exists\varphi).\alpha = \hat{z}(\varphi!z)\}$ Df.
$Kl = \hat{\alpha}\{(\exists\varphi).\alpha = \hat{z}(\varphi!z.Indivi!z)\}$ Df.
$\alpha \subset \beta. =: x \varepsilon \alpha. \supset_x .x \varepsilon \beta$ Df.
$\exists!\alpha. = .(\exists x).x \varepsilon \alpha$ Df.
$\mathrm{V} = \hat{x}(x = x)$ Df.
$\Lambda = \hat{x}\{\sim (x = x)\}$ Df.

記号 Cls でクラスを定義しています. これは命題が定めるクラスのクラスです. そして , 記号 Kl はクラス Cls に包含される個体のクラスになります.
記号 “$\subset$” は現在の集合論で用いられる記号 “$\subset$” と同じ意味で, クラス同士の包含関係を示します.
次の記号 “$\exists!$” も現在の数理論理学で用いられる記号 “$\exists!$” と同じ意味で, 命題函数 ϕx を充す個体が一つのみ存在することを示します. え次の V と Λ は真理値の表現で, Frege の true ($\dot{\varepsilon}(\_\varepsilon = \varepsilon)$) と false ($\dot{\varepsilon}(\neg\_\varepsilon = \varepsilon)$) を思い出させるものです.
なお, 型の理論で重要なことに, 同じ型の対象で構成されるクラスは, それらの対象の上の階の型になります. これによって「クラスのクラス」はそのクラスの成員とは型が異なるために「クラスの

クラス」の成員から自動的に排除されます. このことから「Cantor の逆理」や「Russell の逆理」は PM の体系から排除されます.
次に, 定冠詞 “THE” に相当する記号 “$\imath$” があります. この記号の定義もクラスの定義と同様に函数 f を介在した定義となります:

定冠詞 “THE” に相当する記号$\imath$

$$f\{(\imath x)\phi x\}.\ =:\ (\exists c) : \varphi x.\equiv_x .x = c : fc \quad \text{Df.}$$

具体的には $(\imath x).x - 2 = 0$ は $x - 2 = 0$ となる x として 2 を指示します.

4.20.9　PM の公理系

PM では公理を原始命題 (Primitive Prooposition) と呼んでいます. それから原始命題の表記は定義のように末尾に Pp を置きます.
PM の公理系を次に示しておきましょう:

1. 命題 $p \supset q$ は q が真ならば真である
2. $\vdash : p \vee p. \supset p.$
3. $\vdash : q. \supset .p \vee q.$
4. $\vdash : p \vee q. \supset .q \vee p$
5. $\vdash : p \vee (q \vee r). \supset .q \vee (p \vee r)$
6. $\vdash : q \supset r. \supset : p \vee q. \supset .p \vee r$
7. $\vdash : (x).\varphi x. \supset .\varphi y$
8. $\vdash : \varphi y. \supset .(x).\varphi x$
9. $\vdash : (x).\varphi x. \supset .\varphi a$
10. $\vdash : .(x).p \vee \varphi x. \supset . \vee .(x).\varphi x$
11. $f(\varphi x)$ が任意の変項 x に対して真であり, かつ, $F(\varphi y)$ が任意の変項 y に対して真であれば,$\{f(\varphi x).F(\varphi x)\}$ は任意の変項 x に対して真である
12. 任意の可能な変項 x に対して $\varphi x.\varphi x \supset \phi x$ であれば, 任意の可能な変項 x に対して ϕx は真である
13. $\vdash : .(\exists f) : .(x) : \varphi x. \equiv .f!x.$
14. $\vdash : .(\exists f) : .(x, y) : \varphi(x, y). \equiv .f!(x, y)$

ここで 13 と 14 が「還元可能性公理」になります.
PM の体系は還元可能性公理, 無限公理, 選択公理の 3 個のあまり美的ではない公理を幾つか持っていますが最も体系化されたものです. この PM の体系を数学の形式化で用いたのが次に述べる Hilbert です.

4.21 Hilbertによる形式化

4.21.1 Hilbertの立場

Hilbert は 19 世紀末から 20 世紀半ばにかけて活躍したドイツの大数学者です. 彼の興味の根底に「数学の問題の可解性」があり, このことが数学の基礎付けを行う上での大きな原動力となっているようです ([30] の「解説 (4)」,p.118-170). Hilbert はまず 1899 年の「幾何学基礎論」([43]) で Euclid 幾何学の公理化を行っています. この Euclid 幾何学の公理化に関連した有名な逸話が「点, 線, 面ではなく, 机, 椅子, ビールジョッキで言い換えても幾何学ができる」と Hilbert か語ったというものです[101]. 要するに形式化した体系として幾何学を再構成しても, その本質は残っているということで, 点, 線や面の意味を知らない計算機でも形式的な手続を処理して行けば, 幾何学の処理ができると言い換えても構わないでしょう[102].

David Hilbert(1862-1943)

数学の基礎付けは 1904 年から 1917 年迄の一時的な中断を経て, この中断期に現われた Russell と Whitehead の「数学原論 (Principia Mathematica)」の体系を取り入れ, 命題論理式や述語論理式の形式化を行っています. ただし, Hilbert は論理主義の Frege や Russell とは違い, 論理学から数学を導出することを目的としていないため, Frege や Russell が拘っていることは比較的簡単に済ませ, 基本的に数学の形式化と無矛盾性に焦点を当てていることが Hilbert の著作 ([82], [83] や [44]) から伺えます[103]. ここでは Hilbert の命題の形式化を上記の著作を参照しながら, 現在の流儀を絡めて簡単に眺めてみましょう.

4.21.2 項

「個体記号」は個体を表現する記号のことです. この個体記号としてはギリシャ小文字 “$\alpha, \beta, \gamma, \delta$” 等を用います ([83],p.467).

「変項」は, その値域[104]として個々の個体や後述の論理式を取る対象です. なお, 個体を値域とする変項を「個体変項」, 論理式を値域とする変項を「論理式変項」と呼び, 個体変項をラテン小文字, 論理式変項はラテン大文字で表記します[105]. そして変項は「自由変項」と「束縛変項」と呼ばれる変項の二種類に分類されます. まず,「自由変項」は論理式に中に現われる変項で, 後述の量化詞によって束縛されていない変項です. この自由変項に対して「束縛変項」は量化詞によって束縛されている変項です.

[101] これは有名な発言で, いろいろな所で引用されていますが, その出所と最近の研究については [30], p.166 を参照.

[102] 代数的位相幾何は正にその通りです. この本の §14 に Maxima を使った結び目の不変量の計算を載せているのでその雰囲気を眺めることができるでしょう.

[103]「数学の基礎」[44] は実質的に Bernays が書いていますが, [82] や [83] で Hilbert が示した思想に基づく著作です.

[104][44], p.11 では領域と呼んでいます.

[105][44], p.23 では自由個体変項にラテン小文字 “$a, b, c, k, l, m, n, r, s$” を, 束縛個体変項にはラテン小文字 “$x, y, z, u, v, w$” を用いるとありますが, その直下の註にあるように, この区分は便宜的なものです.

次に「函数記号」は sin のような数学的函数を表現するために用いる記号で, 具体的には, 対象領域の個体, あるいは n 個の対象領域の個体を対象領域の元に対応させるもののことです. また, 変項と函数値が自然数であれば「数論的函数」, 函数値が '真' や '偽' の真理値を取る場合は「論理函数」と呼びます.
さて, これらの自由個体変項, 個体記号, 函数記号といった概念を用いることで「項」を帰納的に定義できます ([44],p.29):

項の定義

(1)　自由個体変項は項である
(2)　個体記号は項である
(3)　$t_1, \ldots, t_n$ が項であり, 記号 f が n 個の変項を持つ函数記号であれば $f(t_1, \ldots, t_n)$ も項である
(4)　上の (1), (2), (3) から得られたものだけが項である

4.21.3　論理記号

「論理記号」は「命題計算の演算」を表現する記号です. ここでは Hilbert の体系で採用されている論理記号の一覧を次に示しておきましょう:

Hilbert の導入した記号

$\overline{}$	$\rightarrow$	&	$\vee$	$\sim$	(x)	(Ex)
否定	ならば	かつ	または	同値	全ての	存在する
$\overline{P}$	$A \rightarrow B$	$A\&B$	$A \vee B$	$A \leftrightarrow B$	$(a)A(a)$	$(Ea)A(a)$

これらの記号の意味を順番に解説しておきましょう.

否定:　記号 "$\overline{}$" は否定と呼ばれ, 命題論理式 'P' に対して '$\overline{P}$' のように論理式全体の上に配置することで命題論理式を否定した命題論理式が得られます. また後述の量化詞を持つ論理式の場合は量化詞の上に配置します. 現在, Frege 由来の記号 "¬" が多く使われ, 他に Principia Mathematica 流儀の記号 "~" も用いられています.

選言と連言:　論理記号 "∨" は選言と呼ばれ, 命題論理式 '$A \vee B$' の意味は 「A または B」 に対応します[106]. なお, 論理式 '$A \vee B$' は '$\overline{A} \rightarrow B$' で置換えられます.
論理記号 "&" は連言と呼ばれ, 命題論理式 '$A\&B$' の意味は 「A かつ B」に対応します. なお, 論理式 '$A\&B$' は '$\overline{A \rightarrow \overline{B}}$' で置換えられ, このように選言と連言には '$A \vee B = \overline{\overline{A}\&\overline{B}}$' という関係があります. そのため, 選言 "∨" か連言 "&" のどちらか一方が定義され, 否定 "$\overline{}$" が定義されていれば, もう一方が得られます. 現在は連言記号として記号 "∧" が主に用いられています.

[106] ラテン語の「または」に対応する "vel" の頭文字 "V" に由来する記号です.

含意: 論理記号 “→” は含意と呼ばれ, 命題論理式「$A \to B$」の意味は「A ならば B」に対応します. この「$A \to B$」は通常の「A ならば B」の感覚とは異なったもので, 命題論理式「$A \to B$」は A が真で B が偽の場合のみに偽となる論理式です. したがって, 命題 A が偽であれば $A \to B$ は常に真となります. 含意記号は他に記号 “⊃” があり, 数理論理学の教科書の多くは記号 “⊃” を用いています. ただし, これらの記号は集合の包含記号 “⊂” や “⊃” と紛らわしこともあって, 数学の本では寧ろ, 記号 “→” がよく用いられます. そして, この本は数学の本なので, 含意として記号 “→” を主に用います. なお, Frege の節でも述べたように, ここでの含意にも「存在含意」は含まれていません[107].

同値: 論理記号 “~” は ‘$A \sim B$’ のように用いられ, ‘$(A \to B)\&(B \to A)$’ と同値, つまり, 左右の命題が「同値」であることを意味します. 同値を表現する記号として他には記号 “↔”, “≡” や “=” が用いられています.

量化詞: (a) と (Ea) は量化詞と呼ばれる記号です. 量化詞には二種類あり, 命題「任意の “x” に対して $P(x)$ は真である」の「任意の “x”」 に対応する “(x)” を「全称記号」, 命題「ある “x” に対して $P(x)$ は真である」の「ある “x” に対して」に対応する “(Ex)” を「存在記号」と呼びます. なお, Hilbert の表記で量化詞を含む論理式の否定は各量化詞や束縛変項の上に否定記号を置きます. たとえば, 命題 ‘$(x)P(x)$’ と ‘$(Ex)Q(x)$’ の否定はそれぞれ ‘$\overline{(x)}P(x)$’ や’$\overline{(Ex)}Q(x)$’ で表記します. そして, 全称記号と存在記号については ‘$\overline{(x)}P(x) = (Ex)\overline{P(x)}$’, あるいは ‘$(x)\overline{P(x)} = \overline{(Ex)}P(x)$’ の関係があります. ここで, この量化詞の束縛変項が作用する範囲を「作用範囲 (scope)」と呼びます. たとえば, 述語 ‘$(x)((Ex)(P(x,y) \vee Q(y)))$’ が与えられたときの最初の量化詞 “$(x)$” の作用範囲は ‘$(Ey)(P(x,y) \vee Q(y))$’, 同じ述語の量化詞 “$(Ey)$” の作用範囲は ‘$P(x,y)$’ となります.
現在, 全称記号は記号 “∀”, 存在記号として記号 “∃” が多く用いられています[108].

論理記号の強さ

論理式の演算を行う論理記号の間には通常の四則演算のように被演算子となる記号表現を引き付ける強さがあります. この強さを記号 “>” で比較したものを次に示しておきます ([44],p.5):

論理記号の強さ

“(x)”, “(Ex)” > ”∨“ > ”&“ > ”→“

このように最も強いのが量化詞で, 引き付ける強さは同じです. このおかげで, Hilbert の体系では PM の “::”. “:.”, “:” や “.” といった区切記号必要としません. また, 論理記号の強さから意味が紛らわしくない場合に括弧が省略できます. たとえば, 「$(A \vee B) \to C$」は「$A \vee B \to C$」と括弧を外せます. その意味では括弧が最も強いとも言えますが, “(x)”,“(Ex)” といった量化詞で括弧を用いるために話を簡単にできないのが不満な点でしょうか. なお, “∨” の方が “∧” よりも強くなっていますが, この順序はのちの計算機言語 Algol に引き継がれているそうです ([84] p.7 の註 8).

[107] このことについては, [44], p10 の註 12 を参照.
[108] 意味は別として, “∀” は ASCII-art でお馴染でしょう (･∀･)

なお, Maxima では §5.3.2 に示すように「演算子の束縛力」という形で演算子の持つ強さが括弧も含めて整数値で表現されています.

リテラル

論理式変項とその否定のことを「リテラル」と呼びます. そして, 有限個のリテラル $L_1, \ldots, L_n$ を論理記号 $\vee$ で結合することで得られた項 $L_1 \vee \ldots \vee L_n$ のことを「節 (clause)」と呼びます.

4.21.4　論理式

基本論理式

「基本論理式」[109]は変項を持たない論理式変項, あるいは, 変項として自由変項のみを持つ論理函数や, この論理函数に対して項を代入して得られる論理式変項のことです.
たとえば,「ミケは猫である」,「$2 = 1 + 1$」 や「$1 < -1$」といった変項を含まない命題は基本論理式になります. これに対し, 命題「$x^2 + 1 > 0$」は自由変項 x を持つ論理函数なので基本論理式になります. しかし, 命題「$(x)(x^2 + 2x + 5 + y)$] は束縛変項 x を持つので基本論理式にはなりません.
この「基本論理式」は二つの論理式に分類できます. まず, 式中に自由変項を持たない基本論理式のことを「論理計算の論理式」と呼び, 自由変項を持つ論理式のことを「述語計算の論理式」, あるいは単に「述語」と呼びます. そして,「論理計算の論理式」を表現する記号を「命題記号」,「述語」を表現する記号を「述語記号」と呼びます.

帰納的な論理式の定義

ここで述語計算の「論理式」は基本論理式を用いて帰納的に定義されます (「数学の基礎」[44],p.24 参照):

述語計算の論理式の定義

(1)　基本論理式は論理式である.
(2)　A, B が論理式であれば $\overline{A}, A\&B, A \vee B, A \to B$ と $A \sim B$ も論理式である.
(3)　$A(c)$ が自由変項 c を含み, 束縛変項 $\mathfrak{r}$ を持たない論理式であれば $(\mathfrak{r})A(\mathfrak{r})$ と $(E\mathfrak{r})A(\mathfrak{r})$ は論理式である.
(4)　上の (1), (2), (3) で構成されたもののみが論理式である.

この「数学の基礎」での述語論理式の定義に対し, 現在の命題論理式の定義を Hilbert 風の記号で示しておきましょう:

[109] [44] では「初等論理式」と訳しています.

命題計算の「論理式」の定義

(1) 命題記号は命題論理式である.
(2) A, B が命題論理式であれば $\overline{A}, A\&B, A \vee B, A \to B$ と $A \sim B$ は命題論理式である.
(3) 上の (1), (2) で構成されたもののみが命題論理式である.

このように命題論理式は述語の定義から量化詞に関連する式を除いた帰納的な定義になります.

代入と書換について

論理式 A の自由変項 a に対する「代入」とは, 論理式 A 内部に出現する全ての a を同一の自由変項 b で置換えることです. このとき, 論理式 A において変項 a を変項 b で代入したと言います.
これに対し,「書換」は論理式内の束縛変項に対して行う操作です. すなわち, 論理式 P にに含まれる束縛変項 $\mathfrak{a}$ に対して, 自由変項 $\mathfrak{b}$ が論理式 P に含まれないときに全ての束縛変項 $\mathfrak{a}$ を束縛変項 $\mathfrak{b}$ で置換える操作です. たとえば, 論理式 $(\mathfrak{a})P(\mathfrak{a})$ に対して個体変項 $\mathfrak{b}$ が論理式 $P(\mathfrak{a})$ に含まれないとき, 束縛変項を $\mathfrak{a}$ から $\mathfrak{b}$ で置換えて新しい論理式 $(\mathfrak{b})P(\mathfrak{b})$ を得る操作です.
現在, 束縛変項に対する書換操作を「α-交換」と呼び, α-交換によって得られる論理式の間には「α-同値」と呼ばれる同値関係成立します.

4.21.5 恒真な論理式

命題計算の論理式に対しては, この命題計算の演算を「真」T,「偽」F の何れかを返す「真理函数」と呼ばれる函数として定義します. このとき, 論理記号については次の関係を充します:

論理記号と真理値

$\overline{T} = F \quad \overline{F} = T \quad T\&T = T \quad T\&F = F \quad F\&T = F \quad F\&F = F$

ここで,「恒等的に真」, すなわち「恒真な」論理式は, tautalogy と呼ばれ, その論理式に現れる論理式変項にどのような真理値を割当てても真になる論理式です. また,「恒等的に偽」である論理式は「恒偽式」, あるいは「充足不能」な論理式と呼びます.

4.21.6 導出

さて, Hilbert は与えられた論理式から推論 (証明) を行うため, 次の図式による規則を 1 つだけを導入します:

推論図式 1

推論図式 1　二つの論理命題式 A と $A \to B$ から B が推論できる

この推論図式 1 は次の図式になります[110]:

[110] [82],p.381 や [83],p.465 を参照

$$\frac{\begin{array}{c}A\\A \to B\end{array}}{B}$$

この図式は Frege でも見られた図式ですね. すなわち, Modus ponensis です. ここで推論図式 1 は「A と $A \to B$ から B が導出できる」, あるいは「A と $A \to B$ から B が証明できる」とも読みます. この図式を現在の線的な表記で記述すると次の図式になります:

$$\frac{A, A \to B}{B}$$

この図式は記号 "$\vdash$" を用いて線的に「$A, A \to B \vdash B$」と表記できます.
この図式に加え, 量化詞を加えた次の図式 (α) と (β) も導入します:

図式 α と図式 β

図式 (α)　$\dfrac{\mathfrak{A} \to \mathfrak{B}(a)}{\mathfrak{A} \to (x)\mathfrak{B}(x)}$

図式 (β)　$\dfrac{\mathfrak{B}(a) \to \mathfrak{A}}{(Ex)\mathfrak{B}(x) \to \mathfrak{A}}$

これらの図式 (α) と (β) は次の基本論理式の逆行型になります:

量化詞に関連する公理

(a)　$(x)A(x) \to A(a)$
(b)　$A(a) \to (Ex)A(x)$

推論図式 1 と図式 (α) と (β) を使って「導出 (=証明)」の定義が出来ます. この「述語計算の導出」は有限の長さの述語計算の論理式の列であり, この列に現われる論理式は次の何れかにあてはまります:

述語計算の導出

1. 命題計算で真の論理式, あるいは論理式 (a), (b) の何れか.
2. 論理式の列で先頭の論理式ではなく, その 1 つ前の論理式から個体変項や論理式変項への代入, 束縛変数の書換, あるいは, 図式 (α), (β) の何れかから得られている.
3. 論理式列の最初の 2 つの論理式ではなく, その前の 2 つの論理式から推論図式 1 から得られている.
4. その前に現われる論理式列の中のどれかの論理式と等しい.

ここで, 1. の論理式や述語計算の導出における先頭の論理式を「初式」と呼びます.
そして, 論理式の導入の初式となる得る恒真な論理式のことを「公理」と呼び, さらに, 論理式変項を 1 つも含まない公理のことを「本来の公理」と呼びます.

それから, 導出における列の最後の論理式を「終式」と呼び, この「終式」を $\mathfrak{B}$ とするときに、導出のことを「論理式 $\mathfrak{B}$ の導出」と呼びます. そして, 論理式がある導出の終式である場合, その論理式のことを「導出可能」であると呼びます.
つまり, Hilbert によると証明は公理を初式として, 終式を証明すべき論理式とする論理式の列として与えられ, その列は図式, 代入や書換による式の変換から得られた論理式で構成されるという訳です.
なお, 現在では与えられた命題が「証明可能」であるとは次のときです:

証明可能について

1. 公理は証明可能である
2. 証明可能な命題論理式に推論図式 1 を適用して得られる命題論理は証明可能である
3. 上の 1. ,2. から得られた命題論理式のみが証明可能である

命題論理式 A が公理からのみ証明可能な場合を「$\vdash A$」と Frege に由来する記号 "$\vdash$" を用いて表記し, さらに, 命題論理式 A が命題論理式「$B_1, \ldots, B_n$」を用いて証明可能な場合は「$B_1, \ldots, B_n \vdash A$」と記述します.
次に A, B, C を述語計算の論理式とし, これらの論理式の自由変数を保全したまま, すなわち, 一切の代入操作を行わずに, A と B から C が導出可能であれば, 論理式 $A \to C$ も論理式 B から導出可能です. このことを演繹定理と呼び, 現在では次のように記述しています:

演繹定理

$A, B \vdash C$ ならば $B \vdash A \to C$

4.21.7 公理系

ここでは Hilbert の「The foundations of mathematics,1927([83])」で挙げている公理の解説を行います. そこで, 最初に含意の公理を示しておきましょう:

Hilbert の公理系 I(含意の公理)

公理 1　$A \to (B \to A)$
公理 2　$(A \to (A \to B)) \to (A \to B)$
公理 3　$(A \to (B \to C)) \to (B \to (A \to C))$
公理 4　$(B \to C) \to ((A \to B) \to (A \to C))$

ここで公理 1 が前提条件の導入, 公理 2 が同じ前提条件の除去, 公理 3 が前提条件同士の交換, 公理 4 が命題の除去となります.
次に選言と連言に関する公理を示します:

Hilbert の公理系 II(選言と連言の公理)

公理 5　$A\&B \to A$
公理 6　$A\&B \to B$
公理 7　$A \to (B \to A\&B)$
公理 8　$A \to A \vee B$
公理 9　$B \to A \vee B$
公理 10　$(A \to C) \to ((B \to C) \to (A \vee B \to C))$

今度は否定に関連する公理を次に示します:

Hilbert の公理系 III(否定の公理)

公理 11　$(A \to B\&\overline{B}) \to \overline{A}$
公理 12　$\overline{\overline{A}} \to A$

ここで公理 11 が矛盾の原理, 公理 12 が二重否定の除去になります.
Hilbert は論理 ε 函数を導入します. この函数を用いることでクラスを定めることが可能になりますが, それと同時に量化詞の定義も行えます. この論理 ε 函数については次の公理があります:

Hilbert の公理系 IV(ε-公理)

公理 13　$A(a) \to A(\varepsilon(A))$

論理 ε 函数は Frege や Russell で見られる函数 A の外延, すなわち, 函数 A を充すクラスを定める函数です. たとえば, 「$\varepsilon(\mathfrak{A})$」は述語 $\mathfrak{A}(x)$ が真となる対象が所属するクラスを定めます. さらに, $\mathfrak{A}(x)$ を充す対象が一つだけ存在する場合, その対象を指定します. そのために a を '$\mathfrak{A}(x)$' を充す対象とするとき, '$\varepsilon(\mathfrak{A}(x)) = a$' となります.
このように函数 ε は空でない集合から成分を取出すことを保証する「選択公理」の選択函数としての働きを持ち, この公理 13 は「選択公理」になります[111]
この論理 ε 函数と公理 13 によって量化詞の定義が次の式で行えます:

量化詞の定義

$(x)A(x) \sim A(\varepsilon(\bar{A}))$
$(Ex)A(x) \sim A(\varepsilon(A))$

ここで, $A \sim B$ は $(A \to B)\&(B \to A)$ のことです.
この ε 函数を用いた量化詞の定義によって, 次の命題が成立することが分ります:

- Aristotle の dictum:　$(x)A(x) \to A(a)$
- 排中律:　$\overline{(x)A(x)} \to (Ex)\overline{A(x)}$

次に対象の同値性については記号 "=" を用います. この対象の同値性に関しては次の公理 14 と公理 15 があります:

[111] [82],P.382 で, この公理は選択公理 (Axiom of choice) として導入されています.

Hilbert の公理系 V(同値性の公理)

公理 14　$a = a$
公理 15　$(a = b) \to (A(a) \to A(b))$

ここで, 有限個の「本来の公理」と公理 15 で構成された公理系を「1 階の公理系」と呼びます.
この同値性の公理に関連して, 「…という性質を持つもの」という概念を形式化する ι-記号と ι-規則があります. まず, ι-記号により自由変項 a を含み, 束縛変数 $\mathfrak{x}$ を含まない論理式 $A(a)$ に対して $\iota_{\mathfrak{x}}A(x)$ を得ます. さらに, 述語 $A(x)$ を充す項 x に対して一意性が満される場合, すなわち:

$$(E\mathfrak{x})A(\mathfrak{x}), \qquad (\mathfrak{x})(\mathfrak{y})\,(A(\mathfrak{x})\&A(\mathfrak{y}) \to \mathfrak{x} = \mathfrak{y})$$

が導出されるときに $\iota_{\mathfrak{x}}\mathfrak{A}(\mathfrak{x})$ を「項」として利用することが可能となり, $\mathfrak{A}^*(\mathfrak{x})$ をと $\mathfrak{A}(\mathfrak{x})$ と α-同値な論理式とするときに, $\mathfrak{A}(\iota_{\mathfrak{x}}\mathfrak{A}^*(\mathfrak{x}))$ を初式として利用することができるというものです.
また, 拡張された ι-規則では, 自由変項 a を含み, 束縛変数 $\mathfrak{x}$ を含まない論理式 $\mathfrak{A}(a)$ による表現 $\iota_{\mathfrak{x}}\mathfrak{A}(\mathfrak{x})$ を認めて次の公理を採用するというものです:

Hilbert の公理系 VI(ι-公理)

ι-公理:　$(Ex)(y)\,(\mathfrak{A}(y) \sim y = x) \to \mathfrak{A}\,(\iota_{\mathfrak{x}}\mathfrak{A}(\mathfrak{x}))$

それから数の公理として公理 16 と公理 17 を導入します:

Hilbert の公理系 VII)

公理 16　$a' \neq 0$
公理 17　$a' = b' \to a = b$
公理 18　$(A(0)\&((a)(A(a) \to A(a')))) \to A(b)$

ここで記号 "′" は数 a に対して a' と用い, 数 a の後者を意味します. したがって,$0, 0', 0'', 0''', \ldots$ は自然数の列を構成します. ここで公理 16 は 0 は任意の数の後者とならないことを意味し, 公理 18 が数学的帰納法の原理になります.
この公理系 (VI) に数記号 "0" と後者記号 "′", そして「等号の公理」の公理 15 を加えた公理系が Peano の公理系となります. この Peano の公理系に和と積の帰納等式を加えた数論的公理系のことを「公理系 (Z)」と呼び, さらに, この公理系 (Z) に述語計算を加えてできる形式的体系を「公理系 (Z) の形式的体系」, あるいは簡単に「体系 (Z)」と呼びます.

最後に, 現在, Hilbert 流の演繹体系と呼ばれる公理系を示しておきましょう:

Hilbert 流の演繹体系 (Hilbert & Ackermann の公理系)

1. $A \supset (B \supset A)$
2. $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$
3. $A \supset (B \supset A \wedge B)$
4. $A \wedge B \supset A$
5. $A \wedge B \supset B$
6. $A \supset A \wedge A$
7. $A \supset A \wedge B$
8. $A \supset A \vee B$
9. $B \supset A \vee B$
10. $(A \supset C) \supset ((B \supset C) \supset ((A \supset B) \supset C))$
11. $(\neg A \supset \neg B) \supset (B \supset A)$

4.21.8 Russell との違い

Russell の PM の体系と Hilbert の体系の違いですが, Hilbert の講演 ([83],p.473) によると, Russell の PM は「無限公理」と「還元可能性公理」に依存しています. ただし, Hilbert の体系ではこれらの公理を必要とはしません.

なお, これらの公理が必要になったのは, 複雑怪奇な「分岐的階型理論」を採用したためですが, Hilbert の体系ではこの分岐的階型理論を採用しておらず, 述語の値域として型が現われる単純な型の理論のみです. まず, Hilbert は変項型 (variable-type) を導入します ([82],p.386-387). この型は論理式の変項の型を階 (Height) で表現し, 変項を分類します. たとえば, 数のような数学的対象 (個体) は 0 階で, その個体を変項の領域とする函数は 1 階の論理式になります. 次に 1 階の論理式を変項の領域とする函数は 2 階の論理式と階が上って行きます.

4.22 集合論の公理化

4.22.1 ZFC-公理系

Cantor の素朴集合論も公理系として再構築されました. この公理化は最初に Zermelo によって試みられます ([98]). その公理系を Fränkel が整理・拡張したものが現在の ZF-公理系 (ZF=Zermelo-Fränkel) と呼ばれる集合論の公理系になります ([79]).
この他の集合論の公理系に von Neuman が構築し, Bernays が簡素化し, Gödel がそれを用いて選択公理と連続体仮説の無矛盾性の証明を行ったことで知られる BG-公理系もあります.
まず, ZF-公理系は上の「外延公理」,「対公理」,「和集合公理」,「冪集合公理」,「空集合」,「無限集合公理」,「正則性公理」と「置換公理」で構成されています. この ZF-公理系に「選択公理」(Axiom of choice) を追加した公理系を ZFC-公理系と呼びます. そして, ZF-公理系や ZFC-公理系では「Russell の逆理」や「Richard の逆理」は排除されます.
ここでは ZFC-公理系について概要を述べます. なお, ここで用いられている記号の意味は §4.21, それから, ZFC-公理系の詳細は「選択公理と数学」([34]) を参照して下さい:

ZFC-公理系 (ZF-公理系+選択公理)

外延公理	$\forall z(z \in x \leftrightarrow z \in y) \to x = y)$
対公理	$\exists z \forall u(u \in z \leftrightarrow u = x \vee z \in y)$
和集合公理	$\exists y \forall z[z \in y \leftrightarrow \exists u(u \in x \wedge z \in u)]$
冪集合公理	$\exists y \forall z(z \in y \leftrightarrow z \subseteq x)$
空集合公理	$\exists x \forall y \neg(y \in x)$
無限集合公理	$\exists x[\emptyset \in x \wedge \forall y(y \in x \to y \cup \{y\} \in x)]$
正則性公理	$x \neq \emptyset \to \exists y(y \in x \wedge y \cap x = \emptyset)$
置換公理	変項 v を含まない任意の述語論理式 $\varphi(x, y)$ に対し, $\dfrac{\forall x \forall y \forall z[\varphi(x, y) \wedge \varphi(x, z) \to y = z]}{\exists v \forall y[y \in v \leftrightarrow \exists x(x \in u \wedge \varphi(x, y))]}$
選択公理	$\forall x \in u(x \neq \emptyset) \vee \forall x, y \in u(x \neq y \to x \cap y = \emptyset)$ $\to \exists v \forall x \in u \exists! t(t \in x \vee t \in v)$

外延公理: 外延公理 (Axiom of extentionality) は二つの集合 x, y の全ての元が一致することと $x = y$ であることが同値であること, すなわち, 集合はその構成要素で決定されることを意味します.

対公理: 対公理 (Axiom of pairing) は集合 x, y が与えられた場合, これらを成分に持つ集合 $\{x, y\}$ が存在することを保証します. この集合の一意性は外延公理によって保証されますが, 集合 $\{x, y\}$ に順序はそのままでは入りません. そこで, $\{x, \{x, y\}\}$ という集合を考え, これを $\langle x, y\rangle$ と表記します. すると集合 x と $\{x, y\}$ の間には包含関係が入るために $\langle x, y\rangle$ の元の x と y に順序を入れることが可能になります. この集合 $\langle x, y\rangle$ を対集合と呼びます. この対集合に関しては $\langle a, b\rangle = \langle c, d\rangle \leftrightarrow a = c, b = d$ が成立します. また, a を n 階の対象,b を $n + 1$ 階の対象とする場合, $\langle a, b\rangle$ を $\langle \{a\}, b\rangle$ で定義が可能です. これによって $\langle a, b, c \ldots\rangle$ も同様に定義できます.

和集合公理:　和集合公理 (Axiom of the union) は集合 x,y の和集合 $x \cup y$ の存在を保証します.

冪集合公理:　冪集合公理 (Axiom of the power set) は集合 x の成分から構成される全ての集合である冪集合 $\mathfrak{P}(x)$ の存在を保証します. なお, この冪集合には自分自身は含まれないことが正則性公理で保証されるため,「Cantor の逆理」は生じません.

空集合公理:　空集合公理 (Axiom of elementary set) は一切元を含まない空集合 $\emptyset$ が存在することを保証します.

無限集合公理:　無限集合公理 (Axiom of infinity) は $\emptyset$ を含む集合 x の元 y に対して $y \cup \{y\}$ を含む集合の存在を保証するものです. ここでの $y \cup \{y\}$ を y の後継と呼びますが, この後継を限りなく生成できることを保証している公理であるために無限公理と呼ばれます. この公理によって順序数の定義で用いた $\{\emptyset, \{\emptyset, \{\emptyset\}\}, \ldots\}$ といった集合の存在も保証されます.

正則性公理:　正則性公理 (Axiom of regularity) は空集合 $\emptyset$ と異なる集合に対して, それ自身と共通要素を持たない要素を含むことを保証します.
この公理は $a = \{a\}$ のような集合, さらには $b_0 = \{b_1\}, b_1 = \{b_2\}, \ldots$ のような集合の無限列 $b_0, b_1, b_2, b_3, \ldots$ を排除します. もしも, これらの無限列を認めると, $\ldots \in b_3 \in b_2 \in b_1 \in b_0$ のような底なしの無限降下列が存在することになって非常に厄介です. この正則性公理によって自己参照を行う集合の定義も排除されることになります.

置換公理:　置換公理 (Axiom of replacement) は図式であり, $\forall x \forall y \forall z[\varphi(x,y) \wedge \varphi(x,z) \rightarrow y = z]$ を仮定とし, この仮定の下で, $\exists v \forall y \left[y \in v \leftrightarrow \exists x \left(x \in u \wedge \varphi(x,y)\right)\right]$ が成立するという図式です.
まず, 置換公理の仮定から $\varphi(x,y)$ は $F(x) = y$ となる変項 x の一変項函数 $F(x)$ とみなせます. そして, この仮定から得られる結論を 1 変項函数 $F(x)$ を用いて言い換えると, 集合 u に対して集合 v が存在し, 集合 u の成分 x の函数 F による像 $F(x)$ が集合 v の成分となり, 逆に集合 v の成分 y に対応する函数 F の逆像 x が集合 u に含まれることを意味します. すなわち, 函数の値域が集合であれば定義域も集合であり, 逆に定義域が集合であれば, その値域も集合になることを保証する公理です.
なお, この置換公理は Fränkel によって Zermelo が最初に導入した分出公理 (Axiom of seperation) : $\forall a \exists b \forall x[x \in b \leftrightarrow x \in a \wedge P(x)]$ の代りに入れた公理で, 置換公理から分出公理を導出することも可能です. 実際, 置換公理の $\phi(x,y)$ の代りに $\phi(x) \wedge x = y$ を用いれば, $\exists v \forall y[y \in v \leftrightarrow y \in u \wedge \phi(y)]$ が得られます. この分出公理は集合 u と述語 ϕ が与えられたときに, 述語 $\phi(x)$ を充す集合 v を集合 u から切り出せることを意味し, このことから,「分出 (seperation)」という名前になっているのです. この分出公理でえられる集合 v を $\{x \in u | \phi(x)\}$ で表現しましょう. この表記からも判るように述語を充す集合 v の成分 x には $x \in u$ という制約が予め入っており, 述語 ϕ のみから集合 v を構築することを禁じています. Frege の「概念 $\Phi(x)$ の値域」"$\grave{\varepsilon}\Phi(\varepsilon)$" は実質的に $\{x|\Phi(x)\}$ で定義されており, このように「概念の値域」には一切の制約がありません. その御陰で Frege の体系では無限公理無しに値域を際限無く生成することが可能であった一方で,「Russell の逆理」に対して何等の対処もできずに呆気無く体系が崩壊してしまった原因となっています. 対象:$\{x|\phi(x)\}$ は現在, クラス/類と

呼ばれる対象で, ZFC-公理系では集合とは別物で, その存在は許容されません. しかし,BG-公理系ではクラスは集合とは別物として, その存在自体は許容されています.

4.22.2 選択公理について

選択公理は空集合 $\emptyset$ でない集合から元を一つ取出すことができるというもの, すなわち, 選択函数が存在することを保証する公理です.
この公理はユークリッド幾何学での「平行線の公理」に幾分似た性格を持つ公理です. この公理は非常に自然に見える性質がありますが, 実はこの選択公理には厄介な問題も持っています.
この選択公理と同値な公理や定理として次のものがあります:

- 整列可能性定理
- Zorn の補題
- Tikhonov の定理

整列性定理は, 任意の集合に対してその元の間に適切な順序を定義すると整列集合, すなわち, 任意の空でない部分集合が最小限を持つ集合になるというものです.
そして,Zorn の補題は順序を入れた集合に対して非常によく用いられる補題です:

Zorn の補題題

任意の空でない帰納的順序集合はその極大元を持つ.

すなわち, 順序付けられた集合に対し, その部分集合の元が順序による上限を持てば必ず最大元が存在するという補題です.
Zorn の補題では有限回で終了するかどうか分らない処理で, その処理には上限があり, さらに何らかの指標で単調な増大列が構成できれば, その処理を継続することで, やがて具体的な値が得られることを意味します. また, その指標の程度で近似解が得られることも保証されるのです.
この補題を用いることで,「全てのベクトル空間は基底を持つ」といった命題を容易に証明することも可能です.
この選択公理は非常にもっともらしい公理で, この公理や同等な命題を用いてさまざまな数学の命題が証明できる一方で,「Banach-Tarski の逆理」と呼ばれる逆理も導出できてしまいます:

Banach-Tarski の逆理

3 次元 Euclid 空間 $\mathbb{R}^3$ において,A, B を内転を持つ任意の有界集合とする. このとき, A, B を適当な同数個

$$\begin{cases} A = A_1 \cup A_2 \cup \cdots \cup A_n \\ B = B_1 \cup B_2 \cup \cdots \cup B_n \end{cases}$$

に分割し, 各 A_i と $B_i (1 \geq i \geq n)$ が合同にできる.

この逆理を適用すると, ゴルフボールの表面を適当に分割し, それらを貼り合せたもので地球が覆えることになります. したがって, 牛の皮であれば砦どころか世界征服も可能と女王 Dido も大喜びな話[112]になりますね.

4.23　論理式の代表的な操作

4.23.1　Skolem の標準形について

任意の論理式は量化詞と ¬, ∨, ∧ のみの式に変換できます. 実際, $P \to Q$ は $\neg P \vee Q$ に置換えられ, $P \leftrightarrow Q$ はそもそも $(P \to Q) \wedge (Q \to P)$ と同値なので, $(\neg P \vee Q) \wedge (\neg Q \vee P)$ に置換えられるからです.
その結果, 任意の論理式は選言標準形 $F_1 \vee \cdots \vee F_m$, あるいは 連言標準形 $G_1 \wedge \cdots G_n$ のいづれかの式に変換できます.
述語 P の自由変項を $x_1, \cdots, x_n$ とするとき, これらの自由変項を全称記号 ∀ や存在記号 ∃ を用いて束縛することが可能です:

閉形式

全称閉形式　$\forall x_1, \forall x_2, \ldots, \forall x_n P$
存在閉形式　$\exists x_1, \exists x_2, \ldots, \exists x_n P$

任意の閉形式に対しては, 量化詞を先頭に置いた次に示す「Skolem の標準形」と呼ばれる式に変換することが可能です:

- $Q_1 x_1 Q_2 x_2 \cdots Q_m x_n \left[(A_{11} \vee \cdots \vee A_{1k_1}) \wedge \cdots (A_{h1} \vee \cdots \vee A_{hk_h}) \right]$
- $Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \left[(B_{11} \wedge \cdots \wedge B_{1l_1}) \vee \cdots (B_{m1} \wedge \cdots \wedge B_{ml_m}) \right]$

ここで, A_{ij} は $A_{ij_1} \wedge A_{ij_2} \wedge \cdots \wedge A_{ij_m}$ のように論理演算子 ∧ のみか論理演算子を持たない述語, B_{ij} は $B_{ij_1} \vee B_{ij_2} \vee \cdots \vee B_{ij_n}$ のように論理演算子 ∨ のみ, あるいは論理演算子を持たない述語です.
この変換手順を以下に示しておきましょう:

標準形変換の手順

1. 量化詞 Qx の作用範囲内に変項 x が出現しない場合, 量化詞を削除する
2. 束縛変項の取り換え式の左から順番に行い, 全ての束縛変項に重複がないようにする
3. 同値な論理式に置換することで論理演算子を ∨,∧,¬ のみにする
4. 量化詞の外にある ¬ は量化詞の作用範囲内に入れる
5. 量化詞を式の左側に移動させる
6. ∨ に対して ∧ を分配, あるいは ∨ に対して ∧ を分配する

[112] 牛の皮で覆えるだけの土地が与えられるという条件で, 牛の皮を細かく切って取り囲んで得た場所から発展したというカルタゴの建国神話

4.23.2 Davis-Putnam の手続

連言標準形に変換することで論理式は $P_1 \wedge P_2 \wedge \cdots \wedge P_n$ の書式, ここで各 P_i はリテラル L_i^j から構成された節 (clause): $P_i = L_i^1 \vee \cdots \vee L_i^m$ として表記されます.
連言標準形の論理式に対しては, その論理式が恒偽 (充足不能) な命題であるかどうかを検証する「**Davis-Putnam の操作**」と呼ばれる操作があります. なお, Davis-Putnam の操作は 1960 年の Davis と Putnam の論文で示された自動証明の手法を Logemann と Loveland が IBM 704 計算機に修正を加えて実装したために,「**DPLL**」とも呼ばれています.

この Davis-Putnam の手続を以下に示します:

Davis-Putnam の手続

- その 1: l をリテラルとするとき, 論理式 P の節で $P_i = l$, $P_j = \neg l$ を充す節 P_i, P_j があれば, 論理式 P は恒偽な命題です.
- その 2: l をリテラルとするときに P の節で $P_i = l \vee \neg l$ となる節 P_i が存在すれば, この節を論理式 P から削除した論理式で置換します.
- その 3: P の節 P_i がリテラル l であれば, 論理式 P から節 P_i を削除します. このときに論理式 P にリテラル $\neg l$ が含まれていれば, $\neg l$ を論理式 P から削除した論理式で置換します.
- その 4: リテラル l が論理式 P に含まれ, その否定 $\neg l$ が論理式 P に含まれていない場合, リテラル l を論理式 P から削除した論理式で置換します.
- その 5: 論理式 P がリテラル l を含み,

  $$(l \vee A_1) \wedge \cdots (l \vee A_m) \wedge (\neg l \vee B_1) \wedge \cdots \wedge (\neg l \vee B_n) \wedge R$$

  の形式であれば,

  $$A_1 \wedge \cdots A_m \wedge B_1 \wedge \cdots \wedge B_n \wedge R$$

  で論理式 P を置換します.

ここで, その 5:の操作によって二つの節 '$L \vee P$' と '$\neg L \vee Q$' で構成された論理式 '$L \vee P) \wedge (\neg L \vee Q)$' から節 '$P \vee Q$' が得られます. この節 '$P \vee Q$' を節 '$L \vee P$' と節 '$\neg L \vee Q$' からの導出節と呼びます.
ここでのその 1 からその 5 の操作の反復の結果, 論理式 P が最終的に空集合になった場合に論理式 P は充足可能な論理式となります. もしも, この一連の操作の結果, 論理式 P が最終的に偽という結果が得られた場合, この論理式 P は恒偽の命題となります.
この DPLL によって機械的な操作で恒偽性の判別がおこなえますが, これは非常に重要なことです. なぜなら論理式 $F_1, \ldots, F_n$ と G に対し, $F_1 \wedge \cdots \wedge F_n \rightarrow G$ が恒真であれば, $F_1 \wedge \cdots \wedge F_n \wedge \neg G$ が恒偽であることが同値だからです. さらに, 演繹定理を加えると, $F_1, \ldots, F_n \vdash G \sim F_1 \wedge \cdots \wedge F_n \wedge \neg G$ は恒偽 となります.

4.23.3　Herbrand の定理

Davis-Putnam の手続を可能にしている定理が Herbrand の定理です. (まだまだ)

4.23.4　λ 計算

λ 式は, そもそも, 函数 $f(x)$ という表記方法が函数 f の表記なのか, それとも, x の値なのかが判別し難いために導入された函数の記述方法です. この問題点は Frege も気付いており, 独自の表記法を開発していますが (§4.19 参照), この Frege とは独立して Church が λ 式による函数表記を考案しています. Church は, この λ 式を用いて完全な数学の体系を目指しましたが, 数学の体系の構築には失敗します. ところが, この λ 式は, その扱い方の便利さで残ったものです.
さて, 函数 $f(x)$ が函数 f の表記なのか,x における値かどうかが不明瞭なことは, 通常の計算で特に問題にならないかもしれません. 実際, 数学で式を $f(x) = \sin(x^2 + 1)$ のように書いているときには, この表記に問題があるといわれても何が問題なのか困惑するかもしれませんね. ところが, 函数 f を一種の演算子として考えてしまうとどうでしょうか？ 具体的には, Basic や Fortran で, この式を表現するときに直接 `f(x)=sin(x**2+1)` と記述しても, 変項 `x` がどこかで値を割当てられていれば, その `f(x)` には `x` の値が入ってしまうので作用素としての f の表現には失敗しますね. プログラムの場合, サブルーチンやプロシジャといったもので表記することになるでしょうが, その場合には引数を使って表現するでしょう. λ-表記はその方法をより体系的に行える手法です.
その方法は, まず, 函数の変項を式の先頭に明示的に取出し, そのうしろに函数の本体を表記するものです. たとえば, $f(x, y) = x \cdot y$ の λ-表記は $\lambda\, xy.x \cdot y$ になります.
ここで, λ-表記された函数のことを λ 項と呼び, λ の直後にある変項をメタ変項と呼びます. たとえば, $f(x) = x^2 + a$ を λ-表記すると $\lambda x.(x^2 + a)$ となりますが, ここでの変項 a のように λ 表記のメタ変項にならない変項を「自由変項」と呼びます. 逆に λ のうしろにあるメタ変項に対応する本体側の変項を「束縛変項」と呼びます. そして, この束縛変項を函数本体に含まれていない変項で置換することを「α 変換」と呼び, λ 項 M が λ 変換で λ 項 N に移る場合, M と N を「α 同値」と呼び, と呼んで '$M =_\alpha N$' と表記します.
現在では λ-表記可能な函数の集合は一般帰納的函数と呼ばれるものや, Turing 機械と同じものであることが知られています.

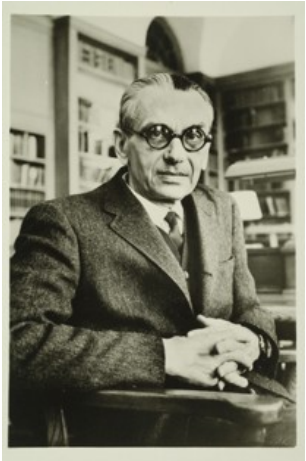
4.24　Gödel の不完全性定理

4.24.1　背景

1928 年に Hilbert は宿敵 Brouwer を雑誌「Mathematische Annalen」の編集会議から追放することで政治的に制圧[113]し, 束の間の勝利を味わっていますが, 肝心の Hilbert 計画は思わぬ所で頓挫してしまいます. これは Gödel の不完全性定理 (1930 年) によるものです. Gödel の不完全性定理には第一不完全性定理と第二不完全性定理の二種類があり, 第一不完全性定理では, 決定不能な命題

[113] この経緯については [30],p.231-241 参照.

の存在性を示し, 第二不完全性定理では, 論理体系の完全性はそれ自身で示すことができないことを示すものです.
まず, 第一不完全性定理はそもそも “Über Formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I” (Principia Mathematica, および, 関連した体系の形式的に決定不能な命題について I) という題名が示すように Peano の自然数の公理系を付加した Russell の Principia Mathematica の公理系にて決定不能な命題が存在するというものです. ここで Hilbert は彼の形式主義を達成するために Principia Mathematica の体系に Peano の公理系を追加した体系を用いているので, この公理系での決定不能な命題の存在は Hilbert 計画の全般的な失敗を意味します.



Kurt Gödel(1906-1978)

4.24.2 体系 P

体系 P の基本記号

まず, Gödel は体系 P の論理式で用いる記号:基本記号を次のように定めます:

1. 定数: 否定 “~”, 連接 “∨”, すべて “Π” [114], 零 “0”, 後者 “f”, 括弧 “(” と “)”
2. (a) 第一型の変数:“x_1”, “y_1”, “z_1”, ...
   (b) 第二型の変数:“x_2”, “y_2”, “z_2”, ...
   (c) 第三型の変数:“x_3”, “y_3”, “z_3”, ...

第 n 型の記号と基本論理式

第 1 型の記号は次の記号の組み合わせです:

$$a, fa, ffa, fffa, \ldots$$

ここで a を 0 か第 1 型の記号とします. この a が 0 の場合, これらの記号を数字と呼びます[115].
b が第 n 型の記号で a が第 $n+1$ 型の記号であるときに $a(b)$ という形を取る記号を「基本論理式」と呼びます.
そして, 論理式の類 (クラス) を次の論理式を含む最小の類として定めます:

1. 全ての基本論理式を含む
2. a, b を論理式の類の元とする場合, 否定: $\sim a$, 選言: $a \vee b$, 全称化 (普遍化): $x\Pi(a)$ を含む

[114] $x\,\Pi(p(x))$ は $\forall x.p(x)$ と同じ意味です.
[115] $f0 \to 1, f1 \to 2, \ldots$ のように自然数の生成ができるのです

次に, 自由変項を持たない論理式を文論理式 [Satzformel] と Gödel は名付けていますが, これは変項が論理式に含まれる場合, その変項全てが束縛変項であることを意味し, このように全ての変項が束縛変項となる論理式を現在は閉論理式と呼んでいます. 次に, n 個の個体変項を持つ論理式を n 項関係記号, そして, 変項が一つの論理式を類記号[116]と呼びます.
そして, 論理式 "a" が論理式 "b" の持ち上げであるとは, 論理式 "b" の変項の全ての型を同じ階数だけ上げることで論理式 "a" が得られる場合です.

体系 P の公理系

体系 P の公理として次の公理を採用します:

1. (a) $\sim (fx_1 \equiv 0)$
   (b) $fx_1 = fy_1 \supset x_1 = y_1$
   (c) $x_2(0).x_1\Pi(x_2(x_1) \supset x_2(fx_1)) \supset x_1\Pi(x_2(x_1))$
2. (a) $p \vee q \supset p$
   (b) $p \supset p \vee q$
   (c) $p \vee q \supset p \vee q$
   (d) $(p \supset q) \supset (r \vee p) \supset (r \vee q)$
3. (a) $v\Pi(a) \supset \text{Subst}\, a \begin{pmatrix} v \\ c \end{pmatrix}$
   (b) $v\Pi(b \vee a) \supset b \vee \text{Subst}\, a \begin{pmatrix} v \\ c \end{pmatrix}$
4. (a) $(Eu)(v\Pi(u(v) \equiv a)$
5. (a) $x_1\Pi(x_2(x_1) \equiv y_2(y_1) \supset x_1 = y_1$

最初 1. の 3 個の公理は自然数の公理系を構成し, 最初の公理は 0 は何如なるものの後者ではないこと, 第 2 の公理は後者が一致する場合は前者も一致することを示します. そして, 第 3 が数学的帰納法になります. この体系 P では Frege のように零や直続関係が何であるかは議論せず, 定数として予め零 "0" や後者 "f" を定めています.
次の 2. の 4 個の公理は任意の論理式に対して成立する論理式です. そして, 3. の二つの公理は代入 (substitute) を伴うものです. ここで函数 $\text{Subst}\, a \begin{pmatrix} v \\ c \end{pmatrix}$ は論理式 a に含まれる項 v を項 c で置換する操作を行う函数です.
たとえば, 論理式 a を「ミケは猫である」とするとき, $\text{Subst}\, a \begin{pmatrix} \text{ミケ} \\ x \end{pmatrix}$ は論理式「x は猫である」になります. Russell の Matrix を用いると, $a/$ミケ (x) と表記できるでしょう.
そして 4. の公理は集合論では「集合の内包公理」と呼ばれる公理で, Gödel はこの公理を還元可能性公理の代りに入れています.
そして, 最後の 5. の公理はクラスはその構成要素によって一意に定まるという公理です.

[116] [42] では単項述語記号と訳しています.

4.24.3 Gödel 数

それから Gödel 数と呼ばれる数を定義します. これは体系 P の基本記号に自然数を一対一対応させて基本記号から構成される命題を自然数の積として表現する手法です. この素数と記号の対応を次の表に纏めておきます:

基本記号と自然数の対応

“0”	⇔	1	“∨”	⇔	7	“(“	⇔	11
“f”	⇔	3	“Π”	⇔	9	“)”	⇔	14
“∼”	⇔	5						

それから n 型の変項に対しては p^n となる形の素数 ($p > 13$) を対応させます. これによって, 任意の基本記号の有限列は自然数の有限列に一対一に対応します. この操作によって得られた素数列 $n_1, n_2, n_3, ldots, n_l$ に対し, $2^{n_1} \cdot 3^{n_2} \cdot 5^{n_3} \cdots p_l^{n_l}$ を対応させます. ここで p_l は 1 番目を 2 とする第 l 番目に小さい素数です. この操作を Gödel 数化と呼び,Gödel 数化によって得られた数を Gödel 数と呼びます. この Gödel 数化と Gödel 数はさまざまな所で用いられています.

4.24.4 述語の算術化

原始帰納的述語

函数 $f(x_1, \ldots, x_n)$ が述語 $P(x_1, \ldots, x_n)$ の表現函数と呼ばれるのは述語 P と函数 f が次の関係を充す場合です:

述語の表現函数

$P(x_1, \ldots, x_n)$ が真	⇔	$f(x_1, \ldots, x_n) = 0$
$P(x_1, \ldots, x_n)$ が偽	⇔	$f(x_1, \ldots, x_n) = 1$

ここで述語 P に対し, その表現函数として原始帰納的函数が存在する場合, この述語 P を原始帰納的述語と呼びます.

17. $a = b$
18. $a \leq b$
19. $a < b$
20. $P(g_1(x_1, \ldots, x_m), \ldots, g_n(x_1, \ldots, x_m))$l

    ここで $P(y_1, \ldots, y_n)$ は原始帰納的述語, $g_1, \ldots, g_n$ は m 変項の原始帰納的函数
21. P, Q を原始帰納的述語とする場合, 次の述語は全て原始帰納的述語になります:

    - $\neg P(x_1, \ldots, x_m)$

- $P(x_1, \ldots, x_m) \vee Q(x_1, \ldots, x_n)$
- $P(x_1, \ldots, x_m) \wedge Q(x_1, \ldots, x_n)$
- $P(x_1, \ldots, x_m) \to Q(x_1, \ldots, x_n)$

22. $R(x_1, \ldots, x_n, y)$ が原始帰納的述語の場合, 次の述語は原始帰納的述語になります:

- $\exists y[y < z \wedge R(x_1, \ldots, x_n, y)]$
- $\forall y[y < z \wedge R(x_1, \ldots, x_n, y)]$

4.24.5　決定不能な命題

決定不能な命題で有名なものに連続体仮説があります. これは自然数の基数 $\aleph_0$ よりも大きく, 実数の基数 $\aleph$ よりも小さな基数が存在しないというものです.
歴史的には Cantor が最初に提出した問題です. この問題は 1900 年のパリで開かれた第 1 回数学者国際会議で取り上げられた Hilbert の 23 問題で第一番目の問題とされた程ですが, 結局, ZFC の体系で証明不能な命題であることは 1963 年に Cohen よって示されています.

4.24.6　数学基礎論の勝者は？

さて, この Gödel の不完全性定理によって Hilbert の目論見も見事に破綻してしまいましたが, では, 数学の基礎を巡る論争に関して論理主義, 形式主義, 直観主義のどちらか勝者となったと言えるでしょうか？
実際は論理主義も形式主義も部分的な目的の達成に成功しています. たとえば, Hilbert の形式主義は「数学の可解性」という側面では不完全性定理によって, 可解でない命題の存在が証明されたことから失敗したといえるでしょう. その一方で数学の形式化と無矛盾性に関して或る程度の成功を収めている面も無視できません.
直観主義についても同様で, 結局の所, 論理主義, 形式主義, 直観主義のどちらも当初の目的を完全に達成することには失敗しており, 最終的にどちらも勝者にも敗者にもなっていないのが実情なのです.

4.25　そして計算機

4.25.1　数学の現代化

ところで, 不完全性定理の言う所の決定不能な命題は, どちらかと言えば, 数学の辺境で生じていることで, 普通に数学を研究する上で Hilbert の形式主義の考えは経験的に問題が全くないために, この形式主義の考えが数学の主流となってさまざまな方面に影響を与えました.
まず, 1930 年代のフランスの若手数学者グループ Bourbaki [117]による数学原論は構成主義と呼ばれ,Hilbert の初期の形式主義を修正したものです.

[117]創立メンバーは Weil, Cartan, Chevaley 等

この構成主義では, 集合論を基盤とした数学的構造を全面に出していますが, 1957 年のスプトーニクの打ち上げを契機に西側諸国の教育界で吹き荒れた新数学 (New Math, 日本では「数学の現代化」) で, この流儀が大きく採用されています. ちなみに, この新数学はアメリカでソビエトに対抗するための技術者育成を目的としたものですが, 集合, 論理, 群やトポロジーを大きく取り上げ, その一方で, Euclid 幾何学の比重が低下する結果を招くことになりました.
この様子を文部省の中学校学習指導要領[118]で眺めてみましょう. まず, 新数学以前の昭和 33 年度と新数学に対処した昭和 44 年度の数学の学習指導要領を比較すると, 従来は A:数, B:式, C:数量関係, D:計量, E:図形と分類されていたものが, A:数式, B:函数, C:図形, D:確率・統計, E:集合・論理と内容が一新されています. そして, 図形は双方にあるものの, 昭和 33 年度には Hilbert 流の公理的ではない図形の科学[119]とも言える Euclid 幾何学が主体であるのに対し, 昭和 44 年度では解析幾何学的なものに置き代わっています. この新数学はあまりにも急進的なものであったために現場での混乱を招き, さらには, 詰め込み教育への反発を起すこととなって最終的には消えています[120]. 実際, 昭和 52 年の指導要領を見ると, 最初に集合・論理が消え, それ以降は内容が減って行きます. ここで注意することとして, Euclid 幾何学は従来の位置に戻ることはなく, その比重は低下したままで, 数学全体の内容が削減されていることでしょう[121]
なお, 高等学校の学習指導要領では, 昭和 45 年度で初めて電子計算機と流れ図, 線形計画法が出ています. なお, 線形計画法は昭和 53 年度の学習指導要領からは消えていますが, 計算機は残り, 平成 10 年度の学習指導要領からは, 新たに「情報科」が増えているのは周知のことです. ただし, この情報科の内容が計算機を利用するための基礎と言うよりは, 非常に即席的な計算機利用[122]に留まっているのが現状です.

4.25.2 LISP と MACSYMA へ

さて, 数学の形式化は命題の意味を考える必要もない, 非常に機械的な処理を行うことによって, その命題の証明が行えることを意味するものです. このことはさまざまな数式の処理に留まらず, 最終的には数学の各種定理の計算機による自動証明の問題にも繋がります. さらにより一般的には, 計算機の基礎としての数理論理学にも関連して行くのです.
たとえば, MIT のテキストの「計算機プログラミングの構造と解釈」[123]の表紙を見てみましょう. ここではユダヤ教のラビと思われる怪しい老人と若い弟子の間に λ が顕示しています. 集合論で $\aleph$ というヘブライ文字が用いられていること, そして λ-表記可能な函数が一般帰納的函数と同値であること, そして, 老人がコンパスと eval と apply の太極マーク風の小道具 (Tao?) を手にしている考えると, この秘教的な表紙の別の側面が見えてより一層楽しくなるでしょう.
さて, Maxima は MIT の MAC 計画から誕生した MACSYMA を母体にしています. この MACSYMA を記述するために MACLisp が開発され, そして, その後継が Common LISP です. そして, MACSYMA や LISP にも共に Hilbert 計画の名残が他の言語と比べて顕著に残っているのです.

[118] http://www.nicer.go.jp/guideline/old/
[119] 小平邦彦, 「幾何学の誘い」 [23]
[120] 私はその末期に田舎の中高生でしたが, トポロジーは教科書にあっても教えられることはなく, 函数概念や群構造についても, 熱心に教わった憶えはありません. ただし, 教員や一般向けの「新しい数学」の解説書は豊富でした.
[121] 赤摂也, 「新講数学 I,II,III」のように異様に贅沢な高校生向けの参考書が消えて久しいことです.
[122] MS-Word や MS-Excel の使い方+Flash で遊ぼう!
[123] `http://mitpress.mit.edu/sicp/full-text/book/book.html` で原書の表紙を見たり, 全文が読めます.

第5章　Maximaの処理原理について

Faust	**ファウスト**
Mir hilft der Geist! Auf einmal seh ich Rat	我を助けよ, 精霊よ!一瞥で我は悟り
Und schreibe getrost: Im Anfang war die Tat!	確信をもちて書下す：原初に業(わざ)あり!
	Göthe,Faust より

この章ではMaximaの述語, 文脈, 宣言と属性の設定, そして, 評価の方法について述べます. これらの処理に関連する事項として演算子とLISPに関連した函数についても簡単に述べます.
Maximaの大きな特徴の一つに, 利用者が必要な述語や規則を定義し, その述語や規則を式に適用して式の評価を行うといった処理手順を踏む点です. これは単純に式を代入して函数で処理を行う場合に意識する事はさほどないでしょう．しかし, 函数によっては式に含まれる成分に関係する述語を付加したり, 処理に適した宣言を行うことで, より円滑に処理が行える様になります.
更に規則を上手く利用する事で作用素や代数的性質を付加する事が容易になります．そして作用素の定義で利用するMaximaの組込の演算子の特性を熟知していれば，より効果的な規則の適用が行えます.

5.1 Maxima の基礎概念

5.1.1 Maxima の原子

Maxima が扱うものを「**対象**」と呼びます. この対象には '1', '2', '3' のような「**数**」を指示する対象, "mikeneko" のように文字の羅列である「**文字列**」, 'true', 'false', 'unknown' のような「**意味**」をもつ対象, '%pi' や '%e' のように常に一定の値を持つ「**定数**」, また, 必要に応じて一定の値を保持する「**変数**」, あるいは「**変項**」と呼ばれる対象, そして, '1 + 2' の中に現われている記号 "+" のように変数, 定数, 数等の対象を繋ぎ合せて新しい対象を生成する機能を有する「**演算子**」と呼ばれる対象, 'x', 'x+1' のように単体, あるいは複数の対象を演算子と組合せることで「**式**」を表現する対象, また, 'matrix([1,2],[3,4])' のような「**函数の名前**」を表現する対象, さらには 'x:2' や 'if(x=1)then y:2 else y:1' のような Maxima によるさまざまな処理を指図する「**文**」と呼ばれる対象があります.

ここで具体例として数式 $(x+1)(y-2)$ を表現する Maxima の式 '(x + 1) * (y - 2)' を階層的に考えてみましょう. この式は二つの Maxima の式 'x + 1' と 'y - 2' を Maxima の演算子 "*" で結合したものとして捉えられますが, さらに式 'x + 1' と式 'y - 2' も分解すれば, 'x', '+', '1', 'y', '-', '2' の計 6 個の対象で構成されています. そして, これらの対象はより小さな対象には分解できません. このように, Maxima の対象として分解できない対象, すなわち, 最下層の対象を「**原子**」(atom) と呼びます.

この Maxima の原子は次に示す対象をその構成要素とします:

Maxima の原子

1. 記号, 文字, 文字列
2. 整数, 浮動小数点数, 多倍長浮動小数点数, 真理値
3. 函数, 演算子

記号 (symbol):　**symbol_壱_Z1** のようにアルファベットや数字, 日本語の漢字, 平仮名や片仮名といった各言語の印字可能な文字と alphabetic 属性を持った ASCII 文字[1]のならびで構成された対象です.

文字列 (string):　**"123"**のように二重引用符**"**で任意の計算機で表現可能な文字のならびを括った対象です.

整数 (fixnum,bignum):　**128** のように '0' から '9' の ASCII 文字に含まれる数字のならびだけで構成された対象, および, 数字のならびの先頭に, 符号として演算子 "**+**" や演算子 "**-**" を配置した対象. ここで fixnum と bignum は Lisp 処理系の整数の型で, fixnum が 1 語長 (Lisp の処理系で決定される), bignum が n-語長の整数です.

[1] ASCII コードで表現される文字

浮動小数点数 (float): **12.8** のように '0' から '9' までの ASCII 文字の数字と一つの小数点を表現する記号 "." で構成された対象, あるいは, **1280e-1** のような整数や **1.28e+2** のように記号 "**.**" を一つ含む数字の並びの末尾に "e⟨ 整数次数 ⟩", あるいは, d⟨ 整数次数 ⟩ を付加した対象, および, これらの対象の先頭に符号として記号 "+" や記号 "-" を配置した対象です.

多倍長浮動小数点数 (bigfloat): **1280b-1** のように整数, あるいは **1.28b+2** のように小数点を表現する記号 "." を一つ含む ASCII 文字の数字のならびに "b⟨ 整数次数 ⟩" を付加した対象. および, これらの対象の先頭に符号として記号 "**+**" や "**-**" を配置した対象です.

真理値 (boolean): **true** と **false**, あるいは **on** と **off** です.

函数 (function): **sin(2)** や **integrate(f(x),x,1,2)** のように ASCII 文字を先頭とする記号の羅列で, Maxima の対象の列を括弧で括ることで, 新たに Maxima の対象を生成することのできる対象です.

演算子 (operator): **1+3**, **3!**や **[1,2,3]** の '**+**', '**!**' や '**[]**' のように新しい Maxima の対象を生成する働きを持つものの単体では被演算子による充足を必要とするために意味を持たない対象です. なお, 函数に所定の属性を与えることで生成できます.
ここで「**属性**」は Maxima の対象に性質を付与する機能を持ち, 1 変項の命題函数として表現されます. Maxima の属性には, 先天的な属性と, 後天的な属性があります. 属性については §5.1.13 を参照して下さい.

なお, Maxima では「**有理数**」と「**複素数**」は原子ではありません. なぜなら有理数は整数対で与えられる式, 複素数は純虚数 '%i' の多項式として表現される対象だからです.

5.1.2 Maxima の記号 (symbol)

Maxima の「**記号**」は次の印字可能文字から構成される対象です:

Maxima の記号

- アルファベットを表現する ASCII 文字
- 0 から 9 までの数字を表現する ASCII 文字
- alphabetic 属性を持つ ASCII 文字
- 計算機で利用可能な各言語の文字

記号単体で「**変数/変項**」と呼ばれる対象を構成し, 式中では「**項**」, あるいは函数や演算子を指示する「**名前**」として利用され, この記号を基に様々な対象が生成されます. ここで, 記号の Maxima への入力では, '**\c\a\t**' のように, 各記号の先頭に記号 "\" [2] を置きます[3]. なお, この記号 "\" は Maxima の表示では省略されます. 実際, '\a' や '*' と入力すると, Maxima 上ではそれぞれ 'a' と '*' と表示されます. さらに, 記号 "\" を先頭に置く必要のない記号があり, それは, アルファベット

[2] "\" は backslash(バックスラッシュ) と呼びますが, 日本語環境では円記号 "¥" で表示される ASCII 印字可能文字です.
[3] "\" 自体は "\ \" となります

と数字を表現する ASCII 文字, alphabetic 属性を持つ記号と各言語の印字可能な文字が該当します. たとえば, '\a' と 'a' は同値です. このように記号 "\" が省略可能な記号のことを簡単に「**文字**」と呼び, 通常はこの文字を用いて変数/変項や函数名といった Maxima の対象の「**名前**」が構成されます.

文字や記号の羅列によって Maxima の変数/変項や数値が構成されます. このとき, アルファベットと数字以外の印字可能な ASCII 文字以外であれば, 先頭の記号を除いて記号 "\" の省略が可能です. さらに, 先頭の記号がアルファベットや後述の alphabetic 属性を持つ記号であれば, 記号 "\" が省略できます. したがって, 一々, '\c\a\t' と入力する必要はなく, 'cat' と入力するだけで済みます. ここで, 単体で文字となる数字については, 数字を先頭とする変数を構築する場合のみ, その先頭となる数字を Maxima の文字として明示的に指定する必要があります. つまり, 記号の羅列 '01234a' は変数として許容されませんが, '\01234a' は変数として使うことができます. なお, 変数 "x0" は, 数記号が先頭にないために, '\x\0' や 'x\0' と入力する必要はなく, 'x0' と入力できます.

最後に, 各言語の文字については LISP と OS 等の利用環境に大きく依存するために一概に言えません. ただし, ここで前提としている OS(UNIX 系や MS-Windows) の日本語環境と CLISP であれば, 記号として日本語の文字が使えます. 勿論, 他の計算機環境で利用することを考慮すれば, その文字コード (EUC, JIS, SHIFT-JIS, UTF8 等) にも注意を払わなければなりません.

alphabetic 属性:　Maxima の記号や文字列に対して declare 函数 を用いて利用者が付与できる属性です. ここで属性の概要は §5.1.13, その詳細は §5.4 を参照して下さい. この alphabetic 属性を持った記号は先頭から記号 "\" を外して入力することが可能となります. さらに alphabetic 属性を持った文字は内部変数*alphabet* に割当てられた LISP のリストの一成分になります (内部変数の概要は §5.1.15 参照). ここで内部変数*alphabet*の既定値として '(_ %)' が割当てられています[4]. なお, 演算子や Maxima で意味を持つ文字として既に利用されている ASCII 文字に対し, alphabetic 属性を与えることは非常に危険です. ここで Maxima で意味を持たない文字とは, 何の属性も与えられていない ASCII 文字と言い換えられます.

ここで初期状態で一切の属性を持たないアルファベットや数字以外の ASCII 文字を示しておきます:

[4] LISP 内部では, '(#_ #\%)' です.

属性が指定されていない ASCII 文字符号

16 進表現	文字	注意事項
34	"	文字列の定義で利用
29	)	matchfix 型演算子 (と組合せて利用
2C	,	列で利用 (alphabetic 属性を絶対与えてはならない!)
5C	\	文字を意味する特殊記号
5D	]	matchfix 型演算子 [と組合せて利用
60	‘	とくになし
7E	~	外積演算として利用

この中で記号の先頭に置いたときに ASCII 文字符号 “\” が省略できる記号は記号 “ ‘ ” と記号 “ ~ ” のみですが, 記号 “ ~ ” は vect パッケージの外積演算子として用いられているので, その利用には注意が必要です.

5.1.3 Maxima の文字列

任意の記号, あるいは文字の羅列を二重引用符 ‘”’ で括った対象です. 二重引用符で括られる羅列では記号”\” は二重引用符を記号として含む場合を除いて不要です. つまり, ”cat” でも”\c\a\t” でも文字列としては同値ですが, 文字列 ”ここで彼女は \”あっ\”と言った” の内部の記号 “\” は省略できません. この意味では, Maxima の文字の羅列と考えた方が正確でしょう.

なお, 区切記号 “,” で区切られた Maxima の対象で構成された対象のことを「**列**」と呼び, 文字列と文字の列を別物として区別します. 実際, 「**文字列**」”123”と「**文字の列**」1, 2, 3 が違うことは明らかでしょう. 実際, Maxima で文字列は内容が評価されませんが, 文字の列では, その内容が評価されるという大きな違いがあります. ただし, 文字列に値や属性を与えた場合, 文字列全体が Maxima で評価されます.

ここで文字列には Maxima の版によって実体が異なることに注意が必要です. Maxima-5.13.0 以前は LISP の文字列型とは異った内部データで, 大文字と小文字の区別がありません. ところが, Maxima-5.14.0 からは LISP の文字列型になります (§6.10,§6.4.1 参照).

文字列は演算子の定義や演算子の処理で用いられますが, この本では, 演算子の表記は問題が無ければ二重引用符を外した表記も利用します.
また, 文字列は記号のように項として利用することや値を割当てることもできます. ただし, このような利用方法はプログラム言語一般でも正規的な利用方法ではないでしょう.

5.1.4 整数の表現

Maxima の整数は **123** のように ‘0’ から ‘9’ までの数字だけで構成された対象です. この Maxima の整数には通常の整数を表現する「**fixnum 型**」と可変桁の「**bignum 型**の二種類の型があります.

これらの型の整数は LISP の整数表現にそれぞれが対応し, これらの整数の差異を利用者が直接認識することはまずありませんが, fixnum 型の上限は内部変数 fixnbound に割当てられた LISP の most-negative-fixnum の値で定められており, この値以上の整数は自動的に内部で bignum 型となります.

5.1.5　実数の表現

Maxima の実数の表記には大きく分けて 4 種類の表記方法があります:

Maxima の実数表記

●小数点表記:	0 から 9 迄の数字の羅列に小数点 "." を一つだけ入れた対象
●E-表記:	整数対, 或いは小数点表記と整数の対の間に e を入れた対象
●D-表記:	整数対, 或いは小数点表記と整数の対の間に d を入れた対象
●B-表記:	整数対, 或いは小数点表記と整数の対の間に b を入れた対象

Maxima では実数を表現する対象として浮動小数点数を用います. この浮動小数点数にも整数と同様の二種類の型があります. まず, 通常の浮動小数点数に対応する「**浮動小数点数**」, そして, Maxima 独特の内部表現を持つ「**多倍長浮動小数点数**」です. これらの浮動小数点数を簡単にまとめておきましょう:

浮動小数点数:　浮動小数点数は小数点表記,E-表記と D-表記で表現される実数です.
たとえば, '120.0', '1.2e+2', '1.2d2', '1200d-1' は全て同じ浮動小数点数 120.0(=実数 120 の近似) を意味します.
Maxima の浮動小数点数は LISP の倍精度浮動小数点型 (float) であり, その上限は内部変数 flounbound で指定され, その値は LISP の most-negative-double-float 変数の値になります.

多倍長浮動小数点数:　多倍長浮動小数点数は B-表記で表現される実数です. たとえば, '120b0', '1.2b+2', '1200b-1' は全て同じ多倍長浮動小数点数 120b0 を意味します. この多倍長浮動小数点数の桁数の制約は計算機の記憶容量による制約を除いてありません. また, 多倍長浮動小数点数は Maxima 内部でリストで表現されていますが, 有理数と異なり Maxima の原子として扱われています. ただし, 多倍長浮動小数点数の書式は **IEEE-754** の定める浮動小数点数の書式とは異なる独特のものです.

5.1.6　真理値の表現

Maxima の真理値には 'true' と 'false' があります. そして, これらの真理値は LISP の 'T' と 'NIL' にそれぞれ対応します. なお, これら他に 'on' と 'off' もありますが, これらは 'true' と 'false' の異名です.

他に重要な対象に 'unknown' があります. こちらは「**述語の値**」[5]の真偽の判定が不能な場合に用いられる Maxima 独自の定数で, この 'unknown' に対応する LISP の真理値はありません.

この本では「**狭義の真理値**」の集合を {true, false} とし, 「**広義の真理値**」の集合を {true, false, unknown} とします. ここで単に「**真理値**」と呼ぶ場合には「**狭義の真理値**」を意味するものとし, 「**広義の真理値**」を意味する場合には注意書きを入れます. そして, 述語の評価, つまり, 述語の真理値を求める計算処理を「**判断**」と呼びます.

なお, Maxima の大域変数 prederror が 'false' の場合に広義の真理値集合 {true, false, unknown} で論理式の判断が実行されますが, この大域変数の値を 'true' に切換えると 'unknown' は 'false' に包含されて「狭義の真理値集合」{true, false} で論理式の判断が実行されます.

5.1.7 有理数と複素数の表現

有理数と複素数は Maxima では原子ではありませんが, これらの表現についても解説しておきます:

有理数と複素数	
●**有理数:**	二つの整数の間に演算子 "/" を置いた対象
●**複素数:**	二つの数 a と b に Maxima の純虚数`%i` を和の演算子 "+" と可換積の演算子 "*" を用いて構築した式 'a + %i * b'

有理数は Maxima 内部で演算子 "/" の対象として表現されています. そして, 複素数も同様に Maxima の式として表現されています. ここで, これらの対象に対応する有理数型や複素数型の対象が Common Lisp にありますが, Maxima ではこれらの型をどちらも利用していません (§6.4 参照). そのために, これらは Maxima の原子ではありませんが, 自動簡易化によって有理数や複素数が整数や実数に変換されることもあります.

5.1.8 Maxima の変数

Maxima の変数 (=変項) となりうる Maxima の対象は, Maxima の記号に限定されます. Maxima の変数には Maxima の対象をその値として割当てたり, 属性を付与することもできます. Maxima の処理文や函数で「Maxima の対象によって充足されるべき場所」[6] を示す対象として利用出来ます.

ここで Maxima の変数には二種類存在します. ひとつは Maxima 上で属性や値が割当てられていない変数で, これを「**自由変数**」と呼びます. もう一つは Maxima 上で属性や値が割当てられた変数, Maxima の函数や処理文の内部処理で割当てを伴うような変数で, これを「**束縛変数**」と呼びます. そして, 束縛変数は大域変数 values に自動的に登録され, kill 函数を使って大域変数 values

[5]「述語の値」を「述語の意味」とも記述します.
[6]Frege の函数の考え方 (§4.19) を参照.

から変数名を削除することで自由変数にすることができます.

具体的に束縛変数について説明しておきましょう. まず, ‘x’ と ‘sin(x*y)’ のように単なる項として Maxima 上にはじめて現われる変数 x, y は自由変数で, 式 ‘x:10’ や式 ‘y:sin(a)’ で値が割当てられた変数 x, y, あるいは処理文 ‘for i in [1,2,3] do i*2’ の変数 i, 函数の定義文 ‘neko(x):=x+1’ の変数 x が束縛変数になります. また, λ 式 ‘lambda([x],x^2 + y)’ の変数 x は束縛変数になりますが, 変数 y は, この λ 式ではじめて出現した変数であれば自由変数になります. この束縛変数は演算子や記号と組合せて利用することを前提とする変数です. 値を持った変数が束縛変数というのは不可思議かもしれませんが, たとえば, 式 ‘x:10’ は ι 記号[7] を用いた論理式 ‘$\iota_x(x = 10)$’ と同値であると考えれば, 変数 x が束縛変数であることが明瞭となるでしょう.

なお, 与えられた自由変数を束縛変数として使えるように, 変数に bindtest 属性を与えることができます. bindtest 属性を付与された変数に実際に値が割当てられるまで, Maxima で入力を行うとエラーが生じますが, 一度, 値を割当てると以降は普通の変数として扱えます.

5.1.9　Maxima の函数と演算子

函数と演算子の概要

Maxima の函数と演算子は, Maxima の記号から構成され, Maxima 上で様々な処理を行うための対象です. 函数と演算子には, 名前と呼ぶ Maxima の記号の羅列があり, 函数であれば, その名前の直後に括弧 “()” で括られた変項や定数等の引数の列が続きます. また, 演算子であれば, その演算子の型に応じて, 変項や定数等の引数が演算子の名前に対して配置されます. こうして, 函数や演算子が機能する最小の対象のことを, 函数であれば函数項, 演算子であれば演算子項と呼びます.

たとえば, 対象 ‘sin(x)’ が函数項, この函数項の括弧 “()” の前の記号 “sin” が函数の名前, 括弧の中の “x” が函数項の引数, あるいは変数/変項となります. また, 対象’x*y’ であれば, ‘x*y’ が演算子項, 記号 “*” が演算子名, 対象 x と対象 y が演算子 “*” の被演算子です. ここで, 演算子は, 演算子項の表記に適合する属性を付与するまでは演算子として扱えず, 演算子としての属性を付与した時点で自動的に大域変数に演算子の名前が文字列として登録されます (演算子の詳細は §5.3 参照).

なお, 函数と変数は Maxima では別の対象のために同名の函数と変数が共存できます. たとえば “quit()” と “quit” は前者が函数, 後者は変数として扱われます.

函数と演算子の評価

Maxima は入力した対象に対して自動的に評価を行います. この評価では対象に含まれる函数, および演算子等の属性, さらには函数や演算子固有の処理函数を用いて対象の処理が行われます (§5.8.2

[7] ι 記号については, 「数学の基礎」[44] や「数学基礎論入門」[55] を参照して下さい.

参照).

このときに函数の書式で記述された対象は, Maxima 上で未定義なものでも Maxima の函数として扱われますが, 未定義の演算子を含む式はエラーになります. これは演算子には属性に対応する書式があるためで, ある対象を Maxima の演算子として利用するためには, 予め, 書式で用いようとする演算子属性を与えておかなければなりません. そして, 式の評価では式の変形に関連する属性を持たない演算子, 実質を持たない函数を含む式に対しては項順序 "$>_m$" によって演算子項や函数項の並べ換えが行われる程度です.

函数項や演算子項を効果的に処理させるためには函数, 演算子に属性を与えたり, その実体となる処理函数や Maxima に既存の簡易化函数が利用する規則等を定義しなければなりません.

函数と演算子の実体

演算子は Maxima の函数に演算子としての属性を与えたものとして考えられるために, ここでは函数について主に述べることとします. ここで Maxima の函数は次の四種類に分類できます:

Maxima の函数

1	形式的な函数
2	Maxima 言語で記述した函数 (§8.1 参照)
3	LISP で直接記述され, 利用者が直接利用可能な函数
4	LISP で直接記述され,Maxima の裏で内部処理を受け持つ函数

ここで形式的な函数とは, 'f(x)' のように Maxima の函数としての書式を持つものの, その実体, すなわち, その定義文を持たない函数です. このような函数に対しては, その函数に付与された属性があれば, その属性に応じた処理のみが実行されます. ここで函数の属性は declare 函数等で与えられます (§5.4 参照). したがって, 函数項の処理函数の実体は属性の表現函数で代用されることになります (§5.4.7 参照).

Maxima から直接利用可能な函数は最初の 1 から 3 の 3 個の函数です. これらの函数を「**Maxima の函数**」と呼び, 最後の「Maxima の裏で動作する函数」のことを「**内部函数**」と呼びます. この内部函数は利用者にとっては直接見えない LISP の函数です. なお, Maxima の函数は内部函数を組合せて構成されたもので, Maxima に入力された式は, 一旦, 内部表現に変換されて式の内部表現の型や属性に対応する内部函数を使って処理されます (§7.2 等参照).

ここで Maxima の函数と演算子は "((%函数名 *SIMP*) 変数$_1$ $\cdots$ 変数$_n$)" の書式の内部書式を持っています. たとえば, sin 函数の場合, 函数項 'sin(x)' の内部表現は '((%SIN SIMP) X)' となり, 内部での函数名は記号 "%" を先頭に置いたものとなります §6.4 参照). この表現の書式は演算子も同様ですが, 利用者が定義した演算子の内部での名前では, 函数とは異なり記号 "%" ではなく記号 "$" が先頭に置かれ, 演算子名もソースファイル nparse.lisp にて def-mheader マクロで指定された名

前があれば, その名前が内部表現では用いられます.

なお, Maxima の函数項には二つの形態があります. 一つは「**名詞型**」と呼ばれ, この場合は項の評価が行われません. もう一つは「**動詞型**」と呼ばれ, この場合は項の評価が行われます. ここで函数名を f とするときに, Maxima 内部では '%f' で名詞型を表現し, $f で動詞型を表現します. なお, 未定義の函数を入力した場合, Maxima は自動的に名詞型の函数が与えられたものとして Maxima は処理を行います. なお, 演算子については, 名詞形はありません.

最後に, 利用者が実体を定義した函数と演算子は大域変数 functions に自動的にその名前が登録されます. 逆に言えば, この登録によって函数は函数項と函数の実体が関連付けられています. そのため, 大域変数 functions から函数名を削除すると, 函数項は実体との関連を失うために形式的な函数となります.

5.1.10　マクロ

Maxima の函数に似た対象として「**マクロ**」があります. これは Maxima の対象の内部表現が S 式であることを応用し, 対象から対象への写像としての性質を持ちます. マクロは定義されると, その名前が大域変数 macros に登録されます. このときに函数とマクロは排他的な関係にあります. すなわち, 同名の対象は大域変数 functions か大域変数 macros のどちらか一方のみ存在が許容されます. したがって, 大域変数 functions と大域変数 macros のどちらか一方に既存の名前を新規に登録する場合, 古い名前の側が上書き, あるいは自動的に削除される仕組になっています.

5.1.11　配列

函数に似た性質を持つ Maxima の対象に配列があります. 配列項は函数項に似た表記で, 配列名の直後に大括弧 "[ ]" で整数で構成された列を括った対象です. この列を構成する整数のことを「**添字**」, 列の長さのことを「**配列の次元**」と呼び, それから, 各列の取り得る整数値の最大値を「**配列の大きさ**」と呼びます. 配列の添字は C とは違って 0 からではなく 1 から開始します.
配列の生成には二つの方法があり, 一つは array 函数で名前と配列の大きさ指定する方法で, こちらは内部で LISP の make-array 函数を用いられています. もう一つの方法は, 形式的な配列の表記に直接, 値を束縛させる方法です. これらの手法で生成した配列は全てその名前が大域変数 arrays に登録されます. また, 配列の実体の削除は大域変数 arrays から配列名を削除することで行えます.
なお, 配列はリストと異なり, その名前は通常の変数として認識され, 配列 a と変数 a が両立し得ることになります. 配列 a の内容が Maxima 側で認識されるのは, 値を参照した場合に限定されます. その意味では函数に似た対象です.

5.1.12　リスト

Maxima の対象を一切合切, 大括弧 "[ ]" で括って構成される Maxima の対象で, 成分の取出は配列と同様に 1 から開始する添字を使って取出すことができます. リストは配列とは異なり, 自由な大

きさの配列を構築することが可能であり, さらに, MATLAB 系の言語と同様の, 成分単位の四則演算と Maxima の式の含めた利用も可能です.

5.1.13 属性

属性とは対象の性質です. この属性には変項/変数が一つの述語が対応します. たとえば,「ξ は奇函数である」のように具体的な項で充当すべき変項 ξ を一つだけ持つ論理函数 (述語) です. さて, この述語を「奇函数 (ξ)」としましょう. すると函数 f が奇函数であることは「奇函数 (f)」が真となることと同値ですが, このことを [f, 奇函数] と対で表現したものが, Maxima の属性の表現の基本的な考えで, この属性を表現するために LISP の属性リストを利用する方法, 属性リストを模した Maxima のリスト構造を用いる方法や函数形で表現する方法が Maxima の内部で用いられています.

ちなみに, この手法は Russell の還元可能性公理 (Axiom of Reducibility): 「**任意の命題に対して同値な述語が存在する**」を思い出させるものです. この還元可能性公理は集合の内包性と密接に関連する公理です.

属性は Maxima の内部データとして表現されていますが, 属性によっては利用者が固有の大域変数に纏めた属性を参照したり, 変更することができます. ここで対象に属性を付加する具体的な方法として, 次の三つの方法があります:

対象に属性を付加する方法

1	put 函数を用いて対象に属性を付加する方法.
2	declare 函数を用いて対象に属性を付加する方法.
3	内部函数 defprop 函数[8]等を用いて対象に属性を付加する方法.

利用者が任意の対象に属性を与えることに適しているのが 1. の put 函数を用いる方法です. これに対し, declare 函数や defprop 函数を用いる手法は, Maxima による式の変換と深く関係します.

まず, declare 函数を用いることで利用者が函数や演算子に線形性等の性質を付加できます. さらに演算子に対しては, 演算子型や被演算子に対する演算子の束縛力の大きさの指定出来ます. このように declare 函数は数学上の性質の付与で用いられ, その性質に応じた内部函数による処理が行われます.

それに対し, 内部函数 defprop 函数を用いる方法は, 利用者が LISP でパッケージを記述したときに主に用いる方法です. ここで行えるのは, 式を入力した時点で自動的に簡易化を行うための内部函数の指定, 微分公式等のさまざまな性質や TEX の書式に変換する際に用いる雛形の設定ができます (§5.4,§5.8.2,§6.4.13 参照).

[8] 正確には LISP のマクロ

5.1.14　Maxima の式

Maxima の式の構成

最初に Maxima の基本的な式について述べます:

Maxima の式
1. 記号, 文字列, 数値, 形式的な函数項や配列項の単体
2. 1. を出力する函数項, 配列項の単体
3. 1. と 2. を演算子で結合することで得られた対象
4. 3. を成分とするリスト
5. 4. を出力とする函数項や配列項
6. 大きさが矛盾しない 4. のリストや 5. の対象を演算子で結合した対象
7. Maxima の対象を用いて定義された対象を含む式

Maxima の基本的な式は, 1. の記号や文字列で名前を指示される変数, さらに, 整数, 有理数, 浮動小数点数や複素数から構成される数値を基に, 2. の函数項や配列項, 4. の演算子で結合することで得られた対象となります.

この基本式を基に, 新たな Maxima の対象が構成されます. その最も重要な対象がリストで, 4. の基本式を成分とするリスト, 6. の大きさが矛盾しないリストを演算子で結合した対象も Maxima の式となります. このリストの構築方法と同様に, Maxima の対象が構築可能です. たとえば, 行列は matrix 函数によって Maxima の式を成分とするリストから生成されます.

この行列のように, Maxima 上で新しい対象も柔軟な LISP の S 式を用いて容易に定義が可能です. 7. で述べた対象は行列のように定義された対象について演算子が矛盾なく定義されており, 必要であれば従来の演算子とも互換性を持つことが要請されます.

Maxima の式の内部表現

Maxima の式は, 表で見えている式と内部で処理されている式では異なった書式を持っています. Maxima 内部で処理される式の表現を内部表現と呼びます. この内部表現は S 式風の前置式表現であり, 式を構成する各項は項順序 “$>_m$” で並び換えられ, 同様に, 演算子項や函数項も演算子や函数の属性によって項順序 “$>_m$” による並び換えが生じることがあります. そして, 函数項や演算子項の名前は “()” で括られて変項よりも一段高い対象であることが明瞭となるように表現されています. また, Maxima の変数項は先頭に “$” が置かれた LISP の記号 (symbol) になります.(§6.4 参照).

Maxima の部分式と項

Maxima の式の「**部分式**」は与式の一部分であり, それ単体で Maxima の式となりうる Maxima の対象です. さらに, 「**項**」は, 和の演算子 “+” や差の演算子 “-” で区切られた部分式のことです. たとえば, 式 ‘a + b * c + d’ の部分 “a + b * c” は与式の部分式で, “a”, “b * c” と “d” が与式を構成す

る項になります. ただし, “a+” や “+b*” のような半端な部分は Maxima の式にはならないために部分式にも項にもなりません.

Maxima に式を入力すると, 式の各項は Maxima の項順序 “$>_m$” で一意に並び換えられます (§5.2 参照). その際に式の単純な簡易化 (代入や項のまとめ) も式の型や大域変数の設定, 函数や演算子の持つ属性に応じて実行されます (§7.2 参照).

5.1.15 大域変数

Maxima にはシステムや函数の制御を行う特別な変数があり, その性質上, 次の二種類の変数に分類されます:

大域変数の種類

大域変数:	Maxima 側から利用者が直接利用可能な大域変数
内部変数:	LISP 側の内部処理で専ら利用される大域変数

ちなみに, これらの変数は LISP の Special 変数が用いられています.

大域変数: 函数やシステムの制御を行うために利用者が必要に応じて演算子 “:” で値を設定できたり, 大域変数名を入力することでその値が参照できる変数です. Maxima 内部では通常の変数と同様に先頭に記号 “$” が付きます.

内部変数: システムが専ら利用することを想定しているため, 表から見えない変数です. 内部変数の命名基準は変数名の先頭と末尾に記号 “*” が置かれた名前 (慣習的な LISP の Special 変数の命名方法) を採用しています. 例外も幾つかありますが, 先頭に記号 “$” が付くことはありません. この内部変数の処理は基本的に LISP 側で処理します (§3, §5.9 参照).

5.1.16 Maxima の論理式

Maxima の論理式は次の方法で構成された Maxima の式です:

Maxima の論理式

1.	真理値の true と false
2.	二つの Maxima の対象を二項間の関係の演算子 (“=”, “>”, “<”, “>=”, “<=”, “#”) で結合した対象
3.	equal 函数項,notequal 函数項
4.	Maxima の真理函数を Maxima の式に作用させた式
5.	1., 2., 3., 4. の式に論理演算子 (“and”, “or”, “not”) を用いて構築した式

1. の true と false は Maxima の真理値そのものです.

2. は式 ‘4>1’ や式 ‘x<1’ のように関係の二項演算子を用いることで二項間の関係を示した Maxima の式です. なお, 関係の演算子を用いて構成された式は Maxima に入力しても直ちに解釈はされません. is 函数, maybe 函数や ev 函数等の論理式の評価を行う函数で評価されることで, その真理値が返されます.

3. の equal 函数や notequal 函数は関係の演算子 “=” と演算子 “#” に似た意味を持つ函数ですが, 演算子 “=” や “#” と異なり, assume 函数 で使えます. これらの函数も論理式の自動的な評価を行いませんが, これらの函数の評価では演算子 “=” と “#” と異なり, 有理式に対して有効な ratsimp 函数による簡易化が実行される利点があります.

4. の真理函数は, 文脈 (§5.1.17 参照) と呼ばれる Maxima の機構を利用することで被演算子として与えられた論理式の評価を行い, true や false 等の真理値を返す函数です. なお, 論理学での真理函数は真理値集合から真理値集合への写像ですが, ここでの真理函数の定義域は真理値集合に限らない一般的な Maxima の対象で, 何等かの評価を伴う函数を指しています. たとえば, ‘atomp(x)’ のように Maxima 組込の真理値を返す函数項, ‘is(x>=0)’ のように論理式の評価を行う函数項, あるいは利用者が構築した Maxima の対象を引数とし, 返却値が真理値となる函数項です.

5. は Maxima の真理函数の構成方法を述べたものです. 論理演算子 “and” , “or” , “not” は論理式の評価を伴う演算子です. ここで演算子 “and” と演算子 “or” は被演算子を文字通り字面で評価するために被演算子の解釈を的確に行う函数 (例えば ev 函数) と組合せて利用する必要があります. そして, Maxima の論理式は文脈と呼ばれる Maxima の対象を用いて, その評価が行われます.

なお, Maxima の論理式で自由変項を持つ論理式を述語と呼びます.

5.1.17　文脈

文脈は Maxima の論理式の保管庫として用いられる対象です. 正確には, 文脈は subc 属性を持った Maxima の記号です. この subc 属性によって他の文脈間の親子関係が規定され, 全体として木構造の階層構造が入ります. subc 属性自体は LISP の属性リストで表現され, 子文脈名が属性値として登録されています (§5.6 参照).

Maxima の述語は assume 函数によって, 文脈の data 属性の属性値として LISP の属性リストを使って登録されます. ここで文脈に登録可能な論理式は以下の方法で構成されたものに限定されます:

Maxima の文脈

1.	二項間の大小関係 (“>”, “>=”, “<”, “<=”) を示す述語
2.	equal 函数による二項間の同値性
3.	notequal 函数による二項間の非同値性
4.	1., 2., 3. の述語の演算子 not による否定
5.	1., 2., 3., 4. の述語を演算子 and で繋いだもの

ここで文脈に登録された論理式を '$P_1, \ldots, P_n$' としましょう. ここで Maxima に論理式 'P_0' が与えられたときに, その論理式の評価は論理式 '$P_0 \wedge P_1 \wedge \cdots \wedge P_n$' で行います. つまり, 文脈は変項を ξ とする述語 '$\xi \wedge P_1 \wedge P_2 \wedge \cdots \wedge P_n$' です. このことが演算子 "or" を含む述語を文脈が受入れられない理由となります.

5.1.18 規則

規則によって演算子や函数の変換が行えます. ここで規則とは, 「**式の並び**」が与えられたときに演算子や函数に対して簡易化函数が行うべき処理を定めたものです. 公式と規則の違いは, ある対象の固有の属性として与られる対象が公式, ある特定の Maxima の対象で構成された式に対して特有の変換を定めたものが規則となります.

具体的には, $\sin 2x \Rightarrow 2\sin x \cos x$ のような数式の変換は sin 函数の性質として与えられます. ところが, その逆の $2\sin x \cos x \Rightarrow \sin 2x$ は函数 $f(x) = \sin x \cos x$ の属性として与えられなくもありませんが, むしろ, $\sin x \cos x$ のような「式の並び」が出現したときに $\sin 2x$ に変換することを定めた方が自然ですね. このように公式は「**対象の属性**」, 規則は「**式の並び**」に対して与えられます. そのために規則の定義は「式の並び」を定義し, その「式の並び」に対して規則を定める手順となります. ここで,「式の並び」では決った定項だけではなく, 函数を扱う必要があるためにさまざまな指定を行う必要があります (§5.7 参照).

5.1.19 式の自動簡易化

大域変数 simp の意味が 'true' であれば Maxima は自動的に式の簡易化を行います. ここで自動簡易化で和 "+", 可換積 "*", 商 "/" 以外の演算子 (以降, 簡単に演算子と略記します), および, 函数の属性を用います. この簡易化で用いられる内部函数の種類を次で纏めておきましょう:

演算子と函数の自動簡易化で用いられる内部函数の種類

一般的な性質	⇔	内部変数`*oper-list` に登録された属性の表現函数
函数固有の性質	⇔	operators 属性に対応する内部函数

線形性のような一般的な演算子や函数の性質は, declare 函数を用いて属性として函数や演算子に付加させ, その属性を用いて与式を自動的に簡易化することができます. この仕組は与えた属性に沿って式を変換する函数である「**属性の表現函数**」と属性のリストを成分とする内部変数*opers-list を Maxima が持っており, 内部函数 simplifya で入力された式の演算子項 (演算子と被演算子で構成された部分式) や函数項 (全ての変数が充足された函数) に対し, その演算子や函数の持つ属性と *opers-list に登録された属性の照合を行なって属性が一致すれば, その属性の表現函数を項に作用させるというものです (§5.4.7 参照).

函数固有の性質による簡易化は内部函数 defprop によって函数毎に operators 属性として与えられた内部函数を函数項に作用させて行います. 具体的には内部函数 simplifya にて, 函数項に対し,

その函数固有の簡易化函数を項に作用させて函数項の簡易化を実施しています (§5.8.2 参照).

なお, 演算子項や函数項だけではなく, より一般的な式の自動簡易化を行う方法として tellsimp 函数や tellsimpafter 函数 による規則を用いた方法があります (§5.7.7 参照). これは内部函数 simplifya に演算子や函数の処理函数を追加する方式と言える方法ですが, この処理を行うためには規則を適用するための「**式の並び**」を予め定義しておく必要があります (§5.7 参照)

式の簡易化では文脈を Maxima は利用します. この文脈による評価では大小関係と同値性を用いて符号処理等による推論が行われます (§5.6 参照).

5.1.20　まとめ

Maxima は与えられた対象が持つ属性, 対象が置かれた文脈に蓄積された論理式を前提に, 大域変数によって制御された Maxima の函数群を用いて処理を行います. この処理は結局, 論理式の機械的処理に他ならず, その意味で Maxima は Hilbert 計画直系の子孫 (「ロボット式数学」[31]) の側面を強く持っています. そして, この属性を活用することで, Maxima は非常に古い数式処理システムでありながら, あらゆる問題に対応出来る柔軟さが得られているのです.

5.2 順序

5.2.1 Maximaの変数順序

Maximaの変数となり得る対象は記号です. そこで, Maximaの記号の集合を A_m と表記します. 次に, この集合 A_m に順序 "$>_m$" を入れます. この順序 "$>_m$" について解説しましょう.

まず, 記号がアルファベットであれば大文字が小文字よりも大という関係が入ります. すなわち X をアルファベットの大文字, x を X に対応するアルファベットの小文字とするとき, '$X >_m x$' は真になります. つぎに, 大文字に限定して比較すると, 記号 "Z" が記号 "A" よりも大, 同様に小文字に限定すると, 記号 "z" が記号 "a" よりも大きいという順序が入ります. さらに, alphabetic 属性を与えた記号は, アルファベット大文字の "Z" よりも大きく, ordergreat 函数の最後の引数よりも小さくなるという順序が入ります. つぎに, "0" から "9" 迄の数字に関する大小関係は通常の数の大小関係 ">" と同値です. それから残りの ordergreat 函数や orderless 函数による順序とは無関係な記号については, ASCII 符号表の 10 進表記で比較されます. この記号の大小関係を判定では内部の LISP の great 函数が使われています.

順序 "$>_m$" は Maximaの式を一意に決めるためにも重要な役割を果たします. 実際, 変数名や函数名は Maximaの記号の羅列として表現されるため, 順序 "$>_m$" によって記号の間に大小関係を入れることができれば, その順序に従って変数名や函数名で並び替えが行えて式の表記は一意に定まります. ここで, 変数名が一つの記号であれば順序 "$>_m$" による比較は分かり易いものですが, 実際は "abc" のように変数名は複数の記号で構成されているので, 記号の羅列に対しても順序 "$>_m$" で順序付ける必要があります.

この方法を簡単に説明しましょう. まず, 二つの変数が与えられ, 一つの変数名を $x_1x_2\ldots x_n$, もう一方の変数名を $y_1y_2\ldots y_m$ とします. ここで x_i, $(1 \le i \le n)$, y_j, $(1 \le j \le m)$ を Maximaの記号とします. このときに $n = m$ でなければ短い変数の側に空白文字を入れて双方の変数名の長さを合わせて同じ記号数の変数名にできます. もし, $i = 1,\ldots,k-1$ に対し $x_i = y_i$, k 番目の記号 x_k と y_k ではじめて異なるとすると, 二つの変数名の順序を順序 "$>_m$" に対し, 記号 x_k と記号 y_y の順序で定めます. つまり $x_k >_m y_k$ であれば $x_1x_2\ldots x_n >_m y_1y_2\ldots y_m$ とするわけです. たとえば二つの変数 abc と aaz が与えられたとき, 双方の変数名の最初の記号は記号 "a" なので二番目以降の記号が大小関係を定めます. ここで, 二番目の記号にて "b"$>_m$"a" となるので 'abc $>_m$ aaz' が得られ, 変数 abc の方が変数 aaz よりも上の順位となります.

Maxima内部の式を構成する変数に対し, 項順序 "$>_m$" で最も順位が高い変数を「**主変数 (mainvar)**」と呼びます. ここで, declare 函数を使って mainvar 属性を付与した変数は Maximaの変数の順序 "$>_m$" に対して最上位の変数となります. そして Maximaの式は主変数を 1 変数とする多項式としてまとめることで式の見通しが良くなるために「**CRE 表現**」(§6.2.2 参照) への変換等の式の処理で多く用いられています.

この Maximaの変数順序を次に纏めておきましょう:

Maxima の変数順序 “$>_m$”

主変数属性を持つ変数	$>_m$	ordergreat の引数$_1$	$>_m$	...	$>_m$
ordergreat の引数$_h$	$>_m$	alphabetic 属性記号$_1$	$>_m$	...	$>_m$
alphabetic 属性記号$_k$	$>_m$	先頭が Z の変数	$>_m$	...	$>_m$
先頭が A の変数	$>_m$	先頭が z の変数	$>_m$	...	$>_m$
先頭が a の変数	$>_m$	orderless の引数$_n$	$>_m$	...	$>_m$
orderless の引数$_1$	$>_m$	宣言されたスカラー	$>_m$	宣言された定数	$>_m$
Maxima の定数	$>_m$	9	$>_m$	...	$>_m$
0					

ここで順序 “$>_m$” は全順序となるために $(A_m, >_m)$ は全順序集合になります.

5.2.2　項式項に対する順序

Maxima の式を構成する項に対して変数の順序 “$>_m$” を拡張します. Maxima の差 “-” を含む式は, Maxima 内部では和 “+” と “-1” の積の式で表現されます. たとえば, 式 ‘a - b’ は内部で ‘a + (-b)’ と表現されます. このように, Maxima の式は内部では演算子 “+” で区切られた部分式に分解されます. この部分式を「**項**」と呼びます. これらの項に対して変数順序 “$>_m$” を自然に拡張した項順序 “$>_m$” は「**辞書式順序**」と呼ばれる順序になります.

この変数順序 “$>_m$” の項への拡張について多項式に限定して解説しましょう. まず最初に Maxima の項を構成する変数を “$>_m$” が定める順序に従って大きなものから並べて列を構成します. たとえば, 項の変数を $x1$ から $x9$ とするとき, ‘$x9 >_m x8 >_m \ldots >_m x1$’ となるので, 並び換えを行うことで変数の列 $x9, \ldots, x1$ が得られます. このような変数の置換は Maxima で自動的に実行されます. さて, 二つの項が入力されると各項の変数は順序 “$>_m$” で並び換えられ,　二つの変数の列から項の「**次数リスト**」を構築します. 次数リストは最初に二つの項を構成する全ての変数を順序 “$>_m$” で変数を並び換え, 次に項毎に変数に対応する次数を左から順に並べてえられた整数リストです. なお, ここで片方の項にない変数があれば次数 0 を入れます. たとえば, 二つの項 $x1\,x2^2\,x8^3$ と $x1\,x2^2\,x3\,x9$ が与えられた場合, 二つの項を構成する変数 $x9, \ldots, x1$ の順序にしたがって変数置換を行います. その結果, $x1\,x2^2\,x8^3$ は $x8^3\,x2^2\,x1$, $x1\,x2^2\,x3\,x9$ は $x9\,x3\,x2^2\,x1$ になります. それから, 各項の変数の次数で変数を置換えますが, 一方の項のみに現れる変数があれば, その変数が現れていない項の該当変数を 0 とします. ここで, 項 $x8^3\,x2^2\,x1$ は $x9$ と $x3$ が抜けているので $x9^0\,x8^3\,x3^0\,x2^2\,x1$ になり, $x9\,x3\,x2^2\,x1$ は $x8$ がないので $x9\,x8^0\,x3\,x2^2\,x1$ となります. このように項を正規化してから変数の次数を左から順に並べたリストを構築します. その結果, 5 成分の次数リスト $(0,3,0,2,1)$ と $(1,0,1,2,1)$ がえられます. 次に, 項の大きさの比較で, これらのリストの先頭から通常の大小関係 “$>$” を使って決めます. この場合, $(0,3,0,2,1)$ の先頭の値が 0 であるのに対して, $(1,0,1,2,1)$ の先頭が 1 となるので, $(1,0,1,2,1)$ の方が大となります. この次数リストの大小関係をそのまま項の大小関係とします. このことから $x1\,x2^2\,x3\,x9 >_m x1\,x2^2\,x8^3$ となります.

この方法で構築した順序 “$>_m$” は ‘$x >_m y$’ であれば, 任意の 0 と異なる項 a に対して ‘$ax >_m ay$’ が成立する事が判ります. この性質を満す順序の事を項順序と呼びますが, このことから順序 “$>_m$” が項順序である事が判ります. なお, 大域変数 simp が true の場合, Maxima は項順序 “$>_m$” に沿って式の項の並び換えを自動的に実行します：

```
(%i16) expr1:x1*x2^2*x8^3+x1*x2^2*x3*x9;
                                2               2   3
(%o16)                    x1 x2  x3 x9 + x1 x2  x8
(%i17) expr2:x1*x2^2*x3*x9+x1*x2^2*x8^3;
                                2               2   3
(%o17)                    x1 x2  x3 x9 + x1 x2  x8
(%i18) :lisp $expr1;
((MPLUS SIMP)((MTIMES SIMP) $X1 ((MEXPT SIMP) $X2 2)((MEXPT SIMP) $X8 3))
 ((MTIMES SIMP) $X1 ((MEXPT SIMP) $X2 2) $X3 $X9))
(%i18) :lisp $expr2;
((MPLUS SIMP)((MTIMES SIMP) $X1 ((MEXPT SIMP) $X2 2)((MEXPT SIMP) $X8 3))
 ((MTIMES SIMP) $X1 ((MEXPT SIMP) $X2 2) $X3 $X9))
(%i18) :lisp (equal $expr1 $expr2)
T
```

この例では同値な式の項の順番を変更して Maxima に入力していますが, 項の順序が変更されて同じ式で返されています. さらに ‘:lisp $expr1’ と ‘:lisp $expr2’ で式 expr1 と式 expr2 の内部表現を表示させていますが, これらの内部表現は同一です.

なお, 演算子 “:lisp” はあとに続く文字列を LISP に直接渡して結果を Maxima に戻す演算子です.

この内部表現で注目して頂きたいことは, 順序 “$>_m$” で小さなものがリストの左側に置かれ, 大きなものが右側に置かれることです. この内部表現に対して式の表示は項順序の大きなものから左側に並べられます. こうすることで数式の通常の書き方に準拠したものになっています.

5.2.3 局所的な順序の変更

Maxima の項順序 “$>_m$” は局所的に変更することが可能です. 項順序 “$>_m$” を変更する函数としては ordergreat 函数と orderless 函数があります：

Maxima の順序変更を行なう函数

ordergreat(⟨ 記号$_1$⟩,...,⟨ 記号$_n$⟩)
orderless(⟨ 記号$_1$⟩,...,⟨ 記号$_n$⟩)
unorder()

ordergreat 函数: 第 1 引数である記号 ⟨ 記号$_1$⟩ が最大で, 以降, 第 2 引数, 第 3 引数…と右側に行くに従って小さくなり, 第 n 番目の記号 ⟨ 記号$_n$⟩ が最小となるように新たな項順序 “$>_m$” が再構築されます.

orderless 函数:　第 1 引数の ⟨ 記号$_1$⟩ が最小で, 第 2, 第 3 と右に行くに従って大きくなり, 第 n 番目の引数 ⟨ 記号$_n$⟩ が最大になる様に Maxima の順序 "$>_m$" が再構築されます.
この ordergreat 函数や orderless 函数で Maxima に入れた順序は unorder(); を実行すれば解除されます.

unoder 函数:　ordergreat 函数や orderless 函数で再構築した順序 "$>_m$" は一度 unorder 函数で元に戻さなければ, これらの函数による順序の再構築はできません:

```
(%i13) ordergreat(c,b);
(%o13)                                done
(%i14) ordergreat(b,z);
Reordering is not allowed.
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i15) unorder();
(%o15)                               [b, c]
```

この例では最初に 'c $>_m$ b' という順序を入れています. そのあとに続けて 'b $>_m$ z' という順序を入れようとしてエラーになっています. それから unorder() を実行し, 順序 "$>_m$" を元に戻しています.

5.2.4　順序に関連する函数

Maxima の項順序は逆アルファベット順の辞書式順序と呼ばれる順序になります.
そこで今度は Maxima で変数順序や項順序がどのように入っているかを実際に調べてみましょう.
二つの与えられた項の順序を調べる函数として Maxima には ordergreatp 函数と orderlessp 函数の二つの真理函数があります:

順序を確認する真理函数
ordergreatp(⟨ 項$_1$⟩ ,⟨ 項$_2$⟩)
orderlessp(⟨ 項$_1$⟩ ,⟨ 項$_2$⟩)

これらの函数は二つの引数を取る真理函数であり, 内部では LISP の great 函数を用いています.

ordergreaterp 函数:　⟨ 項$_1$⟩ の筆頭項が ⟨ 式$_2$⟩ の筆頭項よりも大となる場合に true を返し, それ以外の場合は false を返します.

orderlessp 函数:　⟨ 式$_1$⟩ の筆頭項が ⟨ 式$_2$⟩ の筆頭項よりも小となる場合に true を返し, その他の場合に false を返します:

```
(%i33) ordergreatp(abc,a);
(%o33)                                true
(%i34) ordergreatp(abc,ax);
(%o34)                                false
(%i35) ordergreatp(x^2,y^2);
(%o35)                                false
(%i36) ordergreatp(z^2,y^2);
```

```
(%o36)                                  true
(%i37) ordergreatp(z,y^2);
(%o37)                                  true
(%i38) ordergreatp(z^3,z^2);
(%o38)                                  true
(%i39) ordergreatp(z^2*x*y^2,z^2*x*t^3);
(%o39)                                  true
```

最初の例では変数 abc と変数 a の順序を比較しています. Maxima の項順序 "$>_m$" ではアルファベットには逆アルファベット順で大きさを決め, 変数の比較では左端の文字から両者を比較して最初に大きな文字のある側を大とします. この例では,abc と a では頭は同じですが, a には他の文字がないために 'abc $>_m$ a' となります. 次の abc と ax の場合は頭の a は同じでも, b と x では x の方が大となるために 'ax $>_m$ abc' となります.

次の自由変数 x と y を含む式 'x^2' と 'y^2' の比較では変数 y の方が x よりも大のために 'y^2' の方が大になります. 同様に式 'z^2' と 'y^2' の比較では変数 z の方が y よりも大となるため, 式 'z^2' が大になりますが, 式 'z' と式 'y^2' の比較でも次数とは無関係に式 'z' の方が式 'y^2' よりも大になります. そして, 同じ変数の場合は次数の大きな方が大になるので, 'z^2 x y^2' と 'z^2 x t^3' では, その頭から比較したときに 'y $>_m$ t' となるので, 式 'z^2 x y^2' の方が式 'z^2 x t^3' よりも大になります. このことから Maxima の変数順序は斉次ではなく, 辞書式順序に基く順序であることが理解出来るかと思います.

Maxima では変数の順序を ordergreat 函数や orderless 函数で多少変更する事が可能ですが, 基本的な順序は辞書式順序のみです. S INGULAR のような数式処理では複数の順序を目的に応じて選択できますが, 残念ながら Maxima は違います. この点から Maxima は古風な数式処理システムと言えるでしょう.

5.2.5 函数を含めた順序

Maxima では通常の多項式に加えて exp や sin 等の Maxima 組込の初等函数や利用者定義の函数に対しても順序を入れられます.

Maxima の組込の函数に関しては先ず, 同じ引数の二つの函数に対しては函数名で順序付けられます. 次に同じ函数を別の式に作用させることでえられる二つの式に対しては, 函数を作用させる前の式に対して順序付けることができます. この組込の函数と多項式の項に対しては, 基本的に函数項の方が多項式項よりも大となりますが, 函数項に主変数が含まれず, 多項式項に主変数が含まれている場合には, 多項式の項の方が順序 "$>_m$" で上位になります.

利用者定義の函数の場合, Maxima 側で値を解釈するために ordergreatp 函数や orderlessp 函数の値はその状況に応じて変化します.

次に, その例を示します:

```
(%i77) neko(x):=if x<0 then x^2 else cos(x)^3;
                                                    2          3
(%o77)              neko(x) := if x < 0 then x  else cos (x)
(%i78) assume(p0>0);
(%o78)                                [p0 > 0]
(%i79) ordergreatp(cos(p0),neko(p0));
(%o79)                                  false
(%i80) assume(p1<0);
(%o80)                                [p1 < 0]
(%i81) ordergreatp(cos(p1),neko(p1));
(%o81)                                  true
(%i82) ordergreatp('neko(x),atan(x));
(%o82)                                  true
(%i83) ordergreatp(neko(x),atan(x));
Maxima was unable to evaluate the predicate:
x < 0
#0: neko(x=x)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i84)
```

この例で示すように利用者函数に単引用符を付けなければ Maxima で解釈が実行されて, その結果で ordergreatp 函数の結果が決ります. これに対して単引用符を付けて名詞型で比較すると単純に函数名で比較されます. このように初等函数や利用者定義函数が名詞型の場合は引数も含めた函数名で変数順序 “$>_m$” による比較が実行されます.

5.3 演算子

5.3.1 Maxima の演算子の属性

Maxima の演算子には数学で用いられる四則演算等を模した算術演算子, さらには変数に式を割当てる演算子といったさまざまな演算子があります. 利用者が演算子を新たに定義したり, 既存の演算子の性質を変更することができます.

Maxima の演算子は演算子名と被演算子の関係, すなわち, 演算子の属性から次の 6 種類に分類されます:

Maxima の演算子の属性

•prefix 型	前置表現の演算子.	微分演算子 “*d*/*dx*” のように一つの引数に対して引数の前に配置される.
•infix 型:	内挿表現の演算子.	等号の演算子 “=” のように二つの引数の間に配置される演算子で, 結合律を必要としない場合.
•nary 型:	内挿表現の演算子.	和の演算子 “+” のように二つの引数の間に配置される演算子で, 結合律を必要とする場合.
•matchfix 型:	外挿表現の演算子.	[1,2,3] の “[ ]” のように対象を囲む.
•postfix 型:	後置表現の演算子.	階乗の演算子 “!” の様に演算子のうしろに引数を置く.
•nofix 型:	無引数の演算子.	引数を一切取らない.

ここで Maxima の属性の詳細については §5.4 を参照して下さい.
それでは適当な文字列に演算子としての属性を与えてみましょう:

```
(%i25)  prefix ("mike");
(%o25)                                mike
(%i26)  mike neko;
(%o26)                              mike neko
(%i27)  infix  (\:/) $
(%i28)  x  :/  mike y;
(%o28)                            x  :/  mike y
```

この例では演算子としての実体がないために評価がこれ以上実行されず, 形式的な演算子項となっています. 実際の処理を行うためには演算子の実体を定義するか, 属性を適宜与えておく必要があります. ここで定義した演算子に実体を与える方法としては, あらかじめ定義した函数を演算子として宣言する方法, それと, 函数を定義する要領で直接定義する二種類の方法がありますが, 演算子の定義が先か, 函数の定義が先になるかと単純にその手順が逆になるだけで, その手続の結果, 得られる演算子は同じものです.

今度は, 形式的な演算子に実体を与えたり, 函数を演算子にしてみましょう:

```
(%i29) mike x:=2*x+1;
(%o29)                                mike x := 2 x + 1
(%i30) x :/ y := (x+sin(x))/y;
                                                 sin(x) + x
(%o30)                                x :/ y := ----------
                                                    y
(%i31) pochi(x,y):=x^y;
                                                      y
(%o31)                                pochi(x, y) := x
(%i32) nary("pochi");
(%o32)                                      pochi
(%i33) mike 3;
(%o33)                                        7
(%i34) 5 :/6;
                                          sin (5) + 5
(%o34)                                    ----------
                                              6
(%i35) 4 pochi 2;
(%o35)                                        16
```

この例で示すように演算子として宣言した函数の実体を定義する場合は Maxima の函数の定義と同様に演算子 ":=" が使えます. つまり, 演算子 ":=" の左辺に演算子項を置き, 右側に函数項で用いた変数を利用して, その処理内容を函数の定義と同様に記述するのです (§8.2 参照). この例から判るように Maxima の演算子の実体は演算子属性を持った函数です. 実際, 演算子と函数の内部表現は同じです:

```
(%i1) a:mike(x,y)$

(%i2) :lisp $a;

((SMIKE SIMP) $X $Y)
(%i2) infix ("mike")$

(%i3) b: (x mike y)$

(%i4) :lisp $b

((SMIKE SIMP) $X $Y)
```

この例より, 形式的な函数項 'mike(x,y)' と形式的な演算子項 'x mike y' の内部表現が一致することが判ります.

5.3.2　演算子の束縛力

数学で用いる算術演算子には被演算子を引き止める力があります. たとえば, 数式 '$1 + ab^2c - d$' は, この式を繋ぎ合せている演算子の力関係から, '$(1 + a(b^2)c) - d$' として解釈されます. Maxima では演算子が被演算子を引き込む力を束縛力 bp(Binding Power) と呼び, 200 までの整数値で表現します. さらに, 演算子に対する被演算子の配置から, この束縛力を左右の二つの属性, すなわち, 「**左束縛力 lbp(Left Binding Power)**」と「**右束縛力 rbp(Right Binding Power)**」に分けて指定できる

演算子もあります.

この束縛力は利用者が自由に設定することができます. 演算子の束縛力は LISP の属性で与えられ, Maxima の組込演算子の属性 lbp と rbp の値は Maxima のソースファイル nparse.lisp 内で設定されています. たとえば, 和の演算子 “+” は両方共に 100, 可換積演算子 “*” は lbp のみ 120, 冪演算子は lbp が 140 で rbp が 139, 差の演算子は lbp が 100 で rbp が 134 となっています. そして, 束縛力の値が大きなものほど, 演算子と被演算子の結び付きが強くなります. 実例を挙げて解説しておきましょう. Maxima の式 ‘b^2*c’ が入力されると, この式を構成する可換積の束縛力は 120, 冪の束縛力は 140 と冪の束縛力が大きいため, 冪演算子の直後の対象 2 は冪に引き寄せられ, その結果, 入力式は ‘(b^2)*c’ として解釈されます.

また, 演算子を利用者が定義する際に束縛力を指定しなければ, 既定値の 180 が設定されます. このことを実際に確認してみましょう:

```
(%i1) prefix ("tama")$
(%i2) :lisp (get '$tama 'lbp);
NIL
(%i2) :lisp (get '$tama 'rbp);
180
```

この例では前置表現の演算子 tama を定義し, その左右の束縛力を取出しています. ここで束縛力は LISP の属性を用いて設定するので, LISP の get 函数て調べられます. この例では演算子 tama が前置表現の演算子なので左束縛力が未設定 (NIL), 右束縛力が 180 であることが判ります[9].

今度は後置式表現の演算子 mike を定義して束縛力を調べてみましょう:

```
(%i4) postfix ("mike")$
(%i5) :lisp (get '$mike 'lbp);
180
(%i5) :lisp (get '$mike 'rbp);
NIL
```

この例から前置式演算子と束縛力の設定が逆になっていることが判ります.

演算子の束縛力は, 算術演算子以外の右小括弧 “(” や左小括弧 “)” といった記号にも設定されています. これらの小括弧の束縛力は最大で 200 であり, このことから小括弧 “( )” は外や内部の束縛力を遮断する作用を持つことになります. つまり, 小括弧 “( )” で括られた式に対しては, 外部の演算子の影響が小括弧によって遮断されるだけではなく, 小括弧内部の式に含まれる演算子の束縛力も, この小括弧の外部に及ばないことになります.

ここで二項演算子に対しては左右の束縛力の釣合加減で結合律の成立に影響が出ます:

```
(%i5) infix ("><",100,120) $
(%i6) (a >< b):=a^b;
                                  b
```

[9]本当はソースファイル npase.lisp にて演算子の宣言函数の定義を見れは判ることです.

```
(%o6)                                (a  >< b)  :=  a
(%i7)  a><b><c;
                                                 b  c
(%o7)                                          (a  )
(%i8)   infix  (”><”,120,100) $
(%i9)  a><b><c;
                                                   c
                                                  b
(%o9)                                            a
```

この例では infix 型の内挿型演算子 “><” を最初に左束縛力を 100, 右束縛力を 120 で定義しています．すると ‘a><b><c’ は右束縛力の方が強いために ‘(a><b)><c’ と解釈されます．次に左束縛力を 120, 右束縛力を 100 と逆にすると, 演算子と左側の被演算子との繋がりが強くなるために ‘a><b><c’ は’a><(b><c)’ として処理されます.

束縛力は内挿式, 前置式演算子の実体の定義を行うときでも重要です. 何故なら, 定義で用いる演算子 “:=” にも束縛力があるからです. 演算子 “:=” の束縛力は左が 180, 右が 20 となっています. そのために演算子の束縛力が弱い, すなわち, 演算子の右束縛力が 180 よりも小さい場合, 演算子 “:=” の側に被演算子が引き寄せられます. そのために演算子 “:=” の左側の演算子項全体を小括弧で括らなければ定義そのものが無意味になってしまいます.

このように Maxima の式を記述するときに少しでも式中の演算子の束縛力に疑問があれば, 演算子の束縛力の作用範囲を明確にするために小括弧 “( )” の併用を強く薦めます.

5.3.3　演算子と被演算子の型

積 “*” は数式に対して利用可能で, 結果も数式になりますが, 文字列そのものに作用させても意味はありません. このように演算子には必要とする被演算子や演算の結果がどのようなものであるかが予め決っています. Maxima では演算子が取り得る被演算子と演算子が返す結果には型が指定されています. この型は, Maxima のソースファイルでは二項演算子の左被演算子なら def-lprop, 右被演算子なら def-rprop, 一般の演算子の被演算子に対しては def-prop マクロを用いて指定され, その結果は def-pos マクロで指定されます. また, 新規に演算子を宣言する場合は, その演算子の宣言を行う函数で被演算子と結果の型の指定が行われる仕様となっています.

nary 函数と matchfix 函数の場合:　これらの函数で指定される演算子では, その被演算子が Maxima 内部にて列として表現されるため, 左右に分けて被演算子の型を指定する必要はなく, 被演算子を一元的に指定することになります:

被演算子と返却値の型が設定される属性

- 被演算子の型が設定される属性:　argpos
- 返却値の型が設定される属性:　pos

nary 函数と matchfix 函数以外の函数の場合: この場合, 被演算子がある場合は, 左右のいずれか, あるいは双方に配置されることとなります. ここで, 二項演算子の場合は nary 型とは異なって, 被演算子が均質的である必要がないために演算子の左右の被演算子を独立して設定することになります:

二項演算子の左右の被演算子と返却値の型が指定される属性

• 左被演算子の型が設定される属性:	lpos (left part of speech)
• 右被演算子の型が設定される属性:	rpos (right part of speech)
• 返却値の型が設定される属性:	pos

このように, Maxima では必要とされる被演算子の型は argpos, lpos, rpos といった属性の属性値として, 結果の型は属性 pos の属性値として与えることになります.

設定される属性: ここで Maxima の函数や被演算子が扱うべき対象が何かを考えてみましょう. Maxima は何よりも数式処理システムなので, 数式を処理しなければなりません. また, その数式が正しいものであるかどうかといった数式の意味を考慮しなければなりません. それから, 「計算終了」といった文字列といった数式や真理値以外の対象もあります.Maxima では数式の型を expr で, 式の意味を返す論理式の型を clause で指示し, この分類に収まらない場合, any で式の型を指示します:

演算子の型

型	属性値	概要
expr	algebraic	数式全般, Maxima の数式に付与される型です.
clause	logical	論理式全般, Maxima の述語や論理式に付与される型です.
any	untyped	任意の Maxima の式, expr 型や clause 型を含む Maxima の任意の式に付与される型です.

これらの expr 型, clause 型, any 型の属性として, english があり, その属性値はそれぞれ algebraic(代数的), logical(論理的), untyped(未分類) となっています:

```
(%i1) : lisp (get '$expr 'english)
 algebraic
(%i1) : lisp (get '$clause 'english)
 logical
(%i1) : lisp (get '$any 'english)
untyped
```

なお, 演算子宣言で演算子の型を指定しなければ, 左右の被演算子の型と返却値の型に$any が自動的に設定されます. これらの属性は LISP の属性を用いて実現されているので LISP の put 函数 で無理矢理に設定できます:

```
(%i4) prefix ("mike")$
(%i5) : lisp (get '$mike 'pos)
$ANY
```

```
(%i5) :lisp (put '$mike '$clause 'pos)
$CLAUSE
(%i5) :lisp (get '$mike 'pos)
$CLAUSE
(%i5) mike a := freeof(a,x)$
(%i6) if mike(x^2+1) then print("test1")$
test1
(%i7) :lisp (put '$mike '$expr 'pos)
$EXPR
(%i7) if mike(x^2+1) then print("test1");
Incorrect syntax: Found algebraic expression where logical
expression expected if mike(x^2+1) then
```

最初に定義した前置式演算子 mike の属性は any ですが, LISP の put 函数により返却値の型を指定する属性 pos の値を clause に変更します. それから, mike 本体を定義し, if 文の条件文で演算子 mike を使います. このとき, if が条件文として要求する型はソースファイルの def-rpos マクロの記載から claus 型であることが判り, 演算子 mike の返却する型とは矛盾しないので問題は生じません. そこで, 今度は LISP の put 函数を使って演算子 mike の返却値の属性を expr 型に変更します. すると, 二度目の if 文では演算子 mike の返却型は expr と if が必要とする clause 型と矛盾するためにエラーとなるのです.

5.3.4 演算子の属性を宣言する函数

Maxima では利用者が記号や文字列を演算子として利用できます. また, 函数にも演算子としての属性を付与することで演算子として利用できます. その上, 適当な対象に演算子としての属性を付加してしまえば, それだけで演算子として利用ができます. もし, 演算子が演算子としての実体を持っていなくても, 属性を利用することで特定の値に対する返却値を定めることが可能です. 実際に, この返却値の指定を行っている函数が atvalue 函数で, この函数によって, ある値に対する返却値や勾配を属性として付与することができています.

ここで演算子を宣言する函数には, infix 函数, nary 函数, prefix 函数, postfix 函数, matchdeclare 函数と nofix 函数があります. 以降, これらの函数の解説を行いますが, これらの函数を解説する前に幾つかの表記を確認しておきます. まず, lbp と rbp が左束縛力と右束縛力, lpos と rpos が左右の被演算子の型, pos が返却値の型を示します. ただし, matchfix 函数と infix 函数に関しては束縛力を左右に分けずに束縛力を bp, 被演算子の型を argpos とします.

infix 函数: infix 型の内挿表現の演算子の宣言を行う函数です:

infix 函数の構文

infix(a)	対象 a を infix 型演算子として宣言
infix(a,lbp,rbp)	対象 a を infix 型の演算子として左右の束縛力を含めて宣言
infix(a,lbp,rbp,lpos,rpos,pos)	対象 a を infix 型の演算子として左右の束縛力と被演算子の型を含めて宣言

infix 型の演算子は数式 '$a = b$' の中の演算子 "=" のように被演算子が演算子の左右に配置される演算子であり, 左右の束縛力を変更しておく必要のある演算子であるか, lhs 函数や rhs 函数を用いて被演算子を取出す必要のある式を構築しなければならない場合に用いる演算子の型です. この型の演算子は後述の nary 型のように結合律を暗黙の内に認める演算子ではありません. そのため, infix 型はより一般的な二項演算子の宣言に適しています.

nary 函数 nary 型の内挿表現演算子の宣言で用いる函数です:

nary 函数の構文

nary(a)	対象 a を nary 型演算子として宣言
nary(a,bp)	対象 a を nary 型演算子として束縛力を含めて宣言
nary(a,bp,argpos,pos)	対象 a を nary 型演算子として束縛力と被演算子の型を含めて宣言

nary 函数で宣言可能な演算子は左右の束縛力が等しく, rhs 函数や lhs 函数を用いて左右の被演算子を取り出す必要がないものに限定されます. つまり, 演算子 "△" が nary 型の場合, '(a △ b) △ c' と 'a △ (b △ c)' は内部表現は共に '(($△ SIMP) a b c)' と演算子 "△" の被演算子の列が内部の演算子名のリストのうしろに続くという表現となります. これが infix 型の演算子であれば, これらの式の構造を保ったままの内部表現となります. 実際に, この infix 型と nary 型の内部表現の違いを実際に確認しておきましょう:

infix 型と nary 型の演算子の内部表現の違い: ここでは infix 型演算子 "<>" と nary 型演算子 "><" を定義して, 形式的な演算子項の確認を行います:

```
(%i1) infix ("<>")$
(%i2) nary("><")$
(%i3) eq1:1<>2<>3<>4$
(%i4) eq2:1><2><3><4$
(%i5) : lisp $eq1
(($<> SIMP) (($<> SIMP) (($<> SIMP) 1 2) 3) 4)
(%i5) : lisp $eq2
(($>< SIMP) 1 2 3 4)
```

infix 型演算子 "<>" を使った式の内部表現が二項演算子の痕跡を残しているのに対し, nary 型演算子 "><" を使った式の内部表現が平な S 式になって二項演算子の痕跡がないことに注意して下さい. この内部表現の違いにより, infix 型の演算子に対してのみ lhs 函数と rhs 函数が機能するのです.

この性質の違いは, infix 演算子は結合律の成立を前提とした演算でもなく, 左右の演算子が異った対象であっても構わない演算子であるのに対し, nary 演算子は結合律の成立を前提とした演算子だからです. 実は, nary の名前は「**n-ary 型**」, つまり, 複数の引数を持つ函数として表現できる演算子の名前なのです.

これらの演算子の違いに関連した例をもうひとつ挙げておきましょう. 演算子 “<” は infix 型の演算子で論理式を構成しますが, Maxima の式 ‘a1<a2<a3<a4’ は infix 型の演算子にとっては正しくない構文です. ただし, ‘((a1<a2)<a3)<a4’ 括弧を使って纏めると infix 型の演算子項としては正しい式となります. しかし, このような括弧を使った入れ子の構造は, is 函数や ev 函数といった評価を行う函数では, その式の評価を括弧で構成された構造で最下層から評価をさせなければならないために上手く評価ができません. is 函数や ev 函数で評価を行うことを考慮すると, 正しい式の構文は ‘al<a2 and a2<a3 and a3<a4’ のように論理演算子 “and” を用いて結合しなければなりません. ここで, 論理演算子 “and” は nary 演算子としての性質を持っており, 演算子 and の函数名の S 式に続いて, and で繋がれた論理式が平なリストの成分として並びます. そのために is 函数や ev 函数が有効に作用するのです:

```
(%i10)  a2 :(( x1<x2)<x3)<x4;
(%o10)                                  x1  < x2  < x3  < x4
(%i11)  : lisp  $a2

((MLESSP . #1=(SIMP))  ((MLESSP . #1#)  ((MLESSP . #1#)  $X1 $X2)  $X3)  $X4)
(%i11)  a3 :( x1<x2)  and  (x2<x3)and  (x3<x4);
(%o11)                        (x1  < x2)  and  (x2  < x3)  and  (x3  < x4)
(%i12)  : lisp  $a3

((MAND SIMP) ((MLESSP . #1=(SIMP)) $X1 $X2) ((MLESSP . #1#)  $X2 $X3)
  ((MLESSP . #1#)  $X3 $X4))
```

このように infix 演算子は, 複数の infix 演算子が存在する項では小括弧を外すことができません. しかし, nary 演算子は結合律が前提にあるために括弧を省略することが可能なのです. そして, nary 演算子の内部表現が平な S 式となるために, 評価函数は各成分に map させるだけで良く, infix 演算子項のように全ての階層で評価を行う必要がないのです.

nofix 函数:　無引数演算子の宣言を行う函数です:

nofix 函数の構文

nofix(a)	対象 a を無引数の演算子として宣言
nofix(a, pos)	対象 a を無引数の演算子として結果の型を含めて宣言

nofix 函数はは演算子が引数を持たないことを明示するために用いる函数ですが, 結果の型も指定できます.

prefix 函数:　前置表記演算子の宣言を行う函数です:

prefix 函数の構文

prefix(a)	対象 a を前置表現の演算子として宣言
prefix(a,rbp)	対象 a を前置表現の演算子として右束縛力を含めて宣言
prefix(a,rbp,rpos,pos)	対象 a を前置表現の演算子として右束縛力と被演算子と結果の型を含めて宣言

前置表記の演算子は一個の被演算子のみを持ち, その被演算子は演算子の直後に置かれます. prefix 函数では右束縛力, 被演算子と結果の型を含めて宣言できますが, 無指定が無指定であれば any が指定されます.

postfix 函数: 後置表記演算子の宣言を行う函数です:

prefix 函数の構文

postfix(a)	対象 a を後置表現の演算子として宣言
postfix(a,lbp)	対象 a を後置表現の演算子として左束縛力を含めて宣言
postfix(a,lbp,rpos,pos)	対象 a を後置表現の演算子として左束縛力と被演算子の型を含めて宣言

後置表記の演算子は一つの被演算子のみを持ち, 被演算子は演算子の前に置かれます. この postfix 函数では左束縛力の大きさ, 被演算子と結果の型が指定できます.

matchfix 函数

任意個数の被演算子を二つの対象で囲む演算子を宣言する函数です:

matchfix 函数の構文

matchfix(a,b)	被演算子を対象 a と対象 b で挟む外挿式の演算子を宣言
matchfix(a,b,argpos,pos)	引数の型と結果の型を含めて宣言

matchfix 函数では, 演算子の属性に加え, 被演算子と結果の型の宣言が行えます. この matchfix 型の演算子の演算子名は matchfix 函数の第 1 引数の記号, すなわち, 演算子を構成する左側の記号で代表されます:

```
%i5) matchfix("@-","-@")$
(%i6) @- a,b,c,d,e,f -@:=a*b*c+d*e^f;
                                                        f
(%o6)                  @-a, b, c, d, e, f-@ := a b c + d e
(%i7) @- 1,2,3,4,5,6 -@;
(%o7)                                 62506
(%i8) dispfun("@-");
                                                        f
(%t8)                  @-a, b, c, d, e, f-@ := a b c + d e

(%o8)                                 done
```

この例では matchfix 演算子の定義を行い, dispfun 函数で演算子の実体を表示させていますが, このときに指定する演算子名は演算子 "@- -@" の左側の記号 "@-" で代表できるのです.

演算子宣言函数で定められる属性

演算子を宣言する函数で付与される属性を次に纏めておきます. なお, ここでの左右の束縛力の値は宣言する時点で自動的に割当てられる数値です:

演算子宣言函数で定められる属性

演算子	左束縛力	右束縛力	左変数型	右変数型	返却値型
postfix	180		any		any
prefix		180		any	any
infix	180	180		any	any
nofix					any
nary			any	any	any
matchfix			any	any	any

これらの函数によって変数型と返却型は any に設定されます. matchfix 函数による被演算子の型は他の函数と異なり, argpos 属性で指定しますが, その属性値は他と同じ型の any です.

5.3.5　演算子属性の削除

演算子属性は kill 函数や remove 函数で削除できます. remove 函数は演算子の属性のみを削除し, kill 函数は演算子を全体を削除します:

```
(%i10) nary("tama")$
(%i11) a tama b:=a+b^2$
(%i12) properties ("tama");
(%o12)                          [function , operator , noun]
(%i13) remove("tama",op);
(%o13)                                  done
(%i14) properties ("tama");
(%o14)                                   []
(%i15) prefix ("mike")$
(%i16) mike x:=x!+1$
(%i17) kill ("mike");
(%o17)                                  done
(%i18) properties ("mike");
(%o18)                                   []
```

この例では内挿式演算子 tama を定義し, remove 函数を使って tama の演算子属性を第 2 引数に op を指定して削除しています. 実際に properties 函数で調べても tama の属性がありません. また, kill 函数は演算子名を指定するだけで全てを削除しています. remove 函数は properties 函数で調べた特定の属性だけを削除する函数です:

```
(%i19) nary("tama")$
(%i20) a tama b:=a+b^2$
(%i21) remove("tama",function)$
(%i22) properties ("tama");
(%o22)                                [operator , noun]
(%i23) 3 tama 4;
```

(%o23) 3 tama 4

この例では第 2 引数を function にしたため, 演算子に割当てられた函数が削除されただけで, 演算子としての属性は残っています. そのために 3 tama 4; と入力してもエラーにはなりませんが, 実体が削除されているので形式的な演算子になります.

5.3.6 算術演算子

二項算術演算子

二項算術演算子として次の演算子が定義されています:

二項数式演算子

演算子	属性	例	概要
+	prefix	a + b	a と b の和
-	prefix	a - b	a と b の差
*	nary	a * b	a と b の可換積
/	infix	a / b	a の b による商
**		a ** b	冪 a^b
^		a ^ b	冪 a^b,a**b と同じ
.	infix	a . b	a と b の非可換積
^^		a ^^ b	a と b の非可換積の冪乗

これらの演算子の持つ属性を以下に示しておきます:

二項算術演算子の持つ属性

演算子	左束縛力	右束縛力	左変数型	右変数型	返却値の型
+	100	100	不要	expr	expr
-	100	134	不要	expr	expr
*	120	不要	expr	不要	expr
/	120	120	expr	expr	expr
^	140	139	expr	expr	expr
**	140	139	expr	expr	expr
.	130	129	expr	expr	expr
^^	140	139	expr	expr	expr

これらの演算子項は通常の数式と殆ど同じ表記ですが, 演算子 "-", 演算子 "/" は Maxima 内部では表示された式とは別表現です (§6.4 参照).

たとえば Maxima の式 'x - y' は, Maxima 内部で 'x + (-1) * y', 'x / y' は 'x * y^-1' に対応する内部表現となっています. この内部表現を強制的に本来の形に変換する函数 dispform もあります. さらに, -に関しては大域変数 negsumdispflag を false に変更して内部表現通りに表示できます.

非可換積の冪 “^^” と通常の可換積の冪 “^” は別物です. たとえば, ‘a^^3’ は ‘a . a . a’ と同値であり, ‘a^3’ は ‘a * a * a’ を意味します. ここで可換積 “*” と非可換積 “.”, および, それらに対応する冪 “^” と “^^” が Maxima の式中に混在しても構いません. また, 演算子 “^” を含む項の表示では右側の被演算子が通常の冪, たとえば, x^3 のように上付きで表示されるのに対し, 演算子 “^^” は $x^{\langle n \rangle}$ のように ⟨⟩ で括られて上付きで表示されます:

```
(%i1) a^^b;
                                    <b>
(%o1)                              a
```

これら非可換積 “.” と非可換冪 “^^” は行列で用いられます. このとき, 非可換積 “.” が通常の行列の積を意味し, 可換積 “*” はスカラー積や同じ大きさの行列に対する成分毎の積を取る演算子になります:

```
(%i54) A:matrix    ([1,2],[3,4]);
                                  [ 1   2 ]
(%o54)                            [       ]
                                  [ 3   4 ]
(%i55) B:matrix    ([2,1],[4,3]);
                                  [ 2   1 ]
(%o55)                            [       ]
                                  [ 4   3 ]
(%i56) A*B;
                                  [ 2    2   ]
(%o56)                            [          ]
                                  [ 12   12  ]
(%i57) A.B;
                                  [ 10   7   ]
(%o57)                            [          ]
                                  [ 22   15  ]
```

可換積の冪 “^” と非可換積の冪 “^^” の計算結果が長過ぎて式が表示し切れない場合は記号 “expt” が可換積の冪, 記号 “ncexpt” が非可換積の冪の表記で利用されます.

非可換積を制御する大域変数 dot 一族

非可換積 “.” と非可換冪 “^^” には挙動を制御する先頭が “dot” で開始する大域変数が幾つか存在します. これらの大域変数を, ここでは安易に「**大域変数 dot 一族**」と呼びます. ここでは, これらの大域変数について解説しましょう:

非可換積演算子に関連する大域変数 dot 一族

変数名	既定値	true の場合の作用の概要
dot0nscsimp	true	0 と scalar 属性を持たない対象の非可換積を可換積に変換.
dot0simp	true	0 と scalar 属性を持つ対象の非可換積を可換積に変換.
dot1simp	true	1 との他の項の非可換積を可換積に変換.
dotassoc	true	非可換積項に結合律を適用.
dotconstrules	true	定数と項の非可換積を可換積で置換. この大域変数は dotocimo,dotonscsimp,dot1simp に影響.
dotdistrib	false	非可換積項に分配律を適用.
dotexptsimp	true	同じ式による非可換積を非可換積の冪乗に変換.
dotident	1	非可換冪の 0 乗で返される値を設定.
dotscrules	false	群環の元と scalar 属性を持つ対象の非可換積を可換積に変換.

ここでスカラーと群環の元との非可換積に関連する大域変数は, scalar 属性を付与した対象には制御が効きますが, 通常の数値の場合は大域変数とは無関係に可換積に置換されます:

```
(%i6) 2 . 3 . x . y;
(%o6)                                  6 (x . y)
```

なお, 非可換積を Maxima で記述するときは ‘a . b’ のように空白文字を入れておくべきです. これは引数が数値の場合は必須です. 何故なら, ‘1.2’ は数値 ‘1’ と数値 ‘2’ の非可換積でなく, 浮動点小数 ‘1.2’ を意味し, ‘1. 2’ や ‘1 .2’ のように中途半端に空行が入ったも浮動小数点数と整数を空白文字で繋いだ式と解釈されて無意味な Maxima の式になってしまうからです.

大域変数 dotassoc を false にすると, 複数の非可換積で勝手に括弧を外されないために非可換積に対する規則を定義した場合, その適用が容易になります.

後置式の算術演算子

後置式の算術演算子

!	n!	n が整数の場合,n の階乗を計算. 一般の場合 $\Gamma(x+1)$
!!	n!!	ln が奇数 (偶数) ならば,n 以下の奇数 (偶数) の積

後置式演算子は引数を一つ取ります. Maxima では “!” と “!!” が後置式の演算子となります. これらの演算子の数学的側面の詳細に関しては, §9.1 の階乗や Γ 函数の項目を参照して下さい.

ここで後置式演算子 “!” と “!!” で設定される束縛力や型を示しておきます:

後置式表現演算子の属性

演算子	左束縛力	右束縛力	左変数型	右変数型	返却値型
!	160		expr		expr
!!	160				

5.3.7　論理演算子

Maxima の論理演算子は Maxima の論理式を構成する演算子です (§5.5 参照). 否定 “not”, 論理和 “or”, 論理積 “and” は論理式を評価する演算子ですが, その他の演算子は評価を伴わない二項間関係を表現する演算子です:

論理演算子

演算子	属性	例	概要
not	prefix	not a	述語 a を否定
and	nary	a and b	述語 a と b の論理積
or	nary	a or b	述語 a と b の論理和
=	infix	a = b	a と b が等しい
#	infix	a # b	a と b が等しくない
>=	infix	a >= b	a は b 以上
>	infix	a > b	a は b よりも大
<=	infix	a <= b	a は b 以下
<	infix	a < b	a は b より小

C の論理積 “&&” や論理和 “——” に相当する演算子が演算子 “and” と演算子 “or” になります. これらの演算子 “and” と演算子 “or” は論理式を強引に評価します. この評価では文字通りの字面による解釈が実行されるために false が返される場合には注意が必要です.

二項間の関係を示す演算子は, Maxima 独特の演算子 “#” と “=” を除くと, C や FORTRAN で利用されるものとの違いはありません. ここで注意が必要なのは演算子 “=” です. Maxima では変数への値の割当に演算子 “:” を用い, 方程式で左辺と右辺が等しいことを示す演算子として演算子 “=” を用います. そのため, 演算子 “=” が C の “==” に対応します. この割当の演算子と等号の演算子は何気に混同し易いので注意して下さい.

次に論理演算子の属性値の一覧を示します:

論理演算子の属性値

演算子	左束縛力	右束縛力	左変数型	右変数型	返却値の型
not		70	clause	clause	clause
and	65		clause		clause
or	60		clause		clause
=	80	80	expr	expr	clause
#	80	80	expr	expr	clause
>=	80	80	expr	expr	clause
>	80	80	expr	expr	clause
<=	80	80	expr	expr	clause
<	80	80	expr	expr	clause

5.3.8 割当の演算子

割当の二項演算子

演算子	属性	例	概要
:	infix	a : b	a に b の値を割当てる
::	infix	a :: b	a に b の値を割当てる
::=	infix	a ::= b	本体が b のマクロ a を定義
:=	infix	a:=b	本体が b の函数 a を定義

大域変数 setcheck に割当てた変数リストに含まれる変数に演算子 ":" と "::" を用いて割当を行う場合, 変数名と割当てられた値の表示が行われます:

```
(%i11)  setcheck : false ;
(%o11)                                  false
(%i12)  a1 :20;
(%o12)                                   20
(%i13)  setcheck :[' a1 ,' b1 ];
(%o13)                                [a1, b1]
(%i14)  a1 :10;
a1  SET TO 10
(%o14)                                   10
(%i15)  a1 :20;
a1  SET TO 20
(%o15)                                   20
```

割当の二項演算子の属性

演算子	左束縛力	右束縛力	左変数型	右変数型	返却値の型
:	180	20	any	any	any
::	180	20	any	any	any
::=	180	20	any	any	any
:=	180	20	any	any	any

もし, 演算子の実体を割当の函数で定義する際に演算子の右束縛力が 180 よりも小であれば, 演算子項の右側のメタ変数が割当の演算子に吸い取られます. この場合, 割当の演算子の左側に配置した演算子項全体を小括弧で括る必要があります:

```
(%i7)  infix ("tama ",111,111) $
(%i8)  x  tama y:=  x+y*2;
Improper function  definition :
y
 -- an  error .   Quitting .   To debug this  try  debugmode(true);
(%i9)  (x  tama y):=  x+y*2$
(%i10)  2  tama z;
(%o10)                                  2 z + 2
```

この例では, 左右の束縛力を 111 に設定したために最初の演算子の実体の定義で演算子項の右メタ変数 y が演算子 ":=" に引き寄せられてエラーになっています. そこで, 左側の演算子項 x tama y 全体を小括弧で括ることで割当の演算子 ":=" の束縛力を遮断して, 演算子 "tama" の実体を定義しています.

5.3.9　その他の演算子

式を構成する多くの構成要素が実は演算子として扱われています. その結果, 演算子としての束縛力がプログラムの構築でも影響します.
まず, 括弧や引用符, そしてコンマの束縛力を次に示します:

括弧等の演算子としての束縛力

演算子	左束縛力	左変数型	返却値型
]	5		
[	200	any	any
)	5		
(	200		
'	190		
"	190		
,	10	any	any

ここで括弧に関しては多少解説が必要でしょう. まず, 文字 "[" と文字 "(" は matchfix 型の演算子名として利用され, 共に束縛力が 200 なので, "[" に対応する "]" と "(" に対応する ")" は外部からの演算子の束縛力を完全に遮断することになります.
if 文を構成する対象も同様に束縛力を持ちます:

if 文に関連する演算子の束縛力

演算子	左束縛力	右束縛力	右変数型	返却値型
if		45	clause	any
then	5	25		
else	5	25		
elseif	5	45	clause	any

最後に do 文を構成する対象も同様に束縛力を持ちます:

do 文に関連する演算子の束縛力

演算子	左束縛力	右束縛力	右変数型	返却値型
for	25	200	any	any
from	25	95	any	any
step	25	95	expr	any
next	25	45	any	any
thru	25	95	expr	any
unless	25	45	clause	any
while	25	45	clause	any
do	25	25	any	any

5.3.10 演算子に関連する函数

lhs 函数と rhs 函数

infix 属性を持つ内挿演算子から被演算子を取り出す函数に lhs 函数と rhs 函数があります. lhs 函数が infix 属性の演算子を挟んで左側の被演算子を取り出し, rhs 函数が右側の被演算子を取り出します:

lhs と rhs

lhs(⟨`infix` 演算子を最上層に含む式⟩)
rhs(⟨`infix` 演算子を最上層に含む式⟩)

これらの函数は infix 属性を持つ演算子のみに有効です.

```
(%i55) nary("<>")$
(%i56) A:((x+1)^2) <> ((z+a)^3)$
(%i57) lhs(A);rhs(A);
                                  2                3
(%o57)                        ((x + 1)  ) <> ((z + a)  )
(%o58)                                  0
(%i59) infix("<>")$
(%i61) lhs(A);rhs(A);
                                          2
(%o61)                                (x + 1)
                                          3
(%o62)                                (z + a)
```

この例では最初に nary 函数で演算子 “<>” を宣言していますが, この演算子に対しては lhs 函数がちゃんと動作していません. 次に infix 函数を用いて同名の演算子を宣言します. すると, 今度は lhs 函数と rhs 函数の両方がちゃんと作用しています. この理由は両者の内部表現の違いによるものです.
まず, nary 型の演算子項の内部表現を次に示しておきます:

```
(%i11) nary("<>")$

(%i12) x1 :( x<>y<>z)$

(%i13) : lisp  $x1

((  $<> SIMP) $X $Y $Z)
(%i13)
```

この nary 型の演算子は結合律の成立を前提とした演算子のために演算子項の内部表現は第 1 成分が演算子, その他が被演算子のリストとして表現します. その結果, 演算子項には右左の被演算子という情報が欠落します.
では, infix 型の演算子項の内部表現はどうなるでしょうか？

```
(%i13)  infix ("<>")$

x1 :( x<>y<>z)$

(%i15) : lisp  $x1

((  $<> SIMP)  ((  $<> SIMP) $X $Y)  $Z)
```

この infix 型の演算子は結合律の成立を前提としないために, その演算子項の内部表現では二項演算子としての表現を保ちます. だから, lhs 函数や rhs 函数によって左右の被演算子の取り出しが可能なのです.

op 函数

op 函数は一つの演算子や函数を含む式から演算子や函数を取り出す函数です:

op 函数の構文

op(⟨ 式 ⟩)

なお, この函数の実体は part 函数 (§6.4.10 参照) で, 第 2 引数として 0 を指定したものす. この函数を λ 式として '`x.op(x)` ≡ λ`x.part(x,0)`' の関係があります:

```
(%i18) op(a+b+c+d);
(%o18)                                  +
(%i19) op(a*(b+c+d));
(%o19)                                  *
(%i20) op(sin(a*(b+c+d)));
(%o20)                                 sin
(%i21) part(sin(a*(b+c+d)),0);
(%o21)                                 sin
(%i22) part(a*(b+c+d),0);
(%o22)                                  *
(%i23) part(a+b+c+d,0);
(%o23)                                  +
```

このように op 函数の結果と part 函数の第 2 引数に 0 を指定した函数による結果が一致します.

5.4 属性

5.4.1 Maxima の属性

「**属性**」は述語と呼ばれる変項が一つだけの論理函数で表現されます. この属性によって集合 (類/クラス, あるいは外延) が一つ定められます. Maxima では, 函数や演算子が線形性を持つといった対象が保有する性質を, その対象の属性として表現します. そして, 函数や演算子の線形性のような式の変形を伴う属性に対しては, 「**属性の表現函数**」と呼ばれる Maxima の内部函数を用いて, その属性に適合する処理が実行されます. Maxima の対象に付与できる属性は, その属性値が一つのみのものに限定されますが, 互いに矛盾することのない複数の属性を付与することが可能です. また, Maxima の対象に付与された属性は文脈からも参照され, 属性のその性格上, 特定の文脈に限定されすに大域的に影響を及ぼします. この文脈と属性の関連については §5.6 にて解説します.

5.4.2 属性と設定函数

Maxima の属性には, 利用者が自由に設定できる属性, Maxima であらかじめ準備された数学的な属性, 形式的な函数の値や勾配の設定等の評価に関連する属性があり, これらの属性は, 目的に応じた設定函数を用いて対象に付与されます:

属性と属性の設定を行う函数

	函数	概要
1.	put, qput	一般的な属性の設定
2.	declare	属性を付与することで副作用を伴う属性の設定
3.	atvalue, gradef, deftaylor, depends	函数の値, 勾配, Taylor 展開, 依存関係等を属性で設定

属性が付与された対象は, 大域変数 props に割当てられたリストに登録されます, そして, 対象にどのような属性が付与されているかは, properties 函数を使って表示することができます.

5.4.3 put 函数と qput 函数による属性指定

put 函数と qput 函数は利用者が自由に属性を指定することが可能です. 具体的には, Maxima の対象に属性名とその値を引数として与えることになります. これらの函数を用いて属性を与えた対象は大域変数 props に登録され, その際に, '[user properties]' に分類されています. この属性値は get 函数によって取得され, rem 函数によって対象から属性の除去ができます. また, ここで解説する put 函数, qput 函数, get 函数と rem 函数の実体は内部函数 prop1 です. なお, Maxima であらかじめ用意された数学的な属性の指定は後述の declare 函数を用います.

一般的な属性に関連する函数の構文

put(⟨ 対象 ⟩, ⟨ 属性値 ⟩, ⟨ 属性 ⟩)
qput(⟨ 対象 ⟩, ⟨ 属性値 ⟩, ⟨ 属性 ⟩)
get(⟨ 対象 ⟩, ⟨ 属性 ⟩)
rem(⟨ 対象 ⟩, ⟨ 属性 ⟩)

put 函数と qput 函数:　対象の属性に属性値を付加する函数です. qput 函数はその内部で put 函数を用いてますが, 引数の評価を行わない点で put 函数と異なります (qput=quote put).

get 函数:　put 函数や qput 函数で対象に与えた属性の値 (=属性値) を, 対象と属性を指定することで取得する函数です.

rem 函数:　put 函数や qput 函数で対象に与えた属性を削除する函数で, 対象と属性を引数として取り, その属性と属性値を対象から削除します. ただし, 大域変数 props に割当てられたリストから対象名の削除は行いません.

これらの函数の実行例を示しておきましょう:

```
(%i1) props;
(%o1) [nset, {, }, trylevel, maxmin, nummod, conjugate, erf_generalized, beta,
                                        desolve, eliminate, adjoint, invert]
(%i2)  put("ミケ","3歳","年齢")$
(%i3) get("ミケ","年齢");
(%o3)                                  3歳
(%i4) props;
(%o4) [nset, {, }, trylevel, maxmin, nummod, conjugate, erf_generalized, beta,
                                      desolve, eliminate, adjoint, invert, ミ
                                                                        ケ]
(%i5) rem("ミケ","年齢");
(%o5)                                 done
(%i6) props;
(%o6) [nset, {, }, trylevel, maxmin, nummod, conjugate, erf_generalized, beta,
                                      desolve, eliminate, adjoint, invert, ミ
                                                                        ケ]
(%i7)  get("ミケ","年齢");
(%o7)                                 false
```

この例では put 函数を用いて文字列"ミケ"の属性として '年齢' と, その属性値として '3 歳' を付与します. それから get 函数を使って文字列"ミケ"の '年齢' 属性より先程, 設定した属性値の取得を行っています. つぎに, 大域変数 props には属性を持った Maxima の対象のリストが割当てられており, その中に対象"ミケ"も含まれています. それから rem 函数で"ミケ"から '年齢' の属性値を消去します. その直後に大域変数 props を参照しても, 対象"ミケ"から get 函数で '年齢' の属性値は取り出せません. なお, rem 函数で削除可能な属性は, 函数 put や qput で設定した属性に限定され, declare 函数で付与した属性の削除は rem 函数ではなく remove 函数を用います.

put 函数や qput 函数で与えられる属性は, 利用者の必要に応じて与えられるもので, Maxima にあらかじめ準備された数学的な性質を表現する属性の指定は declare 函数を用います.

5.4.4 declare 函数 について

declare 函数は Maxima の記号と文字列に対し, あらかじめ定義された属性 (features) を付加します. declare 函数によって与えられる属性は Maxima の処理では非常に重要で, §5.6 の文脈で Maxima の式を分析, 処理する重要な役割を担います:

declare 函数の構文

declare(⟨ 対象$_1$⟩, ⟨ 属性$_1$⟩,⟨ 対象$_2$⟩, ⟨ 属性$_2$⟩, ...)
declare(⟨ 対象 ⟩, [⟨ 属性$_1$⟩, ..., ⟨ 属性$_n$⟩)
declare([⟨ 対象$_1$⟩, ..., ⟨ 対象$_m$⟩], [⟨ 属性$_1$⟩, ..., ⟨ 属性$_n$⟩])

引数として対象の列を取る場合, 属性は同じ長さの列となって 対象$_i$ に 属性$_i$ が付加されます. また, 対象に付与する属性をリストで与えることも可能で, この場合は属性リストに対応する対象に, その属性リストに含まれる全ての属性が付加されます. 同様に, 今度は対象をリストで与えると対象のリストに含まれる全ての対象に属性が指定されます:

```
(%i10) declare(a1,[integer,odd])$
(%i11) declare([b1,c1,d1],[integer,odd])$
(%i12) featurep(d1,odd);
(%o12)                                  true
(%i13) featurep(c1,integer);
(%o13)                                  true
```

この例では最初に declare 函数を使って, 記号 a1 が整数で, しかも, 奇数であることを属性として付与しています. 次の例では記号 b1, c1 と d1 が整数で, しかも, 奇数であることを一纏めに宣言しています. このように複数の属性を設定する場合は属性をリストで与え, 複数の対象に同じ属性を与える場合も対象をリストで与えられます. なお, この例では最期に featurep 函数を用いて属性の確認を行っています.

ここで複数の属性を一つの対象に付与する場合, 付与する属性が無矛盾でなければなりません. たとえば, 対象に「**偶数, かつ, 奇数である**」といった矛盾する属性を付与することはできませんが, 「**整数, かつ, 偶数である**」のように属性が矛盾しなければ問題ありません:

```
(%i11) declare(n1,odd)$
(%i12) declare(n1,even)$
Inconsistent Declaration: declare(n1,even)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i13) declare(n2,integer)$
(%i14) declare(n2,even)$
```

ここでの例では, 対象 n1 に奇数としての属性を与え, それから偶数としての属性を与えようとしたためにエラーとなっています. しかし, 対象 n2 に対しては, n2 が整数としての属性を付与し, それから偶数としての属性を与えており, これらの属性が互いに矛盾することがないためにエラーとは

なりません.

declare 函数による属性の付与は, その属性の表現函数の存在を前提としており, 実際に式の評価で, この属性の表現函数を用いて式の変換が遂行されます. ここで, declare 函数による影響は, 文脈 (§5.6 参照) で利用される assume 函数による影響と比較して特定の文脈上に限定されずに大域的に影響を及ぼします. これは declare 函数による属性の付与が対象に対して行われるためで, 文脈に対して対象と属性が関連付けられるためではないからです. ただし, fact 函数を使って declare 函数で付与した属性が表示されるのは, declare 函数を用いた文脈上に限定されます. 具体的には, 文脈 A 上で declare 函数で付与した属性は文脈 A 上でのみで facts(); で確認ができます.

5.4.5　declare 函数で付与可能な属性

declare 函数によって属性を付与されることで, 対象はその属性に応じた処理が行われます. この内部処理の違いに基づいて, declare 函数で付与可能な属性を分類しておきます:

declare 函数で付与可能な属性

	属性	作用
1.	evfun, evflag, nonarray, bindtest	評価に関連する属性
2.	alphabetic, constant, nonscalar, noun, scalar, mainvar	対象の型に関連する属性.
3.	feature	属性を大域変数 features に登録.
4	大域変数 opproperties 内の属性	属性に応じて処理.
5.	大域変数 features 内の属性	属性に応じて処理.

1. から 5. の属性の詳細については §5.4.8 で解説しますが, 4. と 5. の大域変数に含まれる属性について, 補足の説明を行っておきます. まず, 4. の大域変数 opproperties のリストとして含まれる属性は, 演算子や函数が持つ属性で, その表現函数を持ちます. そのために, これらの属性を与えた対象は, 入力した時点で名詞型でない限り, 自動簡易化の対象となります:

大域変数 opproperties に含まれる属性

linear	additive	multiplicative	outative	evenfun
oddfun	commutative	symmetric	antisymmetric	nary
lassociative	rassociative			

これに対し, 5. の大域変数 features のリストに含まれる属性は feature とも呼ばれます. 一部, 大域変数 opproperties にも含まれる属性がありますが, 数学的対象の一般的な性質を述べたもので, 文脈と組合せて利用可能な属性です. この大域変数 features には, 利用者が declare 函数を用いて feature として登録した属性も登録されます. 次に示す属性は Maxima-5.25.1 の既定値として含まれている属性です:

大域変数 features に含まれる属性

integer	noninteger	even	odd	rational
irrational	real	imaginary	complex	analytic
increasing	decreasing	oddfun	evenfun	posfun
commutative	lassociative	rassociative	symmetric	antisymmetric
integervalued				

5.4.6 利用者による属性の追加

declare 函数では属性に feature を指定することで, 利用者が新たに属性を追加できます. ここで追加した属性は declare 函数から利用することも可能です.

declare 函数による属性の追加と関連する函数

declare(⟨ 新属性 ⟩, feature)
featurep(⟨ 対象 ⟩,⟨ 属性 ⟩)

declare 函数で属性を宣言すると大域変数 features に割当てられたリストに新属性が追加されます. ここで対象が大域変数 feaures に関連する属性を持つかどうかは真理函数の featurep 函数を使って調べられます. この函数の真理値集合は {true,false} であり, 対象が指定した属性を持つときに 'true' を返します.

ここでは新しい属性として四元数 (quaternion) を追加してみましょう:

```
(%i3) features;
(%o3) [integer, noninteger, even, odd, rational, irrational, real, imaginary,
complex, analytic, increasing, decreasing, oddfun, evenfun, posfun, constant,
commutative, lassociative, rassociative, symmetric, antisymmetric,
integervalued]
(%i4) declare(quaternion,feature)$
(%i5) features;
(%o5) [integer, noninteger, even, odd, rational, irrational, real, imaginary,
complex, analytic, increasing, decreasing, oddfun, evenfun, posfun, constant,
commutative, lassociative, rassociative, symmetric, antisymmetric,
integervalued, quaternion]
(%i6)  declare(q1,quaternion)$
(%i7)  featurep(q1,quaternion);
(%o7)                                true
(%i8) facts();
(%o8)                          [kind(q1, quaternion)]
```

この例では, declare(quaternion,feature); によって quaternion 属性を新たに宣言します, すると, declare 函数は大域変数 features に quaternion 属性を追加します. それから, declare(q1,quaternion); によって対象 'q1' に quaternion 属性を付与します. このことは featurep 函数や facts 函数を用いて確認できます. 特に facts 函数では形式的な函数 kind の項として表われます. この facts 函数の

詳細については §5.6 を参照して下さい.

このように quaternion 属性を新たに Maxima に入れましたが, この属性は現時点では形式的なものです. 実際, 名前だけで対象が四元数 (quaternion) として持つ性質が一切含まれていないことに注意してください. Maxima では, 別途, その属性を持つ対象を処理する Maxima の函数が必要となります. この処理を行う函数が「**属性の表現函数**」です.

5.4.7　属性の表現函数

属性の表現函数の解説の前に重要な変数の解説をしておきます. まず, 大域変数 opproperties の実体は内部変数 opers に割当てられたリストです:

```
(%i8) opproperties;
(%o8) [linear, additive, multiplicative, outative, evenfun, oddfun,
       commutative, symmetric, antisymmetric, nary, lassociative, rassociative]
(%i9) :lisp opers
($RASSOCIATIVE $LASSOCIATIVE $NARY $ANTISYMMETRIC $SYMMETRIC $COMMUTATIVE
 $ODDFUN $EVENFUN $OUTATIVE $MULTIPLICATIVE $ADDITIVE $LINEAR)
```

ここで示したように, 大域変数 opproperties の内容は内部変数 opers の成分を逆に並べたものですが, 内部変数 opers はさらに内部変数*opers-list に登録されたリストから属性のみを抜き出したものです:

```
(%i9) :lisp *opers-list
(($RASSOCIATIVE . RASSOCIATIVE) ($LASSOCIATIVE . LASSOCIATIVE) ($NARY . NARY1)
 ($ANTISYMMETRIC . ANTISYM) ($SYMMETRIC . COMMUTATIVE1)
 ($COMMUTATIVE . COMMUTATIVE1) ($ODDFUN . ODDFUN) ($EVENFUN . EVENFUN)
 ($OUTATIVE . OUTATIVE) ($MULTIPLICATIVE . MULTIPLICATIVE)
 ($ADDITIVE . ADDITIVE) ($LINEAR . LINEARIZE1))
```

ここで, 属性 commutative に注目して下さい. この属性は LISP 側で$commutative が対応し, 大域変数*opers-list では, リスト ($COMMUTATIVE . COMMUTATIVE1) が対応しています. このリストの第 2 成分 commutative1 が属性の表現函数です, これだけではよく判りませんね. そこで, commutative1 の動作を LISP の函数 trace を用いて確認してみましょう:

```
(%i9) infix("(^_^)")$
(%i10)  declare("(^_^)",commutative)$
(%i11) :lisp (trace commutative1)
(COMMUTATIVE1)
(%i11)  Y (^_^) P - P (^_^) Y;
  0: (COMMUTATIVE1 ((|$(^_^)|) |$y| |$p|) NIL)
  0: COMMUTATIVE1 returned ((|$(^_^)| SIMP) |$p| |$y|)
  0: (COMMUTATIVE1 ((|$(^_^)|) |$p| |$y|) NIL)
  0: COMMUTATIVE1 returned ((|$(^_^)| SIMP) |$p| |$y|)
(%o11)                                  0
```

この例では infix 型の演算子 "(^_^)" を定義し, 可換性 (commutative 属性) を declare 函数で付与しています. それから, :lisp (trace commutative1) によって内部函数 commutative1 に trace させます. それから, この演算子項を含む式を入力すると, 内部函数 commutative1 によって処理が実行

されている様子と結果が返されていますね.

この仕組をもう少し詳しく解説しましょう. まず, Maxima に式が入力されたとき, Maxima は入力式の自動簡易化を担当する内部函数 simplifya を呼出します. ここで, 大域変数 simp の値が 'true' であれば, 内部函数 simplifya は入力式に含まれる函数や演算子の属性を調べ, 大域変数 features に登録された属性が存在すれば内部変数*opers-list に含まれるリストから, その属性に対応する属性の表現函数を用いて式の簡易化を実行します. このとき, 内部変数*opers-list の成分から Lisp の car 函数で取得されるものが属性名に対応し, 同様に Lisp の cdr 函数で取得できるものが, その属性の表現函数の名前となります.

先程の例で解説すると, 最初に入力式を内部函数 simplifya で項に分解します. それから, 各項は全て演算子属性を持つので, それらの項を内部函数 oper-apply に引渡します. それから, 内部函数 oper-apply で内部変数*opers-list に登録された属性と演算子項の属性を照合して, 一致する属性があれば属性の表現函数を用いて演算子項の処理を行いますが, ここでは属性は commutative 属性のみのために, 用いられる内部函数も commutative1 だけです. ただし, 入力式には演算子項が二つあり, 共に commutative 属性を持つので, 二度, 内部函数 commutative1 が呼出されて, 最初の演算子項の被演算子が Maxima の項順序 "$>_m$" で並び換えられます. その結果, 双方の項が一致するために '0' が得られるのです.

このような演算子の自動簡易化を用いたければ, 属性とその属性の表現函数を定義して双方を内部変数*opers-list に登録するか, 後述の tellsimp 函数を用いた「**Maxima の規則**」を用いて内部函数 simplifya に簡易化の手順を教えてやる必要があります (§5.7.7 参照). ただし, 内部変数*opers-list を直接操作する Maxima の函数は存在しないために, この操作は全て LISP 側からの操作になるでしょう.

5.4.8 declare 函数に用意された属性

式の評価に関連する属性

式の評価に関連する属性を次に示します:

式の評価に関連する属性	
declare(⟨f⟩ , evfun)	対象 ⟨f⟩ に evfun 属性を付加
declare(⟨a⟩ , evflag)	大域変数 ⟨a⟩ に evflag 属性を付加
declare(⟨a⟩ , nonarray)	⟨a⟩ が配列名であれば, 名前の評価を禁止
declare(⟨a⟩ , bindtest)	⟨a⟩ が束縛変数であることを宣言

これらの属性では属性を付与する対象の Maxima 内部の名前から記号 "$" を削除して内部変数に登録する副作用があります.

evflag 属性:　evflag 属性を持つ大域変数は, ev 函数の第 2 引数以降にその名前のみを与えると, 自動的に true が設定されて処理されます. この evflag 属性を持つ大域変数を次に示しておきます:

evflag 属性を持つ大域変数

algebraic	cauchysum	demoivre	dotscrules	%emode
exponentialize	exptisolate	factorflag	float	halfangles
infeval	isolate_wrt_times	keepfloat	letrat	listarith
logabs	logarc	logexpand	lognegint	lognumer
m1pbranch	numer_pbranch	programmode	radexpand	ratfac
ratalgdenom	ratsimpexpons	ratmx	simp	simpproduct
simpsum	sumexpand	trigexpand		

evfun 属性:　evfun 属性を持つ対象を ev 函数の第 1 引数以後に与えた場合, 第 1 引数を evfun 属性を持つ対象に自動的に作用させて評価が実行されます. 既定値として evfun 属性を持つ函数を次に示します:

evfun 属性を持つ函数

bfloat	factor	fullratsimp	logcontract
polarform	radcan	ratsimp	trigexpand
trigreduce	rootscontract	ratexpand	rectform

これらの evflag 属性と evfun 属性を持つ対象に扱いについては §5.8.3 を参照して下さい.

nonarray 属性:　この属性を付与した対象が配列であれば, 配列の成分を評価する際に多重評価を行わないという属性です:

```
(%i1) a:b$ b:c$ c:d$

(%i4) a[x];
(%o4)                                  d
                                        x
(%i5) declare(a,nonarray);
(%o5)                                done
(%i6) a[x];
(%o6)                                  a
                                        x
(%i7) x:1$a[x](x);
(%o7)                                  a  (1)
                                        1
(%i8) a;
(%o8)                                   b
(%i9) a(x);
(%o9)                                d(x)
```

ただし, ここでの例で示すように, 評価が行われないのは配列項の場合の対象名のみです. 項を構成する添字や引数の式の評価, 配列項以外の名前は評価が実行されています.

bindtest 属性:　bindtest 属性を持つ対象については, その対象を含む式の評価で LISP の bindp 函数による検査が入り, 'nil' の場合にエラーを返す仕様となっています:

```
(%i37) declare(aa,bindtest);
(%o37)                                done
(%i38) aa+1;

evaluation: unbound variable aa
 -- an error. To debug this try: debugmode(true);
(%i39) aa+1,aa=1;
(%o39)                                 2
```

この例では, 対象 aa に bindtest 属性を付与していますが, 最初の評価では対象 aa に値が割当てられていないためにエラーが返されています. ただし, 'aa+1, aa=1' という非明示的な ev 函数による評価では, 一時的に対象 aa に 1 が割当てられているためにエラーにはなりません.

記号の型に関連する属性

記号の型に関連する属性を次に示します:

記号の型に関連する属性

declare(⟨*a*⟩, scalar)	⟨*a*⟩ をスカラーとして宣言
declare(⟨*a*⟩, nonscalar)	⟨*a*⟩ をスカラーではないと宣言
declare(⟨*a*⟩, constant)	⟨*a*⟩ を定数として宣言
declare(⟨*a*⟩, alphabetic)	⟨*a*⟩ を記号として宣言
declare(⟨*f*⟩, noun)	⟨*a*⟩ を名詞型として宣言
declare(⟨*a*⟩, mainvar)	⟨*a*⟩ を主変数として宣言

scalar 属性や nonscalar 属性: これらの付与は行列やベクトルの処理で大きく影響します:

```
(%i1) A:matrix([1,2,3],[3,2,1]);
                                  [ 1  2  3 ]
(%o1)                             [         ]
                                  [ 3  2  1 ]
(%i2) declare(a1,nonscalar)$
(%i3) a1*A;
                                    [ 1  2  3 ]
(%o3)                           a1  [         ]
                                    [ 3  2  1 ]
(%i4) declare(b1,scalar)$
(%i5) b1*A;
                            [  b1   2 b1  3 b1 ]
(%o5)                       [                  ]
                            [ 3 b1  2 b1   b1  ]
```

この例では, 対象 A に行列を割当て, 対象 a1 に nonscalar 属性, 対象 b1 に scalar 属性を与え, 'a1*A' と 'b1*A' を計算させています. nonscalar 属性を与えた場合, 各成分に a1 が分配されていないことに注意して下さい.

constant 属性:　対象に定数としての属性を与えます. この constant 属性を既定値として持つ代表的な対象に, %pi, %i, %e, %phi, %i, %gamma といった数学定数に加え, inf, minf, und, ind と infinity といった無限大に関連する定数があります.

alphabetic 属性:　対象 (Maxima の記号) に付与することで, 対象は記号 "\" を付けなくてもよい記号としての属性が与えられます.(§5.1.2 参照).

noun 属性:　対象に付与すると内部で nounify 函数 を作用させ, 対象は名詞型になります.

mainvar 属性:　この mainvar(主変数) 属性の付与は多項式の処理に大きく影響します. 何故なら, 多項式は順序 "$>_m$" で最高位の変数の多項式として表現されるためです:

```
(%i1) f1:(x+y)^4$
(%i2) f1,expand;
                      4      3      2  2      3      4
(%o2)                 y  + 4 x y  + 6 x  y  + 4 x  y + x
(%i3) f1,declare(x,mainvar),expand;
                      4      3      2  2      3      4
(%o3)                 x  + 4 y x  + 6 y  x  + 4 y  x + y
(%i4) ans1:%o2$
(%i5) ans2:%o3$
```

まず, 'y$>_m$x' のために最初の式の展開は変数 y の多項式となります. それから, f1,declare(x,mainvar),expand; によって, 多項式 fl の変数 x を主変数とした式の展開を実行しています (§5.8.3 参照). この結果は, 変数 x を主変数として宣言したために変数 x の多項式になります. 次に, 主変数を変数 y とする多項式を変数 ans1, 主変数を変数 x とする多項式を変数 ans2 に割り当てておきます. ここで, ans1 と ans2 の表示の違いは式の内部表現自体が異っていることに由来します:

```
(%i6) :lisp $ans1
((MPLUS SIMP) ((MEXPT SIMP) $X 4) ((MTIMES SIMP) 4 ((MEXPT SIMP) $X 3) $Y)
 ((MTIMES SIMP) 6 ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 2))
 ((MTIMES SIMP) 4 $X ((MEXPT SIMP) $Y 3)) ((MEXPT SIMP) $Y 4))
(%i6) :lisp $ans2
((MPLUS SIMP) ((MEXPT SIMP) $Y 4) ((MTIMES SIMP) 4 ((MEXPT SIMP) $Y 3) $X)
 ((MTIMES SIMP) 6 ((MEXPT SIMP) $Y 2) ((MEXPT SIMP) $X 2))
 ((MTIMES SIMP) 4 $Y ((MEXPT SIMP) $X 3)) ((MEXPT SIMP) $X 4))
(%i6) ans1+ans2;
       4      3      2  2      3      4      3      2  2      3      4
(%o6) y  + 4 x y  + 6 x  y  + 4 x  y + 2 x  + 4 y x  + 6 y  x  + 4 y  x + y
(%i7) ev(%,simp);
                      4      3      2  2      3      4
(%o7)                 2 x  + 8 y x  + 12 y  x  + 8 y  x + 2 y
```

ここで示すように, ans1 が変数 y で式を纏めているのに対して ans2 が変数 x で纏めていることが判ります. ここで, 'ans1 + ans2' の結果はどうなるかと言えば, 最悪なことに ans1 と an2 を単純に繋げ, ans1 の末端と ans2 の先頭の "x^4" を足し合せただけの結果になっています. このように主変数が一致しなければ予想した結果は得られないことがあります. この場合は ev 函数を用いて式を再評価すれば十分です. 実際, ev 函数を用いて簡易化させれば本来の結果が得られます.

大域変数 opproperties に関連する属性

大域変数 opproperties に含まれる属性は函数や演算子の属性で, これらの属性を持つ対象は, その属性の表現函数によって自動的に処理されます:

作用に関連する属性

declare(⟨f⟩ , linear)	対象 ⟨f⟩ の線形性
declare(⟨f⟩ , additive)	対象 ⟨f⟩ の加法性
declare(⟨f⟩ , multiplicative)	対象 ⟨f⟩ の乗法性
declare(⟨f⟩ , outative)	対象 ⟨f⟩ と積の可換性
declare(⟨f⟩ , oddfun)	対象 ⟨f⟩ を奇函数として宣言
declare(⟨f⟩ , evenfun)	対象 ⟨f⟩ を偶函数として宣言
declare(⟨f⟩ , commutative)	対象 ⟨f⟩ の可換性
declare(⟨f⟩ , symmetric)	対象 ⟨f⟩ の対称性
declare(⟨f⟩ , antisymmetric)	対象 ⟨f⟩ の歪対称性
declare(⟨f⟩ , nary)	対象 ⟨f⟩ を複数引数の函数として宣言
declare(⟨f⟩ , lassociative)	対象 ⟨f⟩ の左分配律
declare(⟨f⟩ , rassociative)	対象 ⟨f⟩ の右分配律

ここで, evenfun, oddfun, commutative, symmetric, antisymmetric, lassociative と rassociative 属性は後述の大域変数 feature にも包含される属性です.

linear 属性

linear 属性は additive 属性と outative 属性の両方を持たせた属性で, 対象 f の第 1 引数に対する線形性を付与します. その結果, 次の変換が生じます:

$$f(a*x_1+b*y_1,\ldots) \quad \Rightarrow \quad a*f(x_1,\ldots)+b*f(y_1,\ldots) \qquad a,b \text{ は定数}$$

additive 属性

対象 f の第 1 引数に対する加法性を付与し, その結果, 次の変換が生じます:

$$f(x_1+y_1,\ldots) \Rightarrow f(x_1,\ldots)+f(y_1,\ldots)$$

なお, sum 函数に対しては効果がありません.

multiplicative 属性

対象 f の第 1 引数に対する乗法性を与えます. その結果, 次の変換が生じます:

$$f(x_1*y_1,\ldots) \Rightarrow f(x_1,\ldots)*f(y_1,\ldots)$$

なお, multiplicative 属性は additive 属性と同様に, 式の内部表現に依存するため, product 函数に対して無効になります.

outative 属性

対象 f の第 1 引数に対する可換積 "*" と対象 f の交換性を付与します. その結果, 次の変換が生じます:

$$f(a*x_1, b*x_2, \ldots) \Rightarrow a*f(x_1, b*x_2, \ldots)$$

ここで f の作用域の外に出される対象は数値 (整数, 有理数, 浮動小数点数, 多倍長浮動小数点数) と constant 属性を持つ対象に限定されます. この outative 属性を持つ函数として, sum 函数, integrate 函数と limit 函数があります. この属性の付与は内部函数 defprop を用いて処理されています.

evenfun 属性

対象 f に偶函数としての属性を付与します. このとき, 対象 f は 1 変数の函数でなければなりません:

$$f(-x) \Rightarrow f(x)$$

oddfun 属性

対象 f に奇函数としての属性を付与します. このとき, 対象 f は 1 変数の函数でなければなりません:

$$f(-x) \Rightarrow -f(x)$$

commutative(=symmetric) 属性

対象 f の任意の引数の順序に結果が依存しない性質を付与します. その結果, 対象 f の引数全体に対して項順序 "$>_m$" による変換が生じます:

$$f(x_1, x_2, \ldots, x_n) \Rightarrow f(x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)})$$

ここで n を対象 f の引数の総数, σ は $x_{\sigma n} >_m \ldots >_m x_{\sigma(2)} >_m x_{\sigma(1)}$ を満す, n 次の置換群 $\mathfrak{S}_n$ の元です. なお, 対称性を表現する symmetric 属性は commutative 属性そのものを用いて表現しています.

antisymmetric 属性

歪対称性を表現する属性で, 次の式の変換が発生します:

$$f(x_1, x_2, \ldots, x_n) \Rightarrow (-1)^{\#(\sigma)} f(x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)})$$

ここで n を対象 f の引数の総数, σ は $x_{\sigma n} >_m \ldots >_m x_{\sigma(2)} >_m x_{\sigma(1)}$ を満す, n 次の置換群 $\mathfrak{S}_n$ の元, $\#(\sigma)$ は σ を互換の積に分解したときの互換の総数となります.

nary 属性

nary 属性は引数が 1 以上の函数 (n-arity の函数) としての属性を与えます. 典型的な nary 属性を持つ演算子は演算子 "+" ですが, 函数 f が nary 属性を持つ場合, 演算子 "+" と同様に, 同じ函数 f を作用させた対象を引数としてとった場合に引数を融合して, 函数 f で纏める操作が行われます:

$$f(x_1,\ldots,x_i,f(x_{i+1},\ldots,x_j),x_{j+1},\ldots,x_n) \Rightarrow f(x_1,\ldots,x_i,x_{i+1},\ldots,x_j,x_{j+1},\ldots,x_n)$$

これは引数が一つしかない函数では, 射影函数と同様の性格を持つことになります:

$$f \circ f \circ \cdots \circ f(x) \Rightarrow f(x)$$

この函数は Maxima の内部表現を理解していなければ判り難い函数です. ここで次の Maxima の式の模式的表現を用いて解説してみましょう. なお, Maxima の式の内部表現の詳細は §6.4 を参照して下さい. ここでは対象 $f(x_1,\ldots,x_n)$ の模式的内部表現を模式的に次で表現します[10]:

対象 f の模式的内部表現

Maxima の表示		Maxima の模式的内部表現
$f(x_1,\ldots,x_n)$	$\Leftrightarrow$	$((f)\ x_1\ \ldots\ x_n)$

このとき, 函数 f が nary 属性を持つことは,

$$((f)\ x_1\ \ldots\ x_i\ ((f)\ x_{i+1}\ \ldots\ x_j)\ x_{j+1}\ \ldots\ x_n) \Rightarrow ((f)\ x_1\ \ldots\ x_i\ x_{i+1}\ \ldots\ x_j\ x_{j+1}\ \ldots\ x_n)$$
$$((f)\ ((f)\ x)) \Rightarrow ((f)\ x)$$

となることを意味します.

lassociative 属性と rassociative 属性

これらの属性も nary 属性以上に Maxima の式の内部表現を理解していなければ判り難い属性です. これらの属性は演算子を左がら次々と作用させられるか, 右から作用させられるかという性質に対応するものです.

まず, lassociative 属性が対象 f に付与されると次の式の変換が実行されます:

$$f(x_1,x_2,\ldots,x_n) \Rightarrow f(\ldots f(f(x_1,x_2),x_3)\ldots,x_n)$$

この変換式を nary 属性の解説で用いた模式的な内部表現で見ると次の式になります:

$$((f)\ ((f)\ldots((f)\ ((f)\ x_1\ x_2)\ x_3)\ldots x_{n-1})\ x_n)$$

このように対象 f の引数のリスト $(x_1,\ldots,x_n)$ を左側から括弧を使って対で括り, 各リストの先頭に対象 f を作用させる形になります. ここで, 対象 f を可換積 "*" で置換えると, 変換式後の式は $((\ldots((x_1 * x_2) * x_3)\ldots) * x_n)$ となって, この変換は左結合律に対応することが分ります.

[10] 正確には, ((`$f simp`) $\mathtt{x_1}$ … $\mathtt{x_n}$) となるでしょう

同様に rassociative 属性を対象 f に付与すると次の変換が行われます:

$$f(x_1, x_2, \ldots, x_n) \Rightarrow f(x_1, f(x_2, \ldots f(x_{n-2}(f(x_{n-1}, x_n))) \ldots))$$

この変換式を模式的な内部表現で見ると次の式になります:

$$((f)\ x_1\ ((f)\ x_2\ \ldots\ ((f)\ x_{n-2}\ ((f)\ x_{n-1}\ x_n))\ldots))$$

この場合は対象 f の引数のリスト '$(x_1, \ldots, x_n)$' を右側から括弧を使って対で括り, 各リストの先頭に対象 f を作用させる形になります. ここで対象 f を可換積 $*$ で置換すると, 与式は '$x_1 * (x_2 * (\ldots (x_{n-2} * (x_{n-1} * x_n)) \ldots))$' となって, この変換は右結合律に対応するものであることが分ります.

大域変数 features に含まれる属性

大域変数 features に含まれる属性の大きな特徴は, この属性によって一つの集合が定まり, その集合に対する包含関係を declare 函数を用いて宣言することになります. ここで Maxima では, 集合と対象の関係は形式的な内部函数を用いて表現します. 一つは単純な包含関係を表現する内部函数 kind であり, もう一つは集合の構成を表現する内部函数 par です:

内部函数 par と kind の構文

(par(⟨ 属性$_1$⟩,⟨ 属性$_2$⟩),⟨ 属性$_3$⟩)
(kind ⟨ 属性$_1$⟩,⟨ 属性$_2$⟩)
(kind ⟨ 対象 ⟩,⟨ 属性 ⟩)

内部函数 par:　属性$_3$ が 属性$_1$ と 属性$_2$ の和集合として構成されることを示すことに用いられます:

内部函数 par の実例

関係	概要
par(($even $odd) $integer)	整数が偶数と奇数で構成されることを示す.
par(($rational $irrational) $real)	実数が有理数と無理数から構成されることを示す. なお, この内部式は 2009/10 から削除されている.
par(($real $imaginary) $complex)	複素数が実数と純虚数から構成されていることを示す. なお, この内部式は 2009/10 から削除されている.

上の例で示すように, 内部函数 par を使った表現は以前と比べて減っており, 現在は, 整数が偶数と奇数で構成される関係を示すもののみとなっています.

内部函数 kind: 属性$_1$ で定まる集合が 属性$_2$ で定まる集合に包含される場合と, 対象$_1$ が 属性 で定められる集合に包含される場合を表現します:

内部函数 kind の実例

関係	概要
(kind $imaginary $complex)	純虚数は複素数に包含されることを示す.
(kind $integer $rational)	整数は有理数に含まれることを示す. なお, この内部式は 2009/10 から削除されている.
(kind $%i $noninteger)	純虚数 $%i$ は整数ではない.
(kind $%i $imaginary)	純虚数 i は imaginary 属性を持つ.
(kind $%pi $noninteger)	円周率 π は非整数である.
(kind $%pi $real)	円周率 π は実数である.

この kind を用いた表現は Maxima では多く用いられており, Maxima の文脈に関連するソースファイルの compar.lisp の末尾に定数や函数の属性を設定するために用いられています.

これらの内部函数は「**文脈**」における仮定を構成するものの一つです. そのために, 対象に大域変数 features に含まれる属性を付与した場合, その属性を付与した文脈上で facts(); を実行することで, 付与した属性を, これらの内部函数を用いた表記で確認することが可能であり, また, kindp を用いて表現される関係は内部函数 kindp を使って確認することも可能です:

```
(%i1) declare([x1,x2,x3],integer)$

(%i2) declare([y1,y2,y3],complex)$

(%i3) facts();
(%o3) [kind(x1, integer), kind(x2, integer), kind(x3, integer),
                    kind(y1, complex), kind(y2, complex), kind(y3, complex)]
(%i4) :lisp (kindp '$x1 '$integer)

T
```

Maxima の既定値の内部函数 kind や par を用いた表現は文脈を global にすれば確認できます. このように大域変数 features に包含される属性は文脈と大きく関連します. 文脈の詳細については §5.6 を参照して下さい.

この大域変数 feature に含まれる属性は, 対象の性質を表現するものです. このとき, 対象の数的な特性, つまり, 対象が整数で, 偶数であるといった数的な属性, もう一つは対象の作用素としての属性, つまり, 対象が線形写像であるとか, 偶函数であるといったものに大きく分類できます. ここでは, 数的な属性と作用素の属性に分けて詳細を述べることとします.

大域変数 features に含まれる数的な属性

大域変数 features に含まれる属性より, 数に関連する属性を以下に纏めておきます:

数的な属性

declare(⟨*a*⟩ , integer)	⟨*a*⟩ を整数として宣言
declare(⟨*a*⟩ , noninteger)	⟨*a*⟩ を非整数として宣言
declare(⟨*a*⟩ , even)	⟨*a*⟩ を偶数として宣言
declare(⟨*a*⟩ , odd)	⟨*a*⟩ を奇数として宣言
declare(⟨*a*⟩ , rational)	⟨*a*⟩ を有理数として宣言
declare(⟨*a*⟩ , irrational)	⟨*a*⟩ を無理数として宣言
declare(⟨*a*⟩ , real)	⟨*a*⟩ を実数として宣言
declare(⟨*a*⟩ , imaginary)	⟨*a*⟩ を純虚数として宣言
declare(⟨*a*⟩ , complex)	⟨*a*⟩ を複素数として宣言

これらの属性の概要については問題が無いかと思うので省略しますが, これらの属性を用いて Maxima 組込の数学定数に次の属性が既定値として付与されています:

数学定数に既定値として与えられた属性

定数名		付与された属性
純虚数 %i	⇒	noninteger ∧ imaginary
Napier 数 %e	⇒	noninteger ∧ real
円周率 %pi	⇒	noninteger ∧ real
Euler の定数 %gamma	⇒	noninteger ∧ real
黄金比 %phi	⇒	noninteger ∧ real

函数や演算子の属性

函数や演算子の属性は内部変数 opers と大域変数 features に登録された属性で, これらの属性は演算子や函数の線形性といった作用に関連した属性や函数の単調増加といった函数自体の特徴に関連する属性に大きく二つに分類できます.

なお, 演算子の場合は引数が演算子の型によってまちまちとなりますが, 演算子と函数の内部表現の引数の配置は違いが無いため. ここでは函数を例に解説します. さらに「演算子/函数 f」と表記する代りに, ここだけ「対象 f」と表記しますが, f 等の名前をうしろに伴わない「対象」は通常の Maxima の対象を意味します.

さらに declare 函数で付与可能な対象の属性には, 函数の単調増加性, 単調減少性, 奇函数, 偶函数, 正値函数, 解析的函数といったものもあります:

対象 f の属性

declare(⟨f⟩, analytic)	対象 ⟨f⟩ を解析的函数として宣言
declare(⟨f⟩, increasing)	対象 ⟨f⟩ を単調増加函数として宣言
declare(⟨f⟩, decreasing)	対象 ⟨f⟩ を単調減少函数として宣言
declare(⟨f⟩, posfun)	対象 ⟨f⟩ を正値函数として宣言
declare(⟨f⟩, evenfun)	対象 ⟨f⟩ を偶函数として宣言
declare(⟨f⟩, oddfun)	対象 ⟨f⟩ を奇函数として宣言

ここで, evenfun 属性と oddfun 属性は大域変数 opproperties に含まれている属性ですが, 次に示す Maxima 組込の函数への属性値に含まれているために, ここでも載せています.

Maxima の組込函数が持つ属性

delta	⇒	evenfun
sinh	⇒	increasing ∧ oddfun
cosh	⇒	posfun
tanh	⇒	increasing ∧ oddfun
coth	⇒	oddfun
csch	⇒	oddfun
sech	⇒	posfun
asinh	⇒	increasing ∧ oddfun
acosh	⇒	increasing
atan	⇒	increasing ∧ oddfun
lambert_w	⇒	complex
li	⇒	complex
cabs	⇒	complex
log	⇒	increasing

5.4.9 declare 函数以外の函数による函数属性の付加

askinteger 函数: askinteger 函数の構文を次に示します:

askinteger 函数の構文

askinteger(⟨ 式 ⟩ ,⟨ オプション引数 ⟩)
askinteger(⟨ 式 ⟩)

引数の ⟨ 式 ⟩ は Maxima の式で ⟨ オプション引数 ⟩ は even(偶数), odd(奇数), integer(整数) の何れか一つで, 省略された場合は内部で integer が自動的に設定されます. この函数は文脈を使って ⟨ 式 ⟩ が even, odd, あるいは integer であるかを判別しようとします. 文脈に必要な情報がなければ利用者に質問して, 文脈に情報を蓄えます. このとき, 内部函数 kind を使って属性が表現されます:

```
(%i12) facts();
(%o12)                                [ ]
```

```
(%i13) askinteger(zz)$
Is zz an integer?

yes;
(%i14) facts();
(%o14)                          [kind(zz, integer)]
(%i15) askinteger(zz,odd);
Is zz an odd number?

no;
(%o15)                                  no
(%i16) askinteger(zz,even);
(%o16)                                  yes
(%i17) facts();
(%o17)              [kind(zz, integer), kind(zz, even)]
```

この例では対象 zz が整数属性 integer を持つかどうかを判別しようとしますが, 最初の facts 函数の結果から, 文脈には何も登録されていないために判別ができません. そこで, askintegr 函数は利用者に integer であるかどうかを尋ねます. そこで, yes と答えると内部函数 kind を用いて対象に属性を付与するのです.
この例では対象 zz が整数属性を持つとしました. 次に zz が奇数属性を持たないとしましょう. これによって自動的に対象 zz には偶数属性が付与されます. これは integer が odd か even の何れか一つの属性を持つことが予め内部函数 'par($even $odd) $integer)' を使って compar.lisp の中で設定されているからです.

atvalue 函数:　atvalue 函数の構文を次に示します:

atvalue 函数

atvalue(⟨ 函数 ⟩, ⟨ 変数 ⟩=⟨ 式$_1$⟩, ⟨ 式$_A$⟩)
atvalue(⟨ 函数 ⟩, [⟨ 変数$_1$⟩=⟨ 式$_1$⟩,. . ., ⟨ 変数$_n$⟩=⟨ 式$_n$⟩], ⟨ 式$_A$⟩)

atvalue 函数は第 1 引数に函数項や函数項の微分 (名詞型), 第 2 引数に第 1 引数の項で用いた変数の値, 第 3 引数に第 2 引数で指定した値で函数を充足したときの函数値を指定する函数です. そして, 対象には atvalue 属性として, ここで指定した函数値が割当てられ, 大域変数 props に対象が登録されます.
第 2 引数は単変数に値を指定する場合, ⟨ 変数 ⟩ = ⟨ 値 ⟩ となりますが, 多変数の場合は,⟨ 変数$_i$⟩ = ⟨ 値$_i$⟩ を成分とするリストとなります. ここで, ⟨ 変数$_i$⟩ = ⟨ 値$_i$⟩ の記号 "=" は等号ではなく, ev 函数で用いる記号 "=" と同様に函数に含まれる左辺の変数を左側の対象で充足させることを意味する記号です.
たとえば, 式 'atvalue(f(a),a=a^2,g)' の意味は, 函数項 'f(a)' の変数 a を式 'a^2' で置換することで得られた函数項 'f(a^2)' の取る値が g であることを示します.
atvalue 函数で設定された値を printprops 函数を使って表示する際に, 函数の疑似変数を表記する為に, 記号 "@1", "@2" ,. . . が用いられます.

```
(%i10) atvalue(f(x),x=x^2,g);
(%o10)                                  g
```

```
(%i11) [f(x),f(1)];
(%o11)                                [f(x), f(1)]
(%i12) atvalue(f(x,y,z),[x=a,y=b],g(z));
(%o12)                                   g(@3)
(%i13) f(a,b,10);
(%o13)                                   g(10)
```

ここで,atvalue 属性をもう少し調べてみましょう.atvalue 属性が対象に設定されている事は properties 函数で判り, 具体的な内容は printprops 函数で表示可能です:

```
(%i14) props;
(%o14) [nset, {, }, kron_delta, trylevel, maxmin, nummod, conjugate, desolve,
                                                    eliminate, adjoint, invert, f]

(%i15) properties(f);
(%o15)                                  [atvalue]
(%i16) printprops(f,atvalue);
                                f(a, b, @3) = g(@3)

                                       2
                                    f(x ) = g

(%o16)                                   done
```

この atvalue 属性は Maxima 内部では atvalues 属性となっています. この属性値は内部函数の mget 函数を用いて得られます:

```
(%i6) atvalue(f1(x1,x2,x3,x4,x5),[x1=a,x5=b],a*b*g(x2,x3,x4));
(%o6)                                a b g(@2, @3, @4)
(%i7) :lisp (mget '$f1 'atvalues)
(((0 0 0 0 0) ($A ##### ##### ##### $B)
  ((MTIMES SIMP) $A $B (($G SIMP) &@2 &@3 &@4))))
```

atvalues 属性の属性リストは三成分のリストの並びの繰り返しで構成されています. この例では, 第一成分が函数項の変数の座を示す 0 で構成されたリストで, 第二成分が各変数の値, そして第三成分が返却される値になります. なお, 変数の値で "####" とあるのは,Maxima で変数に値が割当てられていない事を示す内部変数 unbound の値です. なお, 先程の f の様に複数の atvalue 属性が設定されている場合, 次の様なリストが得られます:

```
(%i7) :lisp (mget '$f 'atvalues)
(((0 0 0) ($A $B #####) (($G SIMP) &@3)) ((0) (((MEXPT SIMP) $X 2)) $G))
```

この場合,3 変数の函数 f の atvalues 属性と 1 変数の函数 f の atvalues 属性が並んでいることが判ります.

at 函数: 与えられた式の変数に指定された値を代入した式を返す函数です:

at 函数の構文

at(⟨ 式 ⟩, ⟨ 変数 ⟩=⟨ 式$_1$⟩)
at(⟨ 式 ⟩,[⟨ 変数$_1$⟩=⟨ 式$_1$⟩, ⟨ 変数$_n$⟩=⟨ 式$_n$⟩])

at 函数は代入処理の際に atvalue 属性を調べて式の処理するため, 代入を行う subst 函数とは異なります:

```
(%i28) atvalue(xa(x,y),x=1,cos(y)*gamma(y));
(%o28)                              cos(@2) gamma(@2)
(%i29) at(xa(x,y),[x=1,y=3]);
(%o29)                                 2 cos(3)
(%i30) subst([x=1,y=3],xa(x,y));
(%o30)                                 xa(1, 3)
```

この例では最初に atvalue 函数で 'xa(1,y)=cos(y)*gamma(y)' となることを atvalue 属性を用いて表現し, at 函数で "[x,y]→[1,3]" の場合の処理を行っています. このときに atvalue 属性を参照するので正しく処理が行えます. 一方, subst 函数は式に値を単純に代入するだけなので, 期待する値はえられません.

5.4.10　depends 函数と gradef 函数

Maxima の記号に対し, 変数の従属性と勾配を与えることが可能です:

従属性と勾配を与える函数の構文

depends(⟨ 函数 ⟩ , ⟨ 変数$_1$⟩ , . . ., ⟨ 函数$_n$⟩ , ⟨ 変数$_n$⟩)
gradef(⟨ 函数名 ⟩ (⟨ 変数$_1$⟩ , . . ., ⟨ 変数$_m$⟩), ⟨ 式$_1$⟩ , . . ., ⟨ 式$_n$⟩)
gradef(⟨ 記号 ⟩ , ⟨ 変数 ⟩ , ⟨ 式 ⟩)

depends 函数:　depends 函数は, Maxima の記号に対し, 変数への従属性を与えることが可能です. これは, 形式的な函数表記では "f(x,y,z)" で 3 変数の函数であることが表記できますが, 函数名 "f" に対しては変数が何処にも記載されていないために, 微分を行うと定数と見做されて '0' になります. depends 函数は記号が指定した変数の函数であることを示す函数です.
この depends 函数によって, 対象には dependency 属性が付与され, 大域変数 dependencies に形式的な函数として登録されます.

gradef 函数:　変数 ⟨x_i⟩ による函数の 1 階微分と ⟨ 式$_i$⟩ を結び付ける函数です. まず, gradef 函数の第 1 引数に 'f(x)' のような函数項を与えた場合, 函数名 f に gradef 属性 を付与し, 指定した式が導函数になります. そして, 大域変数 gradefs に函数名が蓄えられます.
gradef 函数の第 1 引数に "f" のような記号を与えた場合, 属性として gradef ではなく atomgrad 属性と dependency 属性が付与され, 同時に記号 "f" は大域変数 atomgrad と大域変数 dependencies に登録されます.
もし, gradef 函数で与える式の数が変数よりも少ない場合, ⟨ 函数 ⟩ の i 番目の引数 x_i が参照されます. ここで x_i は函数定義で用いる疑似変数と同類で, 函数 ⟨ 函数 ⟩ の i 番目の変数を指示するために用います.

gradef と depends に関連する大域変数

変数名	既定値	概要
gradefs	[]	勾配を定義された函数名リスト
dependencies	[]	従属性を宣言された函数名リスト

大域変数 gradefs: gradef 函数で勾配を定義すると大域変数 gradefs に割当てられたリストに函数名が蓄えられます.

大域変数 dependencies: 函数 depends や函数 gradef で指定された函数名が大域変数 dependencies に割当てられたリストに蓄えられます.

```
(%i1) gradef(f(x,y),y,x);
(%o1)                                f(x, y)
(%i2) gradefs;
(%o2)                               [f(x, y)]
(%i3) [ diff(f(x,y),x),diff(f(x,y),y)];
(%o3)                                [y, x]
(%i4) dependencies;
(%o4)                                  []
(%i5) depends(f,x,g,y);
(%o5)                            [f(x), g(y)]
(%i6) gradefs;
(%o6)                              [f(x, y)]
(%i7) depends(h,x,h,y,h,z);
(%o7)                       [h(x), h(y), h(z)]
(%i8) dependencies;
(%o8)                  [f(x), g(y), h(z, y, x)
```

ここで Maxima の組込函数も gradef 属性を用いて微分が定義されていますが, こちらは大域変数 gradefs には登録されていません.

```
(%i104) properties(sin);
(%o104) [deftaylor, rule, noun, gradef, transfun, transfun, transfun, transfun]
(%i105) printprops(sin,gradef);
                              d
                              -- (sin(x)) = cos(x)
                              dx

(%o105)                                done
```

この例では sin 函数の属性を properties 函数で表示させています. ここで表示される属性で, gradef が微分の公式の関係式が登録されている属性なので, printprops 函数を使って gradef 属性を表示させると, sin 函数の微分の公式が現れます. 組込の函数では, この属性の設定で内部函数の defprop 函数を用いています.

numerval 函数

numerval 函数の構文を次に示します:

numer 函数の構文

numerval(⟨ 変数$_1$⟩ ,⟨ 式$_1$⟩ , . . . ,⟨ 変数$_n$⟩ ,⟨ 式$_n$⟩)

⟨ 変数$_i$⟩ に数値属性 numer を付与し, numer の属性値として ⟨ 式$_i$⟩ を設定します. なお, 大域変数 numer が true のときに, numer 属性を付与された ⟨ 変数$_i$⟩ に属性値が自動的に割当てられます:

```
(%i4) numerval(n1,x^2+x+1);
(%o4)                                  [n1]
(%i5) properties(n1);
(%o5)                                [numer]
(%i6) :lisp (mget '$n1 '$numer)

((MPLUS) ((MEXPT) $X 2) $X 1)
(%i7) numer:true;
(%o7)                                  true
(%i8) n1;
                                        2
(%o8)                                  x  + x + 1
```

Taylor 級数を定める函数

deftaylor 函数:　指定した函数の taylor 展開式を指定する函数です:

deftaylor 函数

deftaylor(⟨ 函数名 ⟩(⟨ 変数 ⟩),⟨ 式 ⟩)
deftaylor(⟨ 函数名$_1$⟩(⟨ 変数$_1$⟩),⟨ 式$_1$⟩,. . . ,⟨ 函数名$_n$⟩(⟨ 変数$_n$⟩),⟨ 式$_n$⟩)

deftaylor 函数によって函数には deftaylor 属性が付加され, Taylor 展開式は deftaylor 属性の属性値として設定されます. そして, 函数名が大域変数 props に割当てられたリストに登録されます:

```
(%i1) deftaylor(f1(x),sum(x^(2*i+1)/i!,i,1,inf));
(%o1)                                  [f1]
(%i2) taylor(f1(x),x,0,10);
                                     5    7    9
                                    x    x    x
                              3     --   --   --
(%o2)/T/                     x  + 2  + 6  + 24  + . . .
```

5.4.11　属性を削除する函数

remove 函数:　対象と属性を指定して対象に付与された属性と, その属性値を削除する函数です:

remove 函数の構文

remove(⟨ 記号$_1$⟩ , ⟨ 属性$_1$⟩ , . . ., ⟨ 記号$_n$⟩ , ⟨ 属性$_n$⟩)
remove([⟨ 記号$_1$⟩ , . . .,⟨ 記号$_m$⟩], [⟨ 属性$_1$⟩ , . . .,⟨ 属性$_n$⟩])
remove (”⟨ 記号 ⟩ ”, operator)
remove(⟨ 記号 ⟩ , transfun)
remove (all,⟨ 属性 ⟩)

ここで remove 函数で削除可能な属性を示しておきましょう:

属性の削除のみ:

- assign
  LISP の putprop 函数によって属性リストを持たされた対象から属性リストを削除します.
- atvalue
  函数や記号に付与された atvalues 属性を削除します. なお, atvalues 属性は atvalue 函数 で付与されます.
- autoload
  函数名に付与された autoload 属性を削除します. autoload 属性は setup_autoload 函数 を用いて付与される属性です.
- evfun,evflag,nonarray,mainvar,constant, nonconstant,scalar,nonscalar
  対象に割当てられた属性を単純に削除します. これらの属性は全て declare 函数で与えられたもので, 関連する大域変数が存在せず, 属性値も true のみであるために属性の削除に留まります. これらの属性は declare 函数で与えられます
- matchdeclare
  matchdeclare 属性を対象から削除します. なお, matchdeclare 属性は matchdeclare 函数 で与えられる属性です.
- mode,modedeclare
  modedeclare 函数 (=mode_declare 函数) で与えた mode 属性を削除します.
- numer
  対象の numer 属性を削除します. この属性は属性値が内部函数 mputprop で与えられ, 大域変数 numer が true の場合に属性値が自動的に取り出されます. なお, 大域変数 values に登録された対象に対しても, その値を削除する作用があります.
- transfun
  対象の transfun 属性を削除します. この属性は translate 函数で与えられる属性で, putprop 函数で属性 translated として与えられています.

大域変数の書き換えを伴う:

- alphabetic
  内部変数*alphabet* に登録された Maxima の記号から alphabetic 属性を削除し, Maxima の文字にします. その結果, 内部変数の*alphabet*に割当てられたリストから対象が削除されます.

- array
  大域変数 arrays に登録された対象から, 割当てられた配列本体を削除します. その結果, 大域変数 arrays から対象が削除されます.

- alias
  大域変数 aliases に登録された対象の alias 属性を削除します. このときに大域変数 aliases からも対象が削除されます. ここで alias 属性は alias 函数で与えられます.

- function
  大域変数 functions に登録された演算子や函数の本体を削除し, 形式的な演算子や函数にします. このに大域変数 functions から対象は削除されます. この function 属性は演算子 ":=" や define 函数で与えられます.

- macro
  大域変数 macros に登録された対象の macro 属性と macro の実体を削除します. この macro 属性は演算子 "::=" を用いて与えられます.

- operator と op
  演算子としての属性を削除します. ここで function 属性を持つ演算子は演算子としての属性を削除されるだけなので, 通常の函数になります.

- features
  大域変数 features に登録された対象の feature 属性を削除します. この結果, 対象は大域変数 features に割当てられたリストから削除されます. なお, features 属性は declare 函数で与えられます.

- gradef,grad,atomgrad
  登録された函数の勾配を削除します. なお, 大域変数 gradefs に割当てられたリストに登録された対象は削除されます. ここで atomgrad 属性を持った対象は gradefs に登録されていません. なお, これらの属性は gradef 函数で与えられます

- defatylor と taylordef
  Taylor 展開式を削除し, その結果, 大域変数の deftaylor に割当てられたリストから対象が削除されます. deftaylor 属性は deftaylor 函数で与えられます.

- depends,dependency,depend
  大域変数 dependencies に登録された対象から,dependency 属性を削除します. その結果, 大域変数 dependencies から対象が削除されます. なお, dependency 属性は depends 函数や gradef 函数で与えられます.

- rule
  大域変数 rules に登録された対象から rule 属性を削除します. その結果, 対象は大域変数 rules から削除されます. rule 属性は defrule 函数で与えられます.

- value
  大域変数 values に登録された対象から割当てた値を削除します. その結果, 対象は大域変数 values から削除されます. value は正確には属性ではありません. 演算子 ":" による割当で対象が大域変数 values に登録されます.

なお, remove 函数は与えられた属性が存在しないときでもエラーを返しません. remove 函数の返却値は常に 'done' です.

5.4.12 属性の表示

属性の表示に関連する函数として, properties 函数, propvars 函数と printprops 函数があります:

属性の表示に関連する函数と大域変数

properties(⟨ 対象 ⟩)
propvars(⟨ 属性 ⟩)
props

properties 函数: 対象に付与された全ての属性を記載したリストを表示します. 次の例では put 函数で対象に属性を指定し, properties 函数で対象に指定した各種属性を確認を行って, get 函数を使って属性値を取り出しています:

```
(%i37) put(Mike,"2004/07/4",birthday)$
(%i38) put(Mike,"10[Kg]",Weight)$
(%i39) put(Mike,"White-Black-Red",Color)$
(%i40) properties(Mike);
(%o40)           [[user properties, Color, Weight, birthday]]
(%i41) get(Mike,Color);
(%o41)                          White-Black-Red
```

ここで, リストの先頭の 'user properties' は利用者が定義した属性であることを示します. Maxima で予め付与される属性の場合, 'database info' となります:

```
(%i42) properties(integer);
(%o42)                    [database info, feature]
(%i43) properties(real);
(%o43)                    [database info, feature]
```

propvars 函数: 函数内部で properties 函数を用いる函数で, 大域変数 props に割当てられたリストから属性を持つ対象のリストを返します. たとえば, propvars(atvalue) で atvalue 函数で値が設定された対象のリストを返します:

```
(%i23) atvalue(f(x),x=0, 0)$
(%i24) atvalue(g(x),x=1, 1)$
(%i25) propvars(atvalue);
(%o25)                                [f, g]
```

この例では, 形式的な函数 f と函数 g にそれぞれ 0.0 と 1.0 での値を atvalue 属性の属性値として与えています. 次に, propvars 函数の引数として atvalue 属性を指定すると atvalue 属性を持つ函数 f と函数 g で構成されたリストが返却されています.

printprops 函数:　対象に付与した属性の属性値を表示する函数です. 対象のリストも引数として使えますが, 引数として printprops 函数に与えられる属性は一つだけです:

printprops 函数

printprops(⟨ 記号 ⟩, ⟨ 属性 ⟩)
printprops([⟨ 記号$_1$⟩, ..., ⟨ 記号$_n$⟩], ⟨ 属性 ⟩)
printprops(all, ⟨ 属性 ⟩)

なお, printprops で表示可能な属性は次のものに限定されます:

printprops で表示可能な属性

属性名	属性	概要
atvalue	函数値属性	atvalue 函数で与えられます.
atomgrad	勾配属性	gradef 函数で与えられます.
gradef	勾配属性	gradef 函数で与えられます.
matchdeclare	並びの照合変数の属性	matchdeclare 函数で与えられます (§5.7.3).

第 1 引数に all を指定すると指定した属性を持つ全ての記号と値が表示されます:

```
(%i28) atvalue(f(x),x=0, 0)$
(%i29) atvalue(g(x),x=1, 1)$
(%i30) matchdeclare([_a,_b],true)$
(%i31) printprops(all,atvalue);
                                    f(0) = 0

                                    g(1) = 1

(%i32) printprops(all,matchdeclare);
(%o32)                          [true(_b), true(_a)]
```

ここでの例では, 最初に atvalue 函数を用いて形式的函数 f と g に値を設定し, 次に, 記号_a と記号_b に matchdeclare 属性を与えています. printprops 函数に第 1 引数に all, 第 2 引数に atvalue 属性を指定すると, atvalue 属性の値が表示され, 同様に第 2 引数に matchdeclare 属性を指定すれば, 記号_a と記号_b の matchdeclare 属性値が返却されています.
Maxima には属性が付与された対象の名前から構成された大域変数 props があります.

大域変数 props: declare 函数, atvalue 函数や matchdeclares 函数等で属性が指定された記号が追加されたリストが割当てられる大域変数です. Maxima 起動時の一例を示しておきます:

```
(%i1) props;
(%o1) [nset, {, }, trylevel, maxmin, nummod, conjugate, erf_generalized, beta,
                                                desolve, eliminate, adjoint, invert]
```

なお, Maxima-5.9.3 以前では大域変数 props にシステムに関連する対象の属性が数多く登録されていましたが, Maxima-5.10.0 からシステムと利用者を区分し, 利用者側で属性を指定した対象が登録されるように変更されています.

5.5　論理式

5.5.1　Maxima の論理式について

Maxima の論理式は §5.1.16 で述べているように, 真理値を値 (=意味) として持つ, 帰納的に構成された Maxima の対象です. Maxima の論理式は, Maxima の自動簡易化, あるいは函数を介した判断により, 'true', 'false' や 'unknown' といった「**真理値**」[11]を値に持ちます.

Maxima の論理式の例を幾つか示しておきましょう:

```
(%i124) 3>5;
(%o124)                                3 > 5
(%i125) 4#1;
(%o125)                                4 # 1
(%i126) x^2+y^2+z^2>0;
                                     2    2    2
(%o126)                             z  + y  + x  > 0
(%i127) diff(f(x),x,2)+k*f(x)=0;
                                   2
                                  d
(%o127)                           ––– (f(x)) + k f(x) = 0
                                    2
                                  dx
```

最初の二つの式は演算子 ">" や演算子 "#" を用いた数を比較する論理式ですが,　ここで示すように入力式そのままが返却されています. このように Maxima に入力された論理式がただちに評価されるとは限りません. 論理式の評価は演算子 "and", "or", "not" といった演算子や ev 函数のような真偽の判別を行なう函数がなけれは自動的に実行されません. なお, ここでは論理式の真偽を判別することを「**判断**」と呼びます.

その次の二つの論理式は (自由) 変数や函数を伴う論理式の例になります. ここで変数を持つ論理式を「**述語**」と呼び, 変数を持たない論理式を「**命題**」と呼びます. 述語は変数の値によって述語の意味 (=真理値) が異なります. その一方で述語 '$x - x = 0$' のように変数 x の値に依存せずに真偽が定まる述語のことを「**恒真式 (tautology)**」と呼びます. ここで述語 '$x^2 - 3x + 2 = 0$' は '$x = 1$ または $x = 2$' の場合に限って真になりますが, このように変項の値よっては真になることがある述語を「**充足可能な述語**」と呼び, 変項の値とは無関係に偽となる述語を「**充足不可能な述語**」と呼びます.

5.5.2　論理式の判断

文脈の概要

Maxima での論理式の判断は「**文脈**」と呼ばれる Maxima の機構が用いられます (§5.1.17,§5.6 参照). この処理を簡単に解説しておきましょう: まず, 文脈に登録された論理式を $P_1, \ldots, P_n$, 判断す

[11] ここでは unknown も加えた広義の真理値です.

べき論理式を P_0 とします. ここで新たに論理式 '$P_0 \wedge P_1 \wedge \cdots \wedge P_n$' を構成し, この論理式の判断を行うというものです (§5.1.17, §5.6 参照).

真理値集合

Maxima では論理式の意味となる真理値の集合が選択できます. 選択できるのは, 真理値集合が '{true, false, unknown}' の「**広義の真理値**」と '{true, false}' の「**狭義の真理値**」の二種類で, 大域変数 prederror の設定で大域的に切換えられます:

大域変数 prederror: Maxima の論理式の判断で用いられる真理値集合を切換える大域変数です:

大域変数 prederror

変数名	既定値	概要
prederror	true	真理値集合の切替を遂行

この大域変数 prederror が 'false' の場合は広義の真理値集合 '{true,false,unknown}' による判断結果が返され, 'true' の場合は狭義の真理値集合 '{true,false}' による判断結果が返されます.

ここで広義の真理値集合の場合, 論理式の意味が真であれば 'true', 偽であれば 'false' となり, 真偽の判断ができない場合には 'unknown' となります. それに対して狭義の真理値集合が真理値集合であれば, 判別不能の命題は 'unknown' ではなく 'false', あるいは何らかのエラーが返却されます.

論理式の自動評価

二項間の関係の演算子 ">", "<", ">=", "<=", "=", "#" や equal 函数と notequal 函数のみから構成された論理式が入力されても Maxima は自動的に判断を行いませんが, 演算子 "and", 演算子 "or" と演算子 "not" を含む論理式が入力されればただちに評価を行います:

```
(%i1) 4>=1;
(%o1)                                4 >= 1
(%i2) 1>3 and 4>=1;
(%o2)                                false
(%i3) not(3>1);
(%o3)                                false
(%i4) x>1;
(%o4)                                x > 1
(%i5) not(x>1);
(%o5)                               x <= 1
(%i6) (x+1)^2-expand((x+1)^2)=0 and 4>1;
(%o6)                               false
(%i7) (x+1)^2-expand((x+1)^2)=0 or 4<1;
(%o7)                               false
(%i8) not((x+1)^2-expand((x+1)^2)=0 or 4<1);
(%o8)                                true
(%i9) equal((x+1)^2-expand((x+1)^2),0) and 4>1;
(%o9)                               true
```

```
(%i10) notequal((x+1)^2-expand((x+1)^2),1) and 4>1;
(%o10)                                  true
```

最初の論理式 '4>=1' の入力で Maxima は判断を行っていませんが, 演算子 "and" を作用させることで 'true' と判断しています. 同様に論理式に演算子 "not" を作用させると, 真偽の決定可能な論理式に対しては真理値を返し, 決定不能な論理式に対しては大域変数 prederror が 'false' であれば与えられた論理式の否定と同値となる論理式を返しています.

ここで演算子 "and" と演算子 "or" は, 被演算子となる論理式の判断を行い, それから得られる真理値の演算を実行する演算子です. 論理式の判断では文脈が用いられますが, 論理式に演算子 "=" 演算子 "#" が含まれているときには注意が必要になります. 実際, 論理式 '(x+1)^2-expand((x+1)^2)=0 and 4 >1' と論理式 '(x+1)^2-expand((x+1)^2)=0 or 4 <1' の判断で明瞭に現われています. これらの論理式中の多項式を展開してしまえば真となることが容易に判りますが, Maxima は全て 'false' を返しています. その一方で equal 函数や notequal 関数を用いた論理式では正しい値が返却されています. これは Maxima 内部での処理の違いによるものです. まず, equal 函数と notequal 函数の処理には, 有理数を簡易化する ratsimp 函数による処理が含まれています. しかし, 演算子 "=" と演算子 "#" には, そのような多項式の簡易化が含まれていないために, 別途, ev 函数 (§5.8.3 参照) を論理式に適用しなければ簡易化処理は行われません. このことから, 論理式中の式の簡易化による処理が論理式の判断の前提となる対象では, 同値性の判断には演算子 "=" ではなく equal 函数を, 非同値性には演算子 "#" ではなく notequal 函数を用いることを薦めます. また, 同値性の判断に問題がある場合, 簡易化が十分に行われているか検証することも薦めます.

5.5.3　量化詞を表現する函数

Maxima には量化詞を表現する函数として, every 函数と some 函数があり, これらの函数は全称記号 "∀" と存在記号 "∃" にそれぞれ対応します. これらの函数では述語の名前を第 1 引数に取り, 第 2 引数としては, その述語が真となる対象をリストや集合で表現した対象となります.

every 函数:　全称記号 "∀" に対応する函数で, 真理函数と値域を引数とします:

every 函数の構文

every(⟨ 述語名 ⟩, ⟨ リスト ⟩)
every(⟨ 述語名 ⟩, ⟨ 集合 ⟩)
every(⟨ 述語名 ⟩, ⟨ リスト$_1$⟩, . . . , ⟨ リスト$_n$⟩)

every 函数は第 1 引数を述語とし, 第 2 引数以降の対象に対して述語が全て 'true' となる場合に 'true', 'false' が一つでもあれば 'false' を返却し, それ以外は 'unknown' を返します.

some 函数:　存在記号 "∃" に対応する函数で, 構文は every 函数と同様です:

some 函数の構文

some(⟨ 述語名 ⟩,⟨ リスト ⟩)
some(⟨ 述語名 ⟩,⟨ 集合 ⟩)
some(⟨ 述語名 ⟩,⟨ リスト$_1$⟩,...,⟨ リスト$_n$⟩)

some 函数は第 1 引数を述語名とし, この述語が第 2 引数以降の対象に対して 'true' となる成分があれば 'true', 全て 'false' の場合には 'false', それ以外は 'unknown' を返します.

every 函数と some 函数による判断は次の手順で行われます:

1. 引数が述語名 P と n 成分リスト L, あるいは m 成分の集合 S の場合, $P(L_i)_{1\leq i\leq n}$, あるいは $P(S_i)_{1\leq i\leq n}$ から判断します.
2. 引数が述語名 P と二つ以上のリスト $L_1,\ldots,L_m$ の場合, $P(L_1^i,\ldots,L_m^i)_{1\leq i\leq m}$ から判断します.

1. の述語は対象の属性を表現し, この場合のみ引数に集合が使えます. それから 2. の述語は複数の対象の関係を表現しするもので, 各リストの左の成分から順番に判断が行われるために全てのリストが同じ長さでなければなりません. このように対象の順番に自体に意味があるため, 関係を表現する場合には, 引数として集合は使えません.

every 函数と some 函数に与える述語名は通常の函数であれば, その函数名を与えます. 関係を表現する演算子の場合, その演算子を二重引用符で括って演算子名として与えなければなりません:

```
(%i6) every("<",[1,2,3,4,5],[0,0,0,0,0]);
(%o6)                                true
(%i7) some("<",[-1,-2,-3,4,-5],[0,0,0,0,0]);
(%o7)                                true
```

5.5.4 同値性と非同値性の表現

述語 '$x^2-3x+2=0$' は,'$x=1$' や '$x=2$' といった二つの述語の何れかの述語があれば, この述語の判断が可能になります. この判断に必要な条件を前提条件と呼びますが, この前提条件となりうる述語を蓄える仕組を Maxima では文脈と呼びます (§5.1.16, §5.5 参照). しかし, Maxima の文脈に蓄えられる論理式で利用可能な関係の演算子とそうでない演算子があります:

- 文脈で利用可能な関係の演算子: "¿=", "¿", "¡=", "¡"
- 文脈で利用不可の関係の演算子: "=", "#"

文脈で利用出来ない関係の演算子 "=" と演算子 "#" の代りに, equal 函数と notequal 函数を用いて述語を構成しなければなりません:

equal 函数と notequal 函数の構文

equal(⟨ 式$_1$⟩,⟨ 式$_2$⟩)
notequal(⟨ 式$_1$⟩,⟨ 式$_2$⟩)

equal 函数:　演算子 “=” に対応する函数で, 二つの引数を取り, それらの対象が同値であることを表現する論理式を構築します.

notequal 函数:　演算子 “#” に対応する函数で, 二つの引数を取り, それらの対象が同値でないことを表現する論理式を構築します.

not 演算子の作用による変換:　not 演算子は真理値の ‘true’ を ‘false’ , ‘false’ を ‘true’, ‘unknown’ はそのままにする演算子です. この not 演算子を作用させることで, これら equal 函数と notequal 函数は互いに移り合えます:

```
(%i5) not equal(x^2,y^3);
                                      2    3
(%o5)                              notequal(x , y )
(%i6) not notequal(x^2,y^3);
                                      2    3
(%o6)                               equal(x , y )
```

対応する演算子との相違点:　equal 函数と演算子 “=”, notequal 函数と “#” の意味は同じですが, その内部表現は異なります. このことを簡単な例を使って確認しましょう. ここで内部表現の確認のために演算子 “:lisp” を用います. また, 式の内部表現に関しては §6.4 を参照して下さい:

```
(%i1) eq1:x^2=y^3;
                                       2    3
(%o1)                                 x  = y
(%i2) eq2:x^2#y^3;
                                       2    3
(%o2)                                 x  # y
(%i3) eq3:equal(x^2,y^3);
                                      2    3
(%o3)                               equal(x , y )
(%i4) eq4:notequal(x^2,y^3);
                                      2    3
(%o4)                              notequal(x , y )
(%i5) :lisp $eq1
((MEQUAL SIMP) ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 3))
(%i5) :lisp $eq3
(($EQUAL SIMP) ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 3))
(%i5) :lisp $eq2
((MNOTEQUAL SIMP) ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 3))
(%i5) :lisp $eq4
(($NOTEQUAL SIMP) ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 3))
```

まず, 演算子 “=” を用いた述語 eq1 の内部表現の函数名が “MEQUAL” であるのに対し, equal 函数を用いた述語 eq3 の函数名は “$EQUAL” になっています. 同様に演算子 “#” を用いた述語 eq2 の函数名が “MNOTEQUAL” であるのに対し, notequal 函数を用いた述語 eq4 の函数名は “$NOTEQUAL” になっています. このように演算子は Maxima の函数としての書式を持っていますが, 対応する内部表現は Maxima の内部函数の書式に類似した書式です. これらの機能の違いは,

equal 函数と notequal 函数が必ずしも明確ではない二つの対象の関係を表現することを目的としているのに対し, 演算子 “=” と演算子 “#” は Maxima の自動処理や代入だけで関係が明瞭となる二つの対象の関係, あるいは方程式のように特定の函数を用いて処理しなければならない式の表現を目的としているためです.

そのために文脈での利用を考慮すると, 同値性や非同値性を表現する場合, 函数 equal と函数 notequal を用いた式が妥当であることが判ります. また, assume 函数を用いて文脈に登録可能な論理式は, 演算子 “or”, “=” や “#”, および, 真理函数を含まない論理式に限定されます. また, 論理式の評価で文脈を活用する場合, その論理式に含まれる同値性を表現する論理式には演算子 “=” ではなく函数 equal, 非同値性を表現する論理式も同様に演算子 “#ではなく函数 notequal を用いなければなりません:

```
(%i1) assume(equal(x^2,1));
                                    2
(%o1)                         [equal(x , 1)]
(%i2) x^2=1,pred;
(%o2)                              false
(%i3) equal(x^2,1),pred;
(%o3)                              true
(%i4) equal(x^2-x,1-x),pred;
(%o4)                              true
(%i5) equal(x^2-1,0),pred;
(%o5)                              true
(%i6) x^2-1=0,pred;
(%o6)                              false
(%i7) x^2=1,pred;
(%o7)                              false
```

この例では最初に equal 函数を用いて項 ‘x^2’ と項 ‘1’ が同値であるという条件を文脈に assume 函数を用いて追加しています. 文脈を用いた ev 函数による判断は, equal 函数を用いた論理式に対しては有効であっても, 演算子 “=” を用いた式に対しては文脈の内容が反映されません. 同様に, is 函数, maybe 函数と ev 函数や演算子 “and”, “or” と “not” について言えます.

5.5.5 論理式を評価する函数

論理式を判断する函数には is 函数, maybe 函数, それと ev 函数があります. is 函数と maybe 函数は与えられた論理式を評価して真理値を返却する函数ですが, ev 函数は与えられた式の展開や微分といった評価もできる非常に高機能な函数です. これらの構文を次に示しておきます:

述語を評価する函数

is(⟨ 述語 ⟩)
maybe(⟨ 述語 ⟩)
ev(⟨ 述語 ⟩,pred)
ev(⟨ 述語 ⟩,⟨ 変数$_1$⟩ = ⟨ 値$_1$⟩,...,⟨ 変数$_n$⟩ = ⟨ 値$_n$⟩,pred)
ev(⟨ 述語 ⟩,⟨ 変数$_1$⟩ = ⟨ 値$_1$⟩,...,⟨ 変数$_n$⟩ = ⟨ 値$_n$⟩, ⟨ オプション$_1$⟩,...,⟨ オプション$_n$⟩,pred)

is 函数と maybe 函数:　is 函数と maybe 函数は与えられた論理式の判断を行う函数です. これらの函数が返却する真理値集合は「広義の真理値」{true, false, unknown} にも対応しており, 述語が真であれば 'true', 偽であれば 'false', そして, 判定不能であれば 'unknow' を返します.

大域変数 prederror が 'true' であれば is 函数と maybe 函数は同様の働きをしますが, 'false' であれば is 函数は述語の評価の結果が 'true' にならなかったときにエラー表示を行ないます.

is 函数と maybe 函数は与えられた述語を多少は簡易化して解釈しますが, 式の展開, 微分や値の代入といった操作は一切行いません. これらの函数を利用する場合は予め式を可能な限り目的に合せて変換しておくことと, さまざまな仮定を文脈に登録しておく必要があります.

ev 函数:　ev 函数は論理式に含まれる数式の展開, 微分, 簡易化等の変換, および, 変数の代入といった処理ができる函数で, このような式の処理に加えて, 引数に pred を指定することで第 1 引数に与えた述語の判断ができます. この ev 函数の詳細については §5.8.3 を参照して下さい.

ここでは方程式と大小関係を組合せた述語 '(x+1)^2-x^2-2*x-1=0 and 3>0' の ev 函数による判断を示します:

```
(%i7) (x+1)^2-x^2-2*x-1=0 and 3>0;
(%o7)                                false
(%i8) expr1:'((x+1)^2-x^2-2*x-1=0 and 3>0);
                                  2     2
(%o8)                      (x + 1)  - x  - 2 x - 1 = 0 and 3 > 0
(%i9) ev(expr1,expand,pred);
(%o9)                                true
```

ev 函数で述語を評価する場合, 引数として必ず "pred" を指定しますが, その他に式の性質や必要に応じて "expand", "ratsimp" や代入等を指示しなければなりません. ここでの例では, 方程式の左辺を展開することで得られた式が恒等的に 0 と等しいことと, 3 が 0 より大であることを判断しなければなりません. そのため, 与式を "expand" を指定することで項の展開を行った上で "pred" で得られた式の判断を行うことになります.

では, 方程式の解の検証を行う場合はどうでしょうか？ たとえば, 述語 'diff(f(x),x)-f(x)=0 and f(x)=exp(x)' と入力すれば Maxima は真偽の判定を行うでしょうか. この場合は変数に値を代入す

ることで確認が行えますね. この評価は ev 函数を使えば簡単に行えます. 実際, 式 'ev(diff(f(x),x)-f(x)=0, f(x)=exp(x), pred)' を評価すればよいのです. さらに次の判断もできます:

```
(%i21) infix("->");
(%o21)                                    ->
(%i22) a ->b:=block(ev(b,a,expand,diff,pred));
(%o22)              a -> b := block(ev(b, a, expand, diff, pred))
(%i23) (f(x)=sin(x)) -> '(diff(f(x),x,2)+f(x)=0);
(%o23)                                    true
(%i24) true -> ((x+1)^2-x^2-2*x-1=0);
(%o24)                                    true
(%i25) false -> ((x+1)^2-x^2-2*x-1=0);
(%o25)                                    true
```

ここでの最後の二つの例は, この微分の話に限定して数理論理学の「A ならば B」, すなわち, $A \to B$ の論理演算子 $\to$ と同じ結果になることを確認したものです.

5.5.6 Maxima の真理函数

Maxima の真理函数は, Maxima の対象に対して 'true', 'false' といった真理値を返す函数です. 数理論理学の真理函数は真理値集合から真理値集合への函数であり, この本の真理函数よりも狭義の函数になります.

この真理函数は Maxima には非常に多く存在し, Maxima の式がある属性を持つかどうかを判定する函数の場合, LISP 風に末尾に p が付くことから, 名前で判別できる函数もあります.

ここでは複数の引数を取る特徴のある函数に関して解説します.

zeroequiv 函数: 与えられた式が 0 に等しいかどうかを判別する函数です.

zeroequiv 函数の構文

zeroqeuiv(⟨ 式 ⟩,⟨ 変数 ⟩)

この zeroequiv 函数の真理値集合は {true,false,dontknow} です. この zeroequiv 函数は与えられた式が 0 に等しいかどうかを判別するだけの函数で, 判定不能の場合にのみ, "dontknow" を返しますが, もちろん, この函数の能力はあまり高くはありません. そのために函数 zeroequiv に引渡す式は予め簡易化しておく必要があります:

```
(%i6)  zeroequiv (sin(2*x) - sin(x)*cos(x), x);
(%o6)                                    false
(%i7)  zeroequiv (sin(2*x*y) - sin(x*y)*cos(x*y), x);
(%o7)                                  dontknow
```

freeof 函数と lfreeof 函数:　式中の変数の有無を調べられる函数に freeof 函数と lfreeof 函数があります.

freeof 函数と lfreeof 函数の構文

freeof(⟨ 変数$_1$⟩ ,...,⟨ 変数$_n$⟩ , ⟨ 式 ⟩)	⟨ 変数$_i$⟩ が ⟨ 式 ⟩ に存在しない場合
lfreeof([⟨ 変数$_1$⟩ ,...,⟨ 変数$_n$⟩], ⟨ 式 ⟩)	⟨ 変数$_i$⟩ が ⟨ 式 ⟩ に存在しない場合

この freeof 函数は ⟨ 変数$_i$⟩ が ⟨ 式 ⟩ の中に現われなければ true を返し, それ以外は false を返します. ここで, ⟨ 変数$_i$⟩ は変数, 添字付けられた名前, 函数, あるいは二重引用符 """ で括られた演算子が扱えますが, freeof 函数では sum 函数や product 函数の内部で利用される疑似変数に対して適用できません. 疑似変数を ⟨ 変数$_i$⟩ に指定した場合は式に疑似変数が含まれていても 'true' を返します.

lfreeof 函数も動作は freeof と殆ど同じものです. ただし, 変数をリストで与える点と演算子が扱えない点で異なります.

subvarp 函数:　形式的な函数で, 引数の変数が添字付けられている場合に 'true', それ以外は '0' を返す函数です.

subvarp 函数

subvarp(⟨ 式 ⟩)

この函数の内部処理は与えられた式の形が a[i] の形式, 内部表現で "(($A SIMP ARRAY) $I)" の書式であれば 'true', それ以外は 'false' を返す函数です:

```
(%i1) subvarp(a[1]);
(%o1)                         true
(%i2) subvarp(a[i]);
(%o2)                         true
(%i3) subvarp(x^2+y+1);
(%o3)                         false
(%i4) subvarp("いろはにほへと");
(%o4)                         false
(%i5) a[1]:10;
(%o5)                          10
(%i6) subvarp(a[1]);
(%o6)                         false
(%i7) a[2]:b[100];
(%o7)                         b
                               100
(%i8) subvarp(a[2]);
(%o8)                         true
```

この例で示すように subvarp 函数は入力式の内部表現のみに影響されます.

順序に関連する真理函数

順序に関連する函数の構文を示しておきます.

順序に関連する真理函数
orderlessp(⟨ 式$_1$⟩,⟨ 式$_2$⟩)
ordergreatp(⟨ 式$_1$⟩,⟨ 式$_2$⟩)
cgreaterp(⟨ 文字$_1$⟩,⟨ 文字$_2$⟩)
clessrp(⟨ 文字$_1$⟩,⟨ 文字$_2$⟩)

orderlessp 函数と ordergreaterp 函数: 与えられた二つの式の筆頭項を Maxima の項順序 "$>_m$" を用いて比較する函数です. 詳細は §4.8 を参照して下さい.

cgreaterp 函数と clessp 函数: 文字 (長さが 1 の Maxima の文字列) に対して Maxima の項順序 "$>_M$" を用いて比較を行う函数です.

演算子に関連する真理函数

演算子に関連する真理函数に featurep 函数と operator 函数があります. なお, featurep 函数は declare 函数と組んで用いるため, その詳細は §5.4 を参照して下さい.

operatorp 函数: 与式に指定した演算子が含まれているかどうかを判断する函数です:

operatorp の構文
operatorp(⟨ 式 ⟩,⟨ 演算子 ⟩)
operatorp(⟨ 式 ⟩,[⟨ 演算子$_1$⟩,...,⟨ 演算子$_n$⟩])

operatorp 函数は演算子単体, あるいはリストの成分として与えた演算子が与式に含まれていれば 'true', そうでなければ 'false' を返却します.

unknown 函数: 実体の無い演算子や未定義の函数を持つ式に対して 'true' を返す函数です:

unknown 函数の構文
unknown(⟨ 式 ⟩)

具体的な例として, 演算子 "pochi" を実体を持たない演算子として定義したときの unknown 函数の値と, 実体を定義したあとの unknown 函数の値を比較しておきましょう:

```
(%i39) infix("pochi")$
(%i40) unknown(a pochi b);
(%o40)                          true
(%i41) x pochi y:= x+y+x*y;
(%o41)                x pochi y := x + y + x y
(%i42) unknown(a pochi b);
(%o42)                         false
```

identity 函数:　引数を無評価でそのまま返却する函数です:

identity 函数の構文

identity(⟨ 式 ⟩)

この函数は “(defun $identity (x) x)” で定義された函数で, 与えられた引数をそのまま返却する函数であることが判りますね. なお, この identity 函数は内部で some 函数や every 函数と組合せて多く用いられています.

charfun 函数:　述語の値が ‘true’ であれば ‘1’, ‘false’ であれば ‘0’, それ以外は函数を名詞型で返す函数です:

charfun 函数の構文

charfun(⟨ 述語 ⟩)

この函数は述語の表現函数としての性格を持っています:

```
(%i77)  charfun(3>1);
(%o77)                                1
(%i78)  charfun(4<1);
(%o78)                                0
(%i79)  charfun(x>1);
(%o79)                          charfun(x > 1)
```

この例では ‘3 >1’ のような真である論理式に対しては 1, ‘4 <1’ のように偽の論理式には 0 を返却し, 前提条件を持たない自由変数 x に対する論理式 ‘ x>1’ のように判断不能な論理式に対しては, 入力式をそのまま名詞形で返しています.

5.5.7　引数が一つの真理函数

Maxima には他にも多くの真理函数があります. 特に多いのが引数が一つだけの函数です. そのために, ここでは基本的と思われる真理函数と真理函数が真を返すために象が満すべき条件を纏めた一覧を示すことに留め, 詳細は各対象の解説で行うこととします:

整数に関連する真理函数

整数の性質に関連する真理函数

oddp(⟨ 式 ⟩)	奇数の場合
evenp(⟨ 式 ⟩)	偶数の場合
primep(⟨ 式 ⟩)	素数の場合

文字に関連する真理函数

文字に関連する真理函数

lowercasep(⟨ 式 ⟩)	小文字の文字列の場合
uppercasep(⟨ 式 ⟩)	大文字の文字列の場合
digitcharp(⟨ 式 ⟩)	数字の場合

対象の型 に関連する真理函数

型に関連する真理関数

atomp(⟨ 式 ⟩)	原子の場合
numberp(⟨ 式 ⟩)	数値の場合
bfloatp(⟨ 式 ⟩)	bigfloat 型の場合
floatnump(⟨ 式 ⟩)	浮動小数点数の場合
integerp(⟨ 式 ⟩)	整数の場合
ratnump(⟨ 式 ⟩)	有理数の場合
ratp(⟨ 式 ⟩)	CRE 表現, 或いは taylor 級数型の場合
taylorp(⟨ 式 ⟩)	taylor 級数型の場合
constantp(⟨ 式 ⟩)	定数の場合
scalarp(⟨ 式 ⟩)	数, 定数やスカラとして宣言された変数, 数, 定数, そして, 行列やリストを含まない式の場合
nonscalarp(⟨ 式 ⟩)	非スカラーの場合
matrixp(⟨ 式 ⟩)	行列の場合
diagmatrixp(⟨ 式 ⟩)	対角行列か二次元配列の場合
lstringp(⟨ 式 ⟩)	LISP の文字列の場合
stringp(⟨ 式 ⟩)	Maxima の文字列の場合

5.5.8 その他の函数

compare 函数

compare(⟨ 式$_1$⟩,⟨ 式$_2$⟩)

compare 函数は与えられた式を比較する演算子で, 述語 ⟨ 式$_1$⟩⟨ 演算子 ⟩⟨ 式$_2$⟩ が真となる演算子を返す函数です.
内部的には, どちらか一方が実数でなく, 両方の式が等しければ演算子 “=” を返し, 等しくない場合には notcomparable を返します. 双方が実数の場合, ⟨ 式$_1$⟩ − ⟨ 式$_2$⟩ を計算して, その符号で大小関係を返し, 不定の場合は “#” を返して大小関係が判らない場合や差が計算出来ない場合には “unknown” を返します.

```
(%i67) compare(x^2,0);
(%o67)                                  >=
(%i68) compare(1/(1+x)^2,0);
(%o68)                                   >
(%i69) compare(1/(1+x),0);
(%o69)                                   #
(%i70) compare(1/(1+%i*x),0);
(%o70)                             notcomparable
(%i71) compare(x,y);
(%o71)                                unknown
```

5.6 文脈

5.6.1 文脈の概要

文脈について

「**文脈 (context)**」は文を判断する上での基盤となる事実や仮定の集積です. 貴方が街で「あの人はイケメン/カワイイ」といった会話を耳にしたとき, その「あの人」が誰であるかはそれまでの会話の内容, あるいは, 話者が誰で, その時点で何をして, 何を見ているのかといった様々な情報が必要でしょう. そのような情報を抜きに, 自分がイケメン/カワイイと呼ばれたと一人で喜ぶ訳にはいきません. このことは 19 世紀末になって Frege が「**文は文脈上で解釈されるべき**」と主張したもので, このことを「**文脈原理**」と呼びます.

では数学の文脈とは何になるでしょうか？ たとえば, 4 の平方根は 2 になりますが, x^2 の平方根はどうでしょうか？ x は正か負の実数？ ひょっとすると複素数かもしれません. さらに「猫のタマ」のような文字列かもしれません. この変数 x に加えて「平方根って何？」といった言葉の意味や定義もあるでしょう. このように $\sqrt{x^2}$ を処理させることを考えるだけでも x に対して様々な情報が要求されます. これらは数学の問題を考察する上で必要となる, 定義, 公理, 定理, 仮定や事実が文脈に相当します. そして, Maxima には不完全ながらも文脈を実装しています.

さて, 先程の x^2 の平方根に戻って考えましょう. ここで, 仮定 1 を「x は実数である」, 仮定 2 を「$x \geq 0$ である」とします. 仮定 1 によって $\sqrt{x^2}$ は絶対値 $|x|$ と処理され, 仮定 2 によって x と処理されますが, これらの仮定の内容をもう少し吟味してみましょう. 仮定 1 の「x は実数である」は対象 x が帰属する類/クラスを指示しています. このように対象の帰属先を指示する性格の論理式を「**属性**」と呼びます. Maxima ではこのような属性は, 論理式ではなく declare 函数を用いるために大域的な影響が出ます. その一方で, 仮定 2 の「$x \geq 0$ である」は仮定 1 とは異なった性格で, 二つの対象 x と 0 との関係を述べています. そして, Maxima では二項間の大小関係を表現する論理式のみが文脈に登録可能な論理式となります.

文脈の指定

$\sqrt{x^2}$ の評価を行う際に, 変数 x の値域を '$x \geq 0$' と '$x < 0$' に分けると, 結果として x や $|x|$ が得られます. このように式の評価を行う上で仮定を切り替えることで, より的確な判断が可能となる場合も多々あります. この仮定の切替は文脈の切替に対応します.

Maxima での文脈の切替は, 大域変数 context に文脈名を束縛させることで行われます. たとえば, 文脈 A を '$x \geq 0$', 文脈 B を '$x < 0$' とすると, 仮定 '$x \geq 0$' で式の評価を行う場合には 'context:A' で文脈 A を指定し, 仮定 '$x < 0$' で評価を行いたければ 'context:B' で文脈 B を指定すればよいのです.

文脈の階層構造

このように問題に応じて文脈が切り替えられるということは非常に便利です. $\sqrt{x^2}$ の例では変数 x について文脈 A を '$x \geq 0$', 文脈 B を '$x < 0$' として必要に応じて切替えれば, その文脈上で解釈した結果が得られます. では, 考慮すべき変数が増えたり, その場合分けが増えるとどうなるでしょうか？ そこで, 式 $\sqrt{x^2y^2}$ の評価を考えてみましょう. 通常, この式の評価は次の 4 つの場合に分けて考察します:

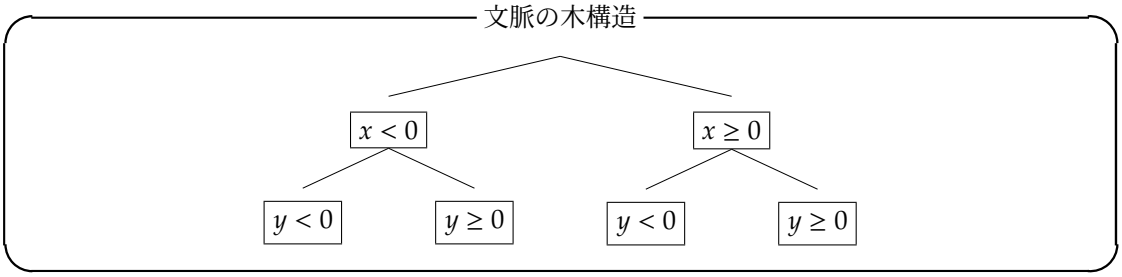
1. $x \geq 0 \wedge y \geq 0$
2. $x \geq 0 \wedge y < 0$
3. $x < 0 \wedge y \geq 0$
4. $x < 0 \wedge y < 0$

文脈もこの場合分けに応じた 4 種類の文脈が与えられます. すなわち, 変数が n 個, 各変数で考えられる場合の数が少なくとも 2 通りならば, 最低でも 2^n 個の文脈が発生するため, それだけの文脈を定義して評価を行うことも大変なことになります. しかし, 現実は, 全ての場合について評価を行わずに可能な限り少ない評価で目的に到達しようとするか, 全ての評価を行うにせよ, より見通しの良い方法を採用するでしょう.

この $\sqrt{x^2y^2}$ の例では, 幾ら文脈が 4 個とはいえ闇雲に評価することは危険です. そこで, 変数を一つ一つ評価させてみましょう. つまり, 第一段として変数 x について '$x \geq 0$' と '$x < 0$' に分けて評価し, 第二段として変数 y について '$y \geq 0$', '$y < 0$' と二段階の評価を行うのです:

1. $x \geq 0$ { $y \geq 0$ ⇒ 文脈 1 ; $y < 0$ ⇒ 文脈 2
2. $x < 0$ { $y \geq 0$ ⇒ 文脈 3 ; $y < 0$ ⇒ 文脈 4

ここで上の式の分類を 90 度回転させてみましょう:



このようにすることで文脈が木構造を持っていることが明瞭になります. 次に重要なことは, この木構造の根本側 (上位) の文脈の仮定は全て直系の下位の文脈に継承されることです. そのために Maxima の文脈も木構造を持っています. この木構造は親子関係に喩えると分り易くなります. まず, 根本側が先祖で枝の先側が子孫となります. そして, 文脈 A が文脈 B の一つ上の根本側の文脈

であれば, 文脈 A のことを「**親文脈**」, 文脈 B のことを「**子文脈**」と呼びます. そして親文脈の内容は子文脈に継承されます. したがって, 先祖にあたる文脈の内容はその子孫にあたる系列上の下位の文脈に継承されてゆくことになります. なお, Maxima の文脈に登録してある条件は, その条件を登録した文脈上でのみ facts 函数を使った確認が可能です.

5.6.2 文脈に登録可能な論理式

Maxima の文脈に登録可能な論理式は二項関係を表現した式に限定されます:

文脈に登録可能な論理式例

数学記号	Maxima の演算子/函数	例
=	equal	equal(x,y)
≠	notequal	notequal(x,y)
≥	>=	x >= y
>	>	x >y
≤	<=	x <= y
<	<	x <y

Maxima 内部では演算子 “<=” の演算子項と演算子 “<” の演算子項は被演算子の左右の入替えにより, それぞれが演算子 “>=” の演算子項と演算子 “>” の演算子項として文脈に登録されます. たとえば述語 ‘x <= y’ は述語 ‘y >= x’ , 述語 ‘x < y’ は述語 ’y > x’ として置換されて登録されます. また, 二つの対象の同値性と非同値性を示す論理式は, 演算子 “=” と演算子 “#” が文脈では使えないため, 同値性の表現が函数 equal, 非同値性の表現は函数 notequal を用いた論理式とする必要があります. なお, 演算子 “not” を用いた論理式は自動的に演算子 “not” を持たない同値な論理式で置換えられます:

演算子 “not” による演算子の変換

x >y	⇐(not)⇒	x <= y
equal(x,y)	⇐(not)⇒	notequal(x,y)

次に, これらの論理式の文脈の登録, 削除と確認を行う函数の解説をしましょう.

5.6.3 論理式の文脈への登録

Maxima の文脈上で論理式の操作を行う函数を次に纏めておきます:

文脈上の論理式に関連する函数

assume(⟨ 論理式$_1$⟩,⟨ 論理式$_2$⟩,...)
forget(⟨ 論理式$_1$⟩,...,⟨ 論理式$_n$⟩)
forget([⟨ 論理式$_1$⟩,...,⟨ 論理式$_n$⟩])
facts(⟨ 事項 ⟩)
facts(⟨ 文脈 ⟩)
facts()

assume 函数:　Maxima に事実や仮定を教える函数は assume 函数です. assume 函数によって論理式は現行の文脈の data 属性の属性値として蓄積されます.

assume 函数は適切でない式や論理式が入力された場合, assume 函数はエラー表示やリストを返します. たとえば, 演算子 “or” を含む論理式に対しては演算子 “or” を含む論理式の処理が出来ないと表示し, 同値演算子 “=” や非同値の演算子 “#” を含む論理式が入力された場合には equal 函数や notequal 函数を使うようにと指示します.

assume 函数は既に登録された論理式が入力された場合, リストの形式で対応する論理式に対して redundant を返し, 式が論理式ではない場合には meaningless を返し, 真理函数を含む論理式に対しては inconsistent を返します.

forget 函数:　文脈に設定した論理式を消去するときに用いる函数です. 削除したい論理式があれば, forget 函数の引数として論理式をそのまま記述するか, 論理式が割当てられている変数を指示します. 忘れさせる論理式が複数あれば, forget 函数に列として与えるか, リストとして与えます. この論理式が沢山になれば, forget 函数で消すよりも文脈の切替も視野に入れた方が良いでしょう.

facts 函数:　文脈に含まれている論理式や属性を表示するために用います. facts 函数は文脈を指定した場合, その文脈に含まれる論理式や属性を返し, 何も指定しない ‘facts()’ を入力した場合は現行の文脈が保持する論理式属性を全て表示します. また, declare 函数で与えられた属性は, その属性を定義した文脈を用いているときのみ表示されます.

文脈への論理式の登録例

これらの函数による実行例を示しておきます:

```
(%i1) assume(y>0)$
(%i2) assume(x>0,z<0)$
(%i3) facts();
(%o3)                          [y > 0, x > 0, 0 > z]
(%i4) sqrt(x^2);
(%o4)                                  x
(%i5) sqrt(z^2);
(%o5)                                 - z
```

```
(%i6) forget(z<0);
(%o6)                                [z < 0]
(%i7) sqrt(z^2);
(%o7)                                abs(z)
```

この例では, assum 函数を使って述語 'x >0', 'y >0' と 'z <0' を文脈に登録しています. そして, 文脈に登録した論理式全体と対象の属性は facts(); で表示ができます. ここで文脈内部では入力した論理式 'z <0' が論理式 '0 >z' で置換えられていることに注意して下さい. それから, 'sqrt(x^2)' と 'sqrt(z^2)' の評価にて, sqrt 函数は文脈から対象 x と対象 z の情報を入手して x と-z をそれぞれに返却しています.

登録した論理式の削除では forget 函数で論理式をそのまま引用します. もし, 変数に割当てられているのであれば変数名を指定することができます. この例では対象 z の情報が削除されたために sqrt(z^2 は abs(z) として簡易化されています.

5.6.4 文脈内部での属性と論理式の表現

今度は, 関係の論理式がどのように内部で表現されるかを観察してみましょう. そこで最初に論理式を調べ, 次に属性がどのように表現されるかを調べましょう:

論理式の内部表現

```
(%i1) assume(x1>y1);
(%o1)                               [x1 > y1]
(%i2) assume(x1<z1);
(%o2)                               [z1 > x1]
(%i3) assume(x1<=w1);
(%o3)                              [w1 >= x1]
(%i4) assume(equal(a1,a2));
(%o4)                          [equal(a1, a2)]
(%i5) assume(notequal(a1,a4));
(%o5)                       [notequal(a1, a4)]
(%i6) :lisp (get '$initial 'data)
(((MNQP $A1 $A4) CON $INITIAL) ((MEQP $A1 $A2) CON $INITIAL)
 ((MGQP $W1 $X1) CON $INITIAL) ((MGRP $Z1 $X1) CON $INITIAL)
 ((MGRP $X1 $Y1) CON $INITIAL))
```

assume 函数の返事から, 不等号を持つ論理式は被演算子の入れ替えを行った上で, 演算子 ">=" や演算子 "<" に置換されていることが判ります. それから, assume 函数に与えられた論理式は内部表現に置き換えられて LISP の putprop 函数によって文脈の data 属性の属性値リストに追加されます. これが文脈への論理式の登録の実態です.

具体的に解説しましょう. 論理式 'x1 <= w1' は被演算子の並換えにより 'w1 >= x1' で置換され. その内部表現は "((MGQP $W1 $X1) CON $INITIAL)" となります. ここで S 式の第 1 成分 "(MGQP $W1 $X1)" が入力した論理式に直接対応します. この第 1 成分が論理式 ' w1 >= x1' の内部表現と

比較して, 先頭の函数名に記号 “$” がなく, 函数名を括る小括弧 “()” もない平坦なリストであることに注意して下さい. このことから函数と異なる与件としての文脈の性格が伺えます. そして, S 式の第 2 成分が “CON”, 第 3 成分が文脈名 (Maxima 側から見ると文脈 initial) となっており, この S 式の性格と所属が読み取れるようになっています.

属性の内部表現

今度は declare 函数を用いて対象に幾つかの属性を指定し, それがどのように文脈に反映されるかを観察してみましょう:

```
(%i6) declare(x1,odd)$
(%i7) declare(a1,complex)$
(%i8) :lisp (get '$initial 'data)
(((KIND $A1 $COMPLEX) CON $INITIAL) ((KIND $X1 $ODD) CON $INITIAL)
 ((MNQP $A1 $A4) CON $INITIAL) ((MEQP $A1 $A2) CON $INITIAL)
 ((MGQP $W1 $X1) CON $INITIAL) ((MGRP $Z1 $X1) CON $INITIAL)
 ((MGRP $X1 $Y1) CON $INITIAL))
(%i8) facts();
(%o8) [x1 > y1, z1 > x1, w1 >= x1, equal(a1, a2), notequal(a1, a4),
                                        kind(x1, odd), kind(a1, complex)]
```

declare 函数で対象に属性を指定すると文脈に反映されます. このとき属性の内部表現は ‘facts()’ で出力されるリストと逆順のリストで, 第 1 成分が論理式, 第 2 成分が文脈を示す ‘CON’, そして第 3 成分が文脈名となります. ここで論理式の内部表現で ‘kind’ の項がありますが, これは対象が feature と呼ばれず属性を有することを示しています. ここでの例で, a1 は complex という属性を持ちますが, これを ‘kind(a1,complex)’ という函数項の形式で文脈に登録しているのです.

論理式と属性の内部表現

論理式や属性の内部表現を次に纏めておきましょう:

文脈内部での論理式と属性の表現

表現名	内部表現	対応する論理式表現	概要
kind	(kind x y)	kind(x,y)	帰属を表現
par	(par x y)	par(x,y)	二項関係を表現 (Maxima 側から定義出来ない)
mgrp	(mgrp x y)	x>y	大小関係 (>) を表現
mgpq	(mgpq x y)	x>=y	大小関係 (≥) を表現
meqp	(meqp x y)	equal(x,y)	同値関係を表現
mnqp	(mnqp x y)	notequal(x,y)	非同値関係を表現

文脈に登録される論理式は、Maxima の対象 a の概念 P への帰属を表現する ‘$a \in P$’ に対応する kind(a,P), 二つの概念 P, Q の包含関係 ‘$P \subset Q$’ を表現する “par(P,Q)”, そして, 同値性と非同値性に加えて大小関係が平坦なリストとして表現されます. ここで内部函数 par を利用者が設定することはできません. Maxima の初期状態では予め登録された事柄 (§5.4.8 参照) に限定されています.

自由に利用可能なのは内部函数 kind で設定可能な属性のみです. この kind で表現される属性は大域変数 features に登録されたもので, 利用者は declare 函数を用いて宣言することが可能です. そして大域変数 features に登録した属性を対象に付与する場合は declare 函数を用いて行います:

```
(%i1) features;
(%o1) [integer, noninteger, even, odd, rational, irrational, real, imaginary,
complex, analytic, increasing, decreasing, oddfun, evenfun, posfun, constant,
commutative, lassociative, rassociative, symmetric, antisymmetric,
integervalued]
(%i2) declare(cat,feature);
(%o2)                                done
(%i3) declare(dog,feature);
(%o3)                                done
(%i4) declare(tama,cat);
(%o4)                                done
(%i5) declare(mike,cat);
(%o5)                                done
(%i6) declare(pochi,dog);
(%o6)                                done
(%i7) features;
(%o7) [integer, noninteger, even, odd, rational, irrational, real, imaginary,
complex, analytic, increasing, decreasing, oddfun, evenfun, posfun, constant,
commutative, lassociative, rassociative, symmetric, antisymmetric,
integervalued, cat, dog]
(%i8) facts();
(%o8)          [kind(tama, cat), kind(mike, cat), kind(pochi, dog)]
```

5.6.5 文脈を用いた推論

文脈を用いた論理式の判断では, 文脈に登録された論理式表現を Modus Ponens(MP) (§4.19 参照) を用いて推論を行うこと, より具体的には論理式表現のリストの結合処理で行います.

たとえば, 条件として '$x_1 > x_2'$, '$x_2 > x_3'$, ..., '$x_{n-1} > x_n$' を文脈に与えたときの '$x_i > x_j$' の判断は x_i から出発して文脈に登録された演算子 ">" に対応する "mgrp" を持つ論理式表現を用いて x_j に到達出来るかどうかで行います. そこで, 最初に x_i を第 2 項に持つ mgrp の論理式を文脈の data 属性値の論理式リストの先頭から順番に捜します. このときに論理式 '(`mgrp $x_i $x_{i_1}`)' が該当する論理式であれば, その第 3 項の x_{i_i} が x_j と一致するかを確認します. ここで一致すれば真であると判断しますが, 一致しなければ, 今度は '$x_{i_1} > x_j$' として x_{i_1} に対して該当する論理式を捜します. あとは同様に処理を行って項の列 $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ が得られますが, この列が最終的に x_j に到達できれば 'true' と判断します.

この操作は '$x_i > x'_{i_1}$, '$x_{i_1} > x'_{i_2}, \ldots,$ '$x_{i_{k-1}} > x_{i_k}$' に MP を適用して最終的に論理式 '$x_i > x_{i_k} \wedge x_{i_k} = x_j$' を得ることに対応しています. もし, x_j に到達出来なければ, 再び, x_i を第 2 項に持つ "mgrp" の論理式を検出して同様の処理を行います. それでも x_i に到達出来なければ大小関係を演算子 "<" に反転させて, x_i から x_k に到達出来るかを試みます. ここで関係 "<" に対応するのが内部函数 dlsf です. これに成功すると偽 'false' と判断しますが, こちらでも最終的に到達出来なければ判断不能

'unknown' と判断します. このとき大域変数 prederror が 'false' であれば判断不能を 'unknown' として返し, それ以外の場合は 'false' やエラーを返します.

この処理をまとめておきます. 文脈に登録された論理式を $P_1, \ldots, P_n$, 真偽を判定すべき論理式を G とするとき, $P_1 \wedge \cdots \wedge P_n \wedge G$ を解釈していることになり, $P_1 \wedge \cdots \wedge P_n \wedge G$ が真であれば, G を真と判定し, 偽であれば, $P_1 \wedge \cdots \wedge P_n \wedge \neg G$ の解釈に移ります. この論理式が真であれば G が 偽であると判断し, そうでなければ解釈不能としているのです. このような手順を踏んで Maxima は文脈上で論理式 G の解釈を行っているのです.

それでは具体的な例を次に示しておきましょう:

大小関係の推論 (その 1)　次に 'x1>x2>...>x6>x7' を仮定して 'x2>x5' を判断させます:

```
(%i1) assume(x1>x2,x2>x3,x3>x4,x4>x5,x5>x6,x6>x7);
(%o1)         [x1 > x2, x2 > x3, x3 > x4, x4 > x5, x5 > x6, x6 > x7]
(%i2) :lisp  (trace dlqf dgqf dlsf dgrf deq dcompare)
(DLQF DGQF DLSF DGRF DEQ DCOMPARE)
(%i2) is(x2>x5);
  0: (DCOMPARE $X2 $X5)
    1: (DEQ $X2 $X5)
      2: (DGRF $X3 $X5 ((MGRP $X3 $X4) CON $INITIAL))
        3: (DGRF $X4 $X5 ((MGRP $X4 $X5) CON $INITIAL))
        3: DGRF returned T
      2: DGRF returned T
    1: DEQ returned T
  0: DCOMPARE returned $POS
(%o2)                                  true
```

この判断の様子を見るために LISP の trace 函数を使って内部函数 dlqf, dgqf, dlsf, deq と dcompare の動作を追跡します. そこで ":lisp" を使って Maxima の土台の LISP に処理させる S 式を評価させています. ここでの判断では内部函数 deqf と内部函数 dgrf が用いられ, dgrf が大小関係の ">" に対応する推論処理を行っていることが分ります. この処理の流れは 'x2>x5' を示すために, 変数 x5 を内部函数 deq の第 2 引数に固定して第 1 引数を動かして行きます. この動かし方は, 論理式の内部表現に対し, その第 2 成分が変数 x2 となる文脈リストの第 3 成分を順番に取り出します. この例では開始項が変数 x2 なので, 変数 x2 に関連する文脈を構成する論理式として 'x1>x2' と 'x2>x3' の二つがありますが, 最初に第 2 成分が変数 x2 の論理式 'x2>x3' を採用するので, プロンプト 3 では変数 x3 を使って比較を行います. そこで変数 x3 を論理式の内部表現の第 2 成分として持つ文脈を構成する論理式を同様に文脈リストの先頭から取り出しますが, 論理式 'x3>x4' が対応するので, 次のプロンプト 4 では変数 x4 と変数 x5 と比較になります. すると, この比較では文脈に論理式 'x4>x5' があるので 'T' となり, 以降, 'x3>x4' と併せることで 'x3>x5' も 'T' となります. それから, 内部函数 dcompare 函数が 'pos' を返しています. このことから分るように Maxima で 'x>y' の判断は式 'x - y' の正値性で判断します. この判断では上記の結果と文脈の 'x2>x3' を併せることで 'x2 - x5' の正値性 ('$POS') を示されるので 'x2>x5' と結論付けるという流れになっています.

大小関係の推論 (その 2)　同じ文脈上で今度は 'x5>x2' の判断をさせてみましょう:

```
(%i3) is(x5>x2);
  0: (DCOMPARE $X5 $X2)
    1: (DEQ $X5 $X2)
      2: (DGRF $X6 $X2 ((MGRP $X6 $X7) CON $INITIAL))
        3: (DGRF $X7 $X2 ((MGRP $X6 $X7) CON $INITIAL))
        3: DGRF returned NIL
      2: DGRF returned NIL
      2: (DGRF $X6 $X2 ((MGRP $X5 $X6) CON $INITIAL))
      2: DGRF returned NIL
      2: (DLSF $X4 $X2 ((MGRP $X4 $X5) CON $INITIAL))
      2: DLSF returned NIL
      2: (DLSF $X4 $X2 ((MGRP $X3 $X4) CON $INITIAL))
        3: (DLSF $X3 $X2 ((MGRP $X3 $X4) CON $INITIAL))
        3: DLSF returned NIL
        3: (DLSF $X3 $X2 ((MGRP $X2 $X3) CON $INITIAL))
        3: DLSF returned T
      2: DLSF returned T
    1: DEQ returned T
  0: DCOMPARE returned $NEG
(%o3)                                false
```

この例は否定的な結論を得る推論の流れになります. ここでは文脈の論理式を辿ることでプロンプト 4 の 'x7>x2' で限界に到達するために 'NIL' を dgrf 函数が返却します. その結果を受けて 'x6>x2' の比較も 'NIL' を返しています. そこから逆に演算子 ">" に対応する dgrf 函数から演算子 "<" に対応する内部函数 dlsf に切り替えて推論を行います. dlsf の場合は dgrf の逆で, 文脈に登録した論理式の内部表現の第 3 成分を指定し, その論理式の内部表現の第 2 成分を取り出します. 二番目のプロンプト 3 で 'x4>x5' の x4 を採用して, 以降同様に進めて行くことで最終的に 'x5 - x2' が負 ('$NEG') であると判定し, そこから 'false' と結論付けています.

対象の同値性の推論　では対象の同値性の推論はどのようにしているのでしょうか？ これは大小関係と同様に差を取って符号の判定を行う手法ですが, 文脈から項の同値性に関する論理式を与式に ratsubst 函数を用いて代入し, 与式が '0' になるかどうかで判定しています:

```
(%i1) assume(equal(x1,x2),equal(x2,x3),equal(x3,x4),equal(x4,x5));
(%o1)     [equal(x1, x2), equal(x2, x3), equal(x3, x4), equal(x4, x5)]
(%i2) :lisp (trace alike $ratsubst)

(ALIKE $RATSUBST)
(%i2) is(equal(x1,x5));
  0: (ALIKE ($X1) ($X5))
  0: ALIKE returned NIL
  0: (ALIKE ($X3 $X4) ($X4 $X5))
  0: ALIKE returned NIL
  0: (ALIKE ($X2 $X3) ($X3 $X4))
  0: ALIKE returned NIL
  0: (ALIKE ($X2 $X3) ($X4 $X5))
  0: ALIKE returned NIL
  0: (ALIKE ($X1 $X2) ($X2 $X3))
  0: ALIKE returned NIL
  0: (ALIKE ($X1 $X2) ($X3 $X4))
```

```
  0: ALIKE returned NIL
  0: (ALIKE ($X1 $X2) ($X4 $X5))
  0: ALIKE returned NIL
  0: ($RATSUBST $X2 $X1 ((MPLUS SIMP) $X1 ((MTIMES SIMP RATSIMP) -1 $X5)))
    1: (ALIKE ($X2 $X1 $X5) ($X2 $X1 $X5))
    1: ALIKE returned T
  0: $RATSUBST returned ((MPLUS RATSIMP) ((MTIMES RATSIMP) -1 $X5) $X2)
  0: ($RATSUBST $X3 $X2 ((MPLUS RATSIMP) ((MTIMES RATSIMP) -1 $X5) $X2))
    1: (ALIKE ($X3 $X2 $X5) ($X3 $X2 $X5))
    1: ALIKE returned T
  0: $RATSUBST returned ((MPLUS RATSIMP) ((MTIMES RATSIMP) -1 $X5) $X3)
  0: ($RATSUBST $X4 $X3 ((MPLUS RATSIMP) ((MTIMES RATSIMP) -1 $X5) $X3))
    1: (ALIKE ($X4 $X3 $X5) ($X4 $X3 $X5))
    1: ALIKE returned T
  0: $RATSUBST returned ((MPLUS RATSIMP) ((MTIMES RATSIMP) -1 $X5) $X4)
  0: ($RATSUBST $X5 $X4 ((MPLUS RATSIMP) ((MTIMES RATSIMP) -1 $X5) $X4))
    1: (ALIKE ($X4 $X5) ($X4 $X5))
    1: ALIKE returned T
  0: $RATSUBST returned 0
(%o2)                                  true
```

Maxima は assume 函数によって論理式が文脈に登録される時点で登録しようとする論理式から矛盾や余剰なものでないことをこれに似た方法で自動的に検証します. もし, 登録しようとする論理式の検証結果が 'unknown', すなわち, 文脈上で矛盾が導かれず, さらに文脈で既知の事実から導かれない場合に登録し, そうでなければ登録を行いません:

```
(%i1) assume(equal(x1,x2),equal(x2,x3),equal(x3,x4),equal(x4,x5));
(%o1)     [equal(x1, x2), equal(x2, x3), equal(x3, x4), equal(x4, x5)]
(%i2) assume(equal(x1,x5));
(%o2)                                [redundant]
(%i3) assume(x1>x5);
(%o3)                              [inconsistent]
(%i4) facts();
(%o4)     [equal(x1, x2), equal(x2, x3), equal(x3, x4), equal(x4, x5)]
```

この例では新たに論理式 'equal(x1,x5)' を文脈に登録しようとしていますが, この論理式は前の論理式から導出されるために 'redundant' として判断され、新規に登録されません. また, 論理式 'x1>x5' は変数 x1 と変数 x5 の同値性と矛盾するために 'inconsistent' と判断され, この論理式も登録されません.

5.6.6　文脈の階層

Maxima の文脈には前述のように階層構造が入ります. この階層構造は文脈 global を最上層 (根源) とし, 文脈 global の一つ下位の文脈として文脈 initial があらかじめ用意されています:

Maxima の起動時に存在する文脈

文脈名	概要
initial	Maxima を立ち上げた時点で利用される文脈名.
global	Maxima の最上位の文脈.

文脈 global が Maxima で最上層の文脈で, この文脈 global には定数の値, 函数の属性といった最も基本的な事柄が登録され, Maxima の全ての文脈に継承されます:

```
(%i8) facts(global);
(%o8) [par(even, integer), kind(imaginary, complex), kind(%i, noninteger),
kind(%i, imaginary), kind(%e, noninteger), kind(%e, real),
kind(%pi, noninteger), kind(%pi, real), kind(%gamma, noninteger),
kind(%gamma, real), kind(%phi, noninteger), kind(%phi, real),
kind(log, increasing), kind(atan, increasing), kind(atan, oddfun),
kind(delta, evenfun), kind(sinh, increasing), kind(sinh, oddfun),
kind(cosh, posfun), kind(tanh, increasing), kind(tanh, oddfun),
kind(coth, oddfun), kind(csch, oddfun), kind(sech, posfun),
kind(asinh, increasing), kind(asinh, oddfun), kind(acosh, increasing),
kind(atanh, increasing), kind(atanh, oddfun), kind(li, complex),
kind(lambert_w, complex), kind(cabs, complex), kind(conjugate, complex),
kind(erf_generalized, antisymmetric), kind(beta, symmetric),
kind(%pi, constant), kind(%i, constant), kind(%e, constant),
kind(%phi, constant), kind(i, constant), kind(%gamma, constant),
kind(inf, constant), kind(minf, constant), kind(und, constant),
kind(ind, constant), kind(infinity, constant), kind(true, constant),
kind(false, constant), equal(%e, 2.718281828459045),
equal(%pi, 3.141592653589793), equal(%phi, 1.618033988749895),
equal(%gamma, .5772156649015329)]
```

文脈 global に与えられた定数や函数の属性は文脈 initial, および Maxima で生成される全ての文脈に継承されます. このことから文脈 global は系全体に関わる事項を登録すべき文脈で, 日常的な利用は文脈 global よりも下位の文脈を用いるべきです. この文脈 global に対し, 文脈 initial は文脈 global の直下の文脈で, 利用者は大域変数 context を使って文脈の切換えを行わない限り文脈 initial 上で処理を行うことになります.

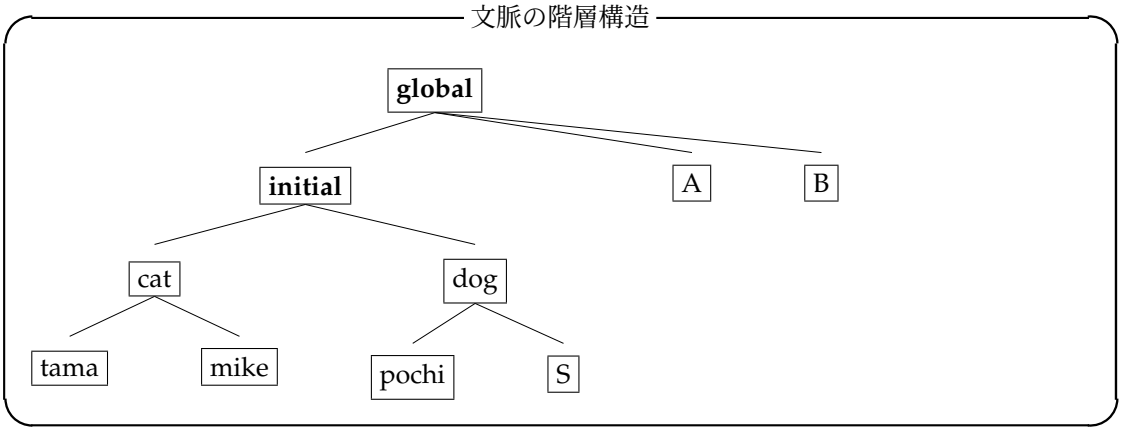
newcontext 函数を用いて新しく生成される文脈は, 大域変数 context で指示される既存の文脈の直下に置かれます. また, supcontext 函数を用いた場合, 新しく生成される文脈は supcontext 函数の第 2 引数で指示する既存の文脈の直下で生成されます. そして, 上位にある文脈の内容は下位の文脈に継承されます. そこで簡単のために文脈の階層を親子関係に喩えて解説します. まず, 文脈 A が文脈 B の直下にある文脈であるとします. このとき文脈 B の内容は文脈 A に継承されます. したがって, 上位にある文脈 B が文脈 A の**親**, に相当するので**親文脈**と呼びます. これに対し文脈 A は文脈 B の**子**に対応するので**子文脈**と呼びます. 具体的には, 文脈 initial は文脈 global の一つ下の階層の文脈となるので, 文脈 initial は文脈 global の子文脈, 文脈 global が文脈 initial の親文脈となります.

この文脈の階層構造の表現は LISP の属性を用いて表現されます. 具体的には文脈に与えた属性 subc の属性値として親文脈名を付与することで表現しています. このことを文脈 global の下位の文脈である文脈 initial で確認しておきましょう:

```
(%i7) :lisp (get '$initial 'subc);

($GLOBAL)
```

この例に示すように文脈 initial の subc 属性値から initial の上位の文脈, すなわち親文脈が文脈 global であることが分ります. このことから各文脈の subc 属性値を調べ, 親文脈を子文脈の一階層上に表記し, この親子関係にある文脈同士を線分で結ぶことで文脈の階層を可視化することができます：



この例で, 文脈 cat は文脈 initial の一つ下の文脈なので, 文脈 cat は文脈 initial の子文脈となります. そして, 文脈 cat は間に文脈 initial を挟んで文脈 global と繋がるために, 文脈 global に対しては孫に相当します. そして, 大域変数 context で指示された文脈の先祖にあたる文脈に登録された論理式は大域変数 context で指示された文脈に継承されます. たとえは, 文脈 tama では文脈 global, 文脈 initial と文脈 cat に登録された論理式が継承されます. しかし, 文脈 S では文脈 cat やその子孫の文脈の内容は当然継承されません. このように文脈は決定問題で用いられる「**意味の木**」(semantic tree) に似た側面を持ちます.

文脈操作函数

次に文脈操作函数の一覧を示します:

文脈操作函数

newcontext(〈 文脈 〉)
supcontext(〈 親文脈 〉, 〈 子文脈 〉)
killcontext(〈 文脈$_1$〉, . . .)
activate(〈 文脈$_1$〉, . . .)
deactivate(〈 文脈$_1$〉, . . .)

ここで挙げた函数は文脈の「生成」,「削除」と「借用」に関連するものです. 文脈の生成は newcontext 函数と supcontext 函数で行い, 削除は kill 函数で行います. さて, 借用の意味ですが, これはある系統の文脈から別の系統の文脈の内容を借用することです. 借用は activate 函数を用い, deactivate 函数で借用の解除を行います.

newcontext 函数: 新しい文脈の生成を行う函数です. 生成された文脈は newcontext 函数を実行した文脈の子文脈となります. そして自動的に大域変数 context に生成された文脈名が割当てられます.

supcontext 函数: 既存の文脈の下位の文脈を生成する函数です. newcontext 函数と supcontext 函数で新規に生成された文脈は, 後述の大域変数 contexts に割当てられたリストに登録されます.

activate 函数: 先祖-子孫関係にない別の系統の文脈を借用する歳に用いる函数です. activate 函数で借用する文脈は大域変数 activecontexts に割り当てられたリストに登録されます.

deactivate 函数: 借用した文脈を切離す函数で, 大域変数 activecontexts に割り当てられたリストから指定した文脈名を削除します. なお, 大域変数 context に割当てられた文脈は deactivate 函数で切離すことはできません.

killcontext 函数: 文脈の削除を行う函数です. この killcontext 函数で削除された文脈は大域変数 contexts からも削除されますが, 大域変数 context で指示された文脈や大域変数 activecontexts に割り当てられたリストに包含される文脈は利用中の文脈であるために killcontext 函数による削除が行えません.

これらの文脈操作函数の具体例を示します:

```
(%i1) supcontext(ペット,initial)$
(%i2) supcontext(猫,ペット)$
(%i3) supcontext(性格,猫)$
(%i4) assume(equal(みけ,穏やか))$
(%i5) assume(equal(たま,不思議と目立たない))$
(%i6) assume(equal(とら,名前どおりヤンチャ))$
(%i7) facts();
(%o7)     [equal(みけ, 穏やか), equal(たま, 不思議と目立たない), equal(とら, 名前
       どおりヤンチャ)]
(%i8) :lisp (get '$猫 'subc)
($ペット)
(%i8) :lisp (get '$性格 'subc)
($猫)
(%i8) :lisp (get '$性格 'data)
(((MEQP $とら $名前どおりヤンチャ) CON $性格)
 ((MEQP $たま $不思議と目立たない) CON $性格) ((MEQP $みけ $穏やか) CON $性格))
```

この例では,supcontext 函数を使って, 文脈 initial の子文脈「ペット」, 文脈「ベット」の子文脈「猫」と文脈「猫」の子文脈「性格」を生成しています. それから, assume 函数を用いて論理式を文脈「性格」に登録しています. ここで文脈に登録した論理式は facts 函数を用いて表示可能です. さらに, 演算子 ":lisp" を用いて LISP の get 函数で, 文脈「猫」や文脈「性格」の subc 属性値が文脈「ペット」であることを示し, 最後に文脈「性格」の data 属性に登録した属性がリストとして登録されていることを示しています.

次に借用の例を示しておきましょう:

```
(%i1) supcontext(X0,initial);
(%o1)                                X0
(%i2) assume(x>0);
(%o2)                              [x > 0]
(%i3) assume(>0);
(%o3)                              [ > 0]
(%i4) supcontext(Y0,initial);
(%o4)                                Y0
(%i5) assume(y>0);
(%o5)                              [y > 0]
(%i6) supcontext(X1,X0);
(%o6)                                X1
(%i7) assume(x>1);
(%o7)                              [x > 1]
(%i8) supcontext(Y1,Y0);
(%o8)                                Y1
(%i9) assume(y>1);
(%o9)                              [y > 1]
(%i10) context;
(%o10)                               Y1
(%i11) activate(X1);
(%o11)                             done
(%i12) sqrt(x^2);
(%o12)                               x
(%i13) sqrt((x-1)^2);
(%o13)                             x - 1
(%i14) sqrt(z^2);
(%o14)                               z
```

この例では X0 の系列と Y0 系列の二つの文脈の系列があります. ここで文脈 Y1 で文脈 X1 の借用を activate 函数を用いて行っています. この文脈 Y0 の系統では一切, 変数 x や変数 z に関する言及がありませんが, 文脈 X1 の借用の結果, 文脈 X1 が継承した内容が文脈 Y1 から参照できていることに注意して下さい.

5.6.7　文脈の指定に関連する大域変数

文脈を必要に応じて生成して行くにつれて, 現在利用中の文脈が何であるか, あるいは他にどのような文脈があるのか判らなくなるかもしれません. Maxima の大域変数 context に利用中の文脈名が割当てられ, 大域変数 contexts に全ての文脈名が登録されたリストが割当てられています:

文脈の指定に関連する大域変数

変数名	初期値	概要
context	initial	現行の文脈名が割当てられた変数
contexts	[initial,global]	Maxima 内部の文脈名リストが割当てられた変数
activecontexts	[]	active 函数で有効にされた文脈を成分とするリストが割当てられた変数

大域変数 context: 式の判断で用いられる文脈名が割当てられています. また, 文脈を切換える必要が出た場合に切換える文脈名を指定することで, 文脈を指示した文脈に切換ることができます. ここで大域変数 context に指定した文脈が既存の文脈でなければ, Maxima は newcontext 函数を用いて context に指定した文脈名の文脈を新規に生成します.

大域変数 contexts: Maxima に存在する文脈の名前を成分とするリストが割当てられている大域変数です.

大域変数 activecontexts: activate 函数によって借用された文脈名で構成されたリストが割当てられています. このリストに登録された文脈に対して deactivate 函数を作用させると, その文脈名が大域変数 activecontexts のリストから削除されます.

では, 以下に文脈の使い方の実例を示しましょう:

```
(%i1) contexts;
(%o1)                          [initial, global]
(%i2) context;
(%o2)                               initial
(%i3) newcontext(mike);
(%o3)                                mike
(%i4) supcontext(neko,mike);
(%o4)                                neko
(%i5) context;
(%o5)                                neko
```

この例では最初に立ち上げた時点で Maxima が持っている文脈を大域変数 contexts を使って表示させています. 次に context 函数を使って最初に用いている文脈が文脈 initial であることを示しています. それから newcontext 函数を使って新しい文脈 mike を生成し, 文脈 mike の親文脈となる文脈 neko を今度は supcontext 函数を使って生成しています. この supcontext 函数では親文脈に既存の文脈を指定しなければなりません.

5.6.8 変数の正値性に関連する大域変数

変数の正値性に関連する大域変数

変数名	初期値	概要
assume_pos	false	assume_pos_pred で指定した真理函数が 'true' となる対象を正値とするためのフラグ
assume_pos_pred	false	真理函数を指定

大域変数 assume_pos と大域変数 assume_pos_pred は対で用いる大域変数で, 引数の正値性の評価で用いられます. 仕組は内部処理で式の正値性の判定で用いられる内部の LISP 函数 sign-any が大域変数 assume_pos が true の時にのみに大域変数 assume_pos_pred に指定した真理函数を用いて引数の評価を行います. この sign-any 函数は大域変数 assume_pos_pred に指定した真理函数が

true を返す場合に内部変数 sign に値として pos を割当てます. ただし, assume 函数による設定が大域変数 assume_pos よりも優先されます:

```
(%i13) declare(aa,even)$
(%i14) featurep(aa,even);
(%o14)                                true
(%i15) assume_pos_pred:lambda([x],featurep(x,even));
(%o15)                    lambda([x], featurep(x, even))
(%i16) assume_pos:true$
(%i17) sqrt(aa^2);
(%o17)                                 aa
(%i18) sqrt(bb^2);
(%o18)                               abs(bb)
```

この例では変数 aa に偶数としての属性 even を与えています. 実際, featurep 函数による検査では 'true' になります. ここで大域変数 assume_pos_pred に featurep 函数による検査を行う函数を割当てます. それから, 大域変数 assume_pos を true に設定すると, 偶数としての属性を持つ変数 aa には正値性が付与されます. そのために 'sqrt(aa^2)' は 'aa' になりますが偶数の属性を持たない変数 bb に対して 'sqrt(bb^2)' は 'abs(bb)' となります.

5.7 規則と式の並びについて

5.7.1 規則の概要

Maxima では, 式を項で纏めたり展開することで式を変形するだけでなく, 「**式の並び**」と呼ばれるある特徴を持った部分式に対してのみ変換を行う「**規則**」を設定することができます. この規則を用いた処理は与式の項を別の指定された式で置換えるもので代入の一種とも言えますが, 代入では項や部分式を直接指定して入れ替えを実行するのに対し, 規則を用いた処理は, 人間が与式から既存の公式をあてはめて式の変換を行うように, 雛形となる式と処理を行うべき項の属性を式の並びとしてあらかじめ定義し, その属性を持った項 (=適合する項) が式中に含まれている場合のみに項の置換を実行する処理です.

より正確にこの処理を説明しておきましょう. まず, Maxima の規則は述語 P と函数 f_P で構成されています. このとき 「**式の並びの照合**」という処理は, Maxima の式 E を函数 $E(x_1,\ldots,x_n)$ の変数 $x_1,\ldots,x_n$ が項 $t_1,\ldots,t_n$ で代入されたものと見做したときに, この項の列 $t_1,\ldots,t_n$ に述語 P を作用させた列 $P(t_1),\ldots,P(t_n)$ を計算することに対応します. そして, $P(t_i)$ が真であれば項 t_i を「述語 P に適合する」, 偽ならば「述語 P に適合しない」と呼びます. そして, 論理式 $P(t_i)$ が真の場合, すなわち, 項 t_i が述語 P に適合する場合に項 t_i を函数 $f_p(t_i)$ で置換えることが「**規則の適用**」になります. したがって, $t_1,\ldots,t_l$ が述語 P に適合する項, $t_{l+i},\ldots,t_n$ を適合しない項とすると, 「規則の適用」により, 本来の式 $E(t_1,\ldots,t_n)$ から新しい式 $E(f_p(t_1),\ldots,f_p(t_l),t_{l+1},\ldots,t_n)$ が得られます.

では, 述語と変換函数はどのように設定すべきなのでしょうか？ そこで, 微分演算子 D の定義を例にちょっと考えてみましょう.

5.7.2 述語と変換函数の定義

規則と並びの照合

まず, 前置式表現の演算子 D が微分演算子と呼ばれるためには次の条件を満さなければなりません:

微分演算子の持つべき性質

☆線形性: $D(a+b) = Da + Db$
☆ Leibniz 則: $D(ab) = (Da)b + a(Db)$

Maxima で前置式表現の演算子 D を微分演算子として使うためには, 線形性と Leibniz 則の二つを演算子 D に与えてやらなければなりません. この性質を全て規則として与えることも考えられなくもありませんが, 一般的な演算子の性質の多くは Maxima の属性として予め定義されています. 実際, 線形性は linear 属性を演算子 D に declare 函数を用いて付加すればよいのです (§5.4.8). ところが, Leibniz 則はありません. したがって, Leibniz 則を規則で表現する必要があるのです. このように規則を定義する前に, Maxima に使えそうな属性があるかどうかを確認し, そのような属性がない場合に規則を用いることを考慮すべきです.

この Leibniz 則は, 二つの Maxima の式 a と b の積 ab に演算子 D を作用させると $(Da)b + a(Db)$ を得るという公式 (変換式) なので, この規則を ‘Leibniz : $D(a\ b) \to (D\ a)\ b + a\ (D\ b)$’ と表記しましょう. このとき, ab のように規則を適用すべき式のことを「**式の並び (パターン)**」と呼びます. そして, 式が与えられたときに, この式の並びに合致する項を検出する操作を「**並びの照合 (パターンマッチング)**」と呼びましょう.

並びの照合と述語

Maxima の規則は述語とそれに対応する変換函数で構成されていると述べました. 人間が公式を適用する場合, 適用可能な式の並びを検出して公式をあてはめます. だから, 雛形さえ与えてしまえば良さそうに見えますが, どうして述語が必要なのでしょうか？ この点について考えてみましょう. ここで演算子 D を変数 x に対する微分 $\dfrac{d}{dx}$ としましょう. このときに演算子 D を作用させる式が x の函数でなければ 0 になりますが, 変換式 ‘$D\ (a\ b) \to (D\ a)\ b + a\ (D\ b)$’ だけしかなければ, 次に述べる困ったことが生じます.
まず, $1 = 1 \cdot 1$ という関係がありますね. この関係を 1 に対して $1 \cdot 1$ で置換えるという関係で, 式 ‘$\dfrac{d1}{dx}$’ の計算で利用してしまうとどうなるでしょうか？ ここで ‘$\dfrac{d1}{dx}$’ は ‘$\dfrac{d(1 \cdot 1)}{dx}$’ になりますが, Leibniz 則を何も考えずに自動的に適用すると ‘$\dfrac{d1}{dx} \cdot 1 + 1 \cdot \dfrac{d1}{dx}$’ となり, 演算項 ‘$\dfrac{d1}{dx}$’ が前よりも一つ増えてしまいます. 以降, 同様の処理を行えは同じ演算子項が増えるばかりで, 一向に計算が終らなくなります. そこで述語として
$P(x) \stackrel{def}{=}$‘x は定数ではない’ を与えて適用する式の被演算子で照合を行えば, この無意味な Leibnitz 則の適用が回避されます.
このような無限ループを防止することの他に実用上の問題もあります. 実際, 規則を定義していても, 式に規則を適用すべき項や式が埋れているので, 与えられた式を検査する必要があります. たとえば, 函数 f を周期 ω_0 を持つ周期函数とし, ‘$\forall n \in \mathbb{Z}\ (f(x + n\omega_0) \to f(x))$’ といった規則を入れたとします. そこで, $f(a_0)$ と $f(a)$ の関係を調べる必要が出た場合, ‘$\exists n_0 \in \mathbb{Z}\ (a = a_0 + n_0\omega_0)$’ を満すかどうかを検査する必要がありますね. これらの理由から規則を述語と変換函数の対として考える必要があるのです.

規則の定義方法

Maxima の規則の設定には三種類の方法があります:

1. tellsimp 函数と tellsimpafter 函数を用いる方法
2. defrule 函数を用いる方法
3. let 函数を用いる方法

まず, tellsimp 函数と tellsimpafter 函数を用いると, Maxima は入力された Maxima の式が条件に合致する場合に自動的に規則の適用を行います.

それに対し, defrule 函数や let 函数を用いると, 式の入力と同時に規則の適用が自動的に実行されず, 別途, 規則の適用を行う函数を用いなければなりません. さらに, defrule 函数で個別の規則を定義すると, 規則を適用する函数は, その規則を帰納的に式に作用させて式の変換を行います.
ここで let 函数を用いると, 複数の規則を含むパッケージが構築され, let 函数で定義した規則を適用する函数は, このパッケージに含まれる規 則を順番に処理して式の変換を行ないます.
このように定義した規則を実際の式に適用する函数, 規則の表示, および, 削除を行う函数は系統毎に異なったものになり, これらの規則に互換性がないことにも注意が必要です.
この節では規則を適用する式の並びを指定する変数に与える述語の説明を最初に行い, そののちに defrule 函数, let 函数, tellsimp 函数と tellafter 函数による規則の定義と適用について解説します.

5.7.3 述語と変数の指定

Maxima では指定した式の並びに対して規則の変換を行いますが, 一般の式では, 指定した式の並びが埋没しているために規則が適用されるべき式であるかどうかを判別する必要があります. さらに規則を適用すべき式かどうかを判断させることで, 処理の効率化や間違った処理を防ぐ必要もあります.
そこで, 規則を適用すべき式の並びであるかどうかは, 式の並びと類似した式が与式に項として包含され, その部分項を構成する函数, あるいは変数等が, 式の並びで用いた函数や変数と何らかの意味で一致するという判断を行わなければなりません.
この判断は「式の並び」が与式の「項」に対応し, さらに, その項を構成する変数が式の並びで用いた変数と対応するという関係か成立するかどうかで行えます. ここで, 最後の関係の判断は変数に関する述語として与えられますが, Maxima ではこの述語を matchdeclare 函数を用いて表現します.

matchdeclare 函数: この函数は変数とその変数に対応する述語を一組として定義します:

matchdeclare 函数
matchdeclare(⟨ 並びの変数$_1$⟩,⟨ 述語$_1$⟩,...,⟨ 並びの変数$_n$⟩,⟨ 述語$_n$⟩)
matchdeclare([⟨ 並びの変数$_1$⟩ , ...,⟨ 並びの変数$_n$⟩],⟨ 述語 ⟩)

matchdeclare 函数の引数は「並びの指定」で用いる「並びの変数」と, その「並びの変数」に対応する「真理函数」の対を引数とします. したがって, 引数の個数は常に偶数個となるので, 函数の引数が偶数個でなければエラーを返す仕様となっています.
ここで「並びの変数」に対応する述語として与えられる式は Maxima の真理値函数, 真理値を返す lambda 式や block 函数を含まない Maxima の文となります. さらに「並びの変数」に制限を追加する必要がなければ述語として true や all も指定できます.
なお, この述語は「並びの変数」の matchdeclare 属性の属性値となります.
たとえば, 「式の並び」で用いる変数_a に対し, 定数であるかどうかを判断する constantp 函数 を結び付けた場合を示しましょう:

```
(%i30) matchdeclare(_a,constantp);
(%o30)                                done
(%i31) properties(_a);
```

```
(%o31)                                  [matchdeclare]
(%i32) printprops(_a,matchdeclare);
(%o32)                                  [constantp(_a)]
```

この例では「並びの変数」は_a, 対応する真理函数は, 定数属性 constant を持つ対象に対して 'true' を返す constantp 函数としています. このとき変数に指定された属性を返す函数 properties を使って変数の属性値を調べると, 変数_a には matchdeclare 属性が付与されていることが判ります. さらに printprops 函数で_a の matchdeclare 属性値を表示させると, constantp(_a) が返却されていますね. このように matchdeclare 属性値として「並びの変数」が満すべき述語が付加されているのです.

今度は matchdeclare 函数による constantp の処理に注目してみましょう. この例では matchdeclare 属性値として constantp(_a) が付与されています. この処理をより一般化して解説しましょう. そこで,「並びの変数」が_a, 対応する真理函数が $P(x_1,\ldots,x_n,\_a)$ であれば, matchdeclare($\_a, P(x_1,\ldots,x_n)$ と入力すればよいのです. また, 真理函数の引数が一つの場合は単に函数名 P で十分です. つまり, matchdeclare(_a,P) とすることで真理函数 P(_a) との対応が付きます.

ここで例を挙げると, 並びの変数_b を変数 x, y, z を含まない式すると, 式 'freeof(x,y,z,_b)' の意味は 'true' になりますが, matchdeclare 函数には 'freeof(x,y,z)' を引数として与えます. こうすうることで, 並びの変数_b に対応する真理函数 freeof(x,y,z,_b) が変数_b に matchdeclare 属性として付与されます. この処理は Maxima の式の内部表現の構造を考えるとより明瞭になります. 何故なら項 'freeof(x,y,z)' の内部表現は "((FREEOF SIMP) (X Y Z))" であり, matchdeclare 函数は, 函数項の内部表現における変数リスト末尾に並びの変数_b を追加し, その S 式を matchdeclare 属性値として「並びの変数」に与えているのです:

```
(%i33) matchdeclare(_b,freeof(x,y,z));
(%o33)                                  done
(%i34) printprops(_b,matchdeclare);
(%o34)                          [freeof(x, y, z, _b)]
```

したがって, 真理函数の引数の総数が n 個あれば, 最期の第 n 番目の変数が matchdeclare で結び付けられる変数に対応しなければなりません.

ここで, この matchdeclare 函数の引数として与えられる変数は Maxima の変数に限定されず, 函数名を変数として扱うこともできます. ここで函数名も変数も共に記号ですが, このことに加えて函数の内部表現も考えれば明確になります. たとえば, 函数 $f(x,\ldots,z)$ の内部表現は "((f SIMP) (X ... Z))" であり, 本質的に規則は条件付きの置換になるので, 結局, 内部表現の変数部分を置換えるか, あるいは, 函数部分を置換えるかは内部表現の S 式の先頭か末尾の何れかを置換することの違いでしかないからです.

5.7.4 並びの指定に関連する函数

defmatch 函数: 与式から指定した式の並びを検出する函数です:

defmatch 函数
defmatch(⟨ 函数名 ⟩ ,⟨ 式の並び ⟩ , ⟨ 変数$_1$⟩ ,. . .,⟨ 変数$_n$⟩)

defmatch 函数は $n+2$ 個の引数を持つ函数で, 第 1 引数が defmatch 函数で生成される ⟨ 函数の名前 ⟩, 第 2 引数が第 1 引数で指定した函数が検証する ⟨ 式の並び ⟩, そして, 第 3 引数以降が第 2 引数で指定した式の並びを構成する ⟨ 変数$_1$⟩, . . . , ⟨ 変数$_n$⟩ です.
この defmatch 函数で構成された検査函数は, 式の照合に成功すると式 ⟨ 助変数$_i$⟩ = 適合する変数で構成された n 個のリストを返します. 逆に照合に失敗すれば単純に false を返します.
なお, ⟨ 式の並び ⟩ 中の ⟨ 変数$_1$⟩, . . . , ⟨ 変数$_n$⟩ 以外の項に対して matchdeclare 函数による属性の付与があった場合, matchdeclare 函数による述語を用いて式の分解が defmatch 函数で定義した函数で行えるようになります.
次の例では与式の線形性を調べる函数 linear の定義と函数 linear による処理を行っています:

```
(%i2) defmatch(linear,a*x+b,x)
(%i3) linear(3*z+(y+1)*z+y^2,z);
(%o3)                                  false
(%i4)  linear(a*z+b,z);
(%o4)                                 [x = z]
(%i5) nonzeroandfreeof(x,e):=if e#0 and freeof(x,e)
        then true else false
(%i6) matchdeclare(a,nonzeroandfreeof(x),b,freeof(x))
(%i7) linear(3*z+(y+1)*z+y^2,z);
(%o7)                                  false
(%i8) defmatch(linear,a*x+b,x)
(%i9) linear(3*z+(y+1)*z+y^2,z);
                                   2
(%o9)                       [b = y , a = y + 4, x = z]
```

この例では最初に defmatch 函数を使って検証函数 linear を定義し, それから, 式 '3*z+(y+1)*z+y^2' が変数 z の一次式であるかどうかを検証しています. この場合は 'false' を返却していますが, これは defmatch 函数の式の並びで用いた変数 a と b に関して, 他に何も情報がないため, linear 函数が与式に記号 "a" と記号 "b" を持たない一次式と判断したためです. そこで, 函数 linear に式 'a*z+b' を与えてみます. すると今度は「式の並び」'a*x+b' に適合する式で, 式の並びの中の変数 x に対応するものが z であると判断したために, linear 函数は '[x = z]' を返却しています.
そこで, mathchdeclare 函数で指定した式の並びを構成する対象 a と b の情報を追加します. そのために述語 nonzeroandfreenof を定義します. この述語では, 第 1 引数が第 2 引数の式に含まれず, 第 2 引数が '0' と異なれば 'true' とし, それ以外を 'false' とします. そして, matchdeclare 函数を使って, 対象 a の属性値として 'nonzeroandfreeof(x,a)' , 対象 b の属性値として 'freeof(x,b)' を与えます. すなわち, 対象 a は '0' と異なり変数 x を含まない対象, 対象 b は変数 x を含まない対象という性質を付与しているのです. この matchdeclare 函数による属性の付与ののちに, defmatch 函数で函数 linear を再定義します. この再定義を行わないと対象 a と b に関する情報は更新されません. それから linear 函数を使って '3*z+(y+1)*z+y^2' を調べると, 今度は与式と式の並び 'a*x+b'

を比較して '[b=y^2, a=y+4, x=z]' を返します.
これで式の並びで用いる変数の解説を終えます. 次に, defrule 函数を用いた規則について解説します.

5.7.5　defrule 函数による規則

defrule 函数による規則の定義

defrule 函数:　規則の定義を行う函数です:

defrule 函数
defrule(⟨ 規則名 ⟩,⟨ 並び ⟩, ⟨ 置換 ⟩)

defrule 函数は与えられた ⟨ 並び ⟩ を指定した ⟨ 置換 ⟩ で置換える規則の定義と規則の名付けを行う函数です. ここで ⟨ 規則名 ⟩ は Maxima の記号であり, 規則の式への適用では, 後述の apply 函数族に帰属する函数を用います. この適用によって ⟨ 並び ⟩ に適合する全ての項が ⟨ 置換 ⟩ で指定した値で置換されます. なお, 照合に失敗すると元の式をそのまま返却します.
defrule 函数で設定した規則は大域変数 rules に割当てられたリストに規則名が登録されます. 単純に rules; と入力すると, その時点で Maxima に追加されている規則名一覧のリストが表示されます. この大域変数の附置は空リスト "[]" ですが, Maxima の函数によっては規則を定義し, その規則を用いて処理を行う函数があります. たとえば, trigsimp 函数を実行すると, trigsimp 函数で利用する 8 個の規則が導入されます:

```
(%i1) rules;
(%o1)                                  []
(%i2) trigsimp(sin(x)^2+cos(x)^2);
(%o2)                                   1
(%i3) rules;
(%o3) [trigrule1, trigrule2, trigrule3, trigrule4, htrigrule1, htrigrule2,
                                                        htrigrule3, htrigrule4]
```

この trigsimp 函数については §9.2.4 を参照して下さい.
さて, ここで挙げる実例では最初に演算子を定義し, その演算子に対して規則を定めてみましょう. ここで定義する演算子 "dfx" は微分演算子のような癖を持った前置表現の演算子とし, この演算子に対する微分としての性質を規則として defrule 函数で入れてみます:

```
(%i2) prefix("dfx");
(%o2)                                  dfx
(%i3) declare("dfx",linear);
(%o3)                                done
(%i4) defrule(leibniz,dfx (a.b), (dfx a).b + a.(dfx b));
(%o4)          leibniz : dfx (a . b) -> dfx a . b + a . dfx b
(%i5) rules;
(%o5)                               [leibniz]
(%i6) dfx( a+b );
(%o6)                             dfx b + dfx a
```

defrule 函数による定義のあとに rules; と入力すれば, その時点での defrule 函数によってさだめられた規則のリストが表示されます. この例で示すように大域変数 rules に規則 leibniz が登録されています.

defrule 函数による規則の表示

disprule 函数: defrule 函数, tellsimp 函数と tellsimpafter 函数による規則の内容を表示する函数です:

規則の表示を行う函数

disprule(⟨ 規則$_1$⟩ ,⟨ 規則$_2$⟩ ,...)
disprule(all)

この disprule 函数は defrule 函数, tellsimp 函数, tellsimpafter 函数で定義した規則の詳細を, defmatch 函数によって定義された並びの名前込みで表示します. また, 引数を 'all' とすると Maxima が持つ, これらの函数によって定義された全ての規則を表示します.
disprule 函数による規則の表示では, 記号 "-¿" の左側に規則を適用される式, そして, 右側に適用した結果が表示されます. この書式は defrule 函数で定義した規則の内部書式に関連します.
先程の defrule 函数の動作を確認してみましょう:

```
(%i7) disprule(chain1);
(%t7)              chain1 : dfx (a . b) -> dfx a . b + a .

dfx b

(%o7)                                    [%t4]
(%i8) disprule(all);
(%t8)              chain1 : dfx (a . b) -> dfx a . b + a . dfx b

(%o8)                                    [%t5]
```

この例では defrule 函数で取り上げた Leibniz 則 "chain1" を disprule 函数で表示させたものです. なお, 引数に all を指定すると大域変数 rules に含まれる規則全ての表示を行います.
次に, disprule 函数の内部的な動作も確認してみましょう. この defrule 函数の実体 (=内部函数) は proc-$defrule 函数です. そこで, 演算子 ":lisp" を用いて, この函数の動作を調べてみましょう:

```
(%i8) :lisp (trace proc-$defrule)
WARNING: TRACE: redefining function PROC-$DEFRULE in top-level, was defined in
         /usr/local/maxima-5.13.0/src/binary-clisp/matcom.fas
;; Tracing function PROC-$DEFRULE.
(PROC-$DEFRULE)
(%i8) defrule(chain1,dfx (a*b),(dfx a)*b+a*dfx b);
1. Trace:
(PROC-$DEFRULE
 '($CHAIN1 (($DFX) ((MTIMES) $A $B))
   ((MPLUS) ((MTIMES) (($DFX) $A) $B) ((MTIMES) $A (($DFX) $B)))))
1. Trace: PROC-$DEFRULE ==>
((MSETQ) $CHAIN1
```

```
 ((MARROW) ((SDFX SIMP) ((MTIMES SIMP) SA SB))
  ((MPLUS SIMP) ((MTIMES SIMP) ((SDFX SIMP) SA) SB)
   ((MTIMES SIMP) SA ((SDFX SIMP) SB)))))
(%o8)                  chain1 : dfx (a b) -> a dfx b + dfx a b
```

ここで示すように defrule 函数で定義される規則の内部表現は “((MARROW) 適用前の式 適用後の式)” の書式であり, disprule 函数は, この S 式のヘッダ “(MARROW)” を単純に “-¿” で置き換えて表示しています.

defrule による規則の適用を行う函数 (apply 函数族)

defrule 函数による規則の適用は apply1,apply2,applyb1 函数を用います. ここで, apply 函数族は defrule 函数で与えられた規則を式に適用させる函数で, apply1 函数と apply2 函数が式の木構造の上側から作用するのに対し, applyb1 函数が木構造の下側 (Bottom) から規則を作用させます:

apply 函数族
apply1(⟨ 式 ⟩ ,⟨ 規則$_1$⟩ , . . . ,⟨ 規則$_n$⟩)
apply2(⟨ 式 ⟩ ,⟨ 規則$_1$⟩ , . . .,⟨ 規則$_n$⟩)
applyb1(⟨ 式 ⟩ ,⟨ 規則$_1$⟩ , . . .,⟨ 規則$_n$⟩)

apply1 函数　与えられた ⟨ 式 ⟩ に対して ⟨ 規則$_1$⟩ を最初に作用させます. このときに ⟨ 式 ⟩ の木構造の根側から大域変数 maxapplydepth で指定される深さ迄の全ての部分式に ⟨ 規則$_1$⟩ を適用させます. これによって得られた ⟨ 式$_2$⟩ に対し, 次の ⟨ 規則$_2$⟩ を同様に適用します. 以降, 帰納的に各部分式に作用させ, ⟨ 規則$_n$⟩ を全ての部分式に作用させて終了します.

apply2 函数　⟨ 規則$_1$⟩ が ⟨ 式 ⟩ の部分式で失敗すると, ⟨ 規則$_2$⟩ を適用する点で apply1 函数とは異なります. 大域変数 maxapplydepth で指定された深度以下の全ての部分式で失敗したときに限って, 全ての規則が次の部分式に繰返し適用されます. もし, 規則の一つが成功すれば, その同じ部分式が ⟨ 規則$_1$⟩ で再実行されます.

applyb1 函数　この函数は apply1 函数と似ていいますが, apply1 函数が ⟨ 式 ⟩ の木構造の上から下へと作用して行くのに対して, applyb1 函数は ⟨ 式 ⟩ の最下層の部分式から作用させて, 規則の適合に失敗すると, もう一つ上の階層の部分式に帰納的に作用させます.
なお, apply1 函数, apply2 函数と applyb1 函数は無制限に階層構造の全ての部分式に規則を適用しません. 大域変数 maxapplydepth で apply1 函数と apply2 函数が規則を適用する階層の深さを指定し, 大域変数 maxapplyheight で applyb1 函数が到達する階層の高さを指定します. ただし, これらの大域変数の既定値が 10000 のために通常の利用ではほとんど無制限と言っても構わないでしょう.
これらの適用函数は与式の木構造, すなわち, 内部構造に密接に関連します. そのために二項演算子の場合は特に nary 型か infix 型であるかで結果が異ります:

```
(%i2) matchdeclare([_a,_b],all);
(%o2)                                done
(%i3) prefix("dfx");
(%o3)                                 dfx
(%i4) nary("><");
(%o4)                                 ><
(%i5) defrule(Leibniz,dfx (_a><_b),(dfx _a)><_b +_a><(dfx _b));
(%o5)       Leibniz : dfx (_a >< _b) -> dfx _a >< _b + _a >< dfx _b
(%i6) applyb1(dfx (a><b><c),Leibniz);
(%o6)                          dfx (a >< b >< c)
(%i7) apply1(dfx (a><b><c),Leibniz);
(%o7)                          dfx (a >< b >< c)
(%i8) apply2(dfx (a><b><c),Leibniz);
(%o8)                          dfx (a >< b >< c)

(%i9) infix("<>");
(%o9)                                 <>
(%i10) defrule(Leibniz2,dfx (_a<>_b),(dfx _a)<>_b +_a<>(dfx _b));
(%o10)      Leibniz2 : dfx (_a <> _b) -> dfx _a <> _b + _a <> dfx _b
(%i11) applyb1(dfx (a<>b<>c),Leibniz2);
(%o11)                dfx (a <> b) <> c + a <> b <> dfx c
(%i12) apply1(applyb1(dfx (a<>b<>c),Leibniz2),Leibniz2);
(%o12)          (dfx a <> b + a <> dfx b) <> c + a <> b <> dfx c
(%i13) apply1(dfx (a<>b<>c),Leibniz2);
(%o13)          (dfx a <> b + a <> dfx b) <> c + a <> b <> dfx c
(%i14) apply2(dfx (a<>b<>c),Leibniz2);
(%o14)          (dfx a <> b + a <> dfx b) <> c + a <> b <> dfx c
```

ここでは二つの演算子 "><" と "<>" を定義し, Leibniz 則を与えています. ところが, nary 型の演算子 "><" を用いた式 'a><b><c' に対しては規則の適用が上手くできないのに対し, infix 型の演算子 "<>" では上手く処理ができています. これは内部表現の違いによるものです.

nary 型演算子項の内部表現は "((nary 型演算子) 被演算子$_1$, ..., 被演算子$_n$)" と nary 演算子が先頭に一つ置かれ, そのうしろに被演算子が並ぶ書式となります. これに対して infix 型の演算子の場合は "((infix 型演算子) (... ((infix 型演算子) 被演算子$_1$, 被演算子$_2$)...), 被演算子$_n$)" となります. そのために項 'a><b><c' が内部で "((><SIMP) A B C)" となるのに対し, 項 'a<>b<>c' は "((<>SIMP)((<>SIMP) A B) C)" となります. ここで規則は項 '_a><_b' や項 '_a<>_b' に対して適用させるので, 変数_a と変数_b に対応する被演算子を捜します. その結果, nary 型の場合は被演算子が平リストとして配置されるために適応するものがないと判断する一方で, infix 型の場合は被演算子が二つ一組になるために照合に成功します. nary 型の演算子に規則を適用させる場合には括弧 "()" を上手く利用して規則の適用を行う函数にとって分かり易い書式にしておく必要があります:

```
(%i15) apply1(dfx((a><b)><c),Leibniz);
(%o15)          (dfx a >< b + a >< dfx b) >< c + (a >< b) >< dfx c
(%i16) apply1(dfx((a><b><c)><d),Leibniz);
(%o16)            dfx (a >< b >< c) >< d + (a >< b >< c) >< dfx d
```

この例では '(a><b)><c' としたため, 最初の適合で, '_a→a¿¡b, _b→c' の対応がつき, さらに, 項 'a><b' に対しても_a と_b との対応がつくために apply1 函数による規則の適用に成功するのです. ところが, 最後の例では項 '(a><b><c)><d' のために最初の照合には成功するものの, 項 'a><bP><c'

の照合に失敗するのです.

defrule 函数で定義した規則の適用に関連する大域変数

規則に関連する大域変数

変数名	既定値	概要
maxapplydepth	10000	apply1 と apply2 が停止する階層
maxapplyheight	10000	applyb1 が停止する階層

大域変数 maxapplyheight は apply1 函数,apply2 函数や applyb1 函数が停止する与式の階層構造の最高位となります. ここで, maxima の式には LISP の S 式のような階層構造があります. apply1 等の函数は maxapplyheight よりも低い個所, すなわち, 木構造の根本側に作用し, それよりも高ければ作用しません (木構造の根元側を最低辺と見做します). ただし, 既定値の 10000 は式のほとんど全てと言える程の高さになるでしょう.

5.7.6　defrule を用いた微分作用素

非可換積項の場合

具体例として matchdeclare 函数で並び変数を定義し, この並び変数に対して defrule 函数で規則を定義して aplly1 函数で規則を適用してみましょう:

```
(%i28) prefix("dfx");
(%o28)                                  dfx
(%i29) declare("dfx",linear);
(%o29)                                 done
(%i30) matchdeclare([_a,_b],lambda([y],not(freeof(x,y))));
(%o30)                                 done
(%i31) defrule(leibniz,dfx(_a._b),(dfx _a)._b + _a.(dfx _b));
(%o31)          leibniz : dfx (_a . _b) -> dfx _a . _b + _a . dfx _b
(%i32) apply1(dfx(sin(x).cos(x)),leibniz);
(%o32)              dfx sin(x) . cos(x) + sin(x) . dfx cos(x)
(%i33) apply1(dfx(sin(x).cos(y)),leibniz);
(%o33)                        dfx (sin(x) . cos(y))
(%i34) printprops([_a,_b],matchdeclare);
(%o34) [lambda([y], not freeof(x, y), b), lambda([y], not freeof(x, y), a)]
```

この例では変数 a と b を変数 x を含む函数とし, 変数 x の函数に対して Leibniz 則を適用させるものです. そのために式 'sin(x).cos(x)' に対しては規則が適用されても式 'sin(x).cos(y)' に対しては規則が適用されていないことに注目して下さい. また, printprops で変数 a, b の matchdeclare 属性を見ると, matchdeclare 函数で指定した函数が属性値として割り当てられていることが判ります. nnnn なお, ここでの Leibniz 則の定義では通常の積演算子 "*" ではなく非可換積の演算子 "." を用いています. これは Maxima の規則では演算子 "+" と演算子 "*" を含む式に対しては規則の適用が上手く動作しないからです.

可換積項の場合

和を使って演算子の線形性, 可換積を使って Leibniz 則を定義した場合に, どのようなことが生じるか確認してみましょう:

```
(%i1) matchdeclare([_a,_b],all);
(%o1)                                done
(%i2) prefix("dfx");
(%o2)                                 dfx
(%i3) defrule(Leibniz2,dfx(_a*_b),dfx(_a)*_b+_a*(dfx _b));
_b _a partitions 'product'
(%o3)          Leibniz2 : dfx (_a _b) -> _a dfx _b + dfx _a _b
(%i4) defrule(Linear2,dfx(_a+_b),dfx _a + dfx _b);
_b + _a partitions 'sum'
(%o4)            Linear2 : dfx (_b + _a) -> dfx _b + dfx _a
(%i5) applyb1(dfx(a*b),Leibniz2);
(%o5)                          dfx (a b) + dfx 1 a b
(%i6) applyb1(dfx(a*b),Linear2);
(%o6)                           dfx (a b) + dfx 0
```

まず, defrule 函数で線形性や Leibniz 則を定義すると可換積 "*" に対しては partitions 'product', 和 "+" に対しては "partitions 'sum'" といった文言を含む警告が出ています. どうも雲行きが怪しそうです. 次に, applyb1 函数で規則を適用させるとどうなるでしょうか？ 線形性の場合には 0, Leibniz 則の場合には 1 が出ていることに注意して下さい. これは非常に危険な事態を招きます. 何故なら, apply1 函数を用いると '1=1*1' や '0=0+0' で式が展開されるために無限ループに陥ってしまうからです.
これは, 並びの式を S 式で表現した場合に, "((演算子) (+ 並びの式の変数$_1$, ..., 並びの式の変数$_n$n))" や "((演算子) * 並びの式の変数$_1$, ..., 並びの式の変数$_n$))" となるときに規則の適用を行うと, 並びの式の変数$_1$ に演算子の被演算子全てが割当てられ, 残りの 並びの変数$_2$, ..., 並びの変数$_n$ に和や積の単位元が割当てられる仕様となっているためです. この処理は和の演算子と可換積の演算子の場合に限定されます. それ以外の演算子ではこのような現象は生じません. なお, ここでの例で applyb1 函数を用いたのは, applyb1 函数が式の木構造の葉の部分, すなわち, 底 (bottom) から式を適用するために無限ループに陥いらないからです. 木構造の上, つまり, 根の部分から規則を適用する apply1 函数を用いると無限ループに陥ります. ただし, 演算子が単位元に対して '0' を返す性質を与えていれば無限ループには陥りません. しかし, 期待した規則の適用による式の変形はできません.
この例で示すように Maxima の和の演算子 "+" と可換積の演算子 "*" は非常に特別な演算子であり, 利用者が属性を付与したり, これらの演算子項を使って何等の規則を与えることが事実上できなくなっていることに注意して下さい.

5.7.7 tellsimp 函数と tellsimpafter 函数による規則の定義

Maxima では入力された式の項を Maxima の項順序 "$>_m$" に従って並び換えたり, 数値の四則演算や, 項の和や差を自動的に簡易化します. この処理を受け持つ函数は内部函数 simplifya で, 大域変数 simp によって制御されています (§7.2).

ここで tellsimp 函数と tellsimpafter 函数は指定した規則を simplifya を用いて簡易化させる函数です. これらの函数の構文を纏めておきましょう:

tellsimp と tellsimpafter

tellsimp(⟨ 並び ⟩,⟨ 置換 ⟩)
tellsimpafter(⟨ 並び ⟩,⟨ 置換 ⟩)

tellsimp 函数と tellsimpafter 函数は規則を定義する函数ですが, defrule 函数や let 函数のように定義した規則を適用する函数を用いて式の変形を行う必要がない点で異ります.
tellsimp 函数と tellsimpafter 函数で定義した規則は自動的に大域変数 rules に追加されます. このときに規則名は自動的に設定されますが, 規則名は並び変数の内部表現の最初の成分, すなわち, 演算子名や函数名に文字列 rule と, この演算子/函数に対応する規則番号を並べた文字列で表現されます. ここで規則番号はその演算子/函数に tellsimp 函数や tellsimpafter 函数で設定された規則の総数に 1 を加えた正整数値で与えられるものです:

```
(%i1) matchdeclare([_a,_b],all);
(%o1)                                done
(%i2) prefix("dfx");
(%o2)                                 dfx
(%i3) nary("><");
(%o3)                                 ><
(%i4) rules;
(%o4)                                 []
(%i5) tellsimp(dfx(_a><_b),(dfx _a)><_b+_a><(dfx _b));
(%o5)                          [dfxrule1, false]
(%i6) rules;
(%o6)                             [dfxrule1]
(%i7) dfx(a><b><c);
(%o7)                          dfx (a >< b >< c)
(%i8) dfx(a><b);
(%o8)                      dfx a >< b + a >< dfx b
(%i9) dfx((a><b)><c);
(%o9)        (dfx a >< b + a >< dfx b) >< c + (a >< b) >< dfx c
(%i10) nary("<>");
(%o10)                                <>
(%i11) tellsimpafter(dfx(_a<>_b),(dfx _a)<>_b+_a<>(dfx _b));
(%o11)                  [dfxrule2, dfxrule1, false]
(%i12) dfx((a<>b)<>c);
(%o12)               dfx (a <> b) <> c + (a <> b) <> dfx c
```

tellsimp 函数と tellsimpafter 函数は共に第 1 引数に式の並び, 第 2 引数に置換式を指定します. この例では演算子 dfx に対して規則を二つ定めていますが, 最初の tellsimp による規則には dfxrule1, 次の tellsimpafter による規則では, dfxrule2 と演算子名, 文字列 rule, 既存の規則の総数に 1 を加えた数値で規則名が構成されていることが判ります.
ここで式の並びでは演算子に注意が必要になります. 基本的に和 "+" や可換積 "*" のみで構成された式の並びに対しては, 式の並びの照合に失敗する可能性が高く, 最悪の場合, 無限ループに陥る可能性もあります. 特に tellsimp 函数の場合は再帰的に規則の適用を行うために, 式の入力と同時に無限ループに陥る羽目になって非常に危険です.

5.7.8 let 函数による規則

let 函数

let 函数による規則の定義を解説しましょう. まず, let 函数の構文を次に示しておきましょう:

let 函数

let([⟨ 式の並び ⟩,⟨ 式 ⟩,⟨ 述語 ⟩,⟨ 変数$_1$⟩,...,⟨ 変数$_n$⟩], ⟨ パッケージ名 ⟩)
let(⟨ 式の並び ⟩,⟨ 式 ⟩,⟨ 述語 ⟩,⟨ 変数 ⟩,...,⟨ 変数$_n$⟩)
let(⟨ 式の並び ⟩,⟨ 式 ⟩)
let([⟨ 式の並び ⟩,⟨ 式 ⟩], ⟨ パッケージ名 ⟩)

この let 函数は ⟨ 述語 ⟩ の意味が 'true' の場合に ⟨ 式の並び ⟩ を ⟨ 式 ⟩ で置換する規則を定義する函数です. なお, defrule 函数, tellsimp 函数や tellsimpafter 函数による規則と違う点は, 規則の適用を行う letsimp 函数が内部で CRE 表現を用いるため, CRE 表現に向いた式, 特に有理式が適しています.
let 函数は式の並びと変換式で構成されたリストをパッケージと呼ばれる大域変数に属性として割当てられたリストに追加する作業を行います. 次に具体的な例を示しておきましょう:

```
(%i5) let(pochi2(_a*_b),pochi2(_a)+3*_a*pochi2(_b));
(%o5)              pochi2(_a _b) --> 3 _a pochi2(_b) + pochi2(_a)
(%i6) :lisp (mget $current_let_rule_package 'letrules)
(((MTEXT) (($POCHI2 SIMP) ((MTIMES SIMP) $_A $_B)) -->
  ((MPLUS) (($POCHI2) $_A) ((MTIMES) 3 $_A (($POCHI2) $_B))))
 ((MTEXT) (($POCHI SIMP) ((MTIMES SIMP) $_A $_B)) -->
  ((MPLUS) (($POCHI) $_A) ((MTIMES) $_A (($POCHI) $_B)))))
(%i6) mike(_a,_b):=true;
(%o6)                              mike(_a, _b) := true
(%i7) let(pochi3(_a*_b),pochi3(_a)+3*_a*pochi3(_b),mike,_a,_b);
(%o7)  pochi3(_a _b) --> 3 _a pochi3(_b) + pochi3(_a) where mike(_a, _b)
(%i8) :lisp (mget $current_let_rule_package 'letrules)
(((MTEXT) (($POCHI3 SIMP) ((MTIMES SIMP) $_A $_B)) -->
  ((MPLUS) (($POCHI3) $_A) ((MTIMES) 3 $_A (($POCHI3) $_B))) WHERE
  (($MIKE) $_A $_B))
 ((MTEXT) (($POCHI2 SIMP) ((MTIMES SIMP) $_A $_B)) -->
  ((MPLUS) (($POCHI2) $_A) ((MTIMES) 3 $_A (($POCHI2) $_B))))
 ((MTEXT) (($POCHI SIMP) ((MTIMES SIMP) $_A $_B)) -->
  ((MPLUS) (($POCHI) $_A) ((MTIMES) $_A (($POCHI) $_B)))))
(%i8)
```

この例では最初に述語を指定せずに式の並びと置換式だけを let 函数で指定し, 次には述語と変数を加えた指定を行っています. この例で判るように規則は大域変数 current_let_rule_package に束縛されたリストに追加されていることが判ります.
⟨ 式の並び ⟩ には, Maxima の原子, sin(x) や f(x,y) のような函数の可換積 "*", 商 "/" や冪 "^" を含む項になります. ただし, 負の冪を用いる場合には大域変数 letrat を true にする必要があります.
⟨ 式$_1$⟩ に含まれる ⟨ 変数$_i$⟩ と対応する ⟨ 述語 ⟩ を省く場合, それらの変数が matchdeclare 函数によって予め true であると宣言されていなければなりません.

let 函数の引数の末尾に 〈 パッケージ名 〉 を追加すれば, 定義した置換規則を指定したパッケージに追加します. 未設定の場合は自動的に大域変数 current_let_rule_package に割当てられたパッケージに追加されます.
これらの置換函数は一度に幾つかの規則の組合せを用いて作用させられます. 規則の組合せは任意数の let 函数で操作された任意の数の規則を含むことが可能で, 利用者が与えた名前で参照されます.
さて, let 函数で定義した規則を式に適用する場合, defrule 函数による定義とは異なり, letsimp 函数を用います. この letsimp 函数の構文を次に示しておきましょう.

letsimp 函数による規則の適用

letsimp 函数

letsimp(〈 式 〉,〈 規則パッケージ名$_1$〉,...,〈 規則パッケージ名$_n$〉)
letsimp(〈 式 〉,〈 規則パッケージ名 〉)
letsimp(〈 式 〉)

ここで letsimp 函数は 〈 式 〉 が指定した規則パッケージに含まれる規則の適用を続けて, 式の変化がなくなるまで規則の適用を続けます.
規則パッケージの指定がない場合, 大域変数 current_let_rule_package に割当てられた規則パッケージを利用します.
パッケージを複数指定した場合, 〈 式 〉 には左端のパッケージから順番に適用します. たとえば, letsimp(expr,package1,package2) を実行すると, 最初に 'letsimp(expr,package1)', 次に 'letsimp(%,package2)' を実行したものと同じ結果が得られます.
ここで規則パッケージを指定して大域変数 current_let_rule_package が切替えられることはありません.

letrules 函数による規則の表示

let 函数で構築した規則の表示は letrules 函数を用いて表示します:

let 函数による規則の表示を行う函数

letrules(〈 規則パッケージ 〉)
letrules()

letrules 函数は引数で指定した 〈 規則パッケージ 〉 に含まれる規則の詳細を表示します. ここで, 引数を指定しない式 'letrules()' の場合, 大域変数 current_let_rule_package に割当てられた規則パッケージに含まれる規則を表示します. なお, この大域変数の附置は default_let_rule_package です.
それでは, 簡単に let 函数と letsimp 函数の動作を確認しておきましょう:

```
(%i1)  matchdeclare([_a,_b],true);
(%o1)                                         done
(%i2)  let([tama(_a)^2,tama(2*_a)+1],nekoneko);
                                     2
(%o2)                        tama (_a) --> tama(2 _a) + 1
```

```
(%i3) letrules();
(%o3)                                    done
(%i4) letrules(nekoneko);
                                2
                          tama (_a) --> tama(2 _a) + 1

(%o4)                                    done
(%i5) letrules(all);
(%o5)                                    done
(%i6)  letsimp(tama(x)^2);
                                           2
(%o6)                                   tama (x)
(%i7)  letsimp(tama(x)^2,nekoneko);
(%o7)                                 tama(2 x) + 1
(%i8) current_let_rule_package:nekoneko;
(%o8)                                   nekoneko
(%i9)  letsimp(tama(x)^2);
(%o9)                                 tama(2 x) + 1
```

この例では sin 函数の倍角公式を模擬したものです. まず, let 函数で nekoneko という名前の規則パッケージに規則を追加し, letrules 函数で規則の表示を行っています. ここで最初の 'letrules()' で大域変数 current_let_rule_package に指定されたパッケージ default_let_rule_package の内容を表示します. 次の letrules 函数では引数にパッケージ nekoneko が指定されているために, パッケージ nekoneko に含まれる規則が表示されます. 次に, letsimp 函数ではパッケージを指定しなければ大域変数 current_let_rule_package で指定されたパッケージの規則が適用されますが, ここには指定がないので, 入力と同じ式を返すことになります. ここで, letsimp 函数にパッケージを指定した場合と, 大域変数 current_let_rule_package にパッケージ nekoneko を指定したた場合には, letsimp 函数によって引数の式に規則が適用されています.
この letsimp 函数は内部的に CRE 表現に式を変換して処理します. したがって, 本質的に多項式の処理となる式の計算では効力を発揮するでしょう.

letsimp 函数に関連する大域変数

letsimp に関連する大域変数

変数名	既定値	概要
default_let_rule_package	'default_let_rule_package	既定値で用いられる規則パッケージ
current_let_rule_package	'default_let_rule_package	規則パッケージを指定しない場合に用いられる規則パッケージ
let_rule_packages	[default_let_rule_packages	規則パッケージ名の一覧
letrat	false	有理式の簡易化に関連

大域変数 default_let_rule_package:　パッケージ名を指定しなかった場合に let 函数で定義された規則が格納されるパッケージになります.

大域変数 current_let_rule_package:　let 函数で構築したパッケージ名を割り当てると, let 函数や letsimp 函数で用いられるパッケージは自動的に current_let_rule_package に割り当てられたパッケージに切り替えられます.

大域変数 let_rule_packages:　let 函数で構築した規則のパッケージのリストを返します. 初期状態では default_let_rule_package のみが存在するために, 大域変数 let_rule_packages の既定値は '[default_let_rule_package]' になります.

大域変数 letrat:　false の場合に letsimp 函数が式の分子と分母を各々別に簡易化して結果を返します. ただし, $n!/n$ を $(n-1)!$ にするような置換はできません. このような置換を行うためには大域変数 letrat を true に設定しておかなければなりません. すると, 分子, 分母の商は要求の通りに簡易化されます.

5.7.9　規則の削除

規則の定義函数と削除函数

規則の削除も規則を定義した函数の系統によって異なりますが, let 函数とそれ以外の函数で大きく二分されます:

規則の定義函数と削除函数

削除函数		規則の定義函数
clear_rules 函数	⇔	defrule 函数,tellsimp 函数と tellsimpafter 函数
remlule 函数	⇔	tellsimp 函数と tellsimpafter 函数
remlet 函数	⇔	let 函数

clear_rules 函数による規則の削除

clear_rules 函数の構文

clear_rules()

clear_rules 函数は引数を取らない函数です. 大域変数 rules に含まれている規則を削除する函数です. clear_rules 函数を実行すると可換積 "*", 可換冪 "^" と和 "+" の規則番号を既定値の 1 に戻します:

```
(%i4) defrule(Leibniz,dfx(_a<>_b),(dfx _a)<>_b+_a<>dfx _b);
(%o4)        Leibniz : dfx (_a <> _b) -> dfx _a <> _b + _a <> dfx _b
(%i5) rules;
(%o5)                                [Leibniz]
```

```
(%i6) tellsimp(dfx(_a<>_b),(dfx _a)<>_b+_a<>dfx _b);
(%o6)                          [dfxrule1, false]
(%i7) rules;
(%o7)                        [Leibniz, dfxrule1]
(%i8) dfx (a<>b);
(%o8)                     dfx a <> b + a <> dfx b
(%i9) clear_rules();
(%o9)                                 false
(%i10) rules;
(%o10)                                 []
```

この例では clear_rules 函数によって, defrule 函数による規則と tellsimp 函数による規則が消去されていることが判ります.

remrule 函数による規則の削除

tellsimp 函数と tellsimpafter 函数に対しては, remrule 函数を用いて直接, 規則を指定して削除できます.

tellsimp 函数と tellsimpafter 函数による規則の削除を行う函数

remrule(⟨ 対象 ⟩,⟨ 規則名 ⟩)
remrule (⟨ 対象 ⟩,all)

remrule 函数は ⟨ 規則名 ⟩ で指定した規則を ⟨ 対象 ⟩ から削除します. 第 1 引数の ⟨ 対象 ⟩ は演算子, または函数を指定します. この理由は tellsimp 函数や tellsimpafter 函数による規則は, 演算子や函数に対して付加される規則だからです. そのために規則名を指定しない場合, 指定した対象に附属する規則をその命名規則に従って全て削除し, 引数が 'all' の場合は大域変数 rules に登録されている全ての tellsimp 函数と tellsimpafter 函数で設定した規則を削除します.

remlet 函数による規則の削除

let 函数で設定した規則の削除は remlet 函数を用います:

let 函数による規則の削除を行う函数

remlet(⟨ 項 ⟩)
remlet(⟨ 項 ⟩,⟨ パッケージ名 ⟩)
remlet(all)
remlet()

これらの函数は全て let 函数で定義された置換規則 ⟨ 項 ⟩ → ⟨ 式 ⟩ を削除します. ⟨ パッケージ名 ⟩ が与えられると指定された規則パッケージから規則を削除します.
それに対し, 引数を指定しない remlet() と clear_rules(), 引数が 'all' の remlet(all) は規則パッケージから代入規則の全てを削除します.

ここで, 規則パッケージ名が, たとえば, 式 ‘relmet(all,⟨ パッケージ名 ⟩)’ で与えられていれば指定された規則パッケージも削除されます. もし, 構築した規則が古い規則を上書きしたものであれば, remlet 函数で新しい規則を削除すれば, 古い規則が復活します.
大域変数 let_rule_package の値は全ての利用者定義の規則パッケージに特殊なパッケージを加えたもののリストとなります. ここで, default_let_rule_package は利用者が特に規則パッケージを指定しない場合に用いられる規則パッケージの名前です.

5.8 式の評価

5.8.1 Maxima での式の評価について

Maxima の式の評価には二種類の式の評価方法があります. 第一の方法は式の入力と同時に入力式の解釈を実行し, その結果, 式が簡易化される自動簡易化, 第二の方法は ev 函数等の函数を用いて利用者が明示的に条件を与え, それによって与式の評価を行う方法です.
第一の方法の自動簡易化では内部函数 simplifya を用いた処理となります. この函数の処理は与式の函数や演算子の属性, および, これらの函数や演算子に関連する大域変数に影響されます. さらに与式を構成する対象に関連する論理式や属性から文脈を用いた式の評価も行われます.
第二の方法の ev 函数を用いる式の評価は利用者が明示的に条件を与える処理になります. この処理では一時的に大域変数や式中の変数の値を変更したり, さらには局所的な変数を利用した処理が可能で, 自動簡易化と比較してより高度な処理ができます.
この節では式の自動簡易化の仕組について概要を述べ, 次に ev 函数による簡易化の仕組, そして, 最後に簡易化に関連する函数や大域変数の解説を行います.

5.8.2 式の自動簡易化

Maxima に入力した式は大域変数 simp が true の場合に自動的に解釈されます. たとえば式 '1+2' や 'cos(%pi/2)' はそれぞれ '3' や '0' に簡易化されます. これは函数や演算子に付加された属性に対応する内部函数を用いて処理を行うためです.
Maxima の組込函数であれば内部函数 defprop を用いて, Maxima の函数や演算子の operator 属性に対して簡易化を行う内部函数が付加されています.
たとえば, cos 函数の場合は内部函数 simp-%cos が対応する内部函数です. この自動簡易化を実行する内部函数の名前の多くは Maxima の函数名の頭に記号 "simp-%" や記号 "-simp" が後部に付いています.
この属性として設定された簡易化を行う内部函数では, Maxima の大域変数によってその式の簡易化制御が行われることもあります.
次に cos 函数の自動簡易化を行う内部函数 simp-%cos の内容を示しておきましょう:

```
(defmfun simp-%cos (form y z)
  (oneargcheck form)
  (setq y (simpcheck (cadr form) z))
  (cond ((double-float-eval (mop form) y))
        ((and (not (member 'simp (car form))) (big-float-eval (mop form) y)))
        ((taylorize (mop form) (second form)))
        ((and $%piargs (cond ((zerop1 y) 1) ((linearp y '$%pi) (%piargs-sin/cos
                (add %pi//2 y))))))
        ((and $%iargs (multiplep y '$%i)) (cons-exp '%cosh (coeff y '$%i 1)))
        ((and $triginverses (not (atom y))
              (cond ((eq '%acos (setq z (caar y))) (cadr y))
                    ((eq '%asin z) (sqrt1-x^2 (cadr y)))
                    ((eq '%atan z) (div 1 (sqrt1+x^2 (cadr y))))
                    ((eq '%acot z) (div (cadr y) (sqrt1+x^2 (cadr y))))
                    ((eq '%asec z) (div 1 (cadr y)))
```

```
                ((eq '%acsc z) (div (sqrtx^2-1 (cadr y)) (cadr y)))
                ((eq '$atan2 z) (div (caddr y) (sq-sumsq (cadr y) (caddr y
                  )))))))
        ((and $trigexpand (trigexpand '%cos y)))
        ($exponentialize (exponentialize '%cos y))
        ((and $halfangles (halfangle '%cos y)))
        ((apply-reflection-simp (mop form) y $trigsign))
        ;((and $trigsign (mminusp* y)) (cons-exp '%cos (neg y)))
        (t (eqtest (list '(%cos) y) form))))
```

ここで Maxima の函数や大域変数には先頭に文字 “$” が付くので, どのような函数や定数が利用されているか判るかと思いますが, ここで%piargs, trigexpand と halfangles が Maxima の大域変数になります. このことからも, これらの大域変数が cos 函数の自動簡易化に大きな影響を持つことが判るでしょう.
なお, 函数の自動簡易化に影響を与える大域変数はソースファイルを調べなくても, options 函数で調べられます.
この cos 函数を options 函数で調べた結果を示しておきましょう:

```
(%i7) options(cos);
(%o7) [float, numer, bfloat, %piargs, %iargs, triginverses, trigexpand,
                                    exponentialize, halfangles, trigsign, logarc]
```

このように options 函数で返されたリストに含まれる大域変数が, この cos 函数の評価で影響を及ぼす大域変数となるのです.
なお, 演算子や利用者が Maxima の函数を用いて定義した函数については declare 函数によって付与された属性に対応する内部函数を用いて式の評価が自動的に実行されます (5.4.7 参照).

5.8.3　ev 函数

ev 函数の概要

ev 函数は Maxima の式の評価を行う上で, 強力, かつ, 柔軟性のある函数です.
ev 函数の処理では, 最初に函数内部で大域変数 simp を ‘true’ にして与式を内部函数 simplifya で簡易化を行います. それから, ev 函数に与えられた引数から局所的な環境を設定し, その環境で式の評価を実行します.
この ev 函数の構文を示しておきます:

ev 函数

ev(⟨ 式 ⟩,⟨ 引数$_1$⟩,...,⟨ 引数$_n$⟩)
⟨ 式 ⟩,⟨ 引数$_1$⟩,...,⟨ 引数$_n$⟩

ev 函数では評価する式 ⟨ 式 ⟩ を第一引数とし, そのうしろに与式を評価するための環境を ⟨ 引数$_1$⟩,...,⟨ 引数$_n$⟩ で設定します. これによって, 大域変数や変数の値の一時的な変更や式を評価する Maxima の函数 (evfun) が指定できるのです.
さらに Maxima の最上層に限定されますが, 函数項としての表現 “ev()” を外した表記も許容されます.

ここで簡単な例を示しておきましょう:

```
(%i1) ev((x+1)^4,expand);
                          4      3      2
(%o1)                    x  + 4 x  + 6 x  + 4 x + 1
(%i2) (x+1)^4,expand;
                          4      3      2
(%o2)                    x  + 4 x  + 6 x  + 4 x + 1
(%i3) x.y.z;
                                 x . y . z
(%i4) (x.y).z,dotassoc:false;
                                (x . y) . z
(%i5) (x.y).z;
                                 x . y . z
(%i6) x^2+2*x+1,factor;
                                        2
(%o6)                            (x + 1)
(%i7) x^2/(y+1)+2*x/(y^2-1)+1,ratsimp;
                           2    2      2
                          y  + x  y - x  + 2 x - 1
(%o7)                     ------------------------
                                    2
                                   y  - 1
```

この例では最初に式 ‘(x+1)^4’ の展開を ev 函数に “expand” を指定することで行います. 最初の書き方が ev 函数の正規の記述方法ですが, Maxima の最上層に限定されますが, 式 ‘(x+1)^4,expand’ のように函数項の表記 “ev()” が省略できます. この表記が使えないのは block 文内部や lambda 函数内部で, Maxima の最上層, すなわち, 通常の入力プロンプトが出ているときや Maxima の batch ファイル内部では, この表記が利用できます.

次の処理では大域変数 dotassoc の値を一時的に変更して式の評価を実行しています. 可換積の結合律を制御する大域変数 dotassoc の既定値は ‘true’ なので, ‘(x . y) . z’ とそのまま入力すると ‘x . y . z’ に自動的に評価されます. ところが, ここでの評価では一時的に大域変数 dotassoc を ‘false’ にしているために ‘(x . y) . z’ がそのまま返されています. ただし, 一時的に変更しているだけなので, そのあとの ‘(x.y).z’ には影響を及ぼしません.

最後の二つは evfun 属性を持つ函数の factor 函数と ratsimp 函数を使って評価しています. evfun 属性を持つ函数名を指定すると与式に該当する函数を作用させた結果が返却されます. evfun 属性を持たない函数名を指定した場合は函数による式の評価は実行されません.

ev 函数では式に含まれる変数に値を一時的に割当てて評価することも可能です. この評価は方程式の解を解いたあとに, その結果を使って他の式の評価や解の検証といった処理で非常に便利です:

```
(%i29) solve([x^2-y^2+x*y-1,x+y-3],[x,y]);
                 sqrt(41) - 9        sqrt(41) - 3
(%o29) [[x = - ------------, y = - ------------],
                       2                   2
                            sqrt(41) + 9         sqrt(41) + 3
                       [x = ------------, y = - ------------]]
                                 2                    2
(%i30) x*y,%[1];
```

```
                                  (sqrt(41) - 9) (sqrt(41) - 3)
(%o30)                        - ------------------------------
                                              4
```

この例では連立方程式 $x^2 - y^2 + xy - 1 = 0, x + y - 3 = 0$ を解き, その結果を使って xy の計算を行うものです. この評価では ev 函数の函数項表記を省略した式 'x*y, %[1]' を用いています. なお, "%[1]" は solve 函数による結果がリストとなるために, 返却値のリストの第一成分を取出す式で, ここでは '[x = -(sqrt(41)-9)/2, y = (sqrt(41)-3)/2]' に対応します. そして, ev 函数では "=" を含む式が第二引数以降に与えられると, その式の左辺が通常の変数であれば左辺の変数に右辺の式を割当てます. そうすることで, 方程式の解が与式に反映されるという訳です.
では,ev 函数の引数について詳細を解説しましょう.

5.8.4　ev 函数の引数について

属性値の変更による式の評価

ev 函数では evflag 属性を持つ大域変数を引数として与えることで, その大域変数の値を一時的に 'true' に設定したり, evfun 属性を持つ函数を用いて与式を評価することが可能です. ここで evflag 属性と evfun 属性は declare 函数を用いて付与可能な属性です (§5.4.8).
この小節では evflag 属性を持つ大域変数とその動作について述べ, 同様に, evfun 属性を持つ函数とその動作について解説しましょう.

evflag 属性　evflag 属性を持った大域変数を指定した場合, ev 函数はその大域変数の値を true に切り換えて与式の評価を行います.
ここで evflag 属性を持つ大域変数を次に纏めておきます:

evflag 属性を既定値で持つ大域変数

exponentialize	%emode	demoivre	logexpand
logarc	lognumer	radexpand	keepfloat
listarith	float	ratsimpexpons	ratmx
simp	simpsum	simpproduct	algebraic
ratalgdenom	factorflag	ratfac	infeval
%enumer	programmode	lognegint	logabs
letrat	halfangles	exptisolate	isolate_wrt_times
sumexpand	cauchysum	numer_pbranch	m1pbranch
dotscrules	trigexpand		

大域変数に evflag 属性を持たせるためには, declare 函数を用います. そして, 大域変数が evflag 属性を持つかどうかは properties 函数を使って調べられます.
では evflag 属性を持つ大域変数を定義し, ev 函数によって評価する例を示しましょう:

```
(%i1) declare(tama,evflag)$
(%i2) tama:false$
```

```
(%i3) ev('(if tama=true then print("nekoneko") else print("1234")));
1234
(%o3)                                  1234
(%i4) ev('(if tama=true then print("nekoneko") else print("1234")),tama);
nekoneko
(%o4)                                 nekoneko
(%i5) properties(tama);
(%o5)                             [value, evflag]
(%i6) :lisp (get '$tama 'evflag)
T
```

この例では変数 tama に evflag 属性を declare 函数を用いて持たせ, それから, if 文で構成された式の評価を行っています. 最初の tama には 'false' を設定しているために評価式では tama が 'false' の場合の処理が実行されています. 次に, ev 函数の引数として tama を与えると, tama に evflag 属性を持たせているために自動的に値に 'true' が設定され, その値を用いて式が評価されて, tama が 'true' の場合の処理が実行されていることが判ります. 次に, properties 函数を用いて属性を調べています. LISP で調べる場合には, 内部表現に対して get 函数で evflag 属性を取出します. evflag 属性があれば 'T' が返却され, そうでない場合には 'NIL' が返却されることから判別できます.

evfun 属性 evfun 属性は函数名に付与される属性です. この evfun 属性を持つ函数名を ev 函数の引数として指定すると, ev 函数による式の評価で該当する函数を与式に作用させた結果が返却されます. この evfun 属性も declare 函数によって付与可能な属性です.
ここで evfun 属性を持つ函数を纏めておきましょう:

evfun 属性を持つ函数

radcan	factor	ratsimp	trigexpand
trigreduce	logcontract	rootscontract	bfloat
ratexpand	fullratsimp	rectform	polarform

次に, declare 函数を用いて函数に evfun 属性を持たせてみましょう. そこで, ここでは簡単な函数を定義して動作を観察してみましょう:

```
(%i1) mike(z):= diff(z,x,2)$
(%i2) properties(mike);
(%o2)                                [function]
(%i3) x^2,mike;
                                          2
(%o3)                                     x
(%i4) declare(mike,evfun)$
(%i5) properties(mike);
(%o5)                          [evfun, function, noun]
(%i6) x^2,mike;
(%o6)                                     2
(%i7) :lisp (get '$mike 'evfun)
T
```

この例では変数 x で二階微分を行う函数 mike を定義しています. この函数は evfun 属性を最初は持っていないために ev 函数の引数として函数名 mike を与えても函数 mike による評価は実行されません. ところが, declare 函数で evfun 属性を付加すれば, 次の ev 函数で同様の評価を行わせてみると今度は函数 mike が実行され, 結果として二階微分が得られます.
ここで函数が evfun 属性を持つかどうかは,properties 函数や LISP の get 函数で実際に調べることができます. この点も evflag 属性と同様です.
ここで evfun 属性を持つ函数が複数 ev 函数に与えられた場合, この evfun 函数の作用の順番は ev 函数の左端の evfun 函数からになります:

```
(%o29) tst(z) := expand(z )
(%i30) declare(tst,evfun)$
(%i31) (x+1)^2,tst,factor;
                                          4
(%o31)                                (x + 1)
(%i32) (x+1)^2,factor,tst;
                           4       3       2
(%o32)                     x  + 4 x  + 6 x  + 4 x + 1
(%i33) tst(factor(x+1)^2);
                           4       3       2
(%o33)                     x  + 4 x  + 6 x  + 4 x + 1
(%i34) factor(tst((x+1)^2));
                                          4
(%o34)                                (x + 1)
```

この例で示すように ev 函数の引数として, evfun 属性を持つ factor と利用者定義函数の tst を与えています. はじめに ev 函数の引数として左から tst, factor の順に与えたために “factor(tst(与式))” を処理しています. 次に, factor, tst の順で ev 函数の引数として引き渡した場合には, 与式は展開されています. すなわち, “tst(factor(与式))” で処理したからです.

大域変数の一時的な変更による評価

evflag 属性を持つ大域変数に対しては, その値を ev 函数が自動的に変更します. 特定の ev 函数の引数を指定すると関連する複数の大域変数の値を自動的に変更する機能があります:

大域変数の一時的な変更を伴う評価

expand	大域変数 expop に大域変数 maxposex の値, 大域変数 expon に大域変数 maxnegex の値を割当てて式を展開.
expand(⟨ 整数$_1$⟩,⟨ 整数$_2$⟩)	大域変数 maxposex に ⟨ 整数$_1$⟩, maxnegex に ⟨ 整数$_2$⟩ を設定し式を展開.
numer	大域変数 numer と大域変数 float を true にして式を評価.
detout	逆行列の計算で行列式を行列の外に出す.

expand : 大域変数 maxposex と maxnegex に設定された値を大域変数 expop と expon に割当て, 与式の展開を行います.
ここで大域変数 expop と大域変数 expon は冪乗を自動展開する次数を指定する大域変数で, 大域変数 maxposex と大域変数 maxnegex は expand 函数による展開に於て, 展開を行う冪項の最大次数と最小次数を設定する大域変数です.
大域変数 maxposex と大域変数 maxnegex の既定値が '1000' となっているため, ev 函数に expand を引数として与えると最大次数が 1000 以下の冪項の自動展開を行います.
この最大次数と最小次数は利用者が明示的に与えることもできます. その場合は expand を函数に似た書式で ev 函数に与えます.

expand(⟨ 整数$_1$⟩,⟨ 整数$_2$⟩): 大域変数 maxposex に ⟨ 整数$_1$⟩, 大域変数 maxnegex に ⟨ 整数$_2$⟩ を設定して与式を展開します:

```
(%i1) (x+2)^1001,expand;
                                                  1001
(%o1)                                        (x + 2)
(%i2) (x+2)^2/(x+1)^3,expand(2,3);
                  2
                 x                          4 x                        4
(%o2)   ------------------- + ------------------- + -------------------
         3       2              3       2              3       2
        x  + 3 x  + 3 x + 1    x  + 3 x  + 3 x + 1    x  + 3 x  + 3 x + 1
(%i3) (x+2)^2/(x+1)^3,expand(2,2);
                                 2
                                x          4 x          4
(%o3)                         -------- + -------- + --------
                                    3           3           3
                              (x + 1)     (x + 1)     (x + 1)
```

最初の例では冪の次数が 1001 と maxposex の値 '1000' を越えているために展開を行いませんが, 次の例では expop に 2, expon に 3 を指定しており, 展開する式の冪が正の冪の次数の絶対値が 2, 負の冪の絶対値が 3 となっているので今度は展開を実行しています. しかし, expop に 2, expon に 2 を指定すると, 指定した値が負の冪の次数の絶対値よりも小さいために負の冪乗の部分式の展開のみが実行されません.

numer:　この引数を ev 函数に与えた場合, 大域変数 numer と大域変数 float を true にして式の評価を実行します:

```
(%i45) sin(%pi/10);
                                    %pi
(%o45)                           sin(---)
                                    10
(%i46) sin(%pi/10),numer;
(%o46)                        .3090169943749474
(%i47) 2*%e*x+%pi/4,numer;
(%o47)              5.43656365691809 x + .7853981633974483
(%i48) 2*%e^x+%pi/4,numer;
                                  x
(%o48)                     2 %e  + .7853981633974483
(%i49)  2*%e^x+%pi/4,numer,%enumer;
                                      x
(%o49)              2 2.718281828459045  + .7853981633974483
```

なお, 式の中に冪乗でない定数%e が存在する場合は大域変数%enumer を true にして式の評価を行います. ただし, %e の冪の場合は%e を浮動小数点数に変換しません. この変換を行いたければ%enumer も追加します.

detout:　大域変数 detout を true にしたときと同じ結果が得られます. これは detout を指定したときに ev 函数は大域変数の doallmxops と大域変数 doscmxops を false, 大域変数の detout を true にして式の評価を実行するからです:

```
(%i7) A:matrix([1,2,3],[4,3,1],[2,4,1])$
(%i8) A^^(-1),detout;
                                [ - 1   10    - 7 ]
                                [                 ]
                                [ - 2   - 5   11  ]
                                [                 ]
                                [ 10     0    - 5 ]
(%o8)                           -------------------
                                        25
```

微分と積分の評価

微分・積分の評価

risch	risch 積分を実行
diff	式中の名詞型の微分を評価.
derivlist($\langle x_1\rangle,\ldots,\langle x_n\rangle$)	名詞型の微分で, $\langle x_1\rangle,\ldots,\langle x_n\rangle$ による微分のみを評価.

risch:　integrate 函数項に対して Risch 積分を実行させます. これは integrate 函数内部で risch が指定される場合に内部函数 rischint を用い, それ以外は内部函数 sinit を用いる仕様となっている

ためにできることです. 積分の計算で疑問があった場合, risch を試してみると良いでしょう. ただし, 両者が食い違った場合は, それだけ難しい計算をさせていると解釈し, グラフ等を用いた検証を行う必要があります.

diff: 式中の名詞型の微分項 ('diff(...)) の評価を実行します.

derivlist: この derivlist では微分を行う変数を指定するために函数に似た構文を持っています:

derivlist の構文

derivlist(⟨ 変数$_1$⟩ ,...,⟨ 変数$_k$⟩)

ev 函数は derivlist で指定した変数 ⟨ 変数$_1$⟩ ,..., ⟨ 変数$_k$⟩ を含む名詞型の微分を評価します:

```
(%i9) a1:'diff('diff(x^2+2*x*y^2+y^4,x),y)$
(%i10) a1,diff;
(%o10)                                  4 y
(%i11) a1,derivlist(x);
                                  d       2
(%o11)                            -- (2 y  + 2 x)
                                  dy
(%i12) a1,derivlist(y);
                                 d       3
(%o12)                           -- (4 y  + 4 x y)
                                 dx
(%i13) a1,derivlist(x,y);
(%o13)                                  4 y
```

変数の割当や局所変数を用いた評価

変数の割当や局所変数を用いた評価は ev 函数を利用する上で最も便利な機能の一つでしょう. この機能を利用することで方程式の検証が非常に簡単, かつ, 効率的に行えます. このときの引数の形を纏めておきましょう:

変数の割当や局所変数の宣言

⟨ 変数 ⟩ = ⟨ 値 ⟩	大域変数や式中の変数に割当を実行.
⟨ 変数 ⟩ : ⟨ 値 ⟩	大域変数や式中の変数に割当を実行.
local(⟨x_1⟩,...,⟨x_n⟩)	ev 内部で用いる局所変数 ⟨x_1⟩,...,⟨x_n⟩ を宣言.

演算子 "=" と演算子 ":": 式中の変数や大域変数に値を割当てて評価を行う場合は演算子 "=" と演算子 ":" の両方が使えます. ev 函数は, この割当を一時的に実行して与式の評価を行います:

```
(%i31) ev(sin(x),x=1);
(%o31)                                sin(1)
(%i32) ev(sin(x),x=1,float);
(%o32)                                sin(1)
(%i33) ev(sin(x),x=1,bfloat);
```

```
(%o33)                          8.414709848078965B-1
(%i34) ev(sin(x),x=1);
(%o34)                                sin(1)
(%i35) ev(sin(x),x=1,bfloat);
(%o35)                          8.414709848078965B-1
(%i36) ev(sin(x),x:%pi/4,bfloat);
(%o36)                          7.071067811865475B-1
(%i37) ev(sin(sqrt(x^2+y^2)),[x:%pi/4,y=1]);
                                                 2
                                              %pi
(%o37)                          sin(sqrt(---- + 1))
                                              16
```

この例では函数項 'sin(x)' の変数 x に '1' や '%pi/4' 等を割当てています. 割当は演算子 "=" でも演算子 ":" でも構いません. 複数の変数に一度に変数を割当てる場合はリストで与えます. この際に演算子 "=" と演算子 ":" が混在していても問題はありません.
この評価方法は algsys 函数や solve 函数等の方程式を解く函数と評価したい式を組合せると強力です:

```
(%i42) algsys([x^5-x^3+5],[x]);
(%o42) [[x = - 1.53955007256894], [x =
- 1.183445980013718 %i - .4590933961159689],
[x = 1.183445980013718 %i - .4590933961159689],
[x = 1.228868436016586 - .7109481105485196 %i],
[x = .7109481105485196 %i + 1.228868436016586]]
(%i43) map(lambda([z],ev(realpart(x^2),z)),%);
(%o43) [2.37021442594703, - 1.189777641253336,
        - 1.189777641253336, 1.004670417145341, 1.004670417145341]
```

この例では方程式 $x^5 - x^3 + 5 = 0$ の数値近似解を求め, その近似解を二乗したものの実部を求めています.

local:　derivlist に似た構文を持ち, ev 函数内部で利用する局所変数の宣言に用いられます. ただし, ev 函数では後述の式への割当があるために local で宣言しなければ局所変数が扱えない訳ではありません:

```
(%i16) solve(x^3+2*x-b,x),local(b),b=3;
                     sqrt(11) %i + 1       sqrt(11) %i - 1
(%o16)         [x = - ---------------, x = ---------------, x = 1]
                            2                     2
(%i17) solve(x^3+2*x-b,x),b=3;
                     sqrt(11) %i + 1       sqrt(11) %i - 1
(%o17)         [x = - ---------------, x = ---------------, x = 1]
                            2                     2
```

式の評価

式の評価に関連する引数

eval	大域変数 infeval を true に変更して式の評価を実施.
noeval	式を無評価.
nouns	名詞型の式を評価.
pred	論理式を評価 (§5.5.5 参照)
infeval	無限評価を実行させる大域変数

eval: 式の評価を実行します. infeval との違いは, eval では内部変数 evalfg を用いて評価の制御を行いますが, infeval の場合は評価によって式が変化する場合に文字通り無限回の評価となる点です:

```
(%i9) x,x=x+1,eval;
(%o9)                                  x + 2
(%i10) x,x=x+1,infeval;
… 無限ループに陥いります …
```

noeval: noeval を指定することで ev 函数は内部函数 resimplify に式の評価を任せます. また, noeval を指定することで ev 函数で指定した処理の一部が無効になります.

nouns: 演算子 “’” を用いた名詞型の式の評価を行います.

pred: pred は与式が論理式の場合, その評価で用います. Maxima で論理式は真理函数や論理演算子によって評価されるために関係の演算子のみの論理式は自動的に評価されません. そのために論理式の評価を行う場合に pred を用います.

infeval: infeval は無限評価を行う大域変数です. この大域変数を指定すると, 与式の変化が止るまで評価を行います. したがって, ‘ev(x,x=x+1)’ のような式に対して infeval を ‘true’ にして評価を実行すると Maxima は無限ループに陥るので注意が必要です.

5.8.5 評価に関連する函数

函数評価に関連する演算子と函数

’⟨ 式 ⟩
nounify(⟨ 記号 ⟩)
”⟨ 式 ⟩
verbify(⟨ 記号 ⟩)
eval(⟨ 式 ⟩)

演算子 “’”:　演算子 “’” は Maxima による評価を防止, すなわち, 引数を名詞化します. たとえば, ’(f(x)) で函数項 ‘f(x)’ を名詞化し, Maxima による式の評価が回避できます. つまり, ’f(x) は変数 x の評価を行って函数 ‘f’ を作用させた名詞型で返す形になるために, 結果として名詞化された函数 ‘f’ の函数項の評価のみが凍結された形になります.
これと似た函数に nounify 函数があります. こちらは記号を名詞化するもので, 引数が演算子 “’” よりも限定されます.

演算子 “’’”:　この演算子は特殊な評価を行います. たとえば, ’’%o4 でラベル ‘%i4’ に割当てられた式を再評価します. さらに ’’f(x) は函数 ‘f’ を変数 x に作用させて動詞型に変更します. そのため, Maxima は函数項 ‘f(x)’ の評価を行います. なお, 記号に対しては verbify 函数も同様の作用をします:

```
(%i65) test:2*%pi$
(%i66) sin(test);
(%o66)                                0
(%i67) test:%pi/4;
                                     %pi
(%o67)                               ---
                                      4
(%i68) ''%i66;
                                     1/2
                                    2
(%o68)                              ----
                                    2

(%i69) ''sin(test);
(%o69)                        .7071067811865475
```

eval 函数は ⟨ 式 ⟩ の評価を行います. この函数は LISP の eval 函数と同様の働きをします.

5.8.6　函数や演算子に影響を与える大域変数を表示する函数

options 函数:　Maxima の函数や演算子にはさまざまな属性や大域変数による制御が加えられています. 属性に関しては properties 函数で表示可能ですが, 影響を与える大域変数の一覧を表示することが可能な函数に options 函数があります.

options 函数の構文

options()
options(⟨ 記号 ⟩)

options 函数は引数を与えなければ, フロントエンドを立ち上げて数字入力で下の階層に進めます. 直接, 函数名や大域変数名を入力しても構いません. なお, 入力行の末尾には記号 “;” や “$” を入れます. そして, このフロントエンドから抜ける場合には exit; と入力します:

```
(%i14) options();
'options' interpreter  (Type 'exit' to exit.)
1 - INTERACTION
```

```
2 - DEBUGGING
3 - EVALUATION
4 - LISTS
5 - MATRICES
6 - SIMPLIFICATION
7 - REPRESENTATIONS
8 - PLOTTING
9 - TRANSLATION
10 - PATTERN-MATCHING
11 - TENSORS
:
sin;
1 - FLOAT (C)
2 - NUMER
3 - BFLOAT (C)
4 - %PIARGS (S)
5 - %IARGS (S)
6 - TRIGINVERSES (S)
7 - TRIGEXPAND (C S)
8 - EXPONENTIALIZE (S)
9 - HALFANGLES
10 - TRIGSIGN (S)
11 - LOGARC (S)
:
exit$
(%o14)                                  done
```

この option 函数の引数として直接, 函数名や演算子名が指定できます. この場合, options 函数は函数や演算子の属性や影響を与える大域変数名で構成されたリストを返却します:

```
(%i1) options(".");
(%o1) [dotassoc, dotscrules, dotconstrules, dotexptsimp, dotdistrib,
                                                              assumescalar]
(%i2) options(sin);
(%o2) [float, numer, bfloat, %piargs, %iargs, triginverses, trigexpand,
                                  exponentialize, halfangles, trigsign, logarc]
```

なお, 演算子名を指定する場合は二重引用符で括る必要がありますが, 函数や大域変数の場合は, 二重引用符で括る必要はありません.

5.9 LISP に関係する函数

5.9.1 Maxima と LISP

Maxima は Common Lisp と呼ばれる LISP の一方言で記述されています. LISP は函数型と呼ばれるプログラム言語で, プログラムはさまざまな函数を定義して, それらの組合せで構成されます. ちなみに C や FORTRAN は手続型と呼ばれる言語です.
Maxima はこの LISP の上で動作する環境ですが, Maxima 自体は PASCAL 風の処理言語を持っており, 構文的にも LISP を意識することは初歩的な利用ではほとんどありません.
ただし, Maxima で酷いエラーを出して LISP のデバッガに落ちることが稀にあります. LISP のデバッガからの抜け方は Maxima を実装した LISP によって微妙に異なりますが, CLISP の場合は ':q' と入力してみて下さい. それで Maxima に戻る筈です.
LISP の特徴は言語仕様が非常に柔軟な点です. LISP には原子 (アトム) と呼ばれる項があり, それらを空行で区切って小括弧 "( )" で括ったリストと呼ばれる対象が原子の次に基本的な対象となります. このリストは「リストのリスト」といったものも許容します. この原子とリストで構成された対象を S 式と呼びます. ここでに LISP のプログラム自体も S 式になります. そのために LISP の函数でプログラムを操作することさえも容易に行えます. このように非常に柔軟な言語であるため, Maxima の機能拡張や Maxima のプログラムの処理速度向上でしばしば利用されます.

5.9.2 Maxima から LISP の利用

Maxima の面白い点は, Maxima 側から LISP を直接利用できることです. これには幾つかの方法があります. 一つは完全に LISP に切換えてしまう方法でもう一つは LISP の函数を Maxima 側から利用する方法です.

to_lisp 函数と to-maxima 函数: Maxima から LISP に切換えてしまう場合, Maxima の to_lisp 函数を利用します. Maxima で to_lisp(); と入力すると裏方の LISP が表に出ます. これで LISP を使って遊べます. この状態から Maxima に戻りたければ (to-maxima) と入力します. すると通常の Maxima に戻ります. このことを例で解説しておきましょう:

```
(%i1) to_lisp();
type (to-maxima) to restart, ($quit) to quit Maxima.

Maxima> (setq $a '1)
1
Maxima> (to-Maxima)
returning to Maxima
(%o1)                                true
(%i2) a;
(%o2)                                  1
```

この例では最初に to_lisp(); で Maxima から LISP に切換えます. するとプロンプトが "Maxima¿" に切替わることに注目して下さい. そこで Maxima の項 a に値を割当てますが, LISP 側では Maxima の変数には先頭に "$" が付いているので, 項 '$a' に '1' を割当てます. それから (to-maxima) で

Maxima に戻りますが, このときにプロンプトが ‘to_lisp()’ を入力した次の入力行のプロンプトになっていることに注意して下さい. なお, to_lisp 函数の返却値は ‘true’ になります. 最後に a; を入力すると, LISP で変数 $a に割当てた値 ‘1’ が返却されます.
この例で注目して頂きたいことは Maxima で表示されているものと LISP 側で見たものと様子が違うことです. すなわち, Maxima で扱う対象, 勿論, 函数それ自体も含めて LISP では別の表記名があります. この LISP 側での表現を単純に内部表現と呼びましょう. この内部表現を通常の処理で気にすることは殆どありませんが, 細かな処理を行う必要が出た時点で初めて意識することになります.

演算子 “?”: Maxima の函数で先頭に記号 “?” が付いているものが幾つか存在します. このような函数は記号 “?” を外した部分は LISP の函数であり, Maxima から裏の LISP で処理させた結果を Maxima 側に返す函数です. この記号 “?” は通常の LISP の函数にも適応可能な演算子であり, この演算子 “?” を頭に付けた函数は Maxima 内部で記号 “?” を外した LISP の函数として処理されます. ただし, Maxima が介在するために演算子 “?” を用いて LISP の函数を利用する場合, その引数は Maxima で見えているもの, すなわち, 内部表現ではない Maxima 側の表現を設定します.
また, 演算子 “?” を変数に付けると, LISP 側の変数に束縛された値を参照したり, LISP 側の変数に値を束縛することさえもできます:

```
(%i3) ?neko:'123;
(%o3)                                  123
(%i4) :lisp neko
123
(%i4) ?neko;
(%o4)                                  123
```

この例では ‘?neko:123’ で LISP 側の変数 neko に値’123 を束縛させています. この操作は LISP での ‘(setq neko ’123)’ の処理に相当します.

大域変数 lispdisp : 演算子 “?” に関連する大域変数で, LISP 側の変数を表示するときに記号 “?” を表示するかどうかを制御する大域変数です. 大域変数 lispdisp の値が ‘true’ の場合のみ記号 “?” を付けて表示させます.
ここで LISP の函数や変数を利用するために演算子 “?” を使おうとして, 迂闊に記号 “?” の直後に空白文字を入れると, Maxima のオンラインマニュアルを呼出そうとするので注意が必要です.

演算子 “:lisp”: 演算子 “?” に似た働きをする演算子として演算子 “:lisp” があります. こちらは直接 LISP の S 式を Maxima 側から入力し, LISP に評価させた値が得られる演算子です. 演算子 “?” との違いは, 演算子 “?” の被演算子が LISP の函数であっても, その LISP の函数の引数は Maxima での表記になり, 演算子 “:lisp” の場合, その被演算子は通常の LISP の S 式になります. すなわち, Maxima の内部表現そのものを与えなければなりません:

```
(%i26) a:x+y+z;
(%o26)                                 z + y + x
(%i27) :lisp $a;
((MPLUS SIMP) $X $Y $Z)
(%i27) :lisp (car $a)
(MPLUS SIMP)
```

```
(%i27) ?car(a);
(%o27)                                ("+", simp)
```

上記の例では変数 a に式 'x + y + z' を割当てていますが, ここで "$a" が変数 a の内部変数名となります. :lisp $a; でこの変数に割当てた値を参照していますが返却値は内部表現そのものです. このような変数の参照は演算子 "?" ではできません.
さらに :lisp (car $a); の値は内部表現そのものですが, ?car(a); の値はそれを Maxima で解釈した "("+", simp)" となっており, 引数に記号 "$" が付いていないことと, 演算子 ":lisp" の結果に%o ラベルがないことに注目して下さい.
このように演算子 "?" は入出力で Maxima が介在するため, 入出力の度に Maxima の評価を受けますが, 演算子 ":lisp" の場合は直接操作となります. この演算子 ":lisp" は Maxima で内部表現を確認する必要がある場合に特に便利です.

5.9.3　LISP から Maxima の函数を利用

さて, ここまでは Maxima から LISP の函数を用いる話でしたが, 逆に Maxima の函数を裏で走っている LISP から利用することもできます. この目的のために mfuncall 函数を用います. mfuncall 函数の引数は Maxima の函数名の頭に記号 "'$" を付け, そのうしろに引数を並べます:

```
MAXIMA> (mfuncall '$diff '$x '$x 1)
1
```

この例では函数 x を diff 函数を使って変数 x で微分するものです. このように引数は全て内部表現に対応したものでなければなりません. したがって, Maxima 上で引数を割当ててある状態でなければ使い難いものです. 現実的には Maxima 函数の虫取りや動作の確認, あるいは速度向上のために LISP で Maxima のプログラムを生成する際に, 既存の Maxima の函数を LISP の函数から利用するときに使うことになるでしょう.

第6章　Maximaの対象とその操作

Der erste	**第一の学生**
Dieses Buch leg' ich in Eure Hand.	この本を貴方のお手に.
Der zweite	**第二の学生**
Von mir erhaltet Ihr den Schlüssel.	これが本を開ける鍵
Der Dritte	**第三の学生**
Diese Briefschaft macht es zu Eurem Eigentum.	この書簡で貴方は法的に所有者だ.
Faust	**ファウスト**
Wie kommt ein solches Geschenk mir zu?	どうして, この様な物を私に？
Die drei	**三人で**
Du bist der Meister!	貴方が道に通じた者であるが故に!

Buzoni,Doktor Faust[101] より

この章では,Maxima の数値, 多項式, リスト, 配列, 集合, 行列, 及び文字列とそれらで構成される式について述べ, それらの直接処理を, 代入, 簡易化に分けて記述しています.
Maxima の式の内部表現に関しては式の節を参照して下さい. この式の内部表現は代入処理や式の簡易化を行う上で非常に重要で,Maxima の要の一つです.
猶, 比較的規模の大きな数値行列の処理に関しては余程の理由でも無い限り, Maxima よりも Octave 等の数値行列処理に長じたソフトを利用される事を勧めます. MATLAB クローンの Octave に関しては,16 章で数値行列の処理方法やファイル処理,Maxima とのインターフェイスに関して記述しているので, 必要があれば参照して下さい.

6.1 数値

6.1.1 Maxima で扱える数値について

Maxima で扱える数値の型には,「**整数**」,「**有理数**」,「**浮動小数点数**」に加え, これらの数値で構成される「**複素数**」があります. この中で整数, 浮動小数点数は Maxima の原子ですが, 有理数と複素数は Maxima の原子ではなく式です (§5.1.1 参照).

整数: 基盤の Common Lisp の整数をそのまま用いています. ここで Common Lisp の整数には 1-語長の fixnum 型と n-語長の bignum 型の二種類があり, 語長は Lisp の処理系で異なります. たとえば, KNOPPIX/Math の Maxima は GCL を用いていますが, ここでの fixnum 型は $-2^{31} = -2147483648$ から $2^{31} - 1 = 2147483647$ の範囲, 同様に, KNOPPIX/Math の CLISP では $-2^{24} = -16777216$ から $2^{24} - 1 = 16777215$ の範囲, そして SBCL では $-2^{29} = -536870912$ から $2^{29} - 1 = 536870911$ の範囲となります[1]. ここでの値は KNOPPIX/Math が 32-bit 環境の OS のために, 64-bit 環境の OS であれば当然異なります. 実際, 64-bit 環境で CLISP は -2^{48} から $2^{48} - 1$, SBCL なら -2^{60} から $2^{60} - 1$ の範囲となります. この様に 32-bit 環境と 64-bit 環境では扱える整数の範囲が格段に異なるのです. そして, bignum 型は記憶容量の許す範囲となり, 大きな整数が扱えます. なお, fixnum 型と bignum 型の切替えを利用者が意識する必要はありません.

有理数: Maxima の内部表現では, Lisp の有理数を表現する ratio 型ではなく, リストになります:

```
(%i6) b:3/4;
                                      3
(%o6)                                 -
                                      4
(%i7) to_lisp();

Type (to-maxima) to restart, ($quit) to quit Maxima.

MAXIMA> $b

((RAT SIMP) 3 4)
```

つまり, 式 'a/b' の内部表現は '((RAT SIMP) a b)' となります. このことから判るように, 有理数の計算は Maxima 内部での変換処理が加わるために, LISP での有理数の処理速度に劣ることになります.

浮動小数点数: ここでは簡単に浮動小数点数について解説しておきます. この浮動小数点数は「**符号部**」,「**仮数 (小数) 部**」,「**指数部**」の三部構成の対象です. ここで符号部で表現される整数を s , 仮数部から構成される有理数を f, 指数部から構成される整数を ε , 基数を正整数 β とすると

[1] 下限と上限は most-negative-fixnum と most-positive-fixnum で調べられます

$(-1)^s \times f \times \beta^\varepsilon$ で対応する実数が復元できます. このときに符号部の s は 0, あるいは 1 が用いられ, 仮数部 f が正整数 d_i, $0 \le d_i < \beta - 1$, $(i = 1, \ldots, n)$ から $d_1/\beta + \cdots + d_n/\beta^n$ で, 同様に指数部 ε が $\delta_1 + \delta_2 \times \beta + \cdots + \delta_m \times \beta^{m-1} - b$ で与えられます. ここで定数 b を「**下駄履き値**」と呼びます. この浮動小数点数は「零」を中心に対称に分布することが構成方法からも判りますが, 浮動小数点数の濃度 (個数) は $2 \times \beta^{m+n} + 1$ と有限で, もし m, n の一方を ∞ としても高々 $\aleph_0$ です. しかし実数の濃度は $\aleph$ なので浮動小数点数で実数を網羅することはできません. そのために浮動小数点数は本質的に近似値としての性格を持ちます. だから, 数式処理できちんと計算したとしても, 浮動小数点数での計算では幾らかの誤差が残留する可能性があり, それが数値計算に悪影響を及ぼす可能性となります.

Maxima では基数を 2 とする全体の長さが 64 ビット, 指数部が 11 ビット, 仮数部が 52 ビット, 符号部が 1 ビットの倍精度浮動小数点数と任意の長さの多倍長浮動小数点数の二種類が扱えます.

この倍精度と多倍長の浮動小数点数の関係はちょうど整数と有理数の関係に似ています. つまり, 倍精度浮動小数点数は Lisp の倍精度浮動小数点数にそのまま対応しますが, 多倍長浮動小数点数はリストで表現される点です:

```
(%i8) a:1.0b0;
(%o8)                                1.0b0
(%i9) to_lisp();

Type (to-maxima) to restart, ($quit) to quit Maxima.

MAXIMA> $a

((BIGFLOAT SIMP 56) 36028797018963968 1)
```

多倍長浮動小数点数の長さ, すなわち, 精度は大域変数 fpprec で指定されます. この大域変数 fpprec の既定値は '16' です. fpprec は内部表現では直接現れていませんが, 内部表現で Lisp の函数 caddar で得られる数値に反映されています. つまり, a を多倍長整数とし, このときの内部表現を '(bigfloat simp n) α) δ)' としたとき, n は $[(\mathrm{fpprec} + 1)\log_2 10]$ で与えられます. ここで "[ ]" は Gau{ss の記号と呼ばれ, 実数 a に対して $[a]$ は実数 a を越えない最大の整数を意味します[2]. また, α は $\mathrm{a}/|\mathrm{a}|2^{[(\mathrm{fpprec}+1)\log_2 10]} \sum_{i=0}^{n} d_i/2^{i-n}$ で与えられ, a は $\alpha/2^{n-\delta}$ で復元されます:

```
(%i9) float(?cadr(1.0b-2)*2^(?caddr(1.0b-2)-floor((fpprec+1)*log(10)/log(2))));
(%o9)                                0.01
(%i10) float(?cadr(1.0b+2)*2^(?caddr(1.0b+2)-floor((fpprec+1)*log(10)/log(2))));
(%o10)                               100.0
(%i11) fpprec:20;
(%o11)                                20
(%i12) float(?cadr(1.0b+2)*2^(?caddr(1.0b+2)-floor((fpprec+1)*log(10)/log(2))));
(%o12)                               100.0
(%i13) float(?cadr(1.0b-2)*2^(?caddr(1.0b-2)-floor((fpprec+1)*log(10)/log(2))));
(%o13)                               0.01
```

このように Maxima の多倍長浮動小数点数は IEEE-754 が定める浮動小数点数と異なった形式であり, その上, 内部ではリストとして処理されるために, Lisp の浮動小数点数としてそのまま扱われる倍精度浮動小数点数と比較して格段に処理速度が落ちることが容易に予想できます.

[2] Maxima の函数では floor 函数が対応します.

また, 大域変数 fpprintprec に表示に必要な桁数を設定すれば, この大域変数 fpprintprec に設定した桁数で多倍長浮動小数点数の表示が行われるものの, 表示桁数は大域変数 fpprec の値を越えることはありません:

```
(%i5) fpprec:30$fpprintprec:20$

(%i7) bfloat(1/3);
(%o7)                          3.33333333333333333b-1
(%i8) fpprec:50$fpprintprec:50$

(%i10) bfloat(1/3);
(%o10)       3.3333333333333333333333333333333333333333333333333b-1
(%i11) fpprec:20$fpprintprec:50$

(%i13) bfloat(1/3);
(%o13)                         3.33333333333333333b-1
```

複素数: 「複素数」は Maxima の整数, 有理数と浮動小数点数とそれらの数に ‘%i’ をかけたものとの和で表現されます. たとえば, 方程式 $x^2 - 4x + 13 = 0$ の根を Maxima では ‘2+3*%i’ と ‘2-3*%i’ で表現します. さらに複素数の実部と虚部はそれぞれ realpart 函数と imagpart 函数で取り出せます. 複素数は整数, 有理数, 浮動小数点数や多倍長浮動小数点数の自然な拡張となっているために, 複素数の実部と虚部はこれらの数で表現されます.

代数的数: この他の数に代数的整数もありますが, 代数的整数は最小多項式や大域変数を用いるために §6.2 にて説明します.

6.1.2 四則演算について

Maxima での数値の四則演算は通常の記号が使えます. また, さまざまな型の数値が混在していてもエラーにはなりませんが, 基本的に優位にある数の型に自動的に変換されます. この順序は 複素数 > 多倍長 > 倍精度 > 有理数 > 整数 ですが, 整数の ‘0’ との積の場合は常に ‘0’ が返却され. 有理数と整数の演算で約分によって整数に簡約化可能な場合には整数が返却されます.
また, 倍精度と多倍長浮動小数点数の演算では, 倍精度浮動小数点数側が多倍長浮動小数点数で「近似」されるために本来の数値と異なった値で置き換えられて演算処理が行われることもあります:

```
(%i26) fpprec:40;
(%o26)                                    40
(%i27) 1.0b04*1.09;
(%o27)          1.0900000000000000079936057773011270910501b4
```

その危険性があるために倍精度から多倍長精度への変換が行われた際に警告を出すためのフラグとなる大域変数 float2bf があります:

```
(%i28) float2bf:false;
(%o28)                                  false
(%i29) 1.0b04*1.09;
Warning:  Float to bigfloat conversion of 1.0900000000000001
```

(%o29) 1.09000000000000007993605777301127091050 1b4

この大域変数 float2bf の既定値は true で, この場合には変換が行われても警告が出ません. 値が 'false' の場合に警告を出す仕様となっています. なお, この大域変数は函数 bfloat に対しても有効です.

6.1.3 数値に関連する大域変数

数値に関連する大域変数

変数名	初期値	取り得る値	概要
domain	real	[real,complex]	多項式の係数環を指定
float2bf	false	[true,false]	float → 多倍長浮動小数点数の際の警告の有無を指定
fpprec	16	正整数	多倍長浮動小数点数の桁数を指定
fpprintprec	0	正整数	多倍長浮動小数点数の表示桁数を指定
bftrunc	true	[true,false]	bfloat 函数の表示を制御
bftorat	false	[true,false]	多倍長浮動小数点数の有理数への変換を制御
m1pbranch	false	[ture,false]	-1 の原始 n 乗根自動変換の有無を指定
radexpand	true	[true,false]	根号の外に出すかどうかを指定

大域変数 domain: 大域変数 domain には既定値として 'real' が設定されています. これは Maxima が主に実数上で処理を行うことを意味しています. したがって, 式 'x+%i*y' が与えられたときに, この式を構成する変数 x と y は実数に限定されるため, 与式 'x+%i*y' の実部は x, 虚部が y となります. ここで大域変数 domain の値を 'complex' にすると, 複素数上で処理が行われることを意味し, この場合, 'x+%i*y' の変数 x と y が複素数値を取り得るために, 実部と虚部は分りません. さらに大域変数 domain の値を 'complex' , 大域変数 m1pbranch を 'true' にした場合, -1 の n 乗根 (原始 n 乗根) は Gauß平面上の点として自動的に変換されます. たとえば, $(-1)^{\frac{5}{7}}$ は $e^{\frac{5i\pi}{7}}$ に自動的に変換されます.

大域変数 float2bf: false の場合に bfloat 函数で浮動小数点数を多倍長浮動小数点数に変換する時点で計算精度が落るとの警告メッセージを表示させます.

大域変数 fpprec: 多倍長浮動小数点数の桁数を定めます. すなわち, 大域変数 fpprec の値を正整数 n に設定すると多倍長浮動小数点数は n 桁になります.

大域変数 bftrunc:　多倍長浮動小数点数の表示を短縮して表示するかどうかを制御する大域変数です. 大域変数 bftrunc の値が 'true' の場合は短縮して表示しますが, 'false' の場合は長く表示します.

大域変数 bftorat:　大域変数 ratepsilon と組合せて用いられる大域変数です. この大域変数 bftorat の値が 'false' の場合, 大域変数 ratepsilon が多倍長浮動小数点数から有理数型への変換で用いられ, 大域変数 bftorat の値が 'true' の場合に生成される有理数は多倍長浮動小数点数になります.

大域変数 m1pbranch:　-1 の原始 n 乗根の自動変換を制御する大域変数です. 大域変数 m1pbranch が 'true' で大域変数 domain が 'complex' の場合, 原始 n 乗根が Gauß平面上の点として自動的に変換されます.

大域変数 radeexpand:　式 $\sqrt{a^2 b}$ の様に根号の外に出せる因子が式に含まれる場合に 'true' であれば, そのような因子を根号の外に出します.

6.1.4　Maxima の数学定数

Maxima の数学定数

%e	Napia 数
%gamma	Euler の定数
%phi	$\frac{1+\sqrt{5}}{2}$
%pi	円周率 π
inf	正の実無限大.
minf	負の実無限大.
infinity	複素無限大, 任意の偏角で無限大
zeroa	0_+.limit 函数で利用
zerob	0_-.limit 函数で利用

円周率 '%pi', 黄金率 '%phi', Napia 数 '%e' と Euler 定数 '%gamma' については予め numer 属性値として 2048 桁の数値が mlisp.lisp にて付与されています
これらの定数の他に zeroa や zerob のように limit 函数だけで用いる定数もあります:

```
(%i55)  limit(1/x,x,zeroa);
(%o55)                                  inf
(%i56)  limit(1/x,x,zerob);
(%o56)                                 minf
```

ただし, 'limit(1/(x-1),x,1,'plus)' のように 'limit(1/(x-1),x,1+'zeroa)' とする使い方はできません.
なお, 'inf' や 'minf' といった極限に関連する数を含む式の評価は limit 函数で行えます. この場合, 'limit(〈式〉)' で式の評価が行えます.

6.1.5 数に関連する真理函数

引数が一つの真理函数

引数が一つの数値に関連する真理函数

函数名	true となる条件
numberp	整数, 有理数, 浮動小数点数や多倍長浮動小数点数
bfloatp	多倍長浮動小数点数
floatnump	浮動小数点数
integerp	整数
evenp	偶数
oddp	奇数
primep	素数
constantp	定数の場合

これらの真理函数は真理値が {true,false} で, 一つの引数を取ります.

引数が二つの数値に関連する真理函数

float_approx_equal 函数と bfloat_approx_equal 函数

float_approx_equal(⟨ 浮動小数点数$_1$⟩,⟨ 浮動小数点数$_2$⟩)
bfloat_approx_equal(⟨ 多倍長浮動小数点数$_1$⟩,⟨ 多倍長浮動小数点数$_2$⟩)

これらの函数は, その真理値集合を {true,false} とする真理函数で, 共に二つの引数を取り, それらの数値の差が大域変数 float_approx_equal_tolerance を用いて計算した値よりも小さくなった場合に等しいと判断して 'true' を返し, それ以外は 'false' を返す函数です.

6.1.6 整数値函数

引数が一つの整数値函数

一変数の整数値函数

函数名	変換前		変換後
isqrt	整数	→	与えられた整数の平方根を越えない整数
fix	実数	→	与えられた実数を越えない最大の整数
entier	実数	→	与えられた実数を越えない最大の整数
floor	実数	→	与えられた実数を越えない最大の整数
ceiling	実数	→	与えられた実数を越える最小の整数
round	数値	→	与えられた実数を四捨五入で丸めた整数

isqrt 函数:　引数 ⟨ 整数 ⟩ の絶対値の平方根を越えない整数値を返す函数です.　なお,　isqrt 函数は内部で inrt 函数を用いており, 式 'isqrt(x)' を内部で "inrt(x,2)" として処理しています:

```
(%i50) isqrt(-3);
(%o50)                                1
(%i51) isqrt(-4);
(%o51)                                2
(%i52) isqrt(10);
(%o52)                                3
(%i53) isqrt(-10);
(%o53)                                3
```

ここで isqrt 函数は引数が整数でなければ入力式をそのまま返却します.

fix 函数,entirer 函数と floor 函数:　これらの函数は ⟨ 数値 ⟩ が実数の場合, ⟨ 数値 ⟩ を越えない最大の整数 n を返す函数で, 両者の違いは全くありません:

```
(%i42) fix(10);
(%o42)                               10
(%i43) fix(-10);
(%o43)                              - 10
(%i44) fix(10.5);
(%o44)                               10
(%i45) fix(-10.5);
(%o45)                              - 11
(%i46) entier(10);
(%o46)                               10
(%i47) entier(-10);
(%o47)                              - 10
(%i48) entier(10.5);
(%o48)                               10
(%i49) entier(-10.5);
(%o49)                              - 11
(%i50) floor(10);
(%o50)                               10
(%i51) floor(-10);
(%o52)                              - 10
(%i53) floor(10.5);
(%o53)                               10
(%i54) floor(-10.5);
(%o54)                              - 11
```

なお, この例のように絶対値で越えない数を返すのではないので注意が必要です.

ceiling 函数:　引数以上の整数で最小の整数を返す函数です:

```
(%i19) ceiling(10);
(%o19)                               10
(%i20) ceiling(-10);
(%o20)                              - 10
(%i21) ceiling(10.5);
(%o21)                               11
(%i22) ceiling(-10.5);
```

```
(%o22)                                  - 10
```

引数が二つの整数値函数

二変数の整数値函数
inrt(⟨ 整数値$_1$⟩,⟨ 整数値$_2$⟩)

inrt 函数: ⟨ 整数値$_2$⟩ 乗が ⟨ 整数値$_1$⟩ の絶対値を越えない最大の整数を返す函数です. なお, inrt(x,2) は内部表現自体が isqrt(x,2) と等しくなります:

```
(%i19) inrt(x^2,2);
                                        2
(%o19)                               isqrt(x )
(%i20) makelist(inrt(128,x),x,1,8);
(%o20)                      [128, 11, 5, 3, 2, 2, 2, 1]
```

6.1.7 一般の数値函数

一般の数値函数	
max(⟨ 実数値$_1$⟩,⟨ 実数値$_2$⟩,$\cdots$)	⟨ 実数値$_1$⟩,⟨ 実数値$_2$⟩,$\cdots$) の最大値
min(⟨ 実数値$_1$⟩,⟨ 実数値$_2$⟩,$\cdots$)	⟨ 実数値$_1$⟩,⟨ 実数値$_2$⟩,$\cdots$) の最小値
sqrt(⟨ 式 ⟩)	与式の平方根

max 函数と min 函数: 実数列の最大値は max 函数, 最小値は min 函数で計算可能です. なお, リストや集合に含まれる数値の最大値や最小値は lmax 函数と lmin 函数で求められますが, これらの函数は実質的に max 函数と min 函数です.

sqrt 函数: ⟨ 式 ⟩ が実数の場合はその平方根を返します. なお, Maxima 内部では通常, ⟨ 式 ⟩^(1/2) で表現されています. そのために substpart 函数等を用いて式の操作を行う場合は注意が必要になります. なお, 大域変数 sqrtdispflag を初期値の false から true に変更すると sqrt ではなく 1/2 の冪乗として表示されます:

```
(%i34) sqrt(x);
(%o34)                                sqrt(x)
(%i35) sqrtdispflag:false;
(%o35)                                 false
(%i36) sqrt(x);
                                        1/2
(%o36)                                 x
```

数値の変換に関連する函数

rationalize	式	→	式中の float 型や bfloat 型を有理数化
float	全ての数	→	浮動小数点数
bfloat	全ての数	→	多倍長浮動小数点数

rationalize 函数:　与えられた式に含まれる float 型や bfloat 型の数値を有理数に変換する函数です. 式は通常の式やリストが扱えます. なお, 与式がリストや集合であれば fullmap 函数で rationalize 函数が作用させられる仕組になっています:

```
(%i54) rationalize({1,2.03,3.0,4.0});
                                  203
(%o54)                       {1, ---, 3, 4}
                                  100
(%i55) rationalize({1,2.03,3.0,4.0,{x^4+1.03}});
                              203          4    103
(%o55)                   {1, ---, 3, 4, {x  + ---}}
                              100               100
(%i56) rationalize([1,2.03,3.0,4.0,{x^4+1.03}]);
                              203          4    103
(%o56)                   [1, ---, 3, 4, {x  + ---}]
                              100               100
```

float 函数:　全ての数と数値函数を浮動小数点数型に変換する函数です.

bfloat 函数:　全ての数と数値函数を多倍長浮動小数点数型に変換する函数です.

6.1.8　疑似乱数に関する函数

疑似乱数に関連する函数

make_random_state(⟨ 正整数 ⟩)
make_random_state(⟨ 乱数状態 ⟩)
make_random_state(true)
make_random_state(false)
set_random_state(⟨ 乱数状態 ⟩)
random(⟨ 数値 ⟩)

Maxima の疑似乱数生成では, 松本眞と西村拓士によって開発された Mersenne Twister 法 (略して MT 法[3]) が用いられています.

make_random_state 函数:　この函数は乱数状態を生成する函数です.

[3] もう一つの由来は Makoto Takushi です.
http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/name.html を参照

set_random_state 函数: 乱数の生成で利用する乱数状態を指定する函数で, その引数は乱数状態です.

random 函数: 引数として ⟨ 数値 ⟩ を取り,0 から ⟨ 数値 ⟩ – 1 の間の整数乱数を返す函数です.

6.1.9 複素数に関連する函数

複素数に関連する函数

cabs	式	→	与式の複素数としての絶対値
realpart	式	→	与式の実部
imagpart	式	→	与式の虚部
carg	式	→	与式の偏角

なお, 一般の式に対しては展開した式の “%i” を含まない部分式を実部, “%i” を含む部分式を虚部としています.

cabs 函数: 複素数の絶対値を返却する函数です:

realpart 函数: 複素数の実部を返却する函数です.

imagpart 函数: 複素数の虚部を返す函数です.

carg 函数: 与えられた複素数の偏角 θ を $\pi \geq \theta > -\pi$ の範囲で返す函数です:

```
(%i46) cabs(x+%i*y);
                                      2    2
(%o46)                              sqrt(y  + x )
(%i47) realpart(x+%i*y);
(%o47)                                x
(%i48) imagpart(x+%i*(z+%i*y));
(%o48)                                z
(%i49) carg(1+%i/2);
                                        1
(%o49)                                atan(-)
                                        2
(%i50) %,float;
(%o50)                          0.463647609000806
```

6.1.10 LISP 由来の数値函数

これらの数値函数は LISP の函数をそのまま利用します. そのために先頭に演算子 “?” が付いています. なお, 引数は内部表現ではなく通常の Maxima の式になります.

LISP 由来の数値函数	
?round	〈 数値 〉の最も近い整数
?truncate	〈 数値 〉の小数点以下を切り捨て

?round 函数: 〈 数値 〉を最も近い整数に丸めます. ここで, 引数は浮動小数点数であり, 多倍長浮動小数点数ではありません.

?truncate 函数: 浮動小数点数の〈 数値 〉を引数とし, 小数点以下を切り捨てる函数です.

6.2 多項式

この節では Maxima の多項式の表現について述べます. なお, 数学的な事項, 順序や式の表現に関しては §4 を参照して下さい.

6.2.1 多項式の一般表現

Maxima の多項式の扱いは数値を扱う様に自然に扱うことができます. SAGE の様にオブジェクト指向の数式処理の場合は, 予め変数が何であるかを明瞭にしておく必要があります. しかし, Maxima ではその様なことはなく, 通常の C や FORTRAN で記述する “x^2+3*x*z+4” や “x**2+3*x*z+4” のような書式で多項式を入力すればよいのです. ところが, 多項式の内部表現は入力式そのままではありません. 大域変数の設定に従って入力された多項式の簡易化を行ない, Maxima の項順序 “$>_m$” に従って項内部の変数や項自体の並び換えを実行したものになります. この様にすることで, 多項式の項や変数の順序等が異なっていても内部表現は同一のものとなります.
では, 実際に入力式と内部式の例を示します:

```
(%i28) a:x+y+z;
(%o28)                                z + y + x
(%i29) :lisp $a;
((MPLUS SIMP) $X $Y $Z)
(%i29) b:z+x+y;
(%o29)                                z + y + x
(%i30) :lisp $b;
((MPLUS SIMP) $X $Y $Z)
(%i31) c:(1+2)*x+3*y+(2+1-2)*z-z;
(%o31)                                3 y + 3 x
(%i32) :lisp $c;
((MPLUS SIMP) ((MTIMES SIMP) 3 $X) ((MTIMES SIMP) 3 $Y))
(%i33) a1*x+a2*x;
(%o33)                               a2 x + a1 x
(%i34) d:x1^2*x8^2*x3;
                                       2       2
(%o34)                               x1  x3 x8
(%i35) :lisp $d;
((MTIMES SIMP) ((MEXPT SIMP) $X1 2) $X3 ((MEXPT SIMP) $X8 2))
```

この例では多項式 ‘x+y+z’ と ‘(1+2)*x+3*y+(2+1-2)*z-z’ の処理を示しています. 最初に変数 a に式 ‘x+y+z’ を割当てています. ここで, :lisp $a; を使って変数 a の内部表現を参照すると ‘((MPLUS SIMP) $X $Y $Z)’ が返却されます. この S 式の頭にある “(MPLUS SIMP)” は式の主演算子が和 “+” であることを示し, そのうしろに被演算子となる項が並んでいます. このように Maxima の多項式の内部表現でも先頭に演算子が置かれる前置式表現になっています. なお, 項の並びは Maxima の項順序 “$>_m$” を変更しない限り, 式の項や変数の入力の順番を変更しただけであれば同一の内部表現となります. この例で示すように式 ‘x + y + z’ も式 ‘z + x + y’ も同じ式 ‘z + y +x’ で置換えられます. これは Maxima の項順序 “$>_m$” に従って, 与えられた多項式の項を構成する変数や項を並べ替えているからです.

この項順序 “$>_m$” は基本的に逆アルファベット順で変数を並べる順序です. 順序一般に関しては §4.8, 順序 “$>_m$” に関しては §5.2 を参照して下さい. 多項式の内部表現では項に対して項を構成する変数を順序 “$>_m$” に従って小さいもの順に並べ, 式を構成する項に対しては順序 “$>_m$” で小さな項で並べます. 最初の例の式 ‘x + y + z’ の内部表現は ‘(+ x y z)’ となります. これは項順序が ‘z $>_m$ y $>_m$ x’ となるためで, 項を順序 “$>_m$” に従って小さいものから順に並べたからです. 次に単項式 ‘x1^2 x8^2x x3’ の場合, この単項式は ‘x1^2 x3 x8^2’ と各変数が並べ替えられています. これは ‘x8 $>_m$ x3 $>_m$ x1’ となるために Maxima 内部で変数を x1, x3, x8 の順に並べ替えた結果です. さらに ‘x*4’ のような変数と数値で構成された項の場合, 数値が順序 “$>_m$” の下位に来るために ‘4*x’ と数値と変数が入れ替えられます.
次に多項式と単項式の一般表現について纏めておきましょう:

多項式と単項式の一般表現

多項式	((mplus simp) 項$_1$ ⋯ 項$_m$)	項$_m$ $>_m$ ⋯ $>_m$ 項$_1$
単項式	((mtimes simp) 数値 変数$_1$の冪 ⋯ 変数$_n$の冪)	変数$_n$ $>_m$ ⋯ $>_m$ 変数$_1$

Maxima では内部表現を基に式の表示を行います. 多項式の場合は大きな項から順番に表示されます. そのために ‘x+y+z’ と入力しても ‘z+y+x’ と表示されます. ところが項に関しては大きな変数からではなく, 小さい順に並べたままで表示されます. そのため, 項 “x1^2*x8^2*x3” は変数を順序 “$>_m$” に従って並べ替えられた “x1^2*x3*x8^2” で表示されます.
次に, 式 ‘x+y’, ‘x-y’, ‘x*y’, ‘x/y’, ‘x^y’ に対し演算子 “:lisp” を使って, その内部表現を調べてみましょう:

```
(%i33) t0:x+y;
(%o34)                                  y + x
(%i34) :lisp $t0;
((MPLUS SIMP) $X $Y)
(%i35) t1:x-y;
(%o35)                                  x - y
(%i35) :lisp $t1;
((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y))
(%i36) t2:x*y;
(%o36)                                   x y
(%i37) :lisp $t2;
((MTIMES SIMP) $X $Y)
(%i38) t2:x/y;
                                          x
(%o38)                                    -
                                          y
(%i39) :lisp $t3;
((MTIMES SIMP) $X ((MEXPT SIMP) $Y -1))
(%i40) t4:x^y;
                                          y
(%o40)                                   x
(%i41) :lisp $t4;
((MEXPT SIMP) $X $Y)
```

この結果から, 式 ‘x-y’ が ‘x+(-y)’, 式 ‘x/y’ が ‘x^(-y)’ に置換されていることが判ります. このように Maxima の内部では可換積や可換積の冪と和を用いて多項式が表現され, 差や商の処理の手間を

減らしています.
この一般表現に対して Maxima には内部表現をより簡潔にし, 係数を有理数に変換した正準有理式表現 (Cannonical Rational Expressions, 略して CRE 表現) もあります.

6.2.2 多項式の CRE 表現

CRE 表現は factor 函数や ratsimp 函数等々で内部的に利用されるもので, 利用者はこの表現をあまり意識する必要はありません. この CRE 表現は本質的に展開された多項式や有理式函数に適したリストによる表現の一つです.
最初に 1 変数多項式 ⟨ 係数$_1$⟩⟨ 変数 ⟩$^{⟨ 次数_1⟩}$ + ⟨ 係数$_2$⟩⟨ 変数 ⟩$^{⟨ 次数_2⟩}$ + ⋯ の CRE 表現は以下の書式で与えられます:

単変数多項式の CRE 表現

(⟨ 変数 ⟩⟨ 次数$_1$⟩ 係数$_1$ ⟨ 次数$_2$⟩ 係数$_2$ ⋯)

なお, この CRE 表現では次数に関して ⟨ 次数$_1$⟩ > ⋯ > ⟨ 次数$_2$⟩ > ⋯ を満しています.
有理数係数の 1 変数多項式とその CRE 表現は項順序 "$>_m$" により一対一に対応します. これを多項式 $3x^2 - 1$ を使って説明しましょう. この多項式は Maxima 内部では $3x^2 + (-1)x^0$ と表現されます. これを λ 式風に考えれば, $\lambda\, x \cdot \left(3x^2 + (-1)x^0\right)$ となるでしょう. 以上から, 係数と次数の対から構成されるリストは '((2 3) (0 -1))' になります. ここでは予め変数が x であるとして処理しているので変数の情報を落していても問題はありませんが, 一般的には変数を明確にしておくべきです. そこで, リストの先頭に変数 x を入れてみましょう. すると '(x (2 3) (0 -1))' となりますが, もっと簡単にできないでしょうか? そこで, リスト中の小括弧を外して '(x 2 3 0 -1)' するのはどうでしょうか? 多項式 $3x^2 + (-1)x^0$ の復元には問題ありませんね. この表記が CRE 表現のアイデアなのです.
多変数多項式の場合も同様に CRE 表現で書換えることができます. ただし, 1 変数の例のような平坦なリストで CRE 表現は表現されず, CRE 表現のリストによる複合リストになります. 多変数の場合で重要なことは変数の間に順序を入れることです. Maxima では辞書式順序を基礎にした順序 "$>_m$" が入っています.
Maxima のデフォルトの順序 "$>_m$" の変数順序の要点だけを述べると, アルファベットの大文字が小文字よりも大きく, アルファベットの中では "z" が一番大きく "a" が一番小さいという逆アルファベット順で順番が付けられています. 変数が二文字以上の場合, 先頭の文字から順番に比較します. もし, 先頭の文字が等しければ次の文字で比較し, 途中で Maxima の順序 "$>_m$" で大小関係がつくと, その順序で変数に順番が入ります. たとえば, "xxz" と "xxy" の場合は頭の二つが文字 "x" なので順序がまだ決まりませんが, 最後の文字 "z" と文字 "y" については 'z $>_m$ y' となるので最終的に 'xxz $>_m$ xxy' となります.
さて, 多変数多項式の CRE 表現に戻りますが, この場合は再帰的な考えで処理します. ここでは例として $2xy + x - 3$ で考えてみましょう. まず, この多項式を x の多項式と看做すと $(2y + 1)x - 3$ となるので, CRE 表現の第一段目は '(x 1 2y+1 0 -3)' となりますね. ただし, 第二成分の "2 y + 1" は CRE 表現ではないので, これを CRE 表現に変換した "(y 1 2 0 1)" で置換える必要があります. 以上から '(x 1 (y 1 2 0 1) 0 -3)' が求める CRE 表現となります. 次に y の多項式として考えると

$2xy + x - 3$ なので '(y 1 2x 0 x-3)' 中の x の式を CRE 表現に置換えると最終的に '(y 1 (x 1 2) 0 (x 1 1 0 -3))' が得られます.
このように多変数の場合, CRE 表現は順序をあやふやにしていると表現が一意に定まるとは限りません. 変数に順序を入れて大きな変数順に式を括れば一意に定まります. Maxima では主変数として宣言された変数が式に含まれている場合, その主変数の多項式と看做して, それ以外の変数に対し, Maxima の変数順序 "$>_m$" を用いて順序を入れます. 式に主変数として宣言された変数が存在しなければ, 順序 "$>_m$" に対して最高位の変数の式中の変数を主変数として CRE 表現を構築します.
ここで, 二つの CRE 表現が与えられたときに, それらの変数順序が異なれば処理は非常に厄介なことになるのが判るでしょう. そのために CRE 表現を用いる函数に関してはあとのことも考えて主変数の設定を行う必要があります.
また, CRE 表現の変数には通常の算術演算子 ("+", "-", "*", "/") や整数冪 ("^") を持たないものを与えます. そのために 'y^2' のような式は変数に使えませんが, 'log(x)' や 'cos(x+1)' のような函数項は使えます.
Maxima の項順序は基本的に辞書式順序ですが, 局所的な変数の順序の変更が ratvars 函数等で指定出来ます.

6.2.3　係数体について

Maxima では多項式環の係数体として有理数 $\mathbb{Q}$ に純虚数 '%i' を付加した体 $\mathbb{Q}[\%i]/\langle \%i^2 + 1 \rangle$ が既定値です.
その他の係数体を Maxima では扱うことが可能です. まず, p を正素数とする場合に多項式の係数体を $\mathbb{Z}_p[\%i]/\langle \%i^2 + 1 \rangle$ とすることも可能です.
さらに Maxima の係数体 $\mathbb{Q}[\%i]/\langle \%i^2 + 1 \rangle$ に代数的整数を追加した体も扱えます. ここで代数的整数とは整数係数の 1 変数多項式で, 最高次数の項の係数が 1 となる monic な多項式の解となる数です. より一般的に整数係数の 1 変数多項式の解となる数は代数的数と呼ばれます. ただし, Maxima では代数的数は扱えません.
代数的整数はその定義から多項式と密接に関連する数です. a が代数的整数であるためには a を解とする多項式が定義から必ず存在します. その a を解とする多項式の中で最小次数の多項式を a の最小多項式と呼びます. 代数的整数の例としては, 純虚数 i や $\sqrt{2}$ が挙げられます. そして, これらの最小多項式はそれぞれ $x^2 + 1$ と $x^2 - 2$ になります.
この係数体に関連する Maxima の大域変数を以下に示しておきます:

環に関連する大域変数

変数名	初期値	概要
modulus	false	剰余 p を設定
algebraic	false	代数的整数の自動簡易化を制御

大域変数 modulus:　この大域変数 modulus に正素数 p を設定すると多項式の CRE 表現への変換の際に p の剰余で計算されます. すなわち, CRE 表現の係数体は $\mathbb{Q}[\%i]/\langle \%i^2+1 \rangle$ から $\mathbb{Z}_p[\%i]/\langle \%i^2+1 \rangle$ になります. この大域変数は mod 函数にも影響を与えます. ただし, mod 函数の第二引数を与えな

い場合, 大域変数 modulus の値が用いられます. また, 大域変数 modulus に正素数以外の値が設定できます.
大域変数 modulus に正素数 p を設定した場合, $p/2$ よりも大きな整数全てを考えなくても良くなります. たとえば, p として 5 を採ると, 整数の剰余は $\{0,1,2,3,4\}$ となりますが, $3 \equiv -2(mod\ 5)$, $4 \equiv -1(mod\ 5)$ となります. 実際, $3-(-2) = 4-(-1) = 5(mod\ 5)$ となるの, 絶対値では $\{0,1,2\}$ だけを考えれば良いのです. このことを利用すれば, 計算をより簡易に行えます.

大域変数 algebraic: Maxima 上で代数的整数の簡易化を行う場合には必ず true にしなければならない大域変数です. 勿論, この大域変数だけでは自動的に代数的整数が処理される訳ではありませんが, 代数的整数を処理する函数や他の大域変数では, この algebraic が true となっている事が前提となっているものがあるために, 代数的整数を扱う場合には true に設定するのが良いでしょう. ここで, 代数的整数に関連する大域変数に大域変数 ratalgdenom があります. これは代数的整数を分母とする項の有理化を制御するものです. これらの変数は ratexpand 函数, ratsimp 函数等の CRE 表現の式を扱う函数に大きく影響します. その他の函数では gcd 函数も影響を受けます.
factor 函数の様に最小多項式を与えることで多項式の処理を代数的整数を付加した係数体上で処理が行える函数もあります. たとえば, 最小多項式が x^2-2 となる代数的整数 a を使って factor 函数で式を分解するときは, 代数的整数をその最小多項式に代入した形で factor 函数に引き渡します:

```
(%i1) factor(x^4-4,a^2-2);
                                      2
(%o1)                        (x - a) (x + a) (x  + 2)
```

この例で判る様に, factor 函数の第二引数で代数的整数 'a' を 'a^2-2' で定義しています.

6.2.4 多項式に関する函数

多項式に関連する真理函数

多項式に関連する真理函数

函数	true を返す条件
ratnump(〈 式 〉)	〈 式 〉が有理式の場合
ratp(〈 式 〉)	〈 式 〉が拡張 CRE 表現の場合

ratnump 函数: 〈 式 〉が〈 有理数式 〉であれば 'true', それ以外は 'false' を返します.

ratp 函数: 〈 式 〉が CRE 表現, あるいは拡張 CRE 表現であれば 'true', それ以外は 'false' を返します.

代数的数の処理

tellrat 函数:　より徹底して代数的整数を利用する場合は tellrat 函数を用います. この tellrat 函数は最小多項式や等式に tellrat 属性を設定するもので, ratsimp 函数や ratexpand 函数で処理を行う際に大域変数 algebraic の値が 'true' であれば, その属性が処理に反映されます.
この tellrat 函数は次の写像を Maxima に組込むものです:

tellrat(式) : `Maxima` の係数環 → `Maxima` の係数環/⟨ 式 ⟩

ただし, tellrat 属性は CRE 表現を扱う ratexpand 函数や ratsimp 函数で処理を行うときのみに反映されます. この tellrat 函数の構文を纏めておきましょう:

tellrat 函数の構文

tellrat(⟨`monic` な多項式 ⟩)
tellrat(⟨ 等式 ⟩)
tellrat()

引数として与えられる式は主変数に対して monic な多項式に限定されます. これは tellrat 函数が代数的整数を Maxima に与える事を目的としているからです. また, 等式として与える場合に等式の左辺は代数的整数の冪乗で係数が '1' のものに限定されます. このときに左辺の変数が主変数と看做され, 右辺を左辺に移した式を最小多項式とする代数的整数が与えられます. したがって, 式 'a^2=c^3-2' を tellrat 函数に与えた場合, この式の主変数が a となるので, 変数 a が Maxima に追加される代数的整数, さらに, その最小多項式が 'a^2-c^3+2' で与えられます. ところが, 'a^2-c^3+2' を引数として与えると, Maxima の項順序 $>_m$ から主変数が c となるために代数的整数が変数 c で表現されることに注意して下さい.
この tellrat 函数は任意個の因子が取れます. また, 'tellrat()' で Maxima に設定された代数的整数を表現する最小多項式のリストを返します:

```
(%i22) tellrat(x^2+x+1);
                                  2
(%o22)                          [x  + x + 1]
(%i23) tellrat(y^3+y^2+y+1);
                             3    2          2
(%o23)                     [y  + y  + y + 1, x  + x + 1]
(%i24) tellrat();
                             3    2          2
(%o24)                     [y  + y  + y + 1, x  + x + 1]
(%i25)
```

tellrat で最小多項式を導入しても即座には反映されません. まず, 代数的整数の簡易化を行うために大域変数 algebraic の値に 'true' を設定していなければなりません. それから ratsimp 等の CRE 表現が扱える函数で処理を行う必要があります.
例として, 'tellrat(a^2-2,b^2=c^4)' の処理を示します:

```
(%i11) tellrat(a^2-2,b^2=c^4);
                                 2     4    2
(%o11)                         [b  - c , a  - 2]
```

```
(%i12) (a+2)^4,algebraic,expand;
                          4       3       2
(%o12)                   a  + 8 a  + 24 a  + 32 a + 16
(%i13) (a+2)^4,algebraic,expand,ratsimp;
(%o13)                                48 a + 68
(%i14) (b+c)^3,algebraic,expand,ratsimp;
                             5       4     3         2
(%o14)                      3 c  + b c  + c  + 3 b c
(%i15) (b+c)^3,expand,ratsimp;
                             3         2       2       3
(%o15)                      c  + 3 b c  + 3 b  c + b
(%i16) algebraic:true;
(%o16)                                true
(%i17) ratexpand((b+c)^3);
                             5       4     3         2
(%o17)                      3 c  + b c  + c  + 3 b c
(%i18) expand((b+c)^3);
                               3         2       2       3
(%o18)                        c  + 3 b c  + 3 b  c + b
```

この例で示すように tellrat で設定した属性は, 大域変数 algebraic の値を 'true' に設定した環境下で ratsimp 函数や ratexpand 函数等の CRE 表現を内部で用いる函数で処理するときに反映されます. なお, この例での式 '(a+2)^4,algebraic,expand,ratsimp' といった入力は ev 函数の表記方法の一つで, 正式には 'ev((a+2)^4,algebraic,expand,ratsimp)' と入力します. 詳細は §5.8.3 を参照して下さい.

次に, tellrat(a^2-2,b^2=c^4); で代数的整数 b を追加したために 'ratexpand((b+c)^3)' の結果が 'b^2 が c^4' で置換されていることに注意して下さい. このように多変数の多項式を用いて代数的整数を入れる場合, 通常は Maxima の項順序 $>_m$ の影響を受けるため, 等式の右辺に代数的整数の項を置きます.

たとえば, 'a=a^2+c^3 や a^2=c^3-a' のようにします. ここで, tellrat 函数を用いて多項式を被約する際に零因子で分母の有理化を行うことに注意する必要があります たとえば, 'tellrat(w^3-1); algebraic:true;rat(1/(w^2-w))' は零による割算になります. なお, このエラーは 'ratalgdenom:false' で大域変数 ratalgdenom に値 'false' を設定することで回避出来ます.

tellrat 函数で設定した属性は tellrat() で見ることができます. この場合, 引数は特に設定する必要がありません.

untellrat 函数: tellrat 函数で設定した属性は untellrat 函数を使えば削除出来ます:

untellrat 函数の構文

untellrat(⟨x⟩)

untellrat 函数で設定した属性を消去する場合, 代数的整数を直接引数として与えます:

```
(%i36) tellrat(a^2-2,b^3-c^2);
                                 2     3    2
(%o36)                          [c  - b , a  - 2]
(%i37) tellrat();
                                 2     3    2
```

```
(%o37)                          [c  - b , a  - 2]
(%i38) untellrat(a);
                                    2    3
(%o38)                             [c  - b ]
(%i39) untellrat(b);
                                    2    3
(%o39)                             [c  - b ]
(%i40) untellrat(c);
(%o40)                               []
(%i41) tellrat(a^2-2,b^3=c^2);
                                  3    2   2
(%o41)                          [b  - c , a  - 2]
(%i42) untellrat(c);
                                  3    2   2
(%o42)                          [b  - c , a  - 2]
(%i43) untellrat(b);
                                    2
(%o43)                             [a  - 2]
```

次に, 複数の変数に対して tellrat と untelrat を実行した例を示します:

```
(%i21)  tellrat(x^2+1,y^2+1);
                                  2       2
(%o21)                          [y  + 1, x  + 1]
(%i22) ev(rat(x^3+1+y^3+y),algebraic);
(%o22)/R/                            - x + 1
(%i23)  untellrat(y);
                                    2
(%o23)                             [x  + 1]
(%i24)  ev(rat(x^3+1+y^3+y),algebraic);
                                   3
(%o24)/R/                         y  + y - x + 1
```

この例では変数 x, y を $x^2+1=0$ と $y^2+1=0$ を満す代数的整数と設定しています. このように複数の変数に対して整係数多項式を tellrat 函数に与えることも可能で, この例で示すように, ev 函数による評価でも的確に評価されています.

次に 'untellrat(y)' で変数 y に関してのみ tellrat で設定した性質 (=変数 y は $y^2+1=0$ を満す代数的整数) であることを除去しています. そのあとは変数 x に関する性質だけで評価が行われています.

なお, tellrat 函数で入力可能な多項式は主変数に関して monic(最高次項の係数が '1') の多項式でなければならなりません. また, 多変数の場合は untellrat 函数は主変数 (maimvar) に対して行います.

```
(%i15) tellrat(x+y+z*y+1);
Minimal polynomial must be monic
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i16) tellrat(x+y+z+1);
(%o16)                           [z + y + x + 1]
(%i17) untellrat(y);
(%o17)                           [z + y + x + 1]
(%i18) untellrat(z);
```

```
(%o18)                                  []
(%i19) tellrat(2*x+y+z+1);
(%o19)                             [z + y + 2 x + 1]
(%i20) untellrat(z);
(%o20)                                  []
```

この例で示すように 'x+y+z*y+1' を tellrat の引数とすると, 主変数が z で, 係数が変数 y となるためにエラーになりますが, '2*x+y+z+1' のように主変数 z の筆頭項が monic でありさえすれば問題ありません. untellrat 函数が主変数のみに使えることも上の例から分ります.

多項式の係数を取り出す函数

多項式の係数を取出す函数

coeff(⟨ 式 ⟩,⟨ 変数 ⟩,⟨ 次数 ⟩)
ratcoef(⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨n⟩)
ratcoef(⟨ 式$_1$⟩,⟨ 式$_2$⟩)
bothcoef(⟨ 式 ⟩,⟨ 変数 ⟩)

coeff 函数: ⟨ 式 ⟩ に含まれる項 ⟨ 変数 ⟩$^{⟨\text{次数}⟩}$ の係数を求めます. ⟨ 次数 ⟩ を省略すると次数は 1 が設定されます. ⟨ 変数 ⟩ は原子, 項, あるいは真部分式です. 具体的には x, $sin(x)$, $a[i+1]$, $x+y$ 等です. なお , 真部分式の場合は $(x+y)$ が式の中に現れていなければならなりません. ここで ⟨ 変数 ⟩$^{⟨\text{次数}⟩}$ の項を正確に求めるために式の展開や因子分解が必要な場合もあります. 何故なら, coeff 函数だけでは自動的に式の展開や因子分解が実行されないからです:

```
(%i1) coeff(2*a*tan(x)+tan(x)+b=5*tan(x)+3,tan(x));
(%o1)                              2 a + 1 = 5
(%i2) coeff(y+x*%e**x+1,x,0);
(%o2)                                y + 1
```

ratcoef 函数: ⟨ 式$_1$ に含まれる項 ⟨ 式$_2$⟩$^{⟨n⟩}$ の係数を返します. ⟨n⟩ が '1' の場合は ⟨n⟩ が省略出来ます. なお, 返却値は ⟨ 式$_2$⟩ に含まれる変数を函数の変数としても含まないものになります. もし, このような係数が存在しない場合は零 '0' を返します. ratcoef は展開等を行って式を簡易化するので単純に ⟨ 式$_2$⟩$^{⟨n⟩}$ の係数を返す coef と異った答を返します.
たとえば, 式 'ratcoeff((x+1)/y+x,x)' は式 '(y+1)/y' を返却しますが, coeff 函数は '1' を返します.
'ratcoef(⟨ 式$_1$⟩,⟨ 式$_2$⟩ ,0)' は ⟨ 式$_1$⟩ から ⟨ 式$_2$⟩ を含まない項の和を返します. そのために ⟨ 式$_2$⟩ が負の冪の項に含まれていれば ratcoef 函数を使ってはいけません. ⟨ 式$_1$⟩ が有理的に簡易化されていれば, 係数は予期したようには現れないかもしれません.

bothcoef 函数: 二成分のリストを返し, このリストの第 1 成分が ⟨ 式 ⟩ 中の ⟨ 変数 ⟩ の係数 (式が CRE 表現であれば ratcoef 函数, それ以外は coeff 函数で見つけたもの) となります. 第 2 成分が ⟨ 式 ⟩ の残りとなります. すなわち, '[a, b]' が返却値であれば, ⟨ 式 ⟩ = a ∗ ⟨ 変数 ⟩ + b となります.

項の総数や次数を返す函数

nterms(⟨ 式 ⟩)
powers(⟨ 式 ⟩,⟨ 変数 ⟩)
hipow(⟨ 多項式 ⟩,⟨ 変数 ⟩)
lopow(⟨ 多項式 ⟩,⟨ 変数 ⟩)

nterms 函数: ⟨ 式 ⟩ を展開した時点の総数を返却します. この函数は多項式以外の函数を含む通常の式でも利用可能ですが, 函数は引数の形式を問わず一つで数えられます:

```
(%i26) nterms((x+1)^2);
(%o26)                                3
(%i27) nterms(sin(x+1)^2);
(%o27)                                1
(%i28) nterms((sin(x+1)+1)^3);
(%o28)                                4
(%i29) nterms((sin((x+1)^10)+1)^3);
(%o29)                                4
```

この例で示すように sin 函数の引数がどのような式であっても, 一つで数えられていることに注意して下さい.

powers 函数: ⟨ 式 ⟩ に現われる ⟨ 変数 ⟩ の次数リストを返します. この函数を利用する為には予め, load(powers); で函数の読込を行う必要があります.

hipow 函数: ⟨ 多項式 ⟩ に含まれる ⟨ 変数 ⟩ の項の最高次数を返します. なお, hipow 函数では式の展開を自分で実行しないために, 予め式を展開しておく必要があります. 次に示す例では与式 '(x+1)^4' を展開せずに hipow を用いた結果と expand 函数を用いて展開した式に対して hipow 函数を利用した結果を示しています:

```
(%i5) hipow((x+1)^4,x);
(%o5)                                 1
(%i6) hipow(expand((x+1)^4),x);
(%o6)                                 4
```

lopow 函数: ⟨ 多項式 ⟩ の部分式 ⟨ 変数 ⟩ の次数で明示的に現われものの中で最も低い次数を返します.

多項式の変数に関連する函数

多項式の変数に関連する函数
ratvars(⟨ 変数$_1$⟩,··· ,⟨ 変数$_n$⟩)
ratweight(⟨ 変数$_1$⟩,⟨ 重み$_1$⟩,··· ,⟨ 変数$_n$⟩,⟨ 重み$_n$⟩)
showratvars(⟨ 式 ⟩)
printlistvar()

ratvars 函数: 与えられた変数リストに含まれる変数に沿った順序を Maxima に入れる函数です. 変数リストは n 個の変数引数で構成し, ratvars 函数を実行したのちにリスト中の一番右側の ⟨ 変数$_n$⟩ が有理式であれば, その変数を有理式の主変数とします. また, 他の変数の順序はリストの右から左への順番に従います. ある変数が ratvars リストから抜けていれば, その変数は一番左側の ⟨ 変数$_1$⟩ よりも低い順序が与えられます.
ratvars 函数の引数は変数, あるいは 'sin(x)' のような有理式とは異なる函数項の何れかでなければなりません. なお, 大域変数 ratvars は, この函数に与えられた引数のリストとなります:

```
(%i26) ratvars(x,y,z);
(%o26)                          [x, y, z]
(%i27) rat(x+y+z);
(%o27)/R/                       z + y + x
(%i28) rat(a+x+y+z);
(%o28)/R/                   z + y + x + a
(%i29) ratvars(z,y,x);
(%o29)                          [z, y, x]
(%i30) rat(a+x+y+z);
(%o30)/R/                   x + y + z + a
```

なお, ratvars 函数で指定した変数リストは printlistvar 函数を用いて表示可能です.

ratweight 函数: CRE 形式の多項式を構成する項の切捨を行うための重みを変数毎に指定する函数で, 大域変数 ratwtlvl と組合せて用います. 引数は ⟨ 変数$_i$⟩ とそれに対応する ⟨ 重み$_i$⟩ の対で, ここで指定した変数と重みはリストとして大域変数 ratweights に登録されます:

```
(%i2) ratweight(a,1,b,1);
(%o2)                          [a, 1, b, 1]
(%i3) ratweights;
(%o3)                          [b, 1, a, 1]
```

このときに, 項の重みは大域変数 ratweights に割当てられたリストに含まれる変数の次数とその重みの積の総和で計算されます. たとえば, 変数 v_1 と変数 v_2 に対応する重みを w_1 と w_2 とするとき, 項 $3 \cdot v_1^2 \cdot v_2$ の重みは $2 \cdot w_1 + w_2$ になります.
大域変数 ratwtlvl の値が 'false' であれば, ratweight 函数による影響はありませんが, 大域変数 ratwtlvl の値として 0 以上の整数を指定すると, この指定以降の CRE 形式の多項式の項の自動的な切捨てが生じます. この項の切捨は Maxima で計算した CRE 形式の多項式の項の重みが大域変数 ratwtlvl を越えたものを '0' に置換えることで実行されます:

```
(%i1) ratweight(a,1,b,2,c,3)$

(%i2) expr:rat(a+2*b+3*c)$

(%i3) ratwtlvl:2$

(%i4) expr^2;
                                        2
(%o4)/R/                               a
(%i5) ratwtlvl:3$

(%i6) expr^2;
                                            2
(%o6)/R/                          4 a b + a
(%i7) ratwtlvl:6$

(%i8) expr^2;
                    2                        2            2
(%o8)/R/          9 c  + (12 b + 6 a) c + 4 b  + 4 a b + a
```

この例では変数 a, b, c に対応する重みを 1, 2, 3 としています. 大域変数 ratwtlvl に '2' を指定して 'expr^2' の計算を行うと, 変数 b と変数 c の重みは既に大域変数 ratwtlvl に割当てた整数値を超過するために切捨の対象となり, その結果, 変数 a のみの項が残ります. 次に, 大域変数 ratwtlvl に '3' を指定して, 'expr^3' を計算すると, 項 '4*a*b' の重みは '3' となりますが, 大域変数 ratwtlvl の値を超過しないために切捨の対象から外れます. 最期に大域変数 ratwtlvl を '6' にすると, 式 'expr^2' の全ての項の重みが超過しないために全ての式が表示されます.
なお, ratfac 函数と ratweight 函数の手法は互換性がないので両方同時に使えません.

showratvars 函数: 〈式〉の大域変数 ratvars のリストを返します:

```
(%i30) exp:x^2+y^2+z^3;
                                  3    2    2
(%o30)                           z  + y  + x
(%i31) showratvars(exp);
(%o31)                             [x, y, z]
```

printlistvar 函数: 内部変数 varlist に束縛された値を返す函数です. この函数は引数を必要としません.

多項式を纏める函数

多項式を纏める函数
factcomb(〈式〉)
fasttimes(〈多項式$_1$〉, 〈多項式$_2$〉)
rootscontract(〈式〉)

factcomb 函数: 〈 式 〉中に現われる階乗の係数を階乗それ自体に置換して纏めます. すなわち, $(n+1)*n!$ を $(n+1)!$ にすることです. ここで, 大域変数 sumsplitfact が false に設定されていれば, minfactorial 函数が factcomb 函数による処理のあとに適用されます.

fasttimes 函数: 多項式の積に対する特殊なアルゴリズムを用いて〈 多項式$_1$〉と〈 多項式$_2$〉の積を計算します. これらの多項式は多変数で各次数に対して係数が '0' でない項が多く, 両者共にほぼ同じ大きさでなければ効果があまり出ません.
n と m を多項式の次数とすると, 古典的な積では $n \cdot m$ の次数で計算を行いますが, fasttimes 函数を用いると $\max(n,m)^{1.585}$ の次数になります.

rootscontract 函数: 有理数次数の冪同士の積を大域変数 rootsmode の値に従って纏めます. たとえば, 大域変数 rootsmode が true の場合, 式 'x^(1/2)*y^(3/2)' を 'sqrt(x*y^3)' で纏めます.
大域変数 radexpand の値が 'true' で大域変数 domain の値が 'real' であれば, rootscontract 函数は abs 函数を sqrt 函数の式に変換します. すなわち, 函数項 'abs(x)' は 函数項 'sqrt(x^2)' で置換えられます. その結果, 式 'abs(x)*sqrt(y)' は 'sqrt(x^2*y)' に変換されます.

rootscontract 函数: logcontract 函数と似た手法で ratsimp 函数を用います.

因子分解に関連する函数

因子分解を行う函数

factor(〈 式 〉)
factor(〈 式 〉,〈p〉)
factorsum(〈 式 〉)
sqfr (〈 式 〉)
factorout(〈 式 〉,〈 変数$_1$〉,〈 変数$_2$〉,$\cdots$)
nthroot(〈 多項式 〉,〈n〉)
polydecomp(〈 多項式 〉,〈 変数 〉)

factor 函数: 〈 式 〉を整数上で既約因子に分解します. 'factor(〈 式 〉,〈p〉)' で最小多項式が 〈p〉 となる代数的数 α を付加した体 $\mathbb{Q}[\alpha]$ 上で式の因子分解を行うことを意味します. なお, factor 函数には動作に影響を与える大域変数が存在します. この大域変数に関しては factor に影響を与える大域変数を参照して下さい.

factorsum 函数: グループ単位で〈 式 〉の因子分解を試みます. このグループの項はそれらの和が因子分解可能なものです. 'expand((x+y)^2+(z+w)^2)' の結果は復元可能ですが, 'expand((x+1)^2+(x+y)^2)' の結果は共通項が存在するために復元出来ません.

sqfr 函数:　無平方 (square-free) の因子に分解して返す函数です. ここで多項式因子 f が無平方であるとは, 定数と異なる多項式 g で, g^2 が f を割切るような因子が存在しない場合です. 特に 1 変数多項式の場合は f と $\frac{d}{dx}f$ が共通の零点を持ちません. そこで, 多項式式 a の無平方因子分解は, 多項式 a の分解 $\prod_{i=1}^{n} a_i^i$ で, その各因子 a_i が無平方であり, $i \neq j$ であれば 'gcd(a_i, a_j)=1' を満すものです. sqfr 函数は与えられた式に対して, この無平方因子分解を計算します.
では, 無平方因子分解 sqfr と因子分解 factor との違いを $4x^4+4x^3-3x^2-4x-1$ で比較して確認しましょう:

```
(%i44)  sqfr(4*x^4+4*x^3-3*x^2-4*x-1);
                                          2    2
(%o44)                              (2 x + 1)  (x  - 1)
(%i45)  factor(4*x^4+4*x^3-3*x^2-4*x-1);
                                                      2
(%o45)                        (x - 1) (x + 1) (2 x + 1)
```

この例で示すように factor 函数は式を徹底して分解しているのに対し, sqfr 函数は程々で止めていることです.
このように多項式の無平方因子分解は通常の因子分解程の手間をかけません. さらに, 有理多項式の積分計算では無平方な因子に分母を分解することで各因子を分母に持つ式に変形して積分を行うアルゴリズムもあり, 無平方分解は幅広く利用されています.

factorout 函数:　〈 式 〉を $f(\langle$ 変数$_1\rangle, \langle$ 変数$_2\rangle, \cdots) * g$ の形式の項の和に書換えます. ここで, g は factorout の引数の各変数を含まない式の積で, f は因子分解された式になります.

nthroot 函数:　与えられた整数係数の 〈 多項式 〉が, ある整数係数の多項式を 〈 正整数 〉で冪乗したものであれば, その多項式を返します. もし, そのような多項式が存在しなければエラーメッセージを表示します.
この函数は factor 函数や sqfr 函数よりも処理が遥かに速いものです:

```
(%i22)  nthroot(x^2+2*x+1,2);
(%o22)                                x + 1
(%i23)  nthroot(x^3+3*x^2+3*x+1,2);
Not an nth power
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i24)  nthroot(1-3*x+3*x^2-x^3,3);
(%o24)                                1 - x
```

polydecomp 函数:　与えられた多項式を指定された変数の 1 変数多項式として多項式の分解を行いってリストで返す函数です. ここでの分解は因子分解によるものとは異ったものです. polydecomp 函数にリスト $[p_1(x), \ldots, p_n(x)]$ が与えられた場合, 本来の多項式 $P(x)$ との関係は *lambda* 表現を用いて次のように記述されます:

$$\lambda\, x.p_1(x)(\lambda\, x.p_2(x)(\cdots(\lambda\, x.p_n(x))\cdots))$$

```
(%i41) polydecomp(x^3+3*x^2+3*x+1,x);
                                      3
(%o41)                          [x , x + 1]
```

因子分解に関連する大域変数

因子分解に関連する大域変数

変数名	初期値	概要
dontfactor	[]	factor 函数で因子分解しない式のリスト
facexpand	true	factor 函数で返される因子を制御
factorflag	false	式に含まれる整数の因数分解を制御
berlefact	true	因子分解のアルゴリズムを制御
intfaclim	1000	factor で用いる最大の約数の設定
newfac	false	因子分解ルーチンの選択
savefactors	false	式の因子の保存を制御
sumsplitfact	true	factcomb の挙動を制御

大域変数 dontfactor: factor 函数で因子分解しない式を登録します. なお, ratvars 函数で入れた変数順序に対して大域変数 dontfactor に割当てられたリストに含まれる変数よりも小さな変数を持つ式の因子分解は実行されません.

大域変数 facexpand: factor 函数で返された既約因子が展開された形式 (既定値) か, 再帰的 (通常の CRE) 形式であるかを制御します.

大域変数 factorflag: false であれば有理式に含まれる整数の因数分解を抑制します.

大域変数 berlefact: false であれば,factor 函数で Kronecher の因子分解アルゴリズムが利用され, それ以外では既定値として Berlekamp アルゴリズムが使われます.

大域変数 intfaclim: 大きな整数の因子分解を行う時に試す最大の約数を指定します. 'false' を指定した場合, つまり, 利用者が factor 函数を明示的に呼び出す場合や, 整数が fixnum, すなわち, 1-語長に適合する場合, 整数の完全な因数分解が試みられます. ここでの語長は Maxima が動作する LISP 環境で異なります. 詳細は §6.1 を参照して下さい. この大域変数 intfaclim の設定値は factor 函数の内部函数の呼び出しで用いられます.

大域変数 intfaclim: 大きな整数の因数分解に長時間を費すのを防ぐために再設定しても構いません.

大域変数 newfac:　'true' であれば, factor 函数は新しい因子分解ルーチンを用います.

大域変数 savefactors:　'true' であれば, 幾つかの同じ因子を含む式の展開の処理速度向上のため, 式の各因子がある函数で保存されます.

大域変数 sumsplitfact:　'true' の場合に factcomb 函数は minfactorial 函数の呼び出しを行います.

共通因子を求める函数

共通因子を求める函数

gcd(⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨ 変数$_1$⟩,$\cdots$)
gcde(⟨ 多項式$_1$⟩,⟨ 多項式$_2$⟩,⟨ 変数 ⟩)
gcdex(⟨ 多項式$_1$⟩,⟨ 多項式$_2$⟩)
gcfactor(⟨Gauß整数 ⟩)
gfactor(⟨ 多項式 ⟩)
gfactorsum(⟨ 多項式 ⟩)
ezgcd(⟨ 多項式$_1$⟩,⟨ 多項式$_2$⟩,$\cdots$)
content(⟨ 多項式 ⟩,⟨ 変数$_1$⟩,$\cdots$,⟨ 変数$_n$⟩)

gcd 函数:　与えられた多項式の最大公約因子を計算します. この gcd 函数は多くの函数, たとえば, ratsimp 函数や factor 函数等でも利用されています.
gcd 函数に直接影響を及ぼす大域変数として, 同名の大域変数 gcd があります. この大域変数 gcd は gcd 函数で用いるアルゴリズムを決定する変数です.

大域変数 gcd に設定可能な値

値	概要
subres	副終結式を利用 (既定値値)
ez	ezgcd 函数を利用
eez	eez gcd を利用
red	被約
spmod	剰余
false	gcd 函数は常に 1 を返却

代数的整数を扱う場合, たとえば, 式 'gcd(x^2-2*sqrt(2)*x+2,x-sqrt(2))' を計算するためには大域変数 algebraic が 'true' で, 大域変数 gcd が 'ez' と 'false' 以外の値でなければなりません.
同次多項式に対しては, 大域変数 gcd の値を 'subres' にすることを推奨します.

gcdex 函数:　3 個の多項式を成分とするリスを返します. このリストを '[a, b, c]' とするとき, 多項式 c が引数の ⟨ 多項式$_1$⟩ と ⟨ 多項式$_2$⟩ の最大公約因子, 多項式 a と多項式 b が共に c = a⟨ 多項式$_1$⟩ + b⟨ 多項式$_2$⟩ を満します. この函数が用いるアルゴリズムは Euclid の互除法に基づくものです.

なお, 多項式が単変数の場合は ⟨ 変数 ⟩ を指定する必要はありませんが, 多変数の場合は多項式を ⟨ 変数 ⟩ で指定した単変数の多項式と看做して, その GCD を計算します.
ここで多変数多項式の場合に変数の指定を行う必要があるかと言えば, 多変数多項式では, 二つの多項式の最大公約因子が存在するとは限らないからです.
数学的には最大公約因子は二つの多項式が生成するイデアルの生成元で, 通常, 多項式の係数が実数や複素数で考えている多項式が単変数のみ, すなわち, 多項式環 $k[x]$ であれば, 任意のイデアルは単項イデアル, つまり, 一つの多項式だけで生成されるイデアルとなるので, 1 変数多項式と看做した場合には最大公約因子が必ず存在します.
そのため, 多変数の場合には二つの多項式の主変数として変数を一つ選択する必要があります. 但し, その選択が妥当でなければ gcdex は適当な答を返すだけです:

```
(%i16) gcdex(x^2+1,x^3+4);
                                  2
                                 x  + 4 x - 1   x + 4
(%o16)/R/                  [- -------------, -----, 1]
                                   17          17

(%i18) gcdex(x*(y+1),y^2-1,x);
                                    1
(%o18)/R/                     [0, ------, 1]
                                   2
                                  y  - 1
(%i19) gcdex(x*(y+1),y^2-1,y);
(%o19)/R/                     [1, 0, x y + x]
```

ここで, 式 'gcdex(x*(y+1),y^2-1,x)' の結果が '1' であることに注意して下さい. この場合, 多項式環 $K(y)[x]$ で処理を行っているので, 共通の因子として期待される 'y+1' になりません. ここで, $K(y)[x]$ は x を主変数とした x と y の多項式環, つまり, x の多項式で, その係数が体 K 上の y の多項式となるものとして, x と y の多項式環 $K[x, y]$ を見直したものです. 一般的に可換環 K が UFD(Unique Factorized Domain:一意分解整域) であれば $K[x]$ も UFD になることが知られています. そのため, Eculid の互除法が利用可能になるので, gcdex は必ずまともな結果を返します.
gcdex(x*(y+1),y^2-1,y); とすれば, 多項式環 $K(x)[y]$ の話になるので '1' ではなく 'x * y + x' になります. ただし, この返却値が良いものとは言い難いものがあります.
gcfactor 函数は Gauß整数上で ⟨*Gauß*整数 ⟩ の因子分解を行います. なお, Gauß整数とは実部と虚部が整数の複素数 $a + bi$ で, 因子は a と b を非負とすることで正規化されています.

```
(%i56) gcfactor(5*%i+1);
(%o56)                          (1 + %i) (3 + 2 %i)
(%i57) gcfactor(2);
                                                2
```

gfactor 函数: Gauß整数上で ⟨ 多項式 ⟩ の因子分解を行なう函数です. これは 'factor(exp,a^2+1)' と同様の結果を返します:

```
(%i3) gfactor(x^4-1);
(%o3)                  (x - 1) (x + 1) (x - %i) (x + %i)
(%i4) factor(x^4-1,a^2+1);
```

```
(%o4)                    (x - 1) (x + 1) (x - a) (x + a)
(%i5)
```

この例で factor を用いたものでは方程式 $x^2 + 1 = 0$ の解となる代数的整数 a を用いて $x^4 - 1$ を因子分解しています.

gfactorsum 函数:　factorsum 函数に似ていますが, factor 函数の代りに gfactor が適用されます:

```
(%i58) gfactorsum(x^2+1);
(%o58)                          (x - %i) (x + %i)
(%i59) factor(x^2+1);
                                      2
(%o59)                               x  + 1
```

ezgcd 函数:　, 第一成分が全ての多項式の GCD, 残りの元が GCD で割った値を成分に持つリストを返します. この ezgcd 函数では ezgcd アルゴリズムが常用されています.

content 函数:　二成分のリストを返し, このリストの第一成分が ⟨ 変数$_1$⟩ を ⟨ 多項式 ⟩ の主変数とした場合の各係数の最大公約因子, 第二成分を第一成分で多項式を割った monic な多項式となります:

```
(%i43) content(2*x*y+4*x^2*y^2,y);
                                              2
(%o43)                              [2 x, 2 x y  + y]
```

剰余を計算する函数

剰余を計算する函数

divide(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数$_1$⟩, ⋯ , ⟨ 変数$_n$⟩)
quotient(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数$_1$⟩, ⋯)
remainder(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数$_1$⟩, ⋯)
polymod(⟨ 多項式 ⟩)
polymod(⟨ 多項式 ⟩, ⟨ 整数 ⟩)

divide 函数:　⟨ 多項式$_2$⟩ による ⟨ 多項式$_1$⟩ の商と剰余を計算します. 各多項式は ⟨ 変数$_n$⟩ を主変数とし, その他の変数は ratvars 函数に現れるものとします. 結果はリストで返却され, 第一成分が商, 第二成分が剰余となります:

```
(%i1) divide(x+y,x-y,x);
(%o1)                                [1, 2 y]
(%i2)  divide(x+y,x-y);
(%o2)                               [- 1, 2 x]
```

quotient 函数: 〈多項式$_2$〉による〈多項式$_1$〉の商を計算します.

remainder 函数: 〈多項式$_2$〉による〈多項式$_1$〉の剰余を計算します.

polymod 函数: 〈多項式〉の第二引数の整数や, 大域変数 modulus で指定した値に対する剰余を計算します. ここで大域変数 modulus の既定値が 'false' のために, この大域変数 modulus に整数値が割当てられるまで, polymod 函数の引数は二つ必要です. 大域変数 modulus が既定値の 'false' のままで一つの引数で計算させようとするとエラーになります:

```
(%i14) polymod(expand((x+2)^3),3);
                                    3
(%o14)                             x  - 1
(%i15) modulus:3;
(%o15)                               3
(%i16) polymod(expand((x+2)^3));
                                    3
(%o16)                             x  - 1
(%i17) modulus:false$
(%i18) polymod(expand((x+2)^3));
Maxima encountered a Lisp error:


MINUSP: NIL is not a real number

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
```

終結式に関連する函数

多項式 f と g の解を α_i と β_j とすると, 多項式 f と g の終結式 $\mathtt{res}(f,g)$ は次の式と等しくなることが知られています:

$$\mathtt{res}(f,g) \stackrel{def}{=} a_m^n b_n^m \prod_{1\le i\le m, 1\le j\le n} (\alpha_i - \beta_j)$$

このことから終結式は〈多項式$_1$〉と〈多項式$_2$〉が共通の定数の因子を持つ場合に限って零になることが判ります.
終結式の計算方法は〈多項式$_1$〉と〈多項式$_2$〉を〈変数〉の多項式と看做した場合の係数から構成される行列の行列式から計算できます.
ここで行列の大きさは〈多項式$_1$〉と〈多項式$_p$〉の次数を m, および n としたときに $m+n$ 次の正方行列となります. bezout では行列操作によって, それよりも小さな行列が得られる場合に, その行列を表示します.
具体的には多項式 f と g を次のものとします:

$$f = \sum_{i=0}^{m} a_i x^i \quad g = \sum_{i=0}^{n} b_i x^i$$

すると次の式で多項式 $f(x)$ と $g(x)$ の終結式 `resultant`(f,g,x) が計算出来ます:

$$\texttt{resultant}(f,g,x) \stackrel{def}{=} \det\left(\begin{pmatrix} a_m & a_{m-1} & \cdots & \cdots & a_1 & a_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & a_m & a_{m-1} & \cdots & \cdots & a_1 & a_0 \\ b_n & b_{n-1} & \cdots & \cdots & b_1 & b_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & b_n & b_{n-1} & \cdots & \cdots & b_1 & b_0 \end{pmatrix}\right)$$

この終結式に関連する函数を次に纏めておきましょう:

終結式に関連する函数

resultant(⟨ 多項式$_1$⟩,⟨ 多項式$_2$⟩,⟨ 変数 ⟩)
bezout(⟨ 多項式$_1$⟩,⟨ 多項式$_2$⟩,⟨ 変数 ⟩)
eliminate([⟨ 方程式$_1$⟩,⟨ 方程式$_2$⟩,$\cdots$,⟨ 方程式$_n$⟩], [⟨ 変数$_1$⟩,⟨ 変数$_2$⟩,$\cdots$,⟨ 変数$_k$⟩])
poly_discliminant(⟨ 多項式 ⟩,⟨ 変数 ⟩)

bezout 函数: ⟨ 多項式$_1$⟩ と ⟨ 多項式$_2$⟩ に対して ⟨ 変数 ⟩ を主変数とした場合のある係数行列を返します. この係数行列の行列式を取ると終結式に等しくなるものです.
この行列 'bezout(f,g,x)' の determinant が多項式 f と g の終結式になります. したがって, bezout 函数と determinant 函数を組合せれば resultant 函数の代替になります.

resultant 函数: 二つの多項式 ⟨ 多項式$_1$⟩ と ⟨ 多項式$_2$⟩ の終結式を計算し, 指定した ⟨ 変数 ⟩ を消去します.
なお, ⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩ が因子分解可能であれば resultant 函数を呼び出す前に factor 函数を呼出すと良いでしょう.

eliminate 函数: 与えられた方程式, あるいは零と等しいと仮定した式から続けて終結式を取ることで指定された変数の消去を行います. eliminate 函数に引渡した k 個の ⟨ 変数$_1$⟩,$\cdots$,⟨ 変数$_k$⟩ を消去した $n-k$ 個の式のリストを返します. 最初の ⟨ 変数$_1$⟩ は消去されて $n-1$ 個の式を生成し, ⟨ 変数$_2$⟩ 以降も同様です.
$k=n$ の場合, 結果リストは k 個の ⟨ 変数$_1$⟩,...,⟨ 変数$_k$⟩ を持たない一つの式となります. そして最後の変数に対応する終結式を解くために solve 函数を呼出します:

```
(%i1) exp1:2*x^2+y*x+z$
(%i2) exp2:3*x+5*y-z-1$
(%i3) exp3:z^2+x-y^2+5$
(%i4) eliminate([exp3,exp2,exp1],[y,z]);
                8          7          6          5
(%o4) [7425 x  - 1170 x  + 1299 x  + 12076 x
                                4          3          2
                           + 22887 x  - 5154 x  - 1291 x
```

```
                              + 7688 x + 15376]

(%i5) eliminate([x+y=2,2*x+3*y-5=0],[x,y]);
(%o5)                                  [1]
(%i6) eliminate([x+y=2,2*x+3*y-5=0],[x]);
(%o6)                                [y - 1]
(%i7) eliminate([x+y=2,2*x+3*y+5=0],[x]);
(%o7)                                [y + 9]
(%i8) eliminate([x+y=2,2*x+3*y+5=0],[x,y]);
(%o8)                                 [- 9]
```

終結式のアルゴリズムを指定する大域変数

変数名	初期値	可能な値
resultant	subres	[subres,mod,red]

大域変数 resultant は同名の resultant 函数による終結式の計算で用いるアルゴリズムを設定します. 指定可能なアルゴリズムを以下に示しておきます.

値	概要
subres	既定値
mod	モジュラー終結式アルゴリズム
red	縮約 prs

殆どの問題では subres が最適です. 単変数の大きな次数や 2 変数問題では mod がより良いでしょう. poly_discriminant 函数は与えられた多項式を指定した変数を主変数として判別式を計算する函数です:

```
(%i146) poly_discriminant(x^2+1,x);
(%o146)                                - 4
(%i147) poly_discriminant(x^2+2*x+1,x);
(%o147)                                 0
(%i148) poly_discriminant(x^2+2*x*y^2+y+1,x);
                                   4
(%o148)                      4 y  - 4 y - 4
(%i149) poly_discriminant(x^2+2*x*y^2+y+1,y);
                                   3
(%o149)                    - 8 x  - 8 x + 1
```

Horner 則

多項式の表記で Horner 則に基く式の表記方法があります. これは X の多項式が与えられた場合に変数 X の次数の高い順番に項を並べます. たとえば, 与えられた多項式が $a_nX^n+a_{n-1}X^{n-1}+\cdots+a_1X+a_0$ とすると, $X((\cdots(a_nX+a_{n-1})+\cdots)+a_1)+a_0$ のように帰納的に変数 X の積で式を纏める方法です. この Horner 則を用いると式の積の回数を減らすことができるので, 複雑な多項式の数値計算を高速化する場合には非常に有効な手段の一つです.

horner 函数:　与式を Horner 表記に変換する函数です. 構文を次に纏めておきます:

Horner 表記に変換する函数

horner(⟨ 式 ⟩,⟨ 主変数 ⟩)
horner(⟨ 式 ⟩)

主変数を指定した場合, その主変数を用いて Horner 則を適用します. 式の主変数を指定しない場合は Maxima の変数順序 "$>_m$" に従って最大の変数を主変数として与式に Horner 則を適用します:

```
(%i3) expr:(x+2*y)^5,expand;
                   5          4        2  3        3  2        4       5
(%o3)          32 y  + 80 x y  + 80 x  y  + 40 x  y  + 10 x  y + x
(%i4) horner(expr,x);
                 5          4           3           2
(%o4)        32 y  + x (80 y  + x (80 y  + x (40 y  + x (10 y + x))))
(%i5) horner(expr);
                                                 2         3         4       5
(%o5)        y (y (y (y (32 y + 80 x) + 80 x ) + 40 x ) + 10 x ) + x
```

なお, 主変数として函数項を指定することも可能です:

```
(%i12) neko:(sin(x)+2*y)^5,expand$
(%i13) horner(neko,sin(x));
             5                  4                    3
(%o13) 32 y  + sin(x) (80 y  + sin(x) (80 y
                                2
               + sin(x) (40 y  + sin(x) (10 y + sin(x)))))
```

CRE 表現に関連する函数

CRE 表現の簡易化に関連する函数

ratexpand(⟨ 式 ⟩)
fullratsimp(⟨ 式 ⟩,⟨ 変数$_1$⟩,$\cdots$,⟨ 変数$_n$⟩)
fullratsimp(⟨ 式 ⟩n)
ratsimp(⟨ 式 ⟩)
ratsimp(⟨ 式 ⟩,⟨ 変数$_1$⟩,$\cdots$,⟨ 変数$_n$⟩)
partfrac(⟨ 式 ⟩,⟨ 変数 ⟩)

ratexpand 函数:　和の積や指数の和をかけ, 共通の分子で因子を纏め, 分子と分母の共通約数を通分し, 分子を分母によって割られた項へと分割して ⟨ 式 ⟩ の展開を行います. 実際は ⟨ 式 ⟩ を CRE 表現に変換し, それから一般形式に戻しています.
そのために ratexpand に影響を与える大域変数として, 大域変数 ratexpand, 大域変数 ratdenomdivide や大域変数 keepfloat といった CRE 表現への変換に影響を与える大域変数が該当します.

fullratsimp 函数: 与式の〈式〉に非有理式が含まれていれば, 普通, 簡易化した結果を返す際に, やや非力な非有理的 (一般的) 簡易化に続いて ratsimp 函数を呼出します. 時には, そのような呼出しが一回以上必要かもしれません. fullratsimp 函数は, この操作を簡易にしたものです.
fullratsimp 函数は非有理的簡易化に続けて ratsimp を式に変化が生じなくなるまで適用します.
たとえば, 式 'exp:(x^(a/2)+1)^2*(x^(a/2)-1)^2/(x^a-1)' に対し, ratsimp(exp) によって
'(x^(2*a)-2*x^a+1)/(x^a-1)' が得られ, ここで 'fullratsimp(exp)' を実行すれば 'x^a-1' が得られます.

ratsimp 函数: 非有理的函数に対して有理的に〈式〉とその部分式の全てを引数も含めて ratexpand 函数のように簡易化します. 結果は二つの多項式の商として再帰的な形式で返されます. すなわち, 主変数の係数は他の変数の多項式となっており, その係数もまた変数の順序に沿って主変数の次に順序の高い変数の多項式の係数と纏められています. 変数は ratexpand 函数のように有理式と異なる函数項 (たとえば, $\sin x^2+1$ を含みますが, ratsimp 函数で非有理的函数に対する引数は有理的に簡易化されます. ratsimp 函数は ratexpand 函数に影響を与える幾つかの大域変数の影響を受けることに注意して下さい.
なお, 'ratsimp(〈式〉,〈変数$_1$〉,...,〈変数$_n$〉)' で大域変数 ratvars に変数の〈変数$_1$〉,... を設定した場合と同様に, この変数の並びの順序で有理的簡易化を行います.

partfrac 函数: 与えられた有理式を有理式の和に分解する函数で, ratsimp 函数の逆操作を行う函数です. 第一引数が分解すべき式で, 第二引数に式の主変数を指定します:

```
(%i135) neko:(y/(x+1)+(y^2+x)/(y*(x+1))+1/(x+1)^2);
                                2
                               y  + x        y         1
(%o135)                      --------- + ----- + --------
                             (x + 1) y    x + 1          2
                                                  (x + 1)
(%i136) ratsimp(neko);
                                          2         2
                               (2 x + 2) y  + y + x  + x
(%o136)                       ------------------------
                                     2
                                  (x  + 2 x + 1) y
(%i137) partfrac(neko,x);
                                  2
                               2 y  - 1     1       1
(%o137)                       --------- + - + --------
                               (x + 1) y    y          2
                                                (x + 1)
(%i138) partfrac(neko,y);
                              (2 x + 2) y + 1        x
(%o138)                      --------------- + ---------
                                2                (x + 1) y
                               x  + 2 x + 1
```

一般表現と CRE 表現への変換を行う函数

Maxima には与えられた式を一般表現から CRE 表現, CRE 表現から一般表現に変換する函数が用意されています:

一般表現と CRE 表現への変換に関連する函数

rat(⟨ 式 ⟩,⟨ 変数$_1$⟩,...,⟨ 変数$_n$⟩)	一般表現	⇒	CRE 表現
ratdisrep(⟨ 式 ⟩)	CRE 表現	⇒	一般表現
totaldisrep(⟨ 式 ⟩)	CRE 表現	⇒	一般表現

rat 函数　与えられた ⟨ 式 ⟩ を展開し, 浮動小数点数を大域変数 ratepsilon で指定された許容範囲以内の有理数に変換し, 全ての項を共通の分母で纏め, 分子と分母の最大公約因子を除去します. ここで変数の順序は無指定であれば Maxima の順序 “$>_m$” に従いますが, ravtars 函数を用いて導入された順序があれば, その順序を用います. rat 函数は四則演算子 “+”, “-”, “*”, “/” と羃乗 “^” の他の函数を一般的には簡易化しません. CRE 表現での原子は一般形式のものと異なります. したがって, ‘rat(x)-x’ は ‘0’ と異なる内部表現の ‘rat(0’) で計算されます:

```
(%i1) exp1:rat(x)-x;
(%o1)/R/                                    0
(%i2) :lisp $exp1;
((MRAT SIMP ($X) (X13157)) 0 . 1)
(%i2) exp0:0;
(%o2)                                       0
(%i3) :lisp $exp0;
0
```

この rat 函数に大域変数 ratfac, 大域変数 ratprint, 大域変数 keepfloat といった直接動作に関連する大域変数を持ちます.

ratdisrep 函数:　CRE 表現から一般表現に引数を変換する函数です.

totaldisrep 函数:　CRE 表現から一般表現に ⟨ 式 ⟩ の全ての部分式を変換する函数です.

有理式の CRE 表現

有理式は多項式の分数ですが, 有理式の CRE 表現の表現は多項式の分母と分子に共通因子がなく, 分母の筆頭項 (leading term) の係数を正にしたものとなります.
次に実例を示しましょう:

```
(%i1) r1: rat((y-1)/((y-x)*z^2+1));
                                        y - 1
(%o1)/R/                            --------------
                                              2
                                    (y - x) z  + 1
(%i2) r2: rat((y-1)/((x-y)*z^2+1));
```

```
                                        y - 1
(%o2)/R/                          - --------------
                                            2
                                   (y - x) z  - 1
(%i3) r3:rat((y-1)/(-(y-z)*x^2+1));
                                        y - 1
(%o3)/R/                          ---------------
                                   2      2
                                  x  z - x  y + 1
(%i4) :lisp $r3;
((MRAT SIMP $(X $Y $Z) (X13180 Y13181 Z13182)) (Y13181 1 1 0 -1)
Z13182 1 (X13180 2 1) 0 (Y13181 1 (X13180 2 -1) 0 1))
(%i4) t3:(y-1)/(-(y-z)*x^2+1);
                                       y - 1
(%o4)                             --------------
                                   2
                                  x  (z - y) + 1
(%i5) :lisp $t3;
((MTIMES SIMP)((MPLUS SIMP) -1 $Y)
 ((MEXPT SIMP)
  ((MPLUS SIMP) 1
   ((MTIMES SIMP)((MEXPT SIMP) $X 2)((MPLUS SIMP)
    ((MTIMES SIMP) -1 $Y) $Z)))
  -1))
(%i5)
```

この例では変数順序 “$>_m$” が逆アルファベット順のために変数順序は z $>_m$ y $>_m$ x になります. そのため, 変数 z が主変数となり, 式は変数 z の多項式として纏められます. 最初の二つの例では, ‘(x-y)*z^2’ は, 式を構成する変数の順序が, ‘y $>_m$ x’ となるため, 自動的に ‘-(y-x)*z^2’ に並び替えられてしまいます. この際に最も順序の高い項となる ‘y* z^2’ の係数を正にするために必要に応じて ‘-1’ がかけられています. この例で示すように式が CRE 表現, あるいは CRE 表現の部分式を含む場合, 記号 “/R/” が行ラベルに続きます.

:lisp $r3; で CRE 表現の内部表現を表示しています. 先頭の MRAT で CRE 表現であることを示し, その後のリストで有理式の変数が X, Y,Z, これらの変数に対応する内部変数が X13180, Y13181 と Z1382 であることを示しています. ここで内部変数は LISP の gensym 函数で生成されるもので, うしろの番号は生成順に付けられるものです. なお, 有理式の変数は Maxima の showvar 函数を使って取り出すことが可能ですが, それらの変数に対応する内部変数は Maxima の内部変数 genvar に登録されるので, :lisp genvar で参照できます

また, 変数リストの順版は順序 “$>_m$” に対して小さい順に左から並ぶために左端が最小で右端が主変数となります. そのため, この式では Z が主変数となります. このような変数リストを含むリスト ‘(MAT ...)’ のうしろに分子となる式が来ます. この式は先頭が Y13181 のために変数 Y の多項式で, うしろの ‘1 1 0 -1’ から次数が ‘1’ でその係数が ‘1’ の項と, 次数が ‘0’ で係数が ‘-1’ であることが判ります. 最後の副リストが分母の式となり, 先頭が Z13182 となっているので, 主変数 Z の多項式であることが判ります. 以降, 分子のときと同様に読めば良いのです. ただし, 式は Maxima の変数順序 “$>_m$” に従って読む必要があります.

この例では ‘Z $>_m$ Y $>_m$ X’ で順序付けられているので, 変数 Z が式中にあれば, 変数 Z の多項式として表現し, 変数 Z がなくて変数 Y があれば変数 Y の多項式, そして, 変数 Z と Y の両方が無けれ

ば変数 X の多項式と帰納的に解釈します.
これに対し, 同じ多項式の一般表現を最後に示しますが, 非常に長いものになっていることが判ります.
なお, CRE 表現で分母が整数の場合 (CRE 表現では浮動小数点数は有理数で近似されます), CRE 表現の内部表現は少し変化します:

```
(%i4) r1:rat((x-1)/5);
                                        x - 1
(%o4)/R/                                -----
                                          5
(%i5) :lisp $r1;
((MRAT SIMP $(X) (X13157)) (X13157 1 1 0 -1) . 5)
```

この例で示すように分子は 'x-1' ですが, 分子と分母の間に記号 "." が入っています. CRE 表現では分子と分母の間に, 分母が整数のときに限って記号 "." を入れています. これは CRE 表現の生成で LISP の cons 函数が用いられていることを示しています. また変数 X のうしろに数値が入っていますが, これは LISP 内部の処理で変数 X に対応する表象を生成した際に割当てられた通し番号です.
拡張 CRE 表現は taylor 級数を表現するために用いられています. 有理式の表記は正の整数ではなく, 正か負の有理数となる様に拡張されており, 係数はそれ自身が多項式と云うよりは, 上で記述したような有理式となっています. これらは内部的に再帰的な多項式形式によって表現され, 多項式形式は CRE 表現に類似していますが, より一般化したものです. なお, 切り捨てられる次数のような情報も追加されています.
式を表示する際に, 拡張 CRE 表現の場合は記号 "/T/" が式の行ラベルに続きます:

```
(%i1) t1:taylor(exp(x),x,0,5);
                                    2    3    4    5
                                   x    x    x    x
(%o1)/T/                1 + x + -- + -- + -- + --- + . . .
                                   2    6    24   120
(%i2) :lisp $t1;
((MRAT SIMP (((MEXPT SIMP) $%E $X) $X) (%e^x13162 X13163)
  ($(X ((5 . 1)) 0 NIL X13163 . 2)) TRUNC)
 PS (X13163 . 2) ((5 . 1)) ((0 . 1) 1 . 1)
                 ((1 . 1) 1 . 1) ((2 . 1) 1 . 2)
 ((3 . 1) 1 . 6) ((4 . 1) 1 . 24) ((5 . 1) 1 . 120))
```

雰囲気は CRE 表現に近いものですが, 係数と冪のリストの書式が cons で結合されたリストであり, 他にリストの第一成分に Taylor 展開のさまざまな情報が追加されていることが判ります.

CRE 表現変換に関連する大域変数

CRE 表現変換に関連する大域変数

変数名	初期値	概要
keepfloat	false	実係数の有理数への近似を制御
ratepsilon	2.0E-8	実係数の有理数近似する際の誤差を指定
ratalgdenom	true	代数的整数を分母とする項の有理化を制御
ratprint	true	CRE 表現変換時のメッセージを制御

大域変数 keepfloat: true であれば浮動小数点数を含む式が CRE 表現に変換される際に浮動小数点数が有理数に近似変換されることを防ぎます. なお, 浮動小数点数が有理数に近似される際に生じる誤差は大域変数 ratepsilon で制御されます.

大域変数 ratepsilon: 式を CRE 表現に変換する際に, 係数を有理数に変換するときに用いられる許容範囲を指定します. 大域変数 ratepsilon よりも小さな浮動小数点数は無視されます. 浮動小数点数を有理数に変換したくなければ大域変数 keepfloat の値として 'true' を設定します:

```
(%i30) ratepsilon;
(%o30)                                2.0e-8
(%i31) ratsimp((1+2.0e-8)*x);

rat replaced 1.00000002 by 1/1 = 1.0
(%o31)                                   x
(%i32) ratsimp((1+2.0e-7)*x);

rat replaced 1.0000002 by 5000001/5000000 = 1.0000002
                                      5000001 x
(%o32)                                ---------
                                      5000000
```

この例で示すように ratsimp 函数を作用させた場合に大域変数 ratepsilon よりも小さな数が無視され, 浮動小数点数が有理数に変換されていることが判ります。

大域変数 ratalgdenom: 式中に代数的整数を項の分母として持つ式に対して大域変数 ratalgdenom の値が 'true' の場合に, その分母を有理化します. これを実行するためには大域変数 algebraic を 'true' に設定し, 式を CRE 表現に変換しておく必要があります:

```
(%i16) algebraic:true;
(%o16)                                true
(%i17) ratalgdenom:true;
(%o17)                                true
(%i18) rat(1/sqrt(2)*x^2+1);
                                          2
                                   sqrt(2) x  + 2
(%o18)/R/                          ---------------
                                         2
(%i19) ratalgdenom:false;
```

```
(%o19)                                  false
(%i20) rat(1/sqrt(2)*x^2+1);
                                         2
                                        x  + sqrt(2)
(%o20)/R/                               ------------
                                          sqrt(2)
```

この例で示すように大域変数 algebraic と大域変数 ratalgdenom を同時に true にすると, 分母に $\sqrt{2}$ を持つ式の分母が有理化されていることに注意して下さい.
大域変数 ratprint が false であれば, 浮動小数点数の有理数への変換を報せるメッセージ出力を抑制します.

CRE 表現式の処理に関連する大域変数

CRE 表現式の処理に関連する大域変数

変数名	初期値	概要
ratdenomdivide	true	分子の項の分離を制御
ratexpand	false	CRE 表現の展開を制御
ratfac	false	CRE 表現式の因子分解を制御
ratsimpexpons	false	ratsimp の自動実行を制御
ratwtlvl	false	近似の際の切捨てを制御
ratweights	[]	重みのリスト
rootsconmode	true	rootscontract 函数を制御
psexpandnd	false	CRE 表現の展開を制御

大域変数 ratdenomdivide: 'false' であれば ratexpand 函数を作用させた式に対して, 分子の項を分離することを抑制します.

大域変数 ratexpand: 'true' であれば, それらが一般形式に変換されるか表示されたときに CRE 式が展開されます.

大域変数 ratfac: 'true' であれば, CRE 有理式に対して部分的に因子分解された形式で出力します. 有理的操作の間に factor 函数を実際に呼ばずに式を可能な限り因子分解します. これでメモリ空間を節約し, 計算時間を幾らかを節約します. 有理式の分子と分母は互いに素とします.
たとえば, 式 'rat((x^2 -1)^4/(x+1)^2)' は '(x-1)^4*(x+1)^2' になりますが, 各部分の因子は互いに素とは限りません.
なお, 大域変数 ratfac と大域変数 ratweights の手法は互換性がないので両者を同時に使っていけません.

大域変数 ratsimpexpons: 'true; であれば簡易化中に式の冪に対し, 自動的に ratsimp 函数が実行されます.

大域変数 ratwtlvl: ratweight 函数を用いた式を纏める際に CRE 表現の切捨ての制御で用いられます. 'false; の場合は切捨は生じません.

大域変数 ratweights: ratweight 函数で設定した変数と対応する重みのリストです. 大域変数 ratweights や ratweight() でそのリストが見られます.

大域変数 rootsconmode: この大域変数は rootscontract 函数 の挙動を定めます. 大域変数 rootsconmodefalse ならば rootscontract 函数は有理数次数の分母が同じ次数の冪だけを纏めます. true の場合に次数の分母が割切れる冪だけを纏めます. そして, all の場合は全ての有理次数の分母の LCM を取って纏めます:

式	rootsconmode	rootscontract の結果
x^(1/2) * y^(3/2)	false	(x * y^3)^(1/2)
x^(1/2) * y^(1/4)	false	x^(1/2) * y^(1/4)
x^(1/2) * y^(1/4)	true	(x * y^(1/2))^(1/2)
x^(1/2) * y^(1/3)	true	x^(1/2) * y^(1/3)
x^(1/2) * y^(1/4)	all	(x^2 * y)^(1/4)
x^(1/2) * y^(1/3)	all	(x^3 * y^2)^(1/6)

大域変数 psexpand: ratexpand と似た作用となります. 大域変数 psexpand の値が 'true' の場合に式全体の展開が実行されます. 大域変数 psexpand の値が 'multi' の場合, 総次数が同じ項毎で纏めて表示されます.

6.2.5 有理式に関連する函数

有理式に関連する函数

combine(⟨ 式 ⟩)
denom(⟨ 有理式 ⟩)
num(⟨ 有理式 ⟩)
ratdenom(⟨ 有理式 ⟩)
ratnumer(⟨ 有理式 ⟩)
ratdiff(⟨ 有理式 ⟩,⟨ 変数 ⟩)
xthru(⟨ 式 ⟩)

combine 函数: ⟨ 式 ⟩に含まれる和の部分式を同じ分母で纏めて一つの項にします. なお, combine 函数の制御で大域変数 combineflag が用いられています. この大域変数 combineflag の既定値は 'true' で, 有理式の分母が整数の場合に共通の分母で式を纏めますが, 'false' の場合はそのままにします:

```
(%i25) combineflag:true$
(%i26) combine(1/4+(y+1)^4/2);
                                        4
                                  2 (y + 1)  + 1
(%o26)                            --------------
                                        4
(%i27) combineflag:false$
(%i28) combine(1/4+(y+1)^4/2);
                                        4
                                  (y + 1)    1
(%o28)                            -------- + -
                                     2       4
```

denom 函数: 〈有理式〉の分母 (DENOMinator) を返します. なお, 有理式が通常の多項式であれば '1' を返します:

```
(%i40) denom((x^2+1)/(y^2+1)/2);
                                        2
(%o40)                                2 (y  + 1)
(%i41) denom(x^2+1);
(%o41)                                    1
(%i42) denom(1/2*x^2+1/2);
(%o42)                                    1
(%i43) denom((x^2+1)/2);
(%o43)                                    2
```

num 函数: 有理式の分子 (NUMerator) を返します.

ratdenom 函数: 〈有理式〉の分母を計算します. 〈有理式〉が一般形式で結果も一般形式のものが必要ならば, denom 函数を用いましょう.

ratnumer 函数: 〈有理式〉の分子を取り出します. 一般形式の〈有理式〉に対して CRE 表現の結果が不要であれば, num 函数を使いましょう.

ratdiff 函数: 〈有理式〉の微分を〈変数〉で行います. 有理式に対しては diff 函数よりも処理が速く, 計算結果は CRE 表現になります. なお, ratdiff 函数は因子分解された CRE 表現には使ってはいけません. 因子分解された式では通常の diff 函数を使いましょう.

xthru 函数: 有理式を一つの分母で纏める函数です. このときに分母の展開を実行せずに多項式を纏めます:

```
(%i34) xthru(1/x+1/y+1/(x*y));
                                    y + x + 1
(%o34)                              ---------
                                       x y
(%i35) xthru(1/(x-1)^2+1/(y+1)^3-(y+1)^2/(x*(x-1)));
```

```
                          3          2                    5
                  x ((y + 1)  + (x - 1) ) - (x - 1) (y + 1)
(%o35)            ------------------------------------------
                                  2          3
                          (x - 1)  x (y + 1)
```

6.2.6 その他の函数

trunc 函数の構文

trunc(⟨ 式 ⟩)

trunc 函数: 内部表現が演算子 “+” で括られた式を引数とし, その式に対して級数のような表示を行います. ただし, 内部表現が演算子 “+” で括られていない式はそのまま返します.
なお, CRE 形式や Taylor 級数に対しては型の変更が生じ, 通常の多項式の型に変更されます.
このことを実際に確認して見ましょう:

```
(%i17) a1:x^3+3*x^2+3*x+1;
                                 3       2
(%o17)                          x  + 3 x  + 3 x + 1
(%i18) b1:trunc(a1);
                                          2     3
(%o18)                      1 + 3 x + 3 x  + x  + . . .
(%i19) :lisp $a1
((MPLUS SIMP) 1 ((MTIMES SIMP) 3 $X) ((MTIMES SIMP) 3 ((MEXPT SIMP) $X 2))
 ((MEXPT SIMP) $X 3))
(%i19) :lisp $b1
((MPLUS SIMP TRUNC) 1 ((MTIMES SIMP) 3 $X)
 ((MTIMES SIMP) 3 ((MEXPT SIMP) $X 2)) ((MEXPT SIMP) $X 3))
(%i19) is(a1=b1);
(%o19)                                  true
```

このように通常の式に対しては内部書式で式の型を示すヘッダの末尾に TRUNC が追加されるだけです.
ところが CRE 形式や Taylor 級数に対しては動作が異なります:

```
(%i1) t1:taylor(sin(x),x,0,5);
                                   3     5
                                  x     x
(%o1)/T/                       x - -- + --- + . . .
                                  6     120
(%i2) r1:taytorat(t1);
                                 5        3
                                x  - 20 x  + 120 x
(%o2)/R/                        ------------------
                                       120
(%i3) tr1:trunc(t1);
                                   3     5
                                  x     x
```

```
(%o3)                          x - -- + --- + . . .
                                   6     120
(%i4) tr2:trunc(r1);
                                 5        3
                                x  - 20 x  + 120 x
(%o4)                            ------------------
                                        120
(%i5) :lisp $tr1;
((MPLUS SIMP TRUNC) $X
 ((MTIMES SIMP) ((RAT SIMP) -1 6) ((MEXPT SIMP RATSIMP) $X 3))
 ((MTIMES SIMP) ((RAT SIMP) 1 120) ((MEXPT SIMP RATSIMP) $X 5)))
(%i5) :lisp $tr2
((MTIMES SIMP) ((RAT SIMP) 1 120)
 ((MPLUS SIMP) ((MTIMES SIMP RATSIMP) 120 $X)
  ((MTIMES SIMP) -20 ((MEXPT SIMP RATSIMP) $X 3)) ((MEXPT SIMP RATSIMP) $X 5)))
```

この例で示すように内部書式では型の変更が生じています.

6.3 級数の扱い

6.3.1 Maxima に於ける級数の表現

Maxima では級数の扱いは二通りあります. 一つは sum 函数を用いた形式的な級数の表記方法です. この場合, 基本的に多項式の表記とさほどの違いはありません. もう一つの方法は Taylor 級数を用いる方法です.
ここで形式的級数を生成する函数として powerseries 函数があります.

powerseries 函数: 形式的冪級数を求める函数です. 出力される函数は有理式の無限和を sum 函数を用いて表現したものになります. したがって形式的な級数です. powerseries 函数は三角函数や双曲函数の級数展開した情報を持っています. これらの情報を用いて与えられた式の冪級数展開を行いますが, 展開が容易にできない場合には “Unable to expand” と表示します.

powerseries 函数の構文

powerseries(⟨ 函数 ⟩,⟨ 変数 ⟩,⟨ 級数展開を行う点 ⟩)

```
(%i10) neko:powerseries(sin(2*x)*exp(x),x,0);
                 inf          inf
                 ====    i8   ====         i8  2 i8 + 1  2 i8 + 1
                 \       x    \       (- 1)   2          x
(%o10)           ( >     ---)  >     ----------------------------
                 /       i8!  /               (2 i8 + 1)!
                 ====         ====
                 i8 = 0       i8 = 0
(%i11) neko:powerseries(sin(2*log(x)),x,0);
(%o11)                          Unable to expand
```

6.3.2 Taylor 級数の内部表現

Maxima の Taylor 級数は通常の多項式表現を拡張したものではなく, より高速な処理の行える CRE 表現を拡張したもので表現されています.
まず, Maxima の Taylor 級数は無限の長さを持つ級数として表現されます. そして, 与件型が Taylor 級数であることを示すために表示の際にラベル “/T/” が先頭に付けられます. 実際,sin 函数を原点 0 の回りで展開した例を示しておきましょう:

```
(%i5) taylor(sin(x),x,0,10);
                                3    5     7      9
                               x    x     x      x
(%o5)/T/                 x - -- + --- - ---- + ------ + . . .
                               6    120   5040   362880
```

この式の内部表現を次に示しておきます:

```
(%i7) :lisp $t1
((MRAT SIMP (((%SIN SIMP) $X) $X)
  (cos(1)15731 sin(1)15730 cos(1)15729 sin(1)15728 cos(x0)15727 sin(x0)15725
   X015723 sin(x)15721 X15722)
  (($X ((10 . 1)) 0 NIL sin(1)15730 . 2)) TRUNC)
 PS (sin(1)15730 . 2) ((10 . 1)) ((1 . 1) 1 . 1) ((3 . 1) -1 . 6)
 ((5 . 1) 1 . 120) ((7 . 1) -1 . 5040) ((9 . 1) 1 . 362880))
```

この内部表現から Taylor 級数は CRE 表現を拡張した書式であることが明瞭になります. したがって, Taylor 級数から通常の CRE 表現への変換も簡単に行えます. Taylor 級数を CRE 表現に taytorat 函数を用いて変換した例を示しておきましょう:

```
(%i7) r1:taytorat(t1);
                      9       7         5          3
                     x  - 72 x  + 3024 x  - 60480 x  + 362880 x
(%o7)/R/             ------------------------------------------
                                    362880
(%i8) :lisp $r1
((MRAT SIMP (((%SIN SIMP) $X) $X)
  (cos(1)15731 sin(1)15730 cos(1)15729 sin(1)15728 cos(x0)15727 sin(x0)15725
   X015723 sin(x)15721 X15722))
 (sin(1)15730 9 1 7 -72 5 3024 3 -60480 1 362880) . 362880)
```

6.3.3　taylor 函数

Maxima で Taylor 級数を生成することができる函数は taylor 函数のみです. この taylor 函数の構文を次に纏めておきます:

taylor 展開を行う函数の構文

taylor(⟨ 式 ⟩,⟨ 変数 ⟩,⟨ 座標 ⟩,⟨ 正整数値 ⟩)
taylor(⟨ 式 ⟩,[⟨ 変数 ⟩,⟨ 座標 ⟩,⟨ 正整数値 ⟩,asymp])
taylor(⟨ 式 ⟩,[⟨ 変数$_1$⟩,...,⟨ 変数$_n$⟩],⟨ 座標,⟨ 正整数値 ⟩)
taylor(⟨ 式 ⟩,[⟨ 変数$_1$⟩,...,⟨ 変数$_n$⟩],[⟨ 座標$_1$⟩,...,⟨ 座標$_n$⟩],⟨ 正整数値 ⟩)
taylor(⟨ 式 ⟩,[⟨ 変数$_1$⟩,...,⟨ 変数$_n$⟩],[⟨ 座標$_1$⟩,...,⟨ 座標$_n$⟩],[⟨ 正整数値$_1$⟩,...,⟨ 正整数値$_n$⟩])
taylor(⟨ 式 ⟩,[⟨ 変数$_1$⟩,⟨ 座標$_1$⟩,⟨ 正整数値$_1$⟩],⋯ ,[⟨ 変数$_n$⟩,⟨ 座標$_n$⟩,⟨ 正整数値$_n$⟩])

函数の変数と展開を行う点を指定し, 級数表示を行う次数の指定を行います. すると, 指定した次数を越えない次数の級数として Taylor 級数の表示を行います.

6.3.4 Taylor 級数に関連する函数

taylor 級数に関連する函数

taytorat(⟨ 式 ⟩)
pade(⟨*Taylor* 級数 ⟩,⟨ 整数$_1$⟩,⟨ 整数$_2$⟩)
taylor_simplifier(⟨ 式 ⟩)
taylorinfo(⟨ 式 ⟩)
taylorp(⟨ 式 ⟩)

taytorat 函数: taylor 展開の内部表現を CRE 表現に変換する函数です.

pade 函数: 与えられた Taylor 級数を近似表現する有理式を返す函数です. 第 2 引数の整数が近似有理式の分子の多項式の最高次数の上限を定め, 第 3 引数の整数が近似有理式の分母の多項式の最高次数の上限を定めます.

taylorinfo 函数: 与式が taylor 展開式であればその情報を返し, taylor 展開式でなければ 'false' を返す函数です.

taylorp 函数: 真理値集合を {true,false} とする真理函数です.

6.3.5 Taylor 級数に関連する大域変数

taylor 級数に関連する大域変数

大域変数	値	概要
maxtayorder	true	
taylordepth	3	taylor 展開で指定された次数 n に対し $n2^{taylordepth}$ 次の項までを計算.
taylor_logexpand	true	
taylor_order_coefficients	true	taylor 級数の係数順序を制御
taylor_truncate_polynomials	true	taylor 級数表示を制御

6.4　式について

Maxima の式は §5.1 の §5.1.14 で述べたように Maxima の記号や数値を演算子や函数項で繋ぐことで構築された対象です.
ところで Maxima の式は二面性を持っています. 一つは Maxima が表示する通常の数式に近い表示であり, もう一つは Maxima を記述する LISP の処理に適したデータ構造, すなわち, LISP の S 式に近い内部表現です. 実際の Maxima の処理では, 単に表に現われる式の操作ではなく, 内部表現を念頭に置いて処理を行う方が動作を理解し易くなります.

6.4.1　変数や文字列の内部表現

まず, 式を構成する上で基礎となる変数と文字列の内部表現を示しましょう:

Maxima の内部表現：変数

Maxima	内部表現
a	$A
?a	A
″a″	″a″
‘a	((MQUOTE) $A)

Maxima の変数 (記号) は内部で先頭に文字 “$” を付与したものです. 一方で Maxima で文字 “?” が先頭に付く対象は, Maxima 内部では文字 “?” が外されています. そのために LISP の函数を Maxima から実行させる場合は先頭に文字 “?” を付けてやれば使えます. ただし, その函数の引数は Maxima を通して与えられるために Maxima の書式で引数を与える必要があります.
次に名詞形’a の内部表現は “((MQUOTE) $A)” です. ここで変数 a が “$a” となるのは理解出来ますが, 何故, “(MQUOTE)” で “MQUOTE” でないのでしょうか？ここで “(MQUOTE)” はリストで, その被演算子$a は原子になりますね. こうすることで函数と被演算子を区別できます. すなわち, 函数が被演算子よりも一階高い対象であることが視覚的にも表現できるのです. そして, この考え方は Maxima の内部表現で一貫しています.
注意が必要なのは文字列の扱いです. Maxima-5.13.0 以前では Maxima の文字列と LISP の文字列は別物です. たとえば, Maxima の文字列 ″ABCD1″ は先頭に文字 “&” を付けてた ‘&ABCD1’ が内部表現でした. しかし, Maxima-5.14.0 以降から Maxima の文字列と LISP の文字列が一致する仕様に変更されています. この点は実用上の理由と思われますが, 抽象化の面では後退していると私は思います. この点は使っている環境毎に確認するようにして下さい.

6.4.2　二項演算の内部表現

代表的な二項演算子 (内挿表現の演算子) の内部表現を示します:

Maxima の内部表現：数値演算

Maxima の式	内部表現
x + y	((MPLUS SIMP) $X $Y)
x - y	((MPLUS SIMP) $X ((MMINUS SIMP) $Y))
x * y	((MTIMES SIMP) $X $Y)
x / y	((MTIMES SIMP) $X ((MEXPT SIMP) $Y -1))
x . y	((MNCTIMES SIMP) $X $Y)
x^2 または x**2	((MEXPT SIMP) $X 2)
x^^2	((MNCEXPT SIMP) $X 2)

基本的に 変数$_1$ 演算子名 変数$_2$ の内部表現は ((演算子名) 変数$_1$ 変数$_2$) となります. なお, Maxima 組込の演算子名は全て “m” から開始していることに注意して下さい. このように Maxima では “m” を先頭につけて Maxima の函数と LISP の函数を区別しています. さらに, 演算子名は () で括られ, 演算子が被演算子よりも一階高い対象であることを明瞭に示しています.
ここで, 差 ‘x - y’ は ‘x + (-y)’ と内部で表現されていることに注意して下さい.
さて, 式 ‘x -y ’ の表現を実際に Maxima で調べてみましょう. 使う函数は LISP の基本の基本, car 函数と cdr 函数です. Maxima から LISP 函数を利用する方法には演算子 “?” を使って Maxima の函数として LISP の函数を利用する方法と演算子 “:lisp” を使って直接 LISP の函数を利用する方法があります. ここでは, 演算子 “?” を使う方法で調べてみましょう:

```
(%i74) ?car(x-y);
(%o74)                          ("+", simp)
(%i75) ?cdr(x-y);
(%o75)                           (x, - y)
```

演算子 “?” を用いて LISP の函数を実行すると, その結果は Maxima で解釈された結果が帰って来ます. この例からも式 x-y が Maxima 内部では ‘x+(-y)’ に対応する式で置き換えられていることが分りますね. では, 今度は演算子 “:lisp” を使って調べてみましょう. この場合, 演算子 “:lisp” のうしろには LISP の S 式を記述します:

```
(%i11) a:x-y$

(%i12) :lisp $a

((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y))
(%i12) :lisp (car $a)

(MPLUS SIMP)
(%i12) :lisp (cdr $a)

($X ((MTIMES SIMP) -1 $Y))
```

この例に示すようにすると式 ‘x - y’ の内部表現が明瞭に分りますね. 演算子 “?” で返却された値はここでの例で示した内部表現を Maxima 側で変換して出力したもので, 実際に Maxima が処理している式はこの内部表現です.
次に論理演算子の一覧を示しますが, 内容的には数値演算のものと同様です:

Maxima の内部表現：論理演算子

Maxima の式	内部表現
not a	((MNOT SIMP) $A)
a or b	((MOR SIMP) $A $B)
a and b	((MAND SIMP) $A $B)
a = b	((MEQUAL SIMP) $A $B)
a >b	((MGREATER SIMP) $A $B)
a >= b	((MGEQP SIMP) $A $B)
a <b	((MLESSP SIMP) $A $B)
a <= b	((MLEQP SIMP) $A $B)
a # b	((MNOTEQUAL SIMP) $A $B)

6.4.3　割当の演算子の内部表現

割当の演算子にも内部表現があります:

Maxima の内部表現：割当の演算子

Maxima の式	内部表現
a:b	((MSETQ SIMP) $A $B)
a::b	((MSET SIMP) $A $B)
a(x):=f	((MDEFINE SIMP) (($A) $X) $F)

最初の式 ‘a:b’ は内部で “((MSETQ SIMP) $A $B)” と表現されていますが, 余計な括弧と “SIMP” を外してしまえば “(MSETQ $A $B)” となって LISP の ‘(setq a b)’ に対応することが分りますね. 函数定義を行う演算子 “:=” も LISP の ‘(defun a(x) f)’ に似た並びで内部表現されていることが判ります. 勿論, ここでも函数名は変項よりも高い階数を表現するために括弧 “( )” を用いて囲うことが徹底されていますね.

6.4.4　Maxima の函数の内部表現

Maxima の内部表現：函数

Maxima の式	内部表現
a(x)	(($A SIMP) $X)
sin(x)	((%SIN SIMP) $X)
diff(y,x)	(($DIFF SIMP) $Y $X 1)
diff(y,x,2,z,1)	(($DIFF SIMP) $Y $X 2 $Z 1)
‘diff(y,x)	((%DERIVATIVE SIMP) $Y $X 1)
integrate(a,b,c,d)	(($INTEGRATE SIMP) $A $B $C $D)
‘integrate(a,b,c,d)	((%INTEGRATE SIMP) $A $B $C $D)

普通の函数項の内部表現では, 函数名の先頭に文字 “$” が付きますが, 単引用符’ が先頭に付いた名詞形の函数項の場合, 内部表現では, その函数名の先頭が文字 “%” で置換えられます. 逆に言えば, 函数項の内部表現で, その函数名の先頭に文字 “%” が付いた函数項は名詞型と解釈され, Maxima は解釈を行いません. また, 函数名は括弧 “( )” で括られることで変項よりも一階高い対象であることが, ここでも視覚的にも表現されています.
微分を行う diff 函数の例に示すように一階の微分を行う場合, 階数の ‘1’ を省略しても構いませんが内部的には補完されていることが分ります:

```
(%i93) ?car('diff(x,y));
(%o93)                          (derivative, simp)
(%i94) ?cdr('diff(x,y));
(%o94)                               (x, y, 1)
```

nounify 函数と verbify 函数

名詞型に関連する函数として, nounify 函数と verbify 函数があります:

nounify 函数と verbify 函数

nounify(⟨ 記号 ⟩)
verbify(⟨ 記号 ⟩)

nounify 函数: Maxima の記号を名詞化して返す函数です. 内部的には先頭の文字 “$” を文字 “%” で置換する函数です.

verbify 函数: nounify 函数の逆操作を行う函数が verbify 函数で, 内部的には先頭の文字 “%” を文字 “$” で置換する函数です:

```
(%i9) test1:a$
(%i10) :lisp $test1;
$A
(%i10) test2:nounify(a)$
(%i11) :lisp $test2;
%A
(%i11) test3:verbify(nounify(a))$
(%i12) :lisp $test3;
$A
```

ここで函数名は記号であるため, これらの函数を用いれば函数項の名詞化や動詞化が容易に行えます.

6.4.5　配列とリストの内部表現

Maxima の内部表現：函数

Maxima の式	内部表現
a[1,2]	(($A SIMP ARRAY) 1 2)
a[1,2](x)	((MQAPPLY SIMP) (($A SIMP ARRAY) 1 2) $X)
[a,b,c]	((MLIST SIMP) $A $B $C)

まず, 配列の内部表現の第一成分は最初に文字 “$” が付けられた配列名があり, そのうしろに毎度の “SIMP” と最後の “ARRAY” で与件が配列であることを示しています.
Maxima のリストも LISP のリストとは別の構造で, 他の演算子等と同様に通常のリストに “(MLIST SIMP)” が先頭に置かれ, 変数には何時もの文字 “$” が先頭に付加されています.

6.4.6　Maxima の制御文の内部表現

ここでは if 文, for 文と block 文の内部表現を示しましょう:

if 文:　if 文の場合, LISP の cond と関連付けて mcond となっています:

Maxima の内部表現：if 文

if a then b	((MCOND SIMP) $A $B T NIL)
if a then b else c	((MCOND SIMP) $A $B T $C)

for 文:　for 文は LISP の do と関連付けて mdo となっています:

Maxima の内部表現：for 文

for i:a thru b step c unless q do f(i)	((MDO SIMP) $I $A $C NIL $B $Q (($F SIMP) $I))
for 1:a next n unless q do f(i)	((MDO SIMP) $I $A NIL $N NIL $Q (($F SIMP) $I))
for i in l do f(i)	((MDOIN SIMP) $I $L NIL NIL NIL NIL (($F SIMP) $I))

block 文:　block 文は LISP の prog と関連付けて mprog となっています:

Maxima の内部表現：block 文

block([l1,l2],s1,s2)	((MPROG SIMP) ((MLIST SIMP) $L1 $L2) $S1 $S2)
block(s1,s2)	((MPROG SIMP) $S1 $S2)

表に示すように Maxima の各制御文の内部表現は S 式になっています. このように S 式で全てが記述されており, その S 式を裏の LISP が解釈しているのです. そして, Maxima はその式をより人間

が理解し易い書式に変換しているのです. したがって, Maxima の入力では内部表現を考慮しながら Maxima に指示を与えてやれば, Maxima はより効率的に式を処理できることになるのです.

6.4.7 表示式と内部表現

式の内部表現では, 式を構成する演算子項や函数項が全て前置式に変換されますが, この他にも表示式と内部表現が微妙に異なる書式を取ることがあります. すなわち, 表示される式が必ずしも内部表現を反映したものではないことがあります.
ここで代表的な表現の違いを次に示しておきましょう:

式の表現の違い

入力	part	inpart
$a-b$	$a-b$	$a+(-1)b$
a/b	$\frac{a}{b}$	$a\,b^{-1}$
sqrt(x)	sqrt(x)	$x^{1/2}$
$x*4/3$	$\frac{4x}{3}$	$\frac{4}{3}x$
exp(x)	$\%e^{x}$	$\%e^{x}$
1/exp(x)	$\%e^{-x}$	$\%e^{-x}$

この表で入力列が Maxima に入力する式, part 列が Maxima で表示される式, 最後の inpart 列が内部表現に対応する式となっています.
これは式を和 "+" で項に分解し, 項は積で纏めるという方針が反映されたものです. そのために後述の part 函数と inpart 函数では結果が異なるのです.

内部表現の変換函数

dispform 函数: 実際に表示される式に適合するように内部表現変を換する函数として dispform 函数があります:

dispform 函数

dispform(⟨ 式 ⟩)
dispform(⟨ 式 ⟩,all)

この dispform 函数は引数の ⟨ 式 ⟩ の内部表現を式の表示をそのまま反映した内部表現に変換する函数です:

```
(%i23) exp1:x-y;
(%o23)                                x - y
(%i24) exp2:dispform(exp1);
(%o24)                                x - y
(%i25) :lisp $exp1
((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y))
(%i25) :lisp $exp2
((MPLUS SIMP) ((MMINUS) $Y) $X)
(%i25) exp1:x/y;
```

```
                                        x
(%o25)                                  -
                                        y
(%i26) exp2:dispform(exp1);
                                        x
(%o26)                                  -
                                        y
(%i27) :lisp $exp1;
((MTIMES SIMP) $X ((MEXPT SIMP) $Y -1))
(%i27) :lisp $exp2;
((MQUOTIENT SIMP) $X $Y)
```

このように内部表現で自動的に変換されていた演算子が内部表現で表現される本来の演算子に置換えられています. この dispform 函数は与えられた CRE 表現もこのような外部表現に変換します. ただし, 'dispform(⟨ 式 ⟩)' では式に含まれた函数の引数自体は内部形式のままです. そのため, 一切の例外を認めずに式全てを外部表現に変換したければ第 2 引数に "all" を指定することで式の変換を行います:

```
(%i40) expr1:f(sqrt(y/x));
                                          y
(%o40)                              f(sqrt(-))
                                          x
(%i41) expr2:dispform(expr1);
                                          y
(%o41)                              f(sqrt(-))
                                          x
(%i42) expr3:dispform(expr1,all);
                                          y
(%o42)                              f(sqrt(-))
                                          x
(%i43) :lisp $expr1
(($F SIMP)
 ((MEXPT SIMP) ((MTIMES SIMP) ((MEXPT SIMP) $X -1) $Y) ((RAT SIMP) 1 2)))
(%i43) :lisp $expr2
(($F SIMP)
 ((MEXPT SIMP) ((MTIMES SIMP) ((MEXPT SIMP) $X -1) $Y) ((RAT SIMP) 1 2)))
(%i43) :lisp $expr3
(($F SIMP) ((%SQRT) ((MQUOTIENT) $Y $X)))
```

この例では $f\left(\sqrt{\frac{y}{x}}\right)$ を内部表現, dispform 函数を作用させたもの, dispform 函数の第 2 引数に 'all' を与えて作用させたものの結果を示しています. 表示は全て同じものですが内部表現では 'all' を指定したもの以外は全て同じですが, 'all' を指定することで式全体が外部表現になっています.

大域変数 display_format_internal

大域変数	既定値	概要
display_format_internal	false	内部表現に沿った表示への切替を制御

大域変数 display_format_internal:　この大域変数の値を切替えることで内部表現に沿った表示ができます. 既定値の 'false' であれば表示の際に前述のような変換を行うために, 内部表現をそのま

ま表示しません. ここで大域変数 display_format_internal の値を 'true' に設定すると内部表現に沿った表示に切替わります:

```
(%i1) display_format_internal;
(%o1)                                  false
(%i2) [a-b,a/b,sqrt(x),x*4/3,exp(x),1/exp(x)];
                                a            4 x     x     - x
(%o2)                  [a - b, -, sqrt(x), ---, %e  , %e    ]
                                b             3
(%i3) display_format_internal:true$
(%i4) [a-b,a/b,sqrt(x),x*4/3,exp(x),1/exp(x)];
                                  - 1    1/2   4       x     (- 1) x
(%o4)            [a + (- 1) b, a b    , x    , - x, %e  , %e        ]
```

6.4.8 変数と変数項

Maxima で変数や変数項として利用可能な対象は Maxima の記号と文字列です. さらに, 変数や変数項には項順序 "$>_m$" が組込まれています (§5.2 参照). Maxima の変数には値が束縛されていない自由変数, 値が束縛されている変数, そして, 函数内部で用いられる変数や Maxima の内部変数で構成される束縛変数があります.

変数の処理

大域変数 values: Maxima 上で値が割当てられた変数は大域変数 values に登録されます.
ただし, 値が割当てられた変数以外の束縛変数 (函数内部の変数, および内部変数) や Maxima の大域変数大域変数 values には登録されません:

Maxima に含まれる変数一覧

大域変数名	既定値	概要
values	[]	値が割当てられた変数が登録されるリスト

```
(%i1) a:1$
(%i2) b::sin(a);
(%o2)                                  sin(1)
(%i3) x;
(%o3)                                     x
(%i4) values;
(%o4)                                  [a, b]
```

この例では大域変数 values には値が割当てられた変数 a, b が登録されていますが, そうでない変数 x は登録されていません.
大域変数 values に登録された変数は remvalues 函数を用いて削除出来ます:

変数値の削除を行う函数

remvalue(⟨変数$_1$⟩,⟨変数$_2$⟩,...)
remvalue (all)

remvalue 函数:　この函数は指定した変数を自由変数にし, 大域変数 values から削除します. この変数は添字されていても構いません. さらに remvalue(all) で全ての束縛変数が削除されます:

```
(%i10) b1:sin(y)$
(%i11) x:1$
(%i12) c1:b1*x$
(%i13) values;
(%o13)                          [b1, c1, x]
(%i14) remvalue(x);
(%o14)                             [x]
(%i15) c1;
(%o15)                           sin(y)
(%i16) values;
(%o16)                          [b1, c1]
(%i17) remvalue(all);
(%o17)                          [b1, c1]
(%i18) values;
(%o18)                             []
```

この例では変数 b1, x, c1 に値を割当て, 最初に変数 x を削除し, 最後に引数を 'all' とすることで全ての変数を削除しています. 変数 c1 の割当では変数 x を利用していますが, 入力直後に評価された値が変数 c1 に割当てられているために変数 x を削除しても問題はありません. なお, 属性を利用して remove 函数 で同様の処理が可能です.

式中の変数の処理

Maxima には式中の変数を取出したりする函数が幾つか用意されています:

式中の変数を扱う函数
args(⟨ 式 ⟩)
listofvars(⟨ 式 ⟩)

args 函数:　⟨ 式 ⟩ が函数項や演算子項であれば, その引数のリストを返し, 通常の式であれば項のリストを返します. すなわち, 内部表現式の引数をリストで返却する函数です.
args 函数は substpart("[",⟨ 式 ⟩,0) と同値の処理を実行する函数で, 多項式の CRE 表現から変数を取出す showratvars 函数とは違います:

```
(%i21) args(sin(x));
(%o21)                             [x]
(%i22) args(sin(x+y));
(%o22)                          [y + x]
(%i23) expand((sin(x)+y)^2);
                              2                    2
(%o23)                       y  + 2 sin(x) y + sin (x)
(%i24) args(%);
                                2                  2
(%o24)                       [y , 2 sin(x) y, sin (x)]
```

ここで args 函数と substpart 函数の両方は大域変数 inflag の設定に依存します.

listofvars 函数: 〈式〉の変数リストを生成します. ここで, 大域変数 listconstvars のが値が 'true' であれば, 〈式〉に Maxima の数学定数 '%e', '%pi', '%i' や定数属性を付与した変数で含まれているものがあれば, listofvars 函数が返すリストに, これらの定数が含められます. ただし, 既定値の 'false' のままの場合は Maxima の数学定数を除外したリストを返却します.

listofvars の動作を制御する大域変数

変数名	既定値	概要
listconstvars	false	Maxima の定数を与式の変数リストに加えるかどうかを制御
listdummyvars	true	疑似変数を与式の変数リストに加えるかどうかを制御

大域変数 listconstvars: 'true' のときに定数属性を付与した変数と Maxima の数学定数 '%e', '%pi', '%i' が式に含まれていれば, listofvars 函数はこれらの定数も変数として加えたリストを返します. 大域変数 listconstvars が 'false' の場合, 数学定数と定数属性を付与した変数が除外されたリストを listvars 函数が返します:

```
(%i6) listofvars(x^2*y+aa+%e);
(%o6)                                [x, y]
(%i7) listconstvars:true;
(%o7)                                 true
(%i8) listofvars(x^2*y+aa+%e);
(%o8)                           [%e, aa, x, y]
```

大域変数 listdummyvars: 'false' であれば, 式の疑似変数は listofvars 函数が出力するリストの中に含まれません. なお, 疑似変数は sum 函数や product 函数等の添字や極限変数や定積分変として利用される変数です.

6.4.9 部分式に分解する函数

Maxima には与えられた式を部分式に分解する函数があります. まず, isolate 函数と disolate 函数は指定した変数を含む式と含まない式に分解して表示します:

指定した変数を分離して式を表示する函数

isolate(〈式〉,〈変数〉)
disolate(〈式〉〈変数$_1$〉, ...,〈変数$_n$〉)

isolate 函数: 〈式〉から〈変数〉との積を持つ部分式と持たない部分式に分けて表示します. 〈変数〉を持たない部分式は中間ラベルで置換えられ, 式全体は中間ラベルと〈変数〉の項の和で表現されます. ここで, isolate 函数は式の展開を行なわずに与式を指定された変数を持つ項と持たない項に分けて表示するだけの函数です.

ここで, 大域変数 isolate_wrt_times の値が 'false' の場合, ⟨ 式 ⟩ を ⟨ 変数 ⟩ との積を持たない部分式と ⟨ 変数 ⟩ との積を持つ部分式に分解して表示します.
大域変数 isolate_wrt_times の値が 'true' の場合, isolate 函数はさらに ⟨ 式 ⟩ を分解し, 項も ⟨ 変数 ⟩ の羃とそれ以外の変数との積に分解して表示します:

```
(%i12) isolate_wrt_times:false$
(%i13) exp1:expand((1+a+x)^2);
                          2                    2
(%o13)                   x  + 2 a x + 2 x + a  + 2 a + 1
(%i14) isolate(exp1,x);

                                  2
(%t14)                           a  + 2 a + 1

                             2
(%o14)                      x  + 2 a x + 2 x + %t14
(%i15) isolate_wrt_times:true$
(%i16) isolate(exp1,x);

(%t16)                                 2 a

                             2
(%o16)                      x  + %t16 x + 2 x + %t14
(%i17) isolate((1+a+x)^2,x);

(%t17)                                a + 1

                                                2
(%o17)                             (x + %t17)
```

disolate 函数:　isolate 函数に似ていますが, こちらでは利用者が一つ以上の変数を同時に孤立させて表示することが出来ます. この isolate 函数に影響を与える大域変数として次の大域変数があります:

isolate 函数に影響を与える大域変数

変数名	既定値	概要
exptisolate	false	指数項を範囲に含めるかを決定
isolate_wrt_times	false	表示を制御

大域変数 exptisolate:　'true' の場合に 'isolate(⟨ 式 ⟩ ,⟨ 変数 ⟩)' の実行で ⟨ 変数 ⟩ に含まれる (%e のような) 原子の指数項に対しても調べます.

大域変数 isolate_wrt_times:　'false' の場合, isolate 函数は指定した変数を含まない項と含む項に分けて表示を行います.
'true' の場合はさらに式を分解し, 指定した変数を含む項も指定した積を除く項と指定した変数の項の積に分解して表示を行います:

```
(%i18) isolate_wrt_times;
(%o18)                                  false
(%i19) exp1:expand((a+b+x)^2);
                        2                    2            2
(%o19)                 x  + 2 b x + 2 a x + b  + 2 a b + a
(%i21) isolate(exp1,x);

                                    2            2
(%t21)                             b  + 2 a b + a

                              2
(%o21)                       x  + 2 b x + 2 a x + %t21
(%i22) isolate_wrt_times:true$
(%i23) isolate(exp1,x);

(%t23)                                  2 a


(%t24)                                  2 b

                             2
(%o24)                      x  + %t24 x + %t23 x + %t21
```

isolate 函数では分離して表示するだけでしたが, 指定した変数を含む部分と含まない部分に分解する函数 partition と純虚数を含む項と含まない項に分解する函数 rectform があります.

部分式に分解する函数

partition(⟨ 式 ⟩,⟨ 変数 ⟩)
rectform(⟨ 式 ⟩)

partition 函数: 与式 ⟨ 式 ⟩ を分解し, 二つの部分式を成分とするリストを返します. これらの部分式は ⟨ 式 ⟩ の第一層に属するもので, 第 1 成分が ⟨ 変数 ⟩ を含まない部分式, 第 2 成分が ⟨ 変数 ⟩ を含む部分式となります:

```
(%i89) part(x+1,0);
(%o89)                                  +
(%i90) partition(x+1,x);
(%o90)                                [1, x]
(%i91) part((x+1)*y,0);
(%o91)                                  *
(%i92) partition((x+1)*y,x);
(%o92)                              [y, x + 1]
(%i93) part([x+1,y],0);
(%o93)                                  [
(%i94) partition([x+1,y],x);
(%o94)                            [[y], [x + 1]]
```

rectform 函数: 与式を 'a+b*%i' の書式で返します. ⟨ 式 ⟩ が複素数の場合, 変数 a と b は実数になりますが, そうでない場合は純虚数 '%i' を持たない部分式と純虚数 '%i' で括られた部分式に分解

しますが, 内部の項順序に従って出力されるために 'a+b*%i' の書式にはなりません:

```
(%i12) rectform((x+%i)^3);
                                3             2
(%o12)                         x  + %i (3 x  - 1) - 3 x
(%i13) rectform((10+%i)^3);
(%o13)                              299 %i + 970
```

6.4.10　部分式を扱う函数

Maxima の式には函数や演算子を節とする木構造表現があります. この表現は式の内部表現でより明確に現われます. ただし, 式の内部表現は前置式表現を取るために判り難いものとなっています. Maxima の式を操作する函数には Maxima 側から見た式の木構造を基に式を操作する函数と, 式の内部表現を直接操作する函数の二種類があります.
同様に, 部分式を取出す函数にも式の内部表現を直接利用する inpart 函数, 一般的な木構造表現を用いる part 函数の二種類があります:

部分式を取出す函数

inpart(⟨ 式 ⟩ ,⟨ 整数$_1$⟩ , . . .,⟨ 整数$_k$⟩)
inpart(⟨ 式 ⟩ , [⟨ 整数$_1$⟩ ,. . .,⟨ 整数$_k$⟩])
part(⟨ 式 ⟩ ,⟨ 整数$_1$⟩ , . . .,⟨ 整数$_k$⟩)
part(⟨ 式 ⟩ , [⟨ 整数$_1$⟩ ,. . .,⟨ 整数$_k$⟩])
op(⟨ 式 ⟩)
pickapart(⟨ 式 ⟩ ,⟨ 整数 ⟩)

inpart 函数:　与式 ⟨ 式 ⟩ の内部表現に直接作用する函数で, ⟨ 整数$_1$⟩ ,. . .,⟨ 整数$_k$⟩ で指定された ⟨ 式 ⟩ の部分式を取出す函数です. この inpart 函数は式の内部表現に直接作用するので, その分, 処理が速くなります.

part 函数:　この函数は inpart 函数に似ていますが, ⟨ 式 ⟩ の木構造表現に対応する部分式を取出します.
部分式の取出し方は最初に ⟨ 式 ⟩ から ⟨ 整数$_1$⟩ で指定される部分式を取出します. 次に取出した部分式から ⟨ 整数$_2$⟩ で指定される成分を取出します. 以降, 同様に ⟨ 整数$_{k-1}$⟩ で指定された部分式から ⟨ 整数$_k$⟩ で指定される成分を取出し, この部分式を結果として返します:

```
(%i15) part((x+1)^3+2,1);
                                      3
(%o15)                          (x + 1)
(%i16) part((x+1)^3+2,1,1);
(%o16)                            x + 1
(%i17) part((x+1)^3+2,1,1,1);
(%o17)                              x
```

なお, [⟨ 整数$_1$⟩, . . . , ⟨ 整数$_n$⟩] のようにリストで指定することもできますが, この場合は木構造の第一層となる ⟨ 整数$_1$⟩ で指定された部分式, 以降, ⟨ 整数$_n$⟩ で指定される部分式が取出され, これらの部分式に part(⟨ 式 ⟩,0) で得られる主演算子を作用させた式が返されます:

```
(%i72) expr:x+y+sin(x^2+2*x+1)+cos(z/w);
                                z              2
(%o72)                     cos(-) + y + sin(x  + 2 x + 1) + x
                                w
(%i73) inpart(expr,[2,4]);
                                  z          2
(%o73)                         cos(-) + sin(x  + 2 x + 1)
                                  w
(%i74) part(expr,[1,4]);
                                          z
(%o74)                                cos(-) + x
                                          w
(%i75) expr2:x*y*z*sin(x^2+1);
                                          2
(%o75)                             x sin(x  + 1) y z
(%i76) inpart(expr2,[1,4]);
(%o76)                                    x z
(%i77) part(expr2,[1,2]);
                                            2
(%o77)                                x sin(x  + 1)
(%i78) inpart(expr,0);
(%o78)                                     +
(%i79) inpart(expr2,0);
(%o79)                                     *
```

この例で示すようにリストで指定した場合には部分式を抜き出して主演算子を作用させたものが返されています. そのため, 和や積, 一つだけの被演算子を取る演算子 (unary) としての演算子 “-”, 差と商を扱う場合, 部分式の順序に注意が必要になります.
ここで inpart 函数と part 函数に影響を与える大域変数を纏めておきましょう:

part 函数に関連する大域変数

変数名	既定値	概要
piece		inpart/part 函数で取出した部分式を一時保存
partswitch	false	inpart/part 函数のエラーメッセージを制御

大域変数 piece: この大域変数に inpart 函数や part 函数を用いて取出した最新の部分式が保存されます.

大域変数 partswitch: この大域変数の値が ‘true’ であれば式に指定した成分が存在しない場合に inpart 函数と part 函数は ‘end’ を返します. ‘false’ の場合はエラーメッセージを返します.

op 函数: この函数は part 函数の機能省略版とも言えます. 実際, ‘op(⟨ 式 ⟩)’ は ’part(⟨ 式 ⟩,0)’ と同じです.

pickapart 函数: 〈整数〉で指定された式の階層に含まれる全ての部分式を%t ラベルに割当て, ラベルを用いた式に〈式〉を変換します. 階層指定は part 函数と同様で, 表示された形式に対して指定を行います. pickapart 函数は大きな式を扱う際に part 函数を使わずに部分式に変数を自動的に割当てることにも使えます:

```
(%i49) exp:(x+1)^3;
                                        3
(%o49)                                (x + 1)
(%i50) pickapart(exp,1);
                                         3
(%o50)                                 %t48
(%i51) exp2:expand((x+1)^3);
                                  3      2
(%o51)                           x  + 3 x  + 3 x + 1
(%i52) pickapart(exp2,1);

                                        3
(%t52)                                 x


                                       2
(%t53)                               3 x


(%t54)                                 3 x

(%o54)                    %t54 + %t53 + %t52 + 1
```

6.4.11　総和と積

Maxima では総和 (∑) と総積 (∏) が扱えます:

∑ と ∏

product(〈式〉,〈添字変数〉,〈下限〉,〈上限〉)
sum(〈式〉,〈添字変数〉,〈下限〉,〈上限〉)
lsum(〈式〉,〈添字変数〉,〈リスト〉)

product 函数: 〈添字変数〉の〈下限〉から〈上限〉までの〈式〉の値の積を与えます. 〈上限〉が〈下限〉より小になると空の積となり, この場合は product 函数はエラー出力ではなく '1' を返します. 評価は sum 函数と似ています.
この product 函数を制御する大域変数としては大域変数 simoroduct があります:

product 函数を制御する大域変数

変数名	既定値	概要
simpproduct	false	product 函数の簡易化を制御

大域変数 simpproduct 既定値は 'false' ですが, この値を 'true' に変更すると product 函数項を自動的に簡易化しようとします:

```
(%i23) product(k,k,1,n);
                                        n
                                      /===\
                                       ! !
(%o23)                                 ! !   k
                                       ! !
                                      k = 1
(%i24) simpproduct:true$
(%i25) product(k,k,1,n);
(%o25)                                 n!
(%i26) simpproduct:false;
(%o26)                                false
(%i27) product(k,k,1,n),simpproduct;
(%o27)                                 n!
```

この例では大域変数 simpproduct の値を 'true' にすることで, product 函数の自動的な簡易化が行われていることに注意して下さい. なお, この大域変数 simpproduct は evflag 属性を持つために ev 函数による評価で指定できます.

sum 函数: 〈添字変数〉を〈下限〉から〈上限〉までの値の〈式〉の和を取ります. 〈上限〉と〈下限〉が整数で異なっていれば, 和の各々の項は評価されて互いに加えられます.
sum 函数に影響を与える大域変数を次に纏めておきます:

sum 函数を制御する大域変数

変数名	既定値	概要
simpsum	false	sum 函数の簡易化を制御
sumexpand	false	総和の積の処理を制御
cauchysum	fale	Cauchy 積の適用を制御
genindex	i	sum や product で利用される疑似変数名を設定
gensumnum	false	疑似変数の番号

大域変数 simpsum: true であれば sum 函数を自動的に簡易化します:

```
(%i33)  simpsum;
(%o33)                                 false
(%i34)  sum(x^n,n,0,m);
                                       m
                                      ====
                                      \        n
(%o34)                                 >      x
                                      /
                                      ====
                                      n = 0
(%i35)  simpsum:true$
(%i36)  sum(x^n,n,0,m);
                                      m + 1
```

```
                                        x       - 1
(%o36)                                  ----------
                                          x - 1
(%i37) sum(x^n,n,0,inf);
Is  abs(x) - 1  positive, negative, or zero?

pos;
(%o37)                                    inf
(%i38) sum(x^n,n,0,inf);
Is  abs(x) - 1  positive, negative, or zero?

neg;
                                          1
(%o38)                                  -----
                                        1 - x

(%i39) simpsum:false$
(%i40) sum(k,k,1,n),simpsum;
                                         2
                                        n  + n
(%o40)                                  ------
                                          2
```

大域変数 simpsum の値が ‘false’ であれば式 ‘sum(x^n,n,0,m)’ の簡易化は実行されませんが, ‘true’ の場合は自動的に簡易化されます. 式 ‘sum(x^n,n,0,inf)’ に関しては, 変数 x の絶対値から ‘1’ を引いたものが正, 負, あるいは零であるかを利用者が指示することで簡易化が行えます. この大域変数 simpsum は evflag 属性を持つために ev 函数による評価でも利用できます.

大域変数 sumexpand:　‘true’ であれば sum 函数の積を纏めて sum 函数の入れ子にします:

```
(%i1) sum(f(x),x,0,m)*sum(g(x),x,0,n);
                                  m          n
                                 ====       ====
                                 \          \
(%o1)                            ( >    f(x)) >    g(x)
                                 /          /
                                 ====       ====
                                 x = 0      x = 0
(%i2) sumexpand:true$
(%i3) sum(f(x),x,0,m)*sum(g(x),x,0,n);
                                m      n
                               ====   ====
                               \      \
(%o3)                           >      >     f(i1) g(i2)
                               /      /
                               ====   ====
                               i1 = 0 i2 = 0
```

なお, この処理は定数 inf や定数 minf を含む総和に対して利用することは非常に危険です.

大域変数 cauchysum:　大域変数 sumexpand と対で用います. 大域変数 sumexpand と大域変数 cauchysum の双方が true であれば, 総和の掛算を纏める際に通常の積ではなく Cauchy 積が適用

されます:

```
(%i1) sumexpand:true$
(%i2) cauchysum:true$
(%i3) sum(f(x),x,0,m)*sum(g(x),x,0,n);
                               m       n
                              ====    ====
                              \       \
(%o3)                          >       >      f(i1) g(i2)
                              /       /
                              ====    ====
                              i1 = 0  i2 = 0
(%i4) sum(f(x),x,0,inf)*sum(g(x),x,0,n);
                              inf      n
                              ====    ====
                              \       \
(%o4)                          >       >      f(i3) g(i4)
                              /       /
                              ====    ====
                              i3 = 0  i4 = 0
(%i5) sum(f(x),x,0,inf)*sum(g(x),x,0,inf);
                            inf      i5
                            ====    ====
                            \       \
(%o5)                        >       >      g(i5 - i6) f(i6)
                            /       /
                            ====    ====
                            i5 = 0  i6 = 0
```

大域変数 genindex と大域変数 gensumnum: これらの大域変数は sum 函数内部の疑似変数を生成するために用いられます. 大域変数 genindex には疑似変数のアルファベット, 大域変数 gensumnum には疑似変数の番号がそれぞれ設定されています. ここで, 大域変数 gensumnum は sum 函数内部で疑似変数を生成する度に一つづつ増加します:

```
(%i1) sumexpand:true$
(%i2) gensumnum;
(%o2)                                  0
(%i3) sum(f(x),x,0,m)*sum(g(x),x,0,n);
                               m       n
                              ====    ====
                              \       \
(%o3)                          >       >      f(i1) g(i2)
                              /       /
                              ====    ====
                              i1 = 0  i2 = 0
(%i4) gensumnum;
(%o4)                                  2
```

この例では, 大域変数 sumexpand の値を 'true' にしたために総和の積が纏められ, その結果, 二つの疑似変数 i1 と i2 が新たに生成されています. このときに大域変数 gensumnum は最初が 0 で, それから二つの疑似変数を生成したために '2' になっています.

最後の lsum 函数は sum 函数に似ていますが, 添字変数の定義域をリストで与えます. ただし, この定義域は必ずリストで与える必要はなく, リストで与えられない場合に名詞型で式を返します:

```
(%i2) lsum(i,i,[1,2,3,4,5,6]);
(%o2)                                  21
(%i3) lsum((i*x+1)^2,i,[1,3,5,7,9]);
                    2             2             2             2             2
(%o3)      (9 x + 1)  + (7 x + 1)  + (5 x + 1)  + (3 x + 1)  + (x + 1)
(%i4) lsum((i*x+1)^2,i,mikeneko);
                                ====
                                \                2
(%o4)                            >      (i x + 1)
                                /
                                ====
                                i in mikeneko
```

この例で示すように mikeneko は単なる記号で Maxima のリストと異なるために sum 函数の名詞型として式が返されています.

sum 函数と prod 函数双方に影響する函数

bashindices(⟨ 式 ⟩)
niceindices(⟨ 式 ⟩)

bashindices 函数:　sum 函数と prod 函数の添字を自動的に変更する函数です. このとき, 大域変数 genindex の値は j(内部的には$j) が割当てられるために与式の sum 函数や prod 函数の添字が ‘j1, j2, ...’ で置換えられます:

```
(%i23) neko:sum(sum(sin(k*x)^l,l,1,inf),k,1,inf);
                                 inf   inf
                                 ====  ====
                                 \     \          l
(%o23)                            >     >     sin (k x)
                                 /     /
                                 ====  ====
                                 k = 1 l = 1
(%i24) bashindices(neko);
                                inf    inf
                                ====   ====
                                \      \          l
(%o24)                           >      >     sin (j2 x)
                                /      /
                                ====   ====
                                j2 = 1 l = 1
```

niceindices 函数:　後述の大域変数 niceindicespref を利用する函数で, sum 函数や prod 函数による総和や積の添字をこの niceindicespref に登録された添字で置換えます:

```
(%i28) neko;
                                  inf   inf
```

```
                                 ====  ====
                                 \     \          l
(%o28)                            >     >     sin (k x)
                                 /     /
                                 ====  ====
                                 k = 1 l = 1
(%i29) niceindices(neko);
                                 inf   inf
                                 ====  ====
                                 \     \          i
(%o29)                            >     >     sin (j x)
                                 /     /
                                 ====  ====
                                 j = 1 i = 1
```

大域変数 niceindicespref: sum 函数と prod 函数の双方に影響を与える大域変数として大域変数 niceindicespref があります:

sum 函数と prod 函数双方に影響する大域変数

大域変数	既定値	概要
niceindicespref	[i,j,k,l,m,n]	sum 函数と prod 函数の双方で用いる添字を指定

大域変数 niceindiciespref には niceindices 函数で置換えられる添字リストが割当てられています. このリストに蓄えられた添字を使い切るとリストの先頭の添字に '0' から開始する番号をうしろに付加した添字を用います:

```
(%i48) neko:prod(sum(sum(m*sin(k*x)^l,l,1,inf),k,1,inf),m,1,inf);
                              inf     inf   inf
                             /===\   ====  ====
                             ! !     \     \          l
(%o48)                       ! !  m  >     >     sin (k x)
                             ! !     /     /
                            m = 1    ====  ====
                                     k = 1 l = 1
(%i49) niceindicespref:[a,b];
(%o49)                                [a, b]
(%i50) niceindices(neko);
                             inf       inf   inf
                            /===\      ====  ====
                            ! !        \     \          a
(%o50)                      ! !   a0   >     >     sin (b x)
                            ! !        /     /
                            a0 = 1     ====  ====
                                       b = 1 a = 1
```

6.4.12　式の様々な操作を行う函数

式の符号を判定する函数

sign 函数

sign (⟨ 式 ⟩)

sign 函数:　sign 函数は Maxima の文脈 (§5.6 参照) を用いて与えられた式の正負を判定する函数です. この sign 函数が返却する値は {pos, neg, zero, pz, nz, pnz} になりますが, これらの値の意味を次の表 6.1 に纏めておきます:

表 6.1: sign 函数の返す値

pos	正の場合
neg	負の場合
zero	零の場合
pz	正か零の場合
nz	負か零の場合
pn	正か負の場合
pnz	判定不能

この sign 函数の実例を以下に示しておきましょう.

```
(%i42) assume(neko>0);
(%o42)                              [neko > 0]
(%i43) sign(neko);
(%o43)                                 pos
```

この例では変数 neko が 0 より大であると assume 函数を用いて仮定したために sign 函数は変数 neko が pos であると返しています.

式の複製を行う函数

copy 函数の構文

copy(⟨ 式 ⟩)

copy 函数:　Maxima の式の複製を行う函数で, 引数は Maxima の任意の式になります.
なお, 函数 copy の実体は内部函数の copy-tree 函数です. この函数は他に copymatrix 函数, copylist 函数でも用いられていますが, それぞれ, matrixp 函数や listp 函数による引数の判定が行われる仕組になっています. ただし, これらの函数は copy-tree 函数を用いて複製を生成する仕組のために実質的に copy 函数の適用範囲を狭めたものでしかありません.

```
(%i33) pet:{みけ,ぽち,たま}$
(%i34) my_pet:copy(pet)$
(%i35) my_pet;
(%o35)                          {たま, ぽち, みけ}
(%i36) eq1:x^2+y+1$
(%i37) eqx:copy(eq1);
                                     2
(%o37)                          y + x  + 1
```

二項演算式の検出を行う函数

assoc 函数の構文

assoc(⟨ キーワード ⟩,⟨ リスト ⟩)
assoc(⟨ キーワード ⟩,⟨ リスト ⟩,⟨ 文字列 ⟩)

assoc 函数: この函数は 'x * y' のような一つの二項演算子と二つの被演算子の項で構成されたリストに対して ⟨ キーワード ⟩ に適合する項があれば, その項の被演算子を返す函数です. ここで, 第 3 引数に文字列を指定すれば, 適合に失敗すると 'false' ではなく指定した文字列を返します:

```
(%i4) assoc(x,[a*b,c^x,x*y]);
(%o4)                                y
(%i5) assoc(x,[a*b,c^x,a*x]);
(%o5)                              false
(%i6) assoc(x,[a*b,c^x,a*x],"残念でした");
(%o6)                              残念でした
(%i7) assoc(x,[a*b,c^x,x*a],"残念でした");
(%o7)                              残念でした
```

この函数は与式の内部構造に対して検出を行うために例の 'x * y' と 'a * x' で結果が異なるのに対し, 'a * x' と 'x * a' の結果が一致する理由となっています.

6.4.13 TeX や FORTRAN の書式に式の変換を行う函数

Maxima の出力式は TEX や FORTRAN の書式に変換出来ます. このために tex 函数や fortran 函数を用います. ここで, Maxima の函数や与件の構造は TEX や FORTRAN の書式とば異なったものです. まず, TEX に関しては通常の C や FORTRAN とは全く異なった書式になります. そこで, Maxima は函数や変数に予め設定した属性を利用して TEX への式の内部表現の変換を実行しています. そのために利用者定義の函数や演算子に対しては texput 函数を用いて TEX の書式との対照関係を入れてやる必要があります.
FORTRAN への式の変換では, Maxima と FORTRAN では共通する式や表現も多いために比較的簡単な置換で済みます. ただし, fortran 函数にとって未知の函数や定義式については利用者が処理する必要があります.

TEX の書式に変換

TEX の変換に関連する函数

tex(⟨ 式 ⟩)
tex(⟨ 式 ⟩,⟨ ファイル名 ⟩)
tex(⟨ ラベル行 ⟩,⟨ ファイル名 ⟩)
texinit(⟨ ファイル ⟩)
texend(⟨ ファイル ⟩)
texput(⟨ 記号 ⟩,⟨ 文字列 ⟩)
texput(⟨ 記号 ⟩,⟨ 文字列 ⟩,⟨ 演算子型 ⟩)
texput(⟨ 記号 ⟩,[⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩],matchfix)
texput(⟨ 記号 ⟩,[⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 文字列$_3$⟩],matchfix)

tex 函数:　与えられた ⟨ 式 ⟩ や ⟨ ラベル行 ⟩ を TEX の書式に変換します. ⟨ ファイル名 ⟩ を指定すると, 出力結果は指定ファイルに保存されます. なお, 指定ファイルが既存すれば, その結果はファイル末尾に追加されます. なお, ラベル行を変換する場合, 式のラベル番号も生成されます.
tex 函数では A から ω 迄のギリシャ文字読みの変数は全て TEX のギリシャ文字表記で置換えられます. そして, Maxima の函数や変数に関しても tex 函数によって置換が行われます:

```
(%i15) tex([Alpha,beta,Gamma,Omega,omega]);
$$\left[ {\rm A} , \beta , \Gamma , \Omega , \omega \right] $$
(%o15)                                 false
(%i16) tex([exp(2*%pi*%i*t)*sin(2*%pi/4*t),cos(u)^2+sin(u)^3]);
$$\left[ e^{2\,i\,\pi\,t}\,\sin \left({{\pi\,t}\over{2}}\right) ,
 \sin ^3u+\cos ^2u \right] $$
(%o16)                                 false
```

この置換は Maxima の内部変数 defprop 函数を用いて texword 属性等の属性として TEX の表記や置換函数を予め格納させており, tex 函数を用いて, その texword 属性で置換えたり, 置換函数を作用させることで変換を行っています.
なお, Maxima は現在の Maxima に存在しない幾つかの函数 (cubrt ,%j 等を TEX の書式に変換する機能も持っています.

texend 函数:　指定したファイルの末尾に \end を書込むだけの函数です.

texput 函数:　指定した記号と TEX の書式を結び付ける函数です. 利用者が定義した函数や変数等を TEX に自動的に変換させるために使えます.
まず, 第 1 引数が tex 函数で TEX の書式に変換すべき記号となります. 第 2 引数はその記号に対応する TEX の書式の雛形となります.
引数が二つだけの場合, *alpha* → α のような対応付けとなります. 変換すべき記号が演算子である場合, その演算子の特性を反映して変換を行う必要があります. この場合, 第 3 引数に演算子の型を指定します.

この例を以下に示しておきましょう:

```
(%i3) infix("|-")$ infix("->")$ nary("|*|")$
(%i6) texput("|-"," \\vdash ",infix)$
(%i7) texput("->"," \\rightarrow ",infix)$
(%i8) texput("|*|"," \\wedge ",nary)$
(%i9) neko:((A->B)|*|(B->C))|-(A->C);
(%o9)                    A -> B |*| (B -> C) |- (A -> C)
(%i10) tex(neko);
$$A \rightarrow B \wedge \left(B \rightarrow C\right) \vdash \left(A
  \rightarrow C\right)$$
(%o10)                                false
```

ここでは $A \to B \wedge B \to C \vdash A \to C$ を tex 函数に生成させようとしたものです.
この tex 函数による結果を次に示しておきましょう.

$$A \to B \wedge (B \to C) \vdash (A \to C)$$

括弧の処理に問題がありますが, まあまあでしょうか.
texput 函数の最後の二つの型は変換すべき演算子が matchfix 型の場合です. この場合, 演算子は式を挟んで左右に存在するために左右の記号の変換を指定しなければなりません.

```
(%i20) matchfix("(:-","(:-(");
(%o20)                                (:-)
(%i21) (:-) x+1 (:-(;
(%o21)                             (:-)x + 1(:-(
(%i22) texput("(:-)",["\\int \\int","\\int \\int"],matchfix);
(%o22)                          [\int \int, \int \int]
(%i23) tex((:-) x+1 (:-();
$$\int \intx+1\int \int$$
(%o23)                                false
(%i24) texput("(:-)",["\\int \\int ","\\int \\int"],matchfix);
(%o24)                          [\int \int , \int \int]
(%i25) tex((:-) x+1 (:-();
$$\int \int x+1\int \int$$
(%o25)                                false
```

ここで, 最初の tex 函数による変換では記号に対応する TEX の書式に空行がないために`\intx+1` の様に,`\int` と `x+1` が結合していることに注意して下さい. なお, Maxima-5.14.0 以降では Maxima の文字列が LISP の文字列型となったためか, 演算子にアルファベット以外の文字を用いていると記号ではないとエラーになります.

6.4.14 FORTRAN の書式に変換

fortran 函数と fortmx 函数

fortran(⟨ 式 ⟩)
fortmx(⟨ 原子 ⟩,⟨ 行列 ⟩)
fortmx(⟨ 原子 ⟩,⟨ 行列 ⟩,⟨ ストリーム ⟩)

fortran 函数:　与式を FORTRAN の構文に変換します. 出力行が長くなり過ぎると継続文字を用います. この文字は '1' から '9' までが順番に振られて足りなくなると, ':', ';', '<', '=+' '>', '?', '@+' 順番で振られ, それから再度 '1' から '9' を繰返します.
この fortran 函数は Maxima の式を FORTRAN の書式に合せます. たとえば, 式中の演算子 "^" を演算子 "**" で置換し, 複素数 'a + b*%i' を '(a, b)' で置換えます.
与式は方程式であって構いません. この場合, 方程式の右辺を左辺に割当てる形で出力します. ここで, 右辺が行列名であれば, fortran 函数は行列の各成分に割当を行うように与式を変換します.
もし, 与式に fortran 函数で解釈出来ないものがあった場合は grind 函数を用いて式の内部表現の変換を行い, 表示されている式と同じ式を出力します:

```
(%i56) expr:(a+b+1)^5;
                                        5
(%o56)                          (b + a + 1)
(%i57) fortran(expand(expr));
      b**5+5*a*b**4+5*b**4+10*a**2*b**3+20*a*b**3+10*b**3+10*a**3*b**2+3
     1    0*a**2*b**2+30*a*b**2+10*b**2+5*a**4*b+20*a**3*b+30*a**2*b+20*a
     2    *b+5*b+a**5+5*a**4+10*a**3+10*a**2+5*a+1
(%o57)                                  done
(%i58) prefix("|-");
(%o58)                                   |-
(%i59) fortran( |- (x^2+y^2-a^2=0) );
      |- (y**2+x**2-a**2 = 0)
(%o59)                                  done
```

この例では最初に多項式の変換を実行しています. その次の例では前置演算子 "—-" を定義し, その式を fortran 函数に与えています. 勿論, fortran 函数にとって演算子 "—-" は未知の代物のために単純に出力を通常の式表示と同じものに変換しています.
fortran 函数の引数に行列が含まれている場合, fortran 函数は fortmx 函数を用いて行列の処理を実行します. この場合, 第 1 引数に fortran プログラムで用いる配列名, 第 2 引数に行列を指定します. なお, 第 3 引数にストリームを指定すると出力が指定したストリームに対して行われます.

```
(%i25) A1:matrix([1,2,3],[4,3,2]);
                                    [ 1  2  3 ]
(%o25)                              [         ]
                                    [ 4  3  2 ]
(%i26) neko:openw("testfile")$
(%i27) fortmx( testarray,A1);
      testarray(1,1) = 1
      testarray(1,2) = 2
      testarray(1,3) = 3
      testarray(2,1) = 4
      testarray(2,2) = 3
      testarray(2,3) = 2
(%o27)                                  done
(%i28) fortmx( testarray,A1,neko);
(%o28)                                  done
```

この例では行列 A1 を fortmx 函数を用いて fortran の書式に変換しています. 最後にストリームを指定し, その結果はストリームの出力先のファイル testfile に書込まれます.

fortran 函数に関連する大域変数

fortindent	0	左側の空白を制御
fortspaces	false	80 カラム出力の制御

大域変数 fortindent は左側の空白を制御します. 既定値の '0' で通常の空白文字 (6-空白文字) となります.
大域変数 fortspaces が true であれば fortran 函数は 80 列で出力を行います.

6.5　リスト

6.5.1　Maxima のリスト

Maxima にはリストと呼ばれる与件型があります. これは配列に似たデータ構造ですが, より柔軟で視覚的にも把握し易い構造を持っており, 多くの数式処理で採用されています.
Maxima のリストは '[1, 2, 7, x+y]' のように対象をコンマ "," で区切った列を大括弧 "[]" で括った対象です. ここで, リストは集合とは違い, 列の順序に意味があるので, 列の順序が違えば異ったリストになります. たとえば, '[1,2,3]', '[1,3,2]' と '[3,2,1]' は全て対象 '1', '2', '3' で構成されたリストですが, 成分の並び順が異なるために全て異なったリストになります. なお, 集合の場合は成分の並び順は無関係で, '{2,3,1}', '{1,3,2}' は '{1,2,3}' と同値な集合になります. Maxima の集合はリストと重なる部分も多いために必要に応じて §6.6 も参照して下さい. なお, Maxima のリストは LISP のリストのように成分の区切に空行を用いないことに注意して下さい.
Maxima のリストは '[1,2,[3,4],[4,[5]]]' のようにリストを入れ子にした複合リストが扱えます. また, MATLAB のようにリストの成分が全て同じ型である必要もありません.
ここで Maxima は LISP で記述されたシステムのため, Maxima の式やプログラムも内部では LISP の S 式と呼ばれるリストで表現されています. さらに Maxima の演算子が前置式でなくても内部では前置表現になっています. そのために Maxima 内部表現では演算子の部分が 0, それ以降の成分に $1, 2, \ldots$ と番号が振られた階層構造を持っています.
このことから一見してリスト処理専用の函数であっても, Maxima の殆どの式で利用可能なことが多くあります. この Maxima の内部表現の詳細は §6.4 を参照して下さい.

6.5.2　リストの生成を行う函数

出力が Maxima のリストとなる函数は沢山あります. ここでは雛形となる式を用いてリストを生成する函数を幾つか紹介しておきましょう:

リストを生成する函数
create_list(⟨ 式 ⟩,⟨ 変数$_1$⟩,⟨ リスト$_1$⟩,...,⟨ 変数$_n$⟩,⟨ リスト$_n$⟩)
makelist(⟨ 式 ⟩,⟨ 変数 ⟩,⟨ 正整数$_1$⟩,⟨ 正整数$_2$⟩)
makelist(⟨ 式 ⟩,⟨ 変数 ⟩,⟨ リスト ⟩)

create_list 函数:　与えられた ⟨ 式 ⟩ に対して式中の変数にリストで指定した値を代入した結果で構成された平リストを返す函数です:

```
(%i30) create_list(x^2,x,[1,3,x1,sin(x)]);
                                        2      2
(%o30)                          [1, 9, x1 , sin (x)]
```

create_list 函数は後述の makelist 函数と違い複数の変数を扱うことができます. この場合, create_list 函数の引数を左端から順に作用させることになります.
たとえば, '`create_list(f(x`$_1$`,...,x`$_n$`),x`$_1$`,L`$_1$`,...,x`$_n$`,L`$_n$`)`' は

''flatten(create_list(...(create_list(f(x_1,...,x_n),x_n,L_n),...,x_1,L_1))' と同じ値となります.
ここで flatten 函数はリストを平リストにする函数です.
このことを実際に確認しておきましょう:

```
(%i28) create_list(i^j+sin(k),i,[0,1,2,3,4],j,[2,3],k,[x1,x2,x3]);
(%o28) [sin(x1), sin(x2), sin(x3), sin(x1), sin(x2), sin(x3), sin(x1) + 1,
sin(x2) + 1, sin(x3) + 1, sin(x1) + 1, sin(x2) + 1, sin(x3) + 1, sin(x1) + 4,
sin(x2) + 4, sin(x3) + 4, sin(x1) + 8, sin(x2) + 8, sin(x3) + 8, sin(x1) + 9,
sin(x2) + 9, sin(x3) + 9, sin(x1) + 27, sin(x2) + 27, sin(x3) + 27,
sin(x1) + 16, sin(x2) + 16, sin(x3) + 16, sin(x1) + 64, sin(x2) + 64,
sin(x3) + 64]
(%i29) flatten(create_list(create_list(create_list(i^j+sin(k),k,[x1,x2,x3]),
       j,[2,3]),i,[0,1,2,3,4]));
(%o29) [sin(x1), sin(x2), sin(x3), sin(x1), sin(x2), sin(x3), sin(x1) + 1,
sin(x2) + 1, sin(x3) + 1, sin(x1) + 1, sin(x2) + 1, sin(x3) + 1, sin(x1) + 4,
sin(x2) + 4, sin(x3) + 4, sin(x1) + 8, sin(x2) + 8, sin(x3) + 8, sin(x1) + 9,
sin(x2) + 9, sin(x3) + 9, sin(x1) + 27, sin(x2) + 27, sin(x3) + 27,
sin(x1) + 16, sin(x2) + 16, sin(x3) + 16, sin(x1) + 64, sin(x2) + 64,
sin(x3) + 64]
```

makelist 函数: 〈変数〉で〈式〉の変数を指定し, その変数に対して値を代入する事でリストを生成します. まず, 引数が 4 個の場合は第 3 引数と第 4 引数は正整数, 第 2 引数で指定した変数の値域は, この第 3 引数と第 4 引数で指定された閉区間に含まれる整数になります. ここで, 第 3 引数は第 4 引数以下でなければなりません.
makelist 函数の引数が 3 個の場合, 変数の値域は第 3 引数のリストになります:

```
(%i5) makelist(x^2+i*x+1=i^2,i,1,4);
         2              2              2              2
(%o5) [x  + x + 1 = 1, x  + 2 x + 1 = 4, x  + 3 x + 1 = 9, x  + 4 x + 1 = 16]
(%i6) makelist(x^2+i*x+1=i^2,i,[1,2,4]);
               2              2              2
(%o6)        [x  + x + 1 = 1, x  + 2 x + 1 = 4, x  + 4 x + 1 = 16]
```

6.5.3 リスト処理に関連する大域変数

リストに関連する大域変数

変数名	初期値	取り得る値	概要
listarith	true	[true,false]	算術演算子によるリスト評価を制御
inflag	false	[true,false]	内部表現への作用を制御

大域変数 listarith: 'true' であれば算術演算子によるリスト評価を実行します.

大域変数 inflag:　'true' であれば, 成分を取り出す函数は与えられた式の内部表現に対して処理を行います. 簡易化は式の並び換えを行うことに注意が必要になります. そのために 'first(x+y)' は大域変数 inflag が 'true' であれば x となりますが, 大域変数 inflag が 'false' ならば y となります.
ここで, この大域変数 inflag に影響を受ける函数を纏めておきましょう:

inflag の影響を受ける函数

函数	概要
part	式の成分を取出す函数
substpart	式の成分の代入を行う函数
first	リストの先頭を取出す函数
rest	リストの残りを取出す函数
last	リストの末尾を取出す函数
length	リストの長さを返す函数
for-in 構文	リストを用いるループ文
map	リストに函数を作用させる函数
fullmap	リストに函数を作用させる函数
maplist	リストに函数を作用させる函数
reveal	式の置換えを行う函数
pickapart	成分を取出す函数

6.5.4　リスト処理に関連する主な函数

リストに関連する真理函数

リストに関連する真理函数

函数	true を返す条件
atom(〈 変数 〉)	原子の場合
symbolp(〈 変数 〉)	記号の場合
listp(〈 変数 〉)	リストの場合
member(〈 式$_1$〉,〈 式$_2$〉)	〈 式$_1$〉が〈 式$_2$〉に含まれている場合

atom 函数:　引数が原子であれば 'true', それ以外は 'false' を返します.

symbolp 函数:　引数が記号であれば 'true', それ以外は 'false' を返します.

listp 函数:　引数がリストであれば 'true', そうでなければ 'false' を返します.
なお, ここでの判定では内部表現は無関係です. そのために 'sin(x)' や 'x+y' のような式では変数に値が束縛されていなければ原子でもリストにもなりません.

member 函数: 二つの引数を取り, ⟨ 式$_1$⟩ が ⟨ 式$_2$⟩ に含まれていれば 'true', それ以外は 'false' を返します:

```
(%i34) member(sin(x),cos(x)+sin(x)+x^2);
(%o34)                                true
(%i35) member(sin(x),[cos(x)+sin(x)+x^2]);
(%o35)                                false
(%i36) member(sin(x),[cos(x),sin(x),x^2]);
(%o36)                                true
(%i37) member(sin(x),[cos(x),sin(x)+x^2]);
(%o37)                                false
(%i38) member(sin(x),f(cos(x),sin(x),x^2));
(%o38)                                true
(%i39) member(sin(x),f(cos(x),sin(x)+x^2));
(%o39)                                false
```

リストの基本処理を行う函数

リストの基本処理を行う函数

length(⟨ リスト ⟩)	⟨ リスト ⟩ の長さを返却
copylist(⟨ リスト ⟩)	⟨ リスト ⟩ の複製
reverse(⟨ リスト ⟩)	⟨ リスト ⟩ の並びを逆にしたリストを返却
append(⟨ リスト$_1$⟩,⟨ リスト$_2$⟩,...)	複数のリストの結合を実行
cons(⟨ 式$_1$⟩,⟨ 式$_2$⟩)	⟨ 式$_2$⟩ を ⟨ 式$_1$⟩ に追加
endcons(⟨ 式 ⟩,⟨ リスト ⟩)	⟨ 式 ⟩ を ⟨ リスト ⟩ の後に結合
delete(⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨n⟩)	⟨ 式$_1$⟩ を ⟨ 式$_2$⟩ の先頭から ⟨n⟩ 個削除
sort(⟨ リスト ⟩,⟨ 述語 ⟩)	⟨ 述語 ⟩ による置換
sort(⟨ リスト ⟩)	順序 $>_m$ を用いてリストの成分の置換を実施

length 函数: リストの長さを返却する函数で, LISP の同名の函数と同じ動作になります. ただし, Maxima の任意の式は内部表現として LISP のリスト構造を持ち, この length 函数は内部表現で先頭の式の識別子を除いたリストの長さを返す函数になります:

```
(%i22) length([1,2,3]);
(%o22)                                3
(%i23) length([1,2,[1,2,[3,4,5]]]);
(%o23)                                3
(%i24) length(x+y+z);
(%o24)                                3
(%i25) length(x+y*z);
(%o25)                                2
```

この例で, 式 'x+y+z' の内部表現は '((MPLUS SIMP) X Y Z)' なので, length 函数は先頭の "(MPLUS SIMP)" を除いたリストの長さを返しています.

copylist 函数:　リストの複製を行う函数です. なお, この函数は copy 函数を listp 函数を用いてリストに限定した函数です.

reverse 函数:　与えられたリストの並びを逆にしたリストを返却します. たとえば, リスト '[a1,a2,a3]' に対して 'reverse([a1,a2,a3])' とすることで, '[a3,a2,a1]' が得られます. ただし, 成分については, それがリストの場合でもそのままです.
この函数も Maxima のリストに限定されず, 式に対しても操作可能です. 特に 'a > b' の様な中置演算子を持つ式の場合, 'b < a' のように演算子を挟んで左右が変換されます. この場合でも, 内部表現の先頭の演算子に対して逆向きになるので, 'reverse(a*c>b*d)' の結果は '(b*d>a*c)' となります.

append 函数:　複数のリストの結合を行います. Maxima の append 函数は LISP の append 函数と同様の働きをします.

endcons 函数:　append 函数と似た函数ですが, append 函数が先頭に成分を追加するのに対して, endcons 函数はうしろに追加します.
つまり, 'econs(⟨ 式$_1$⟩ ,⟨ リスト$_1$⟩)' で ⟨ 式$_1$⟩ を ⟨ リスト$_1$⟩ のうしろに追加します. ここで, Maxima の式は内部的にはリストとなるために, 'econs(⟨ 式$_1$⟩ ,⟨ 式$_2$⟩)' とすると ⟨ 式$_2$⟩ に ⟨ 式$_1$⟩ が追加されます:

```
(%i43) endcons(x+y,[1,2,3,4]);
(%o43)                          [1, 2, 3, 4, y + x]
(%i44) endcons(x+y,sin(x)+cos(y));
(%o44)                        cos(y) + y + sin(x) + x
(%i45) endcons(x+y,sin(x)/cos(y));
                                 sin(x) (y + x)
(%o45)                           --------------
                                     cos(y)
(%i46) endcons(x+y,sin(x)*cos(y));
(%o46)                         sin(x) (y + x) cos(y)
(%i47) endcons(x+y,sin(x)-cos(y));
(%o47)                       - cos(y) + y + sin(x) + x
```

このように endcons 函数はリストではなく, 式に追加する場合はその式の内部表現に依存します.

delete 函数:　⟨ 式$_2$⟩ に含まれる ⟨ 式$_1$⟩ を式の先頭から ⟨n⟩ 個削除します. ただし, ⟨ 式$_1$⟩ が函数, あるいは単項式でなければ除去出来ません.
なお, ⟨n⟩ はオプションで, 指定しない場合と ⟨ 式$_2$⟩ に含まれる ⟨ 式$_1$⟩ が ⟨n⟩ 個よりも少ない場合に ⟨ 式$_1$⟩ を全て削除します.

sort 函数:　第 1 引数のリストの並び換えを行います. 引数が一つの場合は各成分を Maxima の順序 "$>_m$" を用いて並換えを行います. ここで, 引数が二個の場合, 第 2 引数が 2 変数の真理函数となります:

```
(%i47) sort([1,2,3,4,x,y,z]);
(%o47)                          [1, 2, 3, 4, x, y, z]
```

```
(%i48) sort([1,2,3,4,x,y,z],greaterp);
(%o48)                          [z, y, x, 4, 3, 2, 1]
```

指定した部位を取出す函数

指定した部分式を取出す函数

first(⟨ 式 ⟩)
last(⟨ 式 ⟩)
rest(⟨ 式$_1$⟩ ,⟨n⟩)
sublist(⟨ リスト ⟩ ,⟨ 函数 ⟩)
substpart(⟨x⟩ ,⟨ 式 ⟩ ,⟨n_1⟩, . . .,⟨n_k⟩)

first 函数: ⟨ 式 ⟩ の最初の成分を返します.

last 函数: ⟨ 式 ⟩ の最後の成分を返します.

rest 函数: ⟨n⟩ が正整数であれば ⟨ 式$_1$⟩ の先頭から n 個の成分を除いた式を返します.
ここで n が負整数であれば ⟨ 式$_1$⟩ のうしろから ⟨n⟩ 個の成分を除いた式を返します:

```
(%i52) rest(x+y+z,2);
(%o52)                                x
(%i53) rest(x+y+z,-2);
(%o53)                                z
(%i54) rest([x+y+z,sin(x)+cos(x),exp(x)],-2);
(%o54)                            [z + y + x]
(%i55) rest([x+y+z,sin(x)+cos(x),exp(x)],2);
                                       x
(%o55)                              [%e  ]
```

なお, rest 函数, first 函数と last 函数は Maxima の大域変数 inflag によって与式の内部表現に対する操作に変更出来ます. 大域変数 inflag の初期値が false のため, 内部表現に対して操作したければ大域変数 inflag の値を 'true' に変更する必要があります.

sublist 函数: ⟨ 真理函数 ⟩ が 'true' を返す ⟨ リスト ⟩ に含まれる成分を抜出したリストを返します. たとえば, sublist([1,2,3,4],evenp) によって '[2,4]' を得ます.

substpart 函数: ⟨ 式 ⟩ の ⟨n_1⟩ ,. . .,⟨n_k⟩ で指定した成分を ⟨x⟩ で置換えます. ⟨x⟩ は式, 原子, 演算子が指定できます.
⟨n_1⟩ ,. . .,⟨n_k⟩ の指定方法は, ⟨ 式 ⟩ がリスト, 第 m 番目の成分であれば m を指定します. さらに, リストの m 番目がリストで, その n 番目の成分を入れ替えたければ, 列 m,n で指定します:

```
(%i13) substpart(x,[1,2,3,4],2);
(%o13)                                [1, x, 3, 4]
(%i14) substpart(x,[1,[2,3],4],2);
(%o14)                                  [1, x, 4]
(%i15) substpart(x,[1,[2,3],4],2,2);
(%o15)                               [1, [2, x], 4]
```

最初の例ではリスト '[1,2,3,4]' の第 2 成分の 2 を 'x' で置換えるために '2' を指定しています. 複合リスト '[1,[2,3],4]' の場合, 第 2 成分はリスト '[2,3]' なので, 第 2 成分を 'x' で置換すれば '[1,x,4]' になります. 第 2 成分に含まれる '3' を 'x' で置換えたければ第 2 成分のリストの第 2 成分を指定すれば良いのです.

このことは Maxima の一般の式に対しても適応が可能です. ただし, Maxima の場合は入力した式の順番と Maxima に入力されたあとの順番が異なることがあるので注意が必要になります:

```
(%i16) expr:(x+1)/(x^2+x+1)+exp(x);
                                   x      x + 1
(%o16)                           %e  + ----------
                                         2
                                        x  + x + 1
(%i17) substpart(sin(x),expr,1);
                                            x + 1
(%o17)                          sin(x) + ----------
                                           2
                                          x  + x + 1
(%i18) substpart(sin(x),expr,2);
                                                x
(%o18)                             sin(x) + %e

(%i19) substpart(sin(x),expr,2,2);
                                   x + 1       x
(%o19)                             ------ + %e
                                   sin(x)
(%i20) substpart(sin(x),expr,2,1);
                                   sin(x)        x
(%o20)                           ---------- + %e
                                  2
                                 x  + x + 1
(%i21) substpart("+",expr,2,0);
                                  x     2
(%o21)                          %e  + x  + 2 x + 2
```

この例では式 expr の指定した成分を函数項 'sin(x)' で置換える操作を行っています. ここで第 1 成分は入力した順序とは異なり, '%e^x' となっていることに注意して下さい. 次に第 2 成分は有理式全体となりますが, この第 2 成分の模式的な内部構造は '(/, x+1,x^2+x+1)' となっています. Maxima の内部表現では式はリストで表現され, 演算子が先頭の第 0 成分となり, 以下にその引数が続く構造となっています. そのために, "2, 1" で有理式の分子, "2, 2" で有理式の分母, 最後の "2, 0" が演算子となります. そこで, 'substpart("+", expr, 2, 0)' を実行すると割算が和に置換えられて有理式が 'x^2+2*x+2' で置換されてしまいます.

6.5.5 map 函数族

map 函数は LISP ではお馴染の函数で, 基本的に函数をリストに作用させる函数です. これは Mathematica や Maple でも採用されている非常に便利な函数です. Maxima で式は内部的にリスト構造を持っていますが, 表にはそれが現われていないために map 函数の動作が判り難い側面もあります.
ここでは内部形式と絡めて, map 函数のお仲間について解説します.
Maxima の map 函数のお仲間は全て Maxima の式に函数を式やリスト, あるいは行列等に作用させる働きがあります. この map 函数の基本的な考えは, 函数をリストの各成分に作用させたリストを計算させるものです. ところが Maxima の式は内部的に全てリスト (S 式) であるために, この map 函数のお仲間による式への作用は非常に有効な手段です. そこで, 作用させる函数の特性, 作用させる対象の構造といった諸条件に対応するためにさまざまな map 函数のお仲間が Maxima にはあります.
さて, ここで最初に map 函数の例を示しておきましょう:

```
(%i34) map(sin,x*y);
(%o34)                          sin(x) sin(y)
(%i35) map(sin,x*y+y);
(%o35)                        sin(x y) + sin(y)
(%i36) map(sin,factor(x*y+y));
(%o36)                        sin(x + 1) sin(y)
(%i37) map(lambda([x,y],x*y),x+y,w+z);
(%o37)                             y z + w x
```

最初の式 'x*y' の場合, 主演算子が "*", 第 1 層には被演算子の変数 x と y があるために表徴 sin は式 'x*y' の第 1 層の部分式 'x' と 'y' に作用しますが, 演算子の置換を行わないので 'sin(x)+sin(y)' が得られます. 式 'x*y+y' の場合, この式の第 1 層には二つの部分式 'x*y' と 'w' があるので今度は 'sin(x*y)+sin(y)' となります.
ところが同値な式でも内部表現が異なると異った結果になります. 次の例では 'factor(x*y+y)' の結果に map 函数で sin 函数を作用させますが, 'factor(x*y+y)' が '(x+1)*y' となるので, この式の第 1 層には部分式 'x+1' と 'y' があり, 主演算子が "*" なので, 今度は 'sin(x+1)*sin(y)' が得られます.
次の maplist 函数は基本的に map 函数と同様ですが, こちらは主演算子をリストに置換します:

```
(%i15) map(sin,factor(x*y+y));
(%o15)                        sin(x + 1) sin(y)
(%i16) maplist(sin,factor(x*y+y));
(%o16)                      [sin(x + 1), sin(y)]
(%i17) :lisp %o15;
((MTIMES SIMP) ((%SIN SIMP) ((MPLUS SIMP) 1 X)) ((%SIN SIMP) Y))
(%i17) :lisp %o16;
((MLIST SIMP) ((%SIN SIMP) ((MPLUS SIMP) 1 X)) ((%SIN SIMP) Y))
```

この例に示すように内部表現で MTIMES が MLIST の変化していることに注目して下さい.

6.5.6　map 函数族に関連する大域変数

大域変数 maperror

変数名	既定値	概要
maperror	true	map や maplist 函数を制御します

大域変数 maperror:　この大域変数は map 函数と maplist 函数の動作に影響します. まず, map 函数と maplist 函数の引数は ⟨ 函数 ⟩,⟨ 式$_1$⟩,...,⟨ 式$_n$⟩ で, ⟨ 函数 ⟩ は n 変数の函数です. ここで, 大域変数 maperror の値が 'true' の場合, 各 ⟨ 式$_i$⟩ の主演算子は基本的に同じもので, 成分の個数も同じ個数でなければなりませんが, 大域変数 maperror が 'false' の場合には, それ以外の引数でも適用可能になるので次の動作になります:

1. 全ての ⟨ 式$_i$⟩ が同じ長さでなければ, 最短の ⟨ 式$_j$⟩ を終えた時点で停止します.
2. ⟨ 式$_i$⟩ の主演算子が全て同じものでなければ, [⟨ 式$_1$⟩,...,⟨ 式$_n$⟩] に ⟨ 函数 ⟩ を作用させ, apply 函数と同じ動作になります.

大域変数 maperror が 'true' の場合, 上の二つの状況であればエラーメッセージが出力されます:

```
(%i40) maperror:false;
(%o40)                                false
(%i41) map(lambda([x,y],x*y),x+y+a,w+z);
'map' is truncating.
(%o41)                              y z + w x
(%i42) map(lambda([x,y],x*y),x+y+a,w*z);
'map' is doing an 'apply'.
(%o42)                          w (y + x + a) z
(%i43) maperror:true;
(%o43)                                true
(%i44) map(lambda([x,y],x*y),x+y+a,w*z);
Arguments to 'mapl' not uniform - cannot map.
 -- an error.  Quitting.  To debug this try debugmode(true);
```

6.5.7　map 函数いろいろ

map 函数に関連する函数

map(⟨ 函数 ⟩,⟨ 式$_1$⟩, ...,⟨ 式$_n$⟩)
maplist(⟨ 函数 ⟩,⟨ 式$_1$⟩, ⟨ 式$_2$⟩,...)
mapatom(⟨ 式 ⟩) scanmap (⟨ 函数 ⟩,⟨ 式 ⟩)
scanmap(⟨ 函数 ⟩,⟨ 式 ⟩,bottomup)
fullmap(⟨ 函数 ⟩,⟨ 式$_1$⟩, ..., ⟨ 式$_n$⟩, ...)
fullmapl(⟨ 函数式 ⟩,⟨ 式$_1$⟩, ..., ⟨ 式$_n$⟩, ...)
outermap(⟨ 函数 ⟩,⟨ 式$_1$⟩, ..., ⟨$_n$⟩)

map 函数: n 個の式の列 ⟨ 式$_1$⟩,...,⟨ 式$_n$⟩ から n 個の成分を取出し, n 個の引数を取る ⟨ 函数 ⟩ を作用させた結果を返します.

maplist 函数: ⟨ 式$_i$⟩ の各成分に ⟨ 函数 ⟩ を作用させたリストを生成します. ⟨ 函数 ⟩ は函数の名前や lambda 式となります.
maplist が map 函数と異なる点は, map 函数では ⟨ 式$_i$⟩ の主演算子で式を纏めた式が出力されるのに対し, maplis 函数 t では主演算子の代りに Maxima のリスト演算子が置かれる点です:

```
(%i27) maplist(sin,x+y);
(%o27)                          [sin(y), sin(x)]
(%i28) map(sin,x+y);

(%o28)                           sin(y) + sin(x)
(%i29) maplist(lambda([x,y],x*y),x+y,w+z);
(%o29)                             [y z, w x]
(%i30) map(lambda([x,y],x*y),x+y,w+z);
(%o302)                             y z + w x
```

scanmap 函数: scanmap⟨ 函数 ⟩,⟨ 式 ⟩) の場合は再帰的に ⟨ 函数 ⟩ を ⟨ 式 ⟩ の内部表現の頂上から適用します. これは徹底した因子分解が望ましいときには特に便利です.
たとえば, $(a^2+2a+1)y+x^2$ を factor で因子分解しても, そのまま元の式が返却されるだけですが, scanmap 函数を使って factor 函数を式に作用させると与式の部分式 a^2+2a+1 が因子分解された結果が返されます:

```
(%i3) exp:(a^2+2*a+1)*y + x^2
(%i4) factor(exp);
                              2                    2
(%o4)                        a  y + 2 a y + y + x
(%i5) scanmap(factor,exp);
                                     2       2
(%o5)                          (a + 1)  y + x
```

scanmap 函数による結果は式の内部表現に依存します. 上の例の式の内部表現を次に示しておきます:

```
((MPLUS SIMP)
    ((MEXPT SIMP) X 2)
    ((MTIMES SIMP)
          ((MPLUS SIMP) 1 ((MTIMES SIMP) 2 A)
                          ((MEXPT SIMP) A 2)) Y))
```

この内部表現に対して scanmap 函数は factor 函数を上の階層から各部分式に対して作用させます. この式の場合は最初に x^2 と $(a^2+2a+1)y$ に factor 函数を作用させ, その次に x^2 の x と 2 , a^2+2a+1 と y …と下の階層の成分に作用します. その結果, a^2+2a+1(=(+ 1 (* 2 a) (^ a 2))) の展開以外はそのままなので上記の結果を得ます. この作用の様子は factor 函数の代りに形式的函数 f の名詞型'f を与えると判り易くなります:

```
(%i18) scanmap('f,exp);
                    f(2)                                          f(2)
(%o18) (f(f(f(f(a)     ) + f(f(2) f(a)) + f(1)) f(y)) + f(f(x)     ))
```

この性質があるために式を全て展開してしまうと, 各項に factor 函数を作用させるので入力値がそのまま返却されてしまいます:

```
(%i16) expand(exp);
                              2                    2
(%o16)                       a  y + 2 a y + y + x
(%i17) scanmap(factor,expand(exp));
                              2                    2
(%o18)                       a  y + 2 a y + y + x
```

scanmap(⟨ 函数 ⟩,⟨ 式 ⟩,bottomup) は scanmap(⟨ 函数 ⟩,⟨ 式 ⟩) とは逆に内部表現の最下層側から ⟨ 函数 ⟩ を作用させます:

fullmap 函数: ⟨ 函数 ⟩ を第 2 引数以降の式に対して作用させる函数です:

```
(%i38) fullmap(sin,[1,2,3]);
(%o38)                    [sin(1), sin(2), sin(3)]
(%i39) fullmap(lambda([x,y],x-y),[1,2,3],[3,2,1]);
(%o39)                          [- 2, 0, 2]
(%i40) fullmap(sin,a+b+c*d);
(%o40)                 sin(c) sin(d) + sin(b) + sin(a)
```

fullmapl 函数: fullmap 函数と似た働きをしますが, fullmap 函数の第 2 引数以降は全てリスト, あるいは行列となる点で異なります:

```
(%i41) fullmapl(sin,[1,2,3]);
(%o41)                    [sin(1), sin(2), sin(3)]
(%i42) fullmapl(lambda([x,y],x-y),[1,2,3],[3,2,1]);
(%o42)                          [- 2, 0, 2]
(%i43) fullmapl(sin,[a+b+c*d]);
(%o43)                        [sin(c d + b + a)]
(%i44) fullmapl(sin,[[a],[b],[c*d]]);
(%o44)                 [[sin(a)], [sin(b)], [sin(c d)]]
```

outermap 函数: 与えられた ⟨ 函数 ⟩ を ⟨ 式$_1$⟩,...,⟨ 式$_n$⟩ 分配することで生成した各成分の長さが n, 成分の総数が `length`(⟨ 式$_1$⟩)·`length`(⟨ 式$_2$⟩)...`length`(⟨ 式$_n$⟩) 個のリストに作用させる函数です:

```
(%i1) outermap(f,[1,2,3],[4,5],[a,b,c,d,e]);
(%o1) [[[f(1, 4, a), f(1, 4, b), f(1, 4, c), f(1, 4, d), f(1, 4, e)],
[f(1, 5, a), f(1, 5, b), f(1, 5, c), f(1, 5, d), f(1, 5, e)]],
[[f(2, 4, a), f(2, 4, b), f(2, 4, c), f(2, 4, d), f(2, 4, e)],
[f(2, 5, a), f(2, 5, b), f(2, 5, c), f(2, 5, d), f(2, 5, e)]],
[[f(3, 4, a), f(3, 4, b), f(3, 4, c), f(3, 4, d), f(3, 4, e)],
[f(3, 5, a), f(3, 5, b), f(3, 5, c), f(3, 5, d), f(3, 5, e)]]]
```

mapatom 函数: 〈式〉が map 函数によって原子として扱われるときに 'true' を返す函数です:

6.5.8 apply 函数

リストを引数に使う函数で, map 函数と並んで重要な函数として apply 函数があります. この apply 函数は Maple や Mathematica にもある函数です:

apply 函数

apply(〈函数名〉,〈リスト〉)

apply 函数は〈函数名〉を〈リスト〉に適用した結果を与えます.
たとえば apply(min,[1,5,-10.2,4,3]) は-10.2 となります.
函数の呼出しに於て, その未評価の引数の評価で apply 函数は便利です. たとえば, filespec をリスト '[test,case]' とするときに, 'apply(closefile,filespec)' と 'closefile(test,case)' は同値です.
ここで〈函数名〉は Maxima の記号となりますが, この記号を変数名とする変数に何らかの値が割当てられているとき, apply 函数による評価で函数名が評価された結果, 同名の変数に割当てられた値で置換えられてしまいます. この事態を避けるために単引用符 "'" を用いて名詞型として apply 函数に引渡せば安全です. 勿論, 函数と同名の変数を用いないようにすることも重要です:

```
(%i30) neko(x):=2*x;
(%o30)                              neko(x) := 2 x
(%i31) neko:-128;
(%o31)                                   - 128
(%i32) neko(10);
(%o32)                                    20
(%i33) apply('neko,[20]);
(%o33)                                    40
(%i34) appply(neko,[20]);
(%o34)                             appply(- 128, [20])
```

この例では neko を函数名としていますが, 同名の変数 neko には-128 を束縛しています. その結果, apply 函数による評価では単引用符 "'" を用いて名詞型にしていなければ, apply 函数による評価から同名の変数に束縛された値で置換えられていることに注意して下さい.

6.5.9 リストを使った四則演算

Maxima では MATLAB 風にリストを使った四則演算が可能です. つまり, この演算が可能となるのは次の二種類の状況に限定されます:

1. 双方が同じ長さのリスト
2. 一方がリストで, 片方が原子の場合

まず, 1. の場合はリストの成分同士の演算となります. つまり, 演算子を "∘" とするとき $[a_1,\ldots,a_n]\circ[b_1,\ldots,b_n] \Rightarrow [a_1\circ b_1,\ldots,a_n\circ b_n]$ と処理されます:

```
(%i36) [1,2,3,4,5]*[5,4,3,2,1];
(%o36)                          [5, 8, 9, 8, 5]
(%i37) [1,2,3,4,5]+[5,4,3,2,1];
(%o37)                          [6, 6, 6, 6, 6]
(%i38) [1,2,3,4,5]-[5,4,3,2,1];
(%o38)                       [- 4, - 2, 0, 2, 4]
(%i39) [1,2,3,4,5]/[5,4,3,2,1];
                                 1  1
(%o39)                          [-, -, 1, 2, 5]
                                 5  2
```

次の 2. の場合は, ベクトルとスカラーの演算に似た状況になります. すなわち, $[a_1,\ldots,a_n] \circ b \Rightarrow [a_1 \circ b,\ldots,a_n \circ b]$ と処理されます:

```
(%i40) [1,2,3,4,5]*2;
(%o40)                       [2, 4, 6, 8, 10]
(%i41) [1,2,3,4,5]+2;
(%o41)                        [3, 4, 5, 6, 7]
(%i42) [1,2,3,4,5]-2;
(%o42)                       [- 1, 0, 1, 2, 3]
(%i43) [1,2,3,4,5]/2;
                               1    3    5
(%o43)                        [-, 1, -, 2, -]
                               2    2    2
```

このような柔軟なリストの処理も可能ですが, この性質は全ての二項演算子で可能な訳ではありません.

6.6 集合について

6.6.1 概要

Maxima の集合は原子, 式, リストや集合等の成分を中括弧 “{}” で括った対象です. この中括弧 “{}” は matchfix 型演算子として宣言されており, 内部では第 1 成分に “($SET SIMP)” が置かれた S 式として表現されています.
集合の生成は単純に成分を中括弧で括ったものを直接入力して生成するか, setify 函数や fullsetify 函数を用いてリストから生成します.
Maxima の集合は, その集合の成分の並びに意味がないために, orderlessp 函数を用いて成分を並べ替えた対象で置換えられます. なお, この並び換えで用いられる順序は Maxima の順序 “$>_m$” です (§5.2 参照).
なお, 集合を生成する際に各成分の評価が行われ, 同値なものが存在すれば, 代表のみが選択されて Maxima の順序 “$>_m$” に従って成分の並び換えが行われます. なお, 論理式を成分とする集合では, これらの論理式の評価は行われません:

```
(%i117) {1,2,3,4,5,4/2,cos(0)};
(%o117)                          {1, 2, 3, 4, 5}
(%i118) a:1;b:sin(%pi/4)$
(%i120) {1,{2},{{a}},b,{a,{{{b},a*b}}}};
                    1                1          1
(%o120)       {1, -------, {1, {{-------, {-------}}}}, {2}, {{1}}}
                  sqrt(2)          sqrt(2)    sqrt(2)
(%i121) S1:{1>3,3<54,5=5};
(%o121)                        {1 > 3, 3 < 54, 5 = 5}
(%i122) S2:setify([1>3,3<54,5=5]);
(%o122)                        {1 > 3, 3 < 54, 5 = 5}
(%i123) S3:map(lambda([x],ev(x,pred)),{1>3,3<54,5=5});
(%o123)                            {false, true}
```

集合はリストに似ているためにリストのような処理が行えそうですが, リストでできる操作が集合に対して行えるとは限りません.
次に利用可能な函数名を列記しておきますが, 一応使えなくもない程度で, 将来のことを考えると希望する函数が存在しない場合には集合を listify 函数や full_listify 函数で一旦リストに変換してリスト処理の函数を用いる方が無難でしょう:

- 集合に対して適用可能な函数
  length,member,append,cons,delete,endcons,first,rest,last,map,maplist,scanmap
- 無意味な函数
  reverse,substpart

6.6.2 集合の生成に関連する函数

集合の生成は, その成分で構成された列を演算子 “{}” で括ることで生成出来ますが, 定型的な式で構築可能な場合には, その式を用いた方法で集合やリストを生成したり, 既存のリストから集合を生成する方法があります:

集合やリストを生成する函数

makeset(⟨ 式 ⟩,⟨ リスト$_1$⟩,⟨ リスト$_2$⟩)
cartesian_product(⟨ 集合$_1$⟩,...,⟨ 集合$_n$⟩)
divisors(⟨ 正整数 ⟩)
setify(⟨ リスト ⟩)
fullsetify(⟨ リスト ⟩)

makeset 函数: ⟨ 式 ⟩ から集合を生成する函数です. まず, 第 2 引数の ⟨ リスト$_1$⟩ は ⟨ 式 ⟩ に含まれる変数で構成されたリストで, ここで指定した各変数の値域が ⟨ リスト$_2$⟩ で指定するリストになります. なお, ⟨ リスト$_2$⟩ の各成分は ⟨ リスト$_1$⟩ で指定した変数と対応がつかなければなりません:

```
(%i15) makeset(i+j*x,[i,j],[[1,2],[3,1],[4,3]]);
(%o15)                          {x + 3, 2 x + 1, 3 x + 4}
(%i16) makeset(i+j*x,[x,j],[[1,2],[3,1],[4,3]]);
(%o16)                           {i + 2, i + 3, i + 12}
(%i17) makeset(i+j*x,[i],[[1],[3],[4]]);
(%o17)                         {j x + 1, j x + 3, j x + 4}
(%i18) makeset(i+j*x,[i],[[1]]);
(%o18)                                  {j x + 1}
```

cartesian_product 函数: 直積集合を生成する函数です. 集合を $S_1,\ldots,S_n$ とすると $[s_1,\ldots,s_n] \in S_1 \times \cdots S_n$ で構成された集合を返します.
なお, 集合の成分には順序がありませんが, 直積集合の各成分はリストとなるために, 直積集合の各成分には順序があります:

```
(%i26) cartesian_product({1,2,3},{4,5,6});
(%o26) {[1, 4], [1, 5], [1, 6], [2, 4], [2, 5], [2, 6], [3, 4], [3, 5], [3, 6]}
(%i27) cartesian_product({1,2,3},{4,5});
(%o27)          {[1, 4], [1, 5], [2, 4], [2, 5], [3, 4], [3, 5]}
```

divisors 函数: 与えられた整数を割切る正整数 (因子) で構成されたリストを返す函数です. 負の整数が与えられた場合は内部で絶対値を取って処理を行います. なお, この函数の内部では factor 函数が用いられています:

```
(%i36) divisors(-290348);
(%o36) {1, 2, 4, 29, 58, 116, 2503, 5006, 10012, 72587, 145174, 290348}
(%i37) divisors(2903488432);
(%o37) {1, 2, 4, 8, 13, 16, 26, 52, 104, 208, 13959079, 27918158, 55836316,
 111672632, 181468027, 223345264, 362936054, 725872108, 1451744216, 2903488432}
```

setify 函数: 与えられたリストから集合を生成する函数です:

```
(%i18) a:setify([1,2,3,4,5]);
(%o18)                                {1, 2, 3, 4, 5}
(%i19) to_lisp();
```

```
Type (to-maxima) to restart, ($quit) to quit Maxima.

MAXIMA> $a
(($SET SIMP) 1 2 3 4 5)
MAXIMA> (to-maxima)
Returning to Maxima
(%o19)                                  true
(%i20) b:setify([ 10,x,3,104,-5]);
(%o20)                          {-5, 3, 10, 104, x}
```

集合を生成する際に LISP の sort 函数を用いて成分が再配置されますが, その際の順序の判定を行う述語として orderlessp 函数が用いられます. その結果, orderlessp 函数で最小ものが左端に置かれ, 以降右に行くに従い大きくなります.

fullsetify 函数: setify 函数と同様に与えられたリストから集合を生成する函数です. ただし, setify 函数では最上部が集合になるだけで各成分は以前の型をそのまま保ちますが, fullsetify 函数を用いると全てが集合になります:

```
(%i24) setify([[1,2],3,[4],5]);
(%o24)                          {3, 5, [1, 2], [4]}
(%i25) fullsetify([[1,2],3,[4],5]);
(%o25)                          {3, 5, {1, 2}, {4}}
```

集合からリストの生成を行う函数

setify 函数と fullsetify 函数の逆の操作を行う函数が listify 函数と fulllistify 函数です:

集合からリストを生成する函数

listify(⟨ 集合 ⟩)
full_listify(⟨ 集合 ⟩)

listify 函数: 集合からリストを生成します. ただし, 集合の元に集合が含まれていても集合の元はそのままです:

```
(%i28) listify(b1);
(%o28)                          [3, 5, {1, 2}, {4}]
(%i29) listify(a1);
(%o29)                          [3, 5, [1, 2], [4]]
```

full_listify 函数: 集合からリストを生成します. ここで集合の元が集合である場合, その元も全てリストで置換えます:

6.6.3　リスト操作の函数

リスト操作の函数

flatten(⟨ リスト ⟩)
unique(⟨ リスト ⟩)

flatten 函数:　与えられた Maxima の複合リストを Maxima の平リストに変換する函数です. ただし, Maxima のリストに対してのみ効果があり, 集合や式は無変換で, そのままの式を返します:

```
(%i6) flatten([1,2,3,[4,[5,4],4]]);
(%o6)                          [1, 2, 3, 4, 5, 4, 4]
(%i7) flatten({1,2,3,[4,[5,4],4]});
(%o7)                       {1, 2, 3, [4, [5, 4], 4]}
```

unique 函数:　リストの各成分に対して orderlessp 函数を述語とし, 順序 "$>_m$" で成分の並び換えを実行する函数です. 左から右への順序が "$>_m$" の小から大の順に対応します.
ただし, unique 函数はリストの第 1 層の並び換えを行い, より深い階層に作用はしません:

```
(%i3) unique([1,2,34]);
(%o3)                             [1, 2, 34]
(%i4) unique([2,34,1]);
(%o4)                             [1, 2, 34]
(%i5) unique([2,34,[3,1]]);
(%o5)                          [2, 34, [3, 1]]
```

6.6.4　集合演算の函数

集合演算の函数

演算子	構文
∪	union(⟨ 集合$_1$⟩,⋯,⟨ 集合$_n$⟩)
∩	intersection(⟨ 集合$_1$⟩,⋯,⟨ 集合$_n$⟩)
∩	intersect(⟨ 集合$_1$⟩,⋯,⟨ 集合$_n$⟩)
∈	symmdifference(⟨ 集合$_1$⟩,⋯,⟨ 集合$_n$⟩)
\	setdifference(⟨ 集合$_1$⟩,⟨ 集合$_2$⟩)

union 函数:　引数の集合全ての和集合を返す函数です:

```
(%i21) union({1,2,3},{3,4,5});
(%o21)                          {1, 2, 3, 4, 5}
(%i22) union({{1,2,3}},{3,4,5});
(%o22)                       {3, 4, 5, {1, 2, 3}}
(%i23) union({1,2,3},{3,4,5},{a,b,c});
(%o23)                    {1, 2, 3, 4, 5, a, b, c}
```

intersection 函数と intersect 函数: 一つ以上の集合を引数にし, 与えられた集合の共通集合を返します. intersect 函数は intersection 函数の別名といっても良い程で, intersect 函数の内部では apply 函数を用いて intersection 函数を作用させているだけです:

```
(%i26) intersection({1,2,3,4},{4,5,6});
(%o26)                                {4}
(%i27) intersection({1,2,{3,4}},{4,5,6});
(%o27)                                {}
```

setdifference 函数: 第 1 引数の集合から第 2 引数の集合を除去した集合を返します. ここで第 1 引数の集合が空集合 '{}' (= ∅) であれば結果も空集合 '{}' になります. なお, 引数が集合でない場合にはエラーを返します:

```
(%i106) setdifference({1,2,3},{3,4});
(%o106)                              {1, 2}
(%i107) setdifference({1,2,3},{});
(%o107)                             {1, 2, 3}
(%i108) setdifference({},{3,4});
(%o108)                               {}
(%i109) setdifference({1,2,3},{{1}});
(%o109)                             {1, 2, 3}
```

symmdifference 函数: 与えられた複数の集合の共通部分を削除した集合を返却します:

```
(%i87) S_1 : {a, b, c};
(%o87)                              {a, b, c}
(%i88) S_2 : {1, b, c};
(%o88)                              {1, b, c}
(%i89) S_3 : {a, b, z};
(%o89)                              {a, b, z}
```

6.6.5 集合操作の函数

集合操作の函数

adjoin(⟨ 対象 ⟩,⟨ 集合 ⟩)
disjoin(⟨ 対象 ⟩,⟨ 集合 ⟩)
permutation(⟨ 集合 ⟩)
powerset(⟨ 集合 ⟩,⟨ 正整数値 ⟩)
random_premutation(⟨ 集合 ⟩)

adjoin 函数: 第 1 引数の対象を第 2 引数の集合の成分とした集合を返す函数です:

```
(%i15) adjoin(3,{1,4,5});
(%o15)                               {1, 3, 4, 5}
(%i16) adjoin({1,2,3},{a,b});
(%o16)                            {{1, 2, 3}, a, b}
```

disjoin 函数:　第 1 引数の対象を第 2 引数の集合から除去した集合を返す函数です:

```
(%i18) disjoin(a,{a,b});
(%o18)                                {b}
(%i19) disjoin({a,b},{a,b});
(%o19)                               {a, b}
(%i20) disjoin({a,b},{1,2,{a,b}});
(%o20)                               {1, 2}
```

permutations 函数:　引数の集合の第 1 層の成分に全ての置換を作用させて得られたリストから構成される集合を返します.

powerset 函数:　与えられた集合の全ての部分集合で構成された集合を返します. また, 第 2 引数に正整数値を指定することで正整数値で指定された基数を持つ部分集合の集合を返します:

```
(%i78) powerset({1,2,3,4,5});
(%o78) {{}, {1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5},
{1, 2, 3, 5}, {1, 2, 4}, {1, 2, 4, 5}, {1, 2, 5}, {1, 3}, {1, 3, 4},
{1, 3, 4, 5}, {1, 3, 5}, {1, 4}, {1, 4, 5}, {1, 5}, {2}, {2, 3}, {2, 3, 4},
{2, 3, 4, 5}, {2, 3, 5}, {2, 4}, {2, 4, 5}, {2, 5}, {3}, {3, 4}, {3, 4, 5},
{3, 5}, {4}, {4, 5}, {5}}
(%i79) powerset({1,2,3,4,5},3);
(%o79) {{1, 2, 3}, {1, 2, 4}, {1, 2, 5}, {1, 3, 4}, {1, 3, 5}, {1, 4, 5},
                                  {2, 3, 4}, {2, 3, 5}, {2, 4, 5}, {3, 4, 5}}
(%i80)
```

random_premutation 函数:　乱数を用いて集合やリストの成分の置換を行ったものを返す函数です. 置換は集合の第 1 層に対してのみ実行されて集合の成分には及びません:

```
(%i31) random_permutation({1,2,3,4,5});
(%o31)                             [3, 4, 5, 1, 2]
(%i32) random_permutation({1,2,3,4,5});
(%o32)                             [5, 1, 2, 3, 4]
(%i33) random_permutation({1,2,3,4,5});
(%o33)                             [4, 5, 1, 2, 3]
(%i34) random_permutation({1,2,3,{4,5}});
(%o34)                           [1, 2, {4, 5}, 3]
(%i35) random_permutation({1,2,3,{4,5}});
(%o35)                           [1, 2, 3, {4, 5}]
```

Maxima には与えられた集合を指定した函数を用いて分類する函数が幾つかあります. なお, この分類で用いる函数のことを簡単に判別函数と呼ぶことにします:

集合の分類を行う函数

external_subset(⟨ 集合 ⟩,⟨ 函数 ⟩,max)
external_subset(⟨ 集合 ⟩,⟨ 函数 ⟩,min)
equiv_classes(⟨ 集合 ⟩,⟨2 変数の述語 ⟩)
subset(⟨ 集合 ⟩,⟨ 述語 ⟩)
partition_set(⟨ 集合 ⟩,⟨ 述語 ⟩)

extremal_subset 函数: 第 1 引数の集合に第 2 引数に与えた判別函数を作用させ, 第 3 引数が 'max' の場合は第 2 引数の判別函数が最大値を取る成分で構成される集合, 同様に 'min' の場合は判別函数が最小値を取る成分で構成される集合を返します:

```
(%i104) extremal_subset({1,2,3,4,5},lambda([x],abs(x-3)),max);
(%o104)                                {1, 5}
(%i105) extremal_subset({1,2,3,4,5},sin,min);
(%o105)                                  {5}
```

equiv_classes: 与えられた集合を判別函数を用いて同値なものの集合に分類する函数です. ここでの判別函数は 2 変数の述語函数であり, そのために equiv_classes 函数は与えられた集合を同値類で分類する函数になります:

```
(%i115) equiv_classes({1,2,3,4,5},lambda([x,y],abs(x-3)=abs(y-3)));
(%o115)                        {{1, 5}, {2, 4}, {3}}
```

subset 函数: 与えられた集合から第 2 引数で指定した判別函数が 'true' を返す成分から構成される集合を返します. したがって, 第 2 引数の判別函数は 1 変数の述語函数になります:

```
(%i67) subset({1,2,3,4,5},lambda([x],is(x>3)));
(%o67)                                 {4, 5}
(%i68) subset({1,2,3,4,5},lambda([x],ev(x>3,pred)));
(%o68)                                 {4, 5}
```

ここで論理式を評価する場合, 判別函数として is 函数, あるいは, ev 函数に pred を引数に追加した函数も使えます.

partition_set 函数: 与えられた集合を指定の述語を満す成分の集合と述語を満さない成分の集合の二つで構成される集合を返します:

```
(%i70) partition_set({1,2,3,4,5},lambda([x],ev(x>3,pred)));
(%o70)                          [{1, 2, 3}, {4, 5}]
(%i71) partition_set({1,2,3,4,5},lambda([x],ev(x<3,pred)));
(%o71)                          [{3, 4, 5}, {1, 2}]
(%i72) partition_set({1,2,3,4,5},lambda([x],ev(x<8,pred)));
(%o72)                         [{}, {1, 2, 3, 4, 5}]
(%i73) partition_set({1,2,3,4,5},lambda([x],ev(x>8,pred)));
(%o73)                         [{1, 2, 3, 4, 5}, {}]
```

この場合, 集合の第 1 成分が評価函数を満さない元, すなわち, 返される値が false の場合や unknown になる集合で, 第 2 成分が評価函数を満す元の集合になります. なお, 評価函数は 1 変数の述語です.

6.6.6 集合に関連する函数

集合の濃度/基数を返す函数

cardinarity(⟨ 集合 ⟩)

cardinality 函数:　与えられた集合の基数/濃度を返す函数です:

```
(%i58) cardinality({1,2,3,4});
(%o58)                                4
(%i59) cardinality({{},{{}}});
(%o59)                                2
(%i60) cardinality({1,2,{3,4}});
(%o60)                                3
```

集合やリストに含まれる元の最大値や最小値を返す函数

lmax(⟨ リスト ⟩)
lmax(⟨ 集合 ⟩)
lmix(⟨ リスト ⟩)
lmix(⟨ 集合 ⟩)

lmax 函数と lmin 函数:　lmax 函数と lmin 函数は引数として集合やリストを取り, その中で最大値, あるいは最小値を返す函数です. これらの函数の実体は lmax 函数が max 函数, lmin 函数が min 函数で, 単純にリストの演算子 "[]" や集合の演算子 "{}" を外した数列を max 函数や min 函数に引き渡すだけです:

```
(%i38) lmax([1,-2,3,-4]);
(%o38)                                3
(%i39) lmin([1,-2,3,-4]);
(%o39)                               - 4
(%i40) lmin([[1,-2],{3},-4]);
(%o40)                      min(- 4, {- 2, 1}, {3})
(%i41) lmax([[1,-2],{3},-4]);
(%o41)                      max(- 4, {- 2, 1}, {3})
```

reduce 函数族

reduce 函数族は基本的に第 1 引数の函数を第 2 引数のリストや集合に作用させる函数ですが, この作用のさせ方の違いから 4 種類の函数になっています.

reduce 函数族

lreduce(⟨ 函数 ⟩,⟨ 式 ⟩)
lreduce(⟨ 函数 ⟩,⟨ 式$_1$⟩,⟨ 式$_0$⟩)
rreduce(⟨ 函数 ⟩,⟨ 式 ⟩)
rreduce(⟨ 函数 ⟩,⟨ 式$_1$⟩,⟨ 式$_0$⟩)
xreduce(⟨ 函数 ⟩,⟨ 式 ⟩)
xreduce(⟨ 函数 ⟩,⟨ 式$_1$⟩,⟨ 式$_0$⟩)
tree_reduce(⟨ 函数 ⟩,⟨ 式 ⟩)
tree_reduce(⟨ 函数 ⟩,⟨ 式$_1$⟩,⟨ 式$_0$⟩)

lreduce 函数: lreduce(F, L) のときに F(. . . , F(F(l_1, l_2), l_3), . . . , l_n) を返し, lreduce(F, L, X) に対しては F(. . . , F(F(X, l_1), l_2), . . . , l_n) を返す函数です:

```
(%i42) lreduce(f,[x_1,x_2,x_3,x_4]);
(%o42)                    f(f(f(x_1, x_2), x_3), x_4)
(%i43) lreduce(f,[x_1,x_2,x_3,x_4],X);
(%o44)                  f(f(f(f(X, x_1), x_2), x_3), x_4)
```

rreduce 函数: rreduce(F, L) のとき, F(l_1, . . . , F(l_{n-2}, F(l_{n-1}, l_n))) を返し, rreduce(F, L, X) に対しては F(l_1, . . . , F(l_{n-1}, F(l_n, X))) を返す函数です.

```
(%i44) rreduce(f,[x_1,x_2,x_3,x_4]);
(%o44)                    f(x_1, f(x_2, f(x_3, x_4)))
(%i45) rreduce(f,[x_1,x_2,x_3,x_4],X);
(%o45)                  f(x_1, f(x_2, f(x_3, f(x_4, X))))
```

xreduce 函数: 第 1 引数の演算子が nary 型である場合を除いて lreduce 函数と全く同じです.
F を nary 型の演算子するときに xreduce(F, L) は F(((l_1, l_2, l_3, . . . , l_n) を返し, xreduce(F, L, X) に対しては F(X, l_1, l_2, . . . , l_n) を返します:

```
(%i66) xreduce(g,[x_1,x_2,x_3,x_4,x_5]);
(%o66)                  g(g(g(g(x_1, x_2), x_3), x_4), x_5)
(%i67) xreduce(g,[x_1,x_2,x_3,x_4,x_5],X);
(%o67)              g(g(g(g(g(X, x_1), x_2), x_3), x_4), x_5)
(%i68) declare(g,nary);
(%o68)                                done
(%i69) xreduce(g,[x_1,x_2,x_3,x_4,x_5]);
(%o69)                    f(x_1, x_2, x_3, x_4, x_5)
(%i70) xreduce(g,[x_1,x_2,x_3,x_4,x_5],X);
(%o70)                    f(X, x_1, x_2, x_3, x_4, x_5)
```

tree_reduce 函数: tree_reduce(F, L) のときに F(F(. . . , F(F(l_1, l_2), F(l_3, l_4)), . . .), l_n)] を返し, tree_reduce(F, L, X) に対しては F(F(. . . , F(F(X, l_1), F(l_2, l_3)), . . .), l_n)] を返す函数です:

```
(%i49) tree_reduce(f,[x_1,x_2,x_3,x_4,x_5]);
(%o49)                  f(f(f(x_1, x_2), f(x_3, x_4)), x_5)
(%i50) tree_reduce(f,[x_1,x_2,x_3,x_4,x_5],X);
(%o50)              f(f(f(X, x_1), f(x_2, x_3)), f(x_4, x_5))
```

6.6.7 分割に関連する函数

集合を複数の集合に分割したり, 整数を集合に分割する函数として次のものがあります:

分割に関連する函数

set_partitions(⟨ 集合 ⟩)
set_partitions(⟨ 集合 ⟩,⟨ 正整数値 ⟩)
integer_partitions(⟨ 正整数値 ⟩)
integer_partitions(⟨ 正整数値 ⟩,⟨ 正整数値 ⟩)
num_partitions(⟨ 正整数値 ⟩)
num_partitions(⟨ 正整数値 ⟩,list)
num_distinct_partitions(⟨ 正整数値 ⟩)
num_distinct_partitions(⟨ 正整数値 ⟩,list)

set_partition 函数:　与えられた集合の分割を返す函数です. 第 2 引数を指定しない場合は全ての分割を返します.

integer_partition:　指定した正整数値に対し, 和がその整数値になるリストを成分とする集合を返します. 第 2 引数を指定した場合にはリストの長さが第 2 引数となる対象の集合を返しますが, 第 2 引数の値が第 1 引数よりも大きな場合, 第 2 引数を指定しない場合の結果で, 各成分が第 2 引数の値の長さになるように, 0 を追加したリストの集合を返します:

```
(%i64) integer_partitions(5);
(%o64) {[1, 1, 1, 1, 1], [2, 1, 1, 1], [2, 2, 1],
                          [3, 1, 1], [3, 2], [4, 1], [5]}
(%i65) integer_partitions(5,1);
(%o65)                                {[5]}
(%i66) integer_partitions(5,3);
(%o66)      {[2, 2, 1], [3, 1, 1], [3, 2, 0], [4, 1, 0],
                          [5, 0, 0]}
(%i67) integer_partitions(5,5);
(%o67) {[1, 1, 1, 1, 1], [2, 1, 1, 1, 0], [2, 2, 1, 0, 0],
        [3, 1, 1, 0, 0], [3, 2, 0, 0, 0], [4, 1, 0, 0, 0],
                          [5, 0, 0, 0, 0]}
```

num_pertitions 函数:　同一引数の integer_partitions 函数による結果の集合の基数を返す函数です. 第 2 引数に list を指定した場合, 1 から第 1 引数までの整数の分割に対応する分割数を返します. なお, このときに返却されるリストの第 1 成分は 1 で, 実際の分割数は第 2 成分以降になります:

```
(%i88) is(num_partitions(5)=cardinality(integer_partitions(5)));
(%o88)                                true
(%i89) num_partitions(7,list);
(%o89)                    [1, 1, 2, 3, 5, 7, 11, 15]
(%i90) map(lambda([x],cardinality(integer_partitions(x))),[1,2,3,4,5,6,7]);
(%o90)                      [1, 2, 3, 5, 7, 11, 15]
```

num_distinct_partitions 函数: 第 1 引数で指定された正整数に対応する分割で, 各成分が異なる分割の個数を返します. また, 第 2 引数として 'list' を指定すると, 1 から第 1 引数の正整数までの成分が異なる分割の個数をリストで返します. このときに実際の分割数は第 2 成分以降になります:

```
(%i92) integer_partitions(5);
(%o92) {[1, 1, 1, 1, 1], [2, 1, 1, 1], [2, 2, 1], [3, 1, 1], [3, 2], [4, 1],
                                                                        [5]}
(%i93) num_distinct_partitions(5);
(%o93)                                3
(%i94) num_distinct_partitions(5,list);
(%o94)                          [1, 1, 1, 2, 2, 3]
```

6.6.8 集合に関連する真理値函数

集合に関連する真理値函数

setp(⟨ 対象 ⟩)
emptyp(⟨ 集合 ⟩)
elementp(⟨ 要素 ⟩,...,⟨ 集合 ⟩)
subsetp(⟨ 集合$_1$⟩,⟨ 集合$_2$⟩)
setequalp(⟨ 集合$_1$⟩,⟨ 集合$_2$⟩)
disjointp(⟨ 集合$_1$⟩,...,⟨ 集合$_2$⟩)

setp 函数: 与えられた対象が集合型であるかどうかを判別する函数です

emptyp 函数: 与えられた集合が空集合かどうかを判別する函数です.

elementp 函数: 第 1 引数で与えられた対象が第二引数で与えられた集合に含まれるかどうかを判定する函数です.

setequalp 函数: は与えられた二つの集合が同等のものであるかどうかを判定する函数です.

disjointp 函数: 与えられた二つの集合に共通部分があるかどうかを判定する函数で, 共通部分がない場合に 'true' を返します.

6.7　配列

6.7.1　Maxima の配列について

Maxima はリストの他に配列が扱えます. Maxima で配列を生成する場合, 幾つかの方法があります. まず, array 函数で配列を宣言して具体的な値を割当てて行く方法と, 配列を宣言せずに直接値を割当てて行く方法, make_array 函数を使う方法があります:

```
(%i1) a1[1,2]:10;
(%o1)                                10
(%i2) a1[0,3]:1;
(%o2)                                 1
(%i3) array(a3,fixnum,5);
(%o3)                                a3
(%i4) make_array(hashed,5);
(%o4)              {Array:  #(NIL NIL $HASHED NIL NIL G13202)}
```

最初に示しているのが配列を無宣言で生成する方法で, この場合は配列に直接式や文字列等を入力することが可能です. また, Maxima の配列は C の配列と同様に 0 から開始します. 配列への値の代入は, この例の様に直接指定することで行います. 他に, fillarray 函数等を用いて他の配列やリストから生成することも可能です.
配列を函数を用いて生成する場合, 目的に応じて三種類の函数が使えます.
array 函数で配列を生成する場合, 引数に配列名を与える必要があります. そして, 最後の make_array 函数は array 函数の様に配列名を与える必要はありません. これらの函数では共に配列の型を指定することができます.
Maxima には生成した配列を調べる函数として listarray 函数と arrayinfo 函数の二つがあります:

配列の情報を表示する函数

listarray(⟨ 配列名 ⟩)
arrayinfo(⟨ 配列名 ⟩)

最初の listarray 函数は配列データを表示し, arrayinfo 函数は配列の型や大きさを返す函数です. では, 先程の例から listarray 函数と arrayinfo 函数で配列の情報を調べてみましょう:

```
(%i6) listarray(a1);
(%o6)                              [1, 10]
(%i7) arrayinfo(a1);
(%o7)                    [hashed, 2, [0, 3], [1, 2]]
(%i8) arrayinfo(a2);
(%o8)                       [declared, 1, [9]]
(%i9) arrayinfo(a3);
(%o9)                      [complete, 1, [5]]
```

まず, 'listarry(a1)' で配列 a1 に '1' と '10' が設定されます. そして, arrayinfo(a1) で返却されるリストの第 1 成分 "hashed" が配列 a1 の型になります. その次に配列の次元があり, 最後にデータが設定されている個所が表示されています.
ここで表示されている配列の型を詳しく述べるために, 配列を生成する函数について詳細を説明しましょう.

配列の生成

array(⟨ 配列名 ⟩,⟨ 整数$_1$⟩,. . ., ⟨ 整数$_n$⟩)
array(⟨ 配列名 ⟩,⟨ 型 ⟩,. . ., ⟨ 整数$_1$⟩,⟨ 整数$_n$⟩)
array([⟨ 配列名$_1$⟩,. . .,⟨ 配列名$_2$⟩], ⟨ 整数$_1$⟩,. . .,⟨ 整数$_n$⟩)
make_array(⟨ 型 ⟩,⟨ 整数$_1$⟩, . . .,⟨ 整数$_n$⟩)
make_array(functional,⟨ 函数 ⟩, ⟨ 型 ⟩,⟨ 整数$_1$⟩, . . .,⟨ 整数$_n$⟩)
arraymake(⟨ 配列名 ⟩, [⟨ 添字$_1$⟩, . . . , ⟨ 添字$_n$⟩])

array 函数は引数に配列名と次元を指定し,⟨ 整数 ⟩ 次の配列を生成します. ここで ⟨ 整数 ⟩ は 5 以下の整数でなければなりません. すなわち, array 函数で生成可能な配列は 5 次以下の配列です. たとえば, 'array(a,2,3,4,5,6)' は問題ありませんが, 'array(b,2,3,4,5,6,7)' はエラーになります.
この array 函数は大域変数 use_fast_arrays の影響を受けます.

大域変数 use_fast_arrays

変数名	初期値	概要
use_fast_arrays	false	配列の種類を制限

まず, 大域変数 use_fast_arrays の値が 'true' の場合, make_array 函数を用いて any 型の配列を生成します. なお, make_array 函数で any 型の配列を生成する場合は LISP の make-array 函数がそのまま使われて, 初期値が 'nil' の LISP の配列を生成します.
つぎに大域変数 user_fast_arrays が既定値ままの 'false' であれば, array 函数は指定した ⟨ 型 ⟩ を反映した配列を構成します. ここで設定可能な型には, 数値配列の場合は「浮動小数点数」と「整数」, その他に「複素数」があります.

array 函数で指定可能な型

型	引数	概要
flonum	[flonum,float]	浮動小数点データ.array-mode 属性に float を設定
fixnum	[fixnum,integer]	整数データ.array-mode 属性に fixnum を設定
function	function	函数型
complete	complete	その他

⟨ 型 ⟩ で flonum 型が指定された場合, 初期値として '0.0' が設定され, 配列の array-mode 属性に float が設定されます.
⟨ 型 ⟩ で fixnum 型が指定された場合, 初期値として '0' が設定され, 配列の array-mode 属性に fixnum が設定されます.
⟨ 型 ⟩ に function と complete が指定された場合と ⟨ 型 ⟩ が無指定の場合, Maxima 内部では 'nil' が設定されます. この 'nil' を listarray 函数は '####' で表示します.

```
(%i1) array(a1,flonum,5);
(%o1)                                a1
(%i2) listarray(a1);
```

```
(%o2)                    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
(%i3) array(a2,fixnum,3);
(%o3)                                    a2
(%i4) listarray(a2);
(%o4)                                [0, 0, 0, 0]
(%i5) array(a3,3);
(%o5)                                    a3
(%i6) listarray(a3);
(%o6)                    [#####, #####, #####, #####]
```

この例では最初に配列 a1 を flonum 型のデータ配列として生成し, listarray 函数で中身を見ています. 配列の成分が '0.0' で初期化されていますね. 以降, fixnum 型と型の指定を行わずに生成し, listarray 函数を用いて中身を見ています.
make_array 函数は array 函数よりも複雑な配列が生成可能です. この make_array 函数で指定可能な〈型〉には any 型, flonum 型, fixnum 型, hashed 型と functional 型があります. なお, functional 型の場合は第 1 引数のみに設定します.

```
(%i45) make_array('any,5);
(%o45)                   {Array:  #(NIL NIL NIL NIL NIL)}
(%i46) make_array('fixnum,5);
(%o46)                      {Array:  #(0 0 0 0 0)}
(%i47) make_array('flonum,5);
(%o47)                   {Array:  #(0.0 0.0 0.0 0.0 0.0)}
(%i48) make_array('hashed,5);
(%o48)             {Array:  #(NIL NIL HASHED NIL NIL G13873)}
(%i49) make_array(functional,'sin,flonum,3);
(%o49) {Array:  #(NIL NIL $FUNCTIONAL NOTEXIST %SIN #(0.0 0.0 0.0))}
```

flonum 型と fixnum 型を指定した場合, make-array 函数を用いた配列のみを生成します. そのために flonum 型の場合は初期値が '0.0', fixnum 型の場合は初期値が '0' の配列となります. なお, 内部的には LISP の通常の配列となります.
flonum 型と fixnum 型以外の型を指定した場合, 内部的に構造体 mgenarray 型のデータが生成されます. このデータ型の場合, 配列データ本体は mgenarray の content に設定されます.
まず, any 型の場合, LISP の make-array 函数を用いて初期値 'nil' の配列が生成されます. なお, listarray 函数で表示を行うと '####' で初期値が表示されます.
hashed 型と functional 型に関しては content に配列本体が設定される仕様の筈ですが, 通常の配列成分の割当を行うと配列データが壊れるために, これらの型は事実上使えません.
array 函数と make_array 函数で生成した配列は大域変数 arrays に登録されます. make_array 函数で,hashed 型と functional 型を指定した場合, gensym 函数で生成したシンボルがこのリストに登録されます.

配列が登録されるリスト

変数名	初期値	概要
arrays	[]	生成した配列名が登録されるリスト

大域変数 arrays に登録される配列は array 函数で生成した全ての配列と, make_array 函数で型が hashed で生成した配列です. なお, この場合は LISP の gensym 函数で生成された表徴が表示され

ます:

```
(%i1) array(a1,fixnum,5)$
(%i2) array(a2,flonum,5)$
(%i3) array(a3,5)$
(%i4) make_array(hashed,5)$
(%i5) arrays;
(%o5)                          [a1, a2, a3, g13158]
```

ただし, listarray 函数や arrayinfo 函数でこの表徴は使えません. そのために array 函数で生成した配列名のみが登録されたリストと考えた方が実用上問題がありません.
arraymake 函数は funmake 函数と同様に実体の無い形式的な配列を生成する函数です.

6.7.2 配列操作に関連する函数

リストや配列の値を配列に割当てる函数

fillarray(⟨ 配列 ⟩,⟨ リスト ⟩)
fillarray(⟨ 配列 ⟩,⟨ 配列 ⟩)

第 1 引数の ⟨ 配列 ⟩ に第 2 引数に指定した ⟨ リスト ⟩ か ⟨ 配列 ⟩ の値を入れます. ⟨ 配列 ⟩ が浮動小数点数 (整数) 配列ならば, 第 2 引数は浮動小数点 (整数) のリストか浮動小数点 (整数) の配列のどちらかでなければなりません.
第 1 引数の配列には第 2 引数の内容が先頭から順番に入れられますが, もし, 第 1 引数の配列が第 2 引数よりも大きければ, 第 2 引数の最後部の元で第 1 引数の配列の残りの個所を埋めてしまいます.

配列から指定したデータを取出す函数

arrayapply(⟨ 配列 ⟩,[⟨ 添字$_1$⟩,...,⟨ 添字$_k$⟩])

arrayapply は第 1 引数に ⟨ 配列 ⟩ を取り, そのあとで配列の添字リストを指定します. 返却値は指定した添字に対応する配列の値です.

配列の大きさを変更する函数

rearray(⟨ 配列 ⟩,⟨ 次元$_1$⟩, ...,⟨ 次元$_n$⟩)

rearray 函数で配列の大きさの変更を行います. この場合, 新しい配列に古い配列の元は番号順に代入されます. 新しい配列が古い配列よりも大きなものであれば, 残りは '0' か '0.0' の何れかで埋められます.

配列を削除する函数

remarray(⟨ 配列$_1$⟩,⟨ 配列$_2$⟩,...)
remarray(all)

remarray 函数は函数を除去し, 占拠されていた保存領域を解放します. 引数が 'all' であれば全ての配列を除去します.

ここで〈式$_i$〉の内部表現の先頭にある演算子 (主演算子と呼びます) は全て同じもので, map 函数を作用させた結果は〈函数〉を作用させた各成分を主演算子で繋げたものとなります.

6.8 行列

6.8.1 行列の内部表現

Maxima の行列は MATLAB の行列とは異なり, 数値行列だけではなく Maxima の式を成分とする行列も扱えます. この Maxima の行列の内部表現は次の書式になります:

Maxima の行列の内部表現

((MATRIX SIMP) リスト$_1$ ⋯ リスト$_n$)

ここで リスト$_1$,..., リスト$_n$ は行列の行に対応する Maxima のリストの内部表現です.
ちなみに n 成分のリストの内部表現は '((MLIST SIMP) 成分$_1$ ... 成分$_n$)' であり, S 式の第 1 成分が MATRTX であれば行列, MLIST であればリストとなっている他に違いはありません. そのため, substpart 等の内部表現を直接操作可能な置換の函数を用いると行列からリストへ, または逆にリストから行列へ変換できるのです:

```
(%i3) a:matrix([1,2,3],[4,5,6]);
                                    [ 1  2  3 ]
(%o3)                               [         ]
                                    [ 4  5  6 ]
(%i4) :lisp $a;
((MATRIX SIMP) ((MLIST SIMP) 1 2 3) ((MLIST SIMP) 4 5 6))
(%i4) ?car(a);
(%o4)                            (matrix, simp)
(%i5) b:substpart("[",a,0);
(%o5)                          [[1, 2, 3], [4, 5, 6]]
(%i6) :lisp $b
((MLIST SIMP) ((MLIST SIMP) 1 2 3) ((MLIST SIMP) 4 5 6))

(%i6) c:substpart(matrix,b,0);
                                    [ 1  2  3 ]
(%o6)                               [         ]
                                    [ 4  5  6 ]
(%i7) ?car(c);
(%o7)                            (matrix, simp)
```

この例では最初に行列 a を matrix 函数を用いて定義し, その内部表現を演算子 ":lisp" を用いて表示させています. それから, 演算子 "?" を用いて内部表現の先頭を取り出しています. 次に substpart 函数で内部表現の先頭を行列からリストに変換しますが, substpart 函数には表徴 "MLISP" ではなく記号 "[" で置換えます. これによって行列はリストに変換されます. なお, リストから行列への変換は記号 "[" を表徴 "matrix" で置換することでできます.

行列表示に関連する大域変数

仮想端末上等の非 GUI 環境, あるいは xMaxima のような数式のレンダリングを行わない環境で Maxima は ASCII 文字等の記号を用いた数式表示を行います. このときに行列の括弧を構成する記号の指定ができます:

行列の括弧を設定する大域変数

変数名	既定値	概要
lmxchar	[	行列の右側の括弧で用いる文字を設定
rmxchar	]	行列の左側の括弧で用いる文字を設定

大域変数 lmxchar:　行列の (左) の括弧として表示する文字を設定します. 右側は大域変数 rmxchar で指定します.

大域変数 rmxchar:　行列の (右) の括弧として表示する文字を設定します. 左側は大域変数 lmxchar で指定します.
ここで行列の左側の括弧を記号 “(”, 右側の括弧を記号 “—” で置換える例を示しておきましょう:

```
(%i15) a:matrix([1,2,3],[4,5,6]);
                                    [ 1  2  3 ]
(%o15)                              [         ]
                                    [ 4  5  6 ]
(%i16) lmxchar:"("$rmxchar:"|"$
(%i18) a;
                                    ( 1  2  3 |
(%o18)                              (         |
                                    ( 4  5  6 |
```

6.8.2　行列を生成する函数

一般の行列の生成を行う函数

ここでは最初に一般的な行列の生成を行う函数を挙げておきましょう:

一般的な行列の定義を行う函数

matrix(⟨ リスト$_1$⟩ ,...,⟨ リスト$_n$⟩)
entermatrix(⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩)

matrix 函数:　引数の ⟨ リスト$_i$⟩ が行列の i 行に対応します. ここで Maxima のリストは ‘[1, 2, 3]’ のようにコンマで区切った式の列を大括弧で括ったものです:

```
(%i1) a:matrix([1,2,3],[4,3,2],[1,0,0]);
                                    [ 1  2  3 ]
                                    [         ]
(%o1)                               [ 4  3  2 ]
                                    [         ]
                                    [ 1  0  0 ]
```

この例に示すように行列の表示は数式の行列のように文字を使って ASCII-ART 風に表示を行います. この文字による行列の表示では行列の括弧として大括弧 “[]” を用います. この括弧は大域変数

lmxchar と大域変数 rmxchar で指定できますが, TeXmacs や wxMaxima のような GUI を用いていれば, これらの大域変数を意識することは殆どないでしょう.

entermatrix 函数: Maxima の要求に沿って ⟨ 整数$_1$⟩ × ⟨ 整数$_2$⟩ 個の成分を入力することで ⟨ 整数$_1$⟩ 行, ×⟨ 整数$_2$⟩ 列の行列が生成できる函数です:

```
(%i1) entermatrix(3,3);
is the matrix  1. diagonal  2. symmetric  3. antisymmetric
 4. general

answer 1, 2, 3 or 4
1;
row 1 column 1:  a;
row 2 column 2:  b;
row 3 column 3:  c;
matrix entered.
                                  [ a  0  0 ]
                                  [         ]
(%o1)                             [ 0  b  0 ]
                                  [         ]
                                  [ 0  0  c ]
```

この例に示すように entermatrix 函数に行列の大きさを引数で指定したあとは対話的に行列を構成できます.

特殊な行列の生成を行う函数

matrix 函数と entermatrix 函数は一般的な行列を生成する函数です. Maxima は行列の型に応じて専用の行列生成函数を持っており, 効率的に行列を生成することが可能です.
このような函数を纏めておきましょう:

特殊な行列の生成を行う函数

ident(⟨ 整数 ⟩)
zeromatrix(⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩)
ematrix(⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩ ,⟨x⟩ , ⟨ 整数$_3$⟩ ,⟨ 整数$_4$⟩)
diagmatrix(⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩)

ident 函数: ⟨ 整数 ⟩ 次の単位正方行列, すなわち, 対角成分が全て 1 で他が全て 0 となる正方行列を生成する函数です:

```
(%i6) ident(3);
                                  [ 1  0  0 ]
                                  [         ]
(%o6)                             [ 0  1  0 ]
                                  [         ]
                                  [ 0  0  1 ]
```

zeromatrix 函数: 〈整数$_1$〉行〈整数$_2$〉列の零行列, すなわち, 全ての成分が零の行列を生成する函数です:

```
(%i7) zeromatrix(3,2);
                                  [ 0  0 ]
                                  [      ]
(%o7)                             [ 0  0 ]
                                  [      ]
                                  [ 0  0 ]
```

ematrix 函数: 〈整数$_1$〉行〈整数$_2$〉列の行列で, (〈整数$_3$〉,〈整数$_4$〉) 成分のみを 〈x〉 とし, 他が全て零となる行列を生成します:

```
(%i9) ematrix(4,3,1,1,1);
                                  [ 1  0  0 ]
                                  [         ]
                                  [ 0  0  0 ]
(%o9)                             [         ]
                                  [ 0  0  0 ]
                                  [         ]
                                  [ 0  0  0 ]
```

diagmatrix 函数: n 行 n 列の正方行列で, その対角成分が x で他が全て零となる対角行列を返します:

```
(%i19) diagmatrix(3,2);
                                  [ 2  0  0 ]
                                  [         ]
(%o19)                            [ 0  2  0 ]
                                  [         ]
                                  [ 0  0  2 ]
```

なお, 'diagmatrix(n,1)' は 'ident(n)' と同じ n 次元の単位行列を生成します.

連立一次方程式から行列を生成

Maxima では連立一次方程式から, その係数行列を生成できます:

連立方程式から行列の生成を行う函数
coefmatrix([〈方程式$_1$〉,...], [〈変数$_1$〉,...])
augcoefmatrix([〈方程式$_1$〉,...], [〈変数$_1$〉,...])

coeffmatrix 函数: 連立一次方程式の変数リストに対応する係数行列を返します. なお, 連立一次方程式は演算子 "=" の右辺, あるいは左辺の何れかが 0 の場合には式だけで構いません:

```
(%i22) coefmatrix([2*x-3*y-1=0,3*x+3*y+10=0],[x,y]);
                                  [ 2   - 3 ]
(%o22)                            [         ]
                                  [ 3    3  ]

(%i23) coefmatrix([2*x-3*y-z,3*x+3*y+10*z],[x,y]);
                                  [ 2   - 3 ]
(%o23)                            [         ]
                                  [ 3    3  ]
```

augcoefmatrix 函数: 与えられた方程式と指定した変数のリストから係数行列を生成します. 行列は ⟨ 方程式$_1$⟩,... から構成される線形方程式系に含まれる ⟨ 変数$_1$⟩ , ... の係数から構築されます. coefmatrix 函数との違いは生成する係数行列に各方程式の定数項が列として付加される点です:

```
(%i25) augcoefmatrix([2*x-3*y-z,3*x+3*y+10*z],[x,y]);
                                 [ 2   - 3   - z   ]
(%o25)                           [                 ]
                                 [ 3    3    10 z ]
(%i26) augcoefmatrix([2*x-3*y=1,3*x+3*y=10],[x,y]);
                                 [ 2   - 3   - 1   ]
(%o26)                           [                 ]
                                 [ 3    3    - 10 ]
```

配列から行列を生成する函数

配列から行列を生成する函数
genmatrix(⟨ 記号 ⟩, ⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩ , ⟨ 整数$_3$⟩ ,⟨ 整数$_4$⟩)
genmatrix(⟨ 記号 ⟩, ⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩ , ⟨ 整数$_3$⟩)
genmatrix(⟨ 記号 ⟩, ⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩)

genmatrix 函数: 第 1 引数の ⟨ 記号 ⟩ で指定された二次元配列, および, 2 変数函数から行列を生成します. 内部では ⟨ 整数$_1$⟩ と ⟨ 整数$_2$⟩ で行列の大きさを決定します. 次に, 指定した記号が配列であるか, 実体を持たない単なる記号であるかに応じて, lambda 函数を用いて作用 $\lambda(i, j)$ を定めます. ここで, 作用 $\lambda(i, j)$ の第 1 変数 i と第 2 変数 j の開始は ⟨ 整数$_3$⟩ と ⟨ 整数$_4$⟩ で指定されますが, 条件として ⟨ 整数$_1$⟩ ≥ ⟨ 整数$_3$⟩ と ⟨ 整数$_2$⟩ ≥ ⟨ 整数$_4$⟩ を満していなければなりません. 開始位置が無指定の場合には自動的に '1' から開始します.

たとえば, f を配列を定める 2 変数の函数として ‘genmatrix(f,4,3,2,1)’ とした場合を示します:

```
(%i1)  genmatrix (f, 4, 3, 2, 1);
                                [ f       f       f      ]
                                [  2, 1    2, 2    2, 3  ]
                                [                        ]
(%o1)                           [ f       f       f      ]
                                [  3, 1    3, 2    3, 3  ]
                                [                        ]
                                [ f       f       f      ]
                                [  4, 1    4, 2    4, 3  ]
```

この例で示すように行は第 3 引数で指定した整数から開始し, 列も第 4 引数で指定した整数から開始します.

行列から小行列を生成する函数

小行列を生成する函数

minor(⟨ 正方行列 ⟩ ,⟨i⟩ ,⟨j⟩)
submatrix(⟨ 行$_1$⟩, . . . ,⟨ 行$_m$⟩,⟨ 行列 ⟩,⟨ 列$_1$⟩, . . . ,⟨ 列$_n$⟩)
submatrix(⟨ 行$_1$⟩, . . . ,⟨ 行$_m$⟩,⟨ 行列 ⟩)
submatrix(⟨ 行列 ⟩,⟨ 列$_1$⟩, . . . ,⟨ 列$_n$⟩)

minor 函数:　与えられた ⟨ 正方行列 ⟩ の ⟨i⟩, ⟨j⟩ 成分の小行列, つまり, ⟨ 正方行列 ⟩ から ⟨i⟩ 行と ⟨j⟩ 列を抜いた行列を返します.

submatrix 函数:　⟨ 行$_i$⟩ 行と ⟨ 列$_j$⟩ 列が削除された新しい行列を生成します. submatrix 函数は minor 函数よりも多くの行と列が削除出来ます.

行列の複製やリストに変換する函数

行列の複製やリスト変換を行う函数

copymatrix(⟨ 行列 ⟩)
list_matrix_entries(⟨ 行列 ⟩)

copymatrix 函数:　⟨ 行列 ⟩ の複製を行います. この函数は ⟨ 行列 ⟩ を成分毎に再生成する時だけに使います. なお, copymatrix 函数の実体は copy 函数を matrixp 函数を用いて行列に制限した函数です.

list_matrix_entries 函数: 与えられた行列に対して各行を繋げた平リストを返す函数です. 内部的には matrix を mlist で置換して型の変換を実行し, 各行をリストとして順番に追加して行くことで平リストを生成しています:

```
(%i10) A1:matrix([1,2,3],[4,5,6]);
                                    [ 1  2  3 ]
(%o10)                              [         ]
                                    [ 4  5  6 ]
(%i11) list_matrix_entries(A1);
(%o11)                        [1, 2, 3, 4, 5, 6]
```

6.8.3 行列の操作函数

6.8.4 行, 列, 及び成分の操作函数

Maxima の行列の行, 列や成分の操作は, MATLAB の影響を受けた言語, すなわち, MATLAB 風の言語とは異なり, 基本的に函数を通して行います.

行や列の操作函数

col(⟨ 行列 ⟩,⟨ 整数 ⟩)
row(⟨ 行列 ⟩,⟨ 整数 ⟩)
addcol(⟨ 行列 ⟩,⟨ リスト$_1$⟩,⟨ リスト$_2$⟩,...,⟨ リスト$_n$⟩)
addrow(⟨ 行列 ⟩,⟨ リスト$_1$⟩,⟨ リスト$_2$⟩,...,⟨ リスト$_n$⟩)
setelmx(⟨x⟩,⟨i⟩,⟨j⟩,⟨ 行列 ⟩)

col 函数: ⟨ 行列 ⟩ に対し,⟨ 整数 ⟩ 番目の列を行列の形式で返します:

```
(%i1) mat1:matrix([1,2,3],[4,3,2]);
                                    [ 1  2  3 ]
(%o1)                               [         ]
                                    [ 4  3  2 ]
(%i2) col(mat1,2);
                                       [ 2 ]
(%o2)                                  [   ]
                                       [ 3 ]
```

row 函数: ⟨ 行列 ⟩ の ⟨ 整数 ⟩ 番目の行を 1 行の行列として返します:

```
(%i1) mat1:matrix([1,2,3],[4,3,2]);
                                    [ 1  2  3 ]
(%o1)                               [         ]
                                    [ 4  3  2 ]
(%i2) row(mat1,1);
(%o2)                               [ 1  2  3 ]
```

addcol 函数: 〈 行列 〉に対して〈 リスト$_1$〉,...,〈 リスト$_n$〉を列として追加します. このときに各リストの長さと行列の行数は一致していなければなりません:

```
(%i7) mat1;
                                  [ 1  2  3 ]
(%o7)                             [         ]
                                  [ 4  3  2 ]
(%i8) addcol(mat1,[a,b],[x^2,y^3]);
                                  [           2 ]
                                  [ 1  2  3  a  x ]
(%o8)                             [               ]
                                  [             3 ]
                                  [ 4  3  2  b  y ]
```

addrow 函数: 〈 行列 〉に対して〈 リスト$_1$〉,...,〈 リスト$_n$〉を行として追加します. このとき, 各リストの長さと行列の行数は一致していなければなりません:

```
(%i7) mat1;
                                  [ 1  2  3 ]
(%o7)                             [         ]
                                  [ 4  3  2 ]
(%i8) addrow(mat1,[a,b,c],[x^2,y^3,z^4]);
                                  [ 1   2   3  ]
                                  [            ]
                                  [ 4   3   2  ]
(%o8)                             [            ]
                                  [ a   b   c  ]
                                  [            ]
                                  [ 2   3   4  ]
                                  [ x   y   z  ]
```

setelmx 函数: 〈 行列 〉の (〈i〉,〈j〉) 成分を 〈x〉 で置換する函数です:

```
(%i13) A;
                                  [ 4  0  0  0 ]
                                  [            ]
                                  [ 0  4  0  0 ]
(%o13)                            [            ]
                                  [ 0  0  4  0 ]
                                  [            ]
                                  [ 0  0  0  4 ]
(%i14) setelmx(4,2,3,A);
                                  [ 4  0  0  0 ]
                                  [            ]
                                  [ 0  4  4  0 ]
(%o14)                            [            ]
                                  [ 0  0  4  0 ]
                                  [            ]
                                  [ 0  0  0  4 ]
```

直接, A[i,j]:x でも行列 A の (i,j) 成分を x で置換できます.

6.8.5 転置, 上三角, 共役行列を計算する函数

転置, 上三角, 共役行列を計算する函数
transpose(⟨ 行列 ⟩)
triangularize(⟨ 行列 ⟩)
echelon(⟨ 行列 ⟩)

transpose 函数: ⟨ 行列 ⟩ の転置行列を生成します:

```
(%i9) A:matrix([1,2,3],[4,3,1]);
                                  [ 1  2  3 ]
(%o9)                             [         ]
                                  [ 4  3  1 ]
(%i10) transpose(A);
                                   [ 1  4 ]
                                   [      ]
(%o10)                             [ 2  3 ]
                                   [      ]
                                   [ 3  1 ]
```

triangularize 函数: ⟨ 行列 ⟩ の上三角行列形式を生成しますが, ここで行列が正方行列である必要はありません:

```
(%i6) A:matrix([1,2,3,4],[3,4,5,1],[2,3,1,5]);
                                  [ 1  2  3  4 ]
                                  [            ]
(%o6)                             [ 3  4  5  1 ]
                                  [            ]
                                  [ 2  3  1  5 ]
(%i7) triangularize(A);
                              [ 1   2    3    4    ]
                              [                    ]
(%o7)                         [ 0  - 2  - 4  - 11 ]
                              [                    ]
                              [ 0   0    6   - 5   ]
```

echelon 函数: ⟨ 行列 ⟩ の echelon 形式を生成します. これは初等的な行操作で各々の行の最初の非零元を 1, その元を含む列に対しては, その元を含む行よりも下の成分を全て零となる上三角行列を生成します:

```
                              [2   1 - a   -5 b ]
(%o2)                         [                 ]
                              [a     b       c  ]
(%i3) echelon(d2);
```

```
                    [       a - 1          5 b        ]
                    [1    - -----       - ---        ]
                    [         2             2         ]
(%o3)               [                                 ]
                    [                  2 c + 5 a b    ]
                    [0      1          -------------]
                    [                         2       ]
                    [                  2 b + a  - a]
```

行列の成分に函数を作用させる函数

行列の成分に函数を作用させる函数
matrixmap(⟨ 函数名 ⟩,⟨ 行列 ⟩)

matrixmap 函数:　行列 ⟨m⟩ の各成分に ⟨ 函数名 ⟩ で指定した函数を作用させます. リストに対する map 函数の行列版です:

```
(%i11) A:ident(3)*3;
                                    [ 3  0  0 ]
                                    [         ]
(%o11)                              [ 0  3  0 ]
                                    [         ]
                                    [ 0  0  3 ]

(%i12) matrixmap(lambda([x],cos(x)*exp(-x)),A);
                    [   - 3                                  ]
                    [ %e     cos(3)        1              1   ]
                    [                                        ]
(%o12)              [                  - 3                   ]
                    [       1         %e     cos(3)        1   ]
                    [                                        ]
                    [                                 - 3    ]
                    [       1              1         %e     cos(3) ]
```

matrixmap 函数では函数名を第 1 引数として与えるために表記 “f(x)” のように変数を含めた函数名を与えることができません. 既に定義された函数の場合は函数名を指定しますが, そうでない場合は Maxima の lambda 式を用いて無名函数として与えます. この例では行列 A に $\lambda x.e^{-x}\cos x$ を matrixmap 函数を用いて作用させています.

行列の階数と対角和に関連する函数

階数と対角和を計算する函数
rank(⟨ 行列 ⟩)
mattrace(⟨ 行列 ⟩)

rank 函数: 〈行列〉の階数を求めます. 行列の階数は行列から取出した小正方行列の行列式で, 零にならない小行列の最大次数です. なお, rank 函数は行列成分の値が非常に零に近い場合には誤った答を返すことがあります.

mattrace 函数: 〈行列〉が正方行列の場合, 対角和, すなわち行列の主対角成分の総和を計算します. この函数は,ncharpoly 函数で利用されています. この ncharpoly 函数は Maxima の charpoly 函数の代りに使える函数です. なお, mattrace 函数を利用するために予め load("nchrpl"); を実行する必要があります:

```
(%i14) load(nchrpl)$
(%i15) A:matrix([1,2,3],[4,3,1],[-2,0,-2]);
                                  [  1   2   3  ]
                                  [             ]
(%o15)                            [  4   3   1  ]
                                  [             ]
                                  [ -2   0  -2  ]
(%i16) mattrace(A);
(%o16)                                 2
```

6.8.6 行列式に関連する函数

余因子行列に関連する行列

余因子行列の計算に関連する函数

adjoint(〈正方行列〉)
invert(〈正方行列〉)

adjoint 函数: 〈正方行列〉の余因子行列を計算します.

invert 函数: 逆行列を余因子行列を用いた方法で計算します. これは bfloat 値成分や浮動小数点数を係数とする多項式を成分とする行列の逆行列を CRE 形式に変換せずに計算出来ます. determinant 函数は余因子の計算で利用されるので, 大域変数 ratmx が false であれば, その逆行列は成分表現を変更せずに計算されます. 現行の実装は高い次数の行列に対して効率的なものではありません. なお, 大域変数 detout が true の場合は行列式の部分は逆行列の外側に出されたままとなります.
invert 函数が返した結果は展開されていません. 最初から多項式成分を持つ行列の場合, expand(invert(mat)) ,detout; で生成された出力は見栄えが良くなります:

```
(%i28) A:matrix([t,1,t-2],[1,t,0],[t+1,1,t-1]);
                                  [    t     1   t - 2 ]
                                  [                    ]
(%o28)                            [    1     t     0   ]
                                  [                    ]
                                  [ t + 1   1   t - 1  ]
```

```
(%i29) adjoint(A);
        [    (t - 1) t              - 1            - (t - 2) t ]
        [                                                       ]
(%o29)  [      1 - t      (t - 1) t - (t - 2) (t + 1)   t - 2   ]
        [                                                       ]
        [                                             2         ]
        [ 1 - t (t + 1)             1                t  - 1     ]
(%i30) invert(A),expand,detout;
                         [     2                 2 ]
                         [    t  - t    - 1  2 t - t ]
                         [                           ]
                         [    1 - t      2    t - 2  ]
                         [                           ]
                         [    2             2        ]
                         [ - t  - t + 1  1  t  - 1   ]
(%o30)                   -----------------------------
                                    2 t - 1
```

なお, 行列式を行列の中に入れる場合, adjoint(⟨ 行列 ⟩)/determinant(⟨ 行列 ⟩) を計算した結果を expand 函数で展開したり, 有理数係数の多項式が行列の成分に現われる場合は ratsimp 函数を用いると良いでしょう:

```
(%i35) adjoint(A)/determinant(A),expand;
       [        2                                       2     ]
       [       t         t              1        2 t    t     ]
       [    ------- - -------       - -------  ------- - ------- ]
       [    2 t - 1   2 t - 1         2 t - 1  2 t - 1  2 t - 1 ]
       [                                                        ]
       [       1         t              2         t        2    ]
(%o35) [    ------- - -------         -------  ------- - ------- ]
       [    2 t - 1   2 t - 1         2 t - 1  2 t - 1  2 t - 1 ]
       [                                                        ]
       [     2                                  2               ]
       [    t          t         1       1     t          1     ]
       [ - ------- - ------- + -------  -------  ------- - ------- ]
       [   2 t - 1   2 t - 1   2 t - 1  2 t - 1  2 t - 1  2 t - 1 ]
(%i36) adjoint(A)/determinant(A),ratsimp;
                      [    2                      2       ]
                      [   t  - t        1        t  - 2 t ]
                      [  -------    - -------  - -------- ]
                      [  2 t - 1      2 t - 1    2 t - 1  ]
                      [                                   ]
                      [    t - 1        2        t - 2    ]
(%o36)                [ - -------    -------    -------   ]
                      [   2 t - 1    2 t - 1    2 t - 1   ]
                      [                                   ]
                      [    2                     2        ]
                      [   t  + t - 1    1       t  - 1    ]
                      [ - ----------  -------  -------    ]
                      [    2 t - 1    2 t - 1  2 t - 1    ]
```

行列式に関連する函数

行列式に関連する函数
determinant(⟨ 行列 ⟩)
newdet(⟨ 行列 ⟩)
newdet(⟨ 配列 ⟩ ,⟨ 整数 ⟩)
permanent(⟨ 行列 ⟩ ,⟨ 整数 ⟩)

determinant 函数: Gaußの消去法と似た方法で ⟨ 行列 ⟩ の行列式を計算します. 計算結果の書式は大域変数 ratmx の設定に依存します.
疎行列の行列式を計算する特別な方法もあり, ratmx:true と sparse:true に設定した場合に使えます.

newdet 函数: ⟨ 行列 ⟩ や ⟨ 配列 ⟩ の行列式を計算します. この際に Johnson-Gentleman tree minor アルゴリズムを用います. なお, ⟨ 整数 ⟩ を指定した場合, 1 行から ⟨ 整数 ⟩ 行と 1 列から ⟨ 整数 ⟩ 列の正方行列を取出し, その行列式を計算します. この整数値が無指定の場合に ⟨ 行列 ⟩ が正方行列であればそのまま行列式を計算し, ⟨ 行列 ⟩ が正方行列でなければ余分な末尾の行, あるいは列を削除した正方行列の行列式を返します:

```
(%i23) A;
                                [ 1  2  3  4 ]
                                [             ]
(%o23)                          [ 3  4  5  1 ]
                                [             ]
                                [ 2  3  1  5 ]
(%i24) newdet(A,3);
(%o24)/R/                            6
```

permanent 函数: ⟨ 行列 ⟩ の permanent を計算します. なお, ⟨ 整数 ⟩ を指定した場合, 1 行から ⟨ 整数 ⟩ 行と 1 列から ⟨ 整数 ⟩ 列の正方行列を取出し, その行列式を計算します. ここで, permanent は行列式に似ていますが符号の変化のないものです.

特性多項式に関連する函数

特性多項式に関連する函数
charpoly(⟨ 行列 ⟩ ,⟨ 変数 ⟩)
ncharpoly(⟨ 行列 ⟩ ,⟨ 変数 ⟩)

charpoly 函数: ⟨ 行列 ⟩ の特性多項式 det (⟨ 変数 ⟩I – ⟨ 行列 ⟩) を計算します.
determinant (⟨ 行列 ⟩ – diagmatrix (length (⟨ 行列 ⟩) , ⟨ 変数 ⟩)) と同じ結果を返します.

ncharpoly 函数: 〈 変数 〉に対する〈 行列 〉の特性多項式を計算します. これは charpoly 函数とは別物です.
ncharpoly 函数では与えられた行列の冪乗の対角和を計算しますが, 対角和は特性多項式の根の冪乗の総和に等しいものです. これらの諸量から根の対称式の計算が可能ですが, それらは特性多項式の係数です. charpoly 函数は 'varident[n]-a' の行列式を計算しており. この点で ncharpoly 函数は優れています. たとえば, 整数成分の非常に大きな行列の場合は算術的に多項式の計算を避けるためです. 予め, load("nchrpl"); で読込む必要があります.

6.8.7　行列の四則演算

行列の四則演算子

Maxima で行列の四則演算は多項式の四則演算に似た表記で行えます. まず, 行列の和と差は演算子 "+" と演算子 "-" を用いますが, 可換積演算子 "*" と商演算子 "/" は行列の成分同士の積と商になります.
matrix 函数による行列の生成と演算子 "+","-" による結果を示しておきましょう:

```
(%i10) a:matrix([1,0,0],[0,2,0],[0,0,3]);
                                  [ 1  0  0 ]
                                  [         ]
(%o10)                            [ 0  2  0 ]
                                  [         ]
                                  [ 0  0  3 ]
(%i11) b:matrix([1,2,3],[1,3,5],[9,7,5]);
                                  [ 1  2  3 ]
                                  [         ]
(%o11)                            [ 1  3  5 ]
                                  [         ]
                                  [ 9  7  5 ]
(%i12) a+b;
                                  [ 2  2  3 ]
                                  [         ]
(%o12)                            [ 1  5  5 ]
                                  [         ]
                                  [ 9  7  8 ]
(%i13) a-b;
                                 [  0   -2  -3 ]
                                 [             ]
(%o13)                           [ -1   -1  -5 ]
                                 [             ]
                                 [ -9   -7  -2 ]
```

引き続き, 演算子 “*”,“/” による結果も見ておきましょう:

```
(%i14) b*b;
                                [ 1    4    9  ]
                                [              ]
(%o14)                          [ 1    9    25 ]
                                [              ]
                                [ 81   49   25 ]
(%i15) b/b;
                                  [ 1  1  1 ]
                                  [         ]
(%o15)                            [ 1  1  1 ]
                                  [         ]
                                  [ 1  1  1 ]
```

このように演算子 “+” と演算子 “-” は通常の行列の和と差になりますが, 演算子 “*” は勝手が違います. 何故なら, この積演算子 “*” は可換性を属性として持っているために可換性を持たない行列の積が表現できないからです.
そこで, 通常の行列の積演算子は非可換積の演算子 “.” を用います. この演算子 “.” は見落し易いので, ここでは, 非可換積, あるいは非可換積と表記します. この非可換積による演算は行列 A と B に対して, A . B と記述します. この非可換積とその他の演算子に関しては大域変数で, その分配律や結合律が制御出来ます. この非可換積には属性として分配律と結合律が与えられています. 次に, 非可換積に対応する行列の冪乗は演算子 “^^” を用います. たとえば, 行列 A の逆行列が存在する場合, ‘A^^(-1)’ で行列 A の逆行列を表現します. なお, 非可換積の詳細に関しては §5.3.6 を参照して下さい.

6.8.8 行列演算に関連する大域変数

行列演算を制御する大域変数として通常の演算を含めて制御する大域変数の他に, 行列演算のみに影響を与える大域変数があります. ここではこれらの大域変数について解説します.

大域変数 do 一族

行列演算を遂行させる大域変数で先頭が “do” で開始する大域変数があります. ここではこれらの “do” で開始する大域変数を安易に “do 一族” と呼び, それらの特徴について述べましょう:

大域変数 do 一族

変数名	初期値	概要
doallmxops	true	行列演算子の評価に関連
domxexpt	true	行列に対する指数函数の処理
domxmxops	true	対行列, 対リスト間の演算を管理
domxnctimes	false	非可換積の実行に関連
doscmxops	false	スカラと行列間の演算を実行
doscmxplus	false	スカラ+行列の処理に関連

大域変数 doallmxops:　'true' であれば全ての行列演算子が評価されます. ここで 'false' であれば非可換積の演算子 "." を支配する個々の大域変数 dot 一族の設定 (§5.3.6 参照) が優先されます.

大域変数 domxexpt:　'true' の場合, '%e^ matrix([1,2],[3,4])' は 'matrix([%e,%e^ 2],[%e^3,%e^4])' となります. 一般的に, この変換は ⟨ 基底 ⟩⟨ 次数 ⟩ の形式の変換に影響します. なお, ⟨ 基底 ⟩ はスカラか定数の式であり, ⟨ 次数 ⟩ はリストか行列です. この大域変数が 'false' であれば, この変換は実行されません.

大域変数 domxmxops:　'true' であれば行列と行列間の演算子や行列とリストの間の演算子が実行されます. この大域変数が 'false' なら, これらの演算は実行されません. なお, この大域変数はスカラーと行列との間の演算には影響を与えません.

大域変数 domxnctimes:　'false' であれば行列の非可換積が実行されます.

大域変数 doscmxops:　'true' であればスカラと行列間の演算子が実行されます.

大域変数 doscmxplus:　'true' であれば, スカラと行列の和が行列値となります. この大域変数は大域変数 doallmxops から独立した変数です.

大域変数 matrix_element 一族

行列の和や可換積, 及び転置行列の処理で用いられる演算子や函数を指定する大域変数を纏めておきます:

大域変数 matrix_element 一族

変数名	既定値	概要
matrix_element_add	+	行列の和の演算子を指定
matrix_element_mult	*	行列の成分間の積の演算子を指定
matrix_element_transpose	false	転置の際に作用させる函数を設定

大域変数 matrix_element_add:　行列同士の和を計算する際に用いる演算子を設定します. 函数名や lambda 式であっても構いません.

大域変数 matrix_element_mult:　行列の成分同士の積を計算する際に用いる演算子を設定します. 函数名や lambda 式であっても構いません.

大域変数 matrix_element_transpose:　転置行列を計算する際に作用させる函数や lambda 式をを指定します.

行列演算で特定の処理を指定する大域変数

行列演算で特定の処理を指定する大域変数

変数名	既定値	概要
assumescalar	true	引数がスカラー値であると仮定.
mx0simp	true	0 との積を 0 で返すかどうかを制御
scalarmatrixp	true	1 行 1 列行列のスカラへの自動変換を制御
sparse	false	疎行列を計算するかどうかを指定
detout	false	余因子行列を用いた逆行列計算で行列式の処理を制御
ratmx	false	行列成分の CRE 表現の表示を制御

大域変数 assumescalar: 'true' であれば, 自由変数と行列の可換積は, 行列の各成分との可換積で自動的に置換されます. 'false' の場合, 変数は各成分に分配されません.

大域変数 mx0simp: 'true' であれば, 行列と '0' との積は成分が全て '0' の零行列として返却されますが, 'false' であれば, 行列と '0' の積は '0' となります.

大域変数 scalarmatrixp: 'true' であれば二つの行列の非可換積の計算で得られた 1 行 1 列の行列はスカラーに変換されます. もし, 'all' に設定されていれば, この変換で 1 行 1 列の行列は何時でもスカラに変換されます. ただし, 'false' であればこの変換は実行されません.

大域変数 sparse: この大域変数の値が 'true' で大域変数 ratmx の値も 'true' であれば, determinant 函数は疎行列式を計算するためのルーチンを利用します.

大域変数 detout: 'true' であれば逆行列を計算したときに行列式の割算がそのまま行列の外に残されます. この大域変数が効力を持つためには大域変数 doallmxops と大域変数 doscmxops の値がともに 'false' でなければなりません. この設定をその他の二つが設定される ev 函数で与えることも出来ます.

大域変数 ratmx: 'false' であれば, 行列式や行列の和, 差, 積が行列の表示形式で行われ, 逆行列の結果も一般の表示となります. 'true' であればこれらの演算は CRE 表現で実行され, 逆行列の結果も CRE 表現となります. これは成分が往々にして望みもしない展開 (大域変数 ratfac の設定に依存するものの) の原因になります.

6.8.9 eigen パッケージ

Maxima に標準で附属する eigen パッケージには固有値計算に関連する函数と基底の直交化に関連する函数が含まれれています.

ここで説明する函数を利用するために load(eigen); を予め実行します. 'load(eigen)' を実行するとパッケージに含まれる函数が Maxima に読込まれ, 次の大域変数が定義されます.

eigen パッケージで定義される大域変数

変数名	既定値	概要
hermitianmatrix	false	Hermit 行列かどうかを指定
nondiagonalizable	false	非対角化行列かどうかを指定
knowneigvals	false	固有値を既知として扱うかどうかを指定
knowneigvects	false	固有ベクトルを既知として扱うかどうかを指定
listeigvects	[]	固有ベクトルのリスト
listeigvals	[]	固有値のリスト
rightmatrix	[]	行列の対角化で左側に置かれる行列
leftmatrix	[]	行列の対角化で右側に置かれる行列

大域変数 hermitianmatrix: 'true' の場合に与えられた行列が Hermite 行列であると仮定します. ここで大域変数 leftmatrix に割当てられた行列は大域変数 rightmatrix に割当てられた転置行列と複素共役になります. すなわち, 大域変数 rightmatrix に割当てられた行列は ⟨ 行列 ⟩ の正規化した固有ベクトルを列とする行列になります.

大域変数 nondiagonalizable: 'false' であれば大域変数 leftmatrix と大域変数 rightmatrix に行列が割当てられます. そして, `leftmatrix` . ⟨ 行列 ⟩ . `rightmatrix` の対角成分に ⟨ 行列 ⟩ の固有値が現れる対角行列になります.
このように leftmatrix, rightmatrix の意味は単純に ⟨ 行列 ⟩ の左側と右側にそれぞれ配置されたときの積の処理結果が対角行列となるという意味です. ただし, 大域変数 nondiagonalizable の値が 'true' であれば, これらの行列は生成されません.

大域変数 knowneigvals: 'true' であれば行列の固有値は既知で, 大域変数 listeigvals に保存されていると eigen パッケージの函数は仮定します. したがって, 大域変数 listeigvals に割当てられている値は eigenvalues 函数の出力と同じリストになります.

大域変数 knowneigvlects: 'true' の場合に行列の固有値に対応する固有ベクトルが既知であり, 大域変数 listeigvects に保存されていると eigen パッケージの函数が仮定します. そのために大域変数 knowneigvects の値が 'true' の場合, 大域変数 listeigvects には eigenvects 函数の出力と同じリストが設定されていなければなりません.

内積函数

innerproduct($\langle x\rangle$,$\langle y\rangle$)
inprod($\langle x\rangle$,$\langle y\rangle$)(innerproduct 函数の別名)

innerproduct 函数: 内積を表現する函数で, その短縮名は inprod 函数になります. リスト $\langle x \rangle$ と $\langle y \rangle$ を引数として取り, $\langle x \rangle$ の複素共役 . $\langle y \rangle$ で定義されています. ここで非可換積は通常のベクトルの内積演算子と同じものです.
eigen パッケージには以下の行列操作の函数が含まれています.

行列操作の函数

columnvector(〈 リスト 〉)
conjugate(〈 リスト 〉)
conj(〈 リスト 〉)(conjugate 函数の別名)

columnvector 函数: 与えられたリストから列ベクトルを生成します:

```
(%i6) columnvector([1,2,3]);
                                  [ 1 ]
                                  [   ]
(%o6)                             [ 2 ]
                                  [   ]
                                  [ 3 ]
```

conjugate 函数: 引数の複素共役を返します:

```
(%i3) A:matrix([1,2*%i],[1-%i,4]);
                                [    1      2 %i  ]
(%o3)                           [                  ]
                                [ 1 - %i     4     ]
(%i4) conjugate(A);
                               [    1      - 2 %i  ]
(%o4)                          [                   ]
                               [ %i + 1      4      ]
```

固有値の計算に関連する eigen パッケージ函数

eigenvalues(〈 行列 〉)
eivals(〈 行列 〉)(eigenvalues 函数の別名)
eigenvectors(〈 行列 〉)
eivects(〈 行列 〉)(eigenvectors 函数の別名)
similaritytransform (〈 行列 〉)
simtran (〈 行列 〉)(similaritytransform 函数の別名)

eigenvalues 函数 (短縮名 eivals): 引数に一つの行列を取り, 固有値と固有ベクトルを含むリストを返します. 返却されるリストの第 1 成分が固有値リスト, 第 2 成分が固有値リストに対応する重複度のリストとなります.
eigenvalues 函数では charpoly 函数で特性多項式を計算し, solve 函数を使って, その特性多項式の根を求めています. solve 函数は厳密解を求める函数なので高次多項式に対しては, その根を見付け

損なうことがあります. さらに不正確な答を返すこともありますが, eigen パッケージに含まれている conjugate 函数, innerproduct 函数, univector 函数, columnvector 函数と gramschmidt 函数を必要としない函数です.

eigenvectors 函数:　引数として一つの行列を取ってリストを返します. このリストに含まれる最初の副リストには eigenvalues 函数の出力, 他の副リストには行列の各々の固有値に対応する固有ベクトルが含まれています.
なお, algsys 函数が固有ベクトルの計算で使われており, 固有値が不正確な場合, この algsys が解を生成することが出来ないこともあります. この場合, eigenvalues 函数を使って最初に見つけた固有値の簡易化を行うことを勧めます.

similarityransform 函数:　〈行列〉を引数とし,uniteigenvectors 函数の出力結果リストを返します.

ベクトルの正規化に関連する函数

gramschmidt([〈リスト$_1$〉,...,〈リスト$_n$〉])
gschmidt([〈リスト$_1$〉,...,〈リスト$_n$〉]) (gramschmidt 函数の別名)
unitvector(〈リスト〉)
uvect(〈リスト〉)(unitvector 函数の別名)
uniteigenvectors(〈行列〉)
ueivects(〈行列〉)(uniteigenvectors 函数の別名)

gramschmidt 函数:　Gram-Schmidt 法によって直交ベクトルを求めます. この函数は eigen パッケージに含まれる函数なので, 予め load(eigen) を実行して利用します. gramschmidt 函数は引数にリストの列で構成されるリストを取ります. ここで〈リスト$_i$〉は全て長さが等しくなければなりませんが, 各リストが直交している必要はありません.
この gramschmidt 函数は互いに直交したリストで構成されたリストを返します. なお, 返ってきた結果には因子分解された整数が含まれることがあります. これは Maxima の factor 函数が gramschmidt の処理の過程で使われたためで, こうすることで式が複雑なものになることを回避して生成される変数の大きさを減らす助けにもなっています.

unitvector 函数:　〈リスト〉の大きさを '1' にしたリストを返します.

uniteigenvalues 函数:　〈行列〉の固有値と固有ベクトルで構成されたリストを返します. 出力リストの第 1 成分のリストには eigenvalues 函数の出力があり, 第 2 成分の副リストには正規化した固有ベクトル, 第 1 成分のリストの固有値に対応する順番で並んでいます.

uniteigenvectors 函数:　与えられた行列から長さを '1' にした固有ベクトルを返します.

6.9 LAPACK パッケージの利用

6.9.1 BLAS と LAPACK について

LAPACK(Linear Algebra PACKage) は数値行列処理を目的とするライブラリで, 線形方程式や固有値問題を扱うことができます. この LAPACK は BLAS(Basic Linear Algebra Subprograms) と呼ばれるベクトルと行列の基本的な計算を効率良く処理することを目的としたルーチン群上で構築されています. この BLAS の公式標準実装は netlib [4]にて公開されていますが, BLAS の性能が LAPACK の性能に直接関係するために最適化を行った BLAS が幾つか存在します. まず, フリーのものでは ATLAS (Automatically Tuned ALgebra Software) と GotoBLAS2(後藤 BLAS) [5]で, 商用のものには Intel の MKL(Math Kernel Library) や AMD の ACML(AMD Core Math Library) です. これらの最適化 BLAS については, 一般的な利用者は MKL, ATLAS や GotoBLAS で十分, ヘビーユーザなら MKL か GotoBLAS, 新規アルゴリズム開発等の学術向け用途には GotoBLAS が良いようです [22]. なお, BLAS は数値行列処理を高速に行うことを目的にしたものですが, 実際に効果が出るのはサイズの大きな行列といった計算量を必要とする行列に効果があり, 逆に計算量が極端に小さな行列では遅くなる傾向があります. そのために BLAS をブラックボックスとみなして使うべきではなく, どのような原理で動作しているかを理解しておく必要があります [22].

Maxima の LAPACK パッケージは通常の LAPACK, あるいは最適化が行われた ATLAS や GotoBLAS を利用するものではなく, netlib で公開されている標準 BLAS を Common Lisp に f2cl(FORTRAN から Common Lisp への変換ツール) を使って単純に移植したものです. そのために ATLAS や GotoBLAS のような最適化が行なわれたものではなく, Matlisp [6]のように行列操作のための豊富な函数を有するパッケージでもありません. その上, Lapack パッケージに含まれる全てのルーチンが Maxima 上の函数として使える訳ではなく, Maxima の行列から Lisp の配列への変換を行う内部函数 lapack-lispify-matrix にしても, BLAS で利用可能なルーチンで利用可能な配列を網羅したものではありません. とはいえ, 効率的とは言い難い Maxima の数値行列をより扱い易い LISP の配列に変換して処理を行うために, 比較的大きな数値を扱う際には大きな効果が得られるでしょう.

6.9.2 BLAS について

ここでは Maxima の LAPACK パッケージに限定した話ではなく, より一般的な話を中心に行います. まず, BLAS と LAPACK のルーチンには「**精度+行列の型+処理**」という命名規則があります. BLAS や LAPACK のルーチンを利用する場合, この命名規則に基いて適切なルーチンの選択を行う必要があります. この命名規則については §6.9.5 で詳細を述べます.

この BLAS のルーチンは, その扱う計算式の特徴から三つの水準に分類されます:

- 第一水準: ベクトル単体やベクトル同士の演算

[4] http://www.netlib.org/blas/
[5] ATLAS と GotoBLAS2 はに BSDL で配布されています.
[6] http://http://matlisp.sourceforge.net/

- 第二水準: 行列とベクトルの演算
- 第三水準: 行列同士の演算

このように三つの水準がある理由は, BLAS の開発の歴史的な経緯もあります. 1979 年にリリースされた最初の BLAS は 1970 年代に出始めたベクトル型計算機で効率的に動作するように構築され, それがベクトル演算を中心とする第一水準のルーチンでした. その後, ベクトル演算だけでは不十分なことから, 行列とベクトルの演算を行う第二水準 (1987) と行列同士の演算を行う第三水準 (1989) と 10 年かけて拡張されています [78][85]. また, 上記の水準に加え, 複素数の一次ノルムと割算を計算するルーチンを第 0 水準のルーチンと呼んでいます.

BLAS の利用では数値計算の効率化という面が大きくあります. その効率化を考える上で, 計算機内部の動作を考慮する必要があります. まず, CPU が主メモリにアクセスして処理を行うときに実行性能はデータの転送速度で決定されます. ところで, 最近の CPU は動作周波数を向上させるよりも, コアを増やし, 並列処理で処理速度の向上を目指していますが, このようなマルチコアプロセッサでは複数のコアがデータ転送の帯域を奪い合うという問題が顕著になります. そこでキャッシュにデータがある間に並列して演算 (「**データの再利用**」) を行うことが重要となっています.

さて, BLAS の第一水準のルーチンはベクトル演算であるために, 行列を扱う他の水準と比較して演算量が少なく, その性能は CPU の理論的性能とメモリバンド幅に依存します. 実際, ベクトルの大きさを N とすると, データ量は $O(N)$, 演算量も $O(N)$ と同じ水準となります. そして, キャッシュに載ったデータの再利用はほとんどできないために, この水準のルーチンは実装方法による性能向上があまり期待できません.

BLAS の第二水準のルーチンでは行列とベクトルの演算が主体で, 第一水準よりも計算量とデータ量共に $O(N^2)$ に増大し, その性能はメモリバンド幅に依存しますが, ベクトルのみにデータの再利用性があり, この点で第一水準と異なります.

BLAS の第三水準のルーチンは, 行列同士の処理が主体となるために計算量も $O(N^3)$ と最も多く, データ量も $O(N^2)$ となります. 性能も CPU の理論性能値に依存しますが, $O(N)$ 回のデータの再利用性があり, 実装方法による性能向上が望めます [22].

6.9.3　ベクトルと行列の表記について

この文書でのベクトルや行列の表記の方法について記しておきます. 最初に大文字のアルファベット $A, B, C, \ldots$ で行列を, 小文字のアルファベット $v, u, w, \ldots$ で列ベクトルを, 小文字ギリシャ文字 $\alpha, \beta, \ldots$ でスカラーを表現します. なお, 行列 A を格納する配列は a と小文字で表記し, ベクトル v,u,w を格納する配列は x,y,z と表記することにします.

それから, ベクトルや行列のスカラー積は $\alpha \cdot A$ や $\beta \cdot v$ と表記し, 行列 A と 行列 B の積は $A\,B$, 行列 A とベクトル x の積も $A\,x$ と表記します. それから ${}^tA$ を行列 A の転置, 行列 A の転置と共役 ${}^t\overline{A}$ は

${}^{H}A$ で表現します.

6.9.4 配列への格納方法

配列への行列やベクトルの格納方法は Maxima へのインターフェィスを持った LAPACK パッケージの函数であれば考慮する必要は特にありません. しかし, 実際の動作を確認したり, より高度な LAPACK の利用となれば直接, FORTRAN から LISP に移植されたルーチンに対応する函数を利用する必要が生じます. そのためには行列やベクトルがどのように内部で表現されているかを知っておく必要があります. 以下, FORTRAN や C の BLAS や LAPACK の一般的な話を行います.

まず BLAS や LAPACK でベクトルは一次元配列として扱われています. このベクトルに対して行列は行列の添字をそのまま二次元配列で置き換えたり, 列や行を繋げて単純な一次元配列 として格納する方法だけではなく, 行列の特性を生かして効率的な処理が行えるよう配列への収納の工夫があります. この収納方法に加えて行列の特性に合せたルーチンを用いることで処理効率の向上が狙えます.

行列の収納方法には, まず, 行列を一次元的, あるいは二次元的に素朴な方法で配列に転記した「**一般形式**」, 三角行列, 対称行列やエルミート行列に対しては, 行列を規則的に取込むことで一次元配列で表現する「**圧縮格納形式**」, 帯行列向けの「**帯格納形式**」があります.

一般形式: 行列 A の i 行 j 列成分の a_{ij} をそのまま配列 a の成分 a[i,j] に格納するという形式です. また, 行列の列を並べて一次元配列として扱うこともあります. たとえば, $m \times n$-行列 A の (i, j)-成分 a_{ij} は一次元配列 a の $i + m(j-1)$ -成分になります. この列方向で並べる理由は, 本来の BLAS が作成された FORTRAN で, 2 次元配列の列方向へのアクセスがメモリ上で連続したものになるためです. また FORTRAN で配列の行方向へのアクセスはメモリ上を飛び飛びにアクセスすることになるために速度向上の面で致命的になります. これらの理由から FORTRAN では列を優先としますが, C の場合は列ではなく行が優先されるため, C から FORTRAN の BLAS ルーチンを呼び出す場合, C の配列には FORTRAN の配列を転置した状態で格納することになります. なお, C 向けの CBLAS であれば行優先, 列優先の指定や引数を番地渡しではなく値渡しといった指定が可能となります. なお, Maxima の LAPACK パッケージでは, Maxima の行列は eigenvalue.lisp 内部で定義された内部函数 lapack-lispify-matrix によって行列の列を並べて一次元配列に変換し, Lisp の配列から Maxima の行列への変換では内部函数 lapack-maxify-matrix を用いています. これらの変換を行う内部函数は一般形式以外の格納方法には対応していません. そのため, Maxima からは付属の LAPACK パッケージの一般形式に対応したルーチンのみが標準で利用できるだけです.

この形式の行列を扱うルーチンは命名規則の “MM” の箇所が “GE” となっているものが対応します.

圧縮格納形式: この圧縮格納形式 (Packed storage) と呼ばれる格納方法は上下三角行列, エルミート行列や対角行列を一次元配列に格納するために用いられ, 通常形式よりコンパクトに行列を配列に収納することができます. この形式は記憶容量の面で有利で, この書式は後述のルーチンの命名

規則にて “MM” の箇所の二文字目が “P” となるもので利用されます.

この圧縮格納方式では, 上三角行列であれば $i \leq j$ のときに a_{ij} を a(i + j(j - 1)/2) に格納し, 下三角行列であれば, $i \geq j$ のときに a_{ij} を a(i + (2 n - j)(j - 1)/2) に格納する方式です. 具体的に 4×4 の上三角 U と下三角行列 L をどのように格納するのかを見てみましょう:

行列の配列への格納方法

$$U: \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ 0 & & & a_{44} \end{pmatrix} \Rightarrow a_{11}\underbrace{a_{12}a_{22}}\underbrace{a_{13}a_{23}a_{33}}\underbrace{a_{14}a_{24}a_{34}a_{44}}$$

$$L: \begin{pmatrix} a_{11} & & & 0 \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \Rightarrow \underbrace{a_{11}a_{21}a_{23}a_{41}}\underbrace{a_{22}a_{23}a_{24}}\underbrace{a_{33}a_{43}a_{44}}$$

これらの例から判るように, 一般形式で一次元配列に変換する際に, 上三角, あるいは下三角の列のみを並べて一次元配列に取り込んで行く様子が分ります.

この格納方法は対称行列やエルミート行列でも使えます. 何故なら, 対称行列は $a_{ij} = a_{ji}$, エルミート行列は $a_{ij} = \overline{a_{ji}}$ を充すために上下三角行列と同様に行 列の上三角領域さえ登録できれば復元可能となるからです. したがって, 上三角, あるいは下三角行列の圧縮格納形式で配列に格納し, その行列の種類に応じたルーチンを選択すれば良いことになります. 具体的には, 命名規則の “MM” が対称行列であれば “SP”, エルミート行列であれば “HP” のものを用いれば良いのです.

帯格納形式:　帯行列向けの格納方式で, 帯の成分だけを配列に格納する方式です. 行列 $A \in \mathrm{M(m, n)}$ が kl 個の劣対角成分 (対角成分の下側) と ku 個の優対角成分 (対角成分の上側) を持つ帯行列とします:

$$\begin{array}{c} \\ \mathrm{kl+1}\left\{\vphantom{\begin{matrix}a\\a\\a\\a\end{matrix}}\right. \end{array} \begin{array}{c} \overbrace{\hphantom{a_{11} \quad \cdots \quad a_{1(ku+1)}}}^{\mathrm{ku+1}} \\ \begin{pmatrix} a_{11} & \cdots & a_{1(ku+1)} & & O \\ \vdots & \ddots & & \ddots & \\ a_{(kl+1)1} & & \ddots & & \ddots \\ & \ddots & & \ddots & \\ O & & \ddots & & \ddots \end{pmatrix} \end{array}$$

この行列 M の成分 m_{ij} を $(ku + kl + 1) \times n$ の配列 a に格納する方法です. この収納の方法を netlib

で公開されているルーチンのコメント中の記述を見てみましょう:

```
      DO 20, J = 1, N
         K = KU + 1 - J
         DO 10, I = MAX( 1, J - KU ), MIN( M, J + KL )
            A( K + I, J ) = matrix( I, J )
   10    CONTINUE
   20 CONTINUE
```

ここで 'matrix(I,J)' が行列 M の i, j 成分 $m(i, j)$ に対応し, 'A' が配列 a に対応します. ここで配列 a への格納は列単位で行われますが, 最初に行列の第一列先頭の成分 m_{11} は $a_{(ku+1)1}$, 以降の第一列の成分が続いて配列 a の 1 列目に収納され, $i < ku$ を充す i 列の先頭 m_{1i} は $a_{ku-i,i}$ に収められます. そして, $i \geq ku$ を充す i 列の先頭 m_{1i} は配列 a の i 列の先頭 a_{1i} に格納されます. つまり, $i > ku$ を充す配列 a の i 列はその先頭の $a_{(i-ku)i}$ が一行目に配置されて収納されること, すなわち対角成分を挟んで上に行 ku, 下に kl 行と帯成分が収納されることが判ります.

実例として LAPACK のマニュアルにもある $m = n = 6, ku = 1, kl = 2$ の行列の例を示します:

帯行列の格納方法

行列 A							配列への格納状態						
$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & a_{64} & a_{65} & a_{66} \end{pmatrix}$							$\Rightarrow$	$\begin{bmatrix} * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ a_{31} & a_{42} & a_{53} & a_{64} & * & * \end{bmatrix}$					

ここで '$*$' の箇所は代入を行わない箇所です. この例からも判るように帯行列の帯の部分を積み重ねる格納方法で, このようにより小さな 2 次元配列に格納することが可能です. この帯格納形式は一般形式と同様に列で並べた配列に最終的に変換することができます. また, $kl = 0$ であれば上三角行列, $ku = 0$ ならば下三角行列となりますが, これらの場合の格納方法は通常形式と一致します. そして, 通常形式の場合と同様に行列 A が対称行列やエルミート行列であれば, その行列の上三角, あるいは下三角成分のみを帯格納形式で格納しておくことができます.

6.9.5 ルーチンの命名規則

BLAS や LAPACK のルーチンには命名規則があります. 第 1 にルーチンの名前は **PMMAAA** の書式に限定されます. この文字数は FORTRAN の函数名の制約に由来するもので, これらのアルファベットには次の意味があります:

$$\underbrace{\text{P}}_{\text{精度}} \quad \underbrace{\text{MM}}_{\text{行列の型}} \quad \underbrace{\text{AAA}}_{\text{処理内容}}$$

では, これら **P**, **MM** と **AAA** の内容について解説をしましょう.

P:　処理すべきベクトルや行列等の精度, および, 対象が実数であるか複素数であるかを示します. 具体的には次の表に示すものとなります:

精度を表現する文字

	単精度 (32bit)	倍精度 (64bit)	拡張倍精度 (128bit)
実数	S	D	Q
複素数	C	Z	X

なお, Maxima の LAPACK パッケージでは倍精度の実数と複素数を扱うルーチンのみが提供されているため, 倍精度の実数 **D** と複素数 **C** に対応するルーチンのみが含まれ, **S**, **C** といった単精度, **Q**, **X** といった拡張倍精度を扱うルーチンは含まれていません.

MM:　ルーチンが扱う行列の型が指示されます. ここでの行列の型は, 行列が対角行列であれば **D**, 三角行列は **T**, 対称行列は **S**, エルミート行列は **H**, その他の一般の行列であれば **G** で示され, その行列の性質に続いて, 行列の配列への格納方式を示す文字が入ります. 具体的には一般形式は **G**, 圧縮格納形式は **P**, 帯格納形式が **B** となり, これらを組合せた「**行列の性質+格納形式**」の形式になります:

行列の型を示す二文字

BD	二重対角行列	DI	対角行列	GB	帯行列
GE	一般行列	GG	一般行列 (一般行列の対)	GT	一般三重対角行列
HB	エルミート帯行列	HE	エルミート行列	HP	エルミート行列 (圧縮格納形式)
HG	上 Hessenberg 行列	HS	上 Hessenberg 行列	OR	直交行列
OP	直交行列 (圧縮格納形式)	PB	正値対称/エルミート帯行列	PO	正値対称/エルミート行列
PP	正値対称/エルミート行列 (圧縮格納形式)	PT	正値対称三重対角行列/エルミート三重対角行列	SB	対称帯行列
SP	対称行列 (圧縮格納形式)	SY	対称行列	TB	三重対角行列/帯行列
TG	三角行列	TP	三角行列 (圧縮格納形式)	TR	三角行列
TZ	台形行列	UN	ユニタリ行列	UP	ユニタリ行列*圧縮格納形式)

AAA:　ルーチンの処理内容を示します. ここの文字数は 2 文字から 3 文字となり, BLAS の場合は多くが 2 文字, LAPACK の多くが 3 文字となります. BLAS では **M** が行列, **V** がベクトル, **S** が逆行列を求める処理といった意味があり, それらを組合せた名前となっています:

処理内容を示す文字列

MV	行列とベクトルの積
SV	逆行列とベクトルの積
MM	行列同士の積
SM	逆行列と行列の積
EV	固有値問題
QRF	QR 分解

6.9.6 ルーチンの引数について

BLAS のルーチンの引数について取り決めを説明しておきます. まず, 行列 A を格納させる配列は a と小文字で表記し, 行列の行数は m, 列数を n と表記します. ただし, 正方行列の場合は $m = n$ のために双方を区別せずに用いることがあります. そして, ベクトル v, u を格納する配列は x, y と表記します.

次に添字に関連する引数について述べましょう. ベクトル v に対応する配列 x に対して引数 **INCX** があります. この引数は配列 x の成分に対する増分, すなわち, **添字の増分**で 0 より大でなければなりません. この文書では 'd_x' と表記します. この添字の増分は GotoBLAS では 1 の場合のみ「**真面目に最適化**」が行われているとのことです [22].

そして, 行列 A に対応する配列 a に対しては **Leading Dimension** と呼ばれる引数 **LDA** があります. この引数 LDA はこの文書では 'l' と表記します. この Leading Dimension の由来ですが, FORTRAN では列優先のために行列 A を格納した配列 a の列を繋いだ 1 次元配列に対して行列 A の (i, j) 成分へのアクセスが 'a[i + j*LDA]' によって行われます. このことから多くの BLAS ルーチンでは $\max(1, m)$ と等しい, すなわち, 行列 A の行数 m と等しいという制約が入っています. そのために,「**行列の行数を指定すれば良い**」と Leading Dimension の解説で書かれていることも多くあります. しかし, この LDA は行列 A が帯行列で行列データの格納のために「**帯行列格納**」を行うときに, 行列の行数と違う値を設定なければなりません. 具体的には, 対角成分数 ku, 劣対角成分数 kl とする $n \times n$ 行列のとき, その格納先の配列の大きさが $(ku + kl + 1) \times n$ となるために, 引数 LDA には $ku + kl + 1$ を指定することになります.

BLAS のルーチンの引数には, 行列の変換に関するフラグ, 行列が三角行列の場合に上三角か下三角であるかを示すフラグ, そして, 行列が対角成分が全て 1 となる三角行列, すなわち単位三角行列であるかを示すフラグの三種類のフラグがあります.
まず, 行列の変換に関するフラグは, 与えられた行列の転置や転置共役を指示するためのフラグです. ルーチンの引数として **TRANS** と記述されていますが, この本では単純に f と表記します. このとき, f の値で定まる行列の作用素 op_f は次のものになります:

作用素 op_f の挙動

フラグ f の値	$\mathrm{op}_f(A)$ の値	概要
N, n	A	そのまま
T, t	${}^tA$	転置
C, c	${}^t\overline{A}$	転置+共役

なお, フラグ f の値は二重引用符で括った文字列として現われます.
次に, 三角行列が上三角行列か下三角行列を指示するフラグは, ルーチンの引数として **UPLO** で与えられます. この文書では f_U と表記し, この f_U の値と意味を以下にまとめておきます:

フラグ f_U(UPL0) の値と意味

フラグ f_U の値	概要
U, u	上三角行列の場合
L, l	下三角行列の場合

最後に, 行列が単位三角行列であるかどうかを指示するフラグは, ルーチンの引数として **DIAG** で与えられます. この文書では f_D と表記し, その値を以下にまとめておきます:

フラグ f_D(DIAG) の値と意味

フラグ f_D の値	概要
U, u	単位三角行列の場合
N, n	単位三角行列ではない場合

6.9.7　BLAS のルーチン

BLAS に含まれるルーチンは, 計算する行列やベクトル等を含む式の形式から三つの水準に分類されています. 第一水準が最も基本的な処理で, 行列の複製や回転, ベクトルの和やノルムといった処理が含まれます. 第二水準は行列とベクトルの演算を行います. そして第三水準が行列同士の演算となります. ここで第一水準のルーチンでは直接値が返却されますが, 第二, 第三水準のルーチンの多くでは計算結果がルーチンの引数の末端 (左端側) のベクトルや行列に対応する配列に代入されるので注意が必要です.

以降の解説では, 函数名の先頭の精度を示す箇所を表では「型」とし, それ以外の部分を「ルーチン名」とします. ここで Maxima の LAPACK パッケージに含まれないルーチンについては「型」に括弧を付与します. たとえば, ルーチン scabs1 が含まれていないことを示すために型を「**s**」ではなく「**(s)**」と括弧 "()" を付けて表記します. また, 概要ではそのルーチンが処理する式を前述した規則に従って記述し、 その実例は Maxima 上で実際に LAPACK パッケージの函数を使った例を示します. ここで Maxima の LAPACK パッケージで実数は 'flonum, 複素数は '(complex flonum) と型が決っています. そのために適宜, 型の変換を行って例題を実行しています. また, LAPACK

パッケージの呼び出しはルーチン名の前に文字列 'lapack::' を置いた函数名にすることで行います. たとえば, ルーチン 'dcab1' を呼び出す場合は 'lapack::dcabs1' として通常の LISP の函数として用います.

6.9.8 BLAS の第 0 水準のルーチン

第 0 水準のルーチン

	型			ルーチン	概要
(s)	d			cabs1	$\|\Re\mathrm{e}(x)\|_1 + \|\Im\mathrm{m}(x)\|_1$
(s)	d	(c)	z	ladiv	$x/y \;\; x, y \in \mathbb{C}$

cabs1: 構文は cabs1(x) で複素数 x に対して一次ノルムを計算します. 具体的には次の処理を実行するルーチンです:

$$\mathrm{cabs1}(x) = |\Re\mathrm{e}(x)|_1 + |\Im\mathrm{m}(x)|_1$$

ここで $\Re\mathrm{e}(x)$ と $\Im\mathrm{m}(x)$ は複素数 x の実部と虚部を返す函数です. この実例を示しておきます:

```
(%i132) :lisp (blas::dcabs1 (coerce #c(2 10) '(complex flonum)))

12.0
NIL
```

ladiv: 複素数 $x = (a + i\,b)$ と $y = (c + i\,d)$ の割算 $x/y = (a + i\,b)/(c + i\,d)$ を行い, その実部 p と虚部 q を返却するルーチンです. 構文は実数を引数とする場合は dladiv(a, b, c, d, p, q) のように 6 引数, 複素数を引数とする場合は zladiv(x, y) のように 2 引数で, 次の計算が行なわれます:

$$\begin{cases} p \leftarrow \dfrac{ac - bd}{c^2 + d^2} \\ \\ q \leftarrow \dfrac{ad + bc}{c^2 + d^2} \end{cases}$$

実数引数の場合は実部と虚部の p, q が返却され, 複素数引数の場合は複素数 $p + i\,q$ (Lisp では #c(p, q)) が返却されます.

ここでは dladiv ルーチンの例を示しておきます:

```
(%i121) :lisp (lapack::dladiv ($float 1) ($float 1) ($float 1) ($float -2) (
        $float 1) ($float 1))

NIL
NIL
NIL
NIL
-0.2
0.6
```

6.9.9　BLAS の第一水準のルーチン

この水準に含まれる式は, 行列やベクトルの複製, 二つのベクトル同士の和や内積, ベクトルのノルムです.

第一水準の BLAS 函数

型				ルーチン	概要
(s)	d	(c)	z	axpy	ベクトルの和:$u \leftarrow \alpha \cdot v + u$
(s)	d	(c)	z	copy	ベクトルの複製
(s)	d	(c)	z	swap	ベクトルの入替
(s)	d	(c)	z	dot[c]	ベクトルの内積 ${}^t\overline{v}u$
		(c)	z	dotu	${}^t vu$
(sd)	(d)			sdot	ベクトルの内積 ${}^t\overline{v}u$
(s)	d	(c)	z	scal	$v \leftarrow \alpha \cdot v$ を計算
		cs	zd	scal	$\alpha \cdot v$ を計算
(s)	(d)	(c)s	(zd)	rscl	$v \leftarrow v/\alpha$
(s)	d	(c)	z	rot	Givens 平面回転を計算
		(sc)	(dz)	drot	Givens 平面回転を計算
(s)	d	(c)	z	rotg	平面回転を生成
(s)	d	(c)	z	lartg	平面回転を生成
(s)	d			rotm	平面回転を適用
(s)	d			rotmg	平面回転を生成
(s)	d	(sc)	dz	nrm2	ベクトルの 2-norm:$\| x \|_2$
(s)	d	(sc)	dz	asum	$\| \mathrm{Re}(x) \|_1 + \| \mathrm{Im}(x) \|_1$
		(sc)	(dz)	sum1	$\|x\|_1$
is	id	ic	iz	amax	$\max(\| \mathrm{Re}(x) \|_1 + \| \mathrm{Im}(x) \|_1)$
		(ic)	(iz)	max1	
		(c)	z	lacgv	

Maxima の LAPACK パッケージには倍精度の実数 **d** と複素数の浮動小数点数 **c** 以外の型に関連する函数が含まれていないことに注意して下さい.

axpy:　構文は axpy$(n, \alpha, x, d_x, y, d_y)$ で計算結果を配列 y に代入します. このときに次の計算と代入が行われます:

$$y_{1+d_y\cdot(i-1)} \leftarrow \alpha \cdot x_{1+d_x\cdot(i-1)} + y_{1+d_y\cdot(i-1)} \qquad i \in \{1, \ldots, n\}$$

ここで $d_x, d_y \in \mathbb{N}$, スカラー α と 1 次元配列 x, y は型で指定された数で構成されたものです.

以下に daxpy の例を示します. ここで配列 a1, b1, c1 の値は同じ '#(1.0 5.0 9.0 2.0 4.0 8.0 3.0 3.0 7.0 4.0 2.0 6.0)' としています:

```
MAXIMA> (blas::daxpy 4 10.0 b1 1 c1 2)
```

```
NIL
NIL
NIL
NIL
NIL
NIL
MAXIMA> c1

#(11.0 5.0 59.0 2.0 94.0 8.0 23.0 3.0 7.0 4.0 2.0 6.0)
```

この例では, 計算を行う添字を 4 個, 配列 b1 の添字の増分を 1, 配列 c1 の添字の増分を 2 としているため, 配列 c1 の添字 1, 3, 5, 7 に対応する成分に対して計算と代入が行われ, その他の成分には影響が現われていないことに注目して下さい.

copy: 構文は copy(n, xd_x, y, d_y) で, 配列 y に対して次の代入が行われます:

$$y_{1+d_y\cdot(i-1)} \leftarrow x_{1+d_x\cdot(i-1)} \qquad\qquad i \in \{1, \ldots, n\}$$

ここで, x, y は 1 次元配列, $n, d_x, d_y \in \mathbb{N}$ です. 次に dcopy の例を示しますが, c1 は daxpy の結果を利用しています:

```
MAXIMA> b1

#(1.0 5.0 9.0 2.0 4.0 8.0 3.0 3.0 7.0 4.0 2.0 6.0)
MAXIMA> c1

#(11.0 5.0 59.0 2.0 94.0 8.0 23.0 3.0 7.0 4.0 2.0 6.0)
MAXIMA> (blas::dcopy 3 b1 1 c1 3)

NIL
NIL
NIL
NIL
NIL
MAXIMA> b1

#(1.0 5.0 9.0 2.0 4.0 8.0 3.0 3.0 7.0 4.0 2.0 6.0)
MAXIMA> c1

#(1.0 5.0 59.0 5.0 94.0 8.0 9.0 3.0 7.0 4.0 2.0 6.0)
```

この例では, 複製は 3 個, d_x は 1 で d_y を 3 にしています. そのために配列 c1 には添字が 1, 4, 8 の箇所に対して配列 b1 の添字が 1, 2, 3 の箇所を複製することになります.

swap: 構文は swap(n, x, d_x, y, d_y) で, 1 次元配列 x, y に対して以下の入れ換え処理を行います:

$$x_{1+d_x\cdot(i-1)} \leftrightarrow y_{1+d_y\cdot(i-1)}$$

dot:　構文は dot(n, x, d_x, y, d_y) で, 1 次元配列 x, y に対して以下の方法で内積を計算します:

$$\mathrm{dot}(n, x, d_x, y, d_y) = \sum_{i=1}^{n} \overline{x_{1+d_x\cdot(i-1)}} \cdot y_{1+d_y\cdot(i-1)}$$

このルーチンは配列ではなく数値を返却します. そのために配列の書き換えは生じません. ここでは最初に倍精度の浮動小数点数を扱う ddot の例を示します:

```
MAXIMA> b1

#(1.0  5.0  9.0  2.0  4.0  8.0  3.0  3.0  7.0  4.0  2.0  6.0)
MAXIMA> (blas::ddot 5 b1 1 b1 1)

 127.0
NIL
NIL
NIL
NIL
NIL
MAXIMA> (blas::ddot 3 b1 1 b1 3)

 38.0
NIL
NIL
NIL
NIL
NIL
```

この例では最初は d_x, d_y 共に 1 で n が 5 のために配列 x と y の先頭の 5 個の成分に対して内積を計算し, 次の例では d_y のみを 3 として 3 個の成分に対する内積を計算させています. 次に倍精度の複素数の浮動小数点数を扱う zdotc の例を示します:

```
(%i144) :lisp t1

#(#C(0.0  1.0) #C(1.0  1.0))
(%i144) :lisp (blas::zdotc 2 t1 1 t1 1)

#C(3.0  0.0)
NIL
NIL
NIL
NIL
NIL
```

この例では d_x, d_y 共に 1, n も 2 としているために通常の内積による結果が得られています.

dotu:　構文は dotu(n, x, d_x, y, d_y) で, x, y は複素数で構成された 1 次元配列であり, 以下の dot 積の処理を行います:

$$\mathrm{dotu}(n, x, d_x, y, d_y) = \sum_{i=1}^{n} x_{1+d_x\cdot(i-1)} \cdot y_{1+d_y\cdot(i-1)}$$

この函数も dot ルーチンと同様に数値を返却し, 引数の配列の書き換えは生じません:

```
(%i144) :lisp t1

#(#C(0.0 1.0) #C(1.0 1.0))
(%i144) :lisp (blas::zdotu 2 t1 1 t1 1)
  0: (BLAS:ZDOTU 2 #(#C(0.0 1.0) #C(1.0 1.0)) 1 #(#C(0.0 1.0) #C(1.0 1.0)) 1)
  0: BLAS:ZDOTU returned #C(-1.0 2.0) NIL NIL NIL NIL NIL
#C(-1.0 2.0)
NIL
NIL
NIL
NIL
NIL
```

sdot: 構文は dotu(n, x, d_x, y, d_y) で, x, y は実数で構成された 1 次元配列であり, 以下の dot 積の計算を処理します:

$$\mathrm{sdot}(n, x, d_x, y, d_y) = \sum_{i=1}^{n} x_{1+d_x\cdot(i-1)} \cdot y_{1+d_y\cdot(i-1)}$$

このルーチンも数値を返却し, 引数の配列の書き換えは生じません. なお, Maxima の LAPACK パッケージには含まれていません.

scal: 構文は scal(n, α, x, d_x) で, 1 次元配列 x とスカラー α に対して次の処理を行います:

$$x_{1+d_x\cdot(i-1)} \mapsto \alpha x_{1+d_x\cdot(i-1)}$$

このルーチンは引数の配列 x の書き換えが生じます:

```
#(1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0)
(%i148) a1:a;
(%o148)                  [ 1  2  3  4  5  6  7  8  9  10 ]
(%i149) :lisp a1

#(1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0)
(%i149) :lisp (blas::dscal 5 10.0 a1 2)

NIL
NIL
NIL
NIL
(%i149) :lisp a1

#(10.0 2.0 30.0 4.0 50.0 6.0 70.0 8.0 90.0 10.0)
```

この例では d_x を 2, n を 5 としているために添字 1,3,5,7,9 の成分が 10 倍したもので置き換えが発生しています.

```
(%i145) :lisp a1
```

```
#(#C(0.0  10.0) #C(1.0  1.0))
(%i145) :lisp (blas::zscal 2 (coerce #c(10  0) '(complex flonum)) a1  1)

NIL
NIL
NIL
NIL
(%i145) :lisp a1

#(#C(0.0  100.0) #C(10.0  10.0))
```

この例では d_x は 1, n が 2 のために配列の全成分を 10 倍にしたもので置き換えられています.

rscl:　構文は rscl(n, α, x, d_x) で 1 次元配列 x とスカラー α に対して次の処理を行う函数です:

$$x_{1+d_x(i-1)} \mapsto x_{1+d_x(i-1)}/\alpha$$

このルーチンでは配列の置換が発生します. なお, この rscl ルーチンは Maxima の LAPACK パッケージには含まれていません.

rot:　構文は rot$(n, x, d_x, y, d_y, c, s)$ で, c, s は必ず実数値でなければなりません. この構文で以下の処理を $i \in \{1, \ldots, n\}$ について行います.:

$$\begin{cases} x_{1+d_x\cdot(i-1)} & \mapsto & cx_{1+d_x\cdot(i-1)} + sy_{1+d_y\cdot(i-1)} \\ y_{1+d_y\cdot(i-1)} & \mapsto & cy_{1+d_y\cdot(i-1)} - sx_{1+d_x\cdot(i-1)} \end{cases}$$

このように配列 x, y 双方で置き換えが生じます. ここでは drot ルーチンを用いて実例を示しておきます:

```
(%i149) :lisp a1

#(1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0)
(%i149) :lisp b1

#(10.0  9.0  8.0  7.0  6.0  5.0  4.0  3.0  2.0  1.0)
(%i149) :lisp (blas:drot 3 a1 2 b1 2 1.0  -2.0)

NIL
NIL
NIL
NIL
NIL
NIL
NIL
(%i149) :lisp a1

#(-19.0  2.0  -13.0  4.0  -7.0  6.0  7.0  8.0  9.0  10.0)
(%i149) :lisp b1

#(12.0  9.0  14.0  7.0  16.0  5.0  4.0  3.0  2.0  1.0)
```

ここでの例では d_x, d_y 共に 2, n を 3 としているために, 添字が 1, 3, 5 の成分のみの置き換えが配列 $a1$ と $b1$ の双方に発生していることに注意して下さい.

rotg: 構文は rotg(a, b, c, s) で, Givens 回転を行ったときに Y 座標が 0 となる点 P(r, z) の座標と Givens 回転行列のパラメータ c, s の値を返却します. ここで rotg の引数 c, s は必ず実数です:

$$\begin{cases} r \leftarrow \begin{cases} \text{sgn(a)}\sqrt{a^2+b^2} & \| a \| > \| b \| \\ \text{sgn(b)}\sqrt{a^2+b^2} & \text{その他} \end{cases} \\ z \leftarrow \begin{cases} b/r & \| a \| > \| b \| \\ r/a & \| b \| \geq \| a \| \wedge c \neq 0 \\ 1 & \text{その他} \end{cases} \\ c \leftarrow a/r \\ s \leftarrow b/r \end{cases}$$

このルーチンは配列の書き換えは生じません. 以下に drotg の例を示しておきます:

```
MAXIMA> (blas::drotg 1.0 2.0 3.0 4.0)

2.23606797749979
2.23606797749979
0.4472135954999579
0.8944271909999159
```

lart: rotg ルーチンよりも低速ですが, より高精度に値を計算して返却するルーチンです. 構文は lart(f, g, cs, sn, r) で, 以下を充す cs, sn, r を返却します:

$$\begin{cases} \begin{pmatrix} cs & sn \\ -sn & cs \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix} \\ \\ cs^2 + \| sn \|^2 = 1 \quad , \quad cs \in \mathbb{R} \end{cases}$$

このルーチンは引数の配列の書き換えを行わないルーチンです. 次の dlartg による実例を示しておきます:

```
(%i101) :lisp (lapack::dlartg ($float 1) ($float 2) ($float 1) ($float 1) (
        $float 0.1))

NIL
NIL
0.4472135954999579
0.8944271909999159
2.23606797749979
```

rotm: 構文は rotm(n, x, d_x, y, d_y, p) で, 修正 Givens 回転を適用するルーチンです. ここで x, y は 1 次元の実数配列, p は 5 成分の配列です. このルーチンでは配列 x, y の双方に計算結果で値の入れ換えが生じます.

rotmg:　構文は rotmg(d_1, d_2, d_x, d_y, p) で, 修正 Givens 回転行列を生成するルーチンです. ここで p 以外の引数は全てスカラ値で, p のみが 5 成分の配列です.

nrm2:　構文は nrm2(n, x, d_x) で, ベクトル x の 2-norm を計算するルーチンです. 一般的には次の処理を行っています:

$$\mathrm{nrm2}(n, x, d_x) = \sqrt{\sum_{i=1}^{n} \Re\mathrm{e}(x_{1+d_x l\cdot(i-1)})^2 + \Im\mathrm{m}(x_{1+d_x\cdot(i-1)})^2}$$

実数に対する dnrm2 と znrm2 の二つの例を示します:

```
(%i149) :lisp b1

#(12.0 9.0 14.0 7.0 16.0 5.0 4.0 3.0 2.0 1.0)
(%i149) :lisp (blas::dnrm2 5 b1 2)

24.819347291981714
NIL
NIL
NIL

(%i149) :lisp c1

#(#C(1.0 1.0) #C(0.0 1.0) #C(1.0 0.0) #C(0.0 -3.0) #C(4.0 5.0))
(%i149) :lisp (blas::dznrm2 4 c1 1)

3.6055512754639896
NIL
NIL
NIL
```

asum:　構文は asum(n, x, d_x) で, 1 次元配列 x に対し, 1-norm を計算するルーチンです:

$$\mathrm{asum}(n, x, d_x) = \sum_{i=1}^{n} |\Re\mathrm{e}x_{1+d_x\cdot(i-1)}| + |\Im\mathrm{m}x_{1+d_x\cdot(i-1)}|$$

asum ルーチンは 1-norm を数値で返却し, 引数の添字の成分の置き換えは一切行いません. 以下に dasum の例を載せておきますが, この例では n と d_x を変更しているので, その結果の違いを確認して下さい:

```
MAXIMA> a1

#(1.0 5.0 9.0 2.0 4.0 8.0 3.0 3.0 7.0 4.0 2.0 6.0)
MAXIMA> (blas::dasum 3 a1 2)

14.0
NIL
NIL
NIL
MAXIMA> (blas::dasum 3 a1 3)
```

```
 6.0
NIL
NIL
NIL
MAXIMA> (blas::dasum 3 a1 5)

 11.0
NIL
NIL
NIL
MAXIMA> (blas::dasum 3 a1 3)
```

また, 引数が複素数となる dzasum の例を以下に示しておきます. こちらの例では d_x を 1 とし, n のみを 4 や 5 としているので, 配列の頭から 4 成分や 5 成分に対して 1-norm を計算していることに注意して下さい:

```
(%i152) :lisp c1

#(#C(1.0 1.0) #C(0.0 1.0) #C(1.0 0.0) #C(0.0 -3.0) #C(4.0 5.0))
(%i152) :lisp (blas::dzasum 4 c1 1)

 7.0
NIL
NIL
NIL
(%i152) :lisp (blas::dzasum 5 c1 1)

 16.0
NIL
NIL
NIL
```

6.9.10 BLAS の第二水準のルーチン

第二水準に含まれる BLAS のルーチンは行列とベクトルとの積や行列同士の和です. より詳細に分類すると, 行列とベクトルの積, 行列とベクトルの積とベクトルの和, 二つのベクトルから構成した行列と与えられた行列の和となります.

三角行列とベクトルの積

ここでは行列 A を三角形行列に限定し, $\mathrm{op}_f(A)x$ や $\mathrm{op}_f(A)^{-1}x$ を計算するルーチンを紹介します:

第二水準の BLAS 函数 (三角行列とベクトルの積)

型		型		ルーチン	行列の種類
(s)	d	(c)	z	trmv	一般
(s)	d	(c)	z	tbmv	一般帯行列
(s)	d	(c)	z	tpmv	圧縮格納形式
(s)	d	(c)	z	trsv	一般
(s)	d	(c)	z	tbsv	帯行列
(s)	d	(c)	z	tpsv	圧縮格納形式

ここで扱う行列は三角形行列, つまり, 対角成分の下側, あるいは上側が全て 0 となる正方行列です. ここで紹介するルーチンの引数としては, この行列の形に関連する三種類のフラグ f_U, f, f_D に加え, 行列 A とベクトル v の内容を格納した配列と行列の行数に対応する n があります. また, 圧縮形式以外の配列を利用する場合にややこしい引数として $l = \max(1, n)$ を充す変数 l を必要としますが, 実用上は行列の行数に合せることになります. それから, ベクトル v に対応する配列の添字の増分も引数に含まれます. そして, 帯行列を処理するルーチンでは, 帯行列の幅に対応する整数値 k が引数に加わります.

ここで形に関するフラグの取り得る値とその意味を以下に纏めておきます:

三種類のフラグ

フラグ	取り得る値	概要
f_U	{U, u, L, l}	行列が上三角 (U,u) か下三角 (L,l) であるかを指定
f	{N, n, T, t, C, c}	作用素 op_f のフラグ
f_D	{U, u, N, n}	行列が単位三角行列 (U,u) かそれ以外 (N,n) を指定

ここで行列 A が「**単位三角行列 (unit triangular matrix)**」であるとは, 対角成分が全て 1 となる三角行列のことです.

trmv:　構文は trmv($f_U, f, f_D, n, a, l, x, d_x$) で, $n \times n$ の三角行列 A と n 次元のベクトル v に対して $v \leftarrow \mathrm{op}_f(A)lv$ を計算するルーチンです. このルーチンでは結果が配列 x に収められるために配列 x のみ書き換えが生じます. ここでは例として以下の演算の様子を示しておきます:

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

```
(%i166) :lisp a1

#(1.0 0.0 0.0 0.0 1.0 0.0 1.0 -2.0 1.0)
(%i166) :lisp v1
```

```
#(1.0 2.0 3.0)
(%i166) :lisp (blas:dtrmv "u" "n" "u" 3 a1 3 v1 1)

NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
(%i166) :lisp v1

#(4.0 -4.0 3.0)
```

---

tbmv: 構文は tbmv($f_U, f, f_D, n, k, a, l, x, d_x$) で, $n \times n$ 帯三角行列 A に対して $x \leftarrow \mathrm{op}_f(A)x$ を計算するルーチンです. ここでの配列 a は帯行列 A の内容を格納する配列で, k はその帯の幅に対応する整数です. このルーチンの処理結果は配列 x に格納されます. したがって配列 x の書き換えが生じることに注意して下さい.

tpmv: 構文は tpmv($f_U, f, f_D, n, a, x, d_x$) で, $n \times n$ の帯三角行列 A に対して $x \leftarrow \mathrm{op}_f(A)x$ を計算するルーチンです. ここでの配列 a は圧縮形式で行列 A の内容を格納しています. このルーチンの処理結果は配列 x に格納されます. したがって, 配列 x の書き換えが生じていることに注意して下さい.

trsv: 構文は trsv($f_U, f, f_D, n, a, m, x, d_x$) で, $n \times n$ の帯三角行列 A に対して $x \leftarrow \mathrm{op}_f(A)^{-1}x$ を計算するルーチンです. このルーチンの処理結果は配列 x に格納されます. したがって, 配列 x の書き換えが生じていることに注意して下さい.

ここでの例では, dtrmv の例での計算結果を用いて元のベクトルを計算させています. 具体的には次の計算を行っています:

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 4 \\ -4 \\ 3 \end{pmatrix}$$

```
(%i166) :lisp v1

#(4.0 -4.0 3.0)
(%i166) :lisp (blas:dtrsv "u" "n" "u" 3 a1 3 v1 1)

NIL
NIL
NIL
NIL
NIL
```

```
NIL
NIL
NIL
(%i166) :lisp v1

#(1.0 2.0 3.0)
```

tbsv: 構文は tbsv($f_U, f, f_D, n, k, a, l, x, d_x$) で, $n \times n$ の帯三角行列 A に対して $\mathrm{op}_f(A)^{-1}x$ を計算するルーチンです. ここで配列 a は行列 A の帯行列格納形式で, k はその帯の幅に対応します. このルーチンの処理結果は配列 x に格納されます. したがって, 配列 x の書き換えが生じていることに注意して下さい.

tpsv: 構文は tpsv($f_U, f_t, f_d, n, a, x, d_x$) で, $n \times n$ の三角行列 A に対して $\mathrm{op}_f(A)^{-1}x$ を計算するルーチンです. ここで配列 a は行列 A の圧縮格納形式になります. このルーチンの処理結果は配列 x に格納されます. したがって, 配列 x の書き換えが生じていることに注意して下さい.

$\alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ の計算を行うルーチン

行列とベクトルの積にベクトルを加える処理, すなわち, $\alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ の計算を行うルーチンの解説を行います. ここで各ルーチンの引数は行列の型による違いを除き, 引数として行列の転置, 共役転置を与える作用素 op_f のフラグ f, 行列の行数と列数に対応する m, n, ベクトルに対応する配列 x の添字の増分 d_x, スカラの α, β があります. また, $l = \max(1, m)$ を充す引数 l もあります

以下に, この処理に関連するルーチンの名前を挙げます:

第二水準の BLAS ルーチン ($\alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$)

型				ルーチン	行列
(s)	d	(c)	z	gemv	一般の行列
(s)	d	(c)	z	gbmv	一般の帯行列
		(c)	z	hemv	エルミート行列
		(c)	z	hbmv	エルミート帯行列
		(c)	z	hpmv	エルミート行列圧縮格納形式)
(s)	d	(c)	z	symv	対称行列
(s)	d			sbmv	対称帯行列
(s)	d			spmv	対称行列の圧縮格納形式

gemv: 構文は gemv($f, m, n, \alpha, a, l, x, d_x, \beta, y, d_y$) で, 一般の $m \times n$ の行列 A と n 次元のベクトル v, m 次元のベクトル v に対して $u \leftarrow\!\shortmid \alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ を処理します. このルーチンの処理結果は配列 y に格納されます. したがって, 配列 y の書き換えが生じていることに注意して下さい.

最初の例として次の式の計算を行います:

$$2\cdot\begin{pmatrix}1&0&0&1\\1&0&3&2\\4&-2&0&-2\end{pmatrix}\begin{pmatrix}1\\2\\3\\4\end{pmatrix}-3\cdot\begin{pmatrix}1\\0\\-1\end{pmatrix}$$

```
(%i240) :lisp a1

#(1.0 1.0 4.0 0.0 0.0 -2.0 0.0 3.0 0.0 1.0 2.0 -2.0)
(%i240) :lisp v1

#(1.0 2.0 3.0 4.0)
(%i240) :lisp u1

#(1.0 0.0 -1.0)
(%i240) :lisp (blas::dgemv "n" 3 4 2.0 a1 3 v1 1 -3.0 u1 1)

NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
(%i240) :lisp u1

#(7.0 36.0 -13.0)
```

次に作用素 op_f を使った例を示しておきます:

$$2\cdot\mathrm{op}_T\left(\begin{pmatrix}1&0&0&1\\1&0&3&2\\4&-2&0&-2\end{pmatrix}\right)\begin{pmatrix}1\\0\\-1\end{pmatrix}-3\cdot\begin{pmatrix}1\\2\\3\\4\end{pmatrix}$$

```
(%i243) :lisp a1

#(1.0 1.0 4.0 0.0 0.0 -2.0 0.0 3.0 0.0 1.0 2.0 -2.0)
(%i243) :lisp v1

#(1.0 2.0 3.0 4.0)
(%i243) :lisp u1

#(1.0 0.0 -1.0)
(%i243) :lisp (blas::dgemv "t" 3 4 2.0 a1 3 u1 1 -3.0 v1 1)

NIL
NIL
```

```
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
(%i243) :lisp v1

#(-9.0 -2.0 -9.0 -6.0)
```

gbmv: 構文は gbmv($f, m, n, k_l, k_u, \alpha, a, l, x, d_x, \beta, y, d_y$) で, $m \times n$ の帯行列 A を格納した配列 a, ベクトル v, u を格納した配列 x, y に対して $u \leftarrow\!\!\!\rightarrow \alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ を処理します. ここで引数の k_l, k_u はそれぞれ対角成分よりも下側の帯の幅, 上側の帯の幅を指定する正整数です. このルーチンの処理結果は配列 y に格納されます. したがって, 配列 y の書き換えが生じていることに注意して下さい.

hemv: 構文は hemv($f_U, n, \alpha, a, l, x, d_x, \beta, y, d_y$) で, $n \times n$ のエルミート行列 A を格納した配列 a, ベクトル v, u を格納した配列 x, y に対して $u \leftarrow\!\!\!\rightarrow \alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ を処理します. なお, フラグ f_U は上三角 ("u", "U") か下三角 ("l", "L") の何れかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納されます. したがって, 配列 y の書き換えが生じていることに注意して下さい.

ここで次の式を f_U を"l", "u"をそれぞれ指定して計算してみます:

$$(2+i)\cdot\begin{pmatrix} 0 & i & 0 \\ -i & 1-2i & \\ 0 & 2i & 3 \end{pmatrix}\begin{pmatrix} 1 \\ 2i \\ -3i \end{pmatrix} + i\cdot\begin{pmatrix} i \\ 0 \\ 1 \end{pmatrix}$$

最初に f_U = "l" の場合を計算してみます:

```
(%i267) :lisp (blas::zhemv "l" 3 (coerce #c(2 1) '(complex flonum)) c1 3 w1 1 (
        coerce #c(0 1) '(complex flonum)) u1 1)

NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
(%i267) :lisp u1

#(#C(-5.0 -2.0) #C(-13.0 -4.0) #C(1.0 -21.0))
```

次に f_U = "u" で計算してみます:

```
(%i267) :lisp (blas::zhemv "u" 3 (coerce #c(2 1) '(complex flonum)) c1 3 w1 1 (
        coerce #c(0 1) '(complex flonum)) u1 1)

NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
(%i267) :lisp u1

#(#C(-5.0 -2.0) #C(-13.0 -4.0) #C(1.0 -21.0))
```

このように両者は一致しています.
ちなみに Maxima の計算結果も示しておきましょう:

```
(%i273) [c1,w1,u1];
                    [  0      %i      0    ]  [   1     ]  [ %i ]
                    [                      ]  [         ]  [    ]
(%o273)            [[ - %i    1    - 2 %i ], [  2 %i   ], [ 0  ]]
                    [                      ]  [         ]  [    ]
                    [  0     2 %i     3    ]  [ - 3 %i  ]  [ 1  ]
(%i274) (2+%i)*c1.w+%i*u,expand;
                                  [ - 2 %i - 5  ]
                                  [             ]
(%o274)                           [ - 4 %i - 13 ]
                                  [             ]
                                  [  1 - 21 %i  ]
```

hbmv: 構文は hbmv($f_U, n, k, \alpha, a, l, x, d_x, \beta, y, d_y$) で, $n \times n$ のエルミート行列で対角成分からの幅が k の帯行列でもある行列 A を帯行列形式で格納した配列 a, ベクトル v, u を格納した配列 x, y に対して $u \leftarrow\!\shortmid \alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ を処理します. なお, フラグ f_U は上三角 ("u", "U") か下三角 ("l", "L") の何れかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納されます. したがって, 配列 y の書き換えが生じていることに注意して下さい.

hpmv: 構文は hpmv($f_U, n, \alpha, a, x, d_x, \beta, y, d_y$) で, $n \times n$ のエルミート行列 A の圧縮格納した配列 a, ベクトル v, u を格納した配列 x, y を用いて $u \leftarrow\!\shortmid \alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ を処理します. なお, フラグ f_U は上三角 ("u", "U") か下三角 ("l", "L") の何れかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納されます. したがって, 配列 y の書き換えが生じていることに注意して下さい.

symv: 構文は symv($f_U, n, \alpha, a, l, x, d_x, \beta, y, d_y$) で, $n \times n$ の対称行列 A を格納した配列 a, ベクトル v, u を格納した x, y を用いて $u \leftarrow\!\shortmid \alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ を処理します. なお, フラグ f_U は上三角

("u", "U") か下三角 ("l", "L") の何れかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納されます. したがって, 配列 y の書き換えが生じていることに注意して下さい.

例として次の式を計算してみましょう:

$$2\cdot\begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 0 & 3 \end{pmatrix}\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}+5\cdot\begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix}$$

ここではフラグ f_U = "u" とした場合の結果を示しておきます:

```
(%i283) :lisp s1

#(0.0 -1.0 1.0 -1.0 1.0 0.0 1.0 0.0 3.0)
(%i283) :lisp x1

#(1.0 2.0 3.0)
(%i283) :lisp y1

#(1.0 0.0 -2.0)
(%i283) :lisp (blas::dsymv "u" 3 (coerce 2 'flonum) s1 3 x1 1 (coerce 5 'flonum)
         y1 1)

NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
NIL
(%i283) :lisp y1

#(7.0 2.0 10.0)
```

Maxima の結果も示しておきます:

```
(%i285) [s1,x1,y1];
                          [  0   - 1   1 ]  [ 1 ]  [  1  ]
                          [              ]  [   ]  [     ]
(%o285)                   [[ - 1   1   0 ], [ 2 ], [  0  ]]
                          [              ]  [   ]  [     ]
                          [  1    0    3 ]  [ 3 ]  [ - 2 ]
(%i286) 2*s1.x1+5*y1,expand;
                                          [ 7  ]
                                          [    ]
(%o286)                                   [ 2  ]
                                          [    ]
                                          [ 10 ]
```

sbmv: 構文は sbmv($f_U, n, k, \alpha, a, l, x, d_x, \beta, y, d_y$) で, 対称行列で帯行列でもある行列 A を幅 k の帯形式で格納した配列 a, ベクトル v, u を格納した x, y を用いて $u \leftarrow\!\!\rightarrow \alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ を処理します. なお, フラグ f_U は上三角 (”u”, ”U”) か下三角 (”l”, ”L”) の何れかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納されます. したがって, 配列 y の書き換えが生じていることに注意して下さい.

spmv: 構文は spmv($f_U, n, \alpha, a, x, d_x, \beta, y, d_y$) で, 対称行列 A を圧縮して格納した配列 a, ベクトル v, u を格納した x, y を用いて $u \leftarrow\!\!\rightarrow \alpha \cdot \mathrm{op}_f(A)v + \beta \cdot u$ を処理します. なお, フラグ f_U は上三角 (”u”, ”U”) か下三角 (”l”, ”L”) の何れかを用いて計算するかを指定するフラグです. このルーチンの処理結果は配列 y に格納されます. したがって, 配列 y の書き換えが生じていることに注意して下さい.

行列を生成するルーチン

第二水準の最後のルーチンは, 行ベクトルと列ベクトルの積から生成される行列と同じ大きさのベクトルの和を計算するものです. ここで「**階数 1 の更新 (rank-1 update)**」と「**階数 2 の更新 (rank-2 update)**」と呼ばれる処理があります. 階数 1 の更新が $A \leftarrow\!\!\rightarrow \alpha \cdot v^t\overline{u} + A$ を行う処理, 階数 2 の更新が $A \leftarrow\!\!\rightarrow \alpha \cdot (v^t\overline{u} + A$ を v^Hv や $v^Hv + v^Hv$ によって $n \times n$ の正方行列を生成し, 階数 2 の更新が二つの n 次元の列ベクトル v, u から v^Hu や $v^Hu + u^Hv$ より $n \times n$ の正方行列を生成します. そして, 階数 1 や階数 2 の更新で生成した行列と正方行列 A との和を計算するものです:

第二水準の BLAS ルーチン (行列を生成)

型				ルーチン	概要	条件
		(c)	z	her	$\alpha \cdot v^t\overline{v} + A$	$A = {}^t\overline{A}$
		(c)	z	hpr	$\alpha \cdot v^t\overline{v} + A$	$A = {}^t\overline{A}$(圧縮格納形式)
		(c)	z	her2	$\alpha \cdot v^t\overline{u} + u^t\overline{(\alpha \cdot v)}$	$A = {}^t\overline{A}$
		(c)	z	hpr2	$\alpha \cdot v^t\overline{u} + u^t\overline{(\alpha \cdot v)}$	$A = {}^t\overline{A}$(圧縮格納形式)
(s)	d	(c)	z	syr	$\alpha \cdot v^tv + A$	$A = {}^tA$
(s)	d	(c)	z	spr	$\alpha \cdot v^tv + A$	$A = {}^tA$(圧縮格納形式)
	d			ger	$\alpha \cdot v^tu + A$	$A \in M(m, n)$
		(c)	z	gerc	$\alpha \cdot v^t\overline{u} + A$	$A \in M(m, n)$
(s)	(d)	(c)	z	geru	$\alpha \cdot v^tu + A$	$A \in M(m, n)$
(s)	d			syr2	$\alpha \cdot v^tu + \alpha \cdot u^tv + A$	$A = {}^tA$
(s)	d			spr2	$\alpha \cdot v^tu + \alpha \cdot u^tv + A$	$A = {}^tA$(圧縮格納形式)

なお, 階数 2 のルーチンは名前の末尾に 2 が付いているために容易に判ります. 実際, her, hpr,syr,spr,gerc と geru が階数 1 の更新で, her2, hpr2, syr2 と spr2 が階数 2 の更新となります.

her: 構文は her($f_U, n, \alpha, x, d_x, a, l$) で, 階数 1 の更新を行うルーチンです. エルミート行列 A を一般形式で収納した配列 a, ベクトル v を収納した配列 x に対し, $A \leftarrow\!\!\rightarrow \alpha \cdot v^t\overline{v} + A$ を処理します.

hpr:　構文は hpr($f_U, n, \alpha, x, d_x, a$) で, 階数 1 の更新を行うルーチンです. エルミート行列 A を圧縮格納形式で収納した配列 a, ベクトル v を収納した配列 x に対し, $A \leftarrow\!\shortmid \alpha \cdot v^t\overline{v} + A$ を処理します.

syr:　構文は syr($f_U, n, \alpha, x, d_x, a, l$) で, 階数 1 の更新を行うルーチンです. 行列 A が実数行列であれば, A は対称行列, 行列 A が複素数行列であれば A はエルミート行列で, 一般形式で配列 a に格納され, ベクトル v を収納した配列 x に対し, $A \leftarrow\!\shortmid \alpha \cdot v^t\overline{v} + A$ を処理します.

spr:　構文は spr($f_U, n, \alpha, x, d_x, a$) で, 階数 1 の更新を行うルーチンです. 行列 A が実数行列であれば, A は対称行列, 行列 A が複素数行列であれば A はエルミート行列で, 圧縮格納形式で配列 a に格納され, ベクトル v を収納した配列 x に対し, $A \leftarrow\!\shortmid \alpha \cdot v^t\overline{v} + A$ を処理します.

ger:　構文は ger($f_U, n, \alpha, x, d_x, a$) で, 階数 1 の更新を行うルーチンです. 行列 A が実数行列であれば, A は対称行列, 行列 A が複素数行列であれば A はエルミート行列で, 圧縮格納形式で配列 a に格納され, ベクトル v を収納した配列 x に対し, $A \leftarrow\!\shortmid \alpha \cdot v^t\overline{v} + A$ を処理します.

gerc:

geru:

her2:

hpr2:

syr2:

spr2:

第三水準の函数

第三水準の函数では行列同士の積を含む式 $\alpha \cdot \mathrm{op}_{f_1}(A)\mathrm{op}_{f_2}(B) + \beta \cdot \mathrm{op}_{f_2}(C)$ の計算処理を行います:

第三水準の BLAS 函数

型				ルーチン	概要	条件
(s)	d	(c)	z	gemm	αop(A)op(B) + βC	行列 A は一般行列, op(A) → A, ^{t}A, ^{t}A, $C \in M(m,n)$
		(c)	z	hemm	$\alpha AB + \beta C$, $\alpha BA + \beta C$	$A = {}^{t}\overline{A}$, $A = {}^{t}\overline{A}$, $C \in M(m,n)$
(s)	d	(c)	z	symm	$\alpha AB + \beta C$, $\alpha BA + \beta C$	$A = {}^{t}A$, $A = {}^{t}A$, $C \in M(m,n)$
(s)	d	(c)	z	trmm	αop(A)B, αBop(A)	行列 A は三角行列, op(A) → A, A^{T}, A^{H}, $B \in M(m,n)$
(s)	d	(c)	z	trsm	αop(A^{-1})B, αBop(A^{-1})	行列 A は三角行列, op(A) → A, A^{T}, A^{H}, $B \in M(m,n)$
		(c)	z	herk	$\alpha AA^H + \beta C$, $\alpha A^H A + \beta C$	$A = {}^{t}\overline{A}$, $C \in M(n,n)$
(s)	d	(c)	z	syrk	$\alpha A^t A + \beta C$, $\alpha^t AA + \beta C$	$A = {}^{t}A$, $C \in M(n,n)$
		(c)	z	her2k	$\alpha AB^H + \overline{\alpha}BA^H + \beta C$, $\alpha A^H B + \overline{\alpha}B^H A + \beta C$	$A = {}^{t}\overline{A}$, $C \in M(n,n)$
(s)	d	(c)	z	syr2k	$\alpha AB^T + \alpha BA^T + \beta C$, $\alpha A^T B + \alpha B^T A + \beta C$	$A = {}^{t}A$, $C \in M(n,n)$

dgemm: $\alpha \cdot \mathrm{op}(A) \cdot \mathrm{op}(B) + \beta C$ を処理する函数. ここで op(A) は行列 A に対し転置を行うかどうかを定める函数.

dsymm: $\alpha AB + \beta C$ か $\alpha BA + \beta C$ の何れかを処理する函数.

dsyrk: $C \leftarrow\!\rightarrow \alpha A^t A + \beta C$ か $C \leftarrow\!\rightarrow \alpha^t AA + \beta C$ の何れかを処理する函数.

dtrmm: $B \leftarrow\!\rightarrow \alpha \cdot \mathrm{op}(A) + B$ か $B \leftarrow\!\rightarrow \alpha \cdot B\mathrm{op}(A)$ の何れかを処理する函数.

6.9.11 LAPACK の構成

LAPACK のルーチンはドライバ, 計算, 補助の三種類に分類されます.

ドライバルーチン

連立一次方程式を解くといったルーチンです.

- 線形方程式
- 線形最小二乗法問題 (LLS)
- 一般化線形最小二乗法 (LSE や GLM) 問題
- 標準固有値問題と特異値分解
- 一般化固有値問題と特異値分解

線形方程式：　ルーチン名の末尾が「SV」で終わるものと「SVX」で終るものの二種類があります. 「SV」は Simple driver で $AX = B$ の形式の線型方程式を解きます. 「SVX」は Expert driver と呼ばれ, 次に示す計算に対応したものです:

- ${}^tAX = B, A^*X = B$
- 特異点周辺での行列 A の条件数の計算等
- よりよい解の計算, 前方/後方誤差の推定値の計算
- 方程式系の均衡化

ここで「方程式系の均衡化/平衡化」とは, 行列 A を相似変換することで行列の成分の絶対値の差を減らす処理です. この処理を行う理由は, 行列を構成する浮動小数点数はあくまでも数の近似値なので, 行列成分の絶対値でその最大値と最小値の差が大きければそれだけ計算精度を保つ上で不利になります. そこで, 差を小さくすることで精度の問題を低減することを目的とします.

なお,「SVX」ルーチンは通常の「SV」ルーチンよりも特殊な計算であるために二倍程度の記憶容量を必要とします.

計算ルーチン

ドライバルーチン内部の実際の計算で用いられるルーチンです. ドライバルーチンに必要な機能を持つものがなければ, この計算ルーチンを組合せて問題を解くことになります.

補助ルーチン

補助的に用いられるルーチンです. BLAS を拡張したものも含まれます.

DGEMM(倍精度汎用行列演算函数)

DGEMM は BLAS の第三水準で規定される倍精度行列積を計算する函数で, $\alpha AB + \beta C$ を計算します. 行列演算の多くが, この形式に還元可能なため, この函数の実装方法が BLAS の計算処理速度を大きく左右します.

6.9.12 Maxima の函数

ここでは Maxima から直接利用できる函数について解説し, 内部函数については後述します.

行列の三水準に関連する函数

dgemm(⟨ 行列$_1$⟩, ⟨ 行列$_2$⟩)
dgemm(⟨ 行列$_1$⟩, ⟨ 行列$_2$⟩, ⟨ 設定$_1$⟩, . . .)

dgemm 函数: BLAS 演算の第三水準を実現するための函数で倍精度の浮動小数点数行列を返します. ここで計算される行列の基本形は $\alpha AB + \beta C$ の第三水準の形式で, A が ⟨ 行列$_1$⟩ と設定で指示される行列, B が ⟨ 行列$_1$⟩ と設定で指示される行列, C が設定で与えられる行列項となります.

ここで dgemm 函数で指定可能な設定を以下にまとめておきます:

dgemm の設定の内容

キーワード	指定方法	概要
alpha	alpha=⟨ 実数値 ⟩	⟨ 行列$_1$⟩ と ⟨ 行列 ⟩ の積の係数
beta	beta=⟨ 実数値 ⟩	後述 C で指定した行列の係数
C	C=⟨ 行列 ⟩	⟨ 行列$_1$⟩ と ⟨ 行列 ⟩ の積の項に加える行列項
transpose_a	transpose_=⟨ 真理値 ⟩	'true' であれば ⟨ 行列$_1$⟩ の転置行列で処理
transpose_b	transpose_=⟨ 真理値 ⟩	'true' であれば ⟨ 行列$_2$⟩ の転置行列で処理

この函数は lapack パッケージの lapack.mac 内部で定義された函数で, その定義も 'dgemm(a, b, [options]) :=%dgemm(a, b, options);' という簡単なものです. この%dgemm 函数とその実体 %%dgemm 函数は dgemm.lisp 内部で定義されています.

固有値に関連する函数

dgeev(⟨ 行列 ⟩)
dgeev(⟨ 行列 ⟩, ⟨ 真理値$_1$⟩, ⟨ 真理値$_2$⟩)
zgeev(⟨ 行列 ⟩)
zgeev(⟨ 行列 ⟩, ⟨ 真理値$_1$⟩, ⟨ 真理値$_2$⟩)
dgesv(⟨ 行列$_1$⟩,⟨ 行列$_2$⟩)
dgesvd(⟨ 行列 ⟩)
dgesvd(⟨ 行列 ⟩, ⟨ 真理値$_1$⟩, ⟨ 真理値$_2$⟩)
dlange(⟨ ノルム ⟩, ⟨ 行列 ⟩)
zlange(⟨ ノルム ⟩, ⟨ 行列 ⟩)

dgeev 函数: 実数値行列の固有値と固有ベクトルを計算する函数. 第 2 引数が左固有ベクトル, 第 3 引数が右固有ベクトルを計算するためのフラグとなっています.
この函数の返却値は 3 成分のリストで, リストの第 1 成分が固有値, リストの第 2 成分が右固有ベクトルを並べた行列, 第 3 成分が左固有ベクトルを積み上げて構成された行列です.

dgesv 函数:　線形方程式 $Ax = b$ の解を計算する函数. dlartg, dlapy2, dgebak, dtrevc, dhseqr, dorghr, dlacpy, dgehrd, dgebal, dlascl, dlange, dlabad, dlamch,ilaenv に依存します.

dgesvd 函数:　行列の特異値分解 (SVG=Singular Value Decomposition) を計算する函数です. dorglq, dgelqf, dormbr, dorgqr, dlacpy, dbdsqr, dorgbr, dgebrd, dlaset, dgeqrf, dlascl, dlange, dlamch, ilaenv に依存します.

dgesdd 函数:　行列の特異値分解 (SVG=Singular Value Decomposition) を計算する函数です. dgesvd 函数との違いは分割統治法 (divide-and-conquer method) を用いて処理を高速化した点です. dorglq, dgelqf, dormbr, dorgqr, dlacpy, dbdsdc, dorgbr, dgebrd, dlaset, dgeqrf, dlascl, dlange, dlamch, ilaenv に依存します.

dlange 函数と zlange 函数:　ノルムとしては次のものを指定:
max
one_norm
inf_norm
frobenius

QR 分解に関連する函数

dgeqrf(⟨ 行列 ⟩)

dgeqrf 函数:　行列の QR 分解を行う函数で, dgeqrf.lisp にて定義されています. LAPACK パッケージの dgeqrf を用いた函数で, 引数は QR 分解を行う行列一つのみで, オプション等はありません.

6.9.13　LAPACK パッケージの利用方法

Maxima の LAPACK パッケージは FORTRAN 向けの LAPACK パッケージのソースファイルを Common LISP に移植したものです. そのために LAPACK ライブラリがインストールされていても, そのライブラリを利用することはありません.

lapack パッケージの利用は load(lapack); でパッケージの読込を行う必要があります. 特に, この呼出が初めて行われた場合には Maxima の下層にある Common Lisp を使って中間コードファイルを生成し, 二度目の呼出からは中間コードファイルの読込を行います. この中間コードファイルは, UNIX 系 OS の場合, ホームディレクトリ上の.maxima/binary の下に, Maxima が利用する Common Lisp に対応するディレクトリに置かれます. たとえば, SBCL を利用していれば, .maxima/binary/binary-sbcl/share/lapack, ECL なら.maxima/binary/binary-ecl/share/lapack に関連ファルが置かれます. なお, lapack 以下の構成は共通で, binary-sbcl に全体を制御する函数, blas に行列演算の函数が衆力されています.

LAPACK を利用する利点は何といっても実数値行列の処理の高速化が挙げられます. また, LAPACK のような標準的なライブラリを用いることで, C や FORTRAN のプログラムの移植性の向上が望めることも大きな利点です. だだし, 最適化が行われている訳ではないので過大な期待はすべきではありません. また, 行列も倍精度の浮動小数点数のみで, Maxima の数式を含む行列は扱えないことに注意が必要です.

6.9.14 橋渡しを行う函数

eigensys.lisp に基本的な函数が定義されています.

maxia-matrix-dims: Maxima の行列の行数と列数を返す内部函数です. Maxima の行列の書式を利用した方法で行数と列数を計算しており, 行列として成立しているかどうかの検証は含みません.

complex-maxima-matrix-p: Maxima の行列に純虚数が含まれていれば 't', そうでなければ 'nil' を返却する函数です. この函数では Maxima の函数 imagpart を用いており, 行列の行毎に検証し, imagpart 函数が '0' 以外の値を返すと 't' を返却します.

lapack-lispify-matrix: Maxima の行列を LISP 上の LAPACK で使える LISP の行列に変換します. ここでの引数は, Maxima の行列とその行数と列数の三種類です. Maxima の行列から LISP の配列への変換では, LISP の make-array 函数で 1 次元配列と 2 次元配列の二つを生成し, 2 次元配列に ':display-to' オプションを入れて 1 次元配列に結果が引き渡されるように設定しておきます. それから行列の 1 列目から順番に 2 次元配列に収めますが, その結果, 1 次元配列には行列の列から格納した形になります (FORTRAN の列優先の配列に合せています). このときに各成分の型を実数であれば 'flonum, 複素数であれば, '(complex flonum) に変換していることに注意が必要です. なお, ここで生成する一次元配列には一切の元の行列の大きさの情報は含まれていません. 1 次元配列を make-array 函数で生成する際にその大きさを指定することのみに第 2 と第 3 引数を用いています.

```
15 (defun lapack−lispify−matrix (a nrow ncol)
16   "Convert a Maxima matrix A of dimension NROW and NCOL to Lisp matrix
17   suitable  for  use with LAPACK"
18   (setq  a  ($float  a))
19   (let*  ((array−type (if  (complex−maxima−matrix−p a)
20                            '(complex flonum)
21                            'flonum))
22           (mat (make−array (* nrow ncol)
23                            :element−type array−type))
24           (mat−2d (make−array (list ncol nrow)
25                               :element−type array−type
26                               :displaced−to mat))
```

```
27          (r 0))
28      (dolist (row (cdr a))
29        (let ((c 0))
30          (dolist (col (cdr row))
31            ;; Fortran matrices are in column-major order!
32            (setf (aref mat-2d c r) (if (eql array-type 'flonum)
33                                        (coerce col 'flonum)
34                                        (coerce (complex ($realpart col) ($imagpart col))
35                                                '(complex flonum))
36                                        ))
37            (incf c)))
38        (incf r))
39      mat))
```

このソースファイルに示すように, ここで生成される配列は一般形式のみで, 圧縮格納や帯格納形式には対応していません. そのため, BLAS や LAPACK のルーチンで一般形式以外の格納方式を前提とするルーチンを使うためのインターフェイスを自作する必要があります.

lapack-lispify-matrix:　Lisp の一次元配列を指定した行数と列数の Maxima の行列に変換する函数です. 第 1 引数と第 2 引数は Maxima の行列として見た場合の行列の大きさを指定し, 第 3 引数が LISP の一次元配列となります.

```
15  (defun lapack-maxify-matrix (nrow ncol a)
16    "Convert an LAPACK matrix of dimensions NROW and NCOL into a Maxima
17  matrix ( list of lists )"
18    (let ((2d (make-array (list ncol nrow) :element-type (array-element-type a)
19                          :displaced-to a)))
20      (let (res)
21        (dotimes (r nrow)
22          (let (row)
23            (dotimes (c ncol)
24              ;; Fortran arrays are column-major order!
25              (let ((v (aref 2d c r)))
26                (push (add (realpart v) (mul '$%i (imagpart v))) row)))
27            (push '((mlist) ,@(nreverse row)) res)))
28        '(($matrix) ,@(nreverse res )))))
```

6.10 文字列

6.10.1 Maxima の文字列

この節では Maxima の文字列操作に関連する函数, 主に contrib に含まれる stringproc パッケージに含まれる函数の解説を行います.
Maxima には本来は文字列操作の函数は殆どありません. しかし, LISP 自体は文字列操作に関して非常に強力な言語です. この穴を埋めるのが contrib に含まれる stringproc パッケージです. stringproc パッケージは文字列操作を行うためのさまざまな函数を含んでいます. これらの函数は stringproc.lisp 内部で定義されており, ファイル名からも分るように LISP で記述されています. この stringproc パッケージは Maxima の文字操作の弱さを補強するものとなっています.
この stringproc パッケージて定義される函数は大きく分けて四種類あります. 一つは入出力に関連するもので, ファイルやストリームの処理を行う函数群です. 次に, 与えられた引数に対して真偽値を返す真理函数があります. さらに, 与えられた数値を文字に変換する函数のように, 与えられた引数を文字や数値, あるいは LISP の文字列に変換する函数群, それから最後に, 与えられた Maxima の文字列の結合や指定した位置の文字を取出したりする文字列操作の函数群があります.
ここで注意すべきこととして, Maxima の版によっては Maxima の文字列が LISP の文字列型とは異なるとです. §6.4.1 で解説したように旧来の Maxima の文字列データの内部表現は文字列データの二重引用符を取り除いて先頭に文字 “&” を付けた対象でした. このことを Maxima-5.10.0 で具体的に確認しておきましょう:

```
(%i19) neko1:"x^2+x-1";
(%o19)                                x^2+x-1
(%i20) neko2:x^2+x-1;
                                       2
(%o20)                                x  + x - 1
(%i21) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.

MAXIMA> $neko1
&X^2+X-1
MAXIMA> $neko2
((MPLUS SIMP) -1 $X ((MEXPT SIMP) $X 2))
MAXIMA> (stringp $neko1)
NIL
MAXIMA> (atom $neko1)
T
```

この例では変数 neko1 に文字列"x^2+x-1" を割り当て, 変数 neko2 には式 'x^2+x-1' を割り当てています. to_lisp() で裏で動作している LISP に入り, 変数 neko1 と neko2 の内部表現である '$neko1' と '$neko2' の値を表示させています. ここで '$neko1' に束縛された値が '&X^2+X-1' と先頭に文字 “&” が付き, さらに変数 x が大文字になっていることに注意して下さい. そして$neko1 に束縛させた値は LISP の文字列型ではないことを stringp 函数を用いて確認しています. 実際, このデータの型は atom 型になります.
ところが, Maxima-5.14.0 から Maxima の文字列と LISP の文字列の扱いは同じものになっています. この点には注意が必要になります. 今度は Maxima-5.14.0 で確認しておきましょう:

```
(%i2)  neko1:"x^2+x-1";
(%o2)                                  x^2+x-1
(%i3) :lisp $neko1;
x^2+x-1
(%i3) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.

MAXIMA> $neko1
"x^2+x-1"
```

このように文字列の扱いが 5.13.0 以前と 5.14.0 以降で異なることが判るでしょう. さて, Maxima には文字列型のデータがありますが, LISP との大きな違いは, Maxima には「文字型」がないことです. なお, この節を通して長さが '1' の Maxima の文字列を Maxima の文字と簡単に呼ぶことにします.

Maxima と LISP では文字列型が長く異なっていたために, stringproc パッケージで定義される函数には Maxima と LISP との間での文字列型の変換函数や判別函数が多く含まれています. また, LISP にある函数を Maxima の函数に移植する傾向があるために, このパッケージで定義される大域変数は少なく, 函数も単機能なものが多いのが特徴です.

なお, 旧来の Maxima にも文字列操作の函数がない訳ではありません. たとえば, sconcat 函数のように与えられた複数の Maxima の文字列を結合して一つの LISP の文字列を生成する函数があります:

Maxima 固有の文字列操作の函数

concat(⟨ 式$_1$⟩, ··· , ⟨ 式$_n$⟩)
sconcat(⟨*Maxima* の文字列$_1$⟩, ··· , ⟨*Maxima* の文字列$_n$⟩)
string(⟨ 対象 ⟩)

concat 函数: 引数として Maxima の式, あるいは文字列を取り, 文字列の結合を行う函数です. ここで concat 函数で利用可能な式は Maxima の原子, あるいは評価を行うことで数値や文字列, あるいは原子が返される式です. concat 函数は引数の評価を行い, それから結果を文字列に変換して, これらの文字列を結合した文字列を返します:

```
(%i28) z1:x$
(%i29) concat(z1,12);
(%o29)                                   x12
(%i30) concat("123",z1,12);
(%o30)                                 123x12
```

sconcat 函数: LISP の concatenate 函数を Maxima に実装した函数で, macsys.lisp 内部で定義されています. この函数は与えられた複数の文字列を単純に結合するだけですが, 出力は LISP の文字列になります:

```
(%i1) sconcat("これは","テスト","だよ！");
(%o1)                                 これはテストだよ！
(%i2) neko:sconcat("This"," ","is"," ","a"," ","test!");
```

```
(%o2)                                This is a test!
(%i3) ?stringp(neko);
(%o4)                                    true
```

この例では sconcat 函数の出力が LISP の文字列型であることを確認しています. なお, concat 函数, sconcat 函数共に引数を一つ以上必要とします.
Maxima-5.14.0 より Maxima の文字列は LISP の文字列型に統合されましたが, Maxima-5.13.0 以前で結合した文字列も Maxima の文字列であって欲しい場合, LISP の函数を用いて内部データを料理するか, 次に解説する string 函数を用いるか, あるいは stringproc パッケージの sconc 函数を用いなければなりません.

string 函数: 与えられた対象を Maxima の文字列に変換する函数です. この函数は suprv1.lisp で定義されています:

```
(%i40) tama:string(factor(x^2+2*x+1));
(%o40)                                  (x+1)^2
(%i41) stringp(tama);
(%o41)                                   true
```

この例では Maxima の式 'x^2 + 2 * x + 1' を string 函数で Maxima の文字列に変換しています. そして, 最後に stringproc パッケージの函数である stringp 函数を用いて Maxima の文字列であることを確認しています. なお, Maxima-5.14.0 以降では Maxima の文字列型は LISP の文字列型と一致するために ?stringp(tama) の返却する値も 'true' になります.
それでは stringproc パッケージに含まれる函数の解説を始めましょう. 最初に入力操作に関連する函数について次の小節で解説しましょう.

6.10.2 ストリーム処理に関連する函数

stringproc パッケージには純粋に文字列の操作に関係する函数だけではなく, ストリーム操作の函数も含まれています. Maxima は LISP 上で動作している割にストリーム処理が全体的に弱かったのですが, stringproc パッケージを用いることで, Maxima 言語だけで LISP と機能的に大差無い水準に近づけられるのです.

ファイル直接操作の函数

openw(⟨ ファイル名 ⟩)
opena(⟨ ファイル名 ⟩)
openr(⟨ ファイル名 ⟩)
close(⟨ ストリーム ⟩)

openw 函数: Lisp のストリーム出力をファイルに落すために用いる函数です. ファイルが存在しない場合には新たに生成され, 既存の場合にはファイルを開きますが, ファイルはストリームを閉じた時点で書き換えられてしまいます. この函数は LISP の open 函数を利用しています.

opena 函数:　openw 函数に似ていますが, 既存のファイルに対して末尾にストリーム出力を追加する点で異なります. この函数も LISP の open 函数を利用しています.

openr 函数:　入力ストリームとしてファイルを開く函数です. ファイルが存在しない場合はエラーになります. この函数は LISP の open 函数を利用していますが, 他の函数と違い, open 函数のオプションを設定していません.

close 函数:　開いたストリームを閉じる函数です. LISP の close 函数をそのまま利用しただけの函数です. なお, ファイルへの出力を伴うストリームの場合, このストリームが閉じられた時点でファイルの内容が更新されます.

ストリーム処理の函数

make_string_input_stream(⟨ 文字列 ⟩)
make_string_output_stream()
get_output_stream_string(⟨ ストリーム ⟩)
fposition(⟨ ストリーム ⟩)
fposition(⟨ ストリーム ⟩,⟨ 正整数値 ⟩)
flength(⟨ ストリーム ⟩)
readline(⟨ ストリーム ⟩)

make_string_input_stream 函数:　LISP の make-string-input-stream 函数を利用した函数です. 引数の文字列から入力用のストリームを生成します. ここで引数は Maxima の文字列か LISP の文字列型でなければなりません. なお, LISP の with-input-from-string は make_string_input_stream のように Maxima には実装されていません.

make_string_output_stream 函数:　LISP の make-string-output-stream 函数を利用した函数で, 出力ストリームを開きます. なお, LISP の with-output-from-string は make_string_output_stream のように Maxima で実装されていません.

flength 函数:　指定したストリームの成分数を返す函数です. この函数は LISP の file-length 函数をそのまま利用している函数です. そのためにファイルからの出力ストリームに対してのみ利用可能です.

fposition 函数:　file-position 函数を利用した函数で, ストリーム上でポインタを移動させたり, そのポインタの位置を返したりする函数です. まず, 引数がストリームだけの場合はストリームの頭からのポインタの位置を返します. この場合, 1 がストリームの先頭であることを示します. 次に, 引数がストリームと正整数値が指定されている場合, 正整数値で指定した箇所にポインタを移動させます. 勿論, 指定可能な正整数値の範囲は 1 から flength 函数で求めたストリームの長さに限定されます.

readline 函数: 指定したストリームから一行づつ読取を行う函数です.

ストリームに対して出力を行う函数

freshline()
freshline(⟨ ストリーム ⟩)
newline()
newline(⟨ ストリーム ⟩)
printf(⟨ ストリーム ⟩,⟨*Maxima* の文字列 ⟩)
printf(⟨ ストリーム ⟩,⟨*Maxima* の文字列 ⟩,⟨ オプション ⟩)

freshline 函数: LISP の fresh-line 函数を実装した函数です. この函数は指定したストリーム, 引数がない場合には標準出力に改行を出力します.

newline 函数: LISP の terpri 函数を実装したものです. 引数がない場合に, 標準出力にストリームの指定があれば, 指定されたストリームに対して改行を出力します.

printf 函数: Common LISP の format 函数を Maxima で利用するための函数です. 函数名が C 風なのが面白いことですが, 書式は C ではなく LISP の format 函数の書式に準じます:

数値の書式

~$	お金の表記
~d	十進整数
~b	二進整数
~o	八進整数
~x	十六進整数
~br	b 進整数
~f	浮動小数
~e	科学的表記
~g	~f か ~e を数値の大きさで切替える
~r	整数を英語で表記
~p	複数形

では, 実際に数値で試してみましょう:

```
(%i11) printf(true,"d:%",128);
128:
(%o11)                                 false
(%i12) printf(true,"b:%",128);
10000000:
(%o12)                                 false
(%i13) printf(true,"o:%",128);
200:
(%o13)                                 false
(%i14) printf(true,"x:%",128);
```

```
80:
(%o14)                                  false
(%i15) printf(true,"7r:%",128);
242:
(%o15)                                  false
(%i16) 2*7^2+4*7+2;
(%o16)                                   128
(%i17) printf(true,"r:%",128);
one hundred and twenty-eight:
(%o17)                                  false
(%i18) printf(true,"f:%",128);
128.0:
(%o18)                                  false
(%i19) printf(true,"e:%",128);
1.2799999f+2:
(%o19)                                  false
(%i20) printf(true,"g:%",128.0);
128.      :
(%o20)                                  false
(%i21) printf(true,"e:%",128.0);
1.28E+2:
(%o21)                                  false
(%i22) printf(true,"g:%",128.0);
128.      :
(%o22)                                  false
```

‘~r’ の場合に ‘128’ が ‘one hundred and twenty-eight’ となっているのは面白いことでしょう.
数値に関しては表示枠の指定と右寄, 左寄の指定が行えます:

```
(%i55) printf(true,"16,8f:%",128.0);
     128.00000000:
(%o55)                                  false
(%i56) printf(true,"-16,8f:%",128.0);
128.00000000:
(%o56)                                  false
```

この例から判るように ‘~16,8f’ で ‘128.0’ を右詰めで表示し, ‘~-16,8f’ で左詰め表示になります. C と異なる点は ‘16.8f’ ではなく, ‘16,8f’ と表記する点です. これは紛らわしいので注意が必要でしょう.
数値に関する書式で面白い指定が ‘~r’ と ‘~p’ です. ‘~r’ は整数を英語で置換し, ‘~p’ は ‘1’ 以外に対しては ‘s’ を返すものです.

```
(%i70) dog:3;printf(true,"r 柴犬p  are  same  color.(three 柴犬 s are same color.(
```

次に文字列に関する書式があります:

文字列出力に関する書式

~%	改行 (newline)
~&	改行 (fresh line)
~t	タブ (tab)
~a	Maxima の print 函数と同様に文字列の中身を表示
~s	文字列を二重引用符で括って出力する.
~~	~ を表示する.

すでに ‘~%’ は数値の例題に出ていますが, C の\n に相当します. また, ‘~a’ と ‘~s’ の違いは出力が二重引用符で括られるかどうかという点です. 次の例を見ると明瞭になるでしょう:

```
(%i66) printf(true,"最初はねaだけど、次はsだよ最初はねaだけど、次は"s"だよ
```

printf 文の書式では, 同じ書式を反復して利用するといった出力の制御もできます:

制御に関連する書式

~<	justification, ~>で閉じる.
~(	case conversion, ~) で閉じる.
~[	selection, ~] で閉じる.
~{	リストで与えられた引数に対し, 反復処理を行う~} で閉じる

6.10.3 stringproc パッケージの真理函数

ここでは stringproc パッケージに付属する真理函数について解説します.
まず, 引数が一つだけの真理函数について解説します. これらの函数は与えられた対象が目的の対象 (Maxima の文字列等) に合致する場合に ‘true’ を返し, それ以外は ‘false’ を返す函数です:

stringproc パッケージに含まれる真理函数

lcharp(⟨ 対象 ⟩)
charp(⟨ 対象 ⟩)
stringp(⟨ 対象 ⟩)
lstringp(⟨ 対象 ⟩)
constituent(⟨ 対象 ⟩)
alphanumericp(⟨ 対象 ⟩)
lowercasep(⟨ 対象 ⟩)
uppercasep(⟨ 対象 ⟩)
digitcharp(⟨ 対象 ⟩)

lcharp 函数: LISP の characterp 函数に引数をそのまま引渡す函数で, 与えられた対象が LISP 型の文字の場合に ‘true’ を返す函数です. この函数は内部的に LISP の character 函数を用いています. なお, Maxima の長さが 1 の文字列を与えても ‘false’ が返されます.

charp 函数: 引数が Maxima の文字の場合に ‘true’ を返す函数です. 内部では引数が Maxima の文字列であることを確認し, その文字列の長さが 1 の場合を Maxima の文字としています.

lstringp 函数: LISP の stringp 函数に引数をそのまま引渡す函数で, 与えられた対象が LISP 型の文字列の場合に ‘true’ を返す函数です.

stringp 函数: 引数が Maxima の文字列の場合に ‘true’ を返す函数です.

consituent 函数:　引数が画面上で表示される文字の何れかで空行でないときに 'true' を返す函数です. なお, この函数の実体は [19] の P.61(原書では P.67) で定義されている contituent 函数そのもので, LISP の graphic-char-p 函数と char=函数 を用いています.

alphacharp 函数:　引数が LISP の文字型であり, かつ, アルファベットの場合に 'true' を返す函数です. この函数は alpha-char-p 函数 の Maxima への実装になります.

digitcharp 函数:　数の場合のみに 'true' を返す函数です. この函数は内部で char-int 函数を用いて引数の文字を ACII データの数値に変換し, その数値が 47 よりも大きく, 58 よりも小さい場合に 'true' を返します.

alphanumericp 函数:　名前から判るように引数がアルファベットか数の場合に 'true' を返す函数です. この函数は alphanumericp 函数の Maxima への実装に他なりません.

lowercasep 函数と uppercasep 函数:　lowercasep 函数は引数が小文字の場合に 'true' を返す函数で, 同様に uppercasep 函数は引数が大文字の場合に 'true' を返す函数です. それぞれ, lower-case-p 函数と upper-case-p 函数のの Maxima への実装になります.
次に, 引数が二つの真理函数を以下に纏めておきます:

引数が二つの真理函数

cequal(⟨ 対象$_1$⟩,⟨ 対象$_2$⟩)
cequalignore(⟨ 対象$_1$⟩,⟨ 対象$_2$⟩)
cless(⟨ 対象$_1$⟩,⟨ 対象$_2$⟩)
clessignore(⟨ 対象$_1$⟩,⟨ 対象$_2$⟩)
cgreaterp(⟨ 対象$_1$⟩,⟨ 対象$_2$⟩)
cgreaterpignore(⟨ 対象$_1$⟩,⟨ 対象$_2$⟩)

基本的にこれらの函数は二つの引数の比較を行い, ある条件に結果が合致すれば 'true' を返し, それ以外の場合は 'false' を返す函数です. なお, これらの函数には対応する LISP の函数が存在し, 実質的に LISP の函数の Maxima への実装になっています.

cequal 函数:　LISP の char=函数に対応する函数で, 二つの Maxima の文字が等しい場合に 'true' を返し, そうでない場合には 'false' を返します.

cequalignore 函数:　LISP の char-equal 函数に対応する函数で, 二つの Maxima の文字が等しい場合に 'true' を返し, そうでない場合には 'false' を返します.

clessp 函数:　LISP の char<函数に対応する函数で, 二つの Maxima の文字を ASCII コードで比較する函数です. 基本的に第 1 引数の ASCII コードが第 2 引数の ASCII コードよりも小さい場合に 'true' を返します.

clesspignore 函数: LISP の char-lessp 函数に対応する函数で, 二つの Maxima の文字を ASCII コードで比較する函数です. 基本的に第 1 引数の ASCII コードが第 2 引数の ASCII コードよりも小さい場合に 'true' を返します.

cgreaterp 函数: LISP の char>函数に対応する Maxima の函数で, 二つの Maxima の文字を ASCII コードで比較する函数です. 基本的に第 1 引数の ASCII コードが第 2 引数の ASCII コードよりも大きい場合に 'true' を返します.

cgreaterignore 函数: LISP の char-greaterp 函数に対応する函数で, 二つの Maxima の文字を ASCII コードで比較する函数です. 基本的に第 1 引数の ASCII コードが第 2 引数の ASCII コードよりも大きい場合に 'true' を返します.
文字列に対しても同様の函数があります. ただし, これらの函数は cequal 函数と cequalignore 函数の文字列版に対応する函数だけです.

文字列の真理函数

sequal(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩)
sequalignore(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩)

sequal 函数と sequalignoe 函数: 二つの文字列が等しい場合に true を返す函数です. 各々が LISP の string=函数と strign-equal 函数の Maxima への実装となっています.

6.10.4 文字列変換の函数

以下に LISP と Maxima の間での文字列の変換函数等を纏めておきます.

文字列の変換函数

cunlisp(⟨*LISP* の文字型 ⟩)
sunlisp(⟨*LISP* の文字列 ⟩)
lstring(⟨*Maxima* の文字列 ⟩)
cint(⟨*Maxima* の文字 ⟩)
ascii(⟨*Maxima* の整数値 ⟩)

cunlisp 函数: LISP の文字型データを Maxima の文字列型のデータに変換する函数です.

sunlisp 函数: LISP の文字列型データを Maxima の文字列型データに変換する函数です:

```
(%i43) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.

MAXIMA> (setf $neko1 #\a)
#\a
MAXIMA> (setf $neko2 "aBcD")
```

```
"aBcD"
MAXIMA> (to-maxima)
Returning to Maxima
(%o43)                                  true
(%i44) cunlisp(neko1);
(%o44)                                    a
(%i45) sunlisp(neko2);
(%o45)                                  aBcD
(%i46) lstringp(neko1);
(%o46)                                  false
(%i47) lstringp(neko2);
(%o47)                                  true
```

lstring 函数:　Maxima の文字列を LISP の文字型に変換する函数です. この函数は単純に Maxima 内部で Maxima の文字列を LISP の文字列に変換する l-string 函数に Maxima の文字列を引渡すだけの函数です.

cint 函数:　Maxima の長さが 1 の文字列を ASCII コードの数値に変換する函数です. LISP の char-int 函数を実装したものです. これに対して ascii 函数は cint 函数の逆操作を行う函数で, 与えられた整数値から ASCII コードの数値に対応する Maxima の文字型データを返す函数です:

```
(%i33) map(cint,["a","1"]);
(%o33)                                [97, 49]
(%i34) map(ascii,[40,87]);
(%o34)                                 [(, W]
```

6.10.5　文字列操作の函数

文字操作の函数

scopy(⟨ 文字列 ⟩)
smake(⟨ 整数値 ⟩,⟨ 文字列 ⟩)
charat(⟨ 文字列 ⟩,⟨ 整数値 ⟩)
charlist(⟨ 文字列 ⟩)
slength(⟨ 文字列 ⟩)
parsetoken(⟨ 文字列 ⟩)
sconc(⟨ 文字列$_1$⟩,$\cdots$,⟨ 文字列$_n$⟩)
sposition(⟨ 文字 ⟩,⟨ 文字列 ⟩)
sreverse(⟨ 文字列 ⟩)
strim(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩)

scopy 函数:　LISP の copy-seq を用いる函数で, 与えられた Maxima の文字列を複製する函数です.

smake 函数: LISP の make-string 函数を用いる函数で, Maxima の文字を指定した整数個並べて新しい文字列を生成する函数です. たとえば, smake(3,"a") から Maxima の文字列 "aaa" が得られます.

charat 函数: 与えられた Maxima の文字列から指定した整数で表現される位置にある文字を取出します. ちなみに文字列の先頭 (左端) が 1 で, 文字列の末尾が slength 函数で得られる値, すなわち, 文字列の文字数になります.
たとえば, charat("explode",2) を実行すると, 与えた文字列 "explode" の先頭から二番目の文字 x が返されます. ちなみに, この函数は LISP の subseq 函数の Maxima への実装になります.

charlist 函数: 与えられた Maxima の文字列を分解してリストデータに変換する函数です. たとえば, charlist("explode") は '[e, x, p, l, o, d, e]' に変換されます. LISP に依存するものの日本語も可能です:

```
(%i53) charlist("explode");
(%o53)                        [e, x, p, l, o, d, e]
(%i54) charlist("爆発");
(%o54)                              [爆, 発]
```

slength 函数: 与えられた Maxima の文字列の長さを返す函数です. この函数の実体は引数を LISP の文字型に変換し, length 函数を作用させているだけです.

parsetoken 函数: 与えられた文字列が数と空行, コンマ",", セミコロン";", タブや改行で構成されている場合に先頭の数値部分を数値に変換する函数です:

```
(%i106) parsetoken("1;234");
(%o106)                               1
(%i107) parsetoken("1,234");
(%o107)                               1
(%i108) parsetoken("1 234");
(%o108)                               1
(%i109) 1/parsetoken("1234");
                                      1
(%o109)                               ----
                                      1234
```

sconc 函数: Maxima-5.14.0 より concat 函数をそのまま利用する函数に改められています. これは Maxima-5.14.0 より Maxima の文字列の扱いが Maxima 独自の内部表現から LISP の文字列に統合されたためです.

sposition 函数: 第 1 引数で指定した Maxima の文字が第 2 引数の文字列上で何処で最初に表われているかを返す函数です:

```
(%i67) sposition("c","cCcC");
(%o67)                                1
```

sreverse 函数:　Maxima の文字列の順番を逆にする函数です. LISP の reverse 函数を Maxima に実装した函数です:

```
(%i1) sreverse("うえからよんでも山本山");
(%o1)                                山本山もでんよらかえう
```

sinsert 函数:　指定した位置に文字列を挿入する函数です:

```
(%i18) sinsert("、","絶景かな絶景かな",5);
(%o18)                               絶景かな、絶景かな
```

strim 函数,striml 函数と strimr 函数:　LISP の string-trim 函数,string-left-trim 函数と string-right-trim 函数を Maxima に実装したものです. 共に第 2 引数 ⟨ 文字列$_2$⟩ から特定の位置にある第 1 引数 ⟨ 文字列$_1$⟩ を除いた文字列を返す函数です.
これらの函数は ⟨ 文字列$_2$⟩ に指定した文字列 ⟨ 文字列$_1$⟩ 複数含まれている場合, 一つだけを削除し, 全ての ⟨ 文字列$_1$⟩ を削除することはしません.
まず, striml 函数は ⟨ 文字列$_1$⟩ から開始する文字列 ⟨ 文字列$_2$⟩ に対し, ⟨ 文字列$_1$⟩ を取り除いた部分を返します. もし, ⟨ 文字列$_2$⟩ が ⟨ 文字列$_1$⟩ から開始していない場合は ⟨ 文字列$_2$⟩ をそのまま返却します.
strimr 函数は逆に ⟨ 文字列$_1$⟩ を末尾に持つ文字列 ⟨ 文字列$_2$⟩ に対し, ⟨ 文字列$_1$⟩ を除いた頭の部分を返却します.
strim 函数はこれらの striml 函数と strimr 函数の両方の働きをします. なお, strim("abc","123abc123") を実行すると, 真中の abc を抜いた文字列を返すのではなく, "123abc123"をそのまま返却します.

substring 函数:　与えられた Maxima の文字列から指定した位置の Maxima の文字列を取り出す函数です. この函数は LISP の subseq 函数の Maxima への実装になります:

substring 函数

substring(⟨ 文字列 ⟩,⟨ 正整数 ⟩)
substring(⟨ 文字列 ⟩,⟨ 正整数$_1$⟩,⟨ 正整数$_2$⟩)

```
(%i2) substring("うえからよんでも山本山",9);
(%o2)                                  山本山
(%i3) substring("うえからよんでも山本山",3,9);
(%o3)                                 からよんでも
```

split 函数

split(⟨ 文字列 ⟩)
split(⟨ 文字列 ⟩,⟨ 分離文字 ⟩)
split(⟨ 文字列 ⟩,⟨ 分離文字 ⟩,false)

split 函数: 与えられた文字列を分解したリストを返す函数です. なお, 切れ目になる文字はデフォルトでは空白文字 (space) ですが, 別途指定できます. また, 末尾に 'false' を指定した場合は区切文字が複数存在するときに空白文字を入れたリストを返します:

```
(%i53) split("first  third  fourth","  ", false);
(%o53)                    [first , , third , fourth]
(%i54) split("first  third  fourth","  ");
(%o54)                     [first , third , fourth]
```

simplode 函数

simplode(⟨ リスト ⟩)
simplode(⟨ リスト ⟩⟨ 文字列 ⟩)

simplode 函数: explode 函数の逆操作の函数で, 与えられたリストから文字列を生成する函数です.
引数が Maxima のリストのみの場合, リストの成分を繋げた文字列を返します. 第 2 引数として Maxima の文字列を指定した場合, リストの成分を繋げる際に第 2 引数の文字列を間に挿入します:

```
(%i53) simplode(["12","34","56"]);
(%o53)                                123456
(%i54) simplode(["12","34","56"],"*-*");
(%o54)                              12*-*34*-*56
```

この例では第 2 引数に何も指定しない場合と第 2 引数を指定した場合の違いを示しています. 第 2 引数を指定しない場合には単純に文字リストの成分を結合するだけですが, 第 2 引数として文字列 "*-*" を指定すると, 第 1 引数のリストの各成分にこの文字列を挿入していることが判るかと思います.

supcase 函数: 与えられた文字列に対して指定された範囲のアルファベットを大文字に変換する函数で, 後述の sdowncase の逆操作になります:

supcase 函数

supcase(⟨ 文字列 ⟩)
supcase(⟨ 文字列 ⟩,⟨ 正整数値 ⟩)
supcase(⟨ 文字列$_1$⟩,⟨ 正整数値$_1$⟩,⟨ 正整数値$_2$⟩)

```
(%i44) supcase("this is a pen.",1,1);
(%o44)                          this is a pen.
(%i45) supcase("this is a pen.",1);
(%o45)                          THIS IS A PEN.
(%i46) supcase("this is a pen.");
(%o46)                          THIS IS A PEN.
(%i47) supcase("this is a pen.",5);
(%o47)                          this IS A PEN.
(%i48) supcase("this is a pen.",1,2);
(%o48)                          This is a pen.
```

sdowncase 函数:　与えられた文字列に対し, 指定された範囲のアルファベットを小文字に変換する函数です. この函数は前述の supcase 函数の逆操作になります:

sdowncase 函数
sdowncase(⟨ 文字列 ⟩)
sdowncase(⟨ 文字列 ⟩,⟨ 正整数値 ⟩)
sdowncase(⟨ 文字列$_1$⟩,⟨ 正整数値$_1$⟩,⟨ 正整数値$_2$⟩)

```
(%i53) sdowncase("此の猫の名前はMIKEです");
(%o53)                          此の猫の名前はmikeです
(%i54) sdowncase("此の猫の名前はMIKEです",9);
(%o54)                          此の猫の名前はMikeです
(%i55) sdowncase("此の猫の名前はMIKEです",9,11);
(%o55)                          此の猫の名前はMikEです
```

sinvertcase 函数:　与えられた文字列に対して指定された範囲でアルファベットの大小を逆に変換する函数です:

sinvertcase 函数
sinvertcase(⟨ 文字列 ⟩)
sinvertcase(⟨ 文字列 ⟩,⟨ 正整数値 ⟩)
sinvertcasee(⟨ 文字列$_1$⟩,⟨ 正整数値$_1$⟩,⟨ 正整数値$_2$⟩)

```
(%i58) sinvertcase("此の猫のnameはMIKEです");
(%o58)                          此の猫のNAMEはmikeです
(%i59) sinvertcase("此の猫のnameはMIKEです",5,6);
(%o59)                          此の猫のNameはMIKEです
```

6.10.6　真理函数を利用する文字列操作の函数

tokens 函数の構文 tokens(⟨ 文字列 ⟩)
tokens(⟨ 文字列 ⟩,⟨ 真理函数 ⟩)

tokens 函数:　[19] の P.61 にある tokens を取り込んだものです. 真理函数は省略可能ですが, この場合は constituent 函数が用いられます. この函数は与えられた文字列を分解して Maxima のリストに変換する函数です. この分解で真理函数を用います. 具体的には真理函数で 'true' になる部分のみが Maxima のリストの成分として返されます:

```
(%i79) tokens("MIKEandTAMAand123");
(%o79)                          [MIKEandTAMAand123]
(%i80) tokens("MIKEandTAMAand123",constituent);
(%o80)                          [MIKEandTAMAand123]
```

```
(%i81) tokens("MIKEandTAMAand123",lowercasep);
(%o81)                                [and, and]
(%i82) tokens("MIKEandTAMAand123",uppercasep);
(%o82)                               [MIKE, TAMA]
(%i83) tokens("MIKEandTAMAand123",digitcharp);
(%o83)                                  [123]
```

この例で真理函数を指定しない場合と constituent 函数を指定したときの結果は同じものになります. また, lowercasep 函数を真理函数として指定した場合, lowercasep 函数が 'true' を返す "and" の部分のみが返却されています. それに対し, uppercase 函数を指定すると大文字の部分のみ, digitcharp 函数を指定した場合には数値の部分のみが返却されています.

ssubstfirst 函数: 文字列に指定した真理函数に適合する部分文字列が存在する場合に部分文字列の入れ替えを行う函数です:

ssubstfirst 函数

ssubstfirst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 文字列$_3$⟩)
ssubstfirst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 文字列$_3$⟩,⟨ 真理函数 ⟩)
ssubstfirst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩,⟨ 正整数値 ⟩)
ssubstfirst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩,⟨ 正整数値$_1$⟩,⟨ 正整数値$_2$⟩)

名前の通り, 真理函数に適合する部分が複数存在する場合には一番最初の部分が取り換えられます. また, 真理函数が未指定の場合は sequal 函数が用いられます. 末尾の正整数値は真理函数による評価を行うための領域指定で用いられます:

```
(%i13) ssubstfirst("三毛","虎","虎猫");
(%o13)                                三毛猫
(%i14) ssubstfirst("三毛","虎","虎は虎猫ではない");
(%o14)                             三毛は虎猫ではない
(%i15) ssubstfirst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequal,5);
(%o15)                         虎は虎猫ではないが、三毛キチでもない
(%i16) ssubstfirst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequal,5,9);
(%o16)                          虎は虎猫ではないが、虎キチでもない
```

なお, 領域指定の整数を引数として渡す場合には必ず真理函数を記述しなければなりません.

ssubst 函数: ssubstfirst 函数に似た函数で, 真理函数に合致する部分を全て入れ替えてしまう函数です:

ssubst 函数の構文

ssubst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 文字列$_3$⟩)
ssubst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 文字列$_3$⟩,⟨ 真理函数 ⟩)
ssubst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩,⟨ 正整数値 ⟩)
ssubst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩,⟨ 正整数値$_1$⟩,⟨ 正整数値$_2$⟩)

この函数で真理函数を未指定の場合は equalignore 函数が用いられます.

```
(%i19) ssubst("三毛","虎","虎猫");
(%o19)                                三毛猫
(%i20) ssubst("三毛","虎","虎は虎猫ではない");
(%o20)                              三毛は三毛猫ではない
(%i21) ssubst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequalignore)
;
(%o21)                          三毛は三毛猫ではないが、三毛キチでもない
(%i22) ssubst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequalignore,5);
(%o22)                           虎は虎猫ではないが、三毛キチでもない
(%i23) ssubst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequalignore,5,9);
(%o23)                            虎は虎猫ではないが、虎キチでもない
```

ssubst 函数でも領域指定の整数を引数として渡す場合には必ず真理函数を記述しなければなりません.

sremovefirst 函数:　文字列に指定した真理函数に適合する部分文字列が存在する場合, その部分文字列を削除する函数です:

sremovefirst 函数

sremovefirst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩)
sremovefirst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩)
sremovefirst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩,⟨ 正整数値 ⟩)
sremovefirst(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 正整数値$_1$⟩,⟨ 正整数値$_2$⟩)

名前の通り, 真理函数に適合する部分が複数存在する場合に一番最初の部分が削除されます. また, 真理函数が未指定の場合は sequal 函数が用いられます. 末尾の正整数値は判定を行うための領域の指定に用いられます:

```
(%i28) sremovefirst("さようなら","さようなら−ああ,さようなら",sequal,5,7);
(%o28)                             さようなら−ああ,さようなら
(%i29) sremovefirst("さようなら","さようなら−ああ,さようなら");
(%o29)                                −ああ,さようなら
(%i30) sremovefirst("さようなら","さようなら−ああ,さようなら",sequal);
(%o30)                                −ああ,さようなら
(%i31) sremovefirst("さようなら","さようなら−ああ,さようなら",sequal,3);
(%o31)                                さようなら−ああ,
(%i32) sremovefirst("さようなら","さようなら−ああ,さようなら",sequal,5,7);
(%o32)                             さようなら−ああ,さようなら
```

なお, 領域指定の整数を引数として渡す場合には必ず真理函数を記述しなければなりません.

sremove 函数:　文字列に指定した真理函数に適合する部分文字列が存在する場合に, その部分文字列を全て削除する函数です:

sremove 函数の構文

sremove(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩)
sremove(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩)
sremove(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩,⟨ 正整数値 ⟩)
sremove(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 正整数値$_1$⟩,⟨ 正整数値$_2$⟩)

名前の通り, 真理函数に適合する部分が複数存在する場合に全ての部分列が削除されます. また, 真理函数が未指定の場合は sequalignore 函数が用いられます. 末尾の正整数値は評価を行うための領域の指定に用いられます;

```
(%i33) sremove("さようなら","さようなら−ああ,さようなら");
(%o33)                                −ああ,
(%i34) sremovefirst("さようなら","さようなら−ああ,さようなら",sequal,3);
(%o34)                                 さようなら−ああ,
(%i35) sremovefirst("さようなら","さようなら−ああ,さようなら",sequal,3,5);
(%o35)                               さようなら−ああ,さようなら
```

なお, 領域指定の整数を引数として渡す場合には必ず真理函数を記述しなければなりません.

ssort 函数: 与えられた文字列を指定した真理函数で大小関係を判別して並び換える函数です. 真理函数が未指定の場合は clessp 函数が用いられます:

ssort 函数の構文

ssort(⟨ 文字列 ⟩)
ssort(⟨ 文字列 ⟩,⟨ 真理函数 ⟩)

```
(%i63) ssort("bkavbhjsaAA");
(%o63)                                AAaabbhjksv
(%i64) ssort("bkavbhjsaAA",cgreaterp);
(%o64)                                vskjhbbaaAA
(%i65) ssort("いろはにほへと",cgreaterp);
(%o65)                                  ろほへはにとい
(%i66) ssort("いろはにほへと",clessp);
(%o66)                                  いとにはへほろ
```

真理函数を clessp 函数から cgreaterp 函数に変更することで結果が逆になることが判ります. なお, 日本語も扱えます.

smismatch 函数: 二つの文字列を指定した真理函数を用いて比較し, 不適合が出る位置を返します. 真理函数が未指定の場合は ssequal 函数が用いられます:

smismatch 函数の構文

smismatch(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩)
smismatch(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩)

```
(%i72) smismatch("桃も李も桃の内","桃も李も藻も桃も");
(%o72)                                5
(%i73) smismatch("1234123","2345123",clessp);
(%o73)                                5
(%i74) smismatch("1234123","2345123",cgreaterp);
(%o74)                                1
```

最初の例では真理函数を未指定のために ssequal が用いられます. そのため, 両者が初めて異なる「藻」の位置が返されています. 次の例では真理函数を clessp にした場合です. この場合 5 番目から大小関係が逆になるので 5 が返されていますが, 最後に cgreaterp を指定すると最初から不適合になるために 1 が返されています.

ssearch 函数: 第 1 引数の文字列が第 2 引数の文字列に含まれる場合, その開始位置を返します. また, 真理函数を用いて第 2 引数の文字列の部分列で第 1 引数の文字列との関係を満す箇所があれば, その開始位置を返します. 真理函数を指定する場合に限って評価を行う際の第 2 引数の文字列の開始位置や終了位置が指定できます. 真理函数が未指定の場合は内部では sequalignore 函数が用いられています.

ssearch 函数の構文

ssearch(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩)
ssearch(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩)
ssearch(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩,⟨ 正整数値 ⟩)
ssearch(⟨ 文字列$_1$⟩,⟨ 文字列$_2$⟩,⟨ 真理函数 ⟩,⟨ 正整数値$_1$⟩,⟨ 正整数値$_2$⟩)

```
(%i1) ssearch("猫","三毛猫と虎猫");
(%o1)                                 3
(%i2) ssearch("A","AAAABCDA",clessp);
(%o2)                                 5
(%i3) ssearch("A","AAAABCDA",clessp,4);
(%o3)                                 5
(%i4) ssearch("A","AAAABCDA",clessp,7,8);
(%o4)                                 7
(%i5) ssearch("A","AAAABCDA",clessp,8);
(%o5)                             false
```

6.10.7　関連する大域変数

stringproc パッケージで定義されている大域変数は実はあまりありません. 基本的に stringprogs で定義されている函数で大域変数を殆ど用いないことが影響していますが, こうなる理由も基本的に LISP にある函数を Maxima 上で実現させることを目的にしたために函数を制御する大域変数がそんなに必要ではなかったことも挙げられるでしょう. そのため, 関連する大域変数としては, 改行, タブや空白文字を表現する newline, tab, space 程度しかありません.

関連する大域変数

newline	改行
tab	タブ
space	空白文字

6.11　構造体

6.11.1　関連する函数と大域変数

Maxima は構造体を持ちます. この構造体は defstruct 函数で定義し, 対象の生成は new 函数を用います.

構造体に関連する函数と演算子

defstruct(⟨ 構造体$_1$⟩, ··· , ⟨ 構造体$_n$⟩)
new(⟨ 構造体名 ⟩)
⟨ 構造体名 ⟩ @⟨ 項目 ⟩

defstruct 函数:　複数の構造体が定義できる函数です. ここで Maxima の構造体は 'mike(x, y, z, ..)' のような函数風の書式を持ちます. そして, この場合, 構造体名は 'mike' であり, 'x', 'y' 等の括弧内部の全ての対象が構造体の項目となります. ここで構造体の項目の正規表現は "⟨ 構造体名 ⟩@⟨ 項目 ⟩" であり, 対象の項目への割当もこの表現に対して行います.

new 函数:　defstruct 函数で定義した構造体の生成では new 函数を用います. ここでの引数は defstruct 函数で定義した構造体名になります.

演算子 "@":　構造体の各項目に値を設定したり, 値の参照を行う場合は演算子 "@" を用います. この演算子 "@" の左被演算子が構造体名, 右被演算子が構造体の項目になります.

大域変数 structures:　defstruct 函数で生成した構造体名は大域変数 structures に割当てられたリストに登録されます:

構造体に関連する大域変数

structures	[]	定義された構造体が格納されたリスト

6.11.2　構造体の例

さて, 簡単な例題で各函数の動作を確認しておきましょう.

```
(%i1) structures;
(%o1)                                  []
(%i2) defstruct(ペット(名前,歳,体重,長さ,性格),本(著者,分野,出版社));
(%o2)              [ペット(名前, 歳, 体重, 長さ, 性格), 本(著者, 分野, 出版社)]
(%i3) structures;
(%o3)              [ペット(名前, 歳, 体重, 長さ, 性格), 本(著者, 分野, 出版社)]
(%i4) ウチのたま:new(ペット);
(%o4)                          ペット(名前, 歳, 体重, 長さ, 性格)
(%i5) ウチのたま@名前:"たま''$
(%i6) ウチのたま@歳:1$
(%i7) ウチのたま@体重:5$
```

```
(%i8) ウチのたま@長さ:20$
(%i9) ウチのたま@性格:"温和。寂しがり屋さん''$
(%i10) ウチのたま;
(%o10)          ペット(名前 = たま, 歳 = 1, 体重 = 5, 長さ = 20, 性格 = 温和。寂し
          がり屋さん)
```

最初に大域変数 structures の値を確認しています. この大域変数は既定値として空リスト "[ ]" が割当てられています. 次に defstruct 函数によって二つの構造体 (ペットと本) を定義しています. このあとに大域変数 structures に defstruct 函数による影響が現われていることに注意して下さい. defstruct 函数で行えることは構造体を定義することで, defstruct 函数で準備された内容を持った対象を生成することは new 函数で行います.
ここではペットを用いるために 'new(ペット)' で生成した構造体を 'ウチのたま' に割当てます.
それから各項目に値を設定しますが, ここでは演算子 "@" を用います. ⟨ 構造体名 ⟩@⟨ 項目 ⟩ で構造体の項目の参照を行い, さらに演算子 ":" を併用することで構造体の項目への割当ができます.

6.12　ラベル

6.12.1　ラベルの概要

利用者の入力値と Maxima の出力はラベルと呼ばれる対象に保存される仕様となっています. この仕組によって利用者は結果を逐次, 紙に書写したりする必要がなく, さらにはラベルの値の参照を行うことで処理の効率化が望めます.
ここで入力ラベルは ‘%i⟨ 整数 ⟩’ の書式を持った対象で, 通常は Maxima のプロンプトの中に現われている対象です. これに対し, 出力ラベルは処理結果の表示行で, 先頭のプロンプトとして現われている対象です. そして, 中間ラベルは ‘%o⟨ 整数 ⟩’ の書式を持った対象ですが, このラベルは一部の Maxima 函数のみが利用します.
次に, 入力ラベル, 出力ラベルと中間ラベルの例を示しておきましょう:

```
(%i17) factor(x^2+2*x+1);
                                        2
(%o17)                                (x + 1)
(%i18) isolate(x^2+2*x+1,x^2+1);
(%o18)                                 %t13
(%i19) %i17;
                                      2
(%o19)                          factor(x  + 2 x + 1)
(%i20) %o17;
                                        2
(%o20)                                (x + 1)
(%i21) %t13;
                                     2
(%o21)                              x  + 2 x + 1
```

まず, 各行の左側に括弧 “( )” で囲まれた記号がラベル名です. このラベルには “%i”, “%o” と “%t” の 3 種類があり, それぞれが入力ラベル, 出力ラベルと中間ラベルに対応します. そして, これらのラベル名を入力すれば, そのラベルに割当てられた値が返却されます. この例では, ラベル%i17 に “(%i17)” で入力した式 ’factor(x^2 + 2 * x + 1)’ が割当てられており, ラベル%o17 には, ‘(%i17)’ での入力式の評価結果の ‘(x + 1)^2’ が割当てられています. それから, ラベル ‘%t13’ は特殊で, ここでの例では isolate 函数が出力した式が割当てられています.
このようにラベルの参照が可能となっているのは, 大域変数 nolabels の値が ‘false’ の場合に入力式と出力式がラベルに束縛され, さらに, これらのラベル名が大域変数 labels に割当てられたリストに自動的に追加される仕組となっているからです. ここで, 大域変数 nolabels の値が ‘true’ であれば, この束縛と大域変数 labels へのラベルの自動追加が実行されないためにラベルの参照が行えなくなります.
出力ラベル “%o” に割当てられた値を参照する函数に %th があります. この %th 函数は ‘%th(6)’ の様に用いることで, 6 個前の結果を参照します. さらに, ‘%i7’ や ‘%o8’ とすれば, ラベル ‘(%i7)’ の付いた行に入力した式や, ラベル ‘(%o8)’ に表示した結果を参照することができます. しかし, ‘_’ は, そのような使い方ができません. さらに最新の結果は ‘%’ に割当てられており, 最新の入力は ‘_’ から参照できます.
入出力ラベルは ‘kill(labels)’ で全て削除できます. これを実行すると, 入力・出力ラベルに割当て

られた全ての値が消去されて, 各ラベルのカウンタも 1 に戻されます, そのために ‘kill(labels)’ の実行後の入力ラベルは ‘(%i1)’, 出力ラベルも ‘(%o1)’ から開始します:

```
(%i101) 1+2;
(%o101)                                3
(%i102) resultant(x-t,y-t^2,t);
                                         2
(%o102)                                y - x
(%i103) algsys([2*x+3*y=1],[x,y]);
                                              2 %r2 - 1
(%o103)                    [[x = %r2, y = - ----------]]
                                                3
(%i104) %;
                                              2 %r2 - 1
(%o104)                    [[x = %r2, y = - ----------]]
                                                3
(%i105) _;
(%o105)                                %
(%i106) %i101;
(%o106)                                3
(%i107) %o101;
(%o107)                                3
(%i108) %i102;
                                                   2
(%o108)                     resultant(x - t, y - t , t)
(%i109) kill(labels);
(%o0)                                done
(%i1)
```

この例では処理結果を%や_で確認し, 最後に kill(labels) でラベル (%i と%o) の内容を全て消去しています. ラベルの消去を行ったために kill(labels) を入力した (%i109) から (%o1) を経て (%i1) に初期化されていることに注目して下さい.

6.12.2 ラベルに関連する大域変数

ラベルに関連する大域変数

変数名	既定値	概要
labels	[]	入出力等のラベル名が登録されたリスト
nolabels	false	入力値と計算結果をラベルへの束縛を制御
%		最新の処理結果
%%		maxima-break の間に処理された最新の値
inchar	%i	入力ラベルで用いる文字を指定
outchar	%o	出力ラベルで用いる文字を指定
linechar	%t	中間表示の際に用いられる文字を指定
linenum		入力番号が設定されている
prompt	_	demo 函数のプロンプトを指定

大域変数 labals:　入出力ラベル等のラベルを成分とするリストが割当てられます. 大域変数 labels に登録されたラベルの値の照合はラベル名を入力することで行えます:

```
(%i9) integrate(sin(x),x);
(%o9)                                  - cos(x)
(%i10) labels;
(%o10) [%i10, %o9, %i9, %o8, %i8, %o7, %i7, %o6, %i6, %o5, %i5, %o4, %i4, %o3,
                                                          %i3, %o2, %i2, %o1, %i1]
(%i11) [%i9,%o9];
(%o11)                    [integrate(sin(x), x), - cos(x)]
```

大域変数 nolabels:　大域変数 nolabels が true の場合に入力値と計算結果はラベルに束縛されません. すなわち, 大域変数 labels に割当てられたリストに入出力ラベルの追加が行われなくなり, %やラベル名の直接入力等で結果や入力の参照が出来なくなります.

大域変数%%と大域変数%:　これらの大域変数は Maxima の最上層では同じ意味を持ちますが, 大域変数%%は大域変数%が使えない block 文内部のみで利用可能な点で異なります:

```
(%i1) block(integrate(x^2,x,0,3),%*3);
(%o1)                                  3 %
(%i2) block(integrate(x^2,x,0,3),%*3);
(%o2)                                  9 %
(%i3) block(integrate(x^2,x,0,3),%%*3);
(%o3)                                  27
(%i4) %%;
(%o4)                                 %%
(%i5) block(integrate(x^2,x),print(%%),%%*3,print(%%),
            diff(%%,x),print(%%));
 3
x
--
3
 3
x
   2
3 x
                                         2
(%o5)                                 3 x
```

この例では最初に block 文内部の積分を実行して大域変数%に保存された式を 3 倍にしますが, block 文内部の大域変数%は block 文を実行する直前の結果が設定されているので, block 文内部で書換えられません. これに対し, 大域変数%%は block 文内部の局所変数のために block 文内部のみで更新され, block 文が終了すると値が消去されます.

大域変数 inchar:　入力行ラベルで用いられる文字です. 入力行ラベルは (%i1) のように大域変数 inchar で指定した%i の後に番号が続きます.

大域変数 outchar:　出力ラベルで用いられる文字です. 大域変数 inchar と同じ使い方をします.

大域変数 linechar: 中間表示での式の前に置かれる文字を指定します.

大域変数 linenum: その時点での入力行番号が割当てられています:

```
(%i17) linenum;
(%o17)                                  17
```

大域変数 prompt: demo 函数のプロンプト記号を指定する大域変数です. この大域変数の値は playback 函数や break 函数で Maxima-break に入った時点で表示されます:

```
(%i1) integrate(x^2-1,x);
                                     3
                                    x  - 3 x
(%o1)                               --------
                                       3
(%i2) prompt:"(;_;)";
(%o2)                                 (;_;)
(%i3) playback(slow);

(%i1) integrate(%,x);
(;_;)
                                     3
                                    x  - 3 x
(%o1)                               --------
                                       3
(;_;)

(%i2) prompt:"(\;_\;)";
(;_;)

(%o2)                                 (;_;)
(%o3)                                 done
(%i4) break(x-1);
x - 1

Entering a Maxima break point. Type exit; to resume
(;_;) exit;
(%o4)                                  x - 1
```

この例では大域変数 prompt に Maxima の文字列 "(;_;)" を指定しています. ここで 'playback(slow)' を実行すると過去の入力と出力を一纏めにして出力し, 大域変数 prompt を出力して Enter キー の入力待ちとなります.

6.12.3 ラベル処理の函数

Maxima にはラベル処理の函数として, %th 函数と labels 函数を持っています. これらの函数は共に大域変数 labels を用いて処理を行う函数です:

ラベル処理に関連する函数

%th(⟨ 正整数 ⟩)
labels(⟨ シンボル ⟩)

%th 函数:　大域変数 labels に登録された出力ラベルに対して ⟨ 正整数 ⟩ 番目の出力ラベルに束縛された値を返します. 大域変数 nolabels の既定値が 'false' なので通常は ⟨ 正整数 ⟩ 番前の計算結果になります. 実際, %th⟨*i*⟩ を含む式の入力ラベルが%⟨*j*⟩ であれば, %th⟨*i*⟩ は%i⟨*j* − *i*⟩ の結果, すなわち, %o⟨*j* − *i*⟩ の値になります.
この%th 函数は batch ファイルの利用では非常に便利です. これは%o-ラベルの値が batch ファイルをどの時点で処理するかで異なるのに対し, %th 函数はその函数を実行する時点を中心として指定した整数程前の結果を返すからです.

labels 函数:　記号を引数として取り, 大域変数 labels に割当てられたリストを構成する成分と引数を照合し, 適合する成分で構成されたリストを返します. なお, ここでの照合は引数の最初のアルファベットと大域変数 labels に登録されたラベルの最初のアルファベットが一致するかどうかで行います. したがって, labels(%i1) でも labels(i) でも labels(imax) でも引数の最初のアルファベット 'i' で照合を行うために結果として返されるリストは全ての入力ラベルで構成されたものになります. そして, 適合するものがなければ空リスト "[ ]" を返却します.
ここで入出力等のラベルを設定する大域変数 inchar, 大域変数 outchar や大域変数 linechar を再設定すれば大域変数 labels に登録されるラベルが切替えられます. そして, labels 函数の引数もそれに合せて与えると, その引数に対応するラベルのリストを返します.

6.13 Maxima の対象

6.13.1 Maxima の対象とその実体

Maxima の対象は記号, すなわち, 名前であり, この名前に対してさまざまな属性, 属性値が付与されるだけではなく, 変数の場合はその値, 函数であれば函数の実体といったものが与えられます.
そして, 実体を持つ対象はその属性に応じて名前が大域変数に登録され, これによって実体と対象名の関連が保持されます. このような対象は対象に付与される属性から大域変数 infolists に含まれる大域変数に登録されます.
そして, 実体を削除する必要が生じた場合, 関連する大域変数からその名前を削除することで対象名と実体の関連を断つこととなり, これによって対象名の束縛が解消されます. この削除は kill 函数で行われますが, 対象によっては専用の削除用函数を持っているものもあります.
まず, 大域変数 infolists に登録されている大域変数を次に示しておきます:

対象と関連する大域変数

大域変数	概要
aliases	利用者定義の変名リスト
arrays	利用者定義の配列リスト
dependencies	利用者定義の依存性リスト
functions	利用者定義函数と演算子のリスト
gradefs	利用者定義の勾配を持つ函数のリスト
labels	ラベルのリスト
let_rule_packages	let 函数で定めた規則パッケージのリスト
macros	利用者定義マクロのリスト
myoptions	利用者設定のオプションのリスト
props	属性のリスト
rules	利用者定義の規則のリスト
structures	構造体のリスト
values	利用者定義変数のリスト

大域変数 aliases : 大域変数 alias には利用者が定めた対象の別名が登録されたリストが割当てられています. 具体的には alias 函数によって指定された対象の別名, ordergreat 函数, orderless 函数の各引数, および, declare 函数で名詞型属性を付与された対象が登録されます.

大域変数 arrays : 利用者が生成した配列と配列函数の名前が登録されたリストが割り当てられています.

大域変数 dependencies : depends 函数と gradef 函数によって dependencies 属性を付与された対象の名前が登録されたリストが割当てられる大域変数です.

大域変数 functions :　演算子 ":=" や define 函数で定義された利用者定義函数や演算子の名前が登録されたリストが割当てられる大域変数です.
大域変数 macros とは排他処理が実行される大域変数です.

大域変数 gradefs :　gradef 函数で gradef 属性を付与された対象の名前が登録されたリストです.

大域変数 labelsl :　値が割当てられている入出力や中間出力等の Maxima のラベルの名前を成分とするリストが割当てられている大域変数です. このラベルは大域変数 nolabels の値が 'false' の場合に自動的に追加されますが, 値が 'true' であれば対象の自動追加は行なわれません.

大域変数 let_rule_packages :　let 函数を用いて定義された規則名が登録されたリストが割当てられます. 既定値は '[default_let_rule_packages]' です. この理由は, 利用者が明示的にパッケージを指定しなければ大域変数 default_rule_packages に規則が収納されるためです.

大域変数 macros :　演算子 "::=" を使って実体を定義したマクロの名前で構成されたリストが割当てられた大域変数です. 大域変数 functions とは排他処理が行われます.

大域変数 myoptions :　利用者によって再設定された全ての大域変数名が登録されたリストが割当てられています. 初期値は空リスト "[]" です. ここで登録される大域変数はここで解説している対象が自動的に登録されるリストではない, 函数の挙動に影響を与える大域変数です. なお, 一度変更した大域変数を初期値に戻しても, 大域変数 myoption からは削除されません. この大域変数 myoptions の値は kill 函数でも削除できません:

```
(%i2) nolabels:true;
(%o2)                                true
(%i3) display2d:false;
(%o3) false
(%i4) myoptions;
(%o4) [nolabels,display2d]
(%i5) display2d:true;
(%o5)                                true
(%i6) myoptions;
(%o6)                        [nolabels, display2d]
```

大域変数 props :　atvalue 函数, matchdeclare 函数や declare 函数等によって属性を付与された記号が登録されたリストが割当てられています.

大域変数 rules :　利用者定義の並び照合と簡易化の規則で, tellsimp 函数, tellsimpafter 函数, defmatch 函数や defrule 函数で設定された規則名を登録したリストが割当てられています.

大域変数 structures:　defstruct 函数で定義される構造体名が登録されるリストです.

大域変数 values :　利用者定義の束縛変数のリストが割当てられています.

6.13.2 対象の削除

大域変数 infolists を構成する大域変数に割当てられたリストに登録された対象は全て実体を持つ対象です. この実体を削除する方法は, 各対象専用の函数を利用するか, kill 函数を用いて対応する大域変数から削除する方法になります.
たとえば, 式 'x:10' によって束縛変数にされた変数 x を自由変数にするためには, 束縛変数が登録される大域変数 values から変数 x を削除することで行えます. この処理は論理式 '$\iota_x(x = 10)$' を Maxima から削除することに対応します.

kill 函数の構文

kill(⟨ 引数$_1$⟩,⟨ 引数$_2$⟩,...)
kill(⟨ 整数 ⟩)
kill([⟨ 整数$_1$⟩,⟨ 整数$_2$⟩])
kill(all)
kill(allbut(⟨ 引数$_1$⟩,⟨ 引数$_2$⟩,...))

kill 函数: この kill 函数は引数によって削除する対象が異なります:

- 引数が変数, 配列, 函数の場合, 指定された対象は属性を含めて全てが消去されます.
- 引数が大域変数 infolists に含まれる大域変数の場合, その大域変数に含まれる対象とその属性が全てが削除されます.
- 引数が 'tellrats' の場合, 内部変数 tellratslist の値が 'nil' に戻されます.
- 引数が rateweighs の場合, 内部変数*ratweights の値が 'nil', 大域変数 ratweights が空リストに設定されます.
- 引数が feature の場合, 大域変数 features に登録された対象で, 内部変数 feature の属さないものが削除されます.
- 引数が 'all' の場合, 全てが削除されます.
- 引数が allbut 項の場合, allbut で指定した対象を除外して, 引数が 'all' の場合と同様に削除が行われます.

変数のみを指定して式を削除してもラベルの削除を行わない限り, 記憶容量の全体的な占有領域の解放が行われないことに注意が必要です. たとえば, 大きな式が '%i10' で変数 x に割当てた場合, 占有された保存領域を解放するためには 'kill(x)' と 'kill(%o10)' の両方を実行する必要があります. kill 函数は与えられた引数から全ての属性を削除します. したがって, 'kill(values)' は大域変数 values に割当てられたリストの全ての対象に関連する属性を削除しますが, remvalue 函数, remfunction 函数, remarray 函数や remrule 函数は特定の属性のみを削除します. これらの函数群はリスト名か指定した引数が存在しなければ 'false' を出力しますが, これに対して kill 函数は指定した対象が存在しない場合でも常に 'done' を出力します.

第7章　式の操作

Faust
Faust,Faust,nun erfüllt sich dein Augenblick!

Die Zaubermacht in meine Hand gegeben,
die ungeheuren Zeichen mir erschlossen,
heimliche Gewalten mir geknechtet,
und ich kann -ja,ich kann -o,ihr Menschen,die
ihr mich gepeinigt, hütet euch vor Faust!

ファウスト
ファウストよ, ファウストよ, ついにお前の望が叶ったぞ!
魔力が我が手に与えられ,
おぞましい象徴が明らさまとなり, 秘密の威力が我が物となった,
そして, 俺には出来る, そうだ, 俺には出来る, おお, 人間共よ, 俺を苦しめたお前達よ, ファウストを恐れるがよい!

Buzoni,Doktor Faust[101] より

この章では Maxima の式の操作に関連した事柄について解説を行います. ここで取り上げる操作は, 式への代入, 簡易化, 代数方程式の処理, 極限, 微分積分と常微分方程式の処理です.

7.1　代入操作

方程式を求めた結果を早速, 式に代入したいことが多くあります. この場合は ev 函数による評価 (§5.8.3 参照) が便利ですが, 部分式をそのまま入れ換えたり, 演算子項の演算子や函数項の函数を取り換えたくなることもあります. この場合は通常の割当や評価は使えません. そこで代入函数を使うことで Maxima の式を構成する対象の入れ換えが容易に行えます. この代入函数には単純に指定した式や項を別の式や項で置換える通常の代入を行う函数と, 与式の内部構造を利用して式や項, さらには演算子や函数といったものさえも入れ換える代入函数の二種類があります. 前者の函数は, 与式の内部表現を考慮せずに行える割当に対応し, 後者の函数は部分式を取出す part 函数や inpart 函数に対応すると言えるでしょう.

7.1.1　通常の代入函数

最初に与式の内部構造を考慮せずに代入操作が行える函数を纏めておきます:

通常の代入に関連する函数の構文

subst(⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨ 式$_3$⟩)
ratsubst (⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨ 式$_3$⟩)
fullratsubst(⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨ 式$_3$⟩)
lratsubst(⟨ リスト ⟩,⟨ 多項式 ⟩)
sublis(⟨ リスト ⟩,⟨ 式 ⟩)

subst 函数:　⟨ 式$_3$⟩ 中の全ての ⟨ 式$_2$⟩ を ⟨ 式$_1$⟩ で置換します.
⟨ 式$_1$⟩ と ⟨ 式$_2$⟩ は二重引用符で括られた式の演算子や函数名, あるいは, ⟨ 式$_2$⟩ を Maxima の記号や ⟨ 式$_3$⟩ に完全に含まれる部分式とします.
たとえば, 式 'x+y+z' は式 '2*(x+y+z)*w' に完全に含まれる部分式になりますが, 式 'x+y' は部分式になりません. これは式の木構造を考えると明瞭になります. 部分式は式の木構造を考えたときに各階層が構成する式を取出したものになるからです.
もし, ⟨ 式$_2$⟩ が ⟨ 式$_3$⟩ の部分式でなければ, subst ではなく式の階層を直接指定出来る substpart 函数や ratsubst 函数を使いましょう.
subst 函数では ⟨ 式$_2$⟩ が 式$_a$/式$_b$ のように割算を伴う場合, 'subst(⟨ 式$_1$⟩ * ⟨ 式$_b$⟩,⟨ 式$_a$⟩,⟨ 式$_3$⟩)' が使えます. また, ⟨ 式$_2$⟩ が ⟨ 式$_a$⟩$^{1/⟨\ 式_b\ ⟩}$ の書式であれば, 'subst(⟨ 式$_1$⟩^⟨ 式$_b$⟩,⟨ 式$_a$⟩,⟨ 式$_3$⟩)' が使えます.

subst 函数に影響を与える大域変数

変数名	既定値	概要
exptsust	false	指数函数の代入操作を制御
opsubst	true	演算子に代入する事を抑制

大域変数 exptsubst:　'true' であれば式 '%e^(a*x)' の '%e^x' を変数 y で置換える操作が可能になります.

大域変数 opsubst: 'false' であれば subst 函数は式に含まれる演算子に対して代入を行いません. たとえば, '(opsust:true,subst(x^2,r,r+r[0]))' と '(opsubst:false,subst(x^2,r,r+r[0]))' を実行するとき, 大域変数 opsusbt の値が 'true' であれば, subst 函数は与式 'r+r[0]' の全ての変数 r に 'x^2' を代入しますが, 大域変数 opsubst が 'false' であれば, 左側の変数 r のみに代入操作が行われ, 項 'r[0]' の変数 r には式 'x^2' が代入されません:

```
(%i63) (opsubst:true,subst(x^2,r,r+r[0]));
                                        2       2
(%o63)                                 x  + (x )
                                                 0
(%i64) (opsubst:false,subst(x^2,r,r+r[0]));
                                          2
(%o64)                                   x  + r
                                                0
```

ratsubst 函数: 〈式$_3$〉に含まれる〈式$_2$〉に〈式$_1$〉を代入します. 〈式$_2$〉は和, 積, 冪等の演算子でも構いません. subst 函数が代入を行う個所で ratsubst 函数は式が何を意味するかを知っています. そのために式 'subst(a,x+y,x+y+z)' は 'x+y+z' を返しますが, ratsubst 函数は 'z+a' を返します.

ratsubst に影響を与える大域変数

変数名	既定値	概要
radsubstflag	false	冪乗の扱いを制御

大域変数 radsubstflag: 大域変数 radsubstflag(ratsubstflag ではないことに注意) の値が 'true' の場合, ratsubst 函数を使って項の "sqrt" や冪で式の入れ換えが可能となります.
たとえば, 変数 a の値を 'x^(1/3)' とした場合, 変数 x は式 'a^3' に等しくなりますが, だからといって, 式 'x^(1/3)' の変数 x をいきなり a の項で置換えることは通常できません. 大域変数 radsubstflag の値が 'true' の場合, このような代入が行えます:

```
(%i5) radsubstflag:true$
(%i6) ratsubst(a,x^(1/3),x);
                                        3
(%o6)                                  a
```

fullratsubst 函数: ratsubst 函数と同じですが, 結果が変化しなくなるまで自分自身を再帰的に呼出します. この函数は式の置き換えや置き換えられた式が一つ, またはそれ以上の変数を共通に持つ場合に便利です. fullratsubst 函数は lratsubst 函数と同じ引数の書き方ができます. 第 1 引数は単一の代入方程式かそのような方程式のリストで, 第 2 引数は仮定された式となります.

lratsubst 函数: 'subst(〈方程式のリスト〉,〈式〉)' に似ていますが, rstsubst 函数が subst 函数の代りに使われる点で異なります. lratsubst 函数の最初の因子は方程式か方程式のリストで, subst 函数から得られる書式と同一のものでなければなりません. 代入は方程式のリストの左から右の順で処理します:

```
(%i1) load ("lrats")$
(%i2) subst ([a = b, c = d], a + c);
(%o2)                              d + b
(%i3) lratsubst([a^2=b,b=c^2,c^3=d], a^2+b+c^3);
                                                 2
(%o3)                                     d + 2 c
(%i4) subst([b=c^2,a-2=b,c^3=d], a^2+b+c^3);
                                                2    2
(%o4)                                     d + c  + a
(%i4) lratsubst([b=c^2,a-2=b,c^3=d], a^2+b+c^3);
                                               2    2
(%o4)                                 d + c  + b  + 4 b + 4
```

sublis 函数: 〈式〉に〈リスト〉で指定した複数の代入を並行して行ないます. 〈リスト〉には, 'a = b' の書式で式を記述します. 演算子 "=" のん左辺の a に〈式〉に含まれる原子や函数名を指定し, 右辺の b に置換える値や式を設定します:

```
(%i23) sublis([sin=cos,x=2*theta+1],sin(x-1)^2);
                                              2
(%o23)                                  cos (2 theta)
(%i24) sublis([sin=cos,cos=sin],cos(x)^2+sin(x+1)^3);
                                            3                2
(%o24)                                cos (x + 1) + sin (x)
```

なお, 'sublis([sin=cos,cos=sin],cos(x)^2+sin(x+1)^3)' のような入れ換えの指定では〈リスト〉に含まれる式の代入を順番に行うのではなく同時に行うため, "cos" と "sin" が入れ換えられていることに注意して下さい.

大域変数 sublis_apply_lambda: sublis 函数を実行したあとの簡易化を制御します.

7.1.2 式の内部構造を考慮した代入函数

Maxima の代入には与式の部分式を取出す part 函数と 函数に対応する代入函数の substpart 函数と substinpart 函数があります.
ここで part 函数は式の木構造に対して部分式を取出す函数で, inpart 函数は Maxima の式の内部表現に対して部分式を取出す函数です. substpart 函数と substinpart 函数も同様で, 大域変数 inflag を true に設定して part/substpart 函数を呼出すことは inpart/substinpart 函数を呼出すことと同値です.

substpart と substinpart 函数の構文

substpart(〈式$_1$〉,〈式$_2$〉,〈正整数$_1$〉,...,〈正整数$_n$〉)
substinpart(〈式$_1$〉,〈式$_2$〉,〈正整数〉,...)

substpart 函数: ⟨ 式$_2$⟩ から part 函数のように ⟨ 正整数$_1$⟩, . . . , ⟨ 正整数$_n$⟩ でで抜出した部分式に ⟨ 式$_1$⟨ を代入します. ⟨ 式$_1$⟩ に演算子を入れる場合には二重引用符で 'substpart("+",a*b,0)' の中の演算子 "+" の扱いのように括る必要があります. これは Maxima の演算子名は二重引用符で括って文字列として表現するためです.
具体的な例で説明しましょう. 数式 $\frac{1}{x^3+3x^2+1}$ に対応する Maxima の式 '1/(x^3+3*x^2+1' の成分の入れ換えを行いましょう. この式の構造は図 7.1 に示すものになります:

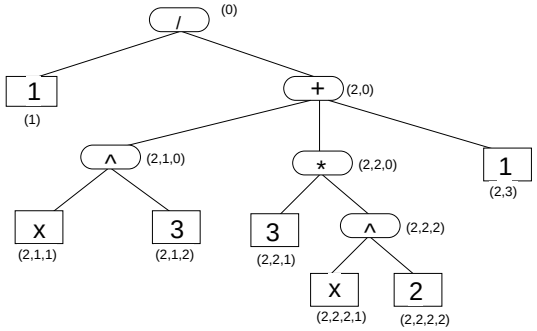


図 7.1: $\frac{1}{x^3+3x^2+1}$ の構造

substpart 函数はこの構造に従って入れ換えを行います. そのために演算子や式で成分を置換えられます:

```
(%i7) 1/(x^3+3*x^2+1);
                                        1
(%o7)                              ------------
                                    3       2
                                   x  + 3 x  + 1
(%i8) substpart(4,%,2,1,2);
                                        1
(%o8)                              ------------
                                    4       2
                                   x  + 3 x  + 1
(%i9) substpart(1,%,2,2,2,2);
                                       1
(%o9)                              -----------
                                    4
                                   x  + 3 x + 1
(%i10) substpart(x,%,1);
                                       x
(%o10)                             -----------
                                    4
                                   x  + 3 x + 1
(%i11) substpart("^",%,0);
                                     4
                                    x  + 3 x + 1
(%o11)                             x
```

```
(%i12) substpart(sin(x),%,1);
                                        4
                                       x  + 3 x + 1
(%o12)                          sin(x)
(%i13) substpart(y,%,2);
                                         y
(%o13)                                  sin (x)
```

このように substpart 函数を用いると, さまざまな処理が容易に行えます. 特にリストの処理が非常に容易になります. この例として, 与えられた方程式が実数解のみを持つかどうかを検証する例を挙げておきます:

```
(%i19) solve([x^4-2*x^3-x+2],[x]);
                                  sqrt(3) %i + 1        sqrt(3) %i - 1
(%o19)      [x = 1, x = 2, x = - --------------, x = --------------]
                                        2                   2
(%i20) map(lambda([x],is(equal(imagpart(rhs(x)),0))),%);
(%o20)                        [true, true, false, false]
(%i21) substpart("and",%,0);
(%o21)                                false
```

この例の lambda 式は演算子 “=” の右辺を取出し, 虚部が ‘0’ であれば ‘true’, それ以外を ‘false’ を返します. この lambda 式を map 函数でリスト作用させることで真理値のリストが得られますが, このリストの全ての成分の論理積を取れば, リストが実数解のみかどうかが判ります.
なお, substinpart 函数は substpart 函数に似ていますが, substpart 函数と違って式の内部表現に対して作用します.

7.2 式の展開と簡易化

7.2.1 自動展開を行う大域変数

Maxima は式を入力すると同時に簡易化が行われます. これは入力式の簡易化を受け持つ内部函数 simplifya が動作するためです. この内部函数 simplifya は大域変数 simp で制御されており, この大域変数の値が 'true' の場合に内部函数 simplifya は式に含まれる函数の operators 属性や演算が持つ属性に対応する内部函数を用いて簡易化と評価が実行される仕組になっています (§5.8.2 参照). ここで大域変数 simp の値を 'false' に設定すると内部函数 simplifya による入力式の簡易化を停止するので, 入力式が自動的に簡易化されることはありません.

```
(%i1) x^2+2-x^2*2+4+4/2;
                                      2
(%o1)                               8 - x
(%i2) simp;
(%o2)                                true
(%i3) simp:false$
(%i4) x^2+2-x^2*2+4+4/2;
                          2          2          4
(%o4)                    x  + 2 + (- x ) 2 + 4 + -
                                                2
```

このような式の簡易化に関連する大域変数が数多くあります. ここでは最初に自動評価に関連する大域変数を纏めておきましょう.

属性を持たない自動化に関連する大域変数

指数表示に関連する大域変数

変数名	既定値	概要
demoivre	false	指数函数の表示を制御
%emode	true	指数函数の簡易化を制御
%enumer	false	Napia 数%e の浮動小数点数への自動変換

大域変数 demoivre: 'true' であれば与式 '%e^(a+b*%i)' の変数 b が実数であれば, '%e^a*(cos(b)+%i*sin(b))' と展開します:

```
(%i18) exp(a+b*%i);
                                     %i b + a
(%o18)                              %e
(%i19) demoivre:true$
(%i20) exp(a+b*%i);
                                a
(%o20)                         %e  (%i sin(b) + cos(b))
```

大域変数%emode:　'true' であれば, 与式 '%e^(%pi*%i*x)' が次のように簡易化されます:

- 変数 x が整数, あるいは '1/2', '1/3', '1/4' や '1/6', あるいは整数の積であれば, 'cos(%pi*x)+%i*sin(%pi*x)' となります.
- その他の数値の場合, '%e^(%pi*%i*y)' となります. ここで変数 y は 'x-2*k', 変数 k は 'abs(y)<1' を満す整数です.

なお, 大域変数%emode が 'false' の場合, 式 '%e^(%pi*%i*x)' の簡易化は何も実行されません:

```
(%i25) %emode:true$
(%i26) exp(%pi*%i/2);
(%o26)                                %i
(%i27) %emode:false$
(%i28) exp(%pi*%i/2);
                                     %i %pi
                                     ------
                                       2
(%o28)                              %e
```

大域変数%enumer:　'true' であれば式中の項 '%e' は 2.718... に変換されます. なお, 式中の項 '%e^x' は指数が整数の場合に限り, この変換が実行されます.

属性を持たない自動展開に関連する大域変数

変数名	既定値	概要
negdistrib	true	$-(A+B+\cdots+Z)$ を $-A-B\cdots-Z$ に-1 を自動的に分配
numer	false	数値を含む式の自動評価を実行
simp	true	入力式の数値部分の自動的簡易化を実行
sumexpand	false	sum 函数の簡易化を実行
simpproduct	false	product 函数の簡易化を制御

大域変数 negdistrib:　この大域変数は '-1' の積を分配するかどうかを決定します. 'true' の場合には '-1' の積が分配されます:

```
(%i1) negdistrib;
(%o1)                                 true
(%i2) -(a+b+c-d);
(%o2)                              d - c - b - a
(%i3) negdistrib:false$
(%i4) -(a+b+c-d);
(%o4)                          - (- d + c + b + a)
```

大域変数 numer:　入力された式の数値を自動的に浮動小数点数に変換させる大域変数です. この大域変数は後述の大域変数 float に影響を及ぼしますが, 大域変数 float の設定の影響は受けません.

大域変数 simp: ‘true’ の場合に内部函数 simplifya を用いて和や差の演算子の簡易化を自動実行します.

大域変数 sumexpand と大域変数 prodexpand: これらの大域変数の詳細に関しては,§6.4 の sum 函数と product 函数を参照して下さい.

属性を持つ自動化に関連する大域変数

自動簡易化に関連する大域変数の中で属性を持つ函数があります. これらの函数は ev 函数による解釈が関連します:

属性を持つ自動評価に関連する大域変数

変数名	既定値	属性	概要
float	false	evflag	true の場合は非整数, および, 非整数を含む式を自動的に浮動小数点数へ変換する
expop	0	fixnum	正の冪乗の自動展開の上限を定める.
expon	0	fixnum	負の冪乗の自動展開の下限を定める.

大域変数 float 大域変数 numer に似た働きをする大域変数です. ‘true’ の場合に Maxima は整数以外の数値を自動的に浮動小数点数に変換します.

大域変数 float 大域変数 numer の影響を受ける大域変数です. 実際, ‘numer:true’ とすることで自動的に ‘float:true’ とされ, 同様に ‘muner:false’ と設定すれば大域変数 float も ‘false’ に設定されます. しかし, 大域変数 float の設定は大域変数 numer に影響を及ぼさず, 影響は大域変数 numer から大域変数 float への一方だけです.

大域変数 expon と大域変数 expop 大域変数 expon は expand 函数とは別個に Maxima が自動的に展開する式に含まれる負の冪の次数を定めます. 同様に大域変数 expop は自動的に展開される正の最高次数を定めます.
大域変数 expon と expop は既定値が ‘0’ に設定されているために, 式 ‘(x+1)^0’ の様に冪の次数が零であれば自動的に ‘1’ に変換されます. 大域変数 expop を例えば 4 に変更すると, 冪の次数が 0 以上, 4 以下であれば Maxima は冪を自動的に展開します. また, expon を 4 にすると負の冪の次数の絶対値が 0 以上, 4 以下であれば自動的に冪を展開します. 以下に簡単な例を示しましょう:

```
(%i38) expon:4$
(%i39) (x+1)^(-3);
                                  1
(%o39)                    ------------------
                           3       2
                          x  + 3 x  + 3 x + 1
(%i40) (x+1)^(-5);
```

```
                                        1
(%o40)                              --------
                                          5
                                    (x + 1)

(%i41) expop:4$
(%i42) (x+1)^4;
                                 4       3       2
(%o42)                          x  + 4 x  + 6 x  + 4 x + 1
(%i43) (x+1)^5;
                                          5
(%o43)                                (x + 1)
```

7.2.2　指数函数の展開に関連する函数

指数函数の表示

demoivre(⟨ 式 ⟩)
exponentialize(⟨ 式 ⟩)

demoivre 函数　大域変数 demoivre の設定や ev 函数による式の再評価なしで変換を行います.

exponentialize 函数　引数 ⟨ 式 ⟩ を指数函数形式に変換します:

```
(%i3) demoivre(exp(x+%i*y));
                                  x
(%o3)                          %e  (%i sin(y) + cos(y))
(%i4) exponentialize(%);
                              %i y      - %i y      %i y      - %i y
                         x  %e      + %e          %e      - %e
(%o4)                  %e  (----------------- + -----------------)
                                  2                     2
```

7.2.3　式の展開に関連する函数

expand 函数の構文

expand(⟨ 式 ⟩,⟨p⟩,⟨n⟩)
expand(⟨ 式 ⟩)
expan(⟨ 式 ⟩,p,n)
expandwrt(⟨ 式 ⟩,⟨ 変数$_1$⟩,$\cdots$,⟨ 変数$_n$⟩)
expandwrt_factored(⟨ 式 ⟩,⟨ 変数$_1$⟩,$\cdots$,⟨ 変数$_n$⟩)
pfet(⟨ 式 ⟩)

expand 函数 expand 函数は和の積や指数函数内の和の展開, 有理式の分子を項に分離, 可換積と非可換積の両方の積を ⟨ 式 ⟩ の全ての階層で加法に対して分配します.
なお, 多項式に対してはより効率的なアルゴリズムを用いる ratexpand 函数を通常用いるべきです. 大域変数の maxnegex と大域変数 maxposex は Maxima が展開する式の負と正の冪の次数の最大値を設定します.

expand 函数を制御する大域変数

変数名	既定値	属性	概要
maxnegex	1000	fixnum	expand で展開される負の冪の次数
maxpogex	1000	fixnum	expand で展開される正の冪の次数

大域変数 maxnegex は expand 函数で展開される絶対値が最大となる負の冪の次数です. 正の冪の最大次数は大域変数 maxposex に設定されています.
大域変数 maxposex は expand 函数で展開される最大の正の冪の次数です. なお, 負の冪の最大次数は大域変数 maxnegex に設定されています.

expan 函数 この函数は式 'expan(⟨ 式 ⟩,p,n)' に対して p を大域変数 maxposex, n を大域変数 maxnegex に割当て, ⟨ 式 ⟩ の展開を行います.

expandwrt 函数 この函数は ⟨ 変数$_1$⟩ ,···,⟨ 変数$_n$⟩ に対して ⟨ 式 ⟩ を展開します. ⟨ 変数$_i$⟩ を含む全ての積は明示的に現れます. 返される形式は ⟨ 変数$_i$⟩ を持つ式の和の積を持たないものとなります. ⟨ 変数$_i$⟩ は変数, 演算子や式でも構いません.
デフォルトで分母は展開されませんが, 大域変数の expandwrt_denom で制御できます. この函数を使うためには予め load(stopex); で読込を実行しておく必要があります.

expandwrt_factored 函数 この函数は expndwrt 函数に似ていますが, 幾分違った式の積を扱います. この expand_factored 函数は要求される展開を処理しますが, 引数リストの中の変数に含まれる ⟨ 式 ⟩ の因子に対してのみ処理を行います. 予め, load(stopex) で読込を実行する必要があります.

pfet 函数 expand 函数に似た函数で式の展開を行います. なお, pfet 函数では ratexpand 函数と内部函数の sssqfr 函数が用いられています. この函数の特徴は主変数に対して式の展開を行うことです. したがって, expand 函数のように式が完全に展開されたものにはなりません:

```
(%i61) pfet((x+y+6)^2+6+z);
                              2                    2
(%o61)                  z + y  + 2 (x + 6) y + x  + 12 x + 42
(%i62) pfet(((x+y)*z+6)^2+6+z);
                                2  2
(%o62)                    (y + x)  z  + 12 (y + x) z + z + 42
(%i63) pfet(((x+z)*x+6)^2+6+z);
                   2  2          2                  4        2
(%o63)            x  z  + 2 x (x  + 6) z + z + x  + 12 x  + 42
```

この例で, 最初の式の主変数は Maxima の項順序 “$>_m$” により変数 y となります. そのために変数 y に対しては展開されますが, その係数は展開されません. あとの二つの例では主変数が変数 z となるために変数 z に対してのみ展開が行われます.

7.2.4　演算子の分配に関連する函数

演算子の分配に関連する函数

distrib(⟨ 式 ⟩)
multthru(⟨ 式$_1$⟩ ,⟨ 式$_2$⟩)
multthru(⟨ 式 ⟩)

distrib 函数　この函数は可換積演算子 “*” に対し和 “+” を分配します. expand 函数との違いは, distrib 函数が式の最上層のみで作用する点です. また, multthru 函数とも最上層の全ての和を展開する点でも異なります.

multthru 函数　⟨ 式 ⟩ の部分展開を行います. すなわち, ⟨ 式 ⟩ が $f_1 * f_2 * \cdots * f_n$ の形式で, 各因子の中で, 冪乗でない ⟨ 式 ⟩ 中で最も左側の因子を f_i とすると, ⟨ 式 ⟩ の f_i 以外の因子と f_i の項との積の和に分解します. たとえば, $(x+1)^2 \cdot (z+1) \cdot (y+1)$ の場合, 最も左側の因子 $y+1$ で式が展開され, $(x+1)^2 \cdot (y+1) \cdot z + (x+1)^2 \cdot (y+1)$ となります.
‘multthru(⟨ 式$_1$⟩ ,⟨ 式$_2$⟩)’ の場合, ⟨ 式$_2$⟩ の各項を ⟨ 式$_1$⟩ 倍にします. つまり, ‘multthru(⟨ 式$_1$⟩ *⟨ 式$_2$⟩)’ と同値です.
なお, ⟨ 式$_2$⟩ には方程式を指定できます. この場合, 演算子 “=” の二つの被演算子に ⟨ 式$_1$⟩ との積が返されます.
この multthru は冪乗された和の展開は行いません. この函数は和に対する可換, あるいは, 非可換積の分配に関して最も速いものです:

```
(%i18) multthru((x+1)^2*(z+1));
                                    2            2
(%o18)                        (x + 1)  z + (x + 1)
(%i19) multthru((x+1)^2*(y+1)^2*(z+1)^2,z+1);
                  2        2          2          2        2        2
(%o19)      (x + 1)  (y + 1)  z (z + 1)  + (x + 1)  (y + 1)  (z + 1)
(%i20) multthru((x+1)^2*(y+1)^2*(z+1)^2,x+1);
                    2        2        2          2        2        2
(%o20)      x (x + 1)  (y + 1)  (z + 1)  + (x + 1)  (y + 1)  (z + 1)

(%i21) multthru((x+1)^2*(z+1)*(y+1));
                              2                    2
(%o21)                 (x + 1)  (y + 1) z + (x + 1)  (y + 1)
(%i22) multthru((x+1)^2*(y+1)^2*(z+1)^2,x^2+1=0);
          2        2        2        2          2        2        2
(%o22)  x  (x + 1)  (y + 1)  (z + 1)  + (x + 1)  (y + 1)  (z + 1)  = 0
```

7.2.5 distrib 函数,multthru 函数,expand 函数の比較

distrib 函数, multthru 函数, expand 函数を比較したものを次に纏めておきましょう:

distrib,multthru,expand の比較

distrib((a+b)*(c+d))	⇒	b*d + a*d + b*c + a*c
expand((a+b)*(c+d))	⇒	b*d + a* d + b*c + a*c
multthru ((a+b)*(c+d))	⇒	(b + a)*d + (b + a)*c
distrib (1/((a+b)*(c+d)))	⇒	1/((b + a)*(d + c))
expand(1/((a+b)*(c+d)),1,0)	⇒	1/(b*d + a*d + b*c + a*c)

7.2.6 sum 函数の簡易化に関連する函数

sumcontract 函数と intosum 函数の構文

sumcontract(⟨ 式 ⟩)
intosum(⟨ 式 ⟩)

sumcontract 函数: 上限と下限の差が定数となる加法の全ての総和を結合します. 結果は各総和の集合に対して, 全ての適切な外の項を加えて一つの総和にしたものを含む式になります. suncontract 函数は全ての互換な総和を結合し, 可能であれば, 総和の一つから添字の一つを用います. sumcontract 函数を実行する前に intosum(⟨ 式 ⟩) の実行が必要かもしれません:

```
(%i18) sum(1/n^2,n,1,m)+sum(1/n^3,n,1,m);
                                 m             m
                               ====          ====
                               \       1     \       1
(%o18)                          >     --  +  >     --
                               /        2    /        3
                               ====   n     ====   n
                               n = 1         n = 1
(%i19) sumcontract(%);
                                  m
                                 ====
                                 \         1      1
(%o19)                            >     (-- + --)
                                 /          2      3
                                 ====     n      n
                                 n = 1
```

intosum 函数: 総和の乗法がなされる全ての物を取り, それらを総和の内部に置きます. 添字が式の外側で用いられていれば, この函数は sumcontract に対して実行するのと同様に適切な添字を探そうとします. これは本質的に総和の outative 属性の観念の逆になりますが, この属性を取り除かずに素通りするだけであることに注意して下さい.
intosum 函数を用いる前に 'scanmap(multthru,⟨ 式 ⟩)' を実行しなければならない場合もあります.

7.2.7　簡易化を行う函数

radcan 函数と scsimp 函数の構文

radcan(⟨ 式 ⟩)
scsimp(⟨ 式 ⟩ ,⟨ 規則$_1$⟩ , ··· ,⟨ 規則$_n$⟩)

radcan 函数:　引数 ⟨ 式 ⟩ には対数函数や指数函数, 冪乗根を含んでいても構いません. ⟨ 式 ⟩ をある変数順序に対する CRE 表現に変換し, 簡易化を行います.
なお, 特定の変数順序に対して CRE 表現は一意に定まります (したがって, CRE 表現は式の正準表現になります). そのために radcan 函数を用いた簡易化も一意に定まります. ところが, この radcan 函数は時間を多く消費します. これは因子分解と指数の部分分数展開を基本とした簡易化の為, 式の成分の間の関係を探索する為です.

scsimp 函数:　scsimp(=Sequential Comparative SIMPlification) 函数は式 (その最初の引数), 同一性や規則 (その他の引数) の集合を取って簡易化を試みます. より小さな式が得られると, その処理が繰返されます. そうでなければ, 全ての簡易化が試みられたあとに元の式が返却されます.

7.2.8　簡易化に関する補助的函数

asksign 函数:　asksign 函数は引数の対象が正, 負, あるいは零の何れかであるかを文脈を使って判別する函数で, 文脈で判断できなければ利用者に直接尋ねる函数です. この構文を次に示します:

asksign 函数の構文

asksign (⟨ 式 ⟩)

判断では Maxima の文脈を用いますが, 文脈だけで決定出来なければ, その演繹を完遂するために必要な質問を利用者に対して行います. この利用者の答は一時的に Maxima に記録されます. ここで, asksign が尋ねる値は 'pos'(正値), 'neg'(負値), 'zero'(零) や 'nonzero'(非零) の何れか一つです.

7.2.9　共通の項で纏める函数

facout 函数の構文

facout(⟨ 式$_1$⟩,⟨ 式$_2$⟩)

facout 函数:　第 2 引数の ⟨ 式$_2$⟩ の各項を第 1 引数の ⟨ 式$_1$⟩ で割って全体を第 1 引数の積で纏めた形式で返す函数です. この函数の性質上, 第 2 引数が単項式であれば第 2 引数の式がそのまま返却されます:

```
facout(sin(x),cos(x)+sin(x));
                                  cos(x)
(%o42)                           (------ + 1) sin(x)
                                  sin(x)
(%i43) facout(sin(x),cos(x));
(%o43)                                cos(x)
```

7.3　代数方程式

7.3.1　Maxima での方程式とその解法について

方程式の書式

Maxima の方程式は演算子 “=” の両側に比較の演算子を持たない式を配置した式で, たとえば, ‘x^2+2*x+1=0’ のような対象です. ここで演算子 “=” は infix 型の演算子のために, この演算子の左右の式は lhs 函数と rhs 函数を使って取出せます:

```
(%i17) eq1:x^2+2*x+1=y^2;
                                  2                2
(%o17)                          x  + 2 x + 1 = y
(%i18) lhs(eq1);
                                   2
(%o18)                           x  + 2 x + 1
(%i19) rhs(eq1);
                                        2
(%o19)                                 y
```

この例では方程式として $x^2 + 2 * x + 1 = y^2$ を eq1 に割当てており, $\boxed{\text{lhs(eq1)}}$ で方程式の左側の ‘x^2+2*x+1’, $\boxed{\text{rhs(eq1)}}$ で方程式右側の ‘y^2’ をそれぞれ取出しています. この lhs 函数と rhs 函数は infix 型の内挿演算子に対してのみ利用可能な函数で, もう一つの内挿演算子である nary 型に対しては利用出来ません.

Maxima で扱える方程式

Maxima で扱える方程式には 1 変数多項式で構成された方程式, 多変数多項式で構成された連立方程式, cos や log 等の初等函数を含むより一般的な方程式があります. 他に名詞型の微分項’diff を含む方程式や名詞型の積分項’integrate を含む方程式もあります. なお, 微分を含む方程式は §7.7 にて詳細を述べるため, ここでは微分と積分を含まない代数方程式 (Algebraic Equation) を中心に述べます.

Maxima では一つの方程式だけではなく, 複数の方程式で構成された系, すなわち, 連立方程式を扱うことも可能です. この場合, $[eq_1, \ldots, eq_n]$ のように方程式を成分とするリストで連立方程式を表現します:

```
(%i25) eq2:[2*x^2-5*y=1,x+y*x+y^2=4];
                          2                2
(%o25)                [2 x  - 5 y = 1, y  + x y + x = 4]
(%i26) eq2[1];eq2[2];
                                    2
(%o26)                          2 x  - 5 y = 1
(%i27)
                                  2
(%o27)                          y  + x y + x = 4
```

この例では二つの方程式 '2*x^2-5*y=1' と 'x+y*x+y^2=4' で構成される方程式系をリストで表現し, それを変数 eq2 に割当てています. 最後の例は連立方程式をリストで表現するために一つの方程式を取り出す場合はリストの成分の取り出しと同じ方式で行えることを示しています.

方程式を解く函数の概要

Maxima では与えられた代数方程式の解法は数値的に近似的に解く方法と代数的数を用いた厳密解を計算する方法に大きく分類出来ます.
まず, 方程式の近似解を数値的に解く函数に allroots 函数と realroots 函数があります. 次に方程式の厳密解を求める函数として, linsolve 函数と solve 函数があります.
そして, 厳密解が計算出来る場合には厳密解を計算し, 厳密解の計算に失敗した場合に近似解を計算する algsys 函数もあります.
これらの函数は与えられた方程式が 1 変数の多項式で構成される場合, 線形連立方程式の場合, 多変数多項式で構成される方程式系の場合, そして, より一般的な初等関数を含む方程式の場合に分類できます. まず, 方程式系が一つの 1 変数多項式で構成される場合, 近似解を計算する allroots 函数と realroots 函数が使えます. 線形連立方程式の場合は linsolve 函数を使って厳密解を計算できます. そして, 多変数多項式で構成された方程式系に対しては algsys 函数によって, 可能であれば厳密解, 厳密解が求められない場合でも数値近似解がを求められます. 最後に, より一般的な方程式に対しては solve 函数を用いて厳密解の計算が行えます.

重要な大域変数

ここで, 代数方程式の求解を行う函数全般に影響を及ぼす重要な大域変数について纏めておきます. ここで解説する大域変数には変数に解を自動代入を行うといった自動代入の制御, 出力書式をリストやラベル付きの与件に切換えること, 重複解があった場合にその重複度を保存する大域変数があります:

重要な大域変数

変数名	既定値	概要
backsubst	true	三角函数化した方程式の代入を抑制
globalsolve	false	解の値の自動代入を制御
multiplicities	not_set_yet	重複度リスト
programmode	true	allroots 函数,linsolve 函数,solve 函数等の出力を制御

大域変数 backsubst: solve 函数と linsolve 函数に対して影響を持つ大域変数です. 'true' であれば連立方程式の解を通常の書式 '$[x = a_1, y = a_2, \ldots, w = a_{n-1}, z = a_n]$' で返しますが, 'false' であれば (上下) 三角行列を用いて式を簡易化した段階で止めた状態の解 '$[x = f_1(y, \ldots, w, z), \ldots, w = f_{n-1}(z), z = a_n]$' を返却します:

```
(%i20) backsubst;
(%o20)                                true
(%i21) linsolve([2*x+y=1,5*x-10*y=4],[x,y]);
                                     14        3
(%o21)                          [x = --, y = - --]
                                     25        25

(%i22) backsubst:false$
(%i23) linsolve([2*x+y=1,5*x-10*y=4],[x,y]);
                                   y - 1         3
(%o23)                       [x = - -----, y = - --]
                                     2           25
```

この例で示すように大域変数 backsubst の値が ‘true’ であれば通常の解を返しています. これに対して大域変数 backsubst の値が ‘false’ の場合, 方程式系をより簡単な方程式系に置換えたものとなっており, Maxima の変数順序 “$>_m$” の大きな変数順に順次代入することで通常の解が得られる解が返却されています.

大域変数 globalsolve:　Maxima で方程式を解いたときに方程式の変数に求めた解を自動代入するかどうかを制御する変数です. 大域変数 globalsolve が true の場合に各変数に対応する解が実際に割当てられます:

```
(c101) globalsolve:true;
(d101)                                true
(c102) solve([xx*2+yy*3-1=0,xx+yy=10],[xx,yy]);
(d102)                        [[xx : 29, yy : - 19]]
(c103) xx;
(d103)                                 29
(c104) yy;
(d104)                                - 19
(c105) globalsolve:false;
(d105)                                false
(c106) solve([mm*2+nn*3-1=0,mm+nn=10],[mm,nn]);
(d106)                        [[mm = 29, nn = - 19]]
(c107) mm;nn;
(d107)                                 mm
(c107)
(d107)                                 nn
```

true の場合, ある方程式を解いたあとで同じ変数の方程式を解こうとすると次のエラーが出るので注意が必要です. たとえば, 上記の例の (c106) 行の方程式を置き換えた場合の結果を次に示します:

```
(c106) solve([xx*2+yy*3-1=0,xx+yy=10],[xx,yy]);
a number was found where a variable was expected -solve
 -- an error.  quitting.  to debug this try debugmode(true);)
(c107)
```

ここで重複度が 2 以上の変数が存在する場合, この自動代入は実行されないことに注意して下さい. そして, 変数に値を束縛せずに式を評価したければ, ev 函数 (§5.8.3 参照) を用いべきです.

大域変数 multiplicities: solve 函数や realroots 函数で返される個々の解に対応する重複度のリストが設定されます:

```
(%i2) multiplicities;
(%o2)                          not_set_yet
(%i3) solve(x^2-4*x+4,x);
(%o3)                            [x = 2]
(%i4) multiplicities;
(%o4)                              [2]
(%i5) realroots(x^4+2*x^3-3*x^2-4*x+4);
(%o5)                       [x = - 2, x = 1]
(%i6) multiplicities;
(%o6)                            [2, 2]
(%i7) solve(x^5+x^4-2*x^3-2*x^2+x+1,x);
(%o7)                       [x = 1, x = - 1]
(%i8) multiplicities;
(%o8)                            [2, 3]
(%i9) factor(x^5+x^4-2*x^3-2*x^2+x+1);
                                   2      3
(%o9)                        (x - 1)  (x + 1)
```

この例では最初に方程式 $x^2 - 4x + 4 = 0$ を solve 函数を用いて解き, 次に, realroots 函数を使って方程式 $x^4 + 2x^3 - 3x^2 - 4x + 4 = 0$ を解いています. そして最後には, $x^5 + x^4 - 2x^3 - 2x^2 + x + 1 = 0$ を解いています. 最初の例では重複度が 2 のために大域変数 multiplicities にはリスト [2] が割当てられています. 次の例では重複度が各々2 であることが判ります. 最後の例では $x = 1$ の重複度が 2, $x = -1$ の重複度が 3 であることが判ります. 実際, 与式を factor 函数で因子分解すれば確認できます.

大域変数 programmode: solve 函数, realroots 函数, allroots 函数と linsolve 函数の返却値に中間行ラベルを付けて出力するかどうかを制御します.
false であれば%t ラベル (中間行ラベル) に解をラベル付けして出力し, true であればリストの書式で解を返却します:

```
(%i4) programmode:false;
(%o4)                             false
(%i5) solve(x^2+1,x);
Solution:

(%t5)                            x = - %i

(%t6)                             x = %i
(%o6)                          [%t5, %t6]
(%i6) programmode:true;
(%o6)                              true
(%i7) solve(x^2+1,x);
(%o7)                      [x = - %i, x = %i]
```

7.3.2　1 変数多項式方程式の場合

数値近似解を求める函数

方程式が多項式で構成された場合について述べます. 方程式系が一つの 1 変数多項式のみで構成されている場合, その数値近似解を allroots 函数と realroots 函数を用いて計算出来ます:

数値解を求める函数

allroots(⟨ 方程式 ⟩)
realroots(⟨ 多項式 ⟩ ,⟨ 許容範囲 ⟩)
realroots(⟨ 多項式 ⟩)

allroots 函数:　単変数の実数係数多項式の実数解と複素解全てを計算します. allroots 函数は多項式が実係数で大域変数 polyfactor が 'true' の場合, 実数上で因子分解を行いますが, 係数に純虚数 '%i' が含まれていれば複素数上で因子分解を行います:

```
(%i14) allroots(%i*x^2+1=0);
(%o14) [x = .7071067811865475 %i + .7071067811865475,
        x = - .7071067811865475 %i - .7071067811865475]
(%i15) polyfactor:true;
(%o15)                                true
(%i16) allroots(x^2+1=0);
                                       2
(%o16)                                x  + 1.0
(%i17) allroots(%i*x^2+1=0);
(%o17) %i (x - .7071067811865475 %i - .7071067811865475)
          (x + .7071067811865475 %i + .7071067811865475)
```

allroots 函数は重複解を持つときに不正確な結果を返すことがあります. この場合は与式に純虚数 '%i' をかけたものを計算すれば解決することがあります.
allroots 函数は多項式方程式以外には使えません. rat 函数を実行したあとに方程式の分子が多項式で, 分母が高々複素数でなければなりません. 大域変数 polyfactor が 'true' であれば allroots 函数の結果として常に同値な式 (ただし , 因子分解されたもの) が返されます.

realroots 函数:　与えられた実単変数多項式 ⟨ 多項式 ⟩ の全ての実根を ⟨ 許容範囲 ⟩ で指定する許容範囲内で求めます. なお, ⟨ 許容範囲 ⟩ が 1 よりも小さければ全ての整数根を厳密に求めます. ⟨ 許容範囲 ⟩ は必要であれば, 任意の小さな数を設定しても構いません. ⟨ 許容範囲 ⟩ を省略した場合, 大域変数 rootsepsilon の値が使われます:

```
(%i34) realroots(x^2-2=0,1.0e-5);
                                  370727       370727
(%o34)                      [x = - ------, x = ------]
                                  262144       262144
(%i35) float(sqrt(2)-rhs(%o34[2]));
(%o35)                        2.289179735770474E-6
```

この例では方程式 $x^2 - 2 = 0$ の解を精度 10^{-5} 以内で求めています. 解は浮動小数点数ではなく有理数で返されます.
realroots 函数は解の大域変数 multiplicities に重複度の情報をリスト形式で追加します. 大域変数 multiplicities に解の重複度リストを設定する函数には他に solve 函数があります. ここで, 重複度リストは求めた解に対応する形で整数のリストとして表現されています.

allroots 函数に影響を与える大域変数

変数名	既定値	概要
polyfactor	false	因子分解の有無
rootsepsilon	1.0E-7	根を含む区間

大域変数 polyfactor: allroots 函数で利用される大域変数で, 'true' であれば多項式が実係数多項式なら実数上で因子分解し, 係数に純虚数 '%i' が含まれていれば複素数上で因子分解を行った結果を返します.

大域変数 rootsepsilon: realroots 函数によって見つけられた根を含む確実な区間を設定する際に使う実数です.

区間内の根の個数を求める函数

nroots 函数: 指定した半開区間に存在する根の個数を計算する函数です. この nroots 函数は 1 変数多項式に対して利用可能です:

区間内の根の個数を返す函数

nroots(⟨ 多項式 ⟩ ,⟨ 下限 ⟩ ,⟨ 上限 ⟩)

nroots 函数は ⟨ 上限 ⟩ と ⟨ 下限 ⟩ で指定された半開区間 '(⟨ 下限 ⟩ ,⟨ 上限 ⟩]' 内部に幾つかの 1 変数多項式 ⟨ 多項式 ⟩ の根があるかを返します.
ここで, 区間の終点は負の無限大と正の無限大にそれぞれ対応する定数 minf と定数 inf でも構いません. このアルゴリズムには Sturm 級数による手法が適用されています:

```
(%i18) nroots(x^2+2,-2,2);
(%o18)                                 0
(%i19) nroots(x^2-2,-2,2);
(%o19)                                 2
(%i20) nroots(x^2-5,-2,2);
(%o20)                                 0
(%i21) nroots(x^2-1,-2,2);
(%o21)                                 2
```

7.3.3 一般の多項式方程式の場合

allroots 函数と realroots 函数は 1 変数多項式の近似解を計算する函数です. これに対して方程式の厳密解を計算出来る函数として linsolve 函数, algsys 函数と solve 函数あります.

linsolve 函数:

linsolve 函数は多変数の線形多項式で構成された方程式系に対して使える函数です:

線形連立方程式を解く函数

linsolve([⟨ 方程式$_1$⟩ ,⟨ 方程式$_2$⟩ ,...], [⟨ 変数$_1$⟩,⟨ 変数$_2$⟩ ,...])

linsolve 函数は与えられた線形連立方程式を変数リストに対して解きます. 各方程式はそれぞれ与えられた変数リストの多項式でなければなりません:

```
(%i68) linsolve([x+y-2=0,y-x+1=0],[x,y]);
                                  3       1
(%o68)                       [x = -, y = -]
                                  2       2
```

linsolve に影響を与える大域変数

変数名	既定値	概要
linsolve_params	true	解に助変数を導入
linsolvewarn	true	linsolve の警告を抑制

大域変数 linsolve_params:　true であれば linsolve 函数は記号 “%ri” を生成し, 大域変数 algsys に記載された任意の助変数を表現するために用います.
false であれば以前のように linsolve 函数が動作します. すなわち, 不定方程式型に対して他の項の幾つかの引数に対して解きます.

大域変数 linsolvewarn:　false であれば dependent equations eliminated(従属方程式が消去された) というメッセージ出力が抑制されます.

より一般の代数方程式

多変数多項式で構成された連立方程式は, solve 函数や algsys 函数で解くことができます. ここで, solve 函数はより一般的な方程式の厳密解が計算できますが, algsys 函数は, 多変数多項式で構成された方程式系の厳密解の計算に失敗すると今度は近似解を計算する点で多項式方程式の計算では使い易いでしょう.

algsysy 函数

algsys 函数

algsys([⟨ 方程式$_1$⟩ ,⟨ 方程式$_2$⟩ ,...], [⟨ 変数$_1$⟩ ,⟨ 変数$_2$⟩ ,...])

algsys 函数では方程式と変数はリストで与えます. 返却される解に ‘%r1’ や ‘%r2’ といった記号が含まれることがありますが, これらは助変数を表示するために用いられる内部的な変数で, 助変数は大域変数%rnum_list に蓄えられています:

```
(%i1) algsys([2*x+3*y=1],[x,y]);
                                          2 %r1 - 1
(%o1)                          [[x = %r1, y = - ---------]]
                                            3
(%i2) %rnum_list;
(%o2)                                  [%r1]
```

この例のように (連立) 方程式の階数が足りない場合には助変数を補って与えられた方程式を解きます. なお, %rnum_list には algsys 函数で使われた助変数が逐次追加されて行きます.
algsys 函数は以下の手順で方程式を解き, 必要であれば再帰的に処理を行います.

1. 最初に方程式を factor 函数で因子分解し, 各因子から構成される部分系 $\mathtt{system}_i$, すなわち, 方程式の集合を構築します.

2. 部分系 $\mathtt{system}_i$ から方程式 $\mathtt{eq}_n$ を取出し, それから変数 var を選択します. この変数の選択は方程式 $\mathtt{eq}_n$ に含まれる変数の中から最小次数のものを選出します. それから方程式 $\mathtt{eq}_n$ と $\mathtt{system}_i$ の部分系 $\mathtt{system}_i \setminus \{\mathtt{eq}_n\}$ に含まれる方程式 $\mathtt{eq}_j$ を変数 var を主変数とする多項式と看做して終結式を計算します. この操作によって新しい部分系 $\mathtt{system}_{i+1}$ は $\mathtt{system}_i$ よりも少ない変数で生成されます. それから 1 の処理に戻ります.

3. 一つの方程式で構成される部分系が最終的に得られると, その方程式が多変数で, 係数が浮動小数点数でなければ厳密解を求めるために solve 函数を呼出します. さらに方程式が単変数で線型, 二次, あるいは四次の多項式であれば solve 函数を再び呼出します.

   係数が浮動小数点数で近似されている場合, 方程式が単変数で線型, 二次, または四次の何れでもなく, 大域変数 realonly が true であれば実数値解を見付けるために realroots 函数を呼出します. 大域変数 realonly が false であれば解を求めるために函数 allroots を呼出します. なお, algsys 函数が要求以下の精度解を生成した場合、大域変数 algepsilon の値をより小さな値に変更しても構いません. もし, 大域変数 algexact が true であれば solve 函数を呼出します.

4. 3 の段階で得られた解を以前の段階に代入して解の計算過程の 1 に戻ります. なお, 浮動小数を近似した多変数方程式に対しては次のメッセージを表示します:

   > "algsys cannot solve - system too complicated." (意味：「algsys では解けません - 系があまりにも複雑です.」)

   radcan 函数を使えば, 大きくて複雑な式が出来ます. この場合, pickapart 函数や reveal 函数を解の計算に用います.

ここで終結式は二つの一変数多項式に対して定義されるもので, たとえば, 多項式 f と g の根を α_i, β_j とすると, $res(f,g,x) = a_m^n b_n^m \prod_{0\le i\le m, 0\le i\le n}(\alpha_i - \beta_j)$ となることが知られています.
このことは f と g に共通の零点が存在する場合には終結式が零になることを意味し, したがって, f と g が多変数の場合には f と g を f と g の共通の変数 x_1 の多項式と看做してその終結式を計算

すれば f と g の共通の根が存在する場合に終結式は零でなければなりません. こうすることで変数 x を含まない新しい多項式 $res(f,g,x)$ が得られます. この操作を方程式系に対して行うことで 1 変数の多項式が得られると, その多項式を解いて一段前の方程式系に代入し, 解を求めて行く方式となっています.
話を簡単にするために一次の連立方程式で簡単に説明しましょう.
最初に次の線形方程式が与えられたとします:

$$f : ax + by + p = 0 g : cx + dy + q = 0$$

この連立方程式を構成する多項式 f と g の終結式は次の行列式を計算すると得られます.

$$res(f,g,x) = \det\left(\begin{pmatrix} a & by+p \\ c & dy+q \end{pmatrix}\right)$$

この行列式は $(ad-bc)y - ap + cp$ です. ここで, f と g の終結式は f と g が共通の解を持つ場合に 0 となります. このことから方程式 $(ad-bc)y - ap + cp = 0$ が得られます. この終結式では前の方程式系から変数 x と方程式が減り, 変数 y の方程式となります. そこで, 終結式から得られた方程式を解くと $y = \frac{ap-cp}{ab-bc}$ が得られます. それから, 一段前の方程式系に戻って変数 y に値を代入すると変数 x の方程式が得られ, その方程式を解くと最終的に $x = \frac{bq-dp}{ad-bc}$ が得られます.
algsys 函数は方程式系を構成する各多項式を因子分解し, 次数を落した因子で構成される方程式の集合に対して終結式を計算し, 新しい方程式系から変数を一つ消去します. この処理を繰返すことで最終的に一変数の多項式方程式が得られると, そこから一つの変数の解を定めます. この計算で 4 次以下の方程式であれば公式を用いた根の計算が可能ですが, それ以外の場合で厳密解が計算出来なければ allroots 函数を用いて近似数値解を求め, それから処理を逆に遡ることで全ての解を求められます. algsys 函数はこのような計算手順を採用しているのです.

algsys 函数に影響を与える大域変数

変数名	既定値	概要
%rnum_list	[]	algsys 函数で解に導入された変数リスト
algexact	false	algsys 函数が solve 函数を呼出すかどうかを制御
algepsilon	10^8	algsys で用いられる変数
algedelta	10^{-5}	algsys で用いられる変数
realonly	false	true の場合,algsys 函数は実数解のみを返却

%rnum_list:　方程式の解を求めたときに導入された変数%r が生成順に追加されるリストです. これはあとで解に代入するときに便利です.

大域変数 algexact:　true であれば algsys 函数は solve 函数を内部で呼出し, realroots 函数を常に利用します. false であれば終結式が単変数でない場合と quadratic か biquadratic な場合のみ solve の呼出しを行います. 大域変数 algexact が true であれば厳密解のみを保証するものではなく, algsys 函数が最初に厳密解を計算しようと試み, 結局, all か失敗したときに近似解のみを生成します.

大域変数 algepsilon: 解の精度を制御する大域変数です.

大域変数 algdelta: 近似解を方程式に代入して零点からのズレを見る際に代入した式に複素数が現われたときに誤差の判定で用いられる大域変数です. 初期値に 1.0e-5 が設定されています.

大域変数 realonly: true であれば algsys 函数は実数解, すなわち, 純虚数%i を包含しない解のみを返します.

solve 函数

solve 函数で扱える方程式としては sin 等の三角函数, 指数函数や対数函数を含んだ方程式が扱えます. ただし, algsys 函数と違い, 厳密解の計算に失敗した場合に数値近似解を計算しません.

solve 函数

solve(⟨ 式 ⟩)
solve(⟨ 式 ⟩ ,⟨ 変数 ⟩)
solve([⟨ 方程式 ⟩ ,···,⟨ 方程式$_n$⟩])
solve([⟨ 方程式 ⟩ ,···,⟨ 方程式$_n$⟩], [⟨ 変数$_1$⟩ ,···,⟨ 方程式$_n$⟩])

solve 函数は代数方程式 ⟨ 式 ⟩ を ⟨ 変数 ⟩ に対して解き, 解のリストを返します.
⟨ 式 ⟩ が方程式でなければ, ⟨ 式 ⟩ が零に等しいと設定されていると仮定します. すなわち, 式 x^2+2*x+1 が ⟨ 式 ⟩ であれば, solve 函数は方程式 x^2+2*x+1=0 が与えられたと solve 函数は解釈します.
⟨ 変数 ⟩ は和や積を除く函数の様な原子でない式でも構いませんが, ⟨ 式 ⟩ が函数 $f(x)$ の多項式であれば, 最初に $f(x)$ に対して解き, その結果が c であれば方程式 $f(x) = c$ を解くことで対処できます. 具体的には次の処理を行います:

```
(%i26) solve(log(x)^2-2*log(x)+1,log(x));
(%o26)                              [log(x) = 1]
(%i27) solve(%o25[1],x);
(%o27)                                [x = %e]
```

⟨ 式 ⟩ が 1 変数のみの場合は ⟨ 変数 ⟩ を省略できます. さらに ⟨ 式 ⟩ は有理式でも良く, その上, 三角函数, 指数函数等を含んでいても構いません.
solve 函数は与えられた方程式が単変数の場合は次の手順で解の計算を行います:

- 方程式が変数 *var* の線形結合であれば, *var* に対して自明に解けます.
- 方程式が $a \cdot var^n + b$ の形式ならば, 解は $(-b/a)^{1/n}$ に 1 の n 乗根を掛けたもので得られます.
- 方程式が変数 *var* の線形結合ではなく方程式に含まれる変数 *var* の各次数の gcd(n とします) が次数を割切る場合, 大域変数 multiplicities に n が追加されます. そして, solve 函数は var^n で方程式を割った結果に対して再び呼出されます.
- 方程式が因子分解されている場合, 各因子に対して solve 函数が呼出されます.

- 方程式二次, 三次, あるいは四次の多項式方程式の場合, 解の公式を必要があれば用います.

solve([⟨ 方程式$_1$⟩ ,···,⟨ 方程式$_n$⟩], [⟨ 変数$_1$⟩ ,···,⟨ 変数$_n$⟩]) の場合, 多項式の方程式系を linsolve 函数, あるいは algsys 函数を用いて解き, その変数で解のリストを返します. ここで, linsolve 函数を用いる場合は第 1 引数のリスト [⟨ 方程式$_i$⟩ ,i=1,···,n] は解くべき方程式を表現し, 第 2 の引数リストは求めるべき未知変数のリストになりますが, 方程式中の変数の総数が方程式数と等しい場合, 第 2 引数のリストは省略しても構いません.
与えられた方程式が十分でない場合, inconsistent と云うメッセージを表示します. これは大域変数 solve_inconsistent_error で制御できます. また, 単一解が存在しない場合は singular と表示されます.

solve 函数の挙動に影響する大域変数

変数名	既定値	概要
solvedecomposes	true	polydecomp を用いるかどうかを制御
solveexplicit	false	陰的な解を許可するかどうかを制御
solvefactors	true	因子分解の実行の有無
solvenullwarn	true	空リストの警告の有無
solveradcan	false	radcan を用いるかどうかを指定
solvetrigwarn	true	方程式を解く際に逆三角函数を利用するかを指定
solve_inconsistent_error	true	階数が不十分な連立方程式に対するエラー表示の有無
breakup	true	解の表示を制御

大域変数 solvedecomposes: 'true' であれば多項式を解く際に solve 函数に polydecomp 函数を導入します.

大域変数 solveexplicit: 'true' であれば solve に陰的な解, すなわち, $f(x) = 0$ の形式で返すことを禁止します.

大域変数 solvefactors: 'false' であれば solve 函数は式の因子分解を実行しません.

大域変数 solvenullwarn: 'true' であれば空の方程式リストや空の変数リストで solve 函数を呼んだ場合に警告が出ます. たとえば, solve([],[]) と入力すると警告と一緒に空リスト "[ ]" が返されます.

大域変数 solveradcan: 'true' であれば solve 函数は内部で radcan 函数を用います. radcan 函数を用いることで solve 函数の全体的な処理速度は低下しますが, 指数函数や対数函数を含む問題に対処できます.

大域変数 solvetrigwarn: 'false' であれば solve 函数は方程式を解くために逆三角函数を利用し, それによって解が失なわれることを警告しません.

solve_inconsistent_error: 'true' であれば solve 函数と linsolve 函数は [a+b=1,a+b=2] のように階数が不十分な線形連立方程式に遭遇すればエラーを表示します. なお, 'false' であれば空リスト "[ ]" を返します.

大域変数 breakup: 'false' であれば solve 函数はデフォルト値の幾つかの共通部分式で構成されたものとしてはなく, 一つの式として三次又は二次の方程式の解の表示を行います. ただし, 大域変数 breakup が true となるのは大域変数 programmode が 'false' のときだけです.

find_root 函数

find_root 函数の構文

find_root(⟨ 方程式 ⟩,⟨ 変数 ⟩,⟨x_0⟩,⟨x_1⟩)
find_root(⟨ 式 ⟩,⟨x_0⟩,⟨x_1⟩)

find_root 函数は閉区間 [⟨x_0⟩,⟨x_1⟩] 内部で与えられた方程式や式の根の近似解を求めます.
find_root 函数で与えられる式は未知変数が一つの方程式や通常の式で, plot2d 函数で描画可能な式であれば十分です.
まず, 引数が 4 個の場合に第 1 引数に対して内部で lhs 函数と rhs 函数が用いられているために infix 型の内挿式演算子で, lhs 函数や rhs 函数で左右の式が取り出せるような束縛力を持った演算子を一つ含む式が与えられます. ただし, 内部では lhs 函数で取出した式と rhs 函数で取出した式の差を計算しているので, 演算子は "=" として処理が行われるのと何等の違いもありません. したがって, この場合には方程式を与えることになります.
これに対して引数が 3 個の場合, 第 1 引数の式に対してこのような前処理を行いません.
それから, 引数が 4 個の場合は演算子の左右の式の差, 引数が 3 個の場合は第 1 引数を plot2d 函数や plot3d 函数で用いられる内部函数 coerce-float-fun に引渡して数値データを生成し, この数値データを基に近似解の検出を二分法, 与式の函数が十分に滑らかな場合には線形近似を適用して近似解を求めます.
なお, find_root 函数で根を求める閉区間は, その両端の点の符号が一致するものであることを前提にしています. そのために開区間の両端の点の符号が異なる場合や両端の点が一致する場合にはエラーメッセージが出力されます.

find_root 函数を制御する大域変数

大域変数	初期値	概要
find_root_error	true	区間が 0 点を包含する場合のエラー処理を制御
find_root_abs	0.0	近似解の精度に関連
find_root_rel	0.0	近似解の精度に関連

find_root 函数の近似解の精度は大域変数 find_root_abs と大域変数 find_root_rel で制御されます.

なお, find_root 函数は以前,interpolate 函数という名前の函数であったために interpolate(⟨ 式 ⟩) を実行すると find_root 函数を使うことを指示するメッセージが出力されます.

7.3.4　漸化式の場合

Maxima では nusum パッケージを用いることで漸化式が扱えます. とはいえ, 機能的にはまだ不十分です.

漸化式を解く函数

funcsolve(⟨ 方程式 ⟩ ,⟨$g(t)$⟩)

funcsolve 函数:　⟨ 方程式 ⟩ を満たす有理函数 ⟨$g(t)$⟩ が存在すれば有理函数のリストを返し, 存在しない場合は空リスト "[ ]" を返します.
ただし, 方程式は ⟨$g(t)$⟩ と ⟨$g(t+1)$⟩ の一次線形多項式でなければなりません. すなわち, funcsolve 函数は一次線形結合の漸化式に対して利用可能です.

```
(%i28)   funcsolve((n+1)*foo(n)-(n+3)*foo(n+1)/(n+1) =
(n-1)/(n+2),foo(n));

dependent equations eliminated:  (4 3)
                                          n
(%o28)                    foo(n) = ---------------
                                   (n + 1) (n + 2)
```

7.4 極限

7.4.1 極限について

Maxima では極限 “lim” が使えます. 一見すると極限 “lim” は代入と似た操作に見えるかもしれませんが, 実際は全く異った操作です.
たとえば $\frac{\sin(x)}{x}$ の原点での値は何になるでしょうか？ 安易に代入すると $\frac{0}{0}$ となりますが, ここで分母と分子が同じ 0 だから 1 が得られると結論付けることはできません. 実際, $\frac{x}{x}$ も $\frac{x^2}{x}$ も単純に x に 0 を代入すると共に $\frac{0}{0}$ を得ますが, ところが $x \neq 0$ ならば前者は $\frac{x}{x} = 1$, 後者は $\frac{x^2}{x} = x$ となるので前者は恒等函数 1 の $x = 0$ での値 1 であるべきで, 後者は写像 x の $x = 0$ の値, すなわち, 0 であるべきです. このように安易に代入した結果として得られる $\frac{0}{0}$ の値は定まったものではないと考える方が自然です. この $\frac{0}{0}$ や $\frac{\infty}{\infty}$ は不定形と呼ばれ, 単純に代入した最終的な式では判断ができず, その元の式を考慮する必要がある数式なのです. そして, その本来の式を使って x が特定の値となった場合に取り得る値を計算する操作を極限と呼ぶのです.
さて, $\frac{\sin(x)}{x}$ に話を戻しましょう. この場合は $\sin(x)$ の原点周りの級数展開を考えると判り易くなります. 実際, $\sin(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$ となり, これを x で割ってしまうと, $\frac{\sin(x)}{x} = 1 + x \cdot (x \text{ の冪級数})$ となります. 以上から x を 0 に近づけると 1 になることが判りますね.
このことを Maxima で試してみましょう. Maxima には極限を計算する limit 函数があり, limit(〈 函数 〉,〈 変数 〉,〈 値 〉) で極限の計算が行えます:

```
(%i39) limit(sin(x)/x,x,0);
(%o39)                                1
(%i40) plot2d(sin(x)/x,[x,-50,50]);
```

Maxima の計算でも, $\lim_{x \to 0} \frac{\sin x}{x} = 1$ となることが判ります. 図 7.2 には, この $\sin x/x$ の閉区間 $[-50, 50]$ での様子を示しておきます:

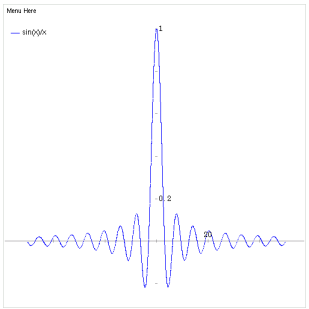


図 7.2: sin(x)/x のグラフ

極限の方向

ここで近付けるという操作では, その近付ける「方向」を考えなければなりません. たとえば, $\frac{1}{x}$ ではどうでしょうか？この函数は $x > 0$ なら正,$x < 0$ で負, そして, $x = 0$ が不連続点になっています. この様子は図 7.3 に示す通りです:

図 7.3: 1/x のグラフ

この処理は Maxima では “plus” や “minus” といったオプションを limit 函数で与える事で対処します:

```
(%i73) limit(1/x,x,0);
(%o73)                                  und
(%i74) limit(1/x,x,0,plus);
(%o74)                                  inf
(%i75) limit(1/x,x,0,minus);
(%o75)                                 minf
```

まず, 最初の limit 函数では方向を指定していないために ‘und’, すなわち, 不定値となっています. 次の例では ‘plus’ を指定しているので limit 函数は右側から ‘0’ に近付けます. そして, ‘inf’, すなわち, 正の無限大 ∞, 左側から近付けた場合には負の無限大 minf, すなわち, $-\infty$ を得ます. このように方向を指定すると, その方向によって結果が綺麗に分れます. つまり, $\lim_{x \to 0} 1/x$ の結果は不定値 und となることを意味するのです.
なお, limit 函数で正の無限大は定数 ‘inf’, 負の無限大は定数 ‘minf’, 複素数での無限大は定数 ‘infinity’, 左右の極限が異なる場合には ‘und’, 未定でも有界なものには ‘ind’ といった表記を返します:

```
(%i89) limit(1/(x^2-1),x,1,plus);
(%o89)                                  inf
(%i90) limit(1/(x^2-1),x,1,minus);
(%o90)                                 minf
(%i91) limit(1/(x^2-1),x,1);
(%o91)                                  und
(%i92) limit(sin(1/x),x,0);
(%o92)                                  ind
(%i93) limit(1/(x^2+1),x,%i);
(%o93)                             infinity
```

7.4.2 limit 函数

limit 函数の構文
limit(⟨ 式 ⟩,⟨ 変数 ⟩,⟨ 値 ⟩,⟨ 方向 ⟩)
limit(⟨ 式 ⟩,⟨ 変数 ⟩,⟨ 値 ⟩)
limit(⟨ 式 ⟩)

limit 函数は与えられた ⟨ 式 ⟩ の極限を計算します. この際, 変数が近づく方向を指定できます. この場合, ⟨ 変数 ⟩ が ⟨ 値 ⟩ に ⟨ 方向 ⟩ で指定したから接近する場合の ⟨ 式 ⟩ の極限を計算します. ここで, 方向は右極限なら 'plus', 左極限なら 'minus' とし, 方向を省略した場合は両側極限が計算されます.
ここで, 原点の極限計算であれば特別に 'zeroa' や 'zerob' も使えます. ここで 'zeroa' が原点の左側 (-側), 'zerob' が原点の右側 (+側) から近付けることを意味します. この場合は 'minus' や 'plus' のような方向を指定する必要はありません.
limit 函数は特別な定数として 'inf'(正の無限大) と 'minf'(負の無限大) を用います. 出力では 'und'(未定義), 'ind'(不定だが有界) と 'infinity' (複素無限大) が使われる場合があります. なお, '-inf' と 'minf', '-minf' と 'inf' は Maxima では意味が全く異なるので注意して下さい. 極限を含む式の計算で limit 函数が利用できます. すなわち, 上記の '-inf' や '-minf' に加え, 'inf-1' ような式に対しては, limit 函数は式のみで評価を行います:

```
(%i51) inf - 1;
(%o51)                                inf - 1
(%i52) limit(%);
(%o52)                                  inf
(%i53) limit(-inf);
(%o53)                                 minf
(%i54) limit(x^2+inf*x);
Is  x  positive, negative, or zero?

pos;
(%o54)                                  inf
```

この例で示すように 'inf-1' のような式は Maxima では自動的に評価されませんが, limit 函数を用いて値を評価することができます. 'limit(x^2+inf*x)' のような式も同様です.

7.4.3 tlimit 函数:

tlimit 函数の構文
tlimit(⟨ 式 ⟩,⟨ 変数 ⟩,⟨ 値 ⟩,⟨ 方向 ⟩)
tlimit(⟨ 式 ⟩,⟨ 変数 ⟩,⟨ 値 ⟩)
tlimit(⟨ 式 ⟩)

大域変数 tlimswitch を内部で true にした limit 函数です. この tlimit 函数は ⟨ 式 ⟩ の Taylor 展開を行い. その展開式に対して極限計算を行います.

7.4.4　極限に関連する大域変数

極限に関連する大域変数

変数名	既定値	概要
lhospitallim	4	limit 函数で用いられる l'Hospital 則の適用階数の上限
limsubst	false	limit の代入を制御
tlimswitch	false	taylor 展開を利用するかどうか

大域変数 lhospitallim:　limit 函数で用いられる l'Hospital 則の適用回数の最大値,
これは 'limit(cot(x)/csc(x),x,0)' のような場合に無限ループに陥いることを防ぐためです.

大域変数 limsubst:　limit 函数が未知の形式に代入を行うことを防ぐ働きがあます.
これは 'limit(f(n)/f(n+1),n,inf)' のような式が '1' となる問題点を避けるためです.
大域変数 limsubst が true であれば, このような代入が許容されます.

大域変数 tlimswitch:　true であれば極限パッケージは可能なときに taylor 展開を利用します.

7.5 微分

7.5.1 微分に関係する函数

微分に関連する函数の構文

diff(⟨ 式 ⟩,⟨ 変数$_1$⟩,⟨ 階数$_1$⟩, $\cdots$,⟨ 変数$_n$⟩,⟨ 階数$_n$⟩)
diff(⟨ 式 ⟩,⟨ 変数 ⟩)
diff(⟨ 式 ⟩)
del(⟨ 式 ⟩)
derivdegree(⟨ 式 ⟩,⟨ 従属変数 ⟩, ⟨ 独立変数 ⟩)

diff 函数: Maxima の式の微分では diff 函数を用いますが, ここで 1 階微分の場合のみ, 'diff(⟨ 式 ⟩,⟨ 変数 ⟩)' のように階数を省略しても構いません. なお, 通常は ⟨ 変数$_i$⟩ と ⟨ 階数$_i$⟩ の一組を指定して ⟨ 式 ⟩ の微分を行います.
'diff(⟨ 式 ⟩)' は全微分を与えます. すなわち, ⟨ 式 ⟩ の各変数に対する微分と各変数の函数 del との積の和になります:

```
(%i41) diff(f(x*y));
                     d                          d
(%o41)          (-- (f(x y))) del(y) + (-- (f(x y))) del(x)
                     dy                         dx
(%i42) diff(g(x+y+z));
          d                                 d
(%o42) (-- (g(z + y + x))) del(z) + (-- (g(z + y + x))) del(y)
          dz                                dy
                                            d
                                       + (-- (g(z + y + x))) del(x)
                                            dx
```

derivdegree 函数: 名詞型の微分を含む式で ⟨ 独立変数 ⟩ に対する ⟨ 従属変数 ⟩ の微分で最も高い階数を見付けます. この函数は多項式の次数を返す函数 hipow と似ています:

```
(%i16) derivdegree('diff(y,x,3)*x^4+'diff(y,x,2)*'diff(y,x),y,x);
(%o16)                                3
(%i17) 'diff('diff(y,x,2),x,3)+'diff(y,x,2);
                                     5      2
                                    d y    d y
(%o18)                              --- + ---
                                      5      2
                                    dx     dx
(%i19) derivdegree(%,y,x);
(%o19)                                5
```

ただし, derivdegree 函数は与式の展開等を行わないので, 場合によっては間違った答を返すこともあるので注意が必要です:

```
(%i26) 'diff('diff(y,x,2),x,3)*(x^2-1)+'diff(y,x,2)
       -'diff(y,x,5)*(x-1)*(x+1);
                                 5                        5      2
                          2     d y                      d y    d y
(%o26)                  (x  - 1) --- - (x - 1) (x + 1) --- + ---
                                  5                        5      2
                                dx                       dx     dx
(%i27) derivdegree(%,y,x);
(%o27)                                    5
(%i28) expand('diff('diff(y,x,2),x,3)*(x^2-1)+'diff(y,x,2)
              -'diff(y,x,5)*(x-1)*(x+1));
                                          2
                                         d y
(%o28)                                   ---
                                           2
                                         dx
(%i29) derivdegree(%,y,x);
(%o29)                                    2
```

この例では与式の 5 階の微分は式を展開することで消去されるものですが, derivdegree 函数は与式を展開せずに安易に式に含まれる y の x による微分で階数の最も高い項を求め, その階数を返却しています.

微分方程式を記述する場合, 函数の名詞型'diff を用います. ここで微分の名詞型の表示はデフォルトで二次元的書式になりますが, 大域変数 display2d を false にすれば入力と同等の式を一行で返します.

この大域変数 display2d は微分の名詞型に限らず, Maxima の計算結果の表示で数学式の表示を制御する大域変数ですが, 名詞型の微分の表示のみを制御する大域変数として大域変数 derivabbrev があります. また, 名詞型の微分項の代入制御を行う大域変数として derivsubst があります.

大域変数 derivabbrev と derivsubst

変数名	既定値	概要
derivabbrev	false	微分の表示を制御
derivsubst	false	名詞型の微分を含む項の代入を制御

大域変数 derivabbrev: 'true' の場合に名詞型の微分は添字で表示されます:

```
(%i30) 'diff(f(x),x);
                                    d
(%o30)                              -- (f(x))
                                    dx
(%i31) derivabbrev:true$
(%i32) 'diff(f(x),x);
(%o32)                                f(x)
                                         x
(%i33) display2d:false$
(%i34) 'diff(f(x),x);
(%o34) 'diff(f(x),x,1)
```

大域変数 derivsubst: 名詞型の微分項の代入制御が行えます. たとえば, $\frac{d^2y}{dt^2}$ は y の t による二階微分ですが, $\frac{dy}{dt}$ を x と置くと, $\frac{d^2y}{dt^2}$ は $\frac{dx}{dt}$ で置換えられます. 大域変数 derivsubst は名詞型の微分を含む式に対し, このような置換を行うかどうかを制御するものです.
この大域変数 derivsubst が 'false' の場合は susbst 函数による微分項の置換ができませんが, 'true' の場合は置換が行えます:

```
(%i33) derivsubst;
(%o33)                                false
(%i34) subst(x,'diff(y,t),'diff(y,t,2));
                                        2
                                       d y
(%o34)                                 ---
                                         2
                                       dt
(%i35) derivsubst:true;
(%o35)                                true
(%i36) subst(x,'diff(y,t),'diff(y,t,2));
                                       dx
(%o36)                                 --
                                       dt
(%i37) subst(x,'diff(y,t),2*t+t^2*'diff(y,t,2));
                                    2  dx
(%o37)                             t  -- + 2 t
                                       dt
```

7.5.2 vect パッケージ

vect パッケージは標準で Maxima に含まれているパッケージで, grad, div, curl や laplace 等の微分演算子や, それらの式を簡易化する函数が含まれています. このパッケージを利用するためには予め load(vect) を実行しておく必要があります.
vect パッケージでは非可換積の演算子 "." を内積演算子として再定義します. そのために非可換積が可換化されてしまうので注意が必要です. さらに非可換積の結合律と対応する冪への簡易化を勝手に実行しないようにするため, 関連する大域変数 dotsassoc と dotexptsimp が 'false' に設定されます. その一方で, スカラーに対して大域変数 dotscrules を 'true' にすることでスカラーとの非可換積が可換に設定されます.
vect パッケージには次の演算子の定義とそれに付随する函数が収録されています:

vect パッケージに含まれる主な函数

express(⟨*expression*⟩)
potential(⟨*grad*⟩
scalefactors(⟨ 座標変換 ⟩)
vectorsimp(⟨ ベクトル式 ⟩)

vect パッケージに含まれる演算子

演算子	演算子の属性	左束縛力	右束縛力
~	内挿式演算子	134	133
grad	前置式演算子		142
div	前置式演算子		142
curl	前置式演算子		142
laplacian	前置式演算子	142	

vect.mac に含まれているこれらの演算子は定義のみが vect.mac で行われており, 演算子の実体は share/vector/vector.mac に含まれています.
これらの演算子の簡易化は大域変数 expandflags に割当てられたリストに含まれる変数で制御されます. ただし, これらの変数を変更しても直ちに上記の演算子が自動的に簡易化を行うのではありません. vect パッケージの函数 vectsimp 函数を用いて簡易化を行います. この vectorsimp 函数自体は内部で演算子の属性を大域変数 expandflag のリストに登録された大域変数から設定して vsimp 函数で実際の処理を実行させています.

expandflags リストに登録された大域変数

大域変数名	既定値	概要
expandall	false	全ての演算子を展開
expandplus	false	被演算子に含まれる和を展開
expanddot	false	和と内積. を展開
expanddotplus	false	和と内積. を展開
expandgrad	false	grad 演算子を展開
expandgradplus	false	被演算子の和で展開
expanddiv	false	div 演算子を展開
expanddivplus	false	被演算子の和で展開
expandcurl	false	curl 演算子を展開
expandcurlplus	false	被演算子の和で展開
expandlaplacian	false	laplace 演算子を展開
expandlaplacianplus	false	被演算子の和で展開
expandprod	false	積に対して展開
expandgradprod	false	grad 演算子にて積を展開
expanddivprod	false	div 演算子にて積を展開
expandlaplacianprod	false	laplace 演算子にて積を展開
expandcurlcurl	false	curl curl を処理
expandlaplaciantodivgrad	false	laplace 演算子を dvi grad に展開
expandcross	false	外積を展開
expandcrosscross	false	外積を外積に対して展開
expandcrossplus	false	外積を和に対して展開
firstcrossscalar	false	外積とスカラーの処理

全てのこれらの大域変数は既定値として 'false' を持ちます. 変数名のうしろに "plus" の付く大域変数は加法性と被演算子の分配性に関連します. 同様にうしろに "prod" の付く大域変数は可換積や内積 (通常は非可換積) 等の積演算に対する被演算子への分配性に関連するものです.

大域変数 expandcrosscross 'p~(q~r)' を '(p,r)*q-(p.q)*r' で置換するかどうかを決めます.

大域変数 expandcurlcurl: 'curl curl p' を 'grad div p + div grad p' で置換するかどうかを決定します.

大域変数 expandcross: 'true' であれば, 大域変数 expandcrossplus と大域変数 expandcrosscross を共に 'true' に設定した場合と同じ効果があります.
二つの大域変数 expandplus と大域変数 expandprod は似た名前の大域変数に 'true' に設定した場合と同効果です. これらが 'true' であれば, 大域変数 expandlaplaciantodivgrad は laplace 演算子を 'div grad' で置換えます.
簡便のために, これら全ての大域変数は load(vect) で vect パッケージの読込を行った時点で evflag として定義されます:

```
(%i1) load(vect)$
(%i2) expandall:true$
(%i3) laplacian(a*V+b*W);
(%o3)                          laplacian (b W + a V)
(%i4) vectorsimp(laplacian(a*V+b*W));
(%o4)                  laplacian (b W) + laplacian (a V)
(%i5) vectorsimp(grad(a*V+b*W));
(%o5)                        grad (b W) + grad (a V)
(%i6) vectorsimp(grad(a*b));
(%o6)                grad (grad a . b) + grad (a . grad b)
(%i7) (V1+V2).(W1+W2);
(%o7)                          (V2 + V1) . (W2 + W1)
(%i8) vectorsimp((V1+V2).(W1+W2));
(%o8)                V2 . W2 + V2 . W1 + V1 . W2 + V1 . W1
```

evflag 属性を持たない大域変数で重要な大域変数として, 大域変数 vector_cross があります:

大域変数 vector_cross

大域変数名	既定値	概要
vector_cross	false	diff 函数による外積の展開を制御

この大域変数 vect_cross は diff 函数による外積の微分の展開を制御する函数です. 既定値の 'false' の場合に diff 函数による外積の展開を行いませんが, 'true' であれば diff 函数による外積の展開を行います:

```
(%i4) vect_cross;
(%o4)                                  false
(%i5) diff(f(x)g(x),x);d(dx(((d d(dx dx
```

express 函数の構文

express(⟨ 式 ⟩)

express 函数は名詞型の微分を展開します. express 函数は vect パッケージで定義される grad, div, curl, laplacian と外積 "~" を認識します. ただし, express 函数は微分を名詞型で返すために, 実際の微分の計算は ev 函数に'diff オプションを付けて行います:

```
(%i1) load(vect);
(%o1)          /usr/local/share/maxima/5.9.2/share/vector/vect.mac
(%i2) e1:laplacian(x^2*y^2*z^2);
                                          2  2  2
(%o2)                          laplacian (x  y  z )
(%i3) express(e1);
                2                   2                   2
               d      2  2  2      d      2  2  2      d      2  2  2
(%o3)          --- (x  y  z ) + --- (x  y  z ) + --- (x  y  z )
                 2                   2                   2
               dz                  dy                  dx
(%i4) ev(%,'diff);
                                 2  2       2  2       2  2
(%o4)                         2 y  z  + 2 x  z  + 2 x  y
(%i5) v1:[x1,x2,x3][y1,y2,y3];(((
```

7.6 積分

7.6.1 記号積分について

Maxima は記号積分, 数値積分の両方の処理が行えます. ここで記号積分に関しては Risch 積分も不完全ながら実装されています. そのために単純に公式を当て嵌めるだけで積分の計算を行うようなシステムよりも一段と優れた処理が行えます.
記号積分の処理で式が非常に繁雑になる場合, 積分記号の文字による表示が不要な場合は大域変数 display2d を false にすると, 一行で結果が表示されるので, 複雑な式の計算を行う場合には非常に便利です.
なお, Maxima 本体では初等函数 (有理式, 三角函数, 対数函数と指数函数) と多少の拡張 (error 函数等) で可積分なものに限定しているので未知函数の積分は形式的な処理以上のことは行いません. より高度な処理が必要な場合, contrib に含まれるパッケージを活用することになるでしょう. とはいえ, Maxima のライブラリは商用の数式処理システムの *Mathematica* や Maple と比較すると貧弱であり, 今後の拡充に注力する必要があるでしょう.

7.6.2 integrate 函数と risch 函数

Maxima で記号積分を行う函数に integrate 函数と risch 函数の二種類があります. integrate 函数から risch 函数の本体を呼出すことが, ev 函数による評価を介在させることでできるので, 記号積分に関しては integrat e 函数だけで済ますことが可能ともいえます:

integrate 函数と risch 函数の構文

integrate(⟨ 式 ⟩,⟨ 変数 ⟩)
integrate(⟨ 方程式 ⟩,⟨ 変数 ⟩)
integrate([⟨ 式$_1$⟩, $\cdots$,⟨ 式$_n$⟩], ⟨ 変数 ⟩)
integrate([⟨ 方程式$_1$⟩, $\cdots$,⟨ 方程式$_n$⟩], ⟨ 変数 ⟩)
integrate(⟨ 行列 ⟩,⟨ 変数 ⟩,⟨ 下限 ⟩,⟨ 上限 ⟩)
risch(⟨ 式 ⟩,⟨ 変数 ⟩)

integrate 函数: integrate 函数は通常の式, 方程式 (演算子 “=” を含む式) や, これらのリストや行列の積分も行えます. 一方で, risch 函数は比較の演算子 (“=”, “>”, “<”, “>=” や “<=”) を含まない通常の式の積分に限定されます.
ここで integrate 函数を使って方程式の不定積分を行うと, 積分定数が結果の式に導入されます. この積分定数は順番が付いており, この順番が大域変数 integration_constant_counter に記録されています.
integrate 函数から risch 函数を用いて Risch 積分を実行させることが可能です. このためには ev 函数 (§5.8.3) を用います:

```
(%i21)  ev(integrate(3^log(x),x),'risch);
                                    log(3) log(x)
                                 x %e
```

```
(%o21)                          ----------------
                                  log(3) + 1
(%i22) trigsimp(%);
                                     log(x)
                                  x 3
(%o22)                            ----------
                                  log(3) + 1
(%i23)
```

なお, integrate 函数は depends 函数 で設定される大域変数 dependencies の影響を受けません. integrate 函数は定積分の計算も出来ます. この場合は defint 函数と同じ引数を取ります. ここで実際に定積分を実行するのは defint 函数です. defint 函数による定積分は数値計算を主体としたものではなく, 記号積分を主体としたものです. ここで数値積分が必要な場合は romberg 函数を用います.

広義の定積分では, 正の無限大には定数 'inf', 負の無限大には定数 'minf', 複素無限大には定数 'infinity' が使えます. この定数 'inf' と定数 'minf' については '-inf' と '-minf' がそれぞれ 'minf' や 'inf' と同値ではないことに注意して下さい. Maxima の広義の積分や, その他の代入操作で定数 'inf' と定数 'minf' が利用できますが, '-inf' や '-minf' を用いると全く無意味な結果になることがあるので注意が必要です. そのために, '-inf' や '-minf' が現れる式は limit 函数で必ず評価をしましょう.

積分形式 (例えば, 幾つかの助変数に関してある数値を代入するまで計算出来ない積分) が必要であれば名詞型 'integrate(...) を利用します.

risch 函数: risch 函数は Risch の積分アルゴリズムから, Transcendential case を用いて式の積分を行います. なお, Maxima では Risch アルゴリズムで代数的な場合は実装されていません. また, integrate 函数とは異なり比較の演算子を含む式の積分が出来ません. その一方で risch 函数は integrate の主要部が処理出来ない入れ子状態の指数函数と対数函数の処理が行えます. そして, integrate 函数は与式がこれらの場合には自動的に risch 函数を与式に適用します:

```
(%i24) risch(x^2*erf(x),x);
                                                                  2
                    3                        2                 - x
                %pi x   erf(x) + (sqrt(%pi) x  + sqrt(%pi)) %e
(%o24)          ------------------------------------------------------
                                     3 %pi
(%i25)  diff(%,x),ratsimp;
                                    2
(%o25)                             x   erf(x)
```

7.6.3 integrate 函数と risch 函数に関連する大域変数

integrate 函数と risch 函数に関連する大域変数

変数名	既定値	概要
integration_constant_counter	0	積分定数のカウンター
erfflag	true	risch 函数による erf 函数の挿入を制御

大域変数 integration_constant_counter: 積分定数の番号割当で用いられる大域変数です. この積分定数は integrate 函数で演算子 "=" を含む式の処理を行ったときに現われる定数です: 実際に, その動作を観察してみましょう:

```
%i1) integration_constant_counter;
(%o1)                                  0
(%i2) integrate(x^2,x);
                                        3
                                       x
(%o2)                                  --
                                       3
(%i3) integrate(x^2=0,x);
                                 3          /
                                x           [
(%o3)                           -- = %c1 + I 0 dx
                                3           ]
                                            /
(%i4) integration_constant_counter;
(%o4)                                  1
(%i5) integrate(x^3=0,x);
                                 4          /
                                x           [
(%o5)                           -- = %c2 + I 0 dx
                                4           ]
                                            /
(%i6) integration_constant_counter;
(%o6)                                  2
```

大域変数 erfflag: risch 函数の動作を制御する大域変数で, 大域変数 erfflag が 'false; であれば erf 函数が被積分函数の中に含まれていない場合に risch 函数が答に erf 函数を入れることを抑制します.

7.6.4 changevar 函数による変数変換

changevar 函数を用いることで被積分函数の変数を新しい変数で置換えて積分ができます. この changevar 函数の構文を纏めておきましょう:

変数変換を行う函数

changevar(⟨ 式 ⟩,⟨$f(x,y)$⟩,⟨y⟩,⟨x⟩)

changevar 函数は ⟨ 式 ⟩ に現われる全ての ⟨x⟩ に対する積分で ⟨$f(x,y)$⟩ = 0 を満す ⟨y⟩ を新しい変数とする変数変換を行います:

```
(%i17) assume(z>0)$
(%i18) I1:'integrate(%e^sqrt(1-y),y,0,1);
                                  1
                                 /
                                 [      sqrt(1 - y)
(%o18)                           I   %e               dy
                                 ]
                                 /
                                  0
(%i19) changevar(I1,y+z^2-1,z,y);
                                       0
                                      /
                                      [           z
(%o19)                           - 2  I      z %e   dz
                                      ]
                                      /
                                       - 1
(%i20) ev(%,risch);
                                            - 1
(%o20)                           - 2 (2 %e      - 1)
```

changevar 函数は総和 (∑) や積 (∏) の添字の変更にも使えます. この場合は添字の変更は単純にずらすだけで, 高度な函数に切換える訳ではありません:

```
(%i3) sum(a[i]*exp(i-5),i,0,inf);
                                  inf
                                  ====
                                  \          i - 5
(%o3)                              >     %e        a
                                  /                 i
                                  ====
                                  i = 0
(%i4) changevar(%,i-5-n,n,i);
                                 inf
                                 ====
                                 \           n
(%o4)                             >       %e   a
                                 /                n + 5
                                 ====
                                 n = - 5
```

7.6.5 有理式の記号積分

多項式 $f(x), g(x)$ の有理式 $\frac{f(x)}{g(x)}$ の積分は比較的容易に行えますが, $\frac{1}{x^3-x^2-x+4}$ のように式の因数分解が容易に出来ない式の積分はそのままでは上手く計算できません:

```
(%i3) integrate(1/(x^3-x^2-x+4),x);
                                 /
                                 [          1
(%o3)                            I ---------------- dx
                                 ]  3    2
                                 / x  - x  - x + 4
```

このような有理式の記号積分に対しては大域変数 integrate_use_rootsof を true に設定することで Maxima に代数的数を導入し, これによって形式的な積分が可能となります:

代数的数の利用を制御する大域変数

変数名	既定値	概要
integrate_use_rootsof	false	代数的数を用いた integrate 函数による積分を実行

先程の例に対して大域変数 integrate_use_rootsof を true にし, integrate 函数を使って再度積分を行ってみましょう:

```
(%i4) integrate_use_rootsof:true;
(%o4)                                true
(%i5) integrate(1/(x^3-x^2-x+4),x);
                 ====
                 \            log(x - %r1)
(%o5)             >       ------------------
                 /                 2
                 ====    3 %r1  - 2 %r1 - 1
                                3     2
                 %r1 in rootsof(x  - x  - x + 4)
```

今度は積分の計算が出来ていますようですね. では, この処理の内容を吟味してみましょう. まず, 大域変数 integrate_use_rootsof が true に設定されると Maxima の integrate 函数は与式の分母 x^3-x^2-x+4 から方程式 $x^3-x^2-x+4=0$ の根を形式的に決めます. これは単純に%r1 等の%r で開始する変数を導入することで行います.

この例では分母の多項式は三次式となるために重複解を含めて根が 3 個あり, それらを α,β,γ とします. そして, これらの根を用いると与式を $\frac{1}{(x-\alpha)(x-\beta)(x-\gamma)}$ と形式的に因子分解できます. このように分母が一次式の有理式の積になると, このような有理式の積分はは簡単に計算できます. 実際の処理では各根に記号を割当てる必要はなく, 方程式 xxx の解 “%r2” といった括り方で十分です. ただし, この方法の気持ちの悪さは方程式の根%r1 が不明瞭ながら式に存在することです. そして, 返却される式は名詞型であるために安易に微分が出来ません. 実際, この処理は単純に公式に当て嵌めているだけで, その意味では形式的な処理だからです.

7.6.6　記号積分の検証について

記号積分の計算は記号微分と比べると格段に難しい問題のために比較的単純な式の計算でも思わぬ結果を得ることがあります. そのために記号積分の結果は検算を行うことを強く薦めます. 最も簡単な方法は積分した結果を微分して同じ結果が得られるかどうかを確認することです.
現在では修正されていますが, Maxima-5.10.0 で integrate 函数や risch 函数を使っても上手く計算出来ない簡単な式の例として $\sqrt{x+\frac{1}{x}-2}$ を挙げておきます. この式は $\sqrt{\frac{(x+1)^2}{x}}$ に変形出来ますが, この式の形の違いで結果が大きく異なります:

```
(%i44)  integrate(sqrt(x+1/x-2),x);
                                    3/2
                                 2 x    - 6 sqrt(x)
(%o44)                           ------------------
                                         3
(%i45)  integrate(sqrt(factor(x+1/x-2)),x);
                                   /
                                   [ abs(x - 1)
(%o45)                             I ---------- dx
                                   ]   sqrt(x)
                                   /
(%i46) assume(x<1 and x>0);
(%o46)                             [x < 1, x > 0]
(%i47)  integrate(sqrt(x+1/x-2),x);
                                    3/2
                                 2 x    - 6 sqrt(x)
(%o47)                           ------------------
                                         3
(%i48) integrate(sqrt(factor(x+1/x-2)),x);
                                     3/2
                                  2 x    - 6 sqrt(x)
(%o48)                          - ------------------
                                          3

(%i49) diff(%,x);
                                                3
                                  3 sqrt(x) - -------
                                              sqrt(x)
(%o49)                          - --------------------
                                          3
(%i50) ratsimp(%);
                                      x - 1
(%o50)                              - -------
                                      sqrt(x)
```

この例では同じ integrate でも $\sqrt{\frac{(x+1)^2}{x}}$ の場合は名詞型を返しており, 何も考えずに $\frac{x-1}{\sqrt{x}}$ の計算を行ってはいません. また, assume 函数を使って論理式 $0 < x < 1$ を文脈に登録した場合でも, $\sqrt{x+\frac{1}{x}-2}$ の integrate の計算は間違っています.
このように Maxima の積分は正しい答を返すとは限りませんが, 内部処理を適切に行う事で正しい答を得ることが可能な場合もあります. これは Maxima に限った話ではなく数式処理一般で言える

ことです. さらに記号積分の結果は面倒でも確認した方が安全な事は強調しておきます.
ではどうして式の形の違いで計算結果に違いが出るのでしょうか？ これは結局, 式の並びの照合等によって処理を行っているために, 式を変形していれば式の並びも異なり, そのために照合もまた違うので, その処理の結果も異なってしまい, 処理の流れが異なってしまうからです. . 数値計算で積分を行う場合, 式の並びは無関係で函数の値を主に見るために, このような現象はまず生じません. 並び照合の結果が違っていても正しく処理が出来ていれば最終的に一致する筈ですが, この例のように何処かの処理を間違えると当然結果が異なります. 積分の計算の場合は特に expand 函数で式を展開したり, factor 函数で因子分解するといった前処理を実行して積分したものと結果を照合することは正しい答を得るために非常に有効な手段です. なお, $\sqrt{\frac{(x+1)^2}{x}}$ の処理は Maxima-5.16.3 では改善されています:

```
(%i24) integrate(sqrt(x+1/x-2),x);
                                /
                                [          1
(%o24)                          I sqrt(x + - - 2) dx
                                ]          x
                                /
(%i25) assume(x<1 and x>0);
(%o25)                            [x < 1, x > 0]
(%i26) integrate(sqrt(x+1/x-2),x);
                                    3/2
                                2 x    - 6 sqrt(x)
(%o26)                        - ------------------
                                        3
```

このように Maxima-5.16.3 の integrate 函数は改善されていますが, risch 函数は相変らずです:

```
(%i27) risch(sqrt(x+1/x-2),x);
                                      2
                                  2 x  - 6 x
(%o27)                            ----------
                                  3 sqrt(x)
```

さて, 検証は結果を微分して一致しさえすればそれで十分そうですが, 実はまだ不十分なのです. たとえば, $\frac{3}{5-4\cos x}$ の積分が該当します:

```
(%i13) integrate(3/(5-4*cos(x)),x);
                                    3 sin(x)
(%o13)                       2 atan(----------)
                                    cos(x) + 1
(%i14) trigsimp(diff(%,x));
                                      3
(%o14)                          - ------------
                                  4 cos(x) - 5
```

積分した結果を微分すると元の式が出ているので, 正しい結果が出ていると判断できそうです. しかし, 残念ながらこの計算は間違っています. 何故でしょう？ ここで被積分函数を良く見て下さい. 被積分函数は滑かな連続函数になりますね, 実際, 分母が 0 になることはなく, 非常に性格の良さそうな函数です. ところが, 結果の方は何故か不連続函数になっています!

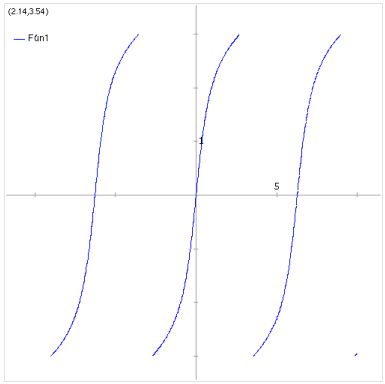


図 7.4: $2\mathrm{atan}\left(\frac{3\sin(x)}{\cos(x)+1}\right)$ のグラフ

このことはグラフを利用して積分した函数を描くと明瞭になります:
このように結果のグラフを描いて見るのも計算結果を確認する上では非常に有効な手段の一つです. このように結果の検証では数値的な側面, 式の形式的な側面や, グラフによる比較といった多目的な検証を行えばより安全です. これは Maxima が Free soft だからではなく, たとえ, それが良質なソフトウエアであったとしても, 利用者の誤用によってとんでもない結果を無批判に受け入れることを避けるためには必要なことなのです.

7.6.7　defint 函数

Maxima は積分処理の為の函数を幾つか持っています. その中でも integrate 函数は最もよく使われるものです. この integrate 函数は不定積分も定積分も処理出来ます. 定積分だけであれば defint 函数もあります. さらに極限操作が必要な場合には ldefint 函数もあります. ただし. この ldefint 函数は曲者で, integrate 函数で記号積分した結果に境界値を代入する代物です. したがって, 区間内に極が存在する場合, その極を検出せずに安易に計算を行うために注意が必要です.
参考までに defint 函数の処理を簡単に説明しておきます. 通常, integrate 函数は LISP 内部で “$integrate” と表記されます. この函数は内部函数 sinint を呼出して, その結果に上限と下限の値を代入しています. この内部函数 sinint でも risch 積分を行う内部函数 rischint を呼出せますが, ev 函数による評価で内部函数 rischint を用いるように指定することが出来ません. この点を修正するためには defint.lisp の antideriv 函数の修正が最低でも必要です.
defint 函数や integrate 函数による定積分の計算では積分の下限や上限に記号や式の正負を尋ねてくることがあります. これは内部の処理で正負の判断が必要となった場合で, 正であれば pos;, 負であれば neg;, 零であれば zero; と入力します. このような正負, あるいは零との同値性に関しては, 内部的に文脈 (§5.6 参照) を用いた処理が行われています. したがって, その大小関係や同値性が明らかな変数に関しては, 予め,assume 函数を用いて文脈に関係を登録しておくことや, 属性を上手く利用しておくことがのちの円滑な処理に繋がります:

```
(%i24) integrate(sqrt(2*x-x^2),x,0,a);
Is a positive, negative, or zero?
```

```
pos;
```

```
        2
                 asin(a - 1) + (a - 1) sqrt(2 a - a )   %pi
(%o24)           ------------------------------------ + ---
                                 2                        4
(%i25) assume(a>0 and a<1);
(%o25)                          [a > 0, a < 1]
(%i26) integrate(sqrt(2*x-x^2),x,0,a);
                                    2
                 (a - 1) sqrt(2 a - a ) - asin(1 - a)   %pi
(%o26)           ------------------------------------ + ---
                                 2                        4
```

7.6.8 Laplace 変換に関連する函数

Maxima には Laplace 変換を行う函数として次の三種類の函数があります:

Laplace 変換に関連する函数
laplace(〈式〉,〈旧変数〉,〈新変数〉)
ilt(〈式〉,〈旧変数〉,〈新変数〉)
delta (t)

laplace 函数: 〈旧変数〉と〈新変数〉に対する〈式〉の Laplace 変換を計算します. なお, 逆 laplace 変換は ilt 函数で行えます.
ここで〈式〉は多項式に函数 exp,log, sin,cos,sinh,cosh と erf 函数を含むもの, atvalue の従属変数が使われている定数係数の線形微分方程式でも構いません.
初期条件は零で指定されていなければならないので, 他の一般解の何処かに押込める境界条件があれば, その境界条件に対して一般解を求めて値を代入して定数消去が出来ます.
〈式〉に畳込み (convolution integral) を含んでいても構いません. laplace 函数が適切に動作するためには函数の従属性を明確に表示していなければなりません. つまり, 函数 f が変数 x と y に従属していれば, laplace('diff(f(x,y),x),x,s) のように函数 f が現れる場合はいつでも f(x,y) と記述する必要があります.
なお, laplace 函数は depends 函数で設定される大域変数 dependencies の影響を受けません:

```
(%i106)  laplace(%e^(2*t+a)*sin(t)*t,t,s);
                                  a
                               %e   (2 s - 4)
(%o106)                       ---------------
                                2           2
                              (s  - 4 s + 5)
```

```
(%i107) ilt(%,s,x);
                                    2 x + a
(%o107)                          x %e          sin(x)
```

ilt 函数: 〈新変数〉と〈旧変数〉に対する〈式〉の逆 Laplace 変換を計算する函数です.
ここで〈式〉は分母が一次と二次の因子を持った有理式でなければなりません. ilt 函数による処理を効率的に行うためには有理式の展開を予め実行しておくと良いでしょう.

delta 函数: Dirac の δ 函数です. laplace 函数は δ 函数を認識しています:

```
(%i38) laplace(delta(t-a)*sin(b*t),t,s);
Is  a  positive, negative, or zero?

pos;
                                          - a s
(%o38)                          sin(a b) %e
```

この例では変数 a に関して何らの仮定や割当がないために Is a positive,negative, or zero?と尋ねています. 予め, assume 函数を用いて論理式 'a ¿ 0' を文脈に登録しておいた場合を示しておきます:

```
(%i39) assume(a<0);
(%o39)                                [a < 0]
(%i40) laplace(delta(t-a)*sin(b*t),t,s);
(%o40)                                  0
```

Laplace 変換を用いた常微分方程式の解法:

laplace 函数と ilt 函数の双方による結果に対し, solve 函数や linsolve 函数を上手く使うと単変数の線形常微分方程式や畳込積分方程式を解くことができます:

```
(%i11) 'integrate(sinh(a*x)*f(t-x),x,0,t)+b*f(t)=t^2;
                    t
                   /
                   [                                    2
(%o11)             I  f(t - x) sinh(a x) dx + b f(t) = t
                   ]
                   /
                    0
(%i12) laplace(%,t,s);
                                    a laplace(f(t), t, s)   2
(%o12)        b laplace(f(t), t, s) + --------------------- = --
                                          2    2             3
                                         s  - a             s
(%i13)  linsolve([%],['laplace(f(t),t,s)]);
                                            2      2
                                         2 s  - 2 a
(%o13)           [laplace(f(t), t, s) = --------------------]
                                          5          2     3
                                       b s  + (a - a  b) s
```

```
(%i14) ilt(rhs(first(%)),s,t);
Is  a b (a b - 1)  positive, negative, or zero?

pos;
                  sqrt(a b (a b - 1)) t
          2 cosh(----------------------)        2
                          b                  a t                2
(%o14)  - ------------------------------ + ------- + ------------------
                3  2       2                a b - 1    3  2       2
               a  b  - 2 a  b + a                    a  b  - 2 a  b + a
```

7.6.9 その他の積分に関連する函数

留数を計算する函数と erf 函数

residue(⟨式⟩,⟨変数⟩,⟨点⟩)
erf(x)

residue 函数: ⟨点⟩回りの⟨式⟩の複素平面上での留数を計算します. ここで留数は式の laurent 級数展開を行ったときの $(\langle 変数 \rangle - \langle 点 \rangle)^{-1}$ の項の係数になります:

```
(%i5) residue(x/(x^2+1),x,%i);
                                        1
(%o5)                                   -
                                        2
(%i6) residue(sin(x)/x^4,x,0);
                                         1
(%o6)                                  - -
                                         6
```

erf 函数: 微分 $\frac{d}{dx}(erf(x))$ が $\frac{2e^{-x^2}}{\sqrt{\pi}}$ となる一変数函数です.

7.6.10 定積分を行う函数

定積分を行う函数

defint(⟨式⟩,⟨変数⟩,⟨下限⟩,⟨上限⟩)
ldefint(⟨式⟩,⟨変数⟩,⟨下限⟩,⟨上限⟩)
tldefint(⟨式⟩,⟨変数⟩,⟨下限⟩,⟨上限⟩)

defint 函数:　内部的に integrate を利用する函数で, 原始函数を求めると単純に〈上限〉と〈下限〉の値を代入したものの差を取るものです. ただし, 後述の ldefint と比較して区間 (〈下限〉,〈上限〉) に於ける〈式〉の極の判定を行っているので ldefint 函数による定積分よりは安全です. なお, romberg 等の数値成分による手法の方が間違いが少ないので, その点は注意して使う必要があります.

ldefint 函数:　〈変数〉の〈上限〉と〈下限〉に関する〈式〉の不定積分に対して limit 函数を用いた評価を行い,〈式〉の定積分を計算します. ここで上限と下限に 'minf' と 'inf' といった定数を与える場合には注意が必要です. また, '-inf' や '-minf' の符号を付けた定数は無意味な結果を得ることがあるので注意が必要になります:

```
(%i20) ldefint(exp(-x)*sin(x),x,0,-minf);
                    minf                  minf
                  %e      sin(- minf)   %e      cos(- minf)   1
(%o20)          - ------------------- - ------------------- + -
                          2                     2             2
(%i21) ldefint(exp(-x)*sin(x),x,0,inf);
                                        1
(%o21)                                  -
                                        2
```

なお, ldefint は内部的で函数の極の判別を一切行わずに integrate 函数で安易に記号積分した結果に上限と下限を limit 函数で代入するだけの函数です. これに対し, ldefint 函数は一応, 右極限と左極限を下限と上限で取るようにしていますが, 単純に上限に対しては'minus, 下限に対しては'plus を内部的に付けるだけなので上限や下限で不連続になる函数の場合は上限と下限の大小関係を逆にして ldefint 函数を用いて計算すれば無意味になる可能性もあります.
ここで, defint 函数や integrate 函数で定積分を行う場合は区間内での極の判別も行っていますが, ldefint 函数では内部的に sinint 函数を用いるだけで, この sinint 函数は極の判別を一切行わずに機械的な記号積分を行います. そのため, あまり性質を知らない函数をいきなり ldefint 函数で積分し, その結果の検証を省くことは薦められません.
次の例をよく吟味して下さい:

```
(%i15) ldefint(1/x^2,x,0,1);
                                        1
(%o15)                          (limit  -) - 1
                                 x -> 0 x
(%i16) ldefint(1/x^2,x,-1,0);
                                         1
(%o16)                         - (limit  -) - 1
                                  x -> 0 x
(%i17) %o15+%o16;
(%o17)                                - 2
(%i18) defint(1/x^2,x,-1,1);
Integral is divergent
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i19) ldefint(1/x^2,x,-1,1);
(%o19)                                - 2
```

この例で示すように%i19 の ldefint 函数の結果は-2 となっています. これは Maxima で $\frac{1}{x^2}$ を安易に記号積分し, 区間の上限と下限の極限を取っているために, このような現象が生じています. これに対して defint 函数や integrate 函数では積分を行う領域に極が存在するために, ちゃんとエラーを返しています.

zeroa と zerob: なお, ldefint 函数では特殊な定数 zeroa と zerob が使えます. 定数 zeroa が 0 の右極限で定数 zerob が 0 の左極限を表現します:

```
(%i7) ldefint(1/x^2,x,zeroa,1);
                                  1
(%o7)                        (limit  -) - 1
                              x -> 0+ x
(%i8) ldefint(1/x^2,x,zerob,-1);
                                  1
(%o8)                        (limit  -) + 1
                              x -> 0- x
```

ただし, 1+'zeroa のような使い方は出来ないので注意します.
そして, ldefint 函数は defint 函数と同様に積分で Risch 積分が使えません.

tldefint 函数: この函数の処理は大域変数 tlimswitch が true に設定された ldefint 函数に相当します. なお, 大域変数 tlimswitch が true の場合に極限の計算で Taylor 展開を利用することを意味します.

7.6.11 数値積分について

Maxima には記号積分だけではなく数値計算による定積分の計算も行えます. この処理を行う函数に romberg 函数があります. そして, 数値的に重積分を行う函数として dbint 函数もあります:

数値積分を行う函数

romberg(⟨ 被積分函数 ⟩,⟨ 積分変数 ⟩,⟨ 実数$_1$⟩,⟨ 実数$_2$⟩)
romberg(⟨ 被積分函数 ⟩,⟨ 実数$_1$⟩,⟨ 実数$_2$⟩)
dblint('⟨F⟩,'⟨R⟩,'⟨S⟩,⟨ 浮動小数点$_1$⟩,⟨ 浮動小数点$_2$⟩)

romberg 函数: この函数は romberg 積分を行う函数です. 最初の引数は translate 函数で変換された函数か compile 函数でコンパイルされた函数でなければなりません. なお, compile 函数でコンパイルされた函数の場合は浮動小数点数を返す函数, すなわち, flonum 型の函数として宣言されていなければなりません. 函数が translate 函数で変換されたものでなければ romberg 函数は translate 函数で変換せずにエラーを返します. 積分の精度は大域変数 rombergtol と大域変数 rombergit で制御されます.
この romberg 函数は再帰的に呼び出されていても構わないので, 二重, 三重積分が実行可能です.
なお, romberg 函数の ⟨ 実数$_1$⟩ と ⟨ 実数$_2$⟩ は内部で倍精度の浮動小数点を用いているため, 多倍長精度 (bigfloat 型) に変換した数値は扱えません.

相対誤差が大域変数 rombergtol よりも小さければ romberg 函数は計算結果を返します. 諦める前に大域変数 romgergit 倍の刻幅を半分にして試みます. romberg 函数は反復と函数評価の大域変数 romgergabs と rombergmin で制御されます:

```
(%i43) showtime:all;
Evaluation took 0.0000 seconds (0.0001 elapsed) using 56 bytes.
(%o43)                                  all
(%i44) romberg(sqrt(2*x-x^2),x,0,1);
Evaluation took 0.9761 seconds (1.3870 elapsed) using 14.122 MB.
(%o44)                          .7853897937007632
(%i45) integrate(sqrt(2*x-x^2),x,0,1);
Evaluation took 0.0840 seconds (0.0816 elapsed) using 928.937 KB.
                                        %pi
(%o45)                                  ---
                                         4
(%i46) bfloat(%);
Evaluation took 0.0000 seconds (0.0001 elapsed) using 1.312 KB.
(%o46)                      7.853981633974483b-1
(%i47) bfloat(integrate(sqrt(2*x-x^2),x,0,1));
Evaluation took 0.0800 seconds (0.0819 elapsed) using 930.227 KB.
(%o47)                      7.853981633974483b-1
(%i48)  romberg(exp(-x)*sin(x),x,0.,1.);
Evaluation took 0.1120 seconds (0.1123 elapsed) using 290.211 KB.
(%o48)                        .2458370426035679
(%i49) bfloat(integrate(exp(-x)*sin(x),x,0.,1.));
Evaluation took 0.0400 seconds (0.0375 elapsed) using 300.836 KB.
(%o49)                      2.458370070002374b-1
```

この romberg 函数に影響を与える大域変数を次に纏めておきましょう:

romberg 函数に影響を与える大域変数

変数名	既定値	概要
rombergabs	0.0	romberg 函数の終了条件を与える変数
rombergit	11	刻幅を設定
rombergtol	1.0E-4	romberg 函数の精度
rombergmin	0	函数評価の最小回数

大域変数 rombergabs:　romberg 函数の終了条件を与える大域変数の一つです. romberg 函数内部の反復処理で生成された値の列を y[0], y[1], y[2], ··· とするときに n 回目の反復処理で,(abs(y[n] - y[n-1])≤ rombergabs か abs(y[n]-y[n-1])/(y[n]=0.0 であれば 1.0, それ以外は y[n] ≥ rombergtol を満した時点で romberg 函数は答を返します. そのために rombergabs が 0.0 であれば相対誤差の検証が出来ます. この追加変数は小さな値域で積分計算を行いたいときに便利です. そこで, 小さな主要な値域で最初に積分する事で相対的精度検証を用い, あとに続く残りの値域上の積分は絶対的精度の検証で用います.

大域変数 rombergtol と大域変数 rombergit　これらの大域変数は romberg 積分命令の精度を制御します. romberg 函数は隣り合った近似解での相対差が rombergtol よりも小さければ値を返し

ます. 上手く行かない場合には計算を諦める前に刻幅の romgergit を半分にして再度試行します.

大域変数 rombergmin romberg 函数による函数評価の最小数を制御します. romberg 函数はその第 1 引数を少なくとも $2^{\text{rombergmin}+2}+1$ 回評価します. これは通常の収束検定が時々悪い通り方をする周期的函数の積分に有効です.

dblint 函数: dbint 函数は二重積分の数値計算を行う函数で, Maxima の処理言語で記述されています. この函数を利用するためには予め load(dblint) で dbint 函数を読込んでおく必要があります. なお, dblint('F,'R,'S,浮動小数点$_1$,浮動小数点$_2$) で次の二重積分を計算します:

$$\int_{\text{浮動小数点}_1}^{\text{浮動小数点}_2}\int_{R(x)}^{S(x)} F(x,y)dydx$$

第 1 引数の ⟨*F*⟩ は x,y の 2 変数の函数, 第 2 引数と第 3 引数の ⟨R⟩ と ⟨S⟩ はそれぞれ変数 x の函数で, dblint 函数には函数名のみを名詞型で引渡します. さらに, これらの函数は全て translate 函数で LISP の函数に変換されたものか, compile 函数でコンパイルされたものでなければなりません. そのために, これらの函数は全て flonum 型の函数でなければならず, 多倍長精度は扱えません.
dblint 函数は ⟨R⟩ と ⟨S⟩ の両方に Simpson 則を用います. そして, dblint 函数は二つの大域変数 dblint_x, dblint_y を持ち, それぞれの大域変数が x と y の区間の分割数を定めます. なお, 収束性を改善するために大域変数 dblint_x と dblint_y を大きくした場合は計算時間が増大します.
dblint 函数は X 軸を大域変数 dblint_x の値で分割し, X 軸上の各値に対して最初に R(x) と S(x) を計算します. それから,R(x) と S(x) の間の Y 軸を大域変数 dblint_y の値で分割し, Y 軸に沿って積分を Simpson 則を用いて計算します.
それから, X 軸に沿った積分を函数の値を Y の積分で Simpson 則を用いて計算します. この手順は色々な理由で数値的に不安定であるが, それなりに速いものです.
ただし, 高周波成分を持った函数や特異点 (領域に極や分岐点) を持つ函数に対する適用は避けて下さい.
Y 軸方向の積分は R(x) と S(x) がどれだけ離れているかに依存し, 距離 S(X)-R(X) が X で急速に変化すれば Y 軸方向の積分で大きな誤差が発生するかもしれません.
函数値は保存されないために, その函数の計算に時間がかかるものであれば何かを変更する度に再計算する羽目になるので, その分時間がかかります.

dblint 函数に影響を与える大域変数: 二重積分を行う函数 dbiint 函数は二つの大域変数 dblint_x,dblint_y を持っています:

dblint 函数に影響を与える大域変数

変数名	既定値	概要
dblint_x	10	X 軸方向の分割数
dblint_y	10	Y 軸方向の分割数

これらの大域変数は X, Y の区間の分割数を定め,$2dblint\_x+1$ 個の点が X 方向に計算され,Y 方向は $2dblint\_y+1$ 個の点となります. これらの変数は勿論, 互いに独立して変更できます.

7.6.12　antid パッケージ

antid パッケージには antid 函数, antidiff 函数と nonzeroandfreeof 函数といった函数が含まれたパッケージです.

antidiff と antid

antid(⟨ 式 ⟩ ,⟨x⟩ ,⟨$u(x)$⟩)
antidiff(⟨ 式 ⟩ ,⟨x⟩ ,⟨$u(x)$⟩)
nonzeroandfreeof(⟨x⟩,⟨y⟩)

antid 函数:　与式の不定積分を行います. antid 函数の返却値は二成分のリストで, このリスト L とするときに L[1]+'integrate(L[2],x) が与式の不定積分となります. ここで, 函数 ⟨$u(x)$⟩ は未知函数でも構いません:

```
(%i8) load(antid);
(%o8)         /usr/local/share/maxima/5.9.2/share/integration/antid.mac
(%i9) antid(sin(3*x+1),x,3*x+1);
                                      cos(3 x + 1)
(%o9)                               [- ------------, 0]
                                            3
(%i10) antid(sin(u(x)+1),x,u(x));
(%o10)                              [0, sin(u(x) + 1)]
```

antidiff 函数:　内部で antid 函数を用いた函数で, antid 函数で不定積分した結果を L[1]+'integrate(L[2],x) の形式に変換て表示します:

```
(%i11) antidiff(sin(3*x+1),x,3*x+1);
                                        cos(3 x + 1)
(%o11)                                - ------------
                                              3
(%i12) antidiff(sin(u(x)+1),x,u(x));
                                      /
                                      [
(%o12)                                I sin(u(x) + 1) dx
                                      ]
                                      /
```

函数 nonzeroandfreeof:　真理函数で,⟨y⟩ が零でなく, ⟨x⟩ が ⟨y⟩ に含まれない場合に true を返します. この真理函数を用いて antid 函数と antidiff 函数は規則を定めています.

7.7 常微分方程式

7.7.1 常微分方程式の書式

常微分方程式の書式は, 通常の代数方程式と同様に演算子 “=” を含み, 最低一つの形式的な函数の微分項が含まれる式です. ここで微分項は形式的な函数に対する微分のために自動的に名詞型になります.
Maxima には常微分方程式の一般解を計算する函数として ode2 函数と desolve 函数の二つがあります. ここで, desolve 函数で扱える常微分方程式は未知函数がどの変数に依存するものかを明示的に記述したものでなければなりません.
たとえば, 次の書式は desolve 函数にとっては正確な書式ではありません. 何故なら, 函数 f と函数 g の変数が何であるかが明確でないからです:

```
'diff(f,x,2)=sin(x)+'diff(g,x);
'diff(f,x)+x^2-f=2*'diff(g,x,2);
```

desolve 函数の場合, 常微分方程式の未知函数が何の函数であるかを下記の様に明確に記述しなければなりません.

```
'diff(f(x),x,2)=sin(x)+'diff(g(x),x);
'diff(f(x),x)+x^2-f(x)=2*'diff(g(x),x,2);
```

desolve 函数に対して ode2 函数は x^2*'diff(y,x)+3*y*x=sin(x)/x の様に函数と変数との関係を明記しなくても構いません.

7.7.2 常微分方程式の解法

desolve 函数による解法

最初に desolve 函数を用いて簡単な例を解いてみましょう:

```
(%i66) eq1:'diff(y(x),x,2)+2*x=sin(x);
                                2
                               d
(%o66)                         --- (y(x)) + 2 x = sin(x)
                                 2
                               dx
(%i67) atvalue('diff(y(x),x),x=0,0);
(%o67)                                 0
(%i68) atvalue(y(x),x=0,0);
(%o68)                                 0
(%i69) desolve([eq1],[y(x)]);
                                                 3
                                                x
(%o69)                      y(x) = - sin(x) - -- + x
                                                3
```

desolve 函数を用いて解く場合, 初期値は atvalue 函数を用いて未知函数の属性として与えます. この場合は常微分方程式を $\dfrac{dy(x)}{dx} + 2x = \sin(x)$ とし, その初期条件 (境界条件) を $\left[y(0) = 0, \dfrac{dy(x)}{dx}|_0 = 0\right]$ としています.

ode2 函数による常微分方程式の解法

ode2 函数を用いる場合, 与えられた一般解と初期条件から特殊解を計算する bc 函数, ic1 函数や ic2 函数を併用します:

```
(%i14) x^2*'diff(y(x),x)+3*y(x)*x=sin(x)/x;
                                  2  d                          sin(x)
(%o14)                           x  (-- (y(x))) + 3 x y(x) = ------
                                    dx                            x
(%i15)  ode2(%,y(x),x);
                                                  %c - cos(x)
(%o15)                                  y(x) = -----------
                                                      3
                                                     x
(%i16) ic1(%o15,x=%pi,y(%pi)=0);
                                                  cos(x) + 1
(%o16)                                  y(x) = - ----------
                                                      3
```

この例では微分方程式 $x^2\dfrac{dy}{dx} + 3xy = \dfrac{\sin(x)}{x}$ を ode2 函数を用いて, その一般解を計算し, それから ic1 函数を用いて $x = \pi$ の場合に y が 0 となる条件で特殊解を求めています.
では変数が二つの場合, 特殊解をどの様にして得るのでしょうか？この場合, 特殊解は bc1 函数ではなく, bc2 函数を用いて求めます:

```
(%i91) 'diff(y,x,2) + y*'diff(y,x)^3 = 0;
                                    2
                                   d y        dy 3
(%o91)                             --- + y (--)  = 0
                                     2        dx
                                   dx
(%i92) ode2(%,y,x);
                                  3
                                 y  + 6 %k1 y
(%o92)                           ------------ = x + %k2
                                      6
(%i93) ratsimp(ic2(%o92,x=0,y=0,'diff(y,x)=2));
                                          3
                                     2 y  - 3 y
(%o93)                             - ---------- = x
                                          6
(%i94) bc2(%o92,x=0,y=1,x=1,y=3);
                                    3
                                   y  - 10 y        3
(%o94)                             --------- = x - -
                                       6            2
```

7.7.3 常微分方程式の一般解を求める函数

一般解を求める函数の構文

desolve([⟨ 方程式$_1$⟩ ,···,⟨ 方程式$_n$⟩], [⟨ 変数$_1$⟩ ,···,⟨ 変数$_n$⟩])
ode2(⟨ 常微分方程式 ⟩ ,⟨ 従属変数 ⟩ , ⟨ 独立変数 ⟩)

desolve 函数: 常微分方程式系を与えられた変数に対して解きます. 常微分方程式の初期条件を設定する場合, desolve 函数を呼出す前に atvalue 函数で与えなければなりません.
desolve 函数の引数としては ⟨ 方程式$_i$⟩ で構成されるリストと従属変数 ⟨ 変数$_1$⟩ ,··· ⟨ 変数$_n$⟩ のリストを指定します.
なお, desolve 函数では函数と変数の関連性を明確に指定しなければなりません. この方法としては, 'depends(f,x)' の様に予め depend 函数を用いて未知函数の変数への従属性を属性として与えるか, 'f(x)' の様に明示的に表記するか何れかを行う必要があります.
解はリストの形式で返却されますが, 解を得られなかった場合に desolve 函数は 'false' を返します.
ここで微分方程式の検証では ev 函数 (§5.8.3 参照) を用いると効率良く作業が行えます:

```
(%i1) 'diff(f(x),x)='diff(g(x),x)+sin(x);
                          d             d
(%o1)                     -- (f(x)) = -- (g(x)) + sin(x)
                          dx            dx
(%i2) 'diff(g(x),x,2)='diff(f(x),x)-cos(x);
                            2
                          d               d
(%o2)                     --- (g(x)) = -- (f(x)) - cos(x)
                            2             dx
                          dx
(%i3) atvalue('diff(g(x),x),x=0,a);
(%o3)                                    a
(%i4)  atvalue(f(x),x=0,1);
(%o4)                                    1
(%i5) desolve([%o1,%o2],[f(x),g(x)]);
                       x                                  x
(%o5)     [f(x) = a %e  - a + 1, g(x) = cos(x) + a %e  - a + g(0) - 1]
(%i6) [%o1,%o2],%o5,diff;
                     x        x       x                  x
(%o6)           [a %e  = a %e  , a %e  - cos(x) = a %e  - cos(x)]
```

この例で '[%o1,%o2],%o5,diff;' は ev 函数の一つの構文です. ここでは, 式 '[%o1,%o2]' に f(x) と g(x) を代入して, ラベル%o1 とラベル%o2 に割当てられた式で微分の名詞型を評価させることで, desolve 函数による結果の検証を行っています. この評価の結果, ラベル%o6 の出力結果で演算子 "=" の両辺の式が等しいことから, 正解であることが判ります.

ode2 函数: ode2 函数は 3 個の引数を取ります. 最初の ⟨ 常微分方程式 ⟩ は一階, または二階の常微分方程式を与えます. なお, 常微分方程式の右側 (rhs 函数で取出せます) が '0' ならば, 左側のみを与えるだけでも構いません. 第 2 引数には ⟨ 従属変数 ⟩, 最後の引数が ⟨ 独立変数 ⟩ となります.

求解に成功すると, 従属変数に対する陽的な解, あるいは陰的な解の何れかを返します. ここで記号 “%c” は一階の方程式の定数, 記号 “%k1” と記号 “%k2” は二階の方程式の定数を表記するために用いられます.
ode2 函数が何らかの理由で解が得られなかった場合, エラーメッセージの表示等のあとに ‘false’ を返します.
現在, 一階常微分方程式向けに実装され, 検証されている解法は, 線型, 分離法, 厳密 (積分因子が多分要求されます), 同次, bernoulli 方程式と一般化同次法があります.
二階の常微分方程式に対しては定数係数, 厳密, 定数係数に変換可能な非定数係数を持つ線型同次方程式, Euler, または同次元方程式, 仮想変位法, そして変形分離で解ける二つの独立な一階の線型な方程式に縮約可能となる方程式を含まないものがあります.
常微分方程式を解く手順では, 幾つかの変数は純粋に情報的な目的, method が記述する解法の集合です. たとえば, linear, intfactor が記述する積分因子を用い, odeindex は Bernoulli 法や一般化同次法の添字を記述し, yp は仮想変位による特殊な解法を記述しています..

初期値の与え方

atvalue 函数:　desolve 函数を用いて初期値問題を解く場合, atvalue 函数を用いて初期値を函数の属性として与えます. ここで atvalue 函数による初期値設定は desolve 函数で微分方程式を処理する前に行わなければ意味がありません.
この atvalue 函数の構文を以下に示しておきます:

atvalue 函数の構文

atvalue(⟨ 式 ⟩, [⟨x_1⟩ = ⟨a_1⟩, ··· , ⟨x_n⟩ = ⟨a_n⟩], ⟨c⟩)
atvalue(⟨ 式 ⟩, ⟨x⟩ = ⟨a⟩, ⟨c⟩)

この atvalue 函数は多変数函数に対し, 第 2 引数に ‘[⟨x_1⟩ = ⟨a_1⟩, . . . , ⟨x_n⟩ = ⟨a_n⟩]’ の書式で点を指定し, 第 3 引数の ⟨c⟩ にその点での値を設定します.
ここで 1 変数函数の場合は第 2 引数をリストの形式ではなく, ‘⟨x⟩ = ⟨a⟩’ の書式にしても構いません.

ode2 函数と併用する函数

境界値問題を解く函数

bc2(⟨ 一般解 ⟩ ,⟨x の値$_1$⟩, ⟨y の値$_1$⟩, ⟨x の値$_2$⟩ ,⟨y の値$_2$⟩)
ic1(⟨ 一般解 ⟩ ,⟨x の値 ⟩ , ⟨y の値 ⟩)
ic2(⟨ 一般解 ⟩ ,⟨x の値 ⟩ , ⟨y の値 ⟩,⟨y の微分値 ⟩)

bc2 函数:　二階の微分方程式の境界条件問題を解きます. ここで ⟨ 一般解 ⟩ は ode2 函数等で計算した微分方程式の一般解です. 二階の方程式の一般解では定数が二つ現われるため、 その特殊解を求めるためには異なった 2 点での連立方程式を解く必要があります.

そこで ⟨x の値$_1$⟩ と ⟨y の値$_1$⟩ が一つの点での値, そして, ⟨x の値$_2$⟩ と ⟨y の値$_2$⟩ がもう一つの別の点での値を定めます. ここで値の与え方は一般解の変数 x, y に対し, ⟨x の値$_1$⟩ と ⟨y の値$_1$⟩ を 'x=x0' や 'y=y0' の様に ⟨ 対応する変数 ⟩ =⟨ 境界値 ⟩ の書式で記述します.

ic1 函数: 初期値問題 (ivp) と境界値問題 (bvp) を解くための ode2 パッケージに含まれる函数です. ⟨ 一般解 ⟩ は ode2 等で計算した微分方程式の一般解です. そして, あとの二つが境界条件を与えます. 一般解の変数を x, y とすると, ⟨x の値$_1$⟩ と ⟨y の値$_1$⟩ は $x = x0$ や $y = y0$ の様に ⟨ 対応する変数 ⟩ =⟨ 境界値 ⟩ の書式になります.

ic2 函数: 二階の常微分方程式の境界値問題を解く函数です. ⟨ 一般解 ⟩ は ode 函数等で計算した微分方程式の一般解となり, うしろの二つがその境界条件を与えます.
一般解の変数を x , y とすると, ⟨x の値 ⟩ と ⟨y の値 ⟩ は, x が ⟨x の値 ⟩ の場合, y の値が ⟨y の値 ⟩ で y を x で微分した函数の, x = ⟨x の値 ⟩ での値が ⟨y の微分値 ⟩ となります.

第8章　プログラム

Sechste Stimme	**第六の声**
Faust, ich bin geschwind als wie des Menschen Gedanke.	ファウストよ, 俺の速さは人間の思考と同じだ.
Faust	**ファウスト**
Als wie des Menschen Gedanke?	人間の思考と同じだと？
Was will ich mehr?	何をそれ以上？
Konnt' ich so viel erhoffen?	それ程の事が望めるのか？
Was will ich mehr denn!	何をそれ以上望もうか!
Als daß Erfüllung schreite mit dem Wunsche,	望めば忽ち叶えられ,
als daß die Tat zugleich ins Leben trete mit der Absicht:	意図すれば直ちに行為となる:
Konnt' ich so viel erhoffen!	それ程の事が期待出来るのか!
Was will ich mehr?	何をそれ以上？
Dein Name?	お前の名は？
Sechste Stimme	**第六の声**
Mephistopheles.	メフィストフェレス.

Buzoni,Doktor Faust[101] より

この章では,Maxima のプログラムに関連した事柄について解説を行います.
Maxima 上でのプログラムでは,Maxima の PASCAL 風の処理言語を用います. この処理言語を用いる事で, 利用者定義の Maxima の函数を構築する事も可能です.
Maxima 言語で記述された利用者定義函数は, 実行時には LISP で解釈され,LISP で処理されます. その為,LISP による解釈で速度の遅延が生じます. この為, 利用者定義函数を LISP の函数に変換する translate 函数や,LISP のコンパイラを利用して高速化を図る optimize 函数があります.

8.1 Maxima でプログラム

Maxima の制御文には if 文, 反復処理に do ループと go といった原始的な構文があります. そして, block 文を用いることで局所変数が利用できます.
Maxima の処理言語は C や PASCAL のような手続型の言語としての側面を持ちます. しかし, Maxima の対象はプログラムも含めて全て S 式として表現されることから, 内部表現を利用した方が効率的なプログラムが記述し易い側面もあります. そして, Maxima 言語で記述した函数等の対象は Maxima の式であることを表現する独特の S 式で表記されており, これを LISP の函数に翻訳することで一層の処理速度向上も望めます.

8.1.1 block 文

block 文

block(⟨ 変数リスト ⟩,⟨ 式$_1$⟩,⟨ 式$_2$⟩,...,⟨ 式$_n$⟩)

Maxima の block は FORTRAN の subroutine, PASCAL の procedure に似た対象です. block 文は文の集合体ですが, block 内部の文にラベル付けが行え, 局所変数も扱えます. 局所変数は block 文外部にある同名の大域変数との名前の衝突を避けることに使えます. 同名の大域変数が存在した場合, block 文の実行中に, その変数はスタックに保存されるので, その間は参照出来ません. block 文が終了した時点でスタックに保存さていた値がもとに戻されますが, block 文で用いた局所変数の値は破棄されます.
なお, 局所変数を用いなければ block 文で局所変数のリストを省略したり, 空のリスト "[]" で置換えても構いません. さらに局所変数に初期値を設定することもできます. この場合, '[a:1,b:2]' のように局所変数に対してコロン ":" による通常の割当を行います.
ここで, block 内部で用いられるものの, block 文の局所変数リストに含まれていない変数は block 文の外部で用いられている変数と同様に大域変数として扱われます. そのために, その変数に割当てられた値は block 文の終了後も保持されます.
block 文の返却値は block 文に return 函数を含まない場合には最後の文の値, block 文が return 函数を含む場合は return 函数に渡された引数の値となります.
函数 go は制御を go の引数でラベル付けされた block 内の文に移動するために使えます. 文のラベル付けは block 内部で原子を文の前に置くことで対処します. たとえば, 'block(⟨ 式$_1$⟩, ..., ⟨ 式$_n$⟩, loop,⟨ 式 ⟩,..., go(loop),...)' のように, ラベル "loop" と go 函数の対で使います. ここで go 函数の引数として与えられるラベル名は, この go 函数を包含する block 内部で現われるラベル名でなければなりません. すなわち, go 函数で飛ぶことのできるラベルは go 函数が包含される block 文に存在するものに限定されます.
return 函数は block 文の変数リスト以外の何処にでも置けます. return 函数を置かなかった block 文では, その block 文の末尾の式の値が返却値となります.
block 文を用いた函数の例を次に示します. この函数では局所変数として, a, k を用いています:

```
(%i67) a:10;
(%o67)                                10
```

```
(%i68) neko(x):=block([a:2,k],k:sin(x)*a,return(k));
(%o68)        neko(x) := block([a : 2, k], k : sin(x) a, return(k))
(%i69) neko(10);
(%o69)                                  2 sin(10)
(%i70) a;k;
(%o70)                                     10
(%i71) k;
(%o71)                                      k
```

この例で示すように局所変数は block 文内部でのみ利用され, 同名の変数には影響を与えていません.

8.1.2 block 文内部で利用可能な函数

block 文内部で利用する函数

go(⟨ ラベル ⟩)
return(⟨ 式 ⟩)

go 函数: block 内部で指定した block 文中のラベルに移動する目的で用いられます. ラベルとして原子を用い, この原子は文の前に置きます:

```
block([x],
      x:1,
  loop,x+1,
      ...,
      go(loop),
      ...)
```

go 函数の引数は同じ block 文の中で現われるラベルでなければなりません. go 函数を含む block 文とば別の block 文中のラベルに移動することはできません.

return 函数: block 文から引数の値を伴って抜けるときに用います. return 文の位置は block 文の何処でも構いません.

8.1.3 if 文

if 文は条件分岐で用います. その構文は C 等の言語と違いはありません.

if 文の構文

if ⟨ 述語 ⟩ then ⟨ 式$_1$⟩ else ⟨ 式$_2$⟩

if 文は ⟨ 述語 ⟩ を評価し, 'true' であれば ⟨ 式$_1$⟩ を実行し, 'false' であれば ⟨ 式$_2$⟩ を実行します.
⟨ 述語 ⟩ はその値が 'true', または 'false' であるかが評価可能な式で, 真理函数や演算子等で構成されたものです.

if 文で利用可能な演算子

演算子	記号	種類
等しい	=, equal	中置式演算子 (infix)
等しくない	#	中置式演算子 (infix)
大きい	>	中置式演算子 (infix)
小さい	<	中置式演算子 (infix)
以上	>=	中置式演算子 (infix)
以下	<=	中置式演算子 (infix)
論理積	and	中置式演算子 (infix)
論理和	or	中置式演算子 (infix)
否定	not	前置式演算子 (prefix)

なお, 前置式演算子 (prefix) と中置子演算子 (infix) は, 演算子を引数の置き方で区分した演算子です. 詳細は §5.3 を参照して下さい.
これに対し, ⟨ 式$_1$⟩ と ⟨ 式$_2$⟩ は任意の Maxima の式 (勿論, if 文を含み, 入れ子になっても構いません) が利用可能です.

8.1.4　do 文による反復処理

Maxima で反復処理は do 文を用います. この do 文の基本的な構文を示しておきます:

do 文の基本構造

for ⟨ 制御変数処理 ⟩ ⟨ 終了条件 ⟩ do ⟨ 本体 ⟩

do 文には do 文内部のみで用いられる局所変数があり, これを制御変数と呼びます. この制御変数は do 文の中だけで効力を持ちます.
do 文では終了条件の判定を行い, 終了条件を満さなければ do 文の本体を実行し, それから制御変数処理で制御変数に変更を加えて終了条件の判定という反復処理を行います.
最初に ⟨ 制御変数処理 ⟩ の個所について述べます.
制御変数処理では最初に制御変数に初期値を割当て, 次に反復処理で再度回って来ると制御変数に新しい値を割当てます.

制御変数への割当

制御変数の初期値の割当には次の二つの同値な書き方があります.

制御変数の初期値の割当

⟨ 変数 ⟩ : ⟨ 初期値 ⟩
⟨ 変数 ⟩ from ⟨ 初期値 ⟩

この制御変数の初期値の割当はどちらを用いても構いませんが, 初期値が '1' の場合は ": ⟨ 初期値 ⟩" や "from ⟨ 初期値 ⟩" を省略できます.

次に, 制御変数を一定の値で増加, あるいは減少させたければ, do 文内部に “step ⟨ 増分 ⟩” を追加します. このとき ⟨ 増分 ⟩ を正実数にすれば通常の増分となり, 負の実数にすれば減少分になります. さらに ⟨ 増分 ⟩ が ‘1’ の場合, “step 1” を省略できます.
制御変数に何等かの函数を割当てたければ, “next ⟨ 制御変数の式 ⟩” とすると制御変数の値に ⟨ 制御変数の式 ⟩ で計算した値が割当てられます.
たとえば, 次の二つの異った書式の反復処理は同値になります.

制御変数の割当方法

```
do i from 1 step 2 ...
do i:1 next i+2 ...
```

これらは全て制御変数 i の初期値を ‘1’, 増分を ‘2’ とする反復処理を実行します. 最初は “step” を用いて増分を定め, 最後は “from” を “next” に置換え, “next” の直後の式 ‘i+2’ で制御変数 ‘i’ の値を定めます.
これらの方法に加えて, リストを用いた制御変数値の割当て方もあります:

リストを用いた制御変数の割当

```
for i in ⟨ リスト ⟩ ...
```

この場合, リストの成分に数値以外の函数や式が与えられます:

```
(%i10) for i in [1,2,3,4,5,6,7,8,9,10] do print(i);
1
2
3
4
5
6
7
8
9
10
(%o10)                                   done
(%i11)for i in [sin,cos,tan] do print(subst(i,f,f(%pi/4)));
sqrt(2)
-------
   2
sqrt(2)
-------
   2
1
(%o17)                                   done
```

この例では最初にリスト [1,2,3,4,5,6,7,8,9,10] の元を表示し, 最後の例では,f(%pi/4) の f に sin,cos,tan を順番に代入した結果を表示させています.

終了条件の与え方

do 文の終了条件の与え方には次の三種類があります:

do 文の終了条件の与え方

thru	制御変数の境界値に達した時点で反復処理を終える.
unless	条件を満した時点で反復処理を終える.
while	条件を満さなくなった時点で反復を終える.

ここで終了条件を与える式は Maxima の述語, すなわち, 'true' か 'false' の何れかが判別可能な Maxima の式です. "unless" を用いた do 文は C の repeat-until 文, while を用いた do 文は C の while 文に相当します.
"thru", "unless" と "while" を用いた例を次に示します. なお, 三種類ともに全て同じ反復処理:「1 から 10 までの数を表示して終了」を実行するものです:

終了条件の例

```
for i:1 thru 10 do print(i);
for i:1 while i <= 10 do print(i);
for i:1 unless i > 10 do print(i);
```

ここで do 文によって返される値は通常は 'done' です. return 函数を用いると本体の中で do 文から早目に抜けて必要な値を与えることに使えます. block にある do 文中の return 函数は do 文を出るだけで block 全体から出る訳ではありません. 同様に go 函数も block 中の do 文から抜けるためには使えません.

8.1.5　エラー処理

エラー処理に関しては, 多くの言語で見られる catch 函数と throw 函数の他に, breakpoint を置く break 函数や, エラー発生時に処理を行う函数があります. ここでは最初に catch 函数と throw 函数について述べます:

catch 函数と throw 函数

catch(⟨ 式$_1$⟩ ,...,⟨ 式$_n$⟩)
throw(⟨ 式 ⟩)

catch 函数:　throw 函数と対で用いる函数です. これは非局所的回帰 (non-local return) で用いる函数で, 最も近い throw 函数に対応する catch 函数に移動します. そのために throw 函数に対応する catch 函数が必ず必要で, そうでなければエラーになります. ⟨ 式$_i$⟩ の評価が何らの throw 函数の評価に至らなかった場合に catch 函数の値は最後の引数 ⟨ 式$_n$⟩ の値となります.

```
(%i51) g(l):=catch(map(lambda([x],
       if x<0 then throw(x) else f(x)),l));
(%o51) g(l) := catch(map(lambda([x], if x < 0 then
                    throw(x) else f(x)), l))
```

```
(%i52) g([1,2,3,7]);
(%o52)                          [f(1), f(2), f(3), f(7)]
(%i53)  g([1,2,-3,7]);
(%o53)                                  - 3
```

函数 g は, 引数 l が非負の数のみであれば l の各要素に対する f のリストを返します. それ以外で, 函数 g は l の最初の負の要素を捉えて, それを放します.

throw 函数: 与えられた〈式〉を評価し, 近くにある catch 函数に値を投げ返します. throw は catch と対で用いられる函数です.

直接的なエラー処理を行う函数

break(〈引数〉,...)
errcatch(〈式$_1$〉,...,〈式$_n$〉)
error(〈引数$_1$〉,...,〈引数$_n$〉)
errormsg()

break 函数: 〈引数〉の評価と表示を行い, (maxima-break) にて利用者がその環境を調べて変更することが出来るようにします. (maxma-break) から抜けて処理を再開させる場合には, 'exit;' と入力します. なお, fbox**Ctrl+a(^a)** で maxima-break に何時でも対話的に入ることができます. "Ctrl+x" は maxima-break 内部で本体側の処理を終了せずに局所的に計算を止めることに用いても構いません.

errcatch 函数: 引数を一つづつ評価し, エラーが生じなければ最後の値のリストを返す函数です. 引数の評価でエラーが生じた場合に errcatch 函数はエラーを捉えて即座に空リスト "[ ]" を返します. この函数はエラーが生じていると疑われる batch ファイルで, エラーを捉えなければたちまち batch を終了させるようにすると便利です.

error 函数: 引数の評価と表示を行い, Maxima のトップレベル, あるいは errcatch 函数にエラーを返します. エラー条件を検知した場合や "Ctrl+^" が入力出来ない場所ならどこでも函数を中断させられるので便利です.
大域変数 error にはエラーを記述するリストが設定されており, 最初のものは文字列, 残りの対象は問題を起している対象です.

errormsg 函数: 最新のエラーメッセージを再表示します. 変数 error にはエラーを記した対象のリストが設定されており, 最初は文字列, 残りは問題の対象です. たとえば, 'ttyintfun:lambda([], errormsg(), print(""))' で利用者中断文字 "(^ u)" によるメッセージの再表示を行うように設定できます.

8.1.6　プログラムに関連する大域変数

プログラムに関連する大域変数

変数名	既定値	概要
backtrace	[]	函数リスト
dispflag	true	block 文中の函数出力を制御
errorfun	false	エラー発生時に起動させたい函数名

大域変数 backtrace:　'all' のときに入力された函数全てのリストを値として持ちます.

大域変数 dispflag:　'false' のときに block 文の中で呼ばれた函数の出力表示を禁止します. 記号 "$" のある block 文の末尾では大域変数 dispflag を 'false' に設定します.

大域変数 errorfun:　引数を持たない函数名が設定されていればエラー発生時に大域変数 errorfun 設定された函数が実行されます. この設定は batch ファイルでエラーが生じた場合に Maxima を終了したり, 端末からログアウトしたいときにも使えます.

8.2 函数とマクロの定義

8.2.1 函数とマクロについて

Maxima の函数とマクロの定義は類似しています. 基本的に函数の定義は演算子 ":=" を用い, マクロの定義は演算子 "::=" で行います. ただし, 函数もマクロも内部的には Maxima の S 式の変換をおこなう函数です.
ここでは簡単のために演算子 ":=" や define 函数で定義される対象を広義の函数と呼びます. たとえば通常の函数 f(x) や配列函数 f[x] が広義の函数となります. この広義の函数は, その名詞形の表記によって二種類に分類されます. まず, f(x) のように名前とその直後に小括弧 "()" で括られた引数列を持つ形式的な項になる対象を狭義の函数, あるいは簡単に函数と呼ぶ対象です. 同様に 'a[i]' のように名前とその直後に大括弧 "[]" で括られた整数値変数の列を持つ形式的な項になる対象を特に配列函数と呼びます. そして, 狭義の函数は大域変数 functions に登録され, 配列函数は大域変数 arrays に登録されています.
ここでマクロも Maxima の対象の函数ですがやや特殊です. まず, Maxima のマクロは対象の内部表現である S 式に作用して S 式を返す函数です. したがって, マクロを用いて狭義の函数ど同様の作用を行う対象を生成することもできます. ただし, 演算子 "::=" で定義される対象は全てマクロであり, マクロの名前は大域変数 macros に登録される点が函数と異なります.
函数とマクロの生成は演算子や函数が異なりますが, これらの対象の定義内容の閲覧は dispfun 函数や fundef 函数が共通で使えます.
そして, これらの函数やマクロの削除も共に remfunction 函数で行え, 引数を切換えることで remove 函数や kill 函数でも削除が行えます.
この函数やマクロの削除は大域変数 functions や大域変数 macros に登録された名前の削除による名前と実体の切断によるものです.
これらの大域変数の操作に関連し, 函数定義で同名のマクロが存在した場合, 大域変数 macros からマクロ名が削除され, 大域変数 functions に函数名が新規に登録されます. 逆に, マクロ定義で同名の函数が存在する場合は大域変数 functions から函数名が削除され, 大域変数 macros にマクロ名が登録されます.
この排他的処理は大域変数 functions と大域変数 macros でのみ行われます. そのため, 大域変数 arrays に同名の配列函数が存在しても, この排他処理の対象にはなりません.

8.2.2 函数の定義

Maxima では予めシステムが持っている函数の他に利用者定義の函数が扱えます. ここでの函数には, 函数名を持つ通常の函数と, lambda 函数による無名函数の二種類があります.

函数定義で用いる演算子と函数

通常の函数定義では演算子 ":=" を用いる方法と define 函数を用いる二つの方法があります:

函数の定義方法

⟨ 函数名 ⟩(⟨ 引数$_1$⟩,...,⟨ 引数$_n$⟩):=⟨ 函数本体 ⟩
⟨ 函数名 ⟩[⟨ 引数$_1$⟩,...,⟨ 引数$_n$⟩]:=⟨ 函数本体 ⟩
define (⟨ 函数名 ⟩,⟨,((⟨ 引数$_1$⟩,...,⟨ 引数$_n$⟩),⟨ 式 ⟩)
define (⟨ 函数名 ⟩,⟨,[⟨ 引数$_1$⟩,...,⟨ 引数$_n$⟩],⟨ 式 ⟩)
define (funmake(⟨ 函数名 ⟩,⟨,[⟨ 引数$_1$⟩,...,⟨ 引数$_n$⟩]),⟨ 式 ⟩)
define (arraymake(⟨ 函数名 ⟩,⟨,[⟨ 引数$_1$⟩,...,⟨ 引数$_n$⟩]),⟨ 式 ⟩)
define (ev(⟨ 式$_1$⟩,⟨⟩,⟨ 式$_2$⟩)

演算子 ":=": 一般的な函数や配列の添字に対する定義は演算子 ":=" で定義できます. ここで演算子 ":=" の右辺の式は定義の際に評価が行われません. そのために, 演算子 ":=" の右辺の被演算子にはコンマで区切った式, block 文, if 文や do 文で構成することが可能です.
そのために何らかの値が割当てられた変数が右辺に含まれていても, 函数定義の際に評価が行われないために変数がそのまま用いられ, 函数項の計算で単純に置換えられます:

```
(%i22) neko(x):=mike*10;
(%o22)                          neko(x) := mike 10
(%i23) neko(10);
(%o23)                              10 sin(x)
```

函数の定義で式の評価が必要であれば, define 函数を用いて函数の定義を行うか, 演算子 """(二重の単引用符) によって函数本体を評価させなければなりません.

define 函数: 二つの引数を取り, 第 2 引数を必ず評価する函数です:

```
(%i25) mike:sin(x);
(%o25)                                sin(x)
(%i26) define(neko(x),mike*10);
(%o26)                          neko(x) := 10 sin(x)
(%i27) neko(100);
(%o27)                              10 sin(100)
```

この define 函数は funmake 函数で定めた函数項や ev 函数による評価を函数項にしたり, arraymake 函数を使って配列の添字に対する函数を定義することが可能です.
ここで define 函数の大きな特徴は, 第 2 引数に与える函数本体が必ず評価されることです. そのために函数本体としては, block 文, if 文や do 文を含まない一つの式で構成可能なものに限定されます.
なお, 第 1 引数が funmake 函数項, arraymake 函数項や ev 函数項の場合, 函数項としてこれらの函数項を評価して得られた項で置換えられます:

```
(%i31) define(funmake(mike,[x]),t^2+x-1);
                                          2
(%o31)                          f(t) := t  + t - 1
(%i32) define(arraymake(mike,[x]),2*(x-1)!);
                                    f  := 2 (t - 1)!
(%o32)                               t
```

```
(%i33) f[10];
(%o33)                                  725760
(%i34) define(tama(x,y),pochi(y,x))$

(%i35) define(ev(tama(a,b)),a-b);
(%o35)                          pochi(b, a) := a - b
```

可変 arity の函数定義

ここで函数の引数の個数 (arity) が可変の函数も演算子 ":=" や define 函数を用いて定義できます. この場合は引数としてあってもなくてもよい複数の変数を一つの変数として纏め, その 1 成分のリストとして表現して函数を定義します:

```
(%i41) f([u]):=u;
(%o41)                                f([u]) := u
(%i42) f(1,2,3,4,5);
(%o42)                              [1, 2, 3, 4, 5]
(%i43) f(a,b,[u]):=[a,b,u];
(%o43)                       f(a, b, [u]) := [a, b, u]
(%i44) f(1,2,3,4,5,6);
(%o44)                         [1, 2, [3, 4, 5, 6]]
(%i45) f(1,2);
(%o45)                              [1, 2, []]
```

最初の函数 f の定義では, その引数を f([u]) とリスト [u] としています. その結果, 可変 arity の函数が定義できています. 次の例では f(a, b, [u]) とすることで, arity が 2 以上の函数が定義されます. ここで最初の二つの引数 a と b は必ず与えられるべき引数ですが, [u] の部分は空でも構いません. ただし, 最初の例と違い f(1,2) の結果は最後の引数が空リスト "[]" として処理され, その結果, 空リストを含むリストが返却されます. このように空リストの処理も考慮して引数をどのように設定するかを明確に決定しておかなければなりません.

無名函数の定義

lambda 函数: 無名函数の構築で用いられる Church の λ 式に由来する函数です:

lambda 函数

lambda([⟨ 変数$_1$⟩,...,⟨ 変数$_n$⟩],⟨ 函数本体 ⟩)

この lambda 函数の構文は最初に函数の変数を宣言し, そのあとに函数本体が続きます. ここでも函数本体は式をコンマで区切った文になります. なお, lambda 函数で構成した函数は函数名を持たない函数で, 基本的に手続を表現する対象です. この無名函数は函数の定義で用いたり, apply 函数や map 函数と組合せた処理で用いられます.
次の例では 'lambda([i],2*i+1)' で引数 i に '1' を加える無名函数を構成し, その函数に map 函数でリスト '[1,2,3]' に作用させた結果と lambda 函数を利用した函数 neko を示しています:

```
(%i58)  map(lambda([i],2*i+1),[1,2,3]);
(%o58)                                [3, 5, 7]
(%i59)  neko(x):=map(lambda([i],sin(2*i+1)),x);
(%o59)           neko(x) := map(lambda([i], sin(2 i + 1)), x)
(%i60)  neko([1,2,3,4,5]);
(%o60)             [sin(3), sin(5), sin(7), sin(9), sin(11)]
(%i61) i;
(%o61)                                  i
```

ここで,lambda 函数内部で用いた疑似変数 i は lambda 函数内部のみで値を持つことに注意して下さい.

8.2.3 函数定義に関連する大域変数

大域変数 functions: define 函数や演算子 ":=" によって通常の函数と配列函数の二種類の (広義の) 函数が定義できます. ここで函数はその項が "f(x)" のように引数を小括弧 "[]" で括った対象で, この書式を項として持つ広義の函数を狭義の函数, あるいは簡単に函数と呼びます. これに対して項が "f[x]" のように配列となる対象を配列函数と呼び広義の函数に含めますが, 狭義の函数には含めません.
狭義の函数に対しては, その函数名が大域変数 functions に登録されますが, 大域変数 macros に同名のマクロが存在すれば, 自動的に大域変数 macros の対象が削除されます. なお, 配列函数に関しては, その函数名は大域変数 functions ではなく大域変数 arrays に登録されますが, 排他処理の対象にはなりません.

大域変数 functions

変数名	既定値	概要
functions	[]	利用者定義の (狭義の) 函数リスト

函数定義による大域変数 functions の様子を見てみましょう:

```
(%i1) functions;
(%o1)                                  []
(%i2) f(x):=sin(x);
(%o2)                             f(x) := sin(x)
(%i3) f(10);
(%o3)                                sin(10)
(%i4) functions;
(%o4)                                [f(x)]
```

この例では函数 f(x) を定義すると立ち上げ時に空リスト "[]" だった大域変数 functions に演算子 :=の左辺の函数名が登録されることが判りますね.
函数定義の演算子 ":=" を用いた函数で最も簡単なものは, 式$_1$, 式$_2$, ..., 式$_n$ のように複数の式を単純に並べた文です. この場合, 函数の返却値は最後の式の結果になります. たとえば, f(x):=(1-x,2+x,2*x) で函数 f を定義した場合, この函数の返却値は最後の式 $2x$ を計算した値になります.

ただし, この函数では割当が実行されていないために Maxima の変数の内容の書換えは一切生じません.
では, f(x):=(y:x,z:2+y,2*z) で定義した函数を実行すると, その影響にはどのようなものがあるでしょうか？

```
(%i1)  f(x):=(y:x,z:2+y,2*z);
(%o1)                    f(x) := (y : x, z : 2 + y, 2 z)
(%i2)  f(x);
(%o2)                                2 (x + 2)
(%i3) y;
(%o3)                                   x
(%i4) z;
(%o4)                                 x + 2
(%i5)  f(2);
(%o5)                                   8
(%i6) x;
(%o6)                                   x
(%i7) y;
(%o7)                                   2
(%i8) z;
(%o8)                                   4
```

まず, この函数の処理では函数本体の文の先頭式から順番に処理され, 最後の式の結果だけが返却されています. さらに, この函数の場合, 内部で用いた変数 y と z の値が書換えられていることに注意して下さい.
このような方法で用いた変数は大域変数として扱われます. そのために変数値の書換が生じます. このことを避けるために, Maxima では函数内部のみで有効な局所変数が扱えます. この局所変数は block 文の中で定義可能です.

8.2.4 函数定義に関連する函数

函数定義に関連する函数

funmake(〈函数名〉,[〈引数$_1$〉,...,〈引数$_n$〉])
local(〈変数$_1$〉,...,〈変数$_n$〉)

funmake 函数: 形式的な函数項を生成する函数で, define 函数と組合せて函数定義に利用できます. 第 1 引数の〈函数名〉と第 2 引数のリストで与えた対象を引数とする形式的な函数項を生成する函数ですが, その際に函数項自体の評価は行いません:

```
(%i2)  funmake(f,[x,y,z]);
(%o2)                                f(x, y, z)
(%i3)  funmake(neko,[x,y,z]);
(%o3)                               neko(x, y, z)
(%i4) funmake(expand,[128,"うちのタマ知りませんか？"]);
(%o4)                        expand(128, うちのタマ知りませんか？)
(%i5) funmake(a,[1,2,3]);
(%o5)                                a(1, 2, 3)
```

```
(%i6)  a:10;
(%o6)                                  10
(%i7)  funmake(a,[1,2,3]);
Bad first argument to 'funmake': 10
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i8)  funmake('a,[1,2,3]);
(%o8)                                a(1, 2, 3)
```

函数名と同名の束縛変数が存在した場合, その値が函数名で置換えられて評価されるためにエラーになります. この場合, 単引用符 "'" を使って函数を名詞型にしなければなりません.

local 函数: 局所変数を定義する函数です. この local 函数は与えられた変数を引数とする函数項を生成します. そして, この local 函数項の引数として現われる変数が局所変数となることを表現します. この local 函数は block 文, 函数定義本体, lambda 函数, または ev 函数内部で一度だけ利用可能です. また, この local 函数に与えられた変数は文脈の影響を受けません.

8.2.5 マクロの定義

Maxima のマクロは文を含む対象の内部表現の S 式を変換する函数です. マクロに対象を引渡すと引数を用いた S 式への変換が実行され, それから S 式の評価が行われます.
函数定義とは異なり, マクロ定義の演算子は ":==" のみであり, さらに定義されたマクロも大域変数 macros のみに登録されますが, 大域変数 functions に同名の対象が存在する場合は大域変数 functions の対象を削除する排他処理が行われます.

マクロの定義に関連する函数と演算子

マクロの定義では演算子 "::=" が用いられますが, マクロ定義のみで用いられる函数として buildq 函数があり, この buildq 函数のみで用いられる函数に splice 函数があります:

buildq 函数 と splice 函数の構文

マクロ名 ::= ⟨ マクロ本体 ⟩)
buildq(⟨ 変数リスト ⟩,⟨ 式 ⟩)
splice(⟨ 変数 ⟩)

演算子 "::=": 比較的単純なマクロ函数の定義では函数のように演算子 "::=" を用いて定義が行えます. ここで演算子 "::=" の左辺がマクロ定義項, 右辺がマクロ定義の本体となります. ここでマクロ定義項の書式は狭義の函数項と同様に函数名のうしろに小括弧 "()" で引数の列を括った書式になります.
ここでマクロ名として利用出来ない表徴は 'all', '%', '%%' です. また, マクロ項を verbify 函数で動詞化した項や配列項も扱えません.
マクロ本体の書式は, 一つの式, block 文等の通常の Maxima の式や文が使えますが, マクロの定義だけに利用可能な函数として buildq 函数があります.

マクロ本体は入力と同時に S 式の変換が行われますが, この変換をマクロの展開と呼びます. このマクロの展開では文脈を用いた式の評価が実行され, マクロ項の引数と式中の同名の変数との同一視が行われます. この同一視を避けたければ, 単引用符 “’” を用いて名詞化します:

```
(%i1) assume(y1<0)$

(%i2) x:1/y1$

(%i3) mike(x):=block([alpha],alpha:(x+1-2*abs(y1)),
                     (1/x^2+2)+alpha+abs(y1)+b);
(%o3) mike(x) := block([alpha], alpha : x + 1 + (- 2) abs(y1),
                                                  1
                                                  -- + 2 + alpha + abs(y1) + b)
                                                   2
                                                  x
(%i4) tama(x):=block([alpha],alpha:'(x+1-2*abs(y1)),
                            (1/x^2+2)+alpha+abs(y1)+b);
(%o4) tama(x) := block([alpha], alpha : '(x + 1 + (- 2) abs(y1)),
                                                  1
                                                  -- + 2 + alpha + abs(y1) + b)
                                                   2
                                                  x
(%i5) mike(a);
                                          1
(%o5)                      y1 + b + a + -- + 3
                                          2
                                         a
(%i6) tama(a);
                              1           1
(%o6)                    y1 + -- + b + -- + 3
                              y1          2
                                         a
```

ここでの例では最初に式 ‘y1<0’ を仮定し, 変数 x には式 ‘1/y1’ を割当てておきます. このとき, 最初のマクロ mike では単引用符 “’” による評価を抑制していないために, 式中の変数 x はマクロ項の変数 x と同一視されて変数 x に割当てられた値の ‘1/y1’ で置換えられることはありません. 次のマクロ tama では, 二番目の式中の変数 x を含む部分式の評価が単引用符 “’” で抑制されているため, マクロの展開でマクロ項の変数との同一視は実行されません. そのため展開後に変数 x の値の評価が実行されて式 ‘1/y1’ で置換えられます.
なお, 演算子 “::=” を用いて定義されたマクロは大域変数 macros に登録されます.

buildq 函数: 無名函数を生成する lambda 函数のマクロ版です. 第 1 引数に変数リスト, 第 2 引数に式の二つの引数を持つ函数で, 変数リストに包含される変数への値の割当を演算子 “:” を用いて行うこともできます.
この buildq 函数の引数は実際の式の代入が実行されるまで Maxima の解釈で勝手に変換されることを防止する必要があります. そのために単引用符 “’” を用います.

splice 函数: splice 函数は buildq 函数内部のみで利用可能な函数項を生成します. splice 函数は buildq 函数内部の変数を引数とし, buildq 函数はその内部で splice 函数項の引数として与えられ

た変数に割当てられたリストの括弧を外します. splice 函数は buildq 函数内部で函数項を生成するだけの函数のために, buildq 函数外部では未定義の函数, すなわち, 形式的な函数項として扱われます:

```
(%i3) buildq([x:'[b,c,d,e,f]],splice(x)+splice(x));
(%o3)                          2 f + 2 e + 2 d + 2 c + 2 b
(%i4) buildq([x:'[b,c,d,e,f]],pochi(splice(x))/apply(tama,x));
                                   pochi(b, c, d, e, f)
(%o4)                          ----------------------------
                               apply(tama, [b, c, d, e, f])
(%i7) splice([1,2,3,4]);
(%o7)                              splice([1, 2, 3, 4])
```

buildq 函数を用いた漸化式の計算

buildq 函数の応用として漸化式の計算があります. ここでは幾つかの数列を buildq 函数を使って定義してみましょう:

A. Fibonacci 数: 次の漸化式を満す数です:

Fibonacci 数

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_{n+1} &= F_n + F_{n-1}
\end{aligned}
$$

では, Maxima の buildq 函数によるマクロを利用して, この数列を定義してみましょう:

```
(%i10) fb:F[n-1]+F[n-2];
(%o10)                                F      + F
                                       n - 1    n - 2
(%i11) define(F[n],buildq([u:fb],u));
(%o11)                               F   := F      + F
                                      n     n - 1    n - 2
(%i12) F[0]:0$F[1]:1$F[2]:1$
(%i15) F[10];
(%o15)                                  55
(%i16) F[140];
(%o16)                 810559000960235041972064086O5
```

ここでの定義では配列函数の定義で buildq 函数を利用しています. したがって, define 函数を用いていても大域変数 functions には F は現われず, 大域変数 arrays に現われています.
なお, この例のように比較的漸化式が単純な場合, buildq 函数を使わずに
'define(F[n],F[n-1]+F[n-2])' や 'F[n]:=F[n-1]+F[n-2]' で定義しても勿論構いません.

B. Legendre の多項式: Legendre の多項式は次の漸化式で定義される多項式です:

Legendre の多項式

$$
\begin{aligned}
P_0(z) &= 1 \\
P_1(z) &= z \\
P_{n+1}(z) &= (2n-1)zP_n - (n-1)P_{n-1}
\end{aligned}
$$

```
(%i10)  pnz:((2*n-1)*z*Pz[n-1]-(n-1)*Pz[n-2])/n;
                        (2 n - 1) Pz       z - (n - 1) Pz
                                   n - 1                 n - 2
(%o10)                  ------------------------------------
                                          n
(%i11) define(Pz[n],buildq([v:pnz],expand(v)));
                              (2 n - 1) Pz       z - (n - 1) Pz
                                          n - 1                 n - 2
(%o11)        Pz   := expand(------------------------------------)
                n                                 n
(%i12) Pz[0]:1$Pz[1]:z$
(%i14) Pz[10];
                  10            8           6           4          2
           46189 z      109395 z     45045 z     15015 z     3465 z     63
(%o14)     --------- - --------- + -------- - -------- + ------- - ---
              256          256         128         128         256     256
```

この例では expand 函数を用いて式を展開しています. この式の展開を省くと結果が複雑になるので注意が必要になります. 次に expand 函数による処理を省略した結果を示しておきます:

```
(%i17) kill(Pz);
(%o17)                          done
(%i18) Pz[n]:=((2*n-1)*z*Pz[n-1]-(n-1)*Pz[n-2])/n;
                          (2 n - 1) z Pz       - (n - 1) Pz
                                        n - 1                 n - 2
(%o18)              Pz   := ------------------------------------
                      n                          n
(%i19) Pz[0]:1$Pz[1]:z$
(%i21) Pz[4];
                                  2
                          5 z (3 z  - 1)
                      7 z (-------------- - 2 z)          2
                                 2                  3 (3 z  - 1)
                      ------------------------- - ------------
                                 3                       2
(%o21)                ---------------------------------------
                                          4
```

この例で示すように式の簡易化が行われていないため, $n = 4$ でも必要以上に複雑な結果となっています.

8.2.6 マクロの展開に関連する函数

このマクロの展開に関連する函数に macroexpand 函数と macroexpand1 函数, 大域変数には macroexpansion があります.

macroexpand 函数と macroexpand1 函数

macroexpand(⟨ 式 ⟩)
macroexpand1(⟨ 式 ⟩)

macroexpand 函数:　与式にマクロが含まれている場合, 与式の評価を伴わずにマクロの展開のみを行います.

macroexpand1 函数:　マクロの展開を行いますが, 式の評価は伴いません.

8.2.7　マクロに関連する大域変数

マクロに関連する大域変数

変数名	既定値	概要
macros	[]	マクロ名が登録される大域変数
macroexpansion	false	マクロ展開の制御で利用

大域変数 macros:　演算子 "::=" で定義されたマクロの名前が登録されるリストです.
大域変数 macros と大域変数 functions には、名前が双方の大域変数に包含されることがないように排除を行う処理が函数定義とマクロの定義の双方で実行される仕組になっています.

大域変数 macroexpansion:　マクロ展開を制御する大域変数です. Maxima のマクロは最初に呼出された時点でマクロが展開されます. ここで, 展開されたマクロを保持していれば次に同じマクロの呼出しがあった場合に展開する手間が不要となり, その分, 効率的な処理が行えます. 一方で, 展開したマクロを保持するためにはメモリが必要となります. そのため, この大域変数 macroexpansion には三種類の指定が行えるようになっています:

- false
  マクロが呼出されるたびにマクロの展開を行います. この場合はマクロ項に展開した文が割当てられることはありません.
- expand
  マクロの呼出によって展開されると, 以後, マクロの展開をしなくても良いように展開した S 式を保持するので記憶容量を必要とします.
- displace
  マクロの呼出によって展開されると, 呼出しを展開式で完全に置換えて保持します. expand を指定した場合よりも記憶領域を効率良く利用します.

8.2.8 利用者定義函数とマクロの確認

利用者が Maxima 言語で定義した函数の内容は dispfun 函数や fundef 函数で参照できます:

定義した函数の内容を表示する函数
dispfun(⟨ 函数名$_1$⟩,...,⟨ 函数名$_n$⟩)
dispfun(all)
fundef(⟨ 函数名 ⟩)

dispfun 函数: 利用者定義の函数 ⟨ 函数名$_1$⟩,...,⟨ 函数名$_n$⟩ の内容を表示します. この函数の表示では函数定義で用いた変数や定数がそのまま表示されます.
ここで, dispfun 函数の引数に all を設定すると, 大域変数 functions と大域変数 arrays に登録された函数を全て表示します.

fundef 函数: fundef 函数は ⟨ 函数名 ⟩ に対応する函数の定義を返します. fundef 函数は dispfun 函数に似ていますが, fundef 函数は display 函数を呼出さない点で異なります:

```
(%i9) neko(x):=sin(x)*exp(x);
(%o9)                    neko(x) := sin(x) exp(x)
(%i10) dispfun(neko);
(%t10)                   neko(x) := sin(x) exp(x)

(%o10)                                done
(%i11) fundef(neko);
(%o11)                   neko(x) := sin(x) exp(x)
```

この例で示すように dispfun 函数の結果は%t ラベルに表示されます. しかし, fundef 函数の結果は通常の%o ラベルに表示されます. ただし, この点以外の違いはありません.

8.2.9 利用者定義函数とマクロの削除

利用者定義函数とマクロの削除の実体は, これらの対象が登録されたリストから名前の削除による名前と実体の割当の解消に対応します. このような削除を行う函数として, 函数とマクロ専用の remfunction 函数と一般的な対象の削除を行う remove 函数と kill 函数が挙げられます:

利用者定義函数とマクロを削除する函数

remfunction (⟨ 函数名$_1$⟩, ⟨ 函数名$_2$⟩, ...)
remfunction (all)
remove (⟨ 函数名$_1$⟩, ⟨ 属性$_1$⟩, ...)
remove ([⟨ 函数名$_1$⟩, ..., ⟨ 函数名$_n$⟩], [⟨ 属性$_1$⟩, ..., ⟨ 属性$_n$⟩])
remove (all,⟨ 属性 ⟩)
kill(⟨ 函数名$_1$⟩, ⟨ 函数名$_2$⟩, ...)
kill(functions)
kill(macros)
kill(all)

remfunction 函数:　引数として与えられた函数やマクロを削除します. その際に大域変数 functions に含まれている函数名と大域変数 macros に含まれているマクロ名を削除します.
ここで引数として all が与えられた場合, 大域変数 functions と大域変数 macros に含まれる全ての利用者定義の函数とマクロが削除されます.

remove 函数:　函数を削除する場合は属性として function, マクロを削除する場合は macro を指定します.

kill 函数:　Maxima の対象を削除する函数です. この函数の詳細は §6.13.2 を参照して下さい. 函数やマクロを削除する場合は直接, 函数名やマクロ名を指定します. さらに, 大域変数 functions や大域変数 macros を引数とすることで, これらの大域変数に包含された対象が削除できます. また, all を指定すれば, Maxima の対象を削除することができます.

8.3　自動的に読込まれる函数

share ライブラリの多くは利用者が予め load 函数を用いて Maxima に読込まなければならないものです. ただし, 一部の函数は必要に応じて Maxima が自動的に読込みます. このような函数は src/max_ext.lisp 内部で設定されています. ここの函数の一覧を挙げておきましょう:

自動的に読込まれる函数

パッケージ名	函数
nusum パッケージ:	nusum, unsum, funcsolve, nusum
bffac パッケージ:	bffac, bfzeta, bfpsi, bfpsi0
trigrat パッケージ:	trigrat
gcdex パッケージ:	gcdex
stopex パッケージ:	expandwrt, expandwrt_factored
facexp パッケージ:	facsum, factorfacsum, collectterms
disol パッケージ:	disolate
declin パッケージ:	linsimp, declare_linear_operator
genut パッケージ:	nonumfactor
eigen パッケージ:	eigenvectors, eigenvalues
trgsmp パッケージ:	trigsimp
ode2 パッケージ:	ode2, ic1, ic2, bc2, desimp, linear2

なお, ここで挙げた函数はシステムが必要に応じて自動的に読込む函数です. 利用者が自分向けに必要に応じて函数の読込を行いたければ, Maxima の初期化ファイル maxima-init.mac と setup_autoload 函数を利用します. このファイルの詳細については §10.10.8, setup_autoload 函数については §10.10.5 を参照して下さい.

8.4　式と函数の最適化

8.4.1　最適化について

Maxima は表に表示されている式とその内部表現は大きく異なっています. Maxima の入力式は S 式で置換えてられ, その S 式に対して処理が実行されます. これは入力式に限定されず, Maxima 言語の文についても同様です. そのために入力式の S 式への変換, 翻訳と LISP による解釈といった手間が入り込み, これが処理の低下に繋がります.
この対処方法としては次の処置ができます:

- 式を Maxima で計算し易い表現で置換える
- Maxima 言語で記述された函数を LISP の函数で直接置換える

これらの処理について, ここでは解説しましょう.

8.4.2　式の最適化

式の最適化を行う optimize 函数の構文

optimize(⟨ 式 ⟩)

optimize 函数:　式の展開を行う函数ではありませんが, 式に含まれる共通部分式を新しい変数で置換え, Maxima 上で効率的に計算出来る Maxima の式に変換します.
この際に block 文が用いられますが, 共通部分式がない場合, ⟨ 式 ⟩ をそのまま返却します:

```
(%i40) optimize((x+1)^3+1/(x+1)^2+exp((x+1)^2));
                                                    2     %2      3    1
(%o40)       block([%1, %2], %1 : x + 1, %2 : %1 , %e    + %1  + --)
                                                                   %2
(%i41) ans1:solve( x^4+x^3+3*x-1=0,x);
               sqrt(5)    sqrt(25 - sqrt(5))    1
(%o41) [x = - ------- - ------------------ - -,
                 4                    1/4      4
                           2 sqrt(2) 5
       sqrt(5)    sqrt(25 - sqrt(5))    1
x = - ------- + ------------------ - -,
         4                    1/4      4
                  2 sqrt(2) 5
       sqrt(sqrt(5) + 25) %i    sqrt(5)    1
x = - --------------------- + ------- - -,
                    1/4           4       4
         2 sqrt(2) 5
    sqrt(sqrt(5) + 25) %i    sqrt(5)    1
x = --------------------- + ------- - -]
                  1/4           4       4
       2 sqrt(2) 5
(%i42) optimize(ans1);
```

```
                                                                  1              1
(%o42) block([%1, %2, %3, %4, %5, %6, %7], %1 : -------, %2 : ----,
                                                  sqrt(2)          1/4
                                                                  5
                                                  %3        %3
%3 : sqrt(5), %4 : sqrt(25 - %3), %5 : - --, %6 : --, %7 : sqrt(%3 + 25),
                                                  4         4
          %1 %2 %4    1              %1 %2 %4    1          %1 %2 %7 %i          1
[x = %5 - -------- - -, x = %5 + -------- - -, x = - ----------- + %6 - -,
             2        4                2        4                2              4
    %1 %2 %7 %i          1
x = ----------- + %6 - -])
          2              4
```

optimize 函数に影響する大域変数

変数名	既定値	概要
otimprefix	%	optimize 函数で生成される記号で利用

大域変数 otimprefix: 設定される文字は optimize 函数で生成された記号に使用される前置詞です.

8.4.3 LISP の函数に変換する函数

Maxima の全ての対象は, その内部表現が S 式になっており, Maxima の裏で動作する LISP がこの内部表現を解釈して処理を実行しています. そのために函数やデータを LISP の函数やデータに変換してしまえば, 内部表現の解釈の手間が省けるので処理速度向上が見込めます:

LISP の函数で変換える函数

LISP 函数に変換する函数

translate(⟨ 函数$_1$⟩, $\cdots$, ⟨ 函数$_n$⟩)
translate(functions)
translate(all)
translate_file(⟨ ファイル ⟩)
translate_file(⟨ ファイル ⟩, ⟨LISP ファイル ⟩)
tr_warnings_get()
declare_translated(⟨ 函数$_1$⟩, $\cdots$, ⟨ 函数$_n$⟩)

translate 函数: Maxima 言語で記述した利用者定義の函数を LISP の函数に変換します. つまり, Maxima 言語で記述された函数は Maxima の裏で動作している LISP で解釈されて実行されますが, この函数を LISP の函数に変換しておけば解釈する手間が省ける分, 処理の高速化が望めるのです.

引数は 〈 函数$_1$〉,$\cdots$,〈 函数$_n$〉 のように利用者定義函数を直接指定する方法に加え, 引数に all や functions を指定して利用者定義函数を一度に変換することもできます. そして, 変換を行った函数は function 属性が破棄されて新たに transfun 属性が付与されます:

```
(%i16) add1(x):=block(mode_declare(x,fixnum),x+1);
(%o16)           add1(x) := block(mode_declare(x, fixnum), x + 1)
(%i17) add1(2);
(%o17)                                3
(%i18) properties(add1);
(%o18)                            [function]
(%i19) translate(add1);
(%o19)                              [add1]
(%i20) properties(add1);
(%o20)         [transfun, transfun, transfun, transfun, transfun]
```

この例で示すように LISP の函数への変換を円滑に行うため, 函数に含まれる局所変数に対し, mode_declare 函数を利用して型を指定しておく必要があります. この mode_declare 函数は §8.4.4 で詳細を述べます.
次に, 函数を translate 函数で変換すると大域変数 savedef が false のときに変換された函数の名前は大域変数 functions に割当てられた函数名リストから削除され, 今度は大域変数 props に割当てられたリストに函数名が追加されます.
当然のことですが, 函数は虫取りが完遂されるまで変換すべきではありません. そして, translate 函数は変換する函数内部の式が予め簡易化されていると仮定しています. そうでなければ最適化されていない LISP 函数が生成されてしまうので, 変換する意味が半減するかもしれません.
そのために大域変数 simp を false に設定して変換式の簡易化を禁じるべきではありません.
なお, translate 函数を用いて LISP の函数に変換した函数は, Maxima や LISP の版が異なってしまうと, これらの整合性の問題から以前と同じ動作をする保証がありません.

translate_file 函数: Maxima 言語で記述したプログラムを含むファイルを LISP 函数のファイルに変換する函数です. translate_file 函数は Maxima のファイル名, LISP のファイル名と translate_file が評価した引数の情報を含むファイル名を成分とするリストを返します.
最初の引数は Maxima ファイルの名前で, オプションの第 2 の引数は生成すべき LISP ファイル名です. 第 2 引数は第 1 引数に trisp の初期設定値の tr_output_file の値を第 2 ファイル名の初期設定値として与えます. たとえば, translate_file(”test.mc”)) でファイル test.mc を LISP ファイルの test.lisp に変換します.
さらに生成されるものには translate 函数が出力したさまざまな重要性の度合を持った警告メッセージのファイルがあります. 第 2 ファイル名は常に UNLISP です. このファイルは変数を含み, それには変換されたコードでのバグ追跡のための情報が含まれています.

tr_warnings_get 函数: 引数を取らない函数で, translate 函数による変換で, この translate 函数が出力する警告のリスト (内部変数*tr-runtime-warnined*に束縛されたもの) を表示します.
この函数変換に関連する大域変数は他の函数と比較しても多く, その上, 名前が長いものが多いために名前と綴を憶えるのは大変ですが, apropos(tr_) を実行すれば,tr_で開始する Maxima の大域変数等のリストが出力されるので, このリストを出して名前を確認すると良いでしょう.

declare_translated 函数: 引数の函数が既に変換されていることを宣言する函数です. Maxima のプログラムファイルを LISP に変換する際に, そのファイル中のどの函数が translate 函数で変換された函数, あるいは compile 函数で変換された函数として呼出されるべきか, そして, どれが Maxima の函数で, どれが未定義の函数であるかを知ることは translate 函数にとって重要なことです.
ファイルの先頭にこの宣言を置くと, ある記号がたとえ LISP 函数の値を持っていなかったとしても, 呼出された時にそれを持つ事を教えます. (mfunction-call fn arg_1 arg_2,$\cdots$) が生成されるのは, ⟨ 函数$_n$⟩ が LISP 函数に変換されるべきものであるかを translate 函数が知らない時です.

コンパイルを行う函数

LISP はコンパイラを持っており, Maxima 言語で記述した函数をコンパイルすることで, translate 函数を使って単に LISP の函数に変換した函数よりも一層の高速化が望めます:

コンパイルを行う函数

compile (⟨ 函数$_1$⟩,$\cdots$,⟨ 函数$_n$⟩)
compile (functions)
compile (all)
compile_file(⟨ ファイル ⟩,⟨ コンパイルされたファイル ⟩,⟨*LISP* のファイル名 ⟩)
compile_file(⟨ ファイル ⟩,⟨ コンパイルされたファイル ⟩)
compile_file(⟨ ファイル ⟩)
compfile(⟨ ファイル ⟩,⟨ 函数$_1$⟩,$\cdots$,⟨ 函数$_n$⟩)

compile 函数: 指定した Maxima の処理言語で記述した函数を LISP の函数に変換し, それを LISP の函数 compile を用いてコンパイルします. なお, compile 函数は函数名リストを返します. ここで引数に functions や all を指定すると大域変数 functions に登録されている利用者定義函数を全てコンパイルします.

compile_file 函数: 指定したファイルのコンパイルを行います. まず, 指定された ⟨ ファイル ⟩ には Maxima のプログラムが含まれており, compile_file 函数は translate 函数で LISP の函数に変換し, それらを compile 函数を使ってコンパイルします. 変換とコンパイルに成功すれば Maxima に結果を読込みます.

compile_file 函数: この函数は引数として四個のファイル名のリストを返します. このリストに含まれるファイル名は, (1) 元の Maxima プログラムファイル, (2)LISP への変換ファイル, (3) 変換に関する註釈ファイル, (4)compile 函数でコンパイルされたプログラムのファイルです. ここでコンパイルに失敗すると返却されるリストの第 4 成分は false になります.

compfile 函数: ⟨ 函数$_1$⟩,$\cdots$,⟨ 函数$_n$⟩ を LISP の函数に変換し,⟨ ファイル ⟩ に書込みます:

```
(%i28) neko(x):=sin(x);
(%o28)                              neko(x) := sin(x)
(%i29) compfile("mike",neko);

Translating neko
(%o29)                              /home/yokota/mike
(PROGN (DEFPROP NEKO T TRANSLATED) (ADD2LNC 'NEKO PROPS)
 (DEFMTRFUN (NEKO ANY MDEFINE NIL NIL) (X) (DECLARE (SPECIAL X))
  (SIMPLIFY (LIST '(%SIN) X))))
```

函数の変換やコンパイルに関連する大域変数

translate 函数と compile 函数によるシステムに関連する大域変数を纏めておきましょう:

translate 函数と compile 函数のシステムに関連する大域変数

変数名	既定値	概要
compgrind	false	compile 函数による出力制御
savedef	true	translate 函数による変換後に元の函数を残す
translate	false	translate 函数による自動変換を制御
transrun	true	変換前の函数の実行
undeclaredwarn	compfile	未定義変数に対する警告を制御

大域変数 compgrind: true であれば,compile による函数定義の出力が整形表示されます.

大域変数 savedef: true であれば, 利用者函数を translate 函数で変換しても元の Maxima のプログラムを残します. そのために大域変数 functions に割当てられた利用者函数名リストから変換した函数名を削除せずに残します. このことは, 函数定義で用いた函数項と函数の実体の関係が保持されることを意味します. つまり, 函数定義に関連する函数によって, 変換された函数の処理ができることを意味します. たとえば, 函数定義を表示する dispfun 函数で, 変換後でも函数定義の実体が表示可能であり, さらには函数の編集もできます.
false の場合, 大域変数 functions に割当てた利用者函数名リストから該当函数名が削除されるので, そのような操作は行えなくなります.

大域変数 translate: true であれば, 利用者定義函数が自動的に LISP 函数に変換されます. なお, Maxima と LISP の整合性の問題から変換される前と同じ動作をするとは限らないことに注意して下さい.

大域変数 transrun: false であれば,translate 函数で変換されたものではなく, 元の Maxima の函数 (それらが存在していれば) が実行されます

大域変数 undeclaredwarn: 次の四種類の設定項目があります:

undeclearewarn の設定項目

設定	動作
false	警告を表示しません
compfile	compfile であれば警告します
translate	translate や translate:true であれば警告します
all	compfile や translate であれば警告します

mode_declare(〈変数〉,any) を実行して〈変数〉が一般の Maxima の変数である事を宣言します. 即ち,float, 又は fixnum である事に限定されません.compile 函数でコンパイルされる利用者定義函数中の変数を宣言する特別な動作は全て無効にしなければなりません.
次に,translate 函数に影響を与えるた大域変数は非常に多くあります. 最初に, 大域変数 tr_state_vars に割当てられたリストに含まれる大域変数を以下に纏めておきます.

tr_state_vars に纏められた変数

変数名	既定値	概要
transcompile	false	compile 函数に必要な宣言を自動生成
translate_fast_arrays	true	配列変換を制御
tr_array_as_ref	true	変換函数の配列評価を指定
tr_function_call_default	general	函数変換を制御
tr_numer	false	数値変数の型を制御
tr_semicompile	false	機械語への翻訳を制御
tr_warn_bad_function_calls	true	不適切な宣言による函数呼出の警告制御
tr_warn_fexpr	compfile	fexpr 型に対する警告制御
tr_warn_meval	compfile	meval 型函数に対する警告制御
tr_warn_mode	all	変数型に対する警告を制御
tr_warn_undeclared	compile	未宣言変数に対する警告を制御
tr_warn_undefined_variable	all	未定義変数に対する警告を制御

大域変数 transcompile: true であれば translate 函数は可能な compile 函数に必要な宣言を生成します. compile 函数は transcompile:true を用います.

大域変数 translate_fast_arrays: true の場合に配列の変換を行います.

大域変数 tr_array_as_ref: 大域変数 translate_fast_arrays が false の場合のみ, translate_file 函数で変換されたプログラムの配列参照に影響を与えます. 大域変数 tr_array_as_ref が true であれば変換された函数は配列を評価しますが, false であれば変換されたプログラム中の単なる記号として配列が現れます.

大域変数 tr_function_call_default:　値として apply,expr,general と false を取り, 初期値は general となります:

- false の場合: Maxima 内部函数 meval を呼出して式を評価する事を意味します.
- expr の場合: 引数固定の LISP 函数と仮定します.
- apply の場合: apply 函数を用いて函数を引数に作用させて変換します.
- general の場合: 内部表現が mexprs と mlexprs に対しては良いコードを与えますが, macros に対しては駄目です. general の場合, コンパイルされた函数中で変数の割当が正確であることを保証します. たとえば, 函数 f(x) を変換する際に, ここで f が値を割当てられた変数であれば警告を出して, apply(f,[x]) のこととして函数を変換します.

なお, 初期設定値で何等の警告メッセージがなければ translate 函数で変換し, compile 函数でコンパイルした利用者定義函数には元の maxima 函数と完全な互換性があることを意味します.

大域変数 tr_numer:　'true' であれば数の属性をそのまま LISP の変数にも継承させます.

大域変数 tr_optimize_max_loop:　考えられる形式で translate 函数でのマクロ展開と最適化工程ループの最大回数を定めます. これマクロ展開エラーを捉えるためで非中断の最適化属性です.

大域変数 tr_semicompile:　'true' であれば translate_file 函数と compile 函数の出力形式は拡張されたマクロになりますが LISP コンパイラで機械語に翻訳されたものにはなりません.

大域変数 tr_warn_fexpr:　内部形式が fexpr 型のものが与えられると警告します. fexpr 型は通常変換されたプログラム内の出力であってはならず, 全ての文法的に正しい特殊なプログラム書式に変換されます.

大域変数 tr_warn_meval:　函数 meval が呼び出されると警告します. meval が呼出されると, 変換の問題点を指定します.

大域変数 tr_warn_mode:　変数が指定した型に対して適切でない値が指定されていれば警告します.

大域変数 tr_warn_undeclared:　未宣言の変数に関する警告を端末に送るべきときを決めます.

大域変数 tr_warn_undefined_variable:　未宣言の大域変数が存在する場合に警告します.
以下に大域変数 tr_state_vars にも含まれない translate 函数に関連する大域変数を纏めておきます.

LISP 函数への変換に関連する大域変数

変数名	既定値	概要
tr_bound_function_applyp	true	変換函数の割当てに対し警告
tr_file_tty_messagesp	false	メッセージ出力制御
tr_float_can_branch_complex	true	逆三角函数の複素数値の制御
tr_optimize_max_loop	100	マクロループの回数
tr_warn_bad_function_calls	true	変換時の警告を制御

大域変数 tr_bound_function_applyp: 'true' であれば函数の引数として割当てられた変数が函数として用いられている場合に警告します. たとえば, 'g(f,x):=f(x+1)' の様な場合です.

大域変数 tr_file_tty_messagesp: 大域変数 translate_file がファイルの変換を行う間に生成されたメッセージを端末に送るかどうかを決めます. false であればファイルの translate 函数による変換に関するメッセージは UNLISP ファイルのみに挿入されます. true であれば UNLISP メッセージは端末に表示され, ファイルにも挿入されます.

tr_float_can_branch_complex: 逆三角函数が複素数値を返しても良いかどうかを宣言します. 逆三角函数は sqrt, log, acos 等です. true の場合に引数 x が float(倍精度浮動小数点型) であったとしても acos(x) は any 型になります. false の場合は x が float 型で, そのときに限って acos(x) は float 型となります.

tr_warn_bad_function_calls: 変換時に不適切な宣言が行われたために函数の呼出しが生じた場合に警告します.

8.4.4 変数型指定に関連する函数

Maxima の記号は値を割当てれば大域変数となりますが, 一部の alphabetic 属性を持つ記号を除けば属性を一切持ちません. また, alphabetic 属性を持つ記号も, その他の属性を初期状態では持っていません (属性全般は §5.4, alphabetic 属性については §5.1.2 を参照).
これは函数の LISP への変換や最適化では不利になります. そのために記号に型を指定します. この型を指定する函数として, mode_declare 函数と modedeclare 函数がありますが, mode_declare 函数は modedeclare 函数と実際は同じ函数です. この変数による変数の型の指定に加え, 初期値を設定する函数として define_variable 函数があります:

変数に型指定を行う函数

modedeclare(⟨ 変数$_1$⟩,⟨ 型$_1$⟩,$\cdots$,⟨ 変数$_n$⟩,⟨ 型$_n$⟩)
mode_declare(⟨ 変数$_1$⟩,⟨ 型$_1$⟩,$\cdots$,⟨ 変数$_n$⟩,⟨ 型$_n$⟩)
mode_identity(⟨ 引数$_1$⟩,⟨ 引数$_2$⟩)
define_variable (⟨ 変数名 ⟩,⟨ 初期値 ⟩,⟨ 型 ⟩)

modedeclare 函数: 〈変数$_i$〉に〈函数$_i$〉や〈型$_i$〉を設定します. mode_declare 函数と modedeclare 函数は内部的では同一の函数です. この函数は translate 函数で変換する利用者定義の函数で用いる変数の型の指定で利用されます. ここで modedeclare 函数で指定した型は, 変数の mode 属性の属性値として次のものが割当てられます:

大域変数の mode 属性

型	modedeclare 函数で利用可能な型	概要
float	float,real,floatp,flonum,floatnum	浮動小数点
fixnum	fixp,fixnum,integer	整数
rational	rational,rat	有理数
number	number,bignum,big	数, 多倍長整数, 多倍長浮動小数点数
complex	complex	複素数
boolean	boolean,boole	論理値 (Boolean)
list	list,listp	リスト
any	any,none,any_check	任意の型

以下に簡単な例を示します:

```
(%i28) modedeclare(x1,integer);
(%o28)                                [x1]
(%i29) :lisp (get '$x1 'mode)
$FIXNUM
(%i29) mode_declare(x2,rat);
(%o29)                                [x2]
(%i30) :lisp (get '$x2 'mode)
$RATIONAL
(%i30) modedeclare(x2,rational);
(%o30)                                [x2]
(%i31) :lisp (get '$x2 'mode)
$RATIONAL
```

この例では変数 x1 を整数型, x2 と x3 を有理数型として型の指定を行っています. これらの型は各変数の mode 属性値として付与されます. そして, この属性値は LISP の get 函数を用いて取出せます.

なお, C とは違って Maxima では変数に指定した型以外の型の値を割当てることができてしまいます. そのために, 指定した型の変数が割当てられているかどうかを検証する函数 mode_identity 函数があります.

mode_identity 函数: 二つの引数を取り, 第 1 引数に型, 第 2 引数に変数名を指定し, 変数が mode_declare 函数で指定された型に適合する値が割当てられているかを検証する函数です:

```
(%i8) x1:128$
(%i9) mode_identity(integer,x1);
(%o9)                                  128
(%i10) x1:256.988$
(%i11) mode_identity(integer,x1);
Warning: x1 was declared mode fixnum, has value: 256.988
```

```
(%o11)                                  256.988
(%i12) mode_identity(float,x1);
(%o12)                                  256.988
(%i13) :lisp (get '$x1 'mode);
$FIXNUM
```

この mode_identity 函数は変数の mode 属性と指定された属性の対象が割当てられているかどうかを検証し, 適合する場合には変数の値を返し, 適合しない場合, この例では単に警告するだけです. mode_identity 函数の動作を制御する大域変数として mode_check_errop と mode_check_warnp があります.

define_variable 函数: 変数の宣言に加え, 初期値と型の設定が行える函数です. この函数は属性を上手く使うことで変数に値を割当てる際に, その変数の利用者が設定した条件に適合するかどうかを検証することもできます.
この define_variable 函数の処理手順を次に示しておきます.

- 引数が 2 個以上あることを確認します.
- 与えた変数が LISP の記号であることを確認します.
- mode_declare 函数を用いて変数型を指定します.
- declare 函数を用いて変数に special 属性を付与します.
- 型が any でなければ属性 assign に属性値 assign-mode-check を設定します. この属性値が設定されていない変数に対して Maxima は変数への値の割当の際に値の検証を行いません. 値の検証を行うためには大域変数の value_check 属性 に真理函数を割当てておく必要があります. この設定では qput 函数が有効です.
- 変数に値が割当てられていなければ指定した初期値を設定します. もしも値が既に割当てられていれば, define_variable 函数は与えられた初期値を割当てず, そのままの値にしておきます.

割当ての際に変数値の検証を行う方法は, qput 函数で付与した大域変数の value_check 属性に変数値を検証する真理函数名を割当ておきます. 次に動作例を示しておきましょう:

```
(%i1) ptest(y):=if not primep(y) then error(y,"is not prime!!")$
(%i2) define_variable(tama,5,integer)$
(%i3) qput(tama,ptest,value_check)$
(%i4) tama;
(%o4)                                   5
(%i5) tama:15;
15 is not prime!!
#0: ptest(y=15)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i6) :lisp (get '$tama 'assign)
ASSIGN-MODE-CHECK
(%i6) define_variable(mike,5,any)$
(%i7) properties(mike);
```

```
(%o7)                          [value, special]
(%i8) qput(mike,ptest,value_check)$
(%i9) properties(mike);
(%o9)          [value, [user properties, value_check], special]
(%i10) mike:15;
(%o10)                                 15
(%i10) :lisp (get '$mike 'assign)
NIL
(%i10) :lisp (put '$mike 'assign-mode-check 'assign)
ASSIGN-MODE-CHECK
(%i10) mike:15;
15 is not prime!!
#0: ptest(y=15)
 -- an error.  Quitting.  To debug this try debugmode(true);
```

この例では最初に素数でなければエラーを返す真理函数 ptest を定義し, それから, define_variable 函数で大域変数 tama に初期値 5 を与えて整数型を指定します. それから, qput 函数を用いて check_value 属性に ptest を与えます. このときに変数 mike に 15 を設定しようとすれば, 15 は素数ではないのでエラーになります.

ここまでの動作をもう少し詳しく解説しましょう. まず, define_variable 函数による宣言で変数型を integer に指定しています. define_variable 函数は変数の型を any 以外に指定していれば, 宣言する大域変数の assign 属性に内部函数の assign-mode-check 函数を割当てます. この内部函数は変数の value_check 属性に設定された函数を用いて変数の評価を実行する函数です. 次に, この value_check 属性の設定で qput 函数を用いています. ここでの例では大域変数 tama の value_check 属性に ptest 函数を割当てていますね. すると, 大域変数 tama に値を割当てるときに assign-mode-check 函数が大域変数 tama の check_value 属性に割当てられた真理函数で割当てようとする値を評価します. この例では ptest 函数に 15 が引渡されますが 15 が素数でないために false となり, 変数に 15 が割当てられません.

次の例では, 大域変数 mike の型を any とした場合です. この場合, 変数 mike の value_check 属性に値を設定しても, 先程のように mike:15 を実行した場合にエラーも何も出ません. この場合, 大域変数 mike の assign 属性に assign-mode-check が設定されていないためです. これは LISP で (get '$mike 'assign) を実行すれば,NIL が返されるので判ります.

そこで, :lisp (put '$mike 'assign-mode-check 'assign) としてみましょう. これによって属性 assign の値として内部函数 assign-mode-check が設定されます. すると, 割当の際に検証が実行されるようになります.

8.4.5 型の検証に関連する大域変数

型の検証に関連する大域変数

大域変数	既定値	概要
mode_checkp	true	定数変数の型を検証するかどうか制御
mode_check_errorp	false	型エラー処理の制御
mode_check_warnp	true	型エラーの表示の制御

大域変数 mode_checkp: true の場合, mode_declare 函数で宣言する変数に割当てられた値の型と新たに宣言する型が矛盾しないか検証し, 矛盾する場合はエラーを返します:

```
(%i17) x0:1.0$
(%i18) mode_declare(x0,integer);
Warning: x0 was declared mode fixnum, has value: 1.0
(%o18)                                [x0]
(%i19) mode_checkp:false$
(%i20) mode_declare(x0,integer);
(%o20)                                [x0]
```

大域変数 mode_check_errorp: true であれば, 既に値が割当てられた変数に対して mode_declare 函数で型の指定を行う際に, 指定する型と割当てられた変数の値が矛盾する場合にエラーを出します.

大域変数 mode_check_warnp: true の場合, mode_identity 函数は変数に割当てられた値の型と指定した型が異なる場合に警告を出します.

第9章　Maximaで扱う数学的対象

この章では,Maxima の様々な数学的対象について解説します.
但し, Maxima が持つ全ての数学的対象を解説するものではない為, 必要に応じて, 配布の Maxima に附属のマニュアルを参照して下さい.

9.1　数論に関連する函数

9.1.1　階乗

階乗に関連する演算子

演算子	例	概要
!	$n!$	n が自然数ならば n の階乗を計算. 一般の場合は $\Gamma(x+1)$
!!	$n!!$	n が自然数で奇数 (偶数) ならば n 以下の奇数 (偶数) の積

演算子 “!” と演算子 “!!” は Maxima の演算子としての属性を持っています. これらの演算子としての詳細は §5.3 の後置式演算子の項目を参照して下さい.

演算子 “!”:　$n \in \mathbb{Z}$ であれば n の階乗 $n!$ を計算します. $n \notin \mathbb{Z}$ で $n \in \mathbb{C} \setminus \mathbb{R}_-$ であれば, $\Gamma(n+1)$ の計算を行います. なお, $\mathbb{R}_-$ は負の実数集合 $\{x | x < 0, x \in \mathbb{R}\}$ です.

演算子 “!!”:　階乗演算子 “!” に似ていますが, $n \in \mathbb{Z}$ の場合, 次の式で定義される演算子です:

$$n!! \stackrel{def}{=} \begin{cases} 1 & , n = 0, 1 \\ \displaystyle\prod_{i=0}^{[n/2]-1} (n-2i) & , n > 1 \end{cases}$$

ここで “[ ]” は Gauß の記号で, 実数 $x \in \mathbb{R}$ に対して $[x]$ で x を越えない最大の整数 n を返却する函数です. Maxima では entier 函数が相当します.
この定義から判るように, $n \in \mathbb{Z}$ が奇数であれば, $n!!$ は n 以下の全ての奇数の積を計算し, $n \in \mathbb{Z}$ が偶数であれば, $n!!$ は n 以下の全ての偶数の積を計算します. したがって, 任意の自然数 n に対して ‘$n! = n!!(n-1)!!$’ が成立します:

```
(%i6) 10!;
(%o6)                                3628800
(%i7) 10!!;
(%o7)                                 3840
(%i8) 9!!;
(%o8)                                  945
(%i9) 10!!*9!!;
(%o9)                                3628800
```

ここで, 階乗と Γ 函数には密接な関連があります. Maxima には Γ 函数項と階乗の函数項を置換する函数もあります:

階乗に関連する函数の構文

genfact(⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨ 式$_3$⟩)
makefact(⟨ 式 1⟩)
multinomial_coeff(⟨a_1⟩, ⋯ ,⟨a_n⟩)
multinomial_coeff()
minfactorial(⟨ 式 ⟩)

genfact 函数: 一般化された階乗を計算する函数です. この函数は次の式で定義されています:

$$\mathrm{genfact}(a,b,c) \stackrel{def}{=} \prod_{i=1}^{b}\{(a-(b-i))c\}$$

したがって, 整数 n に対して 'genfact(n,n,1)' が 'n!' に対応し, 'genfact(n, n / 2, 2)' が 'n!!' に対応します.

makefact 函数: Γ 函数項を階乗に変換するための函数です. なお, Maxima では Γ 函数は gamma 函数で表現されており, makefact 函数は与式に gamma 函数の名詞型を含まない式に対しては, そのまま返却する函数です:

```
(%i62) makefact(gamma(x));
(%o62)                                (x - 1)!
(%i63) makefact(gamma(x)*gamma(y));
(%o63)                             (x - 1)! (y - 1)!
(%i64) makefact(gamma(x)>gamma(y));
(%o64)                           (x - 1)! > (y - 1)!
(%i65) makefact(x^2+1);
                                      2
(%o65)                               x  + 1
```

なお, この makefact 函数の逆操作を行う函数が makegamma 函数です. この函数は makefact 函数と同様に階乗の函数項を gamma 函数項で置換する函数です.

minfacotrial 函数: 階乗を含む式の簡易化を行う函数です. $\frac{m!}{(m-3)!}$ の様な階乗を含む有理式に有効ですが, その一方で, $m(m-1)!$ の簡易化では使えません.

```
(%i17) minfactorial(m!/(m-3)!);
(%o17)                           (m - 2) (m - 1) m
```

9.1.2 剰余

剰余を計算する函数として, mod 函数が存在します:

mod 函数の構文

mod(⟨ 整数$_1$⟩,⟨ 整数$_2$⟩)

mod 函数: 引数を二つ取る函数で, 第 1 引数を第 2 引数で割ったときの剰余を返します. すなわち, 整数 a,b,c,r に対して $a=cb+r$ であれば, 'mod(a, b)' は 'r' を返します:

```
(%i12) mod(129,7);
(%o12)                                3
(%i13) mod(129,3);
(%o13)                                0
```

なお, polymod 函数 の様に大域変数 modulus の影響を mod 函数は一切受けません.

9.1.3 Bell 数

Bell 数 B_n は n 個の元で構成される集合のグループ化の総数です. たとえば, 0 成分の集合は $\{\emptyset\}$ となるので B_0 を 1 で定めます. また, 2 成分の集合 $\{a, b\}$ の場合, $\{\{a\}, \{b\}\}$ と $\{\{a, b\}\}$ の二種類となるために $B_2 = 2$ となります.
Maxima では belln 函数で, この Bell 数の計算を行います:

belln 函数の構文

belln(⟨ 正整数値 ⟩)

ここでは makelist 函数も併用して, 1 個から 10 個の元を持つ集合の Bell 数を計算させてみましょう:

```
(%i16) makelist(belln(x),x,1,10);
(%o16)          [1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975]
```

9.1.4 Bernoulli 数

Bernoulli 数 B_n は歴史的に, 次の冪乗和から得られた数です:

$$\sum_{i=1}^{n} i^k = \sum_{j=0}^{k} \binom{k}{j} B_j \frac{n^{k+1-j}}{k+1-j}$$

ここで, B_n は次の漸化式も満します:

$$\sum_{j=0}^{k} \binom{k+1}{j} B_j = k+1$$

なお, 関孝和も Bernoulli とは独立して Bernoulli 数を発見しています[1].
なお, 歴史的な定義では $B_1 = 1/2$ となりますが, $B_1 = -1/2$ とする流儀もあります. これは, Maxima の bern 函数で採用されており, この場合, Bernoulli 数は次の漸化式で定義されます:

Bernoulli 数の漸化式による定義

- $B_0 = 1$
- $\displaystyle\sum_{i=0}^{n} \binom{n+1}{i} B_i = 0 \qquad (n \geq 1)$

Bernoulli 数は漸化式による定義の他に, 形式的冪級数を用いても定義されます.
まず, 歴史的な Bernoulli 数 ($B_1 = 1/2$) に対応する冪級数 (母函数) による定義を示しておきます:

$$\frac{se^s}{e^s - 1} = \sum_{n=0}^{\infty} \frac{B_n}{n!} s^n \qquad (|s| < 2\pi)$$

次に, Maxima でも使われている Bernoulli 数 ($B_1 = -1/2$) の母函数による定義を次に示します:

[1] Bernoulli の著書「Ars Conjectandi」は 1713 年, 関孝和の著書「括要算法」は正徳 2 年 (1712 年) に出版 [2]

$$\frac{s}{e^s-1}=\sum_{n=0}^{\infty}\frac{B_n}{n!}s^n \qquad (|s|<2\pi)$$

Maxima で Bernoulli 数に関連する函数として次のものがあります:

Bernoulli 数に関連する函数の構文
bern(⟨ 正整数値 ⟩)
bernpoly(⟨ 変数 ⟩,⟨ 正整数値 ⟩)

bern 函数: Bernoulli 数 B_n を出力する函数です. この bern 函数では歴史的な Bernoulli 数ではありません. このことを makelist 函数と taylor 函数を用いて確認してみましょう:

```
(%o42)                                  true
(%i43) makelist(bern(n)/n!,n,0,10);
                 1  1          1          1              1              1
(%o43)     [1, - -, --, 0, - ---, 0, -----, 0, - -------, 0, --------]
                 2  12        720       30240        1209600       47900160
(%i44) taylor(s/(exp(s)-1),s,0,10);
                        2     4       6          8           10
                   s    s     s       s          s           s
(%o44)/T/     1 - - + -- - --- + ----- - ------- + -------- + . . .
                   2    12    720    30240    1209600    47900160

(%i45) taylor(s*exp(s)/(exp(s)-1),s,0,10);
                        2     4       6          8           10
                   s    s     s       s          s           s
(%o45)/T/     1 + - + -- - --- + ----- - ------- + -------- + . . .
                   2    12    720    30240    1209600    47900160
```

初期状態で Bernoulli 数の '0' は除外されていません. もし, Bernoulli 数から '0' を除外する必要がある場合には, 大域変数 zerobern を 'false' に設定します. ここで大域変数 zerobern が 'false' であれば, 当然ながら, '0' の Bernoulli 数が除外されて並び直されるので, 大域変数 zerobern が 'true' の場合と 'false' の場合で 'bern(n)' の値が異なることに注意が必要です：

```
(%i38) zerobern:true;
(%o38)                                  true
(%i39) makelist(bern(n)/n!,n,0,6);
                                1  1          1          1
(%o39)                    [1, - -, --, 0, - ---, 0, -----]
                                2  12        720       30240
(%i40) zerobern:false;
(%o40)                                  false
(%i41) makelist(bern(n)/n!,n,0,6);
                             1  1       1     5        691      7
(%o41)                 [1, - -, --, - ---, ----, - ------, ----]
                             2  12     180   1584     327600   4320
```

この例で判る様に, 大域変数 zerobern を変更することで bern(n) の値が異なって n!とのズレが発生するため, bern(n)/n!の値が異なることになります.

bernpoly 函数:　Bernoulli 多項式 $B_n(x)$ を生成する函数です．この Bernoulli 多項式 $B_n(x)$ は Bernoulli 数 B_n を用いると次の式で定義される多項式です:

$$B_n(x) \stackrel{def}{=} \sum_{k=0}^{n} \binom{n}{k} B_{n-k} x^k$$

Maxima の bernpoly 函数は引数を二つ取り, 第 1 引数が多項式の変数, 第 2 引数が多項式の次数となります.

```
(%i50) makelist(bernpoly(x,n),n,0,4);
                                                  2
                  1    2         1    3    3 x      x    4       3     2    1
(%o50)      [1, x - -, x  - x + -, x  - ---- + -, x  - 2 x  + x  - --]
                  2              6          2      2                      30
```

さらに Bernoulli 多項式 $B_n(x)$ は次の性質を持っています:

Bernoulli 多項式の性質

- $B_n(0) = B_n$
- $\dfrac{d}{dx} B_n(x) = nB_{n-1}(x)$

そこで, $n = 10$ まで $B_n(0) - B_n$ の値と $\dfrac{d}{dx} B_n(x) - nB_{n-1}(x)$ の値を確認しましょう:

```
(%i8) makelist(bernpoly(0,i),i,0,10)-makelist(bern(i),i,0,10);
(%o8)                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
(%i9) makelist(diff(bernpoly(x,i),x),i,1,10)-makelist(bernpoly(x,i-1)*i,i,1,10),
      expand;
(%o9)                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

この様に両者が一致することが判ります.
Maxima には Bernoulli 数に関連する大域変数 zerobern があります:

Bernoulli 数に関連する大域変数

大域変数名	既定値	概要
zerobern	true	函数 bern の制御に関連

大域変数 zerobern:　Bernoulli 数に '0' を含めるかどうかを指定する大域変数です. 真理値を設定し, 'true' の場合には '0' を含めますが, 'false' の場合には '0' を除外します.

9.1.5　B(beta) 函数

B(beta) 函数は第 1 種 Euler 積分と呼ばれる特殊函数で, 次の式で定義されます:

$$B(x, y) \stackrel{def}{=} \int_0^1 t^{x-1}(1-t)^{y-1} dt$$

Maxima では B(beta) 函数は beta 函数で表現されます:

bete 函数の構文

beta(⟨ 式$_1$⟩,⟨ 式$_2$⟩)

beta 函数: beta 函数は次の関係式を用いて定義されています:

$$\mathrm{beta}(x,y)=\frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

```
(%i2) beta(3,9);
                                1
(%o2)                          ---
                               495
(%i3) beta(n,2);
                                1
(%o3)                      ---------
                           n (n + 1)
(%i4) beta(3,n);
                                2
(%o4)                  -----------------
                       n (n + 1) (n + 2)
(%i5) beta(3.0,n);
(%o5)                     beta(3.0, n)
(%i6) beta(3.0,8.);
(%o6)                 .002777777777777777
(%i7) beta(1/2,3/2);
                               %pi
(%o7)                          ---
                                2
```

beta 函数には簡易化に関連する大域変数 beta_args_sum_to_integer があります:

beta 函数に関連する大域変数

大域変数名	既定値	概要
beta_expand	flase	beta 函数の展開で参照
beta_args_sum_to_integer	false	函数 beta の簡易化で参照

大域変数 beta_expand: beta_expand が 'true' の場合に, 'beta(a+n,b)', 'beta(a-n,b)', 'beta(a,b+n)', 'beta(a,b-n)' といった函数項の簡易化が 'n' が整数 (属性として integer 型が与えられたものではなく, LISP の整数型) の場合に実行されます.

大域変数 beta_args_sum_to_integer: beta_args_sum_to_integer の値が 'true' で, 二つの引数の和が整数 (LISP の整数型) であれば, 式の簡易化が実行されます:

```
(%i8) beta(x+1/2,3/2-x);
                                3           1
(%o8)                     beta(- - x, x + -)
                                2           2
(%i9) beta_args_sum_to_integer:true;
(%o9)                           true
```

```
(%i10) beta(x+1/2,3/2-x);
                                          1
                                   %pi (x - -)
                                          2
(%o10)                          ----------------
                                         3
                                sin(%pi (- - x))
                                         2
```

ここでの整数型の判定では, 単純に展開した式を使って判定しており, 文脈は無関係です. また, 単純に引数が有理数で, その和が整数となる場合, この大域変数とは無関係に簡易化が実行されます:

```
(%i11) beta_args_sum_to_integer:false;
(%o11)                                 false
(%i12) beta(1/2,3/2);
                                        %pi
(%o12)                                  ---
                                         2
(%i13) beta(5/2,3/2);
                                        %pi
(%o13)                                  ---
                                        16
```

9.1.6　二項係数

binomial 函数の構文

binomial(⟨ 式$_1$⟩,⟨ 式$_2$⟩)
binomial(⟨ 正整数値$_1$⟩,⟨ 正整数値$_2$⟩)

binomial 函数:　二項係数を返す函数です. 二つの引数を評価した結果が共に正整数であれば, 通常の階乗を用いた計算で処理されますが, 双方が数値で, どちらかが実数の場合, gamma 函数を用いた処理になります. したがって, binomial 函数が返却する数値は整数型か浮動小数点数になります. さらに, Maxima の binomial 函数には整数値以外の式を与えることも可能です. この場合, binomial 函数は ⟨ 式$_1$⟩ と ⟨ 式$_2$⟩ の何れもが数値ではなく, ⟨ 式$_1$⟩ − ⟨ 式$_2$⟩ が正整数となる場合のみに有理式を返却し, それ以外は名詞型で返却します.
引数の双方が数値で, その内, どちらか一方が非整数のときに gamma 函数を用いた計算を実行します:

```
(%i14) binomial(17,3);
(%o14)                                  680
(%i15) binomial(17.0,3.0);
(%o15)                          680.0000000000005
(%i16) gamma(18.0)/(gamma(15.0)*gamma(4.0));
(%o16)                          680.0000000000005
```

```
(%i17) binomial((x-1)^3+11,(x-1)^3+7);
                  3                3                3                3
       ((x - 1)  + 8) ((x - 1)  + 9) ((x - 1)  + 10) ((x - 1)  + 11)
(%o17) ---------------------------------------------------------------
                                    24
(%i18) binomial((x-1)^3+11,1.2);
                                      3
(%o18)                 binomial((x - 1)  + 11, 1.2
```

9.1.7 Euler 数

ここでの Euler 数 E_n は次の式から定義される数です:

$$\frac{1}{\cos x}=\sum_{n=0}^{\infty}\frac{E_n}{n!}x^n$$

euler 函数の構文

euler(⟨ 正整数値 ⟩)

euler 函数: Euler 数 E_n を返す函数です. 引数は Euler 数の性格上, 正整数に限定されます:

```
(%i25) makelist(euler(i),i,0,10);
(%o25)          [1, 0, - 1, 0, 5, 0, - 61, 0, 1385, 0, - 50521]
```

Euler-Mascheroni 定数 γ: 計算方法は Numbers,Constants and Computation[135] の *Euler's constant g* で解説されている Bessel 函数を用いた方法を採用しています.
この Euler の定数は $\gamma=\lim_{n\to\infty}\left(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}-\log(n)\right)$ で計算される定数で, gamma 函数との関係では,$\Gamma'(1)=-\gamma$ となる性質があります:

Euler-Mascheroni 定数

%gamma	0.5772156649015329⋯

9.1.8 Fibonacci 数

Fibonacci 数は次の漸化式で定義される数です:

FIbonacci 数を定義する漸化式

- $F_0=0$
- $F_1=1$
- $F_{n+1}=F_n+F_{n-1}, (n>1)$

Maxima には Fibonnacci 数に関連する函数として, fib 函数と fibtophi 函数があります:

Fibonacci 数に関連する函数の構文

fib(⟨ 正整数値 ⟩)
fibtophi(⟨ 式 ⟩)

fib 函数:　Fibonacci 数を返す函数です.
なお, fib 函数で一度計算を行うと, 一段前の Fibonacci 数が大域変数 prevfib に保存されています.

fibtophi 函数:　fib(n) を $\dfrac{\pi^n - (1-\pi)^n}{2\pi - 1}$ で置換える函数です. ここで引数が fib(⟨ 式 ⟩) の書式であり, ⟨ 式 ⟩ が整数以外の式でなければ, 引数をそのまま返します:

```
(%i60) fibtophi(fib(x));
                                  x              x
                              %phi  - (1 - %phi)
(%o60)                        -------------------
                                  2 %phi - 1
(%i61) fibtophi(fib(1+%i));
                             %i + 1              %i + 1
                         %phi       - (1 - %phi)
(%o61)                   ----------------------------
                                  2 %phi - 1
(%i62) fibtophi(x^3);
                                     3
(%o62)                              x
(%i63) fibtophi(fib(3));
(%o63)                               2
```

大域変数 prevfib

prevfib	函数 fib で計算した Fibonacci 数の一つ前の値を保持

大域変数 prevfib:　fib(⟨n⟩) の計算の際に用いた fib(⟨$n-1$⟩) の値が保存される大域変数です:

```
(%i69) fib(31);
(%o69)                          1346269
(%i70) prevfib;
(%o70)                          832040
(%i71) fib(30);
(%o71)                          832040
```

9.1.9　Γ 函数

Γ 函数は以下の式で定義される函数です:

$$\Gamma(x) \stackrel{def}{=} \int_0^\infty t^{x-1} e^{-t} dt$$

ここで, Γ 函数と正整数 n の階乗 $n!$ との間には $\Gamma(n) = (n-1)!$ となる関係があります.
なお, Γ 函数は Maxima では gamma 函数で表現されます:

Γ 函数に関連する函数の構文

gamma(⟨ 式 ⟩)

gamma 函数: 与えられた引数を評価し, その結果が整数であれば整数を, 浮動小数点数であれば浮動小数点数を返し, それ以外の結果に対しては名詞型を返す函数です.
この gamma 函数の簡易化では, 大域変数 factlim や大域変数 gammalim が用いられます:

gamma 函数に関連する大域変数

大域変数	初期値	概要
factlim	-1	階乗の展開を制御
gammalim	1000000	gamma 函数の簡易化を制御

大域変数 factlim: 正整数の階乗や正整数を引数とする gamma 函数の自動展開を制御します. この値を越える正整数の階乗は自動展開されません. また, gamma 函数は引数の正整数から 1 を引いた値が大域変数 factlim の値を越えると自動展開されません. なお, 大域変数 factlim が '-1' の場合に全ての正整数の階乗や正整数を引数とする gamma 函数が自動的に展開されます:

```
(%i25) factlim:10;
(%o25)                                10
(%i26) 10!;
(%o26)                              3628800
(%i27) 11!;
(%o27)                                11!
(%i28) gamma(11);
(%o28)                              3628800
(%i29) gamma(12);
(%o29)                                11!
```

大域変数 gammalim: gamma 函数の簡易化で用いられる大域変数です.gamma 函数の引数の絶対値が大域変数 gammalim の値よりも小さければ簡易化を行います. 猶,gamma 函数の引数が正整数の場合, 大域変数 factlim も gamma 函数の簡易化に関わります.
ここで形式的に gamma 函数を階乗で置換える函数があります.

makegamma 函数の構文

makegamma(⟨ 式 ⟩)

makegamma 函数: 階乗項を含む式に対して, 階乗項を形式的に gamma 函数項で置換する函数です:

```
(%i67) makegamma(n!);
(%o67)                              gamma(n + 1)
(%i68) makegamma(n!/m!);
                                    gamma(n + 1)
(%o68)                              ------------
                                    gamma(m + 1)
(%i69) makegamma(n!=m!*(m+1));
```

```
(%o69)                    gamma(n + 1) = (m + 1) gamma(m + 1)
```

9.1.10 多重対数函数

多重対数函数 $\mathrm{Li}_s(z)$ は次で定義される函数です:

$$\mathrm{Li}_s(z) \stackrel{def}{=} \sum_{k=1}^{\infty} \frac{z^k}{k^s} \quad ,|z| < 1$$

次数 s が 1 の場合, $-\log(1-z)$ に簡易化されます.
Maxima では多重対数函数 $\mathrm{Li}_s(Z)$ は li 函数で表現されています:

li 函数の構文

li[⟨ 整数 ⟩](⟨ 引数 ⟩)

li 函数: ⟨ 整数 ⟩ が多重対数函数項 $\mathrm{Li}_s(z)$ の次数 s に対応し, ⟨ 引数 ⟩ が引数 z に対応します. li 函数は引数が, 実浮動小数点数か複素数の場合, あるいは ev 函数による評価を行うことで, 函数項ではなく実際の数値を返却します:

```
(%i44) li[3](4);
(%o44)                                  li   (4)
                                          3
(%i45) li[3](4),numer;
(%o45)              4.375154163472729 - 3.018775324000599 %i
(%i46) li[3](4.0);
(%o46)              4.375154163472729 - 3.018775324000599 %i
```

9.1.11 Möbius の函数 μ

Möbius の函数 μ は次で定義されます:

Möbius の $\mu(n)$ 函数の定義

$n = 1$	$\rightarrow$	$\mu(1) = 1$
n が素数の平方で割切れる	$\rightarrow$	$\mu(n) = 0$
n が k 個の異なる素数の積である	$\rightarrow$	$\mu(n) = (-1)^k$

この Möbius の $\mu(n)$ 函数が満す重要な性質に次のものがあります:

Möbius 函数が満す重要な性質

- $\displaystyle\sum_{d|n} \mu(d) = \begin{cases} 1 & n = 1 \text{ の場合} \\ 0 & n > 1 \text{ の場合} \end{cases}$
- $\mu(mn) = \begin{cases} \mu(m)\mu(n) & m \text{ と } n \text{ が互いに素の場合} \\ 0 & m \text{ と } n \text{ が互いに素でない場合} \end{cases}$

さらに, μ 函数は Riemann の ζ 函数の逆数にも現れる函数です:

$$\frac{1}{\zeta(s)} = \sum_{n=1}^{\infty} \frac{\mu(n)}{n^s}$$

Maxima で μ 函数は moebius 函数で表現されます:

moebius 函数の構文

moebius(⟨ 正整数 ⟩)
moebius(⟨ 変数名 ⟩ = ⟨ 正整数 ⟩)

moebius 函数: 引数として正整数を取りますが, 整数値リスト, 整数値集合や整数値行列も扱えます:

```
(%i57) moebius(6);
(%o57)                                1
(%i58) moebius(7);
(%o58)                               - 1
(%i59) moebius([1,2,3,4,5]);
(%o59)                      [1, - 1, - 1, 0, - 1]
(%i60) moebius(matrix([1,2],[3,4]));
                                [  1   - 1 ]
(%o60)                          [          ]
                                [ - 1   0  ]
(%i61) moebius({7,8,9,10,11});
(%o61)                         {- 1, 0, 1}
```

また, 'moebius(⟨ 変数名 ⟩ = ⟨ 正整数 ⟩)' の入力も可能で, この場合は 'moebius(⟨ 変数名 ⟩) = moebius(⟨ 正整数 ⟩)' の形式で答を返します:

```
(%i77) moebius(x=5);
(%o77)                              moebius(x) = - 1
(%i78) moebius(x=[1,2,3,4,5,6]);
(%o78)                  moebius(x) = [1, - 1, - 1, 0, - 1, 1]
```

9.1.12 numfactor 函数

numfactor 函数の構文

numfactor (⟨ 式 ⟩)

numfactor 函数: ⟨ 式 ⟩ から単項式や函数項の数値の因数のみを取り出す函数です. ここで, 返却される数値は numberp 函数が 'true' を返す対象に限定され, declare 函数で integer 等と宣言した対象は含まれません.
また, ⟨ 式 ⟩ が数値であれば, その数値を返却し, 函数項や単項式に数値因子が含まれない場合, 単項式や函数項でない場合は '1' を返します.
なお, 和の中の全ての項の係数の GCD が必要ならば, content 函数を用いましょう.

```
  (%i92) gamma(7/2);
                                      15 sqrt(%pi)
(%o92)                                ------------
                                           8
(%i93) numfactor(%);
                                          15
(%o93)                                    --
                                          8
(%i94) numfactor(10*x^128);
(%o94)                                    10
(%i95) numfactor(10*sin(x));
(%o95)                                    10
(%i96) numfactor(10*sin(x)+2*x^2);
(%o96)                                     1
(%i97) numfactor(128);
(%o97)                                    128
```

9.1.13 digamma(polygamma) 函数 ψ

digamma 函数 ψ は, $\psi_n(x)$ で $\log\Gamma(x)$ の $n+1$ 階の導函数を返す函数です:

$$\psi_n(x) \stackrel{def}{=} \frac{d^{n+1}}{dx^{n+1}} \log\Gamma(x)$$

Maxima には psi 函数と bfpsi 函数の二つがあります:

digamma 函数に関連する函数の構文

psi[⟨ 正整数 ⟩](⟨ 式 ⟩)
bfpsi(⟨ 正整数$_1$⟩, ⟨ 式 ⟩, ⟨ 正整数$_2$⟩,)
bfpsi0(⟨ 式 ⟩, ⟨ 正整数$_2$⟩,)

psi 函数： 'psi[n](x)' で $\log\Gamma(x)$ の $n+1$ 階の微分を計算します. 数値計算が必要であれば, bfpsi 函数を用います.

```
(%i126) diff(log(gamma(x)),x,4);
(%o126)                                  psi  (x)
                                           3
```

bfpsi 函数： 任意精度の浮動小数点数を返却する函数で, 'bfpsi(n, x, m)' により $\log\Gamma(x)$ の $n+1$ 階の導函数の数値を m 桁の精度で計算します. この函数は bffac パッケージに含まれており, 呼出されると自動的に読込まれます.

```
(%i127) bfpsi(0,3,8);
(%o127)                                9.2278433b-1
(%i128) bfpsi(0,3,16);
(%o128)                            9.227843350984671b-1
(%i129) bfpsi(0,3,100);
(%o129) 9.22784335098467139393487909917597568957840664060076401194232765115132\
27322233532906305293670825325O5b-1
```

bfpsi0 函数： 任意精度の浮動小数点数を返却する函数で, 'bfpsi0(x, m)' により $\frac{d}{dx}\log\Gamma(x)$ の数値を m 桁の精度で計算します. つまり, 'bfpsi0(x,m)' と 'bfpsi(0,x,m)' は同じです. この函数も bffac パッケージに含まれており, 呼出されると自動的に読込まれます.

```
(%i132) bfpsi(0,10,10)-bfpsi0(10,10);
(%o132)                                  0.0b0
```

9.1.14 ζ 函数

Riemann の ζ 函数は $\Re(s) > 1$ である $s \in \mathbb{C}$ に対して

$$\zeta(s) \stackrel{def}{=} \sum_{n=1}^{\infty} \frac{1}{n^s}$$

で定義されます. また, Γ 函数を用いても定義ができます:

$$\zeta(s) \stackrel{def}{=} \frac{1}{\Gamma(s)} \int_0^{\infty} \frac{u^{s-1}}{e^u - 1} du$$

さらに素数との関係では次の式 (Euler 積) が知られています.

$$\zeta(s) = \prod_{p\in\{\text{ 素数の集合 }\}} \frac{1}{1-p^s}$$

また, Bernoulli 数との関係では, 次の関係式が知られています:

$$\zeta(2k) = \frac{(-1)^{k-1}}{2}\frac{B_{2k}}{(2k)!}(2\pi)^{2k} \qquad (k \in \mathbb{N})$$

この ζ 函数で有名な未解決問題が Riemann 予想です.

Riemann 予想

複素平面上の $\zeta(s)$ の非自明な零点の実部は全て $\frac{1}{2}$ である.

ここで自明な零点は負の偶数 $(-2, -4, \ldots, 2k, \ldots)$ で, 非自明な零点は $0 < \Re(s) < 1$ のみであることが知られています.
Maxima の函数で ζ 函数に関連するものを示しまします:

ζ 函数に関連する函数の構文

zeta(⟨ 正整数値 ⟩)
bfzeta(⟨ 変数 ⟩,⟨ 正整数値 ⟩)
bfhzeta(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩,⟨ 正整数値 ⟩)

zeta 函数: Riemann の ζ 函数を Maxima に実装した函数で, 引数が整数, 有理数, 浮動小数点数の以外の型であれば, 名詞型の式を返却します.
zeta 函数の簡易化は内部函数 simp-zeta で行われます. ここで, 引数 n が整数の場合, 'n ¿ 1' を満す奇数であれば名詞型, 'n ¡ 2' の奇数であれば '-bern(1-n)/(1-n)' を返却, n が偶数で大域変数 zeta%pi が 'false' なら名詞型, n が偶数で大域変数 zeta%pi が 'true' であれば, '%pi^n*(2^(n-1)/n!)*abs(bern(n))' が返却され, それ以外は名詞型になります:

bfzeta 函数: 任意精度の浮動小数点数を返却する函数です. この函数は引数を二つ取り, 第 1 引数の第 2 引数で指定した桁迄の ζ 函数の値を返します. なお, この函数で第 1 引数は整数, 有理数, 超越数と代数的数を含む無理数, そして, 第 2 引数は正整数でなければなりません. この bfzeta 函数は bffac パッケージに含まれる函数で, 呼出されると自動的に読込まれます.

bfhzeta 函数: Hurwitz の ζ 函数で, 任意精度の浮動小数点数を返却する函数です. 具体的には, 'bfhzeta(s, h, n)' で $\sum_{k=0}^{\infty}(k+h)^{-s}$ の n 桁の数値を返却します. この bfhzeta 函数は bffac パッケージに含まれる函数で, 呼出されると自動的に読込まれます.
ここで, zeta 函数の計算に関連する大域変数として, zeta%pi があります:

zeta 函数に関連する大域変数

大域変数	既定値	概要
zeta%pi	true	zeta 函数の引数 n が偶数の場合,%pi^n を表示するかどうかを指定

大域変数 zeta%pi: zeta 函数の簡易化で参照される大域変数です. zeta 函数の簡易化では内部函数の simp-zeta 函数が用いられますが, 大域変数 zeta%pi が 'true' で, zeta 函数の引数が数値の場合に bern 函数を用いて Bernoulli 数を使った表現に簡易化を行います.

```
(%i10) zeta%pi:false;
(%o10)                                false
(%i11) zeta(10);
(%o11)                               zeta(10)
(%i12) zeta%pi:true;
(%o12)                                true
(%i13) zeta(10);
                                        10
                                      %pi
(%o13)                                -----
                                      93555
```

9.1.15 連分数に関連する函数

連分数に関連する函数の構文

cf(⟨ 式 ⟩)
cfexpand(⟨ リスト ⟩)
cfdisrep(⟨ リスト ⟩)

cf 函数: 与えられた式の連分数による表現をリスト形式で出力する函数です. cf 函数が出力するリストの長さを制御する大域変数が大域変数 cflength です.
なお, cf 函数の引数として利用可能なのは, 整数, 有理数, 浮動小数点数と代数的数であり, それ以外の超越数や複素数等の値や式に対してはエラーを返します.

cfexpand 函数: 連分数表現を有理数表現に戻す函数です. cfexoand 函数は二次の正方行列を返し, この行列の (1,1) 成分を分子, (2,1) 成分を分母とする分数が cf 函数による連分数表現を分数に戻したものに対応します.
なお, この函数は連分数表現に依存するために大域変数 cflength の値に依存します:

```
(%i35) cflength:5;cfexpand(cf((1+sqrt(5))/2));
(%o35)                                 5
                                  [ 4181   1597 ]
(%o36)                            [             ]
                                  [ 2584   987  ]
(%i37) cflength:6;cfexpand(cf((1+sqrt(5))/2));
(%o37)                                 6
```

```
                                    [ 17711   6765 ]
(%o38)                              [              ]
                                    [ 10946   4181 ]
```

cfdisrep 函数:　cf 函数で生成した連分数のリスト表現を通常の連分数の表示に変換する函数です. 内部的には与えられたリストを有理数に変換する函数となっています:

```
(%i31) is(cfdisrep(cf(1.20))-1.2=0),pred;
(%o31)                                 false
(%i32) is(cfdisrep(cf(1.20))-12/10=0),pred;
(%o32)                                 true
```

この例では浮動小数点数が cf 函数によって整数リストで置換えられ, さらに cfdisrep 函数で有理数に変換されたため, 本来の浮動小数点数の差を取っても型の違いによって 0 にならないことを示しています.

連分数に関連する大域変数

大域変数	初期値	概要
cflength	1	cf 函数の出力を制御

大域変数 cflength:　cf 函数が出力する連分数の段数, すなわち, リスト長を制御する大域変数です. この初期値は 1 となっており, この場合に cf 函数が出力する段数は 1, リスト長では `cflength` + 1 になります:

```
(%i1) cf(sqrt(2));
(%o1)                                [1, 2]
(%i2) cfdisrep(%);
                                        1
(%o2)                                1 + -
                                        2
(%i3) cflength:5;
(%o3)                                  5
(%i4) cfdisrep(cf(sqrt(2)));
                                        1
(%o4)                          1 + ----------------
                                           1
                                   2 + -------------
                                              1
                                      2 + ---------
                                                 1
                                         2 + -----
                                                   1
                                              2 + -
                                                   2
```

9.1.16　二次体に関連する函数

二次体に関連する函数の構文

qunit(〈 正整数 〉)

qunit 函数: 一つの引数を取り, その引数が正整数 $\langle n \rangle$ の場合, ノルムが 1 となる二次体 $\mathbb{Z}[\sqrt{\langle n \rangle}]$ の元, すなわち, $a^2 - nb^2 = 1$ (Pell の方程式) を満し, $b > 0$ となる元 $a + b\sqrt{\langle n \rangle}$ を返す函数です:

```
(%i72) qunit(23);
(%o72)                          5 sqrt(23) + 24
(%i73) %*substpart(-1*part(%,1),%,1);
(%o73)                (24 - 5 sqrt(23)) (5 sqrt(23) + 24)
(%i74) %,expand;
(%o74)                                  1
```

この例では substpart 函数と part 函数を用いて $5\sqrt{23} + 24$ を $-5\sqrt{23} + 24$ に変換していますが, これは Maxima の差の内部表現が a+(-b) となっているために subst("-",%,0) とするとエラーになるので, この様な代入を行っています. ここで, qunit 函数は解の計算を行うために内部で isqrt 函数を用いています.

9.1.17 ifactor パッケージに含まれる函数

ifactor パッケージに含まれる函数の構文

ifactor(⟨ 正整数 ⟩)
primep(⟨ 正整数 ⟩)
next_prime(⟨ 正整数 ⟩)
prev_prime(⟨ 正整数 ⟩)
power_mod(⟨ 正整数$_1$⟩,⟨ 正整数$_2$⟩,⟨ 正整数$_3$⟩)
inv_mod(⟨ 正整数$_1$⟩,⟨ 正整数$_2$⟩)

ifactor パッケージには内部変数*small-primes*に 9973 以下の素数のリストが束縛されています. また, 内部変数*largest-small-primes*に内部変数*small-primes*に束縛された素数リストの最大元 9973 が束縛されています.

ifactors 函数: 正整数に対して素数分解を行い, 素数と冪のリストを成分とするリストを返す函数です.
すなわち, $n = p_1^{k_1} \cdots p_m^{k_m}$ に対し, ifactor 函数は素数と冪次数の対のリスト $[[p_1, k_1], \cdots, [p_m, k_m]]$ を返します.
大域変数 ifactor_verbose[2]が true の場合, ifactor 函数は詳細な報告を返します.

```
(%i13) ifactors(818378923834);
(%o13)              [[2, 1], [97, 1], [42283, 1], [99767, 1]]
(%i14) ifactor_verbose:true;
(%o14)                                  true
(%i15) ifactors(818378923834);

Starting factorization of n = 818378923834
Factoring out small prime: 2 (degree:1)
Factoring out small prime: 97 (degree:1)
Factoring n = 4218448061
```

[2]ifactors_verbose でない事に注意!紛らわし事に, この変数だけ ifactor に s がありません.

```
Pollard rho: round #1 of 5 (lim=10000)
Pollard rho: found factor 42283 (5 digits)
========> Prime factor: 42283

========> Prime factor: 99767

(%o15)                [[2, 1], [97, 1], [42283, 1], [99767, 1]]
```

大域変数 factors_only が true の場合, $n = p_1^{k_1} \cdots p_m^{k_m}$ に対し, ifactor 函数は素数の冪のリスト $\left[p_1^{k_1}, \cdots, p_m^{k_m}\right]$ で返します.

```
(%i12) factors_only:true;
(%o12)                                true
(%i13) ifactors(8218344);
(%o13)                          [2, 3, 17, 20143]
(%i14) factors_only:false;
(%o14)                                false
(%i15) ifactors(8218344);
(%o15)              [[2, 3], [3, 1], [17, 1], [20143, 1]]
```

なお, この素因数分解では楕円曲線を用いた手法 (Elliptic Curve Method(ECM)) を採用しています[3].
この函数では,Pollard の Rho 素数判定法と楕円曲線を用いる ECM が用いられています. 最初に Pollard の Rho 素数判定法が適用され, 次に ECM が利用されます.
ここで, Pollard の Rho 素数判定法を制御する大域変数には, 大域変数 pollard_rho_limit_step, 大域変数 pollard_rho_limit と大域変数 pollard_rho_tests があります. そして, ECM を制御する大域変数は大域変数 ecm_number_of_curves, 大域変数 ecm_limit, 大域変数 ecm_max_limit, 大域変数 ecm_limit_delta になります.

primep 函数: 与えられた正整数に対し, 素数であるかどうかを判定する真理函数です. この真理値集合は {true,false} です.
primep 函数は 1 から 17 迄の素数に対しては直接表を参照し, 341550071728321 よりも小さな正整数に対しては内部函数の primep-small を用い, それ以上の正整数に対しては内部函数の primep-prob を用います.
これらの函数での素数判定は内部函数の miller-rabin 函数が用いられています. この miller-rabin 函数は文字通りに Miller-Rabin 素数判定法を用いて判定を行う函数です. なお, Miller-Rabin 素数判定法で合成数を素数と判定する可能性が皆無ではありません.
まず, primep-small 函数では 2 以上 17 以下の素数のリストを用いて Miller-Rabin 素数判定法を用いています.
そして, primep-prob 函数では大域変数 primep_number_of_tests で指定した回数程, Miller-Rabin 素数判定を実行します. ここで素数と判定されると, 最後に primep-lucas 函数による Lucas 素数判定法を用います.
この primep-lucas 函数内部では大域変数 save_primes が true の場合, Lucas 素数判定法を通った正整数は内部変数*large-primes*に追加されます.

[3] ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.Brent/rpb161.dvi.gz 参照

```
(%i2) :lisp *large-primes*
NIL
(%i2) primep(next_prime(818378923834290589778878923337));
(%o2)                                   true
(%i3) :lisp *large-primes*
NIL
(%i3) save_primes:true;
(%o3)                                   true
(%i4) primep(next_prime(818378923834290589778878923337));
(%o4)                                   true
(%i5) :lisp *large-primes*
(818378923834290589778878923337)
(%i5) primep(next_prime(818378923834290589778878923337));
(%o5)                                   true
(%i6) :lisp *large-primes*
(818378923834290589778878923337)
(%i6) primep(next_prime(818378923834290589778878923337));
(%o6)                                   true
(%i7) :lisp *large-primes*
(818378923834290589778878923337 818378923834290589778878923337)
```

next_prime 函数: 与えられた正整数を越える素数で最小の素数を返す函数です. 与える正整数は素数でも合成数でも構いません. この函数の内部で素数判定に primep 函数を用いています. したがって, 341550071728321 よりも大きな素数を求めた場合, 大域変数 save_primes が true であれば内部変数*large-primes*に, その素数が登録されます.

```
(%i4) save_primes:true;
(%o4)                                   true
(%i5) next_prime(34155007172832);
(%o5)                           34155007172837
(%i6) :lisp *large-primes*
NIL
(%i6) next_prime(341550071728321);
(%o6)                           341550071728361
(%i7) :lisp *large-primes*
(341550071728361)
```

prev_prime 函数: 与えられた正整数を越えない素数で最大の素数を返す函数です. 与える正整数は素数でも合成数でも構いません. この函数では素数判定で primep 函数を用いています. そのために 341550071728321 を越える素数を求めた場合, 大域変数 save_primes が true であれば, 内部変数*large-primes* に素数が登録されます.

```
(%i1) save_primes:true;
(%o1)                                   true
(%i2) prev_prime(341550071728321);
(%o2)                           341550071728289
(%i3) :lisp *large-primes*
NIL
(%i3) prev_prime(3415500717283210);
```

```
(%o3)                          3415500717283163
(%i4) :lisp *large-primes*
(3415500717283163)
```

power_mod 函数: power_mod 函数は 3 個の引数を取る函数で,power_mod(a,n,m) は a^n mod m を計算します.

inv_mod 函数: 二つの正整数を引数に取り,inv_mod(m,n) で mod(m,n) の逆元を返します. ここで,m が環 $\mathbb{Z}[n]$ で正則元 (逆元を持たない元) の場合, たとえば, n が m の倍数の場合, あるいは, m が n の倍数の場合に false を返却します.

```
(%i104) makelist(inv_mod(x,5),x,1,4);
(%o104)                        [1, 3, 2, 4]
(%i105) makelist(mod(mod(x,5)*inv_mod(x,5),5),x,1,4);
(%o105)                        [1, 1, 1, 1]
(%i106) makelist(inv_mod(x,4),x,1,4);
(%o106)                    [1, false, 3, false]
```

9.1.18 ifactor パッケージに含まれる大域変数

ifactor パッケージに含まれる大域変数

ifactor_verbose	false	ifactors 函数の動作報告表示を制御
factors_only	false	ifactors 函数の出力形式を制御
save_primes	false	内部変数*large-primes*への素数の登録を制御
primep_number_of_tests	25	Miller-Rabin 素数判定法の適用回
pollard_rho_limit	10000	pollard_rho による判定で利用
pollard_rho_tests	5	pollard_rho による判定で利用
pollard_rho_limit_step	1000	pollard_rho による判定で追加されるステップ数
ecm_number_of_curves	50	ECM による反復回数
ecm_limit	200	ECM による判定条件
ecm_max_limit	51199	ECM による判定条件
ecm_limit_delta	200	ECM で追加されるスッテップ数

大域変数 ifactor_verbose: true であれば,ifactors 函数は結果リストを表示させるだけではなく, より詳細な報告を表示させます. なお, この大域変数のみ ifactors の様に **s** が付かないので注意して下さい.

大域変数 factors_only: true の場合, ifactor 函数は素数の冪のリストとして出力を行います.

大域変数 save_primes: trueであれば,内部変数*large-primes*にprimep函数で判定した341550071728321よりも大きな素数をconsで追加します. この大域変数はMaximaの変数に影響を与えないため, その影響が表から見え難い大域変数です.

大域変数 primep_number_of_tests: primep函数による素数判定で,判定する数が341550071728321よりも大きな場合に用いられる内部函数primep-probでのMiller-Rabin素数判定法を適用する回数を指定します.
因に変数名の先頭にpollardが付く大域変数はPolardの素数判定アルゴリズムで用いられる大域変数です.

大域変数 pollard_rho_limit: ifactor函数が呼出す内部函数get-one-factor-pollard函数で用いられる大域変数です. 内部の判定処理で用いられる大域変数です.

大域変数 pollard_rho_limit_step: get-one-factor-pollard函数内部の反復処理で, 大域変数pollard_rho_limitと組合せて用いられます. この大域変数はpollard_rho_limitで不十分な場合に追加される反復処理のステップ数になります:

```
(%i24) ifactors(913284098373894758374598298931);

Starting factorization of n = 913284098373894758374598298931
Factoring n = 913284098373894758374598298931
Pollard rho: round #1 of 20 (lim=10000)
Pollard rho: round #2 of 20 (lim=11000)
Pollard rho: round #3 of 20 (lim=12000)
Pollard rho: round #4 of 20 (lim=13000)
Pollard rho: round #5 of 20 (lim=14000)
Pollard rho: found factor 11772548497 (11 digits)
========> Prime factor: 11772548497

========> Prime factor: 77577433518887355523

(%o24)          [[11772548497, 1], [77577433518887355523, 1]]
(%i25) pollard_rho_limit:100;
(%o25)                                 100
(%i26) ifactors(913284098373894758374598298931);

Starting factorization of n = 913284098373894758374598298931
Factoring n = 913284098373894758374598298931
Pollard rho: round #1 of 20 (lim=100)
Pollard rho: round #2 of 20 (lim=1100)
Pollard rho: round #3 of 20 (lim=2100)
Pollard rho: round #4 of 20 (lim=3100)
Pollard rho: round #5 of 20 (lim=4100)
Pollard rho: round #6 of 20 (lim=5100)
Pollard rho: round #7 of 20 (lim=6100)
Pollard rho: round #8 of 20 (lim=7100)
Pollard rho: round #9 of 20 (lim=8100)
Pollard rho: round #10 of 20 (lim=9100)
Pollard rho: round #11 of 20 (lim=10100)
```

```
Pollard rho: round #12 of 20 (lim=11100)
Pollard rho: round #13 of 20 (lim=12100)
Pollard rho: round #14 of 20 (lim=13100)
Pollard rho: round #15 of 20 (lim=14100)
Pollard rho: round #16 of 20 (lim=15100)
Pollard rho: round #17 of 20 (lim=16100)
Pollard rho: round #18 of 20 (lim=17100)
Pollard rho: round #19 of 20 (lim=18100)
Pollard rho: round #20 of 20 (lim=19100)
ECM: trying with curve #1 of 50 (lim=200)
ECM: trying with curve #2 of 50 (lim=400)
ECM: trying with curve #3 of 50 (lim=600)
ECM: trying with curve #4 of 50 (lim=800)
ECM: trying with curve #5 of 50 (lim=1000)
ECM: trying with curve #6 of 50 (lim=1200)
ECM: trying with curve #7 of 50 (lim=1400)
ECM: trying with curve #8 of 50 (lim=1600)
ECM: found factor in stage 2: 11772548497 (11 digits)
=========> Prime factor: 11772548497

=========> Prime factor: 77577433518887355523

(%o26)              [[11772548497, 1], [77577433518887355523, 1]]
```

この例からも,pollard_rho_limit を初期値の 10000 から 100 に変更したために, 自動的に上限 (lim) を増やして処理していることが判ります.

大域変数 pollard_rho_tests:　ifactor 函数が呼出す内部函数の get-one-factor 函数で大きな正整数を分解する際に, 内部函数 get-one-factor-pollard 函数による反復処理の上限を定めます.
ちなみに変数名の頭に ecm が付く大域変数は楕円曲線を用いた素数判定法 (Elliptic Curve Method(ECM)) で用いられる大域変数です.

大域変数 ecm_number_of_curves:　内部函数 get-one-factor-ecm 函数の反復処理回数を定める大域変数になります.

大域変数 ecm_limit と大域変数 ecm_max_limit:　ECM による素数の検出を行う上での判定条件を定める大域変数になります. 大域変数 ecm_limit_delta は大域変数 ecm_max_limit と組合せて用いられる大域変数で, 必要に応じて計算の反復処理数を増加させる場合に, 用いられます.

9.1.19　numth パッケージ

numth パッケージに含まれる函数の構文

divsum(⟨ 整数 ⟩)
divsum(⟨ 整数$_1$⟩,⟨ 整数$_2$⟩)
totient(⟨ 整数 ⟩)
jaccobi(⟨ 整数$_1$⟩,⟨ 整数$_2$⟩)
gcfactor(⟨ 式 ⟩)

divsum 函数: 引数が一つの場合, 引数の因子の和を返します. 引数が二つの場合, 第 1 引数の因子を第 2 引数乗したものの和を返します:

```
(%i24) divsum(128);
(%o24)                                255
(%i25) makelist(2^i,i,0,7);
(%o25)                    [1, 2, 4, 8, 16, 32, 64, 128]
(%i26) substpart("+",%,0);
(%o26)                                255
(%i27) divsum(128,4);
(%o27)                             286331153
(%i28) makelist(2^(4*i),i,0,7);
(%o28)      [1, 16, 256, 4096, 65536, 1048576, 16777216, 268435456]
(%i29) substpart("+",%,0);
(%o29)                             286331153
```

この函数は大域変数 intfaclim の影響を受けます.

totient 函数: 与えた正整数以下の整数で, 与えた正整数と互いに素となる整数の数を返します:

```
(%i6) totient(10);
(%o6)                                  4
(%i7) totient(11);
(%o7)                                 10
```

この例では 10 と互いに素な整数は 3, 5, 7, 9 の 4 個, 11 は素数のために 11 を除く 10 個の数と互いに素となります. この函数の内部では factor 函数が用いられており, 大域変数 intfaclim で制御されます.

jacobi 函数: Jacobi 記号を計算する函数です. Jacobi 記号は Legendre の平方剰余記号 $\left(\frac{a}{b}\right)$ を拡張したものです. ここで, Legendre の平方剰余記号は, p を奇素数, a を p と互いに素な整数とします. このとき, $X^2 \equiv a \pmod{p}$ が解を持つ場合に, a は p を法として平方剰余であると呼び, Legendre の平方剰余記号を用いて, $\left(\frac{a}{b}\right) = 1$ と記述します. そして, その様な解が存在しない場合, a は p を法として平方非剰余であると呼んで, $\left(\frac{a}{b}\right) = -1$ と記述します.

Jacobi の記号は, この Legendre の平方剰余記号に対して n を 3 以上の奇数とし, $n = \prod_{i=1}^{k} p_i^{e_i}$ をその素因数分解としたときに, $(a, n) = 1$ となる整数 a に対して,

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_i}\right)_1^e \left(\frac{a}{p_2}\right)_2^e \cdots \left(\frac{a}{p_k}\right)_k^e$$

で $\left(\frac{a}{n}\right)$ を定義したものです.
なお, 任意の整数 a に対し, $\left(\frac{a}{1}\right) = 1$ となります.
ここで Jacobi の記号は次の性質を満すものです:

Jacobi の記号の性質

- $\left(\frac{a}{n}\right) = 0$　　$\gcd(a,n) > 1$
- $\left(\frac{-1}{n}\right) = 1$　　$n = 1 (\mathrm{mod}\ 4)$
- $\left(\frac{-1}{n}\right) = -1$　　$n = 3 (\mathrm{mod}\ 4)$
- $\left(\frac{a}{n}\right)\left(\frac{b}{n}\right) = \left(\frac{ab}{n}\right)$
- $\left(\frac{a}{m}\right)\left(\frac{a}{n}\right) = \left(\frac{a}{mn}\right)$
- $\left(\frac{a}{n}\right) = \left(\frac{b}{n}\right)$　　$a = b (\mathrm{mod}\ n)$

gfactor 函数:　gcfactor 函数は Gauß整数上で多項式の因子分解を行います.

```
(%i23) gfactor(x^7+1);
                                  6     5     4     3     2
(%o23)                  (x + 1) (x  - x  + x  - x  + x  - x + 1)
(%i24) gfactor(x^6+1);
                                      2                   2
(%o24)            (x - %i) (x + %i) (x  - %i x - 1) (x  + %i x - 1)
```

なお, gcfactor 函数内部では jacobi 函数が用いられています.

9.1.20　Kronecker の δ と Stiring 数

nset パッケージに含まれる数論関連の函数の構文

kron(〈式$_1$〉,〈式$_2$〉)
stirling1(〈正整数値$_1$〉,〈正整数値$_2$〉)
stirling2(〈正整数値$_1$〉,〈正整数値$_2$〉)

kron_delta:　引数を二つ取る整数値函数です. 基本的に Kronecker の δ を表現する函数です. したがって, 二つの引数が等しい場合には 1 を返し, そうでない場合には 0 を返します. ただし, 等しいかどうか判別出来ない場合には名詞型で返します:

```
(%i112) kron_delta(1,0);
(%o112)                                0
(%i113) kron_delta("a","a");
(%o113)                                1
(%i114) kron_delta(x^2+2*x+1,(x+1)^2);
(%o114)                                1
(%i115) kron_delta(integrate(x*2,x),x^2);
(%o115)                                1
(%i116) kron_delta(x,y);
(%o116)                      kron_delta(x, y)
(%i117) kron_delta(x,y),x=y;
(%o117)                                1
```

stirling1 函数と stirling2 函数: 第一種と第二種の Stirling 数を返す函数です. なお, Maxima では, これらの Stirling 数には次の同値関係が設定してあります:

stirling1 函数と stirling2 函数の同値関係

(1_a)	stirling1 (0, n)	~	kron_delta(0,n)
(2_a)	stirling1 (n, n)	~	1
(3_a)	stirling1 (n, n - 1)	~	binomial(n,2)
(4_a)	stirling1 (n + 1, 0)	~	0
(5_a)	stirling1 (n + 1, 1)	~	$n!$
(6_a)	stirling1 (n + 1, 2)	~	$2^n - 1$
(1_b)	stirling2 (0, n)	~	kron_delta(0,n)
(2_b)	stirling2 (n, n)	~	1
(3_b)	stirling2 (n, n - 1)	~	binomial(n,2)
(4_b)	stirling2 (n + 1, 0)	~	0
(5_b)	stirling2 (n + 1, 1)	~	1
(6_b)	stirling2 (n + 1, 2)	~	$2^n - 1$
(7_b)	stirling2 (n,0)	~	kron_delta(n,0)
(8_b)	stirling2 (n,m)	~	0 ($m > n$ の場合)

9.2　三角函数

9.2.1　三角函数一覧

Maxima は沢山の三角函数を持っています. 三角函数の恒等式, すなわち, $\cos^2(x) + \sin^2(x) = 1$ や $\cos(2x) = 2\cos^2(x) - 1$ のような恒等式は予め Maxima に組込まれていますが, 多くの恒等式を規則として利用者が付加することができます.
Maxima で予め定義された三角函数は下記のものがあります:

三角函数と双曲函数

函数名	概要	属性
cos	余弦函数	deftaylor, rule, noun, gradef, transfun
cosh	双曲線余弦函数	deftaylor, database info, kind(cosh, posfun), rule, noun, gradef, transfun
cot	余接函数	deftaylor, database info, kind(sinh, increasing), kind(sinh, oddfun), rule, noun, gradef, transfun
coth	双曲線余接函数	deftaylor, rule, noun, gradef, transfun
csc	余割函数	deftaylor, rule, noun, gradef, transfun
csch	双曲線余割函数	deftaylor, database info, kind(csch, oddfun), rule, noun, gradef, transfun
sec	正割函数	deftaylor, rule, noun, gradef, transfun, transfun
sech	双曲線正割函数	deftaylor, database info, kind(sech, posfun), rule, noun, gradef ,transfun
sin	正弦函数	deftaylor, rule, noun, gradef, transfun
sinh	双曲線正弦函数	deftaylor, rule, noun, gradef, transfun
tan	正接函数	deftaylor, rule, noun, gradef, transfun
tanh	双曲線正接函数	deftaylor, database info, kind(tanh, increasing), kind(tanh, oddfun), rule, noun, gradef, transfun

逆三角函数と逆双曲函数

函数名	概要	属性
acos	逆余弦函数	rule, noun, gradef, transfun
acosh	逆双曲線余弦函数	rule, noun, gradef, transfun
acot	逆余接函数	rule, noun, gradef, transfun
acoth	逆双曲線余接函数	rule, noun, gradef, transfun
acsc	逆余割函数	rule, noun, gradef, transfun
acsch	逆双曲線余割函数	rule, noun, gradef, transfun
asec	逆正割函数	rule, noun, gradef, transfun
asech	逆双曲線正割函数	rule, noun, gradef, transfun
asin	逆正弦函数	deftaylor, rule, noun, gradef, transfun
asinh	逆双曲線正弦函数	rule, noun, gradef, transfun
atan	逆正接函数	deftaylor, database info, kind(atan, increasing), kind(atan, oddfun), rule, noun, gradef, transfun
atan2	逆正接函数	rule, gradef
atanh	逆双曲線正接函数	rule, noun, gradef, transfun

なお, 逆正接函数には atan 函数と atan2 函数の二種類があります. ここで atan2 函数は atan2(y,x) のように引数を二つ必要とする函数で, 区間 $(-\pi, \pi)$ の間で atan(y/x) を計算します.
三角函数は倍角公式や角の和の公式等のいろいろな公式を持っています. これらを自動的に入力した式に適応することも可能です. この場合は大域変数 trigexpand を true に, 角の整数倍に関しては大域変数 trigexpandtimes, 角の和については大域変数 trigexpandplus をそれぞれ true に設定すると公式を用いて与式の自動展開を行います:

```
(%i43)  x+sin(5*x)/cos(x),trigexpand=true,expand;
                   5
              sin (x)                  3           3
(%o43)        ------- - 10 cos(x) sin (x) + 5 cos (x) sin(x) + x
              cos(x)
(%i44) trigexpand(cos(3*x+2*y));
(%o44)                cos(3 x) cos(2 y) - sin(3 x) sin(2 y)
(%i45) trigexpand:true;
(%o45)                                  true
(%i46) trigexpandtimes:true;
(%o46)                                  true
(%i47) trigexpandplus:true;
(%o47)                                  true
(%i48) cos(3*x+2*y);
          3                   2          2          2
(%o48) (cos (x) - 3 cos(x) sin (x)) (cos (y) - sin (y))
                                 2                3
                    - 2 (3 cos (x) sin(x) - sin (x)) cos(y) sin(y)
```

この例で最初の x+sin(5*x)/cos(x),trigexpand=true,expand は ev 函数による評価の書式の一つで, 与式 $x + \dfrac{\sin(5x)}{\cos(x)}$ を大域変数 trigexpand を true にした状態で展開を行うことを意味します. この表

記は Maxima のトップレベルだけで利用可能です. この ev 函数の詳細は §5.8.3 を参照して下さい.
三角函数の半角公式は大域変数 halfangles を true にすることで自動展開させることができます. この大域変数は大域変数 trigexpand の影響は受けません.
これらの変数とば別に, 三角函数の引数に含まれる対象への declare 函数による属性の付与で自動簡易化も行えます:

```
(%i2) declare(i,integer,a,even,b,odd);
(%o2)                                done
(%i3) sin(x+(a+1/2)*%pi);
(%o3)                               cos(x)
(%i4) sin(x+(b+1/2)*%pi);
(%o4)                              - cos(x)
(%i5) cos(x+b*2*i*%pi);
(%o5)                               cos(x)
```

この例では最初に対象 i を整数, 対象 a を偶数, 対象 b を奇数として属性を付与しています. それによって, Maxima は三角函数の項を属性による評価を自動的に行い, 簡易化された式を得ています.

三角函数に関連する大域変数

変数名	既定値& 概要	
%piargs	true	%pi と有理数の積を引数とする三角函数の簡易化を制御
%iargs	true	%i と有理数の積を引数とする三角函数の簡易化を制御
halfangles	false	半角公式の自動適用を制御
trigexpandplus	true	和公式の自動適用を制御
trigexpandtimes	true	角度の積による展開を制御
triginverses	all	逆関数との合成による簡易化を制御
trigsign	true	負の引数の簡易化を制御

大域変数%piargs: 'true' であれば sin(%pi/2) のような式を実数に変換します. 大域変数%iargs が true の場合に sin(%i*x) のような純虚数を引数に持つ三角函数を双曲函数に変換します.

大域変数 halfangles: 'true' であれば半角 $\frac{\theta}{2}$ に対して簡易化が実行されます. この変数は大域変数 trigexpand の影響は受けません.

大域変数 trigexpandplus: 和の規則を制御する大域変数です. つまり, 大域変数 trigexpand に true が設定されているときに引数の和を含む $\sin(x+y)$ のような三角函数の自動展開が大域変数 trigexpandplus が true の場合に限って実行されます.

大域変数 trigexpandtimes: trigexpand 函数の積規則を制御する大域変数です. 大域変数 trigexpand が true に設定されているときに三角函数の引数が整数, あるいは有理数倍の式に対して三角函数項の自動展開が実行されます.

大域変数 triginverses: 三角函数, 双曲函数とその逆函数との合成の簡易化を制御します:

- all
  両方, 例えば,atan (tan (*x*)) と tan (atan (*x*)) の両方が *x* に簡易化されます.
- true
  arcfunction(function(x)) の簡易化が切り捨てられます.
- false
  arcfunc(func)) と fun(arcfun(x)) の簡易化が切り捨てられます.

大域変数 trigsign: 'true' であれば三角函数に対して負の引数の自動簡易化を行います. たとえば, 大域変数 trigsin が true のときに限って sin(-x) を-sin(x) に変換します.

9.2.2 三角函数に関連する函数

三角函数の展開と簡易化に関連する函数

trigexpand(⟨ 式 ⟩)
trigreduce(⟨ 式 ⟩,⟨ 変数 ⟩)
trigsimp(⟨ 式 ⟩)
trigrat(⟨ 三角函数を含む式 ⟩)

trigexpand 函数: ⟨ 式 ⟩ に含まれる三角函数や双曲函数に対して倍角公式等を適用することで式の展開を実行します. 最良の結果を得るために予め expand 函数等で ⟨ 式 ⟩ を展開しておくと良い結果が得られることがあります. 簡易化の利用者制御を拡張するため, この函数は一度に一つのレベルのみの角の和と角の積の展開を行います:

```
(%i41) trigexpand((cos(2*x+%pi*4/3*y+%pi/2)+sin(2*y+x))/(cos(x)+sin(y)));
                          4 %pi y                    4 %pi y
(%o41) (- cos(2 x) sin(-------) - sin(2 x) cos(-------) + cos(x) sin(2 y)
                            3                          3
                                                + sin(x) cos(2 y))/(sin(y) + cos(x))
```

なお, 大域変数 trigexpand を true に設定することで trigexpand 函数による式の展開と同じ結果が得られます:

```
(%i44) trigexpand:true$
(%i45) (cos(2*x+%pi*4/3*y+%pi/2)+sin(2*y+x))/(cos(x)+sin(y));
              2          2          4 %pi y                              4 %pi y
(%o45) (- (cos (x) - sin (x)) sin(-------) - 2 cos(x) sin(x) cos(-------)
                                       3                                    3
```

```
                      2          2
      + sin(x) (cos (y) - sin (y)) + 2 cos(x) cos(y) sin(y))/(sin(y) + cos(x))
```

trigreduce 函数: 〈 変数 〉の積を持つ三角函数と双曲正弦函数と余弦函数の積と冪乗を結合します. 分母で現われたこれらの函数を消去することも試みます. なお, 〈 変数 〉が省略されると 〈 式 〉の全ての変数が利用されます:

```
(%i1)  trigreduce(-sin(x)^2+3*cos(x)^2+x);
                       cos(2 x)       cos(2 x)   1         1
(%o1)                  -------- + 3 (-------- + -) + x - -
                          2              2       2         2
```

trigsimp 函数: trigsimp 函数は share/trigonometry/trgsmp.mac で定義された Maxima 言語で記述された函数です. この trgsmp パッケージの詳細は §9.2.4 で解説します.
この trigsimp 函数は trigreduce 函数と組合せてることでよりよい簡易化が得られます:

```
(%i3)  trigreduce(-sin(x)^2+3*cos(x)^2+x);
                       cos(2 x)       cos(2 x)   1         1
(%o3)                  -------- + 3 (-------- + -) + x - -
                          2              2       2         2
(%i4) trigsimp(-sin(x)^2+3*cos(x)^2+x);
                                    2
(%o4)                        4 cos (x) + x - 1
(%i5) trigsimp(trigreduce(-sin(x)^2+3*cos(x)^2+x));
(%o5)                        2 cos(2 x) + x + 1
```

この例では trigreduce 函数のみでは式が纏め切れていません. さらに, trigsimp 函数のみでは cos の冪が残りますが, 両者を合せることでより簡単な式で置換えられていますね.

trigrat 函数: この函数は LISP で記述された trigrat パッケージに含まれる函数で, sin,cos,tan 等の三角函数による有理式の正規簡易化を与えます. 与式に含まれるこれらの三角函数の引数は変数, 有理数と%pi の線形結合です. trigrat 函数の計算結果は簡易化された正弦函数と余弦函数を含む有理式になります:

```
(%i40)  trigrat(sin(2*a)/sin(3*a+%pi/12));
                  24 a + %pi                          24 a + %pi          24 a - %pi
(%o40) - (%i (sin(----------) + (sqrt(3) + 2) cos(----------) + sin(----------)
                     12                                  12                  12
                   24 a - %pi                          24 a + %pi
 + (- sqrt(3) - 2) cos(----------)) + (- sqrt(3) - 2) sin(----------)
                      12                                  12
       24 a + %pi                          24 a - %pi          24 a - %pi
 + cos(----------) + (- sqrt(3) - 2) sin(----------) - cos(----------))
          12                                  12                  12
/((2 sqrt(3) + 4) sin(3 a) + 2 cos(3 a))
```

9.2.3 atrig1 パッケージ

atrig1 パッケージには逆三角函数に対する幾つかの追加の簡易化規則が含まれています. Maxima で既知の規則と共に次の角が実装されています.

0, %pi/6, %pi/4, %pi/3,%pi/2

他の 3 つの象限に於ける角度でも利用可能です. なお, このパッケージを利用するためには予め load(atrig1); を実行しておく必要があります.

9.2.4 trgsmp パッケージ

trigsimp 函数を定義するパッケージで, この函数のために三角函数と双曲函数に属性と規則の定義を行っています.
まず, 属性値は函数 cos, sin, cosh と sinh に対して put 函数を用いて付与します. このときに付与される属性は complement_function, unitconf, complement_conf と type の四種類です. これらの属性は, 属性を付与した函数の二乗の変形で用いるもので, ここで, `trig` を函数 cos, sin, cosh, sinh の何れかとすれば, $\texttt{trig}^2$ を `get(trig,'unitconf)+get(trig,'complement_cof)get(trig,'complement_function)`^{2} で置換えるものです. さらに, 三角函数と双曲函数を函数 cos, sin, cosh と sinh の式として変換するために, 次の三角函数と双曲函数の規則を設定します:

trsmp.mac で定義される三角函数向けの規則

trigrule1	$\tan a \to \frac{\sin a}{\cos a}$
trigrule2	$\sec a \to \frac{1}{\cos a}$
trigrule3	$\csc a \to \frac{1}{\sin a}$
trigrule4	$\cot a \to \frac{\cos a}{\sin a}$

trsmp.mac で定義される双曲函数向けの規則

htrigrule1	$\tanh a \to \frac{\sinh a}{\cosh a}$
htrigrule2	$\mathrm{sech} a \to \frac{1}{\cosh a}$
htrigrule3	$\mathrm{csch} a \to \frac{1}{\sinh a}$
htrigrule4	$\coth a \to \frac{\cosh a}{\sinh a}$

これらの規則は trigsimp 内部で与式に対して apply1 函数を用いて適用し, 次に ratsimp 函数を用いて CRE 表現で簡易化します. その結果, 与式の三角函数と双曲函数は函数 cos, sin, cosh, sinh のみで構成される有理数係数の多項式となります. それから, この式を trigsimp3 に引渡し, 内部で呼出される improve 函数を用いて, これらの函数の冪を処理するのです.
ここで trigsimp3 と improve 函数の動作を trace 函数を用いて追跡してみましょう:

```
(%i3) trace(trigsimp3,improve);
(%o3)                          [trigsimp3, improve]
(%i4) trigsimp(cos(x)^2+sin(x)^2);
```

```
                        2          2
1 Enter trigsimp3 [sin (x) + cos (x)]
                       2          2         2          2
 1 Enter improve [sin (x) + cos (x), sin (x) + cos (x), [[sin(x), cos(x)]]]
                       2          2         2          2
  2 Enter improve [sin (x) + cos (x), sin (x) + cos (x), []]
                      2          2
  2 Exit  improve sin (x) + cos (x)
                      2          2
  2 Enter improve [sin (x) + cos (x), 1, []]
  2 Exit  improve 1
  2 Enter improve [1, 1, []]
  2 Exit  improve 1
 1 Exit  improve 1
1 Exit  trigsimp3 1
(%o4)                                  1
```

このように, trigsimp 函数では三角函数と双曲函数を cos, sin と cosh, sinh の項で構成された有理式に変換し, 次に, これらの項の冪を属性を用いて処理を行っている様子が分ります.
なお, trigsimp 函数を実行することによって, trigrule1 から htrigrule4 までの 8 個の規則が大域変数 rules に登録されます.

9.3 指数函数と対数函数

9.3.1 指数函数と対数函数の概要

Maxima には指数函数 exp と対数函数 log と plog 函数があります.

指数函数と対数函数

函数	属性
exp(⟨ 式 ⟩)	rule, transfu
log(⟨ 式 ⟩)	database info, kind(log, increasing), noun, rule, gradef, transfun
plog(⟨ 式 ⟩)	noun, rule, gradef, transfun

exp 函数: 指数函数を表現する函数で, Maxima 内部で Napia 数%e の冪として表現されています.

log 函数と plog 函数: log 函数と plog 函数 Napia 数%e を底とする自然対数です. ここで log 函数と plog 函数は違いが判り難い函数です. 内部書式も異っていますが, plog 函数は log 函数の機能を内包する函数で, 引数が複素数の場合の処理で大きく異なります:

```
(%i24) log(%i);
(%o24)                                log(%i)
(%i25) plog(%i);
                                      %i %pi
(%o25)                                ------
                                        2
(%i26) log(-(abs(x+1))^2);
                                                2
(%o26)                              log(- (x + 1) )
(%i27) plog(-(abs(x+1))^2);
Is  x + 1  zero or nonzero?

pos;
(%o27)                          2 log(x + 1) + %i %pi
```

これは処理で用いる内部函数の特性の違いによるもので, log 函数は simpln 函数, plog 函数は simpplog 函数と simpln 函数の双方を用います. ここで log 函数で入力される一般の式は正であると仮定した面があり, それに対して plog 函数では正値性を asksign 函数を用いて確認します:

```
(%i38) log((x+1)^2);
1. Trace: (SIMPLN '((%LOG) ((MEXPT SIMP) ((MPLUS SIMP) 1 $X) 2)) '1 'NIL)
2. Trace: (SIMPLN '((%LOG) ((MPLUS SIMP) 1 $X)) '1 'T)
2. Trace: SIMPLN ==> ((%LOG SIMP) ((MPLUS SIMP) 1 $X))
1. Trace: SIMPLN ==> ((MTIMES SIMP) 2 ((%LOG SIMP) ((MPLUS SIMP) 1 $X)))
(%o38)                                2 log(x + 1)
(%i39) plog((x+1)^2);
1. Trace: (SIMPPLOG '((%PLOG) ((MEXPT SIMP) ((MPLUS SIMP) 1 $X) 2)) '1 'NIL)
Is  x + 1  zero or nonzero?

pos;
2. Trace: (SIMPLN '((%LOG) ((MEXPT SIMP) ((MPLUS SIMP) 1 $X) 2)) '1 'T)
```

```
3. Trace: (SIMPLN '((%LOG) ((MPLUS SIMP) 1 $X)) '1 'T)
3. Trace: SIMPLN ==> ((%LOG SIMP) ((MPLUS SIMP) 1 $X))
2. Trace: SIMPLN ==> ((MTIMES SIMP) 2 ((%LOG SIMP) ((MPLUS SIMP) 1 $X)))
1. Trace: SIMPPLOG ==> ((MTIMES SIMP) 2 ((%LOG SIMP) ((MPLUS SIMP) 1 $X)))
(%o39)                                  2 log(x + 1)
```

この例では内部函数 simpln と内部函数 simpplog に LISP の trace 函数を作用させて処理の違いを見たものです. plog 函数では与式に純虚数の積が含まれている場合は単純に%i*%pi/2 を add2*函数を用いて加え, 与式のノルムに対して log 函数を作用させる函数と言えます:

```
 (%i17) plog(%i*x^2);
1. Trace: (SIMPPLOG '((%PLOG) ((MTIMES SIMP) $%I ((MEXPT SIMP) $X 2))) '1 'NIL)
Is  x  zero or nonzero?

pos;
2. Trace: (ADD2* '((MTIMES SIMP) 2 ((%LOG SIMP) $X)) '((MTIMES) 1 ((RAT) 1 2) $%
        I $%PI))
2. Trace: ADD2* ==>
((MPLUS SIMP) ((MTIMES SIMP) ((RAT SIMP) 1 2) $%I $%PI)
 ((MTIMES SIMP) 2 ((%LOG SIMP) $X)))
1. Trace: SIMPPLOG ==>
((MPLUS SIMP) ((MTIMES SIMP) ((RAT SIMP) 1 2) $%I $%PI)
 ((MTIMES SIMP) 2 ((%LOG SIMP) $X)))
                                                 %i %pi
(%o17)                                  2 log(x) + ------
                                                    2
(%i18) plog(-%i*x^2);
1. Trace: (SIMPPLOG '((%PLOG) ((MTIMES SIMP) -1 $%I ((MEXPT SIMP) $X 2))) '1 '
        NIL)
Is  x  zero or nonzero?

pos;
2. Trace: (ADD2* '((MTIMES SIMP) 2 ((%LOG SIMP) $X)) '((MTIMES) -1 ((RAT) 1 2) $
        %I $%PI))
2. Trace: ADD2* ==>
((MPLUS SIMP) ((MTIMES SIMP) ((RAT SIMP) -1 2) $%I $%PI)
 ((MTIMES SIMP) 2 ((%LOG SIMP) $X)))
1. Trace: SIMPPLOG ==>
((MPLUS SIMP) ((MTIMES SIMP) ((RAT SIMP) -1 2) $%I $%PI)
 ((MTIMES SIMP) 2 ((%LOG SIMP) $X)))
                                                 %i %pi
(%o18)                                  2 log(x) - ------
                                                    2
```

この例で示すように plog 函数は $x+1$ の正値性を確認していますが, ここで neg; を入力しても結果は同じ $2\log(x+1)$ になります. ただし, zero; を入力すると, plog(0) is undefined と返されます. このように 0 かどうかをチェックする機能と考えた方が良いでしょう.

exp 函数, log 函数と plog 函数は次の大域変数の設定によって Maxima 上で自動的に簡易化が実行されます.

9.3.2 対数函数に関連する函数

逆三角函数や逆双曲函数を対数函数に変換する函数

logarc(⟨ 式 ⟩)

函数 logarc: 同名の大域変数 logarc の設定とは無関係に逆三角函数 (acos, asin, atan, atan2, asec, acsc, acot) と逆三角函数 (asinh, acosh, atanh, asech, acsch, acoth) を対数函数による有理式に変換して ev 函数を用いた式の再評価を行う函数です:

```
(%i14) logarc(atan(x));
                          %i (log(%i x + 1) - log(1 - %i x))
(%o14)                  - ----------------------------------
                                        2
(%i15) logarc:true;
(%o15)                                true
(%i16) atan(x);
                          %i (log(%i x + 1) - log(1 - %i x))
(%o16)                  - ----------------------------------
                                        2
```

対数函数を潰す函数

logcontract(⟨ 式 ⟩)

logcontract 函数: log 函数を含む式を簡易化で潰す函数です. すなわち, log 函数の係数を引数に取込み, その結果によって log から sqrt 等の函数に変換します. 具体的には ⟨ 式 ⟩ を再帰的に調べ, a1*log(b1)+a2*log(b2)+c の形の部分式を log(ratsimp(b1^a1 * b2^a2))+ c に変換します:

```
(%i8) 2*log(x)+4*log(y)+8;
(%o8)                      4 log(y) + 2 log(x) + 8
(%i9) logcontract(%);
                                    2  4
(%o9)                          log(x  y ) + 8
(%i10) declare(n,integer);
(%o10)                              done
(%i11) logcontract(2*log(x)+4*n*log(y)+8);
                                   2  4 n
(%o11)                         log(x  y  ) + 8

(%i12) logcontract(2*log(x)+4*m*log(y)+8);
                                  4          2
(%o12)                        m log(y ) + log(x ) + 8
```

この logcontract 函数は log 函数の整数係数に対して影響を与える函数です. たとえば, declare(n,integer); を実行して logcontract(2*log(x)+4*n*log(y)+8); を実行すると, 第二式の 4+n+log(y) の係数 4*n は featurep(coeff,integer) を満すために $\log\left(x^2y^{4n}\right)$+8 に簡易化されています. ところが, logcontract(2*log(x)+4*m*log(y)+8); の場合は変数 m は integer として未宣言のために簡易化が途中で停止しています.
なお, この logcontract 函数は大域変数 superlogcon の影響を受けます. この大域変数 superlogcon を初期値の true から 'false' に変更することで内部函数 lgcsqrt の処理を外す結果になります. その

ために与式の主演算子が和の場合と積の場合で処理する以上のことが出来なくなります:

```
(%i54) superlogcon:true$
(%i55) neko: 2*(a*log(x) + 2*a*log(y))$
(%i56) :lisp (trace lgcsort)
WARNING: TRACE: redefining function LGCSORT in top-level, was defined in
         /usr/local/maxima-5.14.0/src/binary-clisp/comm2.fas
;; Tracing function LGCSORT.
(LGCSORT)
(%i56) logcontract(neko);
1. Trace:
(LGCSORT
 '((MTIMES SIMP) 2
   ((MPLUS SIMP) ((MTIMES SIMP) $A ((%LOG SIMP) $X))
    ((MTIMES SIMP) 2 $A ((%LOG SIMP) $Y)))))
1. Trace: LGCSORT ==>
((MTIMES SIMP) $A
 ((MPLUS SIMP) ((MTIMES SIMP RATSIMP) 2 ((%LOG SIMP) $X))
  ((MTIMES SIMP RATSIMP) 4 ((%LOG SIMP) $Y))))
                                          2  4
(%o56)                              a log(x  y )
(%i57) superlogcon:false$
(%i58) logcontract(neko);
                                        2
(%o58)                        2 (a log(y ) + a log(x))
(%i59) logcontract(expand(neko));
                                        4           2
(%o59)                          a log(y ) + a log(x )
```

ただし, この例のように式を展開して与えれば, log 函数毎に式を纏める作用がるので, このような出力が必要であれば大域変数 superlogcon を 'false' にします.

極座標形式に変換する函数

polarform(〈式〉)

polarform 函数:　与えられた〈式〉を r*%e^(%i*theta) の形に変換します. なお, 多項式が実数係数多項式の場合は入力のままで返され, 函数を含む場合には, その函数が負であれば%e^(%i*%pi) をかけた式が返されます:

```
(%i31) polarform((1+%i)^3);
                                          3 %i %pi
                                          --------
                                             4
(%o31)                          2 sqrt(2) %e
(%i32) polarform(x^2+1);
    ppppp                                   2
(%o32)                                 x  + 1
(%i33) polarform((x+1)^2);
Is  x + 1  zero or nonzero?

pos;
                                     2
(%o33)                              x  + 2 x + 1
```

```
(%i34) polarform(sin(x+1));
Is  sin(x + 1)  positive or negative?

neg;
                                        %i %pi
(%o34)                          - %e          sin(x + 1)
```

対数函数に関連する大域変数

変数名	既定値	概要
%e_to_numlog	false	指数に対数を持つ冪乗の簡易化
logabs	false	log を含む不定積分の結果を制御
logarc	false	逆三角函数や逆双曲函数を対数関数で表現
logconcoeffp	false	logcontract で潰される係数を制御
logexpand	true	対数関数の積や冪の自動変換
lognegint	false	対数関数の引数が負の場合の処理を制御
lognumer	false	対数函数の浮動小数点引数の制御
logsimp	true	log を含む指数関数の冪乗の自動化を制御
superlogcon	true	logcontract 函数による簡易化を制御

大域変数%e_to_numlog: 'true' としたときに与式 $e^{r\log x}$ の r が numberp 函数や内部函数 maxima-integerp に対して true を返す場合に与式は x^r に簡易化されます. なお, radcan 函数もこの変換を行います:

```
(%i1) %e^(a1*log(b1));
                                      a1  log(b1)
(%o1)                                %e
(%i2) %e_to_numlog:true;
(%o2)                                  true
(%i3) declare(a1,integer);
(%o3)                                  done
(%i4) %e^(a1*log(b1));
                                        a1
(%o4)                                  b1
```

この例では変数 a1 に属性として 'integer' を指定したために, numberp 函数は 'false' になっても maxima-integerp 函数で 'true' になるので自動変換が実行されています.

大域変数 logabs: 'true' であれば integrate(1/x,x) のように計算結果に log 函数が結果に含まれる場合に log 函数が log abs(···) で置換されます. ただし, 大域変数 logabs が 'false' であれば log(...) の項を持つものになります. なお, 定積分では 'logabs:true' として対数函数が処理されます. これは不定積分の両端点での評価が必要となることが多いためです.

大域変数 logarc: 'true' であれば自動的に逆三角函数, 逆双曲函数を対数函数の書式に変換します.

大域変数 logconcoeffp: logcontract 函数による式の変形で, 式に含まれた log 函数を含む項の係数に対し, この大域変数で指定した真理函数が真となる場合にその係数を log 函数の内部に取

り込む操作を行います．ここで大域変数 logconcoeffp に指定する真理函数は名詞型, すなわち, logconcoeffp:'numberp; のように真理函数名を名詞型にして割当てます. ここで指定する真理函数は一つの引数のみを持つ論理函数でなければなりません:

```
(%i9) logconcoeffp:'numberp;
(%o9)                                numberp
(%i10) logcontract(1/2*log(x));
(%o10)                            log(sqrt(x))
(%i11) logcontract(5/2*log(x));
                                       5/2
(%o11)                               log(x   )
```

この例では最初の logcontract(1/2*log(x)) で logcontract 函数は大域変数 logconcoeffp に設定された真理函数 numberp に log 函数が含まれている項の係数 $\frac{1}{2}$ を引渡します. ここで真理函数の返却値が真理函数の結果が true であれば, この係数を log 函数の中に取込みますが, 'l/2' に対して numberp 函数が true を返すために 'log(sqrt(x))' を返します. なお, 真理函数の返却値が 'false' の場合, 与式をそのままを返します.

大域変数 logexpand:　'true' であれば自動的に $\log(a^b)$ を $b\log(a)$ に変換します．ここで, all の場合は自動的に $\log(ab)$ は $\log(a) + \log(b)$ に変換されます．super の場合では $a = 1$ でない有理数 a/b に対して $\log(a/b)$ を $\log(a) - \log(b)$ に変換します. なお, 整数 b に対して $\log(1/b)$ は大域変数 logexpand とは無関係に簡易化されます. 'false' の場合はこれらの簡易化は全て実行されません.

大域変数 lognegint:　'true ' であれば正整数 n に対し, $\log(-n)$ を $log(n) + i\pi$ で置換える規則が内部的に設定されます.

大域変数 lognumer:　'true' であれば負の浮動小数引数は log 函数に渡される前に常にその絶対値に変換されます.

大域変数 logsimp:　'false' であれば log 函数を含む%e の冪乗の自動簡易化は実行されません.

大域変数 superlogcon:　'false' であれば, logcontract 函数内部で内部函数 lgcsort が省略され, 式中の log 函数を纏めるのではなく log 函数項毎の簡易化になります.

9.4 超幾何微分方程式

この節では

$$\frac{d^2u}{dz^2} + p(z)\frac{du}{dz} + q(z)u = 0$$

の形式の線形常微分方程式に関連する項目について解説します. ここで函数 $p(z)$ と $q(z)$ は複素平面上の領域 S に於て有限個の極を除いて一価の正則函数とします. ここで函数 $f(z)$ が正則であるとは, $\frac{df(z)}{dz} = 0$ となることです.
さらに, この点 c が $p(z)$ の高々一位の極, $q(z)$ の高々二位の極とします. ちなみに, このような極 c のことを確定特異点と呼びます. このときに函数 $p(z)$ と函数 $q(z)$ は正則函数 $P(z)$ と $Q(z)$ を用いて

$$p(z) = \frac{P(z)}{z-c}$$
$$q(z) = \frac{Q(z)}{(z-c)^2}$$

と表現出来ます. これによって次の微分方程式が得られます:

$$(z-c)^2\frac{d^2u}{dz^2} + (z-c)P(z)\frac{du}{dz} + Q(z)u = 0$$

ここで函数 $P(z)$ と $Q(z)$ は点 c 近傍で正則函数となるので級数展開が可能で,

$$P(z) = P_0 + P_1(z-c) + P_2(z-c)^2 + \cdots$$
$$Q(z) = Q_0 + Q_1(z-c) + Q_2(z-c)^2 + \cdots$$

とすることができます. そこで, 微分方程式の解を次の形式的な冪級数

$$u(z) = (z-c)^r\left[a_0 + \sum_{n=1}^{\infty} a_n(z-c)^n\right]$$

としましょう. そして, この $u(z)$ を微分方程式に代入することで, 確定特異点における決定方程式, あるいは特性方程式と呼ばれる方程式 $F(r) \stackrel{def}{=} r(r-1)p_0r + q_0 = 0$ が得られます. そして, この方程式の二つの根 ρ_1, ρ_2 を特性根と呼びます. そして, この特性根に対応する形式的解の係数 $a_0, a_1, \cdots$ の値が

$$a_1 = -\frac{1}{F(\rho+1)}\left[(\rho p_1 + q_1)\right]$$
$$a_n = -\frac{1}{F(\rho+1)}\left[\sum_{i=1}^{n} a_{n-i}\left((\rho+n-i)p_m + q_m\right)\right]$$

で得られ, このことから $\mathrm{Re}\rho \leq \mathrm{Re}\rho'$, $\rho - \rho' \notin \mathbb{N}$ の場合, 点 c の近傍で次の二つの基本解を持つことが知られています:

$$u_1(x) = (z-c)^{\rho}\left[1 + \sum_{n=1}^{\infty} a_n(z-c)^n\right]$$

$$u_2(x) = (z-c)^{\rho'}\left[1 + \sum_{n=1}^{\infty} a' b_n (z-c)^n\right]$$

ここで函数 $p(z)$ と $q(z)$ が複素平面上で有限個の極を除いて一価の正則函数となり, 無限遠点 ∞ を含めて全ての特異点が確定特異点となる場合に Fuchs 型の微分方程式と呼びます. なお, 微分方程式が Fuchs 型であり, $a_1, a_2, \cdots, a_n$ と ∞ を函数 $p(z)$ と函数 $q(z)$ の特異点とする場合に次の性質を持ちます:

$$p(z) = \sum_{i=1}^{n} \frac{A_i}{z-a_i}$$
$$q(z) = \sum_{i=1}^{n}\left(\frac{B_i}{(z-a_i)^2} + \frac{C_i}{z-a_i}\right) \quad \text{ただし,} \sum_{i=i}^{n} C_i = 0$$

Riemann の $\wp$ 函数

$$\frac{d^2u}{dz^2} + \left[\frac{1-\alpha_1-\alpha_2}{z-a_1} + \frac{1-\beta_1-\beta_2}{z-a_2} + \frac{1-\gamma_1-\gamma_2}{z-a_3}\right]\frac{du}{dz} +$$
$$\left[\frac{\alpha_1\alpha_2(a_1-a_2)(a_1-a_3)}{z-a_1} + \frac{\beta_1\beta_2(a_2-a_1)(a_2-a_3)}{z-a_2} + \frac{\gamma_1\gamma_2(a_3-a_1)(a_3-a_2)}{z-a_3}\right]$$
$$\frac{u}{(z-a_1)(z-a_2)(z-a_3)} = 0$$

の一般解を Riemann の $\wp$ 函数 (ペー函数) と呼び,

$$\wp\left\{\begin{array}{ccc} a_1 & a_2 & a_3 \\ \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1 & \beta_2 & \beta_3 \end{array}\right\}$$

と表記します.

Gauss の超幾何微分方程式

$$z(z-1)\frac{d^2u}{dz} + \{\gamma + (1+\alpha+\beta)z\}\frac{du}{dz} + \alpha\beta u = 0$$

9.4.1 Airy 函数

Airy 函数に関連する函数

airy_ai(⟨ 変数 ⟩)
airy_bi(⟨ 変数 ⟩)
airy_dai(⟨ 変数 ⟩)
airy_dbi(⟨ 変数 ⟩)

airy_ai 函数と airy_bi 函数は, 夫々Airy の Ai 函数と Bi 函数を Maxima に実装したものです. また, airy_dai 函数と airy_dbi 函数は Airy の Ai 函数と Bi 函数の導函数になります. この関係は Maxima でも定義されています:

(a) diff(airy_ai(z),z) ⇒ airy_dai(z)
(b) diff(airy_bi(z),z) ⇒ airy_dbi(z)

ただし, 逆は定義されていません.
ここで Airy 函数は二階の常微分方程式 $y'' - yz = 0$ の線形独立な解として得られる函数です. そのために Maxima でこれらの函数も, この微分方程式の解になります:

```
(%i2) diff(airy_ai(z),z,2)-airy_ai(z)*z;
(%o2)                                  0
(%i3) diff(airy_bi(z),z,2)-airy_bi(z)*z;
(%o3)                                  0
(%i4) diff(airy_dai(z),z)-airy_ai(z)*z;
(%o4)                                  0
(%i5) diff(airy_dbi(z),z)-airy_bi(z)*z;
(%o5)                                  0
```

これらの函数は airy.lisp[4] で定義されています.

[4] ©2005 David Billinghurst

9.4.2　Bessel 函数

第 1 種 Bessel 函数

第 1 種 Bessel 函数

函数	属性
bessel_j(⟨ 次数 ⟩,⟨ 変数 ⟩)	transfun
bessel_i(⟨ 次数 ⟩,⟨ 変数 ⟩)	transfun
scaled_bessel_i(⟨ 次数 ⟩,⟨ 変数 ⟩)	system function
scaled_bessel_i0(⟨ 変数 ⟩)	system function
scaled_bessel_i1(⟨ 変数 ⟩)	system function
spherical_bessel_j(⟨ 次数 ⟩,⟨ 変数 ⟩)	

bessel_j 函数:　第 1 種の Bessel 函数です. この函数は引数を二つ取り, 第 1 引数が次数となります. 具体的には次の式で定義されています:

$$\texttt{bessel\_j}(v,z) \stackrel{def}{=} \sum_{k=0}^{\infty} \frac{(-1)^k 2^{-v-2k} z^{v+2k}}{k!\Gamma(v+k+1)}$$

なお, bessel_j 函数は bessel 函数 の実体です.

bessel_i 函数:　第 1 種の変形 Bessel 函数です. この函数は引数を二つ取り, 第 1 引数が次数となります. 具体的には次の式で定義されています:

$$\texttt{bessel\_i}(v,z) \stackrel{def}{=} \sum_{k=0}^{\infty} \frac{2^{-v-2k} z^{v+2k}}{k!\Gamma(v+k+1)}$$

scaled_bessel_i 函数:　第 1 種の変形 Bessel 函数です. この函数は引数を二つ取り, 第 1 引数が次数となります. 具体的には次の式で定義されています.

$$\texttt{scaled\_bessel\_i}(v,z) \stackrel{def}{=} \exp(-|z|)\texttt{bessel\_i}(v,z)$$

scaled_bessel_i0 函数と scale_bessel_i1 函数:　scaled_bessel_i 函数の第一引数を 0 や 1 にした函数に対応します. そのために引数は一つだけになります.

spherical_bessel_i 函数:　第 1 種の球 Bessel 函数です.

第 2 種 Bessel 函数

第 2 種 Bessel 函数

bessel_k(⟨ 次数 ⟩,⟨ 変数 ⟩)	transfun
bessel_y(⟨ 次数 ⟩,⟨ 変数 ⟩)	transfun
spherical_bessel_y(⟨ 次数 ⟩,⟨ 変数 ⟩)	

bessel_k 函数: 次の式で定義されている第 2 種の Bessell 函数:

$$\text{bessel\_k}(v,z) \stackrel{def}{=} \frac{\pi\csc(\pi v)(\text{bessel\_i}(-v,z)-\text{bessel\_i}(v,z))}{2}$$

bessel_y 函数: 次の式で定義されている第 2 種の Bessell 函数:

$$\text{bessel\_y}(v,z) \stackrel{def}{=} \frac{\cos(\pi v)\text{bessel\_j}(v,z)-\text{bessel\_j}(-v,z)}{\sin(\pi v)}$$

spherical_bessel_y 函数: 第 2 種の球 Bessel 函数です.

Bessel 函数に関連する大域変数

Bessel 函数に関連する大域変数

変数名	既定値	概要
besselexpand	false	Bessel 函数の展開を制御

大域変数 besselexpand は Bessel 函数の自動展開を制御する大域変数です. 大域変数 besselexpand が true の場合に次の変換が実行されます:

- bessel_j(1/2)(z) ⇒ sqrt(2/(%pi*z))*sin(z)
- bessel_i(1/2)(z) ⇒ sqrt(2/(%pi*z))*sinh(z)
- bessel_k(1/2)(z) ⇒ sqrt(%pi/(2*z))*exp(-z)
- bessel_y(1/2)(z) ⇒ -sqrt(2/(%pi*z))*cos(z)

関連する大域変数

変数名	概要
iarray	scaled_bessel_i 函数の実行時に生成
yarray	bessel_y 函数の実行時に生成
besselarray	bessel_i 函数の実行時に生成

9.4.3 Hankel 函数

orthopoly パッケージに含まれている Bessel 函数に関連する函数に第 1 種と第 2 種の Hankel 函数があります. これらの Hankel 函数は Bessel の微分方程式

$$\frac{d^2w}{dx^2}+\frac{1}{z}\frac{dw}{dz}+\left(1-\frac{nu^2}{z^2}\right)w=0$$

の独立な解として得られる函数 $H_\nu^{(1)}(z)$ と $H_\nu^{(2)}(x)$ です.

Hankel 函数

$H_\nu^{(1)}(z)=\frac{1}{\pi}\int_{L_1}e^{-iz\sin\zeta+i\nu\zeta}d\zeta$ 第 1 種の Hankel 函数
$H_\nu^{(2)}(z)=\frac{1}{\pi}\int_{L_2}e^{-iz\sin\zeta+i\nu\zeta}d\zeta$ 第 2 種の Hankel 函数

ここで L_1, L_2 は積分経路を表現し, L_1 が $(-\pi+0)+i\infty$ から $-0-i\infty$ へ, L_2 が $+0-i\infty$ から $(\pi-0)+i\infty$ へ至る経路になります.

Hankel 函数

spherical_hankel1(⟨ 正整数 ⟩,⟨ 変数 ⟩)	第 1 種の Hankel 球函数
spherical_hankel2(⟨ 正整数 ⟩,⟨ 変数 ⟩)	第 2 種の Hankel 球函数

Spense 函数

li[⟨ 正整数 ⟩](⟨ 変数 ⟩)

次数 ⟨ 整数 ⟩ とする Spense 函数です. この函数は次の式で定義されています:

$$Li_s(z) = \sum_{i=1}^{\infty} \frac{z^i}{k^s}$$

したがって, $Li_1(z)$ は $-\log(1-z)$ となります.

9.5　hypgeo パッケージ

specint 函数

specint(exp(-⟨ 変数$_1$⟩ * ⟨ 変数$_2$⟩) * ⟨ 式 ⟩,⟨ 変数$_2$⟩)

specint 函数:　⟨ 式 ⟩ の ⟨ 変数$_2$⟩ に対する Laplace 変換を計算します.
上手く積分が出来なかった場合に内部変数を表示することがありますが, これは specint 函数の虫です.

9.6　orthopoly パッケージ

orthopoly パッケージは直交多項式を処理するためのパッケージです.

9.6.1　Chebyshev 多項式

Chebyshev 多項式

chevyshev_t(⟨ 正整数 ⟩,⟨ 変数 ⟩)
chevyshev_u(⟨ 正整数 ⟩,⟨ 変数 ⟩)

9.6.2　Hermite 多項式

Hermite 多項式

Hermite(⟨ 正整数 ⟩,⟨ 変数 ⟩)

9.6.3 超球多項式

Jacobi 多項式の重み函数 $w(x) = (1-x)^{\alpha}(1+x)^{\beta}$ の α と β が等しい場合に得られる多項式が超球多項式, あるいは Gegenbauer 多項式と呼ばれる多項式です.

超球 (Gegenbauer) 多項式

ultraspherical(⟨ 正整数$_1$⟩,⟨ 正整数$_2$⟩,⟨ 変数 ⟩)	

9.6.4 Jacobi 多項式

Jacobi 多項式 $\{P_n^{(\alpha,\beta)}(x)\}$ は重み函数を $w(x) = (1-x)^{\alpha}(1+x)^{\beta}$ とし, 閉区間 $[-1,1]$ で定義される次の多項式です:

Jacobi 多項式

$$P_n^{(\alpha,\beta)}(x) = \frac{(-1)^n}{2^n n!}\frac{1}{w(x)}\frac{d^n}{dx^n}[w(x)(1-x^2)^n]$$

Jacobi 多項式

jacobi(⟨ 正整数 ⟩,⟨ 変数 ⟩)	

9.6.5 Laguerre の多項式

Laguerre 多項式

laguerre(⟨ 正整数 ⟩,⟨ 変数 ⟩)	Laugerre 多項式
gen_laguerre(⟨ 正整数$_1$⟩,⟨ 正整数$_2$⟩,⟨ 変数 ⟩)	一般化 Laugerre 多項式

9.6.6 Legendre の多項式

Legendre の微分方程式は次の形式の常微分方程式です:

$$\frac{d}{dx}\left[(1-x^2)\frac{dy}{dx}\right] + \nu(\nu+1)y = 0$$

ここで *nu* が正整数の場合, 解は ν 次の多項式となります. この多項式のことを Legendre の多項式と呼びます.

legendre 多項式

legendre_p(⟨ 正整数 ⟩,⟨ 変数 ⟩)	第 1 種の Legendre 多項式
legendre_q(⟨ 正整数 ⟩,⟨ 変数 ⟩)	第 2 種の Legendre 多項式
assoc_legendre_p(⟨ 正整数$_1$⟩,⟨ 正整数$_2$⟩,⟨ 変数 ⟩)	第 1 種の Legendre の同伴多項式
assoc_legendre_q(⟨ 正整数$_1$⟩,⟨ 正整数$_2$⟩,⟨ 変数 ⟩)	第 2 種の Legendre の同伴多項式

legendre_p 函数:

9.7　楕円函数

9.7.1　楕円積分の概要

楕円積分について

$p(x)$ を 3 次, あるいは 4 次の多項式とし, $f(x,y)$ を変数 x と y の有理式とします. このとき, $f(x,\sqrt{p(x)})$ を楕円無理函数と呼びます. そして, 楕円無理函数の不定積分: $\int f\left(x,\sqrt{p(x)}\right)dx$ を総称して楕円積分と呼びます. ここで $p(x)=0$ の根を $\alpha_1,\alpha_2,\alpha_3$ と α_4 とするときに, これらの根の中の一つが ∞ であると考えることで $p(x)$ が 3 次の場合に対処できるために $p(x)$ を 4 次式として考えられます. そして, 楕円積分は適当な変数変換によって次の三種類の楕円積分と初等函数の和として表現されることが知られています:

- 第 1 種楕円積分:

$$\int \frac{dt}{\sqrt{(1-t^2)(1-k^2t^2)}}$$

- 第 2 種楕円積分:

$$\int \sqrt{\frac{1-k^2t^2}{1-t^2}}dt$$

- 第 3 種楕円積分:

$$\int \frac{dt}{(1+nt^2)\sqrt{(1-t^2)(1-k^2t^2)}}$$

これらの三種類の楕円積分を Legendre-Jacobi の標準形と呼び, 各積分の変数 k を楕円積分の母数, 第 3 種楕円積分の変数 n を助変数と呼びます.

楕円積分の名前の由来

ここで楕円積分の由来ですが, 楕円の弧長として第 2 種の楕円積分が出現することに由来します. 実際, 楕円の方程式を $\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$ としましょう. それから, 線素を ds とし, 次に $x=a\ \cos\ \theta$, $y=b\ \sin\ \theta$ で変数変換を行うと, $dx=-a\ \sin\ \theta\,d\theta$, $dy=b\ \cos\ \theta\,d\theta$ が得られ, $\mathrm{ds}=\sqrt{\mathrm{dx}^2+\mathrm{dy}^2}$ より楕円の弧長 L は

$$L=4b\int_0^{\frac{\pi}{2}}\sqrt{1-\frac{a^2-b^2}{b^2}\sin^2\ \theta}\,d\theta$$

で与えられます.

ここで, $u = \sin\ \theta, k = \sqrt{(a^2 - b^2)/b^2}$ とすれば, 楕円の弧長は最終的に

$$L = 4b \int_0^1 \sqrt{\frac{1 - k^2 u^2}{1 - u^2}}\, du$$

で与えられますが, この式中に第 2 種の楕円積分が出現していますね.

不完全楕円積分と完全楕円積分

各種楕円積分に対し, $t = \sin\ \phi$ で変数変換を行なった式を考えます. この際に, ϕ について 0 から $\pi/2$ までの積分を行ったものを特に完全楕円積分と呼びます:

- 第 1 種不完全楕円積分:

$$F(k, \phi) = \int_0^{\phi} \frac{d\phi}{\sqrt{1 - k^2\ \sin^2\ \phi}}$$

- 第 1 種完全楕円積分:

$$K(k) = F\left(k, \frac{\pi}{2}\right)$$

- 第 2 種不完全楕円積分:

$$E(k, \phi) = \int_0^{\phi} \sqrt{1 - k^2\ \sin^2\ \phi}\, d\phi$$

- 第 2 種完全楕円積分:

$$E(k) = E\left(k, \frac{\pi}{2}\right)$$

- 第 3 種不完全楕円積分:

$$\Pi(n, \phi, k) = \int_0^{\phi} \frac{d\phi}{(1 + n \sin^2\ \phi)\sqrt{1 - k^2 \sin^2\ \phi}}$$

9.7.2 楕円積分が満す関係式

ここでは楕円積分の満す関係式について簡単に述べておきましょう.

母数 k が零の場合　第 1 種の楕円積分の場合, $F(0,\phi) = \int_0^\phi d\phi$ となるために, $F(0,\phi) = \phi$ となります. したがって第 1 種完全楕円積分であれば $\pi/2$ になります.
第 2 種の楕円積分の場合も同様に $E(0,\phi) = \int_0^\phi d\phi$ となるために $E(0,\phi) = \phi$ となります. ここで第 2 種完全楕円積分は第 2 種不完全楕円積分の 0 から $\pi/2$ までの定積分なので, 第 2 種完全楕円積分の値は $\pi/2$ になります. そして, 第 3 種の楕円積分の場合,

$$\Pi(n,\phi,0) = \int_0^\phi \frac{d\phi}{1 + n\ \sin^2\ \phi}$$

なので, $n = 0$ ならは第 1 種や第 2 種と同じ ϕ, $n \neq 0$ であれば,

$$\frac{\mathrm{atanh}(\sqrt{|n-1|}\ \tan\phi)}{\sqrt{|n-1|}}$$

となります.

母数 k が 1 の場合　第 1 種楕円積分の場合, $F(1,\phi) = \int_0^\phi \frac{d\phi}{\sqrt{1-\sin^2\phi}}$ より,

$$F(1,\phi) = \log\left(\tan\frac{\phi}{2} + \frac{\pi}{2}\right)$$

となります.
また第 2 種楕円積分の場合は $F(1,\phi) = \int_0^\phi \sqrt{1 - \sin^2\ \phi}\ d\phi$ より $\sin\ \phi$ を得ます. したがって, 第 2 種完全楕円積分の値は 1 になります.

Legendre の関係式:　第 1 種と第 2 種の完全積分にたいしては, $k' = \sqrt{1-k^2}$ を補母数と呼び, この補母数に対して $K' = K(k')$, $E' = E(k')$ とおくと, $EK' + E'K - KK' = \frac{\pi}{2}$ が成立します. この関係式のことを Legendre の関係式と呼びます.

9.7.3　Maxima での楕円積分

Maxima での楕円積分について解説します. Maxima では不完全積分の数学で通常用いられる表記と比べ, その引数の配置が異なる点と, Maxima では母数を与えるのではなく, 母数の 2 乗を引渡す点の違いがあります. また, 命名上の規則として, 頭に “elliptic_” が付きます.
なお, Maxima の楕円積分ではその微分は属性 gradef を用いて, その属性値として付与されていますが, taylor 展開には対応していないために, 楕円積分に taylor 函数を作用させても, エラーが出るだけです.
また, 楕円函数の引数が全て倍精度の浮動小数点数の場合, 倍精度の浮動小数点数で評価した値を返却します.
なお, 現時点での Maxima の楕円積分は母数が 0 の場合と, その微分といった, 基本的な性質のみが実装されており, 細かな性質は未実装です. この点は後述の Jacobi の楕円函数の実装でも同様です.

第 1 種楕円積分について

第 1 種楕円積分
elliptic_f(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
elliptic_kc(⟨ 変数 ⟩)

elliptic_f 函数: 次の式で定義された第 1 種不完全楕円積分函数です:

$$\text{elliptic\_f}(\phi, m) \stackrel{def}{=} \int_0^{\phi} \frac{dx}{\sqrt{1 - m\ \sin^2 x}}$$

つまり,elliptic_f$(\phi, m) = F(\sqrt{m}, \phi)$ で, 母数 $= \sqrt{\langle\ 変数_2 \rangle}$ となっていることに注意が必要です. また, この函数の微分は属性 gradef の属性値として付与されています.

elliptic_kc 函数: 次の式で定義された第 1 種完全楕円積分函数です:

$$\text{elliptic\_kc}(m) \stackrel{def}{=} \int_0^{\frac{\pi}{2}} \frac{dx}{\sqrt{1 - m\ \sin^2 x}}$$

すなわち, elliptic_kc$(m) = K(\sqrt{m})$ であり, 母数 $= \sqrt{\langle\ 変数\ \rangle}$ となっていることに注意して下さい. なお, この函数は属性として属性 transfun のみを持っています.
ここで elliptic_f 函数と elliptic_kc 函数に対しては, これらの函数の定義から
'elliptic_f(%pi/2,m)=elliptic_kc(m)' が成立します:

```
(%i22) elliptic_f(%pi/2,m)-elliptic_kc(m);
(%o22)                                0
```

第 2 種楕円積分について

第 2 種楕円積分
elliptic_e(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
elliptic_ec⟨ 変数 ⟩)
elliptic_eu⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)

elliptic_e 函数: 次の式で定義された第 2 種不完全楕円積分函数です:

$$\text{elliptic\_e}(\phi, m) \stackrel{def}{=} \int_0^{\phi} \sqrt{1 - m\ \sin^2 x} dx$$

したがって, elliptic_e$(\phi, m) = E(\sqrt{m}, \phi)$ であり, 母数 $= \sqrt{\langle\ 変数_2 \rangle}$ となっていることに注意が必要です. また, この函数の微分は属性 gradef の属性値として与えられています.

elliptic_ec 函数:　次の式で定義された第 2 種完全楕円積分函数です

$$\text{elliptic\_ec}(m) \stackrel{def}{=} \int_0^{\frac{\pi}{2}} \sqrt{1 - m\sin^2 x}dx$$

つまり, elliptic_ec(m) = $E(\sqrt{m})$ であり, 母数 = $\sqrt{\langle\ 変数\ \rangle}$ となっていることに注意して下さい. また, この函数には属性として属性 transfun のみが付与されています.
ここで elliptic_e 函数と elliptic_ec 函数に関しては, それらの定義から
'elliptic_e(%pi/2,m)=elliptic_ec(m)' が成立します.

```
(%i38)  is(elliptic_e(%pi/2,m)=elliptic_ec(m));
(%o38)                                  true
```

elliptic_eu 函数:　次の式で定義された第 2 種楕円積分函数です:

$$\text{elliptic\_eu}(u,m) \stackrel{def}{=} \int_0^{\tau} \text{dn}^2(x,m)dx \qquad \tau = sn(u,m)$$

この函数の定義からも判るように 母数 = $\sqrt{\langle 変数_2 \rangle}$ となっています. そして, この函数の微分は属性 gradef の属性値として付与されています.

第 3 種楕円積分について

第 3 種楕円積分

elliptic_pi($\langle$ 変数$_1\rangle$,$\langle$ 変数$_2\rangle$,$\langle$ 変数$_3\rangle$)

elliptic_pi 函数:　次の式で定義された第 3 種楕円積分函数です:

$$\text{elliptic\_pi}(n,\phi,m) \stackrel{def}{=} \int_0^{\phi} \frac{dx}{(1 - n\ \sin^2 x)\sqrt{1 - m\ \sin^2 x}}$$

つまり, elliptic_pi(n,ϕ,m) = $\Pi(\phi,n,\sqrt{m})$ であり, 母数 = $\sqrt{\langle 変数_3 \rangle}$ となっていることに注意して下さい.

9.7.4　Jacobi の楕円函数

Jacobi の楕円函数の概要

Jacobi の楕円函数は楕円積分の逆函数として導入されたものです. まず, 第 1 種不完全楕円積分 $F(k,x)$ の母数 k が 0 の場合を考えてみましょう. この場合は, $F(0,x) = \int_0^x \frac{1}{1-t^2}dt$ となりますが, この式は $\sin^{-1}\ x$ です. そこで, $F(k,x)$ を $F_k(x)$ とおくと, 函数 F_0 の逆函数 F_0^{-1} は $F_0^{-1}(x) = \sin x$ を満します.
そこで, より一般的に F_k の逆函数を考え, この逆函数を sn と定義します:

$$\mathrm{sn}(u,k) \stackrel{def}{=} F_k^{-1}(u)$$

この $\mathrm{sn}(u,k)$ は母数 k を省略して単に sn u と一般に表記されますが, 必要に応じて $sn(u,k)$ とも表記します.

この sn を基に函数 cn と函数 dn を定義しておきましょう:

$$\mathrm{cn}(u,k) \stackrel{def}{=} \sqrt{1-sn^2(u,k)}$$

$$\mathrm{dn}(u,k) \stackrel{def}{=} \sqrt{1-k^2sn^2(u,k)}$$

これらの定義式により, $\mathrm{sn}^2+\mathrm{cn}^2=1$ を満し, 母数 $k=0$ の場合, $\mathrm{cn}(u,0)=\cos\ u$, $\mathrm{dn}(u,0)=1$ が成立します. また, 母数 $k=1$ とした場合, 第 1 種の楕円積分は $\int_0^x \frac{1}{1-t^2}dt$ となりますが, この函数は $\tanh^{-1}\ x$ となります. したがって, $\mathrm{sn}(x,1)=\tanh\ x$ となり, さらに, $\mathrm{cn}(x,1)=\mathrm{dn}(x,1)=\mathrm{sech}\ x$ が成立します。

ここで定義した sn, cn と dn を使って, 函数 ns, 函数 sc, 函数 sd と函数 cd 等が定義されます:

$$\begin{array}{ll}
\mathrm{ns}(u,k) \stackrel{def}{=} \frac{1}{\mathrm{sn}(u,k)} & \\
\mathrm{nc}(u,k) \stackrel{def}{=} \frac{1}{\mathrm{cn}(u,k)} & \mathrm{nd}(u,k) \stackrel{def}{=} \frac{1}{\mathrm{dn}(u,k)} \\
\mathrm{sc}(u,k) \stackrel{def}{=} \frac{\mathrm{sn}(u,k)}{\mathrm{cn}(u,k)} & \mathrm{cs}(u,k) \stackrel{def}{=} \frac{\mathrm{cn}(u,k)}{\mathrm{sn}(u,k)} \\
\mathrm{sd}(u,k) \stackrel{def}{=} \frac{\mathrm{sn}(u,k)}{\mathrm{dn}(u,k)} & \mathrm{ds}(u,k) \stackrel{def}{=} \frac{\mathrm{dn}(u,k)}{\mathrm{sn}(u,k)} \\
\mathrm{cd}(u,k) \stackrel{def}{=} \frac{\mathrm{cn}(u,k)}{\mathrm{dn}(u,k)} & \mathrm{dc}(u,k) \stackrel{def}{=} \frac{\mathrm{dn}(u,k)}{\mathrm{cn}(u,k)}
\end{array}$$

また, Jacobi の振幅函数 $\mathrm{am}(u,k)$ は $\mathrm{sn}^{-1}(\sin\phi)$ の逆函数として得られる函数です.

Jacobi の楕円函数の微分

Jacobi の楕円函数 $\mathrm{sn},\mathrm{cn},\mathrm{dn}$ の微分はどうなるでしょうか？ まず, sn の微分を考えてみましょう. $t=sn(u,k)$ とすると定義から $u=\int_0^x \frac{dt}{\sqrt{(1-t^2)(1-k^2t^2)}}$ となり, $du/dt=1/\sqrt{(1-t^2)(1-k^2t^2)}$ と $t=sn(u,k)$ より,

$$\frac{d\mathrm{sn}(u,k)}{du} = \sqrt{(1-\mathrm{sn}^2(u,k))(1-k^2\mathrm{sn}^2(u,k))} = \mathrm{cn}(u,k)\mathrm{dn}(u,k)$$

を得ます.

cn の微分は $\mathrm{cn} \stackrel{def}{=} \sqrt{1-\mathrm{sn}^2}$ より,

$$\frac{d\mathrm{cn}(u,k)}{du} = \frac{-\mathrm{sn(u,k)cn(u,k)dn(u,k)}}{\sqrt{1-\mathrm{sn}^2(u,k)}} = -\mathrm{sn(u,k)dn(u,k)}$$

を得ます. また, dn の微分は dn の定義から,

$$\frac{d\mathrm{dn}(u,k)}{du} = -k\ \mathrm{sn}(u,k)\mathrm{cn}(u,k)$$

を得ます.

函数 ns, 函数 sc, 函数 sd と函数 cd の定義から, これらの函数の微分も容易に計算できます.

9.7.5　Maxima での Jacobi の楕円函数

Jacobi の楕円函数について解説します. Maxima の Elliptic パッケージでは, Jacobi の楕円函数名は必ず “jacobi_” が先頭に付きます:

Jacobi の楕円函数 (その 1)

jacobi_sn(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
jacobi_cn(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
jacobi_dn(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
jacobi_am(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)

jacobi_sn 函数は Jacobi の sn 函数, jacobi_cn 函数は Jacobi の cn 函数, jacobi_dn 函数は Jacobi の dn 函数となります. そして, jacobi_am 函数が振幅函数になります.
これらの函数では ⟨ 変数$_2$⟩ が通常の母数の二乗となっていることに注意が必要です. また, これらの函数の微分は内部函数の属性として指定されています. そして, ⟨ 変数$_1$⟩ = 0, ⟨ 変数$_2$⟩ = 0 と ⟨ 変数$_2$⟩ = 1 の場合は各函数の簡易化函数で予め変換式が設定されていますが, 和公式といったものは実装されていません.
ここで Jacobi の楕円函数 sn,cn と dn の第 1 引数による微分を計算してみましょう:

```
(%i1) diff(jacobi_cn(u,k),u);
(%o1)                  - jacobi_dn(u, k) jacobi_sn(u, k)
(%i2) diff(jacobi_sn(u,k),u);
(%o2)                   jacobi_cn(u, k) jacobi_dn(u, k)
(%i3) diff(jacobi_dn(u,k),u);
(%o3)                 - k jacobi_cn(u, k) jacobi_sn(u, k)
```

このように, Jacobi の楕円函数の微分公式が返されていることが判りますね.

Jacobi の逆楕円函数 (その 1)

inverse_jacobi_sn(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
inverse_jacobi_cn(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
inverse_jacobi_dn(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)

inverse_jacobi_sn 函数:　inverse_jacobi_sn 函数は, $F(m,\Phi)$ = asn(sin ϕ, m) で定義された函数 asm のことです. この函数の微分は対応する内部函数の属性として与えられています.

inverse_jacobi_cn 函数:　inverse_jacobi_cn(x,m) $\stackrel{def}{=}$ inverse_jacobi_sn(sqrt(1-x^2),m) で定義された函数です. この函数の微分は対応する内部函数の属性として与えられています.

inverse_jacobi_dn 函数:　inverse_jacobi_dn $\stackrel{def}{=}$ inverse_jacobi_sn(sqrt(1-x^2)/sqrt(m)) で定義された函数です. この函数の微分は対応する内部函数の属性として与えられています.

Jacobi の楕円函数 (その 2)

jacobi_ns(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
jacobi_nc(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
jacobi_nd(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)

jacobi_ns 函数: 次の式で定義された Jacobi の楕円積分函数です:

$$\texttt{jacobi\_ns(u,m)} \stackrel{def}{=} \frac{1}{\texttt{jacobi\_sn(u,m)}}$$

この函数の微分は対応する内部函数の属性として与えられています.

jacobi_nc 函数: 次の式で定義された Jacobi の楕円積分函数です:

$$\texttt{jacobi\_nc(u,m)} \stackrel{def}{=} \frac{1}{\texttt{jacobi\_cn(u,m)}}$$

この函数の微分は対応する内部函数の属性として与えられています.

jacobi_nd 函数: 次の式で定義された Jacobi の楕円積分函数です:

$$\texttt{jacobi\_nd(u,m)} \stackrel{def}{=} \frac{1}{\texttt{jacobi\_dn(u,m)}}$$

この函数の微分は対応する内部函数の属性として与えられています.

Jacobi の逆楕円函数 (その 2)

inverse_jacobi_ns(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
inverse_jacobi_nc(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
inverse_jacobi_nd(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)

inverse_jacobi_ns 函数: inverse_jacobi_ns 函数は jacobi_sn 函数の逆函数になります. すなわち, inverse_jacobi_ns(jacobi_sn(u,m),m)=u を満します. この函数の微分は対応する内部函数の属性として与えられています.

inverse_jacobi_nc 函数: inverse_jacobi_ns 函数は jacobi_cn 函数の逆函数になります. この函数の微分は対応する内部函数の属性として与えられています.

inverse_jacobi_nd 函数: inverse_jacobi_nd 函数は jacobi_dn 函数の逆函数になります. この函数の微分は対応する内部函数の属性として与えられています.

Jacobi の楕円函数 (その 3)

jacobi_sc(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
jacobi_sd(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
jacobi_cs(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
jacobi_cd(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
jacobi_ds(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
jacobi_dc(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)

jacobi_sc 函数:　次の式で定義された Jacobi 楕円積分函数です:

$$\texttt{jacobi\_sc(u,m)} \stackrel{def}{=} \frac{\texttt{jacobi\_sn(u,m)}}{\texttt{jacobi\_cn(u,m)}}$$

この函数の微分は対応する内部函数の属性として与えられています.

jacobi_sd 函数:　次の式で定義された Jacobi 楕円積分函数です:

$$\texttt{jacobi\_sd(u,m)} \stackrel{def}{=} \frac{\texttt{jacobi\_sn(u,m)}}{\texttt{jacobi\_dn(u,m)}}$$

この函数の微分は対応する内部函数の属性として与えられています.

jacobi_cs 函数:　次の式で定義された Jacobi 楕円積分函数です:

$$\texttt{jacobi\_cs(u,m)} \stackrel{def}{=} \frac{\texttt{jacobi\_cn(u,m)}}{\texttt{jacobi\_sn(u,m)}}$$

この函数の微分は対応する内部函数の属性として与えられています.

jacobi_cd 函数:　次の式で定義された Jacobi 楕円積分函数です:

$$\texttt{jacobi\_cd(u,m)} \stackrel{def}{=} \frac{\texttt{jacobi\_cn(u,m)}}{\texttt{jacobi\_dn(u,m)}}$$

この函数の微分は対応する内部函数の属性として与えられています.

jacobi_ds 函数:　次の式で定義された Jacobi 楕円積分函数です:

$$\texttt{jacobi\_ds(u,m)} \stackrel{def}{=} \frac{\texttt{jacobi\_dn(u,m)}}{\texttt{jacobi\_sn(u,m)}}$$

この函数の微分は対応する内部函数の属性として与えられています.

jacobi_dc 函数: 次の式で定義された Jacobi 楕円積分函数です:

$$\texttt{jacobi\_dc(u,m)} \stackrel{def}{=} \frac{\texttt{jacobi\_dn(u,m)}}{\texttt{jacobi\_cn(u,m)}}$$

この函数の微分は対応する内部函数の属性として与えられています.

Jacobi の逆楕円函数 (その 3)
inverse_jacobi_sc(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
inverse_jacobi_sd(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
inverse_jacobi_cs(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
inverse_jacobi_cd(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
inverse_jacobi_ds(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
inverse_jacobi_dc(⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)

inverse_jacobi_sc 函数: jacobi_sc(u,m) 函数の u に関する逆函数として与えられる函数です. 微分は対応する内部函数の属性として与えられています.

inverse_jacobi_sd 函数: jacobi_sd(u,m) 函数の u に関する逆函数として与えられる函数です. 微分は対応する内部函数の属性として与えられています.

inverse_jacobi_cs 函数: jacobi_cs(u,m) 函数の u に関する逆函数として与えられる函数です. 函数の微分は対応する内部函数の属性として与えられています.

inverse_jacobi_cd 函数: jacobi_cd(u,m) 函数の u に関する逆函数として与えられる函数です. 函数の微分は対応する内部函数の属性として与えられています.

inverse_jacobi_ds 函数: jacobi_ds(u,m) 函数の u に関する逆函数として与えられる函数です. 函数の微分は対応する内部函数の属性として与えられています.

inverse_jacobi_dc 函数: jacobi_dc(u,m) 函数の u に関する逆函数として与えられる函数です. この函数の微分は対応する内部函数の属性として与えられています.

9.7.6 ζ 函数に関連する函数

make_elliptic_e 函数と make_elliptic_f 函数
make_elliptic_e(⟨ 式 ⟩)
make_elliptic_f(⟨ 式 ⟩)

これらの函数は共に, 与式に inverse_jacobi_sc 函数, inverse_jacobi_cs 函数, inverse_jacobi_nd 函数, inverse_jacobi_dn 函数, inverse_jacobi_sn 函数, inverse_jacobi_cd 函数, inverse_jacobi_dc 函数, inverse_jacobi_ns 函数, inverse_jacobi_nc 函数, inverse_jacobi_ds 函数, inverse_jacobi_sd 函数,

inverse_jacobi_cn 函数が含まれている項に対してのみ作用し, それ以外の式はそのままの式を返します.

9.8 力学系と dynamics パッケージ

9.8.1 力学系について

力学系とは, ある時刻の系の状態を定めることで以降の任意の時刻での状態が決定可能となる系のことです. ここで, 系の時間に対する挙動が微分方程式で表現される場合, 時間が連続的になるために「連続力学系」と呼びます. これに対し, 系の挙動が写像で表現される場合は離散的な時間の媒介変数を有するために「離散力学系」と呼びます.
ここで離散力学系で重要な函数を二つ挙げておきましょう. 一つは Poincaré 写像で, もう一つは Hénon 写像です.

Poincaré 写像: この写像は Poincaré が三体問題の研究で導入したもので, 連続力学系の問題を離散力学系で置換えて考察することを可能にする写像です. この Poincaré 写像の構成は一般的な構築方法はありませんが, 周期的な解を持つ力学系に対しては, 容易に構築可能なことが知られています.
まず, $\mathbb{R}^n$ の常微分方程式 $dx/dt = f(x)$ の解が周期 T_0 の周期軌道 $x_0(t)$ であったとします. ここで周期解 $x_0(t)$ 上の一点 $X_0 \in \mathbb{R}^n$ を取り, この点 X_0 で周期軌道 x_0 と横断的に交差する平面 Σ を考えます. なお, 「横断的」とは平面 Σ 上のベクトルと軌道 x_0 上の点 X_0 での接ベクトルの外積が 0 にならないことを意味します. さて, 平面 Σ 上で点 X_0 の十分小さな近傍 D を考えましょう. ここで点 $y \in D$ を考えると, 周期 T_0 に非常に近い時刻 $T(y)$ で再び平面 Σ 上に戻って来ます. すなわち, D の各点は Σ 上の点に再び写されることになります. このように平面 Σ を使って定義される写像 P を「Poincaré 写像」と呼びます:

$$\begin{array}{ccccc} P & : & D & \to & \Sigma \\ & & \cup & & \cup \\ & & y & \mapsto & \phi_{T(y)}(y) \end{array}$$

また, この写像 P の定義で用いた平面 Σ を「Poincaré 平面」と呼びます. この Poincaré 写像が定義出来ると, 点 $y_0 \in D$ に対して点列 $y_0, P(y_0), P^2(y_0), \cdots$ が得られます. なお, P^n は P の n 個の合成 $\underbrace{P \circ \cdots \circ P}_{n}$ を意味します. この点列 $\{P^i(y_0)\}_{0 \le i}$ を点 y_0 を初期値とする「軌道」と呼びます.

たとえば, 写像を $x_{n+1} = 1 - \dfrac{9}{10}x_n^2$ とし, その初期値を 0.8 とした場合の軌道を図 9.1 に示しておきます:

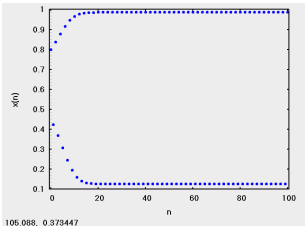


図 9.1: 写像 $1-9/10x^2$ による軌道例

何か二列になっていますね. この図だけでは中間がないので, どのような現象が生じているか判りませんね. では, この軌道を図的に解説してみましょう:

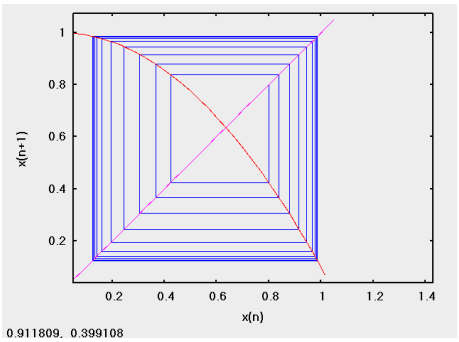


図 9.2: 写像 $1-9/10x^2$ による軌道の分析

この図 9.2 では $y=1-9/10x^2$ と $y=x$ のグラフが描かれています. 初期値は $y=x$ 上の点として置かれています. 四角い線は軌道の動きを表現すものです. ここで初期値は 0.8 なので, 図では函数 $y=x$ 上の点 (0.8,0.8) に置きます. これがこの図的解説の出発点です. 0.8 は函数によって 0.424 に写されます. この様子は点 (0.8,0.8) から $1-9/10x^2$ のグラフに Y 軸に平行に下した線分から 0.424 が読み取れます. 次に, この値を写像 $1-9/10x^2$ に与えるために今度は $y=x$ に向けて X 軸に平行な線分を引いて $y=x$ との交点で止めます. 以降, この操作を繰り返せば升目との交点が軌道となります. その結果, 軌道は徐々に二点に収束している様子が見えます. このことから $n\to\infty$ とすると周期 2 の軌道となることが判ります. このように軌道の $n\to\infty$ の振舞は Poincaré 写像の性質を知る上で非常に重要な情報の一つになります.

次に写像 $y = 1 - ax^2$ の a を $1/2 < a < 1$ の範囲で動かすと軌道はどうなるでしょうか？

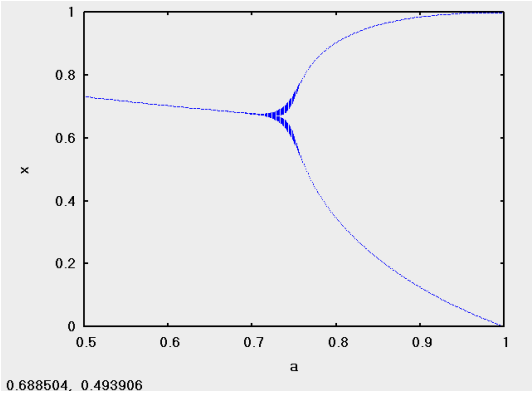


図 9.3: 写像 $1 - ax^2$ の分岐図

この図は Maxima の orbits 函数で描いたものですが, a が 0.7 と 0.8 の間の点に至ると軌道が二つに別れていることが判ります. この図 9.3 のように系の制御用の変数を横軸, 軌道の値を縦軸に描くことで軌道の分岐が読み取れるグラフを「分岐図」と呼びます.

Hénon 写像: Hénon が 3 次元連続力学系の Poincaré 写像のモデルとして導入した写像 $T : (x_i, y_i) \mapsto (x_{i+1}, y_{i+1})$ で次の形の写像です:

$$T : \begin{cases} x_{i+1} = y_i + 1 - ax_i^2 \\ y_{i+1} = bx_i \end{cases}$$

この Hénon 写像は次の三種類の函数 $T_1, T_2, T3$ の合成 $T_3 \circ T_2 \circ T_1$ で出来ています:

折り曲げ: この写像は水平方向は変更しませんが, Y 軸に対称に折り曲げる働きをします:

$$T_1 : \begin{cases} x' = x \\ y' = y + 1 - ax^2 \end{cases}$$

この写像 T では x^2 の係数 a が折り曲げの強さを表現します. なお, 折り曲げだけであれば, この写像によって写される領域は面積を保ちます.

縮小: 水平方向に対して b 倍します:

$$T_2 : \begin{cases} x' = bx \\ y' = y \end{cases}$$

ここで定数 b が '$0 < b < 1$' を満す実数であれば, この写像は領域を水平方向に縮小する写像になります.

入れ換え:　X 座標と Y 座標の入れ換えを実行する写像です;

$$T_3 : \begin{cases} x' = y \\ y' = x \end{cases}$$

この Hénon の写像 T の Jacobi 行列は次に示す行列になります:

$$\frac{\partial(x_{i+1}, y_{i+1})}{\partial(x_i, y_i)} = \begin{pmatrix} -2ax & 1 \\ b & 0 \end{pmatrix}$$

この Hénon 写像 T の Jacobian は $-b$ で, さらに $|b| < 1$ となることから, 写像 T が領域の面積を縮小させる写像であることが分ります.
さて, この Hénon の写像の係数を $a = 1/10, b = 1/2$ とし, 初期値を $(1, 1)$ として軌道を 11 点描いてみましょう:

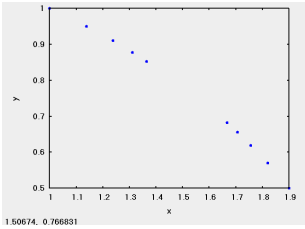


図 9.4: Henon 写像 ($a = 1/10, b = 1/2$) の初期値 (1,1) の軌道 (11 点)

これだけでは今一つ分りませんね. そこで, 軌道を 1001 個にしてみましょう:

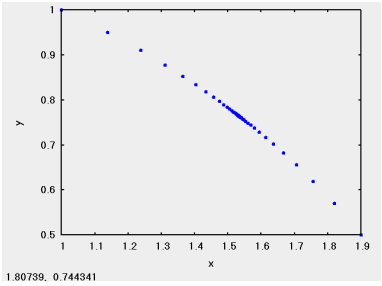


図 9.5: Henon 写像 ($a = 1/10, b = 1/2$) の初期値 (1,1) の軌道 (1001 個)

これで軌道が一点に収束して行く様子が見えますね. 実際, $n \to \infty$ で軌道 (x_i, y_i) が一点に収束した場合, この点は方程式

$$x = 1 + \frac{1}{2}x - \frac{1}{10}x^2, y = \frac{1}{2}x$$

を満すので, 点 $((\sqrt{65} - 5)/2, (\sqrt{65} - 5)/4)$ と点 $(-(\sqrt{65} + 5)/2, -(\sqrt{65} + 5)/4)$ の二点が解になります. 実際, 図 9.5 で軌道が収束している点は $((\sqrt{65} - 5)/2, (\sqrt{65} - 5)/4)$, 近似的には (1.53113, .765565) です.

では, 初期値をもう一つの解の近くの点 $(-(\sqrt{65}+5)/2+1/100, -(\sqrt{65}+5)/4)$ として軌道を描いてみましょう:

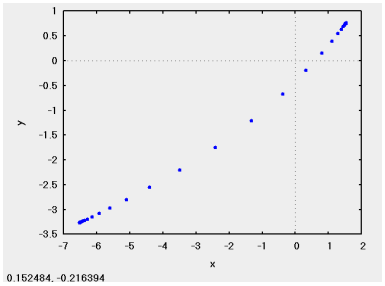


図 9.6: Henon 写像 $(a = 1/10, b = 1/2)$ の初期値 $(-\frac{\sqrt{65}+5}{2}+\frac{1}{100}, -\frac{\sqrt{65}+5}{4})$ の軌道 (1001 個)

すると, この初期値の軌道は近くの解の近くの点ではなく, 図 9.5 の点に収束しています. これはこれらの点の近傍の性格の違いによるものです. まず, 方程式 $x = 1 + y - ax^2,\ y = bx$ の解:

$$(X^{\pm}, Y^{\pm}) = (\frac{b-1 \pm \sqrt{(b-1)^2+4a}}{2a}, b\frac{b-1 \pm \sqrt{(b-1)^2+4a}}{2a})$$

は Hénon 写像 $x_{i+1} = 1 + y_i - ax_i^2,\ y_{i+1} = bx_i$ の不動点, すなわち, Hénon 写像を作用させても動かない点になります. ここで $a > 0$ と $1 \geq b \geq 0$ より $X^+ > 0$ と $X^- < 0$ を満すことが分ります. このことから Hénon 写像の浮動点は第一象限上と第三象限上に一つづつ存在することが分ります.
さて, この Hénon 写像 T の不動点 $P^{\pm} = (X^{\pm}, Y^{\pm})$ におけける Jacobi 行列 $J_T(P^{\pm})$ を考えてみましょう:

$$J_T(P^{\pm}) = \begin{pmatrix} -2aX^{\pm} & 1 \\ b & 0 \end{pmatrix}$$

したがって, この行列の特性多項式として多項式 $\lambda^2 + 2aX^{\pm}\lambda - b$ を得ます. そして, 方程式 $\lambda^2 + 2aX^{\pm}\lambda - b = 0$ を解くと, この行列の二つの固有値 $\lambda_1^{\pm}, \lambda_2^{\pm}$ を得ます:

$$\lambda_1^{\pm} = -\sqrt{(aX^{\pm})^2 + b} - aX^{\pm}$$
$$\lambda_2^{\pm} = \sqrt{(aX^{\pm})^2 + b} - aX^{\pm}$$

ここで f を写像とし, $J_f(p)$ を点 p での写像 f の Jacobi 行列とします. 不動点 p の近傍の点 x_0 に対しては $f(x_0) = J_f(p)(x_0 - p)$ とできます. このとき x_0 を初期値とする軌道は $\{J_f(p)^i\}_{i=0}^{\infty}$ となります. さて, $n \to \infty$ の時にどうなるでしょうか. この場合, 行列の固有値と固有ベクトルで考えましょう. 先ず, 一般の m 次の正方行列 A に対し, その固有値を $\{\lambda_i\}_{1\leq i\leq m}$, これらの固有値に対応する固有ベクトルを $\{v_i\}_{1\leq i\leq m}$ とします. すると, $A^n v_i = \lambda_i^n v_i$ となるので, $|\lambda_i| < 1$ であれば $n \to \infty$ で $A^n v_i \to 0$,$|\lambda_i| > 1$ であれば $|A^n v_i| \to \infty$,$|\lambda_i| = 1$ の場合,$|A^n v_i| = |v_i|$ であることが判ります.
これを不動点 p の周辺の軌道に当て嵌めると次のようになります:

固有値による不動点の分類

1.	固有値の絶対値が全て 1 よりも小	⇔	不動点は吸収点
2.	固有値の絶対値が全て 1 よりも大	⇔	不動点は発散点
3.	固有値の絶対値が全て 1	⇔	不動点周辺は周期的
4.	固有値の絶対値は 1 より小, 或いは 1 より大	⇔	不動点は鞍点

1. と 3. の不動点は「安定多様体」と呼ばれ,2. と 4. の不動点は「不安定多様体」と呼ばれます. さらに 4. の状態は図 9.7 に示す鞍の上にボールを置いた様子と類似しているために, このような不動点を「鞍点」と呼びます.

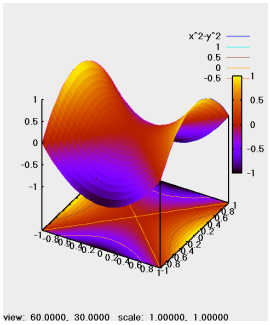


図 9.7: 鞍点の例

さて, Hénon 写像の Jacobi 行列の固有値 $\lambda_1^{\pm}, \lambda_2^{\pm}$ はどうでしょうか？ まず, 不動点 X^- での Jacobi 行列の固有値 λ_1^- と λ_1^- については, $|\lambda_1^-| < 1$ と $\lambda_{-2}^- > 1$ を満す為に鞍点になります. 次に, 不動点 X^+ での Jacobi 行列の固有値 λ^+ と λ_1^+ については, $a > \frac{3}{4}(1-b)^2$ で $|\lambda_1^+| < 1$ と $\lambda_{-2}^+ > 1$ を満す為に鞍点, $a < \frac{3}{4}(1-b)^2$ の場合は $|\lambda_1^+| < 1$ と $|\lambda_{-2}^+| < 1$ を満すために吸い込み点となります.
次に Hénon の写像の係数を $a = 11/10, b = 2/5$ として, 初期値 (1,1) で軌道を 10001 点描いてみましょう:

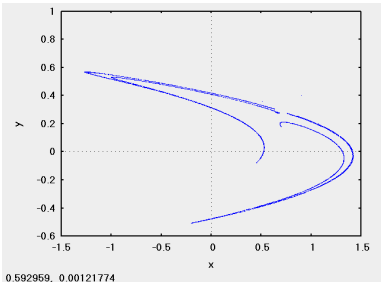


図 9.8: Henon 写像 ($a = 11/10, b = 2/5$) の軌道

何とも不可思議な軌道が現われていますが, この妙な図形は「Strange Attractor」と呼ばれます.
次に軌道の X 座標について折り曲げに関連する係数 a を変数として, 分岐図で軌道の分岐状況を調

べてみましょう:

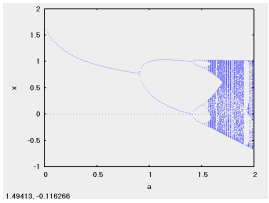


図 9.9: Henon 写像の X 座標に関する分岐図

この図に示すように 0.9 を過ぎたところで軌道が 2 個に分岐し, 以降, $2^2, 2^4, \ldots$ と 2 の倍数で分岐します. ところが, 1.56 を過ぎたところで大きく乱れて不可解な領域が出現します. この領域の事を「chaos」と呼びます. では問題の個所を拡大してみましょう:

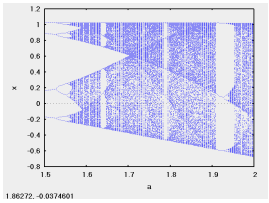


図 9.10: Chaos

さらに, この問題となる領域を拡大してみましょう:

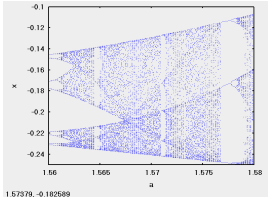


図 9.11: 自己相似性

これらの図に示すように強い自己相似性 (フラクタル構造) があることが分りますね. 基本的に周期軌道が幾つか分岐すると大きく乱れ, そのあとに空白が出現し, その空白では周期軌道が再度出現します. この周期軌道が現われる chaos 領域を「周期的ウィンドウ」と呼びます.

9.8.2 fractal を生成する代表的な函数系について

反復函数系 (IFS)

反復函数系は距離空間 X 上の有限個の Affine 縮小写像 $F_i : X \to X, (1 \leq i \leq N)$ から構成されます. ここで集合 X が距離空間であるとは,「距離の公理」を満す距離函数と呼ばれる函数 $dis : X^2 \to \mathbb{R}^+$

が存在することです. そして, 「距離の公理」は次の公理のことです. なお, $x, y, z \in X$ は任意です:

1. $dis(x, y) \geq 0$
2. $dis(x, y) = 0 \sim x = y$
3. $dis(x, y) = dis(y, x)$
4. $dis(x, y) \leq dis(x, z) + dis(z, y)$

次に, 写像 f が Affine 写像となるのは f が一次変換と平行移動で構成された函数であること, すなわち, 行列 A と平行移動の方向を示すベクトル v を用いて $f(x) = Ax+v$ と表現されることです. ここで一般の写像 $F : \mathbb{R}^n \to \mathbb{R}^n$ が「縮小写像」であるとは, 任意の $x, y \in \mathbb{R}^n$ に対して $|F(x) - F(y) \leq r|x - y|$ を満す $0 \leq r \leq 1$ が存在する場合です. この Affine 写像が縮小写像となるのは, 行列 A の各固有値の絶対値が 1 よりも小の場合になります.
反復函数系 $\cup_{i=1}^{m} F_i$ による fractal 図形は, まず, C を X のコンパクト集合とし, この C に対して,F_i を作用して得られた図形を $\mathbb{F}_i(C) = \cup_{k=0}^{\infty} F_i(C)$ とするときに $\cup_{i=1}^{N} \mathbb{F}_i(A)$ で与えられます.
たとえば, 反復函数系として次の函数を選びます[5]:

$$F_1(x, y) = \begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0.00 \\ 1.60 \end{pmatrix}$$

$$F_2(x, y) = \begin{pmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0.00 \\ 0.44 \end{pmatrix}$$

$$F_3(x, y) = \begin{pmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0.00 \\ 1.60 \end{pmatrix}$$

$$F_4(x, y) = \begin{pmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0.00 \\ 0.00 \end{pmatrix}$$

この函数系を Maxima の ifs 函数を用いて描いた図を 9.12 に示しておきましょう:

```
(%i26) mat1:matrix([0.85,0.04],[-0.04,.85])$
(%i27) mat2:matrix([-0.15,0.28],[0.26,.24])$
(%i28) mat3:matrix([0.2,-0.26],[0.23,.22])$
(%i29) mat4:matrix([0.0,0.0],[0.0,.16])$
(%i30) p1:[0,1.6]$
(%i31) p2:[0,0.44]$
(%i32) p3:[0,1.6]$
(%i33) p4:[0,0]$
(%i34) ifs([40,60,80,100],[mat1,mat2,mat3,mat4],
[p1,p2,p3,p4],[0,0],10000);
```

この図 9.12 は「Barnsley のシダ」と呼ばれる図形です.

[5] http://mathworld.wolfram.com/BarnsleysFern.html 参照

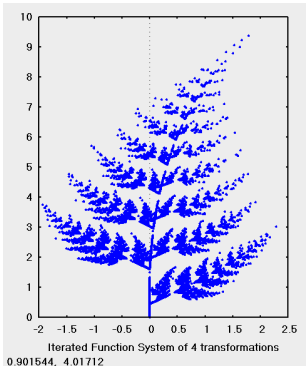


図 9.12: Barnsley のシダ

確率論的反復函数系 (Chaos game)

「chaos game」と呼ばれる手法は多角形と適当に選んだ多角形内部の点を用いて fractal を構成する手法です. 基本的に多角形の辺と多角形内部の点を適当に選んで点と辺の距離を求め, その点にある一定値を掛けて得られた点を加えて, 再度, この処理を繰り返すという処理を行います.
たとえば, 多角形を三頂点 (0,0), (1,1), (2,0) で構成された三角形とします. 次に多角形の内部の点を (1/2,1/2) とし, 点から辺への距離の倍数を 1/2 とすると図 9.13 に示す「Sierpinski の三角形」と呼ばれる図形を得ます:

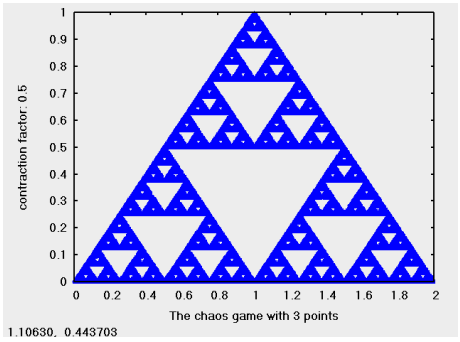


図 9.13: Sierpinski の三角形

確率論的反復函数系 (chaos game) の構成は反復函数系と同様です. ただし, 新たに確率が加わっています.

9.8.3　dynamics パッケージの概要

dynamics パッケージは 2 次元の離散力学系に対処可能なパッケージです. dynamics パッケージを利用することで容易に一次元及び二次元の力学系に関連したグラフが描けます. この中でも特に plotdf 函数は二次連立微分方程式の解曲線を対話処理を通じて綺麗に描くことができるので, 力学系のお勉強を行う上で非常に楽しいパッケージとなっています.
ただし, 専用のソフトウェアと比較すると, 機能的には非力で処理速度も遅いので, その辺は承知して使って下さい.
dynamics パッケージは Maxima のホームディレクトリ下の share/dynamics に収録されており, complex_dynamics.lisp,dynamics.mac と plotdf.lisp の 3 個のファイルで構成されています. パッケージ全体の Maxima への読込は load(”dynamics”); のように load 函数を使います.

9.8.4　dynamics.mac に含まれる函数

dynamics.mac は Maxima の処理言語で記述された函数で, 軌道等の点列を計算する函数を含んでいます. そして, その多くが Maxima を plot2d 函数を流用する函数のためにオプションが指定可能な函数で,plot2d 函数のオプションがそのまま使えます. 一方で, 軌道の総数は Maxima の基底にある Lisp のリストの上限に依存することに注意して下さい.

写像の軌道を描く函数

写像の軌道を描く函数

evolution(⟨ 函数 ⟩,⟨ 初期値 ⟩,⟨ 正整数 ⟩)
evolution(⟨ 函数 ⟩,⟨ 初期値 ⟩,⟨ 初期値 ⟩,⟨ 正整数 ⟩,[⟨plot_options⟩])
evolution2d([⟨ 函数$_1$⟩,⟨ 函数$_2$⟩],[⟨ 変数$_1$⟩,⟨ 変数$_2$⟩],
[⟨ 初期値$_1$⟩,⟨ 初期値$_2$⟩],⟨ 正整数 ⟩)
evolution2d([⟨ 函数$_1$⟩,⟨ 函数$_2$⟩],[⟨ 変数$_1$⟩,⟨ 変数$_2$⟩],
[⟨ 初期値$_1$⟩,⟨ 初期値$_2$⟩],⟨ 正整数 ⟩,[⟨plot_options⟩])

evolution 函数:　実 1 変数写像の自己反復による軌道を描く函数です. f を実 1 変数函数とする時, 初期値 x_0 に対して函数 f を 0 から n 回作用させることで得られる点列 $\left(i, f^i(x_0)\right)_{i=0\cdots n}$ の描画を行います.
ここで ⟨ 函数 ⟩ は 1 変数の函数式ですが, 直接式を引渡しても, Maxima 上で定義した式を含んでいても構いません. 次の引数 ⟨ 初期値 ⟩ を変数の初期値,⟨ 正整数 ⟩ で反復処理の回数を指定します.
この evolution 函数では計算した軌道を plot2d 函数で描画するために plot2d 函数の任意のオプションが指定できます.
ここで簡単な例として, 写像 f を $\exp(-x)\sin x$ とし, この軌道の初期値を 1 として軌道 $\{1, f(1), \cdots, f^{10}(1)\}$ を計算した例を図 9.14 に示します:

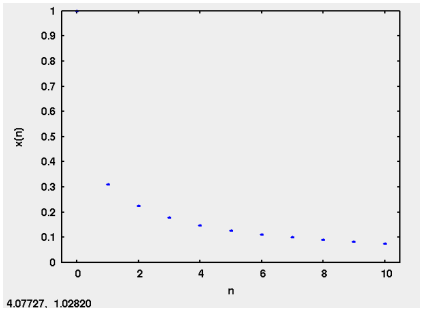


図 9.14: evolution(exp(-x)*sin(x),1,10)

evolution2d 函数: 二次元の離散的力学系で, 写像の自己反復による軌道を二次元グラフで表示する函数です. evolution 函数では $x_0 \in \mathbb{R}$ として点列 $\left(i, f^i(x_0)\right)_{1\leq i\leq n}$ を描きましたが, evolution2d 函数では平面上の点列 $x_0 \in \mathbb{R}^2$ に対し, 写像 $f : \mathbb{R}^2 \to \mathbb{R}^2$ による点列 $f^i(x_0)_{1\leq i\leq n}$ を描きます.
まず, 引数の ⟨ 函数$_1$⟩ と ⟨ 函数$_2$⟩ は夫々⟨ 変数$_1$⟩ と ⟨ 変数$_2$⟩ の実数函数で, f = [⟨ 函数$_1$⟩,⟨ 函数$_2$⟩] となります. 次に初期値は [⟨ 初期値$_1$⟩,⟨ 初期値$_2$⟩] で与え, ⟨ 正整数 ⟩ を自己反復の回数とします. すなわち, 軌道は初期値を含めて ⟨ 正整数 ⟩ + 1 個の平面上の点で構成され, evolution2d 函数はこの点列の表示を行う函数です.

ここで evolution2d 函数も plot2d 函数の任意のオプションが指定できます:

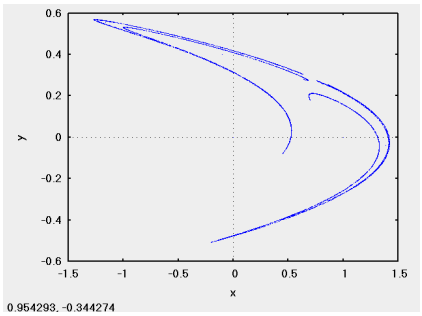


図 9.15: evolution2d([1+y-11/10*x^2,2/5*x],[x,y],[0,0],10000,[style,dots])

この図 9.15 では 1001 個の点で構成された strange attractor が表示されています.

軌道を考察する為の函数

軌道を考察する為の函数

staircase(⟨ 函数 ⟩,⟨ 初期値 ⟩,⟨ 整数値 ⟩,⟨ 値域$_1$⟩)
staircase(⟨ 函数 ⟩,⟨ 初期値 ⟩,⟨ 整数値 ⟩,⟨ 値域$_1$⟩,⟨ 値域$_2$⟩)
staircase(⟨ 函数 ⟩,⟨ 初期値 ⟩,⟨ 整数値 ⟩,⟨ 値域$_1$⟩,⟨plot_options⟩)
staircase(⟨ 函数 ⟩,⟨ 初期値 ⟩,⟨ 整数値 ⟩,⟨ 値域$_1$⟩,⟨ 値域$_2$⟩,⟨plot_options⟩)
orbits(⟨ 函数 ⟩,⟨ 初期値 ⟩,⟨ 正整数$_x$⟩,⟨ 正整数$_y$⟩,⟨ 領域$_1$⟩)
orbits(⟨ 函数 ⟩,⟨ 初期値 ⟩,⟨ 正整数$_x$⟩,⟨ 正整数$_y$⟩,⟨ 領域 ⟩,⟨plot_options⟩)
orbits(⟨ 函数 ⟩,⟨ 初期値 ⟩,⟨ 正整数$_x$⟩,⟨ 正整数$_y$⟩,⟨ 領域$_1$⟩,⟨ 領域$_2$⟩,
⟨plot_options⟩)

staircase 函数: 1 次元写像の軌道の解説図を描く函数です. 引数の ⟨ 函数 ⟩ は 1 変数函数の式, ⟨ 初期値 ⟩ は写像による軌道の初期値, ⟨ 正整数 ⟩ を n とすると, 軌道の集合を $\{f^i(x_0)\}_{0\leq i\leq n}$ とします. そして ⟨ 領域$_1$⟩ は解説図の横軸の区間, ⟨ 領域$_2$⟩ が縦軸の区間を定めます. ここでの領域の書式は閉区間 [⟨ 実数$_1$⟩,⟨ 実数$_2$⟩] を領域とする場合, [⟨ 変数 ⟩,⟨ 実数$_1$⟩,⟨ 実数$_2$⟩] の書式で指定します. この函数でも plot2d 函数が用いられており, plot_options で指定可能なオプションが与えられます. この plot_options の詳細は §11.2 を参照して下さい.
図 9.16 には軌道が一点に収束する様子を staircase(1.5*x*(1-x),0.5,20,[x,0.3,0.6],[y,0.2,0.6]) を実行することで得られた図的解説で示し, 図 9.17 には staircase(3*x*(1-x),0.1,10,[x,0,1],[y,0,1]) を実行することで軌道が二点の周期軌道に移行する様子を示しています:

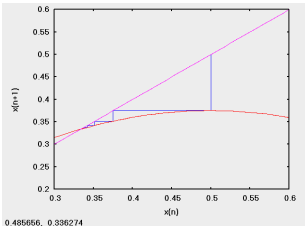


図 9.16: 収束の様子

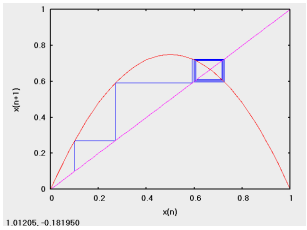


図 9.17: 周期軌道の様子

orbits 函数: ⟨ 函数 ⟩ で指定された 1 次の離散力学系の分岐図を描く函数です. 領域の指定方法は staircase 函数と同様です.
ここで正整数は ⟨ 正整数$_x$⟩ と ⟨ 正整数$_y$⟩ の二つありますが, ⟨ 正整数$_x$⟩ が横方向の点列数に対応し, ⟨ 正整数$_y$⟩ が縦方向の点列数に対応します.
この函数は軌道を描画しますが, 正直な所, あまり綺麗に描きません. その上, 軌道数を多くすると処理時間が非常に長くなるので注意が必要です. 図 9.18 には orbits(a*x*(1-x),1/2,100,1000,[a,0,4],[style,dots]), 図 9.19 には orbits(a*x*(1-x),1/2,200,1000,[a,2.8,4],[style,dots]) で描いた分岐図を示しておきます:

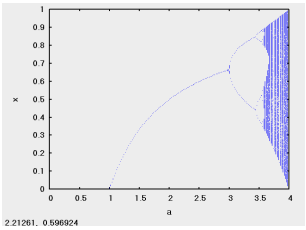


図 9.18: $ax(1-x), (0<a<4)$

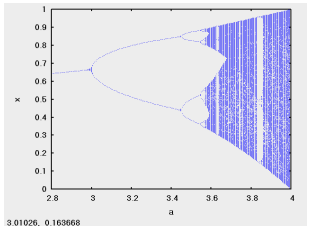


図 9.19: $ax(1-x), (2.8<a<4)$

fractal に関連する函数

dynamic.mac に含まれる函数
chaosgame([⟨ 点$_1$⟩,··· ,⟨ 点$_n$⟩],[⟨ 点$_0$⟩],⟨ 倍率 ⟩,⟨ 表示点数 ⟩)
chaosgame([⟨ 点$_1$⟩,··· ,⟨ 点$_n$⟩],[⟨ 点$_0$⟩],⟨ 倍率 ⟩,⟨ 表示点数 ⟩,[⟨plot_options⟩])
ifs([⟨ 重み$_1$⟩,··· ,⟨ 重み$_m$⟩],[⟨ 行列$_1$⟩,··· ,⟨ 行列$_m$⟩], [⟨ 点$_1$⟩,··· ,⟨ 点$_m$⟩]⟨ 点$_0$⟩],⟨ 表示点数 ⟩)
ifs([⟨ 重み$_1$⟩,··· ,⟨ 重み$_m$⟩],[⟨ 行列$_1$⟩,··· ,⟨ 行列$_m$⟩], [⟨ 点$_1$⟩,··· ,⟨ 点$_m$⟩]⟨ 点$_0$⟩],⟨ 表示点数 ⟩,[⟨plot_options⟩])

chaosgame 函数: 確率論的反復函数系 (chaos game) によって fractal を生成する函数です. これは ⟨ 点$_0$⟩ を初期値とする軌道を描く函数です.
まず, ⟨ 点$_i$⟩$_{0\le i\le m}$ は XY 平面上の多角形の頂点であり, その書式は [⟨X 座標 ⟩,⟨Y 座標 ⟩] となります. そして ⟨ 点$_0$⟩ が軌道の初期値になります. chaosgame 函数では random 函数を用い軌道 P_{orbit} と頂点 P_{poly} を選出し, 新しい軌道を $P_{poly}+(P_{orbit}-P_{polu})$⟨ 倍数 ⟩ で求めます. 以降, この処理を ⟨ 表示点数 ⟩ に到達するまで反復処理する函数です.
たとえば, 多角形を頂点 (0,1),(1,2),(2,1),(1,0) の菱形, 軌道の初期値を (1,1), 倍率を 0.4, 表示点数を 10000 とした場合を図 9.20 に示しておきましょう:

```
(%i16) chaosgame([[0,1],[1,2],[2,1],[1,0]],[1,1],0.4,10000);
```

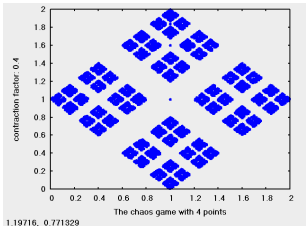


図 9.20: 家紋風

ifs 函数:　Michael Barnsley による反復函数系 (IFS:Iterated Function System) 法を用いて 2 次元のフラクタルを生成する函数です. ⟨ 重み$_1$⟩, $\cdots$,⟨ 重み$_m$⟩ は平面上の「吸収点」(attractor point) ⟨ 点$_1$⟩, $\cdots$,⟨ 点$_m$⟩ に対応する重みです. この重みは吸収点の確率に対応する値で, Maxima の乱数生成函数 random を用いて, この random の値が指定した重みよりも小さい場合に Affine 縮小写像を初期値に作用させ続けます. 次の ⟨ 行列$_1$⟩, *cdots*, ⟨ 行列$_m$⟩ は反復函数系を構成する二次の Affine 縮小写像を表現する 2 次の正方行列, そして, ⟨p_1⟩, *cdots*, ⟨p_m⟩ がこれらの行列に対応する吸収点です. すなわち, $a_i v + p_i$ が Affine 縮小写像に対応します. そして, ⟨ 点$_0$⟩ が初期値となる点です. ここで点の書式は [⟨X 座標 ⟩, ⟨Y 座標 ⟩] となります.

数値的に常微分方程式を解く函数

Runge-Kutta 法で微分方程式を解く rk 函数

rk(⟨ 常微分方程式 ⟩, ⟨ 変数 ⟩, ⟨ 初期値 ⟩, ⟨ 領域 ⟩)
rk([⟨ 常微分方程式$_1$⟩, $\cdots$, ⟨ 常微分方程式$_n$⟩], [⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩],
[⟨ 初期値$_1$⟩, $\cdots$, ⟨ 初期値$_n$⟩], ⟨ 領域 ⟩)

rk 函数:　4 次の Runge-Kutta 法を用いて常微分方程式を解く函数です. ここで扱える常微分方程式は次の形式に限定されます:

$$\frac{dx}{dt} = f(t, x)$$

ここで,t は領域の指定で用いる変数で, この領域は [⟨ 時間の変数 ⟩, ⟨ 開始時刻 ⟩, ⟨ 終了時刻 ⟩, ⟨ 時間刻幅 ⟩] の書式で与え, 上記の常微分方程式の変数 t に ⟨ 時間の変数 ⟩ が対応します. そして, x が ⟨ 変数 ⟩ に対応し, この x の領域で定めた開始時間の初期値に対応するものが ⟨ 初期値 ⟩ になります.
rk 函数は連立常微分方程式も扱えます. この場合, 連立常微分方程式は常微分方程式のリストとし, 初期値も変数リストに対応するリストで与えます.

9.8.5　complex_dynamics.lisp ファイルに含まれる函数

complex_dynamics.lisp には 1 次元の複素力学系で有名な Mandelbrot 集合や Julia 集合を描く函数が定義されています. これらの集合の計算では計算時間の短縮のために Maxima の処理言語ではなく, 直接 Lisp で函数を定義しています. なお, これらの函数は基本的に画像ファイルを出力し, 画面への出力は行いません.

dynamic.mac

mandelbrot()
mandelbrot(⟨ オプション ⟩)
julia(⟨x⟩, ⟨y⟩)
julia(⟨x⟩, ⟨y⟩, ⟨ オプション ⟩)

mandelbrot 函数:　Mandelbrot 集合を描く函数で, 図 9.21 に示すグラフを xpm 形式のファイルで出力する函数です.

julia 函数: Julia 集合を描く函数です. ただし, mandelbrot 函数とは違い, 引数として複素平面上の点を指定しなければなりません. この点は $\langle x\rangle + i\langle y\rangle$ で表現されます.

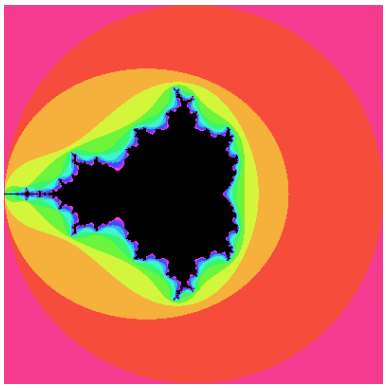


図 9.21: mandelbrot() による Mandelbrot 集合の描画

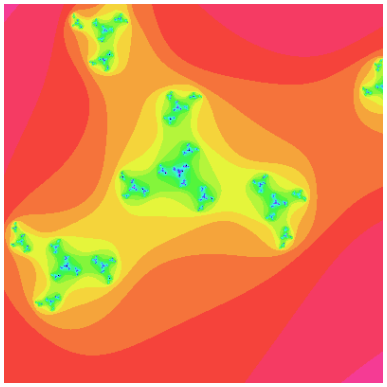


図 9.22: julia(0.1,-1,[center,0.3,0.6], [radius,0.5],[levels,24]) の結果

mandelbrot 函数と julia 函数のオプション: ここで mandelbort 函数と julia 函数に与えられるオプションを纏めておきましょう:

mandelbrot 函数と julia 函数のオプション

オプション名	既定値	概要
size	[size,400,400]	画像の大きさ (画素単位)
levels	[levels,12]	階調の個数 (正整数)
huerage	[huerange,360]	階調数 (正整数)
hue	[hue,-60]	階調の最低値 (正整数)
saturation	[saturation,0.76]	0 から 1 までの浮動小数点数
value	[value,0.96]	0 から 1 までの浮動小数点数
color	[color, "000000"]	
center	[center,0,0]	表示の中心点
radius	[radius,2.0]	表示する領域の指定 (中心点からの半径)
filename	[file,"mandelbrot.xpm"] [file,"julia.xpm"]	画像ファイル名 (後に.xpm が付く)

ここでオプションの書式は表の既定値に示すように [⟨ オプション名 ⟩,⟨ 値$_1$⟩$\cdots$,⟨ 値$_n$⟩] となります. ここで size,huerange,hue は整数値,saturation,value,center,radius には浮動小数点数を与えなければなりません.
hue, saturation, value は画像の rgb による階調表現を生成する内部函数 hsv2rgb で用いられる数値になります.

そして, filename の初期値は mandelbrot 函数では”mandelbrot.xpm”, julia 函数であれば”julia.xpm”となっています. ここでの値で修飾子 “xpm” を省略しても自動的にファイル名に付与されます.
なお, mandelbrot 函数と julia 函数の大きな弱点は計算が遅いことです.

9.8.6　二次の力学系を対話的に描画する函数

plotdf 函数:　二次元の力学系の解曲線等を対話的に描くことができる函数で, 対話処理のために GUI に TCL/TK を利用しています. plotdf 函数は plotdf.lisp で定義されており, この函数を利用するめに予め load(”plotdf”); で読込を実行しておく必要があります.
この plotdf 函数の構文を以下に纏めておきます:

plotdf 函数

plotdf(⟨ 常微分方程式 ⟩)
plotdf(⟨ 常微分方程式 ⟩,⟨ オプション ⟩)
plotdf(⟨ 常微分方程式 ⟩,[⟨ 変数$_1$⟩,⟨ 変数$_2$⟩])
plotdf(⟨ 常微分方程式 ⟩,[⟨ 変数$_1$⟩,⟨ 変数$_2$⟩],⟨ オプション ⟩)
plotdf([⟨ 常微分方程式$_1$⟩,⟨ 常微分方程式$_2$⟩])
plotdf([⟨ 常微分方程式$_1$⟩,⟨ 常微分方程式$_2$⟩],⟨ オプション ⟩)
plotdf([⟨ 常微分方程式$_1$⟩,⟨ 常微分方程式$_2$⟩],[⟨ 変数$_1$⟩,⟨ 変数$_2$⟩)
plotdf([⟨ 常微分方程式$_1$⟩,⟨ 常微分方程式$_2$⟩],[⟨ 変数$_1$⟩,⟨ 変数$_2$⟩,⟨ オプション ⟩)

ここで plotdf 函数で描画出来る常微分方程式は

1. $\frac{dy}{dx} = f(x,y)$
2. $\begin{cases} \frac{dx}{dt} = f(x,y) \\ \frac{dy}{dt} = g(x,y) \end{cases}$

の何れかであり, 1. の場合は $f(x,y)$ を, 2. の場合は $[f(x,y), g(x,y)]$ を plotdf 函数の第一引数に与えます. なお, ここで函数に x, y 以外の変数を用いる必要がある場合, 第二引数として第一引数で用いた函数の変数のリストを与えます. このときにリストの第一成分が XY 平面の X 座標, 第二成分が Y 座標となります.
たとえば, 図 9.23 と図 9.24 に変数リストを逆にした例を示しておきますが, X 座標と Y 座標が入れ替わっていることが容易に判ります:

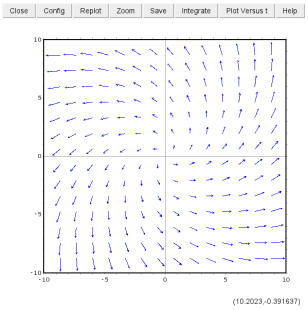


図 9.23: plotdf([u-v,v+u],[u,v])

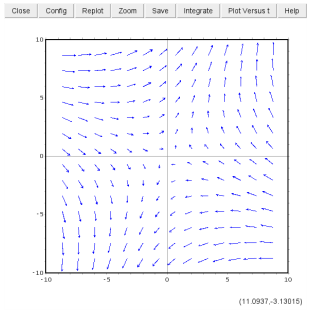


図 9.24: plotdf([u-v,v+u],[v,u])

plotdf 函数のオプション

plotdf 函数にはオプションを指定することができます. このオプションで様々な指定ができます. このオプションの書式は [⟨ オプション名 ⟩,⟨ 値 ⟩] ですが, 値の書式はオプション毎に異なります.

スライダーに関連するオプション

オプション	既定値	概要
parameters		媒介変数とその既定値を指定
sliders		媒介変数の値を変化させるスライダーを設定

オプション parameters: 常微分方程式を定める函数の媒介変数とその初期値を定めます. ここで定めた媒介変数は sliders オプションで自由に変更することができます. なお, parameter の値は文字列として与え, その書式は "⟨ 媒介変数$_1$⟩ = ⟨ 初期値$_1$⟩, . . . , ⟨ 媒介変数$_n$⟩ = ⟨ 初期値$_n$⟩" となります.

オプション sliders: 媒介変数を対話的に変更出来るスライダーを定義するオプションです. この定義によって描画ウィンドウの左下にスライダーが表示され, その値域も sliders で定義されます. ここで sliders の値の書式は "⟨ 媒介変数$_1$⟩ = ⟨ 下限$_1$⟩ : ⟨ 上限$_1$⟩, . . . , ⟨ 媒介変数$_n$⟩ = ⟨ 下限$_n$⟩ : ⟨ 上限$_n$⟩" になります.
たとえば, plotdf([u-m*v,v+k*u],[v,u],[parameters,"m=1,k=1"],[sliders,"m=1:5,k=1:5"]) を実行した場合に現われるウィンドウを図 9.25 に示しておきましょう:

領域指定に関連するオプション

オプション名	既定値	概要
xcenter	0	X 座標の中心点
ycenter	0	Y 座標の中心点
xradius	10	描画する X 軸方向の範囲
yradius	10	描画する Y 軸方向の範囲

オプション xcenter: 描画の中心点の X 座標を定めます.

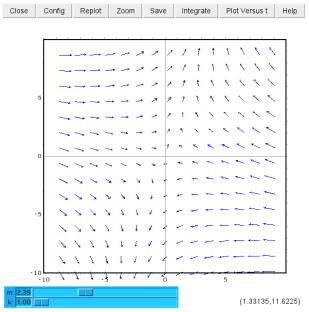


図 9.25: parameters オプションと sliders オプションの例

オプション ycenter:　描画の中心点の Y 座標を定めます.

オプション xradius:　実際に描画する X 座標の範囲を定めます. 描画範囲は `xcenter − xradius` から `xcenter + xradius` になります.

オプション yradius:　実際に描画する Y 座標の範囲を定めます. 描画範囲は `ycenter − yradius` から `ycenter + yradius` になります.

軌道に関連するオプション

オプション	既定値	概要
xfun	""	描画させる 1 変数函数を指定
direction	both	both,foward と backward が指定可能
trajectory_at		軌道の描画で軌道上の点を指定
tstep	0.1	解曲線の時刻の刻幅. 軌道上の点は nsteps で指定
nsteps	100	描画する軌道の点数
tinitial	0.	初期値の時刻を指定

オプション xfun:　xfun に軌道と一緒に描画させる変数 x の函数を文字列として指定出来ます. ここで複数の函数を指定しなければならない場合, 函数の区切としてセミコロン ";" を用います. たとえば [xfun,"sin(x);-sin(x)"] をオプションとして与えると $\sin x$ と $-\sin x$ のグラフも一緒に描かれます. なお, 常微分方程式の変数を x, y 以外に指定していても, xfun で与える函数は x の函数でなければなりません. たとえば, plotdf([u-v,v+u],[u,v],[xfun,"x;-x"]) の結果を図 9.26 に示しておきます:

オプション direction:　描画ウィンドウ常の点をマウスで指定した際に, その点を始点や終点, あるいは通過する軌道を描きます. both でその点を通過する全ての軌道を描き, foward でその点を始点とする軌道, backward でその点を終点とする軌道を描きます.

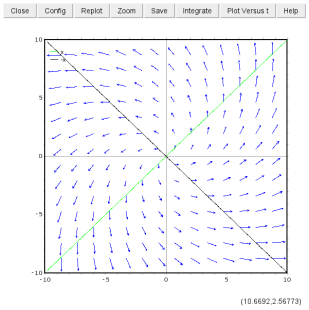


図 9.26: xfun オプションの例

オプション trajectory_at: 平面上の点を指定すると, その点を通過する軌道を描きます. なお, その軌道の方向は direction で指定した方向になります. plotdf 函数のウィンドウメニュー Config から呼出される変数設定ウィンドウで軌道上の点 (x, y) を指定する場合, x y の様に空行を入れます.

オプション tstep: 時間の刻幅を定めるオプションです. 軌道上の点は tstep の刻幅を nsteps で分割した点数になります.

オプション nsteps: 描画する軌道の点数を指定します.

オプション tinitial: 時刻 t の初期値を定めます. この値は軌道の計算で用いられます.

描画ウィンドウに関連するオプション

オプション	既定値	概要
height	400	縦の大きさ
width	400	幅の大きさ
versus_t	0	時刻歴による軌道の表示の有無

オプション height: 描画ウィンドウの縦方向の大きさを指定します.

オプション width: 描画ウィンドウの横方向の大きさを指定します.

オプション versus_t: この versus_t は plotdf 函数のウィンドウメニューの中にも含まれており, plotdf 函数から直接指定する場合には整数値を指定します. この versus_t は trajectory_at で指定した点を通る軌道に対して横軸を時間とした X と Y のグラフを別ウィンドウへの出力を指定ものです. そのためにウィンドウメニューから指定する場合には少なくとも一つの軌道が予め生成されていなければなりません.

9.8.7　幾つかの例題

dynamics パッケージを用いた幾つかの例題を示しておきましょう.

Malthus の人口モデル

産業革命時代のイギリスで Malthus は次の人口増加モデルを提唱しました.

- 人口は等比数列的に増加する
- 食料供給は等差数列的に増加する

最初の人口増加の方程式は時刻 t の人口を N とすると, ある正定数 a を用いて $\frac{dN}{dt} = aN$ と表現されることを意味します. それに対して食料供給は時刻 t の食料を F とすると, ある正定数 F_δ を使って $\frac{dF}{dt} = F_\delta$ となることを意味します.
ここで Malthus の人口モデルを evolute 函数を使って描いてみましょう. たとえば, 定数 a を 1.5 にし, 初期値の人口を 3.0e+7(三千万人, 江戸時代末期の日本のおおよその人口) として, 第 3 世代迄の人口を描いてみましょう. 1.5 にした理由ですが, 単純に, 一つの家庭から成人する子供が 3 人程度と仮定しただけです. ちなみに Malthus 自身は 3 人の子持ちだったようです.

```
evolution(1.5*x,3.e+7,3)
```

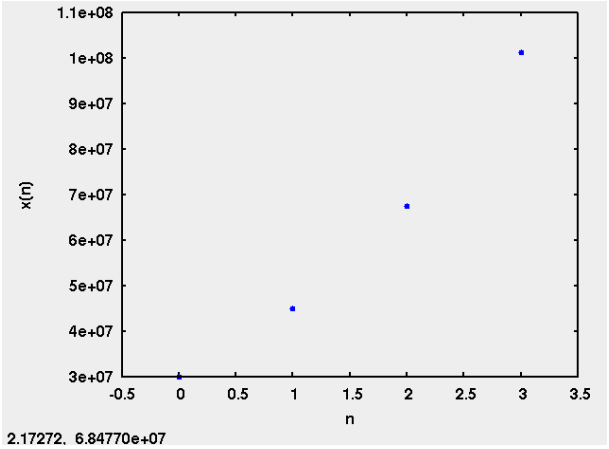


図 9.27: evolution(1.5*x,3.e+7,3) のグラフ

図 9.27 のグラフを見ると, 最終的には 1.1 億人近くになりました. 大体一世代で 30 年なので江戸末期から昭和末期の人口と言えますが, 現在の人口と比較してそんなに外れていないようです. 今度は江戸末期から 10 世代後 (大体 300 年後, 22 世紀) はどうなるでしょうか？

```
evolution(1.5*x,3.e+7,10)
```

図 9.28 を見ると今度は 18 億人を突破してしまいました! 食料供給から考えると通常の方法では到底間に合わないことは明確でしょう.
因に, desolve 函数で, この方程式を解いてみましょう.

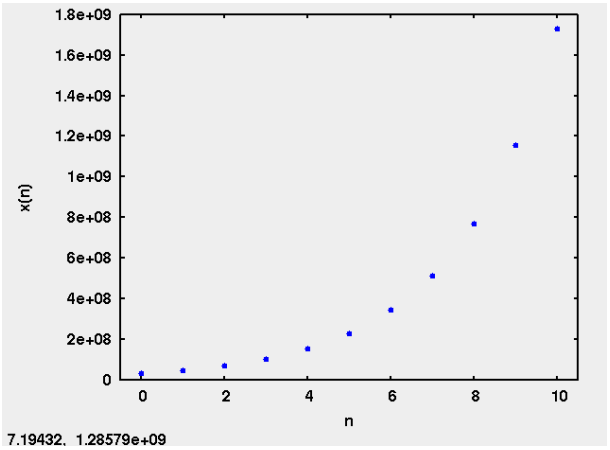


図 9.28: evolution(1.5*x,3.e+7,10) のグラフ

```
(%i54) atvalue(x(t),t=0,3.0e+7);
(%o54)                                3.0E+7
(%i55) desolve(['diff(x(t),t)=3/2*x(t)],[x(t)]);

'rat' replaced -3.0E+7 by -30000000/1 = -3.0E+7
                                                      3 t
                                                      ---
                                                       2
(%o55)                          x(t) = 30000000 %e
```

Malthus の人工モデルでは人口の増加はこのように指数函数になります. では, このような人口爆発が生じていない理由は何故でしょうか？ Malthus は戦争, 飢饉や貧困といった悪があるお陰だと主張しています. ただし, これらの悪が無くなれば人口を抑制するこれらの必要悪が無くなってしまう訳で, そんな場合には Malthus は禁欲的晩婚を主張しています. 無論, それで誰もが納得する訳ではありません. 19 世紀は社会的ダーウィニズムが主張される富国強兵, 弱肉強食の時代のために, このような禁欲的晩婚による人口の抑制の結果によって国力が相対的に落ちてしまえば途端に困る訳です. そこで帝国主義者は逆に植民地獲得によって人口問題を解決することを主張しています[6]. この辺は 20 世紀には生存圏 (Lebensraum) といった形でも見え隠れしたりします.

マメゾウムシの個体数モデル (ロジスティック方程式)

先程の Malthus の人口モデルは指数関数的な人口増大になっていました. しかし, 実際はそうではありません. たとえば, 現在の日本では少子化により人口が減少に転じている程です.
ここで動物の世界, 特に虫に関しては, 食料やその他の様々な問題によって, 個体数が或る一定の値に飽和する現象が見られます.

[6]出来る事なら惑星も併合したい - Cecil John Rhodes

その様な現象を表現するモデルの一つとしてロジスティック (兵站) 方程式と呼ばれる形の微分方程式があります.

$$\frac{du}{dt} = (A - ku)u$$

ここで,u が時刻 t における個体数で, A と k は正実数になります. この方程式は時間経過に伴なって個体数がある一定の飽和値に近づいて行く現象を表現したものです. ちなみに A/k が安定点となり, $u < A/k$ や $u > A/k$ の場合は A/k に漸近します.
このようなモデルとして面白いもので, マメゾウムシの個体数でも知られている方程式を紹介しましょう. マメゾウムシは子孫を残すと親が全て死んでしまうために世代交代が明確になります. そのために上述の微分方程式を離散化したものの方がより良く実際と一致するそうです. なお, 方程式は n 世代の個体数を u_n としたときに

$$u_{n+1} = u_n + (A - ku_n)u_n$$

になりますが, この方程式は非常に怪しい振舞をすることで知られています. つまり, パレメータによって個体数の振動が発生することです. この例はカオスの特徴を説明するものとして広く知られている現象です.

被食者と捕食者のモデル

捕食者 (例えば狼) と被食者 (例えば鹿) の個体数に関する方程式で有名なものに Volterra の微分方程式があります. この方程式は以下の連立微分方程式です.

$$\frac{dx}{dt} = (A - k_1 y)x$$
$$\frac{dy}{dt} = -(B - k_2 x)y$$

ここで被食者の個体数が x , 捕食者の個体数が y となっています. この式では被食者が増加すると, それを食べる捕食者も増加しますが, 捕食者が増え過ぎると今度は被食者が減ってしまうので結果として捕食者が減るといったことを表現したモデルです.
この方程式は一つの安定な解と周期解を持ちます. 安定な解は $\frac{dy}{dt}$ と $\frac{dx}{dt}$ の双方が 0 になる点で, この場合は, $(\frac{A}{k_1}, \frac{B}{k_2})$ が安定点, それ以外は周期解上の点になります.
では, このモデルを Maxima で表現してみましょう.

```
(%i34) load("plotdf")$
(%i35) plotdf([(1-2*y)*x,-(1-x)*y],[x,y],
[xcenter,2],[ycenter,2],[xradius,2],[yradius,2],
[nsteps,400],[trajectory_at,1,1],[versus_t,100])$
```

ここでは軌道を描くために trajectory_at で軌道上の点を指定しています. そして, 軌道上の点数は nsteps で 400 点に指定します. versus_t についてはあとで詳細を述べますが, こちらは適当な整数値で構いませんが, 実数値はエラーになります.
これを実行すると, 二つのウィンドウが出現します. 一つは図 9.29 に示す plotdf 本来のウィンドウで,Openmath を用いたものです.

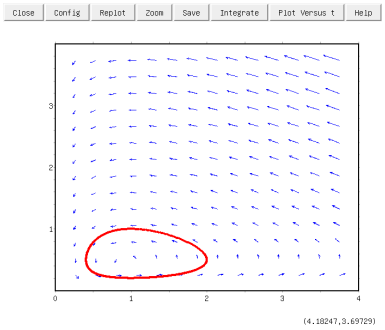


図 9.29: plotdf のグラフ (その 1)

もう一つは versus_t を指定したことで出現した X と Y の時刻歴での解曲線の表示を行う図 9.30 に示すウィンドウで, その名を X and Y versus_t というものです. こちらは plotdf のウィンドウで軌道を新規に表示させると自動的に更新されます.

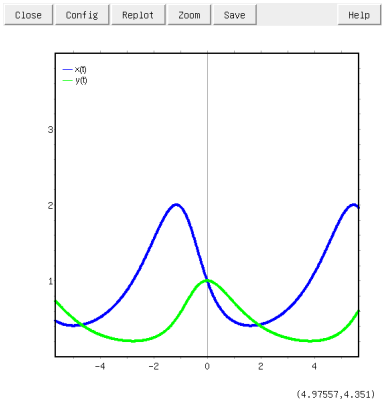


図 9.30: plotdf のグラフ (その 2)

図 9.29 で, 解曲線は赤く表示されているように見えますが, これは解曲線上の点の色です. 実際の解曲線の色は緑です. たとえば, Plotdf 上で (3,3) 付近をマウスで押してみて下さい. すると新規に軌道が表示され, その軌道は緑色, その上に赤い点が載っていることが判りますね.
この色を変更することもできます. これは,Plotdf の Config メニューを押して出てくる “plot setup ウィンド’ ウ” で adamsMoulton の値を変更すれば良いのです.

第10章　Maximaのシステム関連の函数

この章では Maxima のシステム関連の函数と虫取りに関連する函数について述べます.
Maxima は巨大なシステムであると同時に非常に古いシステムであるために, 旧弊なことも紛れ込んだ雑多な内容です.

10.1　計算結果の初期化

起動時の初期状態に戻す函数

reset 函数, reset_verbosely 函数と collapse 函数の構文

reset()
reset_verbosely()
collapse (⟨ 式 ⟩)
collapse ([⟨ 式$_1$⟩, . . . , ⟨ 式$_n$⟩])
collapse (listarray('⟨ 配列 ⟩))

reset 函数と reset_verbosely 函数:　内部的には同じ函数 (reset-do-the-work) 函数を用いる函数です. 共に引数が不要ですが, reset_verbosely 函数は初期化の処理内容が具体的に表示されます. これらの函数は Maxima の内部変数*variable-initial-values* に登録された変数と値のリストを用いて,Maxima を初期状態に戻す函数です. ちなみに大域変数等の設定で内部函数 defmvar を用いた場合, その大域変数と値のリストが内部変数*variable-initial-values*に登録されます.
なお, これらの函数によって内部変数*variable-initial-values*に保存された値に変数に束縛された値が戻るだけで, 何らのデータが削除される訳ではありません. そのために利用者が定義した変数とその値, 以前のラベルに保存されている値は残りますが, ラベルの方はカウンターが 1 に戻されているために処理を進めるにしたがい, 書きされてしまいます:

```
(%i2) display2d:false;
(%o2) false
(%i3) nolabels:true;
(%o3) true
(%i4) reset_verbosely();
reset: bind lispdisp to false
reset: bind display2d to true
reset: bind linenum to 1
reset: bind % to %
reset: bind __ to __
reset: bind _ to _
(%o1)              [lispdisp, display2d, linenum, %, __, _]
(%i2)%i3;
(%o2)                              nolabels : true
```

この例では reset_verbosely 函数による再設定の様子を示しています. 大域変数の値が元に戻されていますが, ラベルが残っていることが判ります.

collapse 函数:　全ての共通部分式を同じセルを用いて式が必要とする記憶容量を削減させる函数です.
この collapse 函数は optimize 函数でも用いられています. save 函数を用いて保存したファイルは Maxima の内部表現をそのまま含む式のために通常の入力式と比較して一般的に大きなファイルとなっています. そのため, ファイルの読込んだ後に,collapse 函数を用いると良いでしょう. さらに配列 a に対して 'collapse(listarray('a))' とすることで, 配列の無駄な成分を潰すこともできます.

10.2 処理の中断

10.2.1 制御文字による中断

Maxima の処理が異常に長い場合や, 間違って計算を実行させた場合, Maxima を一旦中断する必要が生じます. Maxima の計算を中断したければ, 通常は制御文字 "^c(Ctrl+C)" を使います. Maxima は制御文字 "^z(Ctrl+Z)" が入力されても中断しますが, この場合は Maxima を出て, UNIX の shell に戻るので通常は制御文字 "Ctrl+C" を用います:

```
(%i11) factor(2137498127943870982374);
Maxima encountered a Lisp error:

 EXT:GC: User break

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i12)
```

この例は制御文字 "Ctrl+C" で因数分解を中断させています.

10.2.2 函数による意図的な中断

break 函数: Maxima では break 函数を用いてプログラムを意図的に中断させる事が可能です:

break 函数

break(⟨ 式$_1$⟩,...,⟨ 式$_n$⟩)

break 函数の返却値は ⟨ 式$_n$⟩ の処理結果となります. break 函数は引数の全ての式を処理した後に Maxima-break に入ります. Maxima-break ではプロンプトとして大域変数 prompt に割当てた文字列が表示されます. この Maxima-break では通常の函数が利用可能ですが, ev 函数の省略型やラベルを用いた処理が行えません. Maxima-break から抜けるためには exit; と入力します:

```
(%i17) break(integrate(sin(x)*x,x),factor(x^2-1));
sin(x) - x cos(x) (x - 1) (x + 1)

Entering a Maxima break point. Type exit; to resume
_diff(sin(x)-x*cos(x),x);
x sin(x)
_expand((x-1)*(x+1));
 2
x  - 1
_exit;
(%o17)                          (x - 1) (x + 1)
(%i18)
```

この例では, $\int xsin(x)dx$ と x^2-1 の因子分解を実行したあとに Maxima-break に入っています. Maxima-break で出力されるプロンプトの既定値は記号 "_" です. Maxima-break から抜けるためには exit; と入力します. すると, break 函数は最後に処理した式の結果を返却値として返します.

10.3　結果の表示

Maxima は入力行に文字 ";" を付けると,Maxima が評価した値を表示します. 数値の四則演算は入力すると直ちに処理を行い, その結果を表示します. さらに, Maxima は結果を二次元的に表示するのが初期設定となっています:

```
(%i43) 2/5;
                                  2
(%o43)                            -
                                  5
(%i44) integrate(f(x),x,a,b);
                                   b
                                  /
                                  [
(%o44)                            I   f(x) dx
                                  ]
                                  /
                                   a
(%i45) expand((x+1)*(x-1));
                                    2
(%o45)                             x  - 1
```

この表示は式が小さなものであれば味のある良いものですが, 式が長くなると非常に判り難くなります. そこで, 表示を 1 行で済むように指定ができる大域変数 display2d があります. この変数の値が既定値の 'true' であれば二次元的な表示を行い, 'false' を指定した場合に結果を一行で, そのまま入力として利用可能な表示で返します:

```
(%i4) display2d;
(%o4)                              true
(%i5) 'integrate(f(x),x);
                                  /
                                  [
(%o5)                             I f(x) dx
                                  ]
                                  /
(%i6) expand((x+1)^3);
                                3       2
(%o6)                          x  + 3 x  + 3 x + 1
(%i7) display2d:false;
(%o7) false
(%i8) 'integrate(f(x),x);
(%o8) 'integrate(f(x),x)
(%i9) expand((x+1)^3);
(%o9) x^3+3*x^2+3*x+1
```

Maxima にはこの他にも特殊な表示を行う函数があります. 基本的に Maxima はキャラクター端末しかなかった昔の手順に対応したシステムのため, 積分記号のように文字を使って数式を表示するといった, ある意味では涙ぐましい努力の跡があります. しかし, 現在の Window システムから見ると非常に古臭く感じるものが多いのが現状です.

10.3.1 表示に関連する大域変数

表示に関連する大域変数

変数名	既定値	概要
ibase	10	入力数値の基数を指定
obase	10	出力数値の基数を指定
absboxchar	!	絶対値を描く際に用いる文字を指定
display2d	true	結果の二次元表示の有無
leftjust	false	結果の左寄表示の有無
display_format_internal	false	内部表現に対応した表示切替
lispdisp	false	LISP の記号表示に於ける?の有無
%edispflag	false	%e の冪の表示を指定
exptdispflag	true	負の冪を分数で表示
pfeformat	false	有理数の表示書式を決定
powerdisp	false	多項式の項の表示順を決定
stardisp	false	可換積演算子の表示を指定
linel	79	一行に表示する文字数
stringdisp	false	文字列表示で二重引用符の表示の有無
noundisp	false	名詞型表示での単引用符の表示の有無
ttyoff	false	出力の停止制御

大域変数 ibase: 入力数値の基数を指定します. 値は十進数で 2 から 35 が指定可能です. '10' よりも大きな値を設定した場合, "A" から開始する大文字のアルファベットを用います. なお, 大域変数 obase は表示の際に用いる数値の基数を指定する大域変数となります.

大域変数 absboxchar: 絶対値を描く際に用いる文字を指定します. なお, 絶対値の記号は一行の高さしかありません.

大域変数 display2d: 'false' であれば結果表示が二次元的書式ではなく, 一行に収まるように表示します. 長い式や複雑な式や表示に余裕がない場合, あるいは結果を入力に再利用したい場合は特に有効です.

大域変数 leftjust: 結果の表示の左寄表示の有無を設定します. 既定値の 'false' で結果は中央に揃えられますが, 大域変数 leftjust の値が 'true' であれば結果が左寄で表示されます.

大域変数 display_format_internal: 'true' であれば, 内部の数学的表現を隠した表示ではなく内部表現を反映した表示に切替ります. ただし, 内部表現そのもので表示する訳ではありません. ここでの出力は part 函数 に対応するののではなく, inpart 函数 に準じたものになります. ここで part 函数と inpart 函数による式の表現な違いを比較した表を示しておきましょう:

内部書式との比較

入力	part	inpart
a-b;	a-b	a+(- 1)*b
a/b;	a/b	a*bˆ(-1)
sqrt(x);	sqrt(x)	xˆ(1/2)
x*4/3;	4*x/3	4/3*x

大域変数 lispdisp: false の場合に LISP の記号表示で?を省略します.

%edispflag: Napia 数 '%e' の冪表示を定める大域変数です. 既定値の 'false' のとき, '%e' の負の冪は負の冪のまま冪表示されます. たとえば, 'exp(-x)' は '%e' の負の冪'%eˆ(-x))' で表示されます. 'true' の場合, '%e' の負の冪は '%e' の冪の商の書式で表示されます. すなわち, '%eˆ(-x)' は '1/%eˆx' の書式で表示されます:

```
(%i2) exp(-x);
                                    - x
(%o2)                              %e
(%i3) %edispflag:true;
(%o3)                               true
(%i4) exp(-x);
                                     1
(%o4)                                --
                                     x
                                   %e
```

大域変数 exptdispflag: 'true' であれば負の冪を持った項を分数式で表示します. たとえば, 'xˆ(-1)' は '1/x' と表示されます.

大域変数 pfeformat: 'true' であれば有理数は行の中で表示され, 整数の分母は有理数の積として表示されます. たとえば, 入力が 'b/4' であれば '1/4*b' と表示されます. ここで 'a/b' のように変数 a,b の両方が自由変数であれば通常の表示になります.

大域変数 powerdisp: 多項式の冪の表示順序を切換える大域変数です. 既定値が 'false' の場合に項順序 "$>_m$" の大きなものから順に表示しますが, 'true' であれば項順序 "$>_m$" の小さなものから順に表示されます. なお, この変数は taylor 級数には影響を与えません:

```
(%i1) powerdisp;
(%o1)                                false
(%i2) expand((x+y)^3+x+y);
                          3         2       2         3
(%o2)                    y  + 3 x y  + 3 x  y + y + x  + x
(%i3) powerdisp:true;
(%o3)                                true
(%i4) expand((x+y)^3+x+y);
```

(%o4) $x + x^3 + y + 3x^2y + 3xy^2 + y^3$

大域変数 stardisp: ‘false’ であれば可換積演算子 “*” は出力で省略されています. ‘true で’ あれば可換積演算子を表示します.

大域変数 linel: 一行に表示される文字数を設定します. 変更はいつでもできます.

大域変数 stringdisp: ‘false’ であれば文字列の二重引用符を表示しません. ‘true’ であれば二重引用符を含めて文字列の表示を行います.

大域変数 noundisp: ‘false’ の場合, Maxima の名詞型の対象の表示で単引用符を省略します. ‘true’ の場合は単引用符を含めて表示を行います.

大域変数 ttyoff: ‘true’ であれば入力行の表示のみを行い, 通常の出力を止めます. ただし, エラー表示のみは行います. なお, writefile 函数で開いたファイルに対しても, 同じ出力になります.

10.3.2 式の表示を行う函数

式の値の表示を行う函数

式の値の表示を行う函数
display(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, . . .)
disp(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, . . .)
ldisplay(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, . . .
ldisp(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, . . .
print(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, . . .)
grind(⟨ 式 ⟩)

display 函数: 式の表示を行う函数です. その左側が未評価の ⟨ 式$_i$⟩ で, その右側の行の中心がその式の値となります. この函数は block や do 文で, 中途結果の表示を行うのに便利です. display の引数は通常, 原子, 添字された変数や函数呼出しになります.

disp 函数: display 函数に似た函数ですが引数の値のみを表示します.

ldisplay 函数と ldisp 函数: display 函数と disp 函数に似ていますが, 中間ラベルを生成する点で異なります:

```
(%i18) a1[1,2]:128;
(%o18)                                128
(%i19) display(a1[1,2]);
                                  a1      = 128
                                    1, 2

(%o19)                               done
(%i20) disp(a1[1,2]);
                                     128

(%o20)                               done
(%i21) ldisplay(a1[1,2]);
(%t21)                            a1      = 128
                                    1, 2

(%o21)                              [%t22]
(%i22) ldisp(a1[1,2]);
(%t22)                               128

(%o22)                              [%t23]
```

print 函数: ⟨ 式$_1$⟩ がら順番に評価を実行して, その結果を表示します. ここで, ⟨ 式$_i$⟩ に含まれる原子や函数の前に単引用符 “’” が置かれていたり, 文字列の場合は評価を行わずにそのまま表示を行います.

grind 函数: ,⟨ 式 ⟩ を Maxima の入力に適した形式で表示します. grind 函数の返却値は常に done です.

式の内部表現に関連した表示を行う函数

式の内部表現に関連した表示を行う函数

dispterms (⟨ 式 ⟩)
reveal (⟨ 式 ⟩, ⟨ 深度 ⟩)
tcl_output (⟨ リスト ⟩, ⟨ 添字 ⟩, ⟨ 飛幅 ⟩)
tcl_output (⟨ リスト ⟩, ⟨ 添字 ⟩)

dispterms 函数: 与式を内部表現に合せて式の表示を行います.

reveal 函数: ⟨ 整数 ⟩ で指定された成分の長さで ⟨ 式 ⟩ を表示します. 和は sum(n), 積は product(n) として表示されます.
ここで n は和や積の成分の数になります. 指数函数は expt で表現されます:

```
(%i11) aa:integrate(1/(x^3+2),x);
                                                    1/3
                                              2 x - 2
                                         atan(------------)
                2     1/3       2/3            1/3                      1/3
          log(x  - 2     x + 2    )          2     sqrt(3)     log(x + 2    )
(%o11)  - ----------------------- + ------------------ + -------------
                     2/3                   2/3                     2/3
                  6 2                    2     sqrt(3)          3 2
(%i12) reveal(aa,1);
(%o12)                                 sum(3)
(%i13) reveal(aa,2);
(%o13)                     negterm + quotient + quotient
(%i14) reveal(aa,3);
                                          atan          log
(%o14)                 - quotient + ---------- + ----------
                                      product(2)    product(2)
(%i15) reveal(aa,4);
                          log        atan(quotient)    log(sum(2))
(%o15)               - ---------- + -------------- + -----------
                        product(2)      expt sqrt         3 expt
(%i16) reveal(aa,5);
                                          sum(2)
                                   atan(----------)
                   log(sum(3))          product(2)     log(x + expt)
(%o16)           - ----------- + ---------------- + -------------
                     6 expt            2/3                   2/3
                                     2     sqrt(3)          3 2
```

tcl_output 函数: 〈添字〉を展開した〈リスト〉に対応する tcl のリストを表示します. ここで, 刻幅の規定値は '2' で, 引数がリストで構成されたリストではなく, 数値リスト形式の場合は刻幅から外れた全ての要素が表示されます.

結果の再表示を行う函数

playback 函数: 処理結果の再表示を行う函数です. この函数は単純に結果の再表示を行うだけで再計算は行いません:

結果の再表示を行う函数 playback

playback (〈整数〉)
playback ([〈整数$_1$〉,〈整数$_2$〉])
playback ([〈整数〉])
playback ()
playback(input)
playback(slow)
playback(time)
playback(grind)

引数が整数 n であれば, 最近の n 個の式 (入力行%, 出力行%o と中間行%e をそれぞれ 1 個として数えます) を再表示します.
ここで, 引数がリスト [⟨ 整数$_1$⟩,⟨ 整数$_2$⟩] の場合, ⟨ 整数$_1$⟩ から ⟨ 整数$_2$⟩ までの全ての行を再実行します. ⟨ 整数$_1$ = 整数$_2$⟩ であれば, 引数として [⟨ 整数 ⟩] を指定します.
引数が指定されない場合には全ての行が再表示されます.
playback 函数のオプションとして input, slow, time, grind があります:

引数 input:　入力行を再表示行します.

引数 slow:　example 函数によるデモの処理と同様に一つの入力とそれに対応する処理結果を表示すると Maxima-break に入り, Enter キーの入力待ちになります. また, Enter キー以外のキーが入力されると playback 函数を終了します. なお, Maxima-break に入った時点で表示されるプロンプトは大域変数 prompt で設定されています.

引数 time:　計算時間が式と同様に表示されます. ここで大域変数 gctime か totaltime であれば, showtime:all; を用いたのと同様に, 計算時間の詳細が表示されます.

引数 string:　全ての入力行の再表示を文字列として返します.

引数 grind:　再入力が可能な式を出力する grind モードで式の再表示を行います.
なお, playback 函数による行の再表示は, playback([2,5],10,time,grind) のように複数の引数の組合ても構いません.

古風な表示を行う函数

Maxima は歴史のあるシステムです. そのために古風な表示を行う函数が幾つかあります. ここでは古風な函数を愛でることにしましょう:

古風な表示を行う函数

dpart(⟨ 式 ⟩,⟨ 整数$_1$⟩,...,⟨ 整数$_n$⟩)
lpart(⟨ 文字列 ⟩,⟨ 式 ⟩,⟨ 整数$_1$⟩,...,⟨ 整数$_n$⟩)
box(⟨ 式 ⟩)
box(⟨ 式 ⟩,⟨ 文字列 ⟩)
rembox(⟨ 式 ⟩)
rembox(⟨ 式 ⟩,unlabelled)
rembox(⟨ 式 ⟩,⟨ ラベル ⟩)

dpart 函数:　与えられた式の指定された階層を大域変数 boxchar に割当てられた文字を用いて囲って表示する函数です. 昔のテキスト主流の時代では必要な機能だったのでしょうが, 現在では, ややキワモノ的な機能です.

lpart 函数: dpart 函数とほぼ同じ機能の函数です. ただし, lpart 函数は第 1 引数に文字列を指定し, これをラベルとして使えます. 勿論, 第 1 引数を""のように空白文字を指定すれば dpart 函数によるものと同じ結果が得られます:

```
(%i188) dpart(int(f(x),x),0);
                                 """""""""""""""
(%o188)                          "int(f(x), x)"
                                 """""""""""""""
(%i189) lpart("結構歪だね",int(f(x),x),0);
                                 "結構歪だね"""""""""
(%o189)                          "int(f(x), x)"
                                 """""""""""""""
```

box 函数: 式の最上層のみを大域変数 boxchar で指定した文字で囲う函数です l:

```
(%i195)  box(expand((x+1)^3));
                               """""""""""""""""""""""
                               " 3       2           "
(%o195)                        "x  + 3 x  + 3 x + 1"
                               """""""""""""""""""""""
(%i196)  box(expand((x+1)^3),"朝日輝く金亀山");
                               "朝日輝く金亀山"""""""""""""""
                               " 3       2           "
(%o196)                        "x  + 3 x  + 3 x + 1"
                               """""""""""""""""""""""
```

rembox 函数: dpart 函数, lpart 函数や box 函数で生成した ASCII ART(AA) から不要な箱を削除する函数です.

古風な表示を行う函数に関連する大域変数 boxchar

大域変数	既定値	概要
boxchar	"	AA で用いる文字

大域変数 boxchar: dpart 函数等で行う箱表示の際に用いる文字を指定する大域変数です:

```
(%i178) dpart(int(f(x),x),1,1);
                                      """"
(%o178)                          int(f("x"), x)
                                      """"
(%i179) boxchar:"漢"$
(%i180) dpart(1/(ＶＩＰ+1),2,1);
                                        1
(%o180)                           ---------
                                  漢漢漢漢漢
                                  漢ＶＩＰ漢 + 1
                                  漢漢漢漢漢
```

なお, 本来の値に戻したければ,`boxchar:"\""`と入力します.

10.3.3 エラー表示

エラー表示の函数に関連する函数の構文

error(⟨ 式$_1$⟩,...,⟨ 式$_n$⟩)
errormsg()

error 函数: 引数を評価した結果を文字列に変換し, これらの文字列を成分とするリストを大域変数 error に割当てて, その値を表示する函数です.

errormsg 函数: 大域変数 error に割当てた文字列を表示する函数です. 猶, 大域変数 errormsg は error 函数による大域変数 error に割当てた文字列の表示を行うかどうかを指定する大域変数です:

```
(%i27) error("メッ！");
メッ！
 -- an error.  To debug this try debugmode(true);
(%i28) error;
(%o28)                                [メッ！]
(%i29) errormsg();
メッ！
(%o29)                                done
(%i30) errormsg:false;
(%o30)                                false
(%i31) errormsg();
メッ！
(%o31)                                done
(%i32) error("メッ！");
 -- an error.  To debug this try debugmode(true);
```

このように error 函数は大域変数 errormsg の影響を受けますが, errormsg 函数は大域変数 errormsg の影響を受けません.

エラー表示に関連する大域変数

変数名	既定値	概要
error	[No Error.]	エラーメッセージが割当てられる大域変数
errormsg	true	エラーメッセージの表示を制御
error_size	10	エラーメッセージの大きさを制御
error_syms	[errexp1,errexp2,errexp3]	エラーメッセージ

大域変数 error: error 函数で割当てられたエラーメッセージが登録される大域変数です.

大域変数 errormsg: error 函数で大域変数 error に割当てた文字列を表示させるかどうかを指示する大域変数です. 規定値の 'true' であれば表示を行い, 'false' であれば表示を行いません.

大域変数 error_size: エラーメッセージの大きさを指定します. 大域変数 error_size よりも大きな式は文字列に置換され, 文字列には式が設定されています. 文字列は利用者が設定可能なリストから取られます.

大域変数 error_syms: エラーメッセージで大域変数 error_size よりも大きな式は文字列に置換され, その文字列には式が設定されています. 文字列は大域変数 error_syms に割当てられたリストから取られて既定値は 'errexp1', 'errexp2', 'errexp3' 等々となっています. エラーメッセージが表示されたあとで, たとえば, "the function foo doesn't like errexp1 as input." であれば, errexp1; と利用者が入力すると, その式を見ることができます. この大域変数 error_syms には必要ならば別の文字列を設定しても構いません.

10.4 ヘルプに関連する函数

Maxima にはオンラインマニュアルを持っています. このオンラインマニュアルを閲覧するために describe 函数, 記号 "?" や記号 "??" を用います. さらにデモや例題を実行させることもできます,

オンラインマニュアルに関連する函数の構文

describe(⟨ 事項 ⟩)
demo(⟨ ファイル名 ⟩)
example(⟨ 項目 ⟩) example()

describe 函数: オンラインマニュアルを表示させる為の函数です. 引数としては函数名, 演算子名, 大域変数名等になります. 同様のことは演算子 "?" でもできます. なお, 函数名や大域変数名があやふやなときは演算子 "??" を用いた方が良いでしょう.
この describe 函数の仕組みについては §13 で describe 函数の改造の話を書いているので, そこを参照して下さい.

demo 函数: 指定された函数のデモファイルを自動実行する函数です. このデモファイルの構造は通常のバッチファイルと同様の構造, すなわち, 入力と同じ書式になります.
demo 函数でデモファイルを実行すると, デモファイルに記述された Maxima の入力行を処理するたびにプロンプト "_" を出力して処理を止めてキーの入力待ちになります. 通常は, セミコロン ";" や "Enter" キーを入力すると次に進みます.
デモファイルは何処に置いても構いませんが, 大域変数 file_search_demo に割り当てられたリストに登録されていないディレクトリに置いた場合, そのファイルへの経路とファイル名を記述した文字列を demo 函数に与えなければなりません.
もし, 函数名のみを与えたいのであれば, 大域変数 file_search_demo に登録されたディレクトリ上に置く必要があります. ここでファイルの修飾子としては dem, dm1, dm2, dm3, dmt が利用できます. ここでデモファイルは全ての函数やパッケージに用意されているとは限りません. ここで大域変数 file_search_demo にはデモファイルを検出するための経路や修飾子が記述された LISP の文

字列で構成されたリストが割り当てられています. この LISP の文字列の書式は, '経路/###.{拡張子$_1$,...,拡張子$_n$}' のような書式です. この実例を次に示しておきましょう:

```
/usr/local/share/maxima/5.13.0/demo/###.{dem,dm1,dm2,dm3,dmt}
```

したがって自前のデモファイル群を置いたディレクトリを指定したければ, この書式の LISP の文字列を file_search_demo に割り当てられたリストに追加しなければなりません.

example 函数:　指定された函数に対応する例題を返す函数です. example 函数で実行される例題は例題ファイルに登録されたもので, 函数名を文字列ではなく記号として example 函数に与えます. 具体的には, algsys 函数の例題を見たければ, `example(algsys);` と入力し, 文字列 "algsys" を引数にはしません. もしも, 対応する函数が存在しない場合,example 函数に引数を与えなかった場合, example 函数は実行可能な例題の一覧をリストで返します.
この example 函数の例題ファイル名は通常 maxima.demo ファイルであり, このファイルは大域変数 manual_demo に割り当てられています. そして, このファイルの所在は LISP 側の内部変数*maxima-demodir* に束縛されたディレクトリ名になります.
さらに example 函数向けの例題ファイルの構造は demo ファイルとやや勝手が違うものです. そこで, このファイルの構造を次に示しておきます:

example 函数向けの例題ファイルの構造

ラベル$_1$	⟨Maxima の入力行$_1$⟩	
	⟨Maxima の入力行$_2$⟩	
	⋮	
	⟨Maxima の入力行$_n$⟩	&&
/*		
	註釈行$_1$	
	⋮	
	註釈行$_m$	
*/		&&
ラベル$_2$	⟨Maxima の入力行$_1$⟩	
	⟨Maxima の入力行$_2$⟩	
	⋮	
	⟨Maxima の入力行$_n$⟩	&&

このように例題ファイルはラベルで開始する項目と注釈として開始する項目の二種類がありますが, 何れも項目の末尾には "&&" を置きます. この "&&" を省略すると, その項目が終了していないと判断されますが, 題ファイルの末尾の項目に対しては "&&" を省略しても問題はありません.
例題ファイルの注釈は C 風に/*と*/で括りますが, この注釈を終えると必ず末尾には "&&" を入れます. ここでの Maxima の入力行とは通常の Maxima の文で行末に ";" や "$" を入れたものです.
example 函数は例題ファイルで引数に合致するラベルを捜し, 適合するラベルがあれば, そのラベ

ルから “&&” が現われるまでの Maxima の入力行を処理します. ラベルは函数名である必要はありませんが, 出鱈目なものにすると後で困るので, 函数名や話題に沿ったものにすると良いでしょう.

オンラインマニュアル等に関連する大域変数

file_search_demo	demo 函数向けのデモファイルの収納先を指定する大域変数
manual_demo	example 函数向けの例題ファイル名を指定する大域変数
help	help 函数向けの文字列が割り当てられた大域変数

大域変数 file_search_demo: demo 函数で用いるデモファイルが登録されているディレクトリ等がリストとして登録されています. なお, この大域変数は LISP の文字列で構成された Maxima のリストが割り当てられています.

manual_demo: example 函数で用いる例題が登録されたファイル名が割り当てられています. 既定値として manual.demo が割り当てられています.

大域変数 help: この大域変数には次の LISP の文字列が既定値として設定されています:

```
type ‘describe(topic);’ or ‘example(topic);’ or ‘? topic’
```

なお, 函数の help もありますが, 大域変数 help に割り当てられた上記の文字列を表示するための函数です.

演算子”?”

Maxima の演算子 “?” は変った性質を持っています. 函数としてはオンラインマニュアルを表示する演算子の様に振舞いますが, 特殊記号としては裏の LISP に “?” の直後の文字列を流込む働きを行います.

演算子 “?” と演算子 “??”

演算子	構文	概要
?	? 〈事項〉	〈事項〉に関連するオンラインマニュアルを表示
?	?〈LISP に評価させる函数〉	LISP の函数を Maxima の函数風に処理
??	?? 〈事項〉	〈事項〉に関連するオンラインマニュアルの一覧を表示

演算子 “?” は Maxima で二つの意味を持ちます. 一つはオンラインマニュアルの表示で用います. この場合, “?” との間に空白文字や tab を入れて調べたい〈事項〉を入力します. 〈事項〉に関連す

る項目が複数存在する場合, Maxima は一覧表を表示して事項に対応する数値か none が入力されるのを待ちます.
数値が入力されると該当するヘルプを表示して終了し, none の場合は何もせずにそのまま終了します. なお, 返却値は false になります.

```
(%i5) ? prefix;

 0: (maxima.info)prefix.
 1: optimprefix :Definitions for Expressions.
Enter space-separated numbers, `all' or `none': 0

Info from file /usr/local/info/maxima.info:
5.5 prefix
==========

A `prefix' operator is one which signifies a function of one argument,
which argument immediately follows an occurrence of the operator.
`prefix("x")' is a syntax extension function to declare x to be a
`prefix' operator.

   See also `syntax'.

(%o5)                                false
```

もう一つの記号 “?” の働きは引数の間に空白を空けずに利用した場合に, 記号 “?” の直後の引数を LISP の函数として評価して結果を返そうとします. この場合, 記号 “?” の引数に LISP に評価させる函数を記述しますが, ここでは LISP の S 式ではなく Maxima の函数風に表記したものを与えます. そして, 与える引数も Maxima の表の表記にする必要があります. たとえば, Maxima の変数 x は内部で ‘$x’ と表記されています. その変数 x に割当てられた式の car を取出す場合は ‘?car(x);’ と入力し, ‘?(car $x)’ とはしません. ここで, この函数を実行したときの返却値は全て Maxima の与件型に変換されて返されます. ここで LISP の函数で式の評価をする演算子に演算子 “:lisp” もあります. こちらは LISP の S 式を引数に取り, 返却値も LISP の書式になりますが, 演算子 “?” と違ってラベルには保存されません:

```
(%i14) a1:x^2+y+z;
                                  2
(%o14)                          z + y + x
(%i15) ?car(a1);
(%o15)                          ("+", simp)
(%i16) :lisp (car $a1)
(MPLUS SIMP)
```

Maxima には演算子 “??” もあり, オンラインマニュアルを表示する演算子 “?” と似た働きをします. この演算子 “??” は与えられた事項に関連するオンラインマニュアルの一覧を表示して入力を待つものです.
この演算子 “??” と演算子 “?” の違いは, 事項を指定する必要がないことです. 一種のキーワード検索として用いることもできます. たとえば, ‘?? inte’ と入力した結果を示しておきます:

```
(%i91) ?? inte
```

```
 0: Functions and Variables for Elliptic Integrals
 1: Functions and Variables for Integration
 2: Functions and Variables for interpol
 3: Interrupts
 4: Introduction to Elliptic Functions and Integrals
 5: Introduction to Integration
 6: Introduction to interpol
 7: askinteger  (Functions and Variables for Simplification)
 8: display_format_internal  (Functions and Variables for Input and Output)
 9: integerp  (Functions and Variables for Miscellaneous Options)
 10: integer_partitions  (Functions and Variables for Sets)
 11: integrate  (Functions and Variables for Integration)
 12: integrate_use_rootsof  (Functions and Variables for Integration)
 13: integration_constant_counter  (Functions and Variables for Integration)
 14: intersect  (Functions and Variables for Sets)
 15: intersection  (Functions and Variables for Sets)
 16: intervalp  (Functions and Variables for orthogonal polynomials)
 17: linearinterpol  (Functions and Variables for interpol)
 18: nonnegintegerp  (Functions and Variables for linearalgebra)
 19: orthopoly_returns_intervals  (Functions and Variables for orthogonal
      polynomials)
 20: poly_ideal_intersection  (Functions and Variables for grobner)
Enter space-separated numbers, 'all' or 'none':
```

このようにマニュアルで文字列 inte を含む項目を一挙に表示します. ここで, 左端の数字を入力すれば該当する数字のマニュアルを表示し, 'all' と入力すると全ての該当するマニュアルを表示して終了し, 'none' と入力すれば何もせずに終了します.

10.5 システムの状態を調べる

10.5.1 status 函数と sstatus 函数

status 函数: Maxima の内部変数*features* に割当てられたリストの一覧を返却したり, 指定した項目が*features*に含まれているかどうかを真理値で返す函数です:

status 函数

status (feature,⟨ 機能 ⟩)
status (feature)
status (status)

なお, 内部変数*features*は属性の feature や features とは無関係です. 引数が feature のみの場合, status 函数は内部変数*features*に設定されたリストを返却します. このリストには Maxima の生成で用いた LISP に関するもの, 文字コードや国際化対応, ハードウエアや OS 等の情報があります. status 函数で第 1 引数を feature にする場合, 第 2 引数は機能の各項目が内部変数*features*に含まれるかどうかを判断して返却します. ここで, 第 2 引数は二重引用符で括っていても, 括らない表象のままでも構いません:

```
(%i7) status(feature);
```

```
(%o7) [cl, mk-defsystem, readline, regexp, syscalls, i18n, loop, compiler,
clos, mop, clisp, ansi-cl, common-lisp, lisp=cl, interpreter, sockets,
generic-streams, logical-pathnames, screen, ffi, gettext, unicode,
base-char=character, pc386, unix]
(%i8) status(feature,powerpc);
(%o8)                                  false
(%i9) status(feature,"unix");
(%o9)                                  true
```

status 函数の引数が status の場合, feature と status で構成されるリストを返します. この status 函数の利用としては, たとえば, UNIX 環境で利用を前提としたパッケージを記述したときに, 'status(features,unix)' の値が 'false' ならば実行不可の環境である旨を返す様にできます.

sstatus 函数: ここで, status 函数では大域変数*features*に項目があるかどうかを検出することしかできません. そこで, sstatus 函数と組合せます:

sstatus 函数

sstatus (feature,⟨ 情報 ⟩)

この sstatus 函数で内部変数*features*に情報を追加することで, status 函数で情報の検出が行えるようになります:

```
(%i5) status(feature,"KNOPPIX/Math");
(%o5)                                  false
(%i6) sstatus(feature,"KNOPPIX/Math");
(%o6)                                  true
(%i7) status(feature,"KNOPPIX/Math");
(%o7)                                  true
(%i8) status(feature);
(%o8) [KNOPPIX/Math, cl, mk-defsystem, readline, regexp, syscalls, i18n, loop,
compiler, clos, mop, clisp, ansi-cl, common-lisp, lisp=cl, interpreter,
sockets, generic-streams, logical-pathnames, screen, ffi, gettext, unicode,
base-char=character, pc386, unix]
```

この例では最初に statsu 函数で KNOPPIX/Math が feature に含まれているかどうかを調べていますが, 勿論, このような機能はありません. そこで, sstatus 函数で feature に KNOPPIX/Math を追加しています. すると, status 函数で KNOPPIX/Math があるかどうかを検出すると追加されていることが判ります.
status 函数と sstatus 函数を用いることで記述したパッケージが必要とする環境の検証等が予め行えるようになります.

10.5.2 room 函数

room 函数

room ()
room (true)
room (false)

room 函数: 保存領域の状況を詳細な記述で出力します. この room 函数は LISP の room 函数を利用したものです.

10.6 時間に関連する函数

10.6.1 処理時間に関連する函数

処理時間表示の函数

time (⟨出力ラベル$_1$⟩,⟨出力ラベル$_2$⟩,...)

time 函数: ⟨出力ラベル⟩の結果を得るために費した計算時間をミリ秒単位で表記したリストを返します. なお, 大域変数数 showtime を 'true' にすると各出力ラベル (%o-行) の内容表示と共に計算時間が表示されます:

```
(%i17) showtime;
(%o17)                                  false
(%i18) integrate(sin(x)*exp(-x),x,0,inf);
                                          1
(%o18)                                    -
                                          2
(%i19) showtime:true;
Evaluation took 0.00 seconds (0.00 elapsed) using 72 bytes.
(%o19)                                   true
(%i20)  integrate(sin(x)*exp(-x),x,0,inf);
Evaluation took 0.02 seconds (0.02 elapsed) using 109.234 KB.
                                          1
(%o20)                                    -
                                          2
(%i21) time(%o18,%o20);
Time:Evaluation took 0.00 seconds (0.00 elapsed) using 96 bytes.
(%o21)                          [0.015998, 0.015998]
```

時間に関連する大域変数

変数名	既定値	概要
lasttime		直前に入力した計算時間
showtime	false	処理時間の表示の有無

大域変数 lasttime: 直前に入力した式の計算時間をミリ秒単位で time と gctime の組合せを成分とするリストです.

大域変数 showtime: 'true' であれば出力式と共に計算時間の自動表示を行います. つまり, CPU 時間も含めて Maxima は計算処理でのゴミ収集 (gc) に費した時間を零でなければ表示します. この時間は "time=" の時間表示に含まれています. なお, "time=" には計算時間のみが含まれて中間表示時間やファイルを読込む時間は含まれてません. そこで, gc への反応性に分けて認識することが難しいために表示する gctime には計算の実行中に費やした全ての gctime を含んでいます. そこで, 稀に "time=" よりも gctime の方が大きくなこと事があります.

10.6.2 システムの時間を返す函数

システムの時間を返す函数

timedate()
absolute_real_time()
elapsed_real_time()
elapsed_run_time()

timedate 函数: システムの日時を返却する函数です. この函数は引数を一切必要としない函数で, 出力する書式も Maxima の利用環境に無関係に次の書式で定まっています:

timedate 函数が出力する時間の書式

HH:	MM:	SS	Day,	mm/	dd/	yyyy	(GMT n)
時間:	分:	秒	曜日,	月/	日/	年	標準時からの差

```
(%i75) timedate();
(%o75)                    18:50:28 Fri, 3/20/2008 (GMT+9)
```

この time 函数は内部の get-decoded-time 函数を用いて得られた結果を整形して表示している函数です.

absolute_real_time 函数: 引数を必要としない函数で, 1900 年の 1 月 (UTC) から経過した時間を整数で返す函数です. この函数は LISP の get-universal-time 函数を用いています.

elapsed_real_time 函数: 引数無用の函数で, Maxima が起動, あるいは再起動した時点からの経過時間を浮動小数点数で返します. この函数は LISP の get-internal-real-time 函数 を用いています.

elapsed_run_time 函数: 引数無用の函数で, Maxima が起動, あるいは再起動した時点から計算処理に要した時間を浮動小数点数で返します. この函数も get-internal-real-time 函数を用いています.

10.6.3 timer 函数,untimer 函数と timer_info 函数

timer 函数

timer(⟨ 函数$_1$⟩,...,⟩ 函数$_n$⟨)
timer(all)
timer()
untimer(⟨ 函数$_1$⟩,...,⟩ 函数$_n$⟨)
untimer()
timer_info(⟨ 函数$_1$⟩,...,⟩ 函数$_n$⟨)
timer_info()

timer に関連する大域変数

timer_devalue	false

10.7 便利な函数

便利な函数

alias(⟨ 新名称$_1$⟩,⟨ 旧名称$_1$⟩,⟨ 新名称$_2$⟩,⟨ 旧名称$_2$⟩,...)
apropos(⟨ 文字列 ⟩)

alias 函数: 函数, 変数, 配列等に別名を与えます. 引数は新名称と旧名称の一組となるので偶数個の引数が必要になります.
alias 函数は⟨ 新名称$_i$⟩ を大域変数 aliases に追加します. この時,⟨ 新名称$_i$⟩ には alias 属性の属性値として⟨ 旧名称$_i$⟩ が LISP の putprop 函数で付与されます.

apropos 函数: ⟨ 文字列 ⟩ を含む Maxima の函数, 大域変数のリストを返します, たとえば, `apropos(exp);` の結果は expand, exp, exponentialize 等の名前の一部に文字列 exp を含む全ての大域変数や函数のリストになります.
この函数を使えば, 大域変数や函数の名前の一部だけさえ覚えていれば残りの名前を探すことができます.

システムに関連する大域変数

変数名	既定値	概要
aliases	[]	別名リスト
debugmode	false	break loop に入るかどうかを設定
myoptions	[]	オプションを蓄えるリスト
optionset	false	オプション設定時にメッセージの有無

大域変数 alieases:　alias 函数, ordergreat 函数, orderless 函数や原子を名詞型と宣言することによって設定された別名を持つ原子のリストです.

大域変数 debugmode:　true の場合, エラーが生じたときや false で中断モードに入ったときは何時でも Maxima の break loop に入ります. all が設定されていれば実行中の函数のリストに対して backtrace を調べられます.

大域変数 myoptions:　利用者が設定した全てのオプションを蓄えるリストです.

大域変数 optionset:　true であれば Maxima はオプションが再設定された時点でメッセージを表示します. これはオプションの綴りが不確かな場合, 割当てた値が本当にオプションの値となっているかを確認したいときに便利です.

10.8　外部プログラムの起動

system 函数:　Maxima 外部のプログラムを起動する函数です:

system 函数の構文
system(⟨ 命令 ⟩)

OS に実行させる処理全体を文字列として system 函数に引渡します. たとえば, ls -a を実行したければ, system(”ls -a”); と入力します.
なお, UNIX 環境で外部プログラムを長く利用し, その間に Maxima も利用したければ, 命令の末尾に &を入れた文字列を system 函数に引き渡します. たとえば, surf を用いたければ system(”surf&”); のように “&” を使います. もし,“&” がなければ system 函数が外部プログラムが終了するまで実行され続けるため, Maxima による処理は停止したままです.

10.9　Maxima の終了

Maxima の終了は logout 函数や quit 函数を用います. これらの函数は引数を必要としませんが必ず quit() のようにうしろの小括弧 “()” を忘れすに入力します.

Maxima を終了させる函数
logout()
quit()

logout 函数:　単純に LISP の (bye) 函数を実行させるだけです. そのために LISP の処理系によっては上手く動作しないものもあるかもしれません.

quit 函数: Maxima の内部変数*maxima-epilog* に束縛した文字列を表示してから終了します. この quit 函数では様々な Common LISP の処理系 (KCL, CMUC SCL, SBCL, CLISP, MCL, GCL) に対し, それらに応じた終了命令を送り込むので, 通常の利用では quit 函数を用いて終了させるのが問題ないでしょう:

```
(%i1) :lisp (setf *maxima-epilog* "さらば、Maximaの利用者諸君！")
さらば、Maximaの利用者諸君！
(%i1) quit();
さらば、Maximaの利用者諸君！
```

この例では内部変数*maxima-epilog*に文字列を setf 函数を用いて束縛させており, quit 函数を実行すると, この内部変数に束縛した文字列が LISP の princ 函数を用いて表示されています.
なお, Maxima を一時的に停止させるのであれば “Ctrl+C(^C)” を入力します.
to_lisp 函数を用いて LISP 環境に入った場合に Maxima を全て終了させたければ, (\$quit) と入力します. こちらの表記が Maxima の quit 函数の LISP 上に於ける表記になります.

10.10　ファイル操作について

10.10.1　ファイルを使った入出力

この節では Maxima の基本的なファイルを使った入出力について述べます. Maxima は Common LISP で記述されているために, 入出力では LISP の入出力函数を用います. ただし, Maxima は表示されている形式と内部形式が基本的に別物となるので, 結果や式を保存したり, 逆に保存した対象を読込む際には注意が必要になります.
まず, ファイルに画面と同じ出力を行いたければ writefile 函数を用います. この writefile 函数は LISP の dribble 函数を用いたもので, Maxima への入力と Maxima の出力をそのまま指定したファイルに保存します. この writefile 函数は指定したファイルを新規に生成するので, 単純に既存のファイルに記録したければ appendfile 函数を用います. writefile 函数と appendfile 函数で開いたファイルを閉じる場合は close 函数を使って, 'closefile()' で開いたファイルを閉じます.
これらの函数で扱うファイルは実質的に記録ファイルであり, Maxima にそのままの形では再利用は出来ません. 再利用可能なファイルを生成するためには save 函数, stringout 函数や grind 函数を用います. ここで, save 函数は Maxima の内部表現を保存する函数で, load 函数や loadfile 函数を用いて Maxima に読込みます. 一方の stringout 函数と grind 函数は内部形式ではなく Maxima の入力に対応する通常の書式で対象をファイルに保存します
Maxima で C の scan 命令に似た函数として, read 函数と readonly 函数があります. これらの函数は引数として与えた文字列を全て同一行に表示し, キーボードからの入力を待ちます. 利用者は通常の Maxima の入力と同様に式を入力し, このときに行末にはセミコロン ";" か "$" を付けます. すると, read 函数の場合は式を Maxima で評価し, readonly 函数の場合は式を評価せずにそのまま受け取ります.
このように単純なファイル操作に限定されるとは言え, 必要なものが最低限揃っています. とは言え, このままでは流石に不十分で, より柔軟に利用するためには §6.10 で解説する stringproc パッケージに含まれる入出力函数や LISP で補うことになります.

10.10.2　Maxima のファイル検出方法

Maxima には様々なファイルが附属します. そして, 各ファイルは所定の場所, すなわち, ディレクトリ (フォルダ) に収められています.
load 函数, demo 函数や example 函数を用いる上で, このディレクトリ構成を把握しておくことが重要なので, Maxima のディレクトリ構成を簡単に纏めておきましょう:

Maxima のディレクトリ構成

src	Maxima のソースファイル (LISP のプログラムファイル) が収納されたディレクトリ
share	Maxima の函数, パッケージファイルが収納されたディレクトリ
demo	Maxima の函数等のデモファイルが収納されたディレクトリ
tests	Maxima のテスト用プログラムが収納されたディレクトリ
doc	Maxima の文書が収納されたディレクトリ

そして, これらのディレクトリに対応する Maxima の大域変数があります.

ディレクトリに関連する大域変数

変数名	既定値	概要
file_search_maxima		Maxima のプログラムファイル読込に関連する大域変数
file_search_lisp		lisp のプログラムファイル読込に関連する大域変数
file_search_demo		デモファイル読込に関連する大域変数
file_search_usage		texi ファイル等の読込に関連する大域変数

これらの大域変数は全て LISP の文字列型与件を成分とする Maxima のリストが割当てられています. ここでの文字列型与件の形式は基本的に検索経路とファイル型の文字の並びで構成されています. そして, これらの大域変数を load 函数, demo 函数や example 函数等のファイルの読込を行う函数が利用する御陰で, 函数やパッケージの名前だけを記述すればファイルの読込が行えるようになっています. たとえば, surfplot という函数を読込みたい場合に 'load(surfplot);' と入力します. すると, Maxima 内部でパッケージファイルが収納されているディレクトリ情報が登録されている大域変数 file_search_maxima の成分に surfplot という名前があるかどうかを調べ, もしも, 名前が存在する場合にはファイルの読込を行います.
この大域変数の成分は具体的には次に示す書式になっています:

ファイル読込に関連する文字列の書式例

```
/usr/local/share/maxima/5.14.0/share/###.{mac,mc}
```

ここで, "###" の個所に引数の surfplot が代入され, そのうしろの "{mac,mc}" が surfplot 函数が収納されたファイルの修飾子となるべき修飾子の候補になります. この修飾子は大域変数の種類によって異なります. そして, "###" の前に記述されている文字列が読込むべきファイルが置かれているディレクトリです. なお, これらの大域変数の既定値は src/init-cl.lisp で設定されています.
この与件の型は Maxima の文字列型ですが, ここで Maxima-5.13.0 以前の Maxima の文字列型と LISP の文字列は異なるので注意が必要です. Maxima-5.14.0 から Maxima の文字列型が LISP の文字列型になったために, Maxima の文字列を LISP の文字列に変換する処理は不要になっています. そして, 適合するファイルが存在すれば Maxima はファイルの読込を実行しますが, ファイルが存在しなければエラーを返す仕組となっています.

大域変数 file_search_maxima: Maxima のプログラムファイルの読込で用いられる大域変数で, ここで指定されるファイルの修飾子は ".mac" か ".mc" です.

大域変数 file_search_lisp: LISP のプログラムファイルの読込で用いられ, ここで指定されるファイルの修飾子は ".fas", ".lisp" か ".lsp" です.

大域変数 file_search_demo: demo 函数で用いられるデモファイルの読込で用いられます. ここでデモファイルの修飾子は ".dem", ".dm1", ".dm2", ".dm3" か ".dmt" です.

大域変数 file_search_usage:　printfile 函数等で用いられる USAGE ファイルの読込等で用いられる大域変数です. ここでファイルの修飾子は “.texi” か “.usg” です.
これらの大域変数に関連する大域変数として, テンポラリファイルの出力先を指定したり, 利用者独自のパッケージファイルの所在を指定する大域変数もあります:

ディレクトリに関連する大域変数

大域変数	概要
maxima_tempdir	一時ファイルの出力先を指定
maxima_userdir	パッケージファイルの捜索場所を指定

大域変数 maxima_tempdir:　Maxima がグラフ表示等で生成するファイルの出力先を指定します.

大域変数 maxima_userdir:　Maxima が利用者定義の Maxima や LISP のパッケージファイルを捜す場所を指定します. 猶, 利用者全てが利用可能なパッケージは大域変数 file_search_maxima や大域変数 file_search_lisp 等で置き場所を指定する必要があります. 大域変数 maxima_userdir を変更しても, 大域変数 file_search_maxima 等の大域変数は影響を受けない事に注意が必要です.

10.10.3　ファイル検出に関連する函数

ファイル検出に関連する函数

filename_merge(⟨ 文字列$_1$⟩ ,⟨ 文字列$_2$⟩)
file_search(⟨ ファイル ⟩,⟨ 検索経路 ⟩)
file_search(⟨ ファイル ⟩)
file_type(⟨ ファイル名 ⟩)

filename_merge 函数:　⟨ 文字列$_1$⟩ と ⟨ 文字列$_2$⟩ の結合を行います. 内部では先頭に “ #P’ ” を文字列の先頭に付けた対象を生成しますが, Maxima 上では単純に文字列を繋ぎ合せたようにしか見えません.
基本的には Maxima の各種命令でファイルの検索を行う際に経路指定のあるファイル名を生成する際に用いられる函数です.

file_search 函数:　指定したファイルを大域変数 file_search_lisp, 大域変数 file_search_maxima と大域変数 file_search_demo に適合するディレクトリとファイルの修飾子で検索し, ファイルが存在すればファイル名を返し, 存在しなければ false を返します. なお, 上記の検索経路以外で検索する必要がある場合, 第 2 引数として追加分の経路を引渡します. この場合, 検索経路の書式は Maxima の検索経路の書式でなければなりません:

```
(%i10) file_search("neko.usg");
(%o10)                          neko.usg
(%i11) file_search(neko,["/home/yokota/##.{usg,tex}''"]);
```

```
(%o11)                          /home/yokota/neko.usg
(%i12) file_search(neko,["/home/yokota/##.{usg,tex}''"]);
(%o12)                          /home/yokota/neko.usg
(%i13) file_search(neko,["/home/yokota/##.{tex,usg}''"]);
(%o13)                          /home/yokota/neko.tex
```

file_type 函数: 指定したファイルの属性を返します. ただし, ファイル名の末尾で判断する函数のために返却する値も, 'object', 'lisp' や 'maxima' を返します. ここで 'object' はコンパイルされた LISP ファイル, 'lisp' は LISP のテキストファイル, 'maxima' は Maxima 言語で記述されたファイルになります.

10.10.4 バッチ処理に関連する函数

バッチ処理に関連する函数

batch(⟨ ファイル名 ⟩)
batchload(⟨ ファイル名 ⟩)

batch 函数: 指定されたファイルに含まれる Maxima の命令行を逐次評価します. ファイルは経路を含まない場合, 大域変数 file_search_maxima に含まれるディレクトリ上を検索し, ファイルが存在した場合には読込みと実行をします.
ここでバッチファイルの内容は, ほぼ Maxima での入力行と同じです. つまり, 行末には記号 ";" か記号 "$" を置きます. ただし, 記号"%" と記号"%th" を用いて入出力を指定することもできます. なお, 空白文字, Tab, 注釈や改行コードは無視されます. そして, 注釈は C のように "/* */" で括り, 注釈の内容が複数行に亘っても問題ありません.
batch 処理ファイルは通常のテキストエディタで編集する事も出来ますし, Maxima の stringout 函数で出力したものも使えます.
深刻なエラーが生じた場合やファイル末端に達した場合にのみ利用者に制御が戻されます. ただし, 利用者はどの時点でも制御文字"Cntrl-g" を押せば処理を止められます.

batchload 函数: 指定されたファイルのバッチ処理を行います. batch 函数との違いは, batchload 函数はファイルに記述された式の入出力の表示を行わないことです.

10.10.5 ファイルの読込を行う函数

ファイルの読込を行う函数

load(⟨ ファイル名 ⟩)
loadfile(⟨ ファイル名 ⟩)
aload_mac(⟨ ファイル名 ⟩)
auto_mexpr(⟨ 函数 ⟩,⟨ ファイル名 ⟩)
read(⟨ 文字列$_1$,$\cdots$⟩)
readonly (⟨ 文字列$_1$⟩ ,$\cdots$)
setup_autoload (⟨ ファイル ⟩,⟨ 函数$_1$⟩,$\cdots$,⟨ 函数$_n$⟩)

load 函数: 文字列やリストで表現されたファイル名の読込みを行います. ディレクトリが指定されていなければ最初にカレントディレクトリ, それから大域変数 file_search_axima, 大域変数 file_search_lisp や大域変数 file_search_demo に指定されているディレクトリとファイルの修飾子に適合するファイルを読込もうとします.
load 函数はファイルが batch 処理に対応していると判断すると, batchload を用います (これは, 黙って端末に出力やラベルを出力せずにファイルの batch 処理を実行することを意味します).
他のファイルの読込を行う Maxima 命令に loadfile 函数, batch 函数と demo 函数があります. loadfile 函数は save で書込んだファイルに対して動作し, batch 函数と demo 函数は strignout 函数の出力や, 利用者がエディタを使って作成・編集したファイルに対して使えます.

loadfile 函数: 指定されたファイルを読込みます. この函数は以前の Maxima の処理で save 函数で保存した値を Maxima に戻すことに使えます.
ここで, 経路の指定はオペレーティングシステムの経路指定に方法に従います. たとえば unix の場合, /home/user ディレクトリにある foo.mc ファイルを読込むのであれば"/home/user/foo.mc" を引数として loadfile 函数に渡します.
save 函数で保存したファイルを loadfile 函数で読込むと, Maxima は初期化されてしまうので注意が必要です.

aload_mac 函数: Maxima のパッケージファイル (修飾子が mc や mac) の読込で利用可能な函数です. この修飾子は省略しても構いません.

auto_mexpr 函数: ファイルから第一引数で指定した函数の読込を行う函数です. この函数は第 1 引数で指定した函数が存在する場合にはファイルの読込を行わずに false を返します.

read 函数: 画面上に全ての引数を表示して入力を待ちます. 利用者が式を入力すると, 入力した式は Maxima に引渡されて評価が行なわれます:

```
(%i45) a:read("mikeneko");
mikeneko
diff(x^2+1,x);
```

```
(%o45)                                  2 x
```

この例では mikeneko と表示されたあとに diff(x^2+1,x); を入力しています. ここでの入力でも通常の入力と同様に行末に; か$が必要です. この例では入力した値が Maxima に評価され, 結局, 変数 a に 2*x が割当てられています.

readonly 函数: 引数を全て表示し, それから式を読込みます. 基本的には read 函数と同様ですが, read 函数と違うのは読込んだ式を評価しないことです:

```
(%i46) a:readonly("mikeneko");
mikeneko
diff(x^2+1,x);
                                    2
(%o46)                         diff(x  + 1, x)
```

setup_autoload 函数: 指定した函数が呼出された時点で函数が未定義の場合, 指定したファイルの読込を実行します. このファイルの読込では函数名に付与された autoload 属性でファイルの自動読込を判断しています. そのために setup_autoload 函数は引数に配列函数が扱えないことに注意して下さい.

ファイルの読込に関連する大域変数

ファイルの読込に関連する大域変数

loadprint	false	読込に伴うメッセージ表示を制御
packagefile	false	パッケージ作成時の情報を制御

大域変数 loadprint: loadfile 函数や autoload 函数によるファイル読込に伴うメッセージ表示を制御する大域変数です. 大域変数 loadprint が取る値は, true, loadfile, autoload と false の四種類で, それぞれで対応が異なります:

- true であればメッセージが常に表示されます.
- loadfile の場合は loadfile 函数が用いられたときのみに表示されます.
- autoload 函数の場合はファイルが自動的に読込まれたときのみに表示されます,
- false の場合はメッセージが決して表示されません.

大域変数 packagefile: save 函数や translate 函数を用いてパッケージ (ファイル) を作成するときに packagefile:true と設定していれば, ファイルを読込む時点で必要な場所を除いた情報が Maxima の大域変数, たとえば, 大域変数 values や大域変数 functions に追加されることが避けられます. この方法でパッケージに含まれる物は利用者のデータを付け加えた時点で利用者の側からは得られません. これは名前の衝突の問題を解決するものではないことに注意して下さい. この大域変数

は単にパッケージファイルへの出力に影響を与えることに注意して下さい. なお, この変数の値を true に設定すると,Maxima の初期化ファイルの生成でも便利です.

10.10.6　ファイルに書込みを行う函数

stringout 函数:　指定したファイルに Maxima が読込める書式で出力する函数です:

stringout 函数の構文

stringout(⟨ ファイル名 ⟩ ,⟨ 式$_1$⟩ , ⟨ 式$_2$⟩ ,$\cdots$)
stringout(⟨ ファイル名 ⟩ , [⟨m⟩ ,⟨n⟩])
stringout(⟨ ファイル名 ⟩,input)
stringout(⟨ ファイル名 ⟩ ,functions)
stringout(⟨ ファイル名 ⟩ ,values)

⟨ 式$_1$⟩ ,⟨ 式$_2$⟩,$\cdots$ と式を並べると各式を順番にファイルに書込む函数です.
引数に [⟨m⟩ ,⟨n⟩] と指定すれば入力行の m 行から n 行がファイルの書込まれます.
第 2 引数に 'input' を指定すると入力行全てが書込まれます. 第 2 引数に 'functions' を指定すると大域変数 functions に記載された利用者定義の函数が全て保存されます.
同様に第 2 引数に 'values' を指定すると, 大域変数 values に記載された利用者定義の変数の値が全てファイルに書込まれます.
この stringout 函数は writefile 函数の実行中に利用することもできます.
大域変数 grind が true であれば, stringout は文字列ではなく, grind 函数と同じ書式で出力します.

その他の書込を行う函数

appendfile(⟨ ファイル名 ⟩)
writefile(⟨ ファイル名 ⟩)
with_stdout(⟨ ストリーム ⟩,⟨ 式$_1$⟩,$\cdots$, ⟨ 式$_n$⟩)
with_stdout(⟨ ファイル名 ⟩,⟨ 式$_1$⟩,$\cdots$, ⟨ 式$_n$⟩)
closefile()

appendfile 函数:　指定したファイルに Maxima の入出力の追加を行います. writefile 函数との違いは, 同名のファイルが存在した場合に writefile 函数は上書きしますが, appendfile 函数は既存のファイルの末尾に Maxima の入出力を追加する点です. なお, writefile 函数と同様に指定したファイルは 'closefile()' で閉じられます.

writefile 函数:　書込用のファイルを新規に開きます. writefile 函数を実行すると, それ以降の Maxima への入出力処理は全て指定したファイルに記録されます. そのために, このファイルをそのまま Maxima に再度読込ませられません.
ファイル名の指定は文字列で行います. ここで ABCD のように二重引用符なしで指定すると, Maxima は文字 "$" を頭に付けたファイル名, すなわち, この例では '$ABCD' という名前のファイ

ルを生成します. なお, この writefile 函数の実体は LISP の dribble 函数です. ここでファイルを閉じる場合には “closefile()” を用います.
次に簡単な例を示します:

```
(%i1) writefile("test1");
(%o1)           #<OUTPUT BUFFERED FILE-STREAM CHARACTER test1>
(%i2) 1+2+3;
(%o2)                                  6
(%i3) diff(sin(x)*x+2,x);
(%o3)                          sin(x) + x cos(x)
(%i4) closefile();
(%o4)       #<CLOSED OUTPUT BUFFERED FILE-STREAM CHARACTER test1>
```

上記の writefile で生成したファイル test1 の内容を以下に示します:

```
;; Dribble of #<IO TERMINAL-STREAM> started 2005-11-17 06:31:16
(%o1)           #<OUTPUT BUFFERED FILE-STREAM CHARACTER test1>
(%i2) 1+2+3;
(%o2)                                  6
(%i3) diff(sin(x)*x+2,x);
(%o3)                          sin(x) + x cos(x)
(%i4) closefile();
;; Dribble of #<IO TERMINAL-STREAM> finished 2005-11-17 06:31:40
```

このように Maxima の画面入出力そのままが保存されています. この writefile 函数を記録ファイルの生成に利用すれば下手なフロントエンドも不要になります.

with_stdout 函数: 指定されたストリームやファイルを開き, ⟨ 式$_1$⟩,···,⟨ 式$_n$⟩ の評価と書込を行います. このときに各式の評価の標準出力への任意の出力は端末の代りに指定したストリームやファイルに送られ, 端末側には常に false が返されます.
なお, 大域変数 file_output_append によってファイル書込の制御が行われます. この大域変数 file_output_append の初期値は false ですが, その値が true の場合は既存のファイルに対して内容の追加を行い, そうでない場合はファイルの上書を行います.

closefile 函数: ‘closefile()’ で appendfile 函数や writefile 函数で開かれたファイルを閉じます. closefile 函数は LISP の close 函数を使った函数です. この close 函数は開かれたストリームを閉じる函数です.

save 函数: 指定したファイルに指定した式や関数等の値を書込む函数で, 更に, 保存した値は削除されずに Maxima 本体にも残っています:

save 函数の構文

save(⟨ ファイル名 ⟩,⟨ 引数$_1$⟩,⟨ 引数$_2$⟩,···)
save(⟨ ファイル名 ⟩,⟨ 名称$_1$⟩ = ⟨ 式$_1$⟩,⟨ 名称$_2$⟩ = ⟨ 式$_2$⟩,···)
save(⟨ ファイル名 ⟩,[⟨m⟩,⟨n⟩])
save(⟨ ファイル名 ⟩,{values,functions,labels},···)
save(⟨ ファイル名 ⟩,all)

save 函数は ⟨ 引数$_1$⟩ ,⟨ 引数$_2$⟩ ,··· で各引数の値を保存します.
[⟨m⟩ ,⟨n⟩] で m 番目の入力行から n 番目の入力行の内容を保存します.
引数に大域変数 values, 大域変数 functions, 大域変数 labels を指定することもできます. この場合, これらの大域変数に登録されている利用者が設定した対象を全て保存します. 同様に labels を指定すると, 入出力行や中間行の内容が全て保存されます.
最後に引数に all を指定すれば, Maxima の内容をファイルに保存します. この場合は入力や計算結果だけではなく, Maxima の設定も一緒に保存されるので, 処理した内容以上にファイルが膨れ上ることに注意が必要です. なお, 10.1 の collapse 函数と併用すれば不要な内部データを削除できるので, 必要に応じて併用すると良いでしょう. この save 函数の返却値は保存先のファイル名です:

```
(%i1) 1+2+3;
(%o1)                                  6
(%i2) a1:x^2+y^2+1;
                                     2    2
(%o2)                               y  + x  + 1
(%i3) resultant(x-t,y-t^2,t);
                                       2
(%o3)                                y - x
(%i4) save("test",all);
(%o4)                                test
```

save 函数で保存したファイルは loadfile 函数で Maxima に再び読込めます. ただし, loadfile 函数による読込を実行すると, save 函数を実行した時点にまで Maxima 自体を戻す効果があるので注意が必要です.
次に示す例では最初に loadfile 函数でファイル test を読込んでいますが, 行ラベルは上の save で保存する場合と同じものになっていることと二度目に loadfile 函数を実行するとラベルが '(%i8)' から '(%i5)' に戻っていることに注意して下さい:

```
(%i1) loadfile("test");
(%o4)                                test
(%i5) %i2;
                                     2    2
(%o5)                          a1 : y  + x  + 1
(%i6) %i1;
(%o6)                                  6
(%i7) %o3;
                                       2
(%o7)                                y - x
(%i8) loadfile("test");
(%o4)                                test
(%i5)
```

save 函数による出力ファイルの内容は LISP の S 式そのものとなります. ファイルの先頭側に実行内容の内部形式が記述されますが, そのうしろには Maxima の諸設定が保存されます. そのために次に示す例では 1+2+3 から 4 行の入力だけしかしていませんが, save 函数で保存したファイル (test) は 256 行に及ぶファイルとなっています. これは内部形式で記述するとどうしても長くなりますが, それ以上に, 入出力以外の設定 (大域変数の値等) も全て保存されているためです:

save 函数で生成したファイルの例

```
;;; -*- Mode: LISP; package:Maxima; syntax:common-lisp; -*-
(in-package "MAXIMA")
(DSKSETQ $%I1 '((MPLUS) 1 2 3))
(ADDLABEL '$%I1)
(DSKSETQ $%O1 6)
(ADDLABEL '$%O1)
(DSKSETQ $%I2 '((MSETQ) $A1 ((MPLUS) ((MEXPT) $X 2) ((MEXPT) $Y 2) 1)))
(ADDLABEL '$%I2)
(DSKSETQ $%O2 '((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 2)))
(ADDLABEL '$%O2)
(DSKSETQ $%I3
 '($(RESULTANT) ((MPLUS) $X ((MMINUS) $T))
   ((MPLUS) $Y ((MMINUS) ((MEXPT) $T 2))) $T))
(ADDLABEL '$%I3)
(DSKSETQ $%O3
 '((MPLUS SIMP) ((MTIMES SIMP) -1 ((MEXPT SIMP RATSIMP) $X 2)) $Y))
(ADDLABEL '$%O3)
(DSKSETQ $%I4 '($(SAVE) &TEST $ALL))
(ADDLABEL '$%I4)
(DSKSETQ $A1 '((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 2)))
(ADD2LNC '$A1 $VALUES)
以下略
```

この save 函数による出力ファイルは Maxima 内部処理を観察する上で重宝するものですが, とは言え, 作業を一旦中断し, 中断した個所から再度処理を行う必要がなければ, save 以外の命令, たとえば, stringout 函数や grind 函数を用いた方が総合的な使い勝手自体は良いでしょう.

10.10.7 その他のファイルに関連する函数

ed 函数: 引数で指定された名称のファイルを編集します. これは LISP の ed 函数を単純に用いるものです:

ファイルの編集を行う函数 ed の構文 ed(⟨ 文字列 ⟩)

通常, Common LISP 側から ed 函数を用いてファイルを編集する場合, スペシャル変数*editor*で指定されたエディタが利用されますが, Maxima から利用する場合は, このスペシャル変数が参照されず, システム側の環境変数 (UNIX の場合は環境変数 EDITOR) が利用されます. 環境変数に値が未設定の場合, UNIX 環境では vi, MS-Windows 環境ではメモ帳がファイル編集のエディタとして利用されます.

printfile 函数: 指定されたファイルをそのままフロントエンドに表示する函数です:

ファイルの表示を行う函数の構文

printfile(⟨ ファイル ⟩)

この函数は単純に read-char 函数と princ 函数を用いてファイルの EOF (End Of File) が現れるま

で読込と表示を行う函数です. このファイルの検索では file_search1 函数を用い, そのときに検索するディレクトリ/フォルダとファイルの種類は大域変数 file_search_usage で指定します.
この大域変数名からも判るように printfile 函数は Maxima の USAGE ファイル (修飾子が “.usg” の ASCII 形式のファイル) を表示させるためのファイルです. この USAGE ファイルを表示させる場合は二重引用符で括らずに, ファイルの修飾子 “.usg” も付けない記号で与えられます. たとえば, ‘printfile(share)’ を実行すると, share.usg ファイルが表示されます. ただし, 一般のファイルに対しては, その経路も含めて二重引用符で括って引き渡さなければなりません.
なお, 日本語の表示も可能なので, 次のようなことができます:

ファイル neko.usg の内容

```
1 ここで何故,結び目なのかという理由ですが,結び目理論はドイツ語
2 ではKnotten Theorieと呼びます.それにしても,Knotten!!
3 実に,KNOPPIXに似ていますねぇ….そこで,結び目愛好家の為に,
4 Maximaで結び目の不変量を計算して,KNOPPIX/Mathを
5 Knotten Pics/Mathと洒落込もうというのが目的です.
```

このファイル neko.usg を file_search_usage に登録されたデイレクトリに置きます. そして, prinfile 函数を実行すると次の結果が得られます:

```
(%i2) printfile(neko);
ここで何故,結び目なのかという理由ですが,結び目理論はドイツ語
ではKnotten Theorieと呼びます.それにしても,Knotten!!
実に,KNOPPIXに似ていますねぇ….そこで,結び目愛好家の為に,
Maximaで結び目の不変量を計算して,KNOPPIX/Mathを
Knotten Pics/Mathと洒落込もうというのが目的です.

(%o2)                /usr/local/share/maxima/5.14.0/share/neko.usg
```

10.10.8 maxima-init.mac ファイル

Maxima は起動時にカレントディレクトリ上にあるファイル maxima-1init.mac を自動的に読込みます. この maxima-init.mac ファイルを上手く利用れば, Maxima の起動時の環境が容易に変更できます.
maxima-init.mac ファイルには Maxima の函数や大域変数の設定が通常の入力と同様の書式で記述します. このときに setup_autoload 函数や load 函数を用いて必要なパッケージの読込も自動的に行えます.
次に非常に簡単な例を示します:

maxima-init.mac の例

```
1 /*-*-MAXIMA-*-*/
2 showtime:all;
3 put(surfg, d_chain_bisection,root_finder)$
4 put(surfg, 0.0000000001,epsilon)$
5 put(surfg, 20000,iterations)$
6 put(surfg, 500,width)$
7 put(surfg, 500,height)$
```

```
 8 put(surf, yes,do_background)$
 9 put(surf, 5,background_red)$
10 put(surf, 5,background_green)$
11 put(surf, 5,background_blue)$
12 put(surf, 0.14,rot_x)$
13 put(surf, -0.3,rot_y)$
14 setup_autoload("surfplot.mc",surfplot)$
15 load("fox.mc")$
```

ここで Maxima が読込むファイルでの註釈は C と同様に “/* */” の中に記述します. この例では頭に註釈行として “/*-*-MAXIMA-*-*/” と置いて Maxima 言語のファイルであることを示しています. それから大域変数の設定, 属性の設定等を行っています.
次に, setup_autoload 函数と load 函数の二つを用いてファイルの読込を行っています. ここの例では surfplot 函数が Maxima 内部で未定義であれば, surfplot.mc ファイルの読込を setup_autoload 函数は実行します. それに対し, fox.mc ファイルは load 函数で無条件に読込みます.
読込むパッケージが大域変数 file_search_maxima に登録されたディレクトリと修飾子を持っているのであれば, ‘load(”fox.mc”)’ の様に二重引用符と修飾子を外して load(fox) とすることが可能です. この詳細に関しては §10.10.2 を参照して下さい.

10.10.9 maxima-init.mac ファイルの設置場所

maxima-init.mac の効果的な置場所は利用環境によって多少異なります. UNIX 系の OS で, 命令の直接入力で Maxima のフロントエンドを起動する場合, カレントディレクトリ上に maxima-init.mac があれば内容が反映されますが, デスクトップ環境から Maxima のフロントエンドを立上げる場合は話がやや異なります.
UNIX 環境であれば環境変数$HOME で指定されたディレクトリ上に maxima-init.mac を置くと良いでしょう. 厄介なのが MS-Windows の場合です. 基本的に MS-Windows 環境の場合, 利用するフロントエンドの exe ファイルが収納されているフォルダに maxima-init.mac を入れておくのが無難なようです.
maxima-init.mac 内部で指定したファイルが存在しない場合や読込みファイルに重大な間違いがある場合, Maxima が起動しないことがあるので注意して下さい.

10.11　虫取りに関連する函数

Maxima には他のプログラム言語と同様に虫取りの函数が幾つか用意されています. ここでの節では虫取りに関連する函数と大域変数について解説します.

10.11.1　動作追跡に関連する函数

trace 函数の構文

trace (⟨ 函数$_1$⟩, $\cdots$, ⟨ 函数$_n$⟩)
trace(all)
trace()
trace_it(⟨ 函数 ⟩)
untrace (⟨ 函数$_1$⟩, $\cdots$, ⟨ 函数$_n$⟩)
untrace()

trace 函数:　指定した函数の動作追跡を行います. 函数の指定は 'trace(integrate)' や 'trace(integrate,factor)' で行います. ここで引数に 'all' を指定した場合は大域変数 functions に登録された全ての函数に対して動作追跡を行うことになります. そして, 引数を指定しない 'trace()' の場合, trace 函数による動作追跡が設定されている函数のリストを返します.

trace_it 函数:　trace 函数と同じ内部函数 macsyma-trace 函数を用いた函数で, trace_it 函数の引数は一つの函数に限定されます.
それでは簡単な例で確認してみましょう:

```
(%i15) trace(integrate);
(%o15)                              [integrate]
(%i16) integrate(2*sin(x)*cos(x),x);
1 Enter integrate [2 cos(x) sin(x), x]
                        2
1 Exit  integrate - cos (x)
                                          2
(%o16)                                - cos (x)
(%i17) trace();
(%o17)                              [integrate]
(%i18) trace(integrate,risch);

integrate is already traced.
(%o18)                                [risch]
(%i19) trace();
(%o19)                          [risch, integrate]
```

ここでの例では最初に integrate 函数の追跡を行うように指定を行っています. この状態で integrate 函数を実行すると, integrate 函数への入力と integrate 函数の出力が表示されます. 次に, 'trace()' によって動作追跡が指定された函数のリストが返却されています.

次に, integrate 函数と risch 函数を trace 函数で指定しています. この場合, integrate 函数は既に指定済みのために警告が出ますが, risch 函数の登録は実行されます.
さて, integrate 函数ではオプションの risch を ev 函数で指定することで Risch 積分が行えます. この場合の動作はどうなるでしょうか？ それれは, 先程の例の続きを示しておきましょう:

```
(%i25) ev(integrate(3^log(x),x),'risch);
                      log(x)
1 Enter integrate [3       , x]
                       log(3) log(x)
                    x %e
1 Exit  integrate ------------------
                      log(3) + 1
                                    log(3) log(x)
                                 x %e
(%o25)                           ------------------
                                    log(3) + 1
(%i26) integrate(3^log(x),x);
                      log(x)
1 Enter integrate [3       , x]
                        1
                     (------ + 1) log(x)
                      log(3)
                    3
1 Exit  integrate ----------------------
                        1
                    (------ + 1) log(3)
                     log(3)
                                      1
                                   (------ + 1) log(x)
                                    log(3)
                                 3
(%o26)                           ----------------------
                                     1
                                 (------ + 1) log(3)
                                  log(3)
(%i27) risch(3^log(x),x);
                    log(x)
1 Enter risch [3       , x]
                     log(3) log(x)
                  x %e
1 Exit  risch ------------------
                    log(3) + 1
                                    log(3) log(x)
                                 x %e
(%o27)                           ------------------
                                    log(3) + 1
```

この例では risch 函数の本体を使うものの, Maxima の risch 函数そのものは使いません. そのために trace 函数で risch 函数は表に出て来ません.

untrace 函数: trace 函数で指定した動作追跡を解除する函数です. 引数を指定しない場合は無条件で全ての函数の動作追跡を解除します.

trace_options 函数:　単純に trace 函数を函数に作用させるだけではその函数への入出力しか出力されませんでした. より詳細な情報を得るために trace 函数に対するオプション設定が必要になります. このオプションの設定は trace_options 函数を用います:

trace_options 函数

trace_options(⟨ 函数 ⟩, ⟨ オプション$_1$⟩, $\cdots$, ⟨ オプション$_n$⟩)
trace_options(⟨ 函数 ⟩)
trace_options()

まず, 基本的な構文として, 第 1 引数に trace 函数で追跡を行う函数名を指定して, そのあとに指定可能なオプションを記入します.
なお, オプションなしの trace_options(⟨ 函数 ⟩) で追跡を行う函数に設定したオプションを解除して初期状態に戻します.
指定可能なオプションを次の表に纏めておきます:

trace_optons 函数で指定可能なオプション

noprint	true の場合には表示を行いません.
break	true の場合には breakpoint を与えます.
lisp_print	true の場合に LISP の内部形式で表示を行います.
info	
errocatch	true の場合にエラーが捕捉されます.

trace_options 函数で設定した trace 函数のオプションの情報は get 函数を用いて取出せます:

```
(%i9) trace_options(integrate,info);
(%o9)                                [info]
(%i10) get('integrate,'trace_options);
(%o10)                               [info]
```

この例で判るように trace_options 函数によって函数の trace_options 属性にオプションの値が設定されます. このことから get 函数によってオプションの情報が入手可能となる訳です.

10.11.2　bug_report 函数と build_info 函数

bug_report 函数と build_info 函数

bug_report()
build_info()

bug_report 函数:　Maxima の虫取りを行う上で必要となる Maxima の情報を出力する函数です. この函数は引数を必要としません. なお, この函数の実体は build_info 函数で build_info 函数から返された情報に予め用意した文書を追加して表示させているだけです.

build_info 函数: Maxima を構築する再に用いた Common Lisp の情報等, Maxima の生成に関連する情報を返す函数です. この函数は内部変数の*maxima-build-time* , *autoconf-version* , *autoconf-host* , の内容, 内部函数の lisp-implementation-type 函数 や lisp-implementation-version 函数 が出力する情報を纏めて返す函数です.

10.11.3 関連する大域変数

虫取りに関連する大域変数

大域変数名	既定値	概要
setcheck	false	変数をリストで指定し, その挙動を追跡する
setcheckbreak	false	
setval	'setval	Maxima break 時に追跡している変数の値を一時的に保全
trace	[]	追跡を行う函数のリスト
trace_max_indent	15	
trace_break_arg		
trace_safety	true	

大域変数 setcheck: ある特定の変数がどの様に書換えられて行くかを追跡したい場合, その追跡したい変数のリストとして指定します.
それでは実際にその効果を試してみましょう:

```
(%i2) setcheck:['x,'y];
(%o2)                                [x, y]
(%i3) x:128;
x SET TO 128
(%o3)                                 128
(%i4) y:x*2;
y SET TO 256
(%o4)                                 256
(%i5) x:'x;
(%o5)                                  x
(%i6) x:10;
x SET TO 10
(%o6)                                 10
```

このように大域変数 setcheck で割当てた変数リストに含まれる変数に割当てが発生して時点で, "x SET TO 10" のように表示されます.

大域変数 setcheck: all で全ての変数に true 設定しても構いません. ただし, 'x:'x' のように大域変数 setcheck で指定された変数がそれ自身に割当てられている場合は表示されません.

大域変数 setcheckbreak:　true の場合、setckeck リストに収録された変数に値の設定が行われるときに break 函数によって処理が中断されます. この時点で setval 変数に設定されようとする変数の値が保全されます. なお, setval 変数に保全された値を別途の再設定して大域変数 setcheckbreak に設定された変数の値を意図的に変更しても構いません.

大域変数 setval:　追跡している変数の値を一時的に蓄えるために用いられます. この変数は Maxima Break のときにはじめて値が設定され, 利用者はその値の書換もできます.

10.11.4　LISP の trace 函数との併用

Maxima の trace 函数では Maxima 側の処理しか見えません. たとえば, 演算子"and"の動作を Maxima の trace 函数を用いて見てみましょう:

```
(%i4) trace("and");
(%o4)                                ["and"]
(%o4)                                  []
(%i5) (x >1+2) and (y-1>2);
1 Enter "and" [x > 1 + 2 and y - 1 > 2]
1 Exit  "and" x > 3 and y - 1 > 2
(%o5)                          x > 3 and y - 1 > 2
```

このように演算子 "and" で処理する述語の入力と, それに対応する出力が Maxima の式で出力されます. では, この内部処理をより深く見るために演算子 "and" に与えられた与式を簡易化して評価する内部函数の simp-mand の動作を見ます. そのためには演算子 ":lisp" を用います. この演算子 ":lisp" は Maxima 側から :lisp ⟨LISP の S 式⟩ と入力すると与えられた S 式を LISP に引き渡して処理させる函数です. この場合は ':lisp (trace simp-mand)' と入力します.

```
(%i6) :lisp (trace simp-mand)
WARNING: TRACE: redefining function SIMP-MAND in top-level, was defined in
         /usr/local/maxima-5.13.0/src/binary-clisp/compar.fas
;; Tracing function SIMP-MAND.
(SIMP-MAND)
(%i6) (x >1+2) and (y-1>2);
1 Enter "and" [x > 1 + 2 and y - 1 > 2]
1 Exit  "and" x > 3 and y - 1 > 2
1. Trace:
(SIMP-MAND
 '((MAND) ((MGREATERP SIMP) $X 3) ((MGREATERP SIMP) ((MPLUS SIMP) -1 $Y) 2)) '1
 'NIL)
1. Trace: SIMP-MAND ==> ((MAND SIMP) ((MGREATERP SIMP) $X 3) ((MGREATERP SIMP)
        ((MPLUS SIMP) -1 $Y) 2))
(%o6)                          x > 3 and y - 1 > 2
```

このように処理すれば, 内部でどのような処理が行われているかが内部表現から明確になり, より詳細な情報が得られるのです.
なお, LISP の trace 函数を使って追跡されている函数の一覧を見る場合, '(trace)' と入力し, '(untrace)' で全ての追跡を止め, 追跡を停止する函数を指定する場合には, '(untrace ⟨ 函数$_1$⟩, ··· , ⟨ 函数$_n$⟩)' で指定します.

10.11.5 システムの検証計算を行う函数

Maxima には標準でシステムの検証を行う函数 run_testsuite 函数があります.

run_testsuite 函数
run_testsuite()
run_testsuite(t)

run_testsuite 函数: 引数を必要としない函数で, 大域変数 file_search_tests に割当てられたリストに登録されたディレクトリ上のテストプログラムを実行します. なお, この大域変数のディレクトリは内部変数*maxima-tesitdir* に束縛された値です:

```
(%i4) run_testsuite();
Running tests in rtestnset: 502/502 tests passed.
Running tests in rtest1: 27/27 tests passed.
Running tests in rtest1a: 24/24 tests passed.

... 途中省略

Running tests in rtest_sign: 89/89 tests passed (not counting 11 expected errors
        ).


No unexpected errors found.
Real time: 137.03793f0 sec.
Run time: 135.00844f0 sec.
Space: 3293125704 Bytes
GC: 587, GC time: 26.913685f0 sec.
(%o0)                                  done
```

ここで run_testsuite 函数の引数を t とすると, Maxima は計算の過程を詳細に表示しながら検証を行います. この際に既知の虫があれば, その旨も表示しながら処理を行います.
なお, 大域変数 testsuite_files は内部変数*maxima-tesitdir*で示されるディレクトリにある testsuite.lisp ファイルに記述されたリストの値です. この大域変数はテストファイルの名前と既知の虫 (数字で表記) で構成されたリストです.

第11章　Maximaでグラフ表示

この章では Maxima のグラフ表示機能について述べます. ここでは,Maxima のグラフ表示函数の解説に留まらず, 実際に使い熟すために重要と思われる gnuplot の利用方法, さらには Maxima の draw パッケージの利用に関しても解説を行います.

11.1　Maxima のグラフ表示

11.1.1　はじめに

Maxima には目的に応じた描画を可能にするために, いろいろな可視化函数が利用できます. *Mathematica* や Maple といった商用の数式処理システムと比較すると流石に見劣りすることもありますが, ちょっとした式や与件の可視化に十分な機能を持っています. Maxima での描画の特徴として, Maxima で生成したデータを外部のアプリケーションに引渡し, そのアプリケーションで描画を行う点です. 函数によっては, この表示用のアプリケーションを目的に応じて切替えられたりしますが, 固有のアプリケーションだけに対応した函数もあります.
Maxima の標準的な描画函数は plot2d 函数と plot3d 函数です. これらの函数はそれぞれ 2 次元グラフと 3 次元グラフを専門に描く函数です. plot2d 函数と plot3d 函数以外の函数で, 機能的にこれらの函数に勝る物は draw パッケージを除くとありません. ここで draw パッケージは gnuplot のバージョンが 4.2 以上に限定されていて全ての計算機環境で同等の機能を保証するものではありません[1]. そこで, ここでは plot2d 函数と plot3d 函数を中心に解説し, draw パッケージは別途纏めて解説します.

11.1.2　plot2d と plot3d による描画の概要

最初に plot2d 函数と plot3d によるによるグラフ表示について簡単に説明しましょう. まず, 大域変数 plot_options の plot_format に外部アプリケーションの指定を行います. この外部アプリケーションの指定にしたがって Maxima 内部で与件の生成を行い, ファイルとして出力します.
ここで生成される与件ファイルの内容は外部アプリケーションによって詳細は異なりますが, 通常は曲線や曲面を構成する点の座標値を示す数値データを中心に構成されたもので, Maxima で plot2d 函数や plot3d 函数に入力した式そのものではありません. また, 大域変数 plot_options に含まれる run_viewer で指定される外部アプリケーションを起動させずに, 外部アプリケーション向けの与件のみを生成することもできます.
それから Maxima は外部アプリケーションを立上げ, 生成したファイルを引渡しします. そこで, 外部アプリケーションが実際の描画を行いますが, あとの処理は基本的に外部アプリケーション任せになります.

11.1.3　plot2d と plot3d で利用可能な外部アプリケーションについて

この plot2d 函数と plot3d 函数で利用可能な外部アプリケーションで代表的なものを次に列挙しておきます:

[1] MS-Windows 版の Maxima では draw パッケージにも対応した gnuplot が附属している為に最初から利用可能です.

plot2d 函数や plot3d 函数で利用可能なアプリーケーション

gnuplot	汎用 (標準)
openmath	汎用
Geomview	3 次元グラフ描画専用
izic	3 次元グラフ描画専用 (殆ど使われていません)

Maxima-5.10.0 以降の標準の描画アプケーションは gnuplot ですが, Maxima-5.10.0 以前は openmath を用いていました. gnuplot と openmath を比較すると, gnuplot の方が全般的に高機能ですが, Maxima 側からグラフをちょっと描く程度の処理であれば大差はありません. むしろ, openmath の方が 3 次元グラフィックスが綺麗かもしれず, その上, openmath が Maxima のソースファイルに最初から附属しているので, Maxima が利用可能な環境であれば openmath も使えるという長所もあります.
Geomview は 3 次元グラフ専用ですが非常に高品質のグラフ表示が可能であり, さらに Euclid 空間以外の双曲空間等の空間内部でのグラフの表示さえもできます. ただし, Geomview の動作環境は基本的に UNIX 環境に限定されます.
最後の zic は Izic を外部アプリケーションとするものです. この Izic 自体は Tcl/Tk でフロントエンドを記述した古いアプリケーションなので実際には使われないでしょう.

11.1.4 与件ファイルについて

与件ファイルの置かれる場所は UNIX 環境ではホームディレクトリ上, MS-Windows 環境であれば, DOS 窓から Maxima を使うのであれば Documents and Settings フォルダの中にあるログインユーザー名のフォルダ, wxMaxima なら Maxima のフォルダの中にある wxMaxima のフォルダ等になります.
与件ファイルの名前は外部アプリケーション毎に異なっています. 具体的には, openmath で maxout.openmath そして, Geomview は maxout.geomview , gnuplot の場合は maxout.gnuplot や maxout.gnuplot_pipes のように maxout のうしろに plot_format の値に対応するアプリケーション名が付いたファイルになります.
このファイルは対応するアプリケーションで別途利用することもできます. たとえば, gnuplot 向けの与件ファイルにはデータの他に描画命令やオプションの諸設定も記述されたファイルとなっているので, 立上げた gnuplot から `load 'maxout.gnuplot'` で読込むだけでグラフ表示ができます. このときに mouse が on になっていればマウスを使って直接三次元画像を把持して回転や拡大が行えます. もしも, mouse が on でなければ, `set mouse` で on にできます
mgnuplot 向けの与件ファイルであれば gnuplot 向けの曲線, あるいは曲面のデータのみが含まれたファイルになります. そのために plot2d 函数の場合は `plot 'maxout.mgnuplot'` , plot3d 函数の場合, `splot 'maxout.mgnuplot'` を実行すれば描画を行います.

gnuplot による描画

gnuplot を利用する場合, 大域変数 plot_format に gnuplot と gnuplot_pipes の二種類の値が設定できます:

- gnuplot
  グラフを表示させると Maxima 側から設定の変更や再描画は行えず, マウス操作 (拡大や縮小, 3 次元グラフの回転) といったグラフの操作も不可.

- gnuplot_pipes
  Maxima 側からの設定の変更, 再描画, マウス操作 (拡大や縮小, 3 次元グラフの回転) といったグラフの操作が可能

ここでマウスによる画像操作は gnuplot の mouse を 'on' にしています. ここで mouse を 'off' にしてウィンドウの直接操作を停止したければ, グラフを表示しているウィンドウ上でキーボードから直接 m と入力します. 逆に mouse を 'off' から 'on' にしたければ, 同様に m と入力します. さらに gnuplot を立上げて処理を行っている場合, gnuplot の命令入力ウィンドウから set mouse を実行すると 'on' に, uset mouse を入力すると 'off' にできます.
maxout.gnuplot_pipes ファイルと maxout.gnuplot ファイルの場合の違いを解説しておきましょう. 最初の maxout.gnuplot_pipes には描画用の数値与件のみが格納されており, 描画命令や設定等の gnuplot の命令文は Lisp のストリームを用いて gnuplot に送り込まれます.
それに対して maxout.gnuplot ファイルの内容は, maxout.gnuplot_pipes に含まれている曲面や曲線の数値与件に加え, gnuplot の描画命令や設定を含む命令文がその先頭に入っています. そのために別途立ち上げた gnuplot から load 'maxout.gnuplot' で読込めば, Maxima のとき同じグラフが表示されます. さらに gnuplot で mouse を 'on' にしておけばマウスによる画像の回転や拡大操作ができます.

MS-Windows 版 Maxima での gnuplot の利用

MS-Windows 版のみは gnuplot ではなくパッケージに同梱された wgnuplot を利用します. ただし, ここでは gnuplot と wgnuplot は特に区別する必要がなければ特に区別せず, 単に gnuplot と呼んでいます. この wgnuplot は Maxima のフォルダ (通常は Program Files フォルダにある筈です) の bin フォルダにあります. 勿論, この wgnuplot は独立して動かせますが, 最初に立ち上げた時点でフォントが潰れて読めない状態なので, gnuplot のウィンドウで右マウスボタンをクリックし, それから 'Choose Font' を選択して適当なフォントに設定せば改善されます.
ここで MS-Windows 版固有の機能として, mouse を 'off' にした状態からメニューを呼出して表示画像をクリップボードに保存できます. そのためには mouse を 'off' にした状態で右マウスボタンをクリックして 現われるメニューから Copy to Clipboard を選択すれば画像がクリップボードに貼付けられます.

11.1.5 plot2d 函数

plot2d 函数は平面曲線の描画を行う函数です. 次に構文を纏めておきます:

plot2d の構文

plot2d (⟨ 式 ⟩,⟨ 定義域 ⟩,⟨ オプション$_1$⟩,...,⟨ オプション$_n$⟩)
plot2d (⟨ 式 ⟩,⟨ 定義域 ⟩,⟨ 値域 ⟩,⟨ オプション$_1$⟩,...,⟨ オプション$_n$⟩)
plot2d (⟨ 媒介変数式 ⟩,⟨ オプション$_1$⟩,...,⟨ オプション$_n$⟩)
plot2d (⟨ 媒介変数式 ⟩)
plot2d (⟨ 離散式 ⟩,⟨ オプション$_1$⟩,...,⟨ オプション$_n$⟩)
plot2d (⟨ 離散式 ⟩)
plot2d ([⟨ 式$_1$⟩,...,⟨ 式$_n$⟩],⟨ 定義域 ⟩,⟨ 値域,⟨ オプション$_n$⟩)
plot2d ([⟨ 式$_1$⟩,...,⟨ 式$_n$⟩],⟨ 定義域 ⟩,⟨ 値域 ⟩)
plot2d ([⟨ 式$_1$⟩,...,⟨ 式$_n$⟩],⟨ 定義域 ⟩)

plot2d 函数で描ける式は利用者が記述した 1 変数の Maxima の函数, 複素値函数で実部が 1 実数変数を含む式の実部, 媒介変数を用いて記述した函数, そして, 離散的な数値与件の表示です.

通常の Maxima の式: 変数の定義域の設定が必要となります. ここで定義域は [⟨ 変数 ⟩,⟨ 下限 ⟩,⟨ 上限 ⟩] で構成されたリストです. ここでグラフの縦方向の表示範囲を指定する値域も似たリストですが, このときの変数は y に固定されているので [y,⟨ 下限 ⟩,⟨ 上限 ⟩] となります. ただし, 離散式に対する値域の指定は無効になります.

複素函数の実部: 大域変数 plot_options の plot_realpart の項目を 'true' に予め設定しておくか, '[plot_realpart,true]' をオプションとして引渡します.

媒介変数式: Maxima で利用可能な媒介変数式の書式は次の二通りの書式が許容されます:

媒介変数式の書式

[`parametric`,⟨`X` 座標 ⟩,⟨`Y` 座標 ⟩,⟨ オプション$_1$⟩,...,⟨ オプション$_n$⟩]
[`parametric`,⟨`X` 座標 ⟩,⟨`Y` 座標 ⟩]

この場合, X 座標と Y 座標は変数 t を唯一の変数として持つ式でなければなりません. たとえば, 半径 1 の円は '[parametric,cos(t),sin(t)]' と表記します. 通常の plot2d 函数のオプションも入れられますが, 媒介変数 t の定義域は媒介変数式の中でも, plot2d 函数で描く式の定義域として与えても構いません. 図 11.1 に 'plot2d([parametric,cos(2*%pi*t/6),sin(2*%pi*t/6)])' の結果を示します:

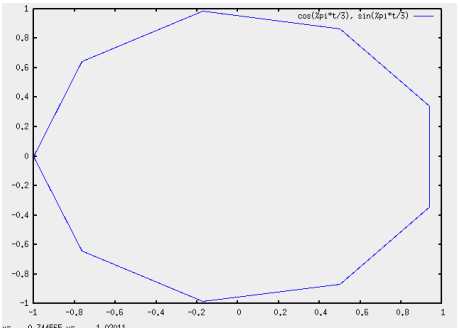


図 11.1: plot2d([parametric,cos(2*%pi*t/6),sin(2*%pi*t/6)]) の結果

ここで媒介変数 t の定義域は省略できます．これは大域変数 plot_options に予め媒介変数 t の定義域が ‘[t,-3,3]’ で設定されているからです．媒介変数 t の定義域を変更したい場合は大域変数 plot_options の項目 t の定義域とする方法と plot2d 函数で定義域を描画の度に指定する方法があります．ここで媒介変数式でグラフの粗さが目立てば大域変数 plot_options の項目の ‘nticks’ を ‘10’ よりも大きな値に設定するか, あるいは ‘[nticks,100]’ のように plot2d 函数のオプションとして引渡します．

点列の描画:　点列の描画は媒介変数式と似た書式になります．したがって二つの書式が可能です:

離散式の書式
[discrete, ⟨X 成分リスト⟩, ⟨Y 成分リスト⟩]
[discrete, [[⟨X 成分$_1$⟩, ⟨Y 成分$_1$⟩], . . ., [⟨X 成分$_n$⟩, ⟨Y 成分$_n$⟩]]]

まず, 双方の書式共に discrete で開始し, それから, X 成分のリストとそれに対応する Y 成分のリストか, X 成分と Y 成分の対 ‘$[x_i, y_i]$’ で構成されるリストの二種類になります．

複数のグラフ表示:　複数のグラフ表示を行う場合, 表示する複数の式を 1 つのリストで与えます．ここでリストに含まれる媒介変数式や離散式以外の Maxima の式は定義域が全て同じものでなければなりません．
媒介変数式を含む場合, その媒介変数式毎に定義域やその他のオプションが設定可能です．図 11.2 に媒介変数式を含む式リストの処理結果を示します:

```
plot2d([x\^{}3+2,[parametric,cos(t),cos(t)*sin(t)]],
       [x,-3,3],[y,-2,4],[t,-5,5],[nticks,100]);
```

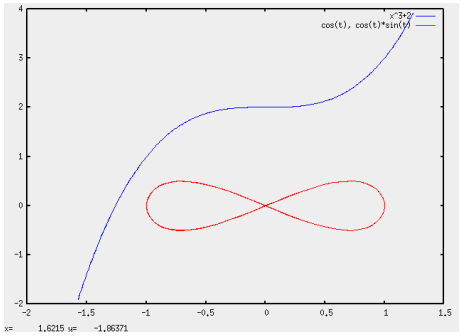


図 11.2: 媒介変数式と通常の式の混在

複数の媒介変数式が存在する場合, 媒介変数式内部にオプションを持たせて媒介変数式毎に設定が行なえます:

```
plot2d([[parametric,cos(t),sin(t),[t,-%pi,%pi],[nticks,20]],
[parametric,2*cos(t),sin(-t),[t,-%pi/2,%pi/4],[nticks,5]]]);
```

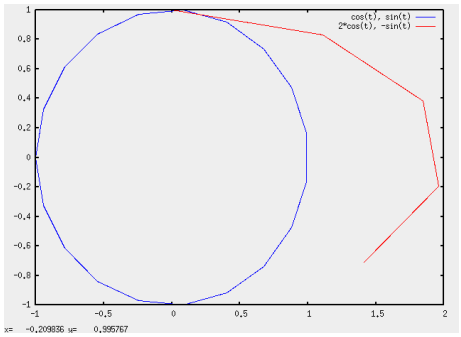


図 11.3: 複数の媒介変数式

この例では最初の半径 1 の円周の描画を 20 個の点で描画しますが, その次の楕円の描画では 5 点で描画しています. ここで nticks の位置に注意して下さい. つまり, nticks の指定は媒介変数式の中に個別に記述しているので, nticks の設定内容の影響は個々の媒介変数式にのみ影響されます. では以下のように変更するとどうなるでしょうか？

```
plot2d([[parametric,cos(t),sin(t),[t,-%pi,%pi],[nticks,20]],
[parametric,2*cos(t),sin(-t),[t,-%pi/2,%pi/4],[nticks,5]]],
[nticks,40]);
```

この場合, 媒介変数式内部の設定が優先されるので外の '[nticks,40]' の内容は無視されて結果として図 11.3 と同じ絵になります.
ところが plot_format で openmath を指定する場合に複数の媒介変数式の中に定義域を記述すると描画が上手くできないことがあります. そのために plot2d 函数を用いる場合は plot_format を既定値の gnuplot にしたままの方が無難でしょう.

plot2d のオプション:　plot2d 函数のオプションは大域変数 plot_options を構成するリストでもあります. この大域変数 plot_options の諸項目リストを設定して, グラフのタイトルやラベル, X, Y 軸のラベル等の細かなグラフの指定が行えます. 特に gnuplot を用いる場合, つまり, plot_format が gnuplot, あるいはオプションとして '[plot_format,gnuplot]' を与えた場合に plot_options の gnuplot_preamble を上手に使って gnuplot の命令文を実行させることができます. この大域変数 plot_options には多くの説明すべき事項があるので, 次の節で詳細を述べることにします.

11.1.6　plot3d 函数

Maxima では f(x,y) の形式の 2 変数の式の描画できます. さらに複素数値函数の実部のみの表示できますが, 式 'x^2+y^2+z^3=1' のような比較の演算子を含む式のグラフはそのままの形式では描けません.
この plot3d 函数の構文は基本的に plot2d 函数の構文を 3 次元にそのまま拡張したものになります:

plot3d の構文

plot3d (⟨ 式 ⟩,⟨ 定義域$_1$⟩,⟨ 定義域$_2$⟩,⟨ オプション$_1$⟩,...,⟨ オプション$_n$⟩)
plot3d (⟨ 式 ⟩,⟨ 定義域$_1$⟩,⟨ 定義域$_2$⟩)
plot3d ([⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨ 式$_3$⟩],⟨ 定義域$_1$⟩,⟨ 定義域$_2$⟩,⟨ オプション$_1$⟩,...,⟨ オプション$_n$⟩)
plot3d ([⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨ 式$_3$⟩],⟨ 定義域$_1$⟩,⟨ 定義域$_2$⟩)
plot3d ([⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨ 式$_3$⟩],⟨ 定義域$_1$⟩,⟨ 定義域$_2$⟩,⟨ 定義域$_3$⟩,⟨ オプション$_1$⟩,...,⟨ オプション$_n$⟩,[*transform_xy*,⟨ 函数 ⟩])

plot3d 函数は plot2d 函数と似た構文を持っていますが, plot2d 函数とは違い複数の曲面を同時に描くことは出来ません. さらに gnuplot で曲面を描く場合は pm3d の設定を行うと曲面を張りますが, 無設定の場合は単なるワイヤーフレーム表示になります. 詳細は §11.2 を参照して下さい.
plot3d 函数は 3 次元空間内の曲面だけではなく空間曲線も描けます. ここで空間曲線を描く場合は x, y, z 成分を ⟨ 定義域$_1$⟩ か ⟨ 定義域$_2$⟩ の変数として記述します.
極座標系の場合も空間曲線を描く方法に似ていますが, この場合は大域変数 plot_options の項目 transform_xy に座標変換を行う函数を指定する必要があります. ここで通常のデカルト座標系から極座標系への変換は単純に polar_to_xy を指定するだけで済みます. ここで, 利用者独自の座標変換函数の指定が必要であれば make_transform 函数を併用する必要があります. 詳細は大域変数 plot_options の transform_xy の個所で述べます.

11.2 大域変数 plot_options

11.2.1 大域変数 plot_options 概要

ここでは plot2d 函数と plot3d 函数向けの設定が行える大域変数 plot_options を解説します.
まず最初に plot_options の値を見てみましょう. これは直接 'plot_options' と入力すれば見られます. たとえば, Maxima-5.10.0 で入力した様子を以下に示しておきましょう.

```
(%i8) plot_options;
(%o8) [[x, -1.75555970201398E+305, 1.75555970201398E+305],
[y, -1.75555970201398E+305, 1.75555970201398E+305], [t, -3, 3],
[grid, 30, 30], [transform_xy, false], [run_viewer, true],
[plot_format, gnuplot_pipes], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 10], [adapt_depth, 10],
[gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]], [gnuplot_curve_styles,
[with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,
with lines 6, with lines 7]], [gnuplot_default_term_command,
set term x11 font "Helvetica,16"], [gnuplot_dumb_term_command,
set term dumb 79 22], [gnuplot_ps_term_command,
set size 1.5, 1.5;set term postscript eps enhanced color solid 24],
[gnuplot_pipes_term, x11], [plot_realpart, false]]
```

このように大域変数 plot_options は二成分リストから構成された複合リストです. この大域変数を構成する二成分リストは '[plot_format, gnuplot]' の書式で, 先頭が plot_options の項目で後がその項目の値になります. たとえば, '[plot_format,gnuplot]' は外部アプリケーションが gnuplot で, 生成するファイルも gnuplot のデータファイルになることを示します. この表示では流石に見難いですね. そこで, get_plot_option 函数を用いると大域変数 plot_options の 1 つの項目に対して項目とそれに対する値の二成分リストを返すので, 内容の把握が容易に行えます.

11.2.2 大域変数 plot_options の設定に関連する函数

大域変数 plot_options の各項目に対応する値を変更を割当の演算子 ":" を使って, このリストを一々全て記述することは効率の良い方法ではありません. 指定した項目を変更する目的で set_plot_option 函数が用意されています. この set_plot_option 函数の引数は項目とその値で構成されるリストを引渡します. ただし, set_plot_option 函数に指定可能な項目は一件に限られます. これらの函数の構文を以下に纏めておきましょう:

大域変数 plot_options に関連する函数

set_plot_options([⟨ 項目 ⟩,⟨ 値 ⟩])
get_plot_option(⟨ 項目 ⟩)

これらの函数の例を示しておきましょう. ここでは plot_format の値を get_plot_option 函数で取出し, この項目の値を set_plot_option 函数で openmath に変更するというものです:

```
(%i9) get_plot_option(plot_format);
(%o9)                    [plot_format, gnuplot_pipes]
```

```
(%i10)  set_plot_option([plot_format,openmath]);
(%o10) [[x, - 1.75555970201398E+305, 1.75555970201398E+305],
[y, - 1.75555970201398E+305, 1.75555970201398E+305], [t, - 3, 3],
[grid, 30, 30], [transform_xy, false], [run_viewer, true],
[plot_format, openmath], [gnuplot_term, default], [gnuplot_out_file, false],
[nticks, 10], [adapt_depth, 10], [gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]], [gnuplot_curve_styles,
[with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,
with lines 6, with lines 7]], [gnuplot_default_term_command,
set term x11 font "Helvetica,16"], [gnuplot_dumb_term_command,
set term dumb 79 22], [gnuplot_ps_term_command,
set size 1.5, 1.5;set term postscript eps enhanced color solid 24],
[gnuplot_pipes_term, x11], [plot_realpart, false]]
(%i11) get_plot_option(plot_format);
(%o11)                          [plot_format, openmath]
```

では, 大域変数 plot_options の内容を分類して解説しましょう.

11.2.3 外部アプリケーションの設定に関連する項目

plot_options には外部アプリケーションの指定と起動に関する項目があります:

外部アプリケーションに関連する項目

項目	既定値	概要
plot_format	[plot_format, gnuplot]	グラフ表示アプリケーションを設定
run_viewer	[run_viewer, true]	true の場合に plot_format で指定したアプリケーションを起動
colour_z	[colour_z, false]	カラーの PS ファイルを出力するかどうかを指定するフラグ
view_direction	[view_direction, 1, 1, 1]	plot_format が ps の場合に 3 次元グラフの視点を指定

まず, plot_format で描画に用いるアプリケーションの指定を行い, run_viewer で外部アプリケーションの起動の有無を指定します.

run_viewer: この run_viewer に 'true', あるいは 'false' を設定します. 'true' であれば plot_format で指定した外部アプリケーションを起動して, 生成した与件ファイルを引渡します. 'false' であれば与件ファイルのみを生成し, 外部アプリケーションを起動しません. ここでの与件ファイルの命名規則と置かれる場所については §11.1.4 を参照して下さい.

plot_format: グラフ表示で用いる外部アプリケーションを指定し, 指定可能な値は 'gnuplot_pipes', 'gnuplot,openmath' ,'geomview', 'pls' と 'zic' です. そして, Maxima-5.12.0 から既定値として

UNIX 環境では 'gnuplot_pipes', MS-Windows 環境では 'gnuplot' が指定されています.
ここで描画に用いる外部アプリケーションは基本的に値と同名のアプリケーションですが, 'ps' を指定した場合だけは ghostview がグラフ表示用の外部アプリケーションとして設定されます.

colour_z: 'true' であればカラーの PostScript ファイル出力を行い, 'false' であれば白黒の PostScript ファイルを出力します. この colour_z は plot_format として ps が指定された場合のみに有効です.

view_direction: 視点の位置を指定しますが plot_format が ps のときだけ有効です.

11.2.4 表示領域に関する項目

大域変数 plot_options の一般的な項目

項目	既定値	概要
grid	[grid 30, 30]	3 次元グラフの解像度を指定
nticks	[nticks, 10]	2 次元グラフの解像度を指定
x	[x,-a,a]	a はシステムによって異なる浮動小数点数. 表示可能な領域の目安
y	[y,-a,a]	a はシステムによって異なる浮動小数点数. 表示可能な領域の目安
t	[t,-3,3]	助変数表示に於ける助変数の定義域

grid: 3 次元グラフの解像度を定め, その初期値は [30,30] です. grid の値は X(横) 方向と Y(縦) 方向の解像度の対で記述し, [grid,50] のように記述できません. 必ず, '[grid,50,50]' のように X と Y の解像度を指定します. 描いた曲線が粗い場合, この grid により大きな整数値を設定します.

nticks: 2 次元グラフの解像度を定め, その既定値は '10' です. 描いた曲線が粗い場合, この nticks の値を大きくすると良いでしょう. 特に媒介変数式の表示では初期値を予め大きく指定しなければ綺麗な曲線は描けないでしょう.

x と y: x と y に設定される値は Maxima の土台にある Common Lisp 処理系の '(/ most-positive-double-float 1024)' の処理結果となります. この値は Common Lisp で扱える数値の限界を示すので, この値を越える数値を Maxima は扱えず, 当然, グラフ表示もできません. なお, グラフ表示可能な数値の上限は Maxima で利用可能なグラフ表示アプリケーション毎に異なるので, ここでの設定以下であっても表示が出来ない場合もあります. たとえば, gnuplot で 'sqrt(x)' のグラフはこの値以下であれば表示できますが, openmath では 10^{203} よりも大きな数値の表示できません.

t:　函数の媒介変数式で利用する変数 t の定義域を定めます. Maxima では媒介変数式の変数名は t に固定されています. この定義域を定めておけばグラフ表示で媒介変数の定義域が省略できます.

11.2.5　描画に直接関連するフラグ

描画に直接関連するフラグ

項目	既定値	概要
transform_xy	[transform_xy, false]	3 次元グラフ表示で座標変換を行うかどうかを指定するフラグ
logx	[logx, false]	2 次元グラフでの X 座標の対数目盛フラグ
logy	[logy, false]	2 次元グラフでの Y 座標の対数目盛フラグ
plot_realpart	[plot_realpart, false]	複素函数実部の表示フラグ
adapt_depth	[adapt_depth, 10]	

transform_xy:　初期値のデカルト座標系から別の座標系への座標変換を指定します. ここで設定可能な値はデカルト座標系であれば 'false', 極座標系を用いる場合には 'polar_to_xy' , あるいは利用者定義の変換函数となります. ただし, この場合は make_tranform 函数を用いる必要があります. この make_transform 函数 の構文を示しておきます:

make_transform 函数の構文

make_transform(⟨ 変数リスト ⟩,⟨f_x⟩,⟨f_y⟩,⟨f_z⟩)

ここで f_x, f_y, f_z はデカルト座標系の X, Y, Z 成分に対応する ⟨ 変数リスト ⟩ に含まれる変数の函数です.
たとえば円筒座標の場合, 'make_transform([r,th,z],r*cos(th),r*sin(th),z)' としますが, これだけでは分り難いので実例も示しておきましょう:

```
(%i6) neko(r,th,z):=make_transform([r,th,z],
r*cos(%pi*th/180),r*sin(%pi*th/180),z);
                                                    %pi th
(%o6) neko(r, th, z) := make_transform([r, th, z], r cos(------),
                                                     180
                                                    %pi th
                                                 r sin(------), z)
                                                     180
(%i7) plot3d(r*th^2,[r,1,2],[th,0,360],[gnuplot_pm3d,true],
[transform_xy,neko(r,th,z)]);
```

この例では plot3d 函数のオプションとして '[transform_xy, neko(r, th, z)]' を引渡していますが, これを '[transform_, make_transform([r, th, z],

r*cos(%pi*th/180), r*sin(%pi*th/180), z)]' にしても同じ結果になります. ここで, 通常の polar_to_xy の角度は弧度法になりますが, 函数 neko では角度を用いています.
注意点として [transform_xy, neko] のような利用者定義の変換函数に対しては函数名のみの記述は許容されず, 必ず [transform_xy, neko(r, th, z)] のように変数も含めて記述しなければなりません. ただし, polar_to_xy については [transform_xy, polar_to_xy] と変数が省略できます. これは polar_to_xy の実体が plot.lisp 内部で定義された LISP の函数だからです.
では, 上記の実行結果を次に示しておきましょう:

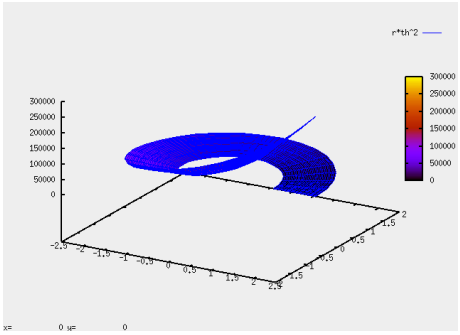


図 11.4: transform_xy と make_transform の組合せ

logx と logy: 設定可能な値は true か false で true の場合に対応する軸の目盛を対数目盛にします. この設定は 2 次元グラフの場合に有効で, 3 次元グラフでは無効になります.

11.2.6 gnuplot に関連する項目

Maxima は gnuplot を標準の描画アプリケーションとしています. そのために gnuplot を制御する項目が沢山存在します:

gnuplot の制御に関連する項目

項目	既定値	概要
gnuplot_out_file	[gnuplot_out_file, false]	gnuplot の画像出力ファイルを指定.
gnuplot_term	[gnuplot_term, default]	gnuplot の term を設定し, 対応するデータを出力する.
gnuplot_default_term_command	[gnuplot_default_term_command, set term x11 font "Helvetica,16"]	gnuplot で実行する term に関連する命令文を記述する
gnuplot_dumb_term_command	[gnuplot_dumb_term_command, set term dumb 79 22]	term を dumb とした場合の設定
gnuplot_ps_term_command	[gnuplot_ps_term_command, set size 1.5, 1.5; set term postscript eps enhanced color solid 24]	PS ファイルに追加する命令
gnuplot_pipes_term	x11	gnuplot の端末を指定
gnuplot_preamble	[gnuplot_preamble,]	gnuplot に引渡す諸設定を文字列で指定

gnuplot_out_file: gnuplot で提供する画像形式を指定することができます. この場合に gnuplot で指定された画像形式のファイルを出力します. 画像ファイルを出力する必要がない場合, 'false' を指定します.

gnuplot_default_term_command: フォントを Helvetica, 文字の大きさを 16 ポイントとする gnuplot の命令が初期値として入っています.

gnuplot_term: gnuplot の出力端末の設定を行います. ここで指定した端末に対応し与件を出力し, gnuplot 内部では 'set term' による端末の設定が行われます. 設定可能な端末の型には X 端末に対応する X11, MacOS の aqua と gnuplot の画像を表示する際に必要な端末の情報, postscript や gif といった画像与件等と非常に多いために, ここでは一々説明しません. この端末の詳細は参考文献 [61] か, gnuplot 上で ? term と入力して起動するオンラインマニュアルにて 'term' に設定可能な値の一覧が表示されるので, そちらを参照して下さい.
なお, gnuplot_term に 'default' 以外の値を設定した場合, Maxima で描画を行うと gnuplot に引渡すデータファイル (maxout.gnuplot, あるいは maxout.gnuplot_pipes) に加え, 別途, グラフ画像の与件ファイルが出力されます. このファイル名は "maxplot.〈 端末名 〉" といった名前になります. たとえば, terminal として tgif を設定した場合で解説しましょう.
このときにグラフ出力時に 'set term tgif' と ' set out '/home/ponpoko/maxplot.tgif/' '(ここで /home/ponpoko/ がホームディレクトリです) が gnuplot に設定されます. 同時にグラフ出力ファイルとして maxplot.tgif が生成され, 画像データが書込まれます. なお, maxplot.tgif を maxplot.obj に変更して tgif に読込ませた結果を図 11.5 に示しておきます:

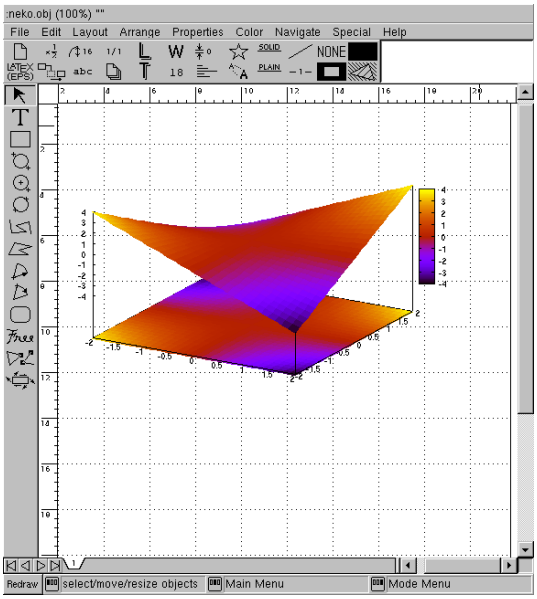


図 11.5: term を tgif にした結果 (maxplot.tgif) を tgif で表示

gnuplot_dumb_term_command: gnuplot の term に dumb を選択した時の設定が行えます. 以下に実例を示しておきます:

```
(%i86) plot2d([sin(x),cos(x)],[x,0,10],[gnuplot_term,dumb],
[gnuplot_dumb_term_command,["set term dumb 70 20"]]);
```

```
    1 ***----$$$$$$-----------+-----------*****-----$$$$$---------++
      + *** $$    $$          +          **+   ** $$$sin(x) $$$$$$ +
  0.8 ++  *$$       $                   **      *$$  cos(x)$******++
  0.6 ++  $$*        $                **        $$*        $$     ++
      |  $$ **        $               *        $$ **        $$     |
  0.4 ++$$   **        $             *        $$   **        $    ++
  0.2 +$$     *        $$           **        $     **        $   ++
      |$       *        $$         **        $       *        $$   |
    0 $+       **        $        **        $$        *        $$ ++
      |         **        $       *        $$         **        $$ |
 -0.2 ++         *        $$     *        $$           **        $++
 -0.4 ++          *        $$   **        $             **       $$+
      |           **        $$ **        $               *        $$
 -0.6 ++           **        $$*        $$                *       ++
 -0.8 ++            **       *$$       $$                  **     ++
      +           +  ***   ***+$$    $$    +           +    **    **
   -1 ++----------+----*****--+--$$$$$-----+-----------+-----*****++
      0           2           4           6           8           10

Output file "/home/yokota/maxplot.dumb".
(%o86)
```

この例では dumb 端末を 70 列 20 行として表示しています. なお, term を dumb にしなければこの設定は無効です.

gnuplot_pipes_term:　gnuplot が用いる端末 (terminal に相当) を指定します.　この変数は Maxima-5.12.0 から導入されています.　なお, MS-Windows 環境でも初期値として x11 が指定されています. 通常, この値を変更する必要はないでしょう.

gnuplot_preamble:　これは gnuplot の制御文を gnuplot に直接引渡すための仕組みで, gnuplot の制御文の間にはセミコロン ";" を入れた Maxima の文字列を指定します. この preamble の設定は plot_option で gnuplot に関連する設定と gnuplot の数値与件の間に置かれます.
preamble の内容がどのように書込まれるかを plot_format を gnuplot に設定した状態で, 次の描画をさせて maxout.gnuplot の内容を調べてみましょう[2].

```
(%i4) nekoneko:"set title 'mike';set xlabel 'X';\
set ylabel 'Y';set zlabel 'height';";
(%o4) set title 'mike';set xlabel 'X';set ylabel 'Y';set zlabel 'height';
(%i5) plot3d(sin(x*y),[x,0,10],[y,0,10],
[gnuplot_pm3d,true],[gnuplot_preamble,nekoneko]);
```

次に maxout.gnuplot の先頭部分を示しておきます:

gnuplot_preamble を反映した maxout.gnuplot の先頭部分

```
set pm3d

set title 'mike';set xlabel 'X';set ylabel 'Y';set zlabel 'height';
splot '-' title 'sin(x*y)' with lines 3
0.0      0.0      0.0
0.3333333333333333      0.0      0.0
0.6666666666666666      0.0      0.0
1.       0.0      0.0
```

gnuplot の splot 命令の直前に gnuplot_preamble の内容が書込まれていることが分りますね. この性質を利用すれば gnuplot の諸設定が容易に行えるのです:

[2]UNIX 環境で Maxima-5.12.0 以降の方は plot_format を gnuplot に変更して試して下さい.

11.2.7 gnuplot の描画に直接関連する項目

gnuplot の描画に直接関連する項目

項目	既定値	概要
gnuplot_curve_titles	[gnuplot_curve_titles, [default]]	描画する曲線の表題を設定.
gnuplot_curve_styles	[gnuplot_curve_styles, [with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,with lines 6, with lines 7]]	曲線の様式を指定
gnuplot_pm3d	[gnuplot_pm3d, true]	gnuplot の PM3D(曲面に面を張るかどうか) のオプション

gnuplot_curve_titles: 曲線や曲面の表題をリストで追加します. その構文は gnulot の plot 命令や splot 命令に引渡す表題の設定に準じます. たとえば, 二曲線に A1, A2 と設定したい場合に 'gnuplot_curve_titles,["title 'A1'","title 'A2'"]]' とします.

gnuplot_curve_styles: 線分の書式を設定します. 具体的には gnuplot の with 命令文を文字列として引渡すために用います.
たとえば, '[gnuplot_curve_styles,["with lines 7","with lines 2"]]' のようにします. ちなみに maxout.gnuplot 内部では gnuplot の描画命令 plot のオプションとして引渡されます. そのために後述の gnuplot_preamble で曲線のスタイルを幾ら設定しても, こちらの設定が優先されるので注意が必要です.

gnuplot_pm3d: gnuplot の pm3d を有効にします. 初期値は false なので曲面はワイヤーフレーム表示ですが, ここに 'true' を設定すれば, gnuplot で 'set pm3d' が指定されるので曲面が張られます. なお, ここでの曲面は裏面が透けて見えるように設定されているので, 向き付けの出来ない Klein の壺のような曲面の表示は図 2.11 のように裏面が欠けて表示されます.
この pm3d のオプションとして 'b', 's', 't' といった文字で構成された語も与えられます.
これらの文字の意味については §11.5.5 で改めて解説します.
最初に pm3d を有効にした例を示しておきましょう:

pm3d の例

```
plot3d(sin(x*y),[x,-3,3],[y,-3,3],[gnuplot\_{}pm3d,true],[grid,50,40])
```

次に, 底面への投影を意味する文字 b と曲面の描画を意味する 's' を組合せた語 "bs" を '[gnuplot_pm3d,bs]' で gnuplot に引渡します:

gnuplot_pm3d の設定例

```
plot3d(sin(x*y),[x,-3,3],[y,-3,3],
[gnuolot_pm3d,bs],[gnuplot_preamble,"unset surf"],[grid,50,40])
```

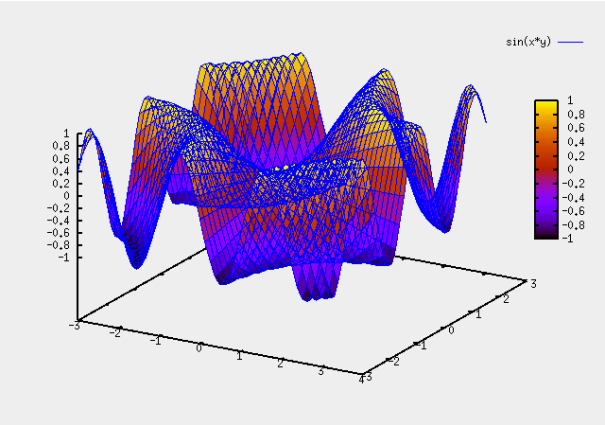


図 11.6: pm3d の例

この例では gnuplot の pm3d のオプションとして語 “bs” の他に gnuplot_preamble を用いて gnuplot に ‘unset surf’ を処理させます. この命令文 ‘unset surf’ は曲面上の網目を消す効果があります. 図 11.7 にその結果を示します:

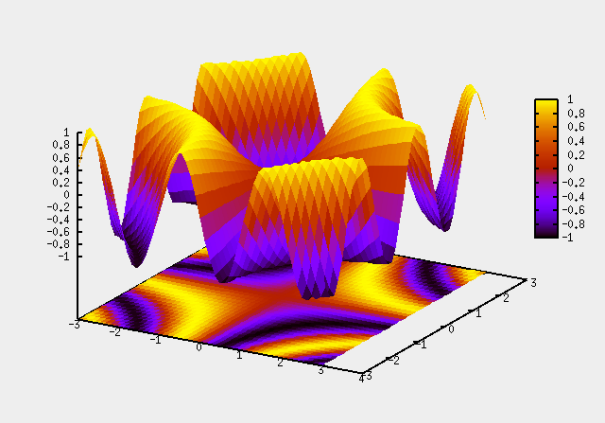


図 11.7: gnuplot_preamble を使った例

この図 11.7 は gnuplot_preamble をだけを用いても描けます:

gnuplot_preamble を使った例

```
plot3d(sin(x*y),[x,-3,3],[y,-3,3],
[gnuplot_preamble,"set pm3d at bs;unset surf"],[grid,50,40])
```

11.2.8 gnuplot との連動に関連する函数

gnuplot に関連した函数
gnuplot_start()
gnuplot_restart( )
gnuplot_close()
gnuplot_pipes
gnuplot_reset()
gnuplot_repolot(⟨ オプション ⟩)

gnuplot_start 函数: 引数を必要としない函数です. MS-Windows 環境でこの函数を実行すると gnuplot が立ち上がります.

gnuplot_restart 函数: 内部的に gnuplot_close 函数を実行し, それから gnuplot_start 函数を実行するだけの函数です. そのためにグラフを表示している場合, グラフのウィンドウが消滅し, MS-Windows 環境であれば gnuplot のウィンドウが現われます.

gnuplot_reset 函数: 引数を必要としない函数で, gnuplot で設定した諸設定を初期化します. この函数をグラフ表示中に実行すると, それ迄のマウス操作による変更が全て無効になって最初の設定のグラフが表示されます.

gnuplot_replot 函数: その名前の示すように gnuplot で再描画を行う函数です.
なお, 'gnuplot_reset()' を最初に実行して 'gnuplot_replot()' を次に実行すれば, マウスによる操作が無効になって最初に描画したグラフが再描画されます.

11.3 その他の描画函数

Maxima の描画函数には上述の plot2d 函数や plot3d 函数の他に幾つかの描画函数があります. とは言え, どちらかと言えば補助的な函数が多く, それも openmath 専用や PostScript への出力のみといった機能や出力が限定されたものが殆どです.
最初に比較的, 汎用性のある openplot_curves 函数の解説をしましょう.

11.3.1 openplot_curves

openplot_plot 函数は大域変数 plot_options の plot_format とは無関係に openmath を使って与えられた点列の描画を行う函数です. とは言え, オプションを変更すると実数値 1 変数函数の描画も可能です.

openplot_curves

openplot_curves ([⟨ 点列リスト$_1$⟩,...,⟨ 点列リスト$_n$⟩])
openplot_cur ves ([⟨ オプション$_1$⟩,⟨ 点列リスト$_1$⟩,...,...,⟨ オプション$_n$⟩,⟨ 点列リスト$_n$⟩)
openplot_curves (⟨ 点列リスト ⟩)
openplot_curves (⟨ オプション ⟩,⟨ 点列リスト ⟩)
openplot_curves (⟨$xfun$ を含むオプション ⟩)

ここでの点列リストの書式は, '$[x_1, y_1, \ldots, x_n, y_n]$' か '$[[x_1, y_1], \ldots, [x_n, y_n]]$' の二種類に限定されます. openplot_curves のオプションは plot_options とは無関係で, openmath のメニューからも設定可能な事項になります. また, 点列リストの直前にあるオプションがその点列の表示で用いられます. openplot_curves の設定項目を以下に示しておきます:

openplot_curves の設定項目

項目	既定値	概要
xfun	なし	指定した変数 x を持つ式を指定
color	blue から開始	曲線の色を指定
plotpoints	0	直線や曲線上に点を打つ為のフラグ. 初期値は 0
linecolors	blue	直線/曲線の色を指定.
pointsize	0	点の大きさを指定
nolines	0	点列を繋ぐ線分表示のためのフラグ
bargraph	0	棒グラフ表示への切替のためのフラグ
xaxislabel	無指定	X 軸のラベルを指定
yaxislabel	無指定	Y 軸のラベルを指定

この openplot_curves のオプションには独特の書式があります. これは plot_options に似たもので, '{ 項目 値 }' の書式の文字列を空白文字で区切って全体を引用符で括ったリストになります. たとえば各点を表示して, その点の大きさを 6 にする場合は '["{plotpoints 1} {pointsize 6}"]' をオプションとして与えます.
次の例では折線毎に点の大きさと折線と点の色を変更し, 各軸にラベルを設定しています:

```
openplot_curves([["{plotpoints 1} {pointsize 6} {color red}"],
[1,2,3,4,5,1],
["{pointsize 10} {xaxislabel time} {yaxislabel neko} {color black}"],
[10,9,9,2,4,2]])
```

xfun を指定することで Maxima の式をグラフに追加できます. 先程の例に正弦函数を追加した例を示しておきましょう:

```
openplot_curves([["{plotpoints 1} {pointsize 6} {color red}"],
[1,2,3,4,5,1],
["{pointsize 10} {xaxislabel time} {yaxislabel neko} {color black}"],
[10,9,9,2,4,2],
["{xfun sin(x)} {color green} {plotpoints 1} {pointsize 1}"]])
```

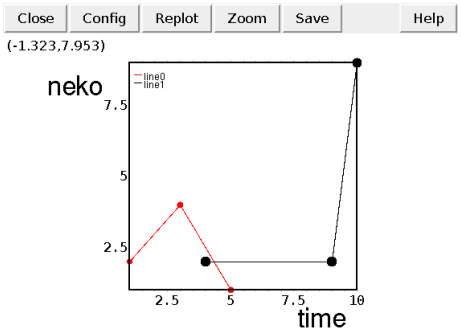


図 11.8: openplot_curves のオプション例

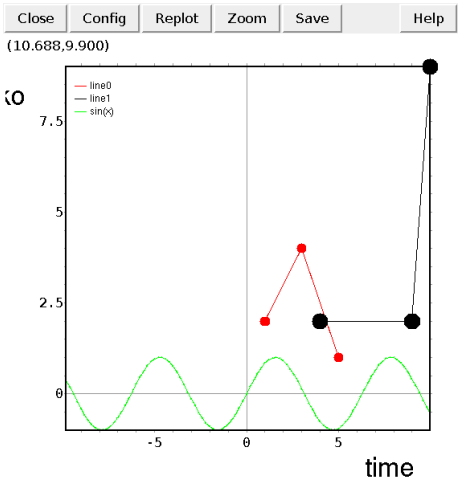


図 11.9: openplot_curves のオプション例 (その 2)

ここで注意することは描画する式の変数は x に限定されることです. また, 函数の描画だけを行いたければ, 点列リストを持たないオプションのみの引数を与えます. ただし, openplot_curves 函数で式の定義域は与えられないようです.

11.3.2　contour_plot

contour_plot 函数の構文

contour_plot(⟨ 式 ⟩)
contour_plot(⟨ 式 ⟩,⟨ オプション ⟩)

contour_plot は与えられた 2 変数実数値函数の等高線を描画する函数です. オプションは plot_format と gnuplot_preamble に限定されます. したがって, グラフにいろいろな細工を行いたければ plot_format は gnuplot でなければなりません.

11.3.3　Postscript に関連する函数

Postscript に関連する函数

plot2d_ps(⟨ 式 ⟩,⟨ 定義域 ⟩)
viewps(⟨PostScript ファイル名 ⟩)
viewps()
psdraw_curve (⟨ 点列リスト ⟩)
psdraw_points (⟨ 点列リスト ⟩)
pscom(⟨PostScript 命令文 ⟩)
closeps()

plot2d_ps 函数:　函数と定義域の二つの引数のみを取る plot2d 函数に似た構文を持っています. ただし, デカルト座標系で函数の表示が可能です. この函数はカレントディレクトリ上に maxout.ps ファイルを生成し, 内部で viewps 函数を呼出して maxout.ps ファイルの表示を行います.

viewps 函数:　実体は ghostview に PostScript ファイルを引渡す函数です. ファイル名を指定しない場合にホームディレクトリ上の maxout.ps ファイルの表示を行います. 当然, ghostview 命令を持たない環境でこの函数は使えません.

psdraw_curve 函数:　点列リストのグラフを PostScript 形式で出力します. ここで点列リストの書式は '$[x_1, y_1, x_2, y_2, \ldots, x_n, y_n]$' と '$[[x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n]]$' の何れかになります. psdraw_curve 函数を実行するとホームディレクトリ上の maxout.ps へのストリームを開きます. このストリームは closeps() で閉じられます.

psdraw_points 函数:　点列リストのグラフを PostScript 形式のファイルで出力します. この函数は psdraw_curve 函数に似ていますが, この函数の引数となる点列リストの書式は '$[[x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n]]$' のみに限定されます. psdraw_curve 函数と同様に psdraw_points 函数を実行するとホームディレクトリ上のファイル maxout.ps へのストリームが開かれます. また, closeps() を実行すると, ストリームが閉じられて maxout.ps ファイルが生成されます.

pscom 函数と closeps 函数: 与えられた PostScript の命令文をホームディレクトリ上の maxout.ps へのストリームに出力する函数です.
なお, これらの psdraw_curve 函数,psdraw_points 函数と pscom 函数は原始的な函数で, これらは全てホームディレクトリ上の maxout.ps へのストリームを開いて, そのストリームに対して出力を行う函数です. そして, closeps() 函数を実行することでストリームが閉じられ, maxout.ps ファイルが生成されます.

大域変数 pstream: ストリームが開いているかどうかは大域変数 pstream で調べられます. ストリームが開いていない場合は pstream には 'false' が割当てられ, 開いていればストリームの情報を返します.
実際に pstream の働きを見ておきましょう.

```
(%i32) psdraw_curve([[1,2],[3,3],[5,1]]);
(%o32)                                false
(%i33) pstream;
(%o33)  #<OUTPUT BUFFERED FILE-STREAM CHARACTER /home/yokota/maxout.ps>
(%i34) closeps();
(%o34)                                true
(%i35) pstream;
(%o35)                                false
```

この例では最初に psdraw_curve 函数を実行することでホームディレクトリ (ここでは/home/yokota) 上のファイル maxout.ps へのストリームを開きます. このことは pstream を見ることで maxout.ps へのストリームが開いていることが分ります. それから closeps() を実行したあとに pstream の値を見ると false になっているのでストリームが閉じていることが分ります.

11.4 plot_option 以外の描画に関連する大域変数

Maxima の描画函数で最も重要な大域変数としては plot_options がありました. Maxima にはこの大域変数の他にも多くの大域変数があります. ただし, これらの変数はどちらかと言えば Maxima の描画函数で内部的に用いることを想定したものが多くて用途も特殊なものが多いのが実情です. そのためにここでは一覧表に纏めて解説しておきます.
ちなみに以下で解説する変数は Maxima のソースファイル plot.lisp の中を見れば色々あることが分ります.

plot_option 以外の描画に関連する大域変数

変数名	既定値	概要
in_netmath	false	plot_format の値が openmath の場合に限って maxout.openmath の内容を画面に表示
show_openplot	false	plot_format が openmath の場合は true, それ以外は false
viewps_command	(ghostview ″ a″)	plot_format を ps にした場合の, PS ファイルの表示命令を設定
ps_scale	[72,72]	PS ファイルの解像度
ps_translate	[0,0]	原点の移動に利用
window_size	[612.0,792]	PS ファイルのウィンドウの大きさを指定

11.5 gnuplot による描画について

ここでは Maxima を使う上で必要な gnuplot の事項のみを記述します. そのために gnuplot 単体の解説ではなく, Maxima から見た gnuplot の解説になります. そこで gnuplot 全般の使用方法や例題を知りたければ「使いこなす GNUOPLOT」 [61] といった gnuplot の解説本や gnuplot に付属するマニュアル等の文献を参照して下さい.

11.5.1 maxout.gnuplot の内容について

maxout.gnuplot ファイルは plot_format を gnuplot に指定した場合に生成されるファイルで, その先頭には数値与件を描画するための gnuplot の命令文とさまざまな設定文が置かれ, ファイルの末尾に描画される式の数値データが記述されています. そのため, 別途, gnuplot を起動して load "maxout.gnuplo"t と gnuplot の入力行に入力すると Maxima から処理したのと同じグラフが表示できます[3].
このことは Maxima 側で行ったグラフの諸設定がどのように maxout.gnuplot に反映されているかを観察すれば, gnuplot での実際の設定や処理が行われているかを理解することが容易になります. そこで, 手始に plot2d 函数による maxout.gnuplot の構造を簡単に解説しておきましょう:

plot2d 函数による maxout.gnuplot

```
set log x          /* gnuplot_logxがtrueの場合に記述 */
set log y          /* gnuplot_logyがtrueの場合に記述 */

<gnuplot_preambleの内容>
plot '-' <gnuplot_curve_titlesの内容> <gnuplot_curve_stylesの内容>
<曲線の数値与件>
```

ここで示すように gnuplot_logx と gnuplot_logy の設定は plot 命令の前に記述されます. さらに plot 命令の直前の行には gnuplot_preamble の内容が書込まれます. そして, plot 命令の直後の行に描画するデータが置かれます. ここで " plot '-' " という文がありますが, この次の行にある数値与件の表示を gnuplot に指示している文です. そして, 曲線の表題や曲線の型の指定が並んでいます.
さて, 今度は plot3d 函数による maxout.gnuplot の構造を示しておきましょう. plot3d 函数で生成した maxout.gnuplot も plot2d 函数と似通った書式になります:

plot3d 函数による maxout.gnuplot

```
set pm3d           /* gnuplot_pm3dがtrueの場合に記述 */

<gnuplot_preambleの内容>
splot '-' <gnuplot_curve_titlesの内容> <gnuplot_curve_stylesの内容>
<曲面の数値与件>
```

今度は gnuplot_pm3d に対応する命令文 'set pm3d' がファイルの先頭に置かれていますね. その他は plot2d 函数で生成した maxout.gnuplot と似ていますが, 描画命令が plot ではなく splot 命令になっていることに注意して下さい.

[3] UNIX 環境では plot_format を gnuplot に変更した場合.

これらのファイルの内容を眺めるだけでも plot 命令や splot 命令が gnuplot の描画命令であることが何となく分りますね. これらの描画命令の前に set 命令から開始して, xlog, ylog や pm3d と続く命令文があります. これらは gnuplot の設定を行う命令文で, まず, set 命令でいろいろな設定を行います. この設定内容は show 命令で確認が行え, unset 命令で設定内容を無効にできます.
さて, Maxima から単純に plot2d 函数や plot3d 函数を用いて maxout.gnuplot を生成しても 'set pm3d at bs' のような gnuplot の命令文の埋込みはできません. gnuplot から直接描画する場合, このような命令文を入れて replot 命令で再描画すれば良いのですが, Maxima から操作したければ plot_format を gnuplot_pipes に設定した場合のみ有効になります. また, MS-Windows 版では gnuplot_pipes が選べないために, この身軽な操作自体ができません.
より高度なグラフ表示を行う場合, 直接 gnuplot の命令文が書込める gnuplot_preamble にいろいろな設定をすると結構楽に処理が行えます. このことからも gnuplot を利用する場合に gnuplot_preamble の使いこなしが非常に重要なことが理解できるでしょう.
それでは先程の maxout.gnuplot に表われた命令の意味を簡単に紹介しましょう.

11.5.2　set 命令

この set 命令は前述のように gnuplot のいろいろな設定を行う命令です. この set 命令で設定した内容は show 命令を使って確認できます.
set 命令による設定を無効したければ unset 命令を用います. これらの命令は gnuplot で非常に重要な命令です.
set 命令, show 命令と unset 命令の基本的な構文を次に示しておきます:

set 命令, show 命令, unset 命令の構文

構文	概要
set ⟨ 項目 ⟩	⟨ 項目 ⟩ を有効にする
set ⟨ 項目 ⟩ ⟨ 設定内容 ⟩	⟨ 項目 ⟩ に ⟨ 設定内容 ⟩ を設定する
show ⟨ 項目 ⟩	⟨ 項目 ⟩ の内容を表示
unset ⟨ 項目 ⟩	⟨ 項目 ⟩ を無効にする

ここで示すように set 命令には単純に項目を指定する場合と, その項目に対して値を指定する場合の二種類があります. この具体的な例は Maxima の諸設定と絡めて詳細を述べましょう.

11.5.3　plot 命令による曲線の表示

gnuplot の plot 命令は変数 x の式や 2 次元の数値与件の描画描画に使えます. この plot 命令の構文を示しておきます:

plot 命令の基本的な構文

plot ⟨ 表示範囲 ⟩⟨ 式 ⟩ title ⟨ 文字列 ⟩ with ⟨ 曲線の様式 ⟩
plot ⟨ 式 ⟩ axes ⟨ 軸の設定 ⟩ title ⟨ 文字列 ⟩ with ⟨ 曲線の様式 ⟩
plot ⟨ 式 ⟩ title ⟨ 文字列 ⟩
plot ⟨ 式 ⟩
plot ⟨ 与件ファイル名 ⟩ title ⟨ 文字列 ⟩ with ⟨ 曲線の様式 ⟩
plot ⟨ 与件ファイル名 ⟩ title ⟨ 文字列 ⟩
plot ⟨ 与件ファイル名 ⟩

表示範囲: 表示範囲は 'plot [-3:3] [-2:2] cos(x)' のように定義域と値域の順番で MATLAB 風のベクトル表記で記述します.
ここで 'plot [-3:3]' のように "[-3:3]" だけを設定すると, "[-3:3]" が式の定義域になり, グラフの値域は式全体が収まるように gnuplot 側で自律的に調整します.
式の表示範囲は複数の式を表示する場合でも, 一つの plot 函数に対して一つだけが設定できます.

gnuplot の式: gnuplot で表記可能な数式, あるいは与件ファイル名や記号 "-" を設定します. ここでの数式は変数 x の式に限定されます.
複数の式を同時に表示させる場合, 式とその式に対する設定を一組としてコンマで区切った式の列を与えます.
たとえば, 正弦函数と余弦函数を同時に表示するときに最も簡単な表記は 'plot sin(x),cos(x)' ですが, 曲線の名前や曲線の様式を指定する場合は次のように入力します:

```
plot sin(x) title 'Sin' with lines 1,cos(x) title 'Cos' with points 5
```

与件ファイル: 与件ファイルの指定では引用符や二重引用符で括られた文字列を用います. なお, Maxima から gnuplot を使って描画する場合, ファイル名やラベルの指定の注意事項として, gnuplot の命令文中の文字列は単引用符で括らなければなりません. たとえば, 曲線の表題を "test" にしたければ, Maxima の大域変数 plot_options の項目 gnuplot_curve_titles の設定を '[gnuplot_curve_titles,"title 'test'"]' で行います. つまり, 二重引用符で括った場合は二重引用符内部の文字の列に二重引用符が出現すれば文字列がそこで一旦途切れてしまいますが, 単引用符の場合は途切れずに gnuplot に引渡されるからです.
あるファイルに plot 命令と与件本体を記載する場合, plot 命令で指定するファイル名の箇所を'-' と記述すれば plot 命令の次の行から開始する与件列の読込を行ってファイルの EOF を検出した時点で描画を行います. この方法が maxout.gnuplot で採用されています.

式と与件ファイルの混在: この場合は引数として数式と与件ファイルをコンマ "," で区切った列を第一引数として引き渡します. たとえば, 先程の正弦函数と余弦函数に加えて与件ファイル test1 の与件を表示させたい場合には 'plot sin(x),cos(x),'test1'' のように入力します.

曲線名の指定:　曲線名の指定は title のうしろに文字列を置くことで行います. この曲線名は gnuplot の右上に表示される凡例 (key) で用いられます. Maxima では大域変数 plot_options の項目 gnuplot_curve_titles の内容がここに記入されます. ここで凡例を調整する場合, 'set key' で行います.
なお, Maxima から gnuplot を利用する場合, この gnuplot_curve_titles の内容が標準入力を示す記号 "-" の直後に置かれます. そこで, このことを利用した阿漕な使い方として別のグラフ与件を指定したり, gnuplot の式を入れることが挙げられます:

gnuplot_curve_titles の阿漕な使い方

```
plot2d(1/x,[x,1.0e-8,1.0e-5],[gnuplot_curve_titles,
"title 'Maxima',1/x title 'GNUPlot' "]);
```

この例では $\frac{1}{x}$ のグラフを描きますが, その一方は Maxima で描いたもので, もう一方は図 11.10 に示す図を描画します:

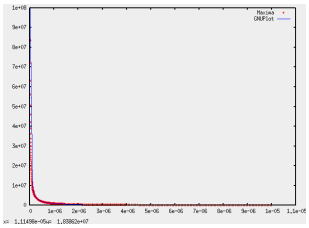


図 11.10: gnuplot_curve_titles の応用で gnuplot の式も描く

gnuplot の式は [gnuplot_curve_titles,"title 'Maxima',1/x title 'GNUPlot'"] で与えていることに注意して下さい. この処理で生成されたファイル maxout.gnuplot のヘッダを次に示しておきます:

ファイル maxout.gnuplot の様子

```
plot '-' title 'Maxima',1/x title 'GNUPlot' with lines 3
1.00000000E-8 100000000.
1.195117187500000100E-8 83673802.90897205
1.390234375000000400000000E-8 71930317.5049171
1.585351562500000600000000E-8 63077491.68411973
1.780468750000000500000000E-8 56164984.642386995
1.975585937500000700000000E-8 50617894.216510125
2.170703125000000600000000E-8 46068022.3141983
2.365820312500000500000000E-8 42268637.001568556
2.560937500000000600000000E-8 39048200.12202562
---- 以下略 ----
```

ここで示すように曲線の書式は with のうしろに設定されます. Maxima では gnuplot_curve_styles に "with" から開始する文字列を設定すると, maxout.gnuplot にその文字列が記入されます. 通常の実線で色を赤にするためには 'with lines 1' のように記述します.

11.5.4 splot 命令による曲面の表示

splot 命令は数値与件や変数 x, y の式で表現された曲面を描く命令です. その構文は plot 命令と同様で単純に plot を 3 次元に拡張した側面を持ちます:

splot 命令の基本的な構文

splot ⟨ 式 ⟩ title ⟨ 文字列 ⟩ with ⟨ 曲面の様式 ⟩
splot ⟨ 式 ⟩ title ⟨ 文字列 ⟩
splot ⟨ 式 ⟩
splot ⟨ 与件ファイル名 ⟩ ⟨ 文字列 ⟩ with ⟨ 曲面の様式 ⟩
splot ⟨ 与件ファイル名 ⟩ title ⟨ 文字列 ⟩
splot ⟨ 与件ファイル名 ⟩

曲面の表題, 曲面の様式 (実際はワイヤーフレームを構成する線分の様式に対応) , 式や与件ファイル名や文字列については plot 命令と違いはありません.
ここで曲面の様式には, lines, point, linespoints, dots, impulses がありますが, 曲面上に表示される網目に対するもので後述の pm3d に関連する項目に影響を及ぼしません.

11.5.5 pm3d

gnuplot では pm3d を有効にすることで面を貼った曲面表示が行えます[4]: この pm3d に関連する設定文を以下に纏めておきましょう:

pm3d の設定

set pm3d
set pm3d at b
set pm3d at s
set pm3d at t
set pm3d map
unset pm3d
show pm3d

set pm3d: pm3d を有効にします. Maxima では大域変数 plot_options の '[gnuplot_pm3d,true]' が対応します. ここで pmd3 のオプションは pm3d のうしろにある "at" で指定します.

set pm3d at b: 曲面の底面への投影を指定します. 後述の map と違って斜め上から曲面と投影の両方を眺める形になります. この指定のみの場合, 曲面本体をワイヤーフレーム表示にしています:

[4] gnuplot ver.3.8 以降から標準の機能となっています.

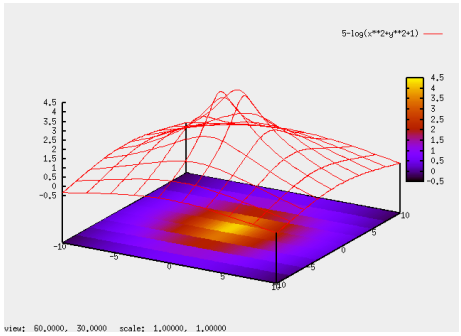


図 11.11: set pm3d at b の場合

set pm3d at s:　曲面に連続的に等高線で変化する色彩付けられた曲面が貼り付けられます. gnuplot の hidden3d を有効にしたり, surface を無効にしていなければ曲面とワイヤーフレームが同時に表示されます:

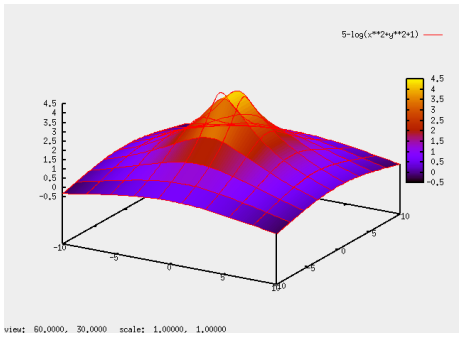


図 11.12: set pm3d at s の場合

set pm3d at t:　曲面の等高線表示の射影を上面に対して行います
この場合は射影が上側に行われることを除いて底面 'b' を指定した場合と違いはありません (勿論, 射影が上に出る違いはありますが…):

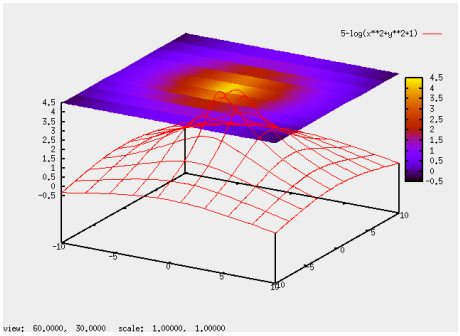


図 11.13: set pm3d at t の場合

set pm3d map: 図 11.14 に示す曲面の等高線図を表示します:

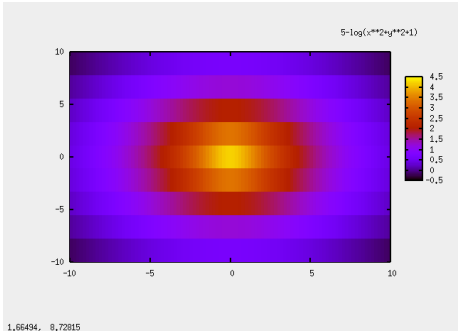


図 11.14: set pm3d map の場合

pm3d のオプションの組合せについて

pm3d のオプション 'b', 's', 't' を組合せて使えます. これは 'set pm3d at bs' のように "at" のうしろに文字 "b", "s" と "t" で構成した語を置くだけですが, 語の文字の順番にはちゃんと意味があります.

ここで GNUPLOT 上で ‘set pm3d at bs’ を処理した結果を図 11.15 に, 同様に ‘set pm3d at sb’ を処理した結果を図 11.16 に示しておきます:

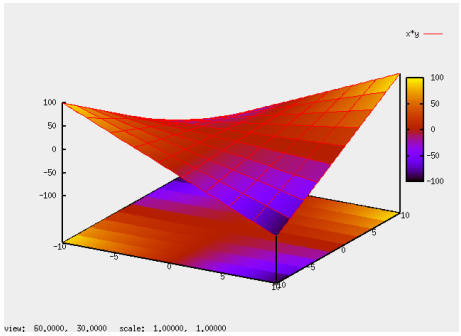


図 11.15: set pm3d at bs の場合

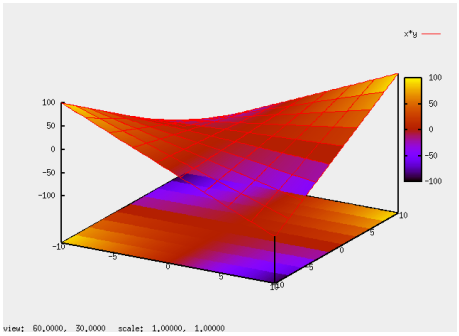


図 11.16: set pm3d at sb の場合

図 11.16 では底部の絵の一部が上にある筈の曲面に覆い被さっていますが, 図 11.15 では逆に底部の絵に曲面が覆い被さっていますね. この理由は:

- bs の場合: 最初に底面 (b) を描いてから曲面 (s) を描くために曲面が底面を覆う.
- sb の場合: 曲面 (s) から描いて次に底面 (b) を描くために底面が曲面を覆う.

すなわち, at のうしろに置く語では左から順番に描画が実行されることを意味します. したがって, pm3d のオプションを設定する場合, その描画順も考慮しなければなりません.

11.5.6　ticslevel による射影平面の調整

底面や上面への曲面の投影図の位置は ticslevel で設定出来ます:

投影面の位置指定の構文

set ticslevel 〈 数値 〉
show ticslevel

ticslevel に与える数値は $\dfrac{\text{投影面の Z 座標} - \text{曲面の最高値}}{\text{曲面の最低値} - \text{曲面の最高値}}$ に等しくなります.
すなわち, ‘0’ を指定した場合は投影面は底辺側, ‘-1’ を指定した場合, 投影面は頂部側になります. このことを $1 - \log\left(x^2 + y^2\right)$ のグラフで確認しましょう. なお, 違いが判り易いように図 11.17 では ‘set pm3d at bs’ として曲面が投影面を覆うようにし, 図 11.18 では ‘set pm3d at sb’ として投影面が曲面を覆うように設定しています:

11.5.7　等高線の階調の変更

曲面の等高線に沿った色彩階調は変更できます. この階調の変更は palette の設定で行います:

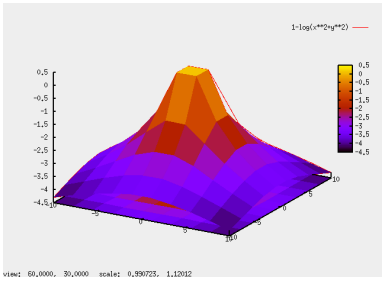


図 11.17: ticslevel が 0 の場合

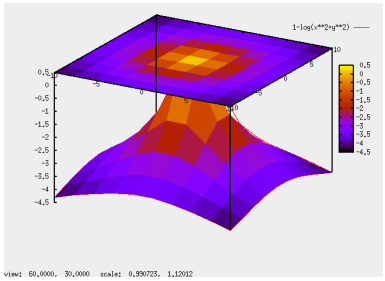


図 11.18: ticslevel が-1 の場合

palette 指定の構文

set palette color
set palette color positive
set palette color nagative
set palette gray
set palette gray positive
set palette gray nagative
set palette rgbformulae ⟨ 数値$_1$⟩,⟨ 数値$_2$⟩,⟨ 数値$_3$⟩
show palette

GNUPLOT の階調には color と gray の二種類があります. それから各階調には, 高い方がより明るくなる ‘positive’ とと逆に高い方が暗くなる ‘negative’ があります.
これらの階調の設定状況は ‘show palette’ で調べられます.
ここでは次の命令文を実行して得られたグラフを使って palette を変更してみましょう:

palette 変更例

gnuplot の命令文	概要
set hidden3d;	曲面上の網目を非表示に設定
set isosample 50;	解像度を 50x50 に設定
splot 10-log(sqrt(x**2+y**2+1));	gnuplot で曲面の描画を実行

ここで gnuplot では描画した曲線や曲面の設定変更のみの場合, replot で再描画が行えます.

set palette color positive と set palette color negative 図 11.19 に示すグラフが通常の等高線の階調で, 図 11.20 が階調を逆にしたものになります:

set palette gray positive と set palette gray negative 図 11.21 に通常の gray 階調, 図 11.22 に階調を逆にしたものを示します:

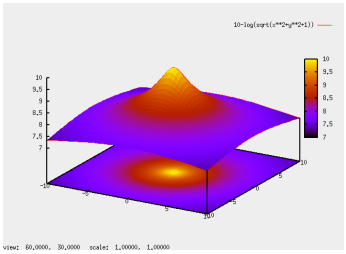


図 11.19: set palette color positive

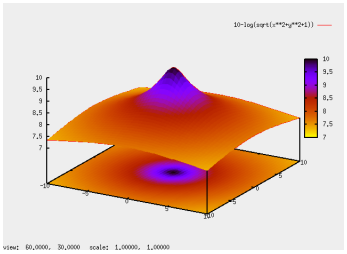


図 11.20: set palette color negative

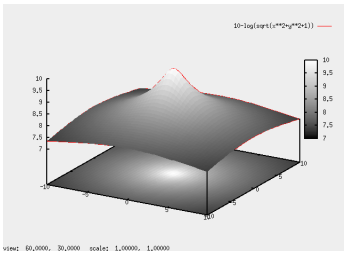


図 11.21: set palette gray positive

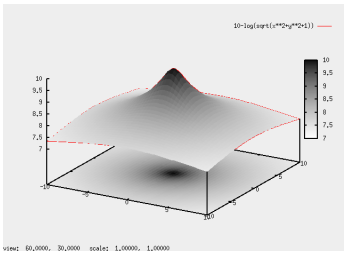


図 11.22: set palette gray negative

rgbformulae:　palette は色々な指定が可能です. たとえば, rgbformulae を使って次の図 11.23 に示すような階調も得られます:

11.5.8　等高線の表示

等高線表示自体は pm3d を有効にしなくても表示されます. この場合は本当に線だけになりますが, pm3d とも共存できます.

等高線の設定

set contour base
set contour surface
set contour both
set contour
show contour
unset contour

等高線は ‘set contour’ で有効になりますが, 単純に等高線の射影が描かれるだけで, 曲面の側に等高線は現われません. これは contour に ‘base’ を指定した場合と同値ですが, contour に ‘surface’ を指定すると, 今度は曲面の側だけに等高線が表示されます. 両方に等高線を表示させるためには ‘both’ を指定します.
このことを次の描画で確認してみましょう:

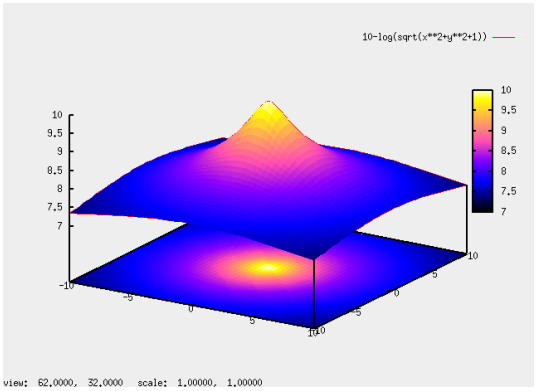


図 11.23: set palette rgbformulae 30,31,32;replot

等高線表示の例

命令文	概要
set isosamples 50;	解像度を 50x50 に設定
set hidden3d;	陰線処理の実行を指定
splot [-4:4] [-4:4] -x*y*cos(x**2+y**2);	函数を描画

set contour と set contour surface 図 11.24 に 'contour' のみ, 図 11.25 に 'surface' を指定した場合を示します:

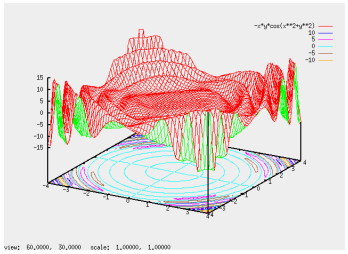


図 11.24: set contour

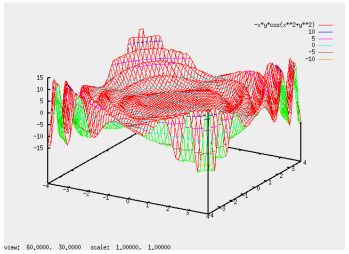


図 11.25: set contour surface

ここで 'unset surf' を実行すれば等高線だけが表示されます.

set contour both: 図 11.26 にその描画結果を示します:
この図 11.26 に示すように 'both' を指定すると曲面と投影面の両方に等高線が描かれています.

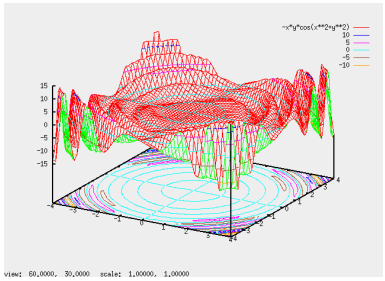


図 11.26: set contour both;replot

11.5.9　contour と pm3d の共存

contour の設定は pm3d の設定と共存できます. たとえば ‘set pm3d at s’ と ‘set contour both’ を設定したグラフの様子を次の入力で見てみましょう:

contour と pm3d の共存を確認する為の描画

入力	概要
set isosamples 100	解像度を 100x100 に設定
set hidden3d	陰線処理を実行
set contour	等高線の設定
set pm3d	pm3d の指定
unset surface	余計な網目の消去
splot -x*y*cos(sqrt(x**2+y**2))	曲線の描画

この結果が図 11.27 になります:

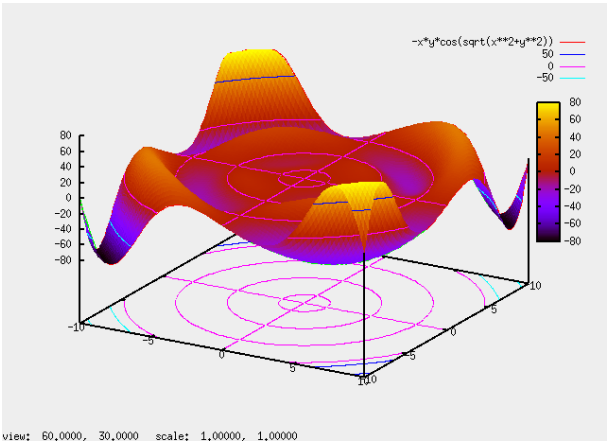


図 11.27: pm3d との共存

11.5.10 等高線の間隔調整

gnuplot では等高線の間隔の調整も行えます. この等高線の間隔等の微調整は cntrparam を設定して行います. 次にその構文を示しておきましょう:

等高線の調整を行う文

set cntrparam linear
set cntrparam cubicspline
set cntrparam bspline
set cntrparam points ⟨ 個数 ⟩
set cntrparam order ⟨ 次数 ⟩
set cntrparam levels ⟨ 等高線総数 ⟩
set cntrparam levels discrete ⟨ 高度$_1$⟩,···,⟨ 高度$_n$⟩
set cntrparam levels incremental ⟨ 開始 ⟩, ⟨ 増分 ⟩,⟨ 終点 ⟩
set cntrparam levels incremental ⟨ 開始 ⟩, ⟨ 増分 ⟩

等高線の補間式設定: 'linear', 'cubicspline', 'bspline' で等高線の補間式を指定します. 既定値は線形補間の 'linear' です. ここで, 'cubicspline' や 'bspline' を利用する場合, points に指定する数値が補間の点数, order で指定する整数値が 'bspline' で用いる補間多項式の次数を定めます.

levels: 等高線の総数, 指定した高さでの等高線の描画, 等高線の間隔の指定が行えます. 等間隔の等高線を描く場合には 'levels auto' で等高線の総数を指定します. このときに等高線は最低点と最高点の 2 点を含めて指定した数だけ高度に対して等間隔に描かれます.
ここで, 指定した高度の等高線だけを描かせる場合には 'levels discrete' で等高線を描く高度を指定します.
このことを次の例で確認してみましょう:

level discrete の例

入力	概要
set contour	等高線の表示を指定
set isosamples 50	解像度を 50x50 に指定
set cntrparam levels discrete -20,0,20	等高線の位置を指定 l
splot x*y*sin(sqrt(x**2+y**2))	函数の描画を実行

ここでの入力は等高線を高さが-20, 0 と 20 の三種類に設定しているので, 図 11.28 の凡例と曲面に 3 個の等高線が出ています.
次に, 一定の間隔で決められた高度の範囲で等高線を描画する場合は incremental を用います. この incremental の引数は始点, 増分と終点の順で 3 個の引数を記述し, 増分は始点と終点が矛盾さえしなければ正でも負でも構いません.
先程の例に次の命令を実行した結果を図 11.29 に示しておきす:

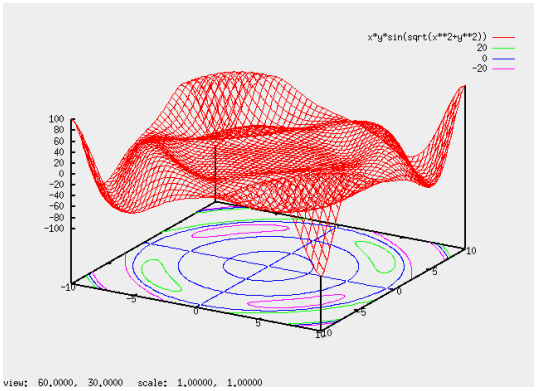


図 11.28: 指定した高さの等高線を描画

incremental の例

入力	概要
set cntrparam level incremental -20,2,20	incremental の指定
replot	再描画

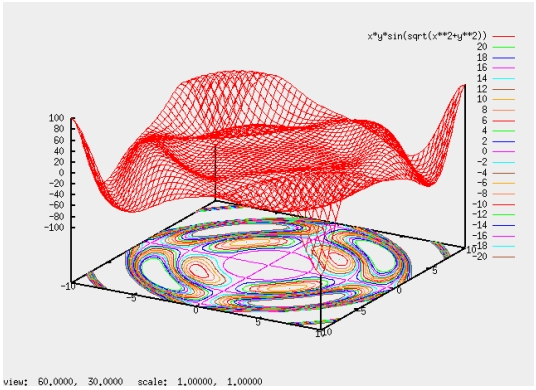


図 11.29: 指定した高さの等高線を描画

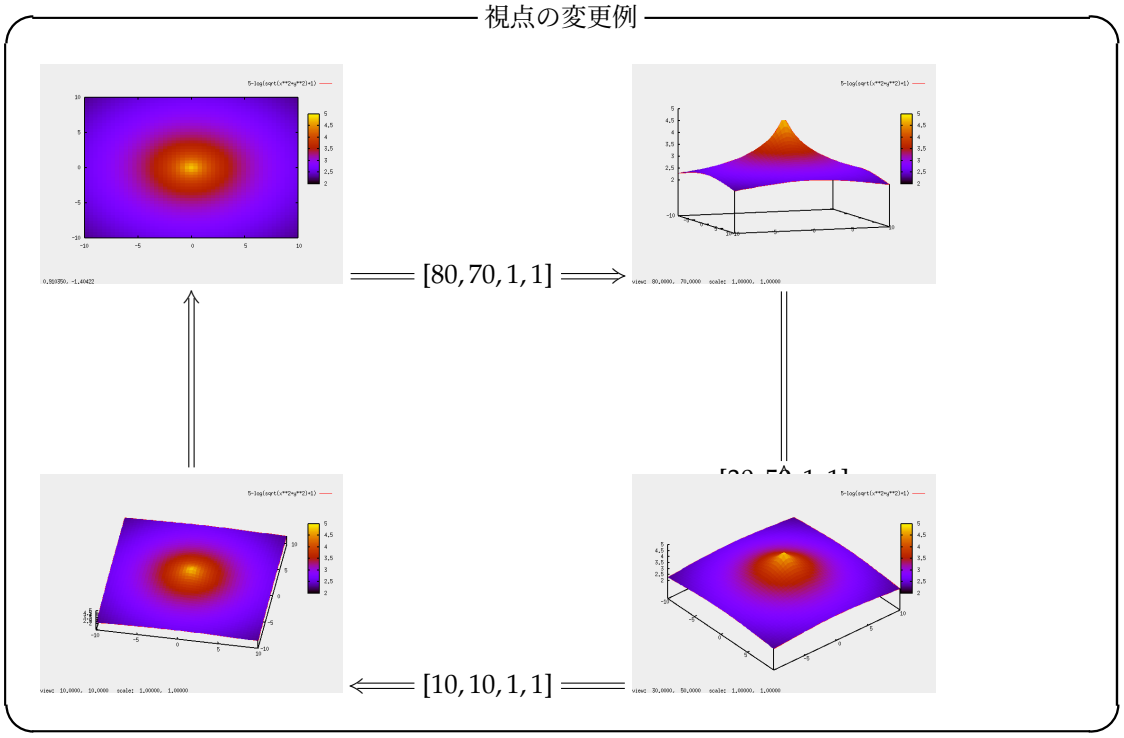
11.5.11　視点の変更

gnuplot で 3 次元グラフの視点は view で設定出来ます:

view に関連する文

set view ⟨X 軸の回転角 ⟩,⟨Z 軸の回転角 ⟩⟨ 拡大率 ⟩,⟨Z 軸方向の拡大率 ⟩
set view ⟨X 軸の回転角 ⟩,⟨Z 軸の回転角 ⟩
set view map
show view

ここで view の初期値は X 軸回りの回転角が 60 度, Z 軸回りが 30 度で各軸方向の拡大率は 1 です. なお, view のオプションに 'map' があります. この効果は 'set pm3d map' と同値で曲面を真上から眺める形になります. これは X 軸回りと Z 軸回りの回転角が 0 度の場合に一致します. 次に視点を変更した例を纏めておきます:



右上の図か 'set view 80,70,1,1' に対応し, 以降, 反時計回りに 'set view 50,70,1,1',
'set view 10,10,1,1' と 'set view map' に対応し, 括弧 "[ ]" の中の数値は view の引数に対応します.

11.5.12 cbrange と cblabel

cbrange: 階調の調整は cbrange で行いますが, その構文は xrange と同じで, 曲面の高さと階調の対応関係を指定します. なお, 曲面の最低値と最高値が cbrange の下限と上限であれば肌理細かな階調表示になりますが, この cbrange の幅が曲面の高低差と比較して大き過ぎる場合には曲面の階調が殆ど出ないのっぺら坊な曲面になることに注意しましょう.

cbrange に関連する文

set cbrange [⟨ 下限 ⟩ : ⟨ 上限 ⟩]
unset cbrange
show cbrange

階調の範囲を指定では MATLAB 風の表記 “[a:b]” を用い, 通常のリスト “[a, b]” とは異なるので注意が必要です.

cblabel:　階調ボックスのラベルを設定することに用います. ここでの設定は X 軸等の各軸にラベルを配置する xlabel 等と同様の構文になります:

cblabel に関連する文

set cblabel ⟨ 文字列 ⟩ ⟨x 座標 ⟩,⟨y 座標 ⟩
set cblabel ⟨ 文字列 ⟩
unset cblabel
show cblabel

ここでの ⟨x 座標 ⟩,⟨y 座標 ⟩ は無指定の場合の位置を基準とする文字列表示位置になります. 座標は縦上方向を Y 軸正方向, 横軸右向きを X 軸の正方向とし, 1 が一文字分になります. そのために画素単位での値は表示フォントや仮想端末を含めた環境で異なります.
さて, 動作確認のために図 11.23 のグラフで次の処理を実行します:

```
set cblabel 'Height' -13,6;
set cbrange [8:10];
replot
```

この結果は図 11.30 に示しておきますが, cbrange を狭くしたために高さが 8 以下は真っ黒になっています.

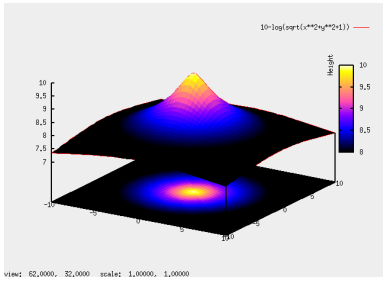


図 11.30: set cbrange [8:10];replot

それから次の処理を行いましょう.

```
set cblabel 'Height' -13,6;
set cbrange [5:10];
replot
```

結果は図 11.31 に示しておきます:

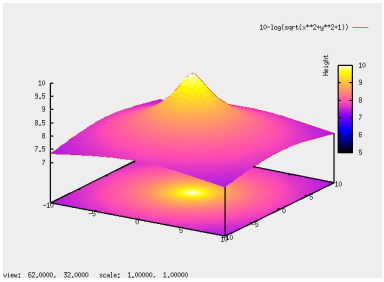


図 11.31: set cbrange [5:10];replot

今度は下に長く取ったので曲面全体が明るくなっていることが分ります.

11.5.13 陰線処理

gnuplot は特に指定をしなければ陰線処理を行いません. この陰線処理をさせるためには hidden3d を有効にします. なお, hidden3d は pm3d とは無関係に利用できます.
ここで pm3d が有効であれば, 'unset surface' を実行すると, pm3d の曲面上の網目を非表示にできますが, そうすると, pm3d の曲面の設定が無効になる副作用があります.
ここでは 'set hidden3d' と 'unset surface ' の違いを確認しましょう. 比較のグラフは次で描画したものとします:

```
set pm3d at bs
set cbrange [-0.05:1]
splot 1/(x**2+y**2+1)
```

図 11.32 にそのグラフを示します:

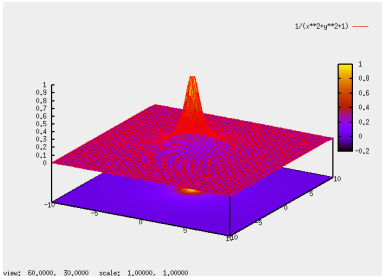


図 11.32: 初期状態

図 11.32 を大本として, 'set hidden3d' を実行した結果を図 11.33, 'unset surface' を実行した結果を図 11.34 に示しておきましょう.
'set hidden3d' と 'unset surface' の違いは 'set hidden3d' では稜線や境界に線分が残っているのに対し, 'unset surf' では一切の線分が消えていることです.

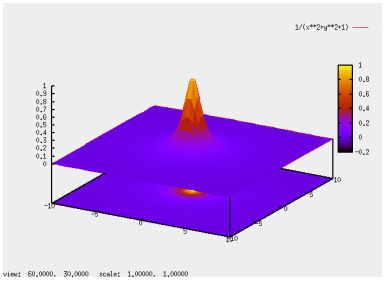


図 11.33: set hidden3d

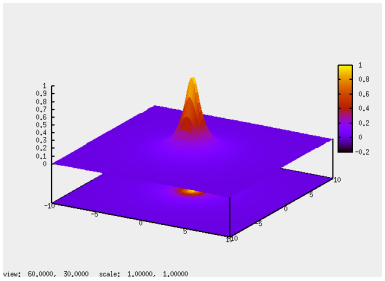


図 11.34: unset surface

11.5.14　Maxima のグラフと gnuplot のグラフの比較

gnuplot で極を持つ数式を描画したときに, この極が埋没する傾向があります. このことを Maxima と gnuplot で同じ式を表示させて確認してみましょう. そのために極のある函数として $\log\left(x^2y^2\right)$ のグラフを描くことにします.
最初に gnuplot で安易に 'splot log(x**2*y**2)' を実行した結果を図 11.35 に示します. こう見ると綺麗な曲面ですね:

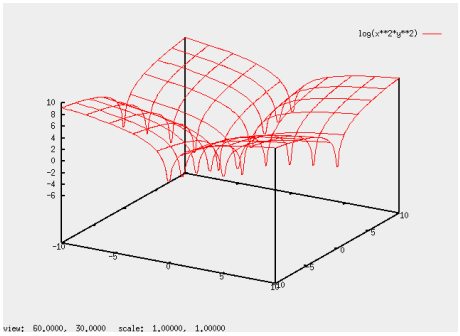


図 11.35: splot log(x**2*y**2)(その 1)

次に 'set pm3d at s' で 'splot log(x**2*y**2)' を実行した結果を図 11.36 に示します. この図では面を張ったために粗さが目立ちますね.

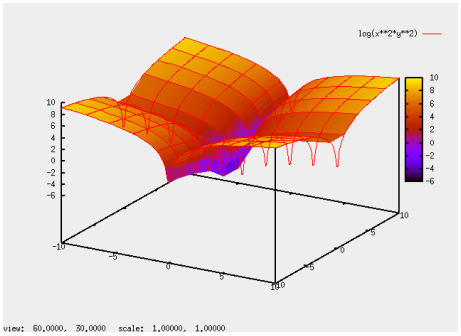


図 11.36: splot log(x**2*y**2)(その 2)

最後に 'set isosample 100' に変更して, より細かな分割で曲面を描画した結果を図 11.37 に示します:

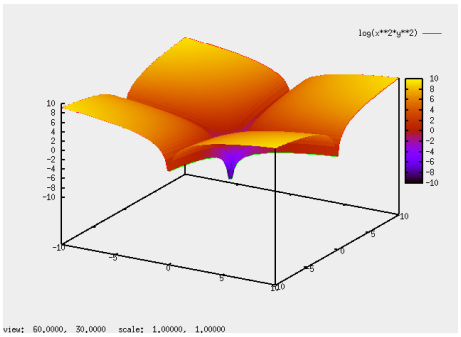


図 11.37: splot log(x**2*y**2)(その 3)

次に, この函数を Maxima から表示してみましょう. gnuplot と同様の細かさを得るために, ここでは次の処理を Maxima に実行させます:

```
plot3d(realpart(log(x^2*y^2)),[x,-10,10],[y,-10,10],
[grid,100,100],[gnuplot_preamble,"set pm3d at s;unset surf"]);
```

この処理結果を図 11.38 に示します:
この図は今迄の gnuplot のものと比べて谷と山が遥かに深くなっています. 何故, 同じ gnuplot でも Maxima から描くと違った形の絵になるのでしょうか？ 単純に拡大率の違いでしょうか？
実は式を良く見れば分る簡単なことですが, X 軸上の点と Y 軸上の点が函数の極になっています. すなわち, gnuplot の表示は大人し過ぎるのです. この意味では Maxima の処理の方がまだ雰囲気が良く出ていると言えます.
さて, この式を Maxima で描画する際に, Maxima からいろいろと質問が出ています:

```
(%i6) plot3d(realpart(log(abs(x))+log(abs(y))),[x,-10,10],[y,-10,10],
[grid,100,100],[gnuplot_preamble,"set pm3d at s;set hidden3d;"]);
Is x zero or nonzero?
```

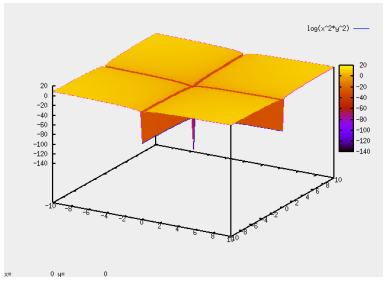


図 11.38: Maxima から $\log\left(x^2y^2\right)$ の描画

```
nonzero;
Is  y  zero or nonzero?

nonzero;
(%o6)
```

ここで Maxima が聞いていることは x と y が零か零でない数値であるかどうかということです. このようなことを Maxima が聞いて来る理由は, x や y の何れかが零になると非常に都合の事態に陥ると Maxima が判断したからです. だから, このような質問が出た場合には函数の定義域に関して吟味する必要があります. この点からも Maxima の plot.lisp で記述されている内容の方が gnuplot よりも遥かに安全であると言えます.

11.5.15　曲線と曲面の細かさの指定

gnuplot の式を表示して粗さが目立つ場合, 曲線の場合は sample , 曲面の場合は isosamples の設定を大きなものに変更します.

sample と isosamples に関連する文

set isosample ⟨ 数値$_1$⟩,⟨ 数値$_2$⟩
set sample ⟨ 数値$_1$⟩
set isosamples ⟨ 数値$_1$⟩,⟨ 数値$_2$⟩
set isosamples ⟨ 数値$_1$⟩
show isosamples

sample と isosamples には二つの数値か一つの数値を指定します. sample の既定値は 100, isosamples の既定値は 10 です.

sample:　X 軸上の総点数が設定されます. $\sin\dfrac{1}{x}$ のグラフを使って確認してみましょう:

```
set xrange [-1:1]
plot sin(1/x)
```

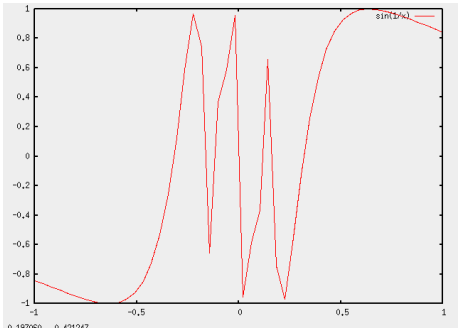


図 11.39: set sample 50 の場合

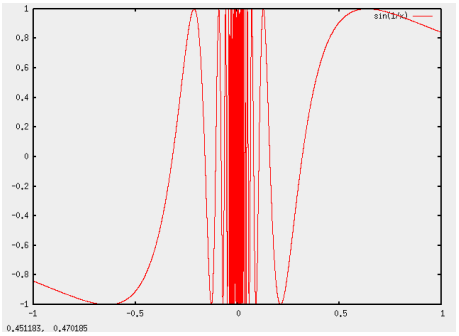


図 11.40: set sample 10000 の場合

isosamples: 〈数値$_1$〉,〈数値$_2$〉で X 軸と Y 軸の分割数を定め, 〈数値$_1$〉だけが指定された場合, X 軸と Y 軸も共に, この分割数になります. ただし, Maxima から曲線や曲面を描画する場合は plot2d 函数や plot3d 函数に与えた Maxima の式から gnuplot の描画に必要な数値与件を計算しますが, この計算では大域変数 plot_options の nticks や grid に設定した値が反映されます. そのために gnuplot_preamble で sample や isosample を指定しても, この設定は gnuplot の式の描画にだけに有効であって, 直接, 役には立ちません. とは言え, gnuplot_curves_titles を使って gnuplot の式を埋込む場合は有効です.

11.5.16 描画の領域設定

plot 命令や splot 命令で描画する gnuplot の式に対し, その描画する範囲は xrange, yrange や zrange で指定します.

描画する領域の指定に関連する文

set xrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション$_1$⟩ ⟨ オプション$_2$⟩
set xrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション$_1$⟩
set xrange [⟨ 下限 ⟩:⟨ 上限 ⟩]
set xrange restore
show xrange
set yrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション$_1$⟩ ⟨ オプション$_2$⟩
set yrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション$_1$⟩
set yrange [⟨ 下限 ⟩:⟨ 上限 ⟩]
set yrange restore
show yrange
set zrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション$_1$⟩ ⟨ オプション$_2$⟩
set zrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション$_1$⟩
set zrange [⟨ 下限 ⟩:⟨ 上限 ⟩]
set zrange restore
show zrange

cblabel の小節でも解説しましたが, ここで領域の設定は MATLAB のベクトルの定義に似た書式となります. すなわち, '[-1:1]' と記述し, '[-1,1]' ではないことに注意して下さい. ここで下限の既定値は '-10', 上限の既定値は '10' になっています. そのために何も指定せずに plot 命令や splot 命令で描画すると, この領域で描画が実行されます.
この領域の設定に, オプションとして軸の向きを逆にする 'reverse', 元に戻す 'noreverse' があります. 早速, 'reverse' を plot と splot で試してみましょう:

```
set xrange [0:10] reverse
plot cos(x)
```

図 11.41: plot cos(x)

```
set zrange [0:10] reverse
splot sqrt(x**2+y**2)
```

図 11.41 では X 軸の向きが逆になっていますね. なお, 図 11.42 の場合は, Z 軸が通常の逆になっています.

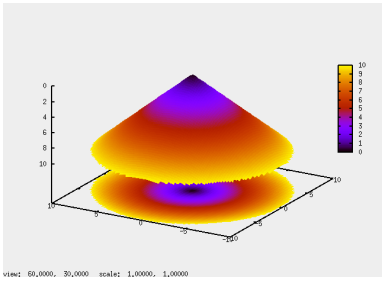


図 11.42: splot sqrt(x**2+y**2)

領域を初期値に戻したければ 'set xrange restore' の様に区間を記述せずに restore を指定すれば良いのです.

11.5.17 ラベル表示と注釈に関連する事項

gnuplot の便利の良さの一つに曲線や曲面にラベルや注釈を色々と入れ易いことが挙げられるでしょう. 実際, このような機能は非常に豊富で, この小節で把握し切れるものではありません. そのために, ここでは本当に一部の機能の解説に留めておきます. 詳細は文献 [61] を参照して下さい. では手始めに座標軸のラベルの指定について解説します.

軸のラベル指定

軸のラベルを指定する場合

set xlabel ⟨ 文字列 ⟩ ⟨X 座標 ⟩,⟨Y 座標 ⟩
set xlabel ⟨ 文字列 ⟩
show xlabel
set ylabel ⟨ 文字列 ⟩ ⟨X 座標 ⟩,⟨Y 座標 ⟩
set ylabel ⟨ 文字列 ⟩
show ylabel
set zlabel ⟨ 文字列 ⟩ ⟨X 座標 ⟩,⟨Y 座標 ⟩
set zlabel ⟨ 文字列 ⟩
show zlabel

xlabel, ylabel, zlabel は X 軸, Y 軸, そして Z 軸のラベルを指定します. 文字列の後に入れる X, Y 座標は通常の文字列が表示される位置を基準として, その位置から何文字分移動するかを指定します. ただし, ここでの値はフォントの指定等で異なる値になります.

曲線の表題表示

複数の曲線を表示した際に右上側に線分と曲線の表題が並んで表示されています. gnuplot ではこれを key と呼んでおり, この key の位置と key に表示する線分の長さ等の指定が行えます.
最初に key の表示位置を設定する構文を纏めておきます:

グラフの key の位置を設定

set key right
set key left
set key top
set key top left
set key top right
set key bottom
set key bottom left
set key bottom right
set key outside
set key below

この key の表示自体の調整も行えます:

グラフの key を設定

set key reverse
set key noreverse
set key samplen 〈 線分の長さ 〉
set key box linetype 〈 線分の様式 〉 linewidth 〈 線分の幅 〉
unset key box
show key

reverse と noreverse:　reverse で線分とその名称の羅列の順序を逆にし, noreverse で初期状態に戻します.

samplen:　key で表示されている線分の長さの調整に用います.

linetype:　set key box linetype で key の線分の様式や幅を変更します.

unset key box:　key を表示したくない場合に利用します.
次に各軸のラベルと key の設定例を次に示します:

key の設定例

命令文	概要
set xlabel 'Time -sec-'	X 軸のラベルを'Time -sec-' に設定
set ylabel 'Height -mm-' 0,40	Y 軸のラベルを'Height -mm-' とし Y 軸の 40 文字程上側に配置
set key outside	key をグラフの外側に配置
set key reverse	線分と曲線名の順序を通常の逆
set key samplen 10	key の線分の長さを 10
plot sin(x),cos(x),sin(x)/x	グラフの描画

この結果を図 11.43 に示しておきます:

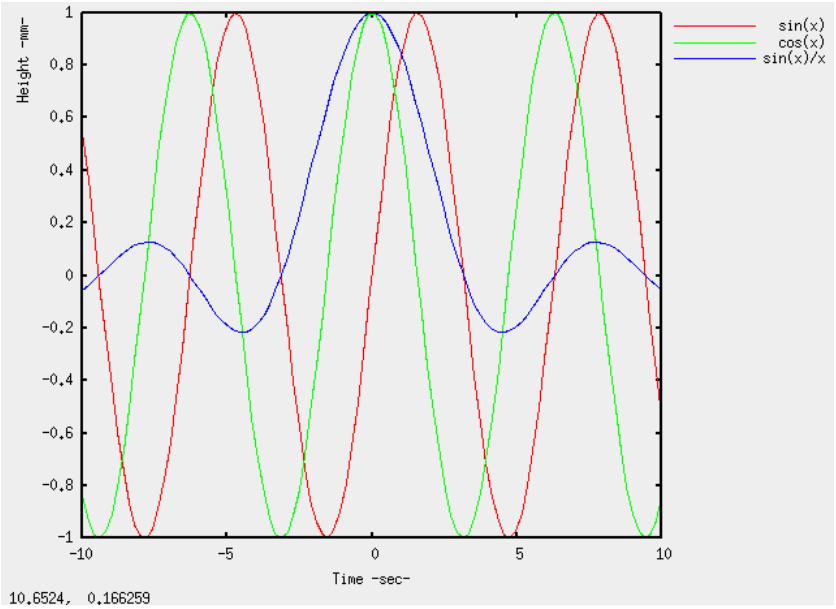


図 11.43: 各軸のラベルと key の設定例

注釈と矢印の追加

グラフの注釈は label で設定します:

註釈の設定

set label ⟨ 文字列 ⟩,⟨ 注釈番号 ⟩ at ⟨*X* 座標 ⟩,⟨*Y* 座標 ⟩
set label ⟨ 注釈番号 ⟩,⟨ 文字列 ⟩ at ⟨*X* 座標 ⟩,⟨*Y* 座標 ⟩
set label ⟨ 文字列 ⟩
set label ⟨ 注釈番号 ⟩ ⟨ 位置合せ ⟩
set label ⟨ 注釈番号 ⟩ font ⟨ フォント名 ⟩,⟨ 大きさ ⟩
set label ⟨ 注釈番号 ⟩ front
set label ⟨ 注釈番号 ⟩ back
set label ⟨ 注釈番号 ⟩ textcolor ⟨ 表示色 ⟩
set label ⟨ 注釈番号 ⟩ rotate by ⟨ 角度 (*deg*)⟩
set label ⟨ 注釈番号 ⟩ norotate
set label ⟨ 注釈番号 ⟩ point pointsize ⟨ 数値 ⟩
unset label ⟨ 注釈番号 ⟩
unset label
show label

set label 文で ⟨ 注釈番号 ⟩ が無指定の場合, 自律的に注釈に適切な番号を割当てます. そのあとのの註釈の回転, 内容やフォントの変更等の注釈の処理はこの番号を指定して行わなければなりません. 注釈の座標はグラフ上の点の位置に対応し, 文字列の左寄, 右寄や中寄といった位置決めは left, right と center で行います.
注釈は rotate by で XY 面内の回転が出来ます. by のうしろには角度を指定します.
では実例として次の処理を gnuplot で実行させてみましょう:

label の例

命令文	概要
set label 1 ’ Comment at Origin’ at 0,0	原点 (0,0) に注釈 1 を配置
set label 2 ’ Comment at (1,1)’ at 1,1	点 (1,1) に注釈 2 を配置
set label 1 point pointsize 6	注釈 1 の文字の大きさを 6
set label 2 point pointsize 2	注釈 2 の文字の大きさを 2
set label 2 rotate by 90	注釈 2 を 90 度回転
plot sin(x)*2	グラフの描画

この結果を図 11.44 に示しておきます:

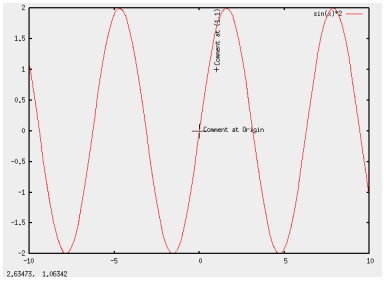


図 11.44: 注釈の設定例 (その 1)

特定の注釈を非表示にしたければ, unset label ⟨ 注釈番号 ⟩ で対処します. ここでは図 11.44 の例に対し, 番号 2 の註釈を unset label 2 を実行したあとに replot した結果を図 11.45 に示します. ここで番号 2 の註釈が実際に消えていることが分ります:

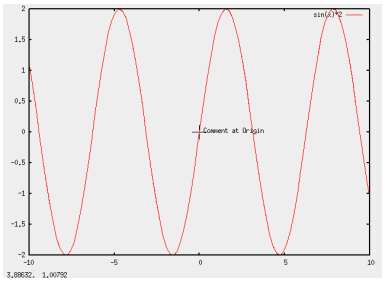


図 11.45: 注釈の設定例 (その 2)

矢印の追加

矢印をグラフ中に追加ができるので, 註釈と併用すれば効果が上ります:

arrow による矢印の設定

set arrow ⟨ 番号 ⟩ from ⟨*X* 座標 ⟩,⟨*Y* 座標 ⟩ to ⟨*X* 座標 ⟩,⟨*Y* 座標 ⟩
set arrow ⟨ 番号 ⟩ from ⟨*X* 座標 ⟩,⟨*Y* 座標 ⟩
set arrow ⟨ 番号 ⟩ to ⟨*X* 座標 ⟩,⟨*Y* 座標 ⟩
set arrow ⟨ 番号 ⟩ size ⟨ 矢の長さ ⟩,⟨ 矢の角度 ⟩
set arrow ⟨ 番号 ⟩ head
set arrow ⟨ 番号 ⟩ heads
set arrow ⟨ 番号 ⟩ nohead
set arrow ⟨ 番号 ⟩ filled
set arrow ⟨ 番号 ⟩ nofilled
unset arrow
show arrow

構文は label を用いた註釈の設定に似ています. まず, 図形としての矢印の設定は set arrow 1 from 0,0 to 1,1 の様に始点と終点を決めるだけで生成されます. このときに矢印の向きは from から to に向きます. なお, from の点や to の点が原点の場合は省略しても構いません.
次に arrowsize で矢印の頭の部分を調整できます. ここでの ⟨ 矢の長さ ⟩ は矢印の頭の矢の長さで, ⟨ 矢の角度 ⟩ は頭の矢の角度になります. なお, この角度は度数です. そして, filled を指定すると矢印の矢の部分が塗り潰されます.
では実際に次の設定により, どのようなグラフが得られるか確認してみましょう:

矢印の例

命令文	概要
set arrow 1 from 0.4,-1 to 1.1,-1	矢印 1 の生成
set arrow 2 from 0.9,1 to 1.1,1	矢印 2 の生成
set arrow 3 from 1,-1 to 1,1	矢印 3 の生成
set arrow 1 nohead	矢印 1 には頭を付けない
set arrow 2 nohead	矢印 2 には頭を付けない
set arrow 3 size 0.1,30 filled heads	矢印 3 の頭の設定
set label 1 ' Width' at 1.05,0	注釈 1 の設定
plot cos(6.283*x);	数式の描画. その 1
set xrange [0:2]	数式の定義域を設定
set yrange [-2:2]	数式の値域を設定
plot cos(6.283*x);	数式の描画. その 2
set arrow 3 filled	矢印 3 を塗り潰した通常の矢印に設定
plot cos(6.283*x);	数式の描画. その 3

ここで矢印の設定は最後に入力した設定だけが有効になり, 他は初期化されてしまうので注意が必要です.
この結果を図 11.46 に示しておきます:

11.5.18　gnuplot の式

gnuplot では FORTRAN で用いられる数式が利用可能です. その他の函数も充実していますが, ここでは簡単に gnuplot の式について述べます.
gnuplot の式の演算子は FORTRAN の書式に準じます. ここで gnuplot の算術演算子を次に示しておきます:

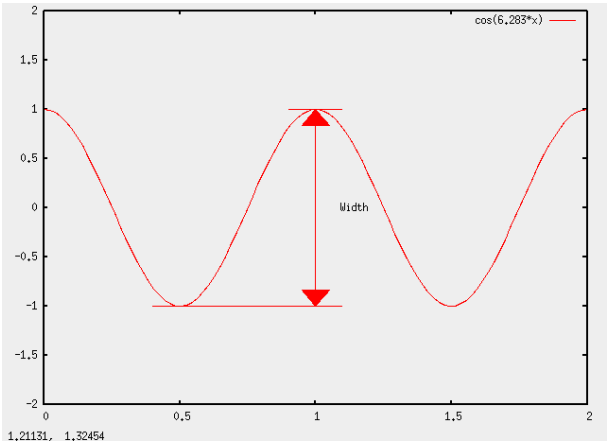


図 11.46: 矢印と註釈の設定例

gnuplot の算術項演算子

演算子	例	概要
+	a+b	和
-	a-b	差
*	a*b	積
**	a**b	冪
/	a/b	商
%	a%b	剰余
-	-a	負の演算子
+	+a	正の演算子
!	a!	階乗

このように gnuplot では一般的な算術演算子の殆どが使えます. なお, 冪は Maxima 等の数式処理でよく用いられる演算子 “^” ではなく, FORTRAN で用いられる演算子 “**” を用います. この点は Maxima と併用する場合には特に間違え易いので注意しましょう.
そして, gnuplot では色々な数学函数が使えます. 次に gnuplot で利用できる函数一覧を示しておきます:

GNUPLOT の数学函数

abs	絶対値
sqrt	平方根を計算
imag	複素数値の虚部を返す函数
real	複素数値の実部を返す函数
rand	乱数函数

三角函数

cos	余弦函数	acos	逆余弦函数
sin	正弦函数	asin	逆正弦函数
tan	正接函数	atan	逆正接函数
atan2	逆正接函数		
cosh	双曲余弦函数	acosh	逆双曲余接函数
sinh	双曲正弦函数	asinh	逆双曲正弦函数
tanh	双曲正接函数	atanh	逆双曲正接函数
exp	指数函数	log	自然底の対数函数
log10	底が 10 の対数函数	arg	偏角を返す函数

特殊函数

ibeta	不完全 B 函数
gamma	Γ 函数
igamma	不完全 Γ 函数
besj0	第一種のベッセル函数 $J_0(x)$
besj1	第一種のベッセル函数 $J_1(x)$
besy0	第二種のベッセル函数 $Y_0(x)$
besy1	第二種のベッセル函数 $Y_1(x)$
erf	誤差函数
erfc	相補誤差函数 ($erfc(x) = 1 - erf(x)$)
inverf	逆誤差函数
norm	正規分布
invnorm	逆正規分布函数
lambertw	ランベルトの W 函数

その他の函数

ceil	与えられた値を越えない整数に 1 を加えた値を返す.　ceil(x) は `floor(x)` + 1 と同じ
floor	与えられた値を越えない整数を返す
int	小数点の切り捨てを行う函数
sgn	符号函数. 引数が負の場合は-1, 正の場合は 1 を返す

これらの算術演算子と函数の他に論理演算もできます:

gnuplot の論理演算子

演算子	例	概要
==	a==b	等号
!=	a != b	等しくない
>=	a¿=b	以上
>	a¿b	大なり
<=	a¡=b	以下
<	a¡b	小なり
&&	a&&b	論理積
||	||	論理和
&	a&b	ビット単位の論理積
|	|	ビット単位の論理和
~	~a	補間
!	!a	否定

ここで独特なのがビット単位の論理積 “&” と論理和 “—” です. これらは与えられた整数に対し, それらを二進数で置換えて各桁で論理積や論理和を各々実行した結果を返す演算子です.
これらの演算子を活用することで複雑な函数を gnuplot 内部で定義し, さらに, その函数を使って描画することもできます. この場合, 函数に与えられる式は ‘sin(x)**+1’ のような gnuplot の函数と算術演算子を用いた式, そして, 変数の範囲で函数を切り替える式の二種類が使えます.

函数の定義

⟨ 函数名 ⟩ (x)= ⟨ *gnuplot* の算術演算子と函数の式 ⟩
⟨ 函数名 ⟩ (x)= ⟨ 条件$_1$⟩ ? ⟨ 式$_1$⟩ : ··· ⟨ 条件$_n$⟩ ? ⟨ 式$_n$⟩
⟨ 函数名 ⟩ (x)= ⟨ 条件$_1$⟩ ? ⟨ 式$_1$⟩ : ··· ⟨ 条件$_n$⟩ ? ⟨ 式$_n$⟩ : ⟨ 各条件を満さない場合の値 ⟩

条件は gnuplot の論理式で構成され, 函数の定義域を定めます. たとえば, -1 から 1 までが x^2 で, それ以外を 1 とする函数を定義したければ次の書式で定義します:

```
f(x)= x>=-1 && x<=1 ? x**2: 1
```

このように指定した区間以外の値は函数の定義の最後に値を設定すれば良いのです. さらに gnuplot 上で定義する函数の変数は x, y, z に限定されません.
なお, 特定の区間のみの値のグラフ表示をしたければ, 特定の区間だけで式を定義しておいて, 最後に 1/0 や-1/0 のよう極を設定すると, グラフ表示で指定した区間だけが表示されます:

```
f(x)= x>=-5 && x<=1 ? cos(x): x>=1 && x<=5 ? sin(x):\
> x>=7 ?sqrt(x) : x<=-7?(x-x**2)/100: 1/0
plot f(x)
```

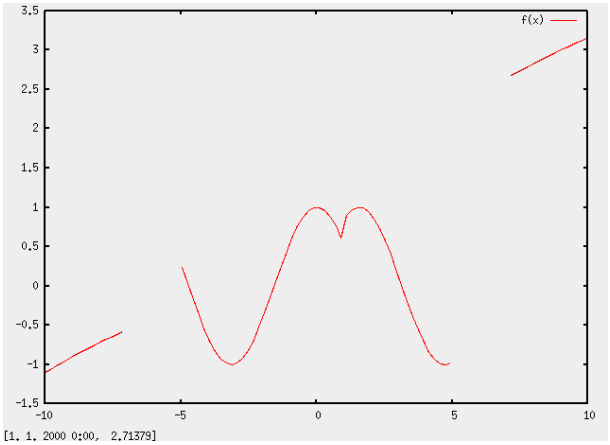


図 11.47: 不連続なグラフの描画

11.5.19　電卓としての gnuplot

このように gnuplot には豊富な函数もあり, 自分で函数も定義できるので, Octave までとははいかなくても簡単な卓上計算機として使いたいものですね. ところが, Maxima と Octave と大きく異なる点は, gnuplot に式を入力してもエラーが返されることです:

```
gnuplot> 1+1
         ^
         invalid command

gnuplot> sin(0)
         ^
         invalid command
```

このことは gnuplot の式はグラフ描画が専門であって, 電卓代りにすることは, あまり考えていなかったからでしょう.
とは言え, 前述のように C や FORTRAN の式が入力可能な程の能力を持っているので, このままにしておくのは非常に勿体ないのでしょう. そこで, 電卓として使う方法について簡単に述べておきます.
とは言っても話は非常に簡単なことで, 式の結果を何等かの変数に代入し, その変数を print 命令に引渡すか, その式を直接 print 命令に引渡せば良いのです:

```
gnuplot> print 1+1
2
gnuplot> print cos(3.14)
-0.99999873172754
```

で, 電卓として使う場合に, この電卓で扱える与件には何があるでしょうか？先程の式の解説では gnuplot で扱える与件に関して頬被りしていましたが, gnuplot には論理演算を行うための真偽値, 整数値, 浮動小数点と複素数値があります.

真偽値: gnuplot では真を 1, 偽を 0 とします. この値は他の数値と混在しても構いません. このことを利用すると, if 文を用いずに式を容易に定義できるのです. たとえば, x が 2 なら 10 を返し, それ以外は-9 を返す函数も次で定義できます:

```
gnuplot> g(x)= (x==2)*10 + (x!=2)*(-9)
gnuplot> print g(2)
10
gnuplot> print g(4)
-9
```

この方法は Octave や Yorick でお馴染の手応です. この機能を使えば gnuplot でも長々と条件分岐を書かなくても済む訳です.

整数値と浮動小数点: 整数は小数点を持たない実数で, 浮動小数点数は小数点を持つ実数です. ここで整数同士の演算結果は整数になるので, 整数同士の割算を行う場合は注意が必要です.

```
gnuplot> print 4/3, 4.0/3
1 1.33333333333333
```

この例では最初の 4/3 が分子, 分母共に整数のために整数剰余が適用されています. ところが, 最後の 4.0/3 では分子が浮動小数点数なので割算の結果は浮動小数点数になります. このように実数式で浮動小数点数を含む式の場合の結果は浮動小数点数に, 後述の複素数を含む式の結果は複素数になります.

複素数: 複素数は大括弧 {} を用います. たとえば, $1 + 2i$ は {1,2} のように数値の対で表現します. 複素数の場合, 実部と虚部は浮動小数点数になります:

```
gnuplot> print 1+{1,3}+{0,3}
{2.0, 6.0}
gnuplot> print 3*{1,4}/9
{0.333333333333333, 1.33333333333333}
```

最初の例で示すように入力側の数値は整数でしたが, 結果は実部と虚部が浮動小数点数になっていることに注意して下さい. この複素数に対しても通常の算術演算子が使えます. そして, 複素数の実部を返す命令が real, 虚部を返す命令が imag になります.
このように gnuplot は電卓としても十分な機能を持っています. この電卓には履歴機能に加えて, グラフ描画機能さえも付いて来るのです. 素晴しいことですね.
なお, 函数の定義で特定区間の描画を行うために函数の値に 1/0 を設定すると申しましたが, print 文でこの式を表示させた場合, undefined value になります.

11.5.20 プログラム言語としての gnuplot

gnuplot では変数が扱えて函数も定義できます. このように Octave のような処理もある程度は行える訳ですが, では, プログラム言語としてはどうでしょうか？

gnuplot では一応, if 文による条件分岐を持っていますが, surf で見られるような if-goto による原始的な loop 文さえありません. この反復処理を行うには reread 命令を用いてファイルを介して行う必要があります.
まず, if 文の構文を以下に示しておきますが, 基本的に C に似た構文で, 文の区切はセミコロンで行います:

if 文の構文

if (⟨ 論理式 ⟩) ⟨ 文$_1$⟩ ; ··· ⟨ 文$_n$⟩
if (⟨ 論理式 ⟩) ⟨ 文$_1$⟩ ; ··· ⟨ 文$_n$⟩; else ⟨ 文$_{n+1}$⟩ ; ··· ; ⟨ 文$_{n+m}$⟩;
if (⟨ 論理式$_1$⟩) ⟨ 文$_1$⟩ ; ··· ; ⟨ 文$_n$⟩ ;else if (⟨ 論理式$_2$⟩) ⟨ 文$_{n+1}$⟩ ; ···

gnuplot の if 文は論理式や else の後に文をセミコロンで区切って並べます. ただし, 複雑な処理は行えません. if 文では条件式に合致する場合に, 条件式の直後から else が出現するまでの文全てを処理し, そうでない場合には else の直後の文から行末の文迄の全ての文を実行する仕様のためです. そのために gnuplot の if 文は文中に if 文を入れて階層的にせずに平面的に処理を行うのが妥当です.
処理言語の多くが条件分岐に加えて反復処理を行う制御文を持っていますが, gnuplot には明示的な反復処理を行う制御文を持っていません. gnuplot では再帰的に reread 命令を用いて gnuplot の文を記述したファイルの再実行による反復処理だけしか行えません.
たとえば, 次のように用います:

ファイル tama の内容

```
1 set isosamples 50;
2 if(amp<5) splot [-3:3] [-3:3] [-1:1] \
3 amp*sin(x*y)*exp(-sqrt(x**2+y**2));\
4 print "Hit any key!";pause -1;\
5 amp=amp+1;reread;else print "The End";
```

次に実行の様子を示しておきましょう:

```
gnuplot> set pm3d at s; set hidden3d;\
> amp=1;load "tama"
Hit any key!

Hit any key!

Hit any key!

Hit any key!

The End
gnuplot>
```

このようにファイルを介した再帰によって反復処理が実現出来ます. なお, ファイルが介するために反復処理の制御を行う変数の初期化は gnuplot 側で実行しておく必要があります. そのため, 現時点の Maxima でアニメーションを実行させる方法は, 結局, ファイル maxout.gnuplot に書き込んで, それを gnuplot に引渡す方式となるので制御変数の初期化もファイル maxout.gnuplot に書

込む方法しかありません. そのために plot2d 函数や plot3d 函数を用いてスマートに処理させることは難しくなります.
強引に実行させるのであれば gnuplot の load 命令と save 命令を用いて, 制御変数を別途臨時ファイルに保存と再読込を行えば実現させられなくもありません:

ファイル tama

```
load "nekonekoxx";
set isosamples 50;
if(amp<5) splot [-3:3] [-3:3] [-1:1] \
amp*sin(x*y)*exp(-sqrt(x**2+y**2));\
print "Hit any key!";pause -1;\
amp=amp+1;save var "nekonekoxx";reread;else print "The End";
```

ここでファイル nekonekoxxx の内容を次に示しておきます:

ファイル nekonekoxx の内容

```
amp = 0
```

この例では予めファイル nekonekoxx に変数 amp の値を設定しておきます. それから, gnuplot でファイル tama を load 命令を使って読込みます. ファイル tama では変数 amp の値をファイル nekonekoxx から読込むことで gnuplot 内部に取り込み, それから amp の値が 5 より小であればグラフの表示と amp に 1 を加えてファイル nekonekoxx に gnuplot 内部の変数を保存し, それから reread 命令を用いて自分自身を呼出します. amp が 5 以上であれば, “The End” と表示します. この手法を Maxima で使えなくもありませんが, plot2d 函数や plot3d 函数で Maxima 側で数値与件として計算した結果を gnuplot に引渡すので, アニメーションにしたい Maxima の式を gnuplot_curve_titles 側に引渡すといった随分と込み入った方法になります. そのために割り切って処理を行う必要があります.
MS-Windows 環境とそれ以外の環境では,Maxima は gnuplot の立ち上げのオプションが異なります. この立ち上げのオプションは大域変数 gnuplot_view_args で指定されています. MS-Windows 環境ではオプションが付きませんが, その他の環境では ‘-persist’ が付きます. このオプションは gnuplot を閉じてもグラフウィンドウを残すという gnuplot のオプションです.
以上で gnuplot の簡単な使い方の解説を終えます. 以降は Maxima での使い方について幾つかの実例の解説をしましょう.

11.6　plot2d 函数と plot3d 函数の活用事例

11.6.1　gnuplot_preamble の使い方

この節では plot2d 函数と plot3d 数で gnuplot を用いた描画例を幾つか示しておきます.
なお, gnuplot は非常に機能が高いアプリケーションで色々なことができますが, 本質的には plot 命令と splot 命令で描画, set 命令で諸設定, そして replot で再描画を行う仕組みになっています.
その一方で, Maxima は plot2d 函数や plot3d 函数に与えられた式から数値与件を生成し, 諸設定は gnuplot の命令を単純にファイルやストリームに出力する方式を採用しています.
そのために gnuplot を Maxima 側から操作することは gnuplot にストリームとして送り込んだり, maxout.gnuplot に書き込んだりする gnuplot の命令文を如何に記述するかという問題に帰着されます.
ここで, gnuplot は非常に高機能のツールであるために Maxima に準備された plot_options の項目だけで全てを網羅することは事実上不可能です. しかし, 大域変数 plot_options の項目の gnuplot_preamble にさまざまな gnuplot の制御文が設定可能となっているので, この特性を利用しないのは幾ら何でも勿体ないことです. ただし, plot_options で設定可能な項目はいろいとと位置が決っていおり, 利用者が思い通りに利用するためにはそれなりの工夫が必要になります. そのために plot_format を gnuplot にして, 描画の際に出力されるファイル maxout.gnuplot の内容を確認しながら進めて行くことは非常に有効な手段です.
さて, 今迄の復習になりますが, グラフの表題を mike, X 軸と Y 軸のラベルをそれぞれ X, Y としたい場合, Maxima でどのように入力すれば良いかを考えましょう.
ここで, gnuplot ではグラフの表題は set title で指定し, X 軸と Y 軸のラベルの設定はそれぞれ set xlabel と set ylabel で行います. 最終的には, これらの gnuplot の命令文をセミコロンで区切り, 全体を二重引用符で括ったものを gnuplot_preamble に設定します.
この処理と描画の実行を以下に示しておきます:

```
(%i5) nekoneko:"set title 'mike'; set xlabel 'X'; set ylabel 'Y';";
(%o5)                set title 'mike'; set xlabel 'X'; set ylabel 'Y';
(%i6) plot2d(sin(x)/x,[x,0,10],[gnuplot_preamble,nekoneko]);
```

この処理の結果得られるグラフを図 11.48 に示しておきます:

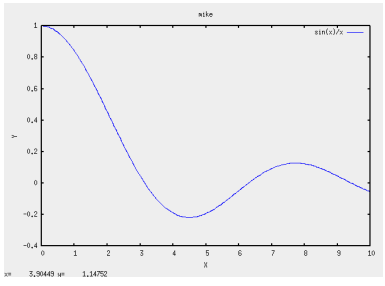


図 11.48: gnuplot_preamble で設定した結果

念のために, この例題で出力されるファイル maxout.gnuplot の先頭部分を確認しておきましょう:

ファイル maxout.gnuplot のヘッダ部分

```
set title 'mike';set xlabel 'X';set ylabel 'Y';
plot '-' title 'sin(x)/x' with lines 3

2.441406250000E-4 0.9999999900658926
4.88281250000E-4 0.9999999602635706
---省略---
```

gnuplot_preamble の内容は gnuplot の描画命令である plot 命令や splot 命令の直前にそのまま埋込まれます. ここで Maxima の式は解釈され, 数値データとして gnuplot の描画命令の直下に書込まれます. したがって, plot 命令や splot 命令のオプションとして設定される曲線の表題や曲線の様式を gnuplot_preamble で設定したとしても, 描画の際には, gnuplot の plot 命令や splot 命令のオプションが優先され, その上, Maxima での入力式で gnuplot は描画を行わないために, sample や isosamples の設定は意味はありません. 曲線の表題や曲線の様式は gnuplot_curve_titles と gnuplot_curve_style に設定すべきであり, 描画する曲線や曲面の細かさは予め nticks や grid で設定すべきであることが理解されるでしょう.
では, 次のように gnuplot_curve_style への設定を行って描画させてみましょう:

```
plot2d(sin(x),[x,-1,1],[gnuplot_curve_styles,"with impulse"])
```

このときのファイル maxout.gnuplot の plot 命令付近を次に示しておきます:

gnuplot_curve_styles の設定

```
plot '-' title 'sin(x)' with impulse
-1.        -0.8414709848078965
-0.975     -0.8277018881672576
--- 略 ---
```

この結果は図 11.49 のようになグラフになります:

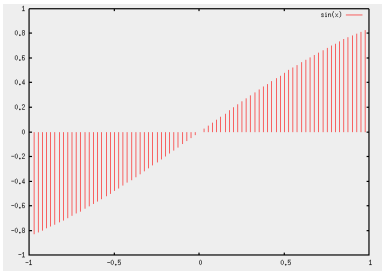


図 11.49: gnuplot_preamble で設定した結果

11.6.2　Maxima のバッチ処理

今度は Maxima で複数のグラフを描く処理を行うために, Maxima にバッチ処理を行わせる方法について簡単に解説しましょう.
まず, gnuplot_preamble に記述する設定の内容は高度なグラフ表示を行わなければならなくなれば, 益々, 膨大なものとなるでしょう. これを一々手入力することは最初の一枚のグラフを得るために試行錯誤することは仕方がないものとしても, 書式が全て決っており, その決められた書式で出力すれば済む話であるのならば自律的な処理を図りたいところです.
ここで Maxima にはバッチ処理機能があります. この場合, Maxima に実行させるファイルを予め準備しておきますが, このファイルの中身は通常の Maxima の式をそのまま記述したものです. このファイルを Maxima に引渡せば, Maxima がファイルの内容に従って処理を行うもので, この処理は Maxima の通常の数式や数値の処理だけではなく, 画像の生成にも使えます.
手始めに簡単な入力ファイル A を記述しておきましょう:

ファイル A の内容

```
1 a:(x+1)^2;
2 b:expand(a);
3 plot2d(b,[x,-1,1]);
4 plot2d(b,[x,-1,1],
5 [gnuplot_preamble,"set term png;set output 'test.png'"]);
```

これは非常に簡単な処理です. 最初の二つの式は通常の Maxima の処理の式です. それから plot2d の処理を行なっていますが, この場合, gnuplot でグラフを生成し, グラフ表示ウィンドウにグラフを表示してものを出力します. そして, 最後の処理では,gnuplot_preamble を利用して描画を行うものです. ここで gnuplot_preamble には, set term png と set output 'test.png' といった二つの gnuplot の設定文を纏めたものを入れています. ここで, 最初の命令文は gnuplot の 'terminal' を PNG に変更し, 次の命令文で出力先をファイル'test.png' に切替えています. すなわち, 最後の二つの命令文によって gnuplot はグラフを PNG データとしてファイル test.png に書込むことになります.
ここで, gnuplot が出力可能な画像データ形式は MS-Windows か Linux 等の OS 環境によって異なります. 一応, PNG であれば問題はないかと思いますが, 確認したければ,gnuplot を立ち上げて, ? term と gnuplot に入力してみましょう. すると色々と説明が表示されますが, 末尾に一覧が表示される筈です. その中にある端末が利用可能なので, その一覧表にある画像データ形式を選択しましょう.
このファイルは Maxima の中から, batch(A); と入力することでも処理ができますが, Maxima を立ち上げなくても次の構文で Maxima にファイルを引渡すことができます:

バッチ処理の構文

maxima -b 〈 ファイル 〉
maxima –batch=〈 ファイル 〉

この処理は MS-Windows でもコマンドプロンプト[5]からの実行が可能です. ちなみに MS-Windows のデスクトップに現われている Maxima のアイコンは wxMaxima のアイコンです. この wxMaxima は Maxima に被せる GUI 環境に過ぎません. そのために上に示した方法は wxMaxima には使えません. この場合は batch 函数を使いますが, 画像の生成を行いたければ wxMaxima ではなく maxima を起動させた方が良いようです. なお, maxima をコマンドプロンプトから起動させるためには MS-Window の環境変数の PATH に maxima への経路を追加する必要があります. この設定はコントロールパネルのシステムから詳細の中にある環境変数を選び, システムの環境変数の Path に書込みを行います. また, バッチファイルで term の設定を行っている場合, MS-Windows 版の Maxima に附属の gnuplot では使えない term が存在するので注意が必要です. たとえば, 附属の gnuplot では jpeg は使えません. ただし, gif と png は利用可能なので, これらを選択すると良いでしょう.

Maxima によるバッチ処理の計算結果は, 出力先をバッチファイル内部で指定しない限り標準出力に出力されます. 重要な計算を行う場合は maxima -b test¿test.log のようにリダイレクトを用いると良いでしょう.

そして, コマンドラインに入力するのが面倒であれば, この内容を含むスクリプトを記述しておくのも良いでしょう.

参考までに maxima -b A を実行した様子を次に示しておきます:

```
yokota@Zuse:~/TEST> maxima -b A
Maxima 5.10.0  http ://  maxima.sourceforge.net
Using Lisp  CLISP 2.37  (2006-01-02)
 Distributed  under the  GNU Public License.  See the  file  COPYING.
Dedicated  to  the  memory of William  Schelter .
This  is  a  development version  of  Maxima. The function  bug_report ()
provides  bug  reporting  information .
(%i1)                                  batch(A)

batching  /home/yokota/TEST/A
                                            2
(%i2)                                 a : (1 + x)
                                              2
(%o2)                                    (x + 1)
(%i3)                                 b : expand(a)
                                       2
(%o3)                                  x  + 2 x + 1
(%i4)                              plot2d(b, [x, - 1, 1])
(%o4)
(%i5)  plot2d(b, [x, - 1, 1], [gnuplot_preamble,
                                                 set  term  png;set  output  ' test .png ' ])
(%o5)
```

この処理では gnuplot のグラフも表示していますが, このグラフは面白味がないので, ここでは示しません. あしからず.

今度はもう少し複雑なグラフを描いてみましょう. そのために次に示す内容のファイル A2 を記述しましょう:

[5]所謂 DOS 窓

ファイル A2 の内容

```
1 nekoneko:"set pm3d at bs;set xrange [-2:2];\
2 set yrange [-2:2];set zrange [-10:20];\
3 set label 1 'top' at 0,0,20;\
4 set arrow 1 from 0,0,20 to 0,0,10;\
5 set arrow 1 size 5,30 filled head;\
6 set hidden3d;\
7 set output 'test1.png';set term png;";
8 plot3d(10*cos(x*y),[x,-2,2],[y,-2,2],[grid,50,50],
9 [gnuplot_preamble,nekoneko]);
```

このファイル A2 では gnuplot_preamble に与える文字列が長くなるので, 適宜, 行が継続していることを示す\を入れて改行しています.
では, maxima -b A2 の結果の text1.png の絵を図 11.50 に示しておきます:

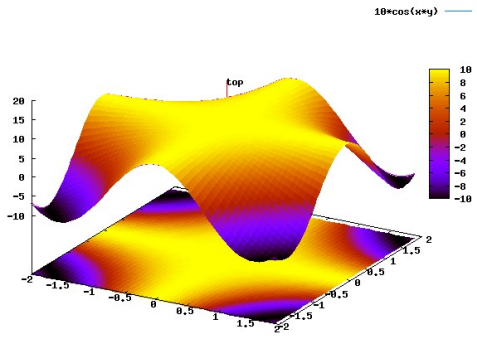


図 11.50: ファイル A2 の実行結果

ここで gnuplot で行う処理が複雑になればなる程, gnuplot_preamble の内容は必然的に長くなります. 画像 1 では設定 1, 画像 2 の場合に設定 2 を使う…等々とする場合, その設定を全て gnuplot_preamble に与える文字列として持たせるという方法は安易な方法ではありますが, 自動処理のことを考えると, そんなに効率的ではありません. むしろ, 共通の設定 1 があって, あとはケース毎に設定が追加される方が汎用性が高いでしょう. ケース毎に画像ファイル等の保存先のファイルを切替えることは普通に行われる処理です.
ところで, gnuplot_preamble に与えられるのは一つの文字列です. そこで Maxima で複数の文字列を纏めて一つの文字列にする操作を行えば良いことになります. Maxima には文字列の結合を行う処理を行う函数として concat 函数があります. この函数は与えられた複数の文字列を一つの文字列に変換する函数です.

そこで今度は concat 函数を使ってみた例を示しましょう:

ファイル B1 の内容

```
A1:"set pm3d at bs;";
R1:"set xrange [-10:10];\
set yrange [-10:10];set zrange [7:12];";
H1:"set hidden3d;set isosamples 50;";
O1:"set output ";
T1:"set term png;";
Ox:concat(O1,"'Fig1.png'",";");
nekoneko:concat(A1,R1,H1,Ox,T1);
assume(x^2+y^2>0);
fxy:10-realpart(log(sqrt(x^2+y^2+1)));
plot3d(fxy,[x,-10,10],[y,-10,10],[grid,20,20],
[gnuplot_curve_titles,"title 'fxy',5+x*y/10 title 'test'"],
[gnuplot_preamble,nekoneko]);
```

この例では,O1 に set output を割当てており, これを雛形として出力先のファイルを O1 に concat 函数で追加した文字列 Ox を利用しています. この方法を使えば条件文やバッチファイルを生成するスクリプトでファイル名の切替えが容易に行えます. さらに, このファイルでは splot に gnuplot の式を gnuplot_curve_titles を使って組込ませるようにしています.
では Maxima に B1 を処理させるシェルスクリプトを次に示しておきましょう:

B1 を処理させるシェルスクリプト

```
#!/usr/local/bin/maxima

maxima -b B1>B1.log
convert Fig1.png Batch2.eps
display Fig1.png
less B1.log
```

このシェルスクリプトでは最初に Maxima でファイル B1 の処理を実行します. その結果, 画像ファイル 'Fig1.png' が生成されますが, 次に convert 命令を使って, この Fig1.png を Batch2.eps に変換し, Fig1.png を display 命令を使って閲覧します. それから, 最後にバッチ処理の記録ファイルの B1.log を見るというものです.
このバッチ処理で生成された画像ファイルを図 11.51 に示しておきます.

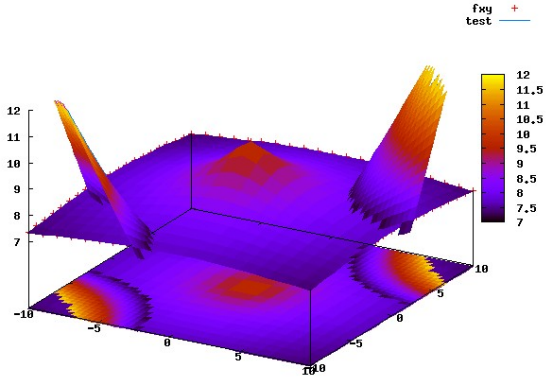


図 11.51: ファイル B1 の実行結果

ここで示したように, Maxima のバッチ処理と gnuplot_preamble が揃えば, あとは awk 等の適切な言語があれば十分複雑な処理が行えることが理解されるでしょう.

11.7 draw パッケージ

11.7.1 draw パッケージの概要

draw パッケージに含まれる描画函数には draw 函数, draw2d 函数と draw3d 函数の三つの函数があります. これらの函数では表示すべき対象を文を用いて構築し, 対象の線分を実線や点線で表示するといった性質を対象の属性として指定する方式となっています. また, 描画すべき曲線や曲面といった描画対象に加え, 座標軸や表題といったグラフも一つの対象として扱うこともできます. そのために複数の対象を様々な組合せで描画することも可能となっています.
さらに draw パッケージの描画函数から描画内容を直接ファイルに出力することも可能で, 従来の plot2d 函数や plot3d 函数よりも柔軟な運用が可能となっています.

11.7.2 外部アプリケーションの切替について

draw パッケージの函数は外部アプリケーションとして GNUPLOT を主に用います. 3 次元の対象のみ draw3d 函数で VTK[6]が利用できます. これらの外部アプリケーションの切替は大域変数 draw_renderer に 'gnuplot_pipes', 'gnuplot' あるいは 'vtk' の何れか一つを指定することで行えます. なお, この大域変数の既定値は 'gnuplot_pipes' で, 'vtk' で描画が行えるのは 3 次元グラフに限定されます. これらの外部アプリケーションを利用するためには利用する環境に予めインストールされていなければなりません. ちなみに MS-Windows 版の Maxima には GNUPLOT が同梱されているので, GNUPLOT の描画を行うことに問題はありません. もし, vtk をインストールした場合は, vtk が Maxima から呼び出せるように環境変数 Path に vtk の exe ファイルへの経路を追加しておく必要があります.
描画アプリケーションとして GNUPLOT が利用されるのは大域変数 draw_renderer の値が 'gnuplot_pipes', あるいは 'gnuplot' のときです. 'gnuplot_pipes' と 'gnuplot' の両者の機能の違いは表からは判り難いものです. まず, 両者共に制御用ファイル maxout.gnuplot とデータファイル data.gnuplot の二種類のファイルを生成し, これらのファイルを GNUPLOT に引き渡します. ここで maxout.gnuplot ファイルには属性に対応する設定文とデータファイル data.gnuplot の所在が記入され, data.gnuplot ファイルには描画対象の座標データのみが出力され, その内容は plot2d 函数や plot3d 函数が出力する maxout.gnuplot_pipes ファイルとほぼ一致します[7]. なお, これらの中間ファイルの名前は後述の属性で変更できます. このように中間ファイルの名前やその内容に大きな違いがありませんが, draw 函数の内部の処理には大きな違いがあります. つまり, 'gnuplot_pipe' の場合は LISP のストリームを用いた処理, 'gnuplot' の場合は単純に system 函数で GNUPLOT を立ち上げて描画するという違いです. より詳細には, LISP のストリームが用いられるのは,Maxima の動作環境が MS-Windows ではない環境で, 属性 terminal の値が 'sceen', 'wxt' か 'aquaterm' であり, 複数のグラフ表示ではないことを示す内部変数*multiplot-is-active* が 'nil' であり, 大域変数 draw_renderer の値が 'gnuplot_pipes' のときです. このときに plot2d 函数や plot3d 函数で用いられている plot.lisp で定義された send-gnuplot-command 函数を用いて実際の描画が行われれています. それ以外は Maxima の system 函数を使って maxout.gnuplot を GNUPLOT に読み込ま

[6]正確には VTK の TCL ラッパーの vtk です.
[7]maxout_guplot の方が無駄に桁数が多いようです.

せて描画させます. このように MS-Windows では L ISP のストリームを用いた処理は行われておらず, MS-Windows 環境は UNIX 環境とは異なり, 描画ウィンドウを閉じるまで Maxima の操作が一切できません.
GNUPLOT の描画では環境によって LISP のストリームを用いた描画が可能ですが, VTK を用いる場合は, ストリームを用いた描画は機能的にありません. VTK を描画用アプリケーションとして利用するためには環境変数 draw_renderer を予め 'vtk' としなければなりません. また, グラフも 3 次元グラフのみに対して利用可能です. そして, x 描画の際には GNUPLOT と同様に制御用のファイル maxout.tcl とデータファイル data.vtk の二種類のファイルがホームディレクトリ上に生成され, これらのデータを用いて vtk による可視化が実行されます. なお, vtk に引き渡される中間ファイルの名前は GNUPLOT の場合と異なり, 属性を使って名前を変更することができない固定の名前となります. また, vtk による表示を行っている間は Maxima の操作は一切できません. そして, draw パッケージは GNUPLOT の利用を前提にしているため, GNUPLOT で可能なことが vtk でも可能であるとは限りません.
最後に draw パッケージの描画函数を用いて幾つかの形式の画像ファイルを直接生成することが可能です. このときの画像への変換は GNUPLOT や vtk を用いています. 詳細は §11.7.7 の属性 terminal を参照して下さい.

11.7.3　対象とその属性について

draw パッケージで実際の描画を行う函数である draw 函数は plot2d 函数のように式をそのまま描画するのではなく, 式等の表示すべきものを対象として定義し, 対象の色や見え方を属性で表現した方式となっています.
ここでは最初に draw 函数を 2 次元グラフ描画向けに限定した draw2d 函数を使って 2 次元グラフの描画を行ってみましょう：

```
draw2d(explicit(sin(x)/x,x,-10,10));
```

この出力を図 11.52 に示しますが, この例では draw2d 函数の引数となっている explicit 文の中に表示すべき数式が明示的に含まれています. 実際, 描画すべき式が '$\sin x/x$' で, 式の変数が x, 変数の動く範囲が $[-10, 10]$ であることを示しています:

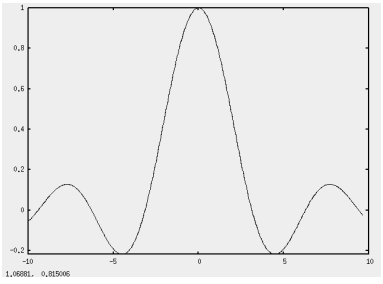


図 11.52: draw2d による単純なグラフ表示

ここでの explicit 文は描画対象の式の変数の定義域が明示的に与えられ, その結果, 値域も明確な対象であることを意味します. この例では定義域を [−10, 10] としており, 実際の描画もこの範囲内で行われ, 値域に適合するように Y 軸側も調整してグラフの表示を行っています. では, 描画範囲を定義域とは別に指定できないのでしょうか？draw パッケージではこのよう描画範囲を定める目的の属性として xrange, yrange や zrange といった属性があります. ここでは変数 x の描画範囲を区間 [−20, 20] にしてみましょう:

```
draw2d(xrange=[-20,20],explicit(sin(x)/x,x,-10,10));
```

この属性 xrange に描画すべき領域をリストとし, 演算子 “=” を用いて値を指定しています. このように draw 函数では演算子 “=” の左辺に属性名を置き, 右辺に属性値を指定します. この結果を図 11.53 に示しますが, explicit 文内の定義域が [−10, 10] であるのに対し, グラフ枠の X 軸側が [−20, 20] であることに注意して下さい:

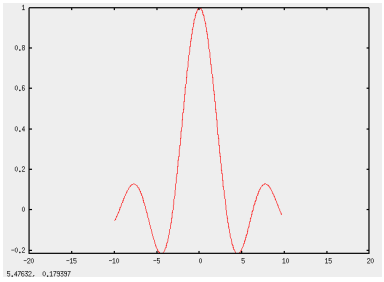


図 11.53: xrange による X 軸側の表示領域設定

draw パッケージの大きな特徴に描画すべき式を一つの対象として扱う点が挙げられます. これは従来の plot2d や plot3d 函数による描画では描画すべき式と領域の情報は一旦描画を実行してしまえば再利用は函数そのものを再度入力する以上のことはできません. ところが draw 函数では対象の定義式や対象の存在範囲等の情報から描画すべき対象を構築する方式であるために, 次に示すように柔軟に対象を扱うことができます:

```
(%i4) neko:explicit(sin(x)/x,x,-10,10);
                                  sin(x)
(%o4)                  explicit(------, x, - 10, 10)
                                    x
(%i5) X1:xrange=[-20,20];
(%o5)                       xrange = [- 20, 20]
(%i6) Y1:yrange=[-0.5,1.5];
(%o6)                      yrange = [- 0.5, 1.5]
(%i7) draw2d(X1,neko,Y1);
(%o7)                        [gr2d(explicit)]
```

この例では explicit 文を変数 neko に, 属性の指定を行う式を変数 X1, Y1 に割当て, それらを draw2d 函数を用いて描画させています. この結果を図 11.54 に示しておきますが, このように描画すべき式とその式が定義された領域といった最も基本的な情報を含む explicit 文を一つの対象として扱うことができるのです:

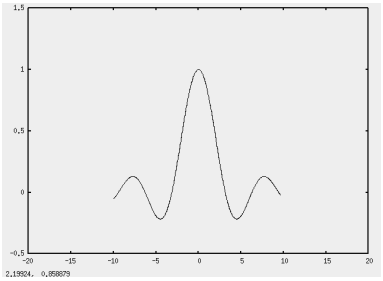


図 11.54: draw2d の表示例

この例で表示領域の指定を行う式は ‘xrange=[-20,20]’ の書式になります. この式の左辺の変数のことをここでは属性と呼び, 右辺の値のことを設定値と呼びます. この例では ‘xrange’ が属性, ‘[-20,20]’ がその属性値です.
この属性 xrange はグラフの X 軸側の表示する範囲を定める属性です. 属性には, このようにグラフの表示範囲を定めたり, グラフの軸の見え方や目盛やラベルの様々な指定, グラフの表題や凡例に関する設定, それと表示する曲線や曲面の色や点線, 破線, 実線やワイヤーフレームか曲面を貼ったソリッド表示にする等の細かな設定等, グラフやグラフに表示される各種の対象の見え方を指定する属性があります. このような属性は対象とグラフに密着して狭い範囲の見え方に影響を及ぼすものであるために**局所的属性**と呼びます. なお, 表示端末の指定, 画像への出力の指定や各種グラフの組合せといった全体に影響を与える**大域的属性**という属性もあります. これらの属性は働きが異なっていますが, その描画函数での配置にも違いがあります. まず, 局所的属性は必ず関係する対象の前に置かなければなりません:

```
(%i28) nekoneko:points([1,2,3],[1,2,3]);
(%o28)                        points([1, 2, 3], [1, 2, 3])
(%i29) draw2d(xrange=[-1,6],
              yrange=[-1,6],
              nekoneko,
              point_size=3,
              point_type=diamant,
              points_joined=true,
              line_type=dots,
              color=red,
              nekoneko);
(%o29)                              [gr2d(points, points)]
```

ここで用いられている対象 points は平面内の点列を定義する対象で, point_size, point_type, points_joined, line_type や color といった局所的属性を持っています. これらの属性についてはあとの小節で詳細を述べますが, 点の大きさ, 点の型, 点列の繋く線分の型と色に関連する属性で, この属性の指定による描画が図 11.55 のものです:
この図に示すように最初の点 nekoneko はあとに続く属性の影響を受けておらず, 領域の設定に留まっています. それに対して二つ目の nekoneko では, 点の大きさが 3, その型が diamant で, 各点を点線で結び, 点と点線の色が赤と, 最初の nekoneko に続く属性の指定に従ったものとなっています. このように draw2d 函数や draw3d 函数では, 引数の先頭から解釈が行われ, 色等の対象の見

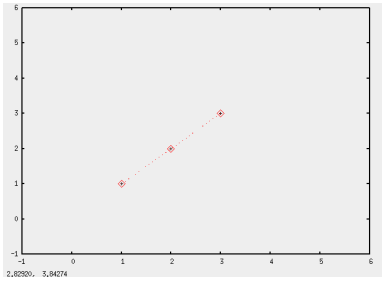


図 11.55: 点と属性の関係

せ方に直接関係する属性は先行する対象に対してではなく, うしろに続く対象全てに対して影響を与えます.
ところが, 大域的属性は描画の全般的な動作を制御する性格の属性であるために, 局所的属性と比べて配置も自由である点で大きく異なります. この点について, draw パッケージの描画函数である draw 函数, draw2d 函数と draw3d 函数の引数と一緒に述べることとします.

11.7.4 draw 函数と draw2d 函数,draw3d 函数との関係

draw2d 函数と draw3d 函数は 2 次元と 3 次元の対象を専門に描く函数です. そして, 複数の対象を表示できるとは言え, それは同じ次元で同一の座標系に重ねて描くことだけで, この機能に限定してしまえば従来の plot2d 函数や plot3d 函数と大差はありません. ここで函数や座標リスト等が表現する形状を対象にするだけではなく, その形状が配置されるグラフそのものを対象にして自由に扱えるようにしてしまえばどうでしょう. そのために 2 次元グラフ全体や 3 次元グラフ全体を対象としてしまうと, 今度はそれら全体をどのように表示するかが問題になります. この「**全体をどのように表示するか**」を制御するのが「**大域的属性**」, 各グラフの中で「**対象をどのように見せるか**」を制御するのが「**局所的属性**」です. その性質上, 大域的属性と違い, 局所的属性はグラフの中に包含される属性となります. そして大域的属性とグラフを纏めて処理する函数が draw 函数です. さらに draw2d 函数は draw 函数を 2 次元のグラフに限定したもの, draw3d 函数は draw 函数を 3 次元のグラフに限定したものとしての特徴を持ちます:

draw2d, draw3d と draw との関係

draw2d(⟨ 対象$_1$⟩,...,⟨ 対象$_n$⟩)	⇒	draw(gr2d(⟨ 対象$_1$⟩,...,⟨ 対象$_n$⟩))
draw3d(対象$_1$,...,対象$_n$)	⇒	draw(gr3d(⟨ 対象$_1$⟩,...,⟨ 対象$_n$⟩))

ここで draw2d 函数や draw3d 函数に対応する draw 函数の構文に現われる 'gr2d' と 'gr3d' は各グラフの**構築子 (Constructor)** としての役目を請負います. 以降, gr2d や gr3d で構成された文のことをまとめて **gr 文**, gr2d や gr3d を用いた文を **gr2d 文**, **gr3d 文**, と呼ぶことにします. これらの gr 文が従来の plot2d 函数や plot3d 函数で描いていたグラフそのものに対応します.
draw2d 函数と draw3d 函数の内部的な動作を簡単に説明すると, これらのの引数として与えた列は各函数の内部で '($grd2)' や '($gr3d)' を Lisp の cons 函数で結合したリストを構築し, それを

draw 函数に引渡すという処理を行なっています. それから draw 函数では与えられたリストの成分で先頭が '=' である成分に対し, その成分が属性値の設定と判断して対応する GNUPLOT の set 文を使った命令文を生成し, あとは gr 文の先頭が '$gr2d' であれば内部函数 make-scene-2d, '$gr3d' であれば内部函数 make-scene-3d を用いてグラフ処理を行っています.

11.7.5　draw 函数

draw 函数は複数の gr2d 文や gr3d 文の列を左から順番に表示する函数です. この draw 函数では対象が 2 次元, あるいは 3 次元であるかに応じて gr2d 文か gr3d 文を用います. ただし, 同じ次元の対象を一つの gr2d 文や gr3d 文で記述するか, 複数の gr2d 文や gr3d 文で記述するかで結果が異なります:

draw 函数の構文 1

A_1	draw(gr2d(⟨ 属性$_1$⟩,...,⟨ 属性$_n$⟩,⟨ 対象 ⟩))
A_2	draw(gr2d(⟨ 属性$_{1\ 1}$⟩,...,⟨ 属性$_{1\ n}$⟩,⟨ 対象$_1$⟩,..., ⟨ 属性$_{m\ 1}$⟩,...,⟨ 属性$_{m\ n}$⟩,⟨ 対象$_m$⟩))
B_1	draw(gr3d(⟨ 属性$_1$⟩,...,⟨ 属性$_n$⟩,⟨ 対象 ⟩))
B_2	draw(gr3d(⟨ 属性$_1$⟩,...,⟨ 属性$_n$⟩,⟨ 対象 ⟩))
C	draw(⟨gr 文$_1$⟩,...,⟨gr 文$_m$⟩)

ここで示した構文は最も基本的な構文を示しています. まず最初の A_1 の draw 文は対象が 2 次元で対象が一つだけの場合で, この場合は draw2d 函数との機能的な違いはありません. 次の A_2 の draw 文は複数の 2 次元対象を同時に表示させる場合です. ここで ⟨ 対象$_i$⟩ から ⟨ 対象$_{i+1}$⟩ で挟まれた ⟨ 属性$_{j\ i+1}$⟩ が ⟨ 対象$_{i+1}$⟩ の属性となります. 同じ属性で相異なる値の指定があれば, 対象の直前の指定が採用されます. この表記も draw2d 函数と同様の結果が得られます.
それから B_1 と B_2 は 3 次元対象の描画で, B_1 が A_1 に, B_2 が A_2 に対応し, これらの表記は draw3d 函数を用いたものと同様の結果が得られます.
最後の C は一つのウィンドウ内部に m 個のグラフを表示する場合の書式です. この書式では, 2 次元と 3 次元の gr 文が混在しても構いませんが, この表記は draw2d 函数や draw3d 函数では実現できません. また, draw 函数で描画を行う際に大域変数 draw_renderer を 'vtk' の状態で 2 次元と 3 次元の gr 文が混在していればエラーになります.

では実例を幾つか挙げておきましょう. 最初に正弦函数と放物線を一つの gr2d 文で記述した例を示しておきます:

```
(%i6) draw(gr2d(explicit(sin(x),x,-10,10),
          explicit(x^2+1,x,-2,2)));
(%o6)                          [gr2d(explicit, explicit)]
```

図 11.56 に示すように一つのグラフの中に正弦函数と放物線が記述されていますね. この draw 函数の応答からも一つの gr2d 文の中に explicit 文が二つありますが, このことから一つのグラフの中に二つの対象が同時に描かれていることが判るのです.

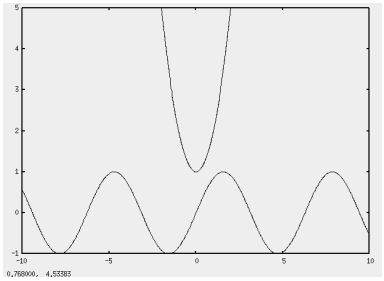


図 11.56: 一つの gr2d 文の場合

そこで, 今度はこれらの正弦函数と放物線を別々に gr2d 文で生成し, draw 函数で表示させて見ましょう:

```
(%i7) draw(gr2d(explicit(sin(x),x,-10,10)),
           gr2d(explicit(x^2+1,x,-2,2)));
(%o7                    [gr2d(explicit), gr2d(explicit)]
```

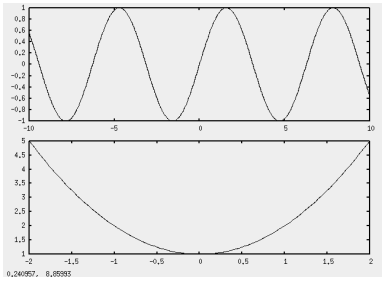


図 11.57: 二つの gr2d 文の場合

図 11.57 から判るように今度は別々にグラフが表示され, 正弦函数のグラフが上, 放物線のグラフが下に描かれています. さらに draw 文の応答は二つの gr2d 文を成分とするリストになっています. このように draw 函数では gr2d 文や gr3d 文で構成される列の先頭から順番に描きます.

11.7.6 属性と draw 函数

draw 函数で複数の gr 文の列を表示させる場合, 通常は縦一列にグラフを表示します. では, これを複数の列で表示させることができるのでしょうか？ 結論から言えば, draw 函数では容易にできることで, そのために属性 columns の値で列数を指定すればよいのです. ところでこの属性 columns の指定はどこに書き込めばよいのでしょうか？ 属性なので draw 函数の引数にある各 gr 文に書き込めばよいのでしょうか？前述のように draw パッケージの属性には, gr 文で表現されるグラフ (=draw2d 函数や draw3d 函数で表示できるグラフ) の中の対象の色, 位置, 座標系の設定等の局所的なグラフの調整のみを行う**局所的属性**, それと gr 文で表現されるグラフ全体を何列で表示するか

といった全体の表示を調整する**大域的属性**の二種類があります. ここで属性 columns は複数の gr 全体の表示を調整する属性のために大域的属性となります.
さて, 大域的属性は局所的属性と draw 文での扱いの違いはあるのでしょうか？局所的属性は各グラフと密接に関わりますが, 大域的属性は全体に関係するもので, 個々のグラフと密接に関係するものではありません. そのために局所的属性は gr 文に強く拘束され, その順番も関連する対象に依存します. ところが, 大域的属性は全体に関わるために特定の gr 文に拘束されません. このことは大域的属性の文は, draw 函数内部の全ての gr 文に置く必要もなく, gr 文の外に出して gr 文とほぼ対等に扱っても良いこと, つまり, 大域的属性の文は draw 函数の中に自由に置けることを意味します. そこで次の構文が最後に挙げられます:

draw 函数の構文 2(大域的属性を含む場合)

D	draw($\langle S_1 \rangle, \ldots, \langle S_m \rangle$)

ここで $\langle S_i \rangle$ は大域的属性の設定文, または gr 文の何れかとなります. なお, 同じ大域的属性の設定文を複数記述した場合, 最後 (=最も右側) にある設定文による設定が採用されます. また, 局所的属性を大域的属性のように gr 文の外に出すとエラーになるので注意して下さい.

では, 属性 columns の値を色々変更してどのようなグラフが描けるかを確認してみましょう:

```
(%i15) draw(columns=1,gr2d(explicit(sin(x),x,-10,10)),
                      gr3d(explicit(x*y,x,-2,2,y,-2,2)),
                      gr2d(explicit(x^2-x+1,x,-2,2)));
(%o15)          [gr2d(explicit), gr3d(explicit), gr2d(explicit)]
```

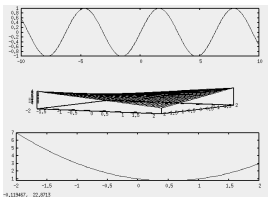


図 11.58: columns=1 の場合

ここでの例では最初が正弦函数, 次が xy で最後が $x^2 - x + 1$ のグラフを描かせています. このように draw 函数では次元が異なるグラフも一緒に表示することができます. ここでは 'columns=1' のために図 11.58 では正弦函数から順に一列に表示されています. そこで今度は 'columns=3' にするとどうなるでしょうか？

```
(%i16) draw(columns=3,gr2d(explicit(sin(x),x,-10,10)),
                      gr3d(explicit(x*y,x,-2,2,y,-2,2)),
                      gr2d(explicit(x^2-x+1,x,-2,2)));
(%o16)          [gr2d(explicit), gr3d(explicit), gr2d(explicit)]
```

この例では属性 columns を先頭に置いています. この属性は大局的属性であるために draw 函数の末尾でも, gr 文の中でも記述しても構いません. この例の結果は図 11.59 に示すように 3 列で各グラフが列の順に表示されています:

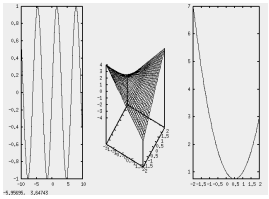


図 11.59: columns=3 の場合

最後に 'columns=2' としたときの例を示しておきましょう:

```
(%i17) draw(columns=2,gr2d(explicit(sin(x),x,-10,10)),
                      gr3d(explicit(x*y,x,-2,2,y,-2,2)),
                      gr2d(explicit(x^2-x+1,x,-2,2)));
(%o17)          [gr2d(explicit), gr3d(explicit), gr2d(explicit)]
```

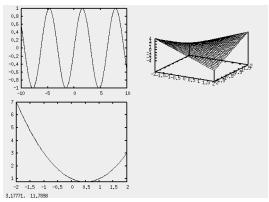


図 11.60: columns=2 の場合

このように対象が 3 個のために最初の 2 個が同じ行に表示され, 3 番目の対象が一段下に表示されています. ここで 3 次元グラフの視点の変更は全く行えません. これは GNUPLOT 上で対象を操作するために mouse を 'on' にすることができるのは GNUPLOT で最後に描画した対象, すなわち, draw パッケージの描画函数の引数として与えた列で最後の対象に限定されるからです.
属性 columns のような大域的属性は draw 文の何処でも, gr 文の任意の位置にさえ置いても構いませんが, 個々の対象に直接関連する局所的属性は, 関連する対象を含む gr2d 文や gr3d 文の外に置くことができません. たとえば属性 xrange はグラフの X 軸方向の表示領域を定める局所的属性であるために大域的属性 columns のように gr 文の外に出すことはできません. このことを確認しておきましょう:

```
(%i6) draw(xrange=[-1,1],gr2d(explicit(sin(x)/x,x,-0.5,0.5)));
Unknown global option xrange
 -- an error.  To debug this try debugmode(true);
(%i7) draw(gr2d(explicit(sin(x)/x,x,-0.5,0.5)),xrange=[-1,1]);
Unknown global option xrange
 -- an error.  To debug this try debugmode(true);
```

draw パッケージで GNUPLOT の利用が半ば前提となっていますが, 各種の属性には対応する GNUPLOT の表示属性があり, 内部では set 文を構築して GNUPLOT 側に反映させる仕組になっています. これらの属性の既定値は draw パッケージ内の grcommon.lisp で設定されていますが,

大域変数とは異なり Maxima 側から属性名を入力すれば属性値の表示が行えるというものではありません.

11.7.7　大域的属性について

大域的属性はグラフ出力の制御を中心とした gr 文に依存しない属性です. ここでは端末の設定, ファイル名の設定, 画像の大きさの設定と GNUPLOT 命令の指定に分類して解説します.

出力端末に関連する大域的属性

出力端末の指定に関連する大域的属性を以下に纏めておきます:

出力端末の設定に関連する大域的属性

属性	型	既定値	概要
columns	正整数値	1	グラフの列数
terminal	文字列	screen	グラフの書式の指定
delay	正整数値	5	GIF アニメーションの遅延時間
background_color	文字列	#ffffff	背景色
file_bgcolor	文字列		背景色 (非推奨)

属性 colums:　draw 函数で複数の gr2d 文や gr3d 文の列を描くときに表示方法を指定する属性です. 通常, columns の初期値は 1 のために columns に値を指定しなければ draw 函数は gr2d 文と gr3d 文の列を左から順番に縦一列に表示します. より一般的に columns を正整数値 n, gr2d と gr3d の列を $a_1, a_2, \ldots, a_m$ とするときに, 部分列 $a_1, \ldots, a_n$ を左から順番にグラフの 1 行目, 部分列 $a_{n+1}, \ldots, a_{2n}$ をグラフの 2 行目と以後同様に表示を行います. このときグラフの行数は m/n を越える最小の整数になります.
外部アプリケーションに GNUPLOT を用いているときに, 座標の読み取りや 3 次元グラフの回転が行える gr 文は, draw 函数の引数として与えられた gr 文の列 $a_1, a_2, \ldots, a_m$ に対して a_m のみが該当します. したがって, グラフの座標の読取, グラフの拡大・縮小, 3 次元グラフの回転は a_m のグラフに対してのみで, その際にグラフの再描画が行われると a_m のグラフのみが再表示されて他のグラフが消えてしまいます.
なお, 外部アプリケーションで vtk を利用する場合, 引数の列に含まれる全ての gr 文のマウスによる操作が可能です. ただし, vtk は 3 次元対象の表示のみが可能なため, GNUPLOT のように 2 次元と 3 次元の対象が混在したグラフ表示は行えません.

属性 terminal:　GNUPLOT の terminal に対応する属性で, 基本はグラフ出力先の端末を指定しますが, 画像をファイルの出力する際のファイル形式の設定にも使えます. draw 函数で利用可能な値を次に纏めておきます:

terminal の取り得る値

値	概要
screen	既定値
aquaterm	MacOSX 上の AquaTerm 向け
wxt	wxWidgets を利用する場合
[⟨ 端末 ⟩, ⟨ 正整数 ⟩]	⟨ 端末 ⟩ が screen, wxt, aquaterm のときウィンドウ ID を指定して描画
dumb	ASCII-Text で端末に表示させる場合
dumb_file	ASCII-Text でファイルに出力
png	PNG 形式で出力
pngcairo	PNG 形式で出力 (cairo+Pango)
eps	EPS 形式で出力
esp_color	EPS 形式で出力
pdf	PDF 形式で出力
pdfcairo	PDF 形式で出力 (cairo+Pango)
jpg	JPEG 形式で出力
svg	SVG 形式で出力
gif	GIF 形式で出力
animated_gif	GIF-アニメーションを出力
tiff	TIFF 形式で出力
obj	Wavefront の Obj 形式で出力
pnm	PNM 形式で出力
vrml	VRML 形式で出力

ここでの属性値の値は Maxima の文字列とは異なり二重引用符で括る必要がはりません. 既定値は 'screen' で, 環境に適合した外部アプリケーションが立ち上げられます. 属性 terminal に指定可能な値は環境毎に異なります. 実際, wxWidgets がインストールされた環境や MS-Windows 環境であれば 'wxt', MacOS X 環境であれば 'aquaterm' が使えます. また 'dumb' に指定すればテキスト端末上でグラフを ASCII-Art 風に表示し, 'dumb_file' を指定すればテキストファイルに ASCII-Art 風のグラフ出力ができます.
出力端末を 'screen', 'wxt', 'aquaterm' とする場合, 表示ウィンドウに ID 番号を指定して表示させることもできます. この場合は, 2 成分のリスト: '[⟨ 端末 ⟩, ⟨ 正整数 ⟩]' を指定することになります. たとえば, '[wxt, 1]' を属性 terminal に設定すると描画ウィンドウの id が 1 の wxWidget を用いた GNUPLOT ウィンドウにグラフが表示されます. ID を指定しなければ, ID 番号が 0 のウィンドウにグラフが表示されます.
画像ファイルに出力する場合, その保存形式の指定を行います. 画像ファイルの生成では GNUPLOT や vtk を用いるので, これらのアプリケーションがインストールされ, Maxima から呼び出し可能であるように環境設定が行われていなければなりません.
まず, GNUPLOT で生成可能な形式には, JPEG 形式, EPS 形式, GIF 形式, PDF 形式, PNG 形式と SVG 形があります. さらに PDF 形式と PNG 形式に関してはグラフィックスでは cairo を利用し,

文字の表示では Pango を用いる ‘pdfcairo’ や ‘pngcairo’ が選択できます. これらのデータ出力の指示は中間ファイルの修飾子が ‘gnuplot’ のファイル (既定値は ‘maxout.gnuplot’) の 2 行目で’set out’ 文を用いて生成されます.
vtk で生成可能な形式には, GNUPLOT と同じ PNG 形式, JPG 形式, EPS 形式に加え, GNUPLOT では対処できない VRML 形式, Obj 形式, TIFF 形式や PNM 形式があります. これらの形式の出力では vtk を利用するために予め大域変数 draw_renderer の値を ‘vtk’ に設定した上で属性 terminal を ‘vrml’ 等の希望する形式に指定します. これらの形式のファイルは UNIX 環境ならホームディレクトリ上, あるいはカレントディレクトリ上, MS-Window 環境であれば Maxima をインストールしたフォルダ内の wxMaxima フォルダ内部に生成されます[8]. VRML 形式のファイルの修飾子は ‘wrl’, Obj 形式のファイルであれば ‘obj’, TIFF 形式は ‘tif’, PNM 形式であれば ‘pnm’ となります. なお, vtk で表示できる対象は 3 次元の対象に限定されるため, これら 4 種類の形式で出力が可能な対象も 3 次元対象に限定されます. なお, VRML 形式や Obj 形式のファイルの可視化は ParaView[9]を用いると良いでしょう.
このように属性 terminal はその環境にインストールしてある GNUPLOT や VTK で利用可能な端末や形式に限定されます. したがって, draw パッケージで利用可能なものでも, その環境上の GNUPLOT や VTK で利用できなければ, 当然, Maxima から使うことはできません.

属性 delay:　属性 terminal の値が ‘animated_gif’ の場合にのみ意味がある属性です. この属性は GIF アニメーションでの遅延時間を指定し, その単位は 1/100[sec] です. この既定値は 5(=5/100[sec]) です. なお, VTK は animated_gif に対応していないために, この属性は利用できません.

属性 background_color:　グラフの背景色を定める属性です.　この属性が有効になるのは属性 terminal で ‘vrml’ や ‘obj’ 以外の画像ファイルの形式を選択した場合に限定されます. ここで指定する値は予め定義されている色名か 6 桁の 16 進数を基にした文字列のいずれかを指定します. たとえば RGB で赤が 255(=ff), 緑が 16(=10) 青が 0(=00) を指定したければ, 文字列として”#ff1000” を指定します. なお, 既定値は#ffffff(=白) です. この色の名前の詳細は §11.7.9 を参照して下さい. また, この属性は GNUPLOT でも vtk でも利用できます.

属性 file_bgcolor:　属性 background_color と同様に画像ファイルの背景色を定める属性で, vtk でも利用可能ですが, 現在, こちらの属性の利用は推奨されておらず, 属性 background_color を用いるように警告が出ます.

ファイル名に関連する属性について

ここで解説する大域的属性はグラフ表示の際に生成される中間ファイルの名前を設定するものです:

[8] MS-Windows 環境であれば, 必ず vtk.exe への Path の設定を忘れないように注意しましょう.
[9] http://www.paraview.org/

ファイル名に関連する大域的属性

属性	型	既定値	概要
file_name	文字列	”maxima_out”	画像の出力ファイル名
data_file_name	文字列	”data.gnuplot”	データファイル名
gnuplot_file_name	文字列	”maxout.gnuplot”	GNUPLOT の出力ファイル名

属性 file_name: GNUPLOT や vtk を用いて生成するときに画像ファイルの名前を指定する属性です. ここで指定するファイル名は二重引用符で括られた Maxima の文字列でなければなりません. ファイル名に画像形式を示す修飾子は不要です. 画像形式を示す修飾子は Maxima 側で付けます. たとえば terminal の値を ‘png’, file_name の値を”tama”と設定していれば, 画像ファイル名は ‘tama.png’ になります.

属性 data_file_name: GNUPLOT による描画で用いるデータファイル名を定める属性で, 既定値は”data.gnuplot” です. ここで指定する文字列は Maxima の二重引用符で括られた文字列でなければなりません. また, GNUPLOT に与えるデータファイルを指定する属性であるために, 大域変数 draw_renderer の値が ‘vtk’ を指定した状態で, この属性の設定が draw 函数や draw3d 函数にあるとエラーとなるので注意が必要です. vtk で処理させるファイルの名前は固定であり, その名前の変更のためにこの属性は使えません.

属性 gnuplot_file_name: GNUPLOT で処理させるための制御文等が記載されたファイル名を指定する属性です. 既定値は ”maxout.gnuplot” です. ここで指定するファイル名は二重引用符で括られた Maxima の文字列でなければなりません. この属性は GNUPLOT でのみ利用可能で, vtk では使えないので注意して下さい.

画像の大きさに関連する大域的属性

ここでの属性は画像の大きさに関連する大域的属性です. ただし, 属性 dimensions を除くと他の属性は古い名前のもので利用することは推奨されていません:

画像の大きさに関連する大域的属性

属性	型	既定値	概要
dimensions	正整数値	[600,500]	ウィンドウの大きさ
pic_width	正整数値	640	画像ファイルの幅 (非推奨)
pic_height	正整数値	480	画像ファイルの高さ (非推奨)
eps_width	正整数値	12	EPS 形式のファイルの幅 (非推奨)
eps_height	正整数値	8	EPS 形式ファイルの高さ (非推奨)
pdf_width	正整数値	12	PDF 形式のファイルの幅 (非推奨)
pdf_height	正整数値	8	PDF 形式のファイルの高さ (非推奨)

属性 dimensions: GNUPLOT や vtk で表示するウィンドウや画像の大きさを指定する属性で, Maxima の 2 成分の整数リストを割当てます. この属性の既定値は '[600,500]' なので, この属性の値が無指定であれば, 横 600 画素, 縦 500 画素のグラフが生成されます. もし, 1000 × 900 にしたければ次のように設定します:

```
(%i217) draw2d(terminal=wxt,dimensions=[1000,900],explicit(sin(x)/x,x,-10,10));
```

この属性は描画ウィンドウの大きさに限りません. 画像をファイルに出力する際の画像の画素数にも対応します:

```
(%i217) draw2d(terminal=jpg,dimensions=[1000,900],explicit(sin(x)/x,x,-10,10));
```

この例では属性 terminal を jpg に変更することで JPEG 形式の画像ファイルの生成を行う指定となっています. このとき dimensions の値として [1000, 900] が割当てられているために, 画像ファイルは 1000 画 × 900 画素 の大きさで生成されます.
この属性 dimension の他に画像の大きさを指定する属性に*_width や*_height の名前の画像別に幅と高さを指定する属性があります. これらは現在, 利用が推奨されておらず, この属性 dimension を用いることが推奨されています. また, vtk でこれらの属性は使えません. 以下に属性毎に説明をしていますが, それらの属性の利用は避けて下さい.

属性 pic_width と属性 pic_height: 属性 dimension を用いることが推奨されている属性です. pic_width が画像の幅, pic_height が画像の縦方向の大きさを画素単位で指定します.

属性 eps_width と属性 eps_height: 属性 dimension を用いることが推奨されている属性です. 画像形式が PostScript や EPS の場合, ここで画像の幅と縦方向の大きさを指定します.

属性 pdf_width と属性 pdf_height: 属性 dimension を用いることが推奨されている属性です. 画像形式が PDF の場合, ここで画像の幅と縦方向の大きさを指定します.

GNUPLOT に命令文を送り込むための大域的属性

GNUPLOT に命令を送り込むための大域的属性

属性	型	既定値	概要
user_preamble	文字列リスト	""	GNUPLOT に引き渡す命令文

属性 user_preamble: user_preamble は GNUPLOT で処理させるべき命令を GNUPLOT に送り込むために用いる属性です. ここで命令文が一つだけであれば, その命令文を一つの文字列として属性 user_preamble の値とします. 複数の命令文を送り込む必要があれば, 各命令文を文字列に置き換えたもののリストを指定します. この属性が指定されると属性 terminal の値は gnuplot になり, maxout.gnuplot と data.gnuplot の二種類の中間ファイルが生成されます. この属性の性格上, vtk では利用不可です.

11.7.8 対象について

draw と worldmap パッケージで定義できる対象と文の一覧を示します:

文と定義される対象の一覧

文	対象	概要
points 文	points, points3d	点列を表現
errors 文	errors	十字や線分
polygon 文	polygon	多角形
triangle 文	triangle, triangle3d	三角形
quadrilateral 文	quadrilateral, quadrilateral3d	四辺形
rectangle 文	rectangle	長方形
ellipse 文	ellipse	楕円
label 文	label	ラベル
bars 文	bars	棒グラフの棒
vector 文	vector, vector3d	ベクトル場
explicit 文	explicit, explicit3d	陽的な曲線や曲面
region 文	region	条件を充す領域
implicit 文	implicit, implicit3d	陰的な曲線や曲面
elevation_grid 文	elevation_grid	曲面
image 文	image	行列,picture
mesh 文	mesh	曲面
parametric 文	parametric, parametric3d	曲線の媒介変数表示
polar 文	polar	曲線の極座標表示
spherical 文	spherical	球面
cylindrical 文	cylindrical	円柱
parametric_surface 文	parametric_surface	曲面の媒介変数表示
tube 文	tube	断面が円となる図形
geomap 文	geomap	世界地図

これらのパッケージには固有の属性を持っている場合もありますが, XYZ 軸の表示に関する属性のように共通する属性も多く持っています. この共通の属性についても, 表示端末の指定のような大域的なものから対象の色のように対象に一つ一つに密接に関わる局所的なものに分類されます. ここでは属性について大域的なものを最初に説明し, それから局所的なものについては実例を踏まえて述べることとします. そのために対象を定義する各構文は後回しにします.

11.7.9　局所的属性について

色彩の属性

ここでは draw パッケージの函数で利用可能な色について解説しておきます. draw パッケージで利用できる色は RGB の組み合わせで 6 桁の 16 進数として指定します. たとえば, 白の場合は R(=赤) を 255, すなわち 'ff', 同様に G(=緑) と B(=青) も 255, すなわち 'ff' とするために 'ffffff' となります. 実際に属性として指示するときは 16 進数であることを示すために Maxima の文字列として先頭に記号 "#" を置いて "#ffffff" を色の属性として指定します. なお, 一部の色については "white" のような名前が指定されています. そのような色を次に纏めておきます:

draw パッケージで指定可能な色の定義

色	値	色	値	色	値
white	#ffffff	black	#000000	gray0	#000000
grey0	#000000	gray10	#1a1a1a	grey10	#1a1a1a
gray20	#333333	grey20	#333333	gray30	#4d4d4d
grey30	#4d4d4d	gray40	#666666	grey40	#666666
gray50	#7f7f7f	grey50	#7f7f7f	gray60	#999999
grey60	#999999	gray70	#b3b3b3	grey70	#b3b3b3
gray80	#cccccc	grey80	#cccccc	gray90	#e5e5e5
grey90	#e5e5e5	gray100	#ffffff	grey100	#ffffff
gray	#bebebe	grey	#c0c0c0	light-gray	#d3d3d3
light-grey	#d3d3d3	dark-gray	#a0a0a0	dark-grey	#a0a0a0
red	#ff0000	light-red	#f03232	dark-red	#8b0000
yellow	#ffff00	dark-yellow	#c8c800	green	#00ff00
light-green	#90ee90	dark-green	#006400	spring-green	#00ff7f
forest-green	#228b22	sea-green	#2e8b57	blue	#0000ff
light-blue	#add8e6	dark-blue	#00008b	midnight-blue	#191970
navy	#000080	medium-blue	#0000cd	royalblue	#4169e1
skyblue	#87ceeb	cyan	#00ffff	light-cyan	#e0ffff
dark-cyan	#00eeee	magenta	#ff00ff	light-magenta	#f055f0
dark-magenta	#c000ff	turquoise	#40e0d0	light-turquoise	#afeeee
dark-turquoise	#00ced1	pink	#ffc0c0	light-pink	#ffb6c1
dark-pink	#ff1493	coral	#ff7f50	light-coral	#f08080
orange-red	#ff4500	salmon	#fa8072	light-salmon	#ffa070
dark-salmon	#e9967a	aquamarine	#7fffd4	khaki	#f0e68c
dark-khaki	#bdb76b	goldenrod	#ffc020	light-goldenrod	#eedd82
dark-goldenrod	#b8860b	gold	#ffd700	beige	#f5f5dc
brown	#a52a2a	orange	#ffa500	dark-orange	#c04000
violet	#ee82ee	dark-violet	#9400d3	plum	#dda0dd
purple	#c080ff				

draw パッケージで予め準備された色名[10]を用いる場合, 二重引用符を用いて "white" と指定しても, 二重引用符を付けずに white と指定しても構いません.

[10] grcommon.lisp 内部で定義されています.

色彩の属性

属性	型	既定値	概要
color	文字列	#0000ff	色を指定
palette	リスト	color	パレットを指定
colorbox	文字列	true	colorbox の有無を指定
cbrange	リスト	auto	colorbox の範囲を指定
cbtics	集合	auto	colorbox に目盛を表示
logcb	論理値	false	colorbox の目盛を対数で表示
enhanced3d		none	表面への palette の適用
wired_surface	真理値	false	グリッドの有無を指定
contour	文字列	none	等高線の指定
contour_levels	正整数	5	等高線の数

属性 color: GNUPLOT や vtk で指定可能な対象の色の属性です. 既定値として '#0000ff' (=青) が指定されています. 色の指定については前述の解説と表を参照して下さい.

属性 palette: GNUPLOT や vtk で対象の高度に応じた色彩を指定するために用いる属性です. 後述の局所的属性の enhanced3d と併用する必要があります. この属性 palette の既定値は 'color' です. また, 'gray' を指定することでグレースケール表示となります.
属性 palette は予め用意された函数に対応する番号を指定することで好きな階調に変更することができます. 函数は 37 種類あり, 0 から 36 までの番号が割当てられており, 各函数の定義域は区間 $[0, 1]$ となるように自動的に調整が行われています. この番号については GNUPLOT 上で help palette と入力することで表示されるオンラインマニュアルにも解説があります. この GNUPLOT のマニュアルで三角函数は弧度法ではなく角度となっていますが, 以下に示す表における三角函数は全て弧度法を採用しています:

属性 palette で用いる函数

番号	函数	番号	函数	番号	函数
0	0	1	0.5	2	1
3	x	4	x^2	5	x^3
6	x^4	7	$\sqrt{x}$	8	$\sqrt{\sqrt{x}}$
9	$\sin \pi x/2$	10	$\cos \pi x/2$	11	$\lvert x-0.5\rvert$
12	$(2x-1)^2$	13	$\sin \pi x$	14	$\lvert\cos \pi x\rvert$
15	$\sin 2\pi x$	16	$\cos 2\pi x$	17	$\lvert\sin 2\pi x\rvert$
18	$\lvert\cos 2\pi x\rvert$	19	$\lvert\sin 4\pi x\rvert$	20	$\lvert\cos 2\pi x\rvert$
21	$3x$	22	$3x-1$	23	$3x-2$
24	$\lvert 3x-1\rvert$	25	$\lvert 3x-2\rvert$	26	$(3x-1)/2$
27	$(3x-2)/2$	28	$\lvert(3x-1)/2\rvert$	29	$\lvert(3x-2)/2\rvert$
30	$x/0.32-0.78125$	31	$2x-0.84$		
32	$x<0.25?4.*x:x<0.42?1:x<0.92?-2.*x+1.84:x/0.08-11.5$				
33	$\lvert 2x-0.5\rvert$	34	$2x$	35	$2x-0.5$
36	$2x-1$				

属性 palette の値としては [±番号$_R$, ±番号$_G$, ±番号$_B$] の 3 成分のリストを指定します. このリストは先頭の 番号$_R$ が赤の階調, 番号$_G$ が緑の階調, 番号$_B$ が青の階調となっています. また '−番号' で番号に対応する函数 $f(x)$ の逆の階調 $f(1-x)$ が用いられます. また, この表の 32 番は GNUPLOT の函数式で表記していますが, その具体的な意味を以下に示しておきます:

$$32:\begin{cases} 4x & x<0.25 \\ 1 & 0.25\le x<0.42 \\ -2x+1.84 & 0.42\le x<0.92 \\ x/0.08-11.5 & 0.92\le x \end{cases}$$

この番号は GNUPLOT でも vtk でも共通です. このような数値を指定するだけではかなりの試行錯誤をしなければならないでしょう. 幸いにして GNUPLOT の rgbformulae のヘルプにパレットの例が掲載されています. これに gray で指定するグレースケールのパレットを追加したものを以下に纏めておきます[11]:

[11] 具体的な階調の一覧は http://gnuplot.sourceforge.net/demo/pm3dcolors.html を参照して下さい

RGB 色空間での素敵な組合せ

7,5,15	...	color に対応する典型的な pm3d(黒-青-赤-黄色)
3, 3, 3	...	グレースケール
3,11, 6	...	緑-赤-紫
23,28, 3	...	ocean(緑-青-白), 並び換えを試すと吉
21,22,23	...	hot (黒-赤-黄色-白)
30,31,32	...	グレースケールでプリント可能な配色 (黒-青-紫-黄色-白)
33,13,10	...	rainbow(青-緑-黄色-赤)
34,35,36	...	AFM hot (黒-赤-黄色-白)
3, 2, 2	...	赤-黄色-緑-水色-青-マゼンタ-赤

ここで二つのパレット [30,31,32] と [23,28, 3] で同じグラフを指定した場合の例を示しておきます:

```
(%i11) draw(terminal=eps_color,columns=2,dimensions=[1200,600],
            gr3d(palette=[30,31,32],enhanced3d=true,
                 explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2)),
            gr3d(palette=[23,28,3],enhanced3d=true,
                 explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2)));
(%o11)                  [gr3d(explicit), gr3d(explicit)]
```

ここでは大域的属性 terminal に eps_color を指定することでカラー EPS ファイルを生成させています. この結果は図 11.61 に示すものです:

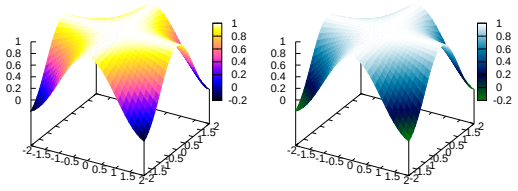


図 11.61: palette の例

さて, 大域変数 draw_renderer が gnuplot_pipes や gnuplot のときに中間ファイルの maxout.gnuplot 内の 'set palette rgbformulae' 文にて属性 palette の値が引き渡されてパレットの指定が行われます. 図 11.61 を描画する際の maxout.gnuplot から左側の図のパレットを定義している rgbformulae 文の箇所を示しておきます:

```
32 set pm3d at s depthorder explicit
33 set colorbox
34 set cblabel ''
35 set palette rgbformulae 30,31,32
36 splot '/home/yokota/data.gnuplot' index 0 t '' w pm3d lw 1 lt 1 lc rgb '#0000ff'
```

また大域変数 draw_renderer が vtk であれば, 中間ファイル maxout.tcl 内の手続 fR1, fG1, fB1 に番号に対応する函数が記述され, vtkLookupTable の lut1 にパレットの値が設定されます. この様子を図 11.61 と同じ絵を vtk で表示したときに生成される中間ファイル maxout.tcl から関連する箇所を示しておきましょう:

```
4  proc unitscale {k} {
5    set n 256.0
6    return [expr $k/($n-1)] }
7
8  proc interval {x x0 x1} {
9    if { $x <= $x0} {return 0}
10   if { $x >= $x1} {return 1}
11   return $x }
12
13 proc fR1 {k} {
14   set x [unitscale $k]
15   set x [expr [interval $x 0.25 0.57] / 0.32 - 0.78125]
16   return [interval $x 0 1] }
17
18 proc fG1 {k} {
19   set x [unitscale $k]
20   set x [expr 2 * [interval $x 0.42 0.92] - 0.84]
21   return [interval $x 0 1] }
22
23 proc fB1 {k} {
24   set x [unitscale $k]
25     if {$x <= 0.42} {
26     set x [expr 4.0 * $x]
27     } elseif {$x <= 0.92} {
28     set x [expr -2.0 * $x + 1.84]
29     } else {
30     set x [expr $x / 0.08 - 11.5]}
31   return [interval $x 0 1] }
32
33 vtkLookupTable lut1
34   lut1 SetNumberOfColors 256
35   lut1 Build
36   for {set i 0} {$i<256} {incr i 1} {
37     eval lut1 SetTableValue $i [fR1 $i] [fG1 $i] [fB1 $i] 1 }
```

この maxout.tcl の内容から判るように, 属性 palette で指定した番号が fR1, fG1 や fB1 で実際の函

数で置き換えられて処理が実行されています.

属性 colorbox: GNUPLOT で 2 次元グラフや 3 次元グラフで表示される colorbox の有無を指定する属性で, vtk では利用できません. 属性 colorbox は等高線表示等で高度が色分けされているときに色分けと高度を関連付ける働きをします. 既定値の 'true' で colorbox が表示され, 'false' で表示されません. また, 任意の Maxima の文字列を指定すると colorbox のラベルとして表示されます:

```
(%i12) draw(terminal=eps_color,columns=2,dimensions=[1200,600],
            gr3d(colorbox=false,palette=[30,31,32],enhanced3d=true,
                 explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2)),
            gr3d(palette=[23,28,3],enhanced3d=true,
                 explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2)))$
```

この例では二つのグラフにて最初の gr3d 文では属性 colorbox を 'false' にし, 第 2 の gr3d 文では既定値のまま (='true') としており, その結果として図 11.62 に示すように左側のグラフには colorbox がなく, 右側のグラフには colorbox があることが判ります:

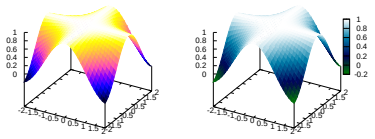


図 11.62: colorbox の有無

属性 cbtics: GNUPLOT で表示したグラフ中の colorbox の目盛を表示するかどうかを指定する属性で, vtk では利用できません. 既定値の 'true' で表示を行い, 'false' で表示を行いません. また, 目盛のラベルを設定することも可能です. この場合, メモリのラベルと対応する数値で構成されるリストを成分とする集合を指定します. たとえば, 属性 cbtics に '{["あつ",55],["よい", 45],["おお",20]}' を割当てると, 数値に対応する箇所にラベルが表示されます:

```
(%i13) draw(gr3d(dimensions=[600,500],
                 cbtics={["あつ",55],["よい",45],["おお",20]},
                 palette=[30,31,32],enhanced3d=true,
                 explicit(40*sin(x*y)/(x*y)+20,x,-3,3,y,-3,3)));
```

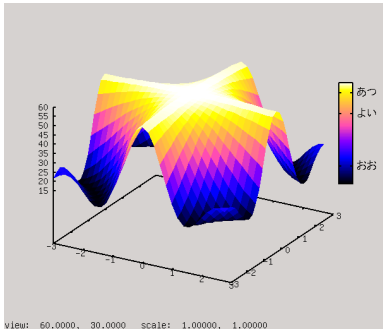


図 11.63: cbtics の設定

属性 logcb:　GNUPLOT で表示したグラフ中の colorbox の目盛を対数目盛で表示するかどうかを指定する属性です. vtk では利用できません. 既定値は 'false' で対数目盛で表示しませんが, 'true' の場合にグラフ中の colorbox の目盛を対数目盛で表示します.

属性 enhanced3d:　GNUPLOT や vtk で表面の描画を指定する属性です. 該当する対象は対象 points3d, implicit3d, explicit3d, elevation_grid, triangle3d, quadrilaterial3d, spherical, cylinder, tube, parametric_3d, parametric3d, parametric_surface, mesh と 3 次元の対象で implicit3d を除く全ての 3 次元対象に存在する属性です. 真理値を指定する場合, 既定値の 'false' では表面の表示を行わないために GNUPLOT ではワイヤーフレーム表示, vtk では属性 color で指定された曲面の色で表示が行われます. 属性 enhanced3d に 'true' を指定すると GNUPLOT では曲面のソリッド表示が行われ, GNUPLOT, vtk 双方で属性 palette で指定した色彩で曲面の色付けがおこなわれます. この属性 enhanced3d に 3 次元のベクトル値函数を指定すると, そのベクトル値函数に沿った色付けが対象に対して行われます.

属性 wired_surface:　属性 enhanced3d と共に用いられ, 既定値の 'false' で曲面の網目を行わず, 'true' で網目の表示を行います. なお, この属性は vtk でも利用可能です.

```
(%i10) draw(columns=2,terminal=eps_color,dimensions=[1200,600],
            gr3d(title="wired\\_surface=false",wired_surface=false,
                 enhanced3d=true,explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2)),
            gr3d(title="wired\\_surface=true",wired_surface=true,
                 enhanced3d=true,explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2)));
```

属性 contour:　GNUPLOT のみで利用可能な属性で, 等高線の描画を指定します. 既定値は 'none' で等高線表示を行いません. また, 等高線の本数は属性 contour_levels で指定します. この属性 contour が取り得る値を以下にまとめておきます:

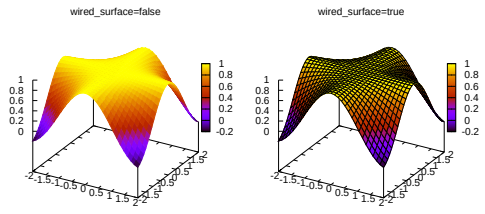


図 11.64: wired_surface

属性 contour の取り得る値

値	概要
none	等高線の非表示
base	等高線を属性 xyplane で指定した XY 平面に投影
surface	曲面に等高線を表示
both	base と surface の双方に対応し, XY 平面と曲面に等高線を表示
map	等高線を XY 平面に投影し, 属性 view を垂直方向に設定

これらの値と gnuplot の命令との対応を次に示しておきます:

属性 contour の値と gnuplot の命令文との関係

surface	set contour surface;set cntrparam levels contour_levels
base	set contour base;set cntrparam levels contour_levels
both	set contour both;set cntrparam levels contour_levels
map	set contour base;unset surface; set cntrparam levels contour_levels

属性 contour_levels: 属性 contour と対で用いる属性で, 等高線の本数, あるいは等高線を表示すべき Z 軸上の高さのリストをその値として設定します. 既定値は '5' で 5 本の等高線が描かれます. ただし, vtk では利用できません.

ここで属性 contour と contour_levels の実例を示します.

```
(%i9)  ep3d:explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2)$
(%i10) draw(columns=3, dimensions=[1500,1200],terminal=eps_color,
            gr3d(title="contour:default", enhanced3d=true,ep3d),
            gr3d(title="contour=base,xyplane=1.5",contour=base,
                 xyplane=1.5,enhanced3d=true,ep3d),
            gr3d(title="contour=surface,contour\\_levels=20",
                 contour=surface,enhanced3d=true,ep3d),
            gr3d(title="contour=both,xyplane=1.5",contour=both,
                 xyplane=1.5,enhanced3d=true,ep3d),
            gr3d(title="contour=map, contour\\_levels=20",contour=map,
                 contour_levels=20,enhanced3d=true,ep3d));
```

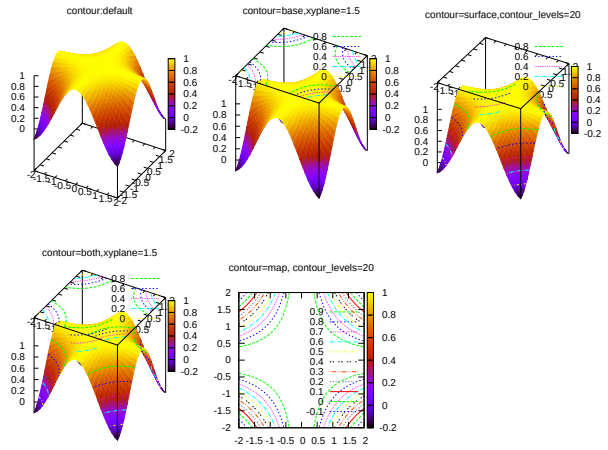


図 11.65: contour の取り得る値

この例では左上より contour の指定なし, 次が contour に ‘base’, xyplane に 1.5 を指定したために上側の XY 平面に等高線が投影されています. 右上は contour に ‘surface’, xyplane に 1.5,contour_levels に 20 を指定したもので曲面のみに等高線が 20 本描かれています. それから, 左下が contour に ‘both’, xyplane を 1.5, contour_levels を 20 としたために XY 平面と曲面の双方に 20 本の等高線が描かれています. それから最後の中央下のグラフでは contour を ‘map’, conter_level を 20 としたため, 視点は曲面を真上から眺める方向で, 等高線が 20 本描かれることとなっています.

フォントに関連する属性

フォントの属性はやや特殊で, gr 文の外に設定式を置くことができない局所的属性ですが, その設定は gr 文内に限定されることなく大域的な影響を及ぼします:

フォントに関連する属性

属性	型	既定値	概要
font	文字列	””	フォントの指定
font_size	整数値	10	フォントの大きさの指定

フォントに関連する属性としては属性 font と属性 font_size の二つがあります. これらの設定は最後の設定文が優先され, それ以前の設定文は事実上無視されることに注意して下さい.

属性 font:　ここで指定するフォントはグラフの各軸, ラベルや表題とグラフ全てで用いられるフォントになります. フォント名は Maxima の文字列として二重引用符で括ったものを与える必要があります. もしも設定値が存在しないフォント名といった不当な値であればシステムのフォントが代用されます. なお, vtk ではフォントは固定です.

属性 font_size: GNUPLOT や vtk でのグラフのフォントの大きさを指定します. 既定値は 10 ポイントとなっています. ここでの指定はグラフ全ての文字, 表題, ラベル, 軸上の目盛の数字等に影響を及ぼします.
ここで簡単な例を示しておきましょう:

```
(%i3) draw(columns=2, terminal=wxt,dimensions=[1200,600],
           gr3d(title="テスト\\_A(既定値)",
                enhanced3d=true,
                explicit(sin(x*y)/(x*y),x,-3,3,y,-3,3)),
           gr3d(title="テスト\\_B(変更)",
                font="みかちゃん", font_size=20,
                enhanced3d=true,
                explicit(sin(x*y)/(x*y),x,-3,3,y,-3,3)));
(%o3)                  [gr3d(explicit), gr3d(explicit)]
```

この例では二つのグラフを並べ, 片方のグラフは既定値のまま, もう片方は font と font_size の値を変更しています. その結果は図 11.66 です:

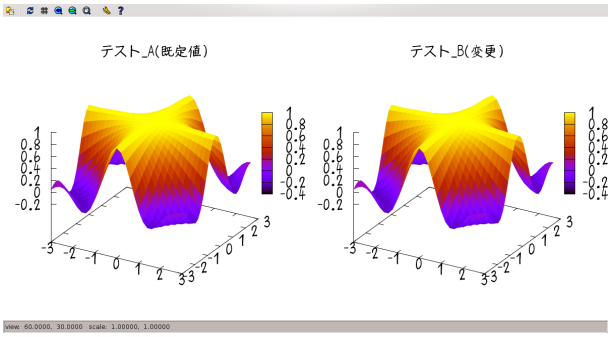


図 11.66: font, font_size の例

このように属性 font と属性 font_size はグラフ全体に影響を及ぼします. そして, draw 文の引数として与えられた gr 文の列で最後の指定が全体の設定となります.

gr 文で定義されたグラフの配置に関連する属性

GNU O T で表示する場合, 各 gr 文で定義したグラフを何処に置くかを具体的に指示することができる属性として属性 allocation があります:

gr 文で定義された対象の配置に関連する属性

属性	型	既定値	概要
allocation	リスト	false	gr 文で定義された対象の位置を指定
proportional_axes	文字列	none	各軸の比率を実寸に合せる
xy_file	文字列	""	データ保存用ファイル
transform	数式リスト	none	形状の変換を行う函数を記述 l

属性 allocation:　gr 文で定義されたグラフを全体のグラフの中で何処するかを指定する属性で, vtk では利用できません. この属性ではグラフ全体に XY 座標を入れますが, その座標で左下を原点 [0,0] とし, 右上が [1,1] となります. 属性 allocation では, 属性 allocation を含む gr 文のグラフの原点をリストの第 1 成分, グラフの右頂点をリストの第 2 成分として指定します.
たとえば, 'allocation=[[0,0,],[1,1]]' とすると allocation を包含する gr 文のグラフで表示ウィンドウが一杯になることを意味します. 既定値は 'false' で, この場合は通常の表示, つまり, '[[0,0],[1,1]]' に対応します. ここで実例を挙げておきましょう:

```
(%i106) draw(dimensions=[1200,600],
             gr2d(allocation=[[0,0],[1/4,1/2]],
                  explicit(sin(x)/x,x,-10,10)),
             gr3d(allocation=[[1/4,0],[1,1]],
                  explicit(sin(x*y)/(x*y),x,-3,3,y,-3,3)));
(%o106)                  [gr2d(explicit), gr3d(explicit)]
```

この例では 2 次元グラフと 3 次元グラフの二つのグラフを allocation を用いて調整しています. 図 11.67 に示すように 2 次元グラフは '[[0,0].[1/4,1/2]]' を指定しているために左下側に小さく表示され, 3 次元グラフは '[[1/4,/1/2],[1,1]]' と指定しているために右側の残りを全て占有しています:

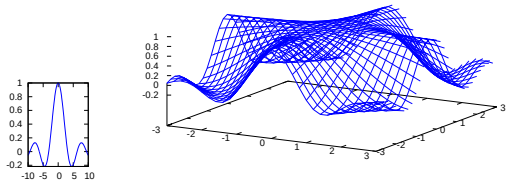


図 11.67: allocation の例

属性 proportional_axes:　gr 文で定義した対象にて指定した軸について実際の比に合せてグラフの表示を行うことを指示する属性です. 既定値は 'none' で, 設定可能な値は gr2d 文については 'xy', gr3d 文については 'xyz' です.

属性 xy_file:　2 次元グラフに対してマウスによる座標の検出を行った結果を保存するためのファイルを指定します. このマウスによる座標の検出は, グラフが表示されているウィンドウ上の任意の点をマウスの左ボタンで押した時点でウィンドウの左下に座標が表示されます. この値をキーボードから文字 "x" を押すことで属性 xy_file で指定したファイルの保存されます. 属性 xy_file で指定したファイルが既存のファイルであれば座標データがファイルの末尾に追加されます. この属性は大域的属性のように思えますが, グラフ自体に深く関わるために gr 文内でしか配置ができない局所的属性です. また, グラフも本質的に 2 次元グラフに限定されます. 実際, 3 次元グラフの描画に属性 xy_file の指定を入れてもエラーにはならず, ファイルも生成されますが, 数値データのデジタイズや格納は行われません.

では 2 次元グラフを使って解説しましょう. 次のグラフを表示し, マウスポインタをグラフの任意の場所に移動して下さい:

```
(%i4) draw(gr2d(xy_file="tama.xy",explicit(sin(x)/x,x,-10,10)));
(%o4)                                [gr2d(explicit)]
```

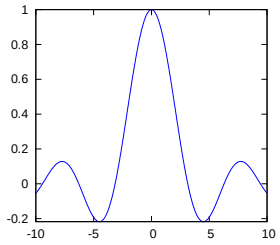


図 11.68: xy_file の例

図 11.68 に表示されたグラフを示していますが, このグラフの左下に座標が現われています. これがポインタの座標で, この値が属性 xy_file で指定したファイルに取込まれる座標です. ここで一度 x とキーを押します. それから好きな場所に移動してまた x をキーを押します. 以降, この操作を繰替えしてみましょう. この例ではファイル名しか指定していないので, ホームディレクトリ上に 'tama.xy' というファイルが生成され, その中に "x" を押した時点の座標が蓄積されています. このファイルの内容を参考のために以下に示します:

```
1 -8.96096904441454 0.0330626375514126
2 -7.56123822341857 0.116921488171606
3 -4.43876177658143 -0.221394561849557
4 -2.23149394347241 0.426431062788887
5 -0.0296096904441452 1.00576129035587
6 2.21534320323015 0.402745738567916
7 4.49259757738896 -0.218833986258101
8 7.69044414535666 0.122042639354519
```

では, 今度は次を実行してみましょう:

```
(%i5) draw(gr2d(xy_file="tama.xy",implicit(x^3+x^2-y^2,x-2,2,y,-2,2)));
(%o5)                                [gr2d(implicit)]
```

ここでは属性 xy_file に同じ名前のファイルを指定していることに注意して下さい. 前回と同様にマウスを動かして x を押してみましょう. すると属性 xy_data で指定されたファイル 'tama.xy' に, 新しいグラフの座標データが追加されていることが判ります.

ところでこの属性 xy_file は複数のグラフを表示させる場合も有効なのでしょうか？そこで次を入力して, gr 文が複数の場合はどうなるのか確認してみましょう:

```
(%i6) draw(gr2d(xy_file="tama1.xy",explicit(sin(x)/x,x,-10,10)),
           gr2d(xy_file="tama2.xy",implicit(x^3+x^2-y^2,x,-2,2,y,-2,2)));
(%o6)                    [gr2d(explicit), gr2d(implicit)]
```

この例では各 gr 文の中で別々のファイルを指定していますが, どのファイルも生成されません. 属性 xy_file が意味を持つのは gr 文が一つの場合のときのみ, gr2d 文であるときに限定されます.

属性 transform:　函数と変数リストを割当てて gr 文内部でそのリストに従って対象の変換を行います. 2 次元と 3 次元のグラフで利用可能です.

属性 transform の書式

次元	リストの書式
2 次元	$[f_x(X,Y), f_y(X,Y), X, Y]$
3 次元	$[f_x(X,Y,Z), f_y(X,Y,Z), f_z(X,Y,Z), X, Y, Z]$

2 次元の場合, f_x, f_y は 2 変数の実数函数, 3 次元の場合, f_x, f_y, f_z は 3 変数の実数函数で, 各リストの函数のうしろに続く X, Y, Z はそれぞれ X,Y,Z 成分を表現し, 函数 f_x, f_y, f_z の定義で用いる変数です. この属性 transform で与えた函数で対象の変換が行われます. 実際に確認してみましょう:

```
tr1:triangle([0,0],[0,1],[1,1])$
draw(gr2d(title="black:Original, red:Moved, green:Shrinked",
          color=black, fill_color=black,tr1,
          transform=[1/2*x1-1,2*y1+3,x1,y1],
          fill_color=red, line_width=5,tr1,
          transform=[1/2,2*y1+3,x1,y1],
          color=green, line_width=5,tr1))$
```

この例では黒い三角形を $(x, y) \to (1/2x - 1, 2y + 3)$ で赤に, $(x, y) \to (1/2, 2y + 3)$ で緑に変換するもので, 図 11.69 に示すように緑は横方向に潰れていますが構いません:

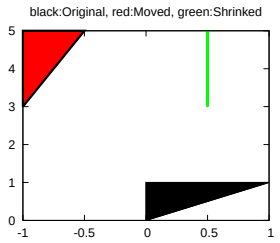


図 11.69: 属性 transform による対象の平行移動

グラフの枠と座標軸に関連する属性

ここで解説する属性はグラフの枠, 各座標軸と目盛の表示や色の設定に関係する属性です. これらの属性は 2 次元, 3 次元共に共用です. ここで解説する属性は各 gr 文が定めるグラフの枠や座標軸を制御する属性であるために gr 文の外に置くことができない局所的な属性です.

まず, グラフの枠に関連する属性で大きさと表示方法に関連する属性を次に示しておきます.

グラフの枠に関連する属性

属性	型	既定値	概要
axis_top	論理値	true	グラフの上側に軸を配置
axis_bottom	論理値	true	グラフの底に軸を配置
axis_right	論理値	true	グラフの右に軸を配置
axis_left	論理値	true	グラフの左に軸を配置
axis_3d	論理値	true	3 次元グラフに枠を配置

属性 axis_top, 属性 axis_bottom, 属性 axis_right と属性 axis_left: GNUPLOT による二次元グラフ表示で上下, 左右の枠の表示を行うかどうかを指定する属性です. たとえば,axis_top が 'false' の場合, グラフの上下, 左右を囲む枠の上側が表示されません. 同様に axis_bottom が 'false' であれば下側, axis_right が 'false' なら右側 ,axis_left が 'false' なら左側の枠が表示されません. 実例として上と右を表示させないようにしてみましょう:

```
(%i114) draw(gr2d(explicit(sin(x)/x,x,-10,10),
                  axis_top=false,axis_right=false));
(%o114)                          [gr2d(explicit)]
```

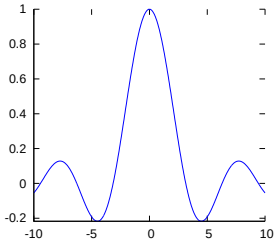


図 11.70: Top と Right の枠を外した例

ここでの例では属性 axis_top と属性 axis_right を 'false' に変更したために図 11.70 に示す図では右の枠と上の枠が消えていることに注意して下さい. ただし, これらの属性は枠に関する属性のために上と右の目盛がそのまま残っていることに注意して下さい. なお, これらの属性は GNUPLOT のみで利用可能であって, vtk では利用できません.

属性 axis_3d: 3 次元グラフでのみ意味のある属性です. この属性は GNUPLOT だけではなく vtk でも利用可能です. この属性に 'false' が割当てられていると 3 次元グラフの全ての枠が非表示になります. この属性の指定による影響は GNUPLOT と vtk で幾らか異なっています. GNUPLOT の場合, この属性は枠に関する属性であって, 目盛の属性とは違うために, ここを 'false' にしていても目盛や数値はそのまま残ります. この様子を図 11.71 に示しておきます:

```
(%i118) draw(gr3d(axis_3d=false,explicit(sin(x*y)/(x*y),x,-3,3,y,-3,3)));
```

```
(%o118)                    [gr3d(explicit)]
```

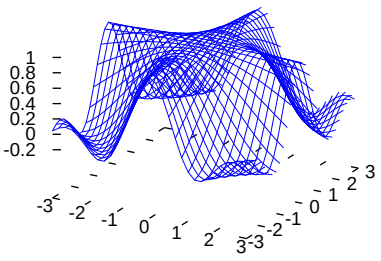


図 11.71: 3 次元表示で枠を外した例

ところが, vtk では枠に表示する目盛や数値は GNUPLOT と違って, 操作できる属性がないために, この属性を 'false' と指定することで枠全体と枠の目盛や数値も図 11.72 に示すように消えてしまいます:

```
(%i119) draw_renderer:vtk$
(%i120) draw(gr3d(axis_3d=false,explicit(sin(x*y)/(x*y),x,-3,3,y,-3,3)));
(%o120)                    [gr3d(explicit)]
```

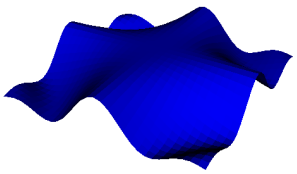


図 11.72: VTK で 3 次元表示で枠を外した例

ここまでグラフの枠の表示に関する属性を説明しました. 次に各軸の表示に関連する属性を解説しましょう. なお, ここで解説する属性は全て GNUPLOT のみで利用可能で, vtk では利用できません:

座標軸の表示に関連する属性

属性	型	既定値	概要
xaxis	論理値	false	軸表示を指定
yaxis	論理値	false	軸表示を指定
zaxis	論理値	false	軸表示を指定
xaxis_secondary	論理値	false	第 2 の目盛を設定
yaxis_secondary	論理値	false	第 2 の目盛を設定
xaxis_width	実数	1	軸の幅を設定
yaxis_width	実数	1	軸の幅を設定
zaxis_width	実数	1	軸の幅を設定
xaxis_type	文字	dots	軸の形を設定
yaxis_type	文字	dots	軸の形を設定
zaxis_type	文字	dots	軸の形を設定
xaxis_color	文字列	"#000000"	軸の色を設定
yaxis_color	文字列	"#000000"	軸の色を設定
zaxis_color	文字列	"#000000"	軸の色を設定

ここで解説する属性は全て GNUPLOT のみで利用可能であり, vtk では利用できません.

属性 xaxis, 属性 yaxis と属性 zaxis: 各軸の表示を指定する属性です. 既定値は全て 'false' のために軸は表示されません. 'true' に変更すると, 該当する軸のみが表示され, その際の軸の太さは軸に対応する属性の *axis_width, 形状は *axis_type, その色は *axis_color で指定されます.

属性 xaxis_secondary と属性 yaxis_secondary: 2 次元グラフにおいて意味のある属性で, 第 2 の目盛を入れるかどうかを指定する属性で, 属性 xtics_secondary, ytics_secondary と組合せて利用します. 'true' であれば対応する軸に二番目の目盛を入れます.

属性 xaxis_width, 属性 yaxis_width と属性 zaxis_width: 軸の表示属性 xaxis, yaxis あるいは zaxis が 'true' の場合に対応する軸の太さを指定します. 軸の太さは実数で指定可能です.

属性 xaxis_type, 属性 yaxis_type と属性 zaxis_type: 軸の表示属性 xaxis, yaxis あるいは zaxis が 'true' の場合に対応する軸の表示状況を定めます. 指定可能な値は 'dots' か 'solid' のいずれかで, 'dots' の場合は点線で描かれ, 'solid' の場合に実線で描かれます. 既定値は 'dots' の点線です.

属性 xaxis_color, 属性 yaxis_color と属性 zaxis_color: 軸の表示属性 xaxis, yaxis あるいは zaxis が 'true' の場合に対応する軸の色を指定します. 色は 6 桁の 16 進数に先頭に 16 進数であることを示すために記号 "#" を追加したものか, §11.7.9 に示す色名の何れかになります. 既定値は Maxima の文字列で "#000000", 色名で 'black' になります.

ここで実例を示しておきましょう:

```
(%i17) draw(columns=2,
            gr2d(xaxis=true,xaxis_type=dots,
                 explicit(sin(x)/x,x,-5,5)),
            gr2d(xaxis=true,xaxis_type=solid,
                 explicit(sin(x)/x,x,-5,5)));
(%o17)                  [gr2d(explicit), gr2d(explicit)]
```

この例では各 gr2d 文で xaxis を ‘true’, 第 1 の gr2d 文で xaxis_type を ‘dots’, 第 2 の gr2d 文で xaxis_type を ‘solid’ にしています. この結果は図 11.73 に示しています:

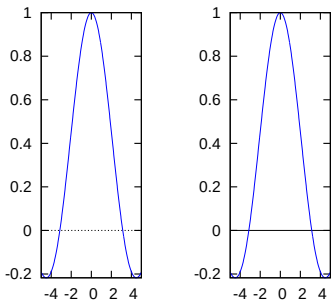


図 11.73: xaxis を ‘true’, xaxis_type を変更した例

この例では枠を外していないので, 単純に X 軸が追加されただけです. 左側のグラフが最初の gr2d 文に対応し, xaxis_type が ‘dots’ のために点線, 右側のグラフが第 2 の gr2d 文に対応し, その中で xaxis_type が ‘solid’ のために実線で軸が表示されています.
グラフの目盛に関連する属性もあります. これらの属性では目盛の有無や目盛を対数表示にするかどうか, そして, グラフ全体に網目を入れるかが指定できます. これらの属性は GNUPLOT のみで利用可能で, vtk では利用できません.

グラフの軸の目盛に関する属性

属性	型	既定値	概要
grid	論理値	false	網目の描画
xtics,ytics,ztics	リスト等	auto	目盛を設定
xtics_seconday,ytics_secondary	リスト等	auto	二番目の軸に目盛を設定
xtics_roate,ytics_rotate,ztics_rotate	論理値	false	目盛を 90 度回転
xtics_axis,ytics_axis,ztics_axis	論理値	false	軸に沿って目盛をするかどうかを指定
logx,logy,logz	論理値	false	対数目盛を設定

属性 grid: 2 次元グラフの網目表示を指定するための属性です. grid に 'true' を指定すると二次元グラフで目盛に対応するように網目の表示が行われます. また, 3 次元グラフの場合は属性 xyplane が既定値のままであれば底面となる XY 平面上, 属性 xyplane に値が設定されていれば, その属性 xyplane で定まる XY 平面上に網目が表示されます:

```
(%i17) draw(columns=2,dimensions=[1200,600],
            gr2d(grid=true,title="2-d, grid=true",
                 explicit(sin(x)/x,x,-5,5)),
            gr3d(grid=true,title="3-d, grid=true, xyplane=1.5",
                 enhanced3d=true,xyplane=1.5,
                 explicit(sin(x*y)/(x*y),x,-3,3,y,-3,3)),terminal=eps_color);
(%o17)                    [gr2d(explicit), gr2d(explicit)]
```

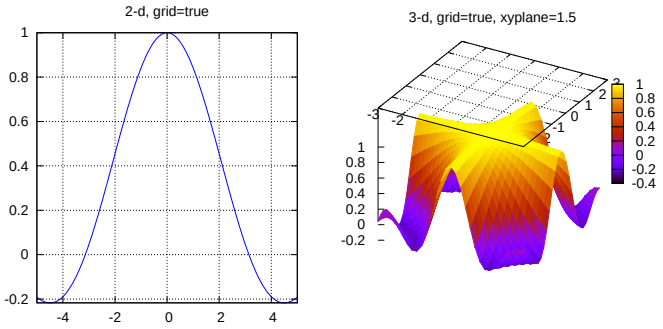


図 11.74: grid を true にした場合の 2 次元と 3 次元のグラフ

属性 xtics, 属性 ytics と属性 ztics: グラフの枠に目盛を入れるかどうかを指定する属性で, 第 2 番目の軸の目盛に関連する属性 xtics_secondary と属性 ytics_secondary と同じ働きをします. これらの属性の値が 'auto' であれば自動的に対応する軸に目盛を入れ, 'none' であれば目盛を入れません. この属性に名称と数値の対のリストを元とする集合を生成すると, 対応する数値に目盛を配置して数値に表示を行います. 各軸の目盛の属性は axis_top 等の枠の表示に関連する属性から独立しています. すなわち, 枠を非表示にしていても軸目盛の表示が 'none' 以外であれば, その軸の目盛はちゃんと表示されます.

属性 xtics_secondary, 属性 ytics_secondary: 2 次元グラフでの属性 xaxis_secondary と属性 yaxis_secondary に対応する属性で, 属性 xtics, 属性 ytics と属性 ztics と同等の働きを第 2 軸に対して行います.

```
(%i24) draw(gr2d(xaxis_secondary=true,
                 xtics_secondary={["A",-4],["B",0],["C",4]},
                 explicit(sin(x)/x,x,-5,5)));
(%o24)                          [gr2d(explicit)]
```

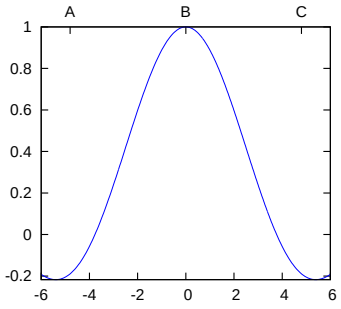


図 11.75: xtics_secondary の例

属性 xtics_rotate, 属性 ytics_rotate と属性 ztics_rotate:　対応する軸上の目盛の数値を 90 度回転させるかどうか指定します. 既定値は 'false' でこの場合は 90 度回転させません. なお, この属性は第 2 の軸には影響しません:

```
(%i25) draw(gr2d(xaxis_secondary=true,
                 xtics_secondary=[["A",-4],["B",0],["C",4]],
                 xtics_rotate=true,
                 explicit(sin(x)/x,x,-5,5)));
(%o25)                          [gr2d(explicit)]
```

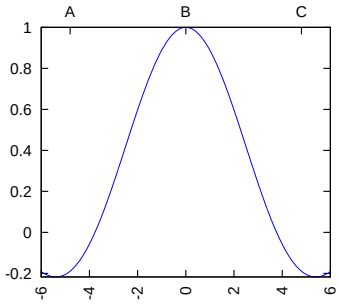


図 11.76: xtics_rotate の例

この例からも判るように第 2 軸の"A","B","C"は回転していません.

属性 xtics_axis, 属性 ytics_axis と属性 ztics_axis:　対応する軸に沿って目盛を表示させるか, 枠に沿って表示させるかを指定します. 既定値は 'false' で枠に沿って目盛が表示されます.

```
(%i26) draw(gr2d(xaxis=true,yaxis=true,
                 xaxis_type=solid,
                 xtics_axis=true,
                 axis_top=false,axis_bottom=false,
                 axis_right=false,xaxis_secondary=true,
                 yaxis_secondary=true,
```

```
                xtics_secondary=[["A",-4],["B",0],["C",4]],
                ytics_secondary=[["top",1],["base",-0.2]],
                explicit(sin(x)/x,x,-5,5)),terminal=eps);
(%o26)                          [gr2d(explicit)]
```

この例では, 属性 xaxis と yaxis を 'true*にすることで表示させ, このときに属性 xaxis_type を 'solid' にすることで実線としています. それからグラフの上下, 右の枠を属性 xaxis_top, xaxis_bottom と xaxis_right を 'false' にすることで外し, xtics_secondary と ytics_secondary に目盛のラベルを指定することで図 11.77 が得られています:

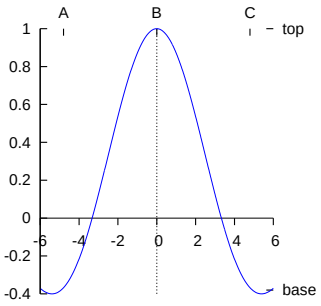


図 11.77: xtics_axis の例

属性 logx, 属性 logy と属性 logz: グラフ枠の目盛の間隔を対数目盛にするかどうか指定する属性で, 対数目盛にする場合には, 必要な軸の属性を 'true' にします.

表題とラベルに関連する属性

表題や軸のラベルは GNUPLOT のみ利用可能です. ここで指定する値には Maxima の文字列を指定します. ここで GNUPLOT では拡張文字モードで処理が行われるため, 拡張文字列制御記号に対しては, その記号に応じた表示が行われます:

GNUPLOT の拡張文字列制御記号

記号	入力	イメージ	解説
^	a^i	a^{i}	上付き添字
_	a_i	a_{i}	下付き添字
@	a@^b_{cd}	a^{a}_{cd}	幅のない文字を挿入. a^b_{cd} なら $a^{b}{}_{cd}$
&	s&{pac}e	s␣␣␣e	&{..} 内部の文字数文の空白を指定
~	~a{.8-}	ā	現時点のフォントの高さの.8 倍の高さ分, 後続の文字を上に上げる

制御記号にはこのような働きがあるため, この機能を用いてより的確なラベル表示が行えます. 逆にファイル名にこれらの記号が含まれていて単なる文字として表示したければ, 記号 "\\" を該当する記号の前に置きます. たとえば, Maxima の文字列"xu_grid" をそのまま表示したければ, 文字

列”xu_grid”に変換しておく必要があります.

グラフの表題や各軸のラベルに関連する属性を次に示します:

表題, ラベルや凡例に関連する属性

属性	型	既定値	概要
title	文字列	””	表題を指定
xlabel,ylabel,zlabel	文字列	””	軸のラベルを指定
key	文字列		凡例 (legend) を記述

ここでの属性は論理値ではなく, 実際に表示する文字列になります. 属性 title にグラフの表題を設定します. これは単純に gnuplot の set title 文に title の内容を引渡すだけです.

属性 title:　グラフの表題を指定します. この表題は Maxima の文字列として与えます. 二重引用符で括られていない単なる文字の場合, Maxima 側で変数名と見做すために自由変数であれば先頭に記号 “$” を付けた変数名が表題となり, 束縛変数の名前であれば, その値が表題として用いられます.

属性 xlabel, 属性 ylabel と属性 zlabel:　X 軸, Y 軸と Z 軸に設定するラベルを指定する属性です. 指定すべき値は二重引用符で文字の列を括った Maxima の文字列で, 二重引用符なしの文字の列を引き渡した場合は Maxima の変数と見做されます. この変数名が自由変数であれば先頭に記号 “$” が付けられたものがラベルとなり, 束縛変数であればその値がラベルとなります. 実質的には, GNUPLOT の ‘set xlabel’ 文等に属性を引渡すだけです.

属性 key:　GNUPLOT の凡例 (legend) を設定するための属性です. つまり, GNUPLUT グラフで右上側にどのグラフが何のグラフに対応するかを説明する箱が描かれていることがありますが, この箱の内容を設定する属性です. 参考のため, 次の処理結果を図 11.78 に示しておきます:

```
(%i264) draw2d(title="2 curves",
               xlabel="Time [sec]",
               ylabel="Height[m]",
               line_type=solid,
               key="sin(x)/x",
               explicit(sin(x)/x,x,-10,10),
               key="x^2-1",
               line_type=dots,
               explicit(x^2-1,x,-3,3));
(%o264)                  [gr2d(explicit, explicit)]
```

図 11.78 に示すように属性 title による表題と各軸ラベルの設定を行い, 図右上に凡例を表示させています. この凡例によって, どれが $\frac{\sin x}{x}$ のグラフで, どれが x^2-1 のグラフであるかが明瞭になっています.

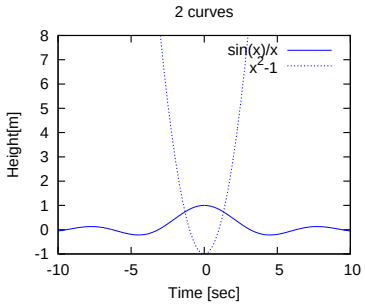


図 11.78: key による凡例

線の属性

対象の線の太さや種類を指定する属性があります. これらの属性は GNUPLOT と vtk の双方で利用することが可能です.

線の属性

属性	型	既定値	概要
line_width	実数	1	線の幅を設定
line_type	文字	solid	軸の形を設定

属性 line_width: 描画する対象に含まれる線の太さを定めます. 既定値は '1' です. 線の種類は属性 line_type で定めます. この属性は vtk でも指定可能です.

属性 line_type: 描画する対象に含まれる線の種類を定めます. 既定値は 'solid' でこの場合は実線が描かれます. 他に 'dots' のみが指定可能です. この属性も vtk で指定可能です.

```
(%i27) draw(columns=2,dimensions=[1200,600],
            gr2d(title="line\\_type:solid",
                 explicit(sin(x)/x,x,-2,2)),
            gr2d(title="line\\_type:dots, line\\_width=5",
                 line_type=dots,line_width=5,
                 explicit(sin(x)/x,x,-2,2)),terminal=eps);
(%o27)                  [gr3d(explicit), gr3d(explicit)]
```

表示領域を指定する属性

表示領域を指定する属性は GNUPLOT のみで利用できます.

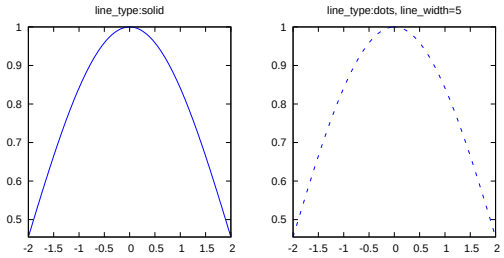


図 11.79: line_width と line_type の例

表示領域を指定する属性

属性	型	既定値	概要
xrange	二成分の実数リスト	自動	X 軸の表示領域を設定
yrange	二成分の実数リスト	自動	Y 軸の表示領域を設定
zrange	二成分の実数リスト	自動	Z 軸の表示領域を設定

属性 xrange, 属性 yrange と属性 zrage:　GNUPLOT で表示する領域, すなわち, 枠の大きさを指定する属性です. これらを無指定にしておくと, 函数側で自動的にその表示対象の大きさから指定を行います. たとえば, 対象 explict や対象 explicit3d の場合, これらを定義する explicit から変数の定義域や値域を自動的に枠の領域として設定します.

３次元グラフの視点に関連する属性

３次元グラフの視点に関連する属性を示しておきます:

視点の変更, 検出した座標値の保存に関連する属性

属性	型	既定値	概要
view	2 成分の数値リスト	[60,30]	視点位置を指定
xyplane	数値		XY 平面の描画
rot_vertical	0 から 180 の間の数	60	Z 軸回りの回転角度 (非推奨)
rot_horizontal	0 から 360 の間の数	30	水平の回転角度 (非推奨)

属性 view:　X 軸と Z 軸の回転で眺める方向を定めます. 既定値は '[60,30]' で最初に X 軸回りに 60 度, 最後に Z 軸回りに 30 度回した方向で視点を定めます. この属性は vtk でも利用可能です.

属性 xyplane:　XY 平面描画の有無と位置の指定を行います. 既定値は 'false' で３次元グラフ表示で XY 平面の描画を行いませんが, この属性 xyplane に XY 平面の Z 座標を入力すると, その座標を通る XY 平面を描きます. なお, この属性は vtk では使えません.

属性 rot_vertical と属性 rot_horizontal: これらの属性は GNUPLOT の 'set view' 文に対応し, 各軸回りの回転を指示しますが, vtk では vertical, horizontak 双方とも使えません. 現在では非推奨で vtk との運用も考慮して属性 view を用いて下さい. ちなみに属性 rot_vertical は 0 から 180 の間の実数で Z 軸回りの回転, 属性 rot_horizontal は 0 から 360 の間の実数で X 軸回りの回転が指定できます.

ここで属性 view と属性 xyplane を用いた例を示しましょう:

```
(%i27) draw(columns=2,dimensions=[1200,600],
            gr3d(title="view=[60,30]",palette=[30,31,32],
                 enhanced3d=true, xyplane=0,
                 explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2)),
            gr3d(title="view=[30,60]",palette=[30,31,32],
                 enhanced3d=true,xyplane=1,
                 explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2),
                 view=[30,60]));
(%o27)                  [gr3d(explicit), gr3d(explicit)]
```

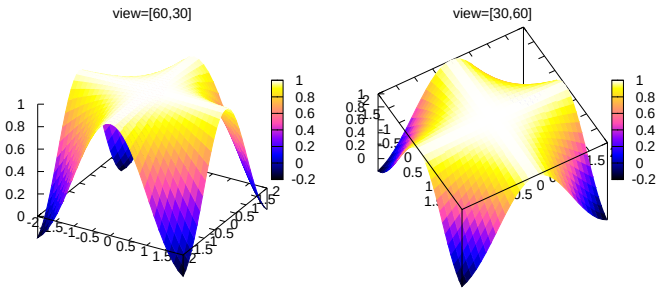


図 11.80: title と view の例

図 11.80 では表題と視点を属性 title と view を用いて変更したものを並べています. 左側のグラフでは平面を 'Z=0' の位置に配置し, 右側のグラフでは平面を 'Z=1' の位置に配置しています.

曲線や曲面近似に関連する属性

曲線や曲面近似に関連する属性を挙げておきます:

曲線・曲面近似に関連する属性

属性	型	既定値	概要
nticks	整数値	29	
x_voxel	整数値	10	X 軸方向の正規格子数
y_voxel	整数値	10	Y 軸方向の正規格子数
z_voxel	整数値	10	Z 軸方向の正規格子数
xu_grid	正整数値	30	X 軸方向の点数
yv_grid	正整数値	30	Z 軸方向の点数

属性 nticks:　曲線の近似の度合いを指定する属性です. この属性は陽的にその座標が定められる対象 ellipse, explicit, parametric, parametric3d に影響を与えます. 既定値は '29' で, X 軸や助変数の定義域を nticks で指定した数で均等に分割します. したがって, この数値を大きくするとそれだけ図形の近似が良好, 具体的には滑かになります.

属性 x_voxel, y_voxel と z_voxel:　対象 implicit, implicit3d と region の属性で, 各 X, Y, Z 軸方向の正規格子の個数を指定します. 既定値は 10 で, この値が大きければ大きい程, 形状の精度が向上します. しかし, 正規格子で近似する性格上, 三角形領域のような単純な図形の頂点付近で領域の抜け落ちが顕著に現われます：

```
(%i21) draw(columns=2,dimensions=[1200,600],
            [gr2d(title="VOXEL=10x10",
                  region(x>0and x<1 and y>0 and y-x<0,
                         x,-0.1,1.1,y,-0.1,1.1)),
             gr2d(title="VOXEL=50x50",
                  x_voxel=50,y_voxel=50,fill_color=green,
                  region(x>0and x<1 and y>0 and y-x<0,
                  x,-0.1,1.1,y,-0.1,1.1))]);
```

この例は対象 region における正規格子数を既定値の 10 と 50 で比較することを目的としています. そのために左側のグラフの X, Y の正規格子点数を既定値の 10x10 に対して右側のグラフの正規格子点数を 50x50 と 25 倍の点数としています. その結果を図 11.81 に示します:

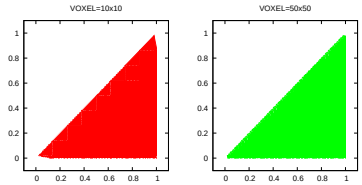


図 11.81: voxel の例

図から判るように格子点数が大きなもののほうがより良好な結果を得ています. ところが, よくよく見ると三角形の頂点付近は良好な筈の 50x50 でもまだ不十分です. そこで同じ格子数で表示領域を変えたもので比較してみましょう：

```
(%i21) draw(columns=2,dimensions=[1200,600],
            [gr2d(title="VOXEL=50x50, AREA=20x20",
                  x_voxel=50,y_voxel=50,
                  region(x>0and x<1 and y>0 and y-x<0,
                         x,-10,10,y,-10,10)),
             gr2d(title="VOXEL=50x50, AREA=1x1",
                  x_voxel=50,y_voxel=50,fill_color=green,
                  region(x>0and x<1 and y>0 and y-x<0,
                  x,-0.1,1.1,y,-0.1,1.1))]);
```

今度は正規格子点数は同じ 50x50 ですが, 表示領域を左側は 20x20, 右側を 1x1 と面積比では 400:1 となっています. この結果を図 11.82 に示します:

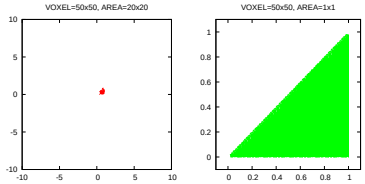


図 11.82: voxel の例 (その 2)

格子点数が同じでも, 左側は対象が相対的に小さくなるためにその分粗くなっていることが判ります. このように不要な領域が広くなればなる程, 描くべき対象が粗くなります. したがって, 対象 implicit, implicit3d や region の描画では, 複雑な図形や鋭角を持つ図形であれば特に格子点数を変更しても描画に大きな差が見られないことを確認しておく必要があります.

属性 xu_grid と属性 yv_grid: 属性 x_voxel や属性 y_voxel に対応する陽的な表示を持つ対象の対象 explicit3d, cylinder, parametric_surface, sphere と tube にて対象を構成する曲面の近似の度合いを指示する属性です. 具体的には, 属性 xu_grid が曲面の X 軸方向の分割数, 属性 yv_grid が曲面の Y 軸方向の分割数に対応します. したがって, より大きな数値を指定することでより細密な形状が得られることになります. なお, これらの属性の既定値は '30' です.

ここで既定値と比較する簡単な例を示しておきましょう:

```
(%i27) draw(columns=2,dimensions=[1200,600],
            gr3d(title="xu\\_grid, yv\\_grid=30x30",palette=[30,31,32],
                 enhanced3d=true,
                 explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2)),
            gr3d(title="xu\\_grid, yv\\_grid=60x60",palette=[30,31,32],
                 xu_grid=60,yv_grid=60,
                 enhanced3d=true,
                 explicit(sin(x*y)/(x*y),x,-2,2,y,-2,2)));
(%o27)                  [gr3d(explicit), gr3d(explicit)]
```

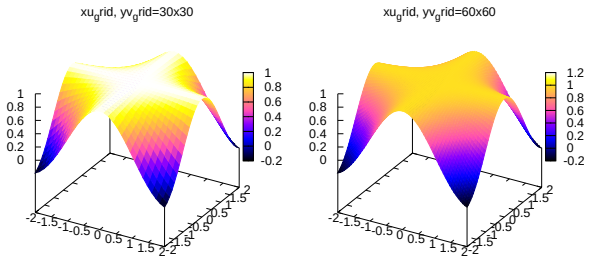


図 11.83: xu_grid と yv_grid を増やした場合の相違

この例では左が既定値の属性 xu_gird,yv_grid が 30 のものです. それに対して右側は属性 xu_gird,yv_grid を共に倍の 60 に変更しています. この分割数の増大によって描画に要する時間は増加しますが, それだけ細かな曲面となります. この結果は図 11.83 に示しておきますが, 明らかに右側のグラフがより滑かな曲面となっていることが判ります.

多角形に関連する属性

２次元グラフの対象で点列から構成された多角形の内部の塗り潰しに関連する属性があります. ここで多角形に関連する対象に polygon, rectangle と ellipse があります. ここで解説する属性は２次元対象の属性であるために vtk では使えません.

多角形の属性

属性	型	既定値	概要
border	真理値	'true'	境界線の有無を指定
fill_color	文字列	red	塗り潰しの色を指定
fill_density	実数値	1	透明度を指定
transparent	論理値	false	塗り潰しを行うかどうかを指定

属性 border:　対象 polygon, rectangle や ellipse にて対象の境界を描くかどうかを指定します. 丁度, 対象 explicit3d や対象 implicit3d の属性 wired_surface に対応します. この境界線の色は属性

color で行い, 後述の属性 fill_color と別の色が指定できます. また, 線の太さと種類はそれぞれ属性 line_width と属性 line_type で行います.

属性 fill_color: 曲線と X 軸で挟まれた領域を塗り潰す色の属性です. 既定値として red(赤) が指定されています.

属性 fill_density: 塗り潰しの透明度を指定する属性で閉区間 [0,1] の範囲の値を取ります. '0' であれば透明, '1' であれば不透明となります.

属性 transparent: 多角形内部の塗り潰しを行うかどうかを指定する属性です. 既定値は 'false' なので属性 fill_color に設定された色で塗り潰しが行われます. 'true' の場合は透明となるので, 塗り潰しは行われず, 属性 fill_density を '0' に設定したのと同じ効果が得られます.

```
(%i17) draw(columns=2,terminal=eps,
            gr2d(title="border=false, fill\\_color=green",
                 border=false, fill_color=green,
                 xrange=[-0.5,1.5],yrange=[-0.5,1.5],
                 polygon([[0,0],[1,0],[1,1]])),
            gr2d(title="border=true, transparent=true",
                 border=true,line_width=5,transparent=true,
                 color=green,xrange=[-0.5,1.5],yrange=[-0.5,1.5],
                 polygon([[0,0],[1,0],[1,1]])));
```

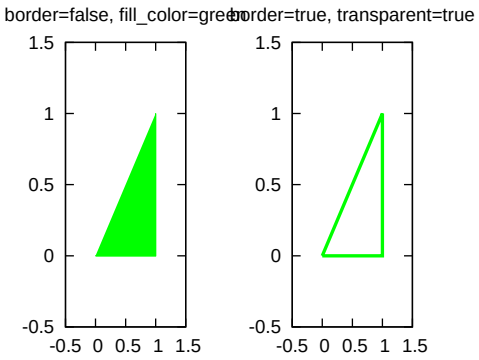


図 11.84: border と transparency の効果

ここでの例では左が属性 border を 'false' にすることで境界線を見えなくし, 属性 fill_color を 'green' にして緑で塗り潰しを行っています. 右は属性 border を既定値の 'true', 属性 color を 'green' , 属性 line_width を '5' に変更することで境界線を通常よりも太く, 既定値の青から緑に変更しています. それから属性 transparent を 'true' にすることで多角形内部を透明にしています. このように多角形では境界線は属性 line_width, line_type で線の太さや種類を定め, 色は属性 color で指定します.

点列に関連する属性

点列に関する属性としては, 点の大きさ, 点の種類と点列同士を繋げるといったものがあります. 点列に関連する対象としては, 対象 points と points3d, それと対象 errors があります.

点列に関連する属性

属性	型	既定値	概要
point_size	正整数値	1	描画時の点の大きさを指定
point_type	文字列	plus(=1)	描画時の点の型を指定
points_joined			

point_size:　点の大きさを定める属性値です. ここでの既定値は '1' です. この属性は GNUPLOT, vtk の双方で利用可能です.

point_type:　点の形状を定める属性です. 値は数値あるいは文字の列 (二重引用符で括らないもの) となります. ここで指定可能な値を以下に纏めておきます:

対象 points の形

形		points_type の値	
記号	none(=ー 1)	dot(=0)	plus(=1)
	multiply(=2)	asterisk(=3)	
円	circle(=6)	filled_circle(=7)	
四角形	square(=4)	filled_square(=5)	
三角形	up_triangle(=8)	down_triangle(=10)	filled_up_triangle(=9)
	filled_down_triangle(=11)		
菱形	diamant(=12)	filled_diamant(=13)	
３次元	sphere	cube	cylinder
	cone		

GNUPLOT を描画アプリケーションとして利用する場合, point_type は上の表の文字列, あるいは括弧内の記号 "=" の右側の数値でもどちらを指定しても構いません. たとえば円を描かせたければ 'point_type=circle' とするか 'point_type=6' とします. また point_type が 'none', 数値で '-1' のときは点の描画は行われません. 数値で指定するときに 14 以上の数値 n を指定した場合は $n \bmod 14 + 1$ で記号が割当てれらます. そして, ３次元の 'sphere', 'cube', 'cylinder' や 'cone' についてはそれぞれ 'plus', 'multiply', 'asterisk', 'square' で代用されます. これら３次元形状の表示は vtk のみで行えますが, 逆に vtk では 3 次元以外の記号の表示はできずにエラーになります.

points_joined:　点列を互いに線分で結ぶかどうかを指定する属性です. GNUPLOT, vtk 共に利用可能です. 点列が $P_1, \ldots, P_n$ の場合, 点 $P_i, i \in [1, n]$ に対して線分 P_iP_{i+1} がひかれます. この線分の

太さや種類は属性 line_width と属性 line_type で指定できます.

ここで簡単な例を示します. まず, GNUPLOT による例で, 点の大きさと形を選び, 各点を結びます:

```
draw(columns=2,dimensions=[1200,600],
     gr2d(title="point\\_type=circle, points\\_joined=false",
          point_size=5, point_type=circle,
          points([[0,0],[0,1],[1,1]])),
     gr2d(title="point\\_type=diamant, points\\_joined=ture",
          xrange=[-0.5,1.5],yrange=[-0.5,1.5],
          point_size=10, point_type=diamant,
          points_joined=true, color=red,
          line_width=5,line_type=dots,
          points([[0,0],[0,1],[1,1]])),terminal=eps);
```

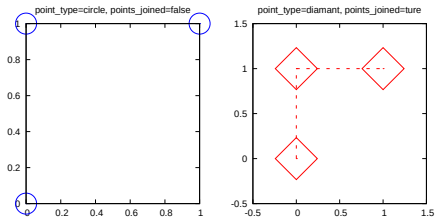


図 11.85: 点列に関連する属性の例 (GNUPLOT)

この図では左側で点を circle とし, 属性 points_joined は既定値の false のままです. 表示領域を指定しなければ点列は枠の端に配置されて表示が行われます. これに対して右側では同じ点列に対して表示領域を大き目に指定し, 点は属性 color を用いて赤を割当て, 結ぶ線分を点線に指定しています.
次は vtk による例です. 上の GNUPLOT と似た例ですが, 属性 point_type を 3 次元の sphere と cone にしたものです:

```
draw_renderer:vtk$
draw(gr3d(point_size=2, point_type=sphere,
          points_joined=true, color=red,
          line_width=5,line_type=solid,
          points([[0,0,0],[0,10,0],[10,10,10]]),
          point_size=4, point_type=cone,
          points_joined=true, color=green,
          line_width=5,line_type=dots,
          points([[10,0,0],[10,0,10],[0,0,10]])),terminal=eps)$
```

この例では赤が点を sphere にして実線で結んだもの, 緑が点を cone にして点線で結んだものです. 3 次元とはいえ線分が三次元的に表示される訳ではありません. また, GNUPLOT による 3 次元グラフの表示で点に配置した記号は視点を変更しても一緒に観点することもなく変化はありませんが, vtk の場合は視点の変化に伴い点に配置した形状も回転します.

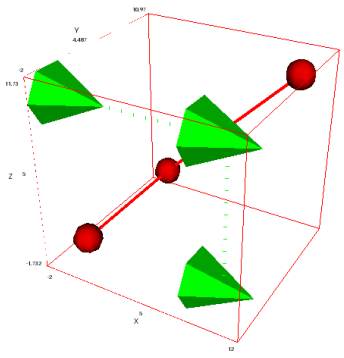


図 11.86: 点列に関連する属性の例 (vtk にて)

11.7.10　対象について

ここでは対象を定義する文ごとに纏めて対象の概要とその対象が有する属性について解説します.

points 文

points 文は平面上の点を表現する対象 points と空間内の点をを表現する対象 points3d を定義するために用います.

points 文の構文

対象	構文
points	points([[⟨X$_1$⟩,⟨Y$_1$⟩],...,[⟨X$_n$⟩,⟨Y$_n$⟩]])
points	points([⟨X$_1$⟩,...,⟨X$_n$⟩],[⟨Y$_1$⟩,...,⟨Y$_n$⟩])
points	points(⟨1 次元配列$_1$⟩,⟨1 次元配列$_2$⟩)
points	points(⟨2 次元配列 ⟩)
points	points(⟨ 行列 ⟩)
points3d	points([[⟨X$_1$⟩,⟨Y$_1$⟩,⟨Z$_1$⟩],...,[⟨X$_n$⟩,⟨Y$_n$⟩,⟨Z$_1$⟩]])
points3d	points([⟨X$_1$⟩,...,⟨X$_n$⟩],[⟨Y$_1$⟩,...,⟨Y$_n$⟩],[⟨Z$_1$⟩,...,⟨Z$_n$⟩])
points	points(⟨1 次元配列$_1$⟩, ⟨1 次元配列$_2$⟩, ⟨1 次元配列$_3$⟩)
points	points(⟨2 次元配列$_1$⟩)
points	points(⟨ 行列 ⟩)

対象 points の定義ではリスト, 配列と行列が利用できます. リストの書式は $[[x_1,y_1],\ldots,[x_n,y_n]]$ と点毎に定義するか, $[[x_1,\ldots,x_n],[y_1,\ldots,y_n]]$ と点列の X 座標と Y 座標のリストで表記する二通りの表記ができます. 配列で定義する場合は, 成分が 2 成分のリストである一つの配列, 二つの 1 次元配列, 一つの 2 次元配列の三種類が選べます. また行列で定義する場合, その行列は $1\times n$, $n\times 1$, $2\times n$, $n\times 2$ の大きさの行列が使えます.
対象 points3d の定義でも同様で, リストを用いる場合は $[[x_1,y_1,z_1],\ldots,[x_n,y_n,z_n]]$ と 3 次元空間内の点として表現するか,$[[x_1,\ldots,x_n],[y_1,\ldots,y_n],[z_1,\ldots,z_n]]$ のように X,Y,Z 座標のリストで表現

するか二通りが選べ, 配列の場合は三個の 1 次元配列, 一つの 2 次元配列, 行列の場合は $3 \times n$ か $n \times 3$ の行列に限定されます.

対象 points 対象と points3d には次の固有の属性があります:

points と points3d の固有の属性	
point_size	point_type

この他にも対象 points には他の対象と共通する属性があります. なお, 属性 enhanced3d のみは 3 次元対象の属性ですが, 他の属性は全て 2 次元, 3 次元で共用の属性です:

対象 points の共通の属性			
points_joined	line_width	line_type	xaxis_secondary
yaxis_secondary	color	transform	key
enhanced3d			

ここで与えられた点列に対して点の型や enhanced3d を有効にした例を示しておきます.

```
draw_renderer:gnuplot_pipes$
draw(gr3d(point_size=2, point_type=sphere,
          points_joined=true, color=red,
          line_width=5,line_type=solid,
          points([[0,0,0],[0,10,0],[10,10,10]]),
          point_size=4, point_type=cone,
          points_joined=true,line_width=5,
          line_type=dots,enhanced3d=true, palette=[33,13,10],
          points([[10,0,0],[10,0,5],[0,0,10]])),terminal=eps_color)$
```

この例では最初の点列については点を実線で繋ぎ, 色を属性 color で赤にしていますが, 第 2 の点列については点を点線で繋いで属性 enhanced3d を 'true' にし, 属性 palette の指定を行っています. その結果を図 11.87 に示しますが, 第 2 の点列はその高度に対して色付けが行われ, これは第 2 の点列に付随する線分も同様です. ただし点の形自体は GNUPLOT のために平面的です.

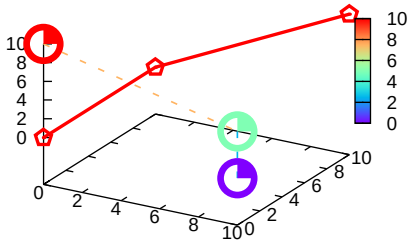


図 11.87: 点列に関連する属性の例, enhanced3d+palette(GNUPLOT)

次に GNUPLOT ではなく vtk で同じ対象を表示したものを示します:

```
draw_renderer:vtk$
draw(gr3d(point_size=2, point_type=sphere,
          points_joined=true, color=red,
          line_width=5,line_type=solid,
          points([[0,0,0],[0,10,0],[10,10,10]]),
          point_size=4, point_type=cone,points_joined=true,
          line_width=5,line_type=dots,enhanced3d=true,
          palette=[33,13,10],
          points([[10,0,0],[10,0,5],[0,0,10]])),terminal=eps)$
```

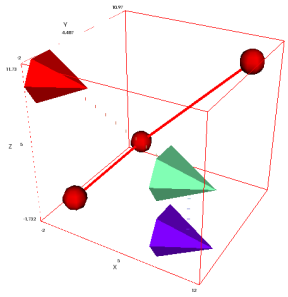


図 11.88: 点列に関連する属性の例, enhanced3d+palette(vtk

図 11.88 に示すように vtk では点が 3 次元形状として表示されます. なお, 点に設定された形状はその点の色が設定されるために形状の高度で色が変化することはありません.

errors 文

errors 文は 2 次元の対象 errors を定義するために用います. この対象 errors は 2 次元の対象 points に似た性格を持ち, 指定した点を基準に位置に特定の長さの水平線, 垂線や箱を配置することができます:

errors を定義する構文

error_type	構文
無指定	errors($[[\langle X_1\rangle,\langle Y_1\rangle,\langle \delta Y_1\rangle],\ldots,[\langle X_n\rangle,\langle Y_n\rangle,\langle \delta Y_n\rangle]]$)
x	errors($[[\langle X_1\rangle,\langle Y_1\rangle,\langle \delta X_1\rangle],\ldots,[\langle X_n\rangle,\langle Y_n\rangle,\langle \delta X_n\rangle]]$)
y	errors($[[\langle X_1\rangle,\langle Y_1\rangle,\langle \delta Y_1\rangle],\ldots,[\langle X_n\rangle,\langle Y_n\rangle,\langle \delta Y_n\rangle]]$)
xy	errors($[[\langle X_1\rangle,\langle Y_1\rangle,\langle \delta X_1\rangle,\langle \delta Y_1\rangle],\ldots,[\langle X_n\rangle,\langle Y_n\rangle,\langle \delta X_n\rangle,\langle \delta Y_n\rangle]]$)
boxes	errors($[[\langle X_1\rangle,\langle Y_1\rangle,\langle \delta X_1\rangle,\langle \delta Y_1\rangle],\ldots,[\langle X_n\rangle,\langle Y_n\rangle,\langle \delta X_n\rangle,\langle \delta Y_n\rangle]]$)

errors 文の引数はリストであり, その成分は対象 errors を配置する点 P_i の X 座標 X_i, Y 座標 Y_i と対象の形を指定する X 側の差分 δX_i, あるいは Y 側の差分 δY_i で構成されたリストです. このリストの座標を除く成分は対象の形に関連し, 対象の形は属性 error_type に依存します:

対象 errors 固有の属性

属性名	取り得る値	既定値	概要
error_type	x, y, xy, boxes	y	対象 erros の形を指定

error_type が 'x' あるいは 'y' の場合: 'x' の場合は指定された点 $P_i = (X_i, Y_i)$ を中心に端点までの長さが δX_i の水平線, 同様に 'y' の場合は指定された点 $P_i = (X_i, Y_i)$ を中心に端点までの長さが δY_i で指定した垂線になります. なお, error_type の既定値は 'y' です.

error_type が 'xy' あるいは 'boxes' の場合: 'xy' の場合は指定された点 $P_i = (X_i, Y_i)$ を中心に, X 軸方向と Y 軸方向の端点までの長さを δX_i と δY_i で指定した水平線と垂線, 同様に 'boxes' の場合は中心点が $P_i = (X_i, Y_i)$, 箱の高さが $2\delta Y_i$, 幅が $2\delta X_i$ の箱を描きます.

次に共通の属性を纏めておきますが, 2 次元の対象 points の属性と共通した属性を持ちます:

対象 errors の属性

points_joined	line_width	line_type	xaxis_secondary
yaxis_secondary	color	fill_density	key

ここで属性 line_types と属性 line_width は属性 points_joined を 'true' にすることで得られる対象 errors の基準点を結ぶ線分の属性になります. また, 属性 fill_density は属性 errors_type を 'box' とした対象にのみ効果があります. では, 実際にこれらの形を確認しておきましょう:

```
draw( columns=3,dimensions=[1800,1200],
      gr2d(title="error\\_type=x",color=blue,error_type=x,
           points_joined=true,xrange=[0,3],yrange=[0,4],
           line_width=2, line_type=dots,errors([[1,2,0.5],[2,3,0.5]])),
      gr2d(title="error\\_type=y",color=green,error_type=y,
           points_joined=true,xrange=[0,3],yrange=[0,4],
           line_width=2,errors([[1,2,0.5],[2,3,0.5]])),
      gr2d(title="error\\_type=xy",color=red,error_type=xy,
           points_joined=true,xrange=[0,3],yrange=[0,4],
           line_width=10,line_type=dots,
           errors([[1,2,0.5,0.5],[2,3,0.5,0.5]])),
      gr2d(title="error\\_type=boxes",color=red,error_type=boxes,
           points_joined=true,xrange=[0,3],yrange=[0,4],
           line_width=2,errors([[1,2,0.5,0.5],[2,3,0.5,0.5]])),
      gr2d(title="error\\_type=boxes,fill\\_density=1",color=red,
           error_type=boxes,points_joined=true,fill_density=1,
           xrange=[0,3],yrange=[0,4],line_width=2,
           errors([[1,2,0.5,0.5],[2,3,0.5,0.5]])));
```

ここでの例では errot_type で指定した形に従って errors 文のリストの成分の長さが異なっていることに注意して下さい. つまり, 'x' と 'y' では 3 成分, 'xy' と 'boxes' では 4 成分となっています. また, この例では全て属性 joints_joints を 'true' としているために基準点が線分で結ばれます. そして, 基準点に配置する形状と互いを結ぶ線分の属性は line_width と line_type で指定されます. この例では errors_type を 'xy' にしたもののみ '10' を指定し, 他は全て '2' を指定しています. また,

属性 line_type は errors_type が ‘x’ と ‘xy’ の場合のみ ‘dots’ を指定し, 他は既定値の ‘solid’ としています. また, 属性 errors_type が ‘box’ のものでは属性 fill_density に 0 より大の数値を指定することで属性 color で指定した色で塗り潰しが行われるために, 最後の ‘box’ の例のみ ‘1’ を指定しています. この結果は図 11.89 に示しておきます:

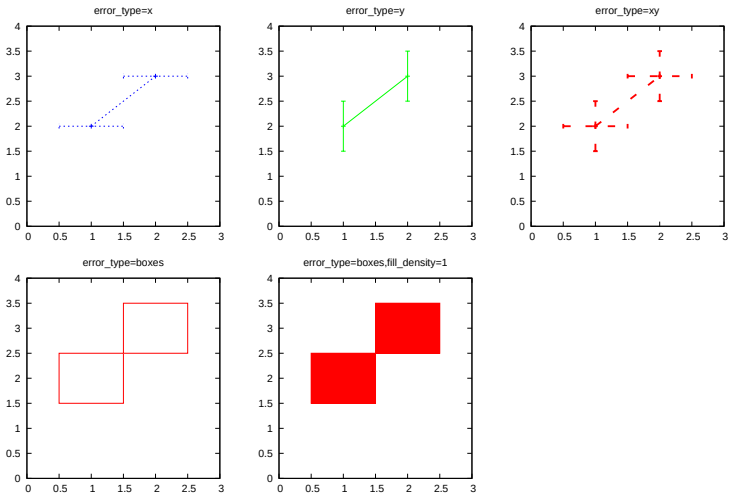


図 11.89: errors_type の型

この図 11.89 に示すように対象 errors の線は属性 line_type, line_width と color の影響を受けることに注意して下さい.

polygon 文

polygon 文は平面上の多角形 (=閉じた折線) を表現する対象 polygon の定義で用いられます. polygon 文では頂点を構成する点列を列記するため, points 文と同様の構文になります:

polygon 文の構文

対象	構文
polygon	polygon([[$\langle X_1\rangle, \langle Y_1\rangle], \ldots, [\langle X_n\rangle, \langle Y_n\rangle]$]
polygon	polygon([$\langle X_1\rangle, \ldots, \langle X_n\rangle$], [$\langle Y_1\rangle, \ldots, \langle Y_n\rangle$])

なお, 対象 polygon は内部で対象 triangle, quadrilateral と rectangle の構築に用いられています. そのために対象 polygon の属性は対象 triangle, quadrilateral と rectangle の属性と共通します:

対象 polygon の共通の属性

transparent	fill_color	color	border
line_width	line_type	xaxis_secondary	yaxis_secondary
transform	key		

簡単な例を次に示しておきます. ここでの例では境界線を太く点線に変更し, さらに同じ多角形を写像 $(x, y) \to (x+2, y+2)$ で変換したものを赤としています：

```
p1:polygon([0,1,2,1],[0,1,0,-1])$
draw(gr2d(title="yellow:original, red:transformed",
          color=black,line_width=5,line_type=dots,
          fill_color=yellow,p1,
          color=black,transform=[x+2,y+2,x,y],
          transparent=true,color=red,
          line_type=solid,line_width=10,p1))$
```

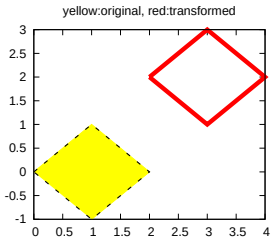


図 11.90: polygon の例

対象 points の点列を属性 points_joined を 'true' としたものと違い, 対象 polygon では自動的に閉じた折線となります.

triangle 文

平面上の三角形を表現する対象 triangle の定義に用います:

triangle 文の構文

triangle([[⟨X1⟩,⟨Y1⟩],[⟨X2⟩,⟨Y2⟩],[⟨X3⟩,⟨Y3⟩]]

引数は三点の XY 座標を表現する 3 個のリストとなります. この対象 triangle の属性には固有のものはありません. triangle の定義で対象 polygon を流用していることもあり, polygon と同じ属性を持ちます:

対象 triangle の共通の属性

transparent	fill_color	color	border
line_width	line_type	xaxis_secondary	yaxis_secondary
transform	key		

簡単な例を次に示しておきます. ここでの例では境界線を太く点線に変更し, さらに同じ多角形を写像 $(x, y) \to (x+2, y+2)$ で変換したものを赤としています：

```
tr:triangle([0,0],[1,0],[1,1])$
draw(gr2d(title="yellow:original, red:transformed",
          color=black,line_width=5,line_type=dots,
```

```
fill_color=yellow,tr,
color=black,transform=[x+2,y+2,x,y],
transparent=true,color=red,
line_type=solid,line_width=10,tr,
xrange=[-0.5,3.5],yrange=[-0.5,3.5]))$
```

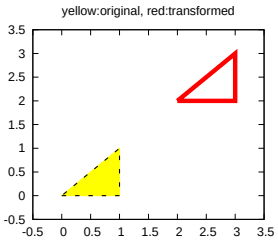


図 11.91: triangle の例

quadrilateral 文

２次元空間での四辺形を表現する対象 quadrilateral と３次元空間内の四辺形を表現する対象 quadrilateral3d の生成に用います:

quadrilateral 文の構文

対象	構文
quadrilateral	quadrilateral($[\langle X_1\rangle,\langle Y_1\rangle],[\langle X_2\rangle,\langle Y_2\rangle],$ $[\langle X_3\rangle,\langle Y_3\rangle],[\langle X_4\rangle,\langle Y_4\rangle]$)
quadrilateral3d	quadrilateral($[\langle X_1\rangle,\langle Y_1\rangle,\langle Z_1\rangle],[\langle X_2\rangle,\langle Y_2\rangle,\langle Z_2\rangle],$ $[\langle X_3\rangle,\langle Y_3\rangle,\langle Z_3\rangle],[\langle X_4\rangle,\langle Y_4\rangle,\langle Z_4\rangle]$)

四辺形を表現するために quadrilateral 文の引数は 4 点を必要とします．２次元の対象 quadrilateral の場合, 線分は $i \in [1,3]$　とするとき $[\langle X_i\rangle,\langle Y_i\rangle]$ と $[\langle X_{i+1}\rangle,\langle Y_{i+1}\rangle]$ の間に引かれます．しかし, ３次元の対象 quadrilateral3d はやや勝手が異なり, $[\langle X_1\rangle,\langle Y_1\rangle]$ と $[\langle X_2\rangle,\langle Y_2\rangle]$, $[\langle X_4\rangle,\langle Y_4\rangle]$ と $[\langle X_3\rangle,\langle Y_3\rangle]$ と引かれ, 残りの二本は $[\langle X_2\rangle,\langle Y_2\rangle]$ と $[\langle X_4\rangle,\langle Y_4\rangle]$, そして, $[\langle X_3\rangle,\langle Y_3\rangle]$ と $[\langle X_1\rangle,\langle Y_1\rangle]$ の間に引かれるので注意が必要です．
属性については, まず, ２次元の対象 quadrilateral は内部で対象 polygon を流用しているために対象 polygon の属性の共通を持ちます:

対象 quadrilateral の属性

transparent	fill_color	color	border
line_width	line_type	xaxis_secondary	yaxis_secondary
transform	key		

ここで簡単な例を示しておきます. その結果は図 11.92 です. この例題は本質的に対象 polynomial, triangle の例題と全く同じものです:

```
qd:quadrilateral([0,0],[1,1],[2,0],[1,-1])$
draw(gr2d(title="yellow:original, red:transformed",
          color=black,line_width=5,line_type=dots,
          fill_color=yellow,qd,
          color=black,transform=[x+2,y+2,x,y],
          transparent=true,color=red,
          line_type=solid,line_width=10,qd,
          xrange=[-0.5,4],yrange=[-1.5,4]))$
```

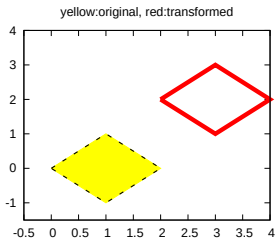


図 11.92: quadrilateral の例

3次元の対象 quadrilateral3d は対象 mesh を流用して定義されます. そのために対象 quadrilateral3 の属性は対象 mesh の属性と共通です:

対象 quadrilateral3d の属性

line_width	line_type	color	key enhanced3d	transform

この例では enhanced3d を 'true' にしています. そのために表に入れていない属性 enhanced3d と関連する contour や colorbox 属性等が扱えます.

```
qd:quadrilateral([0,0,0],[1,1,1],[1,-1,-2],[2,0,2])$
draw(columns=2,dimensions=[1600,600],
     gr3d(title="quadrilateral3d, enhanced=true",
          enhanced3d=true, palette=[30,31,32],contour=both,
          line_width=1,qd,transform=[x+2,y+2,z,x,y,z],line_width=5,
          line_type=solid,line_width=10,color=green,qd),
     gr3d(title="quadrilateral3d, enhanced=false",
          line_width=5,line_type=dots,color=red,qd,
          transform=[x+2,y+2,z-1,x,y,z],contour=surface,
          line_type=solid,line_width=10,color=green,qd))$
```

ractangular 文

2次元の対象 rectangular の生成で用います:

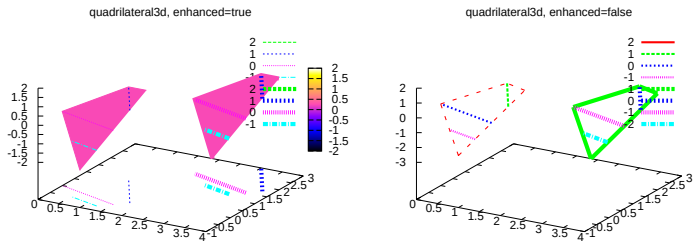


図 11.93: quadrilateral の例 (3d)

rectangle 文の構文

rectangle([⟨X1⟩,⟨Y1⟩],[⟨X2⟩,⟨Y2⟩])

こでこ [⟨X1⟩,⟨Y1⟩] と [⟨X2⟩,⟨Y2⟩] が定める点によって長方形の対角線が決ります. この対象 rectangle も対象 polygon を内部で用いているために属性は polygon と共通のものになります:

対象 rectangle の共通の属性

transparent	fill_color	color	border
line_width	line_type	xaxis_secondary	yaxis_secondary
transform	key		

以下の例は対象 polygon, triangle, quadriteral2d の例に属性 key を追加したものです:

```
rt:rectangle([0,0],[1,1])$
draw(gr2d(title="yellow:original, red:transformed",
          color=black,line_width=5,line_type=dots,
          fill_color=yellow,key="Original",rt,
          color=black,transform=[x+2,y+2,x,y],
          transparent=true,color=red,key="Transformed",
          line_type=solid,line_width=10,rt,
          xrange=[-0.5,4],yrange=[-1.5,4]))$
```

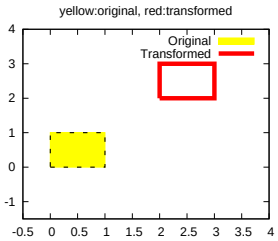


図 11.94: rectangle の例

ellipse 文

ellipse 文は楕円を表現する 2 次元の対象 ellipse を定義するために用います:

ellipse 文の構文

ellipse(⟨X⟩, ⟨Y⟩, ⟨R_x⟩, ⟨R_y⟩, ⟨α⟩, ⟨β⟩)

(⟨X⟩,⟨X⟩) で楕円の中心点, ⟨R_x⟩ で X 軸方向の半径, ⟨R_y⟩ で Y 軸方向の半径とします. ⟨α⟩ と ⟨β⟩ は 0 から 360 までの角度で, ⟨α⟩ が動径の開始位置における X 軸となす角度, ⟨β⟩ は動径の動く角度に対応します. この対象 ellipse も内部で対象 polygon を用いているために, 対象 polygon と共通の属性を持ちます. さらに楕円の近似の度合いを調整するための属性として属性 nticks を持っており, この nticks に大きな値を与えることでより良い近似が得られるようになっています. 以下に対象 ellipse の属性を纏めておきます:

対象 ellipse の属性

nticks	transparent	fill_color	color	border
line_width	line_type	xaxis_secondary	yaxis_secondary	transform
key				

ここでの例は基本的に対象 polygon, triangle, quadriteral と rectangle と同様の属性を用いています. draw パッケージから GNUPLOT を利用するときは拡張モードであるために拡張制御記号に加えてフォントの切り替えも行えます. この例では"{/Symbol ⟨a⟩}" によって a に対応するギリシャ文字のアルファベット α を出力させています:

```
e1:ellipse(0,0,7,5,0,360);
e2:ellipse(0,0,7,5,60,60);
draw(dimensions=[1000,1000],
     gr2d(title="yellow:{/Symbol b-a=2p}, red:{/Symbol b-a=p/3}",
          color=black,line_width=5,line_type=dots,
          fill_color=yellow,key="{/Symbol b-a=2p}",e1,
          transform=[x+2,y+2,x,y],transparent=true,
          color=red,key="{/Symbol b-a=p/3}",
          line_type=solid,line_width=10,e2,
          xrange=[-8,8],yrange=[-6,8]));
```

label 文

label 文は平面グラフ中の対象 label や 3 次元グラフ中の対象 label3d を定義するために用います. なお, label 文では複数のラベルを同時に定義できます.

label 文の構文

label	label([⟨S_1⟩, ⟨X_1⟩, ⟨Y_1⟩], ..., [⟨S_n⟩, ⟨X_n⟩, ⟨Y_n⟩])
label3d	label([⟨S_1⟩, ⟨X_1⟩, ⟨Y_1⟩, ⟨Z_1⟩], ..., [⟨S_n⟩, ⟨X_n⟩, ⟨Y_n⟩, ⟨Z_n⟩])

⟨S⟩ が文字列で, 文字列が配置される位置は属性 label_allgnment で指定され, その後に続く座標に配置されます. なお, この属性の既定値は 'center' であるために文字列の中心の座標を指定するか

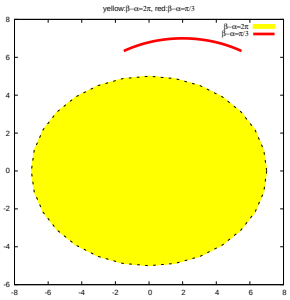


図 11.95: ellipse の例

たちになります. ここでの文字列は Maxima の文字列, すなわち, 二重引用符で文字の列を括ったものです. 二重引用符で括っていなければ, Maxima の変数と見做されるため, 束縛変数であればその値が表示され, 自由変数であれば LISP 内部の書式で処理されるために先頭に記号 “$” が付いた文字の列が表示されます.
なお, 対象 label3d に含まれる文字列は 3 次元の対象として生成されません. あくまでも GNUPLOT や vtk 側の機能を用いて表示させています. そのために視点に依存することなく文字列は常に正面を向きます. これは GNUPLOT で対象 points3d に記号を指定したときに視点に依存することなく常に正面を向いているのと同様です[12].
対象 label 固有の属性として, ラベルの位置合せと並べ方に関する属性があります:

label 固有の属性

属性名	取り得る値	既定値	概要
label_alignment	center,left,right	center	ラベルの位置合せ
label_orientation	horizontal,vertical	horizontal	ラベルの並び

label_alligment:　ラベルの位置合せを center(=中央), left(=左寄), right(=右寄) で行います. 既定値は center となっています.
それ以外の対象 label の属性を示しておきます:

label_orientation:　対象 label の向きを horizontal(=水平), vertical(=垂直) で指定します. 既定値は horizontal のため, 文字列は X 軸に平行に表示されます.

[12] vtk で points3d を表示するとき, 点に配置した形状は普通に回転します

対象 label のその他の属性

color	xaxis_secondary	yaxis_secondary

ここでの例では, ラベルを中央, 左詰め, 右詰めと色を変更したものです. また, 文字列には拡張モードの機能を使っています:

```
text1:label(["test: Center {/Symbol a_1}",0,0]);
text2:label(["test: Left  {/Symbol ~b{.8-}}",0,0.5]);
text3:label(["test: Right  {/Symbol c^2}",0,1]);
draw(gr2d(title="label: test",
          xrange=[-0.5,0.6],yrange=[-0.2,1.2],
          color=black, label_alignment=center,text1,
          color=blue,  label_alignment=left,text2,
          color=red,   label_alignment=right,text3));
```

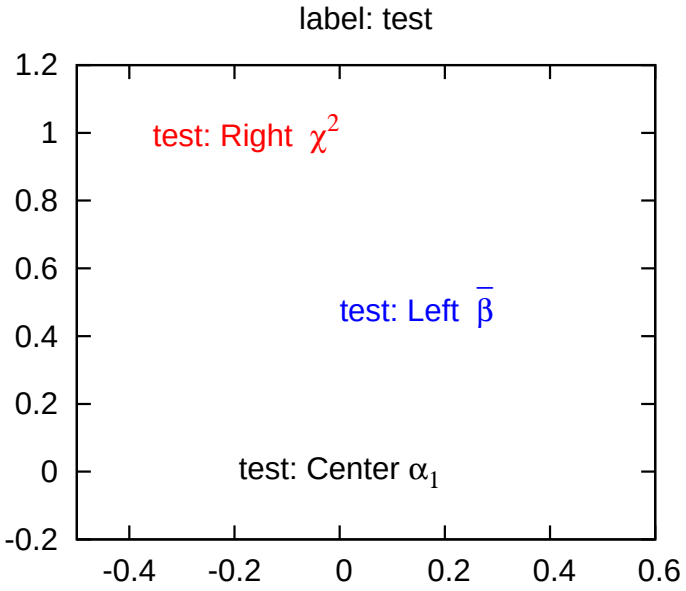


図 11.96: label の例

bars 文

bar 文は 2 次元の対象 bars を定義するために用います:

bars 文の構文

bars([⟨X$_1$⟩,⟨H$_1$⟩,⟨W$_1$⟩],...,[⟨X$_n$⟩,⟨H$_n$⟩,⟨W$_n$⟩])

三成分リスト [⟨X$_i$⟩,⟨H$_i$⟩,⟨W$_i$⟩] により, ⟨X$_i$⟩ で指定された X 軸上を中心に高さ ⟨H⟩, 幅 ⟨W⟩ の棒を定義します. 棒を複数定義する場合, 各棒に対応する 3 成分のリストの列を bars 文に引数として与えます.

対象 bars の属性

fill_color	fill_density	line_width	xaxis_secondary	yaxis_secondary
key				

ここでの例では属性 fill_color で対象 bars の色を決め, line_width で各棒の境界線の太さを決めています. 属性 fill_density は内部の透明度に影響を与えますが, 対象の境界線には影響を及ぼしません:

```
bars1:bars([0,1,0.5],[1,3,0.5],[2,2,0.5]);
bars2:bars([0.5,2,0.5],[1.5,1,0.5],[2.5,4,0.5]);
bars3:bars([3,1,0.5],[3.5,3,0.5],[4,2,0.5]);
draw(gr2d(title="bars: test",yrange=[0,6],line_width=10,
          fill_color=yellow,fill_density=0.2,bars1,
          fill_color=blue,fill_density=0.5,bars2,
          fill_color=red,fill_density=0.8,bars3));
```

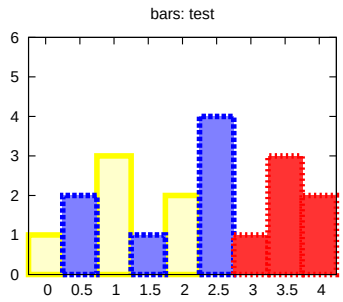


図 11.97: bars の例

なお, bars 要素でグラフを記述する場合, 属性 yrange を指定していなければ, 最も低い棒の頂点と最も高い棒の頂点で Y 座標の領域が区切られてしまい, 最も低い棒が見えないという状況に陥いるので注意しましょう. 上の例ではその状況を避けるために属性 yrange の指定を行っています.

vector 文

vector 文はベクトルを表現する矢印を生成します. 平面上の矢印は対象 vector, 3 次元空間内の矢印は対象 vector3d で表現されています:

vector の構文

vector	vector([⟨X⟩, ⟨Y⟩], [⟨δX⟩, ⟨δY⟩])
vector3d	vector([⟨X⟩, ⟨Y⟩, ⟨Z⟩], [⟨δX⟩, ⟨δY⟩⟨δZ⟩])

ベクトルは基点となる点の座標と, 向きを表現する座標の組み合わせで表現されます. 2 次元の場合, 基点の X 座標が ⟨X⟩, Y 座標が ⟨Y⟩ とし, X 軸方向の向きを ⟨δX⟩, Y 軸方向の向きを ⟨δY⟩ としています. 3 次元の場合も同様に基点の各座標を ⟨X⟩, ⟨Y⟩, ⟨Z⟩ とし, 各軸の向きを ⟨δX⟩, ⟨δY⟩, ⟨δZ⟩ とします.
対象 vector に固有の属性は矢印の矢の部分の属性となります:

vector 固有の属性

属性	型	既定値	概要
unit_vectors	真理値	false	単位ベクトルの描画の有無
head_angle	正整数値	45	矢印の頭の角度
head_both	論理値	false	矢印の矢の設定
head_length	正整数値	2	矢印の柄の長さ
head_type	lilled,empty,nofilled	filled	矢印の頭の矢の型

unit_vectors: 単位ベクトルで描画を行うかどうかを指示する属性です. 既定値は 'false' で値が 'true' の場合に単位ベクトルで描画を行いますが, 'false' の場合は実際の長さのままで描画を行います.

```
v1:vector([0,0],[3,5]);
p1:points([[0,0],[3,5]]);
v2:vector([0,1],[3,2]);
p2:points([[0,1],[3,3]]);
draw(columns=2,dimensions=[1200,600],
     gr2d(title="unit\\_vectirs:false", head_length=0.25,
          point_size=5,unit_vectors=false, color=red,v1,color=blue,
          point_type=circle,p1,v2,color=red,point_type=square,p2,
          xrange=[-1,4],yrange=[-1,6]),
     gr2d(title="unit\\_vectirs:true", head_length=0.25,
          point_size=5,unit_vectors=true, color=red,v1,color=blue,
          point_type=circle,p1,v2,color=red,point_type=square,p2,
          xrange=[-1,4],yrange=[-1,6]));
```

head_angle: ベクトルの矢印の頭の傾きを指定する属性です. 単位は [度] で, その既定値は 45 です.

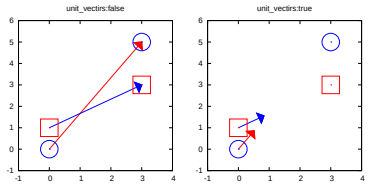


図 11.98: vector の例 (unit_vectors)

head_both:　ベクトルの矢印を片方のみとするか, 両側に持たせるかを定める属性です. 既定値は 'false' ですが, 'true' であれば両方に矢印を持ちます.

head_length:　ベクトルの矢印の箇所の長さを定めます. 既定値は '2' です. ベクトルの全体の長さは vector の定義で入れた δX, δY, δZ から定まりますが, ベクトルの頭はこの値で長さが固定されます. つまり, 大きさを全体の比率で定めるものではありません. そのために既定値のままで描画していれば, 小さな範囲で描けば頭だけで画面が埋まり, 大きな範囲で描けはベクトルの頭がまるで判らないという間の抜けたことになるので注意が必要です.

head_type:　ベクトルの矢印の箇所の有無, 塗り潰しを指定します. 選べる値は矢印の頭の塗り潰しを行う 'filled', 頭の塗り潰しのみがない 'empty' と頭が上の笠だけの 'nofilled' の三種類で, 既定値は 'filled' です:

```
v1:vector([0,0],[3,5]);    v2:vector([0,1],[3,2]);
v3:vector([0,-1],[1,2]);   p1:points([[0,0],[3,5]]);
p2:points([[0,1],[3,3]]); p3:points([[0,-1],[1,1]]);
l1:label(["filled",3,5.5]); l2:label(["empty",3,3.5]);
l3:label(["nofilled",1,1.5]);
draw(gr2d(title="HEAD TYPE", head_length=0.25,head_type=filled,
          point_size=5,color=red,v1,color=blue,head_type=empty,
          point_type=circle,p1,l1,v2,color=red,point_type=square,p2,l2,
          head_type=nofilled,color=green,v3,color=black,point_type=square,
          p3,l3,xrange=[-1,4],yrange=[-2,6]),dimensions=[600,600]);
```

対象 vector は線分としての性格も持つために, 他と共通の属性は線分としての属性が主になります:

vector のその他の属性

line_width	line_type	color	key

explicit 文

explicit 文で定義可能な対象は 2 次元グラフの対象 explicit と 3 次元グラフの対象 explicit3d です. ここで対象 explicit は $y = f(x)$ の書式の 1 変数実数函数のグラフを表現する対象です. そして, 対象 explicit3d は $z = f(x, y)$ の書式の 2 変数実数値函数のグラフを表現する対象です. なお, 式の値域は絶対値が '1.75555970201398d+305' を越えない必要があります:

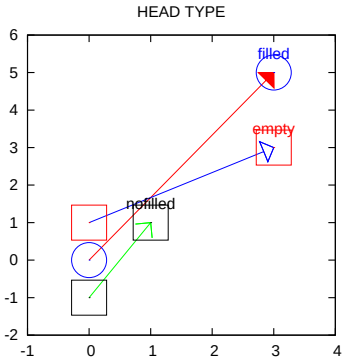


図 11.99: vector の例 (head_type)

explicit 文の構文

対象	構文
explicit	explicit($\langle f(X)\rangle, \langle X\rangle, \langle X_0\rangle, \langle X_1\rangle$)
explicit3d	explicit($\langle f(X,Y)\rangle, \langle X\rangle, \langle X_0\rangle, \langle X_1\rangle, \langle Y\rangle, \langle Y_0\rangle, \langle Y_1\rangle$)

対象 explicit の定義では, その函数の実体と変数, そして定義域を与える必要があります. この構文での函数 $\langle f(X)\rangle$ は変数を $\langle X\rangle$ とする 1 変数函数で, この変数の定義域は閉区間 $[\langle X_0\rangle, \langle X_1\rangle]$ であるために $\langle X_0\rangle \leq \langle X_1\rangle$ を充していなければなりません. 同様に対象 explicit3d の定義にて, 利用する函数は 2 変数函数 $\langle f(X,Y)\rangle$ で, 第 1 変数の $\langle X\rangle$ の定義域を閉区間 $[\langle X_0\rangle, \langle X_1\rangle]$, 第 2 変数の $\langle Y\rangle$ の定義域を閉区間 $[\langle Y_0\rangle, \langle Y_1\rangle]$ で定めます. したがって, $\langle X_0\rangle \leq \langle X_1\rangle$ かつ $\langle Y_0\rangle \leq \langle Y_1\rangle$ でなければなりません.

対象 explicit 固有の属性

属性	型	既定値	概要
adapt_depth	正整数値	10	描画助変数
filled_func	真理値等	false	塗り潰しの領域を指定

属性 adaptive_depth: 2 次元の対象 explicit のみで用いられる属性です. この属性は plot2d 函数で参照される大域変数 plot_option に含まれる同名のオプション adapt_depth に対応します. この理由ですが, plot2d 函数や対象 explicit で用いる適応型描画ルーチンは YACAS の plot2d 函数をベースにした函数 adaptive-plot を用いており, その一つの引数として adapt_depth の値が参照されます. この値は適応的描画での最大の分離数として用いられ, 大きな値程, より滑かに表示が行えることになります. 既定値が '5' となっていますが, explict 文では倍の '10' が既定値となっています.

この応的描画は滑かな函数に対しては然程の効果はありませんが, $\sin 1/x$ の原点付近の様に激しく

振動する箇所の描画で, 適切に領域の分割を行って描画を行うために特に効果があります. ここでは $\sin 1/x$ に対して adapt_depth を 2, 5, 20 で比較してみましょう:

```
rsx:explicit(sin(1/x),x,-1/10,1/10);
draw(dimensions=[1500,900],
      gr2d(title="adapt\\_depth=2", adapt_depth=2, color=black,rsx),
      gr2d(title="adapt\\_depth=5", adapt_depth=5, color=blue, rsx),
      gr2d(title="adapt\\_depth=20",adapt_depth=20,color=red,  rsx));
```

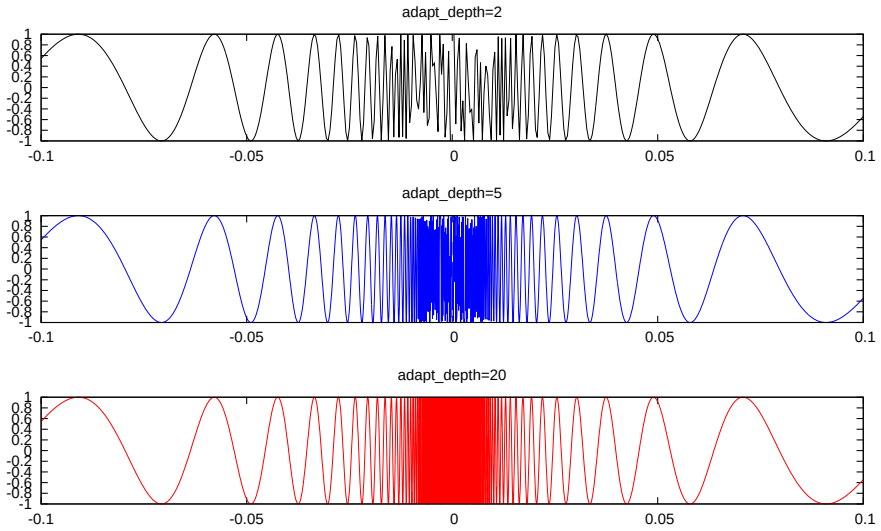


図 11.100: adapt_depth の効果

図 11.100 に示すように属性 adapt_depth の値が大きい程, 原点付近の振動が綺麗に表示されています. とは言え, あまり数が大きいとその分, 処理時間も長くなります.

属性 fill_func:　2 次元の対象 explicit のみで利用可能な属性で, 塗り潰しの領域を定める函数を指定します. この既定値は 'false' で, このときは塗り潰しを行わず, 'true' であれば X 軸と対象 explicit の定める図形を境界とする領域, 対象 explicit の変数を用いて定義した 1 変数の実数函数を指定した場合は, 対象 explicit とその函数で挟まれた領域の塗り潰しを行います. 結果は対象 region で得られる結果に似ていますが, こちらは陽的に座標が定まるという大きな違いがあります. なお, exlicit で描かせる函数と fill_func に指定する函数の双方が函数の定義域で特異点を持たなければなりません. $\sin x/x$ の原点のような除去可能な特異点でもエラーになることがあるので注意して下さい. また, 属性 filled_func が与えられた gr2d 文内の属性 color, line_width と line_type は全て無視されます.

```
rsx:explicit(sin(1/x)/x,x,0.02,0.1);
draw(columns=2,dimensions=[1200,600],
      gr2d(title="filled\\_func=true",filled_func=true, fill_color=red, rsx),
```

```
gr2d(title="filled\\_func=1/x", filled_func=1/x, fill_color=blue,rsx));
```

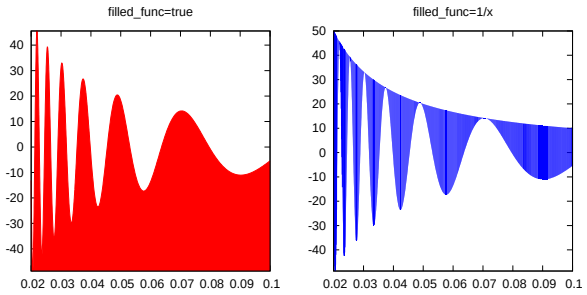


図 11.101: filled_func の効果

以下に共通の属性を纏めておきます:

対象 explicit の属性

nticks	line_width	line_type	color	fill_color	key

対象 explicit3d には以下の属性があります:

対象 explicit3d の属性

xu_grid	yv_grid	contour	contour_levels	enhance3d
line_width	line_typ	color	key	

region 文

region 文は 2 変数の述語函数を充す領域を表現する対象 region の定義で用いられます:

対象 region の構文

region(⟨P(X, Y)⟩, ⟨X⟩, ⟨X_0⟩, ⟨X_1⟩, ⟨Y⟩, ⟨Y_0⟩, ⟨Y_1⟩)

第 1 引数の述語は 2 変数の論理式です. より具体的には, 1 変数, または 2 変数の不等式や等式を論理和 'or' と論理積 'and' で結合した式です. たとえば, 'x¿0', 'x¡1' ,'y¿0' と 'y-x¡0' で囲まれる領域であれば 'x¿0 and x¡1 and y¿0 and y-x¡0' が述語となります. それに続いて, 述語を充す領域を探索する範囲を続けて記述します. この領域については, 第 1 変数とその範囲, 第 2 変数とその範囲の順に指定します. ここでの区間の指定は対象 explicit3d を定義する際の explicit 文の構文と同じです.

この対象 region の属性を以下に纏めておきます:

region の属性

fill_color	x_voxel	y_voxel	key

対象 region は対象 implicit や対象 implicit3d と同様に, 第 1 引数で指定された式や述語を充す領域を探索する手法のため, 各軸方向の領域の分割数が足りなければ粗い結果しか戻らない可能性が十分にあります. その場合は属性 x_voxel や属性 y_voxel の数値を大きくする必要があります. ただし, これらの数値を大きくすることで計算量が増加するために計算時間は当然長くなります.

ここで図 11.101 の領域を対象 region で描いてみましょう:

```
rg1:region(y-sin(1/x)/x<=0,x,0.02,0.1,y,-40,40);
rg2:region(y-sin(1/x)/x>=9 and y<=1/x,x,0.02,0.1,y,-40,40);
draw(columns=2,dimensions=[1200,600],
     gr2d(title="x\\_voxel,y\\_voxcel=10x10",x_voxel=10,y_voxel=10,rg1),
     gr2d(title="x\\_voxel,y\\_voxcel=10x10",x_voxel=10,y_voxel=10,rg2),
     gr2d(title="x\\_voxel,y\\_voxcel=100x100",x_voxel=100,y_voxel=200,rg1),
     gr2d(title="x\\_voxel,y\\_voxcel=100x100",x_voxel=100,y_voxel=200,rg2));
```

この例では各 voxel を既定値の 10 に対して 100 や 200 に変更したものの比較を行います.

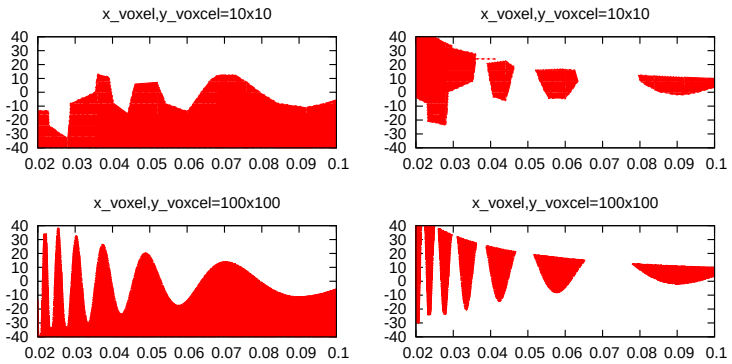


図 11.102: region の例

この図 11.102 では上の各 voxel が既定値の結果は全く駄目で, 100x200 でかろうじてその雰囲気が判る程度になっています. これは対象 explicit では陽的に座標函数が与えられているので簡単に点列が計算できるのに対し, 対象 region では領域を voxel で指定した区画に分割し, その区画が条件を充すかを一々判別しているからです. これは対象 region に限らず, 対象 implicit や対象 implicit3d でも同様です.

implicit 文

implicit 文は 2 次元の対象 implicit や 3 次元の対象 implicit3d の生成で用います. この対象 implicit は $x^2+y^2-1=0$ のような 2 変数, $x^2+y^2+z^3-1=0$ のような 3 変数の陰函数のグラフの表示に用います.

対象 implicit の構文

対象	構文
implicit	implicit(⟨f(X, Y)⟩, ⟨X⟩, ⟨X_0⟩, ⟨X_1⟩, ⟨Y⟩, ⟨Y_0⟩, ⟨Y_1⟩)
implicit3d	implicit(⟨f(X, Y, Z)⟩, ⟨X⟩, ⟨X_0⟩, ⟨X_1⟩, ⟨Y⟩, ⟨Y_0⟩, ⟨Y_1⟩, ⟨Z⟩, ⟨Z_0⟩, ⟨Z_1⟩)

対象 implicit の定義では 2 変数の式と各変数の取り得る範囲, 対象 implicit3d の定義では 3 変数の式と各変数の取り得る範囲を引数とします. ここでの式は 'x + y' のように等号 "=" を持たない式でも 'x + y = 0' のように等号 "=" を持つ式のどちらでも使えます. もし, 式に等号を持たない場合は与式が 0 となる領域の描画を行います. つまり, 'y-sin(x)/x' であれば 'y - sin(x)/x=0' を充す領域が描かれます.

次に 2 次元の対象 implicit 固有の属性を次に示しておきます:

対象 implicit 固有の属性

属性	型	既定値	概要
ip_grid	二成分の正整数リスト	[50,50]	点数
ip_grid_in	二成分の正整数リスト	[5,5]	点数

属性 ip_grid: 1 次抽出で用いる網目を定めます. 既定値は '[50,50]' です.

属性 ip_grid_in: 2 次抽出で用いる網目を定めます. 既定値は '[5,5]' です.

では, 属性 ip_grid と属性 ip_grid_in の効果を確認しましょう. そのために, ここでは各属性の既定値の半分と逆にその既定値の倍を組み合わせてみましょう:

```
dfl:implicit(x^3+x^2-y^2,x,-2,2,y,-2,2);
draw(columns=2,dimensions=[1200,1200],
     gr2d(title="ip\\_grid=[25,25], ip\\_grid\\_in=[2,2]",
          ip_grid=[25,25],ip_grid_in=[2,2],color=red,line_width=2,dfl),
     gr2d(title="ip\\_grid=[25,25], ip\\_grid\\_in=[10,10]",
          ip_grid=[25,25],ip_grid_in=[10,10],color=blue,line_width=2,dfl),
     gr2d(title="ip\\_grid=[100,100], ip\\_grid\\_in=[2,2]",
          ip_grid=[100,100],ip_grid_in=[2,2],color=blue,line_width=2,dfl),
     gr2d(title="ip\\_grid=[100,100], ip\\_grid\\_in=[10,10]",
          ip_grid=[100,100],ip_grid_in=[10,10],color=red,line_width=2,dfl));
```

この例から判るように属性 ip_grid は全体の滑かさに影響を与えます. この値が小さい上の段の代数曲線 (Decaltes の葉形曲線) は下段と比べて明らかに滑かさで劣ります. 実際, この属性に従って領域の分割が行われます. ところで, 左側の図は上下共に連結ではありません. それと比べると右側は連結性は上下共に良く表現できていると言えるでしょう. このように属性 ip_grid_in の値が大きい方が領域の連続性をきちんと表現することができます. このように両者をひたすら大きくするのではなく, その要求に合せてこれらの属性を調整すると無駄に長い計算をしなくて済むでしょう.

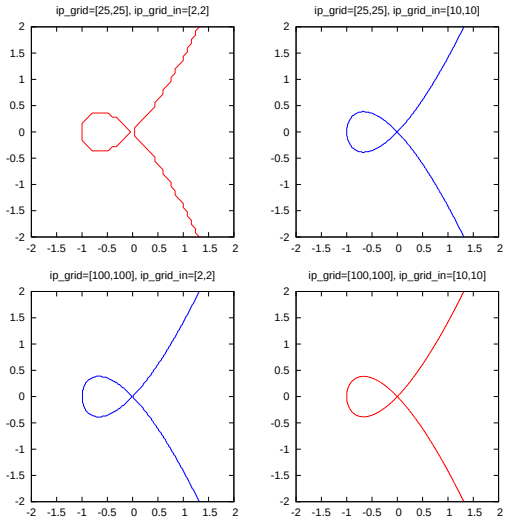


図 11.103: ip_grid, ip_grid_in について

対象 implicit と implicit3d の属性を以下にまとめておきます:

implicit と implicit3d の属性

x_voxel	y_voxel	z_voxel	line_width	line_type
color	enhanced3d	wired_surface		

対象 implicit や対象 implicit3d では対象 region と同様に第 1 引数で指定された式が 0 となる領域を探索する手法のため, 各軸方向の領域の分割数が足りなければ粗い結果となります. その場合は属性 x_voxel や属性 y_voxel の数値を大きくすれば改善することもありますが, その反面, 計算時間が長くなることや, 計算時間がかかった割に不十分な結果しか得られない場合もあります. 代数曲線や代数曲面の描画は surf, surfer や surfex で行う方がより綺麗な結果が得られます. Maxima から Surfer を使う場合は §15 の「surf を使う話」を参照して下さい.

elevation_grid 文

対象 elevation_grid は行列で定義された曲面を表現する対象です. そのために対象 elevation_grid は 3 次元の対象となります:

対象 elevation_grip の構文

elevation_grip(⟨M⟩, ⟨X_0⟩, ⟨Y_0⟩, ⟨W⟩, ⟨H⟩)

第 1 引数の ⟨M⟩ は行列です. 曲面の X 軸は行列 ⟨M⟩ の行, Y 軸は行列 ⟨M⟩ の列, 曲面の Z 座標は ⟨M⟩ の値が対応します. 曲面の XY 平面における原点は (⟨X_0⟩, ⟨Y_0⟩) で与えられ, 曲面の XY 平面での幅が ⟨W⟩, XY 平面での高さが ⟨H⟩ で定められます.

elevation_grid の属性

x_voxel	y_voxel	z_voxel	line_width	line_type
color	enhanced3d	wired_surface		

画像行列を生成しますが, ここではより一般的な画像にします. ただし, あまりにも大きな画像は莫大な記憶容量と計算能力を必要とするのでそこそこの大きさの画像とします. そこで次に示す歌川国吉の浮世絵を使ってみましょう:

図 11.104: 相馬の古内裏

この画像は wikipedia の「歌川国吉」の頁から入手した画像ですが, ここでは 600x289 画素に大きさを縮小しています. とは言え, これでも大きな画像です. この画像をグレースケールの画像に変換し, Maxima の行列として取込む必要があります. 画像の取り込み方法としては draw パッケージに含まれる read_xpm 函数が使えますが, こちらは画像の書式が XPM に限定されることに加え, 出来も今一つ良いものではありません. そのために, ここでは処理を Yorick で行います. Yorick は多元配列の処理を得意とする C 風の対話処理言語を持ったソフトウェアです. 何故, Yorick でなければならないかという理由ですが, それは非常に単純で GNU Octave と比較して非常に軽量で処理が十分に速いことと, Octave と比べて C 風の言語なので言語の説明をしなくても意味が大体判るからです. また, MathLibre には Yorick が使える環境にあるという点も大きいのです. 今回, 画像ファイルは 'KodairiS.jpeg' として固定しておきます. そして, Yorick では画像を読み込んで各点の RGB の平均を計算し, その値を整数に変換します. この時点で 2 次元配列となっているので, あとは Maxima の行列を定義する matrix 函数の行ベクトルへの書き出しを while 文を使って行います:

```
1 A=img_read("KodairiS.jpeg"); /* 画像の読込 */
2 B=long(A(avg,));             /* RGB の平均を計算し, long 型の整数に変換 */
3 sz=dimsof(B);                /* 配列 (行列)の大きさ */
4 f=open("kodairi.mac","w");   /* ファイルのオープン */
5 /* matrix 函数で定義する行列の行の書き出し */
6 write,f,"K:matrix(";         /* Maxima の matrix 文の頭の部分 */
7 for(i=1;i<=sz(2);i++){
8     if(i!=sz(2))px=",";
```

```
9       else            px="";
10      write,f,format="[%s","";
11      for(j=1;j<sz(3);j++) write,f,format="%d,",B(i,j);
12      write,f,format="%d]%s\n",B(i,sz(3)),px;};
13  write,f,format="%s);\n","";
14  close,f;                        /* ファイルを閉じる */
```

この内容を直接 yorick に入れても構いませんが, 'test1.i' といった名前のファイルに保存し, 'yorick -batch test1.i' とバッチ処理を行えば Maxima へ引き渡す行列を含むファイル 'kodairi.mac' が生成されます. このファイルの中身は 'K:matrix(...);' という式のみが含まれており, このファイルの読込を行うと行列 K が構成されます. あとはグラフの表示のみです:

```
load("kodairi.mac")$
kodairi:elevation_grid(K,0,0,2,1)$
draw(gr3d(enhanced3d=true,palette=gray,kodairi));
draw(gr3d(enhanced3d=true,palette=gray,view=[0,180],
          kodairi),dimensions=[300,600]);
draw(gr3d(enhanced3d=true,contour=map,kodairi));
```

最初は視点を標準のままで palette を gray とすることでグレースケール表示を行なうもので, 結果は図 11.7.10 に示すものです:

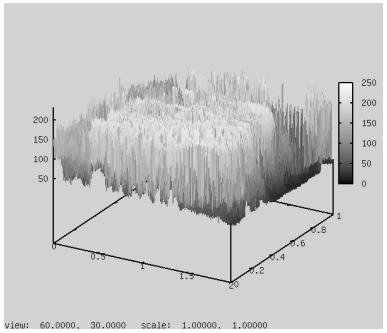
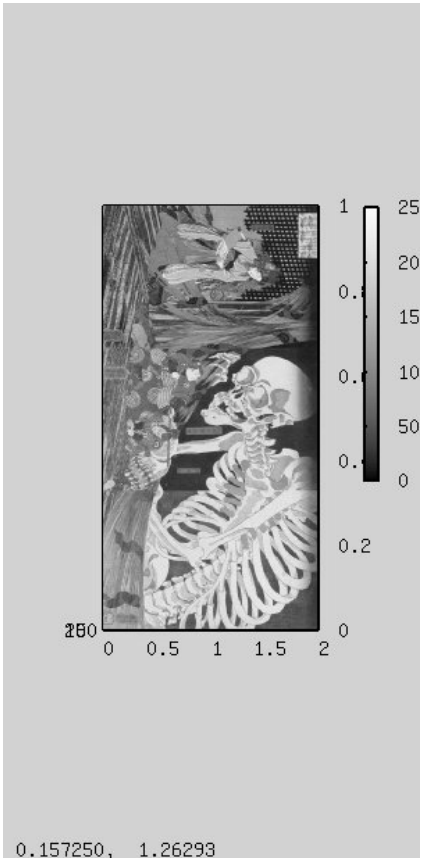
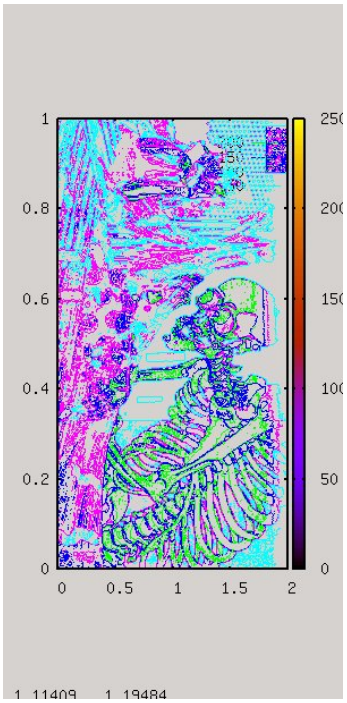


図 11.105: palette=gray

この図 11.7.10 から判るように行列 M の [i, j] 成分の M[i, j] がグラフの高さ, [$i/(W-X_0)$,$j/(H-Y_0)$] で X-Y 座標が与えられます. 次に視点を変えたり, contour=map にした例を示します:



view[0,180]



contour=map

図 11.7.10 から判るように確かに「相馬の古内裏」は読み込まれていることが判ります. また, この図を等高線で表示させたものが図 11.7.10 で, 階調が同じ箇所を描くことになるので, 骸骨が浮び上っています. とはいえ, 他の登場人物はやや？な状態です.
これらのように対象 elevation_grid で用いる行列を高さを色の濃淡で表示している場合, 真上から見たほうが判り易いものです. これを elevation_grid で処理させてゆくのも一手ですが, ここまでの画像表示はそれなりに時間が掛っています. 別に 2 次元の対象 image を用いる手段もあります.

image 文

image 文は対象 image を定義するために用いられます. なお, image 文では elevation_grip 対象の定義に用いる画像行列に加え, read_xpm 函数で読み込んだ画像や行列から函数 make_level_picture を用いて構成される picture という対象によって定義が行われます:

image 文の構文

image(〈 実数行列 〉, 〈X_0〉, 〈Y_0〉, 〈W〉, 〈H〉)
image(〈*pciture*〉, 〈X_0〉, 〈Y_0〉, 〈W〉, 〈H〉)

ここでの引数は対象 elevation_grid の引数と同じ性質のものですが, elevation_grip は 3 次元で可視化を行う関係上, カラー画像のように赤, 青, 緑の RGB の各チャンネルを持つ画像は elevation_grid の例のように三色の平均を取るか, チャンネル別に対象を生成する必要があります. この対象 image は RGB のチャンネルを持つ画像を取込むことのできる対象です. なお Maxima で画像の取り込みは, elevation_grid を用いて画像行列として読み込む方法と XPM 形式のファイルを直接, 読み込む方法の二種類があります. なお, 画像行列の読込では, 対象がカラーであれば赤 (R), 青 (G), 緑 (B) の三原色に対応する行列に分解し, それを Maxima で読み込んだのちに make_level_picture 函数で各チャンネル別に picture に変換し, それを make_rgb_picture 函数で一つの picture に統合するという手順を踏みます. 後者の画像読込では read_xpm 函数を用いますが, XPM ファイルの註釈行の処理に問題があり, 上手く動作しないことがあります. これらの詳細については §11.8.1 にまとめておきます.
対象 image の属性は, その表示対象の性質から, 色彩の属性 palette と色彩の凡例の colorbox の表示に関する属性 colorbox がある程度です:

image の属性

colorbox	palette

ここで先程の「相馬の古内裏」を利用した例を示しましょう.

```
(%i27) load("kodairi.mac")$
(%i28) D:read_xpm("KodairiS.xpm")$
(%i29) pic1:image(K,0,0,2,1)$
(%i30) pic2:image(D,0,0,2,1)$
(%i31) draw(gr2d(palette=gray,pic1),dimensions=[300,600]);
(%o31)                          [gr2d(image)]
(%i32) draw(gr2d(pic2),dimensions=[600,300]);
(%o32)                          [gr2d(image)]
```

この例では elevation_grid のときの画像行列を読み込んだものと, 同じ画像を XPM に変換したファイル[13]を read_xpm 函数を用いて読み込んだものの二種類の処理を行っています. 画像行列は輝度の強弱しかないために, ここでは palette を 'gray' としてグレースケールで表示させ, 画像ファイルを読込したものではそのまま表示させています:
直接読込を行ったものの図 11.7.10 は本来の画像 11.104 と同じ向きのカラー画像となっています. なお, R, G, B の各チャンネルの画像行列を一つに合せる方法や, XPM 函数の読込に関しての注意点については §11.8.1 にて詳細を述べることとします.

[13] ImageMagick の convert で変換し, 註釈行を削除したもの.

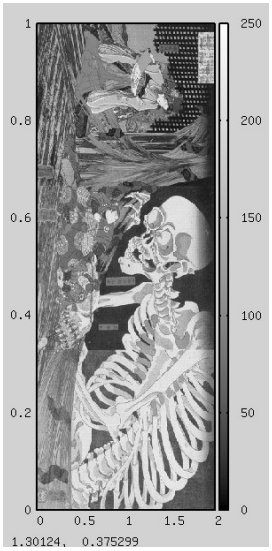


image の例 (その 1)

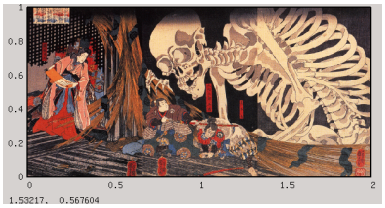


image の例 (その 2)

mesh 文

対象 mesh の生成に用います:

対象 mesh の構文

mesh([[x_{11}, y_{11}, z_{11}], ..., [x_{1n}, y_{1n}, z_{1n}]],
[[x_{21}, y_{21}, z_{21}], ..., [x_{2n}, y_{2n}, z_{2n}]],
...
[[x_{m1}, y_{m1}, z_{m1}], ..., [x_{mn}, y_{mn}, z_{mn}]])

mesh の属性

line_width	line_type	color	enhanced3d	wired_surface	transform

```
draw3d(enhanced3d=true, wired_surface=true,
       mesh([[0,0,0],[0,1,0],[0,1/3,1],[0,0,0]],
            [[1,0,0],[1,1,0],[1,1/3,1],[1,0,0]]), view=[45,45]);
```

parametric 文

parametric 文は平面曲線を表現する対象 parametric と空間曲線を表現する対象 parametric3d の生成で用います.

対象 parametric の構文

parametric	parametric($\langle X(t)\rangle, \langle Y(t)\rangle, \langle t\rangle, \langle t_0\rangle, \langle t_1\rangle$)
parametric3d	parametric($\langle X(t)\rangle, \langle Y(t)\rangle, \langle Z(t)\rangle, \langle t\rangle, \langle t_0\rangle, \langle t_1\rangle$)

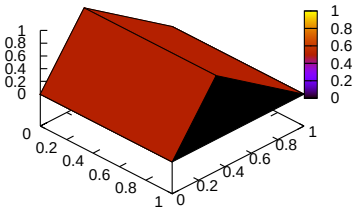


図 11.106: mesh の例

基本的に各座標の函数に続けて媒介変数とその区間を表記します. 対象 parametric の場合, 函数は X 座標と Y 座標の二つで変数は一つのみ, 対象 parametic3d の場合, 函数は X 座標, Y 座標, Z 座標の三つで, 媒介変数は曲線なので一つのみです. 媒介変数を用いた空間曲面は対象 parametric_surface を用います.

parametric の属性

nticks	line_width	line_type	color	enhanced3d	key

以降, 平面曲線と空間曲線の例を挙げておきます:

```
pc:parametric(11*cos(t)-6*cos(11/6*t),11*sin(t)-6*sin(11/6*t),t,0,12*%pi);
draw2d(nticks=1000,line_width=5,color=purple,pc,xaxis=true,yaxis=true,
       xaxis_type=solid,yaxis_type=solid,axis_left=false,axis_right=false,
       axis_bottom=false,axis_top=false,xtics_axis=true,ytics_axis=true);
```

この例は平面曲線の例です. 複雑な曲線の場合, 属性 nticks に大き目の数値を当てるとより滑らかな曲線が描けます. この例の曲線は nticks が既定値の 29 では全く足らず, ここでは 100 を指定します:

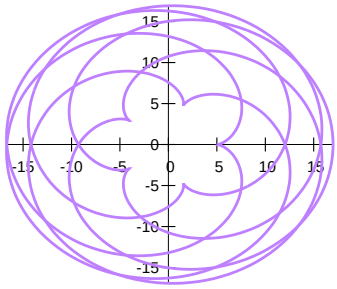


図 11.107: parametric の例 (平面曲線)

次に空間曲線の例を示します. この例ではトーラスとそのトーラス上の閉曲線を描いています. ここでトーラスは後述の parametric_surface, 閉曲線は parametric を用いて描いています:

```
ps:parametric_surface(cos(t)*(5+cos(s)),sin(t)*(5+cos(s)),
                      sin(s),t,0,2*%pi,s,0,2*%pi);
pc:parametric(cos(s/3)*(5+4*cos(s/2)),sin(s/3)*(5+4*cos(s/2)),
                      sin(s/2),s,0,12*%pi);
draw(gr3d(proportional_axes=xyz,surface_hide=true, ps,nticks=2000,
          line_width=5,color=red,pc));
```

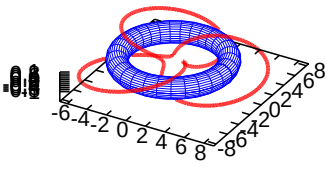


図 11.108: parametric の例 (空間曲線)

polar 文

polar 文は極座標表示された式のグラフを表現する対象 polar の定義で用います:

対象 polar の構文

polar($\langle f(\alpha)\rangle$, $\langle\alpha\rangle$, $\langle\alpha_0\rangle$, $\langle\alpha_1\rangle$)

ここで $\langle f(\alpha)\rangle$ は角度 α の函数で, 動径の長さに対応します. 角度 α の範囲は区間 $[\alpha_0, \alpha_1]$ で定められます.

対象 polar の属性

nticks	line_width	line_type	color	key

```
Archimedes:polar(theta,theta,0,12*%pi);
draw(gr2d(nticks=1000,color=red,line_width=5,Archimedes));
```

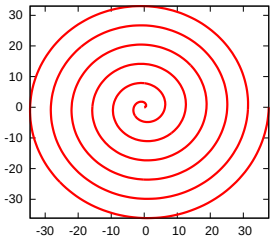


図 11.109: polar の例 (Archimedes の螺旋)

spherical 文

球面を定義する対象 spherical の生成で用います.

対象 spherical の構文

spherical(⟨R⟩, ⟨α⟩, ⟨α_0⟩, ⟨α_1⟩, ⟨β⟩, ⟨β_0⟩, ⟨β_1⟩)

spherical 文で定義可能なのは球面とその断面に限定されます.

spherical の属性

xu_grid	yv_grid_in	line_width	line_type	color
enhanced3d	wired_surface			

以下に簡単な例を示しておきます. 基本は半径と XZ, YZ 平面側での回転角度の指定です:

```
(%i47) draw3d(enhanced3d=[sqrt(x^2+y^2),x,y],wired_surface=true,
              spherical(1,a,0,%pi,b,%pi/2,2*%pi/3));
(%o47)                          [gr3d(spherical)]
```

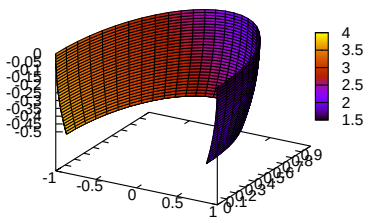


図 11.110: spherical の例

cylindrical 文

対象 cylindrical の生成で用います.

対象 cylindrical の構文

cylindrical(⟨R⟩, ⟨Z⟩, ⟨α⟩, ⟨α_0⟩, ⟨α_1⟩)

cylindricall 文で定義可能なのは円柱と軸に沿った断面に限定されます.

cylindrical の属性

xu_grid	yv_grid_in	line_width	line_type	color
enhanced3d	wired_surface			

以下に簡単な例を示しておきます. spherical と似た引数となっています:

```
(%i48) draw3d(enhanced3d=[sqrt(x^2+y^2),x,y],wired_surface=true,
              cylindrical(1,a,0,10,b,%pi/4,2*%pi/3));
(%o48)                          [gr3d(spherical)]
```

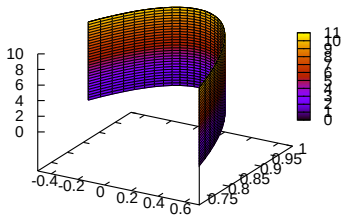


図 11.111: cylindrical の例

parametric_surface 文

parametric_surface 文は媒介変数表示された空間曲面を表現する対象 parametric_surface を定義する際に用います:

対象 parametic_surface の構文

parametric_surface(⟨X(s,t)⟩, ⟨Y(s,t)⟩, ⟨Z(s,t)⟩, ⟨s⟩, ⟨s_0⟩, ⟨s_1⟩, ⟨t⟩, ⟨t_0⟩, ⟨t_1⟩)

構文自体は parametric 文を拡張したものとなっています. つまり, 媒介変数による X,Y,Z 軸の函数と各媒介変数名とその範囲を引数として与えることになります.

parametric_surface の属性

xu_grid	yv_grid	line_width	line_type	color	key

```
draw3d(title= "Sea shell", xu_grid = 100, yv_grid = 25,
       view = [100,20], wired_surface=true,
       enhanced3d = true,palette = color,
       parametric_surface(0.5*u*cos(u)*(cos(v)+1),
                          0.5*u*sin(u)*(cos(v)+1),
                          u*sin(v) - ((u+3)/8*%pi)^2 - 20,
                          u, 0, 13*%pi, v, -%pi, %pi) )$
```

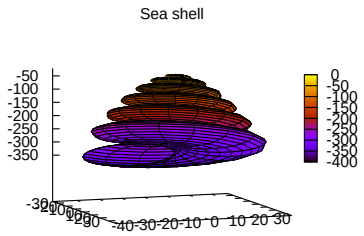


図 11.112: parametric_surface の例

```
ps1:parametric_surface(cos(t)*(5+cos(s)),sin(t)*(5+cos(s)),
                       sin(s),t,0,2*%pi,s,0,2*%pi);
ps2:parametric_surface(cos(t/3)*(5+4*cos(t/2))+1/2*cos(s),
                        sin(t/3)*(5+4*cos(t/2))+1/2*cos(s),
                       sin(t/2)+1/2*sin(s),t,0,12*%pi,s,0,2*%pi);
draw(gr3d(proportional_axes=xyz,surface_hide=true,
          enhanced3d=true,palette=color,ps1,ps2));
```

tube 文

対象 tube の生成に用います.

対象 tube の構文

tube ⟨X(*t*)⟩,⟨Y(*t*)⟩,⟨Z(*t*)⟩,⟨R(*t*)⟩,⟨*t*⟩,⟨t_0⟩,⟨t_1⟩)

tube は媒介変数 *t* を使って記述した座標と半径で描かれる曲面を表現します. tube には固有の属性として tube_extremes があります:

tube 固有の属性

属性	型	既定値	概要
tube_extremes	二成分の文字列リスト	[open,open]	両端を塞ぐかどうかを指定

この tube_extrems は対象 tube の両端をどのように処理するかを明示的に指示する属性です. この対象は二成分リストをその値として持ち, リストの成分の値は 'open' か 'close' のどちらかに限定

されます. たとえば, '[close,open]' が指定された場合, 媒介変数 $t \in [t_0, t_1]$ に対し, $t = t_0$ のときに 'close' のために蓋が描かれ, $t = t_1$ にて蓋が 'open' のために描かれません.

tube の属性

xu_grid	yv_grid	line_width	line_type	color
enhanced3d	wired_surface	surface_hide		

以下に簡単な例を示しておきます.

```
(%i9) draw3d(enhanced3d=true,wired_surface=true,
             xu_grid=50,yv_grid=50,
             tube(cos(4*t),t^2+sin(4*t),cos(t),t^2,t,0,2*%pi));
(%o9)                          [gr3d(tube)]
```

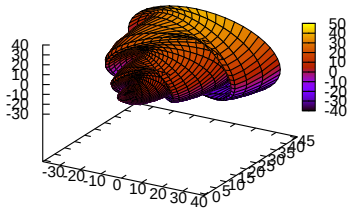


図 11.113: tube の例

11.7.11 その他の函数

その他の函数

add_zeroes(⟨ 整数 ⟩)

11.7.12 大域変数について

draw パッケージは MS-Windows 環境でも UNIX 環境でも利用可能です. ただし, MS-Windows 版では gnuplot ではなく, wgnuplot が同梱されていたり, ストリームを用いない方式となっているために, その他の環境との違いを吸収する際に, 次の大域変数を用いています.

環境の違いを調整する大域変数

変数名	既定値	概要
draw_pipes	true	Windows 環境の場合は false
draw_command	gnuplot	Windows 環境の場合は wgnuplot

大域変数 draw_pipes:　plot2d や plot3d 函数でも採用された Lisp のストリームを用いる方式であることを示します. 前述のように中間ファイルとして maxout.gnuplot_pipes か maxout.gnuplot の何れかが使えますが, 大域変数 draw_pipes が true であれば maxout.gnuplot_pipes を生成し, ストリームを用いて Maxima 側から gnuplot へ制御命令を送り込めます. しかし, 大域変数 draw_pipes が false の場合, maxout.gnuplot ファイルを生成するものの, この中間ファイルを用いる場合に Maxima は gnuplot への描画データだけではなく, 各種設定文や描画命令を含むファイル maxout_gnuplot を生成し, そのファイルを gnuplot に引渡す方式となっています. そのために Maxima 側からデータを生成したあとの面倒は一切見ません.
UNIX 環境の場合は true にも false にも出来ます. ただし, false にするとマウスを使った図形の拡大・縮小, 三次元グラフの回転といったマウス操作が出来ません. さらに, グラフ画面に m と入力しても gnuplot の mouse が on になりません. その代わりに別途,gnuplot を立ち上げて maxout.gnuplot ファイルを読込んで, それから mouse を on にすればマウスによる操作が行えます. mouse の on/off はグラフ画面で m と入力することで行えます.
なお, UNIX 環境でこの変数の値が true の場合, マウスによるグラフの操作も, gnuplot の mouse の 'on' と 'off' の切替もできます.
この変数の具体的な利用方法としては, グラフ描画の設定文を取り込んだ maxout.gnuplot ファイルを出力するために使えます. この maxout.gnuplot ファイルは非常に便利なファイルで, このファイルさえあればファイルを編集するだけで, あとは gnuplot 単体で色々な処理が行えるからです.
次の draw_command は MS-Windows では wgnuplot, その他の環境では gnuplot を用いる為に用いる大域変数です. 独自の gnuplot を使いたい場合に, この変数にアプリケーション名を指定しておきます. そのような特殊な大域変数のために, この変数を弄る必要はまずないでしょう.

multiplot_mode 函数:　一つの画面に複数のグラフ表示を行うための設定を行う函数です. この函数は MS-Windows 環境では使えません.

add_zeros 函数:　E 表現の浮動小数点数を 10 桁以下の小数点表示に変換する函数です.

draw_file 函数：　表示画像をファイルに保存する函数です. 保存ファイルの書式は terminal 属性に指定します. また, 画像ファイルの大きさは dimensions 属性, ファイル名は file_name 属性, 画像の背景色は background_bgcolor 属性で指定します.

active_window 函数:　描画ウィンドウを活性化するための函数です. 通常は描画の都度, 新しいウィンドウを生成し, そのウィンドウが活性化していますが, active_window でウィンドウ番号を直に指定することで, 指定したウィンドウの活性化が行われます.

11.8 drawutils パッケージ

drawutils パッケージは draw パッケージを必要とするパッケージで, load(drawutils) を実行すると必要に応じて draw パッケージの読込を行います. このパッケージは 2 次元ベクトル場の描画を行う plot_vector_field 函数と 3 次元ベクトル場の描画を行う plot_vector_field3d の二つが収録されています:

drawutils パッケージの函数の構文

plot_vector_field([⟨ 函数$_x$⟩, ⟨ 函数$_y$⟩], ⟨ リスト$_x$⟩, ⟨ リスト$_y$⟩)
plot_vector_field([⟨ 函数$_x$⟩, ⟨ 函数$_y$⟩], ⟨ リスト$_x$⟩, ⟨ リスト$_y$⟩, ⟨ 属性$_1$⟩, ...)
plot_vector_field3d([⟨ 函数$_x$⟩, ⟨ 函数$_y$⟩, ⟨ 函数$_z$⟩], ⟨ リスト$_x$⟩, ⟨ リスト$_y$⟩, ⟨ リスト$_z$⟩)
plot_vector_field3d([⟨ 函数$_x$⟩, ⟨ 函数$_y$⟩, ⟨ 函数$_z$⟩], ⟨ リスト$_x$⟩, ⟨ リスト$_y$⟩, ⟨ リスト$_z$⟩, ⟨ 属性$_1$⟩, ...)

ここで ⟨ リスト$_i$⟩ の書式は [⟨ 変数 ⟩, ⟨ 最小値 ⟩, ⟨ 最大値 ⟩] の書式の 3 成分のリストです. ここでの変数は函数リスト [⟨ 函数$_x$⟩, ⟨ 函数$_y$⟩] あるいは [⟨ 函数$_x$⟩, ⟨ 函数$_y$⟩, ⟨ 函数$_z$⟩] の変数と適合していなければなりません. また, X-軸, Y-軸, Z-軸のラベルは ⟨ リスト$_x$⟩, ⟨ リスト$_y$⟩, ⟨ リスト$_z$⟩ の第一成分が割り当てられます.

plot_vector_field: 2 次元のベクトル場を描く函数です.

```
(%i129) plot_vector_field([y,x],[x,-10,10],[y,-10,10],
        terminal=wxt,head_angle=45,color=red);
```

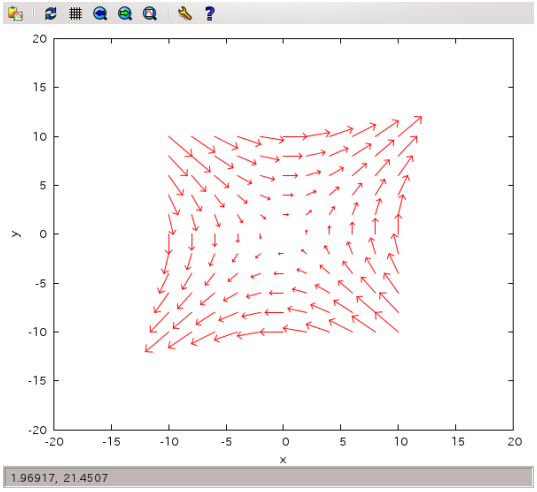


図 11.114: plot_vector_field の例

plot_vector_field3d:　3 次元のベクトル場を描く函数です.

```
(%i137) plot_vector_field3d([y,x,z],[x,-10,10],[y,-10,10],[z,-10,10],
        terminal=wxt,head_angle=45,head_type=filled ,color=red);
```

ここでの例では, 属性 head_angle を 45, head_type を filled とし, ベクトル場の色を赤としています.

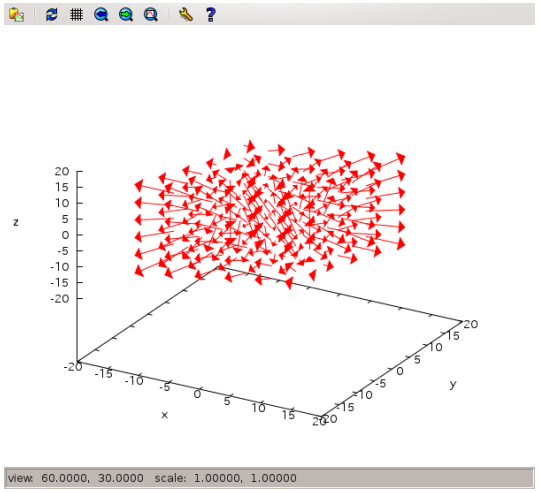


図 11.115: plot_vector_field3d の例

11.8.1　picture パッケージ

picture パッケージには画像与件を格納するための対象である picture の定義や画像を処理するための函数が定義されています. ただし, picture.lisp の註釈にもあるように不安定です.

対象 picture

対象 picture は画像与件を格納することを目的とした対象で, その性格上,　level-picture と rgb-picture の二種類があります. これらの picture の構造を以下にまとめておきます:

種類	対象 picture の構造
level-picture	picture(level,⟨W⟩,⟨H⟩,⟨M⟩)
rgb-picture	picture(rgb,⟨W⟩,⟨H⟩,⟨M⟩)

picture 文の第 1 の成分には表現する画像が何であるかを意味します. level-picture と rgb-picture への分類はこの第 1 成分で区分したもので, 'level' で picture の格納する画像がカラーではなく単

色であることを示し, ここが 'rgb7 であれば picture の格納する画像がカラー画像であることを意味します.
第 2 成分と第 3 成分の ⟨W⟩, ⟨H⟩ が表現する画像の幅と高さの画素数を意味し, 第 4 成分の ⟨M⟩ が画像本体の与件となります. この部分は Maxima の配列として表現され, その成分の値は 0 から 255 までの整数値を取ります.
level-picture の生成は, リストや行列から make_level_picture 函数で生成し, rgb-picture の生成は赤, 緑, 青の各チャンネルに対応する 3 個の level-picture から make_rgb_picture 函数で生成するか, read_xpm 函数で直接, 画像ファイルの読込で行います.

ここで picture パッケージに含まれる函数を紹介します:

picture に関連する函数

make_level_picture(⟨M⟩)
make_level_picture(⟨M⟩, ⟨W⟩, ⟨H⟩)
pciturep(⟨X⟩)
picture_equalp(⟨X⟩, ⟨Y⟩)
make_rgb_picture(⟨R⟩, ⟨G⟩, ⟨B⟩)
take_channel(⟨P⟩, ⟨color⟩)
negative_picture(⟨P⟩)
rgb2lebel(⟨P⟩)
get_pixel(⟨P⟩, ⟨X⟩, ⟨Y⟩)
read_xpm(⟨file⟩)

make_level_picture 函数: Maxima のリストや行列から対象 picture の level-picture を生成する函数です. ここで生成する対象 picture の第 1 成分は level になります:

```
(%i42) a:make_level_picture([1,2,3],3,1);
(%o42)              picture(level, 3, 1, {Lisp Array: #(1 2 3)})
(%i43) b:make_level_picture(matrix([1,2,3],[3,2,1]),3,2);
(%o43)          picture(level, 3, 2, {Lisp Array: #(1 2 3 3 2 1)})
```

picturep 函数: 与えられた対象が picture であるかどうかを判断する函数で, 'true' の場合, 引数として与えられた対象が picture であることになります.

picture_equalp 函数: 二つの対象 picture が等しいかどうかを判断する函数です. 等しければ 'true' を返却します.

make_rgb_picture 函数: 赤, 青, 緑の各チャンネルに対応する level-picture から一つの RGB を持った rgb-picture を生成する函数です. ここでは Yorick を用いて各チャンネルに対応する Maxima の行列を次のスクリプトで生成します:

```
A=img_read("KodairiS.jpeg");
B=long(A);
C=["K_R","K_G","K_B"];
sz=dimsof(B);
f=open("kodairi.mac","w");
for(k=1;k<4;k++){
    write,f,C(k)+":matrix(";
    for(i=1;i<=sz(3);i++){
        if(i!=sz(3))px=",";else px="";
        write,f,format="[%s","";
        for(j=1;j<sz(4);j++) write,f,format="%d,",B(k,i,j);
        write,f,format="%d]%s\n",B(k,i,sz(4)),px;};
    write,f,format="%s);\n","";};
close,f;
```

次に, Maxima で行列を読み込んでそれらを make_level_picture 函数を用いて level-picture に変換し, 最後に make_rgb_picture で一つの対象 picture を生成しましょう:

```
(%i1) load("kodairi.mac")$
(%i2) P_R:make_level_picture(K_R)$
(%i3) P_G:make_level_picture(K_G)$
(%i4) P_B:make_level_picture(K_B)$
(%i5) P:make_rgb_picture(P_R,P_G,P_B)$
(%i6) draw(gr2d(image(P,0,0,1,2)),dimensions=[300,600])$
(%i7) draw(gr2d(image(P_R,0,0,1,2),palette=[3,0,0]),dimensions=[300,600])$
(%i8) draw(gr2d(image(P_G,0,0,1,2),palette=[0,3,0]),dimensions=[300,600])$
(%i9) draw(gr2d(image(P_B,0,0,1,2),palette=[0,0,3]),dimensions=[300,600])$
```

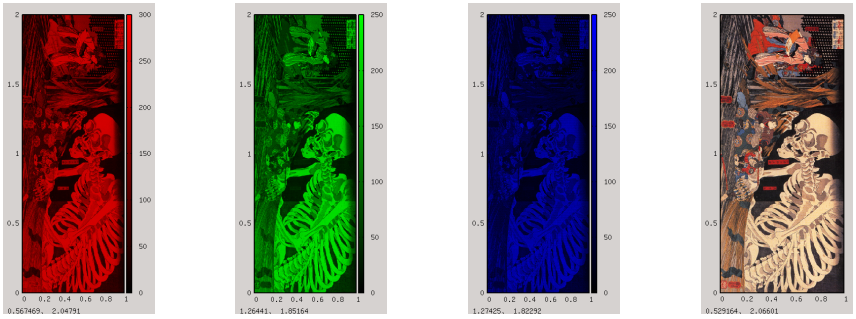


図 11.116: 一連の操作で生成した絵

take_channel 函数:　make_rgb_picture 函数の逆の操作を行う函数で, RGB の picture を赤, 青, 緑の輝度の picture に分解します.

```
(%i11) A1:take_channel(P,red);
(%o11) picture(level, 289, 600, {Lisp Array: #(42 95 105 125 52 33 40 40 33 32\
 32 35 40 126 37 95 42 36 57 78
                34 34 46 57 47 41 67 151 115 109 55 45 84 171 121 71 72 41 42 5\
4
                33 34 114 109 74 112 55 44 33 94 89 126 114 138 36 46 41 123 50
                101 142 176 183 45 110 53 32 33 131 123 193 58 38 38 33 30 74
                130 165 98 23 22 43 24 27 26 56 131 138 148 102 31 34 28 23 35
                38 122 160 158 ...)})
(%i12) picture_equalp(P_R,A1);
(%o13)                                  true
```

negative_picture 函数: 輝度を反転したネガ画像を生成する函数です.

```
(%i32) P1:read_xpm("KodairiS.xpm")$
(%i35) draw(gr2d(image(negative_picture(P1),0,0,2,1)),dimensions=[600,200])$
```

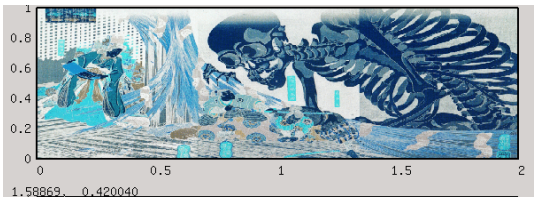


図 11.117: negative_picture 函数の例

rgb2level 函数: rgb-picture に対し, その各 RGB の平均値で構成した level-picture を生成します. 要するに対象 elevation_grid で Yorick を用いてカラー画像行列を輝度のみの画像行列に変換しましたが, それと同じことを処理する函数です.

get_pixel 函数: picture から指定した画素の値を取り出します:

```
(%i27) get_pixel(P,100,100);
(%o27)                          [236, 145, 126]
(%i28) get_pixel(P_R,100,100);
(%o28)                                  236
(%i29) get_pixel(P_G,100,100);
(%o29)                                  145
(%i30) get_pixel(P_B,100,100);
(%o30)                                  126
```

read_xpm 函数: XPM 形式のファイルを読み込みます. この函数の処理手順は非常に安易なもので, 最初の二行をコメント等として跳ばし, 3 行目からを与件本体として make-hash-table で hash-table として取込むものです. しかし, この処理には非常に大きな問題があります. 最初に XPM の基本的な書式を以下に示しておきます:

```
1 static char * 〈 ファイル名 (修飾子無し)〉[] = {
2 "〈 幅 〉〈 高さ 〉〈 色数 〉〈 文字のビット数 〉",
3 "〈 文字1〉c 〈 色情報1〉",
4 ...
5 "〈 文字〈 色数 〉〉c 〈 色情報〈 色数 〉〉",
6 "長さが〈 文字のビット数 〉×〈 幅 〉個の文字の列の文字列"
7 "..."
```

このように本文の頭に C の配列として名前が定義され, 以下に画像の具体的な情報が文字列として並びます. その情報として先頭に画像の大きさと色数に関する情報行, それから文字とその文字が表現する色の情報が画像で実際に用いられる色数分並び, 最後に画像の上から前に定義した画像幅に相当する個数の文字の羅列で構成された二重引用符で括られた文字列が画像の高さ分並びます. ここで文字の羅列の各文字が画像の画素に対応し, その文字の色情報が対応する画素の色となる訳です. そして文字の大きさは"文字のビット数"で制御されています. ここで XPM 形式の註釈行は C と同様に '/*' で開始し '*/' で終了し, その性格上, 何処にあっても構いませんが, 先程の配列の定義の上, その下と実際の文字の羅列の開始前に置かれています.
たとえば, ImageMagick の convert で変換した KodairiS.xpm の中身を見てみましょう:

```
1 /* XPM */
2 static char *KodairiS[] = {
3 /* columns rows colors chars-per-pixel */
4 "600 289 256 2 ",
5 "    c #0D0A0B",
6 ".   c #0C0708",
```

この例では, 画像は 600x289 画素の画層で, 全体の色数は 256 個なので色の情報は 256 行記述され, 次に文字のビット数が 2 バイトなので, 文字の羅列で用いられる画素に対応する文字は 'X_1X_2' と二文字で表現され, 画像を表現する文字列の長さは 2×289 であることが判ります. この画像の情報に続いて, 文字と対応する色の情報が続きます. この文字と色の情報は色数分あり, それが終ると画像を表現する文字列が続きます. この文字と色の情報が終了する時点で"/* pixels */" という註釈行が入ります:

```
259 "IX c #F6E0C3",
260 "UX c #7B8079",
261 /* pixels */
262 "= = * - - * - & O ,  .....
```

ここで read_xpm 函数の処理は, 最初の二行を註釈と C 風の配列定義と見做し, 実体は 3 行目から開始すると想定した手続となっています. しかし, convert が出力する画像は上記のように註釈行が入るものの, そのような行の排除を行わない手続のために読込に失敗するというものです. この問題に対してはフィルターを通すなりの修正を加えれば良いのですが, ここでは安易に XPM ファイ

ル側の註釈の削除という方法を用いています[14]

11.9 世界地図 (worldmap パッケージ)

11.9.1 世界地図の概要

Maxima には世界地図のデータが収録されています. 世界地図を利用するためには予め load(worldmap) で関連する函数やデータの読込を行う必要があります. この worldmap パッケージの読込で, 実際のデータと対象が定義されている wbd.lisp の読込も同時に行なわれます. この世界地図で利用されているデータは 'World Boundary Databank' の 3 次元データで, Maxima に worldmap パッケージの読込を行った時点で配列 boundary_array に収録されています.
世界地図の描画は draw2d や draw3d 函数を用いる場合, 国名, 大陸名, 海岸線名等のリストから geomap 対象を構成すれば描画が行えます:

```
(%i95) draw3d(terminal=wxt,geomap([Japan,Taiwan,Russia]))$
(%i96) draw2d(terminal=wxt,geomap([North_America, European_Union]));
```

この例では最初に日本, 台湾, ロシアの境界線を 3 次元で表示し, 次の例では北アメリカと EU 加盟国を 2 次元で表示しています.

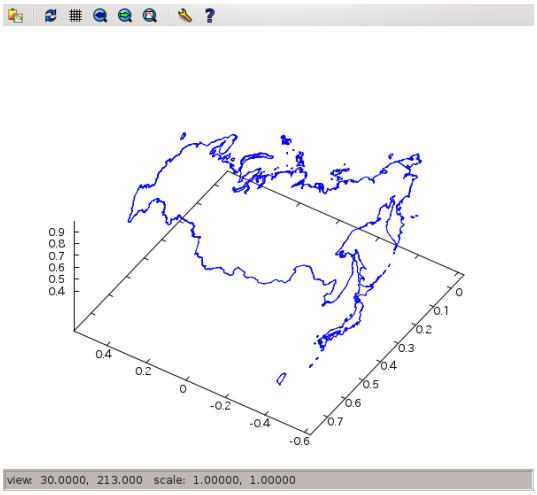


図 11.118: 3 次元表示 (日本, 台湾, ロシア)

11.9.2 worldmap パッケージの函数

worldmap パッケージの Maxima 函数の構文を次に示しておきます:

[14] ImageMagick の convert には '-strip' オプションで余計な註釈行を外すことができます. ただし, ここでの例に挙げた二箇所の註釈行を挿入して他の不要な註釈行を外すため, read_xpm 函数には不十分です.

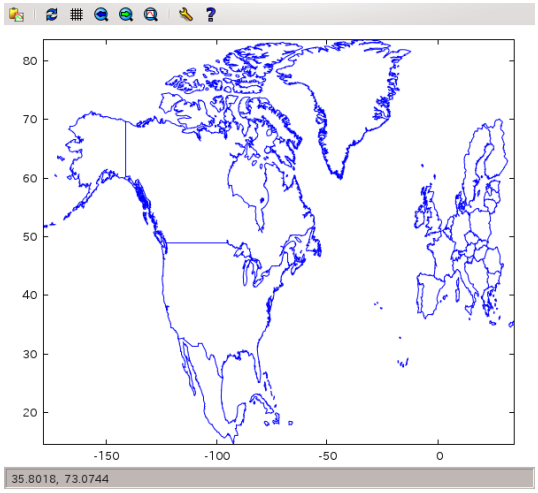


図 11.119: 2 次元表示 (北アメリカと EU)

worldmap の函数の構文

list_to_pairs(⟨ リスト ⟩)
numbered_boundaries(⟨ 整数リスト ⟩)
region_boundaries_plus
region_boundaries
make_polygon
make_poly_country
make_poly_continent
extr_coord
coastlines

list_to_pairs:　偶数成分のリストの各 $2i-1$ 成分と $2i$ 成分で構成される対リストを成分とするリストを生成します. なおリストが奇数成分であればエラーを返します.

```
(%i21) list_to_pairs([1,2,3,4,5,6]);
(%o21)                    [[1, 2], [3, 4], [5, 6]]
(%i22) list_to_pairs([1,2,3,4,a,b]);
(%o22)                    [[1, 2], [3, 4], [a, b]]
(%i23) list_to_pairs([1,2,3,4,[a,b],c]);
(%o23)                  [[1, 2], [3, 4], [[a, b], c]]
```

numbered_boundraries:　境界を描く函数です.

region_boundraries:

make_polygon: 国の境界線データを生成する函数です. draw パッケージの対象 polygon を生成します.

make_poly_country: 国の境界線データ [を生成する函数です. 引数は国名, あるいは国名を成分とするリストになります. なお、 この函数は make_polygon 函数を map 函数で作用させる函数として定義されています.

```
(%i44) draw2d(make_poly_country(Japan));
```

make_poly_continent: 大陸や国の境界線データを生成する函数です. 引数が大陸名の場合は大陸とその大陸に存在する国の国境が描かれます. 国名のリストを与えた場合, リストに含まれる国の境界線を描きます. この函数は map 函数を使って make_poly_country 函数を引数のリストに作用させ, flatten 函数でリストを平坦にする函数として定義されています.

extr_coord:

coastlines: aaa

海岸線

Africa_coastline	Africa, Lesotho
America_coastlines	North_America,Central_America,South_America
Antarctica_ coastlines	Antarctica
Eurasia_coastlines	Europe, San_Marion, Asia
Oceania_coastlines	Oceania
World_coastlines	America_coastlines, Eurasia_coastlines, Africa_coastlines,Oceania_coastlines, Antarctica_coastlines

大陸

Africa	Asia	Europe	Oceania
North_America	Central_America	South_America	

国名 (Africa)

Burundi	Comoros	Djibouti	Ethiopia
Kenya	Madagascar	Malawi	Mauritius
Mozambique	Reunion	Rwanda	Seychelles
Somalia	Tanzania	Uganda	Zambia
Zimbabwe	Angola	Cameroon	Central_African_Republic
Chad	Congo_Republic	Equatorial_Guinea	Gabon
Algeria	Egypt	Libya	Morocco
Sudan	Tunisia	Western_Sahara	Botswana
Lesotho	Namibia	South_Africa	Swaziland
Benin	Burkina	Cabo_Verde	Cote_Ivoire
Gambia	Ghana	Guinea	Guinea_Bissau
Liberia	Mali	Mauritania	Niger
Nigeria	Senegal	Sierra_Leone	Togo
Zaire	Sao_Tome_and_Principe		

国名 (Asia)

azakhstan	Kyrgyzstan	Tajikistan	Turkmenistan
Uzbekistan	Japan	Taiwan	Mongolia
North_Korea	South_Korea	China	Russia
Cambodia	Laos	Malaysia	Myanmar
Philippines	Singapore	Brunei	Thailand
Vietnam	Afghanistan	Bangladesh	Bhutan
India	Iran	Maldives	Nepal
Pakistan	Sri_Lanka	Armenia	Azerbaijan
Bahrain	Palestinian_Authonomy	Georgia	Iraq
Israel	Jordan	Kuwait	Lebanon
Oman	Qatar	Saudi_Arabia	Syria
Turkey	United_Arab_Emirates	Yemen	

国名 (Europe)

Austria	Belgium	Bulgaria	Cyprus
Czech_Republic	Denmark	Estonia	Finland
France	Germany	Greece	Hungary
Ireland	Italy	Latvia	Lithuania
Luxembourg	Malta	Netherlands	Poland
Portugal	Romania	Slovakia	Slovenia
Spain	Sweden	United_Kingdom	Belarus
Moldova	Ukraine	Iceland	Norway
Albania	Andorra	Bosnia_and_Herzegovina	Croatia
Macedonia	Serbia	Liechtenstein	Monaco
Switzerland	San_Marino		

国名 (European_Union)

Austria	Belgium	Bulgaria	Cyprus
Czech_Republic	Denmark	Estonia	Finland
France	Germany	Greece	Hungary
Ireland	Italy	Latvia	Lithuania
Luxembourg	Malta	Netherlands	Poland
Portugal	Romania	Slovakia	Slovenia
Spain	Sweden	United_Kingdom	

国名 (North_America)

Greenland	Canada	United_States	Mexico	Cuba
Bahamas	Jamaica	Puerto_Rico	Haiti	

国名 (Central_America)

Belize	Costa_Rica	El_Salvador	Guatemala
Honduras	Nicaragua	Panama	Trinidad_and_Tobago
Grenada	Barbados	Martinique	Santa_Lucia
St_Vincent_and_Grenadines	Dominica	Guadalupe	Antigua_and_Barbuda
St_Kitts_and_Nevis			

国名 (South_America)

Argentina	Bolivia	Brazil	Chile	Colombia
Ecuador	Malvinas	French_Guiana	Guyana	Paraguay
Peru	Suriname	Uruguay	Venezuela	

国名 (Oceania)

Australia	New_Zealand	Fiji	Papua_New_Guinea
Solomon_Islands	Indonesia		

第12章　積分函数の動きを観察しよう

この章では非常に簡単ですが,Maxima の積分処理の流れに関して実際に Maxima がどの様に解釈し, 処理を行っているのか実例を示したいと思います. その為, 多少の LISP の知識 (この本の LISP についての内容) があれば良いでしょう. 計算機で数式を扱う事は一見人工知能に関係しそうですが, 実際は数値計算と同様に機械的な処理である事がなんとなく理解出来るかと思います.

12.1　積分函数ツアー募集要項

この章では Maxima の処理の流れを調べてみましょう. そこで, 何かと問題のある integrate 函数の動作を眺めてみましょう. 流石に, integrate 函数は色々と混み入っているので, 私が旗を持って観光旅行風に案内しましょう.
なお, このツアーに必要なことを以下に列記しておきましょう:

積分函数ツアー募集要項

- car,cdr,cons,append 等のリスト操作の函数が理解出来る.
- if や cond が理解出来る.
- lambda 函数が理解出来る.

等々と列記していますが, 一番重要なものはなによりも「**好奇心**」です.

12.2　Maxima のソースファイル

Maxima のソースファイルは, KNOPPIX/Math 2010 であれば “/usr/share/maxima/5.17.1/src” にあります. Windows 版の Maxima-5.22 を Maxima を “C;\Program Files” にインストールしていれば, この “Program Files” フォルダにある Maxima-5.22 フォルダ中の “share\maxima\5.22” にある src フォルダに収録されています. この様に Maxima はバージョン番号のディレクトリ/フォルダの中に src ファイルが置かれています.
Maxima は Common Lisp で記述されてるので, ソースファイルは LISP のプログラムファイルです. このファイルは通常のテキストエディタで開いて閲覧できます. なお, 用心のために, ここでの作業では必ずソースファイルの複製を作って複製の方を書換えて下さい.

12.3　integrate 函数

Maxima の integrate 函数をこれから調べるのですが, src ディレクトリには沢山のファイルがあり, どのファイルに integrate 函数が定義されているか判らないでしょう. そこで, ファイル内部の検索を行います. このときに Linux や Cygwin 等の UNIX 環境であれば grep 命令が使えます. この命令は指定した文字列が含まれる行をファイル名と一緒に表示できます. ここで, 検索する文字列をどうするかという問題があります. というのも, Maxima の函数や変数は Maxima 内部では頭に記号 “$” が付いているので, integrate 函数の実体は “$integrate” 函数なのです. ところで, Maxima 側で直接操作しない函数や変数の頭に記号 “$” を付ける必要はありません.
そこで, `grep \$integrate *.lisp` と入力しましょう. すると文字列 “$integrate” を含む行と修飾子が.lisp のファイル名と一緒に示されます. そして, その中に以下の表示がある筈です.

```
simp.lisp:(defmfun $integrate nargs
```

この grep では, 記号 ":" の左側にファイル名, ここでは右側に文字列 "$integrate" を含むファイル内の行が表示されています. ここでの "defmfun" は Common Lisp にない函数ですが, その名前から函数定義の命令と推測できます. 事実, "$integrate" が含まれる他の行には, この様な函数定義らしいものはありません. そこで, simp.lisp の該当個所を抜出したものを以下に示しておきましょう:

```
3064 (defmfun $integrate (expr x &optional lo hi)
3065   (let ($ratfac)
3066     (if (not hi)
3067         (with-new-context (context)
3068           (if (member '%risch nounl :test #'eq)
3069               (rischint expr x)
3070               (sinint expr x)))
3071         ($defint expr x lo hi))))
```

先程述べた様に Maxima 内部では Maxima の函数や変数には先頭に記号 "$" が付いています. Maxima で処理させるときに記号 "$" を全く考える必要はありませんが, LISP 側から処理するときには, このことを忘れてはいけません.

ここで先頭が "defmfun" となっていますが, これは Maxima の函数定義に用いるマクロで, Common Lisp の defun 函数と類似のものと考えて構いません. このマクロは Macsyma が記述された LISP(MacLisp) の構文に合せる目的で用いられており, Maxima には他にもこのようなマクロが多くあります.

この$integrate 函数の処理は上のリストを見ても判るように if 函数による場合分け以上のことをしていません. この if 函数は '(if(述語)(函数$_1$)(函数$_2$))' の形で, 述語 が 'T であれば, 函数$_1$ を実行し, そうでなければ 函数$_2$ を実行するというものです. ここでは述語が '(not hi)' なので, オプションの lo と hi のうち, hi が存在しなければ, 不定積分を行い, そうでない場合には defint 函数による定積分を行うことが判ります. さて, ここで, '(not hi)' が 'T の場合ですが, この場合は, with-new-context 函数を使うことが判ります. この函数の第 1 引数の (context) は名前から文脈であることが予想できますね. すると, 第 3 の引数は, その文脈を使って整理すべき式であることが判りますが, その式で, nounl に '%rish という表徴があれば rischint 函数を実行し, そうでなければ sinint 函数を実行していることが判りますね.

ざっと, ここでは if 函数しか明瞭に判っていませんが, それだけでも, 函数名からおおよその処理が読めた訳です.

ではもう少し詳しく内容を吟味してみましょう. ここでは積分に興味があるので, $integrate 函数が rischint と sinint のどちらの内部函数を用いるかを調べます. ここで, リスト nounl に '%risch が含まれている場合に Risch 積分を行う rischint 函数に引数が引渡されますね. それ以外は, sinint で積分を行います. さて, integrate 函数で Risch の積分を行うためには, リスト nounl に '%risch が含まれていなければなりませんね. では何処で nounl の設定が行われているのしょうか? このことを調べてみましょう. ここで nounl の設定個所の捜し方ですが, KNOPPIX や UNIX 環境があれば, src ディレクトリで grep nounl *.lisp と入力してみましょう. そうすると, 色々出てきますが, 変数に割当を行う setq 函数や setf 函数がある行に注目して下さい. すると以下の行があります:

```
mlisp.lisp:     (setq nounl (cons ($nounify fl) nounl)))
```

ここで左側がファイル名で, 右側に検索する文字列があることを示しています. このことから mlisp.lisp に文字列に nounl を設定する行があるということが判ります. そこで, ファイル mlisp.lisp

を開いて該当個所を見ると, 実は, ev 函数の本体の処理であることが判ります. この ev 函数の定義は長いのでここでは示しませんが, ev 函数の詳細は 5.8.3 を参照して下さい.
ここで ev 函数での nounl の処理の意味は, Maxima の nounify 函数 (内部では$notify) を用いて函数名を名詞化した結果を LISP の cons 函数で nounl リストに追加することです. したがって, ev 函数によって nounl に指定した名詞化された函数名が cons されることが判ります. だから, integrate 函数を ev 函数を用いて評価する際に, %risch を指定すれば integrate 函数は Risch 積分を実行すると判断できるのです.
そこで, 実際に動作を確認してみましょう. ここでの確認は単純に LISP の print 函数を$integrate 函数に埋め込んでみましょう. 最初に src ディレクトリから simp.lisp の複製を適当なディレクトリに置きます. それから simp.lisp の$integrate 函数の定義の個所に print 函数を二箇所に埋め込みます. ここで表示させるのは nounl の内容としましょう. 要するに, '(print nounl)' を追加すれば良いのです.
以下に改造した$integrate 函数を示します.

```
(defmfun $integrate (expr x &optional lo hi)
  (let ($ratfac)
  (print nounl)
  (if (not hi)
        (with-new-context (context)
          (if (member '%risch nounl :test #'eq)
              (and(print nounl)(rischint expr x))
              (sinint expr x)))
        ($defint expr x lo hi))))
```

ここで, 函数の変更を Maxima に反映させるためには幾つかの処理が必要です. 最も大袈裟な方法は Maxima をコンパイルして Maxima をインストールすることです. もし, Maxima が C 等のコンパイラで記述されていればこうするしか方法はないでしょう. しかし, いくら何でもこの方法は幾ら何でも大袈裟です. ところが, 対話的処理が可能な Common Lisp で記述された Maxima ではもっと簡単に済ます方法があります.
一番簡単な方法は ,Maxima を立ち上げて修正した simp.lisp を load 函数で読込む方法です.
もう一つの安易ですが, 場合によっては面倒な方法は, Maxima で to_lisp(); と入力して Maxima の裏の LISP を表に出して上記の$integrate 函数のリストを全て入力し, (to-maxima) と入力して Maxima に戻る方法です. こちらは Maxima で全て済ませることができる点が長所ですが, 長 いプログラムの打ち込みを行うのであればファイルの読込の方が楽です.
ここでは simp.lisp を書換えて, load 函数を使って読込み, それから ev(integrate(3^log(x),x),'risch); と入力して Risch 積分が行われる事を確認してみましょう.

```
(%i8) load("./simp.lisp");
(%o8)                          ./simp.lisp
(%i9) integrate(3^log(x),x);
```

```
NIL                                    1
                                (------ + 1) log(x)
                                 log(3)
                               3
(%o9)                          ---------------------
                                   1
                               (------ + 1) log(3)
                                log(3)
(%i10) ev(integrate(3^log(x),x),'risch);

(%INTEGRATE %RISCH)
(%INTEGRATE %RISCH)                                    log(3) log(x)
                                 x %e
(%o10)                           -----------------
                                    log(3) + 1
```

ev 函数を利用しない場合は nounl 変数が既定値の 'NIL のために sinint 函数が実行されていることが判ります. そこで, ev 函数で'risch を指定すると nounl にリスト '(%INTEGRATE %RISCH)' が割当てられ, nounl に'%RISCH が含まれているので, (memberq '%risch nounl test #'eq)) が 'T となるので rischint が実行されたのです.
ここでは内部変数の動きを見るために integrate 函数を改造しましたが, この調子で函数の修正や改造が Maxima では簡単にできてしまうのです.
では, もう一方の sinint 函数を調べてみましょう. この sinint 函数は sin.lisp の中で定義されています. 該当個所を以下に示します:

```
;; This is the top level of the integrator
(defmfun sinint (exp var)
  ;; *integrator-level* is a recursion counter for INTEGRATOR.  See
  ;; INTEGRATOR for more details.  Initialize it here.
  (let ((*integrator-level* 0))
    (declare (special *integrator-level*))
    (cond ((mnump var) (merror "Attempt to integrate wrt a number: ~:M" var))
          (($ratp var) (sinint exp (ratdisrep var)))
          (($ratp exp) (sinint (ratdisrep exp) var))
          ((mxorlistp exp)    ;; if exp is an mlist or matrix
           (cons (car exp)
                 (mapcar #'(lambda (y) (sinint y var)) (cdr exp))))
          ;; if exp is an equality, integrate both sides
          ;; and add an integration constant
          ((mequalp exp)
           (list (car exp) (sinint (cadr exp) var)
                 (add (sinint (caddr exp) var)
              ($concat $integration_constant (incf $integration_constant_counter
                    )))))
          ((and (atom var)
                (isinop exp var))
           (list '(%integrate) exp var))
          ((let ((ans (simplify
                       (let ($opsubst varlist genvar stack)
                         (integrator exp var)))))
             (if (sum-of-intsp ans)
                 (list '(%integrate) exp var)
```

```
                ans))))))
```

この函数は cond 函数による分岐を持った函数です. 先ず, 最初の mnump 函数は変数 var が数値の場合に 'true' を返す Maxima 内部の真偽函数です. この個所は変数が数値の場合のエラー処理を行う個所です. 実際, 以下の処理を行います:

```
(%i1) integrate(sin(x),3);
Attempt to integrate wrt a number: 3
-- an error. To debug this try: debugmode(true);
```

ここで, "Attemp to integrate wrt a number: 3" という文言との対応が付きますね. さて, それに続く 2 つの$ratp 函数で変数 var と式 exp が CRE 表現の場合の処理を定めます. $ratp 函数は Maxima の ratp 函数です. この函数は引数が CRE 表現であれば 'true' を返す Maxima の真理函数です.

そして mxorlistp 函数は式 exp がリストか行列の場合に 'NIL にならない函数です. ここでの処理では各成分に sinint 函数を作用させます. この個所で lambda 函数を用いて変数 y に対して '(sinint y var)' を対応させる無名函数を定め, その函数を式 exp の内部表現の頭の部分を除いた行列やリストの各成分に mapcar 函数を使って作用させています. この処理の御陰で, 次に示すようにリストや行列で与えても計算が行えるのです:

```
(%i2) integrate([sin(x),cos(x),exp(x)],x);
                                                  x
(%o2)                          [- cos(x), sin(x), %e ]

(%i3) integrate(matrix([sin(x),cos(x),exp(x)],[tan(x),1/x,log(x)]),x);
                    [                                  x        ]
(%o3)               [  - cos(x)     sin(x)        %e          ]
                    [                                          ]
                    [ log(sec(x))   log(x)   x log(x) - x      ]
```

mequalp 函数は式 exp に二項関係子 "=" が含まれた場合に 'T となる函数です. ここでの処理は, 式 exp の右辺と左辺に sinint を作用させて右辺に積分定数を加える処理を行っています. この積分定数の処理で用いられている concat 函数は変数 integrationconstant に大域変数 integration_constant_counter の値を付加するものです. すなわち, ここの処理で, 方程式の積分を行うと右辺に integrationconstant1 等の積分定数を意味する文字列を追加する処理を行っているのです. 如何ですか？ 随分と機械的ですね…. それから変数 var が原子であり, '(isinop exp var)' が 'T の場合に積分を名詞形で返します. それ以外の場合は integrator 函数を用いて積分を実行します.

このことから問題の処理は integrator 函数で生じていることになります. ところで, integrator 函数は大きなプログラムになるので, 実際の式の積分で動きを見てみましょう. 式は $\sqrt{x+\frac{1}{x}-2}$ とします. この式は単純な処理を行うと間違える式です:

```
(%i19) integrate(sqrt(x+1/x-2),x);
                                  /
                                  [          1
(%o19)                            I sqrt(x + - - 2) dx
                                  ]          x
                                  /
```

```
(%i20) integrate(sqrt(factor(x+1/x-2)),x);
                                /
                                [      1
(%o20)                          I sqrt(-) abs(x - 1) dx
                                ]      x
                                /
(%i21) assume(x>1);
(%o21)                                [x > 1]
(%i22) integrate(sqrt(x+1/x-2),x);
                                   3/2
                                2 x    - 6 sqrt(x)
(%o22)                          -------------------
                                        3
(%i23) forget(x>1);
(%o23)                                [x > 1]
(%i24) assume(x<1);
(%o24)                                [x < 1]
(%i25) integrate(sqrt(x+1/x-2),x);
                                                1   2
                                          2 sqrt(-) x
                                   1            x
(%o25)                       2 sqrt(-) x - -------------
                                   x            3
```

この例は, Maxima-5.9.2 で 'integrate(sqrt(x+1/x-2),x)' を処理すると $\dfrac{2x^{\frac{3}{2}}-6\sqrt{x}}{3}$ を返していました. この結果は $x \geq 1$ の場合は正しくても,$x < 1$ の場合は嘘になります. ところで, Maxima-5.22.1 では上に示すようにきちんと処理ができていますね. ここで, 'integrate(sqrt(x+1/x-2),x)' と 'integrate(sqrt(factor(x+1/x-2)),x)' の結果は共に名詞型ですが, factor 函数を用いた後者では 'abs(x - 1)' の項が出ており, より進んだ形になっていますね. そこで, $\sqrt{x+\dfrac{1}{x}-2}$ と $\dfrac{|x-1|}{\sqrt{x}}$ の内部表現の違いを確認しておきましょう:

```
(%i69) exp1:sqrt(x+1/x-2);
                                        1
(%o69)                          sqrt(x + - - 2)
                                        x
(%i70) exp2:sqrt(factor(x+1/x-2));
                                  abs(x - 1)
(%o70)                            ----------
                                   sqrt(x)
(%i71) :lisp $exp1

(#1=(MEXPT . #2=(SIMP)) ((MPLUS . #2#) -2 (#1# $X -1) $X) ((RAT SIMP) 1 2))
(%i71) :lisp $exp2

(#1=(MTIMES . #2=(SIMP)) (#3=(MEXPT . #2#) (#3# $X -1) ((RAT SIMP) 1 2))
 ((MPLUS . #2#) 1 (#1# -1 $X)))
```

ここでの例では Common Lisp として SBCL を用いているので, 表記が CLISP や GCL とは幾分違っていますが, 実体は全く同じです.

さて, 演算子 ":lisp" を用いて Maxima の式の内部表現を表示させていますが, この演算子 ":lisp"

は空行から後の改行までの入力式を裏の LISP に流し込む演算子です. ここでは式 exp1 と式 exp2 の LISP 内部の名前 (頭に記号 “$” が付きます) を入力して, その表示を調べています. そこで, 式 exp1 の内部表現の caar が MEXPT, 式 exp2 の内部表現の caar が MTIMES になることに注意しましょう. そして, sinint で実際の積分を行う内部函数は integrator ですが, この integrator 函数には以下の処理があるからです:

```
485       (setq y (cond ((eq (caar exp) 'mtimes)
486                      (cdr exp))
487                     (t
488                      (list exp))))
```

この処理は与式 exp の主演算子が mtimes であれば, その内部表現の cdr を取り, そうでなければ, そのままをリストにするという処理です. この処理は内部函数 integrator で実際の積分処理を行う函数の分岐に関係します.

12.4　integrate_use_rootsof を用いた積分

Maxima の有理式の記号積分で大域変数 integrate_use_rootsof があります. この大域変数を true に設定すると, 有理式の積分で分母が一次式の積に分解できないものでも記号積分が可能となります. ここでは実際の動きを見てみましょう.

```
(%i53) integrate_use_rootsof:false$
(%i54) ans:integrate(1/(x^3-x^2-x+4),x);
                                /
                                [          1
(%o54)                          I ---------------- dx
                                ]  3    2
                                / x  - x  - x + 4
(%i55) integrate_use_rootsof:true$

(%i56) ans:integrate(1/(x^3-x^2-x+4),x);
               ====
               \            log(x - %r7)
(%o56)          >       ------------------
               /                  2
               ====     3 %r7  - 2 %r7 - 1
                                3    2
               %r7 in rootsof(x  - x  - x + 4)
```

大域変数 integrate_use_rootsof が ‘false’ の場合, $\frac{1}{x^3-x^2-x+4}$ の積分は失敗していますが, integrate_usr_rootsof が ‘true’ の場合はちゃんと計算ができているようです. ここで, 計算結果 ans の内部表現を式 ‘sum(x^2,x,0,2)’ で比較してみましょう:

```
(%i57) :lisp $ans

((%LSUM . #1=(SIMP))
 (#2=(MTIMES . #1#)
  ((MEXPT . #1#)
   (#3=(MPLUS . #1#) -1 ((MTIMES SIMP . #4=(RATSIMP)) -2 $%R2)
```

```
     (#2# 3 ((MEXPT SIMP . #5=(RATSIMP)) $%R2 2)))
    -1)
  ((%LOG . #1#) (#3# (#2# -1 $%R2) $X)))
 $%R2
 (($ROOTSOF . #1#)
  (#3# 4 ((MTIMES SIMP . #4#) -1 $X) (#2# -1 ((MEXPT SIMP . #5#) $X 2))
   ((MEXPT SIMP . #5#) $X 3))))
(%i57) test:'sum(x^2,x,0,4);
                                        4
                                       ====
                                       \        2
(%o57)                                  >      x
                                       /
                                       ====
                                       x = 0
(%i58) :lisp $test
((%SUM SIMP) ((MEXPT SIMP) $X 2) $X 0 4)
```

このように,'sum(x^2,x,0,4) の内部表現と計算結果を比較すると, 計算結果に '($ROOTSOF SIMP)' が第1成分のリストが, 変数の上限と下限を設定する個所に置かれています. では, integrate_use_rootsof を 'true' にすると, どの様な処理が実行されるのでしょうか？ さっそく調べてみましょう.
そこで, grep integrate_use_rootsof *lisp を src ディレクトリ上で実行してみましょう. すると, sinint.lisp に, この大域変数が含まれていることが判ります. この該当個所の抜粋を次に示しておきましょう:

```
(defvar $integrate_use_rootsof nil
"Use the rootsof form for integrals when denominator does not factor")

(defun integrate-use-rootsof (f q variable &aux qprime ff qq
                              (dummy (make-param)) lead)
  ;; p2e is squarefree in polynomial in cre form p1e is lower degreee
  (setq lead (p-lc q))
  (setq qprime (disrep (pderivative q (p-var q))))
  (setq ff (disrep f) qq (disrep q))
  '((%lsum) ((mtimes)
             ,(div* (mul* lead (subst dummy variable ff))
                    (subst dummy variable qprime))
             ((%log) ,(sub* variable dummy))) ,dummy
    (($rootsof) ,qq)
    )
  )
```

integrate_use_rootsof が true の場合,integrate-use-rootsof が用いられます. ここでの処理は実の所, 決った雛形に式を当て嵌めているだけです.lead や qprime に ff の処理はその当て嵌める為の式を生成しているだけで, それを '((%lsum) で開始するリストに当て嵌めています. このリストの中に, (($rootsof) ,qq) がありますが, この函数$rootsof は他では定義されていないダミー函数です. 又,Maxima の sum 函数には, リストの形式で与えられた助変数の集合に対する処理は行えません. その意味でも, ここでの処理は非常に形式的な処理です.
但し, 内部の計算は出来ているので, 以下の様な処理を行えば, 最終的な結果を得る事が可能です.

```
(%i1) integrate_use_rootsof:true;
```

```
(%o1)                                    true
(%i2) ans:integrate(1/(x^3-x^2-x+4),x);
                ====
                \          log(x - %r1)
(%o2)            >      ------------------
                /            2
                ====    3 %r1  - 2 %r1 - 1
                                3     2
                %r1 in rootsof(x  - x  - x + 4)
(%i3) r1s:subst(%r1,x,solve(x^3-x^2-x+4,x))$
(%i4) ans1:substpart("+",map(lambda([z],ev(inpart(ans,1),z)),r1s),0)$
(%i5) diff(ans1,x)$
(%i6) trigsimp(%);
                                        1
(%o6)                              ---------------
                                    3     2
                                   x  - x  - x + 4
```

この方法では大域変数 integrate_use_rootsof を ‘true’ にして式の積分を行います. その結果を ans に割当て, sum の中身を part 函数を使って ‘part(ans,1)’ で取出します. 次に, 与式の分母の解を solve 函数を用いて計算します. この与式の分母は 3 次式なので厳密解が計算できますが, 5 次以上の方程式であれば algsys 函数を用いることになるでしょう. ここで algsys 函数は厳密解が計算出来れば厳密解を計算し, それが無理ならば近似解を返す函数です. この際に変数 x を積分結果で用いられている解%r1 に置換えます. それから, lambda 函数を使って ev 函数による評価を行い, substpart 函数を使ってリストから和に変換させています. なお, その結果の ans1 は長い式になるので非表示にしています. 次に微分を行い, trigsimp 函数を使って式を整理すれば元の函数が出ているので, ちゃんと計算できていることが判ります.

この様にエレガントとは程遠い, とてもエレファントな計算をさせています. 勿論, この計算では式の極を考えていないので形式的な結果です,

如何でしょうか. 数式の積分は計算機の中の大勢の小人さん達が捻り鉢巻で計算していると感じていた方も多いかもしれませんが, 実際の処理は数値計算で方程式を解くときに色々な定式化に基いて計算処理を行うのと同様に, 数式処理でも定式化に基づいて記号の処理を行っているのです. 「**計算機の中に小人が居る**」といったロマンティックなことは残念ながらありません.

第13章　Maximaの簡単な改造

13.1　使い勝手の向上を目指して

Maxima は非常に面白いシステムです. 特に面白い点は, §12 で行ったように, その改造が容易にできることです. その上, ちょっとした改造を行うために貴方が **LISPER**(LISP のエキスパート) である必要は全くありません. LISP の多少の知識があれば, あとは Maxima のソースファイルを活用するだけで済むのです.
ここで, Maxima は単純に数式を処理するだけではなく, 様々な入出力を行っています. 実際, グラフィックスでは外部のアプリケーションを立ち上げたりもしている程で, 何かを最初から作り上げる必要性は意外になく, 似た処理を行う函数を観察して, その函数の仕組を真似たり, 場合によってはその函数の複製を作り, 挙句の果てには, その函数を乗っ取ってしまっても構わないのです.
勿論, 乗っ取りを行った結果, 他との互換性が崩れてしま可能性も十分にあるので, 大局的に影響を及ぼすような処理, 例えば, Maxima の再構築といった処理は, あまり勧められません. しかし, 函数の複製を作って, 特定の処理だけを行うようにすれば互換性の問題はないでしょう. また, 修正したファイルを個々人のディレクトリに置いて, maxima-init.mac ファイルで読込むように設定しておくのも良いでしょう.
この章を始めるにあたって, Maxima で不満な点はありませんか？ 数式処理の能力が足りない？ それはとても重要なことですが, ここでの簡単な改造からは外れてしまう問題もあるし, 貴方にとって不満でも他の人にとっては然程でもないかもしれませんね.
では, 一般の人が最も考え込むのは何でしょうか？ このような技術系のソフトウェアの場合, ヘルプシステムの重要度は高くなります. ここで, Maxima の Help システムはそれなりの良くできています. しかし, 最近の数式処理システムでは HTML 等を使った Hyperlink を用いたり視覚的にも優れたものです. それと比較すると, Maxima のマニュアルは texinfo を用いる方式は**古い**ものです. そこで, 今風に HTML や PDF が表示できれば如何でしょうか？ それだけでも使い易くなると思いませんか？
そんな訳で, ここではオンラインマニュアルの表示に関わる describe 函数を改造してみましょう.

13.2　describe函数は何処にある？

まず, describe 函数を改造したければ, describe 函数の定義の所在を明確にし, それがどのようなものであるかを知っておく必要があります.
describe 函数の捜し方は色々ありますが, UNIX 環境であれば, Maxima のソースファイルが収録されたディレクトリ src に移動し, grep describe *.lisp—less を実行してみましょう. なお, MS-

Windows 版にも Maxima のソースファイルが付属しているので, 何らかのアプリケーションを使って捜せば良いでしょう.
では次に結果を一部示しておきましょう:

```
macdes.lisp:(defmspec $describe (x)
mdebug.lisp:                    '((displayinput) nil (($describe) ,line $exact))))
mdebug.lisp:                    '((displayinput) nil (($describe) ,line $inexact))))
option.lisp:       ((eq '$describe (caar ans)) (mdescribe (decode (cadr ans))))
option.lisp:(subc $describe (c))
option.lisp:(subc $general-info () $describe $example $options $primer $apropos)
option.lisp:(subc $options (c) $down $up $back $describe $exit)
option.lisp:(subc $user-aids () $primer $describe $options $example $apropos
       $visual-aids)
trans1.lisp:(def%tr $describe              $batcon)
```

ここでは Maxima-5.22.1 を利用しているので他の版では多少出力が違っているかもしれません. しかし, 捜し方は単純で, 函数定義では defun 函数や defmspec といったマクロが用いられているので, "$describe" と一緒にこれらの文字列が出ている箇所を捜せは良いのです. この例では, "macdes.lisp:(defmspec $describe (x)" という行があるので, describe 函数は macdes.lisp で定義されていることが判ります.
そこで今度は macdes.lisp の複製を貴方の作業ディレクトリに作って, それを適当なエディタを用いて編集してみましょう.

13.3　describe 函数の動作

describe 函数の定義式を次に示してしておきます. なお, Maxima-5.22.1 の describe 函数なので他の版では多少異っているかもしれません:

describe 函数の定義

```
118 (defmspec $describe (x)
119   (let ((topic ($sconcat (cadr x)))
120         (exact-p (or (null (caddr x)) (eq (caddr x) '$exact)))
121         (cl-info::*prompt-prefix* *prompt-prefix*)
122         (cl-info::*prompt-suffix* *prompt-suffix*))
123     (if exact-p
124         (cl-info::info-exact topic)
125         (cl-info::info topic))))
```

ここで, describe 函数の定義を簡単に解説しておきましょう. 最初の defmspec は Maxima の函数定義のためのマクロで, ここで引数は x です. さて, let 函数は内部変数に値を割当てます. ここでは '($sconcat (cadr x))' を処理し, その結果を変数 topic に束縛させています. この $sconcat は Maxima の函数 sconcat に対応し, 引数を文字列に変換する函数です. このことを describe 函数に print 文を挿入して確認しましょう:

```
(%i6) to_lisp();
```

```
Type (to-maxima) to restart, ($quit) to quit Maxima.

MAXIMA> (defmspec $describe (x)
(print x)
  (let ((topic ($sconcat (cadr x)))
        (exact-p (or (null (caddr x)) (eq (caddr x) '$exact)))
        (cl-info::*prompt-prefix* *prompt-prefix*)
        (cl-info::*prompt-suffix* *prompt-suffix*))
    (if exact-p
        (cl-info::info-exact topic)
        (cl-info::info topic))))


#<FUNCTION (LAMBDA (X)) {1005B56089}>
MAXIMA> (to-maxima)
Returning to Maxima
(%o6)                                   true
(%i7) describe(sconcat);

(($DESCRIBE) $SCONCAT)
 -- Function: sconcat (<arg_1>, <arg_2>, ...)
     Concatenates its arguments into a string.  Unlike `concat', the
     arguments do not need to be atoms.

          (%i1) sconcat ("xx[", 3, "]:", expand ((x+y)^3));
          (%o1)                xx[3]:y^3+3*x*y^2+3*x^2*y+x^3


(%o7)                                   true
```

ここでの例では, to_lisp 函数で Maxima の裏側の LISP に入って, $describe 函数の再定義を行い, describe 函数を起動させています. ここで describe 函数には let の前に '(print x)' が挿入されていることに注意して下さい. そうして再定義した describe 函数を使って sconcat 函数を調べると, "(($DESCRIBE) $SCONCACT)" と表示されていますね. これが変数 x に束縛されてた値なのです. したがって, topic に割当てられる値は "$SCONCAT", すなわち, describe 函数の引数なのです. さて, この '(caddr x)' の値が空, あるいは '$exact に等しい場合に変数 exact-p に 'T, それ以外は 'NIL を割当てます. 最後に cl-info パッケージの変数*prompt-prefix*と変数*prompt-suffix*に*prompt-prefix*と*prompt-suffix*の値を束縛させます.
この変数への割当処理に続くのが, if による分岐処理です. ここでは exact-p の値が 'T であれば, cl-info パッケージの info-exact 函数に topic の値を引き渡します. 'NIL であれば cl-info パッケージの info 函数に topic の値を引き渡します. 通常の処理では info-exact に topic の値が引渡されて texinfo によるマニュアルが表示されます.

13.4 函数 ponpoko の仕様

describe 函数の動作は大体判りました. 今度はどのように改造するかを考えてみましょう. 最近のアプリケーションは困ったときに HTML ブラウザを使ってヘルプや PDF を表示させますね. そこを目的にしましょう. さて, describe 函数を基にして作る函数の名前は, そのオリジナリティのなさ,

それにも関わらず「**打ては響く**」様な動作を期待して, ここでは狸らしく “ponpoko” とします. そして, この函数を定義するファイル名を “ponpoko.lisp” とします.
函数 ponpoko は, 函数名を指定すれば通常の info ファイルを用いたヘルプを表示し, 文字列 “html” を指定すると HTML のマニュアルを表示, 文字列 “pdf” を指定すると PDF のマニュアルを表示する仕様とします, ここで, HTML マニュアルと PDF マニュアルは私のホームページに置いてあるものを用います[1].

13.5　大域変数の作り方

ここで HTML ブラウザとしては Firefox, Google-Chromme や konqueror, あるいは, w3m や lynx, それから, Internet Explore と色々あります. PDF も同様で acroread や xpdf に gv もあって環境によって色々切り替えられるべきでしょう. そして, HTML ファイルや PDF ファイルの格納先も別途定めておく必要がありますね.
このことから必要となる大域変数はブラウザとファイルの格納先の 4 個となります. そこで, Maxima 側の変数名を hbrowser で HTML ブラウザ, pbrowser で PDF ブラウザ, jhtmldir で HTML ファイルの格納先, jpdfdir で PDF ファイルの格納先を指示することにします.
また, 他の函数でも用いられている大域変数で, これらのブラウザを切替えることにします. さて, Maxima 内部で大域変数を定義したい場合は defvar マクロを用います. この defvar マクロの構文を次に示しておきましょう:

defvar マクロの構文

```
(defvar 変数名 値)
```

この defvar は LISP のマクロで, スペシャル変数として変数の宣言と既定値の設定を行います. なお, defparameter という似たマクロがありますが, これらの違いは, defvar 函数で宣言した変数が値を持っている場合には, その値は変更されませんが, defparameter 函数の場合は新しい値で置換えられます. Maxima では両者に使い分けがあり, 利用者が扱う大域変数の宣言では defvar 函数, Maxima 内部で利用者が設定することを想定していない, システムの環境変数の宣言で defparameter 函数が主に利用されています.
さて, Maxima の変数の内部表現は, 先頭に文字 “$” が付きます. そのために hbrowser 函数の名前は LISP 側では “$hbrowser” になります. このことに留意して大域変数を定義しましょう.

ponpoko.lisp での大域変数の定義

```
(defvar $hbrowser "firefox")
(defvar $pbrowser "xpdf")
(defvar $jhtmldir "Maxima/ManualBook/")
(defvar $jpdfdir "Maxima/ManualBook/Book-New/")
```

[1] http://www.bekkoame.ne.jp/ ponpoko/Math/books/ManualBook.tgz,
http://www.bekkoame.ne.jp/ ponpoko/KNOPPIX/MaximaBook.pdf. 2011 年 2 月現在, HTML ファイルの方は PDF と比べて格段に古いものです. ごめんなさい.

ここで, “$jhtmldir” と “$jpdfdir” の値は私の環境の値を設定しています. この値は皆さんの環境に合わせて指定するようにしてください.
さて, 次に函数の定義を行いましょう. 今回は describe 函数の実質的な乗取りが目的なので, 引数に “html” や “pdf” が指定されたときに ponpoko 函数に引渡す処理を挿入すれば良いのです. これは LISP の cond 函数を用いた処理が妥当でしょう.
つまり, 次に示す函数のイメージになります:

函数 ponpoko のイメージ

```
118 (defmspec $ponpoko (x)
119   (let ((topic ($sconcat (cadr x)))
120         (exact-p (or (null (caddr x)) (eq (caddr x) '$exact)))
121         (cl-info::*prompt-prefix* *prompt-prefix*)
122         (cl-info::*prompt-suffix* *prompt-suffix*))
123         (cond
124             ((topicに"html"の情報がある場合)
125              (hbrowserを起動しろ))
126             ((topicに"pdf"の情報がある場合)
127              (pbrowserを起動しろ))
128             (t
129              (if exact-p
130                  (cl-info::info-exact topic)
131                   (cl-info::info topic))))))
```

本来の describe 函数にこちらの要求を取り込めばこのような形になるでしょう. すると, topic がどのような書式なのか. そして, 外部のアプリケーションに大域変数の値を組合せたパラメータを引き渡して起動させれば良いのかが問題になりますね. 今度はこれらの課題を吟味してみましょう.

13.6 判別について

変数 topic には ‘($sconcat (cadr x))’ の値が束縛されます. ここで函数名に “$” が付いていることに注目しましょう. 何故なら, $sconcat は Maxima の函数 sconcat そのものだからです. ここで, ‘(cadr x)’ は引渡された変数に束縛された対象を LISP 内部で処理する際に不要な箇所を落す処理です. sconcat 函数は Maxima の式を結合して Maxima の文字列型に変換する函数です. 通常, describe 函数は一つの文字列を入れるために, ここでは単純に Maxima の文字列を LISP の文字列に変換する函数として用いていることが判ります.
このことから topic を用いた条件分岐では, topic に束縛された値が “html” や “pdf” という文字列であるかどうかを判別すれば良いことになります.
この判別に LISP の equal 函数が使えます. たとえば, topic が文字列 “html” に等しいかどうかは ‘(equal mantype ”html”)’ で判別できますね.
さて, 判別の方はこれでできたことになります. 今度は外部のアプリケーションの立ち上げの方法を調べましょう.

13.7　外部アプリケーションの立ち上げ方

外部アプリケーションを Maxima はどうやって立ち上げているのでしょうか？ここで参考になるのは外部のアプリケーションを利用している Maxima の函数です. さて, 何があるでしょうか？ ここでグラフを Maxima に描かせる場合, 通常は gnuplot が立ち上がっていますが, これはどのようにしているのでしょうか？この方法を真似てしまえば良いと思いませんか？ そこで, 今度は “gnuplot” をキーワードにソースファイルを調べると, どうやら plot.lisp で定義している函数で gnuplot の記述があることが判ります.
そこで, plot.lisp を適当なエディタで開いて調べてみましょう. すると, plot2d 函数の定義の尾に次の記述があります:

plot.lisp より mgnuplot の起動の個所

```
1596        ($mgnuplot
1597         ($system (concatenate 'string *maxima-plotdir* "/" $mgnuplot_command)
1598                  (format nil " -plot2d ~s -title ~s" file "Fun1"))))
1599  output-file))
```

ここで示した個所は MS-Windows 版の Maxima に付属の mgnuplot の起動を行う箇所です. そして, $system 函数は Maxima から外部のアプリケーションを立ち上げる際に用いられ, この函数に Maxima 外部のアプリケーション等に実行させたい処理を Maxima の文字列として引き渡せば良いのです. この$system 函数の引数に concatenate 函数が用いられていますが, この concatenate 函数は LISP の函数で, 第 1 引数が'string の場合に後続の文字列を結合して新しい文字列を生成すします.
このことから外部アプリケーションは$system 函数を用い, ブラウザと開くファイルの指定は concatenate 函数を用いてしまえば良いことになります. したがって, $hbrowser で指定したブラウザを使って$jhtmldir で指定したディレクトリにある “index.html” ファイルを開きたければ, 次の記述で良いことになります:

```
($system (concatenate 'string $hbrowser " "
                              $jhtmldir "index.html" "&")))
```

さらに, HTML ファイルや PDF ファイルの名前も大域変数にしておくとどうでしょうか？ それだけで随分と汎用性が出ますね. そこで, ブラウザで開く HTML ファイル名や PDF ファイル名を指定する変数名をそれぞれ hfile と pflle とし, 既定値を “index.html” と “MaximaBook.pdf” にしておきましょう.
これを纏めると函数 ponpoko は完成です.

13.8　完成

以上の考察を基に函数 ponpoko を完成させてみましょう:

函数 ponpoko

```
(defvar $hbrowser "firefox")
(defvar $pbrowser "xpdf")
```

```
(defvar $hfile "index.html")
(defvar $pfile "ManualBook.pdf")
(defvar $jhtmldir "Maxima/ManualBook/")
(defvar $jpdfdir  "Maxima/ManualBook/")
(defmspec $ponpoko (x)
   (let
    ((topic ($sconcat (cadr x)))
     (exact-p (or (null (caddr x)) (eq (caddr x) '$exact)))
     (cl-info::*prompt-prefix* *prompt-prefix*)
     (cl-info::*prompt-suffix* *prompt-suffix*))
    (cond
         ((equal topic "html")
          ($system (concatenate 'string $hbrowser " "
                    $jhtmldir $hfile "&")))
         ((equal topic "pdf")
          ($system (concatenate 'string $pbrowser " "
                    $jpdfdir $pfile "&")))
         (t (if exact-p
             (cl-info::info-exact topic)
             (cl-info::info topic))))))
```

説明は特に不必要かと思いますが, jhtmldir と jpdfdir は貴方の環境に合せて修正して下さい. 私の環境では HTML ファイルをホームディレクトリ上の Maxima/MaximaBook 以下に置き, PDF ファイルも同じディレクトリに置いています.
ここで, この函数の遊び方ですが, 兎に角, Maxima 側から load 函数で読込んでしまえば良いのです. ponpoko.lisp をカレントディレクトリ上に置いていれば, ディレクトリの指定をする必要もなく, 単純に load("ponpoko.lisp"); と入力すれば良いのです. これで函数 ponpoko が読み込まれて, あとは ponpoko("html"); や ponpoko("pdf"); を実行すれば良いのです.
では, 実行例を示しておきましょう.

```
(%i42) load("ponpoko.lisp");
(%o42)                         ponpoko.lisp
(%i43) ponpoko("to_lisp");

 -- Function: to_lisp ()
     Enters the Lisp system under Maxima. '(to-maxima)' returns to
     Maxima.


(%o43)                                 true
(%i44) ponpoko("html");
(%o44)                                  0
```

読込んだあとに to_lisp 函数を調べています. この場合は describe 函数と同じ働きをします. 次に, 引数を "html" にすると既定値の firefox が立ち上ります. ここで, system 函数に送り込んだ文字列の末尾に "&" を入れているので, Maxima は firefox を立ち上げると入力プロンプトに復帰しています. もしも, ここで "&" を末尾に入れていなければ, 起動した外部アプリケーションが終了するまで処理が Maxima に戻りません.
ここで, 函数 ponpoko を一々読込むのが面倒であれば, maxima-init.mac ファイルに予め 'load("ponpoko.lisp");' を記入しておきます. 例として, lstringproc パッケージも読込む様にした maxima-init.mac を次に

示しておきます.

maxima-init.mac の例

```
load("ponpoko.lisp");
load("stringproc.lisp");
```

このファイルは UNIX の場合, ホームディレクトリに置けば, 準備が完了です.

13.8.1 MS-Windows 環境の場合

MS-Windows 環境の場合は UNIX 環境と比べ, 幾つか注意しなければならないことがあります. まず, フォルダ (ディレクトリ) の区切は MS-Windows では逆スラッシュ("\") を用いていますが, Maxima 内部では UNIX と同様にスラッシュ("/") を用います. ところで, 考えておかなければならないのは ponpok.lisp と maxima-init.mac, そしてマニュアルを置くフォルダです.
MS-Windows を利用する方は基本的にコマンドプロンプト (以下, DOS 窓とも略記) を開いて処理する人よりも, wxMaxima 等の GUI 環境を用いる方の方が大多数でしょう. そのために利用する環境により, maxima-init.mac 等の置場を変更する必要があります.
何がなんでも DOS 窓を利用する方の場合, maxima-init.mac と ponpoko.lisp の置場は Maxima による作業を行うフォルダに入れておく必要があります.
これに対し, wxMaxima や XMaxima を利用する場合は勝手が異なります. Windows メニューのアプリケーションがら Maxima や XMaxima を選んで立ち上げる場合, `C:\Documents and Settings` フォルダにある利用者固有のフォルダに maxima-init.mac と ponpoko.lisp を入れておく必要があります. ところが, wxMaxima はやや勝手が違い, wxMaxima が置かれた場所がそのホームディレクトリとなるので, ここに maxima-init.mac ファイルを置かなければなりません.
勿論, wxMaxima を DOS 窓で `C:\Documents and Settings` フォルダにある個人フォルダに移動して立ち上げてしまえば良いのですが, そのときに wxMaxima や maxima の実行ファイルのあるフォルダを環境変数 Path に追加しなければなりません. このように MS-Windows では何かと面倒なことが生じるのです. そこで, 何も考えずにできる方法は, 兎に角, 個人フォルダと wxMaxima のフォルダの二箇所に別々に置いてしまうことです. wxMaxima だけを利用するのであれば wxMaxima のフォルダだけで十分です.
さて, この wxMaxima は Maxima のフォルダの直下にあります. Maxima はインストール時に特に設定を行なわなければ `C:\Program Files\Maxima-5.13.0` (利用する Maxima が 5.13.0 の場合) にある筈です. そして, maxima-init.mac, ponpoko.lisp や Maxima マニュアルのフォルダも一緒に置きます. ここで, Maxima マニュアルのホルダは下に ManulaBook フォルダを置き, その中に HTML ファイル, PDF ファイルや利用する画像ファイルを置きます.
firefox 等のアプリケーションも MS-Windows の場合は Path が通っていないことがあるので, 別途, 環境変数 Path にアプリケーションへの経路を設定しなけれなりません. この環境変数 Path の設定はコントロールパネルのシステムをダブルクリックし, その中にある環境変数のボタンを押します. ユーザー環境変数とシステム環境を編集する為のウィンドウが出て来るので, システム環境変数から,Path を選んで編集ボタンを押すと, 入力用のウィンドウ欄が出るので, そこに firefox 等のアプリケーションが置かれたフォルダの位置を追加します. それで使える筈です.

なお, UNIX 環境と MS-Windows 環境で切り替えを行う際に, Maxima の内部函数の*autoconf-win32*を用いて判別を行う仕様にしています. 実際, グラフ処理を行うために環境によってアプリケーションの切り替えを行う必要がある plot.lisp では動作する環境が MS-Windows 環境であるかどうかを, (string= *autoconf-win32* "true") のようにして判別しています. つまり, この S 式を評価して true になる場合が MS-Windows 環境で, そうでなければ UNIX 環境として処理を行っています.

13.9 大域変数の変更について

ところで, ponpoko.lisp で定義した大域変数の変更は少し面倒です. 何故なら, Maxima の文字列と LISP の文字列は別物だからです. そこで, この変換に stringproc パッケージに付属する lstring 函数を用います. たとえば, hbrowser を初期値の firefox から opera に変更してみましょう.

```
(%i1) load("stringproc.lisp");
(%o1) /usr/local/share/maxima/5.13.0/share/contrib/stringproc/stringproc.lisp
(%i2) hbrowser;
(%o2)                                firefox
(%i3) hbrowser:lstring("opera");
(%o3)                                 opera
(%i4) ponpoko("html");
(%o4)                                   0
```

この方法は ponpoko.lisp で定義した大域変数全てで使えます. なお, 注意することにディレクトリを指定する場合は末尾に必ず "/" を入れ忘れないで下さい. これがないとファイル名と繋って別のファイルを指すことになってしまうからです.
さて, 如何でしょうか？ 残念ながら, まだブラウザを別個に立ち上げる手間が省けただけですね. 今後, 必要な機能としては指定した項目が載っている HTML ファイルを検索してちゃんと開くといった操作が必要になりますね. これは私にとってはとても大変なことで, 安易な方法を目指したこの章の範囲を越えてしまいそうです. そこで, 以降の作業は皆さんに投げておきます.

第14章　結び目のAlexander多項式

この章では三次元空間内の結び目, そして結び目から得られる結び目群の表現と結び目群の不変量としての Alexander 多項式について簡単な解説をします. なお, ここでの核心は Maxima に於ける演算子の定義方法と規則の適用です. したがって, 結び目に興味がなくても演算子の定義と規則の適用の事例として参考になると思います.

ここで「**何故, 結び目なのか？**」という素朴な疑問を抱かれる方も多いかと思いますが, そもそも, 結び目理論はドイツ語で「**Knotten Theorie**」と呼びます. それにしても Knotten!! 実に KNOPPIX に似ていますねぇ…. そこで, 結び目愛好家のために Maxima で結び目の不変量を計算して KNOPPIX/Math を Knotten Pics/Math と洒落込もうというのが目的です. ところで, この本では大雑把な結び目理論の話しかできませんので, 結び目理論の全般の話はクロウエル, フォックス [46], 本間 [54], 河内 [20], 村上 [58] 等の立派な本を参照して下さい. なお, 亀甲結びといった別の方面の愛好家の方の要望には沿える内容ではありませんので悪からず.

14.1　結び目の概要

結び目には「**蝶々結び**」とか色々な紐の結び方があります. ここで, 結び目のある二つの紐が与えられたときに同じ結び方であるかどうかを判別するにはどうすれば良いでしょうか？ 現実問題としては紐を引いたりして同じ形に変形できるかどうかで判別する方法がありますが数学ではどうするのでしょうか？

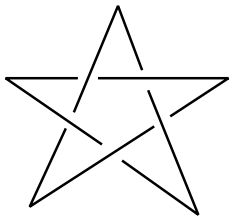


図 14.1: 星型結び目 (5_1)

まず, 結び目理論で扱う結び目は紐の両端を繋いだ図 14.1 に示すような 3 次元空間内部の円です. ここでどうして繋いで円にする必要があるのでしょうか？ 結び目の両端が切れたままだと端の引き方によっては結び目が消えてしまいますね. これはアニメの「トムとジェリー」の一場面, 子鼠の Nibbles がテーブルから下したパスタを一気に吸引してテーブルに登るやりかたを想像すると良いでしょう. ところが, 紐の両端を繋げて輪にしてしまうと解けるものと解けないものが出てくるでしょう. だから円にして考えます.

それから結び目は自分自身が交わることのないように 3 次元空間 $\mathbb{R}^3$ に置かれています. この状態を円 S^1 の 3 次元空間 $\mathbb{R}^3$ への「**埋め込み**」と呼びます. また, 結び目は一つの円の埋め込みですが, 複数の円を互いに交差しないように埋め込んだものは**絡み目** index+mbook む@む!むすひめ@結び目! からみめ@絡み目と呼ばれます. そして, 結び目が埋め込まれた空間, たとえば $\mathbb{R}^3$ から結び目 K を取り除いた空間 $\mathbb{R}^3 - K$ のことを結び目 K の「**補空間**」と呼び, $C(K)$ と記述します. また, 結び目

K を 3 次元空間内部で自分自身が交差しないように太らせて, ちょうど絡んだドーナッツのようにしたものを**管状近傍**と呼びます. この結び目 K の管状近傍を $N(K)$ と表記します. ここで結び目 K の管状近傍を取り除いた空間 $\mathbb{R}^3 - N(K)$ と結び目の補空間 $C(K)$ に位相幾何学的な違いはありません. そして, 管状近傍を除去した空間の方が後述する手術や被覆空間の構成で便利が良いので, 特に問題がなければ管状近傍を除いた空間を結び目の補空間とこの本では呼ぶことにします.

つぎに結び目に現実的な制約を入れましょう. まず, ここで扱う結び目は有限個の折線で近似できるものに限定します. この有限個の折線で近似できる結び目を「**順 (tame) な結び目**」と呼びます. 実際, 図 14.1 の結び目は 4 本の線分で構成されていますね. ちなみに無限個の折線が必要な結び目を「**野性的 (wild) な結び目**」と呼びます. それから結び目をゴム紐でできたものと考え, 結び目を $\mathbb{R}^3$ 内部で紐を切ったり, 紐を互いに交点させずに変形して行くことで互いに移り合える結び目を「**同値な結び目**」と呼びます. このことを数学では二つの結び目 K_1 と K_2 の間に「**ambient isotopy**」と呼ばれる写像が存在することに対応します. 具体的には結び目 $K_1, K_2 \subset \mathbb{R}^3$ に対して次の性質を充す写像 $F : \mathbb{R}^3 \times I \to \mathbb{R}^3, I = [0, 1]$ が存在することです:

ambient isotopy

- $t \in I$ に対して $F_t (= F(, t))$ は $\mathbb{R}^3$ の同相写像
- $t \in I$ に対して $F(K_1, t)$ は円 S^1 と同相
- $F(K_1, 0) = K_1$かつ $F(K_1, 1) = K_2$

二つの結び目 K_1 と K_2 の間に ambient isotopy が存在する場合, '$K_1 \Leftrightarrow K_2$' と表記することにします.

この ambient isotopy が二つの結び目 K_1 と K_2 に存在するということをより直観的に解説するならば, 結び目 K_1 から K_2 へと紐を切らずに変形だけで変形させる映画が存在することです. この映画では結び目の一部を別の時間に飛すことで交差を一時的に消してその間に結び目を動かして全体の交差点を解消するようなトリックを行わないもの, つまり, 結び目が時間とともに変形してゆく様子が見られる映画になります. そして, ambient isotopy を持つという関係 "⇔" は同値関係になります. この理由を**映画がある**ということで説明してみましょう. 同値関係を充すためには反射率, 対称律と推移律の三つの条件を充さなければなりません. 最初の反射率:$K \Leftrightarrow K$ については常に K を映し続ける映画があるので成立します. 対称律については $K_1 \Leftrightarrow K_2$ であれば, K_1 から K_2 への変形の映画を逆回しにすれば K_2 から K_1 への映画が得られます. つまり $K_2 \Leftrightarrow K_1$ となり, 対称律を充すことが判ります. 最後の推移律ですが, $K_1 \Leftrightarrow K_2$ と $K_2 \Leftrightarrow K_3$ が成立するとき, これらの映画があるので, それら二つの映画を繋ぎ合せれば K_2 を経由して K_1 から K_3 へ変形する映画ができます. つまり, $K_1 \Leftrightarrow K_3$ であることが判ります. このことから結び目を互いに変形して移り合えるものを同一視することができるのです.

では, ここで結び目を特徴付けるものに何があるのでしょうか？ まず, 結び目は 3 次元空間に埋め込まれた幾何学的対象です. そのままで考察するには流石に難しいので通常は平面に射影して考察します. これが結び目の「**射影図**」と呼ばれる図です. この射影図には, 結び目の紐が交差する点,

つまり交点が現われることがあります. この結び目の交点の総数も結び目を特徴付けるものの一つです, もちろん, 射影の方法を変えることで交点総数は異なります. たとえば, 平面 $\mathbb{R}^2$ に置かれた $x^2 + y^2 - 1 = 0$ で定義される円を自明な結び目と呼びますが, この自明な結び目を Y 軸を中心に捻じったものを XY 平面に射影すれば, 任意の個数の交点を持った射影図が得られます. しかし, 結び目の射影図が持ち得る最小の交点数を考えるとどうでしょうか？ まず, 自明な結び目の場合は 0 個ですね. そして, 一般の結び目では交点数は 0 以上の自然数の集合となります. ここで任意の自然数の集合は必ず最小元を持つことが自然数の性質から保証されるので, 最小の交点数で結び目を分類する方法が考えられます. では, この最小交点数による分類の実体はどうでしょうか？ 交点数が 0 の結び目, つまり, 交差を一切持たない結び目と同値な結び目は自明な結び目の他には存在しないので, この場合は一つだけです. では交点数だけで全ての結び目を分類できるでしょうか？ この点については残念なことに交点数だけで結び目の分類はできません. たとえば, 数学辞典 [59] の付録「公式 7 の結び糸」を参照して頂ければ判りますが, 交点数が 3 個と 4 個の場合は一つだけなのに, 交点数が 5 個以上になると同じ交点数を持つ結び目が複数個存在しています. このように交点数だけで一意に結び目は定まりませんが, 結び目の複雑さの度合いを示す値としては十分に価値があるので, 上記の結び目表等で用いられているように, 基本的な分類を行うために用いられています.

このように個々の結び目を分類するためには交点数は不十分です. もっと細かく分類するためにはどうすればよいのでしょうか？ そこで, 天下的になりますが「**結び目群**」と呼ばれる群を利用します. この結び目群は, 結び目 K の補空間の「**基本群**」と呼ばれる群で, 結び目の補空間の基本群はその補空間に固有の群です. そして, **3 次元空間内部の結び目はその補空間で分類できる**ことが知られているので, この結び目群に結び目の違いが現われるのです.
この結び目群の計算では結び目の射影図を利用します. この計算方法には二つの有名な方法があります. 第一の方法が Dehn による「**Dehn 表示**」, 第二の方法が Wirtinger による「**Wirtinger 表示**」です. 最初の Dehn による手法は結び目を境界とする向き付け可能な Seifert 曲面と呼ばれる曲面を構成して求める手法です. この Seifert 曲面は非常に重要な曲面で, 後述の Alexander 多項式に対応する**被覆空間**は Seifert 曲面で切り開いた曲面を繋ぎ合せることで容易に構成できます. この Dehn 表示と比べ, Writinger による手法は交差点近傍の情報だけで計算することができます. そこで, ここでは Wirtinger による結び目群の表示を採用することにしましょう.

ところで結び目群はそのままでは非常に扱い難い代物です. たとえば, Dehn の表示や Wirtinger 表示による群の表示は「**自由群**」に「**関係子**」を入れた群の表示で与えられ, このような群の表示が二つ与えられたときに, それらが同値な群であるかを判断することは結構, 面倒な問題です. この同値性の検証については「**Tietze 変換**」と呼ばれる操作で移り合える群は同じものであることが知られています. ここで群 F の表示を $\langle x_1, \ldots, x_n \mid r_1, \ldots, r_m \rangle$ とし, $\{r_1, \ldots, r_m\}$ が生成する群 (**帰結群**と呼ばれます) を R としています:

Tietze 変換

I	$\langle x_1, \ldots, x_n \mid r_1, \ldots, r_m \rangle$	$\Rightarrow$	$\langle x_1, \ldots, x_n \mid r_1, \ldots, r_m, u \rangle$	$u \in R$
I′	$\langle x_1, \ldots, x_n \mid r_1, \ldots, r_m, u \rangle$	$\Rightarrow$	$\langle x_1, \ldots, x_n \mid r_1, \ldots, r_m \rangle$	$u \in R$
II	$\langle x_1, \ldots, x_n \mid r_1, \ldots, r_m \rangle$	$\Rightarrow$	$\langle x_1, \ldots, x_n, y \mid r_1, \ldots, r_m, y\zeta^{-1} \rangle$	$y \notin \{x_1, \ldots, x_n\}, u \in F$
II′	$\langle x_1, \ldots, x_n, y \mid r_1, \ldots, r_m, y\zeta^{-1} \rangle$	$\Rightarrow$	$\langle x_1, \ldots, x_n \mid r_1, \ldots, r_m \rangle$	$y \notin \{x_1, \ldots, x_n\}, u \in F$

ここで帰結群とは具体的には群の表示の関係子で生成される最小の群 F の正規部分群ことで, 帰結群 R の元は具体的には $\Pi_{r \in \{r_1, \ldots, r_m\}, h \in F} h r^n h^{-1}$ の形で表記されるものになります. そして, これらの Tietze 変換を有限回繰替えすことで互いに移り合える群の表示は同じ群の表示であると結論付けることができます. ここでは Fox の本 [46] の P.59 にある例題を使って, 二つの表示 $\langle x, y, z \mid xyz(yzx)^{-1} \rangle$ と $\langle x, y, a \mid xa(ax)^{-1} \rangle$ が同じ群の表示であることを見てみましょう:

Tietze 変換の例

$$
\begin{aligned}
\langle x, y, z \mid xyz(yzx)^{-1} \rangle &\xrightarrow{\text{II}} \langle x, y, z, a \mid xyz(yzx)^{-1}, a(yz)^{-1} \rangle \\
&\xrightarrow{\text{I}} \langle x, y, z, a \mid xa(ax)^{-1}, a(yz)^{-1}, xyz(yzx)^{-1} \rangle \\
&\xrightarrow{\text{I}'} \langle x, y, z, a \mid xa(ax)^{-1}, a(yz)^{-1} \rangle \\
&\xrightarrow{\text{I}} \langle x, y, z, a \mid xa(ax)^{-1}, z(y^{-1}a)^{-1}, a(yz)^{-1} \rangle \\
&\xrightarrow{\text{I}'} \langle x, y, z, a \mid xa(ax)^{-1}, z(y^{-1}a)^{-1} \rangle \\
&\xrightarrow{\text{II}'} \langle x, y, a \mid xa(ax)^{-1} \rangle
\end{aligned}
$$

ここで示したように, 有限回の Tietze 変換の列によって二つの群の表示が同値であることはできます. しかし, この手法の難点は, そのような Tietze 変換が簡単に判るとは限らず, その上, そのような Tietze 変換の列を見付けることができないからといって別の群の表示であると結論付けられない点です.

このことから結び目の表示自体は, その表示だけを使って二つの結び目の根本的な違いを即座に知るという目的に適してはいません. しかし, 情報量は豊富に持っているので, この結び目群の表示から「**群を特徴付ける値**」を取り出して, それを使って比較したいものです. この「**群を特徴付ける値**」は「**不変量**」と呼ばれるもので, 色々な量があり, 最初に挙げた結び目の最小交差点数も結び目の不変量の一つですが, 前述のように強力なものではありません. ここでは結び目群に基く不変量として,「**結び目群の表示**」の関係子から得られる「**Alexander 多項式**」を計算する方法を紹介しましょう.

14.2 結び目の射影図

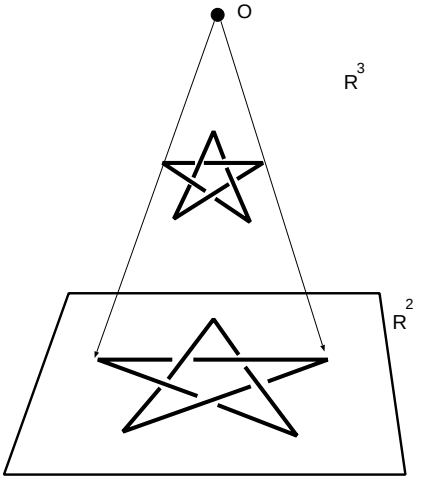


図 14.2: 結び目の射影図

最初に「**結び目の射影図**」について説明しましょう. これは 3 次元空間 $\mathbb{R}^3$ 内部の結び目を平面 $\mathbb{R}^2$ に図 14.2 に示す要領で, 光源 O から出る灯で結び目を射影したものです. この射影図上での結び目の紐自体が交点する個所のことを「**交点**」と呼びますが, この交点を十字で描くだけではどちらが上で, どちらが下かが判らないので, 高速道路のジャンクションを地図で描く要領で上を通る紐で下を通る紐が切断されるように描きます. その結果, 図 14.1 のような絵が得られます. そして, 結び目を道に見立てて紐が交差する個所を「**交差点**」, その交差点の上を通る紐を「**上道**」, 下側の紐を「**下道**」と呼びます. なお, 図 14.1 のように綺麗な射影図ではなく, 図 14.3 の左側に示すように $n(\geq 2)$ 重の交差点や直交しない交点が生じることがありますが, このような交点は順な結び目に高々有限個しか現れません. 何故なら順な結び目は有限個の折線で近似されるために, その射影も有限個の折線で近似できるからです. その上, 性格の悪い交点達は図 14.3 の右側に示す要領で紐を局所的に動かすことで交点を二重点だけにすることができます:

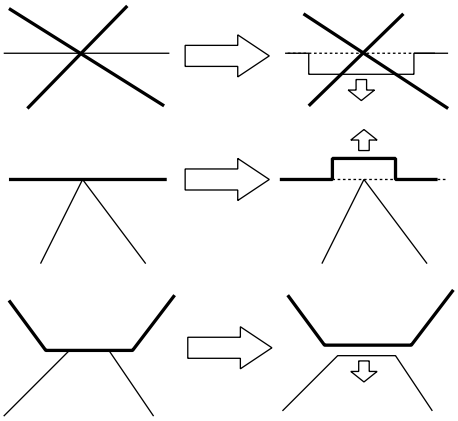


図 14.3: 変形可能な交点

このようにして得られた有限個の二重点のみの交点を持つ射影図を「**正則な射影図**」と呼びます. しかし, 同じ結び目でも射影方向の違い等の要因で同じ正則な射影図が得られるとは限りません. ではどうすればよいのでしょうか？ここで正則な射影図に対しては図 14.4 に示す「**Reidemeister 移動**」と呼ばれる局所的な変形操作があり, 同値な結び目は Reidemeister 移動を繰替えして行うことで, 相互に変形できることが知られています.

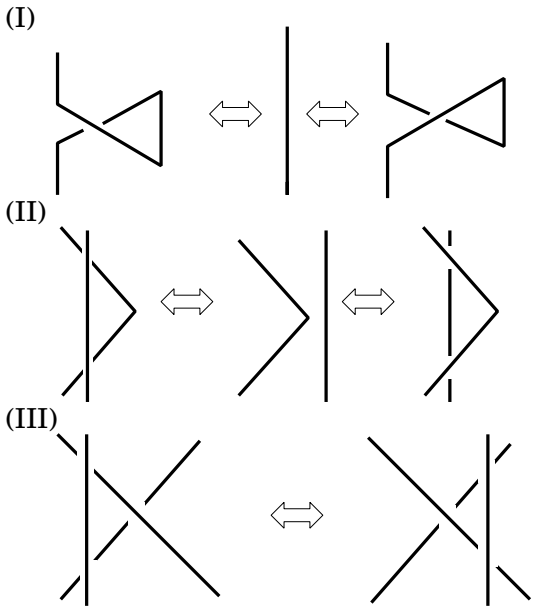


図 14.4: Reidemeister 変形

14.3　結び目群の Wirtinger 表示

Wirtinger 表示による結び目群の計算では, 最初に結び目 K の正則な射影図を作成しておきます. ここで結び目の正則な射影図は交点で幾つかの上道で分割されているので, 結び目 K に向きを入れて各上道に適当な変数を割当てます.

ここでは星型結び目 5_1 を使って手順を示しましょう. まず, 星型の結び目の正則射影図を図 14.5 に示します. この射影図の各上道に変数 v, w, x, y, z を割当てておきます:

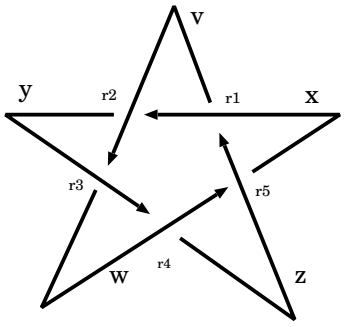


図 14.5: 星型結び目 5_1 と変数

これらの射影図上の曲線に割当てた変数が Wirtinger 表示による結び目群の生成元となります. 次

に結び目群の関係子を計算しなければなりませんが, ここでの関係子は図 14.6 に示す方法で交点毎に決定されます:

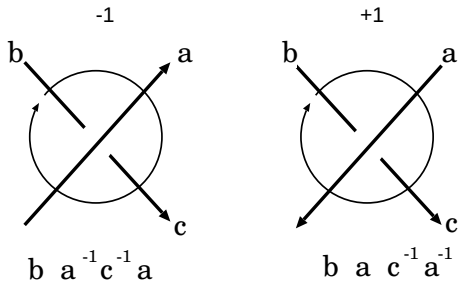


図 14.6: Wirtinger 表示

この図 14.6 の各交点の上にある +1 と −1 は「**交点の符号**」と呼ばれる結び目の各交点に付与される整数値で, 結び目の交点の符号の総和は「**符号和**」と呼ばれる重要な結び目の不変量の一つです.

この手順によって得られる結び目群の Wirtinger 表示は次のものです:

結び目群の Wirtinger 表現

$$\text{結び目群} = \langle \text{上道}_1, \ldots, \text{上道}_n | \text{交点}_1, \ldots, \text{交点}_n \rangle$$

ここで生成元の数と関係子の数は同じ n 個としていますが, n 個の関係子の内から一つの関係子は他の $n-1$ 個の関係子から生成できるので $n-1$ 個の交差点から求められる関係子だけで十分です.

ではどうして Wirtinger 表示で結び目群が表現できていると言えるのでしょうか？このことを天下り的ですが解説しましょう. 図 14.7 に結び目の交差点と各閉じた道を示しておきます.
結び目の射影図の上道に付けた変数は $\mathbb{R}^3 - K$ 内部の点 P から出発して上道を一回りして点 P に戻る閉じた道が対応します. ここで上道の回り方は図 14.7 に示すように上道の向きに対して右回りとします. この閉じた道は点 P に根本が固定されていますが, ゴム紐のように伸縮自在で結び目を取り除いてできた穴 (トンネル) に邪魔されなければ点 P に潰れてしまう性質を持っています. そして, 点 P を基点とする閉じた道 a_1 と a_2 が等しくなるのは, a_1 と a_2 が空間内部で連続的に変形することで互いに移り合える場合です.

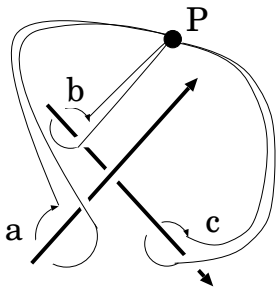


図 14.7: 閉じた道

この閉じた道には群の構造を持たせることができます. まず, 閉じた道 a と b の積 ab は点 P を出て a の道を辿って今度は b の道をたどる閉じた道で定義できます. そして, 点 P から動かない道が, ここでの積の単位元となり, 閉じた道 a の逆元 a^{-1} が a の逆方向に上道を一周する閉じた道となります. このことは図 14.8 の左上から矢印にしたがって aa^{-1} を変形させてゆくことで最終的に図の右上に示すように点 P に潰れることが理解できるでしょう:

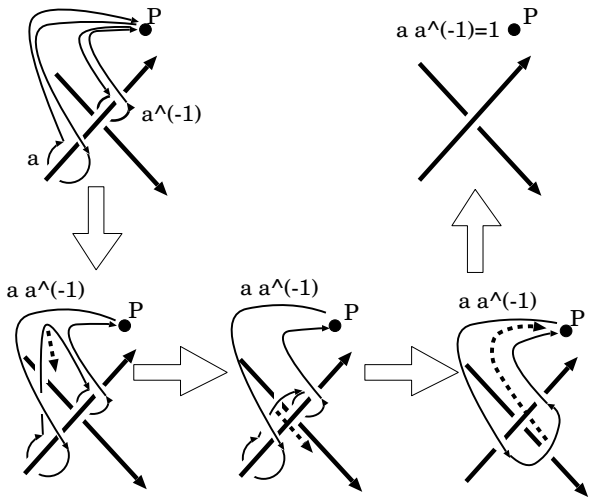


図 14.8: 閉じた道 a , a^{-1} と aa^{-1} の意味

この手順を解説しましょう. まず, 図左上に a と a^{-1} を示します. ここでの図で a^{-1} が a が重なると判り難いので各閉道を移動させています. ところで, これらの閉じた道 a と a^{-1} は結び目の上道に遮られているので a と a^{-1} が単体で上道を越えて点 P へと潰れることはできません. しかし, 両者の積 aa^{-1} は閉じた道 a を一回りしてから a^{-1} の道に沿って動きます. ここで a^{-1} は a の道の逆方向に動くこととなり, a と a^{-1} の結合点は点 P ですが, 積をとった時点で結合点を点 P に拘束する必要がないので結び目の補空間内部を自由に動かせます. そこで結合点を点 P から外して動かしてやれば aa^{-1} は図 14.8 の下段左に示す道になります. この調子で結合点を矢印に沿って動かすと下段右に示す道になりますが, この道は結び目の上道とは絡んでおらず, そのまま点 P に潰すことができます. ここで点 P から動かない点が基本群の単位元 1 なので aa^{-1} が単位元 1 であることが分ります.

ここで aa^{-1} と同様に Wirtinger 表示で現われる関係子の意味も調べてみましょう. この検証の様子を図 14.8 に示しておきます:

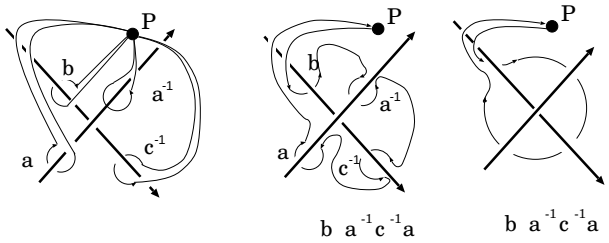


図 14.9: Wirtinger 表記の意味

まず, 閉じた道を関係子の順で繋ぐと曲線から外れた点 P から出て点 P に戻る向き付けられた円になります (図中央). そして, この円を整えると図右側に示すように結び目に引掛かからずに点 P に

潰せる円になります. このことから関係子は結び目補空間の単位元 1 に対応することが判ります. ここで閉じた道が基本群の単位元 1 になるということは閉じた道が空間上で邪魔されることなしに基準点 P に潰れることを意味しますが, さらに, この閉じた道は結び目の補空間に埋め込まれた円盤, すなわち, 自分自身が交点した変な円盤ではなく, 単純に歪んでいるだけの円盤の境界であることが知られています (「**Dehn の補題**」). ここで結び目群は結び目の補空間の基本群です. その基本群の任意の元が 1 , すなわち, 任意の閉道が潰れる空間のことを「**単連結**」な空間と呼びます. 近年証明された Poincaré 予想は「**単連結な閉 3 次元多様体は 3 次元球と同相である**」というものです. ここでいきなり「**多様体**」という言葉が出ましたが, これは物理学で用いられる宇宙空間と理解してよいでしょう. 空間が多様体のときの大きな御利益には, 多様体の何処でも局所的には通常の Euclid 空間 $\mathbb{R}^n$ と同じもので, さらに, そこに座標系を設定すると, これらの座標系同士には変換が存在することが保証された空間になるということです. このことは様々な物理現象を一斉に多様体の各点で観察したときに観測結果を互いの座標系上の値に変換して実験結果の比較・検討が行える空間であることを意味します. 逆に, この変換の存在が保証されなければ観測結果は観測した点に依存するために, 一般性を持つものとして議論ができるものにはなりません. この座標変換が微分可能であれば「**可微分多様体**」, 連続写像であれば「**位相多様体**」と呼びます. なお, 結び目は有限個の折線で近似可能なものに限定していますが, この仮定から, ここでの結び目の補空間は「**PL 多様体**」(PL=Piecewise Linear) と呼ばれる多様体になります. この PL 多様体と可微分多様体は同値であることが知られています. そして, Poincaré の補題の別の意味は, 「**2 次元球面を境界に持つ単連結な位相多様体は 3 次元球と同相になる**」ということです. すなわち, Poincaré の補題は Dehn の補題の 3 次元版になっていると言えるのです.

では, 実際に Wirtinger 表示を求めてみましょう. そのために図 14.5 の星型結び目 5_1 を考えます. このとき生成元は図中の大文字の V, W, X, Y, Z で, 関係子は小文字の r_1, r_2, r_3, r_4, r_5 とします. そして, これらの関係子の計算は図 14.7 の関係式から計算できます. ここではその結果のみを示しておきます:

星型結び目 5_1 の Wirtinger 表現

$$\langle v, w, x, y, z | xz^{-1}x^{-1}v, vx^{-1}v^{-1}y, yv^{-1}y^{-1}w, wy^{-1}w^{-1}z, zw^{-1}z^{-1}x \rangle$$

Wirtinger 表示による結び目群は図 14.4 に示す Reidemeister 移動の不変量となります. このことは表 14.1 に示す語の「**Tietze 変換**」を使って示すことができます. この詳細はクロウエル, フォックス [46] 等の結び目理論の本を参照して下さい.

14.4 群環と Fox の微分作用素

結び目群 $G(K)$ の生成元を $x_1, \ldots, x_n$ とするときに, これらでの生成元で生成される自由群 $F = \langle x_1, \ldots, x_n \rangle$ とその群環 $\mathbb{Z}F_n$ を考えます. ここで群環 $\mathbb{Z}F_n$ は群の元を使って, 積は群の積 "$*$" をそのまま用い, 可換演算子 "+" を導入して群の成分の形式的な和を導入したものから構成します. つまり, $a, b \in F_n$ のとき, 形式的な和 '$a + b$' を考え, '$a + b = b + a$' とします. また, m 個の $a \in F_n$ の和 '$a + \cdots + a$' は '$m * a$' と, 特に '$m * 1$' は 'm' と表記します. すると群環 $\mathbb{Z}F_n$ は係数として自然数 $\mathbb{Z}$

を持つ項を F_n の元とする多項式みたいな代物であることが判ります. そして, この群環では群の積演算子 “$*$” をそのまま積として流用しているので積 “$*$” に対しては閉じ, 和 “+” については結合律や分配律を満すように入れるために自然に環になります. なお, 以降の表記で必要が無い限り積 “$*$” は略記することにします.

この群環 $\mathbb{Z}F_n$ に対して新たに「**Fox の微分子**」Δ と呼ばれる演算子を導入します. ここで Fox の微分子は次の性質を持っています:

Fox の微分子 Δ

- $\Delta(x+y) = \Delta(x) + \Delta(y)$
- $\Delta(xy) = t(y)\Delta(x) + x\Delta(y)$

この Fox の微分子 Δ は最初の性質から $\mathbb{Z}F_n \to \mathbb{Z}F_n$ の線形写像 ($\mathbb{Z}$-準同形写像) になります. そして, 語の積に対しては普通の積の微分公式 (Leibniz 則) に似た性質を持ちます. なお, ここで現われる函数 $t : \mathbb{Z}F_n \to Z$ は群 F_n の各生成元に 1 を代入する線形写像です. これらの基本性質から Fox の微分子が群 F_n の元 x に対して次の性質を持つことが容易に確認できます:

Fox の微分子 Δ の性質

- $\Delta(m) = 0 \quad m \in \mathbb{Z}$
- $\Delta(x^m) = \sum_{i=0}^{m-1} x^i \Delta(x) \quad m \in \mathbb{N}$
- $\Delta(x^{-m}) = -\sum_{i=-m}^{-1} x^i \Delta(x) \quad m \in \mathbb{N}$

第一の性質は自然数 n に対しては 0 となるというもので, これは通常の微分と同様の性質です. 第二と第三の性質は $x \in F_n$ の冪の計算方法を与えるものです. では Maxima を使って, この Fox の微分子を表現してみましょう.

14.5　Maxima で遊ぶ Fox の微分子

14.5.1　函数 t の表現

Fox の微分子を Maxima で表現するために Fox 微分の積の性質で出て来た函数 t を Maximade 定義することにしましょう. なお, 函数名が ‘t’ のままでは通常の変数 ‘t’ と紛らわしいので, 函数名を ‘t1’ とします.

さて, この函数 ‘t1’ はどのような性質を充さなければならないでしょうか? まず, この函数は線形性を持っています. だから語に対して函数を定義しておき, declare 函数を使って線形性を函数 t1 の

属性として付与すれば十分です. そして, この函数は群 F_n の生成元を全て 1 で置き換える函数です. そこで, 次に示す Maxima の函数 t1 として函数 t を定義します:

函数 t1

```
t1(x):=block([vars:listofvars(x),n,i],
              n:length(vars),
              for i in vars do
                 (x:subst(1,i,x)),
                  return(x));
```

この函数 t1 は引数として与えられた語の全ての (自由) 変数に 1 を代入します. この変数の抽出では listofvars 函数 を用います. この listofvars 函数は与えられた語を演算子で分解し, 結果をリストとして返却する函数です. こうして得られた変数リストを用いて代入を行えば良い訳で, そのために反復処理のために do 文を, 代入の処理のために subst 函数を用います. ここで引数が語の場合に 1 を返すように函数 t1 を定義し, それから, declare 函数を使って linear 属性を付与するだけでも十分ですが, ここでは subst 函数と listofvars 函数を使いたかったのであえて複雑にしています.

14.5.2 群環 $\mathbb{Z}F$ の表現

この函数 t1 の値域となる群環 $\mathbb{Z}F_n$ は Maxima でどのように表現すればよいでしょうか？ まず, 和 “+” は Maxima の通常の和演算子 “+” をそのまま用いても良いでしょう. では, 群 F_n の積演算子 “*” はどうしましょう？ Maxima の積演算子 “*” は可換演算子で, F_n の演算子は通常は非可換です. そこで, 群 F_n の積 “*” として Maxima の非可換積である dot 積 “.”, 冪を非可換積の冪 “^^” を用いることにしましょう. この方法の長所は新たに非可換積と非可換積の冪の二つを定義する手間が省けることです. 一方の問題点は扱う式に行列が混っていると勝手に行列演算になることや, Maxima 付属のパッケージによっては dot 積に影響を与える大域変数 dotassoc を勝手に ‘false’ にして以後の処理が難しくなる可能性が挙げられます. それから整数と語の積に対しても非可換積演算子 “.” は使えます. しかし, 整数の場合は小数点と紛らわしいため, 整数と語の積については Maxima の可換積 “*” を用いることにします. それから最後に語同士の和は演算子 “+” を用いることにします. 以上から Maxima は和 “+”, 非可換積 “.” と可換積 “*” が混在した式を扱わなければなりませんが, このような式の計算を Maxima は難無く実行します. このことを実際に確認しておきましょう:

```
(%i76) 2*x.y+3*x.y;
(%o76)                                5 (x . y)
(%i77) 2*x.y.z+3*x.y;
(%o77)                      2 (x . y . z) + 3 (x . y)
```

この例に示すように全く問題がありません. なお, 独自に積演算子を定義してももちろん構いませんが, その場合は積演算子の型に注意する必要があります.

14.5.3 真理函数 wordp の構成

では, いよいよ Fox 微分の定義と行きたいところですが, その前に与式が語であるかどうかを判別する真理函数を定義しなければなりません. ここで語は先程, 定めた非可換積演算子 “.” を使って帰納的に構成された対象です:

語の構成方法

1. 原子 a は語である
2. 原子 a と語 w の非可換積 $a.w$ は語である
3. 整数 n と語 w の可換積 $n*w$ は語である
4. 語 w と整数 n の非可換積の冪 w ^^ n は語である

このことから語を判定する真理函数は 1. から 4. の構成方法に対して真を返す函数でなければなりませんが, 1. の場合は簡単で, 真理函数 atom で判断すれば良いことになります. そして, 2. から 4. については与式の内部表現 (木構造) を用いれば容易な問題となります. このことを形式的な内部表現を使って解説しましょう. まず 2. の場合, その内部表現は形式的に '(. a w)' の前置表現となります. 以降, 同様に 3. の場合は '(* n w)', そして, 4. の場合は '(^^ w n)' といった前置表現で表現されます. このことから与式が語となるためには, その与式が原子であるか, 与式の内部表現の演算子が可換積 "*", 非可換積 ".", あるいは, 非可換積の冪 "^^" のいずれかでなければなりません. それから, 被演算子が全て語であることを確認すればよいのですが, ここで入力式は Wirtinger 表現の関係子なので, 非可換積項とその非可換冪に限定され, Fox の微分子による処理で現われる式も演算子 "+" と可換積 "*" が現れるだけです. このことから内部表現のリストの指定した成分を Maxima の対象として返す函数 inpart を用いて次の判定条件に弱めても問題がないことが判ります:

与式 w が語となるための判定条件

1. atom(w) が true
2. inpart(w,0) が可換積 "*"
3. inpart(w,0) が非可換積 "."
4. inpart(w,0) が非可換積の冪 "^^"

この判定条件に基づく Maxima の真理函数を wordp とします. この wordp 函数は最初に与式が原子であるかどうかを調べ, 原子であれば 'true', そうでなければ与式の内部表現の第 1 成分を取出して, それが非可換積積 ".", 可換積 "*", 非可換冪 "^^" の何れかであれば 'true' を返し, それ以外は 'false' であれば良い訳です:

真理函数 wordp

```
wordp(w) :=
      if atom(w) then true
      else if member(inpart(w,0),[".","\^{}\^{}","*"]) then true
      else false;
```

この wordp 函数は inpart 函数から, 'inpart(w,0)' で与式の内部表現の第 1 成分を取出し, member 函数により, 与式の内部表現の第 1 成分が非可換積 ".", 非可換積の冪 "^^" か可換積 "*" の何れかであれば 'true' を返す仕様です.

では早速, この函数を試してみましょう:

```
(%i59) wordp(w);
(%o59)                                true
(%i60) wordp(2*w.z);
(%o60)                                true
(%i61) wordp(w.z);
(%o61)                                true
(%i62) wordp(w.z+2*x.y);
(%o62)                                false
```

このようにして与式が語であるかを検証する真理函数ができました. この真理函数が使えることで, 真理函数が真となる対象のみに処理を行うこと, すなわち, 規則が定義可能となります.

14.5.4 Fox の微分子の構成

属性の付与

ここで Fox の微分子 “D_fox:” を Maxima で表現しましょう. まず最初に演算子名 “D_fox:” に演算子属性を付与します. この演算子は引数の前に置くことを想定しているので prefix 函数 か nary 函数で定義することになります. ところで, nary 函数は名前から類推出来るように n-ary(n-変数) の函数で, prefix 函数とは勝手が異なります. この本ではあとの処理の利便性から prefix 函数として定義します.

ここで語のみに対する演算子を定義するだけでなら prefix(”D_fox:”) で十分ですが, 実際は群環が相手なので和 “+” や自然数との可換積 “*” が入り組んだ式が相手となります. そこで, 演算子と被演算子を結び付ける力, すなわち, 演算子の束縛力もここで一緒に定めましょう. この演算子の束縛力とは, たとえば, 数式 $1+2\cdot 3^4-4$ が与えられたときに, 数式を構成する和, 積, 差と冪等の演算に順位があるために, この式は $(1+(2\cdot(3^4)))-4$ と解釈されます. このように演算子が被演算子を引き付けておく力のことを Maxima では束縛力と呼び, その強さを整数で表現しています. Maxima では和 “+” が 100, 可換積 “*” の左束縛力が 120, 非可換積 “.” の左束縛力が 130 で右束縛力が 129, 非可換積の冪 “^^” の左束縛力が 140 で右束縛力が 139 と予め指定されており, 何も指定しなければ既定値として 180 が付与されます. もし, Fox 演算子 “D_fox:” に束縛力を指定しなければ, 既定値の 180 が与えられますが, この値は非可換冪の束縛力 140 よりも大きいために, 式 ‘D_fox:x^^5’ は ‘(D_fox:x)^^5’ と Maxima は解釈します. そうではなく ‘x^^5’ 全体に演算子 “D_fox:” を作用させたければ ‘D_fox:(x^^5)’ のよううに小括弧 “()” で括らなければなりません. そんなことを一々するよりは演算子 “D_fox:” の束縛力を演算子 “^^” よりも小さくする方が無難です. そこで右束縛力を 128 とします. なお, 演算子の属性の付与を行う前に, その演算子を具体的に記述する必要は全くなくて形式的なもので十分です.

この演算子の属性の付与に続けて今度は線形性 ‘$D(x+y)=D(x)+D(y)$’ も追加しましょう. この線形性の付与には declare 函数を用います:

```
(%i1) prefix(”D_fox:”,128);
(%o1)                                 D_fox:
```

```
(%i2) declare("D_fox:",linear);
(%o2)                                    done
```

これで演算子 “D_fox:” の前置演算子としての属性と線形性が付与されました. ここで属性 linear を持つ演算子 “f” については ‘f (x+x+x)=f(x)+f(x)+f(x)’ を充しますが, 右辺は ‘3*f(x)’ なので, 整数 n に対して ‘f (n*x) =n * f x’ を自動的に充すことになります. つまり, これらの性質によって $a \in \mathbb{Z}F_n$ に対して 和 “+” と可換積 “*” を含まない項, つまり, 整数 $\mathbb{Z}$ と F_n の元の処理を定めれば良いことになりますが, 整数は Fox 演算子の性質から 0 となるので, 結局, F_n の元, つまり, 語の処理だけを考えれば良いことになります.

規則の設定

ここでは和演算子 “+” と可換積演算子 “*” を持たない語そのものの処理, すなわち, $w_1.w_2 \in F_n$ に対して ‘$\Delta(w_1, w_2) = t(w_2)\Delta(w_1) + w_1\Delta(w_2)$’ を Maxima の演算子 “D_fox:” で実現することです. この性質を実現するためには与えられた与式を分解して群 F_n の元だけに対して処理を行わなければなりません. このような操作を行うために Maxima には**規則**と呼ばれる仕組があります. この帰属では, まず引数がその変換を適用すべき対象であるかどうかを判断し, 該当する対象であれば式の変換を行います. この判断では以前定義した wordp を利用します.

ここで Maxima の規則を利用するためには次に示す手順に沿って作業を進める必要があります:

Maxima で規則を利用するために必要なこと

1. 規則を記述する際に用いる変数の宣言
2. 規則の定義

第一の処理で変数の宣言があります. この変数の宣言は, あとで定める規則の定義で用いるための変数で, この変数のことを「**並びの変数**」と呼びます. 数学の公式集等で変数を $x, y, z, \ldots$ と記述するのと似た目的と思って良いでしょう Maxima 内部では並びの変数とその変数が満すべき述語を matchdeclare 函数に引渡し, 各変数の matchdeclare 属性値として述語に紐付けします. 次の規則の定義では, この並びの変数を使って規則を適用すべき項と適用によって得られる式, 要するに公式を Maxima に教えますが, そのときに defrule 等の函数を用います.

では実際に規則を与えてみましょう. ここでは二つの語で構成された非可換積項 ‘w1.w2’ に対して Fox の微分子 “D_fox:” を作用させると ‘D_fox:w1 * t1(w2) + w1 . D_fox:w2’ になることを, 最初に matchdeclare 函数で, 規則に用いる変数と, その変数に割当てられた値が充すべき性質を持つかどうかを判別する真理函数を与えます. それから, matchdeclare で宣言した変数を使って ‘D_fox: (w1 . w2)’ が ‘D_fox:w1 * t1(w2) + w1 . D_fox:w2’ に変形されることを記述します:

```
(%i3) matchdeclare([_x,_y],wordp);
(%o3)                                    done
(%i4) defrule(Dfox_Prod,D_fox:(_x._y),
                          D_fox:_x*t1(_y)+_x.D_fox:_y);
(%o4) Dfox_Prod : D_fox: (_x . _y) ->
```

```
D_fox:_x  t1(_y)+_x.D_fox:_y
```

この例は重要なので詳細に解説しておきます. 最初に matchdeclare 函数を使って並びの変数を定義しています. Maxima の規則は matchdeclare 函数で宣言した並びの変数を用いて定義を行う必要があります. この理由は, 規則を適用する際に条件を満す式であるかを Maxima が属性として付与された述語を用いて検証するためです. もしも, この宣言がなければ, 規則の適用は規則の定義で用いた変数に対する適用に限定されて一般性を失ないます. なお, 並びの変数が満すべき条件が不要であれば, 述語として 'true' を与えます. ここでは並びの変数を_x と_y の二つとし, それらが満すべき述語は wordp 函数で表現されている, すなわち, 並びの変数に束縛された対象は語であるとしています.
次に規則の定義を行いますが, ここでは並びの変数を用いて変換すべき式と変換後の式を記述して規則を定義します. この規則の定義では defrule 函数に規則名, 規則を適用する式, 規則の適用後の式を引数として与えます. ここでの例では規則名が "Dfox_Prod" で, この規則 Dfox_Prod によって 'D_fox:(_x . _y)' が与えられると 'D_fox:_x * t1(_y) + _x . D_fox:_y' に置換するという処理を行うという規則を定めています. ここで並びの変数_x, _y は語, すなわち, 述語 wordp を作用させると 'true' を返す対象であるということを前提となっています. それ以外の式が与えらえた場合は, 演算子 "D_fox:" の線形性で式が展開されることになります.
このようにして非可換積に対する微分の性質が付加されました.

ここで注意しなければならないことは, ここでの定義で二項の非可換積に対して, その左辺と右辺に分けて処理を与えていることです. そのため, 被演算子が右側と左側で抽出できなくなると, この処理は間違った計算を行う可能性があります. このような現象が生じるのは, 非可換積の演算子が infix 型ではなく nary 型の場合ですが, infix 型の演算子である非可換積 "." についても, その演算子に影響を与える大域変数 dotassoc を既定値の 'true' のままにしていると, 式 '(x . y) . z' は自動的に 'x . y . z' と平坦化してしまい, ここで定めた規則の適用が難しくなります. この理由は非可換積に与えられた規則によって nary 型の演算のように自動的に内部表現が形式的に '(. x y z)' で置換されるためです. ここでは非可換積 "." を用いているので, 結び目群を定義する前に 'dotassoc:false' で大域変数 dotassoc に 'false' を割当て, この処理を回避する必要があります.

与式が定数や非可換冪の場合の規則

項として残っているのは整数の場合と非可換冪の項の場合のみです. まず, 与式が定数の場合に Fox の微分子は 0 を返します. ここで環は整数環 $\mathbb{Z}$ なので定数は integer として仮定して構いません. Maxima には整数であれば 'true' を返す integerp 函数があるので, この函数が真となる対象に対する対する規則として次の規則を入れます:

```
(%i5) matchdeclare(_a,integerp);
(%o5)                                done
(%i6) defrule(Dfox_Const,D_fox:_a,0);
(%o6)                    Dfox_Const : D_fox: _a -> 0
```

このようにして対象が整数の場合に 0 で置き換える規則 Dfox_const を定めました. すると残りは与式が非可換冪の場合の処理だけです. この非可換冪の処理では, 次数が正整数の場合と負の整数

の場合に分けて処理を定める必要がるので posintp 函数と negintp 函数の二つの述語を定義しましょう:

posintp 函数と negintp 函数

```
posintp(x):= if featurep(x,integer) then
                if is(x>0) then true;
negintp(x):= if featurep(x,integer) then
                if is(x<0) then true;
```

ここで定義した posintp 函数と negintp 函数は同じ操作を行う函数です. 処理の内容は, featurep 函数を使って整数属性 integer を持つかを判断し, 'true' であれば正負の判定を行えば良いのです. これらの函数の動作を確認しておきましょう:

```
(%i11) declare(n1,integer);
(%o11)                                done
(%i12) assume(n1>0);
(%o12)                              [n1 > 0]
(%i13) posintp(n1);
(%o13)                                true
(%i14) declare(n2,integer);
(%o14)                                done
(%i15) assume(n2<0);
(%o15)                              [n2 < 0]
(%i16) negintp(n2);
(%o16)                                true
(%i17) negintp(n1);
(%o17)                                false
```

そして最後に非可換冪項の次数が正の冪の場合と負の冪の場合の展開規則を演算子 "D_fox:" に付与します. この規則の与え方は先程の非可換積項の展開規則 "Dfox_Prod" と同様です. 最初に規則を与える変数を matchdeclare 函数を用いて宣言し, その変数を用いて defrule 函数を使って規則を定義します:

```
(%i9)matchdeclare(_ap,posintp)$
(%i10)matchdeclare(_an,negintp)$
(%i11)defrule(Dfox_PPower, D_fox:(_x^^(_ap)),
              sum(_x^^(i),i,0,_ap-1).D_fox:_x);
(%o11) Dfox_PPower : D_fox: (_x      )
              -> sum(_x    , i, 0, _ap - 1) . D_fox: _x
(%i12)defrule(Dfox_NPower, D_fox:(_x^^(_an)),
             -sum(_x^^(i),i,_an,-1).D_fox:_x);
(%o12) Dfox_NPower : D_fox: (_x      )
              -> - sum(_x    , i, _an, - 1) . D_fox: _x
```

この例では, 冪が正の場合は規則 'Dfox_PPower', 冪が負の場合に規則 'Dfpx_NPower' を適応するというものです.

以上で Fox の微分子を利用するために必要な規則が全て揃いました.

規則の適用

規則は全て揃いましたが実際にどのように使えばよいのでしょうか？ここでは $xyz^{-1}y^{-1} \in F_n$ に対応する Maxima の式 'x . y . z^^(-1).y^^(-1)' に Fox 微分演算子をそのまま作用させてみましょう:

```
(%i8) r1:(x . (y . (z^^(-1) . y^^(-1))));
                                        <- 1>     <- 1>
(%8)                          x . y . z       . y
(%i9) dfr1:D_fox:(r1);
                                          <- 1>     <- 1>
(%o9)                  D_fox: (x . (y . (z       . y      )))
```

このように Fox 微分演算子の項がそのまま返されるだけで規則が適用されていません. Maxima では規則を定義するだけでは意味がなく, その規則を式に適用させる函数が別途必要なのです. 規則を作用させる函数を Maxima は幾つか持っており, その中から今回は apply1 函数を使うことにします. この例の計算では, Fox 微分演算子の計算に必要な規則は語の非可換積の展開規則 Dfox_Prod と負の冪の規則 Dfox_NPower の二つです. このように複数の規則を利用する場合, apply1 に適用する規則を適用する順番で並べます. ここでは最初に展開規則を用いて展開を行い, それから負の冪の展開規則を用いてみましょう:

```
(%i11) apply1(r1,Dfox_Prod,Dfox_NPower);
                    <- 1>                <- 1>      <- 1>
(%o11) x . (y . (- z       . D_fox: z - z       . (y       . D_fox: y))
                          + D_fox: y) + D_fox: x
```

この例では $xyz^{-1}y^{-1}$ の第一項 x とその他の項 $yz^{-1}y^{-1}$ の非可換積として Fox 微分演算子の積規則を適用して $\Delta x + x\Delta y(z^{-1}y^{-1})$ を得ますが, この式の第二項にも積規則が適用可能な項が現われるために積規則が再び適用されて $\Delta x + x(\Delta y + y\Delta z^{-1}y^{-1})$ が積規則による展開で得られます. 次に負の冪規則による展開を行いますが, この式の第 3 項に $\Delta z^{-1}y^{-1}$ があるため, この項のみに負の冪規則が適用されて最終的な結果が得られます. ここでの例で示したように展開規則の順番も重要です. 簡単に確認しておきましょう:

```
(%i16) apply1(D_fox:r1,Dfox_Prod);
                  <- 1>            <- 1>            <- 1>
(%o16) x . (y . (z       . D_fox: y       + D_fox: z      ) + D_fox: y) + D_fox: x
(%i17) apply1(D_fox:r1,Dfox_NPower);
                                          <- 1>     <- 1>
(%o17)                  D_fox: x . (y . (z       . y      ))
(%i18) apply1(D_fox:r1,Dfox_NPower,Dfox_Prod);
                  <- 1>            <- 1>            <- 1>
(%o18) x . (y . (z       . D_fox: y       + D_fox: z      ) + D_fox: y) + D_fox: x
```

この例では積規則のみ, 負の冪規則のみ, 負の冪規則を適用したあとで積規則を適用しています. 負の冪規則は $xyz^{-1}y^{-1}$ では対応しないためにそのままの式が返却され, その結果, 積規則のみの展開と負の冪規則の次に積規則を適用した結果は同じ結果となります. このように規則を適用する順番についても注意して下さい.

14.6　プログラムファイルの構成

以上の Fox の微分子や規則の定義等を纏めてファイル fox.mc に記入しておきましょう. このファイルの中に注釈を入れておくとあとで読み易く, 管理を行うのも容易になります. Maxima のプログラムファイルでの注釈は C 風に “/*” と “*/” の間に記述します. このファイル fox.mc の内容を以下に示しておきましょう:

fox.mc

```
/* MAXIMA */
/* 函数t1

Fox微分で現われる函数. 線形性を持ち, 語を1に写す函数.
語の変数はlistofvars函数で取出し, 全ての変数に1を代入する.
代入はsubst函数を用い, 線形性属性の付与はdeclare函数を利用する.
*/
t1(x):=block([vars:listofvars(x),n,i],
              n:length(vars),
              for i in vars do
                 (x:subst(1,i,x)),
              return(x));

declare(t1,linear);

/* Foxの微分子の定義.

演算子はprefix函数を用いて前置式演算子としての属性を与えます.
なお, 被演算子の右束縛力は128とし, 非可換冪よりも小さな値にします.
さらに線形性属性はdeclare函数で与えます.
*/

prefix("D_fox:",128);
declare("D_fox:",linear);

/* Foxの微分子の置換規則で用いる変数宣言と真偽函数定義  */

/* _xと_yを語とします. */
matchdeclare([_x,_y],wordp);

/* 語の述語 */
wordp(w) :=
if atom(w) then true
else if member(inpart(w,0),[".","^^","*"]) then true
else false;

/* _aを整数とします. 述語はMaxima組込のintegerpを用います. */
matchdeclare(_a,integerp);

/* _apを正整数, _anを負の整数とします.ここで述語は,
正整数はposintp,負整数はnegintpを用います.*/
matchdeclare(_ap,posintp);
matchdeclare(_an,negintp);

/* 語の述語 */
```

```
46 posintp(x):= if featurep(x,integer) then
47                 if is(x>0) then true;
48 negintp(x):= if featurep(x,integer) then
49                 if is(x<0) then true;
50
51 /* 積規則 Dfox_Prod */
52
53 defrule(Dfox_Prod,D_fox:(_x._y),D_fox:_x*t1(_y)+_x.D_fox:_y);
54
55 /* 定数に対する規則Dfox_Const */
56 defrule(Dfox_Const,D_fox:_a,0);
57
58 /* 正の冪に対する規則Dfox_PPower */
59 defrule(Dfox_PPower, D_fox:(_x^^(_ap)),
60                       sum(_x^^(i),i,0,_ap-1).D_fox:_x);
61
62 /* 負の冪に対する規則Dfox_NPower */
63 defrule(Dfox_NPower, D_fox:(_x^^(_an)),
64                      -sum(_x^^(i),i,_an,-1).D_fox:_x);
65
66 /* 非可換積の結合律の適用を阻止. 積規則を利用する為に必要 */
67 dotassoc:false;
```

このファイルの演算子, 函数や規則を用いたければ load 函数を用いて load(”fox.mc”); と入力します. さらに, Maxima の起動時に直ちに使いたければ Maxima を起動するディレクトリに maxima-init.mac という名前のファイルを作って ‘load(”fox.mc”);’ を記入しておきます. それから maxima-init.mac があるディレクトリ上で maxima を起動すると自動的に maxima-init.mac 内部が解釈されて fox.mc が読込まれます.

これで結び目の Alexander 多項式を計算するための道具が漸く揃いました.

14.7 Alexander 多項式

14.7.1 Alexander 行列の計算

Alexander 多項式は Alexander 行列と呼ばれる行列から計算されます. この Alexander 行列は結び目群 $G(K)$ の関係子で構成されたベクトル $(r_1,\ldots,r_n)$ に対して通常の微分の代わりに Fox 微分演算子で計算した Jacobian が対応します.

この計算をもう少し具体的に解説しましょう. 普通の微分では変数 x に対する微分 d/dx を用いますが, これは Fox の微分子も同様で, 結び目群 $G(K)$ の Wirtinger 表現の生成元 $x_i\ i\in(1,\ldots,n)$ に対して $\partial/\partial x_i$ を今迄議論した演算子 Δ の性質に加え, $\partial x_j/\partial x_i=\delta_{ij}$ で定めます. たとえば星型結び目の生成元は x,y,z,v,w の 5 個あるので, Fox 微分演算子は $\partial/\partial x,\ldots,\partial/\partial w$ と生成元の数だけ存在します.
ところで, 最初に議論した Fox の微分子 Δ とそれを Maxima 上で定義した演算子 “D_fox:” では生成元の情報がありませんね. 何故そうしたのでしょうか？この理由は単純に, その方が便利だから

です. 実際, 結び目が複雑になれば演算子も生成元の数だけ増大するので, 演算子を生成元に対して定義する手間が増えます. ところが, 演算子 "D_fox:" の定義では微分としての共通の性質だけで演算子を定め, その微分に対して積, 冪といった規則を交互に適用すれば, 最終的に Δx_i に相当する微分項のみを含む式に変換できます. その式に対して $\partial x_i/\partial x_j = \delta_{ij}$ という関係を適用してしまえば生成元 x_j に対応する Fox の微分子 $\partial/\partial x_j$ による計算結果が得られる訳です. さらに関係子で構成された行ベクトル $(r_1, \ldots, r_n)$ に Fox 微分演算子 Δ を作用させた行ベクトル $\delta R = (\Delta r_1, \ldots, \Delta r_n)$ を計算します. そして, $j \in (1, \ldots, n)$ を固定して $\partial x_i/\partial x_j = \delta_{ij}\ i \in (1, \ldots, n)$ を適用し, ベクトル内の式中に残った生成元を全て t で置換し, 非可換積と冪を可換積と可換積の冪で置き換えた行ベクトルを δR_j とします. すると, これらの行ベクトルを積み上げた行列 $\begin{pmatrix} \delta R_1 \\ \vdots \\ \delta R_n \end{pmatrix}$ が求める Jacobian, すなわち Alexander 行列となります. この手法では生成元毎の規則や演算子の定義が不要なので全体の処理が結び目の生成元の数に依存せず, そのお陰で非常に機械的に行なえるという非常に大きな長所があるのです.

14.7.2　Alexander 多項式の計算

結び目群 $G(K)$ の Alexander 多項式は, 結び目群 $G(K)$ の Alexander 行列の余因子行列 $\tilde{A}$ の各成分の最大公約因子です. より正確には 1 次の Alexander 多項式ですが, 結び目理論では単純に Alexander 多項式と呼びます. この Alexander 多項式は Laurant 多項式環 $\mathbb{Z}[t^{-1}, t]$ のイデアルの生成元になります. そのために Alexander 行列の計算結果に $t^n, n \in \mathbb{Z}$ の違いが生じることがありますが, Alexander 多項式として定数項が零にならないように各項の次数が正となるものを選択すると一意に定まります.

これらの処理をプログラム calcAlexanderPoly に纏め, このプログラムを収録したファイル AlexanderPoly.mc の内容を次に示しておきましょう:

AlexanderPoly.mc

```
1  /* MAXIMA */
2
3  /* calcAlexanderPoly
4
5  fox.mcと併用する事が前提です.
6
7  結び目Gはリストで表現します.
8  すなわち G=<x1,...,xn| r1,...,rm>をG:[[x1,...,xn],[r1,...,rm]]
9  で表現します.
10
11 calcAlexanderMatrixでは,Wirtinger表示の結び目群に対して
12 Alexander行列の計算を行います. そのために関係子r1,..,rmの
13 個数mはnかn−1に等しくなければなりません. したがって, 関係子の
14 個数がnかn−1に等しくなければエラーになります.
15 */
16 calcAlexanderPoly(G):=
17 block([vars:G[1],rels:G[2],amat,Alex:[],
```

```
18          Ia,dfx,rdfx,rdfx,crdfx,mrow,n:length(G[1]),
19          m:length(G[2]),AlexanderPoly,APolyData:false,
20          tmp,lst:[]],
21          /* 結び目群のWirtinger表示に限定 */
22          if m=n or m=n-1 then (
23            Ia:subst("[",matrix,ident(n)),
24            for i from 1 thru n do lst:append(lst,[i]),
25            dfx:map("D_fox:",vars),
26            /* 関係子リストに演算子を作用 */
27            rdfx:map(lambda([x],apply1(D_fox:x,Dfox_Prod,
28                                       Dfox_PPower,Dfox_NPower)),rels),
29            /* 微分に1か0を設定 */
30            for i from 1 thru n do(
31                mrow:Ia[i],rdfx2:rdfx,
32                for j from 1 thru n do (
33                    rdfx2:map(lambda([x],subst(mrow[j],dfx[j],x)),rdfx2) ),
34                /* 非可換積の冪を可換積の冪に変換 */
35                crdfx:map(lambda([x], subst("^","^^",x)),rdfx2),
36                /* 非可換積を可換積に変換し,簡易化を実施 */
37                amat[i]:ratsimp(map(lambda([x],subst("*",".",x)),crdfx))),
38            /* Alexander行列を配列として最初に構築. */
39            for i from 1 thru n do Alex:append(Alex,[amat[i]]),
40            /* 配列データから行列データに変換 */
41            Alex:substpart(matrix,Alex,0),
42            /* 変数をtに変換します */
43            for i in vars do Alex:subst(t,i,Alex),
44            Alex:ratsimp(Alex),
45            /* Wirtinger表現で関係子の総数が生成元の個数nよりも
46               一つ少ない場合の処理.Maximaの組込函数では余因子
47               行列の生成は正方行列に限定されます.
48            */
49            if m=n-1 then (
50               /* nx1の零行列をn列目に追加. */
51               tmp:addcol(Alex,zeromatrix(n,1)),
52               /* 小行列式を計算. */
53               tmp:map(lambda([x],
54                       determinant(minor(tmp,x,n))),lst) )
55             else
56               tmp:ratsimp(adjoint(Alex)),
57            /* 余因子行列をリストに変換. */
58            tmp:substpart("[",tmp,0),
59            /* LGCDでAlexander多項式を抽出 */
60            AlexanderPoly:num(LGCD(map(LGCD,tmp))),
61            /* Alexander行列と多項式のリストを生成. */
62            APolyData:[Alex,AlexanderPoly] )
63        else
64            /* エラー処理.Error!と表示するだけのシンプルなもの */
65            print("Error!"),
66        /* Alexander行列を返します. */
67        return(APolyData) )$
68  /* リストから最大公約因子を計算します.
69
70  Lp: 多項式,或いは,整数で構成されるリスト.
71
```

```
72 長さが1の場合，その成分をそのまま返します．
73 長さが2以上の場合，最小次数の多項式を一つ取り出し，その
74 多項式と残りの多項式の最大公約因子のリストを求め，以降，
75 再帰的にLGCDを呼出して最大公約因子を計算します．
76 */
77 LGCD(Lp):=
78 block(
79       [a1,n,lgcd:false,rLp:[]],
80        /* Lpがリストであるかを確認． */
81        if listp(Lp) then (
82           /* Lpの長さを求めます． */
83           n:length(Lp),
84           /* Lpの長さが1の場合，成分をそのまま返します． */
85           if n=1 then  lgcd:Lp[1]
86           else (
87              plst:map(lambda([x],hipow(x,t)),Lp),
88              if member(0,plst)=false then (
89                 j:1, m:substpart(min,plst,0),
90                 for i in plst do (
91                     if i=m then (idx:j,break) else j:j+1 ),
92                 j:0,
93                 for i in Lp do (
94                      j:j+1,
95                      if j#idx then rLp:append(rLp,[gcd(i,Lp[idx])]) ),
96                 lgcd:LGCD(rLp) )
97            else lgcd:1 ) ),
98        return(lgcd) )$
```

ここで紹介する AlexanderPoly.mc ファイルは calcAlexanderPoly 函数と LGCD 函数の二つで構成されています．そして，calcAlexanderPoly 函数が Alexander 行列と多項式をリスト形式で返却します．

この calcAlexanderPoly 函数内部の処理を順番に解説しましょう．まず，与えられた結び目群から生成元と関係子を取出して，生成元の個数 n を求め，その生成元に対応する大きさの単位行列 I_n から行ベクトルのリスト Ia を生成します．それから生成元に対しては演算子 “D_fox:” を map 函数で作用させることで D_fox :x_i $i \in (1,\ldots,n)$ のリストを生成します．このリストと Ia の k 成分のリストは k 番目の成分だけが 1，その他は 0 となり，のちの D_fox :x_i の値の代入で利用します．

結び目群の関係子に対しては三つの規則 Dfox_Prod, Dfox_PPower と Dfox_NPower を順番に apply1 函数を使って作用させる無名函数を lambda 函数を用いて構成し，この無名函数を map 函数を使って関係子の行ベクトルを作用させて Jacobian の雛形を構築します．

次に実際に Alexander 行列の計算に入ります．考え方は Jacobian の雛形である行ベクトル rdfx に対して $\partial x_i/\partial x_j \to \delta_{ij}$ による変換を行い，非可換積と冪を可換積と通常の冪に置き換える作業を行います．具体的には，Jacobian の i 行は x_i による微分であるために $\partial x_k/\partial x_i \to \delta_{ki}$ という代入を $k \in (1,\ldots,n)$ に対して行うことになります．この代入は subst 函数でリスト Ia の i 成分のリストの値を順番に入れることでこの変換を実現させています．それから非可換積を可換積，非可換積の冪

を可換積の冪へと変換しますが, この変換では substpart 函数を用います. このときに Maxima の演算子は式の表現で 0 番目の位置に置かれるために 0 番目の成分を入れ換えるようにしています. ここでも相手がリストのために map 函数を効果的に適用できるように lambda 函数を使って無名函数を定義します. それからリストの形式から行列データに substpart 函数を用いて変換し, 各変数を全て t に置換えると Alexander 行列が出来ます.

それから Alexander 行列の余因子行列を計算します. この余因子行列の計算では adjoint 函数や小行列を求める minor 函数を用いますが, どちらも正方行列にしか使えません. ところが Wirtinger 表示では生成元の個数が関係子の個数よりも一つ少なくても群の表示に問題がないために, そのような群の表示が与えられると問題が生じる可能性があります. そこで生成元が関係子の個数よりも一つ少ないときに Alexander 行列の n 列目に n 次の零ベクトルを addcol 函数を使って追加して胡麻化しておきましょう. それから minor 函数で Alexander 行列 Alex の小行列 $\mathrm{Alex}_{1\leq i\leq n,n}$ を minor 函数を用いて計算し, determinant 函数を map 函数で作用させて必要な余因子行列が計算できます. それから substpart 函数を用いて行列からリストに変換してから多項式の最大公約因子を求めるために LGCD 函数を使います.

この LGCD 函数はリストに含まれる各成分の最大公約因子を計算する函数で, リストの長さが 1 であればリストに含まれる式をそのまま返して終了し, 長さが 2 以上であればリストに含まれる多項式の最小次数を求め, 最小次数が 0 のものが含まれていれば 1 を返して終了し, そうでない場合は最小次数の多項式を一つ取り出して, その他の多項式との最大公約因子のリストを構成します. そして, リストを再び LGCD 函数に与えるという再帰的な手法を用いています. この手法の計算量は $n!$ のオーダーに比例するもので, この点について工夫の余地が残っています. それから最後に Alexander 行列と Alexander 多項式のリストを返却して処理を終えます.

では次の節で Alexander 多項式を計算してみることにしましょう. ここで計算を始めるにあたって, 予め 'load("fox.mc")' と 'load("AlexanderPoly.mc")' で Maxima にプログラムの読込を実行しておいて下さい.

14.8 Alexander 多項式で結び目を分類しよう

星型結び目 (5_1)

最初に星型結び目の Alexander 多項式を計算しましょう. そのためには星型結び目の結び目群の表示が必要です. この結び目群の Wirtinger 表示を次に示しておきましょう:

星型結び目の結び目群

$$\langle v,w,x,y,z | xz^{-1}x^{-1}v, vx^{-1}v^{-1}y, yv^{-1}y^{-1}w, wy^{-1}w^{-1}z, zw^{-1}z^{-1}x\rangle$$

次に, この結び目群の表示をリストとして与えて calcAlexanderPoly で計算してゆく様子を示しておきます:

```
(%i14) star:[[x,y,z,v,w],
[x.z^^(-1).x^^(-1).v,v.x^^(-1). v^^(-1). y,
y.v^^(-1).y^^(-1). w,
w.y^^(-1). w^^(-1).z,
z.w^^(-1).z^^(-1).x]]$
(%i15) K5_1:calcAlexanderPoly(star)$
(%i16) K5_1[2];
                                  4     3     2
(%o16)                            t  - t  + t  - t + 1
(%i17)tex(K5_1[1]);
$$\pmatrix{{{t-1}\over{t}}&-1&0&0&{{1}\over{t}}\cr  0&{{1}\over{t}}&{{
 t-1}\over{t}}&-1&0\cr  -1&0&0&{{1}\over{t}}&{{t-1}\over{t}}\cr  {{1
 }\over{t}}&{{t-1}\over{t}}&-1&0&0\cr  0&0&{{1}\over{t}}&{{t-1}\over{t
 }}&-1\cr  }$$
(%o17)                                  false
```

最初なので細かく説明しておきましょう．最初に結び目群をリストの形式で与えます．このリストは複合リストで, 第一成分のリストに生成元 x, y, z, v, w を並べたリスト, 第二成分のリストが関係子 $xz^{-1}x^{-1}v, vx^{-1}v^{-1}y, yv^{-1}y^{-1}w, wy^{-1}w^{-1}z, zw^{-1}z^{-1}x$ を並べたリストとなります．この結び目群のリストによる表記は変数 star に割当てています．それから calcAlexanderPloy 函数に与えて結果は変数 K5_1 に代入します．calcAlexanderPoly 函数は第一成分が Alexander 行列, 第二成分が Alexander 多項式です．だから, 'K5_[2]' で Alexader 多項式を確認することができるのです．

さて, calcAlexanderPoly 函数の処理によって星型結び目の Alexander 多項式が $t^4 - t^3 + t^2 - t + 1$ であることが判りました．ここで自明な結び目の Alexander 多項式は 1 なので, このことから星型結び目は自明な結び目と違うことが判ります．如何でしょうか．結び目の分類もこのようにすれば Tietze 変換で群の表示を弄くり倒すよりは楽ですね．

なお, calcAlexadnerPoly 函数では Alexander 行列を返却値のリストの第一成分に保存しています．そこで, この第一成分をそのまま tex 函数に与えてみましょう．ここで tex 函数は入力式を TEX のソースに変換する函数です．この出力結果を TEX 形式のファイルに追加して, あとはコンパイルすれば以下の表示が得られます[1]:

星型結び目の Alexander 行列

$$\begin{pmatrix} \frac{t-1}{t} & -1 & 0 & 0 & \frac{1}{t} \\ 0 & \frac{1}{t} & \frac{t-1}{t} & -1 & 0 \\ -1 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} \\ \frac{1}{t} & \frac{t-1}{t} & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & -1 \end{pmatrix}$$

[1] ここでは center で中寄せしています

クローバー結び目 (3_1)

クローバー結び目は三葉結び目 (trefoil) と呼ばれ図 14.10 に示すように三枚の葉がある結び目です:

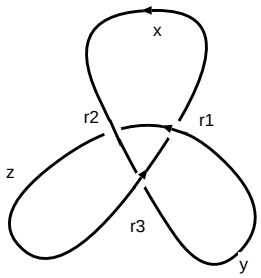


図 14.10: クローバー結び目 (3_1)

クローバー結び目の結び目群

$\langle x, y, z \mid xyz^{-1}y^{-1}, yzx^{-1}z^{-1}, zxy^{-1}x^{-1} \rangle$

```
(%i20) Trefoil:[[x,y,z],
[x.y.z^^(-1).y^^(-1),y.z.x^^(-1).z^^(-1),z.x.y^^(-1).x^^(-1)]]$
(%i21) calcAlexanderPoly(Trefoil);
                       [    1      - t    t - 1 ]
                       [                        ]    2
(%o21)                 [[ t - 1     1      - t  ], t  - t + 1]
                       [                        ]
                       [  - t     t - 1     1   ]
```

この結果からクローバー結び目の Alexander 多項式は t^2-1+1 であることが判ります. Alexander 多項式が 1 でないことからクローバー結び目は自明な結び目ではなく, 星型結び目の Alexander 多項式 $t^4-t^3+t^2-t+1$ とも異なることから, クローバー結び目は星型結び目とは別の結び目であることが判ります.

14.9　結び目の連結和と Alexander 多項式

二つの結び目が与えられたときに「**連結和**」と呼ばれる操作から新しい結び目を生成することができます:

結び目の連結和

1. 二つの結び目 K_1 と K_2 を用意し, 各結び目に向きを入れます.
2. 結び目 K_i 上の点 P_i を中心とする半径 r の球 $B_i(r), i \in \{1,2\}$ を取り出します. これらの球は結び目の一部を含みますが, この際 jni 半径 r を十分小さく取れば $B_i(r)$ をドーナッツのような形状にすることができます.
3. これらの球 $B_i(r)$ を結び目 $K_i, i \in \{1,2\}$ から結び目の曲線を含めて削除します.
4. 取り除いた個所で結び目の向きが一致するように結び目 K_1 と K_2 を繋ぎ合せます.

この操作で生成された結び目のことを, 結び目 K_1 と K_2 の「**連結和**」と呼び, この結び目を $K_1\#K_2$ と表記します. この連結和 "#" を入れることで結び目の集合に「**半群**」の構造が入ります. このとき, 単位元は自明な結び目で, 与えられた結び目が連結和で生成されたものでなければ**既約な結び目**と呼びます.

ここで 3_1 と 5_1 の連結和の Alexander 多項式を計算してみましょう. まず, $3_1\#5_1$ は図 14.11 に示す結び目になります:

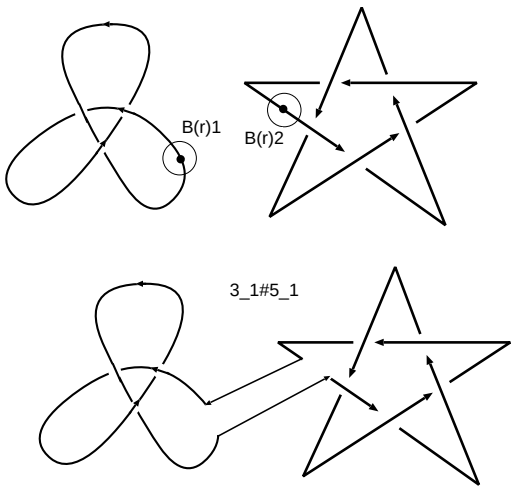


図 14.11: 結び目 3_1 と 5_1 との連結和

この計算で結び目 $3_1\#5_1$ の Wirtinger 表示を直接計算しても構いませんが, 前節で計算した 3_1 と 5_1 の Wirtinger 表示があるので, それらを利用しましょう：
$3_1\#5_1$ の場合, 図 14.12 に示すように K_1 側の結び目群はそのままにして K_2 側の変数名を変更して

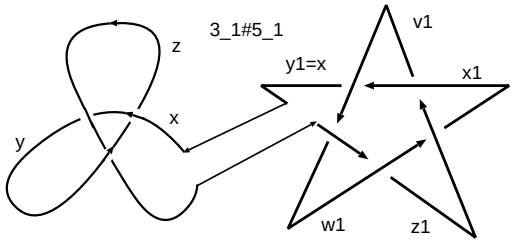


図 14.12: 結び目 3_1 と 5_1 との連結和の結び目群

おきます. そして, 繋げる個所の生成元に関係式を追加します. たとえば, この例では x と y_1 の道を繋げるので $x = y_1$. すなわち, xy_1^{-1} を追加します. この処理でちゃんと群の表示になっていることは **Van Kampen の定理**で保証されています. なお, calcAlexanderPoly は簡易的なプログラムなので関係子を一つ抜いておく必要があります. そのために k_2 側の本来の関係子を一つ抜いて, その代りに xy_1^{-1} を入れておきます.

$3_1\#5_1$ の結び目群

$$\left\langle\ x,y,z,x_1,y_1,z_1,v_1,w_1 \ \middle|\ \begin{array}{l} xyz^{-1}y^{-1}, yzx^{-1}z^{-1}, \\ zxy^{-1}x^{-1}, v_1x_1^{-1}v_1^{-1}y_1, y_1v_1^{-1}y_1^{-1}w_1, \\ w_1y_1^{-1}w_1^{-1}z_1, xy_1^{-1} \end{array}\ \right\rangle$$

```
(%i61)  star1:subst(x1,x,star)$
(%i62)  star1:subst(y1,y,star1)$
(%i63)  star1:subst(z1,z,star1)$
(%i64)  star1:subst(v1,v,star1)$
(%i65)  star1:subst(w1,w,star1)$
(%i66)  ts:[append(Trefoil[1],star1[1]),
append(append(Trefoil[2],rest(star1[2],1)),[x.y1^^(-1)])]$
(%i67) cs:calcAlexanderPoly(ts)$
(%i68) factor(cs[2]);
                          2              4     3     2
(%o68)                  (t  - t + 1) (t  - t  + t  - t + 1)
```

このように連結和の Alexander 多項式は二つの結び目の Alexander 多項式の積になります. これは何故でしょうか？ 実は非常に簡単なことで, 連結和の Alexander 行列を見るとおのずから判ります.

$3_1\#5_1$ の Alexander 行列

$$\begin{pmatrix}
1 & -t & t-1 & 0 & 0 & 0 & 0 & 1 \\
t-1 & 1 & -t & 0 & 0 & 0 & 0 & 0 \\
-t & t-1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & \frac{1}{t} & 0 \\
0 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & -1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & 0 \\
0 & 0 & 0 & \frac{t-1}{t} & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & -1 & 0
\end{pmatrix}$$

如何でしょうか？ クローバ結び目と星型結び目の Alexander 行列の成分が対角線上で二つのブロックとして出現していますね. この行列から余因子行列を計算するので当然, 二つのブロックの行列の行列式の積になります. このことから, 与えられた結び目の Alexander 多項式が既約でなければ, この結び目は Alexander 多項式の各既約因子に対応する結び目連結和で構成されている可能性があります. 逆に言えば, 結び目の Alexander 多項式が既約であれば, 結び目も既約である可能性があります. ところが残念なことに, 既約であるとは言い切れません. 何故なら, Alexander 多項式が 1 となる非自明な結び目が存在することが知られているからです. つまり, Alexander 多項式が 1 となる結び目との連結和を構成することで, 同じ Alexander 多項式を持つ結び目が幾らでも構築できるからです.

14.10　結び目の鏡像と Alexander 多項式

結び目 K の鏡像 $\overline{K}$ について考えましょう. 結び目の鏡像とは, ある平面 P に対する対照像 (面対照な像) であり, 正則射影図では交差点での紐の上下を入れ替えたものです. クローバー結び目の例を図 14.13 に示しておきます:

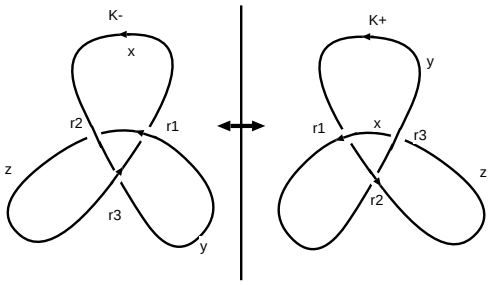


図 14.13: クローバー結び目の鏡像

ここで, K_+ とその鏡像 K_- の違いは何でしょうか？ 先程, 述べた様に交点の符号が逆になることです. K_+ の場合, 交点の符号は全て +1 になりますが, K_- では全て −1 になっています. そのために K_- と K_+ は本質的に違う結び目の筈です. そこで,Alexander 多項式を計算してみましょう.

K_-の結び目群

$$\langle x, y, z \mid xzx^{-1}y^{-1}, yz^{-1}y^{-1}zy, yxy^{-1}z^{-1}\rangle$$

```
(%i20) TrefoilM:[[x,y,z],
[ x.z^^(-1).x^^(-1).y,z.y^^(-1).
z^^(-1).x,y.x^^(-1).y^^(-1).z]]$
(%i21) tm:calcAlexanderPoly(TrefoilM);
                          [ t - 1     1          ]
                          [ -----     -      - 1 ]
                          [   t       t          ]
                          [                      ]
                          [   1             t - 1 ]    2
(%o21)                   [[   -      - 1   ----- ], t  - t + 1]
                          [   t               t  ]
                          [                      ]
                          [         t - 1     1  ]
                          [ - 1     -----     -  ]
                          [           t       t  ]
```

ここでの結果から, K_- の Alexander 多項式は $t^2 - t + 1$ となることが判ります. ところが, K_+ の Alexander 多項式も同じ $t^2 - t + 1$ でした. このように, クローバー結び目では鏡像の区別がつきません. このことから, Alexander 多項式が万能ではないことが判ります.

14.11 Alexander 多項式の意味

この Alexander 行列と Alexander 多項式の意味をもう少し詳しく説明しましょう.
まず, 結び目群 $G(K)$ の Wirtinger 表現による生成元を $x_1, \ldots, x_n$, これらで生成される自由群を F_n, その群環を $\mathbb{Z}F_n$ とします. ここで Alexander 多項式の計算では群 $G(K)$ の生成元 x_i を全て変数 t で置換しています. では, この変数 t は何を意味するのでしょうか？.

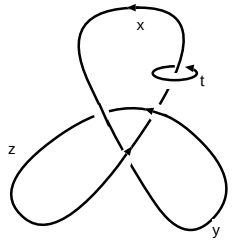


図 14.14: t の意味

ここで基本群の話に立ち返ります. この結び目群 $G(K)$ の生成元 $x_i \in (1, \ldots, n)$ は補空間 $C(K)$ 内の一点 P を起点とする上道を一回りする閉道が対応しました. したがって生成元を全て同じ変数 t で置き換えるということは, これらの閉道を同一視するということを意味します. この変数 t の幾何学的なイメージを図 14.14 に示します. ここで t は結び目を一周する閉道で, 基本群の生成元とは異なり, 基点 P に束縛されていないものです. そして, この閉道の数学的な意味は結び目の補空間 $C(K)$ の 1 次の Homology 群 $\mathrm{H}_1(C(K))$ の生成元となります. ここでいきなり Homology 群というものが出てきましたが, 結び目の補空間に限定してしまえば, その 1 次 Homology 群 $\mathrm{H}_1(C(K))$ は基本群 $\pi_1(C(K), P)$ を可換化したもので与えられることが知られています (「**Hurwiz の定理**」). つまり, $n(n-1)/2$ 個の関係子 $x_i x j x_i^{-1} x_j^{-1} = 1$ $i, j \in \{1, \ldots, n\}$ を追加した群の表示となります. この 1 次の Homology 群と基本群の違いは, Homology 群は可換群で基本群は可換群であるとは限らないことと一次 Homology 群の元で 0 になる元は空間内部に埋め込まれた曲面の境界ですが, 基本群の場合は Dehn の補題によって埋め込まれた円盤の境界とな

ります. このように生成元を変数 t で置き換えるという処理は群の可換化に関連した処理で, 結び目の補空間の 1 次の Homology 群は t だけで生成されるために $\mathbb{Z}$ と同型の群となります.

ここで結び目群の $G(K)$ 可換化写像を *tau* とします. この写像は $\tau : G(K) \to \mathbb{Z}$ となります. そして, Fox の微分演算子 $\partial/\partial x_i$ は群環 $\mathbb{Z}F_n$ から群環 $\mathbb{Z}F_n$ への環準同形写像, それと包含写像 $F_n \to G(K)$ から自然に導かれる環準同型 $\iota : \mathbb{Z}F_n \to \mathbb{Z}G(K)$ を使って次の写像の列が考えられます:

$$\mathbb{Z}F_n \xrightarrow{\partial_i} \mathbb{Z}F_n \xrightarrow{\iota} \mathbb{Z}G(K) \xrightarrow{\tau} \mathbb{Z} \xrightarrow{0} 0$$

ここで Alexander 行列の元となる Fox 微分演算子による Jacoian を ∂_i の個所に置くと, この Jacobian の像が可換化写像 τ の核になります.

さて, t は一次の Homology 群 $\mathrm{H}_1(C(K))$ の生成元に対応すると述べましたが, これだけではまだ釈然としないでしょう. この Alexander 行列と多項式には別の幾何学的な表現があります. この幾何学的な表現は, 結び目群の別の表現である Seifert 表現に関連する手法と「**Dehn 手術**」と呼ばれる手法を用いて可視化する方法です. どちらの方法も最終的には結び目の補空間から**普遍被覆空間**と呼ばれる空間を構成し, その被覆空間の構造に関連する不変量として Alexander 多項式が現われるというものです. これらの視覚化は「Publish or Perish」[2]という凄い名前の出版社から出ていた Rolfsen の名著「Knots and Links」[89] に詳しく説明されています. 意味は判らなくても不思議な絵を図書館で眺めるだけの価値が十分あります.

そこで Seifert 表現に関連する手法から説明しましょう. この Seifert 表現は Wirtinger 表現と並ぶ結び目群の表現の一つで, この表現を構成するために Seifert 曲面と呼ばれる曲面を作成する必要があります. この Seifert 曲面は, その境界が結び目となるとても「**性質の良い曲面**」です. 先程の Alexander 多項式の計算で「**結び目の連結和**」という言葉が出ましたが, この連結和は Seifert 曲面の連結和にそのまま対応します. その他の結び目の処理でも色々と用いられ, ここで解説する「**被覆空間**」の構成でも重要な役割を果します. この曲面の構成は非常に簡単です:

[2]「鹹が嫌なら論文を書け」という意味. 微分幾何学が専門で「Comprehensive Introduction to Differential Geometry」や [The Joy of TEX:A Gourmet Guide to Typesetting With the AMS-TEX Macro Package and The Hitchhiker's Guide to Calculus」といったユニークな著作で有名な M.Spivak が設立者だそうです.

Seifert 曲面の構成手順

1. 結び目の射影図に向きを入れる
2. 交差点で道に分解する
3. 道を選択して向きに沿って円を構築する. このときに次の規則で円を構築する:
   - 交差点でつきあたった下道に移動する
   - 交差点でつきあたった上道に移動する
   - 道が閉じて円ができると他の上道に対しても同じ処理を行う
4. 道が全て円で置き換えられると結び目から誘導された円弧の向きに沿って円盤の塗り潰しを行う. 塗り潰す側は進行方向の左側 (X 軸正方向に向いていれば上半平面側 ($Y > 0$)) とする. この塗り潰した箇所が貼り合わせる円盤に対応する. もしも無限遠点を含む円盤が現われた場合, 紙面裏側に円盤を貼る.
5. 交差点で境界が交差と一致するバンドを取り付けて円盤を繋ぎ合せる

この手順をクローバ結び目に対して行った様子を以下に示します:

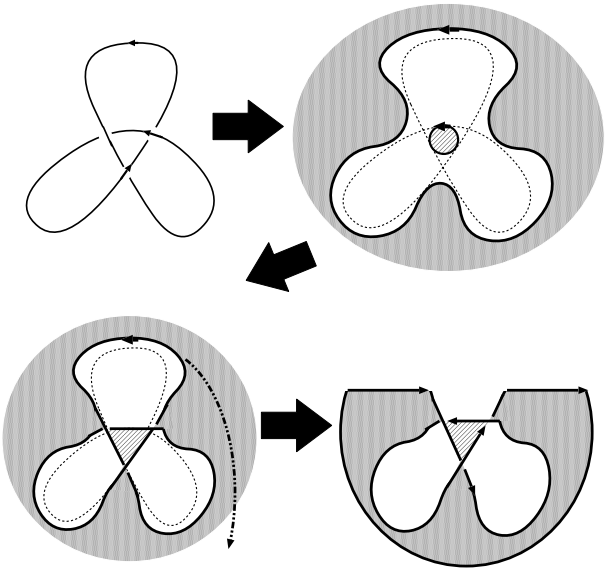


図 14.15: Seifert 曲面の作り方

この図では左上が結び目の射影図に向きを入れて上道に分割したものを示しています. 右上が円を構成した時点で, 各円を境界とする円盤を塗り潰しています. ここで外側の円を境界とする円盤は無限遠点を含む円盤となるので, 紙面から見て袋のように下側に張っています. 次に左下の図では各交差点に本来の結び目の交差に対応するようにバンドを貼ります. こうして出来た曲面が Seifert 曲面ですが, まだ平面的ではないので左下の図で一番上側にある結び目の上道だった箇所を垂れた

袋に沿ってぐるりと X 軸に 180 度回し, 右下の図のように下側にもってゆきます. それによって下側に垂れていた円盤は右下の図のように平面的なものへと変形され, より分かり易い Seifert 曲面が得られます.

Seifert 曲面はその境界が結び目となる**向き付け可能な曲面**です. ここで向き付け可能な曲面とは, 円盤やドーナッツの表面のように表面と裏面の区別がある曲面のことです. 3 次元空間内の円盤 $\{(x,y,z) \in \mathbb{R}^3 | x^2 + y^2 \leq 1\}$ を例に考えてみましょう. この円盤の裏と表はどのように決めると良いでしょうか？ 通常は Z 軸の上側が表, Z 軸の下側を裏としますね. つまり, 円盤の法線ベクトル $v_o = (0,0,1)$ 一つを定めて, 見ている面の法線ベクトル v と v_o の内積を計算して, その値が正なら表, そうでなければ裏と判断できる訳です. このように曲面の向き付けには法線ベクトルが大きく関連します. さて, このことを一般化すると, 「向き付け可能な曲面とは, 3 次元空間に埋め込まれた曲面 M 上の各点で長さが 1 となる法線ベクトルを与える連続函数 v_p が定義できること」と定義できます. 先程の円盤の例では, 法線ベクトル v_o は円盤のどこの点でも移動させることができますね. 要するに, 曲面 M 上の点 p で定めた法線ベクトルという旗竿を曲面上の至る所に自由に移動することができて, その長さは何時でも一定にすることのできる曲面なのです. そして, 3 次元空間 $\mathbb{R}^3$ にこのような空間が埋め込まれた場合, 曲面全体を法線ベクトルに沿って, 元の曲面にぶつからないように上下に僅かに動かすことが可能です. このことから向き付け可能な曲面の $\mathbb{R}^3$ での近傍は $M \times I$ と同相になることが判ります.

その一方で向き付け不可能な曲面という曲面があります. この曲面で有名なものとして Möbius の輪や Klein の壷で, これらの曲面には全体で裏面と表面がないという特徴があります. つまり, $\mathbb{R}^3$ に埋め込まれたときに, 曲面全体で一定の長さの法線ベクトルを与える連続函数が与えられない曲面です. たとえば, 図 14.16 に示すように Möbius の輪の上を動いている蟻が裏に回って見えなくなったり, やがて表に回って見えるといったことが生じます. 特に中央下側の蟻が一周すると裏面に回っていることに注意して下さい:

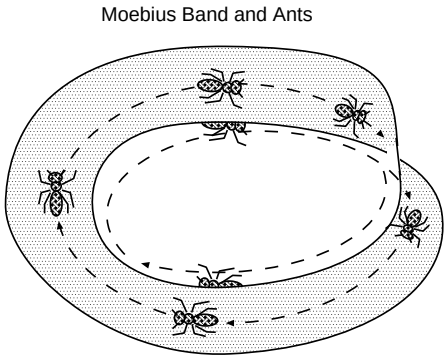


図 14.16: 向き付けできない曲面の例

つまり, 点 p に旗竿を立てて, Möbius の輪の中心軸に沿って一周して点 p に戻ると旗竿が逆向きに立ってしまうことになります. このことから向きを定める法線ベクトルを与える函数が曲面全体で

は定義できないことが判るでしょう. また, 曲面を通常の 3 次元区間 $\mathbb{R}^3$ に滑かに埋め込んでいるときに, 向き付け可能な曲面であれば局所的に定めた法線ベクトルを曲面全体に延長可能で, 法線ベクトルに沿って曲面上の曲線を曲面と交差しないようにそっと押し出すことができます. 一方で, 向き付け不可能な曲面では勝手が違います. 図 14.17 に Möbis の輪の例を示しておきましょう:

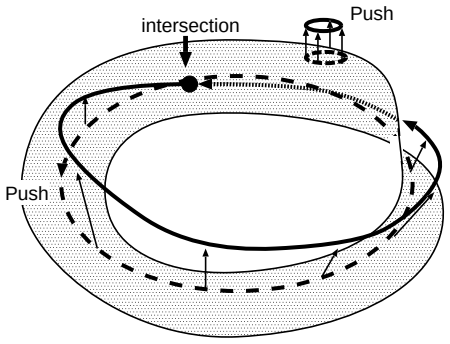


図 14.17: 向き付け不可能な曲面上の閉曲線の押し出し

向き付け不可能な曲面でも局所的には曲面の裏と表が一意に定まるので, 局所的に曲面と交差しないように曲線を押し出すことができます. この様子は図の右上にある小さな円が Möbius の輪と交差することなく押し上げられている様子から判るでしょう. しかし, 向き付け不可能な曲面上の全ての閉曲線が押し上げられるとは限りません. 実際, Möbius の輪の中心軸となる円を押し上げると必ず曲面との交差が生じます. なぜなら, 局所的に定めた法線ベクトルを中心軸に沿って延長しようとしても一周した時点で法線の向きが逆向きになってしまうために法線ベクトルを出発点で 0 にして辻褄合せをしなければ中心軸に沿って法線ベクトルを与える滑らかな函数は構成できないからです. ここで図 14.17 で交差なしに押し上げられている閉曲線が Möbius の輪の基本群の単位元で, Möbius の輪の中心軸が基本群の生成元となります. このように曲面上で円盤の境界となる閉曲線を**非本質的な曲線**, 曲面上で円盤の境界とならない曲線のことを**本質的な曲線**と呼びます. 先程の例では, Möbius の輪から曲面と交差せずに押し上げられる円は非本質的な曲線, 後者の中心軸は本質的な曲線になります. なお, M を任意の曲面とするときに区間 $I = [-1, 1]$ との積空間 $M \times I$ を考えましょう. この空間は曲面 M が金太郎飴上に重なった空間で, $M \times 0$ 上の任意の曲線は I に対して好きな方向に押し出すことができます. このように 3 次元空間に埋め込まれた曲面 M の近傍が $M \times I$ と同相になる場合に, この曲面 M を「**両側曲面**」と呼びます. また, R^3 空間内に埋め込まれた Möbius の輪のように, その近傍が $M \times I$ と同相にならない埋め込まれ方をした曲面 M のことを「**片側曲面**」と呼びます. このように曲面の空間への埋め込まれ方で状況が異なります.

さて, 曲面には向き付け可能な曲面と不可能な曲面の他に, 境界を持つかどうかといった違いもあります. ここで境界を持つ曲面を**開曲面**, 境界を持たない曲面のことを**閉曲面**と呼びます. たとえば, 円盤と Möbius の輪は境界が円となるので開曲面で, 一つ穴のドーナッツの表面と Klein の壷は境界を持たないので閉曲面となります. ここで一つ穴のドーナッツの表面を**トーラス**, あるいは**輪環面**と呼びます. そして, トーラスは二つの円周の直積 $S^1 \times S^1$ に同相となります. また, 中身の詰った

ドーナッツのことを**ソリッド・トーラス**, あるいは**トーラス体**と呼びますが, この本では親しみ易さから安易にドーナッツと呼びことにします. また, $g \geq 1$ 個の穴の開いたドーナッツの表面の場合, その穴の数 g のことを曲面の**種数**と呼び, g 個の穴開きドーナッツの表面を種数 g のトーラスと呼びます. さて, 種数 $g \geq 1$ のトーラスは種数 1 のトーラスを g 個用意して球面に貼り付けたものの表面と考えることができます. このとき, この種数 1 のドーナッツは丁度, 把手のような形になります. ここで曲面を種数 1 のトーラスと球面に分解する操作のことを「**ハンドル分解**」と呼びます:

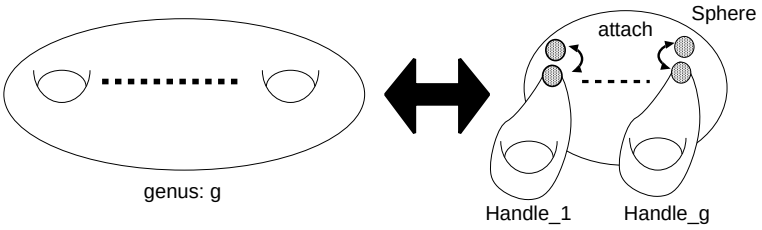


図 14.18: ハンドル分解

図 14.18 に種数 g の曲面のハンドル分解を示しておきますが, この図に示すように各ハンドルはトーラスから円盤を除いたものと同相で, 球面への取り付けは, 球面から円盤を取り除き, その取り除いた円盤の境界 (=円) で接着します. なお, 向き付け不可能な曲面の場合は向き付け可能な曲面から m 個 ($m > 0$) の円盤を抜き, その境界の円に Möbius の輪を貼り付けたものと同相になることが知られています. このことは, 閉曲面ではハンドルの総数と Möbius の輪の総数で分類ができることを意味しています.

このハンドルの接着のように, 二つの曲面から円盤, すなわち 2 次元球 B^2 を取り外し, その境界となる円, すなわち 1 次元球面 S^1 で互いに貼り合わせることで新しい曲面を構成する方法を曲面の**連結和**と呼びます. そして, 曲面 A, B との連結和を $A\#B$ と表記します. この連結和は $n \geq 3$ 次元の多様体にも拡張可能です. 実際, A, B を n 次元の多様体とするときに, A, B の双方から n 次元球 B^n を取り出し, その境界の S^{n-1} で貼り合わせたものを多様体 A, B の連結和と呼び, このときに構成される多様体も $A\#B$ と表記します. 連結和によって n 次元の連結な多様体の集合には半群の構造が入り, 単位元となるのは n 次元球面 S^n です. ちなみに, 結び目の連結和を以前紹介しましたが, この結び目の連結和は多様体の連結和の結果として得られるものです. つまり, 結び目の上道の一部を含む 3 次元球を取り出し, 貼り合せのときに結び目の向きに適合するように球面の穴を合せることで得られます.

ここで Seifert 曲面の話に戻ります. Seifert 曲面は向き付け可能な曲面で, その境界が円周となることから, ある種数 g のトーラスから円盤を一つ除去したものと同相になります. なお, ここでの話は曲面が埋め込まれた空間のことは考えていません. だから, Seifert 曲面の境界に貼り付けられる円盤は 3 次元空間内部で埋め込まれた円盤になるとは限らないことに注意して下さい. さて, ここで種数 1 のトーラスに穴を一つ開けたものを考えてみましょう. この穴には 3 次元空間 $\mathbb{R}^2$ で円盤が貼れるので境界は自明な結び目です. そして, このトーラスが粘土でできていると考えて下さい.

トーラスを千切らずに穴を徐々に広げてゆけば, やがて図 14.19 に示すようにバンドが二本生えた円盤に変形できることが判ります:

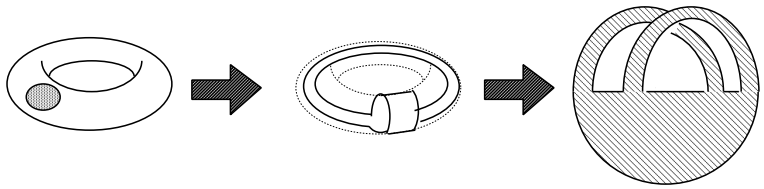


図 14.19: 穴開きトーラス

では種数 g のトーラスに穴を開けたときに, この曲面はどのような曲面になるでしょうか? 結論を言ってしまえば図 14.20 に示すように $2g$ 本のバンドが生えた円盤と同相になります:

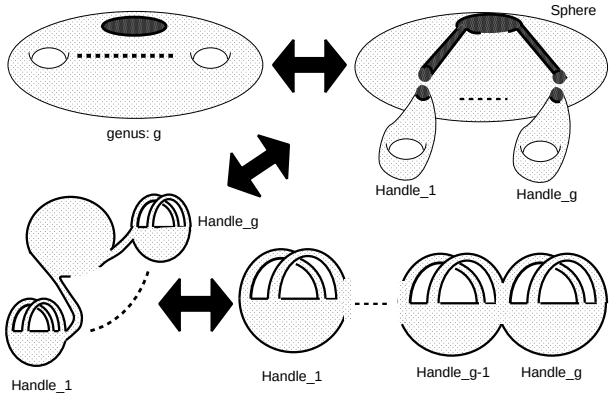


図 14.20: 穴開きの種数 g のトーラス

この図の左上に穴を開けた種数 g の曲面があります. この曲面上の穴の境界を図右上のようにバンドで各ハンドルの根本に延長します. それからハンドルと球面に分けて考えます. まず, ハンドルは図 14.19 の右の曲面に変形できますが, 穴の境界 (各ハンドル根本の太線) でない箇所は球面を開いてできた円盤に繋るために左下の絵のように円盤に帯で繋った曲面に変形できます. これをさらに変形して整理すると右下に示すように g 個の図 14.19 の右の曲面を繋げた曲面, すなわち, $2g$ 個のバンドが円盤から生えた曲面が得られます.

さて, 今度は我々の空間 $\mathbb{R}^3$ に埋め込まれた穴が一つ空いた種数 g の曲面が埋め込まれていたとします. この曲面は先程の図 14.20 で示したように連続的に変形すると $2g$ 本のバンドを持つ円盤が得られます. ただし, このバンドは境界となる結び目によって互いに絡みあったものになります. また, 単純に変形しただけでは円盤からバンドが二本出ていると明瞭でない場合もあります. そこで, 曲面のバンドの根本を動かす**バンド移動**という処理があります.

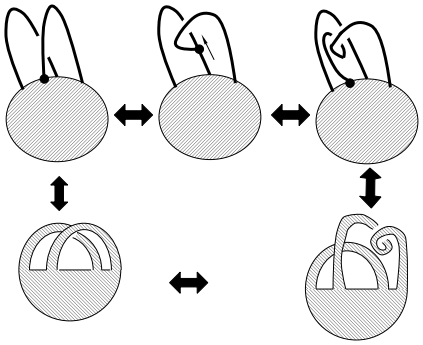


図 14.21: バンド移動

この処理の様子を図 14.21 に示しておきますが, 図左下の自明な結び目の Seifert 曲線の右側のバンドをもう一方の左側のバンドに沿って移動させることを上段で模式的に示しています. この操作からも判るように, バンドの移動は Seifert 曲面の境界を変える変形ではありませんが, この移動によって動かしたバンドのみに $\pm 2\pi$[rad] の捻りが入ります. このバンド移動は構成した Seifert 曲面が, 円盤に $2g$ 本のバンドが生えていることを明瞭にするために行われたり, バンドに $\pm 2\pi \times n$ $n \in \mathbb{Z}$ の捻れがあった場合に, その捻れの解消といったことに用いられます. このバンド移動の結果, 得られた曲面に左下の曲面が幾つか繋った曲面が新しい曲面として得られることがあります. この場合は左下の曲面の部分の境界は自明な結び目なので, この部分を削ってしまっても結び目本体には影響が現われません. このように自明な結び目の境界となるハンドルに対応する曲面を削除したり, 逆に取り付ける操作から得られる曲面を**ハンドル同値な曲面**と呼びます:

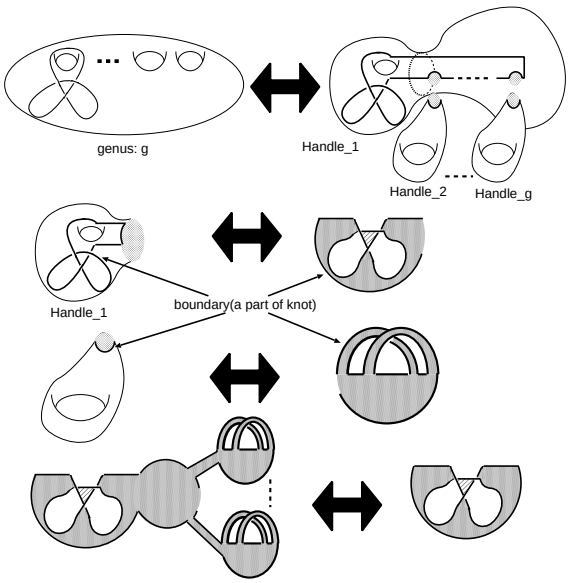


図 14.22: バンドル同値

図 14.22 にはクローバー結び目の例を示しています. 最上段は概念的な絵で, 左上にクローバー結び目を境界として持つ種数 $g > 1$ の Seifert 曲面を描いています. そして右上が曲面をハンドル分解した様子を示しています. ここでクローバー結び目は種数 1 の結び目のために何処かのハンドルの上にあり, その結び目の境界を短冊状に Seifert 曲面上に延し, 各ハンドルの付け根に触れるようにしています. そして図の中段には上段の Seifert 曲面のハンドル分解で得られたクローバー結び目を境界として持つハンドルと自明なハンドルのそれぞれに対応する曲面を示しています. そして下段にはこれらの分解で判ったハンドルを繋いだ Seifert 曲面を左下に示し, 右下に自明なハンドルを除去した Seifert 曲面を示しています. 右下の曲面は 2 本のハンドルを持っているために種数が 1 の曲面であることが判ります. なお, Seifert 曲面の種数が 0 であるためには円盤の境界とならなければならないことから自明な結び目でなければなりませんが, このクローバー結び目は Alexander 多項式の結果からも判るように自明な結び目ではないので, クローバー結び目の Seifert 曲面の最小種数が 1 であることが判ります.

このようにハンドルはドーナッツの表面に一つ穴を開けた曲面と同相であるために種数が 1 となります. そして, ハンドルを曲面から一つ削除すれば曲面全体の種数が 1 つ減ります. この操作を反復することでより種数の小さな Seifert 曲面が得られます. そうして得られた曲面の最小種数のことを**結び目の種数**と呼びます. この結び目の種数は重要な位相不変量で, 自明な結び目の Seifert 曲面はハンドルが削除できるので円盤となるので種数は 0, クローバー結び目は 2 本のバンドを持つ Seifert 曲面となるので種数が 1 となります. また, Alexander 多項式 Δ_K の次数 $\deg(\Delta_K)$ と結び目の種数 g との間には $\deg(\Delta_K) \leq 2g$ の関係があることが知られています. このことは種数 g の Seifert 曲面のを持つ結び目では, Alexander 行列の大きさが $2 \times g$ となることから何となく想像できるでしょう.

さて, これからいよいよ結び目補空間の被覆空間を構築します. この**被覆空間**というものは元の空間を一つのブロックとして持つ金太郎飴のような空間と思って下さい. より正確には, 空間 A が空間 B の被覆空間であるとは, 次の性質を充す写像 $p : A \to B$ が存在するときです:

被覆空間の定義

- 写像 $p : A \to B$ は全射
- 任意の $p \in B$ に対し, 点 p の近傍 $U_p \subset B$ で $p^{-1}(U_p) = \cup_i V_i,\ V_i \cap V_j = \emptyset, i \neq j$ を充す A の開集合 $V_i,\ i \in \mathbb{Z}$ が存在する
- $p : V_i \to U, i \in \mathbb{Z}$ は同相写像である

空間 B がその被覆空間として A を持つのとき, 空間 B のことを**底空間**と呼びます. そして, 空間 A の自己同相写像 t で $p \circ t = p$ を充すものの集合は群としての構造を持ちます. 実際, 恒等写像 id が単位元, 同型写像 $a : A \to A$ の逆写像 a^{-1} も同相でその逆写像 a^{-1} については $a^{-1} \circ a = a \circ a^{-1} = \mathrm{id}$ を充すので逆元も存在します. あとの結合律 $(a \circ b) \circ c = a(\circ b \circ c)$ が成り立つことも問題ないでしょう. 被覆空間 A の自己同相写像で特に $p \circ t$ を充す自己同相写像 t の集合 T を考えます. すると, この写像には少なくとも恒等写像 id が存在するので空ではありません. そして, T には写像の合成で積が自然に入り, $t_1, t_2 \in T$ に対して $p \circ (t_1 \circ t_2) = p$ を充すので T は部分群となります. この群は $\{p^{-1}(B)\}$ の元を別の元に移動させる性質を持っているので, **被覆変換群**と呼ばれます. これを金太郎飴で説明しましょう。

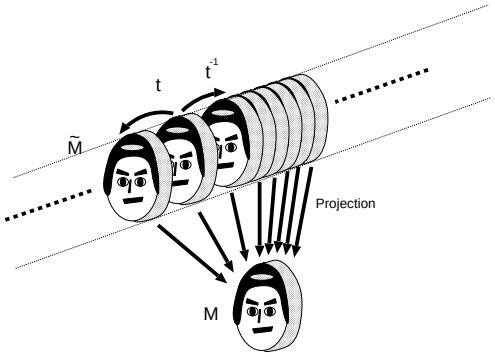


図 14.23: 金太郎飴

金太郎飴で被覆空間を例えると, 金太郎飴が底空間 M, 飴を切り外す前の棒状の大きな飴が被覆空間 $\tilde{M}$ に対応します. そして射影 $p : \tilde{M} \to M$ は棒から金太郎飴を取り出す操作, 変換群は棒の中の飴という塊を水平方向にどう動かすかを定める写像, もっと直感的には飴を切り離す包丁の位置を定める操作となるでしょうか. さて, この金太郎飴の場合, 被覆空間は棒状の大きな飴となりますが, ある基準点を定めて切り出す包丁を動かす操作で, 右隣りの飴となる部位に移動する操作を t, 左隣の飴となる部位に移動する操作を t^{-1} としましょう. ここで任意の自然数 n に対して右方向に n 個移動させる操作を t^n, 左方向に移動させる操作を t^{-n}, 飴を動かさない操作を $t^0 = 1$ とします. さて, 包丁を t^n 程動かし, それから今度は t^m 程動かすと包丁の位置は t^{n+m} となります. また, 最初に t^m 動かし, それから t^n 動かせば t^{m+n} の位置に包丁が動いていますが, これは最初の t^{n+m} と同じ位置ですね. そして指数だけを見ると $n + m = m + n$ と通常の整数の和になっています. この考察から, この変換は可換であり, 二つの変換の合成はその指数の和が対応することが判ります. もし, この金太郎飴の棒が無限に続くのであれば, 左右どちらでも無限に移動させられるので, この変換の集合 $T = \{t^n | n \in \mathbb{Z}\}$ は $\mathbb{Z}$ に一対一対応し, 変換の合成と整数の通常の和に対応させることで可換群となります. では, 金太郎飴の棒が 7 個で構成されている場合はどうでしょうか？この場合, 金太郎飴 $\{A_1, A_2, \ldots, A_7\}$ で tA_7 や $t^{-1}A_1$ をどう処理するかになりますが, ここでの解決方法は単純に変換 t の冪 n を 7 で割ったときの剰余を変換の冪として用います. たとえば, $A_7 = t^6 A_1$ なので, $tA_7 = t^7 A_1$ ですね. ここで 7 を 7 で割った余りは 0 となるので $t^0 = 1$, したがって $tA_7 = A_1$ となります. このことから $t^{-1}A_1 = t^{-1}tA_7 = A_7$ も得られ, 至って自然に変換が定まります. このように n 個の部位で構成された被覆の変換群は $\mathbb{Z}_n$ となります.

では底空間が金太郎飴の絵となるとどうでしょうか？ この場合, 被覆変換群に相当する群は $\mathbb{R}$ で, 底空間の点 P の逆像 $p^{-1}(P)$ は $\mathbb{R}$ となります. そして点 P の近傍 U_P の射影 p による逆像は $U_P \times \mathbb{R}$ と同相で, この様子は $\mathbb{R}$ を束ねた状態です. このように底空間の任意の点で適切な近傍を取ると, その逆像が直積空間として表現される空間を fiber bundle と呼びます. fiber bundle の代表的な例には物理学でもお馴染の接空間 T_pM を fiber として持つ接バンドル空間 $\mathrm{T}M$ を挙げておきましょう.

さて, 結び目補空間の被覆空間の構成に戻ります. はじめに補空間 $C(K)$ を Seifert 曲面 S_K で切り開きます. ここでの切り開きとは, Seifert 曲面を補空間内部で太らせた代物, つまり Seifert 曲面の補空間から取り除く処理が対応します. ここで, Seifert 曲面は向き付け可能な曲面なので, 補空間内部で両側曲面となるので, その正則近傍は Seifert 曲面と開区間 $I = (0, 1)$ との直積 $S_K \times I$ に同相になります. この太らせた Seifert 曲面を補空間から抜き出すと Seifert 曲面の表面 S_+ と裏面 S_- が, その境界として現われることが容易に理解できるでしょう. また, Homology 群の生成元の t は, この金太郎飴を抜いた空間の境界として現われる Seifert 曲面の裏面 S_- と表面 S_+ を互いに結ぶ曲線と

して現われます. それから C の複製 $C_i\, i \in \mathbb{Z}$ を準備して, 各 $C_{i-1}, C_i\, i \in \mathbb{Z}$ を各部位に現れる Seifert 曲面の表 $S^{i-1}_{K^+}$ と裏 $S^i_{K^-}$ で貼り合わせます. こうしてできた空間を $\tilde{C}$ と表記しますが, この空間は結び目補空間 $C(K)$ の**被覆空間**となっています.

この構成を図 14.24 に示す例で確認しておきしょう. 上段左がクローバ結び目, 右がその Seifert 曲面です. それから Seifert 曲面の切り開きを行いますが, 中段左にてドーナッツ状に補空間を示しています. このようにドーナッツとして見える理由は, 結び目の補空間の境界がトーラスに同相であり, 今迄は外側から結び目の管状近傍を眺めていましたが, それを結び目側から管状近傍の表面を眺めるのです. その結果, 図左中の絵のように外見がドーナッツ状の空間内部に Seifert 曲面が入っており, Seifert 曲面の境界がトーラスの本質的な曲面となっているのです. ここでドーナッツ表面には二つの本質的な曲面があります. まず, ドーナッツ内側で円盤が張れる閉曲線は meridian と呼ばれ, ここでは μ と表記します. もう一方の閉曲線はドーナッツの穴を一周し, meridian と一度だけ交差する閉曲面で, longitude と呼ばれ, ここでは λ と表記しますが, この λ こそが結び目補空間の一次 Homology 群の生成元 t に対応します. ここでのドーナッツ状の空間は通常のドーナッツ, つまりトーラス体とは異なった空間です. 実際, meridian μ は Seifert 曲面の境界となる結び目 K と一致するので μ がドーナッツ内部の滑かな円盤の境界となるとは限りません. このドーナッツ内部の構造は **Dehn の手術**を施すことで可視化することができます. この Dehn 手術は 3 次元や 4 次元多様体の構成で非常に重要な技術であり, 面白い図式を使うので後に解説します.

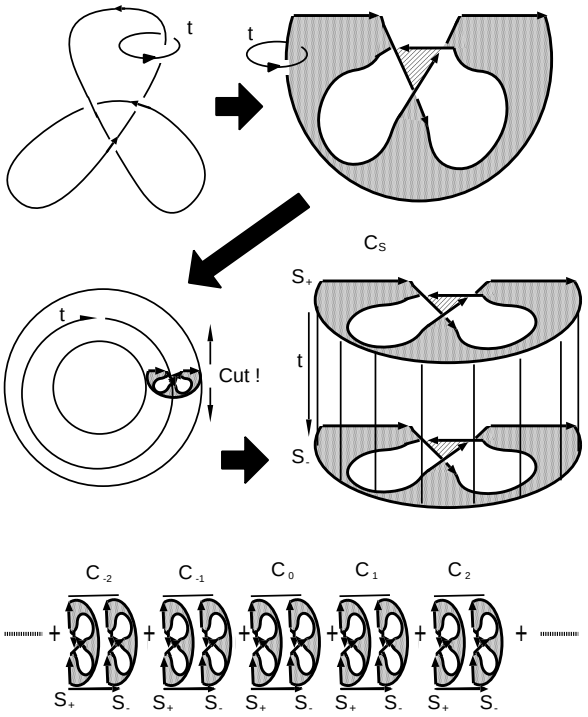


図 14.24: 被覆空間の構成

それから Seifert 曲面でこの空間を切り開きます. ここでは Seifert 曲面に少し厚みを付けたものを除去してしまえば良いのです. 図の下段には, この空間の複製を $n \in \mathbb{Z}$ 程, 用意して, 各空間 C_i の表面 S_+ と隣合う空間の C_{i+1} の表面 S_- を貼り合わせている様子を示しています. 被覆空間 $\tilde{C}$ はこのような構成で得られます. ここで曲線 t ですが, 写像 t を $t \cdot C_i = C_{i+1}, i \in \mathbb{Z}$ で C_i を C_{i+1} に移す操作, t^n で t を n 回作用させる操作とすると, $t^{-1} \cdot C_{i-1} = C_i$ で逆操作が自然に表現できるので集合 $\{t^i | i \in \mathbb{Z}\}$ は群としての構造を持ちます. さらに $p \circ t^n = p$, $n \in \mathbb{Z}$ を充すので写像 t は $\tilde{C}$ の被覆変換群になります.

ここでクローバー結び目で被覆空間の 1 次のホモロジー群を計算してみましょう. ここでの一次の Homology 群は, 基本群を可換化したものと同じものになるため, 円盤や曲面の境界にならない閉じた道が生成元になります. 被覆空間の一次の Homology 群の生成元は Seifert 曲面の周りに存在します.

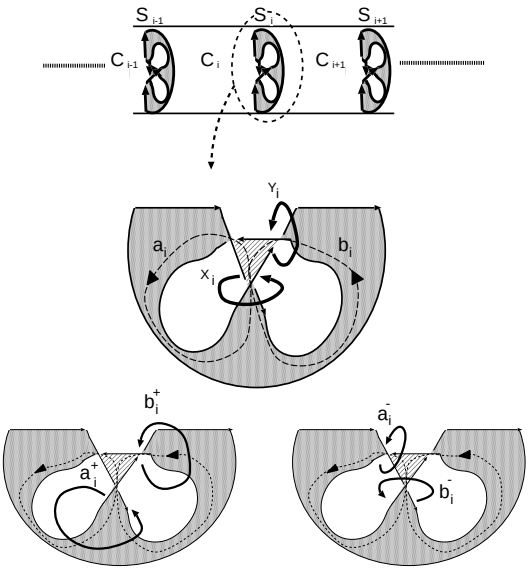


図 14.25: 一次 Homology 群の生成元

図 14.25 に被覆空間の様子を示しておきます. まず, 上段の絵が被覆空間の様子で, $C_i, i \in \mathbb{Z}$ が Seifert 曲面で切り開いた結び目の補空間の複製を示します. 底空間と同相になる部位の区切として Seifert 曲面 $S_i, i \in \mathbb{Z}$ が現われています. この Serifet 曲面 S_i には本質的な曲線として図中段の a_i, b_i が存在していますが, これらの曲線は Homology 群の生成元の一部で, この他に被覆空間内部でバンドを一周する x_i と y_i も Homology 群の生成元となります. ここで Seifert 曲面 S_i 上の本質的な曲線 a_i と b_i を Seifert 曲面の表側に押し出したものを a_i^+, b_i^+, 曲面の裏側に押し出したものを a_i^-, b_i^- としましょう. まず, Seifert 曲面の表側に押し出された曲線は裏側から押し出された曲線と同じものとなります. つまり, $a_i^+ = a_{i+1}^-, b_i^+ = b_{i+1}^-$ を充します. それから図の下段から判るように x_i, y_i との関係として, $a_i^+ = -b_i^- = x_i$, $b_i^+ = y_i$, $a_i^- = x_i + y_i$ を充します.

以上からクローバ結び目一次元の Homology 群の表示は

$$\left\langle\ x_i, y_i, a_i^{\pm}, b_i^{\pm}\ \middle|\ \begin{array}{l} a_i^+ = b_i^- = x_i, b_i^+ = y_i,\ a_i^- = x_i + y_i,\ a_i^+ = a_{i+1}^-, b_i^+ = b_{i+1}^-, \\ x_i y_i x_i^{-1} y_i^{-1},\ a_i^{\pm} b_i a_i^{\mp} b_i^{\mp},\ a_i^{\pm} x_i a_i^{\mp} x_i^{\mp},\ b_i^{\pm} x_i b_i^{\mp} x_i^{-1},\ a_i^{\pm} y_i a_i^{\mp} y_i^{-1}, b_i^{\pm} y_i b_i^{\mp} y_i^{-1} \end{array}\ \right\rangle$$

となります. ここで二行ある関係子のうち, 下の関係子は可換性に関係するもので, 上の関係子を整理しましょう. すると, $a_i^{\pm}, b_i^{\pm}$ は x_i, y_i で生成可能なので生成元や関係子から除くことができます. また, $x_i + y_i = a_i^-$ は $a_i^- = a_{i-1}^+$ と $a_i^+ = x_i$ より $x_i + y_i = x_{i-1}$, そして, $y_i = b_i^+, b_i^+ = b_{i+1}^-$ と $b_i^- = -x_i$ から $y_i = -x_{i+1}$, この結果から被覆空間の一次の Homology 群として

$$\langle x_i, | x_{i+1} - x_i + x_{i-1} \rangle$$

が得られます. さて, この被覆空間の変換群は t で生成されており, t の作用は平行移動を行う群です. その変換群を用いると $x_{i+1} = t \cdot x_i$ なので, 関係子は $x_{i+1} - x_i + x_{i-1} = t^2 \cdot x_{i-1} - t \cdot x_{i-1} + x_{i-1} = (t^2 - t + 1) x_{i-1}$ となり, 被覆空間の一次の Homology 群は最終的に $\mathbb{Z}[t^{-1}, t]/(t^2 - t + 1)$ となります. ここで注目すべき点は関係子に現われた多項式 $t^2 - t + 1$ です. この多項式はクローバー結び目の Alexander 多項式です. このように結び目の Alexander 多項式は結び目の補空間の被覆空間を考えたときに, そ

の一次の Homology 群の関係子として現われる多項式なのです.

ここで結び目の補空間 $C(K)$ をドーナッツとして見なす話をもう少し解説しておきましょう. 結び目の補空間は結び目を除いた空間です. 要するに結び目の糸をチューブ状に太らせて, そのチューブを取り除いた空間です. そのため補空間の境界はトーラス, つまり, ドーナッツの表面と同じ曲面です. すると, 結び目を太らせたものは捻りが入ったドーナッツで, 補空間はその境界がトーラスになった空間になります. さて, 結び目が自明な結び目の場合はどうでしょうか？

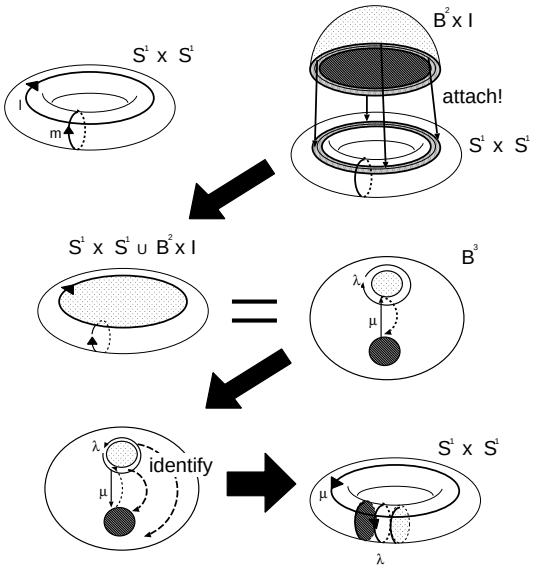


図 14.26: ドーナッツ二つで 3 次元球面が完成

まず, 結び目を太らせたもの, つまり管状近傍はドーナッツそのもの (トーラス体) なので, その表面はトーラスになります. ではその補空間はどうのような空間なのでしょうか？ このときの様子を図 14.26 に示します. まず, 補空間の表面はトーラスになります. そこで, このトーラスの longitude と meridian をそれぞれ左上の図のように μ, λ とします. このトーラスの中に結び目が入っており, 自明な結び目であれば結び目と λ は平行なので, λ を境界とする円盤を補空間内部に貼ることができます. このときにこの円盤を表方向と裏方向に補空間内で太らせて円盤の管状近傍 ($B^2 \times I$) を構成してトーラスに貼ってしまいます. 図 14.26 の右上の図がトーラスに円盤の管状近傍を貼ろうとしている様子で, 帽子上のものが円盤の管状近傍になり, トーラス上の円環 $S^1 \times I$ が $B^2 \times I$ との接着面を表現しています. そして, 管状近傍を貼ったあとの様子を中段左絵に示しています. これを中段右のように膨らませると 3 次元球であることが簡単に理解できるでしょう. そして, 2 次元球面 S^2 が二つの 2 次元球, つまり円盤を貼り合わせて出来ているのと同様に, 3 次元球面 S^3 は二つの 3 次元球を境界で貼り合せることで構成されています. このことから, このことから補空間から先程の円盤の管状近傍を除いた空間も 3 次元球面であることが判ります.

さて, 図中段右の 3 次元球では, 先程の円盤の管状近傍の裏面と表面, もともとのトーラスの longitude の λ と meridian の μ も一緒に描いています. この図の中段左に示す球面の内部には結び目が包含され, もう一方の 3 次元球は我々の視点を含んでいます. そこで, 視点を結び目側に変更して中段左図の球面を眺めることにします. そのために行うことを簡単に言ってしまえば球面を裏返してして見ることににになります. この裏返しを行ったものが下段左の絵で, 裏返したことで μ や λ に対応する曲線の向きが逆になっていることに注意して下さい. ここで, 結び目の補空間側に先程貼り付けた円盤の管状近傍ですが, この近傍は我々の視点側に球面上に残っている円盤の裏と表を繋ぐ円柱となっています. それを見えるように描くと右下のドーナッツが得られます. このことから 3 次元

球面 S^3 は二つのドーナッツの貼り合せで構成されていることが判ります.

このように二つのドーナッツを表面で貼り合せることで構成される空間は**レンズ空間**と呼ばれる空間です. このレンズ空間では, 共通のトーラスの longitude と meridian を λ, μ とするとき, 互いに素な $\alpha, \beta \in \mathbb{Z}$ に対して $\alpha \cdot \lambda + \beta \cdot \mu$ で定まるトーラス上の閉曲線に円盤を貼り付けて最後に 3 次元球を充填することで構築される空間で, この空間のことを $L(\alpha, \beta)$ と表記します.

たとえば, 図 14.26 で示した球面では l に円盤を貼り付けるだけで 3 次元球面が得られているので $L(1,0) = S^3$ であり, $L(0,1)$ は二つのドーナッツ表面の meridian を longitude に沿って貼り合せることになります. この場合, 双方の meridian はトーラス体に埋め込まれた円盤の境界となるので, この貼り合せで 2 次元球面が出来上がります. そして, これらの球面は 1 次元球 S^1 の λ に沿って現われることから $L(0,1) = S^1 \times S^2$ であることが判ります.

レンズ空間のことを簡単に解説しましたが, 「**Dehn の手術**」は結び目や絡み目の管状近傍に対してレンズ空間の構築を流用したものです. まず, 絡み目 $L = \cup_i S^1_i$ を用いた Dehn 手術は, 絡み目の成分 S^1_i $i \in \{1, \ldots, n\}$ の管状近傍をトーラス体で埋め戻す操作のことです. Dehn の手術に関連する重要な定理としては, 「**任意の閉３次元多様体は３次元球面 S^3 に埋め込まれたある絡み目 L に対する Dehn 手術から得られる**」という **Lickorish-Wallance の定理**があります. 結び目の交差の解消で用いられる Dehn 手術の様子を図 14.27 に示しておきましょう:

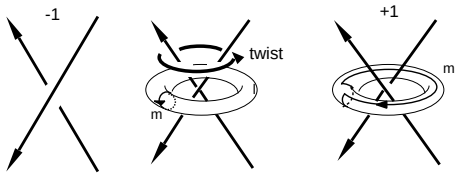


図 14.27: Dehn 手術について

図の左端に結び目の交差点を示します. この交差点の近傍でドーナッツを抜き出したものが中央の図で, 交点をドーナッツ中央に配置し, ドーナッツの longitude と結び目の交差数が 0 となるように, 具体的には結び目の一方の弧ともう一方の弧の向きが図のように逆向きになるように配置します. この配置の理由は Seifert 曲面のリボンをドーナッツで挟む状態にしたいからです. さて, ここで捻りを入れてドーナッツの貼り替えを行いますが, この捻りはドーナッツ上の longitude l と meridian m を使って定義します. つまり, 交差点での向き $p = \pm 1$ に従って $l \to l$, $m \to m \pm l$ で longitude と meridian を写し, もともとドーナッツ内部の円盤の境界となる m の像 $m \to m \pm l$ に円盤を貼り付けます. こうすることでドーナッツを新たに貼り付けることができます. 図 14.27 の場合, 右側に Dehn 手術の結果を示しています. そして, この手術を結び目が自明な結び目になるまで行いますが, 精々, 結び目の交差点数分の手術で済みます. これは, 結び目の射影図上で一点を定め, 結び目に入れた向きに従って移動させます. ここで自明な結び目でなければ交差点に到達する筈です. この交差点をはじめて通過する場合は, 現在, 移動している道が交差点の下道となるよう交差を変更します. もし, 以前通過した交差点であれば, 交差点をそのままにして通過します. この操

作を行った後, 結び目の射影図を横から見ると, 交差点で下に下へと移動しているので, 自明な結び目が得られます. そして, この操作は交差点の個数を越えることがありません. ここまで来ると簡単です. 最終的には自明な結び目と有限個のレンズ空間を包含する結び目の補空間が現われます.

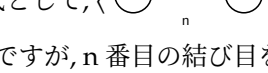
14.12 おまけ：スケイン多項式

Alexander 多項式の計算では, Wirtinger 表示を用いた方法の他に「**Seifert 曲面**」と呼ばれる曲面を結び目に貼って計算する幾何学的な方法, Dehn 手術と普遍被覆空間の構成で視覚的 (？) に求める方法があります. これらの方法も面白い方法ですが, これらとは全く別の方法で結び目の交点の局所的な入れ替えによる「**スケイン関係**」と呼ばれる関係式から多項式を計算する方法があります. このスケイン関係からは, Alexander 多項式だけではなく,「**Conway 多項式**」,「**Kauffman 多項式**」, そして,「**Jones 多項式**」等の結び目の不変量となる多項式が色々計算できます.
ここでは簡単にスケイン関係を用いた結び目多項式の計算を解説します. ここで結び目の多項式は Laurant 多項式環 $\mathbb{Z}[A, A^{-1}, Z, Z^{-1}]$ のイデアルの生成元とします.

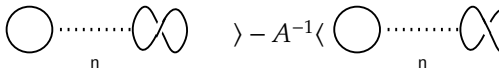
まず,「**スケイン関係式**」とは局所的に結び目の上下関係を入れ換えたものに対して結び目の多項式の間で成立する関係式のことです:

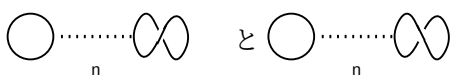
スケイン関係式

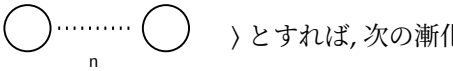
1. $\langle$ $\rangle = 1$
2. $A^{-1}\langle$ $\rangle - A\langle$ $\rangle = Z\langle$ $\rangle$

ここで, 上記の 1, 2 を使うと n 個の自明な結び目が並んでいる n 成分の自明な絡み目 ○…○ に対して, その絡み目の多項式として, $\langle$  $\rangle = \left(\frac{A-A^{-1}}{Z}\right)^{n-1}$ が得られます.

この多項式の計算方法ですが, n 番目の結び目を捻じった に上の 2 を適用すると次の関係式が成立しますね.

$A\langle$  $\rangle - A^{-1}\langle$ $\rangle = Z\langle$ $\rangle$

ここで,  と は に同値なので,

$K_n = \langle$  $\rangle$ とすれば, 次の漸化式が得られます:

スケイン関係式から得られる漸化式

- $K_1 = 1$
- $(A - A^{-1})K_n = ZK_{n+1}$

この漸化式から Skein 多項式が計算出来ます.

なお, Skein 多項式の A と Z に変数を設定しなおすことで, さまざまな結び目多項式が得られます:

スケイン多項式から得られる結び目多項式

多項式名	A の値	Z の値
Alexander 多項式	1	$t^{1/2} - t^{1/2}$
Conway 多項式	1	Z
Jones 多項式	t	$t^{1/2} - t^{1/2}$

ただし, Alexander 多項式は適当に $\pm t^{n/2}$ 倍する必要があります.

では, ここで図 14.28 に示すクローバー結び目のスケイン多項式を計算してみましょう:

この図 14.28 で丸で囲った交点からスケイン関係を適用しましょう. この計算で本質的なことは図 14.29 に示すスケイン関係に関連する結び目や絡み目をひたすら描くことです.

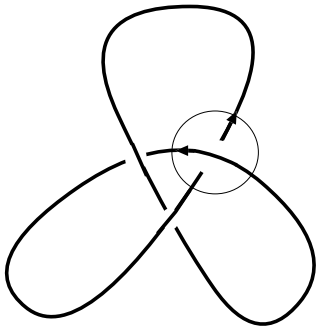


図 14.28: クローバー結び目

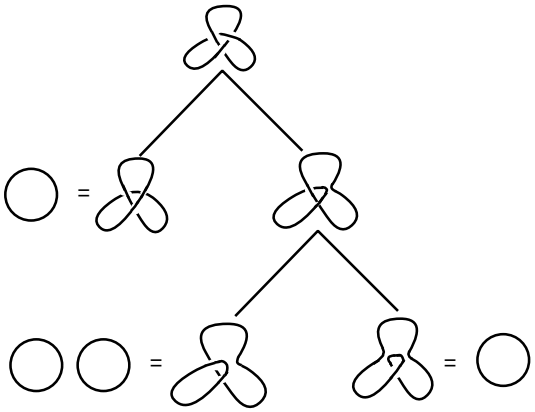


図 14.29: クローバー結び目のスケイン関係

このクローバー結び目の樹形図 14.29 の上 3 個から次のスケイン関係式が得られます: $A^{-1}\langle$ $\rangle - A\langle$ $\rangle = Z\langle$ $\rangle$

ここで, は自明な結び目と同値になるために, この結び目の多項式は 1 になります. また, は Hopf 絡み目と呼ばれる有名な絡み目です. そこで, Hopf 絡み目に対するスケイン関係式から多項式を計算します. 図 14.29 の下段がそれに対応します.

この下段から以下の関係式を得ます.

$A^{-1}\langle$ $\rangle - A\langle$ $\rangle = Z\langle$ $\rangle$

すると, が自明な結び目と同値で が二成分の自明な絡み目となるので, Hopf 絡み目 のスケイン多項式 $\langle$ $\rangle$ は $Z^{-1}A^{-2}\left(-AZ + A - A^{-1}\right)$ となり, この結果から, のス

ケイン多項式〈 〉は $A^{-4}\left(A^2Z^2+2A^2-1\right)$ が得られます.

Alexander 多項式の場合, $A=1, Z=t^{1/2}-t^{-1/2}$ を代入することで $t+t^{-1}-1 \sim t^2-t+1$ が得られます.

このスケイン多項式で鏡像を計算する場合は単純に A を A^{-1} に, Z を $-Z$ に置換えるだけで計算が出来ます.

すると, 先程の三葉結び目の鏡像のスケイン多項式は,$A^2Z^2-A^4+2A^2$ となります. これを Alexander 多項式に変換しても,t^2-t+1 となるので, 違いは判りません.

では, Jones 多項式で比較するとどうなるでしょうか？

三葉結び目の鏡像で Jones 多項式を比較

図 14.28 の Jones 多項式	t^3+t-1
図 14.28 の鏡像の Jones 多項式	t^3-t-1

このように Jones 多項式では多項式自体が異なります. すなわち, Jones 多項式は Alexander 多項式や Conway 多項式よりも強力な結び目多項式になります.

以上に示したように基本群から Fox の微分子を使って計算した Alexander 多項式は, 最終的には中学生でも出来そうなお絵描きと多項式の計算になりました v(^_^).

第15章　surfを使う話

外部アプリケーションを Maxima から利用する方法について, 代数曲線や曲面を描画するアプリケーション surf, あるいは surfer を用いて解説します. 具体的には surf/surfer を呼出して曲線や曲面を描画する surfplot という名前の函数を構築します. この函数は 2 変数の多項式を与えれば曲線, 3 変数の多項式を与えると曲面を描きます. 面白くて綺麗な絵が描けるので遊んでみて下さい.

15.1　代数曲面

Maxima でグラフを描ける数式は $y = f(x)$ や $z = h(x, y)$ といった形式, あるいは媒介変数を用いた函数にほぼ限定されます. そのために $x^2 + y^2 - 1 = 0$ を満す点のグラフを描きたい場合, $y = \sqrt{1 - x^2}$ と $y = -\sqrt{1 - x^2}$ のグラフを同時に描くか, 媒介変数を用いた $x(t) = 2t/t^2 + 1$ と $y(t) = t^2 - 1/t^2 + 1$ を描く等の工夫が必要になります.
そこで 2 変数や 3 変数の多項式の零点集合が簡単に描ける surf や surfer の登場となります.

15.2　surf/surferの概要

surf[1]は変数 x, y, z の多項式 p が定める零点集合 $V(p)$ を描くアプリケーションです. surf は GTK+ ライブラリを用いた GUI と C 風の原始的な処理言語を持ちます. この処理言語で行えることは, 幾つかの基本的な設定を行って多項式の零点を描く処理が中心で, あとは goto 文で多少の反復処理ができる程度です.
surf は UNIX 環境と MS-Windows 環境の双方で動作しますが, MS-Windows 環境では surf の GUI が使えませんが, GUI なしでも画像の生成が可能のために, 画像ファイルを生成する描画エンジンとして surf を利用する方法が中心です. 実際, surfex[2] surfer[3] では, surf を曲面描画専門のエンジンとして用いられています.
最初の surfex は JAVA で記述されたアプリケーションで, 曲面の計算で surf を用いていますが, surfex のスクリプトと surf のスクリプトには互換性が全くありません. もう一つの surfer は UNIX 環境や MS-Windows 環境[4]の双方で動作します. surfex よりも多少の互換性はあり, 曲面に対し, 単純な描画程度であれば互換性があります. ここでは図 15.1 に MS-Windows 版の Surfer の様子を示しておきます:

[1] http://surf.sourceforge.net/
[2] http://www.surfex.algebraicsurface.net
[3] http://www.imaginary2008.de/surfer.php
[4] surfer を MS-Windows 環境にインストールする際にユーザー名が日本語であればインストールに失敗することがあります. この場合は ID を ASCII 文字 (たとえば Administrator) でインストールすれば問題ありません

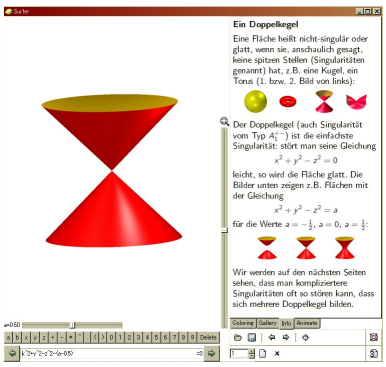


図 15.1: Surfer(MS-Windows 版)

なお, surfex, surfer 共に代数的曲面に限定され, 代数曲線の描画は行えません. そこで, 単純に画像の生成では surf のみ, 回転や拡大といった対話処理は surfer も用いる方針でプログラムを構築します. なお, この本では surf を利用するために必要な事項しか述べません. surf の詳細については surf のオンラインマニュアル[5], あるいは私の webpage に置いた翻訳[6] を参照して下さい.

15.3　Maxima から surf/surfer を使う方法

Maxima から外部のアプリケーションを使う方法として標準的な手法は system 函数からアプリケーションを呼出すことです. ここでは Maxima で計算した多項式を surf/surfer に引渡して描画させるのが目的ですが, system 函数で呼出しても Maxima 側の多項式が surf/surfer には引渡されません. そこで surf 言語のファイルを Maxima 側で生成し, それからファイルを surf/surfer に引き渡す方法を採用します. ここで UNIX 環境で surf を GUI 付きで描画するのであれば起動時にオプション “-x” を用います. たとえば, surf 言語ファイルが surf.pci であれば ‘surf -x surf.pci’ と入力すればよいのです. そして UNIX 環境で surf の GUI なしに描画エンジンとして使う場合はオプション “–no-gui” を用い, ‘surf –no-gui surf.pci’ とします. なお, MS-Windows 版の surf や surfer を用いる場合は単に ‘surf surf.pci’ や ‘surfer surf.pci’ でスクリプトを処理します.
そうとなれば Maxima で surf 用のスクリプトを生成してファイルに書出す函数を構築すれば十分です. その際に多項式の変数の制約がありますが, 最も簡単な方法は多項式の変数を完全に x, y, z に限定し, 変数が 2 個あれば曲線, 3 個ならば曲面を描くように予め用意した surf の設定ファイルに曲線や曲面の方程式として多項式を引渡しても良いのです.
ただし, この方法では x, y, z 以外の変数が使えません. それに設定ファイルを準備するのもあまり洗練された方法ではなく, 設定ファイルの内容を必要に応じて Maxima から変更できない点も面白くありません. ここではもう少し Maxima に複雑な処理を実行させましょう.

[5] http://surf.sourceforge.net/doc.shtml
[6] http://www.bekkoame.ne.jp/ ponpoko/Math/surf/SurfExamples.html

15.4 surf の設定と Maxima への取込み方法

15.4.1 共通設定

surf で絵を描かせる場合, 方程式を満す点を求めるだけではなく, 描画する絵の大きさを指定しておく必要があります. このために surf には次に示す助変数を持っています:

surf の求解に関する助変数

助変数名	概要	ここでの設定例
root_finder	解法の設定	root_finder=d_chain_bisection;
epsilon	解の精度	epsilon=0.0000000001;
iterations	反復の回数	iterations=20000;
width	絵の横方向の画素数	width=500;
height	絵の縦方向の画素数	height=500;

root_finder: 解法を指定します. ここで指定可能な解法はいろいろありますが, 自己交差を持つ曲面, 曲面と平面との断面が綺麗に描ける d_chain_bisection 法を採用します.

epsilon: 解の精度を指定する助変数です. ここでは既定値をそのまま用いますが, 必要に応じて小さくしても構いません.

iteration: 反復計算の上限を定めます. ここでの値も既定値のままでも構わないでしょう.

width と height: これらの助変数は絵の大きさを画素単位で指定します. 既定値の画像の大きさは 200×200 と最近の PC では流石に小さ過ぎるので, 見映えの良い 500×500 で表示させることにしましょう. 勿論, 計算機が強力であれば貴方の好みの値で構いません.

15.4.2 曲面固有の設定

曲面の場合は共通設定に加えて次の設定を追加します:

曲面の場合の設定

助変数	概要	
do_background	背景の描画	do_background=yes;
background_red	背景色の設定 (R)	background_red=255;
background_green	背景色の設定 (G)	background_green=255;
background_blue	背景色の設定 (B)	background_blue=255;
rot_x	対象の回転 (X)	rot_x=0.14;
rot_y	対象の回転 (Y)	rot_y=-0.3;
rot_z	対象の回転 (Z)	rot_z=0;
scale_x	対象の拡大 (X)	scale_x=1.0;
scale_y	対象の拡大 (Y)	scale_y=1.0;
scale_z	対象の拡大 (Z)	scale_z=1.0;
transparence	対象の透過度	transparence=0;
illumination	対象の照明	illumination=transmitted_light +reflected_light+diffuse_light +ambient_light;

背景の設定: 背景の設定は 'do_background=yes;' で背景色の設定の有無を指定し, 'background_red', 'background_green' と 'background_blue' の RGB で背景色を 0 から 255 の範囲の整数で指定します.

対象の回転: 対象の回転は rot_x , rot_y rot_z にそれぞれ X, Y, Z 軸に対する回転角を弧度で指定することで行います.

対象の拡大: 対象の拡大は scale_x, scale_y と scale_z の 3 個の助変数で行います.

対象の透過度: transparence には物体を透過する光の割合を 0 から 100 までの数値で指定します. 0 の場合は光は曲面を全て透過せずに反射されますが, 100 になると全ての光が曲面を透過します. 既定値は 0 です.

対象の照明: 対象の照明 (illumination) には全体光 (ambient_light), 散乱光 (diffuse_light) と反射光 (reflected_light) があり, 考慮すべき光を単純に演算子 "+" でつないだものを指定します. ここでは全てを考慮することにしましょう.

15.4.3 助変数の Maxima での表現方法

これらの値をファイルに書込むのが面倒であれば, Maxima の大域変数として表現する方法が第一に挙げられます. ただし, この方法の弱点は変数値の管理が弱い点です. そもそも, 変数が多くなれば Maxima 固有の大域変数がどれで, surf の設定変数がどれがが全く不明瞭になります. ここでは

趣を変えて Maxima の属性を用いてみましょう. こうすると surf の助変数の値を一括して扱うことが容易になります.
そこで Maxima の初期化ファイル maxima-init.mac に次の設定を行います:

```
put(surfg, "d_chain_bisection",root_finder );
put(surfg, 20000,iterations );
put(surfg, 500,width);
put(surfg, 500,height);

put(surf, "yes",do_background);
put(surf, 255,  background_red);
put(surf, 255,  background_green);
put(surf, 255,  background_blue);
put(surf,   0.14, rot_x );
put(surf,  -0.3,  rot_y );
put(surf,   0.0,  rot_z );
put(surf,   1.0,  scale_x );
put(surf,   1.0,  scale_y );
put(surf,   1.0,  scale_z );
put(surf,   0,    transparence);
put(surf, transmitted_light+reflected_light+diffuse_light
        +ambient_light,illumination);
```

ここでは surf 用に二種類の大域変数 surfg と大域変数 surf を用意し, これらの大域変数の属性として助変数, 属性値として助変数に割当てる値を与えます. なお, 2 次元と 3 次元の共用の設定を大域変数 surfg の属性として与え, 曲面専用の設定を大域変数 surf の属性として与えることにします.
さて, Maxima にて対象の属性値設定では put 函数を用い, 属性値の取出しでは get 函数を用います. たとえば, surf の background_red 属性に 255 を設定したければ 'put(surf, 255,background_red);', background_red の属性値を取出したければ, 'get(surf,backgraound_red);' とします.
次に surf の描画ファイルに大域変数 surf と大域変数 surfg に記述した属性の書出を行う個所の構成を考えましょう. ここで surf と surfg の属性は合せて 10 個あります. この属性を一々函数で指定して行くことは流石に面倒なので, properties 函数を使って surf と surfg の属性に何があるか一覧を出させ, その属性名と属性値の等式を作ります. 要するに次の処理を自動的に行うように do 文を組込めば良いのです:

```
(%i18) properties(surfg);
(%o18) [["user properties",height,width,iterations,root_finder]]
(%i19) properties(surf);
(%o19) [["user properties",scale_z, scale_y, scale_x, rot_z,rot_y,rot_x,
          background_blue,background_green,background_red,do_background]]
(%i20) %[1][6]=get(surf,%[1][6]);
(%o20) rot_y = -0.3
```

この例では properties 函数で大域変数の属性リストを出力させています. 属性リストは通常, 第 1 成

分のリストに属性名が登録され, 先頭が ‘user properties’ なので第 2 番目以降の成分を演算子 “=” の左側に置き, その成分の属性値を演算子 “=” の右側に置けば surf 向けの等式ができあがります. この処理で構築した等式 (上の例では ‘rot_y=-0.3)’ を Maxima の配列に入れて適当なファイルへの書出しを行う函数, たとえば, stringout 函数を用いて配列の内容をファイルに書出してしまえば良いのです. このようにしておけば別に変数名 (Maxima では属性) を憶えていなくても機械的に処理が行えるので, その属性の減増が容易になります.

15.5　多項式の処理

surf で使える数値は整数, 代数的整数や倍精度浮動小数点数で, 多倍長浮動小数点数や複素数が使える訳ではありません. さらに Maxima の浮動小数点数の書式は surf 側でエラーになります. そこで, このような余計な問題を避けるために多項式を expand 函数で展開したあとに ratsimp 函数で簡約化します. ここで ratsimp 函数によって多項式の係数は有理数に近似されるので, 浮動小数点の書式に関する問題は生じなくなります.
次に重要なことですが, surf で利用可能な多項式は変数 x, y, z の多項式に限定されます. 勿論, surf は x, y, z 以外の変数も扱えますが, これらの変数は内部の補助的な変数として扱われ, あくまでも描かれるグラフは XY-平面, あるいは XYZ-空間に限定されます.
そのために多項式の変数を 2 から 3 個に限定し, 変数名が, x, y, z 以外の変数を, x, y, z の変数名で置換える操作もあった方が良いでしょう.
ここで, 多項式の変数を返す函数として Maxima には showratvars 函数があります.
たとえば ‘showratvars(x+yˆ2+aˆ3);’ の結果は ‘[a,x]’ となります. つまり, 返却値のリストは Maxima の変数順序 “$>_m$” で小さな順で変数が左から並びます. この Maxima の変数順序 “$>_m$” は基本的に逆アルファベット順 (“a” よりも “z” が小さい. または小文字は大文字よりも小さい) のために変数を順序 “$>_m$” に対して小さいものから順番に並べてしまうと実際は通常のアルファベット順になります. そこで showratvars 函数で得られた変数リスト vars の左側から順番に x, y, z を対応させれば良いことになります. 具体的には ‘vars[1]’ を ‘x’, ‘vars[2]’ を ‘y’ , ‘vars[3]’ を ‘z’ で置換するのです.
この目的には subst 函数が使えそうですが, subst 函数は置換リストの左から右へと代入を逐次遂行するため, ‘subst([a=x,x=y,y=z],a+x+y)’ は ‘3*z’ となり．期待した値にはなりません．そこで, 置換を段階を踏んで処理します．すなわち, 最初に多項式の変数名を局所的変数名 ‘surf_tmp_x’, ‘surf_tmp_y’ と ‘surf_tmp_z’ で置換え, 次に, これらの変数名を ‘x’, ‘y’, ‘z’ に最終的に置換えるのです.
ところで, 与式に ‘sqrt(2)’ のような函数項が含まれているとき, showratvars 函数は函数項を変数リストに追加してしまいます. そこで, 与式に予め float 函数を作用させて函数項や係数を倍精度浮動小数点に変換しておけば, この事態を避けられます.

15.6　曲線と曲面の描画の違いについて

surf で曲線や曲面を描くためには, その方程式に対応する surf の描画命令をスクリプトの末尾に追加しなければなりません. この処理は曲線で異なるので曲線と曲面の場合に分けて surf の描画命令

を追加します.

曲線の場合: 多項式を変数 curve に割当て, 最後に draw_curve 命令にを追加します.

曲面の場合: 多項式を変数 surface に割当て, 最後に描画命令である draw_curve 命令を追加します.

surf による画像ファイルの出力と準備: 前述のように surf の GUI は MS-Windows 環境や最近の UNIX のデスクトップ環境でちゃんと動作しないことがあります. そのために surf を GUI なしで利用するか, surfer を利用するかを選択する必要があります. ただし, surfer は名前のとおり曲面を描くことが専門で曲線は描けません. そのために surf を使った画像ファイル出力について解説しておかなければなりません. 今回, 利用する surf の画像ファイルに関連する変数を纏めておきましょう:

画像ファイル出力に関連する変数

変数	概要	
color_file_format	画像ファイルの形式	color_file_format=jpg;
filename	表示画像名	filename=”surf.jpeg”;
save_color_image	表示画像の保存	save_color_image;

ここで MS-Windows 版の surf は画像ファイル形式を “ppm”」にしていなければまともに動作しないようです. この PPM 形式のファイルが表示可能なアプリケーションが素の MS-Windows では見当りません. ここで PPM 形式が扱える商用のアプリケーションを画像閲覧用に指定するもの良いでしょうが, 誰もが自由に使えるソフトウエアを使いたいところです. そのために MS-Windows 環境では NetPbm パッケージ[7]の ppmtojpeg や ImageMagick[8]の convert を使って surf で生成した PPM 形式のファイルを JPEG 形式に変換し, それを explorer で表示するようにします. パッケージサイズとしては NetPbm の方が ImageMagick よりも小さいのですが, ImageMagick の方が機能が上なので, ここでは ImageMagick を入手しているものとして話を進めますが, NetPbm を使う話も併記しておきます.
なお, MS-Windows 環境での画像生成, 変換と描画はバッチ処理になるので, surfer や NetPbm といったアプリケーションのインストールに加え, これらを DOS 窓から利用するために環境変数 Path の設定がさらに必要になります. この処理を次に纏めておきます:

[7]NetPbm for Windows: http://gnuwin32.sourceforge.net/packages/netpbm.htm から入手可能
[8]ImageMagick: http://www.imagemagick.org/www/binary-releases.html#windows から入手可能

MS-Windows 環境での事前準備

1. surfer をインストール
2. surfer をインストールしたディレクトリ/フォルダを環境変数 Path に追加
3. NetPbm か ImageMagick のどちらかをインストール. ここでは ImageMagick を推奨
4. NetPbm を選んだ場合は NetPbm インストールしたフォルダ (通常は で C:\Program Files\GnuWin32)+の bin フォルダを環境変数 Path に追加

さて, これでプログラムを UNIX 環境と MS-Windows 環境向けの二つを書いても良いのですが, ここでは UNIX 環境と MS-Window 環境のどちらでも利用可能なプログラムを書きます. では, どのようにして UNIX 環境か MS-Windows 環境かを判別すれば良いでしょうか？ ここで参考になるのが Maxima そのものです. Maxima ではグラフ表示で gnuplot を指定していても, MS-Windows 環境のみ wgnuplot を用います. この設定は内部変数*autoconf-win32*を用いて行います. MS-Windows 環境に限って大域変数 gnuplot_command の値が 'wgnuplot' になっていますが, この大域変数 gnuplot_command は内部変数*autoconf-win32*で決定されているのです. だから, 内部変数*autoconf-win32*を判断に使えば良いのです. ここで UNIX 環境と MS-Windows 環境の処理の違いをまとめておきましょう:

UNIX 環境と MS-Windows 環境の処理の相違点

1. 曲面の表示:surfer を利用するために処理の違いはない
2. surf のオプション:UNIX 環境ではオプションに "–no-gui" が必要だが, M S-Windows 環境では不要
3. 曲線の画像データ:UNIX 環境では JPEG 形式, MS-Windows 環境では PPM 形式のファイルを生成. MS-Windows 環境で PPM 形式のファイル閲覧ソフトがないために, より一般的な JPEG 形式のファイルに NetPbm パッケージの ppmtojpeg や ImageMagick の convert を利用する
4. 曲線の表示:UNIX 環境では JPEG 形式のファイルを ImageMagick の display で閲覧. MS-Windows 環境では explorer.exe を利用して閲覧
5. echo 命令:UNIX 環境のシェルの echo と MS-Windows 環境の DOS 窓の echo は仕様が少し違う. UNIX 環境のシェルの echo では文字列は二重引用符で括らなければならないが, DOS 窓ではその必要はない.

さて, 最期にシェルと DOS 窓の話があります. これは曲面や曲線の表示に依存する項目を echo 命令を利用して surf のスクリプトファイルに追加するためです. この方法を上手に利用すると後述するように曲面の断面を描くといった, さまざまな利用が可能になるからです. ここで UNIX 環境のシェルの echo 命令と DOS 窓の echo 命令では多少勝手が異なりますが, 殆ど同じ使い方ができます. ここで注意することは, UNIX 環境で echo 命令は表示させる文字の列を二重引用符""で括らなければなりません:

UNIX 環境での命令文

```
echo "color_file_format=jpg;filename=\"surf.jpeg\";save_color_image;">>surf.
    tmp
```

この例では surf の設定を surf.tmp ファイルに追加するものです. これと同じことを DOS 窓の echo で行う際に DOS 窓で文字の列を二重引用符で括ると, その二重引用符も一緒に表示されるので, UNIX 環境と同じ結果を得るためには二重引用符が不要です:

MS-Windows 環境での命令文

```
echo color_file_format=ppm;filename="surf.ppm";save_color_image;>>surf.tmp
```

ここで system 函数では文字列を引数とするので, これらの命令を二重引用符で括らなければなりません. そのときに解釈の間違いを避けるために命令中の “\” と二重引用符には “\” を追加します. なお, 命令中の “\”” は二重引用符”と “\” の双方に追加するので “\\\”” で置換えられます.

15.7 surfplot.mc

上述の方針に従って構成した Maxima の函数 surfplot を次に示します:

```
1  /* MAXIMA */
2
3  /* 属性の設定.*/
4  /* surfg の属性設定
5
6     surfg には平面曲線と空間曲面を描く際に用いる共通の設定を入れます.
7     root_finder 零点を計算する際の解法の指定.
8                 d_chain_bisection を用いると自己交差も綺麗に描きます.
9     iterations : 零点を計算する際の繰返しの上限を設定
10    width:      画像の横幅
11    height:     画像の高さ
12 */
13 put(surfg, d_chain_bisection, root_finder );
14 put(surfg, 0.0000000001,epsilon);
15 put(surfg, 20000, iterations );
16 put(surfg, 500,width);
17 put(surfg, 500,height);
18
19 /* surf の属性設定
20
21    surf には空間曲面を描く際に用いる設定を入れます.
22    surf では色の指定は RGB で行い, 0から 255までの整数を指定します.
23    do_background:   背景色
24    background_red:  背景色の指定 (赤)
25    background_green: 背景色の指定 (緑)
26    background_blue: 背景色の指定 (青)
27    rot_x:           X 軸回りの回転角度(rad)
28    rot_y:           Y 軸回りの回転角度(rad)
29    rot_z:           Z 軸回りの回転角度(rad)
30    scale_x:         X 軸方向の倍率
31    scale_y:         Y 軸方向の倍率
```

```
32      scale_z:              Z 軸方向の倍率
33      transparence:         透明度 0−100, 0でsolid, 100で透明
34
35  */
36  put(surf, yes,do_background);
37  put(surf, 0,background_red);
38  put(surf, 0,background_green);
39  put(surf, 0,background_blue);
40  put(surf, 0.14, rot_x );
41  put(surf,−0.3, rot_y );
42  put(surf, 0.0, rot_z );
43  put(surf, 1.0, scale_x );
44  put(surf, 1.0, scale_y );
45  put(surf, 1.0, scale_z );
46  put(surf, 0, transparence);
47  put(surf, ambient_light+diffuse_light+
48              reflected_light +transmitted_light,
49             illumination);
50  /* surfplot
51
52      引数は多項式. 多項式の変数は 2個か 3個でなければエラーになります.
53      描画はsurf を用いますが,こ の函数では臨時ファイルとしてsurf.tmp に
54      surf のスクリプトを書込み, system 函数で画像の生成を行います.
55      猶, 曲面をsurfer で生成と描画を行い, 曲線はsurf で生成した画像を
56      Viewer で表示させます.
57  */
58
59  surfplot(f):=block(
60  [
61   poly,poly0,vars, lls1 :0, lls2 :0,
62   f:ratsimp(expand(f)),tmp,
63   str , target , j , sl , obj ,
64   ls1 : properties(surfg),
65   ls2 : properties(surf ),
66   delcmd,drwcmd,cnvcmd,execmd,dspcmd
67  ],
68  vars:showratvars(float(f )),
69  n:length(vars),
70  display2d:false ,
71  if n=2 or n=3 then
72     ( lls1 : length(ls1 [1])−1,
73       for i :1 thru lls1 do
74           ( str : ls1 [1][ i +1],
75             ( if str=epsilon then tmp:rat(get(surfg, str ))
76                  else tmp:get(surfg,str )),
77             surf_settings [ i ]: str=tmp
78             ),
79       if n=3 then
80          ( lls2 : length(ls2 [1])−1,
81            for i :1 thru lls2 do
82                ( str : ls2 [1][ i +1],
83                  j : i+lls1 ,
84                  surf_settings [ j ]: str=get(surf, str )
85                  ),
```

```
86          /* 変数の入換を行います.*/
87          poly0:subst([vars[1]=surf_tmp_x,vars[2]=surf_tmp_y,
88                       vars[3]=surf_tmp_z],f),
89          poly:subst([surf_tmp_x=x,surf_tmp_y=y,surf_tmp_z=z],poly0),
90          /* 曲面を描く為の設定 */
91          target:surface,
92          obj:draw_surface)
93      else
94          (
95          /* 変数の入換を行います.*/
96          poly0:subst([vars[1]=surf_tmp_x,vars[2]=surf_tmp_y],f),
97          poly:subst([surf_tmp_x=x,surf_tmp_y=y],poly0),
98          /* 曲線を描く為の設定 */
99          target:curve,
100         obj:draw_curve),
101   /* 配列sl を定義し,描画設定と曲線/曲面の方程式と描画命令を入れます */
102    j:lls1+lls2,
103    array(sl,j+2),
104   (for i:0 thru j-1 do
105         sl[i]:surf_settings[i+1]),
106    sl[j]:target=poly,
107    sl[j+1]:obj,
108    /* 只今,描画中… */
109    print("Surf is now drawing ", poly,". Please wait ...."),
110    /* stringout 函数で,スクリプトの式を書込みます. */
111    if n=2 then
112      (stringout("surf.tmp",clear_screen,sl[0], sl[1], sl[2], sl[3],
113                                          sl[4], sl[5], sl[6]))
114    else
115      (stringout("surf.tmp",clear_screen,sl[0], sl[1], sl[2], sl[3],
116                 sl[4], sl[5], sl[6], sl[7], sl[8], sl[9], sl[10], sl[11],
117                    sl[12], sl[13], sl[14], sl[15], sl[16], sl[17], sl[18])),
118    if n=3 then
119    /* system 函数で surfer を起動します.surf の GUI 付が動作するのであれば,
120       "surf -x surf.tmp"としても構いません. */
121       system("surfer surf.tmp >surf.log&")
122    else
123    /* MS-Windows であるかは, 大域変数gnuplot_command の値で判断します
124     このプログラムをMS-Windows 上で動作させるためには
125     NetPbm for Windows か ImageMagick のどちらかと,
126     surfer のインストールが必要です. そして.これらのアプリケーションへの
127     環境変数Path の設定が不可欠です. なお, ここではImageMagick を
128     使う事を前提にしていますが, NetPbm の場合は cnvcmd を
129     cnvcmd:"ppmtojpeg surf.ppm>surf.jpg"
130     に変更するだけで大丈夫です.
131    */
132       (if gnuplot_command="wgnuplot" then
133         (delcmd:"del surf.ppm surf.jpg",
134          drwcmd:"echo color_file_format=ppm;\
135 filename=\"surf.ppm\";save_color_image;>>surf.tmp",
136          execmd:"surf surf.tmp>surf.log",
137          /* ImageMagick の場合 */
138          cnvcmd:"convert surf.ppm surf.jpg",
139          /* NetPbm の場合 */
```

```
/*          cnvcmd:"ppmtojpeg surf.ppm>surf.jpg", */
            dspcmd:"explorer surf.jpg")
          else
           (delcmd:"rm surf.jpeg",
            drwcmd:"echo \"color_file_format=jpg;\
filename=\\\"surf.jpeg\\\";save_color_image;\">>surf.tmp",
            execmd:"surf --no-gui surf.tmp>surf.log",
            cnvcmd:"",
            dspcmd:"display surf.jpeg&"),
       /* system 函数による準備
          surf が GUI 付きで利用可能であれば, 以下のsystem 函数を
          全て削除し, system("surf -x surf.tmp>surf.log&")で
          置換します.
       */
         system(delcmd),
         system(drwcmd),
         system(execmd),
         system(cnvcmd),
         system(dspcmd)
          )
       )
else
/* 多項式が,2変数でも 3変数でもなければエラーを表示します. */
print("Error !"));
```

15.7.1 surfplot.mc の使い方

まず, surfplot.mc ファイル を読込みましょう. なお, Maxima の利用者定義の函数を含むファイルは load 函数で読込めます. ここで surfplot.mc が Maxima を起動したディレクトリ上にある場合は `load("surfplot.mc");` で読込めますが, もし, surfplot.mc が Maxima を起動したディレクトリ上になければ surfplot.mc を置いたディレクトリを直接指定します. このとき, 相対経路 (カレントディレクトリを基準とする経路) でも絶対経路 (ルートディレクトリを基準とする経路) のどちらでも構いません.

load 函数で surfplot を読込んだあとは, surfplot に多項式を書込むだけで描画を行います. では, 曲面や曲線をいろいろ描いて遊んでみましょう.

15.8 簡単な例

手始めに半径が 1 の球面を描いてみましょう. 球面の方程式は $x^2+y^2+z^2-1=0$ で与えられます. したがって, `surfplot(x^2+y^2+z^2-1)` と入力すると surf が立ち上がって球面を描きます. これだけでは面白く無いので, もう少し捻りの効いた方程式を描いてみましょう. そこで, 次の 3 個の球面の方程式の積はどのような曲面を描くでしょうか？

3 個の球面の方程式の積

$$(x^2+y^2+z^2-4)\left((x-2)^2+(y-2)^2+z^2-\sqrt{2}\right)\left((x+2)^2+(y-2)^2+z^2-\sqrt{2}\right)$$

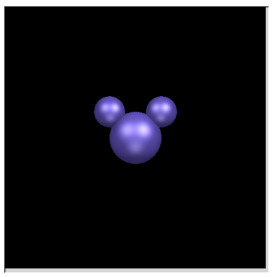


図 15.2: ？曲面

この 3 個の球面の方程式の積で表現される式は図 15.2 に示す 3 個の球面で構成された曲面になります. それにしても, この曲面は何処かで見たことがあるような気がしますが, この曲面は原点を中心とする半径 2 の球面を中心に,$(2,2,0)$ と $(-2,2,0)$ を中心とする半径 $\sqrt{2}$ の球面でできています. このように複数の曲面の方程式の積でできた方程式から得られる曲面は各曲面の和集合になります. 正確に言えば, 多項式 $f \in K[x_1,\cdots,x_n]$ に対して $V(f)$ を多項式 f の零点集合とするときに $V(f_1 \cdot f_2 \cdots f_n)$ は $V(f_1) \cup V(f_2) \cup \cdots \cup V(f_n)$ となります.
このことから, 多項式の零点集合を描くことは多項式の各因子の零点集合の和集合を描くことを意味します. すなわち, 多項式の既約因子分解ができると, あとは各既約な多項式を調べてしまえば十分なことが判ります. さらに既約な元は多項式環 $K[x_1,\cdots,x_n]$ の素イデアルの生成元となるので, 結局, 曲線や曲面は素イデアルと密接に関連します.
このことを怪しい絵を描く方法に適用するのであれば, 片っ端から描きたい方程式の積を surf/surfer で描かせれば良いのです. たとえば, 以下の 3 個の円と楕円の積を描いたものは図 15.3 に示す「アヒルの顔」になります:

3 個の円の方程式と楕円の積

$$(x^2+y^2-9)\cdot((x-2)^2+(y-1)^2-1)\cdot((x+2)^2+(y-1)^2-1)\cdot(x^2/9+2*(y+1)^2-1)$$

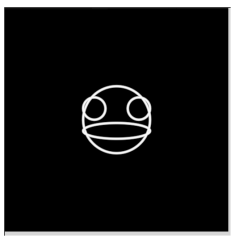


図 15.3: アヒル？

こんな使い方だけではなく, 座標を付けて曲線を描きたい場合にも使えます. この場合は座標軸の方程式と曲線の方程式の積を描けば良いのです. たとえば, 原点を通る X 軸と Y 軸を描きたいのであれば, X 軸の方程式が '$y=0$' で Y 軸の方程式が '$x=0$' となるので, 描きたい曲線に '$x*y$' をかけた多項式を描けば良いのです.

具体例を挙げておきましょう. `surfplot((x^2*(9-x^2)-4*y^2)*x*y);` と入力した結果を図 15.4 に示しておきます:

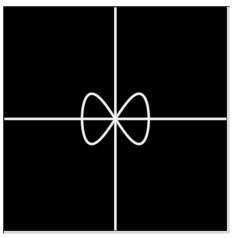


図 15.4: レムニスケート

このようにマジメな使い方もできます. が, いろいろとふざけた使い方をしてみるのも楽しいかと思います.

15.9　Barth Diec

さて, surfplot を使って派手な曲面を描いてみましょう. そこで, 次に示す式の零点集合の曲面を描いてみましょう:

Barth Diec

$$\begin{cases} 8(x^2-\tau^4y^2)(y^2-\tau^4z^2)(z^2-\tau^4x^2)(x^4+y^4+z^4-2(x^2y^2+y^2z^2+z^2x^2)) \\ +(3+5\tau)(x^2+y^2+z^2-1)^2(x^2+y^2+z^2-(2-\tau))^2 \\ \\ \tau=\frac{1+\sqrt{5}}{2} \end{cases}$$

この式の零点集合は図 15.5 に示す派手派手な曲面になります. ここで τ を代数的数ではなく浮動小数点数にするとどうなるでしょうか？ その結果を図 15.6 に示します:

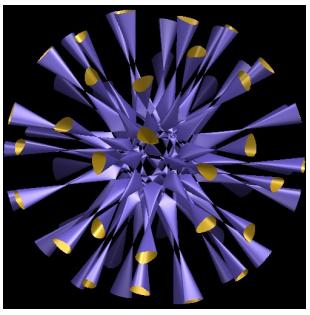


図 15.5: Barth Diec

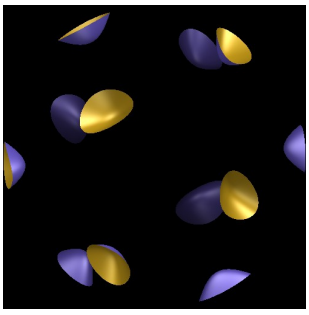


図 15.6: 近似計算による Barth Diec

このように図 15.6 の結果は図 15.5 とは随分と異なる図形になります.

この現象が生じたのも, $\tau = \dfrac{1+\sqrt{5}}{2}$ を代数的数ではなく浮動小数点数という近似値 $\tau = 1.618033988749895$ を用いたためです. この様に, 多項式の近似とは言え, その零点集合に関しては全く近似になっていないことを示しています.
surf にしても最終的には浮動小数点数の計算を行っているので, この事態を単純に「数式処理 VS. 数値計算」の構図に持ち込むことはできません. 寧ろ, 扱う式や数の意味を考慮しないで機械的な処理をさせた悪い例なのです.

15.10 Steiner のローマ曲面

15.10.1 曲面の概要

今度は Steiner のローマ曲面を描いてみましょう. このローマ曲面は $x^2y^2 + x^2z^2 + y^2z^2 - 17xyz = 0$ を満す点の集合で, 写像 $f : (x, y, z) \to (xy, yz, zx)$ による原点を中心とする半径 1 の球面の像で, 2 次元の実射影空間の 3 次元空間への「嵌込み」と呼ばれる閉曲面の 3 次元空間での配置の一つになります. ここで閉曲面とは, 球面やドーナッツ状の曲面のように縁がない曲面です. そして, 2 次元の実射影空間 (=実射影平面) は図 15.7 に示すように Möbius の帯の境界 (=円) に円盤を貼ってできる曲面と位相幾何学的には同じもの (=同相) になります:

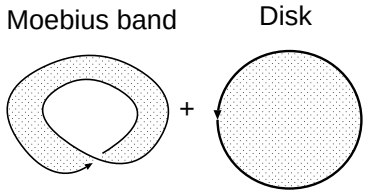


図 15.7: 実射影平面

この曲面には裏と表がありません. 実際, 帯をひとひねりして構成する「Möbius の帯」には裏表がありませんが, この帯の境界に円盤を貼っても裏表がないことに変わりはありません. このように裏表のない曲面のことを「向付け不可能な曲面」と呼びます. その一方で, 裏表のある曲面を「向付け可能な曲面」と呼びます. この「向付け」には別の見方があります. これは通常の 3 次元空間 ($\mathbb{R}^3$) 内部で曲面上の円を曲面の一つの法線に沿って動かしても向付け可能な曲面であれば円を曲面と交叉せずに浮かすことができますが, 向付け不可能な曲面ではどのように動かしても必ず曲面と交叉する円が存在するという特徴があります. この様子を図 15.8 に示しておきましょう:
図 15.8 に示すよう向付け可能な曲面では, その表面の円を曲面と交叉しないように持ち上げることができます. この例では左側の Torus で曲面上の一点に潰れない円を曲面と交叉しないように動かしています. これは裏と表がある場合は「表から XX 上」とか「表から YY 下」とすることができるので, 曲面上に置いた円を曲面と交叉しないように持ち上げられるのです. つまり, 向付け可能な曲面では, その曲面の法線を大域的に決められます.
ところが, Möbius の帯の基軸となる中心線を持ち上げると, どのように動かしても必ず曲面と交叉します. この様に, 向き付け不可能な曲面上に置いた円を曲面から剥そうとしても, 必ず曲面と交差

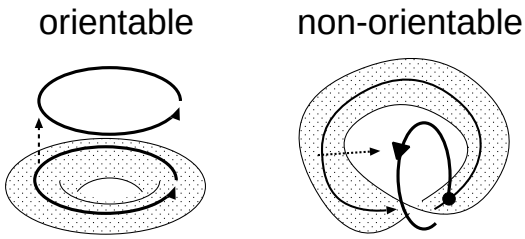


図 15.8: 向付けについて

してしまいます. つまり, 向き付け不可能な曲面で, 法線ベクトルを返す函数を考えると, その函数は何処かで必ず 0 になること, すなわち, 曲面の法線が大域的に決められないことを意味します.
ここで曲面の分類では, 向付け可能な曲面は 穴の沢山あるドーナッツ状の曲面, すなわち, 球にハンドルを貼り付けた曲面に分類され, さらに, ハンドルの本数 (=穴の数, つまり, genus) で一意に決まることが知られています:

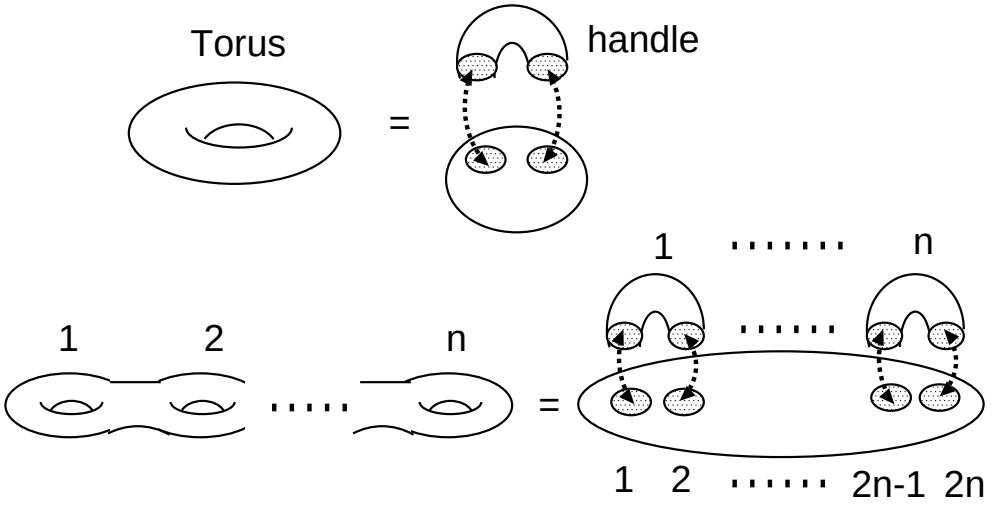


図 15.9: 向付け可能な平面の分類

まず, 図 15.9 の左上には Torus と呼ばれるドーナッツ状の平面があります. この Torus は右側に示すように球面から二つ円盤を外し, その円盤の境界に沿ってハンドル, すなわち, 円筒の境界を接着することで得られます. このとき, 円筒と曲面の向きから得られる境界の向きに沿って接着を行います.
より一般の場合, 図 15.9 の下に示すように球面から $2n$ 個の円盤を除いてできた境界に沿って, n 本のハンドルを Torus のように向きに注意して貼り合せます.
これに対し, 向付け不可能な曲面は向付け可能な曲面から円盤を取出して, そこに円盤を取り除いた射影曲面 (=Möbius の帯) を境界に沿って貼り合わせた曲面として得られることが知られています:

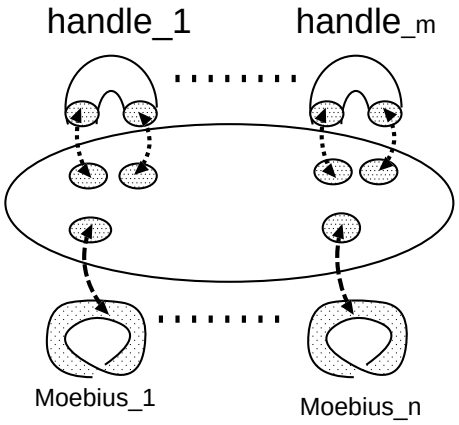


図 15.10: 向付け不可能な平面の分類

ここで向き付け不可能な曲面として Klein bottle を挙げておきましょう．この Klein bottle は図 15.11 に示す様に二つの Möbius の輪で構成された向付け不可能な曲面です:

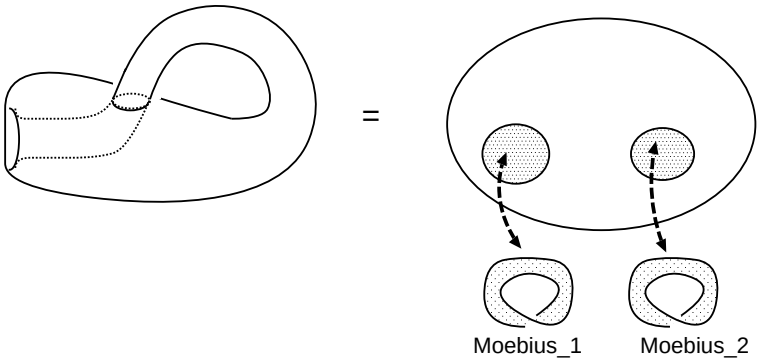


図 15.11: Kleinbottle の構成

以上の話を纏めておきましょう. 先ず, 向き付け可能な閉曲面はハンドルの本数で分類され, 向き付け不可能な曲面はハンドルの本数と射影曲面の個数で分類できます. ここで向き付け可能な曲面の射影曲面の個数を 0 とすれば, 閉曲面はハンドルと射影曲面の個数でだけで分類できることが判りますね. この様に曲面の分類は判り易いのが特徴です.

15.10.2 Steiner のローマ曲面の描画

さて, surfplot を使ってローマ曲面を描いてみましょう.
`surfplot(x^2*y^2+x^2*z^2+y^2*z^2-17*x*y*z);` と入力して下さい.
すると図 15.12 に示す絵が得られます:

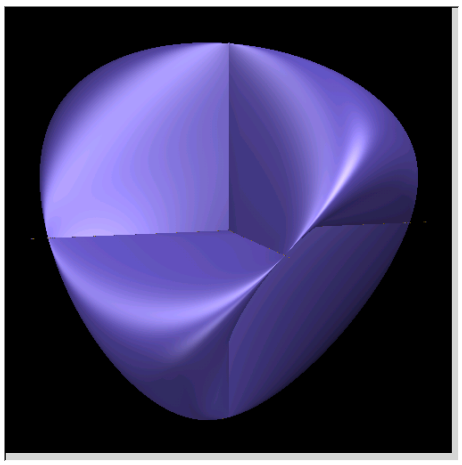
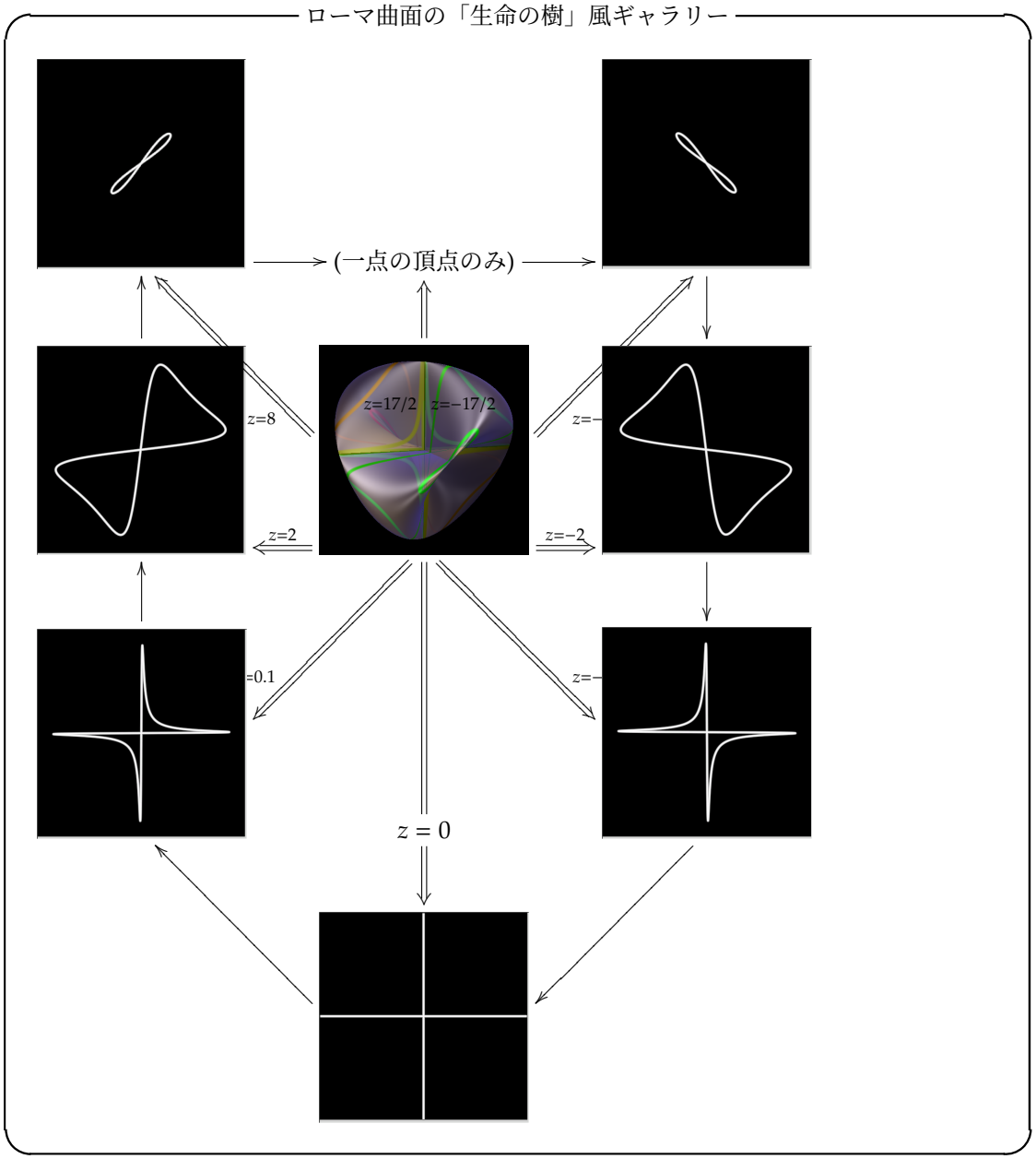


図 15.12: Steiner のローマ曲面

この絵で判る様にローマ曲面は X, Y, Z 軸で潰れた閉曲面になっています. これだけでは何がどうなっているのか判りませんね. そこで, 曲面を平面で切った面, すなわち, 平面による断面を見てみましょう. そのためには断面の方程式を Maxima で計算しなければなりませんが, これは簡単なことです.

まず, 平面の方程式は $(0,0,0)$ と異なる 3 個の実数 (a,b,c) と実数 d を用いて '$ax+by+cz-d=0$' で表現されます. ここで a,b,c のどれか一つは 0 ではないので, ここでは c が零でないとしましょう. すると, 方程式を c で割ることで '$z+ax+by-c=0$' の形式の方程式が得られます. このことからローマ曲面の方程式の z に $c-ax-by$ を代入した式を Maxima で計算し, その式を描けば, それが求める断面になります.

ここで z を [-8, -2, -0.1, 0, 0.1, 2, 8] として描いた結果を次に示しておきます:



ここでの断面図は単純に Z 軸を法線とする平面による断面として描いたものですが, このローマ曲面の面白さが出ています.

15.10.3　頂点の計算

このローマ曲面の頂点は与式の左辺 $x^2y^2 + x^2z^2 + y^2z^2 - 17xyz$ の x と y による 1 階微分が 0 になる点です. この零点を Maxima で計算してみましょう:

```
(%i15) roman:(x^2+y^2)*z^2+x^2*y^2-17*x*y*z;
                                  2    2   2                2  2
(%o15)                          (y  + x ) z  - 17 x y z + x  y
(%i16) solve([diff(roman,x),diff(roman,y)],[x,y]);
                                                                  2
                sqrt(- 2 z - 17) sqrt(z)          sqrt(- 2 z  - 17 z)
(%o16) [[x = -------------------------, y = - --------------------],
                        sqrt(2)                          sqrt(2)
                                                        2
        sqrt(- 2 z - 17) sqrt(z)        sqrt(- 2 z  - 17 z)
[x = - -------------------------, y = --------------------],
                 sqrt(2)                         sqrt(2)
                                                          2
        sqrt(17 - 2 z) sqrt(z)          sqrt(17 z - 2 z )
[x = - ----------------------, y = - ------------------],
                sqrt(2)                         sqrt(2)
                                                        2
      sqrt(17 - 2 z) sqrt(z)        sqrt(17 z - 2 z )
[x = ----------------------, y = ------------------], [x = 0, y = 0]]
              sqrt(2)                       sqrt(2)
```

ここで z の値域は '$\frac{17}{2} \geq z \geq -\frac{17}{2}$' となることから '$[x,\ y,\ z] = [0,\ 0,\ \pm\frac{17}{2}]$' が求める頂点であることが判ります. なお, このギャラリーではやや強引に「生命の樹」風にするために上頂点 ($z = 17/2$) と下頂点 ($z = -17/2$) を一緒にしているので注意して下さい.
さて, 上下の頂点 ('$z = 17/2$ と $z = -17/2$') と $z = 0$' を除くとローマ曲面の断面は自己交叉点を一つ持つ 8 の字になっています. そして, 頂点から '$z = 0$' に近付くにつれて 8 の字の上下が潰れて交差点付近が XY 軸に貼り付き, それから '$z = 0$' を通り過ぎると, 再び 8 の字が出現しますが, 今度は潰れた 8 の字が 45 度回転した形で出現し, 最終的には一点に潰れます. この曲線で自分自身が交わる点で接ベクトルの次元は 2 になります.

15.10.4　surf による断面の描画

ここで surf の機能をフルに使って断面も同時に表示させてみましょう:
この図 15.13 では手前が Z 軸正方向になります. この描画では surf の切断面描画機能と曲面の半透明表示機能の両方を用いています. この絵を描くために surf の次の命令を用います:

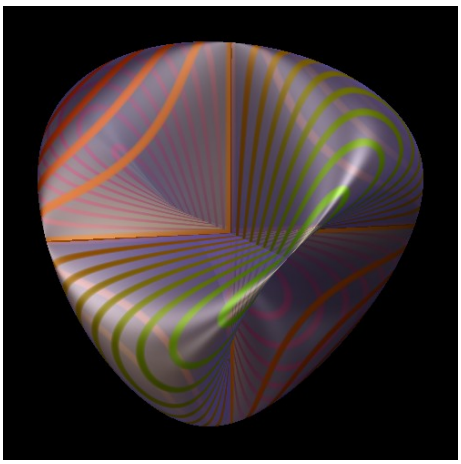


図 15.13: Steiner のローマ曲面と断面

切断面を描く為に必要な命令

変数	例	概要
plane	plane=z-1	平面の方程式を定義
curve_width	curve_width=10	切断面の曲線の幅を指定
curve_red	curve_red=10	切断面の曲線の色 (Red) を指定
curve_green	curve_green=10	切断面の曲線の色 (Green) を指定
curve_blue	curve_blue=10	切断面の曲線の色 (Blue) を指定
cut_with_plane	cut_with_plane	切断面を描画

それから surf のスクリプトファイル, ここでの例では surf.tmp に次の内容を追加すればよいのです:

```
color_file_format =jpg;
filename="uum.jpeg";
save_color_image;

rot_x = 0.14;
rot_y = -0.3;
int i=-8;
int j=0;
int k=100;
curve_width=5;
loop:
    k=k+1;
    plane=z-i;
    curve_red=255-j;
    curve_green=j;
    curve_blue=0;
    j=j+10;
    cut_with_plane;
    i=i+1;
if (i<9) goto loop;
```

```
21 save_color_image;
```

なお, ここでは UNIX 環境を前提とした例を挙げていますが, MS-Windows 環境であれば出力画像を ppm に修正するだけです. このスクリプトの追加はエディタでやっても構いませんし, バッチファイルを使っても構いません.
新しいスクリプトファイルができたら, UNIX 環境では `surf –no-gui surf.tmp`, MS-Windows 環境であれば DOS 窓から `surf surf.tmp` を実行すると画像ファイルが生成されます.

```
1  color_file_format =jpg;
2  filename="uum.jpeg";
3  save_color_image;
4  rot_x  = 0.14;
5  rot_y  = −0.3;
6  int  i=−8;
7  int  j=0;
8  int  k=100;
9  curve_width=5;
10 loop:
11     k=k+1;
12     draw_surface;
13     plane=z−i;
14     curve_red=255−j;
15     curve_green=j;
16     curve_blue=0;
17     j=j+10;
18     filename="Roman"+itostrn(3,k)+".jpeg";
19     cut_with_plane;
20     save_color_image;
21     i=i+1;
22 if (i<9) goto loop;
```

この例では複数の断面の画像ファイルができます. これらを使えばアニーメーションファイルも構成できます. たとえば, ImageMagick の convert を利用すると簡単に GIF アニメーションができます. たとえば, `convert -loop 10 Roman*jpeg Roman.gif` で GIF アニメに変換されるので, あとは FireFox 等の Web browser で Roman.gif ファイルを開くと楽しいでしょう.

15.11　SINGULAR を使ってみよう

可換群の計算では Maxima のような汎用の数式処理よりも専門のシステムの方が効率的に処理が行えることがあります. ここでは SINGULAR を用いた処理を簡単に紹介しておきましょう.
SINGULAR は可換代数, 代数幾何と特異点理論に特化した計算機代数システムです. SINGULAR は C 風の処理言語があり, この処理言語で記述したライブラリで機能を拡張できます. SINGULAR 自体には GUI やエディタ, さらにはグラフ表示機能を持っていませんが, その代りに Emacs, Surf や Surfex といった外部プログラムで代用ができます.

15.11.1 SINGULAR 初歩

SINGULAR の式の入力は Maxima と同様に式の末尾にセミコロン ";" を必ず付けます. セミコロンがなければ Maxima と同様に入力が継続していると SINGULAR が判断し, それから入力行の評価を行わずにプロンプトを "¿" から "." に切換えて入力が完遂されるまで待ちます.
SINGULAR の対象には全て型があります. 整数の和, 差, 積のみはそのまま入力しても結果が帰って来ますが, それ以外は最初に基礎環 (Base Ring) を定義しなければ何も計算できません.
ここで基礎環の定義では次の構文を用います:

基礎環の定義方法

ring ⟨ 環の名前 ⟩ = ⟨ 係数体, (⟨ 変数$_1$⟩, ··· , ⟨ 変数$_n$⟩), ⟨ 順序 ⟩;

基礎環の係数体は実数 $\mathbb{R}$, 複素数 $\mathbb{C}$, 有理数 $\mathbb{Q}$, 整数 $\mathbb{Z}$ 等が選べ, 標数 p は 2147483629 以下の整数が指定できます. さらに指定可能な項順序で代表的なものには辞書式順序 (lp), 斉次辞書式順序 (Dp), 斉次逆辞書式順序 (dp), 重み付き逆辞書式順序 (wp), 重み付き辞書式順序 (Wp), 負辞書式順序 (ls), 負斉次逆辞書順序 (ds), 負斉次辞書順序 (Ds), 負重み付き逆辞書式順序 (ws) に負重み付き辞書式順序 (Ws) 等が選べます. 項順序は標準基底 (Gröbner 基底) の計算で非常に重要であり, この順序の選択一つで処理速度が大きく異なるので, 多様な順序が選択できることは非常に重要です.
基礎環を定義すれば, その基礎環上の多項式 (poly) やイデアル (ideal) , 更に商環 (qring) が扱えます. 次に多項式, イデアルと商環の定義方法を示しましょう:

多項式, イデアルと商環の定義方法

poly ⟨ 多項式名 ⟩ = ⟨ 多項式 ⟩;
ideal ⟨ イデアル名 ⟩ = ⟨ 多項式$_1$⟩, ··· , ⟨ 多項式$_n$⟩;
ideal ⟨ 商環名 ⟩ = ⟨ イデアル ⟩;

商環 R/I の定義で指定するイデアル I は単純に ideal で宣言したものも使えますが, その場合はイデアルが標準基底はないいと警告します. イデアルの標準基底は std 命令で `std(⟨ イデアル ⟩);` から得られるので, この結果を用いましょう.
なお, 標準基底を計算する命令に groebner もありますが, std 命令と違って groebner 命令は環の順序に制約があります.
SINGULAR は入力された大文字と小文字を判別します. 一度定義した対象は名前を SINGULAR に入力すればその内容が表示されます. この点は Maxima と同じです.
では各対象の定義例を次に示しておきましょう:

```
> ring r=0,(x,y,z),dp;
> poly unit_circle_eq=x^2+y^2-1;
> ideal unit_circle=x^2+y^2-1;
> unit_circle_eq;
x2+y2-1
> unit_circle;
unit_circle[1]=x2+y2-1
> ideal points=unit_circle_eq,y-x;
> points;
points[1]=x2+y2-1
points[2]=-x+y
```

```
> points[1];
x2+y2-1
```

この例では基礎環として $r = \mathbb{Q}[x,y,z]$ を定義し, それから多項式の unit_circle_eq とイデアルの points を定義しています.
このように SINGULAR は Maxima と違い, 値の割当で演算子 "=" を使います. 対象の内容は単純に対象名を入力すると, 多項式 unit_circle_eq の場合のように返されます. ここでイデアルは番号付で返されるので, イデアルを生成する多項式を取り出したいときには, この例の points[1] のようにすれば一番目の生成元が取り出せます. なお, 取り出した生成元の型は多項式 (poly) 型になります.
今度は商環の定義の例を示しましょう:

```
> qring ri=std(x^2+y^2-1);
> ri;
//   characteristic : 0
//   number of vars : 3
//        block   1 : ordering dp
//                  : names    x y z
//        block   2 : ordering C
// quotient ring from ideal
_[1]=x2+y2-1
> r;
//   characteristic : 0
//   number of vars : 3
//        block   1 : ordering dp
//                  : names    x y z
//        block   2 : ordering C
```

この例では前の基礎環 $\mathbb{Q}[x,y,z]$ 上で, 商環の定義で必要なイデアルを std(x^2+y^2-1); で与え, 商環 ri を $\mathbb{Q}[x,y,z]/\langle x^2+y^2-1\rangle$ で定義しています.
SINGULAR では環を複数定義することができますが, 定義する毎にポインタが新規に定義された環に移動します. 多項式, イデアル, 写像等はポインタが置かれた環上で定義されるので, 必要に応じてポインタを戻す必要があります. ポインタを別の環に移す場合, setring 命令を使います. その使い方は *setring* ⟨ 環の名前 ⟩; のように環の名前を直接指定するだけです.
SINGULAR では写像が扱えます. この写像の定義では名前が他の数式処理のものと紛らわしい名前ですが, map 函数を用います. この場合, 対象が含まれる環を基礎環にした状態で写像の定義を行います. 次に写像の定義例を示しましょう:

```
> ring r1=0,(x,y,z),dp;
> ring r2=0,(a,b),dp;
> map f=r1,a,b,0;
> f;
f[1]=a
f[2]=b
f[3]=0
```

この例では環 r1 と環 r2 をそれぞれ $\mathbb{Q}[x,y,z]$,$\mathbb{Z}[a,b]$ とし, 環 r1 から環 r2 への写像 $f(x,y,z) \to (a,b,0)$ を写像の値域となる基礎環の変数に対して一つ一つ定めています.
SINGULAR の写像の定義方法は基礎環の変数がどのように写されるかを定めるだけで行えます.

面白いことに, 定義した写像によるイデアルの逆像が計算可能な事でしょう. たとえば, 上の例の環 r2 のイデアル i2 の写像 f による逆像は preimage 命令 を使って `preimage(r1,f,i2);` で計算できます. 次の例ではイデアル i2 を零イデアル (0) とし, 写像 f の核 $\ker f$ を計算します:

```
> ring r1=0,(x,y,z),dp;
> ring r2=0,(a,b),dp;
> map f=r1,a,b,0;
> ideal i2=0;
> setring r1;
> preimage(r2,f,i2);
_[1]=z
```

なお, この例では零イデアルを ideal i2=0; で定義していますが, 単純に ideal i2 としても同じ意味になります.

15.11.2 SINGULAR で surf を使ってみよう

SINGULAR はその処理言語を用いてライブラリが構築できます. このライブラリの読込は LIB 命令 (大文字) を用います. SINGULAR には様々なライブラリが附属していますが, 標準で附属するライブラリで, surfplot.mc のように SINGULAR で定義した函数を surf を用いて可視化する surf.lib があります. 実は, surfplot.mc は surf.lib の内容も参考にしています.
この surf.lib を用いたローマ曲面の描き方の例を以下に示しておきます. ただし, Maxima と同じことをしても面白くはないので, ここでは写像の定義, その写像の逆像を用いて表示する例を示しておきましょう:

```
> ring r=0,(x,y,z),dp;
> poly sp4=x2+y2+z2-16;
> ideal i1=sp4;
> map f=r,xy,yz,zx;
> ideal steiner=preimage(r,f,i1);
> steiner;
steiner[1]=x2y2+x2z2+y2z2-16xyz
> plot(steiner,"background_red=0;background_green=0;
. background_blue=0;rot_x=2;rot_y=0.5;");
```

この例では写像 $f : (x, y, z) \to (xy, yz, zx)$ による半径 4 の三次元球面の逆像を preimage 命令で求め, plot 命令に surf の背景色を設定する命令と一緒に引渡しています. ここで, この背景色の設定が長いために途中で改行を入れていますが, SINGULAR は行末のセミコロン ";" が未入力のために行が継続中であると判断し, ピリオド "." を出しています. この例で表示される曲面は向きは多少異なりますが図 15.12 と同じ曲面が得られます. このように Steiner のローマ曲面は球面の写像 f による逆像としても得られます.
では, 今度は SINGULAR の方から曲線や曲面の新しい描き方を提案しましょう. 曲線や曲面が助変数表示されている場合は surf で絵を描けません. この場合は Maxima の描画函数の方が上手く描くかもしれません.
ところが, SINGULAR の eliminate 命令を併用すると描ける場合があります. ここで SINGULAR の eliminate 命令は不要なイデアルの変数を削除する命令です. 以下に eliminate 命令を使って図 15.14 に示す Decartes の葉状曲線を描く例を示しましょう:

```
> ring r1=0,(x,y,t),dp;
> ring r2=0,(x,y),dp;
> map f=r1,x,y,0;
> setring r1;
> ideal i1=(1+t^3)*x-3*t,(1+t^3)*y-3*t^2;
> ideal i2=eliminate(i1,t);
> poly c2=i2[1];
> c2;
x3+y3-3xy
> setring r2;
> plot(f(c2));
```

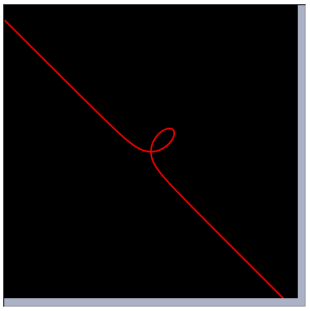


図 15.14: Decartes の葉状曲線

この例では媒介変数を含めた環 r1 とそれを除いた環 r2, 環 r1 から環 r2 への写像 f, それから, 環 r1 上でイデアル i1 を定義します. この葉状曲線は媒介変数 t を用いて $x = 3t/(1+t^3), y = 3t^2/(1+t^3)$ で表現されていますが, 基礎環 r1 が $\mathbb{Q}[x, y, t]$ のためにイデアル i1 の定義で分子をかけた形にしています. 次の eliminate 命令では媒介変数 t を削ったイデアル i1 の表現を求めています. この eliminate 命令でイデアルが返却され, それをイデアル i2 と定義しています. 次にイデアルは生成元を c1 に示すように取り出せます. それから基礎環を setring 命令で環 r2 にし, 写像 f による c1 の像を plot 命令で描いた結果が図 15.14 になります. ここで環 r1 ではなく環 r2 上で曲線を描いた理由は, 単純に surf.lib で環の変数の総数で曲線と曲面を判別するために環 r1 では変数 [x, y, t] と 3 変数になるので曲面が描かれるからです.

15.12　Maxima で終結式を使ってみよう

ここで話を SINGULAR から Maxima に戻します. 先程の例では媒介変数表示された Decartes の葉状曲線を SINGULAR の eliminate 命令を用いて x, y の多項式に変換しました. これと同じことを Maxima でするにはどうすればよいでしょうか？

Maxima で同じことを行うために eliminate 函数を用いれば良いのですが, これでは幾ら何でも呆気ないので, eliminate 函数で利用されている多項式の終結式を使ってみましょう. Maxima では終結式の計算に resultant 函数が使えます. また, bezout 函数を使って行列を求め, そ の行列の determinant を計算する方法もあります. ここではいろいろ試してみましょう.

終結式を求める resultant 函数

resultant(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数 ⟩)

resultant 函数は引数として二つの多項式と一つの変数を必要とします. ここで指定する変数は二つの多項式から消去したい変数になります. この終結式は二つの多項式を指定した変数の多項式として並び換えを行います.

Maxima では resultant 函数は bezout 函数と determinant 函数の合成で表現できます. ここでは Decartes の葉状曲線を用いて順番に作業を追ってみましょう:

```
(%i1) p1:(1+t^3)*x-3*t;
                                  3
(%o1)                           (t  + 1) x - 3 t
(%i2) p2:(1+t^3)*y-3*t^2;
                                  3              2
(%o2)                           (t  + 1) y - 3 t
(%i3) expand(p1);
                                  3
(%o3)                            t  x + x - 3 t
(%i4) expand(p2);
                                  3             2
(%o4)                            t  y + y - 3 t
(%i5) m1:matrix([x,0,-3,x,0,0],
                [0,x,0,-3,x,0],
                [0,0,x,0,-3,x],
                [y,-3,0,y,0,0],
                [0,y,-3,0,y,0],
                [0,0,y,-3,0,y]);

                        [ x   0   - 3   x    0   0 ]
                        [                          ]
                        [ 0   x    0   - 3   x   0 ]
                        [                          ]
                        [ 0   0    x    0   - 3  x ]
(%o5)                   [                          ]
                        [ y  - 3   0    y    0   0 ]
                        [                          ]
                        [ 0   y   - 3   0    y   0 ]
                        [                          ]
                        [ 0   0    y   - 3   0   y ]
(%i6) expand(determinant(m1));
                                  3                 3
(%o6)                        - 27 y  + 81 x y - 27 x
```

この例では最初に二つの多項式多項式 $p1 = (1+t^3)x-3t$ と $p2 = (1+t^3)y-3t^2$ を定義しています. ここで, 多項式 $p1$ と多項式 $p2$ を展開すると, それぞれが $p1 = xt^3+0t^2-3t+x$ と $p1 = yt^3-3t^2+0t+y$ とになります. 多項式 $p1$ と $p2$ の終結式は, この t の多項式と看做した場合の係数を並べたものになります. 両方の多項式の次数が 3 の為, 構築する行列 $m1$ は次の 6 次の正方行列になります:

$$m1 = \begin{bmatrix} x & 0 & -3 & x & 0 & 0 \\ 0 & x & 0 & -3 & x & 0 \\ 0 & 0 & x & 0 & -3 & x \\ y & -3 & 0 & y & 0 & 0 \\ 0 & y & -3 & 0 & y & 0 \\ 0 & 0 & y & -3 & 0 & y \end{bmatrix}$$

この行列の行列式を計算したものが終結式になります.
この終結式には面白い性質があります. それは終結式と二つの多項式の解の関係を示すものです.
変数 x を主変数とする多項式 f と g が次で表現されていたとします:

$$f = \sum_{i=0}^{m} a_i x^i$$

$$g = \sum_{i=0}^{n} b_i x^i$$

これらの多項式の終結式は次で表現されます:

$$\mathrm{resultant}(f, g, x) = \det \begin{pmatrix} a_m & a_{m-1} & \cdots & \cdots & a_1 & a_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & a_m & a_{m-1} & \cdots & \cdots & a_1 & a_0 \\ b_n & b_{n-1} & \cdots & \cdots & b_1 & b_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & b_n & b_{n-1} & \cdots & \cdots & b_1 & b_0 \end{pmatrix}$$

ここで, 多項式 f と g の解を α_i, β_j とすると, 多項式 f と g の終結式は解 α_i,β_j を用いて次の式に等しくなります:

$$\mathrm{resultant}(f, g, x) = a_m^n b_n^m \prod_{1 \le i \le m, 1 \le j \le n} (\alpha_i - \beta_j)$$

このことは多項式 f と g が共通の解を持ては終結式が常に 0 になり, 同時に終結式が 0 になるのは多項式 f と g が共通の解を持つときに限ることを意味します. ここでは零点集合を表現する多項式を計算することが目的なので終結式の零点集合と多項式 $p1$ と $p2$ が同時に 0 になる零点集合は一致しなければなりません. この性質から終結式を使って助変数を排除しているのです.
なお, Maxima ではこの行列を効率良い行列で書き出す bezout 函数があります. この函数で上記の多項式 $p1$ と $p2$ の係数から構成される行列と determinant 函数による結果を示しておきます:

```
(%i7) bezout(p1,p2,t);
                                [   0     3 x   - 3 y ]
                                [                     ]
(%o7)                           [  3 y    - 9    3 x  ]
                                [                     ]
                                [ - 3 x   3 y     0   ]
```

```
(%i8) expand(determinant(%));
                                  3                  3
(%o8)                        - 27 y  + 81 x y - 27 x
```

この計算を resultant 命令で一度に実行したものを以下に示します:

```
(%i5) p1:(1+t^3)*x-3*t;
                                  3
(%o5)                          (t  + 1) x - 3 t
(%i6) p2:(1+t^3)*y-3*t^2;
                                  3              2
(%o6)                          (t  + 1) y - 3 t
(%i7) i1:resultant(p1,p2,t);
                                    3             3
(%o7)                        - 27 (y  - 3 x y + x )
```

これで SINGULAR と定数項を除いて等しい多項式が得られました. ここで定数項は無視しても構いません. 実際, a を定数, $f(x_1,\cdots,x_n)$ を多項式とするときに $f(x_1,\cdots,x_n)$ の零点集合と $af(x_1,\cdots,x_n)$ の零点集合が一致するからです.

ちょっと脇道に逸れますが, 多項式 f の零点集合 $V(f)$ と多項式 $g(x_1,\cdots,x_n)$ と多項式 $f(x_1,\cdots,x_n)$ の積の零点集合 $V(fg)$ の関係を思い出して下さい. $V(fg)$ は多項式 f と g の零点の両方を含み, $V(f)\cup V(g)$ になります. さらに f で割切れる多項式 h の零点集合 $V(h)$ に対しては $V(h)\supset V(f)$ が成立します.

ここで, 多項式 f で生成されるイデアル (f) を考えると, $V(f)$ がイデアル (f) に含まれる多項式の零点集合の中で最小の集合になります. イデアルで考えてしまえば多項式が定数倍であることは大きな問題になりません. ですから, 多項式の零点集合を考える場合はそのイデアルを考えることが妥当な手段になります.

次に, 応用で猿の腰掛と呼ばれる曲面を表示してみましょう. この曲面は助変数表示では以下の関係式を満すものです:

猿の腰掛の助変数表示

$$\begin{cases} x-u=0 \\ y-v=0 \\ z-u^3+3uv^2=0 \end{cases}$$

後の式の変数 u , v を x と y で置換えてしまえば済む話ですが, ここでは終結式を使って猿の腰掛の変数 x , y, z の式に変換してみましょう. ここで注意することは, resultant(x-u,y-v,u) のように片方の式にしか存在しない変数を使ってはいけません. 無意味な式が返却されるだけです. この例では変数 u と v が含まれているのは z-u^3+3 u v^2=0 だけなので, resultant 函数では, この多項式を中心に x-u から開始し, y-v で終える手順になります:

```
(%i1) p1:x-u;
(%o1)                                x - u
(%i2) p2:y-v;
(%o2)                                y - v
(%i3) p3:z-u^3+3*u*v^2;
                                        2     3
(%o3)                          z + 3 u v  - u
```

```
(%i4) a1:resultant(p1,p3,u);
                                    3       2
(%o4)                         - z + x  - 3 v  x
(%i5) a2:resultant(a1,p2,v);
                                        2     3
(%o5)                         - z - 3 x y  + x
(%i6) surfplot(a2);
```

これで多項式として $-z-3xy^2+x^3$ が得られました. このグラフを surf で描いたものが図 15.15 で, 見たとおり, 尻尾の生えた猿にちょうど良い腰掛みたいな曲面が描かれます:

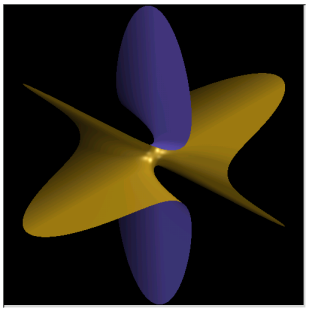


図 15.15: 猿の腰掛

なお, Maxima には eliminate 函数が存在するので, 実際は
`eliminate([x-u,y-v,z-u^3+3*u*v^2],[u,v]);` で十分です.
試しに動作を確認してみましょう:

```
(%i13) eliminate([x-u,y-v,z-u^3+3*u*v^2],[u,v]);
                                        2     3
(%o13)                        [- z - 3 x y  + x ]
(%i14) %[1];
                                        2     3
(%o14)                         - z - 3 x y  + x
```

長々とした計算を行っていましたが, 本当は eliminate 函数を使えば簡単にできることでした. ご苦労さまでした.

15.13　デモファイルでも書いてみよう

surfplot.mc で色々遊んでみたので今度はデモ用のファイルを作ってみましょう. これはバッチファイルでも良いのですが Maxima には函数やパッケージのデモ用に demo 函数があります. このデモ用のファイルの内容は batch ファイルと基本的に同じもので, 要するに,Maxima が実行する内容を並べておけば良いのです.
ただし, demo 函数は Maxima の一行を実行するとプロンプト "_" を出して一旦停止します. ここで, セミコロン ";" や Enter キーを押せば次の行の処理に移ります. そのために入力行に対する処理がどのように実行されるかが判り易くなっています.

このデモファイルの修飾子は dm, dm1, dm2, dm3, dmt と dem が用いられます. さらにデモファイルが大域変数 file_search_path に記述されたディレクトリ上にある場合, ファイル名の修飾子を外した名称だけを引数にできます. すなわち, デモファイル名が surfplot.dem の場合に `demo(surfplot);` と入力するだけでデモが実行されることを意味します.
ただし, 大域変数 file_search_path に登録されていないディレクトリ上にデモファイルがある場合, `demo("Desktop/surfplot.dem");` のようにカレントディレクトリ上にないファイルに対してはパスも含めて二重引用符でファイル名を括る必要があります.
大域変数 file_search_path に強引に自分のデモファイルを登録したディレクトリを登録することもできますが, Maxima の版によって文字列の扱いが異なるために注意が少し必要です. つまり, Maxima-5.13.0 以前では Maxima の文字列と LISP の文字列は別のデータ型です. したがって, 大域変数 file_search_path には Maxima の文字列を LISP の文字列に変換して追加する必要があります. そのために to_lisp 函数を使って LISP に移動し, そこで大域変数$file_search_path に強引にパスを追加する方法もあります. ところが, Maxima-5.14.0 以降から Maxima の文字列と LISP の文字列の型が共通になるのでこの変換は不要になり, Maxima 上で追加処理が済ませられます.
ここで追加する LISP の文字列は "⟨*PATH*⟩/###.{dem,dmt}" の形になります. ちなみに ⟨*PATH*⟩ はディレクトリで {dem,dmt} がファイルの修飾子を指定しています.
さて, dem ファイルの内容は Decartes の葉状曲線, レムニスケート, お遊びのアヒルを入れておきましょう. それに加えて Steiner のローマン曲面の断面を描くものや Bath Diec も入れてみましょう. ここでローマン曲面の断面を描くために細工を行います. それは surf に曲線の透明度を最初に指定し, それからローマン曲面に交差する平面とその平面との交わりの色を指定すること, そして, surf でそれらを描く surf のスクリプトファイルを生成することです. つまり, 最初に標準的なローマン曲面を表示させます. この作業は surfplot を使って行いますが, それによって surf.tmp ファイルが描画の副産物として生成されます. そこで, 透明度や平面と断面の指定をこの surf.tmp に書込むのです. ここでは簡単に済ませるために system 函数を使って UNIX の mv 命令と shell の echo 命令のみを用いて実現させています:

```
1  /* decartes curve */
2  delta:-1/4;
3  poly:x^3+y^3-3*x*y -delta;
4  surfplot(poly);
5
6  /* lemniscate(1)*/
7  poly:(x^2*(9-x^2)-4*y^2);
8  surfplot(poly);
9  /* lemniscate(1)*/
10 poly:(x^2*(9-x^2)-4*y^2)*x*y;
11 surfplot(poly);
12
13 /* Ducky */
14 poly:(x^2+y^2-9)*((x-2)^2+(y-1)^2-1)*
15 ((x+2)^2+(y-1)^2-1)*(x^2/9+2*(y+1)^2-1);
16 surfplot(poly);
17
18 put(surf ,0.6, rot_x );
19 put(surf ,0.6, rot_y );
20 put(surf ,0.6, rot_x );
```

```
21 put(surf ,0.6, rot_z );
22 put(surf ,0.06, scale_x );
23 put(surf ,0.06, scale_y );
24 put(surf ,0.06, scale_z );
25 put(surf ,50,   transparence);
26 poly:x^2*y^2+x^2*z^2+y^2*z^2-x*y*z;
27 surfplot(poly);
28 /* surf.tmp の内容を surf.tmpx の末尾に追加します */
29 system("cat surf.tmp>>surf.tmpx");
30 /* surf.tmpx を surf.tmp に名前を変えます */
31 system("mv surf.tmpx surf.tmp");
32 /* 平面x+y+x-0.005=0による断面を指定 */
33 system("echo \"plane=x+y+z-0.05;\">>surf.tmp");
34 /* 以降,断面の色を指定 */
35 system("echo \"curve_red=255;\">>surf.tmp");
36 system("echo \"curve_green=0;\">>surf.tmp");
37 system("echo \"curve_blue=0;\">>surf.tmp");
38 /* 断面を描く事を指定 */
39 system("echo \"cut_with_plane;\">>surf.tmp");
40 /* 平面x+y+z+0.05=0による断面を指定 */
41 system("echo \"plane=x+y+z+0.05;\">>surf.tmp");
42 /* 以降,断面の色を指定 */
43 system("echo \"curve_red=0;\">>surf.tmp");
44 system("echo \"curve_green=255;\">>surf.tmp");
45 system("echo \"curve_blue=0;\">>surf.tmp");
46 /* 断面を描く事を指定 */
47 system("echo \"cut_with_plane;\">>surf.tmp");
48 /* surf に surf.tmp 内容を読み込んで実行 */
49 system("surf -x surf.tmp>surf.log&");
50
51 /* Barth Diec */
52 /* set rot_x,y,z and scale_x,y,z */
53 put(surf ,0.6, rot_x );
54 put(surf ,0.2, rot_y );
55 put(surf ,0.6, rot_z );
56 put(surf ,0.3, scale_x );
57 put(surf ,0.3, scale_y );
58 put(surf ,0.3, scale_z );
59 /* set tau */
60 tau:(1+sqrt (5))/2;
61 /* polynomial */
62 poly:8*(x^2-tau^4*y^2)*(y^2-tau^4*z^2)*(z^2-tau^4*x^2)*
63 (x^4+y^4+z^4-2*(x^2*y^2+y^2*z^2+z^2*x^2))+
64 (3+5*tau)*(x^2+y^2+z^2-1)^2*(x^2+y^2+z^2-(2-tau))^2;
65 surfplot(poly);
66
67 /* Barth Diec */
68 /* set rot_x,y,z and scale_x,y,z */
69 put(surf ,0.6, rot_x );
70 put(surf ,0.2, rot_y );
71 put(surf ,0.6, rot_z );
72 put(surf ,0.3, scale_x );
73 put(surf ,0.3, scale_y );
74 put(surf ,0.3, scale_z );
```

```
/* set tau */
tau: float((1+sqrt (5))/2);
/* polynomial */
poly:8*(x^2-tau^4*y^2)*(y^2-tau^4*z^2)*(z^2-tau^4*x^2)*
(x^4+y^4+z^4-2*(x^2*y^2+y^2*z^2+z^2*x^2))+
(3+5*tau)*(x^2+y^2+z^2-1)^2*(x^2+y^2+z^2-(2-tau))^2;
surfplot(poly);
```

MS-Windows 環境で利用する場合は, DOS 窓の echo の性質に合せて修正を加え, surf で画像の生成, 画像の変換, 画像の表示を追加すれば良いでしょう. これでデモファイルができました. では早速, 実行してみましょう.
ここではカレントディレクトリ上に surfplot.mc と surfplot.dem が置かれているものとします. そして, 注意になりますが, このデモファイルには surfplot.mc ファイルの読み込みを記述していないので, デモを実行する前に load("surfplot.mc"); で surfplot.mc の読込を行い, それから, demo("surfplot.dem"); を実行しましょう.
実行の様子を頭の数行だけ示しておきます:

```
(%i2) demo("surfplot.dem");

batching /home/yokota/Desktop/surfplot.dem

 At the _ prompt, type ';' followed by enter to get next demo
                                                  - 1
(%i3)                                    delta : ---
                                                   4
_
                                                        3    3
(%i4)                        poly : - delta - 3 x y + y  + x
```

Enter キーを押して行くとグラフが出たりすれば成功です.

15.14 例題ファイルもついでに…

さて, demo 函数用のデモファイルを作ったので今度は example 函数向けの例題ファイルを作ってみましょう.
例題ファイルの在処は, LISP の変数*maxima-demodir*に束縛された値を演算子 ":lisp" で調べると判ります. また, ファイル名は Maxima の大域変数 manual_]demo に割り当てられています.

```
(%i54) :lisp *maxima-demodir*
/usr/local/share/maxima/5.13.0/demo
(%i54) manual_demo;
(%o54)                                    manual.demo
```

これらのことから, この Maxima の例題ファイルは, /usr/local/share/maxima/5.17.0/demo にある manual.demo ファイルである事が判ります.
ここで件の manual.demo ファイルの内容を覗いてみましょう:

```
/* This file is to be run by the EXAMPLE command, and may not
   otherwise work.
```

```
3     The following are either acceptable lines to maxima, or they are
4     two successive '&' characters immediately following a '$' or ';'
5     and then followed by the name of the section of examples, and then followed by
6     a sequence of maxima forms.
7  */
8  &&
9  additive declare(f,additive);
10 f(2*a+3*b);&&
11 algsys  f1:2*x*(1-l1)-2*(x-1)*l2$
12 f2:l2-l1$
13 f3:l1*(1-x**2-y)$
14 f4:l2*(y-(x-1)**2)$
15 algsys([f1,f2,f3,f4],[x,y,l1,l2]);
16 f1:x**2-y**2$
17 f2:x**2-x+2*y**2-y-1$
18 algsys([f1,f2],[x,y]);&&
19
20    … 中略 …
21
22 zeroequiv zeroequiv(sin(2*x)-2*sin(x)*cos(x),x);
23 zeroequiv(%e^x+x,x);
24 zeroequiv(log(a*b)-log(a)-log(b),a);
```

ここで, 例題ファイルの注釈は “/* */” で括ることが判りますね. そして書式は Maxima のバッチファイルに似ていますが, 末尾に “&&” がある行, additive declare(f,additive); のようにラベルと Maxima の文が記述されている箇所がありますね.
例題ファイルの書式はラベルを置いた場合に空白を空けてから Maxima の入力文を記述し, 一通り終了すると末尾に “&&” を置く書式になっているのです. ここでラベルは自由です. だからといって出鱈目なもの, たとえば, surfplot のような函数名ではなく「三毛猫大王」のようなラベルにすると内容が判らず, あとで苦労する羽目になるでしょう.
さて, example 函数は与えられた引数に合致するラベルを検出し, そこから “&&” が検出される迄の行を処理する仕組になっています.
したがって次のような改造が可能です:

```
    …  略 …

zeroequiv(%e^x+x,x);
zeroequiv(log(a*b)-log(a)-log(b),a);&&
/* 三毛猫大王
これでどうかな？ */&&
surfplot print("Decartes!!");
surfplot(x^3+y^3-3*x*y+1/4);
```

このように改造した例題ファイルを所定の位置に置くか, 内部変数*maxima-demodir*の値の変更を予め行ない, 肝心の surfplot.mc を読込んでいれば何時でも `example(surplot);` を実行すると例題が処理されます:

```
(%i56) example(surfplot);
(%i57) print("Decartes!!")
```

```
Decartes!!
(%i58) surfplot(1/4-3*x*y+y^3+x^3)

'rat' replaced 9.999999999999999E-11 by 1/10000000000 = 9.999999999999999E-11
Surf is now drawing  (4*y^3-12*x*y+4*x^3+1)/4 . Please wait ....
(%o58) 0
(%i59) false
(%o59) done
(%i60)
```

第16章　MATLAB風言語で遊ぶ話

16.1　数値計算を主体にした環境

この本はMaximaの解説書ですが, ここで何故, MATLABクローンの話が乱入するのか, その理由を説明しておきましょう. まず, 数式処理システムの多くが任意精度による数値計算が可能であり, 行列も比較的扱い易いという長所を持っています. しかし, 残念なことに処理速度自体は速いものではなく, 行列の処理では遅さが目立ちます. さらに, MaximaはLISPで 記述されているため, Maxima言語でプログラムを記述していれば, 一度, LISPに変換されてから処理を行うために一層不利になります. この辺を回避するためにはLISPで直接処理プログラムを書いたり, 数値行列計算ライブラリのLAPACKをLISP側から利用する方法があり, これらの手法で大きな改善が見込めますが, 使い込むためには相応の知識が必要になります.
ここで頭を切り換え, 何れにしても数値計算ライブラリを利用するのであれば, 最初から数値行列を扱うことを目的としたMATLAB風の言語を使うというのは如何でしょうか？ そもそも, MATLAB風の言語自体, 数値行列の処理に適した様々な工夫がある訳で, 単純な処理速度の向上に留まらない, 作業自体の効率化も一緒に望めるのです.

16.2　MATLABとMATLAB風言語

MATLABは, 大学の教育でLINPACK等の数値計算ライブラリを容易に学生が使えるようするために開発されたアプリケーションです. 現在はThe Mathworks,Inc. が開発と販売を行っており, Toolboxと呼ばれる膨大なライブラリ群を従え, 数値行列処理を目的としたソフトウェアの標準的存在になっています[1].
このMATLABは数値配列, 特に1次元配列 (ベクトル) や1次元配列 (行列) の処理を目的とした各種のアプリケーションに大きな影響を与えており, MATLAB言語を一通り理解していれば, 他の言語の理解が容易となります.
ここでMATLABに類似したソフトウェアには, Euler Math Toolbox, Freemat, Octave と Scilab, おまけに Yorick があります.

Euler Math Toolbox　MATLABと同時期に開発されたアプリケーションで, Maximaと連動が可能なことを大きな特徴とします. 言語的にMATLABと類似していますが, 独自のシステムです. なお, Euler Math ToolboxはMS-Windows上で開発されています. このMS-Windows版はUNIX環境上でもWINEを使えば動かすことができます. また, GCLやCLISPで動作するMaximaであ

[1] 構成は http://www.mathworks.co.jp/products/pfo を参照.

れば, Euler と連動させることも可能です. Euler Math Toolbox の古い版は UNIX 環境にも移植されていますが, こちらには Maxima との連携機能はありません.

FreeMat　MATLAB のクローンであるよりも MATLAB を越えることを目指したアプリケーションです. MATLAB とは 95%程の互換性を持ち, GUI 環境は最も MATLAB に類似しています.

Octave　様々な環境に移植され, GNU の MATLAB クローンと呼んでも差し支えない程, MATLAB との高い互換性を持ちます. また, GUI 環境も qtOatave を用いれば MATLAB と似た GUI 環境で作業が行えます. この Octave は単なる MATLAB クローンに留まるシステムではありません. 実際, ファイルの扱いでは MATLAB よりも柔軟な処理が行える長所もあります. ただし, グラフ処理では外部アプリケーションとして gnuplot を利用するため, MATLAB との互換性が下る傾向があり, 機能的にも MATLAB と比べてやや劣ります.

Scilab　フランスの INRIA で開発された, MATLAB に類似した独自のシステムです. 実際, 言語形態は MATLAB に類似していますが, 微妙な点で異なり, Octave のような高度の類似はありません. しかし, この Scilab は OSS のものの中では最も機能やライブラリが充実しており, MATLAB と比べて見劣りしない程です.

Yorick　MATLAB とは全く違う, 対話処理の可能な C といえる言語仕様で, ここで解説したどのアプリケーションよりも軽量です. この Yorick の特徴は, 軽量で高速な処理, 高度な描画やアニメーションが可能な点で, 添字を使った独特の数値配列の処理によって, 多次元配列の処理が容易に行える長所を持っています.
この Yorick の詳細は KNOPPIX/Math に付属の「たのしい Yorick」[65][2] や「数値計算&可視化ソフト・Yorick」[66] を参照して下さい.
この本では MATLAB の影響を特に強く受けた, FreeMat, Octave, Scilab のことを一括りで「**MATLAB 風の言語**」と呼ぶことにします.

16.3　オンラインヘルプ

基本的に MATLAB 系の言語で, 函数等のオンラインヘルプは help 命令を用います. たとえば, Octave のオンラインヘルプは単純に help〈命令〉と入力すれば, 調べたい命令のオンライヘルプが表示されます. この help 命令で表示されるヘルプの内容は, Octave や MATLAB では **M-file**, すなわち, ファイルの修飾子が ".m" の命令が記述されたファイルの先頭の注釈部分を表示しています. この M-file は MATLAB 系の言語で記述された命令と一対一に対応するファイルで, ファイルの先頭の注釈の部分を命令の解説とする書式が定まっています. ちなみに, Yorick の場合は函数毎に注釈を入れる方式, Scilab は別途 XML で記述した MAN ファイルと呼ばれるファイルを別途用意する方式です. このファイルは函数の解説だけではなく, 一般的なオンラインマニュアルが記述できます.

[2] KNOPPIX/Math2010 に付属. /usr/share/doc/knoppix-math-doc/ja/YorickBook.pdf を参照.

MATLAB と Octave でオンラインマニュアルを表示する命令として doc 命令があります, ここで MATLAB の doc 命令は JAVA を使ったマニュアルを表示する命令ですが, Octave の doc 命令は info ファイルのマニュアルを表示する仕様となっています.

16.4 基本的な対象

MATLAB 系の言語では, 数値や文字列が扱えます. そして, 1 次元配列に対応するベクトル, 2 次元配列に対応する行列もあります. それから, これらの対象から構成される構造体もあります. さらに Yorick の場合は多次元配列と LISP 風のリストを持ちます. ここで対象の型を調べる命令として, MATLAB と Octave には class 命令, Scilab と Yorick には typeof 命令があります.

16.4.1 数値

数値には整数, 浮動小数点数とこれらの数値と純虚数で構成された複素数があります. MATLAB, Octave と Yorick では純虚数を '1i' と表記し, Scilab では '%i' と表記します. 数値の型の判別は, Yorick では typeof で行えますが, MATLAB と Octave の class 命令では, 対象が複素数か実数かは判断できません:

```
octave:6> class(1*2+3)
ans = double
octave:3> (1+1i)^2
ans =  0 + 2i
octave:7> class((1+1i)^2)
ans = double
```

これは Scilab でも同様で, type 命令や typeof 命令では「実数や複素数の定数」と一括りで分類さています. 実際, MATLAB 系の言語では, 数値の型を明示的に指定しない限り, 数値計算は倍精度浮動小数点数として処理されるために, 配列の添字として小数点数以下が 0 の実数の浮動小数点数が使えます. さらに MATLAB と Octave では虚部が '0' の 複素数さえも添字として使えます:

```
octave:7> a=[1:10];
octave:8> a(2.0000)
ans =  2
octave:9> a(2+0i)
ans =  2
```

Scilab では虚部が '0' でも複素数添字はエラ-になります. また, Yorick では配列の添字は整数に限定されます.
MATLAB 系の言語では数値計算は倍精度浮動小数点数で処理されるために, 整数を表現する int32 等の型を利用する場合には, 型変換の函数を用いる必要があります. ここで整数の型は整数を表現する bit 長で区分されるために, 型の範囲外の数値は表現できません. ここで MATLAB 系の言語では, その型の整数の下限や上限を返します:

```
octave:1> int32(2^31)
ans = 2147483647
octave:2> int32(-2^31)
```

```
ans = -2147483648
octave:3> int32(2^31+1)
ans = 2147483647
octave:4> int32(2^31)*2
ans = 2147483647
```

一方で, Yorick の場合は「1 の補数」となるので注意が必要です.
たとえば, KNOPPIX/Math は 32-bit 環境のため, Yorick の整数は 32 ビット長の符号付き整数となり, 整数は -2^{31} から $2^{31}-1$ の範囲です. そして, その範囲を越えた整数を入力すると, 1 の補数が返されます:

```
> 2^31
-2147483648
> 2^32
0
> 2^30+2^30
-2147483648
> 2^30+(2^30-1)
2147483647
```

16.4.2　論理値

MATLAB 系の言語や Yorick では, 論理値の偽が '0', 真が '1' で表現し, 通常の数値との演算が行えます. ここで論理値を被演算子とする数値演算を行うと, MATLAB 系の言語では倍精度の浮動小数点数となります. なお, Yorick では論理値は int 型の対象となりますが, 演算は基本的に被演算子の優先度で定まるために, int 型同士の演算であれば int 型となり, MATLAB 系の言語とは異なります.

16.4.3　ベクトルと行列

MATLAB 系の言語と Yorick では, 数値ベクトルや行列の処理に向いています. 簡単に説明するならば, ベクトルは 1 次元的な構造を持つ対象, 行列は 2 次元的な構造を持つ対象です. たとえば, MATLAB 系の言語のベクトルは '[1,2,3]', 行列は '[1,2;3,4]' の様に表記されます. なお, Yorick では C 風の配列が根底にあり, MATLAB 系の言語の様にベクトルと行列の表記とは異なる表記になり, 3 次元以上の構造を持つ多次元配列が自然に扱えます. このベクトルと行列の詳細は §16.6.1 で解説することにします.

16.4.4　文字列

文字列の扱いは MATLAB, Octave と Scilab や Yorick で大きく異なります. MATLAB と Octave で文字列は単引用符' や二重引用符""で括った文字の列ですが, この列は一方でベクトルとしての性格も持ちます:

```
octave:1> '123'==['1','2',"3"]
ans =

   1   1   1

octave:2> length('123')
ans =  3
octave:3> "123"(3)
ans = 3
octave:4> class('1')
ans = char
```

ここでの例では, 単引用符と二重引用符を混在させて用いていますが, この様に混在させて用いることが可能です. そして, 文字列は char 型の対象となります.
Scilab でも文字列も単引用符' や二重引用符""で括った文字の列で, typeof 命令で string 型となる対象です:

```
-->typeof("123")
 ans  =

 string

-->length("1234")
 ans  =

    4.

-->"1234"==["1","2",'3','4']
 ans  =

  F F F F

-->"1234"=='1234'
 ans  =

  T
```

ただし, MATLAB とは異なり, 文字列はベクトルとしての性質はありません. そのために式 '"123"==["1","2",'3','4']' の結果は左側の文字列を右側のベクトルの成分と比較した結果となり, MATLAB や Octave との結果と異なる結果が得られます. そして, Scilab の文字列では演算子 "+" による結合処理が可能です:

```
-->"1234"+'4568'
 ans  =

 12344568
```

しかし, MATLAB と Octave では, char 型は整数に対応するために, 整数値の演算となってしまいます:

```
octave:1> "abc"+"edf"
ans =
```

```
   198   198   201
octave:2> "三毛猫"+"にゃお"
ans =

   455   313   308   457   305   286   458   269   309
```

ここでの例で, '"abc"+"edf"' は '["a"+"e", "b"+"d", "c"+"f"]' と等しく, さらに "a" から "f" の文字は ASCII 文字として 97 から 102 の整数に対応するために上記の結果が得られます. 日本語は, システムで用いる文字コードに依存しますが, ここでの例では UTF-8 の環境で動作させており, この場合には日本語の一文字は 3byte 長となるために 3 つの整数のベクトルとして表現され, "三毛猫"と"にゃあ"は 9 成分の整数ベクトルになります.
一方で, Yorick は最も C 風で, 文字列は二重引用符""で括り, 単引用符' は使えません. そして, 文字列はベクトルにはならず, 文字の結合は演算子 "+" で行えます.

16.5　基本的な計算式の入力と値の代入

MATLAB 風の言語や Yorick では, 四則演算を C と同様の式で表現します. また, 入力行末にセミコロン ";" をつけておくと計算結果を表示しません:

```
octave:1> 1+1;
octave:2> 1+1
ans = 2
```

この機能は大きな数値配列/行列を扱う言語では必須でしょう. また, MATLAB 系の言語や Yorick で, 変数への値の代入は等号記号 "=" を用います:

```
octave:3> a=1.2
a = 1.2000
octave:4> a
a = 1.2000
octave:5> s=a^2*pi
s = 4.5239
```

なお, MATLAB 系の言語と Yorick では自由変数は扱えず, 束縛変数のみが扱えます. たとえば, 立上げたばかりの Octave でいきなり 'a' と入力しても, 未定義の変数であると返されます. 一方で, Yorick では, 演算子 "=" で値の束縛を行っていない変数には全て値 '[]' を自動的に束縛しているので, 事実上, Yorick には自由変数はありません.

16.5.1　数学定数

MATLAB 系の言語には円周率 π ,Napia 数 e や純虚数 i 等の数学定数が予め用意されていますが, これらの値は立ち上げのときに初期化ファイルを使って設定しています. 実際, MATLAB と Octave ではそれらの変数名 (実際は組込函数) は 'pi', 'e', 'i' 等ですが, 大きな問題は, これらの変数値が全く保護されていないことです. たとえば, for 文を使う際に何気なく i を変数として利用すると定数 'i' が呆気なく書換えられてしまいます:

```
octave:1> pi
pi = 3.1416
octave:2> i
i = 0 + 1i
octave:3> e
e = 2.7183
octave:4> i*i
ans = -1
octave:5> a=1+i;
octave:6> b=1-i;a*b
ans = 2
octave:7>
```

この様に MATLAB や Octave, 同様に Yorick でも定数 'i', 'e', 'pi' は予約語として特別に保護されていないのでうっかり利用しないように注意しなければなりません. また, MATLAB や Octave では システム変数かどうかは type 命令で調べられ, who 命令で自分が設定した変数かどうかを確認することができます. 実際に, 定数 'i' と 'pi' を書換えている例を示しておきます:

```
octave:1> who
octave:2> type i
i is a builtin function
octave:3> i=2
i = 2
octave:4> i*i
ans = 4
octave:5> type i
i is a user-defined variable
2
octave:6> type pi
pi is a builtin function
octave:7> pi=10
pi = 10
octave:8> i=sqrt(-1)
i = 0 + 1i
octave:9> pi=acos(-1)
pi = 3.1416
octave:10> who

*** local user variables:

i    pi

octave:11> type pi
pi is a user-defined variable
3.1416
octave:12> for i=1:10
> i*2;
> end;
octave:13> i
i = 10

octave:14> type i
i is a user-defined variable
```

10

この例では一番最初に who 命令を実行した時点では何も表示されませんでしたが, 変数 i や変数 pi に値を入れたあとに who を実行すると利用者定義の変数として ‘i’ と ‘pi’ が表示されます. 実際, 変数の型が調べられる type 命令えば, これらの定数は書換えを行ったために利用者定義変数になっていることが分ります. そして, for 文の例では ‘i’ の値が最終的に 10 で置換えられていることを示しています. ところで, i は数学では式の添字で多用されることもあって, for 文等の反復処理で利用する可能性が高い文字です. そこで, 習慣として変数の末尾には必ず 1, 2, 3 等の数字を付けたり, ‘ii’ のように変数を全て二文字以上にする等の工夫をすることを薦めます.
ただし, MATLAB, Octave と Yorick の純虚数は正確には ‘0+1i’ で, ‘1i’ の ‘1’ と ‘i’ は不可分です. この ‘1i’ は変数ではないので, 変数 ‘i’ の様な書き換えは行えません.
ところで, Scilab の場合はより優れた手法を用いています. 先ず, 組込定数には頭に記号 “%” を付けています. たとえば, 純虚数 i は ‘%i’, 円周率 π は ‘%pi’ と表記します. さらに, これらの変数は書換に対して保護されています.
MATLAB 系の言語には微小量を表現する定数があります. この定数は計算機イプシロン呼ばれる定数で, MATLAB と Octave では ‘eps’, Scilab では ‘%eps’ と表記されます. この計算機イプシロン (ϵ) は, 浮動小数点数で $1 + x \neq 1$ を満す最小の浮動小数点数を指し, 倍精度浮動小数点数では $\epsilon = 2^{-52}$ で与えられます. 計算機イプシロンは零による割算を避けることや, while 文等の反復処理で定数 eps の何倍かに誤差が収まった場合に反復処理を停止するといった使い方もできます[3].

16.6　行列処理

16.6.1　ベクトルと行列の書式

MATLAB 系の言語では 1 次元や 2 次元の数値配列が容易に扱える様に工夫されています. ここで配列の添字は Fotran 同様に 1 から開始します. Yorick の場合, 高次元の配列操作が添字を使って容易に行える様に工夫されていますが, この特性が独特で, 分かり難いのが難点です. MATLAB 系の言語では 1 次元配列をベクトル, 2 次元配列を行列と呼び, 視覚的にも判り易く表示を行います. そして, 行列の書式は MATLAB 系の言語では共通です:

```
octave:20> [1,2,3;4,5,6]
ans =

   1   2   3
   4   5   6

octave:20> [1,2,3;4,5,6]'
ans =

   1   4
   2   5
   3   6
```

[3] MATLAB 系の言語や Yorick では, ‘x=0’ のときに ‘0’, それ以外は ‘1/x’ を計算するときは ‘(x!=0)/(x+(x==0))’ とできます

```
octave:21> [1,2,3]
ans =

   1   2   3

octave:22> [1,2,3]'
ans =

   1
   2
   3
```

この様に, MATLAB 系の言語では行列やベクトルを視覚的に表示します. ここで行列の要素の区切は空白文字かコンマ “,” を用い, 行の区切ではセミコロン “;” を用います.
なお, Yorick では扱う配列が多次元の配列となるために, MATLAB 風の表記ではなく, 記号 “[ ]” で成分を括ることで次元を表現します. たとえば, 2 次元配列の場合は, MATLAB 系の ‘[1,2,3;4,5,6]’ は ‘[[1,4],[2,5],[3,6]]’ と表記し, ‘[[1,2,3],[4,5,6]]’ は MATLAB 系の ‘[1,4;2,5;3,4]’ に対応し, 表記は丁度, 転置の関係になります.

16.6.2 行列の大きさを返す命令

MATLAB 系の言語では, 行列の大きさは size 命令を使えば取得できます. 似たものに length 命令があり, こちらはベクトルの長さを返します:

```
octave:23> a=[1 2 3;4 5 6]
a =

   1   2   3
   4   5   6

octave:24> a(1,2)
ans = 2
octave:25> a(4)
ans = 5
octave:26> a(3)
ans = 2
octave:27> size(a)
ans =

   2   3

octave:28> length(a)
ans = 3
```

Yorick の場合, dimsof を用いて配列の大きさを 1 次元配列で返します. この返却値の第 1 成分が配列の次元, 以降が各成分の大きさとなります:

```
> a=[[1,2,3]]
> dimsof(a)
[2,3,1]
> a=[[1,2,3],[1,2,3]]
```

```
> dimsof(a)
[2,3,2]
> a=[[[1],[2],[3]],[[1],[2],[3]],[[1],[2],[3]]]
> dimsof(a)
[3,1,3,3]
```

この例で示す様に, 配列の次元が記号 “[]” による深さが対応しています.

16.6.3　ベクトルと行列の成分の取出し

ベクトル a の i 成分は ‘a(i)’, 同様に行列 a の i 行 j 列の成分を ‘a(i,j)’ で表現します. この様に, 書式自体は様々な言語の配列の書式と同様ですが, C の様に ‘a[i,j]’ と記号 “[]” を用いないことに注意して下さい. また, ベクトルと行列の添字は数学の行列表記と同様に 1 から開始します.
ここで, MATLAB 系の言語でのベクトルや行列の生成は, 上記の例の様に一気に設定する方法の他に, ‘a(1,2)=1’ の様に成分を指定して設定する方法があります. もし, 行列 a が未定義の場合に ‘a(1,2)=1’ と入力すると, 行列 a の 1 行 2 列目の成分 ‘a(1,2)’ のみが ‘1’ で他が零の一行 2 列の行列が生成されます. この状態からさらに ‘a(3,2)=10’ と入力すると, 行列 a は 3 行 2 列の行列に拡大され, 値が代入された個所以外の値は全て ‘0’ が割当てられた行列となります. なお, Yorick は MATLAB と比較すると C 風で, array 函数で配列を定義し, 配列の拡大も grow 函数等の専用の函数を用いて行う必要があり, MATLAB 系の言語の気楽さはありません.
さて, ここで行列をベクトルと看做すこともできます. たとえば, 行列 a を m 行 n 列の行列とするときに ‘a(i,j)’ を ‘a((j-1)*m+i)’ のベクトルの成分としても解釈できるのです. しかし, この場合は a が何等の方法で予め定義された場合に限ります. つまり, 未定義の a をベクトルとして成分に値を設定すれば, 当然, a はベクトルになります.
ベクトルや行列の一部の取出は非常に容易に行えます. ベクトルであれば, i 番目の成分から j(≥ i) 番目の成分を取出す場合は ‘a(i:j)’ で取出せます. 行列の場合は, このベクトルの取出を拡張したものとなります. 実際, 行列の a(i,j) 成分と a(m,n) 成分を対角線とする小行列の取出は ‘a(i:m, j:n)’ で行えます. さらに, i 行目だけを行ベクトルとして取出す場合は ‘a(i,:)’ , 同様に, j 列目で構成される列ベクトルは ‘a(:,j)’ で得られます:

```
octave:31> a(4:6)
ans =

  4  5  6

octave:32> a=[1:4;5:8;8:11]
a =

   1   2   3   4
   5   6   7   8
   8   9  10  11

octave:33> b=a(2,:)
b =

  5  6  7  8
```

```
octave:34> c=a(:,2)
c =

  2
  6
  9
```

この様に MATLAB 系の言語ではコロン ":" を用いて行列同士の代入を簡略化が可能なのです. この特性は Yorick でも同様ですが, Yorick の大きな特徴として, 配列の添字を用いてさらに複雑な処理が行える性質があります. この添字を用いた処理の詳細は参考文献 ([65], [66]) を参照して下さい. なお, Python でも Yorick の配列の可変添字 (rubber index) と呼ばれる添字があります.
MATLAB 系の言語や Yorick では, 行列を列, あるいは行ベクトルの集合と看做して新しい行列も容易に生成できます:

```
octave:37> a=rand(4,4)
a =

  0.204195   0.372247   0.195707   0.529230
  0.063849   0.915721   0.857846   0.630308
  0.191641   0.602701   0.667216   0.162591
  0.261553   0.435798   0.732046   0.561905

octave:38> b=zeros(4,4)
b =

  0  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0

octave:39> b(:,2)=a(:,4)
b =

  0.00000   0.52923   0.00000   0.00000
  0.00000   0.63031   0.00000   0.00000
  0.00000   0.16259   0.00000   0.00000
  0.00000   0.56191   0.00000   0.00000
```

ここで 'a(:,1)' で代入を行う場合, 代入する側と代入される側のコロン ":" で指定した部位が同じ大きさでなければなりません:

```
octave:39> a
a =

  0.204195   0.372247   0.195707   0.529230
  0.063849   0.915721   0.857846   0.630308
  0.191641   0.602701   0.667216   0.162591
  0.261553   0.435798   0.732046   0.561905

octave:40> c=zeros(2,2)
c =
```

```
  0   0
  0   0

octave:41> c(:,2)=a(:,2)
error: a(i, j) = x: x must be a scalar or the number of elements in i must
error: match the number of rows in x and the number of elements in j must
error: match the number of columns in x
error: assignment failed, or no method for 'matrix = matrix'
error: evaluating assignment expression near line 41, column 7
octave:41> a(:,2)=a(:,2)
error: a(i, j) = x: x must be a scalar or the number of elements in i must
error: match the number of rows in x and the number of elements in j must
error: match the number of columns in x
error: assignment failed, or no method for 'matrix = matrix'
error: evaluating assignment expression near line 41, column 7
octave:41> a=a(:,4)
a =

  0.52923
  0.63031
  0.16259
  0.56191
```

この例で示す様に, コロン ":" で指定した行列の右辺と左辺の大きさが違う場合はエラーになります. さらに左辺が未定義の場合, コロン ":" を用いて成分を指定してもエラーになります. なお, Yorick では大きさが相違する配列への割当処理や, 添字の処理で, 大きさが異なる場合にシステムが落ち易い傾向があります. そのために, 明示的に代入記号 "=" の両辺の大きさを予め同じ大きさになる様に調整しておくべきです:

```
octave:43> d=zeros(10,2)
d =

  0   0
  0   0
  0   0
  0   0
  0   0
  0   0
  0   0
  0   0
  0   0
  0   0

octave:44> a
a =

  0.204195   0.372247   0.195707   0.529230
  0.063849   0.915721   0.857846   0.630308
  0.191641   0.602701   0.667216   0.162591
  0.261553   0.435798   0.732046   0.561905

octave:45> d([7:10],1)=a(:,1)
d =
```

```
0.00000   0.00000
0.00000   0.00000
0.00000   0.00000
0.00000   0.00000
0.00000   0.00000
0.00000   0.00000
0.20419   0.00000
0.06385   0.00000
0.19164   0.00000
0.26155   0.00000
```

記号 ":" を使って数値ベクトルの間隔を指定できます. たとえば, "1:5" は "1:1:5" と同じ意味で間隔が 1 の場合, 真中の間隔指定は不要です. なお, "1:2:6" の場合,6 を越えない値が設定されるために, 得られるリストは '[1 3 5]' になります.

16.7 MATLAB 系言語での演算

16.7.1 四則演算を含む基本的な演算

MATLAB 系の言語や Yorick の四則演算は C 等のプログラム言語とほぼ同様の書式ですが, 被演算子が行列の場合, MATLAB 系の言語では行列の演算となって, 成分毎の演算となる Yorick と異なることに注意して下さい.
ここでは行列を大文字, スカラーを小文字で表記して, MATLAB 系の演算を纏めておきます:

MATLAB 系言語の演算

演算	例	概要
和 "+"	A+B,a+b,A+b	同じ大きさの行列, 行列とスカラーの和
差 "-"	A-B,a-b,A-b	同じ大きさの行列, 行列とスカラーの差
積 "*"	A*B,a*b,a*B	行列同士の行列積, 行列とスカラーの積
冪 "^"	a^b,A^b,a^B	行列の冪
商 "/"	A/B,a/b,A/b	行列同士の場合, 演算子右側を逆行列にして行列積を計算. A/B=A*B^(-1) と同値
商 "\"	A \ B,a \ b,a \ B	行列同士の場合, 演算子左側を逆行列にして行列積を計算. A \ B=A^(-1)*B と同値
積 ".*"	A.*B,a.*b,a.*B	同じ大きさの行列の成分毎の積, 行列とスカラーの積
冪 ".^"	A.^B,a.^b,A.^b, a.^B	行列の冪
商 "./"	A./B,a./b,a./B	同じ大きさの行列の成分毎の商 (/), 行列とスカラーの商
商 (.\ )	A.\ B,a.\ b,a.\ B, A .\ b	同じ大きさの行列の成分毎の商 (\), 行列とスカラーの商
転置 "'"	A'	行列の転置

実際に動作を確認してみましょう. 行列 A と行列 B は以下のものとします:

```
octave:1> A=[1,2;3,1];
octave:2> B=[5,1;2,1];
```

和 “+” と差 “-”　行列同士の演算であれば, 両者が同じ大きさでなければなりません. また, この場合は成分毎の演算となります. 一方で, 行列とスカラーの演算も可能で, この場合はスカラーが行列の各成分に分配されます:

```
octave:3> A+1
ans =

  2  3
  4  2

octave:4> A-1
ans =

  0  1
  2  0

octave:5> A+B
ans =

  6  3
  5  2
```

積 “*” と積 “.*”　MATLAB 系の言語では, 演算子 “*” と演算子 “.*” の二つの積演算子があります. この演算子は, 被演算子のどちらか一方がスカラーであれば同じ結果になりますが, 両方が行列の場合は意味が違います. まず, 演算子 “*” は通常の行列の積で, 演算子 “.*” は成分毎の積, つまり, 行列 $A = (a_{ij})$ と $B = (b_{ij})$ に対し, $A * B$ の (i, j) 成分は ‘$\sum_{k=1}^{m} A(i,k)B(k,j)$ と通常の行列の積に対応し, ‘$A.*B$’ の各 (i, j) 成分は ‘$(a_{ij} * b_{ij})$’ と成分単位の積に対応します. ここで MATLAB 系の言語では, 演算子 “.*” の様に先頭に記号 “.” が付く演算子は全て成分毎の演算を意味します.

```
octave:9> A*B
ans =

   9   3
  17   4

octave:10> A.*B
ans =

  5  2
  6  1
```

ところで, Yorick の場合は基本的に演算は成分単位の演算であり, 逆に行列の積は添字を用いた処理を行う必要があります. つまり, 行列 A, B の積 AB は $A(,+) * B(+,)$ と $\sum_k A(i,k)B(k,j)$ を意識し

た表記となります. そして, この表記は他の次元にも拡張可能なために, 高次元の数値テンソルの演算も容易に表現可能であるという特徴を持つのです.

```
> A=[[1,3],[2,1]]
> B=[[5,2],[1,1]]
> A+1
[[2,4],[3,2]]
> A+B
[[6,5],[3,2]]
> A*B
[[5,6],[2,1]]
> A(,+)*B(+,)
[[9,17],[3,4]]
> B(,+)*A(+,)
[[8,5],[11,5]]
```

冪 “^と “.^” MATLAB 系の言語では, 通常の冪 “^” のみ, 双方が行列の場合は使えません:

```
octave:29> A.^2
ans =

  1  4
  9  1

octave:30> 2^A
ans =

  5.6453  4.3104
  6.4656  5.6453
```

なお, Yorick では演算は基本的に成分単位となるので, MATLAB 系での演算子 “^^” は存在せず, Yorick の演算子 “^” は MATLAB 系の言語での演算子 “.^” に対応します.

商 “/”, 商 “\” 同じ大きさの行列に対しては ‘A/B=A*B^(-1)’, ‘A\B=A^(-1)*B’ が成立します. ここで行列とスカラーの場合, スカラーを分母にする計算は可能ですが, 行列を分母にする計算, たとえば, ‘B \ 2’ や ‘2/A’ の計算はエラーになります:

```
octave:15> A/B
ans =

  -1.00000   3.00000
   0.33333   0.66667

octave:16> A\B
ans =

  -0.20000   0.20000
   2.60000   0.40000

octave:17> A*B^(-1)
ans =
```

```
   -1.00000    3.00000
    0.33333    0.66667

octave:18> A^(-1)*B
ans =

   -0.20000    0.20000
    2.60000    0.40000

octave:19> 2\B
ans =

   2.50000   0.50000
   1.00000   0.50000

octave:20> A/2
ans =

   0.50000   1.00000
   1.50000   0.50000
```

Yorick では, 演算子 "/" は MATLAB 系の言語の演算子 "./" に対応し, 純粋に逆行列を計算する演算子はありません.

16.7.2　演算の処理速度の比較

対話処理可能な言語による処理は, C の様なコンパイラが生成した実行ファイルによる処理と比べて処理速度が劣るのが通常です. しかし, MATLAB 系の言語で行列演算は数値演算ライブラリを利用するために, 実用で問題になる程の速度の低下はありません. しかし, 処理言語の比重が高まったり, for 文による反復処理があれば, 大きな速度低下が発生します. また, 四則演算でさえも対処方法を誤れば, 計算時間を無駄に消費しかねません.
たとえば, 行列の成分単位の積や冪でも, 処理方法の違いで速度が異なります. ここでは Octave 上で乱数行列を生成する rand 命令 から 1000 行 1000 列の行列を生成し, この行列を使って確認してみましょう.
なお, ここでの "ans =" に続く数値が処理時間 (cputime) で, ここの数値が小さい方が処理が高速です. この計算は計算機 (core2duo E6700 2.66GHz, openSUSE 11.3 環境の PC) で実行しています:

```
octave:1> a=rand(1000);
octave:2> t1=cputime;a.*a;t2=cputime;t2-t1
ans =  0.011997
octave:3>  t1=cputime;a.^2;t2=cputime;t2-t1
ans =  0.010998
octave:4>  t1=cputime;exp(2*log(a));t2=cputime;t2-t1
ans =  0.080987
octave:5>  t1=cputime;a.*a.*a.*a.*a.*a.*a.*a.*a.*a.*a.*a;t2=cputime;t2-t1
ans =  0.14698
octave:6>  t1=cputime;a.^12;t2=cputime;t2-t1
ans =  0.019996
octave:7>  t1=cputime;exp(12*log(a));t2=cputime;t2-t1
```

```
ans =  0.079988
```

いかがでしょうか？ 二乗の処理だけに限定しても, 最悪で 0.088, 最善で 0.011 と随分幅がありますね. また, 12 乗の処理は安易に 12 回の積で表現したものが最悪で, 次に指数函数と対数函数を用いたもの, 最善は冪演算子 “^” を用いたものです. この結果は各言語で結果が異なり, その違いがアプリケーションの内部処理や利用するライブラリの違いに結び付きます. 実際, 冪演算子と積は Scilab と Yorick では冪が小さければ積表現の方が高速になりますが, MATLAB と Octave では, ここで示すように冪演算子の方が僅かに速い傾向があります.
ここで, 安易な積が最悪の結果になっていますが, ここで冪を適当に区切って計算するとどうなるでしょうか？ 今度は 9 乗の計算で 3 個単位で纏めて計算させたものも含めて比較してみましょう:

```
octave:15> t1=cputime;a.*a.*a.*a.*a.*a.*a.*a.*a;t2=cputime;t2-t1
ans =  0.10616
octave:16> t1=cputime;b=a.*a.*a;b=b.*b.*b;t2=cputime;t2-t1
ans =  0.051572
octave:17>  t1=cputime;b=a.^9;t2=cputime;t2-t1
ans =  0.020127
octave:18> t1=cputime;exp(9*log(a));t2=cputime;t2-t1
ans =  0.080215
```

この様に, 安易な積表現が最も遅く, 次に指数函数と対数函数を用いたもの, それから 3 個単位で分けて計算したものが続き, 最善は冪演算子 “^” を用いたものになります. この傾向は MATLAB 系の言語と Yorick で観察されるものです. この結果から比較的大きな冪については, 安易な積表現は避けるべきであると言えます.
この様に冪や積の様な演算でも工夫の余地があることが分りましたが, それ以上に, MATLAB 系の処理言語では for 文を用いるだけで処理速度を低下させられます. これは配列から指定された行列の成分を取出して複製し, その複製を使って計算した結果を再び配列に戻す処理を行うからです. これに対し, 行列やベクトルの演算では標準的な行列演算ライブラリを直接利用するために, 下手なプログラムを組むよりも高速な処理が行える訳です. しかし, 行列の成分に対して条件分岐で値を定めなければならないときは for 文の様な反復処理が必要に思えます. しかし, MATLAB 系の言語では,「**並びの照合**」を上手く使うことで対処できるのです.

16.8 並びの照合

MATLAB 系の言語や Yorick では「**並びの照合**」と呼ばれる処理が行えます. この並びの照合を効果的に適用すれば, 処理速度を引き起し易い反復処理を排除でき, その上, 見通しが良く簡潔なプログラムも構築できるのです.
並びの照合は基本的に, 与えられた行列で, 与えた条件に適合するものがあるかどうかを検証する手法です. 次の例では与えられたベクトルから 2 に等しい成分があるかを検証し, その場所を find 命令を用いて探す処理を実行しています.
まず, MATLAB 系の言語と Yorick では真が ‘1’, 偽を ‘0’ として扱います. そして, MATLAB 系の言語には ’0’ と異なる箇所を検出する命令として find 命令があります :

```
octave:66> x=[1:5,5:-1:1]
```

```
x =

  1   2   3   4   5   5   4   3   2   1

octave:67> x==2
ans =

  0   1   0   0   0   0   0   0   1   0

octave:68> y=find(x==2)
y =

  2   9

octave:69> x(y)
ans =

  2   2
```

この例では '2' に等しいものを検出していますが, C と比べて簡潔な処理で '2' に等しい元の位置を検出しています. ちなみに, Yorick では 1 次元配列として検出する場合には where, 配列の次元に合せた形式で検出する場合は wehre2 命令を用います.
この検出は '0' か '0' でないかのみを問題とするので, 等号だけではなく, 不等号に対しても適用できます:

```
octave:83> x=[1:5,5:-1:1]
x =

  1   2   3   4   5   5   4   3   2   1

octave:84> y=find(x>3)
y =

  4   5   6   7

octave:85> z=x>3
z =

  0   0   0   1   1   1   1   0   0   0

octave:86> z.*x
ans =

  0   0   0   4   5   5   4   0   0   0
```

find 命令は与えられた行列で '0' と異なる成分の位置を返す命令です. MATLAB 系の言語で行列の成分は "(i,j)" で指定しなければならないので, 並びの照合の対象がベクトルでなければ 'x(find(x¿3))' の様な処理は間違になります. ここで 'find(x¿3)' で返された列ベクトルは 'x¿3' で生成された行列を列ベクトルから構成されたものと看做しています. 具体的には m 行 n 列の行列の (i,j) 成分は, m*n 個の成分の列ベクトルの m*(j-1)+i 番目の成分に対応します:

```
octave:5> aa=rand(5);
```

```
octave:6> bb=aa>0.5
bb =

  1  1  0  0  1
  1  1  0  1  0
  0  1  1  0  1
  1  0  0  1  0
  0  1  1  1  0

octave:7> find(bb)
ans =

   1
   2
   4
   6
   7
   8
  10
  13
  15
  17
  19
  20
  21
  23

octave:8>aa(find(bb))
error: single index only valid for row or column vector
error: evaluating index expression near line 8, column 1
octave:8>
```

MATLAB 風の言語では, 'y=x(x¿3)' の様に find 命令に相当する命令を利用せずに処理する事もできます. この様に並びの照合を適用して処理の簡略化が行えます. たとえば, 上記の与えられたベクトルから '3' よりも大きな数値に対してのみ 2 倍する事も簡単にできます:

```
octave:89> x=[1:5,5:-1:1]
x =

  1  2  3  4  5  5  4  3  2  1

octave:90> y=find(x>3)
y =

  4  5  6  7

octave:91> for i=x(y)
> 2*i
> end
ans = 8
ans = 10
ans = 10
ans = 8
```

Yorick でも同様で, 1 次元配列の場合は find 命令の箇所を where で置き換え, 高次元配列であれば where2 で置き換えられ良いのです.
前置はここまでとし, for 文を除外する方法について簡単に解説しておきましょう. 天下り的ですが, 与えられたベクトルに対し, '3' よりも大なら 2 倍, それ以外は '0' にするという処理例を示しておきましょう:

```
octave:1> x=[1:5,5:-1:1]
x =

   1   2   3   4   5   5   4   3   2   1

octave:2> z=zeros(size(x));
octave:3> z(x>3)=2*x(x>3)
z =

    0    0    0    8   10   10    8    0    0    0
```

ここでは 10 成分の配列 x を生成し, その x の各成分に対して '3' よりも大の箇所を検出し, その部位を 2 倍にしています. 通常のプログラム言語であれば, for 文を用いて成分を取り出し, 取り出した成分に対して if 文を用いた条件分岐を入れるでしょう. ここでは零行列 z を生成し, 'x¿3' を満す箇所 ('(x¿3)') を 2 倍にして代入するという処理を 'z(xtextgreater3)=2*x(x>3)' だけで行っているのです. 如何でしょうか？ for 文が必要となるのは結局, 成分を取り出して加工する箇所であり, 単純な計算であれば何らかの行列処理で置き換えられるものです. しかし, if 文といった処理の多くは, '0' と '1' だけの真理値行列に変換して考えれば, 行列の成分単位の演算と, 対応する成分の取り出しだけに帰着できるのです.

16.8.1　for 文と並びの照合の処理速度の比較

ここでは並びの照合による処理と for 文による反復を比較してみましょう:

```
octave:8>   x=rand(100000,1);
octave:9>   t1=cputime;(x>=0.5)*2+(x<0.5)*(-1);t2=cputime;t2-t1
ans =  0.0040000
octave:10>   t1=cputime;tmp=(x>=0.5);tmp*2+(1-tmp)*(-1);t2=cputime;t2-t1
ans =  0.0039990
octave:11>   t1=cputime;tmp=(x>=0.5);tmp*2+tmp-1;t2=cputime;t2-t1
ans =  0.0040000
octave:12>   t1=cputime;for i1=[1:length(x)]
>   if x(i1)>=0.5; x(i1)=x(i1)*2; else x(i1)=-x(i1);
>   end;end;t2=cputime;t2-t1
ans =  1.1658
```

最初の例では $x \geq 0.5$ と $x \geq 0.5$ の二種類の並びの照合を実行しています. 2 番目の例では '0.5' の並びの照合のみを行っていますが, ここで '-1' の積を余計に実行しています. 3 番目の例では 2 番目の例に似ていますが, 最後の積を予め実行したものです. そして, 最後の例は for 文を用いて同じ処理を繰返したものです. 最速のものと比較して実に 291 倍もの差が生じていますね. この Octave の現象は極端な結果ですが, MATLAB にせよ Yorick にしても, 25 倍近くの処理速度の低下が生じ

ます. だから, 処理速度を気にするのであれば, for 文等の反復処理の利用は可能な限り避け, 行列の演算に置換えられるものは徹底して書換した方が安全なのです. また, その書換によってプログラムが非常に見易いものになるという御利益もあるのです.

16.8.2 any と all

ここで条件を満す成分は find 命令等で捜し出せますが, そこまでしなくても「存在する」のか「存在しない」だけが問題となることもあります. このときに便利な MATLAB と Octave の命令に any と all 命令があります.
any 命令は行列データに零でない成分があれば '1', 零行列であれば '0' を返します. 一方で, all 命令は全ての成分が '0' でない場合のみ '1' , それ以外であれば '0' を返します:

```
octave:1> a=rand(4,5)-rand(4,5)
a =

  -0.671536   0.539990   0.205556   0.171495   0.276634
  -0.784795   0.585699  -0.274086  -0.448760   0.131415
  -0.072425   0.276092   0.355440  -0.257676   0.357314
   0.356246   0.061500  -0.318618   0.241485  -0.295221

octave:2> if any(a(:,1)>0)
> lst=find(a(:,1)>0);
> b=exp(a(lst,1));
> end;
octave:3> b
b = 1.4280
```

この例では行列 A の一列目に正の成分があれば, その成分に exp 函数を作用させています. 次に all 命令の例を示します:

```
octave:11> all(a(:,1)==a(:,3))
ans = 0
octave:12> a(:,1)=a(:,3);
octave:13> all(a(:,1)==a(:,3))
ans = 1
octave:14>
```

この例では, 先程の行列 a の 1 列目と 3 列目が等しいかどうかを判断させ, 次に行列 a の 1 列目に 3 列目を代入して, 1 列目と 3 列目が等しいか判断させています.
この any と all は Scilab では mtlb_any と mtlb_all, Yorick では anyof と allof に対応します.

16.9 便利な行列の定義方法

16.9.1 等間隔のベクトルの生成

MATLAB や Octave では等間隔のベクトルが [⟨ 初期値 ⟩ : ⟨ 刻み幅 ⟩ : ⟨ 終値 ⟩] で簡単に設定できます:

```
octave:111>bb=[0:0.1:0.5]
bb =

   0.00000   0.10000   0.20000   0.30000   0.40000   0.50000
```

先程調べた様に, 下 手に for 文を用いると速度の低下に繋がり易い問題がありますが, これは行列の生成でも同様です. 処理速度を比較してみましょう:

```
octave:116> t1=cputime;for i=1:100; bb(i,i)=0.01*i;end;t2=cputime;t2-t1
ans = 0.0047450
octave:117> t1=cputime;dd=[1:100];t2=cputime;t2-t1
ans =  4.9114e-05
```

この場合でも for 文を使うと著しく速度が低下していますね.

16.9.2　対角行列の生成

対角成分を除いて全て零となる対角行列に対しては幾つかの便利な命令があります. まず, 対角成分に数値を設定するためには diag 命令 を用います. この diag 命令には対角成分に設定する数値をベクトルで与えます. ここで, 対角成分を指定した列ほど移動させたり, 与えられた行列の対角成分を抜出すこともできます:

```
octave:118> diag([1,2])
ans =

  1  0
  0  2

octave:119> diag([1,2],1)
ans =

  0  1  0
  0  0  2
  0  0  0

octave:120> diag([1,2],2)
ans =

  0  0  1  0
  0  0  0  2
  0  0  0  0
  0  0  0  0

octave:121> diag([1,2],-2)
ans =

  0  0  0  0
  0  0  0  0
  1  0  0  0
  0  2  0  0
```

```
octave:122> a=rand(3,3)
a =

  0.58905   0.61873   0.63411
  0.19251   0.11602   0.18785
  0.54143   0.83113   0.83952

octave:123> diag(a)
ans =

  0.58905
  0.11602
  0.83952
```

対角成分が全て 1 で他が全て 0 の行列の生成は eye 命令 を使います:

```
octave:124> eye(3,2)
ans =

  1  0
  0  1
  0  0

octave:125> eye(2,3)
ans =

  1  0  0
  0  1  0
```

16.10 多項式の扱い

Octave や MATLAB では多項式が扱えますが, 数式処理の様に直接多項式は扱えず, 多項式の係数ベクトルの書式で扱います:

多項式の変換

$a_n x^n + a_{n-1} x^{n-1} + \cdot + a_1 x + a_0$	$\Leftrightarrow$	$[a_n, a_{n-1}, \ldots, a_1, a0]$

この様に多項式は 1 変数のものに限定されます.
ところで, 多項式の積は conv 命令, 多項式の商と剰余は deconv 命令, 変数に値を代入した場合の多項式の値は polyval 命令で求めます:

```
octave:2> conv([1,2],[1,-2])
ans =

   1   0  -4

octave:3> conv([1,2,2,1],[1,-2])
ans =

   1   0  -2  -3  -2
```

```
octave:4>   [aa,bb]=deconv([1,3,3,1],[1,1,1])
aa =

  1   2

bb = -1

octave:5> polyval([1,2,3,4],2)
ans = 26
```

この例では最初に ‘conv([1,2],[1,-2])’ で 多項式 $(x+2)(x-2)$ の展開を計算しています. この結果はベクトル ‘[1,0,-4]’ で与えられ, $x^2+0\cdot x-4$, すなわち, x^2-4 に対応します. 次に x^3+3x^2+3x+1 を x^2+x+1 で割ったときの商と剰余は dconv を用いて計算しています. この結果から $x^3+3x^2+3x+1 = (x^2+x+1)(x+2)-1$ であることが判ります. この他にも多項式を扱う函数が幾つか存在しますが, どれも直接的な計算を行うものではなく, あくまでも多項式をリストで扱うものです.
この様に Octave では多項式の処理は得意とは言い難い面があります. だからこそ, この様な処理で複雑なものは Maxima 等の数式処理に任せ, 逆に数値行列の演算に徹するのが効率の面でも良いでしょう. 実際, 何でも一つの道具だけで済まそうとすることはあまり賢明な手段はありません.
なお, Scilab であればよりもう少しまともな処理が行えますが, 一寸した伝達函数の処理に使える程度で, 数式処理 Maxima 等と比較して勝るものではありません.

16.11　M-file

MATLAB や Octave では, ファイルの修飾子が “.m のファイル (M-file) に命令を記述します. この M-file がカレントディレクトリ, あるいは M-file が存在するディレクトリが検索経路に含まれていれば自動的に M-file をの読込んで実行します.
では次の函数 nekoneko の MATLAB や Octave への取込について解説しましょう:

函数 nekoneko

```
function [z]=nekoneko(x,y)
         if length(x)>length(y)
            z=x;
         else
            z=x./y;
         end;
```

基本的に M-file はファイルの内容です. だから, emacs 等のエディタで作成し, それを MATLAB や Octave で読込む手順になります. ここで, Octave ならば直接函数を定義することができまが, MATLAB では直接定義することはできません. GUI を利用の場合は付属のエディタで編集することが前提となります. Scilab や Yorick では直接定義することが可能ですが, Yorick の場合はヘルプの内容は反映されないので注意して下さい.
ここでは Octave への直接入力の様子を示しますが, 行先頭の記号 “¿” は Octave の入力待ちのプロンプトです:

```
octave:1> function [z]=nekoneko(x,y)
> if length(x)==length(y)
> z=x./y;
> else if length(x)>length(y)
> z=x;
> else
> z=y;
> end
> end
> end
octave:2> nekoneko([1:3],[3:-1:1])
ans =

   0.33333   1.00000   3.00000

octave:3> nekoneko([1:3],[3:-1:0])
ans =

   3   2   1   0

octave:4> nekoneko([1:5],[3:-1:0])
ans =

   1   2   3   4   5
```

ここで示した入力方法はMATLABでは行えません. MATLABでは基本的にM-fileに表記し, それをMATLABが読込むという手法となっています. では単純な数学的函数を表現する場合もM-fileを記述しなければなないのでしょうか？MATLABやOctaveではλ式に似た表記で函数が定義可能なのです:

```
octave:7> fx=@(x) x^2-1
fx =

@(x) x ^ 2 - 1

octave:8> fx(10)
ans =  99
octave:9> fxyz=@(x,y,z) x^2+x*y+y*z-z^2-1;
octave:10> fxyz(1,2,3)
ans = -1
```

ここで定義された対象はfunction_handleと呼ばれる型になります. MATLABやOctaveで記述した命令/函数の内容を見たい場合, type命令を利用します:

```
octave:11> type nekoneko
nekoneko is a user-defined function:

function z = nekoneko (x, y)
  if length (x) == length (y)
    z = x ./ y;
  else
    if length (x) > length (y)
      z = x;
```

```
    else
      z = y;
    endif
  endif
endfunction
octave:12> type fx
fx is a variable
@(x) x ^ 2 - 1

octave:13> class(fx)
ans = function_handle
```

ここで type 命令で内容が見られるものは, M-file, function_hanlde と利用者定義変数で, 組込函数は無理です. ここで, 組込函数かどうかは who 命令で調べられます:

```
octave:8> who

*** currently compiled functions:

length     nekoneko

*** local user variables:

aa  bb

octave:9> type aa
aa is a user-defined variable
[ 1, 2, 3 ]
octave:10>
```

16.12　外部アプリケーションの起動命令

MATLAB と Octave には外部命令を実行する命令として, system 命令と exec 命令の二種類があります. ここで MATLAB は記号 "!" を先頭に付けて外部の命令を起動させますが, 行末に; を追加すると外部命令のオプションとして判断されるので注意して下さい. Octave では記号 "!" は否定を表わす not の意味になるので, MATLAB と Octave の双方で動作するプログラムを開発する場合には特に注意が必要になります.

exec 命令と system 命令の違いは, exec 命令は正常に処理の実行を終えると Octave 自体を終了させますが, system 命令は実行後に Octave に戻ります. ここでは system 命令を用いた安易な例を以下で紹介しましょう.

まず, system 命令で起動させるプログラムは外部のプログラムであれば何でも構いません. ここでは次のシェルスクリプト (tama) とします：

シェルスクリプト tama

```
1 #!/bin/sh
2 ls -l | awk '{print $5}'>x1
```

このスクリプト tama はカレントディレクトリ上でファイルサイズのみをファイル x1 に出力するものです. それと, system 命令を用いるプログラム mike を以下に示します:

M-file mike

```
1 function x = mike ()
2   system ("tama");
3   load ("x1");
4   x = sum (x1);
5 endfunction
```

M-file の mike はシェルスクリプト tama を system 命令で起動させ, 結果ファイル x1 を読込み, その総和を計算します. system 命令で実行する命令 tama は環境変数 path で設定されたディレクトリ, あるいはカレントディレクトリ上にあれば大丈夫です. なお, ディレクトリを 'system("/usr/bin/tama")' のように直接指定しても良いでしょう. さらに system 命令で実行するプログラムが引数を必要とする場合, 'system("tama mike")' のよう通常利用する命令を単に二重引用符""で括ります.
たとえば, 次のシェルスクリプト pochi で指定したディレクトリ上のファイルサイズを x1 に出力します. 上述の tama との違いは二行目に$1 が追加されている点です.

シェルスクリプト pochi

```
1 #!/bin/sh
2 ls -l $1 |  awk '{print $5}'>x1
```

この pochi に対して, mike を改良した kuro を以下に示します.

M-file kuro

```
1 function x = kuro (wrd)
2       evl=["pochi ",wrd];
3       system (evl);
4       load ("x1");
5       x = sum (x1);
6 endfunction
```

この kuro では evl で文字列の結合を行っています. たとえば wrd として "/usr" が与えられると, /usr ディレクトリ上のファイルサイズの総和を計算することになりますが, evl では文字列 "pochi " と word として与えられた "/usr" が結合された "pochi /usr" が代入されます. この値を system 命令で評価 (そのために evl を二重引用符""で括っていません) して生成された x1 を用いて総和が計算されます.
このように system 命令を用いることで外部プログラムの利用が可能となり, Octave への外部プログラムの組込みに悩む前に安易にシステム動作の確認が行えるので重宝します.
さらに, Octave には cd, ls や pwd といった命令が利用可能で, その働きも UNIX のそれと全く同じものです.
なお, Scilab には unix 命令が, Yorick には system 命令が, MATLAB の system 命令と同様の働きを行います.

16.13　グラフ表示機能

MATLAB 風の言語では高度なデータ可視化が行えます. このデータ可視化では, 言語による違いが出易いところで, ここでは詳細の解説は行いません. 実際, グラフ処理では MATLAB を基準にすると, Octave が最も類似しており, 次に Scilab が似た使い方が可能です. それに対し, Yorick は MATLAB との仕様の違いが目立ちます. そこで, ここでは MATLAB 系の言語に限定し, さらに, 違いの少ない 2 次元グラフに限定して簡単に解説しておきましょう.
MATLAB 系の言語で, 2 次元グラフの表示は plot 命令を用います:

plot 命令

plot(⟨Y − データ ⟩,⟨ 書式 ⟩,...)
plot(⟨X − データ ⟩,⟨Y − データ ⟩,⟨ 書式 ⟩,...)

plot 命令に Y 軸データ単体を与える場合, 行や列ベクトル, あるいは行列が与えられます. ここで, m 行 n 列の行列 A を与えた場合, plot 命令は A の n 個の列ベクトルを描きます. この場合, X 軸は 1 から n の整数で, Y 座標は, その X 座標に対応する列ベクトルの成分になります.
X と Y のデータを分けて描くこともできます. この場合, X と Y は同じ大きさのベクトル, あるいは行列を設定します.

網目を入れたい場合

グラフに網目を入れたければ, grid 命令を使います. この grid 命令は引数として文字列の “on” か “off” だけを取ります. “on” の場合に網目を入れ, “off” の場合には網目の表示を解除します.

グラフの重ね描きをしたい場合

色々なデータのグラフを重ね描きをしたい場合は hold 命令を使います. この hold 命令も grid 命令と同様に引数として文字列の “on” か “off” だけを取ります. “on” の場合は重ね描きを行い, “off” の場合には重ね描きを行わずにグラフの更新を行います.

グラフにラベルや表題を入れたい場合

X 軸, Y 軸, Z 軸上にラベルを入れたければ,xlabel , ylabel ,zlabel 命令を用います. 上側に表題を入れたければ title 命令を用います. これらの命令は一つの文字列を引数とします.
では, 実際に曲線を描いてみましょう:

単純な描画

```
octave:1> a1=[0:0.05:1]*2*pi;
octave:2> b1=sin(a1);
octave:3> plot(b1);
```

最初に配列 b1 を描くと図 16.1 に示す様に X 軸には配列番号が振られます:

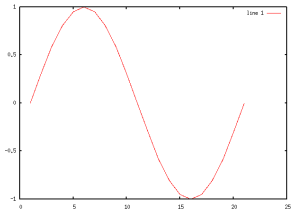


図 16.1: 正弦函数の描画 (Y 座標のみ)

X-Y グラフに表題や軸のラベルの表示

```
octave:4> plot(a1,b1);
octave:5> xlabel("X")
octave:6> ylabel("Y")
octave:7> title("sine curve");
```

配列 a1 と配列 b1 の対にすると, 図 16.2 に示すように配列 a1 を X 軸, 配列 b1 を Y 軸の座標としてグラフを描き, xlabel 命令と ylabel 命令で X 軸と Y 軸にラベルを入れ, title 命令でグラフの表題を入れたものが図 16.3 になります. なお, xlabel, ylabel と title 命令を実行すると, グラフは再描画されます:

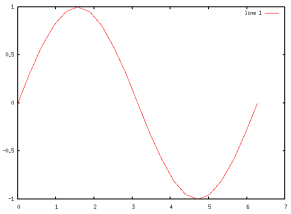


図 16.2: 正弦函数の描画

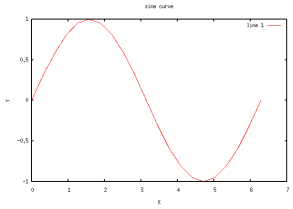


図 16.3: グラフにラベルと表題を追加

網目の表示

```
octave:8> grid("on");
```

'grid("on")' で網目を入れたものが図 16.4 になります:

複数のグラフを表示

```
octave:9> hold("on");
octave:10> c1=cos(a1);
octave:11> plot(a1,c1);
octave:12> title("red:sin, green:cos");
```

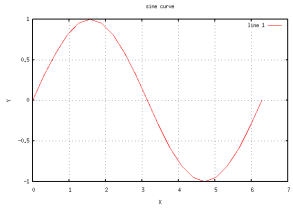


図 16.4: 網目を追加

```
octave:13> hold("off");
octave:14> plot(a1,b1,a1,c1);
octave:15> X=[a1;a1];
octave:16> Y=[b1;c1];
octave:17> plot(X,Y);
octave:18> plot(X',Y');
```

複数のグラフを同時に描く方法には二種類あります. 一つは hold 命令で前の描画を消さない様にして繰返して描く方法, もう一つは一度に描く方法です.
plot 命令で一度に描く場合, 'plot(a1,b1,a1,c1)' の様に X と Y の値の対を並べる方法, 'X=[a1;a1]' と 'Y=[b1;c1]7 の様に X の値と Y の値の行列を作り, 'plot(X',Y')' で一度に描く方法があります. どちらの方法でも図 16.5 の様なグラフが描けます. ここで 'plot(X,Y)' とすると, plot 命令は列ベクトル単位で描くために図 16.6 の様な意味不明のグラフを得る羽目になります:

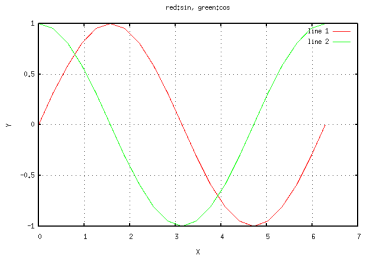


図 16.5: plot(X',Y') のグラフ

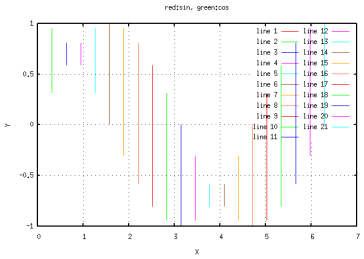


図 16.6: plot(X,Y) のグラフ

16.14 Octave で file を利用する話

この節では Octave でファイルを利用する方法について簡単に説明します. そのため, ここで説明する命令についても, その機能の一部のみを用いるだけで, 全てを活用しているとは限りません.

16.14.1 load 命令によるデータファイルの処理

最も基本的な, 数値行列を含むファイルの読込処理は load 命令で対処できます.
実際に試してみましょう. ここではファイル名を neko にして, 行列データは次の数値にしましょう:

ファイル neko の内容

```
1 \begin{fixedc}
2 1 2 3
3 4 5 6
```

ファイルの読込は単純に load neko の様にファイル名を必要があれば検索経路付きで指定するだけです:

```
octave:1> load neko
octave:2> neko
neko =

   1   2   3

   4   5   6
```

これでファイルの内容の読込みができました. この例で示す様に行列の名前はファイル名が割当てられていますね. では, 続けて同じ名前のファイルを読込んでみましょう. どうなりますか？

```
octave:3> load neko
warning: load: local variable name 'neko' exists.
warning: use 'load -force' to overwrite
error: load: unable to load variable 'neko'
error: evaluating index expression near line 3, column 1
```

色々と文句を言わていますね. Octave では自動的に読込まれたファイル名と同じ名前の変数が既に利用されていると既存のデータを上書きせずに保護します.
そこで,load 命令に-force オプションを付けてみましょう:

```
octave:3> load -force neko
octave:4> neko
neko =
   1   2   3
   4   5   6
```

今度はどうでしょうか. 上書きされていますね. では, 行列名はどの様な規則で付けられているのでしょう. neko と同じ内容のファイルで, neko.matrix という名前のファイルを用意して続けて読込んでみましょう:

```
octave:5>load neko.matrix
warning: load: local variable name 'neko' exists.
warning: use 'load -force' to overwrite
error: load: unable to load variable 'neko'
error: evaluating index expression near line 5, column 1

octave:5> load -force neko.matrix
octave:6> who

*** local user variables:

neko

octave:7>
```

行列名は拡張子を外した名前の部分が使われるので, 変数名は,neko.matrix の場合も neko になります. そして,load 命令に-force オプションが無かったため, 変数の内容保護機能が働いてエラーを返した訳です.

16.14.2 save 命令による行列の保存

数値行列ファイルの読込みができるなら, その逆はどうするのでしょう. 単純な数値行列データのファイルへの保存には save 命令が使えます.save 命令は who 命令を実行して表示される利用者定義データ全ての指定ファイルへの保存や個別のデータ保存の両方が行え, その上, 既存ファイルへのデータの追加も行えます:

```
octave:1> a=rand(4,4);
octave:2> b=rand(3,1);
octave:3> save neko a
octave:4> save test
octave:5> who

*** local user variables:

a  b
```

この例では行列 a,b を各々rand 命令で生成した 4 行 4 列と 3 行 1 列の行列としています. save neko a で, 行列 a をファイル neko に保存し, save test で who で表示されるデータ全てをファイル test に保存しています. ここでファイル neko とファイル test の内容を確認しましょう:

ファイル neko の内容

```
1 # Created by Octave 2.0.16, Thu May 10 08:21:44 2001
2 # name: a
3 # type: matrix
4 # rows: 4
5 # columns: 4
6  0.590789258480072 0.222358718514442 0.876821994781494 0.949454307556152
7  0.741063475608826 0.656238257884979 0.365377485752106 0.979949235916138
```

```
 0.395543217658997  0.417380422353745  0.444111585617065  0.857901215553284
 0.568090081214905  0.558982253074646  0.0379265695810318  0.475694209337234
```

ファイル test の内容

```
# Created by Octave 2.0.16, Thu May 10 08:21:44 2001
# Created by Octave 2.0.16, Thu May 10 08:21:53 2001
# name: a
# type: matrix
# rows: 4
# columns: 4
 0.590789258480072  0.222358718514442  0.876821994781494  0.949454307556152
 0.741063475608826  0.656238257884979  0.365377485752106  0.979949235916138
 0.395543217658997  0.417380422353745  0.444111585617065  0.857901215553284
 0.568090081214905  0.558982253074646  0.0379265695810318  0.475694209337234
# name: b
# type: matrix
# rows: 3
# columns: 1
 0.628571033477783
 0.415022879838943
 0.216913774609566
```

この様に save 命令で変数を指定すると, 変数に割当てられた値がファイルに保存され, 変数を指定しないと, who 命令で表示される変数に割当てられた値がファイルに保存されます.
既存ファイルにデータの追加する場合, save 命令に-append オプションを付けます. たとえば, 上記の test ファイルに行ベクトル c を追加したければ, 次の処理を行います:

```
octave:6> c=[1,2,3];
octave:7> save -append test c
```

では c を追加後のファイル test の内容を確認しておきましょう:

ファイル test の内容

```
# Created by Octave 2.0.16, Thu May 10 08:21:53 2001
# name: a
# type: matrix
# rows: 4
# columns: 4
 0.590789258480072  0.222358718514442  0.876821994781494  0.949454307556152
 0.741063475608826  0.656238257884979  0.365377485752106  0.979949235916138
 0.395543217658997  0.417380422353745  0.444111585617065  0.857901215553284
 0.568090081214905  0.558982253074646  0.0379265695810318  0.475694209337234
# name: b
# type: matrix
# rows: 3
# columns: 1
 0.628571033477783
 0.415022879838943
 0.216913774609566
# name: c
# type: matrix
# rows: 1
# columns: 3
```

```
21  1 2 3
```

ファイルの保存の場合は load 命令と異なり, 指定したファイルが存在していても無警告で処理が実行されます. 実際, test ファイルが存在する状態で test に行列 a と c を保存する例を以下に示しておきます:

```
octave:8> save test
octave:9> save test a c
octave:10>
```

save 命令で保存した行列 a と c を含むファイル test の中身を以下に示します:

ファイル test の内容

```
 1  # Created by Octave 2.0.16, Thu May 10 08:29:54 2001
 2  # name: a
 3  # type: matrix
 4  # rows: 4
 5  # columns: 4
 6   0.590789258480072 0.222358718514442 0.876821994781494 0.949454307556152
 7   0.741063475608826 0.656238257884979 0.365377485752106 0.979949235916138
 8   0.395543217658997 0.417380422353745 0.444111585617065 0.857901215553284
 9   0.568090081214905 0.558982253074646 0.0379265695810318 0.475694209337234
10  # name: c
11  # type: matrix
12  # rows: 1
13  # columns: 3
14   1 2 3
```

そして save 命令で保存したデータは load 命令でそのまま読込めます:

```
octave:1> load test
octave:2> who
*** local user variables:
a  c
octave:3> a
a =
  0.590789  0.222359  0.876822  0.949454
  0.741063  0.656238  0.365377  0.979949
  0.395543  0.417380  0.444112  0.857901
  0.568090  0.558982  0.037927  0.475694
octave:4> c
c =
  1  2  3
octave:5>
```

MATLAB や Octave ではもっと複雑な処理ができます. 両方とも言語的にが C に似ているため,C 風にファイルの操作ができます. この機能を用いれば, 非常に複雑な形式のファイルの読込と書込も可能です. ところが,Octave の方が処理言語の機能が上なので,MATLAB でも利用可能なプログラムを構築しなければならい時には注意が必要になります. ここでは MATLAB との互換性に留意して記述を纏め,Octave 独自の機能は利用しない様にしています.

16.14.3 ファイルの Open と Close

ファイルを開く場合, fopen 命令を用います:

fopne 命令の構文

id	=	fopen(⟨ ファイル名 ⟩,⟨ 指定 ⟩)

先ず,⟨ ファイル名 ⟩ には'neko' や'test/neko.dat' の様なディレクトリを含めたファイル名前を設定します. 次に,fopen 命令の ⟨ 指定 ⟩ は以下のものを用います:

fopen のファイルの読込指定

指定	概要
'r'	読込み
'w'	新規に書込み
'a'	末尾に追加
'r+'	読込み. 内容の更新も可
'w+'	書込み. 内容の更新も可
'a+'	末尾に追加. 内容の更新も可

この fopen 命令は整数値 id を返します. この値は以降のファイル操作で利用し, 操作するファイルの指定に利用します. なお, ファイルが存在しない等のファイルのオープンでエラーを出した場合に-1 が返されるので, この返却値を利用してエラー処理ができます.
ファイルの close は C と同様に fclocse 命令を使って,fclose(id) で指定したファイルを閉じます.
ファイル内部の移動ではポインタを用います. ファイルを開くとポインタはファイルの先頭に置かれていますが, ファイルの読込等を行うと下の方に移動して行きます. そこで, ファイルの先頭にポインタを動かす必要が出て来ると,frewind 命令を使います.
データの読込や書込では fgets,fputs,fscanf 等の命令が存在し, 基本的な処理は C と同様の命令が揃っています. ところが, 実際の機能は C の同名の命令とは微妙に異なるので注意が必要になります. 特に MATLAB との互換性を考慮すると,Octave の機能をフルに活用したプログラムはそのままでは使えず, 修正が必要になってしまいます. これは入力と出力書式の指定で顕著です.MATLAB では基本的に行単位で書式が固定され, 微妙な調整が行えません.
そのため, 一行に整数, 文字列, 浮動小数が混在する場合, 入出力の書式を指定して読込む方式は避け,fgets で一行を stream として取込んで,stream を分解して sscanf で形式の変換を行う事を薦めます. この方法に関しては次の節で説明しましょう.

16.14.4 データの読み込み

ファイルのデータ読込みは fgets,fscanf 等の命令が使えます. なお,Octave の fscanf 命令は上述の様に MATLAB のものと比べ機能が強化されており, その上,C-flag を立てる事によって C 言語の fscanf と同じ利用が可能です. ところが,MATLAB では書式が行毎で数値や文字単位ではないため,Octave 専用のプログラムになってしまうので, MATLAB との互換性を重視したプログラムでは, 文字, 数値が混在する行を扱う場合, fgets 命令を使って一行を stream として取り込み, その stream

を文字列照合や長さで分割したものに対して sscanf を用いて形式の変換を行う方が, 処理は繁雑になるかもしれないが安全です.
以下に単純な実例を示しましょう. ここでの例では単純な数値行列データに日本語を含めた文字列を含むものです：

ファイル neko.txt の内容

```
1 1 2 3
2 3 2 1
3 はい,1 2 3ある日森の中,熊さんに出逢った.
```

このファイルは 1 行と 2 行が数値ですが, 3 行目は数値と文字が混在しています. このファイル ('neko.txt') を open 命令で開き, fgets 命令を用いて一行づつ読込む様子を以下に示しましょう:

```
octave:43> fid=fopen('neko.txt','r');
octave:44> L1=fgets(fid)
L1 = 1 2 3
octave:45> L2=fgets(fid)
L2 = 3 2 1
octave:46> L3=fgets(fid)
L3 = はい,1 2 3ある日森の中,熊さんに出逢った.
octave:47> frewind(fid)
ans = 0
octave:48> L4=fgets(fid)
L4 = 1 2 3
```

この例ではファイルの内容参照を行うために fopen 命令 の型の指定で “r” 指定しています. 一般的には “r” か “r+” でファイルを開きます. ところで, 間違って “w” や “w+” を指定すると, 直ちにファイルが更新されて以前の内容が消去された状態となるので注意が必要です.
fgets 命令は例で示す様に, ファイルの先頭から一行づつ読込みを行います. ここで, ファイルの先頭にポインタを移動させるために frewind 命令を使います.
ここで fgets で返される値の型は実は文字列型です. この例で一見するとベクトルにしか見えない 'L1' は文字列型です. 実際に, 'L1' の和を計算しようとすると以下のエラーが出ます:

```
octave:56> L1+L1
error: invalid conversion from string to real matrix
error: invalid conversion from string to real matrix
error: evaluating assignment expression near line 56, column 3
octave:56> size(L1)
ans =
   1   6
octave:57> L1
L1 = 1 2 3
```

この様にテキストファイルデータを fgets 命令で読込むと, fgets か返す値は全て文字列型になってしまいます. そこで, 必要に応じて型の変換を行わなければなりません. 型の変換は C と同様に sscanf 命令 を用います:

sscanf の型の指定

指定		概要
%d	⇒	整数型データに変換
%f	⇒	浮動小数点型データに変換
%s	⇒	文字列型データに変換

次に, 文字列 L1=1 2 3 と L2=3 2 1 を sscanf 命令を使って型の変換を行った実例を示します:

```
octave:51> a11=sscanf(L1,'%d')
a11 =
  1
  2
  3
octave:52> a12=sscanf(L1,'%s')
a12 = 123
octave:53> a12=sscanf(L1,'%f')
a12 =
  1
  2
  3
octave:54> a11=sscanf(L1,'%d')
a11 =
  1
  2
  3
octave:55> a12=sscanf(L2,'%d')
a12 =
  3
  2
  1
octave:56> a11+a12
ans =
  4
  4
  4
```

なお,'%d' の場合,Octave では自動的に列ベクトルになります. ところが, MATLAB では行ベクトルになります. これは Octave が列ベクトルを基準としている傾向があるためでしょう. この点は Octave と MATLAB の両方で動作するプログラムを作成する際には, 特に注意が必要な個所の一つです.

この様にファイルが文字列か数字列の何れかで構成されている場合,fgets 命令で行毎読んで,sscanf で形式の変換を一度に行えば良い事になります. ところが,L3 の様に数と文字が混合している場合は厄介です.L3 の様に意地の悪い代物は, 悪意を持って例として示しているので仕方がありませんが, 次のファイル tama の様な形式は非常に普通でしょう:

ファイル tama の内容

```
1 # No.  Flag   Value
2 1      t        10
3 2      f       -10
```

```
4 3       t        20
5 4       f       -20
```

この様なデータの場合,sscanf で一気に変換する事は意味が無く, 本来なら書式指定で各々を変換するのが本来の姿です. ところが,MATLAB には与えられたストリームを一気に変換する事しかできません.
先ず, ファイル tama を開き, 安易に一行づつ sscanf で整数型 ('%d') や文字列型 ('%s') へと一気に変換した例を示しましょう:

```
octave:73> fid2=fopen('tama','r')
fid2 = 3
octave:74> tama1=fgets(fid)
tama1 = # No.  Flag   Value

octave:75> tama2=fgets(fid)
tama2 = 1       t        10

octave:76> tama3=fgets(fid)
tama3 = 2       f       -10

octave:77> tama4=fgets(fid)
tama4 = 3       t        20

octave:78> tama5=fgets(fid)
tama5 = 4       f       -20

octave:79> st1=sscanf(tama1,'%d')
st1 = [](0x1)
octave:80> st1=sscanf(tama2,'%d')
st1 = 1
octave:81> st1=sscanf(tama3,'%d')
st1 = 2
octave:82> st1=sscanf(tama4,'%d')
st1 = 3
octave:83> st1=sscanf(tama5,'%d')
st1 = 4
octave:84> st1=sscanf(tama1,'%s')
st1 = #No.FlagValue
octave:85> st1=sscanf(tama2,'%s')
st1 = 1t10
octave:86> st1=sscanf(tama3,'%s')
st1 = 2f-10
octave:87> st1=sscanf(tama4,'%s')
st1 = 3t20
octave:88> st1=sscanf(tama5,'%s')
st1 = 4f-20
```

この様に sscanf による変換では書式に対応しない個所以降の列は除外されます. 例えば,'%d' で整数型に変換する個所を見て頂くと判りますが, 二列目の flag に当たって変換が途中で終了してしまい,flag の前の No. に相当する数のみが返されています. 更に, ファイル先頭行の文字列は変換ができずに空白文字が返されています. その上, ストリームを一気に文字列に変換した場合, 空白文字やタブは省略されています. 実際,2 f -10 が 2f-10 に変換されたりしていますね.

この様に MATLAB の sscanf,fscanf の互換性のある方式ではこれ以上の処理ができません. そこで,Octave で sscanf 命令に C フラグを立てると C と同様に複雑な形式のファイルにも対応できる様になります:

```
octave:94> [s1,s2,s3,s4]=sscanf(tama1,'%s %s %s %s','C')
s1 = #

s2 = No.

s3 = Flag

s4 = Value

octave:95> [n1,flg,n3]=sscanf(tama2,'%d %s %d','C')
n1 = 1

flg = t

n3 = 10
```

この様に Octave であれば, より C 風に sscanf や fscanf を利用できますが, 難点は, この C フラグを立ててしまうと今度は MATLAB では利用できない事です.. そこでスマートではありませんが, 次の様にストリームを分けて対処すれば互換性に問題が生じる事もなく上手に処理が行えます:

```
octave:96> find(tama2=='t' | tama2=='f')
ans = 7
octave:97> n1=sscanf(tama2(1:6),'%d')
n1 = 1
octave:98> n2=sscanf(tama2(8:length(tama2)),'%d')
n2 = 10
octave:99> flg=sscanf(tama2(7),'%s')
flg = t
```

最初に find を用いている個所はストリーム tama2 からフラグ値の t か f のどちらかが存在する個所を求め, その個所から前後に分けて整数に変換しています. ここで, 記号 "|" は Octave/MATLAB の論理和になります.
同様にコメント行かどうかは'#' がストリームに存在するかどうか検証するだけでできてしまいます. だから, 全てを一旦文字列に変換し, 先頭が'#' であるかを判別する様にすれば良い事になります.
なお, 重要な事に Octave と MATLAB の両方では, 扱う行列データは文字か数値のみしか許容されず, 両者が混在した行列を扱う事はできない事です.
この様に行列データに制約があるので, 文字と数値が混在する表の扱いでは, 色々な行列データを準備する必要がある様に思えます.
Octave と MATLAB の双方に C の構造体と同様のデータ構造を用いる事が可能なので, そちらを使うと多くの行列を利用する必要が無くなってしまいます. その上,MATLAB と Octave で構造体を利用する場合は, 他の変数と同様に予め宣言する必要はありません. ここでは実際に Octave で構造体を用いた例を示します:

```
octave:102> [neko.n1(1),neko.flg(1),neko.n2(1)]=sscanf(tama2,'%d %s %d','C')
neko.n1 = 1
```

```
neko.flg = t

neko.n2 = 10

octave:103> [neko.n1(2),neko.flg(2),neko.n2(2)]=sscanf(tama3,'%d %s %d','C')
neko.n1 = 2

neko.flg = f

neko.n2 = -10
```

この例では sscanf で変換したデータを neko.n1,neko.flg,neko.n2 に割当てています. ここで各々の配列は数値と文字列になっている事に注意して下さい.
MATLAB と Octave ではこの様に構造体を用いて文字と数値が混在したデータを一括して扱えます. ここで, データが構造体かどうかは直接データ名を入力する事でも判別可能ですが, この場合, 全てのデータが一度に表示されるので,Octave の場合, is_struct 命令で構造体かどうかを判定し, struct_elements 命令で構造体の構造が調べられます. 但し,MATLAB の場合は命令の名前は似ているが, 全く別の命令を用います. そのため, 互換性に注意が必要です:

```
octave:104> neko
neko =
{
  n2 =

     10

    -10

  flg =

t
f

  n1 =

    1
    2
}

octave:107> is_struct(neko)
ans = 1
octave:108>
octave:109> struct_elements(neko)
ans =

n2
flg
n1
```

この様に MATLAB との互換性を考慮すると, 泥臭い処理が必要になりますが, fgets と sscanf 命令を上手く用いれば, 通常のファイルの処理もできます.

16.14.5 ファイルの更新とデータの追加の例

fopen の指定でファイルの更新, 内容の入れ替えや, データの追加が行えます. ファイルの更新では,'w' を指定します. こうすると, 既存のファイルの内容は消去され, ファイルが存在しない場合は指定された名前のファイルを新規に生成します. もし, 既存のファイルを利用する場合,'a' を指定すれば既存のファイルの末尾に続けて書込めます. ここでは与えられた行列データをフラグに従って指定されたファイルに追加や置換を行うプログラムを示します:

M-file appendDATA の内容

```
function [err]=appendDATA(fname,mv,flg)
        err   = 0;
        % ファイル名の設定.
        vfname=[fname,'.vdt'];
        % フラグflg==0であれば上書き,それ以外は末尾にデータを
        % 追加する.
        if flg==0
                vfp = fopen(vfname,'w');
        else
                vfp = fopen(vfname,'a');
        end;

        % データの書き込み
        [m,n]=size(mv);
        if m>0
                if flg==0
                        fprintf(vfp,' %d ',mv(1,:));
                        fprintf(vfp,'\n');
                end;
                for k=[2:m]
                        fprintf(vfp,'%22.15e',mv(k,:));
                        fprintf(vfp,'\n');
                end;
        else
                err=1;
        end;
        fclose(vfp);
```

この例では, ファイル名は修飾子 ".vdt" を抜いた型で与え, ファイルの先頭行はカラムを分類するために整数としており, 2 行目以降が実際のデータで, 行列 mv もその様な形式となっています. このデータ行の出力並びは fprintf 命令の'%22.15e' で指定したもので, この書式の設定は C や FORTRAN のそれと同じ形式になります. さらに, この処理を"%データの書き込み"と註釈を入れた個所から下で行っています.

以上のように Octave のファイル処理では命令やオプションがほぼ C に似た仕様になっているので MATLAB との互換性を考慮しなければ, C 風に記述して C オプションを立てておけば便利です. さらに扱うデータを数値行列にすれば非常に苦労も少なく, 非常に気楽にファイル操作が行える仕様となっています.

しかし, MATLAB との互換性を考慮すれば, MATLAB で書式指定を伴う処理が行単位で行われるので, 一行に文字と数値が混在データファイルを扱う場合には工夫が必要になります. その上, Octave

の独自の機能に頼ると MATLAB で動作しないプログラムになる可能性が非常に高くなってしまうので注意しましょう.

16.15　Maxima との簡単なインターフェイス作製

この節では簡単な Maxima と Octave のインターフェイスを作製しましょう.
最初に Octave と Maxima の行列の定義方法を確認しておきましょう:

Octave と Maxima の行列の定義を比較

mat1=[1,2,3;4,5,6];	⇔	mat : matrix([1,2,3],[4,5,6]);

両者の違いは, Maxima では行を大括弧 “[ ]” で括るのに対し, Octave ではセミコロン “;” で区切り, Maxima では matrix 函数を用いるのに対し, Octave では直接, 成分を列記し, 割当が Octave では “=”, Maxima はコロン “:” となることです.
ところで, Maxima の行列をリストに変換すると,Octave の行列にもっと似てきます. この場合, `matrix`(...) が [...] になります. そのために行ベクトルの場合, Maxima の行ベクトルと Ocatve のデータは一致します. また, Octave では [[1,2,3],[4,5,6]] は [1,2,3,4,5,6] の様に一つの行ベクトルとして評価されます. このことから, Maxima から Octave に変換するために, Maxima 上で行列を配列に変換し, その行列の大きさを示すデータと一緒に Octave の m-file として記述しておけば, あとは Octave で m-file 名を実行すれば行列が行ベクトルとして定義され, それを本来の行列となるようにプログラムも起動するようにしてしまいましょう.
まず, Maxima のプログラムを示します.

Maxima のプログラム

```
1  SIZEofMAT(mat):=
2  block(
3          [ans:false],
4           if matrixp(mat) then
5             ans:map(length,map(lambda([x],substpart("[",x,0)),
6                                [col(mat,1),col(transpose(mat),1)]))
7           else
8             false,
9           return(ans)
10 )$
11
12 m2oct(filename,x):=
13 block(
14        [size,tmp,tmp1],
15        if matrixp(x) then
16          (
17           tmp:substpart("[",x,0),
18           size:SIZEofMAT(x),
19           tmp:[filename,'maximat_size=size,'maximat=tmp],
20           apply(stringout,tmp)
21          )
22         else
23          print("Please enter MATRIX data!")
```

```
24 )$
```

このプログラムは単純に, 計算した行列の大きさを変数 maximatrix_size に入れ, 行列は一つの行ベクトルを生成する Octave の M-file を生成するものです.
これに対して Octave の Maxima で生成した行ベクトルを行列に変換する変換プログラムと, Octave で生成した行列を Maxima の命令ファイルに変換するプログラムを示します.

M-file m2octmat の内容

```
1  function [z] = m2octmat(maximat,maximat_size)
2   m=maximat_size(1);
3   n=maximat_size(2);
4   for i=[1:m]
5      z(i,[1:n])=maximat([1:n]+(i-1)*n);
6   end;
7  end;
8
9  function []= oct2maximat(fname,mat)
10  [m,n]=size(mat);
11  id=fopen(fname,'w');
12  fprintf(id,'%s\n',"maximat:matrix(\n");
13  for i=[1:m]
14       fprintf(id,'%s%22.15e',"[",mat(i,1));
15       if n>1
16          fprintf(id,', %22.15e',mat(i,[2:n]));
17       end;
18       if i!=m
19          fprintf(id,'%s\n',"],");
20       else
21          fprintf(id,'%s\n',"]");
22       end;
23  end;
24  fprintf(id,'%s\n',");");
25  fclose(id);
26 end;
```

これらのプログラムは非常に初歩的なものですが最低限のことができます.

第17章　Maximaを動作させる環境について

17.1　道標

この章では, Linux 版や MS-Windows 版の Maxima をインストールする方法, KNOPPIX/Math を仮想計算機環境を用いて KNOPPIX/Math を立ち上げる方法, について簡単に述べます.
ここで, 最初の FAQ で述べたように, この本では, a. 我慢強い方, b. 軟派な方, c.windows で済ませたい方の三種類に読者の皆さんを分類しています. そこで, 次に道標を示しておきましょう:

a. の我慢強い方:　Maxima に対応した Common LISP をインストールした上で Maxima のコンパイルしてみましょう. §17.3 に Maxima のコンパイルとインストールの概要を解説しています.

b. の軟派な方:　とにかく KNOPPIX/Math を入手しましょう. KNOPPIX/Math には三種類存在します. 一つはプレス版と呼ばれるオープンソースカンファレンスや日本数学会といったイベントで KNOPPIX/Math Project が配布している DVD です. この DVD は再配布に制約のあるパッケージを含んでいるために, 複製の再配布は御遠慮下さい.
もう一つはダウンロード版で, こちらの実体は ISO イメージファイルです.
http://www.knoppix-math.org/wiki/index.php?KNOPPIX/Math/Download から入手できますが, プレス版から再配布に問題のあるパッケージを除外したもので, こちらの複製の再配布はプレス版と異なり問題がありません.
最後は書籍に付属の KNOPPIX/Math です. たとえば, この文書の元の書籍の「はじめての Maxima」には CD-ROM が付属しています. こちらは KNOPPIX/Math 2006 と古い版です. それに対し,「数値処理・画像処理ソフト Yorick」,「Octave の精義」や「理工 PC 初心者のための KNOPPIX 活用法」には KNOPPIX/Math 2010 が付属しています[1]
プレス版や書籍に付属の KNOPPIX/Math で遊ぶ場合には, PC の boot ディバイスの順位の設定を予め行って下さい. この設定は PC によって異るので, マニュアルを参照して下さい. また, PC が比較的新しいものであればハードウェアの認識が上手く行かないこともあります. その場合には仮想計算機の利用も視野に入れるとよいでしょう.
ダウンロード版は ISO イメージとなるので, 通常の利用を考えているのであれば, DVD に焼く必要があります. とは言え, 仮想計算機を利用するのであれば, DVD に焼く必要もなく, 仮想計算機の DVD/CD ドライブとして ISO イメージを割り当てておけば容易に遊べます. この詳細は §18.2 を参照して下さい.

[1] Yorick 本の KNOPPIX/Math は商用利用で問題のないパッケージのみを収録した KNOPPIX/Math 2010 です.

c. の MS-Windows で全てを済ませたい方と右も左もおぼつかない初心者の方: §17.4 でインストール方法を解説しているので, そちらを参考にして下さい.

17.2　Maxima の初期設定

ここで Maxima の初期設定の方法について簡単に解説しておきましょう. Maxima では maxima-init.mac という名前のファイルを所定の場所に置くことで初期設定が行えます. この maxima-init.mac の書式は通常の Maxima の入力文です. だから, 予め変数に値を設定したり, load 函数を使うことで, 自分専用のプログラムを読込せることが簡単にできます. この応用として, Maxima の修正・改良は src ディレクトリから取出したソースファイルを適当なディレクトリに置いて修正し, それを maxima-init.mac で読込むようにするだけでできてしまいます.
さて, 問題となるのは, このファイルの置き場所です. GUI ではなく, 仮想端末や DOS 窓から立ち上げる場合, その立ち上げる時点でのディレクトリ上に置かれた maxima-init.mac が参照されます. OS の GUI を用いる場合, 例えば, OS のメニュー等から起動するとき, UNIX 系の OS ならば各利用者のホームディレクトリ上の maxima-init.mac が参照されますが, MS-Windows の場合はまちまちです. 基本的に wxMaxima や xMaxima 等のフロントエンドが置かれているフォルダ内の maxima-init.mac の内容が反映されます. 詳細は §17.4.2 を参照して下さい.

17.3　我慢強い方

ここで述べるのは, 訳があって KNOPPIX/Math が使えない/使いたくない場合, Maxima を愛するがあまりに改造に燃える方向けです. 当然, rpm や deb といったパッケージ管理システムを用いてインストールする話は除外します. あくまでも古来からの ‘configure ⇒ make’ という手順で説明します.

17.3.1　Common Lisp の選択

さて, ここで KNOPPIX/Math が使えない/使いたくない理由として, PC の環境的な問題に加え, KNOPPIX/Math が現時点では 32 bit 環境だという事実があるでしょう. また, KNOPPIX/Math の Maxima は GCL を基盤としていますが, この GCL よりも高速な処理が行える Common LISP もちゃんと存在しています. このような事情を考えて予め計算機環境を整えておく必要があります.
ここで無課金で利用可能な Common Lisp で代表的なものとして GCL, CLISP, CMUCL, SBCL, ACL, ECL, OpenMCL 等があります. なお, 利用可能な Common Lisp を調べたければ, Maxima のソースファイルに含まれる configure を ‘–help’ オプションで起動させて表示されるヘルプの内容で, “–enable” オプションの解説の個所で色々と出てきます.
非常に大雑把な選択としては, 処理速度を重視するのであれば SBCL, share ライブラリ等の互換性を重視するのであれば GCL, 幅広い環境で利用したけれは CLISP といったところでしょうか.

ちなみに, これらの LISP を処理速度順で並べると, SBCL>GCL>CLISP となる様です. ここで CLISP の処理速度は今一つですが, CLISP は様々な環境に移植され, その上, 安定しているので, 複数の環境で利用する場合は CLISP で統一しておくとあとの処理が楽な面があります.
ここで, Maxima と Common Lisp のバージョンには注意して下さい. たとえば, 古い CLISP では新しい Maxima が上手く動作しないことがよく発生しています. この場合, LISP の版を変更するか別の LISP でコンパイルを行うことになるでしょう. 猛者は勿論, Maxima を作り替えても構いません. その場合, Maxima のソースファイルは src ディレクトリに含まれているので, 問題の個所を一つ一つ潰して行くことになります.

17.3.2 コンパイルの手順

現在の Maxima のコンパイルは非常に簡単になっています. 基本的には, 入手した書庫ファイルを展開, 附属の configure で利用する LISP やインストール先等の諸設定を行い, make でコンパイルを行い, make install でインストールを行うという手順になります.

書庫ファイルの展開: 適当なディレクトリ上で Maxima のソースファイルの書庫ファイルを展開します. 書庫ファイルには gzip や bzip2 で圧縮したものがありますが, 新しい tar 命令を使えば, `tar -xvf 書庫ファイル` で展開できます.

configure の実行: 展開してできた Maxima のディレクトリに移動し, `./configure` を実行します. 通常はこれだけで済みますが, 環境によっては細かい設定を行いたい場合もあるでしょう. たとえば, CLISP や GCL 等の Common Lisp が複数存在する場合に, コンパイルに用いる Common Lisp を設定する必要もあるでしょう. このときに設定可能なオプションは `./configure –help` を実行すれば表示されます. ここで configure が出力する情報は多いので, `./configure –help—less` の様に less を併用すると良いでしょう.
configure による設定で個人の環境に合せるため, インストール先のディレクトリを指定したい方が多いのではないかと思います. この場合は–prefix オプションを使います. 具体的には, –prefix=”/usr/share” の様にディレクトリを直接指定します. 何も指定しなかった場合, /usr/local/share/maxima に Maxima のバージョンに対応するディレクトリが生成されます. たとえば, 2011 年 1 月の最新版は 5.23.0 なので, インストール先の既定値は/usr/local/share/maxima/5.23.0/ になります.

コンパイルとインストール: コンパイルは `make` を実行します. make が無事に完了すると `make test` によって, 出来上った Maxima に問題がないことを確認し, `make install` で Maxima のインストールを行います. このインストールでは configure の ‘–prefix’ オプションで指定したディレクトリの下に maxima ディレクトリが生成され, このディレクトリの下にバージョンに対応したディレクトリが生成され, 以下のディレクトリを包含します:

- demo ディレクトリ:デモファイルを格納
- doc ディレクトリ:マニュアル等の文書を格納

- msgs ディレクトリ:ru.msg を格納
- share ディレクトリ:Maxima のライブラリを含む
- src ディレクトリ:ソースファイルを格納
- tests ディレクトリ:テスト用ファイルを格納
- emacs ディレクトリ:emacs をフロントエンドとして利用する為のファイルを格納
- xmaxima ディレクトリ:xmaxima 関連のファイルを格納

このディレクトリ構成は MS-Windows 版でもほぼ同様です.

Maxima の (魔) 改造:　Maxima のソースを変更してコンパイルとインストールを行う通常の方法に加え, より手軽な方法として, src ディレクトリに含まれるソースファイルを適当なディレクトリに複製を取り, そのファイルを修正して Maxima から load 函数で読込む方法があります. 勿論, コンパイルを行った方が処理は速いのですが, 一寸した修正を行う程度であれば, この方法の方が大局的な影響が皆無なために安全で, 効果の確認が容易であるという効率の良さといった長所があります.
ちなみに, src ディレクトリに含まれるファイルは全て LISP のプログラムです. したがって, プログラムの修正は LISP のプログラムの修正となります. だからといっても, LISP 通である必要がありません. 適当に LISP のプログラムに print 函数を埋め込んで, S 式がどの様に書換えられるかを観察し, それから自分の望む処理を行う様に修正する方法であれば, とても気楽に修正ができてしまいます.
そして, 修正・改良したプログラムを Maxima の立ち上げ時に自動的に反映させるのであれば, maxima-init.mac ファイルに load 函数を用いて読込むようにすれば十分です. この load 函数では, Maxima 言語のファイルでも LISP のファイルでも読込が行えます.

17.4　MS-Windows 環境への Maxima のインストール

17.4.1　Maxima のインストール

Maxima のインストールは基本的にライセンスに同意するかどうかチェックを入れる個所を除くとあとはそのまま Next> ボタンを押し続けるだけでインストールが行えます. ただし, 意味も判らずに Next を押し続けても面白くはないので, ここでは詳細を述べることにします.
では, デスクトップ上の Maxima のインストーラをマウスでダブルクリックして立ち上げましょう. すると以下の図 17.1 に示すウィンドウが出て来る筈です:
右下に二つボタンが並んでいますが, 今回は Maxima のインストールを行うので, ここでは当然 Next> を押します.
すると, 図 17.2 に示す表示に切替わります. これは Maxima が GPL に従うソウフトウエアであり, そのライセンスに従うかどうかを尋ねるものです:

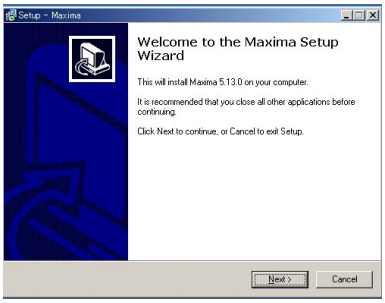


図 17.1: maxima-5.13.0a.exe を実行して出てきたウィンドウ

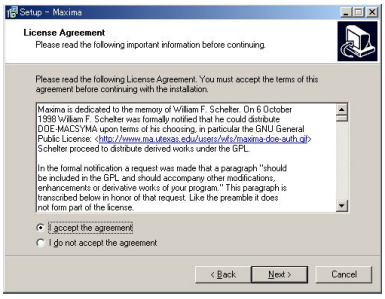


図 17.2: ライセンスに同意するかな？

このライセンス条項に違反する心配は通常の利用で皆無なので, 迷わずに
— acept the agreement のラジオボタンにチェックを入れ, 図 17.2 の状態にします. なお, 同意しない限り, 先には進めません.
次に進むと図 17.3 の表示に切替わります. ここでも迷わずに Next> を押します:

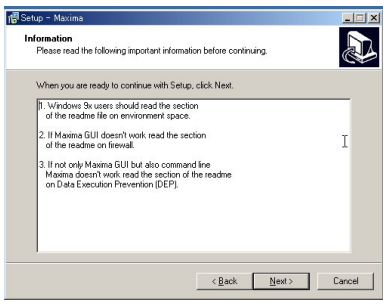


図 17.3: インストールに関する注意書き

ちなみに, この図 17.3 で記述されている事柄は, 第一に MS-Windows 9x 利用者への注意事項, 第二に Maxima が動作しない場合は readme の中の Firewall の項目を参照すること, 最後に GUI の Maxima だけではなく, コマンドラインの Maxima さえも動作しない場合, readme の DEP の個所を参照することといった注意書です.
最初の MS-Windows 9x の注意事項は, Maxima を MS-Windows 9x で動作させて, “Out of environment space” というエラーメッセージが出たときの対処方法です. この対処方法は附属の readme ファイルにあります.
第二の Maxima が動作しないという問題は, MS-Windows に入れている firewall ソフト (anti-virus ソフトも含まれます) によってインストールができなかったり, インストールができていても Maxima 本体とフロントエンド間の通信が阻害されるために結果が返って来ない現象です. まず, 注意することとして, Maxima の GUI 環境は Maxima そのものではなく, 独立したアプリケーションであることです. MS-Windows 版には現在, wxMaxima と XMaxima の二つの Maxima 専用のフロントエンドがあります. これらのフロントエンドは利用者が Maxima の処理文を書込むと, それらを Maxima に送り込んで結果を Maxima から送り返してもらってから結果の表示を行っています. この Maxima への通信が Anti-virus アプリケーションや Firewall アプリケーションで阻害されてしまうと, いつになっても結果が表示されなかったり, フロントエンド側に「Timeout した」といったエラーが出るのです. この対処方法は貴方の anti-virus アプリケーションや firewall アプリケーションの設定に依存するので, 具体的な設定方法を述べることができませんが, 重要なことは, Anti-Virus ソフトや firewall ソフトが Maxima の GUI アプリケーション (wxMaxima や XMaxima) の通信を阻害しないように設定すれば良いのです
そして, 最後の注意書きは MS-Windows の DEP(Data Exception Prevention) の対策です. 多分, この問題は Maxima の下層にある LISP で生じることで, バイナリ配布の Maxima では生じないのではないかと思います.
では, Next> ボタンを押して次に移りましょう. すると図 17.4 に示すインストール先の設定に移動します:

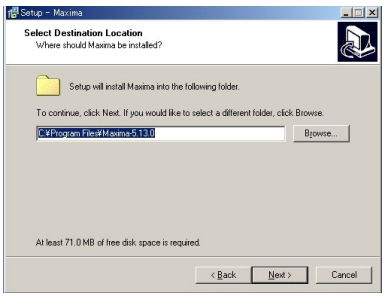


図 17.4: インストール先の指定

通常は `C:\Program File` の下に, Maxima の版に対応するフォルダ, たとえば,Maxima-5-16.3 のような名前, にインストールされます. なお, 初期化ファイル maxima-init.mac を利用したい方は, このフォルダを置いた場所のことを良く憶えておきましょう. そして Next> を押しましょう.
今度は図 17.5 に示す様にパッケージの指定が行えます:

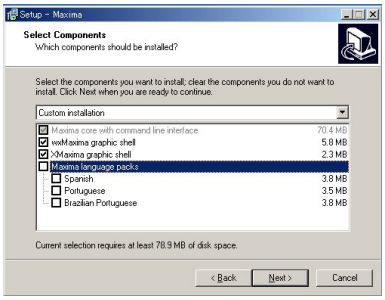


図 17.5: パッケージの指定

なお, Maxima の Language Pack には日本語は残念ながらありません. スペイン語やポルトガル語で使う必要がない限り, このパッケージを入れておく必要性はないでしょう. 勿論, そのままでも構いませんが, 不要ならチェックの入った箱を押せばチェックが外れます.
では, Next> を押しましょう.
今度は図 17.6 に示す様にスタートメニューに登録するフォルダ名の指定が行えます:
これは MS-Windows2000, あるいは MS-Windows XP で Classic な表示を選択した際に, スタートメニューを押して出て来るアプリケーションに含まれているフォルダ名を指定するものです. これも変更しても構いませんし, そのままでも問題はありません.
色々出てきて疲れましたか？ もう少しです. Next> を押しましょう.
今度は図 17.7 に示す表示になります. これはデスクトップに表示する Maxima のフロントエンドのアイコンを選択するものです:
ここで MS-Windows 版の Maxima には wxMaxima と XMaxima の二つのフロントエンドが付属しています. 初心者の方には wxMaxima の方が使い易いと思いますが, デフォルトでなので, そのままで十分でしょう. GUI フロントエンドを選択したら Next> を押しましょう.

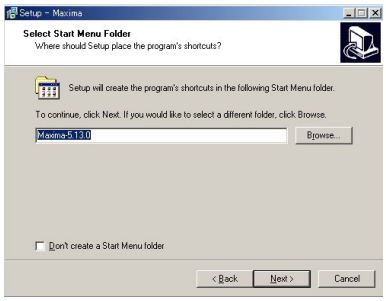


図 17.6: スタートメニューフォルダ名の指定

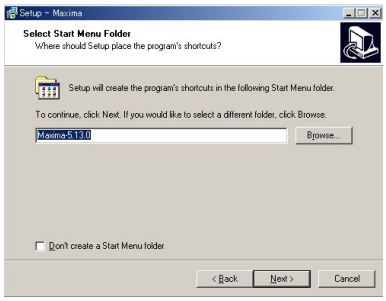


図 17.7: スタートメニューフォルダ名の指定

すると図 17.8 に示す表示になります:

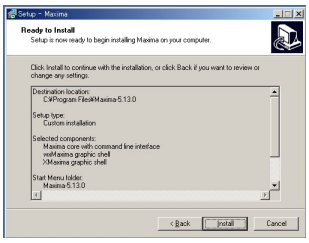


図 17.8: インストール内容の確認

これは今までの設定を纏めたものです. 問題があれば, <Back を押して後に戻って設定し直しても構いません.
お疲れ様でした！ ここで Install を押すといよいよ図 17.9 に示すようにインストールを開始します:

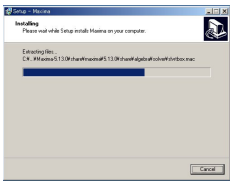


図 17.9: 只今インストール中

この作業も暫く時間がかかるので, ここでちょっと一服しましょう.
さて, インストールが終了すると図 17.10 に示すように readme の内容が表示されます:

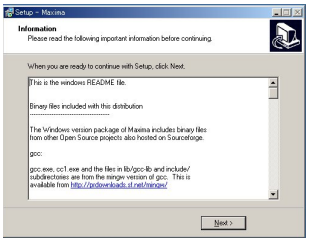


図 17.10: readme を表示中

ここで表示されているファイルは readme_en ファイルで, フォルダの指定がデフォルトのままであれば, `C:\Program Files\Maxima-x-xx-x` の直下にある筈です. 今度も Next> を押しましょう.
すると, 図 17.11 の画面が出ます. ここで Finish を押すとインストール作業は完了です.
これでインストール作業は終りです. お疲れさま！

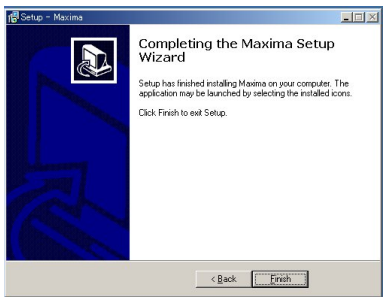


図 17.11: インストール終了のお知らせ

17.4.2　起動時の注意

インストールも終ったので, ここでを終えても良いのですが, Maxima で遊ぶために必要な事柄について, もう少しお話しておきます.

Firewall のこと

wxMaxima や xMaxima といったフロントエンドを利用する方は, Firewall や Anti-virus ソフトがこれらのフロントエンドと Maxima 本体との通信を妨げないように設定をして下さい. また wxMaxima や xMaxima に 1+1 のような非常に簡単式を入れても一行に結果が表示されない場合には, このことを第一に疑って下さい.
初心者で右も左も分らないと自認する方であれば, MS-Windows XP 以降の環境で wxMaxima や xMaxima を立上げたときにシステムから通信を許可するかどうか尋ねられたら必ず許可するようにして下さい.

Maxima の初期化ファイルのこと

カタマイズに関連することですが, MS-Windows 上でも Maxima の初期化ファイル maxima-init.mac が利用できます. この初期化ファイルには Maxima の通常の文を書込むと Maxima が立ち上がるときに, このファイルの内容を自動的に実行します.
そのために日常的に用いるパッケージや函数の読込, 定数の設定等に使えます. ここで MS-Windows 環境の場合はマウス操作による立ち上げもあるので, この初期化ファイルの置場所に注意する必要があります. ここでは以下の各種の場合に分けて解説しておきましょう:

wxMaxima を利用する場合:　wxMaxima をスタートメニューやデスクトップから立ち上げる場合, 初期化ファイルの置場を wxMaxima 本体と同じ階層のフォルダ, すなわち, wxMaxima フォルダにします. デフォルトで Maxima のフォルダは `C:\Program Files` に置かれますが, その Maxima のフォルダの中に wxMaxima のフォルダがあります.
具体的には, Maxima-5.17.0 の wxMaxima のフォルダはデフォルトで
`C:\Program Files\Maxima-5.17.0\wxMaxima` にあります. もし, wxMaxima を使っていて, グラ

フ表示のグラフ出力を画像ファイルに変更した場合は, このフォルダの中に画像ファイルが生成されます.
ただし, 特定のフォルダに移動して DOS 窓から立ち上げる場合は事情が異なります. 基本的には立ち上げたフォルダの中に初期化ファイルが必要になります.

スタートメニュから XMaxima や Maxima を選ぶ場合: まず, XMaxima や Maxima をスタートメニューから立ち上げる場合は初期化ファイルを `C:\Docment and Settings\`以下にある利用者個別のフォルダに置きます. たとえば,PC のログイン名が ponpoko であれば, ディスク C の Docment and Settings フォルダの中にある ponpoko フォルダの中に置きます.

DOS 窓から立ち上げる場合: この場合は Maxima を何処で立ち上げるかに依存します. 逆に言えば,Maxima を立ち上げる直前に移動したフォルダに初期化ファイルを置いておけば良いのです. たとえば,DOS 窓で `D:\Mike\Neko` に移動して, そこで Maxima や wxMaxima を立ち上げるのであれば, `D:\Mike\Neko` に maxima-init.mac を置けば良いのです.

17.4.3 環境変数 Path の設定

ここで述べることはデスクトップにある Maxima のアイコンをダブルクリックして遊んでいる方には不要なので読まなくても構いません. ちょっとしたシステムの改善が関係します.
コマンドプロンプトから Maxima を立ち上げる場合, 環境変数 Path の設定を行っていないと上手くできないことがあります. そのために何処でも Maxima を呼出せるように MS-Windows のシステムの環境変数である Path の設定を行う必要あ生じます.
ここで環境変数 Path を簡単に説明しておきましょう. MS-Windows ではアイコンをクリックしてアプリケーションを立ち上げたりしますね. これはリンクと呼ばれる手法で, 要するに飼犬を名札付きの紐で繋いたようなものです. アイコンをクリックする操作は名札付きの紐を引くことで譬えられるでしょう. ところで, コマンドプロンプトから操作する場合はアイコンをクリックする手段とは異なります. そこで, 計算機にアプリケーションを探させて立ち上げさせるのです. やり方は環境変数 Path に予めアプリケーションの在処を記録しておけば, アプリケーションの呼出を受けた計算機が Path に登録されたフォルダを探して該当するアプリケーションを立ち上げるという方法です. 逆に言えば, この環境変数 Path に含まれていなければ計算機はアプリケーションを見付けられないので, 立ち上げられないことになります.
さて, このシステム環境変数 Path の設定は, コントロールパネルのシステムから設定ができます. このときに注意することは, 既に値が設定されているので迂闊にその内容を消さないように Maxima の所在を追加することです.
さて, Maxima をここでは `C:\Program Files\Maxima-5.17.0` にインストールしていたとしましょう. このときに Maxima の実行ファイルは Maxima のフォルダの下の bin フォルダに置かれています. そのために環境変数に追加する値は
`C:\Programu Files\Maxima-5.17.0\bin` となります. ここで環境変数 Path はこのような表記をセミコロン”;” で繋いだ文字列で構成されます. そのために上記の文字列の前にセミコロン”;”を置いて先程の文字列を追加すれば良いのです.

これで何が良いかと言うと, バッチ処理と呼ばれる自動処理が容易になるからです. バッチ処理と呼ばれる処理は, 処理手順を予めファイルに書き下しておき, それを計算機に処理させる方法です. こうしておくと, 定期的に作業を計算機に自動実行させることや, 手間を掛けずに定型的な処理を実行させる事が容易になるのです. 実際, 定期的な自動実行をさせる場合, 計算機にアイコンをマウスでダブルクリックさせるのは現時点では現実的ではないので, 別のアプリケーションソフトから起動させる手順になるでしょう. このときに Path が明確でなければ, Maxima を立ち上げられません.

第18章　KNOPPIX/Math 2010の活用

18.1　はじめに

ここでは KNOPPIX/Math を利用するための仮想計算機環境の構築と, KNOPPIX/Math の活用について簡単に触れておきましょう.

18.2　仮想計算機環境について

現在の X86 の環境では Multi-core 化が進んでいます. これは CPU の高周波数化とは別の方向の計算機の高速化技術で, 計算機の CPU パッケージ内部に CPU のコアと呼ばれる部分を複数実装させることで一つのチップ上に複数の計算機を実装する方法です. これは CPU の動作クロックの高周波化に伴って周辺回路に与えるノイズの影響が大きな問題として顕在化したことや CPU の排熱の問題といったさまざまな問題によって CPU の動作周波数の高速化が停滞したことが一つの原因です. この Multi-core 化によって並列処理を行わせることで周波数を無理に向上させなくても, プログラムを multi-thread 化することで処理性能の向上が図れることとなりますが, 並列処理に適したプログラムにしておかなければ有難味はさほどありません. だからといっても, 何時もフルで CPU を利用する訳でもありません. そこで遊休気味のコアを独立した計算機に仕立てて別の処理をさせる手段があり, この手段がここで解説する仮想計算機です. この仮想計算機の歴史は古く, 大型計算機で旧機種との互換性を高めるために用いられていました. 現在の仮想計算機環境は CPU の Multi-core 化によって計算機の能力に余裕があることに加え, 仮想化支援機能として Intel VT(VT-x, VT-i, VT-d) や AMD-V といった機能も CPU に実装されたこともあって実用的な水準になりました. ここで実用的に使える仮想計算機の総数はコア数の 2 倍程度と言われます. したがって, 現在の計算機は携帯電話も含めて仮想計算機を一つ運営するだけのハードウエア環境は最初から揃っている状況にありますが, このときに本質的に問題となるのは実装メモリの大きさです.

仮想計化には大きく分けて二種類の仮想化があります. 一つは「**完全仮想化**」と呼ばれるもので, ハードウエア側の支援を受けて完全に独立した計算機として扱う手法です. この手法が最も自由度が高いものですが, 土台の OS の上に別の OS がそのまま載るために I/O が土台と別 OS の二つ存在するので, 専用のドライバがなければ, この I/O のオーバーヘッドが生じて処理の低下が発生し易くなります. そして, もう一つが「**準仮想化**」と呼ばれる手法で, 完全に独立した計算機として扱わずに I/O をある程度共通化して用いる手法です. この手法は完全仮想化と比較して自由な構成は行えませんが I/O が共通化されるので逆に処理速度全般の向上が望めます. そのために, 均質的な仮想計算機ではむしろこちらが向いています.

現在, Linux 上の仮想化環境として「**KVM**」や「**Xen**」が広く用いられていますが, 単純に仮想計算機環境の長所だけを気楽に使いたければ, アプリケーションとして仮想計算機を利用する方か何かと楽です. そこで気楽に使えるツールとして **VirtualBox** と **VMware Player** を順番に紹介しましょう.

18.3　VirtualBox で KNOPPIX を利用する場合

18.3.1　VirtualBox の概要

「**VirtualBox**」は Innotek GmbH が作成した仮想計算機環境で, Sun Microsystems Inc. に買収されてから Sun xVM VirtualBox, その Sun が Oracle に買収されて, 現在は Oracle VM VirtualBox がその正式名称となっています. VirtualBox の入手は http://www.virtualbox.org/wiki/Downloads から可能で, Open Source 版 (OSE と略記) と商用版の二種類があります. ここで OSE は GPL version 2 に基いてソースコードのみが配布[1] され, 商用版のバイナリには x86 と x64 環境の MS-Windows, LINUX [2] と OpenSolaris, Intel 版の MacOS X 環境に対応したものがあり, 個人的利用と教育的利用, あるいは評価目的の利用であれば無償で利用できます. ここでは VirtualBox が既にインストールされていると仮定して解説を行います.

18.4　設定方法

VirtualBox 上での仮想計算機の生成と設定について手順を追って説明してましょう. VirtualBox を立ち上げると図 18.1 に示すウィンドウが現れます:

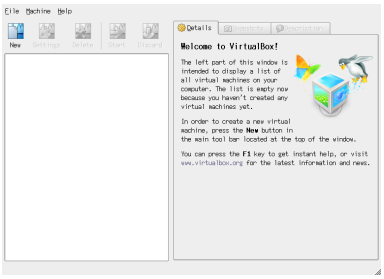


図 18.1: VirtualBox のウィンドウ

このウィンドウのメニュー群の下にアイコンが幾つか並んでいますが, こでらのアイコンを押して現われる Wizard に沿って処理を進めます. ここで仮想計算機の生成では図 18.1 の左上に並んだアイコンの

を押して図 18.2 に示す Wizard を起動させます:

[1] バイナリの OSE がパッケージ化されたディストリビューションもあります.

[2] Debian, Fedora, Mandriva, OpenSolaris, openSUSE, Ubuntu, RedHat Enterprize, openSUSE といった各 LINUX ディストリビューション向けと LINUX 環境全般向けがあります.

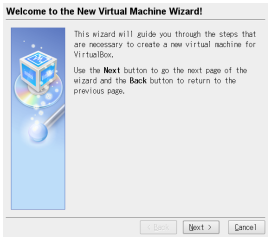


図 18.2: 仮想計算機生成 Wizard

それから 18.2 の右下の 次へ (N)¿ ボタンを押して図 18.3 の画面に進みます:

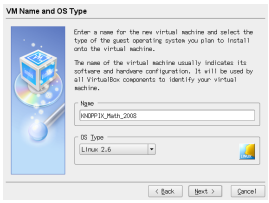


図 18.3: 仮想計算機の名称と OS の設定

ここでは仮想計算機の名称と OS を選択してウィンドウ右下の 次へ (N)¿ ボタンを押せば図 18.4 に移って仮想計算機の記憶容量の設定が行えます:

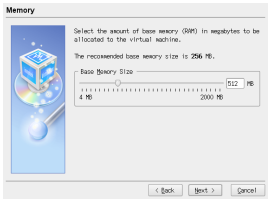


図 18.4: 仮想計算機の記憶容量の設定

記憶容量は KNOPPIX/Math 向けには最低で 128MB, KDE のような豪華なデスクトップ環境を使いたければ最低 256MB を必要としますが, KNOPPIX は計算機のメモリや書込みの領域一式をここでの記憶容量で賄うためにできるだけ多く設定すると良いでしょう. 記憶容量を指定すると今度はウィンドウ右下の 次へ (N)¿ ボタンを押して図 18.5 に示す仮想ハードディスクの設定に移ります:

図 18.5: 仮想計算機のハードディスクの設定

ここで仮想計算機が利用するハードディスクの設定を行います．もし KNOPPIX だけを起動させるのであれば仮想計算機に割当てられる記憶容量と KNOPPIX の DVD/CD-ROM を読込むための DVD/CD-ROM ドライブか DVD/CD-ROM の ISO イメージファイルのみで十分ですが, 仮想ハードディスクがあれば処理結果や環境等の保存が行えます．ハードディスクが不要であれば「起動ディスク (プライマリマスター)(D)」のチェックを外して 次へ N¿ を押せば図 18.6 のウィンドウが出ます:

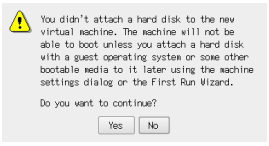


図 18.6: ハードディスクイメージを生成しない場合のメッセージ

この警告の内容は特に気にする必要はありませんが, 続ける を押せば図 18.7 のウィンドウに切り替り, ここで 完了 F を押せば仮想計算機の生成が終了します:

図 18.7: 仮想計算機のハードディスクなしの環境

仮想ハードディスクを新規に作成するのであれば図 18.5 の 次へ (N)¿ ボタンを押して図 18.8 に示す仮想ハードディスク生成の Wizard に移動します:

図 18.8: 仮想ハードディスク生成の Wizard

このウィンドウで 次へ (N)¿ ボタンを押すと今度は図 18.9 に移動してハードディスクのイメージファイルの種類を指定します. 一番上の「可変サイズのストレージ (D)」を指定しておけば必要な領域のみを確保するので無駄に膨れ上がることがありません:

図 18.9: 仮想ハードディスクイメージの型を指定

イメージファイルの型を指定すると 次へ (N)¿ ボタンを押しましょう. すると図 18.10 に示すハードディスクのイメージファイルの大きさに指定に移動します:

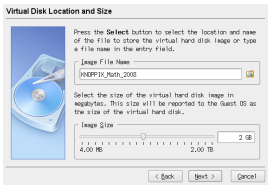


図 18.10: 仮想ハードディスクイメージファイルとその大きさを指定

この設定のあとに 次へ N¿ ボタンを押しましょう. すると, これから生成する仮想計算機の概要が図 18.11 に示すように表示されます:

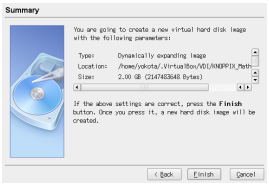


図 18.11: ハードディスクイメージファイルの概要

この設定で良ければ 完了 (F) を押して図 18.12 に切替えます:

図 18.12: 仮想計算機の概要

これで仮想計算機のハードディスクのイメージファイルが指定されました. これで良ければ 完了 (N) ボタンを押しましょう. すると図 18.13 に示すウィンドウに移りますが, このウィンドウの右側には生成した仮想計算機の概要が表示されています:

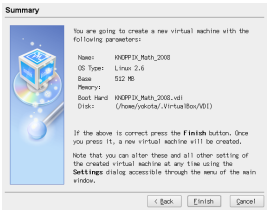


図 18.13: 生成した仮想計算機の概要

今度は仮想計算機の周辺機器の設定を行いましょう. ここで左側のリストから仮想計算機を選択すれば対応する仮想計算機の概要が右側のウィンドウに表示されているので, そこから「CD/DVD-ROM」の箇所をクリックします. すると図 18.14 の画面に切り替わります:

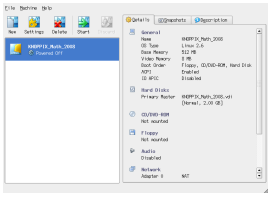


図 18.14: 仮想計算機の CD/DVD-ROM の設定画面

ここで KNOPPIX/Math は CD-ROM/DVD-ROM, あるいは ISO イメージファイルとして提供されます. ISO イメージファイルを用いる利点は CD/DVD-ドライブを用いるよりも読込速度が格段に速いこととネット経由では ISO イメージファイルとして配布されているので入手した ISO イメージファイルをわざわざ CD や DVD に焼かなくて済むことが挙げられます. ISO イメージファイルを利用するのであれば「ISO イメージファイル」のラジオボタンにチェックを入れて, その右側の アイコンを押せば図 18.15 の画面に移動します:

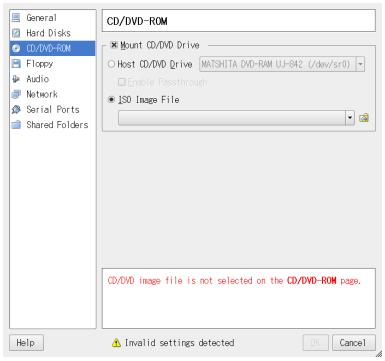


図 18.15: 仮想計算機の CD/DVD-ROM のイメージファイルを選択

このウィンドウの上に並んだアイコンを押せば仮想計算機のハードディスクやフロッピー等のイメージファイルの生成, 指定, 開放や削除が行えます. ここで左上にある Add アイコンを押せば仮想計算機にメイディアのイメージファイルが設定できます. これで良ければ 選択 (S) ボタンを押して図 18.16 に戻ります:

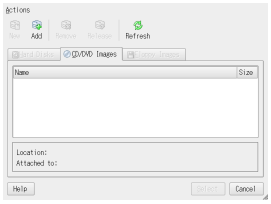


図 18.16: 仮想計算機の CD/DVD-ROM 設定画面

そこで Ok を押しましょう. すると図 18.17 に示す内容に切り替わっている筈です:

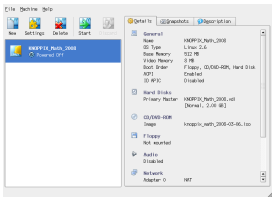


図 18.17: 仮想計算機の設定内容 (変更後)

これで準備が完了しました. 仮想計算機の起動は図 18.17 の左側のリストから仮想計算機を指定し, それから左上の

アイコンを押すと指定した仮想計算機が別ウィンドウで起動するだけです. このときに図 18.18 に示すウィンドウが出てきます:

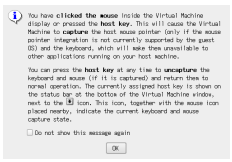


図 18.18: マウスポインタの移動について注意事項

これはマウスのポインタを VirtualBox の仮想計算機ウィンドウ内をマウスでクリックすることで移動させると特定のキーを押すか仮想計算機を停止しない限り, マウスのポインタが移動しないことを知らせるものです. このキーは仮想計算機の右下の枠に のように表示されており, 既定値は 右 Ctrl キーとなっています. この設定は VirtualBox の「ファイルメニュ」の 環境設定 (P)... から変更できます. 具体的には 環境設定 (P)... を選んで現われたウィンドウの左側の項目で「入力」を選択すれば図 18.19 に切替わってキーの設定できます:

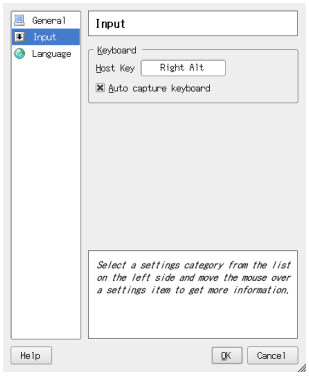


図 18.19: VirtualBox のマウスポインタ切替キーの設定

このウィンドウで ホストキー (K): の右枠にマウスポインタを置いて好きなキーを押せば, そこで指定したキーに切り換えられます.

18.5 VMware Player で KNOPPIX を利用する場合

18.5.1 VMware Player について

VMware Player は VMware, Inc. の製品で, http://www.vmware.com/products/player/ から入手できます. 以前の VMWare Player は QEMU というアプリケーションを使って仮想計算機を生成したり, 設定ファイルを直接編集する必要がありましたが, 現在の VMWare Player では Virtual Box と同様に Wizard 形式で仮想計算機の生成と設定が行えるようになっています.

18.5.2 設定方法

最初に VMWare Player を起動してみましょう. ちなみに Linux 環境で locale が EUC であれば起動に失敗するようなので, この場合は “LANG=C” にしておくと良いでしょう. なお, UTF-8 であれば問題はありません:

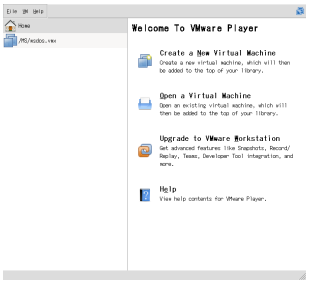


図 18.20: VMWarePlayer の起動画面

新規に仮想計算機を生成するので図 18.20 の左側の「Create a New Virtual Machine」を選択すると図 18.21 に示す Wizard が起動します:

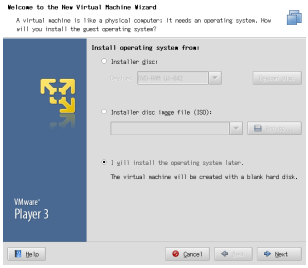


図 18.21: 新規生成の主画面

ここででは一番下の「I will install the operating system later」にチェックを入れて underlineNext を押します．すると図 18.22 に移動し，ここでは「Linux」にチェックを入れて下の Version から「Other Linux 2.6.x Kernel」を選択します:

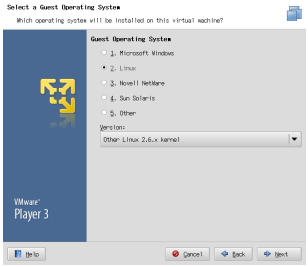


図 18.22: OS の指定

この設定を行うと図 18.23 に移動します:

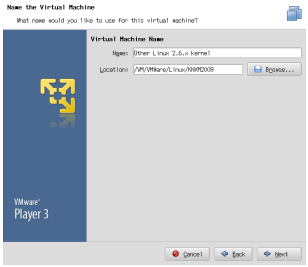


図 18.23: 仮想計算機の置き場所の指定

ここでは仮想計算機のファイルを置く場所を指定します. Location の欄に直接書込むか Browse を使って指示することもできます. この指定が終わると仮想ディスクの指定を行うための図 18.24 に示す Wizard に移動します:

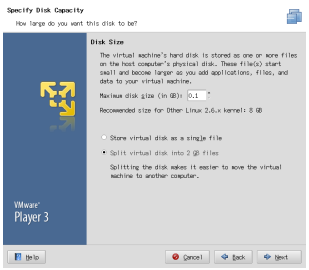


図 18.24: 仮想ディスクの指定

ここでの指定は VirtualBox と同様で, KNOPPIX/Math を起動するだけであれば仮想ディスクは不要ですが, VMWarePlayer では仮想ディスクの大きさは最低 0.1MB が指定されます. この指定を終えると仮想計算機の諸設定を行う図 18.25 に示す Wizard が起動します:

図 18.25: 仮想計算機の設定

ここでの設定は仮想計算機に割当てる記憶容量, CD/DVD ドライブ, サウンドカードや USB コントローラ等の設定が行えます. ここでは ISO ファイルを指定するので Cunstomize Hardware... を押して図 18.26 に示す Wizard を起動します:

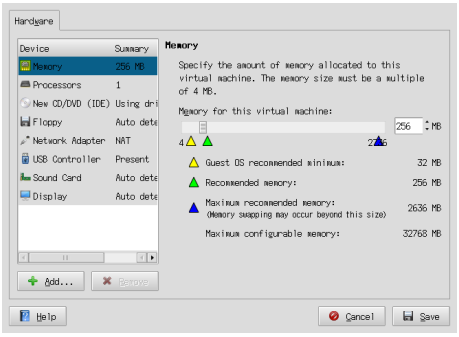


図 18.26: 仮想計算機の詳細設定

ここで左側の Summary より「New CD/DVD (IDE)」を選択し，「Use ISO image:」を選択し，ISO ファイルを下の空欄に直接記入するか Browse を選択して指定します．これで VMPlayer の準備は完了です．この状態で VMWare Player の起動画面を図 18.27 に示しておきます:

図 18.27: 仮想計算機の起動 (VMPlayer)

左側のリストから該当する仮想計算機を選択し，右下の Play virtual machine を押すと仮想計算機が起動します VMWare Player の場合，仮想計算機へのキーの切替は仮想計算機のウィンドウをピックすればよく，ホスト側への切替は Ctrl+Alt でできます．

18.6 仮想計算機と既存環境との共存

ここで openSUSE 上で KNOPPIX/Math2010 を起動させている様子を図 18.28 に示しておきます:

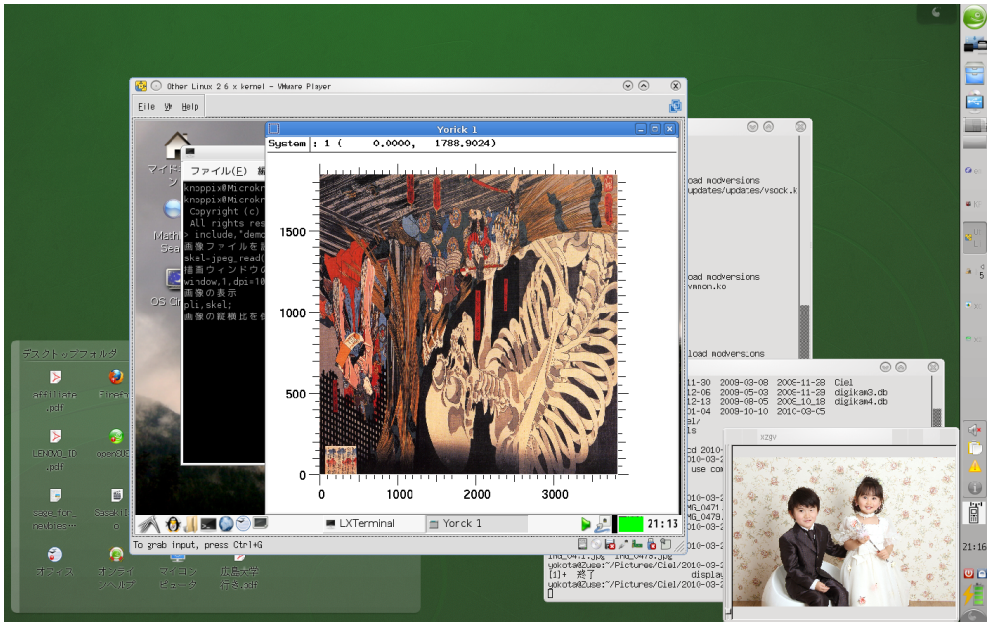


図 18.28: openSUSE と KNOPPIX/Math の共存

中央の大きなウィンドウが仮想計算機です. この様にアプリケーションと同様の状態で起動して従来の環境と併用することができます. VirtualBox や VMWare Player でも仮想計算機のウィンドウを閉じれば仮想計算機がハイバネートされ, 次に仮想計算機を立上げればウィンドウを閉じた状態から作業が続けられます[3]. また, KNOPPIX/Math は立上げ時にネットワークに自動的に接続するので, ネットワークを介して本体側と仮想計算機側のファイルの遣り取りもできます.

[3]VirtualBox や VMWare Player の既定値で, 電源を落す等の処理に変更することもできます.

18.7　KNOPPIX/Math2010 の使い方

ここで KNOPPIX/Math2010 のデスクトップの様子を示しておきましょう:

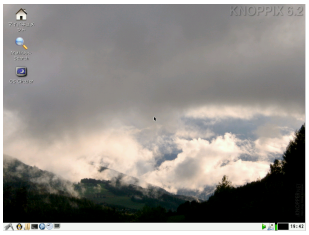


図 18.29: KNOPPIX/Math2010 のデスクトップ

左下端の が「**LXDE メニュー**」というもので, KDE の「**KDE メニュー**」, あるいは MS-Windows の「**スタートメニュー**」に相当します. それから隣にある が「KnxmLauncher」という昔の MacOS やワープロ専用機で見られたアプリケーションラウンチャーを起動させます. このラウンチャーには「**Math**」, 「**Internet**」, 「**Learn**」, 「**Play**」と「**設定**」の 5 種類のラウンチャー画面があり, それぞれの項目で主要なアプリケーションが分類されています. たとえば, Yorick を起動させたければ図 18.30 に示す「**Math**」から Yorick の髑髏のアイコンを見付けてクリックすれば良いのです:

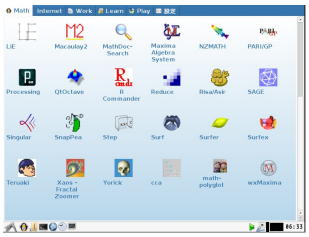


図 18.30: 「Math」の内容

なお, このラウンチャーからは立上げられないアプリケーションを起動させる場合には左端から 四番目の「**Terminal emulator**」のアイコン を押して仮想端末の LXTerminal を起動させて, そこからアプリケーションの起動を行います.

18.7.1　Flash memory へのインストール

KNOPPIX/Math 2010 より USB メモリディスク等の Flash memory へのインストールが容易に行えるようになりました. Flash memory へのインストールを行うと何が良いかと言えば, CD/DVD-ROM よりも読込が高速であることと, Flash memory に個人用のディレクトリを同時に作成しておくことで作業データも保存ができるようになり, Flash memory を持ち歩いていさえすれば何処

でも貴方の仕事ができるという訳です. 因に現在の KNOPPIX/Math 2010 は 4GB 程度を必要とするので, 8GB 程度の Flash memory があれば 4GB 程度作業領域に利用できます.
インストールは非常に簡単です. LXDE の lxdelauncher をクリックして上にある「**設定**」を押しましょう:

図 18.31: 「設定」の内容

ここで「**install KNOPPIX to flash disk**」をダブルクリックすると Flash memory 用のインストーラが起動します. ここでインストーラを起動するとディバイスが幾つか現われている筈です. インストールでは媒体のフォーマットを行うので指定したディバイスに保存されたデータは消えてしまいます! 内蔵ディスクは現在 ATA のものが多いので ATA の名前があるディバイスを決して指定しないようにして下さい。

18.7.2 KNOPPIX-Math-Start

さて, KNOPPIX/Math 2010 を利用する上で重要な「機能」が KNOPPIX-Math-Start です. この KNOPPIX-Math-Start は LXDELauncher では

です. この KNOPPIX-Math-Start から開かれたページを図 18.32 に示しますが, ここでは収録アプリケーションの概要とリンクが記載されています.

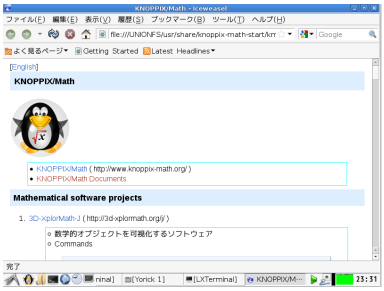


図 18.32: KNOPPIX-Math-Start

また, 文書の先頭にある KNOPPIX/Math Documents は KNOPPIX/Math 2010 に収録したディレクトリ**/usr/share/knoppix-math-doc/ja** へのリンクになっています. このリンク先のディレクトリの様子を図 18.33 に示しておきます:

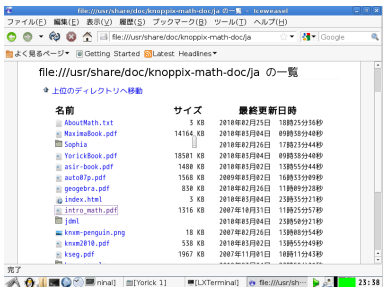


図 18.33: KNOPPIX-Math-Startja

このディレクトリに含まれているファイルの概要は **index.html** に記述されているのでここでは詳細は述べませんが, Maxima の優れた入門書である中川さんの「Maxima 入門ノート」, 私の解説書になりますが「はじめての Maxima(α版)」,「たのしい Yorick」等の PDF 文書や資料があります. ここでは特に重要な jdml について述べておきましょう.

18.7.3 JDML

jdml は “Japan Digitak Mathematics Library” が示すように日本国内の数学系雑誌の情報を集約して再構築する活動の 1 つの成果物です. JDML には大学や研究所等が出している雑誌に掲載された論文の情報が収録され, 論文の PDF をリンク先から入手することができます. ここでは図 18.34 に “index.html” を開いた様子を示しています:

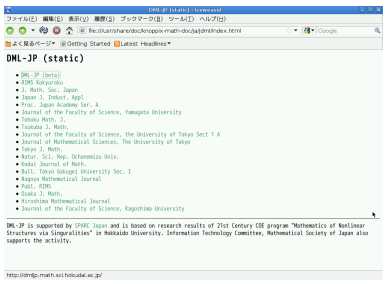


図 18.34: JDML/index.html を開いたところ

ここで “DML-JP(beta)” から http://dmljp.math.scihokudai.ac.jp/に飛ぶことができて, 著者や項目などで論文の検索を行うこともできます. たえば「knot alexander」で検索した結果を図 18.35 に示しておきましょう:

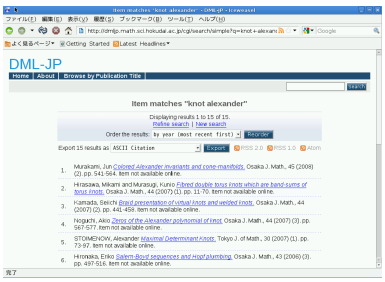


図 18.35: “knot alexander” での検索結果

18.7.4 KNOPPIX/Math 上での全文検索

KNOPPIX/Math のデスクトップの左上に というアイコンがありますね. このアイコンをクリックすると Namazu による KNOPPIX/Math 全文検索システムが立ち上がります:

図 18.36: MathDoc-search による全文検索

ここで検索式の箇所で, たとえば「Frege 概念記法」と入力して Enter キーを押せば該当語句を含む文書の検索を行います. ただし, バイナリファイルであれば該当箇所へのリンクとは限らず, 開いて自分で捜す必要があります. 先程の例でリンク先の MaximaBook.pdf を開いて該当箇所を自分で捜した結果を図 18.37 に示しておきます:

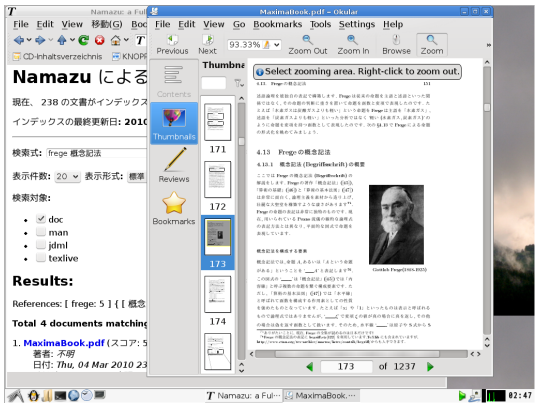


図 18.37: 該当箇所を見付けたところ

このようにマニュアル類を全て自分で調べなくても絞り込みが行えるのです.
以上の説明からもお判りのように, KNOPPIX/Math は数学アプリケーションを集積した Linux 環境だけではなく, 数学に関連する文献やその情報を集めた「数学支援」環境, すなわち, 数学を楽しむための**「数学の玩具箱」**なのです.

第19章　最後に

Faust
Was ich verbaute,richte du grade,
was ich versäumte, schöpfe du nach.
So stell' ich mich über die Regel,
umfaß in einem die Epochen
und vermenge mich den letzten
Geschlechtern:
Ich,Faust,ein ewiger Wille!

ファウスト
俺が歪に築上げた事を, お前は正せ,
俺が怠けて損ねた物を, お前は掴み取るのだ.
そうすれば, 宿命を越えて俺は生き続け,
様々な時代を一つに包含し,
そして, 俺は遂には同胞(はらから)に混ざり合う:

我はファウスト, 永遠の意思!
Buzoni,Doktor Faust[101] より

この本は Maxima と銘打っておきながら, 実体は KNOPPIX/Math に収録されたアプリケーションの紹介も目的の一つでした. 皆さんは楽しんで頂けたでしょうか？
Maxima には, ここで解説したものの他に, 様々なライブラリも付属しています. 勿論, LISP の環境によっては十分に使えないものや, メンテナンスが十分に行なわれていないものもありますが, これらに関しては,Maxima に付属の文書を参照して下さい.
この本で Maxima の計算原理に関連することに重点を置いた訳は, ソフトウェアを使う際に, 使いこなせない問題の多くが, ソフトウェアの仕様や原理への理解不足, 加えて, 対処しようとする問題自体への理解不足と, ソフトウェアと問題の双方の理解不足に大きな問題があると感じることが多いためです. そこで, この本では Maxima の函数の一覧や, その例題集を作るよりは, あえて, 入力した式がどの様に内部で処理されてゆくかに焦点を当てています. 実際, 内部で式がどの様に表現され, どの様に処理されているかを理解さえしていれば, あとは自分が必要とする函数も朧げながらにも見当が付き, Maxima に附属の文書やネット上の情報を調べれば済みますが, そもそも, 何処をどの様に調べれば判らなければ対処の方法はないでしょう.
この Maxima が Maple や *Mathematica* と比べて最も優れている点は, 誰もが疑問を持ったら自由に中身を調べられる点でしょう. 勿論, Maple ではライブラリを自ら調べられますが, 無条件で誰でもシステム全体を調べられることは Maxima の大きな長所です. その際に, Reduce の様に独自の言語を取得する必要もなく, 比較的, メジャーな Common Lisp の知識が多少でもあれば良いのです, 実際, 自分が必要とされる函数に, どの様な函数や大域変数が用いられているか調べられ, さらには, ソースファイルの函数内部に print 函数を挿入して, S 式がどの様に変化するかを眺めるだけでも十分なのです. それだけでも, 何も知らずに使うのとは全く違います. 積極的に Maxima を弄ることで, Maxima の限界や問題点だけではなく, それらを越える新たな可能性にも気付かされる筈です.
「数学の色々なこと」には, Maxima で遊ぶために最低限必要と私か感じた事項をとりとめも無く書いています. 冒頭のロゴス賛歌の様に, 脈絡も何もなく混沌としたアトムの世界を論理 (logic) で束ね, そこから色々なものが派生して行く様子を表現したかったのですが, 私の力不足で十分とは思

えません. この部分には, 私が嘗て高校生の時分に読んだ, 赤先生の「新講 数学シリーズ」の影響もあります. これらの本は非常に贅沢な数学の教科書で, 軽薄短小の逆を行く, 重厚で, とても滋味豊富な本で, 今でも非常に面白い本です. 良書が時を越える 1 つの好例でしょう.
さらに, Frege の解説が必要以上に多いと思われるかもしれませんが, Frege の著作にはそれだけの魅力 (魔力?) があると理解して頂ければ幸いです. 兎に角, 一つの世界を創造してゆく過程を目の当たりにすることができるのです! また, 個人的な思い出になりますが, Frege の「算術の基礎」を読んでいて, 成績不良の中学生の頃に「何故,1+1=2 なのか」と考えていたことを懐しく思い出したことも付け加えておきます. 優秀な同級生は私の質問に真面に答えることはなく, 寧ろ, 「勉強の出来ない真面目な子」の方が説明しようとしたものです. 中には鉛筆と鉛筆キャップを使って説明してくれた子も居ましたが, 意地悪にも, 鉛筆キャップを鉛筆に付けて, 一つになったから '1 + 1 = 1' じゃないかと言うと, その子は流石に私を憐む様な顔をして答に窮していましたが….
結び目の章は, 規則とその評価に重点を置いています. 基本的に, 計算機に入力される式は, ASCII データの羅列でしかありません. これにどうやって意味を持たせ, どの様に処理するかが問題となります ここでは, 演算子で式を区切り, 演算子の属性, さらには函数の持つ性質 (公式等) で式を簡易化して行く訳ですが, そこで重要なのは置換規則, 変数並びの指定等で, 如何に適切に規則を与えられた並びに対して適用するかが問題となります. この章は十分描き切れたとは思えませんが, 空間図形から妙な文字の羅列が構成され, それに色々と細工をしてゆくに従って, 意味が生じる所を楽しんで頂ければと思っています.
surf でお絵描きを行う章は, 代数学の様々な概念を気楽に楽しんでもらおうと思って記述しました. SINGULAR も引っ張ってきましたが, ドイツ流儀で見掛けは堅苦しい SINGULAR も surf と併用すれば案外軟派な使い方ができるので, Maxima で不十分と感じれば, 色々と試してみて頂ければと思っています.
Octave の話は, Maxima に限らず数式処理で強引に非常に大きな数値行列の処理を実行したものの, 処理や速度に難があり, それが数式処理一般の評価を下げる面があると感じた点もあります. 数値行列を数式処理で扱うためには, 数値行列を扱うためのライブラリを組込むことになりますが, 現時点では, 数値行列は MATLAB や Octave の様なシステムで処理すべきで, 数式処理は MATLAB や Octave が苦手とする記号処理で活用すべきであると私は思っています. しかし, その割には貧弱なインターフェイスプログラムしか書いていませんが, これを参考にして良いものを作ってみて下さい.
Maxima の簡単な改良の話は, 目標とするものと, その結果の落差が大きく, 腰砕け気味ですが, 色々と改造してゆけるのは Open Source ならではのことです.
インストールの話では自力でコンパイルする話の分量が少なくなっています. これは, 少し前と比べて格段に Maxima を使う環境が整っていることの現れと理解して下さい. ここでは寧ろ, 仮想計算機環境のことを記述していますが, この仮想計算機は少し前の OS 論争を完全に過去のものとするだけの威力が十分にあります.
最後に, Maxima を自分流に使って楽しんで下さい.

関連図書

[1] アプレイウス (著), 呉茂一 (訳), 黄金のろば (上下), 岩波文庫, 岩波書店,1956.

[2] 荒川恒男, 伊吹山知義, 金子昌信 (著) ベルヌーィ数とゼータ関数, 牧野書店,2001.

[3] アリストテレス, 形而上学 上下, 岩波文庫,2007.

[4] 飯田隆, 言語哲学大全 I 論理と言語, 勁草書房,2003.

[5] 井筒俊彦, イスラーム思想史, 中公文庫, 中央公論社,1991.

[6] 岩熊幸雄, ライプニッツの内包論理-Generales Inquistiones の一解釈-, 哲学論叢 1-Aug-1978, 京都大学哲学論叢刊行会,http://hdl.handle.net/2433/24432

[7] アンドレ・ヴェイユ, アンドレ・ヴェイユ自伝-ある数学者の修行時代, シュプリンガー・フェアラーク,1994.

[8] ヴォルテール, カンディード:他五篇, 岩波文庫, 岩波書店,2005.

[9] G.H. フォン・ウリグト, 論理分析哲学, 講談社学術文庫, 講談社,2000.

[10] H.D. エビングハウス,H. ヘルメス,F. ヒルツブルッフ,M. ケッヒャー,K. マインツァー, J. ノイキルヒ,A. ブレステル,R. レンメルト著, K. ラモトケ編, 成木勇夫訳, 数 (上, 下), シュプリンガー・フェアラーク,1991.

[11] 大山晃,Principia Mathematica における命題函数, 分析哲学の誕生 [41] に収録

[12] 河上徹太郎, 近代の超克, 冨山房百科文庫 23, 冨山房,1979.

[13] 桂紹隆, インド人の論理学, 中公新書, 中央公論社,1998.

[14] ガリレオ, 新科学対話 (上), 岩波文庫, 岩波書店,2007.

[15] クセルゴン, 自由・平等・清潔-入浴の社会史, 河出書房新社,1992.

[16] クラウス, 人類最後の日々, カール・クラウス著作集 〈9〉, 法政大学出版局,1971.

[17] クレムペラー, 第三帝国の言語「LTI」ーある言語学者のノート, 法政大学出版局,1974.

[18] キューン, クヴァンダー共著, 岩下眞好他訳, グスタフ・マーラー その人と芸術、そして時代, 泰流社,1989.

[19] ポール・グレアム,ANSI Common Lisp, ピアソン・エデュケーション,2002.

[20] 河内明夫編, 結び目理論, シュプリンガー・フェアラーク東京,1990.

[21] J. リヒター-ゲバート/U.H. コルテンカンプ著, 阿原一志訳, シンデレラ 幾何学のためのグラフィックス, シュプリンガー・フェアラーク東京,2001.

[22] 後藤和茂,BLAS の概要 (http://jasp.ism.ac.jp/kinou2sg/contents/RTutorial_Goto1211.pdf), 2006.

[23] 小平邦彦, 幾何学の誘い, 岩波書店,1991.

[24] 斎藤憲, ユークリッド「原論」の成立 古代の伝承と現代の神話, 東大出版会,1997.

[25] 佐藤雅彦, フレーゲの計算機科学への影響, 分析哲学の誕生 [41] に収録

[26] 柴田有, グノーシスと古代宇宙論, 勁草書房,1982.

[27] 下地貞夫, 数式処理, 基礎情報工学シリーズ, 森北出版,1991.

[28] セネカ, アポコロキュントシス 神君クラウディウスのひょうたん化,「サテュリコン—古代ローマの諷刺小説」に併載, 岩波文庫, 1991.

[29] ソーマディーヴァ(著), 上村勝彦 (訳), 屍鬼二十五話-インド伝奇集, 東洋文庫 323, 平凡社,1978.

[30] ゲーデル, 林晋・八杉満利子訳・解説, 不完全性定理, 岩波書店,2006.

[31] 高木貞治, 復刻版 近世数学史 数学雑談, 共立出版,1997.

[32] 高木貞治, 数の概念, 岩波書店,1970.

[33] 田畑博敏, フレーゲの論理哲学, 九州大学出版会,2002.

[34] 田中尚夫, 選択公理. 遊星社,1987.

[35] 寺坂英孝編, 現代数学小事典, ブルーバックス, 講談社,2005.

[36] デーデキント著, 河野伊三郎訳, 数について 連続性と数の本質, 岩波文庫,1996.
Project Gutenberg による英訳 (Essays on the Theory of Numbers):
http://www.gutenberg.org/etext/21016

[37] 長尾真, 淵一博, 論理と意味, 岩波講座 情報科学-7, 岩波書店,1985.

[38] 中川義行,Maxima 入門ノート,
http://www.wakaba.jp/ moriarty/works/index.html

[39] 中根美知代,$\epsilon - \delta$ 論法の形成過程の考察:解析学の基礎の転換の要因, 数理解析研究講究録,1195 巻,2001.

[40] 箱崎総一, カバラ ユダヤ神秘思想の系譜, 青土社,1988.

[41] 日本科学哲学学会 [編], 野本和幸 [責任編集], 科学哲学の展開 [1], 分析哲学の誕生, フレーゲ・ラッセル, 勁草書房,2007.

[42] 広瀬健 横田一正, ゲーデルの世界 -完全性定理と不完全性定理-, 海鳴社,1985.

[43] ヒルベルト, 幾何学基礎論, 筑摩書房,2005.

[44] ヒルベルト ベルナイス, 数学の基礎, シュプリンガー・フェアラーク,1993.

[45] R. フィンスター,G. ファン・デン・ホイフェル著, 沢田允茂監訳, 向井久他訳, ライプニッツ その思想と生涯, シュプリンガー・フェアラーク東京,1996.

[46] クロウエル, フォックス, 結び目理論入門, 現代数学全書, 岩波書店,1989.

[47] プラトン, テアイテトス, 岩波文庫, 岩波書店,2007.

[48] フレーゲ, フレーゲ著作集 1 概念記法, 勁草書房,1999.

[49] フレーゲ, フレーゲ著作集 2 算術の基礎, 勁草書房,1999.

[50] フレーゲ, フレーゲ著作集 3 算術の基本法則, 勁草書房,2000.

[51] フレーゲ, フレーゲ著作集 6 書簡集 付「日記」, 勁草書房,2000.

[52] ポアンカレ (著), 吉田洋一 (訳), 科学と方法, 岩波文庫, 岩波書店,1953.

[53] 薮内清, 墨子:東洋文庫, 平凡社,1996.

[54] 本間龍雄, 組合せ位相幾何学, 共立出版,1980.

[55] 前原昭二, 数学基礎論入門, 朝倉書店,2007.

[56] 牧野, 円周率 100,000,000 桁表, 暗黒通信団,2007.

[57] 丸山茂樹, クレブナー基底とその応用, 共立叢書 現代数学の潮流, 共立出版,2002.

[58] 村上順, 結び目と量子群, 数学の風景 3, 朝倉書店,2000.

[59] 日本数学会編, 数学辞典 第 3 版, 岩波書店,1987.

[60] 谷沢淳三, 論証の学としてのインド論理学:帰納法と演繹法,
人文科学論集. 人間情報学科編 信州大学 Vol41(20070315) p.233-252, http://ci.nii.ac.jp/naid/110006389066/

[61] 矢吹道郎, 大竹敢, 使いこなす GNUPLOT(改訂新版), テクノプレス,2001.

[62] 山川偉也, 条件文についての古代の論争：メガラ・ストア論理学の理解のために (共同研究 :「言語の本質」についての総合的研究), 総合研究報 St.Andrew's University,Bulletin of Research Institute, Vol.8,No.1(19820930)p. 1-14, 桃山学院大学 ISSN:03850811

[63] 山本光雄, 戸塚七郎 訳編, 古代ギリシア哲学者資料集, 岩波書店,1985.

[64] 湯浅太一, 萩谷昌己,Common Lisp 入門, 岩波書店,2007.

[65] 横田博史, たのしい Yorick,
http://www.bekkoame.ne.jp/ ponpoko/KNOPPIX/YorickBook.pdf,2010.

[66] 横田博史, 数値計算&可視化ソフト Yorick,I/O ブックス, 工学社, 2010.

[67] 吉田利信, 芹沢昭生, はじめての LISP, 技術評論社,1985.

[68] B. ラッセル (著), 野田又夫 (訳, 私の哲学の発展, みすずライブラリー,
みすず書店,1997.

[69] ルキアノス, ルキアノス選集, 叢書アレキサンドリア図書館第 8 巻, 国文社,1999.

[70] ルキアーノス, 神々の対話 他 6 篇, 岩波文庫, 岩波書店,1953.

[71] ルーキアーノス, 高津春繁 (訳), 遊女の対話 他 3 篇, 岩波文庫, 岩波書店,1953.

[72] Abramowitz and Stegun, Handbook of Mathematical Functions
http://www.math.sfu.ca/ cbm/aands/ .

[73] S.Blackburn,Oxford Dictionary of Philosphy second edition revised, Oxford university press,2008.

[74] G.Boole, The Calculus of Logic,The Cambridge and Dublin
Mathematical Journal,vol. 3 (1848),
http://www.maths.tcd.ie/pub/HistMath/People/Boole/CalcLogic/.

[75] G.Boole, Investigatio of the Law of Thought,Dover,1973,
http://http://www.gutenberg.org/etext/15114.

[76] H.Cohen, A Course in Computational Algebraic Number Theory,GTM 138, Springer-Verlag,New York-Berlin,2000.

[77] D.Cox,J.Little and D. O'Shea,Ideals, Varieties, and Algorithms,UTM,
Springer-Verlag,New York-Berlin,1992.

[78] Dongarra Oral History, by Thomas Haigh, 26 April, 2005, University of Tennessee, Knoxville TN. Society for Industrial and Applied Mathematics, Philadelphia, PA,
http://history.siam.org/oralhistories/dongarra.htm .

[79] Fränkel,The notion "definite" and the independence of the axiom of choice, 1922b,Heijenoort([81],p.284-289).

[80] Gert-Martin Greuel, Gerhard Pfister, A Singular Introduction to Commutative Algebra, Springer-Verlag,New York-Heiderberg-Berlin,2000.

[81] Heijenoort, From Frege to Gödel, A Source Book in Mathematical Logic ,1879 - 1931, HARVARD UNIVERSITY PRESS,1976.

[82] Hilbert,On the infinite,[81],p.367-392 に収録.

[83] Hilbert,The foundations of mathematics,[81],p.464-479 に収録.

[84] Kleene, Mathematical Logic, Dover.

[85] Lawson Oral History. Thomas Haigh, 6 and 7 November, 2004, San Clemente, California. Society for Industrial and Applied Mathematics, Philadelphia, PA.
http://history.siam.org/oralhistories/lawson.htm .

[86] Mario Livio,The Golden Ratio The Story of Phi, the World's Most Astonishing Number,Broadway Books,2003

[87] John McCarthy,Recursive Functions of Symbolic Expressions
and Their Computation by Machine,Part I,April 1960,
http://www-formal.stanford.edu/jmc/recursive.html.

[88] Peano,The principle of arithmetics,presented by a new method, [81],p.81-103 に収録.

[89] D.Rolfsen, Knots and Links. Publish or Perish, Inc.,1975.

[90] B.Russell, The Principles of Mathematics,W.W.Norton & Company,Inc.,1996.

[91] B.Russell,The Mathematical logic as based on the theory of types,1908, [81],p.150-182 に収録.

[92] B.Russell & A.N.Whitehead,Principia Mathematica to *56, Cambridge Mathematical Library,Cambridge University Press,1997.

[93] B.Russell, Introduction to Mathematical Philosophy, Cosimo Classics, 1918.

[94] B.Russell, Introduction to Mathematical Philosophy, http://people.umass.edu/klement/russell-imp.html, PDF 版等がある.

[95] Hal Schenck, Computational Algebraic Geometry London Mathematical Society student texts;58,2003.

[96] Michael Schroeder, A BRIEF HISTORY OF THE NOTATION OF BOOLE'S ALGEBRA, Nordic Journal of Philosophical Logic, Vol.2, No.1,p.41 -62, http://www.hf.uio.no/ifikk/filosofi/njpl/vol2no1/history/history.pdf.

[97] Seneca,APOCOLOCYNTOSIS, OR LUDUS DE MORTE CALUDII: THE PUMPKINIFICATION OF CLAUDIUS, Project Gutenberg,
http://www.gutenberg.org/files/10001/10001-h/10001-h.htm

[98] Zermelo,Investigations in the foundations of set theory I,1908a,
Heijenoort([81],p.199-215).

[99] Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/contents.html

[100] Frege's Logic, Theorem, and Foundations for Arithmetic, Stanford Encyclopedia of Philosophy. http://plato.stanford.edu/entries/frege-logic

[101] Ferruccio Busoni,Doktor Faust. (ERATO)

[102] Open AXIOM のサイト http://wiki.axiom-developer.org/FrontPage

[103] DERIVE のサイト http://www.derive.com

[104] GAP のサイト http://www-gap.dcs.st-and.ac.uk/

[105] Macaulay2 のサイト http://www.math.uiuc.edu/Macaulay2/

[106] Mathsoft Engineering & Education, Inc. http://www.mathcad.com/

[107] Wolfram Research Inc. http://www.wolfram.com/

[108] Waterloo Maple Inc. http://www.maplesoft.com/

[109] Maxima の SOURCEFORGE のサイト http://maxima.sourceforge.net/

[110] MuPAD のサイト http://www.mupad.de/

[111] PARI/GP のサイト http://pari.math.u-bordeaux.fr/

[112] REDUCE のサイト http://www.zib.de/Symbolik/reduce/

[113] Risa/Asir 神戸版 http://www.math.kobe-u.ac.jp/Asir/asir-ja.html

[114] OpenXM(Open message eXchange for Mathematics)
http://www.math.sci.kobe-u.ac.jp/OpenXM/index-ja.html

[115] SAGE のサイト http://www.sagemath.org/

[116] SINGULAR のサイト http://www.singular.uni-kl.de/

[117] SWI-Prolog のサイト http://www.swi-prolog.org/

[118] 株式会社シンプレックスのページ htto://www.simplex-soft.com/

[119] The MathWorks,Inc. http://www.mathworks.com/

[120] Octave WebPage http://bevo.che.wisc.edu/octave/

[121] Scilab のサイト http://www.scilab.org/

[122] Yorick の公式サイト ftp://ftp-icf.llnl.gov/pub/Yorick/
Yorick の非公式サイト http://www.maumae.net/yorick/doc/index.php

[123] R のサイト http://www.r-project.org

[124] Insightful Corporation のページ http://www.insightful.com/

[125] Cinderella のサイト
本家：http://www.cinderella.de/
日本語版ホームページ http://cdyjapan.hp.infoseek.co.jp/

[126] dynagraph のサイト http://www.math.umbc.edu/ rouben/dynagraph

[127] KSEG のサイト http://www.mit.edu/ ibaran/kseg.html

[128] Geomview のサイト http://http.geomview.org/

[129] surf の sourceforge のサイト http://surf.sourceforge.net/

[130] surfer のサイト (IMAGINARY2008) http://www.imaginary2008.de/surfer.php.

[131] surfex のサイト http://www.surfex.algebraicsurface.net.

[132] XaoS のサイト http://wmi.math.u-szeged.hu/ kovzol/xaos

[133] Begriffsschrift in LaTeX
http://arche-wiki.st-and.ac.uk/ ahwiki/bin/view/Main/BegriffsschriftLaTeX

[134] fge パッケージのサイト http://sview01.wiredworkplace.net/pub/jjg/en/code/fge.html

[135] Numbers,constants and computation, http://numbers.computation.free.fr/Constants/constants.html

[136] QEMU のサイト http://bellard.org/qemu/

索引

逆引
数学
数式の内部表現の考え方, 79, 85
順序の考え方, 82
同値類の考え方, 63
アプリケーション
gnuplot の使い方を知りたい, 753
LISP から Maxima に戻る, 43, 324
Maxima から LISP に移動, 43, 324
Maxima から外部アプリケーションを起動, 708, 952
Singular の使い方, 972
プログラム言語としての gnuplot, 785
グラフの描画
曲線グラフの解像度を上げげたい, 739
曲面グラフの解像度を上げげたい, 739
gnuplot のフォントが潰れて読めない!, 732
plot2d 函数の使い方, 733
plot2d による媒介変数式の表示, 733
plot3d 函数の使い方, 736
曲面をソリッド表示にしたい, 745
座標軸を対数目盛にしたい, 741
三角函数
三角函数を含む式の展開を行いたい, 633
三角函数を含む式を簡単にしたい, 636
倍角公式を使って式の展開をしたい, 635
冪を倍角公式で纏めたい, 636
式
Maxima に式の自動展開をさせたい, 517
式の因子分解をしたい, 351
式を展開したい, 520
常微分方程式
解の検証, 567
微分方程式の書き方, 565
常微分方程式の簡単な解き方, 565
数値
Maxima の数学定数, 332
精度を変更したい, 331
積分
Laplace 変換を使いたい, 557
数値積分を行いたい, 561
代数的数を使って積分したい, 553
留数の計算をしたい, 559
定積分を行いたい, 559
変数変換をしたい, 552
微分
式の微分をしたい, 543
評価
函数や演算子に影響を与える大域変数を知りたい, 322
方程式の解を手軽に他の式に入れたい, 320
式を手軽に展開したい, 317
式を手軽に評価したい, 318
方程式
自動的に方程式の解を変数に入れたい, 528
決った範囲内で実数解だけを求めたい, 531
実数解だけを求めたい, 530
連立方程式を解きたい, 532

え
演算子, 198, 199
演算子, 16

非演算子, 16
演算子項, 204
お
オンラインマニュアル, 15
か
函数, 199
函数項, 204
き
記号, 198, 199
規則
規則, 293
〜の削除, 308
規則の適用, 293
行列
〜の固有値の計算, 30
〜の特性多項式, 31
〜の差, 29
〜の定義, 28
〜の和, 29
け
原子 (=atom), 198
こ
項, 208, 214
し
式, 208
式の並び, 293, 294
真理函数, 210
真理値, 199
狭義の真理値, 203
広義の真理値, 203
せ
整数, 201
整数 (fixnum,bignum), 198
そ
属性, 199, 207
属性の表現函数, 211
た
大域変数, 209
対象, 198
多倍長浮動小数点数
多倍長浮動小数点数 (bigfloat), 199
て
定数, 198
と
動詞型, 206
な
内部函数, 205
内部変数, 209
名前, 199, 200
並びの照合, 294
は
パターン, 294
パターンマッチング, 294
バッチ処理, 14
バッチファイル, 14
判断, 203
ふ
複素数, 199
浮動小数点数
B-表記, 202
D-表記, 202
E-表記, 202
小数点表記, 202
浮動小数点数 (float), 199
部分式, 208
文, 198
へ
変項, 198
変数, 198
自由変数, 203
束縛変数, 203
め
名詞型, 206
も
文字, 200
文字列, 198, 201
ゆ
有理数, 199
れ
列, 201

ろ
論理式
述語, 210
き
行列
〜の積, 29
き
行列
〜の逆行列の計算, 30
グラフの属性
C
cbrange, 811
colorbox, 811
P
palette, 811
記号
$>_m$, 213, 956
$>_n$, 208
A_m, 213
%i, 502
%o, 502
%t, 502
\(=¥), 199
expt, 230
nexpt, 230
演算子
〜の型, 223
〜の束縛力, 19, 220, 917
〜の束縛力 (bp), 220
〜の左束縛力 (lbp), 220
〜の右束縛力 (rbp), 220
外挿表現の〜, 219
可換積, 229
後置表現の〜, 219
前置表現の〜, 219
内挿表現の〜, 219
非可換積, 229
無引数の〜, 219
A
and, 210, 232
D
do, 235
E
else, 234
elseif, 234
F
for, 235
from, 235
I
if, 234
if 文で利用可能な演算子, 574
in, 575
N
next, 235, 575
not, 210, 232
O
or, 210, 211, 232
S
step, 235, 575
T
then, 234
thru, 235, 576
U
unless, 235, 576
W
while, 235, 576
記号
:lisp, 215, 325, 726
!!, 231, 606
! , 231, 606
″, 321
′, 321, 585
**, 229
*, 229
+, 229
-, 229
., 229
/, 229
::=, 233, 260, 584
::, 233

:=, 14, 233, 260, 580
:, 14, 233
;, 13
¡=, 232
¡, 232
=, 14, 232, 526, 565
¿=, 232
¿, 232
??, 701
?, 16, 325, 701
@, 500
#, 232
$, 13
^^, 30, 229
^, 229

型
A
any, 223, 600, 601
any_check, 600
B
big, 600
bignum, 201, 600
boole, 600
boolean, 600
C
clause, 223
complex, 600
E
expr, 223
F
fixnum, 201, 600
fixp, 600
float, 600
floatnum, 600
flonum, 600
I
integer, 600
L
list, 600
listp, 600
N
none, 600
number, 600
R
rat, 600
rational, 600
real, 600
た
多倍長浮動小数点数, 202
ふ
浮動小数点数, 202

1 函数
S
system, 952

函数
?
?round, 338
?truncate, 338
%
%ith, 506
%j, 396
%th, 14
A
absolute_real_time, 706
acos, 633
acosh, 633
acot, 633
acoth, 633
acsc, 633
acsch, 633
activate, 288
add_zeros, 871
addcol, 435, 927
addrow, 435
adjoin, 417
adjoint, 439, 927
airy_ai, 646
airy_bi, 646
airy_dai, 646
airy_dbi, 646

algsys, 27, 532
alias, 260, 707
allroots, 530
aload_mac, 714
alphanumericp, 487
antid, 564
antidiff, 564
append, 403
appendfile, 37, 716
apply, 411
apply1, 300
apply2, 300
approps, 707
apropos, 16
args, 382
array, 425
arrayapply, 427
arrayinfo, 424
arraymake, 425
ascii, 489
asec, 633
asech, 633
asin, 633
asinh, 633
askinteger, 253
asksign, 524
assoc, 395
assume, 21, 210, 240, 280
at, 255
atan, 633
atan2, 633
atanh, 633
atom, 402
atomp, 275
atvalue, 254, 259, 568
augcoefmatrix, 432
auto_mexpr, 714
B
bashindices, 392
batch, 713, 790
batchload, 713
bc2, 28, 568
belln, 608
bern, 609
bernpoly, 609
bessel, 648
bessel_i, 648
bessel_j, 648
bessel_k, 648
bessel_y, 648
beta, 611
bezout, 358
bfhzeta, 620
bfloat, 275, 336
bfloat_approx_equal, 333
bfloatp, 333
bfzeta, 620
binomial, 612
block, 313, 572, 583
bothcoef, 347
break, 577, 689
bug_report, 724
build_info, 724
buildq, 584
b
bfpsi, 619
bfpsi0, 619
C
cabs, 337
cardinality, 419
carg, 337
cartesian_product, 414
catch, 576
ceiling, 333
cequal, 488
cequalignore, 488
cf, 621
cfdexpand, 621
cfdisrep, 621, 622
cgreaterp, 488

cgreaterpignore, 488
changevar, 552
chaosgame, 675
charat, 490
charfun, 274
charlist, 490
charp, 487
charpoly, 31, 441
cint, 489
clearrule, 308
cless, 488
clessignore, 488
close, 484
closefile, 37, 716
closeps, 750
coeff, 347
coefmatrix, 432
col, 435
collapse, 688
columvector, 447
combine, 367
compare, 275
compfile, 595
compile, 31, 595
compile_file, 595
concat, 482
conj, 447
conjugate, 447
constantp, 275, 295, 333
constituent, 487
content, 354, 618
copy, 394, 404, 434
copylist, 394, 403
copymatrix, 394, 434
cos, 632
cosh, 632
cot, 632
coth, 632
coutou_plot, 750
create_list, 400
csc, 632
csch, 632
cubrt, 396
D
dblint, 561
deactivate, 288
declare, 200, 205, 211, 213, 239, 259, 260, 314
declare_translated, 593
define, 260, 580
define_variable, 599
define_variable 函数の処理手順, 601
defint, 559
defmatch, 297
defrule, 261, 294, 298
defstruct, 500
deftaylor, 258, 260
del, 543
delete, 403
delta, 557
demo, 699
demoivre, 520
denom, 367
depends, 256, 260, 550
derivdegree, 543
derivlist(ev 函数の引数), 318, 319
describe, 15, 699, 896
desolve, 567
determinant, 441, 927
detout(ev 函数の引数), 317
diagmatrix, 431
diagmatrixp, 275
diff, 25, 543
diff(ev 函数の引数), 318, 319
digitcharp, 275, 487
dipslay, 693
disjoin, 417
disjointp, 423
disp, 693
dispform, 229, 379

dispfun, 227, 589
displate, 383
disprule, 299
dispterms, 694
distrib, 522
divide, 356
divisors, 414
do, 574
dpart, 696
draw, 800
draw2d, 799
〜と draw2d,draw3d との関係, 799
draw3d, 799
〜の gr3d 文, 799
〜の gr2d 文, 799
〜の gr 文, 799
〜のきよくしよてきそくせい, 798
〜函数の構文 1, 800
〜函数の構文 2, 802
〜のたいいきてきそくせい, 798
dvsum, 628
ldisplay, 693

E

echelon, 437
econs, 403
ed, 719
eigenvalues, 31, 447
eigenvectors, 31, 447
eivals, 447
eivects, 447
elapsed_real_time, 706
elapsed_run_time, 706
elementp, 423
eliminate, 358
elliptic_e, 655
elliptic_ec, 655
elliptic_eu, 655
elliptic_f, 655
elliptic_kc, 655
elliptic_pi, 656
ematrix, 431
emptyp, 423
endcons, 403
entermatrix, 430
entier, 333
equal, 210, 267
equiv_classes, 418
erf, 559
errcatch, 577
error, 577, 698
errormsg, 577, 698
euler, 613
ev, 22, 210, 270, 312, 567
eval, 321
eval(ev 函数の引数), 321
evenp, 274, 333
every, 266, 274
evolution, 672
evolution2, 673
example, 15, 699
exp, 639
expand, 20, 317, 520, 956
expand(ev 函数の引数), 317
exponentialize, 520
express, 548
external_subset, 418
ezgcd, 354

F

facout, 524
factcomb, 350
factor, 20, 351
factorout, 351
factorsum, 351
facts, 280
facttimes, 350
feature, 241
featurep, 241, 920
features, 239
Ffortmx, 397
Ffortran, 397

fib, 614
fibtophi, 614
file_search, 712
file_type, 712
filename, 712
fillarray, 427
first, 405
fix, 333
flaten, 401
flatten, 416
flength, 484
float, 275
float_approx_equal, 333
float_approx_equal_tolerance, 333
floatnump, 333
floor, 333
forget, 280
fortran, 31
fposition, 484
freeof, 272
freshline, 485
full_listify, 415
fullmap, 336, 408
fullmapl, 408
fullratsimp, 360
fullratsubst, 512
fullsetify, 414
funcsolve, 538
fundef, 589
funmake, 583
G
Γ 函数, 614
gamma, 612, 615
gcd, 354
gcde, 354
gcdex, 354
gcfactor, 354, 628
genfact, 606
genmatrix, 433
get, 238, 955
get_plot_option, 737
gfactor, 354
gfactorsum, 354
gnuplot_close, 747
gnuplot_pipes, 747
gnuplot_replot, 747
gnuplot_reset, 747
gnuplot_restart, 747
gnuplot_start, 747
go, 573
gradef, 256, 260
gramschmidt, 448
grind, 38, 693
gschmidt, 448
H
help, 701
hipow, 348, 543
horner, 24, 360
I
ic1, 568
ic2, 28, 568
ident, 431
identity, 274
if, 573
ifactor, 623
ifs, 676
ilt, 557
imagpart, 330, 337
infeval(ev 函数の引数), 321
infix, 224
inflag の影響を受ける函数, 402
innerproduct, 446
inpart, 386, 512, 514, 691, 916
inprod, 446
inrt, 334, 335
integer_partitions, 422
integerp, 275, 333
integrate, 26, 248, 549
interpolate, 538
intersect, 416

intersection, 416
intosum, 523
inv_mod, 623
inverse_jacobi_cd, 661
inverse_jacobi_cn, 658
inverse_jacobi_cs, 661
inverse_jacobi_dc, 661
inverse_jacobi_dn, 658
inverse_jacobi_ds, 661
inverse_jacobi_nc, 659
inverse_jacobi_nd, 659
inverse_jacobi_ns, 659
inverse_jacobi_sc, 661
inverse_jacobi_sd, 661
inverse_jacobi_sn, 658
invert, 439
is, 270
isolate, 383
isqrt, 333, 623
ivert, 30
J
jaccobi, 628
jacobi_am, 658
jacobi_cd, 660
jacobi_cn, 658
jacobi_cs, 660
jacobi_dc, 660
jacobi_dn, 658
jacobi_ds, 660
jacobi_nc, 659
jacobi_nd, 659
jacobi_ns, 659
jacobi_sc, 660
jacobi_sd, 660
jacobi_sn, 658
jullia, 677
K
kill, 14, 502, 509
killcontext, 288, 289
kron_delta, 630
k
kron_delta, 630
L
labels, 506
lambda, 313, 581, 926
laplace, 557
last, 405
lcharp, 487
ldefint, 559
ldisp, 693
length, 403
let, 294, 305
letrules, 306
letsimp, 306
lfreeof, 272
lhs, 235, 526
li, 616
limit, 248, 541, 550
linsolve, 27, 532
list_matrix_entries, 434
listarray, 424
listify, 415
listofvars, 382, 915
listp, 402
lmax, 335, 420
lmin, 335
lmix, 420
load, 31, 38, 714, 923, 962
loadfile, 38, 714
local, 583
local(ev 函数の引数), 319
log, 639
logarc, 641
logcontrcat, 641
logout, 708
lopow, 348
lowercasep, 275, 487
lpart, 696
lratsubst, 512
lreduce, 420

lstring, 903
lstringp, 487
lsum, 388, 392
stringp, 275
l
lstring, 489
M
kill, 228
macroexpand, 588
macroexpand1, 588
mainvar, 346
make_array, 425
make_elliptic_e, 661
make_elliptic_f, 661
make_fransform, 740
make_random_state, 336
make_string_input_stream, 484
make_string_output_stream, 484
makegamma, 607, 615
makelist, 400, 609
makeset, 414
mandelbrot, 676
map, 408, 926
mapatom, 408
maplist, 408
matchdeclare, 259, 295
matchfix, 227
matrix, 28, 430
matrixmap, 438
matrixp, 275
mattrace, 438
max, 335
maybe, 270
member, 402, 916
mfuncall, 326
min, 335
minfactorial, 351, 606
minor, 434, 927
mod, 607
mode_declare, 259, 599
mode_identity, 599
modedeclare, 259, 599
moebius, 617
multinomial_coeff, 606
multthru, 522
m
makefact, 606
N
nary, 225, 917
ncharpoly, 441
new, 500
newcontext, 288, 289
newdet, 441
newline, 485
next_prime, 623
niceindices, 392
noeval(ev 函数の引数), 321
nofix, 226
nonscalarp, 275
nonzeroandfreeof, 564
notequal, 210, 267
nounify, 246, 321, 377
nouns(ev 函数の引数), 321
nroots, 531
nterms, 348
nthroot, 351
num, 367
num_distinct_partitions, 422
num_partitions, 422
numberp, 275, 333
numer(ev 函数の引数), 317
numerval, 258
n
numfactor, 618
O
oddp, 274, 333
ode2, 28, 567
op, 236, 386, 387
opena, 484
openplot_curves, 747

openr, 484
openw, 483
optimize, 592
options, 312, 322
orbits, 674
ordergrear, 215
ordergreatp, 216
orderless, 215
orderlessp, 216, 413
outermap, 408
print, 693
P
iprimep, 623
pade, 373
parmanent, 441
parsetoken, 490
part, 386, 512, 514, 691
partfrac, 360
partition, 385
partition_set, 418
permutation, 417
pfet, 520
pickapart, 386
playback, 695
ploarform, 642
plog, 639
plot_format, 11
plot2d, 32, 733
plot2d_ps, 750
plot3d, 33, 736
plotdf, 678
polar_to_xy, 740
poly_discriminant, 358
polydecomp, 351
polymod, 356, 607
ponpoko, 900
postfix, 227
power_mod, 623
powers, 348
powerseries, 371
powerset, 417
pred(ev 函数の引数), 321
prefix, 226, 917
prev_prime, 623
primep, 274, 333
printf, 485
printfile, 719
printlistvar, 349
printprops, 262, 296
product, 247, 388
properties, 237, 261, 296, 314, 955
propvars, 261
pscom, 750
psdraw_curve, 750
psdraw_points, 750
put, 238, 955
p
psi, 619
Q
qput, 238, 601
quit, 15, 43, 708
qunit, 622, 623
quotient, 356
R
radcan, 524
random, 336
random_permutation, 417
rank, 438
rat, 362
ratcoef, 347
ratdenom, 367
ratdiff, 367
ratdisrep, 362
ratexpand, 360
rationalize, 336
ratnumer, 367
ratnump, 275, 343
ratp, 275, 343
ratsimp, 20, 266, 360, 366, 956
ratsubst, 512

ratvars, 342, 349
ratweight, 349, 367
ratweights, 367
read, 714
readline, 485
readonly, 714
realpart, 330, 337
realroots, 27, 530
rearray, 427
rectform, 385
rem, 238
remainder, 356
remarray, 427
remfunction, 590
remlet, 309
remove, 228, 259, 382
remrule, 309
remvalue, 381
reset, 688
reset_verbosely, 688
residue, 559
rest, 405
resultant, 358
return, 573
reveal, 694
reverse, 403
rhs, 235, 526
risch, 26, 549
risch(ev 函数の引数), 318
rk, 676
romberg, 550, 561
room, 705
rootscontract, 350, 367
round, 333
row, 435
rreduce, 420
run_testsuite, 727

r

rembox, 696

S

cunlisp, 489
save, 38, 717
scalarp, 275
scaled_bessel_i, 648
scaled_bessel_i0, 648
scaled_bessel_i1, 648
scaled_bessel_y, 648
scanmap, 408
sconc, 490
sconcat, 482
scopy, 490
scsimp, 524
sdowncase, 494
sec, 632
sech, 632
set_partitions, 422
set_random_state, 336
setdifference, 416
setelmx, 435
setequalp, 423
setify, 414
setp, 423
set_plot_option, 737
setup_autoload, 259, 714
showratvars, 382, 956
showvars, 349
sign, 394
similaritytransform, 447
simplode, 493
simtran, 447
sin, 632
sinh, 632
sinvertcase, 494
slength, 490
smake, 490
smismatch, 497
solve, 27, 535
some, 267, 274
sort, 403
specint, 650

splice, 584
split, 492
sposition, 490
sqfr, 351
sqrt, 335
sremove, 497
sremovefirst, 496
sreverse, 490
ssearch, 498
ssort, 497
sstatus, 704
staircase, 674
status, 703
stirling1, 630
stirling2, 630
strim, 490
string, 482
stringout, 38, 716, 956
stringp, 275, 487
sublis, 512
sublist, 405
submatrix, 434
subset, 418
subsetp, 423
subst, 256, 512, 915, 926, 956
substinpart, 514
substpart, 405, 514, 927
substring, 492
subvarp, 272
sum, 247, 248, 388, 389
sumcontract, 523
sunlisp, 489
supcase, 493
supcontext, 288, 289
surfplot, 959
susbst, 495
susbstfirst, 495
symbolp, 402
symmdifference, 416
system, 37, 708

s

sequal, 489
sequalignore, 489

T

tan, 632
tanh, 632
taylor, 372, 609
taylor_simplifier, 373
taylorinfo, 373
taylorp, 275, 373
taytorat, 372, 373
tcl_output, 694
tellrat, 344
tellsimp, 212, 243, 304
tellsimpafter, 212, 304
tex, 31, 396
texend, 396
texinit, 396
texput, 396
throw, 576
time, 705
timedate, 706
timer, 707
timer_info, 707
tldefint, 559
tlimit, 541
to-maxima, 324
to_lisp, 43, 324
tokens, 494
totaldisrep, 362
totient, 628
tr_warnnings_get, 593
trace, 722
trace_it, 722
translate, 31, 259, 593
translate_file, 593
transplate, 593
transpose, 437
tree_reduce, 420
triangularize, 437

trigexpand, 635
trigrat, 635
trigreduce, 635
trigsimp, 21, 635
trunc, 369
U
ueivects, 448
union, 416
uniteigenvectors, 448
unitvector, 448
unknown, 273
unorder, 215
untimer, 707
untrace, 723
untrllrat, 345
uppercasep, 275, 487
use_fast_arrays, 425
uvect, 448
u
unique, 416
V
verbify, 321, 377
viewps, 750
W
with_stdout, 716
writefile, 37, 693, 716
X
xreduce, 420
xthru, 367
Z
zeroequiv, 271
zeromatrix, 431
zeta, 620
き
行列
Echelon 形式, 437
記号
I
inf, 540
M
minf, 540
グラフ
三次元～表示, 33
二次元～表示, 32
グラフの属性
A
allocation, 819
axis_3d, 823
axis_bottom, 823
axis_left, 823
axis_right, 823
axis_top, 823
B
background_color, 804
C
cbtics, 811
color, 811
columns, 804
D
data_file_name, 807
delay, 804
dimensions, 807
E
eps_height, 807
eps_width, 807
F
file_bgcolor, 804
file_name, 807
fill_color, 836
fill_density, 836
font, 818
font_size, 818
G
gnuplot_file_name, 807
grid, 826
K
key, 830
L
line_type, 831
line_width, 831

logcb, 811
logx, 826
logy, 826
logz, 826
P
pdf_height, 807
pdf_width, 807
pic_height, 807
pic_width, 807
point_size, 838
points_joined, 838
point_type, 838
proportional_axes, 819
R
rot_horizontal, 832
rot_vertical, 832
T
terminal, 804
title, 830
transform, 819
transparent, 836
U
user_preamble, 808
V
view, 832
X
x_range, 832
xaxis, 825
xaxis_color, 825
xaxis_secondary, 825
xaxis_type, 825
xaxis_width, 825
xtics, 826
xtics_axis, 826
xtics_rotate, 826
xtics_secondary, 826
xy_file, 819
xyplane, 832
x
xlabel, 830
Y
y_range, 832
yaxis, 825
yaxis_color, 825
yaxis_secondary, 825
yaxis_type, 825
yaxis_width, 825
ylabel, 830
ytics, 826
ytics_axis, 826
ytics_rotate, 826
ytics_secondary, 826
Z
z_range, 832
zaxis, 825
zaxis_color, 825
zaxis_type, 825
zaxis_width, 825
zlabel, 830
Ztics, 826
ztics_axis, 826
Ztics_rotate, 826
し
色彩の属性, 811
た
多角形の属性, 836
て
点列の属性, 838
せ
絶対経路, 962
そ
相対経路, 962
属性
演算子の属性, 253
函数の属性, 253
記号の属性, 245
数的な属性, 252
A
additive, 247
algebraic, 223

alias, 707
alphabetic, 200, 245
analitic, 253
antisymmetric, 247
argpos, 222
assign, 601, 602
assign-mode-check, 601
atomgrad, 256, 262
atvalue, 262
autoload, 715
B
bindtest, 204, 243
C
commutative, 247
complex, 252
constant, 245
D
data, 281
decreasing, 253
dependency, 256
E
english, 223
even, 252
evenfun, 247, 253
evflag, 243, 314
evflag 属性を持つ大域変数, 314
evfun, 243, 313, 315
evfun 函数の作用の順番, 316
evfun 属性を持つ函数, 315
G
gradef, 256, 262
I
imaginary, 252
increasing, 253
integer, 252
irrational, 252
L
lassociative, 247
linear, 247
logical, 223
lpos, 223
M
mainvar, 245
matchdeclare, 262, 295
mode, 600
mode_check_errorp, 602
mode_check_warnp, 602
mode_checkp, 602
multiplicative, 247
N
nary, 81
nonarray, 243
noninteger, 252
nonscalar, 245
noun, 245
n
nary, 247
O
odd, 252
oddfun, 247, 253
outative, 247
P
pos, 222, 223
posfun, 253
R
rassociative, 247
rational, 252
real, 252
rpos, 223
S
scalar, 245
special, 601
symmetric, 247
T
transfun, 594
U
untyped, 223
V
value_check, 601
大域変数

beta_args_sum_to_integer, 611
beta_expand, 611
%
%%, 503
%, 503
%e_to_numlog, 643
%edispflag, 691
%emode, 517
%enumer, 318, 517
%gamma, 613
%iargs, 634
%num_list, 534
%piargs, 634
%
%, 14
A
absboxchar, 691
activecontexts, 290
algebraic, 342
algedelta, 534
algepsilon, 534
algexact, 534
aliases, 260, 507, 707
arrays, 206, 260, 426, 507
assume_pos, 291
assume_pos_pred, 291
assumescalar, 445
atomgrad, 256
B
backsubst, 527
backtrace, 578
berlefact, 353
besselarray, 649
besselexpand, 649
bftrunc, 331
boxchar, 697
breakup, 536
b
bftorat, 331
C
cauchysum, 389
cfdlength, 622
combineflag, 367
compgrind, 596
context, 290, 291
contexts, 290
current_let_rule_package, 307
D
dblint_x, 563
dblint_y, 563
debugmode, 707
default_let_rule_package, 307
demoivre, 517
dependencies, 256, 257, 260, 507, 550, 557
derivabbrev, 544
derivsubst, 544
detout, 318, 445
dispflag, 578
display_format_internal, 380, 691
display2d, 14, 549, 690, 691
doallmxops, 318, 443
domain, 331, 332
domxexpt, 443
domxmxops, 443
domxnct, 443
dontfactor, 353
doscmxops, 318, 443
doscmxplus, 443
dot0nscsimp, 231
dot0simp, 231
dot1simp, 231
dotassoc, 231
dotconstrules, 231
dotdistrib, 231
dotexptsimp, 231, 545
dotident, 231
dotsassoc, 545
dotscrules, 231, 545

draw_command, 871
draw_pipes, 871
draw_renderer, 795
E
ecm_limit, 626
ecm_limit_delta, 626
ecm_max_limit, 626
ecm_number_of_curves, 626
erfflag, 551
error, 698
error_size, 698
error_syms, 698
errorfun, 578
errormsg, 698
expon, 317
expop, 317
exptdispflag, 691
exptisolate, 384
exptsubst, 512
F
facexpand, 353
factorflag, 353
features, 241
file_output_append, 717
file_search_demo, 711
file_search_lisp, 711
file_search_maxima, 711
file_search_path, 981
file_search_usage, 711
file_search_demo, 699, 701
float, 318, 518
float2bf, 331
fortindent, 399
fortspaces, 399
fppintprec, 330, 331
fpprec, 329, 331
functions, 206, 508, 582, 722
f
factlim, 615
G
gcd, 354
genindex, 389
gensumnum, 389
globalsolve, 527
gnuplot_view_args, 787
gradefs, 256, 257, 260, 508
g
gammalim, 615
H
halfangles, 634
help, 701
hermitianmatrix, 446
I
iarray, 649
ibase, 691
in_netmath, 752
inchar, 503
inflag, 382, 401, 514
infolists, 507
integrate_use_rootsof, 553
integration_constant_counter, 551
intfaclim, 353
isolate, 384
i
ifactor_verbose, 626
ifactors_only, 626
K
keepfloat, 365
knowneigvals, 446
knowneigvects, 446
L
labels, 503, 508
lasttime, 705
leftjust, 691
leftmatrix, 446
let_rule_packages, 307
let_rule_packages, 508
letrar, 307
letrat, 305
lhospitallim, 542

ligarc, 643
limsubst, 542
linechar, 503
linel, 691
linenum, 503
linsolve_params, 532
linsolvewarn, 532
lispdisp, 325, 691
listarith, 401
listconstvars, 383
listdummyvars, 383
listeigvals, 446
listeigvects, 446
lmxchar, 430
loadprint, 715
logabs, 643
logconcoeffp, 643
logexpand, 643
lognegint, 643
lognumer, 643
logsimp, 643
M
m1pbranch, 331
macroexpansion, 588
macros, 206, 508, 585, 588
manual_demo, 700
manual_demo, 701
maperror, 408
matrix_element_add, 444
matrix_element_mult, 444
matrix_element_transpose, 444
maxapplydepth, 302
maxapplyheight, 302
maxima_tempdir, 712
maxnegex, 317, 521
maxpogex, 521
maxposex, 317
maxtayorder, 373
modulus, 342, 607
multiplicities, 527
mx0simp, 445
mymacros, 508
myoptions, 707
N
negdistrib, 518
negsumdispflag, 229
newfac, 353
niceindicespres, 393
nolabels, 503
nondiagonalizable, 446
noundisp, 691
numer, 318, 518
O
obase, 691
opsubst, 512
optimprefix, 593
optionset, 707
outchar, 503
P
packagefile, 715
partswitch, 387
pfeformat, 691
piece, 387
plot_options, 737
～の colour_z, 739
～の gnuplot_curve_styles, 745
～の gnuplot_curve_titles, 745, 756
～の gnuplot_default_term_command, 742
～の gnuplot_dumb_term_command, 743
～の gnuplot_out_file, 742
～の gnuplot_pipe_term, 744
～の gnuplot_pm3d, 745
～の gnuplot_preambles, 744
～の gnuplot_term, 742
～の grid, 739
～の logx, 741
～の logy, 741
～の nticks, 739

〜の plot_format, 738
〜の run_viewer, 738
〜の t, 740
〜の transform_xy, 740
〜の view_direction, 739
〜の x, 739
〜の y, 739
polar_rho_limit, 626
polar_rho_limit_step, 626
polar_rho_tests, 626
polyfactor, 531
powerdisp, 691
prederror, 203, 265, 270
prevfib, 614
primep_number_of_tests, 626
prompt, 503, 689, 696
props, 204, 237, 254, 261, 508
ps_scale, 752
ps_translate, 752
psexpand, 366
pstream, 751
R
radexpand, 331
radsubstflag, 513
rataigdenom, 365
ratdenomdivide, 366
ratepsilon, 332, 365
ratexpand, 366
ratfac, 366
ratmx, 445
ratprint, 365
ratsimpexpons, 366
ratweights, 366, 367
ratwtlvl, 366
readonly, 534
resultant, 359
rightmatrix, 446
rmxchar, 430
rombergabs, 562
rombergit, 562
rombergmin, 562
rombergtol, 562
rootsconmode, 366
rootsepsilon, 531
rpgrammode, 527
rules, 261, 298, 508
S
save_primes, 626
savedef, 596
savefactors, 353
scalarmatrix, 445
setcheck, 233, 725
setcheckbreak, 725
setval, 725
show_openplot, 752
showtime, 705
simp, 211, 215, 243, 311, 312, 517, 518
simpproduct, 388, 518
simpsum, 389
solve_inconsistent_error, 536
solvedecomposes, 536
solveexplicit, 536
solvefactors, 536
solvenullwarn, 536
solveradcan, 536
solvetrigwarn, 536
sparse, 445
sqrtdispflag, 335
stardisp, 691
stringdisp, 691
structures, 500
sublis_apply_lambda, 514
sumexpand, 389, 518
sumsplitfact, 351, 353
superlogcon, 643
T
taylor_coefficients, 373
taylor_logexpand, 373
taylor_truncate_polynomials, 373
taylordepth, 373

timer_devalue, 707
tlimswitch, 542, 561
tr_array_as_ref, 597
tr_bound_function_applyp, 599
tr_file_tty_messagesp, 599
tr_float_can_branch_complex, 599
tr_function_call_default, 597
tr_numer, 597
tr_optimize_max_loop, 599
tr_semicompile, 597
tr_warn_bad_function_calls, 597, 599
tr_warn_fexpr, 597
tr_warn_meval, 597
tr_warn_mode, 597
tr_warn_undeclared, 597
tr_warn_undefined_variable, 597
trace, 725
trace_break_arg, 725
trace_max_indent, 725
trace_safety, 725
transcomile, 597
translate, 596
translate_fast_arrays, 597
transrun, 596
trigexpandplus, 634
trigexpandtimes, 634
triginverses, 634
trigsign, 634
ttyoff, 691
U
undeclaredwarn, 596
undeclearewarn の設定項目, 597
V
values, 261, 381, 508
vect_cross, 547
W
window_size, 752
Y
yarray, 649
Z
zeobern, 610
大域変数 F
features, 260
大域変数 P
prederror, 284
対象
B
～構文, 852
～の属性, 852
～の fill_color, 852
～の fill_density, 852
～の key, 852
～の line_width, 852
～の xaxis_secondary, 852
～の yaxis_secondary, 852
C
～の構文, 869
～の color, 869
～の enhanced3d, 869
～の line_type, 869
～の line_width, 869
～の wired_surface, 869
～の xu_grid, 869
～の yv_grid, 869
～文, 869
c
～の属性, 869
E
～文, 860
～の属性, 861
～の color, 861
～の enhanced3d, 861
～の line_type, 861
～の line_width, 861
～の wired_surface, 861
～の x_voxel, 861
～の y_voxel, 861
～の z_voxel, 861
ellipse, 849
～構文, 849

〜文, 849
〜の属性, 849
〜の border, 849
〜の color, 849
〜の fill_color, 849
〜の key, 849
〜の line_type, 849
〜の line_width, 849
〜の nticks, 849
〜の transform, 849
〜の transparent, 849
〜の xaxis_secondary, 849
〜の yaxis_secondary, 849
〜構文, 842
〜文, 842
〜の固有の属性, 843
〜の属性, 843
〜の color, 843
〜の error_type, 843
〜の fill_density, 843
〜の key, 843
〜の line_width, 843
〜の points_joined, 843
〜の xaxis_secondary, 843
〜の yaxis_secondary, 843
explicit, 854
〜固有の属性, 855
〜の構文, 855
〜の属性, 857
〜の adapt_depth, 855
〜の color, 857
〜の fill_color, 857
〜の filled_func, 855
〜の key, 857
〜の line_type, 857
〜の line_width, 857
〜の nticks, 857
〜文, 854
explicit3d, 854
〜の構文, 855
〜の属性, 857
〜の color, 857
〜の contour, 857
〜の contour_levels, 857
〜の enhance3d, 857
〜の key, 857
〜の line_type, 857
〜の line_width, 857
〜の xu_grid, 857
〜の yv_grid, 857
G
gr2d, 799
gr3d, 799
I
image, 863
〜構文, 864
〜の属性, 864
〜の colorbox, 864
〜の palette, 864
〜文, 863
implicit1d, 858
implicit3d, 858
〜固有の属性, 859
〜の構文, 859, 860
〜の属性, 860
〜の color, 860
〜の enhanced3d, 860
〜の ip_grid, 859
〜の ip_grid_in, 859
〜の line_type, 860
〜の line_width, 860
〜の wired_surface, 860
〜の x_voxel, 860
〜の y_voxel, 860
〜の z_voxel, 860
〜文, 858
L
label, 849
〜文, 849
〜構文, 849

〜の属性, 850
〜の label_alignment, 850
〜の label_orientation, 850
label3, 849
〜構文, 849
〜の属性, 851
〜の color, 851
〜の xaxis_secondary, 851
〜の yaxis_secondary, 851
M
〜の構文, 865
〜の属性, 865
〜の color, 865
〜の enhanced3d, 865
〜の line_type, 865
〜の line_width, 865
〜の transform, 865
〜の wired_surface, 865
P
parametric, 865
parametric_surface, 869
〜構文, 869
〜の属性, 869
〜の color, 869
〜の key, 869
〜の line_type, 869
〜の line_width, 869
〜の xu_grid, 869
〜の yv_grid, 869
〜文, 869
〜の構文, 865
〜文, 865
parametric3d, 865
〜の属性, 866
〜の color, 866
〜の enhanced3d, 866
〜の key, 866
〜の line_type, 866
〜の line_width, 866
〜の nticks, 866
points, 840
points3d, 840
〜の属性, 841
〜の point_size, 841
〜の point_type, 841
〜の共通の属性, 841
〜構文, 840
〜の color, 841
〜の enhanced3d, 841
〜の key, 841
〜の line_width, 841
〜の points_joined, 841
〜の transform, 841
〜の xaxis_secondary, 841
〜の yaxis_secondary, 841
〜文, 840
polar, 867
〜の構文, 867
〜の属性, 867
〜の color, 867
〜の key, 867
〜の line_type, 867
〜の line_width, 867
〜の nticks, 867
〜文, 867
polygon, 844
〜の共通の属性, 844
〜の構文, 844
〜の border, 844
〜の color, 844
〜の fill_color, 844
〜の key, 844
〜の line_type, 844
〜の line_width, 844
〜の transform, 844
〜の transparent, 844
〜の xaxis_secondary, 844
〜の yaxis_secondary, 844
〜文, 844
Q

〜の属性, 847
〜の color, 847
〜の enhanced3d, 847
〜の key, 847
〜の line_type, 847
〜の line_width, 847
〜の transform, 847
〜の属性, 846
〜の border, 846
〜の color, 846
〜の fill_color, 846
〜の key, 846
〜の line_type, 846
〜の line_width, 846
〜の transform, 846
〜の transparent, 846
〜の xaxis_secondary, 846
〜の yaxis_secondary, 846
〜文, 846
〜の構文, 846

R

〜文, 847
rectangle, 847
〜の共通の属性, 848
〜の構文, 848
〜の border, 848
〜の color, 848
〜の fill_color, 848
〜の key, 848
〜の line_type, 848
〜の line_width, 848
〜の transform, 848
〜の transparent, 848
〜の xaxis_secondary, 848
〜の yaxis_secondary, 848
region, 857
〜の構文, 857
〜の属性, 857
〜の fill_color, 857
〜の key, 857
〜の x_voxel, 857
〜の y_voxel, 857

S

〜の構文, 868
〜の属性, 868
〜の color, 868
〜の enhanced3d, 868
〜の line_type, 868
〜の line_width, 868
〜の wired_surface, 868
〜の xu_grid, 868
〜の yv_grid, 868
〜文, 868

T

〜の共通の属性, 845
〜の border, 845
〜の color, 845
〜の fill_color, 845
〜の key, 845
〜の line_type, 845
〜の line_width, 845
〜の transform, 845
〜の transparent, 845
〜の xaxis_secondary, 845
〜の yaxis_secondary, 845
〜文, 845, 852, 865
〜の構文, 845
〜固有の属性, 870
〜の構文, 870
〜の属性, 871
〜の color, 871
〜の enhanced3d, 871
〜の line_type, 871
〜の line_width, 871
〜の surface_hide, 871
〜の wired_surface, 871
〜の xu_grid, 871
〜の yv_grid, 871
〜文, 870

V

vector, 853
〜の構文 (2d) , 853
〜の構文 (3d) , 853
〜の属性, 853
〜の color, 854
〜の head_angle, 853
〜の head_both, 853
〜の head_length, 853
〜の head_type, 853
〜の key, 854
〜の line_type, 854
〜の line_width, 854
〜の unit_vectors, 853
〜のそのほかの属性, 854
〜文, 853
vector3d, 853
き
曲線・曲面近似の属性, 834
け
形状の属性 border, 836
形状の属性 contour, 811
形状の属性 contour_level, 811
形状の属性 enhanced3d, 811
形状の属性 nticks, 834
形状の属性 wired_surface, 811
形状の属性 xu_grid, 834
形状の属性 x_voxel, 834
形状の属性 yv_grid, 834
形状の属性 y_voxel, 834
形状の性属 z_voxel, 834
定数
%e, 318, 332
%gamma, 332
%phi, 332
%pi, 332
false, 202
inf, 332, 550
infinity, 332, 550
minf, 332
off(=false), 202
on(=true), 202
true, 202
unknown, 203
zeroa, 332
zerob, 332
な
内部表現, 374
内部函数
$quit, 43
assign-mode-check, 602
D
dcompare, 284
deqf, 284
dgrf, 284
defprop, 248, 257, 311, 396
defprop 函数の operators 属性, 517
defprop の operator 属性, 311
kind, 250
kindp, 251
lisp-implementation-type, 725
lisp-implementation-version, 725
oper-apply, 243
P
proc-$dfefrule, 299
par, 250
prop1, 237
R
rishint, 318
S
sign-any, 291
simpcospsimp-%cos, 311
sinint, 318
simplifya, 243, 303, 517
内部変数
A
alphabet, 200, 260
autoconf-host, 725
autoconf-version, 725
F
features, 703

fixnbound, 202
flounbound, 202
M
maxima-build-time, 725
maxima-epilog, 709
maxima-tesitdir, 727
maximae-demodir, 700
O
*opers-list, 242
opers, 242
R
ratweights, 509
S
sign-, 292
U
unbound, 255
V
variable-initial-valuses, 688
ファイル
AlexanderPolyl.mc, 924
fox.mc, 922
maxima-init.mac, 923, 955, 1037
maxima-init.mac の置き場所, 1040
ponpoko.lisp, 898
surfplot, 962
surfplot.mc, 959
グラフ
maxout.geomview, 731
maxout.gnuplot, 731
maxout.gnuplot_pipes, 731
maxout.openmath, 731
初期化ファイル (maxima-init.mac), 955
文脈
子文脈の生成, 289
〜の切替, 291
〜の削除, 289
〜の生成, 289
〜を切離す, 289
〜を借用する, 289
global, 286
initial, 22, 286
ほ
方程式, 526
A
α-交換, 179
α-同値, 179
arity, 581
B
Bell 数, 608
C
Cantor の定理, 115
Cauchy 列, 105
Church
〜の真理値, 142
〜の提唱, 99
D
Dehn
〜の補題, 913
De Morgan の法則, 77, 133
E
EOF, 46
$\varepsilon - \delta$ 論法, 116
external import, 130
F
Fox の微分子, 914
H
Hilbert 計画, 122
Honer 則, 359
Hurwiz の定理, 933
J
Jacobi 記号, 629
Julius Caesar 問題, 154
K
Klein の壺, 33
L
λ – 計算, 52
Lgendre 平方剰余記号, 629
M
Mersenne Twister 法 (MT 法), 336
Möbius の輪, 34

N
New Math, 195
P
Peano 曲線, 93
PID(=主イデアル整域), 75
Pythagoras 学派, 102
S
scope, 177
Seifert 曲面, 947
SQUARE, 129
T
term, 167
Tietze 変換, 913
Turing 機械, 190
い
意義 (=Sinn), 140
イデアル
イデアル, 75
極大イデアル, 75
素イデアル, 75
単項イデアル, 75
左イデアル, 75
右イデアル, 75
両側イデアル, 75
意味 (=Bedeutung), 140
意味の木, 288
因明, 123
え
演算
〜が閉じている, 70
可換, 71
逆元, 70
結合律, 70
正則元, 70
前置表現, 80
単位元, 70
中置表現, 80
内挿表現, 80
非可換, 71
左分配律, 74
分配律, 74
ポーランド, 80
右分配律, 74
お
黄金数, 103
か
概念
Begriff, 141
Concept, 167
〜に属する対象, 141
〜の外延, 141
外延, 60
下位概念, 60
概念, 60
〜の外延 (=クラス), 154
Γ-概念, 141
基数の〜, 157
個体, 60
個体概念, 60
種概念, 60
上位概念, 60
属性, 60
単独概念, 60
徴表, 60
内包, 60
内包外延反比例増減の法則, 60
範疇, 60
明晰な概念, 60
明瞭な概念, 60
明確な概念, 60
類概念, 60
概念記法, 135
移行記号, 139
条件線, 136
水平線, 135
対偶変換, 150
内容線, 135
判断線, 136
否定, 136
融合, 136

解の自動代入, 528
環
環, 73
可換環, 74
局所化, 77
局所環, 75
群環, 75
斜体, 77
主イデアル整域, 75
商環, 78
剰余環, 78
整域, 74
多元環, 75
多項式環 , 75
標準基底, 87
零因子, 74
関係, 142
α 同値, 190
関係, 65
逆関係, 156
強先祖関係, 160
後者, 95, 192
後続, 159
弱先祖関係, 160
直続, 158
包含関係, 62
函数記号, 176
き
片側曲面, 937
記号
n 項関係記号, 192
類記号, 192
基数
基数, 61, 92, 110, 156
有限基数, 160
有限基数のクラス, 163
規則, 65
約分, 64
帰納法
一般帰納的函数, 99
帰納的定義, 97
帰納法の原理, 96
原始帰納的函数, 97
再帰的, 97
始函数, 97
基本列, 105
逆理
Banach-Tarski の逆理, 187
Cantor の〜, 115, 118
Richard の〜, 114, 117
Russell の〜, 113, 118
Russell の〜, 164
意味論的〜, 115
うそつきの〜, 113
クレタ人の〜, 113
床屋の〜, 114
論理的〜, 115
行列, 28
単位行列, 431
極, 7
非本質的な曲線, 937
本質的な曲線, 937
両側曲面, 937
く
クラス
クラス, 61, 141, 186
類, 61, 186
群, 930
可換群, 71
〜表示, 907
Wirtinger 表示, 907
関係子, 907
関係子 (=relator), 72
語, 72
自由群, 72, 907
準群, 70
剰余群, 73
正規部分群, 73
生成元, 72
半群, 70

非可換群, 71
け
形式主義, 118
言語
函数型, 8
スタック型, 44
手続型, 8
論理型, 8
原子, 324
原理
Hume の原理, 154, 157
悪循環原理, 119, 171
帰納法の原理, 96
区間縮小法の原理, 110
文脈原理, 21, 140
こ
項, 176
ξ-項場所, 141, 142
ζ-項場所, 142
項, 141, 167
項場所, 141
項順序, 215
公理
Archimedes の公理, 110
Zorn の補題, 187
外延公理, 185
還元可能性公理, 119, 172, 207
空集合公理, 185
集合の内包公理, 192
正則性公理, 185
選択公理, 185
代入則, 152
置換公理, 185
対公理, 185
二重否定の除去, 146
排中律, 120, 131, 152
分出公理, 186
冪集合公理, 185
無限集合公理, 185
矛盾律, 131
和集合公理, 185
公理系
BG-公理系, 185
Peano の公理系, 95
ZFC-公理系, 185
概念記法の公理系, 144
算術の基本法則の公理系, 151
自然数の公理系, 95
実数の公理系, 109
個体 (=individual), 167
ご
語, 63
さ
三段論法, 130
Modus Ponens(=前提肯定), 138
Modus Ponense, 125
仮言三段論法, 130, 149
後件肯定, 139
選言三段論法, 130
前提肯定 (MP), 125, 138, 283
定言三段論法, 130
ディレンマ, 130
両刀論法, 130
し
CRE 表現, 341
式
monic な多項式, 89
最小多項式, 89
正準表現, 86
代数方程式, 89
多変数多項式の正準表現, 86
筆頭項, 88
木構造, 80
縮小写像, 670
順序
順序の定義, 82
辞書式順序, 214
(Maxima の) 次数リスト, 214
思想 (=Gedanke), 140
写像

ambient isotopy, 906
核, 73
準同形写像, 73
全射, 78
全単射, 78
単射, 78
同型, 78
集合
Dedekind 無限集合, 93
外延的定義, 61, 129
可算無限集合, 92
可附番集合, 92
共通集合, 62
空集合, 62
元, 61
差集合, 62
集合, 61
集合の表記, 61
真部分集合, 62
成員, 61
整列集合, 187
積閉集合, 77
対集合, 185
内包的定義, 61, 129
部分集合, 62
冪集合, 62
補集合, 62
有限集合, 62
和集合, 62
述語
tautology(=恒真式), 264
可述的函数, 169
原始帰納的述語, 193
恒真式, 264
充足可能な述語, 264
充足不可能な述語, 264
述語, 61, 264
述語記号, 178
第 1 階述語, 119
表現函数, 193, 274
命題, 264
主変数, 246
mainvar, 213
～の宣言, 245
順序, 82
順序集合, 82
逆辞書式順序, 84
項順序, 83
斉次逆辞書式順序, 84
斉次辞書式順序, 84
辞書式順序, 83
推移律, 82
整列集合, 99, 111
整列順序, 99
整列性, 111
全順序, 82, 214
全順序集合, 82, 214
対称律, 82
反射律, 82
半順序, 82
順序数
順序数, 110, 111
超限順序数, 112
有限順序数, 112
証明法
帰謬法 (=背理法), 120
数学的帰納法, 96
対角線論法, 93, 114
背理法, 120
除去可能な極, 7
真理函数, 271

す

数
Gauß整数, 89
代数的数, 89
代数的整数, 68, 89
超越数, 89
有理数, 65
数学の危機, 118
数学の現代化, 195

数論的函数, 176
スケイン関係式, 948
スタックマシン, 44
せ
切断
上組, 106
下組, 106
隙間, 106
正常, 106
切断, 106
跳躍, 106
前件肯定, 139
全称記号, 142
選択公理, 187
前提肯定, 139
全微分, 543
そ
属性
作用に関連する属性, 247
～を追加, 241
属性の表現函数, 237
存在含意, 130
た
体
体, 77
対当
対当関係表 (SQUARE), 129
小反対対当, 130
大小対当, 130
反対対当, 130
矛盾対当, 130
対話処理言語, 13
楕円積分, 652
楕円無理函数, 652
多項式
最小多項式, 342
多項式と単項式の一般表現, 340
多様体, 913
単変数多項式の CRE 表現, 341
断面, 968
～の方程式, 968
単連結, 913
ち
値域
重積値域, 142
値域, 141
無気息記号の作用域, 141
中間元, 106
稠密, 106
重複度のリスト, 529
直感主義, 118
て
定理
Pythagoras の定理, 102
Tikhonov の定理, 187
三平方の定理, 102
整列可能性定理, 187
手続, 65
α 変換, 190
Euclid の互除法, 100
Tietze 変換, 907
互除法, 100
と
同値関係
反射律, 65
対称律, 65
推移律, 65
同値関係, 65
同値類, 65
同値類の代表, 65
に
二項函数, 142
の
濃度
可算個, 92
可算濃度, 92
可附番濃度, 92
濃度, 61, 110
濃度 (=基数), 92
は

判断
肯定判断, 129
全称肯定判断, 129
全称判断, 129
全称否定判断, 129
特称肯定判断, 129
特称判断, 129
特称否定判断, 129
判断, 129
否定判断, 129
ひ
非共測, 101
微分方程式
初期条件, 28
び
微分方程式
常微分方程式, 28
ふ
普遍例化, 152
フラクタル次元, 93
文脈, 21, 277
分離規則, 139
へ
平面
平面の方程式, 968
変項, 61
作用範囲, 142, 177
実際の変項 (real variable), 167
自由変項, 142, 175, 190
束縛変項, 142, 175, 190
明瞭な変項 (=Apparent variable), 168
変数
～順序, 214, 956
ほ
方程式, 27
母数, 652
母体, 169
む
無限小, 116
無限小解析, 116
無限大, 7
結び目
～群の表示, 908
～の Alexander 多項式, 923
～の Alexander 多項式, 908
～の Reidemeister 移動, 909
～の下道, 909
結び目の交点の総数, 907
～の射影図, 909
結び目の射影図, 906
～の上道, 909
～の正則な射影図, 909
～の符号和, 911
～の連結和, 930
埋め込み, 905
管状近傍, 906
交差点, 909
交点の符号, 911
順な結び目, 906
補空間, 905
野性的な結び目, 906
無平方, 352
無平方因子分解, 352
め
命題
可述的, 117, 170
基本命題 (=Elementary proposition), 168
基本命題函数, 168
繋辞 (=copula), 128
原始命題, 174
主語 (=subject), 128
述語 (=predicate), 128
非可述的命題, 119
命題, 128
命題記号, 178
命題の判断, 129
命題函数, 167
+mbook もくこうそう
木構造

親, 82
子, 82
節, 82
根, 82
葉, 82
木構造, 82
ゆ
有限の立場, 122
よ
予定論, 127
り
力学系
chaos, 669
Hénon 写像, 665
Poincaré 写像, 663
Poincaré 平面, 663
strange attractor, 668
軌道, 663
周期的ウィンドウ, 669
分岐図, 665
力学系, 663
離散力学系, 663
連続力学系, 663
れ
連続
Aristotle の連続, 107
Dedekind の意味で, 107
連続性, 93
連立方程式, 27, 526
ろ
ローマ曲面, 968, 971
論理函数, 176
論理記号
含意, 177
選言, 176
全称記号, 177
存在記号, 142, 177
同値, 177
否定, 176
連言, 176
ろんりしき
論理記号, 176
論理式
Horn 節, 6
Skolem の標準形, 188
演繹定理, 181
書換, 179
基本論理式, 191
恒真式 (=tautology), 145
tautology(=恒真式), 145
公理, 180
個体記号, 175
個体変項, 175
終式, 181
述語, 178
述語計算の論理式, 178
初式, 180
節 (clause), 178
代入, 179
導出可能, 181
判断, 264
文論理式, 192
閉論理式, 192
変項, 175
本来の公理, 180
リテラル, 178
論理計算の論理式, 178
論理式の持ち上げ, 192
論理式変項, 175
論理主義, 118
き
開曲面, 937
閉曲面, 937
く
レンズ空間, 946
記号
~, 65
+, 71
., 71
*, 71

card(M), 61
=, 61
∃, 142
∀, 142
⌒, 155
⌓, 163
\, 141
$\mathfrak{p}$, 157
⇸, 156
$\mathfrak{f}$, 159
⚹, 156
⌴, 156
∈, 61
ι, 141, 153
Ip, 155
ker, 73
LM, 88
LT, 88
ω, 112
$\mathfrak{P}(S)$, 62
$\mathbb{R}[x]$, 67
$> qq\xi$, 155
⌣̸, 160
⊂, 62
⊆, 62
$X/\sim$, 65
$\aleph$, 92
$\aleph_0$, 92
⊢, 136
¬, 136
⊨, 136
人名
A
Aristotle, 107, 113
Averroës(=Ibn Rushd), 128
Avicenna(=Ibn Sina), 128
B
Boole, 118, 133
Brouwer, 120
Buchberger, 87
C
Cantor, 92, 93, 105, 109, 110, 113–115, 118
Cauchy, 105, 116
Chrysippos, 124
D
De Morgan, 132
Dedekind, 93, 106, 116
Dehn, 907
E
Epimenides, 113
F
Frege, 94, 107, 117, 118, 135
H
Heyting, 121
Hilbert, 87, 116, 118, 120, 122, 152, 175
Hippasus, 102
K
Kronecker, 120
L
Leibniz, 94, 131
N
Newton, 132
P
Peano, 95, 115, 118
Poincaré, 113, 117, 119
R
Ramsey, 115
Richard, 114
Rushd, 128
Russell, 113, 117, 119, 164
S
Schelter, 2
Sina, 128
Steiner, 965
T
Theoddoros, 104
W
Weierstrass, 116

Wirtinger, 907
に
西村拓士, 336
ぼ
墨子, 123
ま
松本眞, 336
Soffware
Application
REDUCE, 8
SAGE, 8
Software
Application
AutoCAD, 42
AXIOM, 8
Cantor, 11
Common Lisp, 3
DERIVE, 8
Euler Math Toolbox, 987
FreeMat, 988
GCL, 2
Geomview, 36, 731
GNU Emacs, 42
gnuplot, 12, 35, 731
kan/sm1, 44
Macaulay2, 12
MACSYMA, 2
Maple, 8, 42
MathCAD, 8
Mathematica, 8, 42
MATLAB, 8
MuPAD, 8
Octave, 12, 988
openmath, 35, 731
PARI/GP, 12
R, 8
Risa/Asir, 12
S, 8
SAGE, 2
SciLab, 988
SINGULAR, 217, 972
S-PLUS, 8
surf, 951
surfer, 37, 951
TeXmacs, 12
VirtualBox, 3
VMware Player, 3
wxMaxima, 10
xmaxima, 10
Yorick, 988
カルキング, 8
OS
Debian, 3, 4
FreeBSD, 3
KNOPPIX, 3
KNOPPIX/Math, 2, 3
KNOPPIX Edu, 2
Linux, 3
MacOSX, 3
MS-Windows, 3
OpenSUSE, 4
RedHat, 3
Slackware, 3
Solaris, 3
Ubuntu, 2
Vine, 4
言語
Allegro Common Lisp, 58
CLISP, 3, 43
CMUCL, 58
FORTRAN, 42
GCL, 42
InterLisp, 42
KCL(Kyoto Common Lisp), 42
LISP, 42
MACLisp, 42
OpenMCL, 58
PostScript, 44
Prolog, 5
PS, 44

SBCL, 3, 58
Scheme, 42
GNUPLOT
演算子
~, 783
*, 781
**, 781
+, 781
-, 781
/, 781
%, 781
!=, 783
!, 781, 783
¡=, 783
¡, 783
=, 783
¿=, 783
¿, 783
&&, 783
&, 783
{}, 785
——, 783
—, 783
記号
'-', 755
数学函数
abs, 781
acos, 782
acosh, 782
arg, 782
asin, 782
asinh, 782
atan, 782
atan2, 782
atanh, 782
besj0, 782
besj1, 782
besy0, 782
besy1, 782
ceil, 782
cos, 782
cosh, 782
erf, 782
erfc, 782
exp, 782
floor, 782
gamma, 782
ibeta, 782
igamma, 782
imag, 781
int, 782
invert, 782
invnorm, 782
lambertw, 782
log, 782
log10, 782
norm, 782
rand, 781
real, 781
sgn, 782
sin, 782
sinh, 782
sqrt, 781
tan, 782
tanh, 782
A
arrow, 779
auto, 765
B
b, 757, 758
both, 763
bspline, 765
C
cblabel, 768
cbrange, 767, 768
cntrparam, 765
contour, 762, 763
base, 762
both, 762
surfacee, 762

cubicspline, 765
H
hidden3d, 769
I
if, 786
if 文の構文, 786
incremental, 765
isosamples, 772
K
key, 756, 776
L
label, 778
levels, 765
linear, 765
M
map, 759
P
palette, 760
color, 761
gray, 761
negative, 761
positive, 761
rgbformulae, 762
plot, 755
dots, 757
impulse, 757
lines, 757
linespoints, 757
point, 757
plot 命令の構文, 755
pm3d, 757
pm3d の設定, 757
print, 784
R
replot, 761
reread, 786
S
s, 758
sample, 772
set, 754
show, 754
show palette, 761
splot, 757
splot 命令の構文, 757
surface, 763, 769
T
term, 790
ticslevel, 760
U
unset, 754
V
view, 766, 767
W
with, 756
X
xrange, 773
Y
yrange, 773
Z
zrange, 773
い
陰線処理, 769
か
階調の調整, 767
函数の定義, 783
き
曲面を面を張る, 758
曲面の上面への投影, 758
曲面の底面への投影, 757
曲面の様式, 757
し
条件分岐, 786
真理値, 785
せ
整数, 785
と
等高線, 759
等高線の高度, 765
等高線の設定, 762
等高線の調整, 765

等高線の補間式, 765
は
反復処理, 786
凡例, 756
ひ
表題, 757
ふ
複素数, 785
浮動小数点数, 785
LISP
A
acos, 45
alhpha-char-p, 488
alphanumericp, 488
and, 49
append, 48
apply, 52
aref, 49
asin, 45
atan, 45
C
caar, 48
cadr, 48
car, 47
cddr, 48
cdr, 48
char-code, 46
char-int, 488
char=, 488
characterp, 487
close, 55
code-char, 46
concatenate, 46
cond, 49, 53
cons, 48
cos, 45
D
defun, 47, 53
deof, 54
dribble, 37
E
eval, 50
exp, 45
F
file-length, 484
format, 57, 485
fresh-line, 485
G
get, 54, 221
get-decoded-time, 706
get-internal-real-time, 706
get-universal-time, 706
gethash, 50
graphic-char-p, 488
great, 213, 216
I
:initial-element, 49
if, 49, 53
inof, 54
L
lambda 式, 52
let, 50
let*, 51
load, 55
log, 45
lower-case-p, 488
M
make-array, 49
make-hash-table, 50
mapcar, 52
most-negative-double-float, 202
most-negative-fixnum, 202
N
nil, 46
O
open, 55, 483
or, 49
P
pop, 54
prin1, 57

princ, 57
print, 56, 57
push, 54
P リスト, 54
ut, 223
R
read, 56
room, 705
S
setf, 50, 54
setq, 50
sin, 45
string, 46
stringp, 487
symbol-plist, 54
S 式, 46
T
t, 46
tan, 45
terpri, 485
U
upper-case-p, 488
LISP の記号
*, 45
+, 45
-, 45
/, 45
え
S 式, 324
き
局所変数, 51
け
原子, 44
さ
三角函数, 45
し
指数函数, 45
四則演算, 45
述語, 49
剰余, 45
そ
属性リスト, 54
た
対数函数, 45
は
配列, 49
ハッシュ表, 50
ふ
複素数, 46
も
文字データ, 46
文字列, 46
り
リスト, 46, 47
ろ
論理積, 49
論理和, 49
SINGULAR
E
eliminate, 976
G
groebner, 973
L
LIB, 975
M
map, 974
P
preimage, 975
S
setring, 974
std, 973
surf.lib, 975
い
イデアル, 973
イデアルの定義, 973
き
基礎環, 973
基礎環の係数体, 973
基礎環の定義, 973
し

写像, 974
商環, 973
商環の定義, 973
た
多項式, 973
多項式の定義, 973
れ
零イデアル, 975
Octave
表記
a(:,j), 996
a(i,:), 996
a(i,j), 996
a(i:j), 996
A
all, 1007
any, 1007
C
cd, 1013
conv, 1009
D
deconv, 1009
diag, 1008
E
e, 992
eps, 994
exec, 1012
eye, 1009
F
fclose, 1021
fgets, 1021, 1022
find, 1003
fopen, 1021, 1022
for, 992
for 文による反復処理, 1002
fprintf, 1027
frewind, 1021, 1022
fscanf, 1021
G
grid, 1014
H
help, 988
hold, 1014
I
i, 992
is_struct, 1026
L
length, 995
load, 1017
load -force, 1017
ls, 1013
M
M-file, 988, 1010
O
Octave の演算, 999
P
pi, 992
plot, 1014
polyval, 1009
pwd, 1013
R
rand, 1002
S
save, 1018
save -append, 1019
size, 995
sscanf, 1022
sscanf の型の指定, 1023
struct_elements, 1026
system, 1012
T
title, 1014
type, 993, 1011
W
while, 994
who, 993, 1012, 1018
X
xlabel, 1014
Y
ylabel, 1014

Z
zlabel, 1014
Octave の記号
\, 999
!, 1012
', 999
*, 999
+, 999
,, 995
-, 999
.*, 999
./, 999
.^, 999
/, 999
:, 996, 997, 999
;, 992, 995
=, 992
^, 999
お
オンラインヘルプ, 988
き
行列の書式, 994
行列の成分, 996
偽, 1003
行列, 994
く
組込函数, 1012
グラフ
グラフに網目, 1014
グラフにラベルや表題を入れる, 1014
グラフの重ね描き, 1014
け
計算時間, 1002
こ
構造体, 1025
し
四則演算, 992
真, 1003
た
多項式の商と剰余, 1009
多項式の積, 1009
多項式の変数に値を代入, 1009
な
並びの照合, 1003
ふ
ファイルの close, 1021
ファイルの open, 1021
ファイルの読込, 1017
ファイルへの保存, 1018
不等号, 1004
へ
ベクトルの成分, 996
ほ
ポインタ, 1021
surf
C
curve, 957
D
draw_curve, 957
draw_surface, 957
E
epsilon, 953
H
height, 953
I
iterations, 953
R
root_finder, 953
rot_x, 954
rot_y, 954
rot_z, 954
S
scale_x, 954
scale_y, 954
scale_z, 954
surface, 957
W
width, 953

函数

T
to_lisp, 981
き
記号代数処理, 5
す
数式処理, 5

K
KVM, 1044
V
VirtualBox, 1044
VMware Player, 1051
X
Xen, 1044

か
仮想化
完全仮想化, 1043
準仮想化, 1043

ふ
浮動小数点数
仮数部, 328
下駄履き値, 329
指数部, 328
符号部, 328