簡易 Maxima マニュアル

東芝インフォメーションシステムズ株式会社
横田博史

平成 19 年 2 月 14 日 (水)

まえがき

数式処理システム Maxima は 1960 年代に開発された MACSYMA を Common Lisp 上に移植したソフトウエアです.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, 見掛け以上に強力なソフトウエアであり,GPL の下で配布される数式処理システムの中では非常に魅力的な汎用の数式処理ソフトウエアの一つです.

このマニュアルは「はじめての Maxima」からマニュアル部分を抜き出したものです. このマニュアル部分は時間的な問題と, 本の厚さから, グラフ表示の話や, 大部分のパッケージを省略した簡易的なマニュアルとなっています.

ここでのマニュアルは, 簡易マニュアルを核に拡充したものとなる予定でしたが, 個人的な都合で残念ながら拡充がまだ出来ていません. 何となく義理チョコめいたマニュアルですが, もう暫く, 辛抱をお願いします.

平成 19 年 2 月 14 日 (水) 宇佐子のバレンタイン義理チョコ　　横田博史

目 次

第1章　Maximaの処理原理について

この章で解説する事:

- Maxia の順序
- Maxima の文脈
- 宣言と属性
- 演算子
- 規則と式の並び
- 評価の方法
- 式の評価
- LISP に関連する函数

1.1　順序について

数学には順序という考えがあります. 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. 例えば, 集合を実数とすると, 大小関係 $\geq$ は順序関係の一例です. この大小関係を一般化したものです.

ここで, 集合 S の二項間の関係 $\geq_{order}$ が以下の性質を満す時に順序と呼び, 順序の入った集合 S の事を順序集合と呼びます.

順序の定義

反射律:　$x \geq_{order} x$

対称律:　$x \geq_{order} y$　かつ　$y \geq_{order} x$　$\Rightarrow$　$x = y$

推移律:　$x \geq_{order} y$　かつ　$y \geq_{order} z$　$\Rightarrow$　$x \geq_{order} z$

集合 S の順序 $\geq_{order}$ で, 集合の任意の元 x, y に対し, $x \geq_{order} y$, 或いは $y \geq_{order} x$ の何れかが成立する場合, 順序 $\geq_{order}$ の事を全順序, 集合 S の事を全順序集合と呼びます.

例えば, 集合を実数 $\mathbb{R}$ とすると, 二項関係 $\geq$ は全順序, 集合 $\mathbb{R}$ が全順序集合となります. 但し, 二項関係を包含関係 $\supseteq$ とした場合, 包含関係 $\supseteq$ は集合 $\mathbb{R}$ の順序になりますが, 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 B の間には包含関係がありませんね. 全順序となる為には, 包含関係が無ければなりません. 従って, 全順序にはなりません.

数式処理の場合, 集合は数, 変数や函数で構築された式の集合を考えます. 式は和や積の様に可換な演算で結合された数, 変数や函数の為, 数と変数には変数の方が大きいと順序付け, 変数よりも函数が大きいと順序付け, 変数同士にも順序を入れると, 後は項の順序が残ります.

1 変数の場合, 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, より一般の多変数の場合はどうすれば良いでしょうか？この場合, 変数の間に順序を入れ, その変数の順序で変数を入れ換えた項の次数リストで比較する方法があります.

例えば, 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, $y\,z$ の様に変数が抜けている場合, 抜けている変数の次数を 0 とすれば, 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. 従って, n 変数 $x_1, \cdots, x_n$ の単項式 A と B が与えられた時, n 個の変数 $x_1, \cdots, x_n$ 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. こうする事で, 単項式 A と B を表現する長さが n の次数リスト $(\alpha_1, \cdots, \alpha_n)$ と $(\beta_1, \cdots, \beta_n)$ 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事になります. 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, $x^{\alpha+\gamma} > x^{\beta+\gamma}$ を満すものの方が都合が良いですね. この項に対する性質 $x > y \Leftrightarrow x \cdot a > y \cdot a$ を満す順序の事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 $x\,y^2\,z^3 (= (1, 2, 3))$ が得られた時, 順序はどの様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ね. この様に多項式の場合, 項の順序には色々な考え方があります. 次の小節では代表的

な項の順序について説明しましょう.

1.1.1 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, これらの順序に, 変数の重みを加味したもの等があります. 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. 尚, 変数の並び順を $x_1, \cdots x_n$ とし, $x_1^{a_1}, \cdots x_n^{a_n}$ の次数リストを $a = (a_1, \cdots, a_n)$ $x_1^{b_1}, \cdots x_n^{b_n}$ の次数リストを $b = (b_1, \cdots, b_n)$ とします. 更に, $|a| = a_1 + \cdots + a_n$ で次数リスト a に含まれる次数の総和を表記します. 又, 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です.

辞書式順序 $>_{lex}$

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

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, ここでは $>_{lex}$ を使います.

辞書式順序では二つの項を比較した時に, 左側の変数の次数が大きなものが大きな項となります.

例えば, $x^2\,y^2\,z$ 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 $(2,2,1) >_{lex} (1,2,3)$, 即ち, $x^2\,y^2\,z >_{lex} x\,y^2\,z^3$ となります.

斉次辞書式順序 $>_{glex}$

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

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, ここでは $>_{glex}$ 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方式となっています.

例えば, $x^2\,y^2\,z$ 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, $|x^2y^2z| = 5$ と $|xy^2z^3| = 6$ となるので, $(1,2,3) >_{glex} (2,2,1)$, 即ち, $x\,y^2\,z^3 >_{glex} x^2\,y^2\,z$ 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 $>_{revlex}$

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

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, ここでは $>_{revlex}$ を使います. 辞書式との違いは, 先ず, 調べる方向が逆で, 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の 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, 後から先に移動

して, リストの先頭の 3 と 1 に到達すると, 定義から $(1,2,3) >_{revlex} (3,2,3)$, 即ち, $xy^2z^3 >_{revlex} x^3y^2z^3$ を得ます.

斉次逆辞書式順序 $>_{grevlex}$

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

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, ここでは $>_{grevlex}$ を使います.

この順序は項の総次数で最初に項を比較し, 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、総次数が各々,5 と 6 になるので, $xy^2z^3 >_{grevlex} x^2y^2z$ となります. 因に, 逆辞書式の場合は, 次数リストの右端の z の次数から見るので, $x^2y^2z >_{revlex} xy^2z^3$ となり, 斉次逆辞書式順序とは逆の結果となります.

これらの項順序は多項式の計算で非常に重要です. 特に Groebner 基底の計算では, 項順序を用いた計算を行います. この Groebner 基底は項順序に対しては一意に決まりますが, 順序を替えると, 普通は別の基底が得られます. この事もあって為, 順序は新しい数式処理システムでは目的に応じて様々な順序が扱える様になっています.

尚, 代数学の様々な定理は, 対象に順序を入れる事で証明出来ものが多くあります. その意味でも, 項順序は非常に重要です.

1.2　多項式の表現

ここでは多項式の表現について考えてみましょう.

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により, 式が一意に決定します. 但し, 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. その為, 多項式は予め展開しておき, 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, 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.

とは言え, 内挿表現は無駄が多い表現です. これをより簡単にしたものに, 正準表現というものがあります. この正準表現を解説する為に, 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.

先ず, 与式が何の変数の多項式であるかが重要ですね. その重要なものを内挿表現から前置表現に変換する時と同じ様に先頭に変数 x を置きましょう. 次に, この式は $3x^2+(-1)x^0$ と同値になります. するとこの式は 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 で,0 次の係数の項は -1 ですね. そこで $(x\ (2\ 3)\ (0\ -1))$ と記述すればどうでしょう. 前置表現であれば, $(+\ (*\ 3\ (\hat{}\ x\ 2))\ -1)$ となりますが, 随分とすっきりしましたね. 更に, 内部の括弧は実は冪と係数の対と決まっているので, $(x\ 2\ 3\ 0\ -1)$ で十分ですね. すると, この表現 $(x\ 2\ 3\ 0\ -1)$ と多項式 $3x^2+(-1)x^0$ は一対一で対応します. この様に, 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, 以下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$_2$⟩ $\cdots$)

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の場合はどうでしょうか？実際, 多変数多項式 $K[x_1,\cdots,x_n]$ に対しても, 同様の手法で正準表現が構成出来ます. 但し, 多変数多項式の場合で重要な事は, 項に順序を入れる事です. そこで, 辞書順序 $>_{lex}$ を多項式環 $K[x_1,\cdots,x_n]$ に入れてみましょう.

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

次に, 多項式 $\sum a_{(i_1,\cdots,i_n)}x_1^{i_1}\cdots x_n^{i_n}$ が与えられると, 最初に x_1 の多項式と看做して, 一変数の場合と同様の考え方で表現を構成します. 先ず, 式に含まれる変数が x_1 だけの場合は, 一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場合は, 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 $(\in K[x_1,\cdots,x_{n-1}])$ となります. 今度は各係数に対し, 同様の考え方で 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事で, 以下の様な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$_2$⟩ $\cdots$)

具体的に多項式環 $\mathbb{Z}[x,y]$ の元 $y\,x+2\,x\,y^3-3$ に対し, 順序を辞書式順序 $>_{lex}$ で考えましょう. 尚, 変数の並びは x,y とします. この場合, 最初に与式を x の多項式と看做し, 変数 x で式を纏めましょう. その結果 $(2\,y^3+y)\,x-3$ が得られます. そこで, 第一段目は $(x\ 1\ 2\,y^3+y\ 0\ -3)$ となります. 次に係数の処理に移ります. この場合, 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, 第二成分の $2\,y^3+y$ 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 $y\,x+2\,x\,y^3-3$ の正準表現として得られます.

この様に, 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と同じ多項式でも表現が一般的に異なります.

1.3 Maxima に導入された順序

1.3.1 Maxima の変数と順序

Maxima で用いられる変数や項の順序について簡単に解説します. 順序の一般的な話は前節の 1.1 節を参照して下さい.

先ず,Maxima で変数として扱える文字はアルファベットの小文字と大文字, それと記号_等の幾つかの記号, そして, alphabetic 属性が与えられた文字です. 又, 変数の先頭以外ならば 0 から 9 の数も利用可能です.

例えば, 記号@や~はそのままでは変数として使えませんが, alphabetic として declare 函数で宣言すれば使えます. 以下に,@と~を変数名に用いる例を示しておきます.

```
(%i1) @x12;
Incorrect syntax: x12 is not an infix operator
@x12;
   ^
(%i1) ~123x;
Incorrect syntax: 123 is not an infix operator
~123x
   ^
(%i1) declare(@,alphabetic,~,alphabetic);
(%o1)                                done
(%i2) @y+@z+@z*4-~12x;
(%o2)                          - ~12x + 5 @z + @y
```

但し,Maxima の演算子として用いられている記号の様に alphabetic として宣言しても使えないものもあります.

Maxima で使える変数の集合を A_m とします. すると, この集合 A_m には順序が入っています. この順序を以降 $>_m$ と記述します.

順序 $>_m$ により Maxima の変数集合 A_m に入る順序を以下に示します.

Maxima の変数順序

宣言された主変数	$>_m$	ordergreat の第一引数	$>_m$	$\cdots$	$>_m$
ordergreat の最後の引数	$>_m$	*alphabetic* 宣言した文字$_1$	$>_m$	$\cdots$	$>_m$
alphabetic 宣言した文字$_n$	$>_m$	頭文字が Z の変数	$>_m$	$\cdots$	$>_m$
頭文字が A の変数	$>_m$	頭文字が z の変数	$>_m$	$\cdots$	$>_m$
頭文字が a の変数	$>_m$	orderless の最後の引数	$>_m$	$\cdots$	$>_m$
orderless の第一引数	$>_m$	宣言されたスカラー	$>_m$	宣言された定数	$>_m$
Maxima の定数	$>_m$	9	$>_m$	$\cdots$	$>_m$
0					

先ず, アルファベットに関しては, 大文字が小文字よりも大きく,Z が A よりも大きくなっており, 通常のアルファベットと逆の順番になっています. 次に,alphabetic 属性を与えた文字は, アルファベットの大文字の Z よりも大きく,ordergreat 函数の最後の引数よりも小さな順番となります.

0 から 9 迄の数値に対しては $>_m$ は通常の大小関係 $>$ と同値です. 基本的に ordergreat 函数や orderless 函数で入れた順序に無関係な文字や記号同士では ASCII コードの大きい方が順位が高くなります. これは, 文字の大小関係で内部の LISP の great 函数を用いて判別する為です.

変数が一文字であれば $>_m$ による比較は分かり易いものです. しかし, 変数は通常, *abc* の様に複数の文字で構成されます. この複数の文字に対しても, 順序 $>_m$ で順序付ける事が可能です. この考

え方を簡単に説明しましょう. 二つの変数 $x_1x_2\cdots x_n$ と $y_1y_2\cdots y_m$ が与えられた時, $n=m$ でなければ, 短い変数の側に空白を入れて等しい長さにしたものを考えます. この時, $i=1,\cdots,k-1$ では, $x_i=y_i$, k 番目の文字 x_k と y_k で初めて異なるとします. すると, 文字 x_k と y_y の順序は $>_m$ の順序で決める事が出来ます. そこで, 全体もその順序に従うものとしましょう. すると, $x_k>_m y_k$ であれば, $x_1x_2\cdots x_n>_m y_1y_2\cdots y_m$ となります.

具体的には, 二つの変数 abc と aaz が与えられた場合, 頭は同じ文字 a となる為, 二番目以降の文字で大小関係を定めます. すると, 二番目の文字で $b>_m a$ となる為, $abc>_m aaz$ となります.

こうする事で一般の変数に対しても $>_m$ で順序を入れる事が出来ます.

Maxima の変数で最も順位が高いものが, 主変数 (mainvar) として宣言された変数です. CRE 表現への変換等の多くの函数は主変数を用いますが, 式の中で mainvar として宣言された変数が存在しない場合, 式中の変数で順序 $>_m$ による最高位の変数が主変数として用いられます.

これで変数の順序が与えられました. 次に,Maxima の変数の集合 A_m の元の積で構成される項の順序について説明しましょう.

先ず,Maxima の順序 $>_m$ は辞書式順序と呼ばれる順序です. この順序による項の比較の方法を説明しましょう. 先ず, 項を構成する変数を $>_m$ の順序に従って大きなものから並べます. 例えば, 項の変数が $x1$ から $x9$ の場合, $x9>_m x8>_m\cdots>_m x1$ となるので, 変数の並びは $x9,\cdots,x1$ となります.Maxima ではこの項の変数順序の入れ替えは自動的に実行します. 次に, 二つの項が与えられて順序 $>_m$ を用いて大きさを比較する場合, 変数の並びから各項の次数リストを構築します. この次数リストは $>_m$ によって二つの項を構成する変数を並び換え, 各変数に対応する次数を左から順に並べたリストです. ここで, 項にある変数が抜けている場合には, 次数 0 を入れます. 例えば, 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 $x9,\cdots,x1$ の順序に従って変数の並び換えを行います. その結果, $x1\,x2^2\,x8^3$ は,$x8^3\,x2^2\,x1$ に,$x1\,x2^2\,x3\,x9$ は $x9\,x3\,x2^2\,x1$ になります. それから, 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では 0 の冪で置換します. すると, $x8^3\,x2^2\,x1$ 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 $x8$ が無いので, $x9\,x8^0\,x3\,x2^2\,x1$ とします. この様に正規化してから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 $(1,0,1,2,1)$ が得られます. 次に, 項の大きさの比較では, 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, $(1,0,1,2,1)$ 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. 従って, $x1\,x2^2\,x3\,x9>_m x1\,x2^2\,x8^3$ となります.

この順序 $>_m$ 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.

Maxima では, この項順序 $>_m$ に従って入力された多項式の項の変数と項の順番の並び換えを自動的に行っています.

```
(%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. しかし,Maxima が返す式は順番が変更されて同じ式で返されています. 又,:lisp $expr1 と:lisp $expr2 で表示させた式 expr1 と expr2 の内部表現も同じものになっています. 尚,:lisp は後に続く文字列を LISP に直接渡して結果を Maxima に戻す函数です.

この内部表現で注目して頂きたいのは, 順序 $>_m$ で小さなものが, リストの左側に置かれ, 大きなもの程, 右側に置かれる事です. 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. 但し, 項の変数順序に関しては逆で, 左側に小さなものが表示されます. この辺は, 数式の通常の書き方に準拠したものと言えるでしょう.

1.3.2 変数順序の変更

Maxima では順序 $>_m$ を局所的に変更する事が可能です. 尚, 前述の様に, 新たに記号をアルファベットとして宣言する事で, 新たに順序に記号が加えられますが, これも既存の枠に新しい記号を嵌め込む以上のものではありません.

この変数順序を変更する函数には 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($\langle v_1 \rangle, \cdots, \langle v_n \rangle$)
orderless($\langle v_1 \rangle, \cdots, \langle v_n \rangle$)
unorder()

ordergreat 函数によって, 引数左端の変数 $\langle v_1 \rangle$ が最大で, 右側に行くに従って小さくなり, $\langle v_n \rangle$ が最小となる順序が新たに Maxima の順序 $>_m$ に組込まれます.orderless 函数の場合も同様に, 左端の $\langle v_1 \rangle$ が最小で, 右に行くに従って大きくなり, $\langle v_n \rangle$ が最大になる順序が Maxima の順序 $>_m$ に組込まれます.

この ordergreat 函数や orderless 函数で Maxima に入れた順序は unorder(); を入力する事で全て解除出来ます.

尚,ordergreat 函数や orderless 函数で入れた変数順序は unorder 函数で無効にしない限り, これらの函数を用いて新しい変数順序を入れる事は出来ません.

```
(%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]
```

この例で示す様に,c $>_m$ b という順序を入れましたが, その次に,b $>_m$ z という変数順序を入れようとするとエラーが出ています. 因に,ordergreat(b,z) の代わりに ordergreat(z,b) でもエラーが出ます. そこで, unorder() を入力する事で ordergreat(c,b) で入れた変数順序を解除しています.

ordergreat 函数や orderless 函数で入れた変数順序で,Maxima 全体の変数順序の変更を伴なう為,ordergreat 函数や orderless 函数で入れた変数順序を解除しない限り, 更新を認めない方法を採用しているお陰で, 色々と順序の追加を行なって互いに矛盾した順序が出来上る恐れがない分, 安全と言えます.

1.3.3 順序に関連する函数

Maxima の項順序は, 変数に辞書の逆順で順序が入っていますが, 項順序自体は辞書式順序と呼ばれる順序が入っています.

そこで今度は Maxima で変数順序や項順序がどの様に入っているか, 実際に調べてみましょう. 二つの与えられた式の順序を調べる函数として Maxima には ordergreatp 函数と orderlessp 函数の二つの述語函数があります.

順序を確認する述語函数

述語函数	true となる条件
ordergreatp(⟨ 式$_1$⟩ ,⟨ 式$_2$⟩)	⟨ 式$_1$⟩ が ⟨ 式$_2$⟩ よりも大となる場合
orderlessp(⟨ 式$_1$⟩ ,⟨ 式$_2$⟩)	⟨ 式$_1$⟩ が ⟨ 式$_2$⟩ よりも小となる場合

ordergreatp 函数と orderlessp 函数は, 与えられた二項に対し,Maxima の項順序 $>_m$ による判定結果を返す述語函数です. 尚,ordergreatp 函数と orderlessp 函数は LISP の great 函数を用いて変数の比較を行っています.

```
(%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, アルファベットに関しては逆アルファベット順で大きさが決ります. 変数の比較では, 左端の文字から両者を比較し, 最初に大きなものがあった方を大とします. この例では,abc と a では頭は同じですが,a には他の文字が無い為, abc $>_m$ a となります. 次の,abc と ax の場合, 頭の a は同じでも,b と x では x の方が大となる為,ax $>_m$ abc となります.

以降, 項に対する比較となります. 項順序に関しては,z の方が y よりも大きい為, 次数とは無関係に z の方が y^2 よりも大になります. 同じ変数の場合は次数の大きな方が大になり,z^2xy^2 と z^2xt^3 の場合は, 頭から比較して,y の方が t よりも大になる為, z^2xy^2 の方が z^2xt^3 よりも大になります. この事から,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事が理解出来るかと思います.

この様に 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

1.3.4　函数を含めた順序

ここまでは多項式の項順序について述べましたが,Maxima では通常の多項式に加えて exp や sin 等の Maxima 組込の初等函数にも順序が入ります.

この場合, 同じ引数の函数の比較では函数名で $>_m$ による順序が入ります. 次に, 函数と他の変数では, 函数に含まれる変数に主変数があれば, 函数が変数よりも上位になりますが, 函数に主変数が含まれずに比較する変数が主変数の場合, 変数の方が上位になります. その為, 函数の引数が比較する項を構成する変数よりも順位の低い変数であれば, 項の方が順位が高くなります.

ところで, 利用者定義の函数の場合, 一度 Maxima 側で値を解釈する為に ordergreatp 函数や orderlessp 函数の値はその状況に応じて変化します.

以下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)
```

この例で示す様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, 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. この様に初等関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.

1.4 文脈の概要

Maxima には文脈 (context) があります. この文脈とは要するに, 問題を考える上での様々な仮定や事実の積重ねです. 式の計算や簡易化といった式の処理は, この文脈を利用して行います.

例えば, 非常に簡単な例ですが,4 の平方根は 2 になりますが, x^2 はどうでしょうか？ x は正か負か, ひょっとすると複素数かもしれません. この様に $\sqrt{x^2}$ を考えるだけでも, x に関して様々な情報が必要になります. 先ず, 実数であれば $|x|$ となりますね. 更に,$x \geq 0$ であれば答は x になります.

この処理では, 以下の x に対する仮定を用いています.

- x は実数である.
- $x \geq 0$ である.

文脈とは, この様な問題を考える上で必要となる情報の蓄積の事です.

Maxima には複数の文脈を持たせる事が可能です. 更に, 文脈には階層構造もあります. 先ず, 文脈 global が最上位の文脈です.Maxima を立ち上げた時には既に文脈 global の子文脈 initial があります. デフォルトの文脈はこの initial です.

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デフォルトの文脈名.
global	Maxima の文脈全体の親文脈.

文脈には様々な事実や仮定を蓄積して行く事になりますが, その操作を行う函数を以下に纏めておきます.

文脈上の述語に関連する函数

assume(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, $\cdots$)
forget(⟨ 述語$_1$⟩, $\cdots$, ⟨ 述語$_n$⟩)
forget([⟨ 述語$_1$⟩, $\cdots$, ⟨ 述語$_n$⟩])
facts(⟨ 事項 ⟩)
facts(⟨ 文脈 ⟩)
facts()

Maxima に事実や仮定を教える函数は assume 函数です. 様々な事実や仮定は,Maxima で述語と呼ばれる Maxima の式で表現されます. 述語は評価を行う事で true か false が返却される論理式, 例えば, $x > 0$ や, $x < 1$ and $x > 0$ の様な二項関係に論理積 and, 論理和 or, 否定 not 等の論理演算子が作用した Maxima の式です.

この述語は assume 函数によって現行の文脈上に設定されます. これに対し, 教えた事実や仮定を忘れさせる, 即ち, 文脈に設定した述語を消去する場合,forget 函数を用います. そして, 或る事項に関して, どの様な述語が含まれているかを調べたければ,facts 函数 を用います.facts 函数は文脈を指定した場合, その文脈に含まれる述語を返します. 何も指定しない facts() を入力した場合, 現行の文脈が保持する述語を全て表示します.

Maxima では文脈を複数持たせる事が可能です. この事は, 文脈 A では $x > 0$ を仮定して述語を蓄積し, 文脈 B では $x < 0$ を仮定して述語を文脈 A とは独立して蓄積させる事が可能です.Maxima では, 文脈を新たに生成したり, 既存の文脈の間に親子関係を入れる等の操作が行えます.

以下に文脈操作函数の一覧を示します.

文脈操作函数

activate(⟨ 文脈$_1$⟩, $\cdots$)
deactivate(⟨ 文脈$_1$⟩, $\cdots$)
killcontext(⟨ 文脈$_1$⟩, $\cdots$)
newcontext(⟨ 文脈 ⟩)
supcontext(⟨ 親文脈 ⟩, ⟨ 子文脈 ⟩)

先ず, 既存の文脈を有効にする場合,activate 函数を用い, 逆に無効にする場合は decativate 函数を用います.

新しい文脈の生成は newcontext 函数を用い, 既存の文脈の親文脈を生成する場合は supcontext 函数を用います.

文脈の削除は killcontext 函数で行いますが, 利用中の文脈や文脈が有効 (activate) の場合は削除が出来ません.

デフォルトの文脈は initial ですが, 文脈を生成して行くと, 現在利用中の文脈が何で, どの様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 に現在, 利用中の文脈名が割当てられ, 大域変数 contexts に Maxima の中にある全ての文脈名のリストが割当てられています.

文脈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

先ず, どの様な文脈が Maxima に存在するかは contexts; と入力すれば文脈の一覧がリストの形式で表示されます. 例えば,Maxima を立ち上げた時点では, 文脈 initial と文脈 global の二種類の文脈がある事が判り, context; と入力すると, 文脈 initial が現行の文脈, 即ち,assume 函数等で設定した述語が文脈 initial に蓄積される事が判ります.

```
(%i1) contexts;
(%o1)                          [initial, global]
(%i2) context;
(%o2)                               initial
```

複数の文脈が存在する場合に, 文脈を切替えたければ, 大域変数 context に文脈名を直接指定します.

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 contexts 函数で表示しています. 次に context 函数を使って最初に用いている文脈が文脈 initial である事を示しています. それから,newcontext 函数を使って新しい文脈 mike を生成しています. それから文脈 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 函数には子文脈に既存の文脈を指定しなければなりません.

次に,assume 函数を用いて文脈上に述語を設定し, その述語を用いて式を簡易化する例と文脈の切り替えによる効果を見ましょう.

```
(%i6) assume(y>0);
(%o6)                               [y > 0]
(%i7) assume(x>0,z<0);
(%o7)                          [x > 0, z < 0]
(%i8) facts();
(%o8)                      [y > 0, x > 0, 0 > z]
(%i9) sqrt(x^2);
(%o9)                                 x
(%i10) context:initial;
(%o10)                              initial
(%i11) sqrt(x^2);
(%o11)                              abs(x)
(%i12) facts();
(%o12)                                []
(%i13) activate(neko);
(%o13)                               done
(%i14) context;
(%o14)                              initial
(%i15) sqrt(x^2);
```

```
(%o15)                                x
(%i16) facts();
(%o16)                               []
(%i17) deactivate(neko);
(%o17)                             done
(%i18) sqrt(x^2);
(%o18)                            abs(x)
(%i19) killcontext(neko);
(%o19)                             done
(%i20) contexts;
(%o20)                  [mike, initial, global]
```

先程の文脈の定義に続いて, 文脈 neko で assum 函数を使って変数 x,y,z に対して仮定, 即ち, 述語を設定します. 利用している文脈で設定した述語全体は facts(); で表示が出来ます. 文脈 neko で abs(x^2) を実行すると, 述語 $x > 0$ より x が返されます. 次に, 文脈の切替は変数 context に文脈名を割当てる事で行います. ここの例では, 文脈 neko から文脈 inital へと現行の文脈を context:initial; で変更しています. それから, sqrt(x^2) を実行すると abs(x) が返されます. これは何故でしょうか? 文脈には $x > 0$ という情報が無いからです. 実際, facts(); を実行すると, 文脈 initial には何も述語が無い事が分ります. ここで, 文脈 neko の述語を文脈 initial で利用したければ, actvate(neko); を実行します. その結果,sqrt(x^2) の結果が x になります. 但し, 文脈 initial の内容には変化がありません. 不要になった文脈は deactivate 函数で無効にしたり, killcontext 函数で削除する事が可能です.

次に, 大域変数 features には Maxima に組込まれた属性のリストが割当てられています. Maxima の式に features のリストに含まれる属性を持たせたければ, declare 函数を用います.

例えば birthday を整数として宣言する事で, 整数属性を持たせたければ, declare(birthday,integer); とします. 逆に式 p が属性 q を持つかどうかを調べる場合は featurep 函数を使って, featurep(p,q); で調べます.

尚,declare 函数による宣言は,assume 函数と違い大域的なものになります. その一方で,featurep 函数は declare 函数を用いた文脈上でしか使えません. 詳細は,1.5.1 節を参照して下さい.

```
(%i1) newcontext("mike");
(%o1)                               mike
(%i2) supcontext("neko","mike");
(%o2)                               neko
(%i3) context:mike;
(%o3)                               mike
(%i4) declare(bb,lassociative);
(%o4)                               done
(%i5) assume(x>0);
```

```
(%o5)                                [x > 0]
(%i6) facts();
(%o6)                    [kind(bb, lassociative), x > 0]
(%i7) bb(bb(a,b),bb(c,d));
(%o7)                        bb(bb(bb(a, b), c), d)
(%i8) sqrt(x^2);
(%o8)                                  x
(%i9) context:initial;
(%o9)                                initial
(%i10) bb(bb(a,b),bb(c,d));
(%o10)                       bb(bb(bb(a, b), c), d)
(%i11) aa(aa(a,b),aa(c,d));
(%o11)                       aa(aa(a, b), aa(c, d))
(%i12) facts();
(%o12)                    [kind(kron_delta, symmetric)]
(%i13) sqrt(x^2);
(%o13)                               abs(x)
```

この例では, 文脈 mike と neko を生成し, context:mike; で文脈をデフォルトの文脈 initial から文脈 mike に切替えています. そこで,bb に左結合律が成立する事を宣言し, $x > 0$ も assume 函数で文脈 mike に入れています.facts 函数で文脈に蓄積した述語を確認し,bb(bb(a,b),bb(c,d)) が左結合律によって,bb(bb(bb(a,b),c),d) に自動的に変形される事と,sqrt(x^2) が x に簡易化される事を確認しています. それから, context:initial; でデフォルトの文脈 initial に移動します.

ここで,declare 函数による宣言は assume 函数による述語とは違い, 大域的な影響を及ぼす為,bb に関しては左結合律が成立します. その一方で,$x > 0$ は文脈 mike 上の仮定の為, 文脈 init には影響を与えません. その為,sqrt(x^2) は abs(x) になります. 更に,bb に関しては facts() には表われません.

最後に, 文脈で多く設定されるものの一つが変数の正値性です. この正値性の設定では通常,assume 函数を用いて設定を行いますが, 大域変数を用いて assume 函数による設定を省く事も可能です.

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 で指定した述語函数が true となる対象に対し, 正値であると仮定.

先ず, 大域変数 assume_pos は, 大域変数 assume_pos_pred と対で用います. 大域変数 assume_pos は, 大域変数 assume_pos_pred で指定した述語函数が true となる Maxima の式に対して, その正値性を設定します. 但し, assume_pos が true で assume_pos_pred をデフォルトの false のままにしている場合,symbolp 函数が true となる Maxima の式に対し, その正値性を設定します. 尚,assume 函数による設定がある場合, その assume 函数の設定が大域変数 assume_pos よりも優先されます.

```
(%i13) declare(aa,even);
(%o13)                                done
(%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;
(%o16)                                true
(%i17) sqrt(aa^2);
(%o17)                                 aa
(%i18) sqrt(bb^2);
(%o18)                               abs(bb)
```

この例では変数 aa に偶数として属性を与えています. 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. その為,sqrt(aa^2) は aa になりますが, 偶数としての属性を持たない変数 bb は abs(bb) となります.

1.5　属性の宣言と属性値

Maxima ではアトムや式に属性や属性値を付加する事が出来ます. この処理を行う代表的な函数に declare 函数と put 函数があります.

declare 函数は Maxima に予め設定された属性をアトムや式が持つ事を宣言する事に用います. 尚,declare で宣言する事で式に与えられる属性は, 基本的に数値であれば, 偶奇性, 函数の場合は線形性といった数学上の性質が主です.

declare 函数は Maxima のアトムや式に属性を与える事で式の変形を促進する作用もあります. 例えば, 函数 $f(x)$ が linear, 即ち, 線形写像であると宣言していれば, Maxima は $f(x+y)$ が与えられると $f(x)+f(y)$ に自動的に簡易化します.

```
(%i3) declare(f1,linear);
(%o3)                                done
(%i4) f1(x+y);
(%o4)                           f1(y) + f1(x)
```

又, 利用者が必要な属性を新たに Maxima に付加し, その属性を持つアトムや式に対して様々な函数を作成する事も可能になります.

これに対し,put 函数はアトムに対して用いるもので, 利用者の指定する属性に対応する値をアトムに設定する事が出来ます.

尚,put 函数で与えた属性値の取出しは,get 函数で属性値を設定したアトムと値を取出したい属性を指定する事で行います. 更に, アトムに指定した属性とその値の削除は rem 函数や remove 函数で行います.

1.5.1　declare 函数

declare 函数はアトムや式に対し,Maxima 上で定義された属性を付加する函数です.

declare 函数の構文

declare($\langle a_1\rangle, \langle$ 属性$_1\rangle, \langle a_2\rangle, \langle$ 属性$_2\rangle, \cdots$)
declare($\langle a\rangle, [\langle$ 属性$_1\rangle, \cdots, \langle$ 属性$_n\rangle$)
declare($[\langle a_1\rangle, \cdots, \langle a_m\rangle], [\langle$ 属性$_1\rangle, \cdots, , \langle$ 属性$_n\rangle$)

declare 函数による宣言で $\langle a_i\rangle$ に $\langle$ 属性$_i\rangle$ が付加されます. 基本的に,declare 函数の引数は属性を付加されるものと属性の対になります. 更に, 属性がリストで与えられる場合, $\langle a_i\rangle$ は属性リストに含まれる全ての属性を持つ事になります.

この事を以下の例で説明しましょう.

```
(%i10) declare(a1,[integer,odd]);
(%o10)                               done
(%i11) declare([b1,c1,d1],[integer,odd]);
```

```
(%o11)                                  done
(%i12) featurep(d1,odd);
(%o12)                                  true
(%i13) featurep(c1,integer);
(%o13)                                  true
```

最初の例では変数 a1 が整数で, 奇数となる事を宣言しています. 次の例では変数 b1,c1 と d1 が整数で奇数となる事を宣言しています. この様に, 複数の属性を設定する場合は属性をリストで与え, 複数の式にある属性を与える場合には, 式をリストで指定します.

但し, 複数の属性を一つの式に設定する場合, 設定する属性は互いに無矛盾なものでなければなりません. 例えば, 偶数, 且つ奇数であると宣言する事は出来ません.

```
(%i11) declare(n1,odd);
(%o11)                                  done
(%i12) declare(n1,even);
Inconsistent Declaration: declare(n1,even)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i13) declare(n2,integer);
(%o13)                                  done
(%i14) declare(n2,even);
(%o14)                                  done
```

この例では, 最初に奇数として宣言してから偶数と宣言すると, 偶数且つ奇数となる数が存在しない為に declare 函数はエラーを返します. 但し, 整数である事と偶数である事は矛盾しない為,declare 函数で別個に宣言しても問題はありません.

declare 函数による宣言の影響は, 文脈で利用される assume 函数による設定と異なり, 特定の文脈上に留まらずに大域的に影響を及ぼします. その一方で, `facts();` で表示される内容は declare 函数を用いた文脈上でのみ設定されています. その為, 文脈 A で宣言した内容を文脈 B 上で `facts();` と入力して確認する事は出来ませんが, 宣言した事実を利用する事は可能です.

declare 函数は Maxima が持っている属性を与えるだけの函数ではありません. 利用者が Maxima に新たな属性を与える為に利用する事も可能です.

属性を追加する場合

declare(⟨ 新属性 ⟩,feature)

Maxima に属性を新規に追加する場合, 第一引数に属性名を設定し, 第二引数を feature にします. この宣言を行うと, 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. 尚, 式がある属性を持つかどうかは述語函数の featurep 函数を使って調べられます.

述語函数 featurep

featurep(⟨ アトム ⟩,⟨ 属性 ⟩)

featurep 函数は,⟨ アトム ⟩ が ⟨ 属性 ⟩ を持つかどうかを調べる函数です. ここで, ⟨ 属性 ⟩ は大域変数 features に割当てられたリストの成分でなければなりません.

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.

```
(%i8) declare(quaternion,feature);
(%o8)                                done
(%i9) features;
(%o9) [integer, noninteger, even, odd, rational, irrational,
 real, imaginary, complex, analytic, increasing, decreasing,
 oddfun, evenfun, posfun, commutative, lassociative,
 rassociative,  symmetric, antisymmetric, quaternion]
(%i10) declare(q1,quaternion);
(%o10)                               done
(%i11) featurep(q1,quaternion);
(%o11)                               true
```

この例では, declare(quaternion,feature); で quaternion を features に追加しています. それから, declare(q1,quaternion); で q1 が quaternion であると宣言しています. 最後に,featurep 函数で q1 が quaternion である事を確認しています.

これで, 四元数 quaternion という属性を入れる事が出来ました. 後は四元数を扱える Maxima の函数を定義すれば良いのです.

では, 以降の小節では,Maxima に予め用意された属性について順番に解説しましょう.

1.5.2 Maxima に用意された属性

最初に物事の基本となるアトムに対して与えられる属性について説明します. declare 函数を使う事で, スカラー, 定数, 主変数といった属性, 更には, 変数として利用可能な文字を宣言する事が出来ます.

アトムの属性

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⟩ ,nonarray)	⟨a⟩ を配列ではないと宣言
declare(⟨a⟩ ,constant)	アトム ⟨a⟩ を定数として宣言
declare(⟨a⟩ ,mainvar)	変数 ⟨a⟩ を主変数として宣言
declare(⟨a⟩ ,alphabetic)	⟨a⟩ をアルファベットとして宣言
declare(⟨a⟩ ,special)	⟨a⟩ を special 変数として宣言

最初のスカラーの宣言は行列やベクトルの演算で特に影響します.

主変数の宣言は Maxima の多項式の変換等で大きな影響が出ます. 通常,Maxima の変数順序 $>_m$ より式に含まれる変数の中で最大の変数を主変数とします. そして, 自動的に主変数を筆頭に順序 $>_m$ に沿って項と変数を並び換えます. 但し, この主変数は局所的なものです. これに対し, 変数に対して mainvar 宣言を行うと, 宣言された変数は順序 $>_m$ で最高位の変数になります. その為, mainvar 属性を持った変数は常にその変数が含まれる式の中では主変数として処理されます.

その為,mainvar 属性を持った変数が存在すると, 式の内部表現にも影響が及ぶ為に, 式の表示に限らず, 様々な処理も異なる事があります.

```
(%i1) f1:(x+y)^4;
                                      4
(%o1)                          (y + x)
(%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$
(%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
```

この例では, 最初に多項式 $(x+y)^4$ を展開しています. ここでは, デフォルトの順序 $>_m$ で展開される為, y の多項式として展開されます. それから, x を主変数として宣言して展開した為, 多項式 $f1$ は x の多項式として展開されています. ここで, f1,declare(x,mainvar),expand は ev 函数による評価の方法で,$f1$ を展開する際に x を主変数として評価しています. この ev 函数内部での宣言は ev 函数による式の評価以外には影響を与えません. 詳細は 1.8.1 節を参照して下さい.

ここで主変数を宣言しなかった場合とした場合では, 同じ式の表示の順序が異なっています. これは内部表現自体が異っている事を意味します. 実際,mainvar 宣言をしなかった場合 (ans1) の内部表現と mainvar 宣言を行った場合 (ans2) の内部表現は ans1 が変数 Y で式を纏めているのに対して ans2 が変数 X で纏めている点で異っています. これだけであれば, 単純に式の表現の違いだけで済みますが,ans1+ans2 の和を計算させても, 単純に ans1 と an2 を繋げて ans1 の末端と ans2 の先頭の x^4 を足し合せただけの結果を返します. この様な場合,ev 函数を用いて式を再評価する必要があります. 実際,ev 函数を用いて簡易化させる事で本来の結果が得られています.

alphabetic で引数の文字を Maxima のアルファベットとして利用可能にします. 尚,Maxima で変数等で利用可能な文字は, デフォルトで a から z 迄の小文字と大文字, それに%と. を加えたものです. それ以外の文字を変数で利用したければ, この alphabetic で宣言を行う必要があります.

最後に,special で変数を special 変数として宣言します. この宣言はあまり表に出る事はありません. この変数は函数の最適化等でも用いられますが, 変数に属性を指定する事で, 変数に割当てられる値に制限を加える事も可能になります. この方法は,define_variable 函数 で解説します.

次に,declare 函数で宣言可能な数値属性を以下に示します.

数値属性

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

数値属性には整数, 非整数, 偶数や奇数, 有理数や無理数, 実数, 純虚数に複素数があります. これらの宣言を行う事で Maxima の函数で動作が異なるものも多くあります. 又,-1 の整数による冪乗や三角函数では引数に含まれている整数が偶数であるか, 或いは奇数であるかによって式自体が自動的に簡易化される事もあります.

次に, 函数に対する属性について説明しましょう. 函数に関する属性としては函数と積や和といった Maxima の演算子や引数の並び替えといった作用に関連した属性, 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けられます.

函数属性 (作用に関して)

declare(⟨f⟩ ,additive)	函数 ⟨f⟩ の加法性
declare(⟨f⟩ ,multiplicative)	函数 ⟨f⟩ の乗法性
declare(⟨f⟩ ,outative)	函数 ⟨f⟩ と積の可換性
declare(⟨f⟩ ,linear)	函数 ⟨f⟩ の線形性
declare(⟨f⟩ ,commutative)	函数 ⟨f⟩ の対称性
declare(⟨f⟩ ,symmetric)	函数 ⟨f⟩ の対称性
declare(⟨f⟩ ,lassociative)	函数 ⟨f⟩ の左分配律
declare(⟨f⟩ ,rassociative)	函数 ⟨f⟩ の右分配律

函数 f に対する additive 属性の宣言は, 函数 f の第一変数, 即ち, 左端の変数に対する加法性を宣言します. 即ち, $f(x_1+y_1,\cdots)$ は $f(x_1,\cdots)+f(y_1,\cdots)$ と同値になります. 尚, この宣言は式の内部表現に依存する為,sum 函数に対しては効果がありません.

函数 f に対する multiplicative 属性の宣言は, 函数 f の第一変数に対する乗法性を与えます. 即ち, $f(x_1*y_1,\cdots)$ は $f(x_1,\cdots)*f(y_1,\cdots)$ と同値になります. 尚,multiplicative 属性は,additive 属性と同様, 式の内部表現に依存する為,product 函数に対して無効になります.

outative 属性の宣言では, 函数 f に可換積*との交換性を付与します. 即ち, $f(a*x_1,\cdots)$ は $a*f(x_1,\cdots)$ と同値になります. 但し, 外に出される a はアトムに限定されます.

デフォルトで sum 函数,integrate 函数と limit 函数 の名詞型が outative 属性を持つものとして宣言されています.

linear 属性は二つの属性 additive と outative の両方を持たせたものに相当し, 更に, 函数 f の第一変数に対して線形性を宣言します. 即ち, $f(x_1+y_1,\cdots)$ は $f(x_1,\cdots)+f(y_1,\cdots)$ に同値になり, 定数 a に対し, $f(a*x_1,\cdots)$ は $a*f(x_1,\cdots)$ に同値になります.

函数 f に対する commutative 属性の宣言は, 函数 f の引数の順番を入換えても結果が変化しない事, 即ち, 変数順序の可換性を意味します. 又,commutative 属性は symmetric 属性, 即ち, 対称性と同値です. ここで commutative 属性は演算の可換性に関連し,symmetric 属性は函数の引数の対称性に対処します. 但し, これらの基本的な動作は同じです.

ここで, 一度引数の置換を行うと符号が反転する函数, 即ち, $f(x,y)=-f(y,x)$ を満す場合を f は歪対称性があると呼びます.Maxima では属性 antisymmetric を持つ事を意味します.

属性 antisymmetric で函数 f は丕対称として宣言されます. これは一度函数 f の引数の順番を入れ換えたものが元の値を-1 倍したものになる性質です. 例えば, $f(x,y,\cdots)=-f(y,x,\cdots)$ となる性質です.

lassociative 属性で函数 f の左結合律性を宣言します. $f(f(a,b),f(c,d))$ が $f(f(f(a,b),c),d)$ と同値である事を意味しますが, ここで f を演算子 $*$ で置換えると, $(a*b)*(c*d)=((a*b)*c)*d$ を意味し, これは左側からの結合律を満すものになっています.

rassociative 属性の宣言により函数 f は右結合律律を満す函数となります. 即ち, $f(f(a,b),f(c,d))$ は $f(a,f(b,f(c,d)))$ に簡易化されます. これも, 函数 f を演算子 $*$ で置換えると, $((a*b)*(c*d))=a*(b*(c*d))$ に簡易化される事を意味し, これは右側からの結合律を満すものになっています.

declare 函数を用いた函数の属性としては, 増加, 減少, 奇函数, 偶関数, 正値函数, 解析的函数等の性質を指定する事も可能です.

函数の属性

declare(⟨f⟩ ,analytic)	函数 ⟨f⟩ を解析的函数として宣言
declare(⟨f⟩ ,increasing)	函数 ⟨f⟩ を増加函数として宣言
declare(⟨f⟩ ,decreasing)	函数 ⟨f⟩ を減少函数として宣言
declare(⟨f⟩ ,oddfun)	函数 ⟨f⟩ を奇函数として宣言
declare(⟨f⟩ ,evenfun)	函数 ⟨f⟩ を偶函数として宣言
declare(⟨f⟩ ,posfun)	函数 ⟨f⟩ を正値函数として宣言
declare(⟨f⟩ ,noun)	函数 ⟨f⟩ を名詞型函数として宣言

最後の重要な属性に evfun 属性と evflag 属性があります. これらの属性を持つ函数や大域変数は ev 函数を用いた評価で利用されます. これらの属性は全て declare 函数を用いて設定する事が可能です.

ev 函数で用いる evfun,evflag 属性

declare(⟨f⟩ ,evfun)	函数 ⟨f⟩ に evfun 属性を付加
declare(⟨a⟩ ,evflag)	大域変数 ⟨a⟩ に evflag 属性を付加

詳しくは,1.8.1 節で述べますが,ev 函数では引数として与えられた evflag 属性を持つ大域変数は,ev 函数内部でのみ true が設定され, 又,evfun 属性を持つ函数を ev 函数の引数として与えると,evflag 属性を持った引数と, その函数を与式に適用して処理を行います. 利用者が定義した大域変数や函数に対して declare 函数を用いて宣言する事で,ev 函数の引数として与えられる様になります.

1.5.3　put 函数による属性値の指定

Maxima ではアトムに対して様々な属性値の指定が行えます.declare 函数による宣言の他に,put 函数でアトムと属性に対応する属性値を指定し,get 函数でアトムと属性を指定して対応する属性値を取出す事が出来ます.

属性に設定に関連する函数

put(⟨ アトム ⟩ ,⟨ 属性値 ⟩, ⟨ 属性 ⟩)
qput(⟨ アトム ⟩ ,⟨ 属性値 ⟩, ⟨ 属性 ⟩)
get(⟨ アトム ⟩ ,⟨ 属性 ⟩)

put 函数は ⟨ アトム ⟩ に ⟨ 属性 ⟩ で指定した ⟨ 属性値 ⟩ を加えます. これは利用者がアトムに任意の属性を与える事が可能になります. この設定した属性値は get 函数で取出せます.

put 函数による属性の設定では属性を幾つも指定可能で, 利用者独自の指定も出来ます. 但し, アトムと属性に対応する属性値は一つです. アトムに設定された属性は properties 函数で確認出来ます. 属性値の取出しは get 函数を用いて, アトムと属性を指定すると得られます.

qput 函数は put 函数に似ていますが, 引数の評価を行わない点で put 函数と異なります.

get 函数は, ⟨ アトム ⟩ に設定された属性で, ⟨ 属性 ⟩ に対応する属性値を取出す函数です. 属性値自体は put 函数で与えます.

1.5.4 変数宣言

Maxima で値を割当てた変数は基本的に大域変数になります. この変数は通常は型も属性も何もありません. この大域変数に特定の型を設定したり, 初期値を割当てる事も出来ます.

先ず, 変数に予め型を指定する函数として mode_declare 函数と modedeclare 函数があります. 但し,mode_declare 函数と modedeclare 函数は Maxima 内部では同じ函数です. この変数の宣言に加えて, 初期値を設定する函数に define_variable 函数があります.

変数宣言を行う函数

mode_declare($\langle$ 変数$_1\rangle, \langle$ 型$_1\rangle, \cdots, \langle$ 変数$_n\rangle, \langle$ 型$_n\rangle$,)
modedeclare($\langle$ 函数$_1\rangle, \langle$ 型$_1\rangle, \cdots, \langle$ 函数$_n\rangle, \langle$ 型$_n\rangle$,)
mode_identity($\langle$ 引数$_1\rangle, \langle$ 引数$_2\rangle$)
define_variable ($\langle$ 変数名 $\rangle, \langle$ 初期値 $\rangle, \langle$ 型 $\rangle$)

mode_declare 函数は $\langle$ 変数$_i\rangle$ に $\langle$ 函数$_i\rangle$ や $\langle$ 型$_i\rangle$ を設定します. 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宣言で利用されます. ここで mode_declare 函数で指定した型は変数の mode 属性に割当てられます.

大域変数の mode 属性

型	mode_declare 函数で利用可能な表記	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	任意精度の数
complex	complex	複素数
boolean	boolean,bool	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) mode_declare(x1,integer);
(%o28)                                [x1]
(%i29) :lisp (get '$x1 'mode)
$FIXNUM
(%i29) mode_declare(x2,rat);
(%o29)                                [x2]
(%i30) :lisp (get '$x2 'mode)
$RATIONAL
(%i30) mode_declare(x2,rational);
(%o30)                                [x2]
(%i31) :lisp (get '$x2 'mode)
```

```
$RATIONAL
```

この例では変数 x1 を整数型,x2 と x3 を有理数型として宣言しています. これらの値は各変数の mode 属性に蓄えられています. この属性は LISP の get 函数を用いて取出せます. この様に,mode_declare 函数は基本的に型の指定を行う函数です. mode_declare 函数で指定した変数と型は mode_identity 函数を用いて検証が出来ます.

```
(%i9) mode_identity(integer,x1);
(%o9)                                  128
(%i10) x1:256.988;
(%o10)                               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
```

この様に,mode_declare で宣言した型以外の値を与える事は可能で, mode_identity も変数に割当てられた型と異なると文句を言うだけです. mode_identity は変数の mode 属性と指定された属性が同じものかどうかを検証するのではなく, 単純に変数に割当てられた型が mode_identity の引数で指定された型と同じものかどうかを検証する函数で, 正しい場合には変数の値を返し, 異なる場合には警告するだけです.

型の検証に関連する大域変数

大域変数	初期値	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 が 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 が true であれば,mode_declare 函数で値が割当てられた変数に対し, 異なる型で変数の宣言を行う場合にエラーを出力します.

大域変数 mode_check_warnp が 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.

define_variable 函数は変数の宣言に加え, 初期値と型の設定が行える函数です. 但し, この函数は属性を上手く使う事で, 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事も行える様に出来ます.

この define_variable 函数の処理手順を以下に示しておきます.

- mode_declare 函数を用いて変数型を宣言します.
- declare 函数を用いて変数が special 属性を持つと宣言します.
- 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. 値の検証を行う為には大域変数の value_check 属性に述語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に, 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 函数を用いて宣言した大域変数の value_check 属性に変数値を検証する述語函数名を割当ておきます. 以下に, 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述語函数 ptest を定義し, それから, define_variable 函数で大域変数 tama に初期値 5 を与えて整数変数として宣言します. それから,qput 函数を用いて check_value 属性に ptest を与えます. この時, 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っています.

ここまでの動作をもう少し詳しく解説しましょう. 先ず,define_variable 函数による宣言で, 変数の型を integer と宣言しています.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 に値を割当てる時に,assign-mode-check 函数が, 大域変数 tama の check_value 属性に割当てられた述語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. この例では 15 が素数でない為に 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, 先程の様に 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為です. 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) として見ましょう. これによって, 属性 assign の値として assign-mode-check を設定されます. すると, 割当の際に検証が実行される様になります.

1.5.5　atvalue 函数による函数値の設定

Maxima では属性は色々な所で用いられています.atvalue 函数等を用いる事で, 函数の値の設定も属性で与える事が出来ます.

atvalue 函数

atvalue(⟨ 式 ⟩ ,⟨ 点リスト ⟩ , ⟨ 境界値 ⟩)
at(⟨ 式 ⟩ ,⟨ 点リスト ⟩)

atvalue 函数による境界値の設定では, 函数や式に対して点をリストで指定し, そのリストに対応する境界値を設定します.

この atvalue 函数で値を指定すると,atvalue 属性に座標と函数値が対応付けられ, ⟨ 式 ⟩ の properties リストに atvalue 属性が追加されます.

尚, 引数の ⟨ 式 ⟩ には函数 f($\langle v_1 \rangle$,$\cdots$, $\langle v_n \rangle$) やその微分が指定可能ですが, その場合には変数の従属性が明確になる様に指定しなければなりません.

⟨ リスト ⟩ は ⟨ 変数$_1$⟩ =⟨ 式 ⟩ の様に変数と値を演算子=で繋ぎ合せた式のリストになります.

atvalue(f(x),x=x^2+x+1,0) とした場合,x=x^2+x+1 の左辺は函数 $f(x)$ で用いた疑似変数であり,=も方程式で用いる演算子=とは意味が異なっています. その為,x^2+x+1 で用いている変数 x とは関係ありません. この事から, $x^2+1=0$ の解 $x=\pm i$ を函数 f に代入しても函数 f は零にならない事に注意して下さい. 函数 f が零になるのは $f(x^2+x+1)$ の時です.

```
(%i1) atvalue(f(x),x=x^2+x+1,0);
(%o1)                                 0
(%i2) f(x^2+1);
                                     2
(%o2)                             f(x  + 1)
(%i3) f(x^2+x+1);
(%o3)                                 0
```

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,$\cdots$ が用いられます.

```
(%i35) atvalue(h(x,y,z),[x=1,y=0,z=0],10);
(%o35)                                10
(%i36) atvalue(diff(h(x,y,w),w),[x=1,y=0,w=0],0);
(%o36)                                0
(%i37) printprops(h,atvalue);
                                 !
                d                !
               --- (h(@1, @2, @3))!                      = 0
               d@3               !
                                 !@1 = 1, @2 = 0, @3 = 0

                             h(1, 0, 0) = 10
```

但し,atvalue 函数で設定した函数の値は get 函数で取出せず, 削除も rem 函数では出来ません.

```
(%i1) put(f,C-inf,type);
(%o1)                                C - inf
(%i2) atvalue(f(x),x=0,0);
(%o2)                                  0
(%i3) properties(f);
(%o3)                 [atvalue, [user properties, type]]
(%i4) get(f,type);
(%o4)                                C - inf
(%i5) rem(f,atvalue);
(%o5)                                 false
(%i6) remove(f,atvalue);
(%o6)                                 done
(%i7) properties(f);
(%o7)                       [[user properties, type]]
```

しかし,atvalue で設定した属性値は他のものと同様に printprops 函数で函数名と属性を指定する事で表示されます.

```
(%i19) atvalue(f(x),x=0,0);
(%o19)                                 0
(%i20) atvalue(g(x),x=0,1);
(%o20)                                 1
(%i21) atvalue(g(x),x=1,2);
(%o21)                                 2
(%i22) printprops(all,atvalue);
                                  f(0) = 0

                                  g(0) = 1

                                  g(1) = 2

(%o22)                                 done
```

次の at 函数は atvalue 函数と対で用います. 〈式〉を atvalue 函数に対して与えられるものと同じ方程式のリスト, 方程式中に含まれる変数に対し, atvalue 函数で指定した値を入れて評価します.

部分式が atvalue 函数で指定された値を持たない為に, 評価出来ない場合, その部分式に作用する at 函数の名詞型が返されます. 又, その部分式は二次元的書式で表示されます.

勾配と従属性の設定

gradef(⟨ 函数名 ⟩ (⟨ 変数$_1$⟩ ,···, ⟨ 変数$_m$⟩),⟨ 式$_1$⟩ ,···, ⟨ 式$_n$⟩)
gradef(⟨ アトム ⟩ ,⟨ 変数 ⟩ , ⟨ 式 ⟩)
depends(⟨ 函数 ⟩ ,⟨ 変数$_1$⟩ ,···, ⟨ 函数$_n$⟩ ,⟨ 変数$_n$⟩)

gradef 函数は ⟨ 函数 ⟩ の n 個の引数に対する微分を $\frac{d\langle f\rangle}{dx_i} = \langle$ 式$_i\rangle$ で定めます. 尚,gradef 函数で勾配を定義すると, 大域変数 gradefs に函数名が蓄えられます. 更に, 指定した函数には gradef 属性が付与されます.

引数が変数の総数 m · n 個の勾配 n よりも少ない場合, 最初の ⟨ 函数 ⟩ の i 番目の引数が参照されます.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 番目の変数を指定する為に用います.

最初の引数を除く全ての gradef 函数の引数は ⟨ 式 ⟩ が定義された函数ならばその函数が呼出されて結果が用いられます. 勾配は函数が第一階微分を除いて正確に判らない場合で, より高階の微分を得たい時に必要となります.

gradef(⟨ アトム ⟩ ,⟨ 変数 ⟩ ,⟨ 式 ⟩) で変数 ⟨ 変数 ⟩ による ⟨ 函数 ⟩ の微分が ⟨ 式 ⟩ となる事を宣言します. この場合, 大域変数 gradefs には登録されず, 属性も atomgrad 属性になります.

この時, depends(⟨f⟩ ,⟨x⟩) も実行されるので, depends 函数によって属性 dependency が付加され, 同時に大域変数 dependencies にも ⟨ 函数 ⟩ が追加されます.

gradef 函数は Maxima の既に定義された函数の微分を再定義する事にも使えますが, 添字された函数に対して gradef 函数は使えません.

depends 函数で, 函数の変数に対する従属を宣言します. 例えば, depends(f,x) で函数 f が変数 x に従属する性質を付加します. 尚,depends 函数による従属性の宣言にて, 函数自体を予め定義する必要はありません.

```
(%i41) depends(neko,[tama,mike]);
(%o41)                          [neko(tama, mike)]
(%i42) diff(neko,tama);
                                      dneko
(%o42)                                -----
                                      dtama
(%i43) diff(diff(neko,tama),tama);
                                       2
                                      d neko
(%o43)                                ------
                                           2
                                      dtama
(%i44) depends([rat1,rat2],[cheese,milk]);
(%o44)              [rat1(cheese, milk), rat2(cheese, milk)]
(%i45) depends([rat1,rat2],[cheese,milk],neko,[tama,mike]);
(%o45)    [rat1(cheese, milk), rat2(cheese, milk), neko(tama, mike)]
```

勿論,depends 函数を実行していなければ,diff 函数を実行した時点で 0 になります. depends 函数で neko が変数 tama と mike に従属する函数と宣言した為に, 微分を行っても零になりません. 最初の例では函数 neko が 1 成分しかない為に, リストの大括弧を外しています.

函数の変数に対する従属性は大域変数 dependencies に登録された函数の情報から調べられます.

gradef と depends に関連する大域変数

変数名	初期値	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

gradef 函数で勾配を定義すると, 大域変数 gradefs に函数名が蓄えられます.

大域変数 dependencies には,depends や gradef で指定された函数名が蓄えられます.

これらの変数は共にデフォルト値が空のリスト [] です.

```
(%i4) gradef(f(x,y),y,x);
(%o4)                                f(x, y)
(%i5) gradefs;
(%o5)                               [f(x, y)]
(%i6) diff(f(x,y),x);
(%o6)                                   y
(%i7) diff(f(x,y),y);
(%o7)                                   x
(%i8) dependencies;
(%o8)                                  []
(%i9) depends(g,x,y,z);
(%o9)                           [g(x), y(z)]
(%i10) dependencies;
(%o10)                          [g(x), y(z)]
(%i11) gradefs;
(%o11)                            [f(x, y)]
(%i12) gradef(h,x,x^2);
(%o12)                                  h
(%i13) dependencies;
(%o13)                     [g(x), y(z), h(x)]
```

この例で示す様に,gradef や depends 函数を用いる事で,gradefs や dependencies に函数名が蓄積されます. 尚,gradef を用いて dependencies に函数を追加する為には, gradef(⟨ アトム ⟩ ,⟨ 変数 ⟩ ,⟨ 式 ⟩) の形式に限定されます.

1.5.6 属性と属性値の削除

属性値の削除は rem 函数や remove 函数で行えます. この場合はアトムと属性を指定すると, アトムに設定した属性と属性値が一緒に削除されます.

属性や属性値を削除する函数

rem(⟨ アトム ⟩ ,⟨ 属性 ⟩)
remove(⟨ アトム$_1$⟩ ,⟨ 属性$_1$⟩ ,···, ⟨ アトム$_n$⟩ ,⟨ 属性$_n$⟩)
remove([⟨ アトム$_1$⟩ ,···,⟨ アトム$_m$⟩], [⟨ 属性$_1$⟩ ,···,⟨ 属性$_n$⟩])
remove (”⟨ アトム ⟩ ”,operator)
remove(⟨ アトム ⟩ ,transfun)
remove (all,⟨ 属性 ⟩)

rem 函数は ⟨ アトム ⟩ から ⟨ 属性 ⟩ で指定した属性とその属性値の両方を削除します.

これに対し,remove 函数は変数や関数に関連した属性の一部や全てを削除します. この属性はシステム側が定義したものでも利用者が与えたものや function, mode_declare でも構いません.

remove(⟨ アトム$_1$⟩ ,⟨ 属性$_1$⟩,···, ⟨ アトム$_n$⟩ ,⟨ 属性$_n$⟩) で, ⟨ 属性$_i$⟩ を ⟨ アトム$_i$⟩ から削除します.

ここで指定するアトムと属性は各々が対応するリストでも構いません.

属性に operator や op を指定した場合,declare 等で宣言した prefix(前置式), infix(内挿式),nary(内挿式),postfix(後置式),matchfix や nofix(無演算子) といった演算子の属性が削除されます. 尚, 演算子には必ず二重引用符で括る必要がある事に注意して下さい.

属性が transfun の場合,translate 函数で変換された LISP 函数が削除されます. この後は, Maxima 版の函数が用いられます.

引数にアトムではなく all を指定した場合, 指定された属性を持つアトム全てから, その属性が削除されます.

尚,remove 函数は与えられた属性が存在しない時でもエラーを返しません. 返却値は常に done です.

1.5.7 属性の表示

属性の表示に関連する函数としては,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(⟨ アトム ⟩)
propvars(⟨ 属性 ⟩)
props

properties 函数は ⟨ アトム ⟩ に関連する全ての属性を記載したリストを生成します.

以下の例では put 函数でアトムに属性を指定し,properties 函数でアトムに指定した属性に何があるかを確認し, 指定した属性に対応する属性値を取出しています.

```
(%i37) put(Mike,"2004/07/4",birthday);
(%o37)                              2005/07/4
(%i38) put(Mike,"10[Kg]",Weight);
(%o38)                               10[Kg]
(%i39) put(Mike,"White-Black-Red",Color);
(%o39)                          White-Black-Red
(%i40) properties(Mike);
(%o40)          [[user properties, Color, Weight, birthday]]
(%i41) get(Mike,Color);
(%o41)                          White-Black-Red
```

propvars は大域変数 props のリストに含まれる成分から, ⟨ 属性 ⟩ を持つアトムのリストを生成します. 例えば, propvars(atvalue) で atvalue 函数で値が設定されたアトムのリストを生成します.

```
(%i23) atvalue(f(x),x=0,0);
(%o23)                                0
(%i24) atvalue(g(x),x=1,0);
(%o24)                                0
(%i25) propvars(atvalue);
(%o25)                              [f, g]
```

大域変数 props には declare 函数,atvalue 函数や matchdeclares 函数等で属性が指定されたアトムが追加されたリストが割当てられています.

```
(%i1) props;
(%o1) [nset, kron_delta, dva, %n, %pw, %f, %f1, l%, solvep, %r, p, %cf,
algebraicp, hicoef, genpol, clist, unsum, prodflip, prodgunch, produ, nusum,
funcsolve, dimsum, ratsolve, prodshift, rforn, rform, nusuml, funcsol,
desolve, eliminate, bestlength, trylength, sin, cos, sinh, cosh, list2,
trigonometricp, trigsimp, trigsimp3, trigsimp1, improve, listoftrigsq,
specialunion, update, expnlength, argslength, pt, yp, yold, %q%, ynew, method,
%f%, %g%, msg1, msg2, intfactor, odeindex, singsolve, ode2, ode2a, ode1a,
desimp, pr2, ftest, solve1, linear2, solvelnr, separable, integfactor, exact,
solvehom, solvebernoulli, genhom, hom2, cc2, exact2, xcc2, varp, reduce, nlx,
nly, nlxy, pttest, euler2, bessel2, ic1, bc2, ic2, noteqn, boundtest, failure,
adjoint, invert]
(%i2) properties(invert);
(%o2)                        [transfun, transfun]
(%i3) properties(failure);
(%o3)                        [transfun, transfun]
```

```
(%i4) properties(kron_delta);
(%o4)    [symmetric, database info, kind(kron_delta, symmetric), rule]
(%i5) propvars(rule);
(%o5)                  [kron_delta, sin, cos, sinh, cosh]
```

この例で示す様に, 大域変数 props には属性が設定された Maxima の函数や変数が登録されています.

printprops 函数

printprops(⟨ アトム ⟩ ,⟨ 属性 ⟩)
printprops([⟨ アトム$_1$⟩ ,···,⟨ アトム$_n$⟩], ⟨ 属性 ⟩)
printprops(all,⟨ 属性 ⟩)

printprops 函数は ⟨ アトム ⟩ と ⟨ 属性 ⟩ に対応する属性値を表示する函数です.⟨ アトム ⟩ のリストも指定可能ですが, ⟨ 属性 ⟩ は一つだけです.

尚,printprops で表示可能な属性は以下のものに限定されます.

printprops で表示可能な属性

属性名	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	並びの照合変数属性.matchdeclare 函数で与えられます.

第一引数に all を指定すると, 指定した属性を持つ全てのアトムと値が表示されます.

```
(%i30) matchdeclare([_a,_b],true);
(%o30)                                done
(%i31) printprops(all,matchdeclare);
(%o31)                       [true(_b), true(_a)]
```

1.6　演算子

1.6.1　Maxima の演算子について

Maxima には数学で用いられる算術演算子, 変数に式を割当てる演算子等の様々な演算子があります. 演算子は利用者が新規に定義する事も可能で, 既存の演算子の属性を変更する事も容易に行えます.

Maxima の演算子には前置表現, 内挿式, 後置表現, 無引数の 5 種類の演算子の属性があります. 前置式は 1 引数の演算子で, 演算子が引数の前に置かれる書式になります.

例えば, 数学の微分演算子 $\frac{d}{dx}$ の様な書式になります. 次の内挿表現の演算子は + の様に二つの引数を演算子の左右に置く演算子です. そして, 後置式の演算子は 3! の様に演算子の後に引数を置く演算子で, 最後の無引数の演算子は引数を取らない演算子になります.

Maxima では演算子 (或いは函数) をその引数の数に応じて前置, 内挿, 後置表現演算子として指定したり, 既存の演算子の属性を変更する事が可能です. 演算子の属性は Maxima の属性を利用して与えています. この節では演算子の属性に限定して解説しますが, 属性の詳細に関しては,1.5 節を参照して下さい.

最初に簡単な例として, 適当な文字列を演算子として宣言してみましょう.

```
(%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, 函数としての実体が無い為に評価はこれ以上行われていません. この宣言の後に演算子の定義を行う事や函数の定義の後に演算子としての宣言を行う事も出来ます.

```
(%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
```

この例では函数 (mike や pochi) の定義を行っています. この定義で用いている:=も内挿表現の演算子の一つです. この様に Maxima での演算子は引数を括弧で括る必要がない函数とも看做せます.

1.6.2 演算子の束縛力

普通, 数学の演算子には順位があります. 例えば,1+a*b^2*c-d が与えられたとしましょう. 通常はこの式を (1+a*((b^2))*c))-d の様に演算子と被演算子と結び付きを演算子の強さで解釈していますね.Maxima の場合, この演算子の強さを束縛力 bp (Binding Power) と呼び,200 までの整数で表現しています. 更に, 演算子と被演算子の配置関係から束縛力は二つの属性, 即ち, 左束縛力 lbp(Left Binding Power) と右束縛力 rbp(Right Binding Power) に分けて指定出来る演算子もあり,Maxima では左右の束縛力を指定可能な演算子が大多数です.

この束縛力は演算子の属性に応じて利用者が自由に設定する事が出来ます. 因に,Maxima の組込演算子の属性 lbp と rbp の値は,nparse.lisp で設定されています. 例えば, 和演算子+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, 最後に差の演算子は lbp が 100 で rbp が 134 となっています. 左右の束縛力の値が大きなもの程, 演算子と被演算子の結び付きが強くなります. 例えば, 可換積が 120 ですが, 一方で冪は 140 となっています. これは冪の方が被演算子を引き付ける力が大きな事を意味し, それ故に,b^2*c は (b^2)*c と解釈される事になります.

但し, 演算子の束縛力を指定せずにしておく事も可能です. 特に束縛力を指定しない場合, デフォルトで 180 が設定されます. この事を実際に確認してみましょう.

```
(%i1) prefix("tama");
(%o1)                               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. この束縛力の設定は Maxima では LISP の属性リストを用いています. その為,get 函数を用いて調べられます. この例では, 演算子 tama が前置表現の為, 左側に演算子は不要となるので左束縛力を設定する必要はありません. 但し, 右束縛力は必要なので 180 が設定されている事が判ります.

次に, 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");
(%o4)                                mike
(%i5) :lisp (get '$mike 'lbp);
180
(%i5) :lisp (get '$mike 'rbp);
NIL
```

この場合は前置式演算子とは束縛力が逆になっている事が判りますね.

この束縛力は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. ここで, 小括弧の束縛力は最大で 200 となっています. その為, 処理に不安がある場合は, 小括弧でひと纏めにすると演算子の影響は括弧の中に及ばない事になります.

この束縛力で重要な事として, 演算子の左右の結合律にも影響する事が挙げられます.

```
(%i5) infix("><",100,120);
(%o5)                                 ><
(%i6) (a ><  b):=a^b;
                                                b
(%o6)                           (a >< b) := a
(%i7) a><b><c;
                                       b c
(%o7)                                 (a )
(%i8) infix("><",120,100);
(%o8)                                 ><
(%i9) a><b><c;
                                         c
                                        b
(%o9)                                   a
```

この例では演算子><を最初に左束縛力を 100, 右束縛力を 120 で定義しています. その為,a><b><c は右束縛力の方が強い為に,(a><b)><c で処理されます. ところが, 次に左束縛力を 120, 右束縛力を 100 と逆にした為, 演算子と左側の被演算子との繋がりが強くなった為に,a><b><c は a><(b><c) で処理されます. その為, 左右の結合律に関しては演算子の左右の束縛力の差も考慮しなければなりません.

又, 束縛力は内挿式や前置式の演算子本体の定義でも重要です. 函数として定義する場合, その時に用いる演算子:=の束縛力がある為です. この演算子:=の束縛力は左が 180. 右が 20 となっています. その為, 演算子の束縛力が弱い, 即ち, 演算子の右束縛力が 180 よりも小さい場合には演算子:=の側に被演算子が引き寄せられる事になります. この場合, 先程述べた様に小括弧で被演算子と一緒に演算子を括ります. この小括弧は最大値の 200 が左右の束縛力として設定されているからです. その為,Maxima の式を記述する際に, 演算子の束縛力に疑問があれば, 式を小括弧で一纏めにすると良いでしょう.

1.6.3 演算子の型

演算子の被演算子と結果に型があります. この型の指定は, 演算子の宣言を行う函数で行えますが, この時, 演算子の属性として指定されています. 先ず,nary 函数と matchfix 函数で宣言される演算子の場合, 被演算子の型は, argpos 属性に指定し, 返却値の型は pos 属性に設定します. その他の函数では, 左右の被演算子を独立して宣言します. 実際, 左被演算子と右被演算子は lpos(Left part of speech) 属性と rpos(Right part of speech) 属性, そして結果は pos(Part of speech) で指定します. その指定可能な型を以下に示します.

演算子の型

型	概要	属性値
expr	数式全般	algebraic
clause	論理式全般	logical
any	任意の Maxima の式	untyped

最初に expr は Maxima で扱う数式全般になります. これに対し clause は評価を行う事で,true や false を返す述語や論理式です.any は, これらの式を含む Maxima の任意の式となります.

尚,expr,clause,any には属性として english があり, その属性値は各々algebraic,logical,untyped 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
```

演算子宣言の函数で演算子を宣言する際に型を指定しなければ, 左右の被演算子の型と返却値の型として$any が自動的に設定されます. 尚, これらの属性は LISP の属性リストを用いて実現されている為,LISP の put 函数でも強引に設定が出来ます.

```
(%i4) prefix("mike");
(%o4)                                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);
(%o5)                       mike a := freeof(a, x)
```

```
(%i6) if mike (x^2+1) then print("test1");
test1
(%o6)                                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
                                       ^
```

この例では前置式演算子 mike を定義しています. デフォルトでは$any になっているものを LISP の函数 put を使って属性 pos の値を clause に変更します. 次に mike 本体を定義しています.if 文では mike は true か false の何れかを返す述語函数となっているので問題ありませんが, 次に LISP の put 函数を使って, 今度は型を expr 型にしています. ここで,if は述語を要求するので, 式 expr 型とは異なります, その為, エラーになっています.

1.6.4　演算子の属性を宣言する函数

ここでは最初に演算子の属性を宣言する函数から述べましょう. Maxima では利用者が函数やアトムを演算子として利用する事が可能です. 特に, 適当な文字列を演算子の属性を宣言する函数を用いて属性を与えてしまえば, それだけで演算子として利用が出来ます. 更に, 値もきちんと函数として定義しなくても, atvalue の様な属性で与える事も可能です. 以下に組合せを計算する演算子 C の定義の様子を示します.

```
(%i62) nary("C");
(%o62)                                C
(%i63) m C n:= m!/(n!*(m-n)!);
                                         m!
(%o63)                    m C n := -----------
                                   n! (m - n)!
(%i64) 5 C 3;
(%o64)                                10
```

演算子の属性を宣言する函数には以下のものがあります.

演算子の属性を宣言する函数

infix	infix(a)	文字列 a を内挿表現の演算子として宣言
infix	infix(a,lbp,rbp)	内挿表現の演算子 a に左右の束縛力を含めて宣言
infix	infix(a,lbp,rbp,lpos,rpos,pos)	内挿表現の演算子 a に左右の束縛力と型を含めて宣言
nary	nary(a)	文字列 a を内挿表現の演算子として宣言
nary	nary(a,bp)	内挿表現の演算子 a に束縛力を含めて宣言
nary	nary(a,bp,argpos,pos)	内挿表現の演算子 a に束縛力と型を含めて宣言
nofix	nofix(a)	文字列 a を無引数の演算子として宣言
nofix	nofix(a,pos)	無引数の演算子 a に出力の型を含めて宣言
postfix	postfix(a)	文字列 a を後置表現の演算子として宣言
postfix	postfix(a,lbp)	後置表現の演算子 a に左束縛力を含めて宣言
postfix	postfix(a,lbp,rpos,pos)	後置表現の演算子 a に左束縛力と型を含めて宣言
prefix	prefix(a)	文字列 a を前置表現の演算子として宣言
prefix	prefix(a,rbp)	前置表現の演算子 a に右束縛力を含めて宣言
prefix	prefix(a,rbp,rpos,pos)	前置表現の演算子 a に右束縛力と型を含めて宣言
matchfix	matchfix(a,b)	変数を文字列 a と文字列 b で挟む演算子を宣言
matchfix	matchfix(a,b,argpos,pos)	引数の型と結果の型を含めて宣言

この表で,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 が返却値の型です. 又,matchfix 函数と infix 函数では左右に分けずに演算子の束縛力を bp, 被演算子の型を argpos としています.

最初の infix は内挿 (infix) 表現の演算子の宣言を行います. 内挿表現の演算子は二つの引数を持つ函数で, $a + b$ の和の演算子 + の様に引数が演算子の前後に置かれるものです.nary 函数も内挿表現の演算子の宣言を行います.

nary 函数も, 函数を内挿表現演算子として宣言する事が出来ます. この nary 函数で宣言可能な演算子は左右の束縛力が同じでも構わないものに限定されます. とは言え, デフォルトでは全て 180 が設定されるので, 左右の束縛力を調整する必要が無ければ, nary 函数の方が変数が少ないので使い易いかもしれません.

nofix 函数は無引数演算子の宣言を行います. 無引数演算子は演算子が何等の引数を持たない事を明示する為に使われます.

postfix 函数は後置表記演算子の宣言を行います. 後置表記の演算子は一つの引数のみを持ち, 3! の様にその引数は演算子の前に置かれます.

prefix 函数は前置表記演算子の宣言を行います. 前置表記の演算子は一個の引数のみを持ち, その引数は演算子の直後に置かれるものです.

matchfix 函数は任意個数の引数を二つの文字列で囲む演算子を宣言します.

```
%i5) matchfix("@-","-@");
(%o5)                                  @-
(%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
```

尚,dispfun 函数を使って matchfix で定義した演算子に割当てた利用者定義函数の内容を見る場合, 演算子の先頭の文字列だけを二重引用符で括ったものを dispfun 函数の引数とします.

上記の函数で設定した属性は kill 函数や remove 函数で削除する事が可能です. 尚,remove の場合は属性のみを削除しますが,kill 函数の場合は演算子として定義した文字列そのものを削除してしまいます.

```
(%i10) nary("tama");
(%o10)                                 tama
(%i11) a tama b:=a+b^2;
                                                    2
(%o11)                          a tama b := a + b
(%i12) properties("tama");
(%o12)                     [function, operator, noun]
(%i13) remove("tama",op);
(%o13)                                 done
(%i14) properties("tama");
(%o14)                                  []
(%i15) prefix("mike");
(%o15)                                 mike
(%i16) mike x:=x!+1;
(%o16)                           mike x := x! + 1
(%i17) kill("mike");
(%o17)                                 done
(%i18) properties("mike");
(%o18)                                  []
```

この例では, 中置演算子 tama を定義し, 属性を削除する remove 函数で演算子 tama を削除しています. ここで,tama の属性は properties 函数を用いて調べられます. 演算子の属性を remove を実行された時点で tama の属性全てが削除されています. 又, 演算子 mike に対しては kill 函数を使って演算子 mike 全てを削除しています. この様に remove 函数と kill 函数で削除する事が可能です.

尚, この例では remove 函数は演算子の属性を指定した為, 副作用として演算子全体が削除されてしまいましたが,properties で調べた特定の属性だけを削除する函数です.

```
(%i19) nary("tama");
(%o19)                                tama
(%i20) a tama b:=a+b^2;
                                                    2
(%o20)                        a tama b := a + b
(%i21) remove("tama",function);
(%o21)                                done
(%i22) properties("tama");
(%o22)                          [operator, noun]
(%i23) 3 tama 4;
(%o23)                             3 tama 4
```

この例では, 函数を削除しただけで演算子としての属性は残しています. その為, `3 tama 4;` と入力してもエラーにはなりません.

演算子定義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

基本的に, 変数型と返却型は$any に設定されます. 尚,nary 函数と matchfix 函数で宣言された演算子の被演算子型は他の函数と異なり, 左右の変数型属性 lpos や rpos ではなく,argpos 属性で指定し, その属性値は$any です.

1.6.5　数式演算子

Maxima には数式に関連する二項演算子として以下のものがデフォルトで定義されています.

二項数式演算子

+	a + b	a と b の和
−	a − b	a と b の差
*	a * b	a と b の可換積
/	a / b	a の b による商
**	a ** b	冪 a^b
^	a ^ b	冪 a^b.a**b と同じ
.	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.

二項数式演算子の持つ属性

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

これらの演算子は通常の数値計算で利用される演算子と殆ど同じ表記になります. 但し,Maxima の内部では −,/ の扱いは表示された式とは別の表現となっています. 例えば,$x-y$ は $x+(-1)*y$,x/y は $x*y^{-1}$ 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. 又, 非可換積 . と非可換積の冪^^は Maxima 独特の演算子で, 関連する非可換積の冪^^と通常の可換積の冪^は別物です. 例えば,a^^3 は a . a . a を意味し,a^3 は a*a*a を意味します. 尚, 可換積*と非可換積., そして, それらに対応する冪乗^と^^は混在しても構いません. Maxima のデフォルトの表示では,^は x^3 に似せて上付きで表示され, 演算子^^は $x^{\langle n\rangle}$ の様に ⟨⟩ で括られて表示されます.

```
(%i1) a^^b;
                                        <b>
(%o1)                                  a
```

尚, 非可換積. と非可換冪^^は行列演算で特に利用します. この場合, 非可換積が通常の行列の積を意味し, 可換積は同じ大きさの行列に対し, 成分毎の積に対応します.

```
(%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 ]
```

Maxima では可換積の冪^と非可換積の冪^^の計算結果が長くて式が表示し切れない場合, 記号 expt と ncexpt が各々の演算の表記で利用されます.

非可換積. と非可換冪^^にはその挙動を制御する様々な大域変数が存在します. 以下に, 代表的な大域変数を示しておきます.

非可換積演算子に関連する大域変数

変数名	初期値	概要
dot0nscsimp	true	true の場合, 零と非スカラの項の非可換積を可換積に変換します.
dot0simp	true	true の場合, 零のスカラ項の非可換積を可換積に変換します.
dot1simp	true	true の場合,1 との他の項との非可換積を可換積に変換します.
dotassoc	true	true の場合, $(a.b).c$ と $a.(b.c)$ を $a.b.c$ に変換します. 即ち, 結合律を式に対して自動的に適用します.
dotconstrules	true	true の場合, 定数と項の非可換積が可換積で置換されます. この大域変数は dotocimo,dotonscsimp,dot1simp も影響を与えます.
dotdistrib	false	true の場合,$a.(b+c)$ を $a.b+a.c$ に置換します. 即ち, 分配律を自動的に適用します.
dotexptsimp	true	true の場合, 複数の同じ式による非可換積を非可換積の冪乗に変換します.
dotident	1	非可換冪の 0 乗で返される値を設定します.
dotscrules	false	true の場合, 群環の元 a とスカラー b との非可換積を可換積に変換します.

尚, ここでのスカラーと群環の元との非可換積に関する大域変数は declare 函数で scalar 属性を与えた式に対して制御が効きますが,1,2 等の数値に対しては, これらの大域変数とは無関係に可換積で置換されます.

```
(%i6) 2 . 3 . x . y;
(%o6)                                6 (x . y)
```

尚, 非可換積を Maxima で記述する場合,a . b の様に空白を空けるべきです. これは引数が数値の場合は必須です. 例えば,1.2 は 1 と 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 の様に半端に空行を入れてしまうと Maxima の式としての意味を持ちません.

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為, 非可換積に対する規則を定義した場合に, その適用が容易になります.

後置式の数式演算子

!	$n!$	n が整数の場合,n の階乗を計算. 一般の場合 $\Gamma(x+1)$
!!	$n!!$	n 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 では階乗に関連する！と !! が後置式の演算子となります. 但し, !! は正確には演算子ではありませんが, 演算子の属性を持たせた函数です.

先ず, $n!$ は n が整数の場合に n の階乗を計算します. 一般の数値に対しては $\Gamma(n+1)$ の計算を行います. ここで, $\Gamma(n+1)$ の Γ は Γ 函数と呼ばれる函数で, $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$ で定義されています.

$n!!$ は少し変った函数で, 整数 n が奇数の場合, n 以下の全ての奇数の積を計算し, n が偶数の場合は同様に n 以下の全ての偶数の積を計算します. その為, $n! = n!!(n-1)!!$ が成立します. この函数は, $\prod_{i=0}^{entier\left(\frac{n}{2}\right)}(n-i)$ の計算を行っています. この $n!!$ の計算では, $n!$ と違い整数以外の数値に対して Γ 函数を用いるものではありません.

```
(%i6) 10!;
(%o6)                                3628800
(%i7) 10!!;
(%o7)                                 3840
(%i8) 9!!;
(%o8)                                  945
(%i9) 10!!*9!!;
(%o9)                                3628800
```

これらの演算子の属性を以下に纏めておきます.

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				

1.6.6 論理演算子

Maxima の論理演算子は, 式や述語を被演算子とし,true や false といった値を返す Maxima の式, 即ち, 述語を構成する演算子です. 否定の not, 論理和の or や論理積の and を除くと基本的に二項間関係を表すものとなります.

論理演算子

not	not a	述語 a を否定
and	a and b	述語 a と b の論理積
or	a or b	述語 a と b の論理和
=	a = b	a と b が等しい
#	a # b	a と b が等しくない
>=	a >= b	a は b 以上
>	a > b	a は b よりも大
<=	a <= b	a は b 以下
<	a < b	a は b より小

先ず,C の&&や——に相当するものが,and と or になります. 二項間の関係を示す演算子は,Maxima 独特の#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では:を用い, 等しい事を示す関係子に=を用います.C の==に相当します. この割当の演算子と等号は混同し易いので注意して下さい.

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

1.6.7　割当の演算子

割当の二項演算子

:	a : b	a に b の値を割当てる
::	a :: b	a に b の値を割当てる
::=	a ::= b	本体が b のマクロ a を定義
:=	a:=b	本体が b の函数 a を定義

通常の変数に何等かの値を割当てる場合には:を利用します. ここで, 演算子=は前述の様に割当ではなく, 二つの式が等しい事を示す演算子となるので意味が異なります.

C で割当に用いられる演算子:=は Maxima で函数の定義で用います.

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

尚, 割当の演算子の左束縛力は演算子を宣言する函数で宣言した演算子の束縛力と同じ大きさですが,infix 函数等で右束縛力を 180 よりも小さくすると, 割当の演算子に吸い取られてしまうので演算子の定義で, 演算子と被演算子を含む左辺全体を小括弧括る等の注意が必要になります.

```
(%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;
(%o9)                          (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 11 に設定した為, 最初に演算子を定義した時に, 仮の被演算子 y が演算子:=に引き寄せられてしまい, その結果エラーになっています. そこで,x tama y 全体を小括弧で括ります. この括弧の束縛力は 200 の為,:=の束縛力はそこで遮断され, 演算子 tama が無事に定義出来ます.

1.6.8 その他の演算子

Maxima では式を構成する多くが内部的には演算子としての扱いになっています. 特に束縛力は式の評価やプログラム作製で大きく影響します.

記号の属性

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

先ず, 大括弧や小括弧にも束縛力があります. この束縛力を上手く使う事で, block 文内部の Maxima の式や被演算子をグループ化する事を可能にしています.

更に,if 文や do 文, そして関連する語句にも同様の属性があります.

if 文に関連する演算子

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

do 文に関連する演算子

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

1.7　規則と式の並びについて

Maxima は単純に式を纏めたり展開する事で式を変形するだけではなく, 函数や変数に規則を与える事で式の置換操作を行う事が可能です.

例えば, $\tan(x)$ と $\frac{\sin(x)}{\cos(x)}$ が同値であるという事も規則のひとつです.

Maxima にはデフォルトで幾つかの規則が設定されています. これらは主に三角函数に関するものです. 規則名だけを知りたい場合, 大域変数の rules に規則名のリストが設定されているので, rules; と入力してみて下さい. 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.

```
(%i1) rules;
(%o1) [trigrule0, trigrule1, trigrule2, trigrule3, trigrule4,
       htrigrule1,htrigrule2, htrigrule3, htrigrule4]
```

ここで表示される規則の具体的な内容はどの様なものなのでしょうか？それを表示する函数として disprule 函数や letrules 函数が存在します.

規則の表示を行う函数

disprule(⟨ 規則$_1$⟩ ,⟨ 規則$_2$⟩ ,$\cdots$)
disprule(all)
letrules(⟨ 規則パッケージ ⟩)
letrules()

disprule 函数や letrules 函数は Maxima に設定した規則の表示を行う函数です. disprule 函数は defrule 函数,tellsimp 函数,tellsimpafter 函数で利用者が設定した規則の詳細を 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. 特に, 引数を all とすると Maxima が持つ全ての規則を表示します.

これに対して letrules 函数は let 函数で定義した規則を表示する函数です. 引数として指定した ⟨ 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 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. 尚, この大域変数の初期値は default_let_rule_package です.

では, 実際に `disprule(all);` を実行してみましょう.

```
(%i10) disprule(all);
                                          sin(a)
(%t10)                  trigrule0 : tan(a) -> ------
                                          cos(a)

                                          sin(a)
(%t11)                  trigrule1 : tan(a) -> ------
                                          cos(a)

                                            1
(%t12)                  trigrule2 : sec(a) -> ------
                                          cos(a)

                                            1
(%t13)                  trigrule3 : csc(a) -> ------
                                          sin(a)

                                          cos(a)
(%t14)                  trigrule4 : cot(a) -> ------
                                          sin(a)

                                           sinh(a)
(%t15)                 htrigrule1 : tanh(a) -> -------
                                           cosh(a)

                                             1
(%t16)                 htrigrule2 : sech(a) -> -------
                                           cosh(a)

                                             1
(%t17)                 htrigrule3 : csch(a) -> -------
                                           sinh(a)

                                           cosh(a)
(%t18)                 htrigrule4 : coth(a) -> -------
                                           sinh(a)
```

disprule 函数を使って表示される規則は, 先ず, 左端が規則名,->の左側が適用される函数, 右側が適用される函数が置換される函数となります. 例えば, 最初の trigrule0 は $\tan(x)$ を $\frac{\sin(x)}{\cos(x)}$ で置換する規則です.

ところで,Maxima に tan(x); と入力しても, 直ちに変換される訳ではありません. 実際の利用では函数 tan は式の中にあり, その変数も x の様な単純なものであるとは限りません. その為, 与えられた式の並び方 (パターンとも呼びます) を指定し, その並び方に対して規則を当て嵌める処理を実行しなければなりません. この処理を並びの照合 (パターンマッチング) と呼びます.

ここで trigrule0 は単純に $\tan(x)$ を $\frac{\sin(x)}{\cos(x)}$ で置換する事を意味しますが, この置換の為には, 置換を行う函数を必要とします. disprule 函数で表示される規則は apply1,apply2,applyb1 函数を用います.

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$_n$⟩)

apply 函数族は基本的に defrule 函数で与えられた規則を式に適用させる函数ですが, 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) から作用させます.

最初の apply1 函数は与えられた ⟨ 式 ⟩ に対して ⟨ 規則$_1$⟩ を最初に作用させます. この時, ⟨ 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 函数は ⟨ 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時に限って, 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 函数は apply1 函数と似ていいますが,apply1 函数が ⟨ 式 ⟩ の木構造に対し, 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, 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.

尚,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. 但し, これらのデフォルト値は 10000 の為, 通常の利用では殆ど無制限と言っても構わないでしょう.

では,apply1 函数を使って, 函数 tan を含む式に Maxima に組込の規則 trigrule0 を適用してみましょう. この規則は式中の $\tan(x)$ に規則 trigrule0 を適用する事で $\frac{\sin(x)}{\cos(x)}$ に置換します. この時, 函数 tan の変数は x である必要はありません. 一般的な式でも構いません.

```
(%i19) tan(x);
(%o19)                                tan(x)
(%i20) apply1(tan(x),trigrule0);
                                      sin(x)
(%o20)                                ------
                                      cos(x)
```

```
(%i21) apply1(tan(a1*x+y+b1),trigrule0);
                              sin(y + a1 x + b1)
(%o21)                        ------------------
                              cos(y + a1 x + b1)
```

1.7.1 規則の定義と適用

Maxima の規則を利用者が定義する事も可能です. この場合, 規則の定義は defrule 函数や let 函数で行います.

ここでは最初に defrule 函数による規則の定義方法を示しましょう.

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.

この場合 ,⟨ 規則名 ⟩ で指定された規則が apply 函数族の函数の一つで式に適用されると, ⟨ 並び ⟩ に適合する全ての部分式が ⟨ 置換 ⟩ で指定した値で置換されます. 尚, 照合に失敗すると元の式を返却します.

次に,defrule 函数を使って前置式演算子 dfx に適当な規則を入れてみましょう.

```
(%i1) prefix("dfx");
(%o1)                                dfx
(%i2) defrule(chain1,dfx(a.b),dfx(a).b+a.dfx(b));
(%o2)           chain1 : dfx (a . b) -> dfx a . b + a . dfx b
(%i3) apply1(dfx(a.b),chain1);
(%o3)                        dfx a . b + a . dfx b
(%i4) apply1(dfx(x.y),chain1);
(%o4)                             dfx (x . y)
```

最初に dfx が前置式演算子である事を prefix 函数で宣言します. 次に,dfx が dfx(a.b) → dfx(a).b+a.dfx(b)) となる規則 chain1 を defrule 函数で定めます.

それから apply1 函数を用いて dfx(a.b) に規則 chain1 を適用すると期待通りの結果を得ます. ところが,dfx(x.y) を計算させると残念な事に,dfx(x.y) のままです. これは何が悪かったのでしょうか？

この例では, 式の並びを定義する際に用いた変数に対し, その変数が満すべき条件を定義していないからです.Maxima では変数が満すべき述語が与えられなければ, defrule 函数や let 函数で用いられた変数の並びに対してのみ規則が適用される仕様となっています. 従って, より一般的な規則を定義したければ,defrule 函数や let 函数で規則を定義する前に, 予め ⟨ 並び ⟩ で用いる変数が満すべき条件を付加する必要があります. この処理を行うのが matchdeclare 函数です.

matchdeclare 函数

matchdeclare(⟨ 変数 ⟩ ,⟨ 述語函数 ⟩ , ···)
matchdeclare([⟨ 変数$_1$⟩ ,···,⟨ 変数$_1$⟩], ⟨ 述語函数 ⟩ ,···)

matchdeclare 函数は引数として ⟨ 変数 ⟩ と ⟨ 述語函数 ⟩ の変数と述語函数名の対を取ります. 従って, 引数は常に偶数個になります. ここで指定する述語函数は,matchdeclare で宣言する変数を与えると,true か false の何れかが返却される函数です. 述語函数として, 特に指定する函数が存在しない場合は true, その他には,Maxima 組込の述語函数,lambda 函数や block 函数を含まない利用者定義函数が使えます.

例えば, matchdeclare(q,freeof(x,%e)) とすれば, 変数 q は x と%e を含まない全ての式に適合します. 又, 述語には複数の引数を持つ函数で, 最後の引数に matchdeclare 函数で宣言する変数が入るものを与える事も可能です.

尚,matchdeclare 函数によって ⟨ 変数 ⟩ には matchdeclare 属性が付加されます. この属性は printprops 函数で参照する事が可能です.

matchdeclare 函数に与える変数と述語函数の対は, 後で行う規則の定義で, 変数が満すべき条件を述語函数が与え, この述語が true となる変数に対してのみ defrule 等で定めた規則が適用されます. その為, 無条件に規則を適用したければ, 述語函数を true とします. 又, 複数の変数が同じ述語を満す場合 に ⟨ 変数 ⟩ の個所を ⟨ 変数リスト ⟩ で置換える事も可能です.

では,matchdeclare 函数で並び変数を予め定義し, それから defrule 函数で規則を定義してみましょう.

```
(%i1) prefix("dfx");
(%o1)                                dfx
(%i2) matchdeclare([_a,_b],true);
(%o2)                               done
(%i3) defrule(chain1,dfx(_a._b),dfx(_a)._b+_a.dfx(_b));
(%o3)         chain1 : dfx (_a . _b) -> dfx _a . _b + _a . dfx _b
(%i4) apply1(dfx(a.b),chain1);
(%o4)                        dfx a . b + a . dfx b
(%i5) apply1(dfx(x.y),chain1);
(%o5)                        dfx x . y + x . dfx y
```

この例では, 変数_a と_b を常に true とし, これらの変数に対して規則をあてはめています. その為, 一般の変数に対しても同様に規則の適用が実行されます.

但し,matchdeclare で宣言した変数の可換積 $*$ や和 $+$ を取る場合には注意が必要です.

```
(%i7) matchdeclare([_c,_d],true);
(%o7)                               done
(%i8) defrule(chain1,dfx(_c*_d),dfx(_c)*_d+_c*dfx(_d));
_d _c partitions 'product'
```

```
(%o8)            chain1 : dfx (_c _d) -> _c dfx _d + dfx _c _d
```

この例では,defrule で規則を定義すると,`_d _c partitions 'product'` と表示されています. この調子で apply1 函数を実行するとまともな処理が出来ずに, Maxima が落ちてしまいます.

matchdeclare 函数で宣言した変数の可換積*や冪^を含む式に対しては, let 函数を用いる事が出来ますが, 上記の例の様に函数の引数に可換積*や冪^を含ませる場合には,let 函数は有効ではありません. 寧ろ, 非可換積を用いるか, 適当な演算子を宣言して defrule 函数を用いる方が良いでしょう.

let 函数

let(⟨ 項 ⟩ ,⟨ 式 ⟩ ,⟨ 述語 ⟩ , ⟨ 変数$_1$⟩ ,···,⟨ 変数$_n$⟩ , ⟨ パッケージ名 ⟩)
let(⟨ 項 ⟩ ,⟨ 式 ⟩ ,⟨ 述語 ⟩ , ⟨ 変数 ⟩ ,···,⟨ 変数$_n$⟩)
let(⟨ 項 ⟩ ,⟨ 式 ⟩)

let 函数は ⟨ 述語 ⟩ が true の場合, ⟨ 項 ⟩ を ⟨ 式 ⟩ で置換する規則を定義する函数です.

この ⟨ 項 ⟩ には, アトム,sin(x) や f(x,y) の様な函数の可換積 $*$, 商/や冪^を含む項となります. 但し, 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 ⟩ を省く場合には, 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, 自動的に 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. 各々の規則の組合わせは任意数の let 函数で操作された任意の数の規則を含む事が可能で, 利用者が与えた名前で参照されます.

let 函数で定義した規則を式に適用する場合は,letsimp 函数を用います.

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 ⟩)

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, 式の変化が無くなるまで規則の適用を続けます.

尚, 規則パッケージの指定が無い場合,current_let_rule_package に割当てられた規則パッケージが利用されます.

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されます. 例えば, letsimp(expr,package1,package2) を実行すると, 最初に letsimp(expr,package1) を実行し, 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 current_let_rule_package が切替えられる事はありません.

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)^2-1,tama(2*_a));
                              2
(%o2)                    tama (_a) - 1 --> tama(2 _a)
(%i3) letsimp(tama(x)^2);
                                      2
(%o3)                               tama (x)
(%i4) let(tama(_a)^2,tama(2*_a)+1);
                              2
(%o4)                    tama (_a) --> tama(2 _a) + 1
(%i5) letsimp(tama(x)^2);
(%o5)                            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.
最初に,let(tama(_a^2-1,tama(2*_a))としている為,letsimp 函数による規則の適用に失敗している事に注意して下さい.let 函数では置換される並びは項に限定されます.

tellsimp 一家

tellsimp(⟨ 並び ⟩ ,⟨ 置換 ⟩)
tellsimpafter(⟨ 並び ⟩ ,⟨ 置換 ⟩)

tellsimp 函数と tellsimpafter 函数は各々規則に沿った置換を実行します. これらの函数は共に似ていますが, ここでの置換は, 規則を適用する前に置換を行うか, 規則を適用した後で置換を行うかの違いがあります.

最初の tellsimp 函数は, 新しい情報を古い情報の前に置いて, 組込の簡易化規則よりも前に適用します. この tellsimp 函数は簡易化が実行される前に式を改変する事が必要な場合に用います. ここでの ⟨ 並び ⟩ には和, 積, 単変数や数値は使えません. 置換には規則名のリストを指定し,defrule 函数, defmatch 函数,tellsimp 函数や tellsimpafter 函数で追加されたものです.

tellsimpafter 函数は tellsimp 函数と同様に ⟨ 並び ⟩ に対する ⟨ 置換 ⟩ を定義します. ここで指定する ⟨ 並び ⟩ は Maxima 組込みの簡易化規則の適用後に用いるものになります. この ⟨ 並び ⟩ には単変数や数を除く任意の式が設定出来ます.

defmatch 函数

defmatch(⟨ 函数名 ⟩ ,⟨ 並び ⟩ , ⟨ 助変数$_1$⟩ ,$\cdots$,⟨ 助変数$_n$⟩)

defmatch 函数は n+1 個の引数を持ち, 特定の並びに適合するかどうか式を検査する函数を ⟨ 函数名 ⟩ で指定した名前で生成する函数です.

defmatch 函数の ⟨ 並び ⟩ は変数と ⟨ 助変数$_1$⟩,$\cdots$,⟨ 助変数$_n$⟩ を含む式になります. ここで変数

が既に matchdeclare 函数で宣言されていたとしても, defmatch 函数の ⟨ 助変数$_i$⟩ の引数として defmatch 函数に与えます.

函数の最初の引数は並びに対して照合する式で, 他の n 個の引数は式中の実際の値, 並びの中で変数として置かれるものです.

defmatch で構成された函数は照合に成功すると ⟨ 助変数$_i$⟩ = 適合する変数のリスト を返します. ここで, 照合に失敗すれば false を返します.

次の例では, 与えられた函数の線形性を調べる函数 linear の定義を行い, 函数 linear を実行しています.

```
(%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]
```

最初に defmatch で linear を定義し, 次に式 `3*z+(y+1)*z+y^2` が変数 z の一次式であるかどうか検証しています. この例では false が返却されています. これは最初の変数 a と b に関して何等の情報も無い為, 単純にアトム a とアトム b を持たない一次式と判断されて false となっています. 次の a*z+b の場合, 並び変数 x に対応するものが z であると判断した為に [x = z] が返却されています.

そこで, 並び変数 a と b の情報を追加します. この例では, 最初に is(e#0 and freeof(x,e)) と同値な函数 nonzeroandfreeof を定義します. それから並び変数 a と b に対して, 共に 0 ではなく, 変数 x も含まない式であると matchdeclare 函数を用いて宣言しています. この宣言の後に 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, 並び変数 a と b に関する情報は更新されません. これによって並び変数 a と b が変数 x から独立したもので, 変数 x に関して線形であれば, 対応する成分リストを返す函数 linear が定義されます.

そこで, linear(3*z+(y+1)*z+y^2,z) を実行すると,linear で与式と並びの式 a*x+b を比較し, 今度は [b=y^2, a=y+4, x=z] を返します.

1.7.2 規則の削除

規則の削除を行う函数

remlet(⟨ 項 ⟩)
remlet(⟨ 項 ⟩ ,⟨ パッケージ名 ⟩)
remlet(all)
remlet()
remrule(⟨ 函数 ⟩ ,⟨ 規則 ⟩)
remrule (all)

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() や引数が 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, 例えば, 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 で新しい規則を削除すれば, 古い規則が復活します.

remrule 函数は ⟨ 規則 ⟩ で指定した規則を ⟨ 函数 ⟩ から削除します. この ⟨ 規則 ⟩ は,defrule 函数,defmatch 函数, tellsimp 函数や tellsimpafter 函数で設定したものです.remrule 函数も引数が all の場合は全ての規則を削除します.

1.7.3 規則に関連する大域変数

規則に関連する大域変数

変数名	初期値	概要
maxapplydepth	10000	apply1 と apply2 が停止する階層
maxapplyheight	10000	applyb1 が停止する階層
current_let_rule_package	default_let_rule_package	利用規則パッケージ名
letrat	false	letsimp の動作に影響
let_rule_packages	default_let_rule_package	規則パッケージのリスト

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 式の様な階層構造があります.apply1 等の函数は maxapplyheight よりも低い個所, 即ち, 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してみます). 但し, デフォルト値の 10000 は式の殆ど全てと言える程の高さになるでしょう.

大域変数 current_let_rule_package に let_rule_package で利用した規則パッケージ名が設定されます.let 命令で定義したどの様な規則パッケージ名も, この変数に再設定して構いません.let パッケージに関連するどの様な函数でも, current_letl_rule_package が何時でも使えます.

大域変数 letrat が false であれば,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. 但し, $n!/n$ を $(n-1)!$ にする様な置換は出来ません. この様な置換を行う為には, 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.

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.

1.8　評価

1.8.1　ev 函数

この節では Maxima で最も重要な函数の一つである ev 函数を中心に解説します.

ev 函数

ev(⟨ 式 ⟩, ⟨ 引数$_1$⟩, ··· , ⟨ 引数$_n$⟩)
　⟨ 式 ⟩, ⟨ 引数$_1$⟩, ··· , ⟨ 引数$_n$⟩

ev 函数は与式 ⟨ 式 ⟩ の評価を実行しますが, その際に, 評価を行う為の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. この環境は ⟨ 引数$_1$⟩, ··· , ⟨ 引数$_n$⟩ で設定されるもので,Maxima の様々な大域変数を true にしたり, 式の変数に値を割当てる事, 更には, 式を評価する函数 (evfun) を指定する事で設定されるものです.

尚,Maxima の最上層のみで ev() を外した表記も可能です.

```
(%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 $(x+1)^4$ の展開を行います. 最初の書き方が ev 函数の正規の記述方法ですが,Maxima の最上層に限定して (x+1)^4,expand の様に ev() を省略出来ます. 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, 即ち, 通常の入力プロンプトが出ている場合や Maxima のバッチファイル内部にはこの表記が利用出来ます.

次に, 大域変数 dotassoc の値を変更して式の評価を実行しています. ここで, 非可換積の結合律を制御する大域変数 dotassoc はデフォルトで true となっている為, (x . y) . z は x . y . z に自動的に評価されます. ここで,ev 函数で評価を行う際に, 一時的に大域変数 dotassoc を false にしています.ev 函数の評価と大域変数の値が無関係な事に注目して下さい.

最後に evfun 属性を持つ函数である factor 函数と ratsimp 函数を使って評価しています. ここで evfun 属性を持たない函数を用いた場合, 式が評価されずにそのままの式が返されます.

更に,ev 函数では式に含まれる変数に値を一時的に割当てて式を評価する事も可能です. この評価は方程式の解を解いた後に, その結果を使って他の式の評価する場合や解の検証で便利です.

```
(%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, 最初に連立方程式 $x^2 - y^2 + xy - 1 = 0, x + y - 3 = 0$ を解き, その結果を用いて式 xy を評価しています. これは ev 函数の性質で, 引数に論理演算子=を含む式で, 左辺が通常の変数であれば, 左辺の変数に右辺の式が割当てられます.

ここで, 具体的に ev 函数で利用可能な引数を示しましょう.

ev 函数で利用可能な引数

evflag 属性を持つアトム	evflag 属性を持つアトムに true を一時的に設定.
evfun 属性を持つ函数	evfun 属性を持つ函数を, 引数の左側から順番に式に適用.
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 の値を各々割当てて式を展開.
expand(⟨ 整数$_1$⟩,⟨ 整数$_2$⟩)	大域変数 maxposex に ⟨ 整数$_1$⟩ ,maxnegex に ⟨ 整数$_2$⟩ を各々設定し式を展開.
eval	式を評価.
noeval	式を無評価.
nouns	式を名詞型で評価.
numer	大域変数 numer と大域変数 float を true にして式を評価
risch	risch 積分を実行
diff	式中の名詞型の微分を評価.
derivlist($\langle x_1\rangle,\cdots,\langle x_n\rangle$)	名詞型の微分で, $\langle x_1\rangle,\cdots,\langle x_n\rangle$ による微分のみを評価.
変数への式の割当	大域変数や式中の変数に割当と式の評価を実行.
local($\langle x_1\rangle,\cdots,\langle x_n\rangle$)	ev 内部で用いる局所変数 $\langle x_1\rangle,\cdots,\langle x_n\rangle$ を宣言.
detout	逆行列の計算で行列式を行列の外に出す.

ev 函数では,evflag 属性を持つ引数を true に設定したり, 大域変数に値を設定する事で式の評価の為に局所的な環境設定を行い, それから, 与式に代入を実行し, evfun 属性を持つ函数や, 大域変数の値による Maxima の自動変換によって式を評価します.

先ず,evflag 属性を持つ大域変数はデフォルトで以下のものがあります.

evflag 属性をデフォルトで持つ大域変数

```
float,pred,simp,numer, detout, exponentialize, demoivre,
keepfloat, listarith, trigexpand, simpsum, algebraic,
ratalgdenom, factorflag, %emode, logarc, lognumer,
radexpand, ratsimpexpons, ratmx, ratfac, infeval, %enumer,
programmode, lognegint, logabs, letrat, halfangles,
exptisolate, isolate_wrt_times, sumexpand, cauchysum,
numer_pbranch, m1pbranch, dotscrules,logexpand
```

大域変数に evflag 属性を持たせる為には, declare 函数を用います. 更に, ある大域変数が evflag 属性を持つかどうかは,properties 函数を使って調べられます.

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以下に示します.

```
(%i1) declare(tama,evflag);
(%o1)                                done
(%i2) tama:false;
(%o2)                                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. 最初の例では tama には false を設定している為, 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為, 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, 結果として tama が true の場合の処理が実行されている事が判ります. 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 が返却される事で判別出来ます.

evflag 属性に似た属性に evfun 属性があります. これは函数に対する属性で,ev 函数の引数として利用可能な函数である事を意味します. デフォルトで evfun 属性を持つ函数を以下に示しておきます.

evfun 属性を持つ函数

```
factor,trigexpand,trigreduce,bfloat, ratsimp,ratexpand,
radcan,logcontract,rectform,polarform
```

evflag 属性と同様に,declare 函数を用いて函数に evfun 属性を持たせる事が出来ます. ここでは簡単な函数を定義して, 実際にその作用を観察してみましょう.

```
(%i1) mike(z):=diff(z,x,2);
(%o1)                      mike(z) := diff(z, x, 2)
(%i2) properties(mike);
(%o2)                              [function]
(%i3) x^2,mike;
                                     2
(%o3)                                x
(%i4) declare(mike,evfun);
(%o4)                                done
(%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為に ev 函数による評価が出来ません. ところが,declare 函数で evfun 属性を付加すれば,ev 函数による評価では mike が実行され, 結果として二階微分が得られます. 函数が evfun 属性を持つかどうかの判定は,properties 函数や, LISP の get 函数を用いて行えます. この点も evflag 属性と同様です.

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 函数から順番に式に作用させます.

```
                                                2
(%o29)                       tst(z) := expand(z )
(%i30) declare(tst,evfun);
(%o30)                                done
(%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)
```

この例で示す様に,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. 先ずはじめに ev 函数の引数として左から tst,factor の順に与えた為,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, 式は展開されています. 即ち,tst(factor(与式)) で処理したからです.

expand を指定すると, 大域変数 maxposex と maxnegex に設定された値を大域変数 expop と expon に各々割当てて式の展開を行います. 尚, 大域変数 expop と expon は $(x+1)^3$ の様な冪乗で自動展開する次数を指定する大域変数です.

大域変数 maxposex と maxnegex は expand 函数で展開する冪乗の最大と最小次数を設定する大域変数です. デフォルトで大域変数 maxposex と maxnegex の値が 1000 の為, 冪の最大次数が 1000

以下であれば式の展開を自動実行します. expand(⟨ 整数$_1$⟩,⟨ 整数$_2$⟩) の場合は大域変数 maxposex に ⟨ 整数$_1$⟩ ,maxnegex に ⟨ 整数$_2$⟩ を各々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 を越えている為に展開を行いません. 次の例では,expop に 2,expon に 3 を指定していますが, 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 の為, 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為に, 負の冪乗の部分式の展開のみが実行されません.

noun の場合は与式を評価しません.

numer の場合, 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
```

尚, 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っています. 但し,%e の冪の場合は%e を浮動小数点に変換しません.

risch は integrate 函数に 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, rischint 函数を用い, それ以外は sinit 函数を用いる様になっている為です.

derivlist は函数に似た構文を持っています.

derivlist

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);
                              2
                             d      4        2    2
(%o9)                        ----- (y  + 2 x y  + x )
                             dx dy
(%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
```

この derivlist 引数に似たものに local 引数があります. この local も derivlist に似た構文を持ちます,

local

local(⟨ 変数$_1$⟩ ,···,⟨ 変数$_k$⟩)

こちらは ev 函数内部で利用する局所変数の宣言に用いられます. 但し,ev 函数では後述の式への割当がある為,local で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
```

式の割当は,x=1 や x:1 等によって, 評価する式の変数に値を割当てる事で与式の評価を実行するものです.

```
(%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,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には, リストで与えます. この際に, 演算子=と: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.

detout は大域変数 detout と似た結果が得られます.detout を指定した場合, 大域変数の doallmxops と doscmxops を false にし, 大域変数の detout を true にした場合と同じ結果が得られます.

```
(%i7) A:matrix([1,2,3],[4,3,1],[2,4,1]);
                                  [ 1  2  3 ]
                                  [         ]
(%o7)                             [ 4  3  1 ]
                                  [         ]
                                  [ 2  4  1 ]
(%i8) A^^(-1),detout;
                                [ - 1  10   - 7 ]
                                [               ]
                                [ - 2  - 5  11  ]
                                [               ]
                                [ 10    0   - 5 ]
(%o8)                           -----------------
                                       25
```

1.8.2　評価に関連する函数

評価に関連する述語函数

equal(⟨ 式$_1$⟩,⟨ 式$_2$⟩)
is(⟨ 述語 ⟩)

equal 函数は is 函数と一緒に使われ, ⟨ 式$_1$⟩ と ⟨ 式$_2$⟩ が ratsimp 函数で指定された全ての可能な変数値に対して等しければ true, 等しくない場合には false を返します.x が不定元であっても is(equal((x+1)^2,x^2+2*x+1)) は true を返しますが,is((x+1)^2=x^2+2*x+1) は false を返します.

is(rat(0)=0) は false ですが,is(equal(rat(0),0)) は true となる事に注意して下さい. もし,equal で判別出来ない場合, 同値だが簡易化された形式で返されますが,=を使っていれば常に true か false が返されます. 式中の全ての変数は実数値であると予め仮定しています.

尚, ev(⟨ 式 ⟩,pred) は is(⟨ 式 ⟩) と同じ事を実行しています.

```
(c1) is(x^2 >= 2*x-1);
(d1)                                true
(c2) assume(a>1);
(d2)                                done
(c3) is(log(log(a+1)+1)>0 and a^2+1>2*a);
(d3)                                true
```

is 函数は ⟨ 述語 ⟩ が,Maxima の文脈や宣言等に含まれている事象に適合するかどうかを判定します.is 函数が true と返すのは, ⟨ 述語 ⟩ に含まれる変数に関して, 全ての値で述語が true となる場合で, そうでない場合は false を返します. それ以外は大域変数 prederror の設定に依存します. つまり, 大域変数 prederror が true の場合,is 函数はエラーを出力を行い,falase であれば unknown を返します.

函数評価に関連する函数

'⟨ 式 ⟩
"⟨ 式 ⟩
eval(⟨ 式 ⟩)

単引用符' は Maxima による評価を防止します. 例えば,'(f(x)) とする事で Maxima に式 f(x) を評価しない事を報せます. 一方で'f(x) は x に函数 f を作用させ, その名詞型で返す形になります. この際,f の評価は行なわれませんが x は評価されてしまいます.

二つの単引用符" は特殊な評価を行います. 例えば,"%o4 で%i4 を再評価します. 又,"f(x) は函数 f を x に作用させたものを動詞型で返します.

```
(%i65) test:2*%pi;
(%o65)                                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.LISP の `eval` 函数と同じ働きをします.

1.9　LISP に関係する関数

1.9.1　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, プログラムは様々な函数を定義し, それらを組合せて行く作業とも言えます. 因みに,C や FORTRAN は手続型と呼ばれます.

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 を意識する事は単純な利用では殆どありません.

但し,Maxima で酷いエラーを出すと LISP のデバッガに落ちる事があります. 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. すると Maxima に戻ります.

LISP の特徴は言語仕様が非常に柔軟な点です.LISP にはアトムと呼ばれる変数があり, 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データが, 最も基本的なデータとなります. 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. このアトムとリスト等で構成されたデータを S 式と呼びます. 因に,LISP のプログラム自体も S 式です. その為,LISP の函数でプログラムを操作する事も容易に行えます.

その為,Maxima のプログラムでは足りない所を LISP で代用する事も多くあります. 又,Maxima から LISP を直接利用する事も可能です. この場合は 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.

```
(%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 to_lisp(); で LISP に入って, アトム$a に 1 を割当てています. それから (to-maxima) で Maxima に戻っています.to_lisp 函数の返却値は true です. 最後に a; を入力すると,LISP で$a に割当てた値 1 が返却されます. これは Maxima でのアトムの内部表現では$a の様に先頭に$が付くからです.

この例で注目して頂きたい事は,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事です. 即ち,Maxima で扱うデータ, 更には函数それ自体も LISP 側では別の表記方法があります. この LISP 側での表現を単純に内部表現と呼んでいます. この内部表現を通常の処理で気にする事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事になります.

尚,Maxima の函数名で先頭に?が付いているものが幾つか存在します. この様な函数は?を外した部分は LISP の函数で,Maxima から裏の LISP で処理させて, その結果を Maxima 側に持って来る函数です.?は通常の LISP の函数にも適応可能で?を頭に付けた函数は,Maxima 内部では?を外して,LISP の函数として処理されます. この様に Maxima が介在する為,?を用いて LISP の函数を利用する場合には, 引数は Maxima で見えているものを設定します.

尚,?と函数の間に空行を入れると 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.

?と似た函数に:lisp 函数があります. こちらは, 直接 LISP の S 式を Maxima 側から入力し,LISP に評価させた値を得る為の函数です.?演算子との違いは,?演算子では引数が LISP の函数で, その引数は Maxima に準じたものとなりますが,:lisp の場合はより一般的な LISP の S 式となります. その為, 引数も Maxima の内部表現そのものとなります.

```
(%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 x+y+z を割当てていますが,$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. この様な変数の参照は?では出来ません.

更に, :lisp (car $a); の値は内部表現そのものですが, ?car(a); の値はそれを Maxima で解釈した (”+”, simp) となっており, 引数に$が付いていない事と:lisp の結果に%o ラベルが無い事に注目して下さい.

この様に,?は入力と出力に Maxima が介在する為に, 入出力をする度に Maxima の評価を受ける事になりますが,:lisp の場合は直接操作となります. :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な函数です.

尚,Maxima は LISP で記述されている為, そのデータだけではなく, 全てが LISP の S 式で,LISP の函数で処理が可能となります. この事を利用して Maxima の機能を拡張する事が容易に行えます.

Maxima の函数を LISP から利用する場合,mfuncall 函数を用います. この場合, 引数は Maxima の函数名の頭に'$を付け, その後に引数を並べます.

```
MAXIMA> (mfuncall '$diff '$x '$x 1)
1
```

この例では, 変数 x を diff 函数を使って x で微分するものです. この様に, 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. 従って,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で用いる程度でしょう.

第2章　Maximaのデータとその操作

この章で解説する事:

- 数値
- 多項式
- 式について
- 代入操作
- 式の展開と簡易化
- リスト
- 配列
- 行列

2.1 数値

2.1.1 Maxima で扱える数値について

Maxima で扱える数値には, 整数, 有理数, 浮動小数点と複素数があります. その他に代数的整数もありますが, 代数的整数の場合はその最小多項式を用いる為, 多項式の節で説明します.

整数値の入力は C 等と同じ入力になります.C との違いはメモリ制約を除けば桁数に限界はありません. 有理数は 128/8989 の様に, 演算子/の両辺に整数を置いたもので表現されます. この有理数に関しても整数と同様にメモリの制約を除くと分母, 分子の桁数に上限はありません.

尚, 整数と有理数の演算を行うと, 基本的に有理数になります. 但し, 分母を消去出来る様な計算を整数や有理数で行うと, 結果は整数になります. 尚, 整数には実際は通常の整数の fixnum 型と可変桁の整数の bignum 型の二種類があります. 整数の場合, この二種類の違いを認識する必要は特にありません.

Maxima の浮動小数点には float 型と bigfloat 型の二種類があります.float 型は 16 桁の倍精度で `1.2` や `1.3e-4` の様に入力が出来ます.bigfloat 型は任意精度の実数で大域変数 fpprec で指定した桁数の実数です. この fpprec の値を大きくすると精度も当然高くなります. ここで計算精度がそれなりに必要でも, 結果の表示には少ししか必要がない場合, 大域変数 fpprintprec に大域変数 fpprec とは別の表示の桁数を設定すれば, 大域変数 fpprintprec に設定した桁数で bigfloat 型の数値が表示されます.float 型から bigfloat 型への変換は bigfloat 函数で行います.

浮動小数点と整数や有理数の演算は, 型の変更を実行しない限り浮動小数点となります. 又,bigfloat 型と float 型を混在して計算する場合はエラーになります. その為に変換函数による処理が必要になります.

複素数は実部と `%i` をかけた虚部との和で指定されます. 例えば, 方程式 $x^2 - 4x + 13 = 0$ の根は Maxima では `2+3*%i` と `2-3*%i` で表現されます. ここで複素数の実部は realpart 函数, 虚部は imagpart 函数で取り出せます. 複素数は整数, 有理数, 浮動小数点の自然な拡張となっている為, 実部と虚部は, これらの数で表現されています.

2.1.2 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 → bigfloat の際の警告の有無を指定
fpprec	16	正整数	bigfloat 型の桁数を指定
fpprintprec	0	正整数	bigfloat 型の表示桁数を指定
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 は函数等の動作に影響を与えます.domain はデフォルトで 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事を意味しています. これに対して値を complex

にすると,Maxima で処理する世界は複素数の世界となる事を意味します.

更に,domain を complex, 大域変数の m1pbranch を true にした場合,-1 の n 乗根は原始 n 乗根に自動的に変換されます.

大域変数 float2bf は,false の場合に bfloat 函数で浮動小数点数を bigfloat 型の数値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 は bigfloat 型数値の桁数を定めます. 即ち,fpprec を正整数 n に設定すると bigfloat 型は n 桁になります.

大域変数 radeexpand は式 $\sqrt{a^2b}$ の様に根号の外に出せる因子が式に含まれる場合,true であれば, その様な因子を根号の外に自動的に出させる大域変数です.

2.1.3 Maxima の定数

Maxima の定数

%e	自然底 e
%gamma	Euler の定数
%phi	$\frac{1+\sqrt{5}}{2}$
%pi	円周率 π
false	Bool 代数の定数. 偽 (LISP の nil)
true	Bool 代数の定数. 真 (LISP の t)
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

Maxima には幾つかの定数があり, 大雑把に 3 種類に分類出来ます.

最初の一つは%pi の様な通常の定数, もうひとつは,t や nil といった真偽値, 最後に Maxima が用いる inf の様な定数です. 更に,zeroa や zerob は limit 函数だけで用いる定数です.

```
(%i55) limit(1/x,x,zeroa);
(%o55)                                inf
(%i56) limit(1/x,x,zerob);
(%o56)                               minf
```

但し,limit(1/(x-1),x,1,'plus) の意味で limit(1/(x-1),x,1+'zeroa) の様な使い方は出来ません.

尚,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. この場合は,limit(⟨ 式 ⟩) で式の評価が行えます.

2.1.4　数値に関連する函数

数値に関連する述語函数

函数名	true となる条件
numberp	整数, 有理数, 浮動小数や可変長実数
bfloatp	bigfloat 型の数値
floatnump	浮動小数点数
integerp	整数
evenp	偶数
oddp	奇数
constantp	定数の場合

これらの述語函数は全て引数が一つです.

max と min

$\max(\langle$ 実数値$_1\rangle, \langle$ 実数値$_2\rangle, \cdots)$	$\langle$ 実数値$_1\rangle, \langle$ 実数値$_2\rangle, \cdots)$ の最大値
$\min(\langle$ 実数値$_1\rangle, \langle$ 実数値$_2\rangle, \cdots)$	$\langle$ 実数値$_1\rangle, \langle$ 実数値$_2\rangle, \cdots)$ の最小値

Maxima は与えられた実数列の最大値を max 函数, 最小値を min 函数で求めます.

数値に関連する函数

函数名	変換前		変換後
bfloat	全ての数	→	bigfloat 型
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
random	数値	→	0 から与えられた数値-1 迄の間の乱数
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
cargs	式	→	与式の偏角
sqrt	式	→	与式の平方根

最初の bfloat は全ての数と数値函数を bigfloat 型に変換します.

isqrt 函数は引数 ⟨ 整数 ⟩ の絶対値の平方根を越えない整数値を返します.

```
(%i50) isqrt(-3);
(%o50)                                1
(%i51) isqrt(-4);
(%o51)                                2
(%i52) isqrt(10);
(%o52)                                3
(%i53) isqrt(-10);
(%o53)                                3
```

尚,isqrt 函数は引数が整数で無ければ, 入力式をそのまま返却します.

fix 函数と entirer 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
```

尚, この例の様に, 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.

random 函数は, 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.

cabs 函数,realpart 函数と imagpart 函数は各々複素数の絶対値, 実部と虚部を返す函数です. 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 を含む式を虚部とします.

carg 函数は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.

sqrt は ⟨ 式 ⟩ が実数の場合はその平方根を返します. 尚,Maxima 内部では ⟨ 式 ⟩^ (1/2) で表現されています.

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

これらの数値函数は LISP の函数をそのまま利用します. その為, 先頭に?が付きます.

?round 函数は〈数値〉を最も近い整数に丸めます. ここで, 引数は float 型であって,bigfloat 型ではありません.

?truncate 函数は float 型の〈数値〉を引数とし, 小数点以下を切り捨てます.

2.2 多項式

この節では,Maxima の多項式の表現について述べます.

2.2.1 多項式の一般表現

Maxima の多項式は通常の C や FORTRAN で記述する x^2+3*x*z+4 や x**2+3*x*z+4 の様な書式で入力されます. ところが, 内部表現と表示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簡易化を行ない,Maxima の順序 $>_m$ 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が返されます.

以下に, 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 x+y+z を割当てています. :lisp $a; で変数 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 + である事を示し, その後に被演算子となる項が並んでいます. この様に 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. 尚, 項の並びは同じ式であれば項や変数の順番を変更しても同じ結果になります. この例で示す様に x+y+z でも z+x+y と入力しても同じです. これは Maxima の変数順序 $>_m$ 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 1.1 節, 順序 $>_m$ に関しては,1.3 節を参照して下さい. 多項式の内部表現では, 項に対して, その項を構成する変数を順序 $>_m$ に対して小さいもの順に並べ, 式を構成する項に対して, 順序 $>_m$ で小さな項で並べます. 最初の例の式 x+y+z の内部表現リストは (+ x y z) となります. これは, 項順序が z $>_m$ y $>_m$ x となる為で, 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, 単項式は x1^2 x3 x8^2 に並べ替えられています. これは,x8 $>_m$ x3 $>_m$ x1 となる為に Maxima 内部で変数を x1,x3,x8 の順に並べ替えた為です. 更に,x*4 の様な変数と数値の場合, 数値が $>_m$ の下位に来る為に,4*x になります.

以下に, 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$_1$ $\cdots$ 項$_m$)	項$_m$ $>_m \cdots >_m$ 項$_1$
単項式	((mtimes simp) 数値 変数$_1$の冪 $\cdots$ 変数$_n$の冪)	変数$_n$ $>_m \cdots >_m$ 変数$_1$

式の表示では, この内部表現を基に式の表示を行います. 多項式の場合, 大きな項から順番に表示されます. その為,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. その為,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.

では x+y,x-y,x*y,x/y,x^y についても :lisp 函数を使って, 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)
```

Maxima では x-y が x+(-y),x/y が x^(-y) に置換されている事が判ります. この様に 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ます. この様に変換しておく事で, 差の処理や商の処理の手間を減らす事が出来るのです.

この一般表現に対し,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.

2.2.2 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 $\langle$ 係数$_1\rangle\langle$ 変数 $\rangle^{\langle\text{次数}_1\rangle} + \langle$ 係数$_2\rangle\langle$ 変数 $\rangle^{\langle\text{次数}_2\rangle} + \cdots$ 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

($\langle$ 変数 $\rangle$ $\langle$ 次数$_1\rangle$ 係数$_1$ $\langle$ 次数$_2\rangle$ 係数$_2$ $\cdots$)

尚, この CRE 表現では, 次数に関し, $\langle$ 次数$_1\rangle > \cdots > \langle$ 次数$_2\rangle > \cdots$ 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, $3x^2+(-1)x^0$ で表現されます. これを λ 式風に考えれば, $\lambda\ x\cdot\left(3x^2+(-1)x^0\right)$ となりますね. 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たので, 変数の情報を落していても問題はありません. しかし, 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 x を入れてみましょう. すると (x (2 3) (0 -1)) となります. 更に, リスト中の小括弧を外して (x 2 3 0 -1) としても $3x^2+(-1)x^0$ の復元には問題ありません.

多変数多項式の場合も同様に CRE 表現で書換える事が出来ます. 但し,1 変数の場合の様に平坦なリストで 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事は変数の間に順序を入れる事です.Maxima の場合, 辞書式順序を基礎にした順序 $>_m$ が入っています. この順序の詳細は 1.3 節を参照して下さい.

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,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ます. 途中で Maxima の変数順序 $>_m$ で大小関係がつくと, その順序で変数に順番が入ります. 例えば,xxz と xxy の場合は頭の二つが x なので順序がまだ決まりませんが, 最後の z と y に関しては,z $>_m$ y となるので,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. 先ず, 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. 但し, 第二成分の $2y+1$ 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. 結局, (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, 中間的には (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.

この様に多変数の場合,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は, その主変数の多項式と看做し, それ以外の変数に対しては 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 表現が与えられた時に, それらの変数順序が異なれば, 処理は非常に厄介な事になるのが判るかと思います. その為,CRE 表現を用いる函数に関しては, 後の事も考えて主変数の設定を行う必要があります.

又,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. その為, y ^ 2 の様な式は変数に使えませんが, log(x) や cos(x + 1) の様な函数は使えます.

Maxima の変数順序は基本的に辞書式順序です. 尚, 局所的な変数の順序の変更が ratvars 函数等で指定出来ます.

2.2.3 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$_1$⟩, $\cdots$, ⟨ 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

先ず, 最初の rat 函数は与えられた ⟨ 式 ⟩ を展開し, 浮動小数点を 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 $+,-,*,/$ と冪乗の他の函数を一般的には簡易化しません.CRE 表現でのアトムは一般形式のものと異なります. 従って,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 函数には ratfac,ratprint,keepfloat といった直接動作に関連する大域変数があります.
ratdisrep 函数は CRE 表現から一般表現に引数を変換します.
totaldisrep 函数は CRE 表現から一般表現に ⟨ 式 ⟩ の全ての部分式を変換します.

2.2.4 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無く, 分母の筆頭項 (leading term) の係数を正にしたものとなります.
以下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為, 変数の順序は z $>_m$ y $>_m$ x になります. その為,z が主変数となり, 式は 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,y $>_m$ x となる為に, -(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 の係数を正にする為に, 必要に応じて-1 がかけられています. この例で示す様に, 式が CRE 表現, 或いは 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 表現である事を示し, 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 である事を示しています. ここで内部変数は LISP の gensym 函数で生成されるもので, 後ろの番号は生成順に付けられるものです. 尚, 有理式の変数は Maxima の showvar 函数を使って取り出す事が可能ですが, それらの変数に対応する内部変数は Maxima の内部変数の genvar に登録され, 参照する場合には :lisp genvar の様にします.

又, 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為, 左端が最小で右端が主変数となります. その為, この式では Z が主変数となります. この様な変数リストを含むリスト (MAT cdots) の後に分子となる式が来ます. この式は先頭が Y13181 の為, 変数 Y の多項式で, 後の 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 である事が判ります. 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 の多項式である事が判ります. 以降, 分子の時と同じ様に読めば良いのです. 但し, 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 の順なので, 変数 Z が式中にあれば, Z の多項式として表現し,Z がなくて変数 Y があれば 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事が判ります.

尚,CRE 表現で分母が整数の場合 (CRE 表現では, 浮動小数点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)
```

この例で示す様に分子は 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, 分母が整数の時に限って記号. 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事を示しています. また変数 X の後に数値が入っていますが, これは LISP 内部の処理で変数 X に対応するシンボル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 級数を表現する為に用いられています. 有理式の表記は正の整数では無く, 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, 多項式と云うよりは, 上で記述した様に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. 尚, 切り捨てられる次数の様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 展開の様々な情報が追加されている事が判ります.

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 が true であれば, 浮動小数点を含む式が CRE 表現に変換される際に, 浮動小数点が有理数に近似変換される事を防ぎます. 尚, 浮動小数点が有理数に近似される際に生じる誤差は ratepsilon で制御されます.

大域変数 ratepsilon は式を CRE 表現に変換する際に, 係数を有理数に変換する時に用いられる許容範囲となります. 大域変数 ratepsilon よりも小さな浮動小数点は無視されます. 浮動小数点を有理数に変換したくなければ, 大域変数 keepfloat を true に設定します.

```
(%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
```

この例で示す様に,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, 浮動小数点が有理数に変換されている事が判ります。

大域変数 ratalgdenom は式中に代数的整数を項の分母として持つ式に対し, 大域変数 ratalgdenom が true の場合に, その分母を有理化します. これを実行する為には, 大域変数 algebraic を true に設定して, 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)
```

この例で示す様に, 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事に注意して下さい.

大域変数 ratprint が false であれば, 浮動小数点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 表現の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

大域変数 ratdenomdivide が 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事を抑制します.

大域変数 ratexpand が 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時に CRE 式が展開されます.

大域変数 ratfac が 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,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.

例えば,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.

大域変数 ratfac と大域変数 ratweights の手法は互換性が無いので, 両者を同時に使っていけません.

大域変数 ratsimpexpons が 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 は 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. 尚, デフォルト値が false であれば, 切捨ては生じません.

大域変数 ratweights は ratweight 函数で設定される指定された重みのリストです. 大域変数 ratweights や ratweight() 函数でそのリストが見られます.

大域変数 rootsconmode は rootscontract 函数の挙動を定めます. 大域変数 rootsconmode が false 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, 次数

の分母が割切れる冪だけを纏めます all の場合, 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
$x\hat{}(1/2)*y\hat{}(3/2)$	false	$(x*y\hat{}3)\hat{}(1/2)$
$x\hat{}(1/2)*y\hat{}(1/4)$	false	$x\hat{}(1/2)*y\hat{}(1/4)$
$x\hat{}(1/2)*y\hat{}(1/4)$	true	$(x*y\hat{}(1/2))\hat{}(1/2)$
$x\hat{}(1/2)*y\hat{}(1/3)$	true	$x\hat{}(1/2)*y\hat{}(1/3)$
$x\hat{}(1/2)*y\hat{}(1/4)$	all	$(x\hat{}2*y)\hat{}(1/4)$
$x\hat{}(1/2)*y\hat{}(1/3)$	all	$(x\hat{}3*y\hat{}2)\hat{}(1/6)$

2.2.5　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 $\mathbb{Q}[\%i]/\langle \%i^2+1\rangle$ がデフォルトです.

その他の係数体を Maxima では扱う事が可能です. 先ず,p を正素数とする場合に, 多項式の係数体を $\mathbb{Z}_p[\%i]/\langle \%i^2+1\rangle$ とする事も可能です.

更に,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の事です. 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. 但し,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為には,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, 純虚数 i や $\sqrt{2}$ 等が挙げられます. これらの最小多項式は各々 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 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. 即ち,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. 但し,mod 函数の第二引数を与えない場合, 大域変数 modulus の値が用いられる為です. 又, 大域変数 modulus に正素数以外の値を設定する事も可能です.

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. 例えば, p として 5 を採った場合, 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, $3-(-2)=4-(-1)=5(mod\ 5)$ となるので, 絶対値では $\{0,1,2\}$ だけを考えれば良い事になります. この事を利用すれば, 計算をより簡単に行う事が可能になります.

次の大域変数 algebraic は Maxima 上で代数的整数の簡易化を行う場合, 必ず true にしなければなりません. 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為, 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としては, 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事で, 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. 例えば,factor 函数の場合, 最小多項式が x^2-2 となる代数的整数 a を用いて式を分解する場合, 最小多項式に, その代数的整数を代入した形で 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)
```

この例では, 多項式 x^2-4 を環 $\mathbb{Q}[\sqrt{2}][x]$ 上で因子分解を行っています.

2.2.6 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 が true となっていれば, 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.

$$tellrat(\text{式}) : Maxima \text{の係数環} \to Maxima \text{の係数環}/\langle \text{式} \rangle$$

但し,tellrat 属性は CRE 表現を扱う ratexpand 函数や ratsimp 函数で処理を行う時のみに反映されます.

tellrat 函数

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. 又, 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. この時, 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. 従って,$a^2=c^3-2$ を tellrat 函数に与えた場合, 主変数が a となり, この a が Maxima に追加される代数的整数となり, その最小多項式が a^2-c^3+2 で与えられます. ここで a^2-c^3+2 を引数として与えると, 主変数は Maxima の項順序 $>_m$ から c となる為に代数的整数は c となる事に注意して下さい.

この tellrat 函数は任意個の因子を取る事が可能です.

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. 先ず, 代数的整数の簡易化を行う為に大域変数 algebraic を true に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) と入力した場合を示します.

```
(%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
```

この例で示す様に,tellrat で設定した属性は大域変数 algebraic 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時に反映されます.

尚, この例での (a+2)^4,algebraic,expand,ratsimp といった表記は,ev 函数の表記方法の一つで, ev((a+2)^4,algebraic,expand,ratsimp) と同値です.

次に, tellrat(a^2-2,b^2=c^4); で代数的整数 b を追加した為, ratexpand((b+c)^3) で,b^2 が c^4 で置換されている事に注意して下さい. この様に, 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 $>_m$ の影響を受ける為, 等式の右辺に代数的整数の項を置きます. 例えば,a=a^2+c^3 や a^2=c^3-a の様にします.

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試みない様に注意しなければなりません.

例えば,tellrat(w^3-1);algebraic:true;rat(1/(w^2-w)) は零による割算になります. このエラーは ratalgdenom:false に設定する事で回避出来ます.

tellrat 函数で設定した属性は tellrat() で見る事が出来ます. この場合, 引数は特に設定する必要がありません.

untellrat 函数

untellrat(⟨x⟩)

tellrat 函数で設定した属性は untellrat 函数を使えば削除出来ます. この場合,untellrat で代数的整数を直接指定します.

```
(%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)
```

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 が $x^2+1=0$ と $y^2+1=0$ を満す代数的整数と設定しています. この様に複数の変数に対して整係数多項式を tellrat に入力する事も可能で, ev でも的確に評価されています.

次に untellrat(y) で y に関してのみ,tellrat で設定した性質 (y は $y^2+1=0$ を満す代数的整数) である事を除去しています. その後には,x に関する性質だけで評価が行われています.

尚,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) なものでなければならなりません. 又, 多変数の場合,untellrat 函数は主変数 (maimvar) に対して行います.

```
(%i15) tellrat(x+y+z*y1);
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)                                []
```

この例で示す様に,x+y+z*y+1 に関しては主変数が z で係数が y となる為にエラーになります. 但し,2*x+y+z+1 の様に主変数 z が monic でありさえすれば問題はありません.untellrat 函数が主変数のみに使える事も上の例から分ります.

2.2.7 Horner 表記

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, 変数 X の次数の高い順番に項を並べます. 例えば, 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, $X\left((\cdots(a_nX + a_{n-1}) + \cdots) + a_1\right) + a_0$ の様に帰納的に変数 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事が出来るので, 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 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には, 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,Maxima の変数順序 $>_m$ に従って,$>_m$ で最大の変数を主変数とし, その変数に対して与式に 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
```

尚, 主変数として函数を指定する事も可能です.

```
(%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)))))
```

2.2.8　多項式に関する函数

多項式に関連する述語函数

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 函数は ⟨ 式 ⟩ が ⟨ 有理数式 ⟩ であれば true, それ以外は false を返します.

ratp 函数は ⟨ 式 ⟩ が CRE 表現, 或いは拡張 CRE 表現であれば true で, それ以外は false を返します.

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 函数は,⟨ 式 ⟩ に含まれる項 ⟨ 変数 ⟩$^{⟨ 次数 ⟩}$ 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 ⟩ はアトムか真の部分式です. 具体的には,x,$sin(x)$,$a[i+1]$,$x+y$ 等です.

尚, 真の部分式の場合,$(x+y)$ 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 ⟩$^{⟨ 次数 ⟩}$ の項を正確に求める為には, 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 は ⟨ 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. 尚, 返却値には,⟨ 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です.

この様な係数が存在しない場合は零を返します.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.

例えば,ratcoeff((x+1)/y+x,x) は (y+1)/y を返却しますが,coeff 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. その為,⟨ 式$_2$⟩ が負の冪の項に含まれれば,ratcoef は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, 係数は予期した様に現れないかもしれません.

bothcoef 函数は二成分のリストを返し, このリストの第一成分が ⟨ 式 ⟩ 中の ⟨ 変数 ⟩ の係数 (式が CRE 表現であれば ratcoef, それ以外は coeff 函数で見つけたもの) となります. 第二成分が ⟨ 式 ⟩ の残りとなります. 即ち,[a,b] が返却値であれば,⟨ 式 ⟩ $= a *$ ⟨ 変数 ⟩ $+ b$ となります.

項数や次数を返す函数

nterms(⟨ 式 ⟩)
powers(⟨ 式 ⟩ ,⟨ 変数 ⟩)

nterms 函数は ⟨ 式 ⟩ を展開した時の項数を返却します. 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
```

この例で示す様に,sin 函数の引数がどの様な式であっても, 一つで数えられている事に注意して下さい.

次の powers 函数は ⟨ 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.

多項式の項の次数を返す函数

hipow(⟨ 多項式 ⟩,⟨ 変数 ⟩)
lopow(⟨ 多項式 ⟩,⟨ 変数 ⟩)

hipow 函数は ⟨ 多項式 ⟩ に含まれる ⟨ 変数 ⟩ の項の最高次数を返します. 尚,hipow では式の展開を自分で実行しない為, 予め式を展開しておく必要があります. 以下の例では $(x+1)^4$ を展開せずに hipow を用いた結果と expand で展開した式に対して hipow を用い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 函数は ⟨ 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 ⟩)

ratvars 函数は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 函数を実行した後, リスト中の一番右側の ⟨ 変数$_n$⟩ が有理式にあれば, その変数を有理式の主変数になります. 又, 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$_1$⟩ よりも低い順序が付けられます.

ratvars 函数の引数は変数, 或いは $sin(x)$ の様な非有理的函数の何れかでなければなりません. 尚, 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
```

ratweight 函数は,⟨ 重み$_i$⟩ を ⟨ 変数$_i$⟩ に割当てます. その重みが大域変数 ratwtlvl の値に従って項が 0 で置き換えられます. 項の重みは項の中の変数の冪を重みに掛けた積の和となります. 例えば, $3 \cdot v_1^2 \cdot v_2$ の重みは $2 \cdot w_1 + w_2$ となります. この切捨ては, 式の CRE 表現の積や冪乗を実行する時のみに生じます.

尚,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 函数は ⟨ 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]
```

多項式を纏める函数

factcomb(⟨ 式 ⟩)
fasttimes(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩n)
rootscontract(⟨ 式 ⟩)

factcomb 函数は ⟨ 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. 即ち, $(n+1)*n!$ を $(n+1)!$ にする事です. ここで, 大域変数 sumsplitfact が false に設定されていれば,minfactorial が factcomb の後に適用されます.

fasttimes 函数は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, 両者共に殆ど同じ大きされ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 $n*m$ のオーダーで計算を行いますが,fasttimes 函数を用いると $\max(n,m)^{1.585}$ のオーダーになります.

rootscontract 函数は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. 例えば,rootsmode が true の場合,x^(1/2)*y^(3/2) の様な根同士の積を sqrt(x*y^3) に纏めます.

大域変数 radexpand が true で大域変数 domain が real であれば,rootscontract 函数は abs を sqrt に変換します. 即ち,abs(x)*sqrt(y) を sqrt(x^2*y) に変換します.

rootscontract 函数は logcontract 函数と似た手法で ratsimp 函数を用います.

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 ⟩)

combine 函数は ⟨ 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.

denom 函数は ⟨ 有理式 ⟩ の分母 (DENOMinator) を返します. 尚, 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 函数は有理式の分子 (NUMerator) を返します.

ratdenom 函数は ⟨ 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 函数は ⟨ 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 函数は ⟨ 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. 尚,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.

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$_k$⟩])

bezout 函数は, ⟨ 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. 従って,bezout 函数と determinant 函数と組合せれば resultant 函数の代替になります.

resultant 函数は二つの多項式 ⟨ 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.

ここで, 多項式 f と g の解を各々α_i,β_j とすると, 多項式 f と g の終結式 $res(f,g)$ は以下の式と等しくなる事が知られています.

$$res(f,g) = a_m^n b_n^m \prod_{1\le i\le m, 1\le j\le n} (\alpha_i - \beta_j)$$

その為, 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多項式となる事が判ります.

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出来ます. 行列の大きさは, ⟨ 多項式$_1$⟩ と ⟨ 多項式$_p$⟩ の次数を各々 m,n とすると, $m+n$ 次の正方行列となりますが,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は, 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.

$$f = \sum_{i=0}^{m} a_i x^i$$

$$g = \sum_{i=0}^{n} b_i x^i$$

すると,

$$resultant(f,g,x) = \det \begin{pmatrix} a_m & a_{m-1} & \cdots & \cdots & a_1 & a_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & a_m & a_{m-1} & \cdots & \cdots & a_1 & a_0 \\ b_n & b_{n-1} & \cdots & \cdots & b_1 & b_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & b_n & b_{n-1} & \cdots & \cdots & b_1 & b_0 \end{pmatrix}$$

で終結式は計算出来ます. 尚,⟨ 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 函数は与えられた方程式, 或いは零と等しいと仮定した式から続けて終結式を取る事で, 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$_1$⟩, $\cdots$, ⟨ 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為に solve 函数を呼出します.

```
(%i1) exp1:2*x^2+y*x+z;
                                        2
(%o1)                          z + x y + 2 x
(%i2) exp2:3*x+5*y-z-1;
(%o2)                        - z + 5 y + 3 x - 1
(%i3) exp3:z^2+x-y^2+5;
                                 2    2
(%o3)                           z  - y  + x + 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]
```

終結式のアルゴリス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.

デフォルト	subres
モジュラー終結式アルゴリズム	mod
縮約 prs	red

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.

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 函数は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, 例えば,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 (デフォルト値)
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, 例えば,gcd(x^2-2*sqrt(2)*x+2,x-sqrt(2)) の GCD を計算する為には, 大域変数の algebraic が true であり, 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,gcd:subres を用いる事を推奨します.

gcdex 函数は 3 個の多項式を成分とするリスト [a,b,c] を返します. 多項式 c が引数の ⟨ 多項式$_1$⟩ と ⟨ 多項式$_2$⟩ の最大公約因子で,a,b は共に c = a ∗ ⟨ 多項式$_1$⟩ + b ∗ ⟨ 多項式$_2$⟩ を満す多項式となります. この函数が用いているアルゴリズムは Euclid の互除法に基づくものです.

尚, 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, 多項式を ⟨ 変数 ⟩ で指定した単変数の多項式と看做して GCD を計算します.

何故, 多変数多項式が相手の場合に, 変数の指定を行う必要があるかと言えば, 多変数多項式の場合は, 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となります. 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, 即ち, 多項式環 $k[x]$ であれば, 任意のイデアルは単項イデアル, 即ち, 一つの多項式だけで生成されるので, この場合は最大公約因子が必ず存在します.

その為, 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, 最後の多変数の例で,gcdex(x*(y+1),y^2-1,x) の結果で,GCD として 1 を返している事に注意して下さい. 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 にはなりません. 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 になる事が知られています. その為,Eculid の互除法が利用可能になるので,gcdex は必ず結果を返します.

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 $xy+x$ になります. 但し, 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. 尚,Gauss 整数とは, 複素数 $a+bi$ で,a と b が整数になります. 因子は a と b を非負とする事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 函数は Gauss 整数上で ⟨ 多項式 ⟩ の因子分解を行ないます. 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 $a(=i)$ を用いて x^4-1 を因子分解しています.

gfactorsum 函数は factorsum に似ていますが,factor 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 函数は, 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 函数は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]
```

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⟩)

factor 函数は ⟨ 式 ⟩ を整数上で既約因子に分解します. factor(⟨ 式 ⟩, ⟨p⟩) の場合, 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います. 尚,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 函数はグループ単位で ⟨ 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為に復元出来ません.

sqfr 函数は factor 函数と似ていますが, 多項式因子は無平方 (square-free) となります. 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 を割切る様なものが存在しない場合です. 特に 1 変数多項式の場合,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 は無平方であり, $i \neq j$ であれば $\gcd(a_i, a_j)$=1 を満すものです.Maxima の sqfr 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)
```

この例で示す様に,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事です.

この様に, 多項式を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. 更に, 有理多項式の積分計算では, 無平方な因子に分母を分解して, 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ます. この無平方分解は幅広く利用されています.

factorout 函数は ⟨ 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 は因子分解されたものになります.

nthroot 函数は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, その様な多項式が存在しなければ, エラーメッセージが表示されます.

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
```

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

大域変数 dontfactor に factor 函数で因子分解しない式を登録します. 尚,ratvars 函数で入れた変数の順序に対し, 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 は factor 函数で返された既約因子が展開された形式 (デフォルト) 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 が 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 が 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, それ以外ではデフォルトの Berlekamp アルゴリズムが使われます.

大域変数 intfaclim は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, 即ち, 利用者が factor 函数を明示的に呼び出す場合や, 整数が fixnum, つまり,1 機械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.intfaclim の設定は factor の内部呼び出しで用いられます.

大域変数 intfaclim は大きな整数の因数分解に長時間を費すのを防ぐ為に再設定しても構いません.

大域変数 newfac が true であれば,factor は新しい因子分解ルーチンを用います.

大域変数 savefactors が true であれば, 幾つかの同じ因子を含む後の式の展開の処理速度向上の為, 式の各因子がある函数で保存されます.

剰余を計算する函数

divide(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩)
quotient(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数$_1$⟩, $\cdots$)
remainder(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数$_1$⟩, $\cdots$)
mod(⟨ 多項式 ⟩)
mod(⟨ 多項式 ⟩, ⟨ 整数 ⟩)

divide 函数は ⟨ 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 函数は ⟨ 多項式$_2$⟩ による ⟨ 多項式$_1$⟩ の割り算の商を計算します.

remainder 函数は ⟨ 多項式$_2$⟩ による ⟨ 多項式$_1$⟩ の剰余を計算します.

mod 函数は ⟨ 多項式 ⟩ を大域変数 modulus で指定した値に対する剰余を計算します.

尚, 大域変数 modulus はデフォルトでは false の為,mod(⟨ 多項式 ⟩) を実行するとエラーになります.

大域変数 modulus の値と無関係に, 多項式の剰余を計算したければ, mod(⟨ 多項式 ⟩,⟨ 整数 ⟩) で ⟨ 多項式 ⟩ の ⟨ 整数 ⟩ による剰余を計算します. 尚, 大域変数 modulus の値は変更されません.

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$_n$⟩)

ratexpand 函数は和の積や指数の和を掛け, 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. これは ⟨ 式 ⟩ を CRE 表現に変換し, それからを一般形式に戻しています.

その為,ratexpand に影響を与える大域変数には ratexpand,ratdenomdivide や keepfloat といった CRE 表現への変換に影響を与える大域変数があります.

fullratsimp 函数は ⟨ 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 を呼出します. 時には, その様な呼出しが一回以上必要であるかもしれません. fullratsimp 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迄適用します.

例えば, 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,fullratsimp(exp) を実行すれば x^a-1 が得られます.

ratsimp 函数は, 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 の様に簡易化します. 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. 即ち, 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 の様に非有理的函数 (例えば,sinx^2+1 を含みますが,ratsimp 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 は ratexpand に影響を与える幾つかの大域変数の影響を受ける事に注意して下さい.

尚,*ratsimp*(⟨ 式 ⟩,⟨ 変数$_1$⟩,$\cdots$,⟨ 変数$_n$⟩) で,ratvars に変数の ⟨ 変数$_1$⟩,$\cdots$ 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.

2.3　式について

Maxima では殆どのものが式 (Expression) になります. 例えば,128, $1+2+3$ や $\sin(x)$ の様に, 数値, 変数といった複数の Maxima のアトムを演算子や函数で結び付けたものです. 但し, 式は文脈で設定された各種の仮定や大域変数の指定に従って解釈されます. その為,1+2+3 を入力すると 6 が返されますが,a:sin(x) の場合は x にどの様な値が割当てられているか, 或いは関連する大域変数の設定によって結果が異なります.

複数の式をコンマで区切って並べたものの全体を小括弧 () で括ったものが式の列となります. 尚,Maxima 言語で生成した函数も実体は式の列です. その為, 利用者定義函数もリスト処理函数による処理の対象となります. これは Maxima が動作する LISP 側でも Maxima 言語による函数を操作出来る事を意味しています. その為, 少しでも効率の良いプログラムを記述したければ, LISP 函数を記述する事になります.

Maxima は LISP で記述されています. その為,Maxima のデータや式等は LISP 内部では別の表現を持っています. ここでは簡単に Maxima の内部表現を示しましょう.

2.3.1　変数や文字列の内部表現

Maxima の内部表現：変数

Maxima	内部表現
a	$ a
?a	a
”a”	& a
‘a	((mquote) $a)

変数は内部で先頭に$を付けたものです. 逆に,Maxima で?が先頭に付くものは内部では?を外したものになります. その為,LISP の函数を Maxima から実行させる場合, 先頭に?を付けてやれば使える事になります. 但し, 引数等は Maxima を通して与えられる為,Maxima の函数の様に引数を与える必要があります.

2.3.2 二項演算の内部表現

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	((mquotient simp) $x $y)
x .y	((mnctimes simp) $x $y)
x^ 2 又は x**2	((mexpt simp) $x 2)
x^ ^ 2	((mncexpt simp) $x 2)

基本的に $\boxed{\text{変数}_1\ \text{演算}\ \cdots\ \text{変数}_n}$ は内部では $\boxed{((\text{演算名})\ \text{変数}_1\ \cdots\ \text{変数}_n)}$ で表現されています. 更に演算名は全て m から開始している事に注意して下さい.m を先頭に付けて Maxima の函数と LISP の函数を区別しているのです.

更に, 差 x-y は実質的に x+(-y) で内部では表現されている事に注意して下さい.

ここで x-y の表現を調べてみましょう. 使う函数は LISP の基本の基本,car と cdr 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,:lisp 函数を使って直接 LISP の函数を入れる方法があります. 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) になっている事が分るかと思います.

次に, 論理演算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

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)

2.3.3 割当の内部表現

次に,Maxima で行う割当にも同様の内部表現があります.

Maxima の内部表現:割当

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,a:b は内部では ((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) となり, LISP の (setq a b) に対応する事が分りますね.

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事が判ります.

2.3.4 Maxima の函数の内部表現

MaximaX 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)

函数名には基本的に$が先頭に付きますが, 単引用符' が先頭に付いた場合は%で置換えられます. この%が先頭に付いた函数は Maxima で自動的に解釈される事はありません.

微分を行う diff 函数の例に示す様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事が分りますね.

```
(%i93) ?car('diff(x,y));
(%o93)                          (derivative, simp)
(%i94) ?cdr('diff(x,y));
(%o94)                               (x, y, 1)
```

2.3.5 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)

Maxima 内部の配列は, 多少勝手が違います. 内部表現の第一成分は最初に$が付けられた配列名があり, その後に毎度の simp と最後の array でデータが配列である事を示しています.

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$が先頭に付加されています.

2.3.6 Maxima の制御文の内部表現

Maxima の内部表現:制御文

if a then b	((mcond simp) $a $b t nil)
if a then b else c	((mcond simp) $a $b t $c)
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([l1,l2],s1,s2)	((mprog simp) ((mlist simp) $l1 $l2) $s1 $s2)
block(s1,s2)	((mprog simp) $s1 $s2)

この表に示す様に,Maxima の制御文も内部表現では LISP の S 式となっています. Maxima は結局 S 式で全てが記述されており, その S 式を裏の LISP が解釈して結果を表に出しているのが実状です.

以上からも, この内部表現を利用して新規に Maxima の制御文を LISP で構築する事が可能な事が分るかと思います.

2.3.7　表示式と内部表現

入力式とその式の内部表現は, 入力式の演算子が全て内部表現では前置式に変換されています. それ以外にも細かな書式の違いがあります.

この内部表現への変換で変更される式を具体的に示しておきましょう.

式の表現の違い

入力	part	inpart
$a-b;$	$a-b$	$a+(-1)b$
$a/b;$	$\frac{a}{b}$	$a\,b^{-1}$
$sqrt(x);$	$sqrt(x)$	$x^{1/2}$
$x*4/3;$	$\frac{4x}{3}$	$\frac{4}{3}x$
$exp(x)$	$\%e^x$	$\%e^x$

先ず, 入力列が Maxima に入力する式で,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. その為, これらの式に対して, 後述の part 函数と inpart 函数では結果が異なる事になります.

尚,Maxima には内部表現を part 列で示す Maxima で表示される式に適合する様に式を変換する函数として dispform 函数があります.

dispform 函数

dispform(⟨ 式 ⟩)
dispform(⟨ 式 ⟩ ,all)

dispform 函数は ⟨ 式 ⟩ の内部表現から構築される式から式に含まれる演算から構成される式の前置式表現 (外部表現と呼んでいます) に変換します. 具体的には Maxima は入力された式を内部表現に変換する際に,x-y を x+(-y), x/y を x*y^(-1) 等に変換します.dispform は内部表現が表現する式に含まれる全ての演算子から,Maxima に用意された演算子を用いて, 式に一対一に対応する前置式表現に作り直します.

```
(%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)
```

この様に, 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 表現もこの様な外部表現に変換します.

但し,dispform(⟨ 式 ⟩) では, 式に含まれた函数の引数は内部形式のままです. この様に, 一切の例外を認めずに式全てを外部表現に変換する為, all オプションを用いて,dispform(⟨ 式 ⟩ ,all) で変換を行います.

```
(%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 を作用させたもの,dispform に all オプション付きで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 オプションを付けたもの以外は全て同じで,all オプションを付けたものは, 式全体が外部表現になっています.

2.3.8　変数

Maxima で利用可能な変数名は, 基本的にアルファベットの大文字や小文字, 幾つかの記号と alphabetic 属性を与えた文字です. 0 から 9 の数値の様に変数の先頭以外ならば利用可能な文字もあります. この変数に関しては, 逆アルファベット順を基礎とした変数順序 $>_m$ が組込まれています. 詳細は,1.3 節を参照して下さい.

Maxima には式に指定した変数が含まれているかどうかを判別する述語函数 freeof があります.

変数の述語函数

freeof(⟨ 変数$_1$⟩ ,$\cdots$,⟨ 変数$_n$⟩ , ⟨ 式 ⟩)	⟨ 変数$_i$⟩ が ⟨ 式 ⟩ に存在しない場合
lfreeof([⟨ 変数$_1$⟩ ,$\cdots$,⟨ 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$_i$⟩ は変数アトム, 添字付けられた名前, 函数, 或いは二重引用符””で括られた演算子が扱えます.

尚,freeof 函数は,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する事は出来ません. 疑似変数を ⟨ 変数$_i$⟩ に指定した場合, 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. 但し, 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.

次に,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.

Maxima に含まれる変数一覧

大域変数名	初期値	概要
values	[]	利用者設定のアトム変数の一覧

この大域変数 values には,Maxima の大域変数を除いて, 演算子:, 演算子::や函数的な束縛のある変数名を含むリストが割当てられます.

```
(%i1) a:1;
(%o1)                                1
(%i2) b::sin(a);
(%o2)                             sin(1)
(%i3) x;
(%o3)                                x
(%i4) values;
(%o4)                             [a, b]
```

この例では a と b には値を割当てていますが, 単純に x は入力しただけです. すると, 大域変数 values には値が束縛された変数 a,b が登録されています.

大域変数 values に登録された変数は, 次の remvalue 函数を用いて Maxima から削除する事が出来ます.

変数値の削除を行う函数

remvalue(⟨ 変数$_1$⟩ ,⟨ 変数$_2$⟩ ,···)
remvalue (all)

remvalue 函数は指定した大域変数 values の変数リストに登録された変数を Maxima から削除します. 変数は添字されていても構いません.

remvalue(all) で, 全ての利用者変数が削除されます.

```
(%i10) b1:sin(y);
(%o10)                                sin(y)
(%i11) x:1;
(%o11)                                  1
(%i12) c1:b1*x;
(%o12)                                sin(y)
(%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 all 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為, 変数 x を削除しても問題はありません.

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 函数は ⟨ 式 ⟩ が函数の場合, 函数の引数を返します. 但し, 多項式の CRE 表現から変数を取出す showratvars 函数とは異なり, 内部表現の先頭の成分を Maxima のリストに変換したもの, 具体的には, substpart("[",⟨ 式 ⟩,0) と同じ結果となります. その為, 複雑な式になると部分式のリストが返される事になります.

```
(%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)]
```

尚,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 函数は〈式〉の変数リストを生成します. ここで, 大域変数 listconstvars が true であれば,〈式〉に Maxima の数学定数%e,%pi,%i や定数として宣言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. 但し, デフォルトの false の場合, 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	初期値	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 が true であれば, 定数として宣言した変数と Maxima の数学定数%e,%pi,%i が式に含まれていると,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, 数学定数と定数として宣言された変数は除外され,listvars が返すリストには含まれません.

```
(%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 が 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. 尚, 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.

2.3.9 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. 先ず,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 函数は ⟨ 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. 尚,isolate 函数は式の展開を行いません. 単純に, 与えられた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.

尚, 大域変数 isolate_wrt_times が false の場合,⟨ 式 ⟩ は ⟨ 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 が true の場合,isolate は更に ⟨ 式 ⟩ を分解し, 項も ⟨ 変数 ⟩ の冪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;
(%o12)                                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;
(%o15)                                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 函数は isolate(⟨ 式 ⟩ ,⟨ 変数 ⟩) 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事が出来ます

isolate 函数に影響を与える大域変数

変数名	初期値	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 が true であれば, isolate(⟨ 式 ⟩ ,⟨ 変数 ⟩) の実行で ⟨ 変数 ⟩ に含まれる (%e の様な) アトムの指数項に対しても調べます.

大域変数 isolate_wrt_times が 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, 更に式を分解し, 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.

```
(%i17) eq1:expand((a+b+x)^2);
                     2                   2           2
(%o17)              x  + 2 b x + 2 a x + b  + 2 a b + a
(%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
(%i20) isolate_wrt_times;
(%o20)                              false
(%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;
(%o22)                              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 函数は与えられた ⟨ 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, 第一成分が ⟨ 変数 ⟩ を含まない部分式, 第二成分が ⟨ 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 函数は与式を a+b*%i の形式で返します.⟨ 式 ⟩ が複素数の場合,a と b は実数になりますが, そうでない場合,%i を持たない部分と%i で括られた式に分解し, 内部の項順序に従って出力される為, 正確に a+b*%i の形式にはなりません.

```
(%i12) rectform((x+%i)^3);
                             3          2
(%o12)                      x  + %i (3 x  - 1) - 3 x
(%i13) rectform((10+%i)^3);
(%o13)                             299 %i + 970
```

2.3.10　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表にあり, それを内部で表現する内部表現の二つがあります. 式の木構造では, 演算子や函数の情報は比較的分かり易いものですが, 内部表現は式の前置式表現を基にしたものの為, 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, 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.

先ず, 部分式を取出す函数には, 内部表現を利用する inpart 函数, 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$_k$⟩])
pickapart(⟨ 式 ⟩ ,⟨ 整数 ⟩)

inpart 函数は ⟨ 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て, 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
```

尚,[⟨ 整数$_1$⟩, ··· , ⟨ 整数$_n$⟩] の様にリストで指定する事も可能ですが, この場合は, 第一層の ⟨ 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]);
                                   x z
(%o76)
(%i77) part(expr2,[1,2]);
                                     2
(%o77)                           x sin(x  + 1)
(%i78) inpart(expr,0);
                                    +
(%o78)
(%i79) inpart(expr2,0);
                                    *
(%o79)
```

この例で示す様に, 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. その為, 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.

part 函数に関連する大域変数

変数名	初期値	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 には 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 が true に設定していると, 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.

pickapart 函数は ⟨ 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事にも使えます.

```
(%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
```

2.3.11 総和と積

∑ と ∏

product(⟨ 式 ⟩ ,⟨ 添字変数 ⟩ , ⟨ 下限 ⟩ ,⟨ 上限 ⟩)
sum(⟨ 式 ⟩ ,⟨ 添字変数 ⟩ , ⟨ 下限 ⟩ ,⟨ 上限 ⟩)

product 函数は ⟨ 添字変数 ⟩ の ⟨ 下限 ⟩ から ⟨ 上限 ⟩ 迄の ⟨ 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,product 函数はエラー出力ではなく 1 を返します. 評価は sum 函数と似ていますが, 総和函数 sum とは異なる点は,product の簡易化は無い事です.

product 函数を制御する大域変数

変数名	初期値	概要
prodhack	false	product 函数の簡易化を制御

大域変数 prodhack が true であれば,product(f(i),i,3,1) は 1/f(2) となります. これは a > b の場合,product(f(i),i,a,b) = 1/product(f(i),i,b+1,a-1) とする為です. 尚,prodhack がデフォルトの false

で,a < b の場合, product(f(i),i,a,b) はエラーになります.

```
(%i12) prodhack:true;
(%o12)                                true
(%i13) product(f(x),x,5,1);
                                       1
(%o13)                          --------------
                                f(2) f(3) f(4)
```

sum 函数は ⟨ 添字変数 ⟩ を ⟨ 下限 ⟩ から ⟨ 上限 ⟩ 迄の値の ⟨ 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 函数を制御する大域変数

変数名	初期値	概要
simpsum	false	sum 函数の簡易化を制御
sumhack	false	上限と下限の入換えの制御
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 が true の場合,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;
(%o35)                                 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
```

この様に simpsum が false の場合,sum(x^n,n,0,m) の簡易化は実行されませんが, simpsum が true となると, 自動的に簡易化が実行されます.sum(x^n,n,0,inf) の場合, x の絶対値から 1 を引いたものが正か負か零であるかを利用者が指定する事で簡易化が行えます.

大域変数 sumhack が true の場合,sum(f(x),x,a,b) の a,b が a > b の場合, sum(f(x),x,a,b) を-sum(f(x),x,b-1,a+1) で置換えます. 但し,sumhack が false の場合はエラーになります.

```
(%i12) sumhack:true$
(%i13) sum(f(i),i,3,1);
(%o13) -f(2)
(%i14) sum(f(i),i,5,1);
(%o14) -f(4)-f(3)-f(2)
(%i15) sumhack:false;
(%o15) false
(%i16) sum(f(i),i,5,1);
Lower bound to sum: 5
is greater than the upper bound: 1
 -- an error.  Quitting.  To debug this try debugmode(true);
```

大域変数 sumexpand が true の場合,sum 函数同士の積を纏めて 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;
(%o2)                                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
```

但し, この処理は inf や minf を含む総和に対して利用する事は危険です.

大域変数 cauchysum は大域変数 sumexpand と対で用います. 大域変数 sumexpand と cauchysum の両方が true の場合, 総和の掛算を纏める際に, 通常の積ではなく Cauchy 積が利用されます.

```
(%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 は sum 函数内部の疑似変数を生成する為に用いられる大域変数です. 大域変数 genindex には疑似変数のアルファベットが, 大域変数 gensumnum には疑似変数の番号が各々設定されています. 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 を true にした為, 総和の積が纏められてしまい, 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. この時,gensumnum は最初が 0 で, それから二つの疑似変数を生成した為,2 になっています.

2.3.12 式の最適化

式の最適化を行う函数

optimize(⟨ 式 ⟩)

optimize 函数は ⟨ 式 ⟩ をより効率的に計算出来る Maxima の式を返却します.optimize は式の展開を行うものではありませんが, 式に含まれる共通部分式を内部変数で置換えて, 効率的に計算出来る様な式に変換します. この際に,block 文が用いられます. 但し, 共通部分式が無い場合は, ⟨ 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	初期値	概要
otimprefix	%	optimize 函数で生成される記号で利用

大域変数 otimprefix に設定される文字は optimize 函数で生成された記号に使用される前置詞です.

2.4 代入操作

多項式の計算で, 方程式を求めた結果を早速, 式に代入したい事があります. この場合, 規則による代入や, 直接変数に対して x:a の様に割当てを行う事もありますが, これらとは別に subst 等の代入用の函数を用いる方法があります.

この節では最初に通常の代入函数について解説します. ここで通常とは,Maxima での式の表現に注意を払わなくても済む函数の事です. 例えば, 式中の変数 x に 2 を代入する様な函数です.Maxima には式の表現から部分式や演算子を指定して入れ換える函数があります. この函数は与式から部分式を取出す part 函数や inpart 函数 に対応するものです.

2.4.1 通常の代入函数

subst 函数

subst(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, ⟨ 式$_3$⟩)

subst 函数は ⟨ 式$_3$⟩ 中の ⟨ 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関数名, 或いは,⟨ 式$_2$⟩ は アトムや ⟨ 式$_3$⟩ に完全に含まれる部分式でなければなりません.

例えば,$x+y+z$ は $2*(x+y+z)*w$ に完全に含まれる部分式になりますが, $x+y$ 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時に, 各階層が構成する式を取出したものになるからです. ここで,⟨ 式$_2$⟩ が ⟨ 式$_3$⟩ の部分式でない場合,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$_b$ の様に割算を伴う場合,subst(⟨ 式$_1$⟩ $*$ ⟨ 式$_b$⟩, ⟨ 式$_a$⟩, ⟨ 式$_3$⟩) が使えます. 又, ⟨ 式$_2$⟩ が ⟨ 式$_a$⟩$^{1/\langle\text{式}_b\rangle}$ の形式であれば, subst(⟨ 式$_1$⟩^ ⟨ 式$_b$⟩, ⟨ 式$_a$⟩, ⟨ 式$_3$⟩) が使えます.

subst に影響を与える大域変数

変数名	初期値	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 が true であれば%e^(a*x) の%e^x を y で置換える操作が可能になります.

大域変数 opsubst が 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.

例えば, (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, 大域変数 opsusbt が true であれば,subst 函数は与式 r+r[0] の全ての r に x^2 を代入しますが, 大域変数 opsubst を false にすると左側の r のみに代入操作が行われ,r[0] の r には 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

ratsubst (⟨ 式$_1$⟩, ⟨ 式$_2$⟩, ⟨ 式$_3$⟩)

ratsubst 函数は *langle* 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判っています. その為,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	初期値	概要
radsubstflag	false	冪乗の扱いを制御

大域変数 radsubstflag(ratsubstflag ではない事に注意) 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.

例えば,a を x^(1/3) とした場合,x は a^3 に等しくなりますが, この様に, 式 x の x^(1/3) を a で置換える事は通常出来ません. 大域変数 radsubstflag が true の場合, この様な代入が行えます.

```
(%i5) radsubstflag:true;
(%o5)                                true
(%i6) ratsubst(a,x^(1/3),x);
                                       3
(%o6)                                 a
```

fullratsubst 函数

fullratsubst(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, ⟨ 式$_3$⟩)

fullratsubst 函数は ratsubst 函数と同じですが, 結果が変化しなくなる迄, 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又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出来ます. 最初の引数は単一の代入方程式かその様な方程式のリストで, 第二の引数は仮定された式となります.

lratsubst 函数

lratsubst(⟨ リスト ⟩, ⟨ 多項式 ⟩)

lratsubst 函数は 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

sublis(⟨ リスト ⟩,⟨ 式 ⟩)

sublis 函数は ⟨ 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 の形で式を記述します. 演算子=の左辺の a が ⟨ 式 ⟩ に含まれるアトムや函数名を指定し, 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)
```

尚,`sublis([sin=cos,cos=sin],cos(x)^2+sin(x+1)^3)` の様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為,cos と sin が入れ換えられている事に注意して下さい.

大域変数 sublis_apply_lambda が sublis 函数を実行した後の簡易化を制御します.

2.4.2 substpart 函数と substinpart 函数

substpart 函数と substinpart 函数は与式の部分式を取出す part 函数と函数に各々対応する代入函数です.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となっています. substpart 函数と substinpart 函数の対応もこれと同様です. その為, 大域変数 inflag を true に設定して part/substpart 函数を呼出す事は,inpart/substinpart 函数を呼出す事と同じ事になります.

substpart と substinpart 函数

substpart(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, ⟨ 正整数$_1$⟩, $\cdots$, ⟨ 正整数$_n$⟩)
substinpart(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, ⟨ 正整数 ⟩, $\cdots$)

substpart 函数は ⟨ 式$_2$⟩ から part 関数の様に ⟨ 正整数$_1$⟩, $\cdots$, ⟨ 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) の様に括る必要があります.

具体的な例で説明しましょう. 式 $\frac{1}{x^3+3x^2+1}$ の成分の入れ換えを行いましょう. この式の構造は, 図 2.1 に示すものになります.

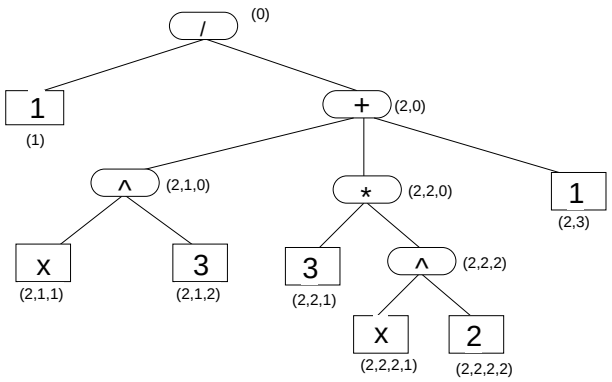

図 2.1: $\frac{1}{x^3+3x^2+1}$ の構造

substpart 函数はこの構造に従って入れ換えを行います. その為, 演算子や式で成分を置換える事も可能です.

```
(%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)
```

substinpart 函数は substpart 函数に似ていますが,substpart 函数と違い, 式の内部表現に対して作用します.

2.5　式の展開と簡易化

2.5.1　自動展開を行う大域変数

Maxima では x+x の様な簡単な入力に対して, 入力式を Maxima の内部表現に変換する際に式の簡易化を実行しますが, 式の展開迄は通常実行しません. ところが, 大域変数の設定を変更する事で,Maxima は与式が多項式や指数関数を含む場合に与式を自動的に展開したり, 簡易化を行う様になります.

自動展開に関連する大域変数

変数名	初期値	概要
demoivre	false	指数函数の表示を制御
%emode	true	指数関数の簡易化を制御
%enumer	false	自然底の浮動小数点への自動変換
expon	0	自動展開される負の冪の次数を設定
expop	0	自動展開される正の冪の次数を設定

大域変数 demoivre を true で, 与式%e^(a+b*%i) の b が実数の場合, 与式は%e^a*(cos(b)+%i*sin(b)) に自動的に展開されます.

```
(%i18) exp(a+b*%i);
                                    %i b + a
(%o18)                             %e
(%i19) demoivre:true;
(%o19)                               true
(%i20) exp(a+b*%i);
                              a
(%o20)                      %e  (%i sin(b) + cos(b))
```

大域変数%emode が true であれば,%e^(%pi*%i*x) が次の様に簡易化されます.

- x が整数, 或いは 1/2,1/3,1/4 や 1/6 と整数の積であれば, 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 y は x-2*k,k は abs(y) $<$ 1 となる整数です.

大域変数%emode が false の場合,%e^(%pi*%i*x) の特殊な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;
```

```
(%o27)                                false
(%i28) exp(%pi*%i/2);
                                      %i %pi
                                      ------
                                        2
(%o28)                               %e
```

大域変数%enumer が true であれば,%e は 2.718$\cdots$ に変換されます.%e^x の指数が整数の場合に限り, この変換が行なわれます.

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も高い次数を定めます.

大域変数 expon と expop は初期値が 0 に設定されている為,(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. 又,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;
(%o38)                                 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;
(%o41)                                 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)
```

2.5.2　簡易化に関連する函数

述語函数の unknown 函数

函数名	true を返す条件
unknown(⟨ 式 ⟩)	⟨ 式 ⟩ が演算子を一つ持ち, 簡易化函数が不明の場合

⟨ 式 ⟩ が一つの演算子を持ち, その簡易化函数が分らない場合は true を返します.

2.5.3　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 函数は大域変数 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 函数は ⟨ 式 ⟩ を指数函数形式に変換します.

2.5.4　式の展開に関連する函数

expand 函数

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$_n$⟩)

expand 函数は和の積や指数函数内の和を展開し, 有理式の分子を各々の項に分離し, 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.

尚, 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 を通常用いるべきです.

大域変数の maxnegex と 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	初期値	概要
maxnegex	1000	expand で展開される負の冪の次数
maxpogex	1000	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 maxposex です.

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. 尚, 負の冪の最大次数は maxnegex です.

expan 函数は,expan(⟨ 式 ⟩,p,n) の場合,p を大域変数 maxposex, n を大域変数 maxnegex に割当てて ⟨ 式 ⟩ の展開を行います.

expandwrt 函数は ⟨ 変数$_1$⟩ ,···,⟨ 変数$_n$⟩ に対し,⟨ 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 でこれは制御が出来ます. この函数を使う為には予め load(stopex); で読込を実行します.

expandwrt_factored 函数は expndwrt に似ていますが, 幾分違った式の積を扱います.expand_factored 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.

演算子の分配に関連する函数

distrib(⟨ 式 ⟩)
multthru(⟨ 式$_1$⟩ ,⟨ 式$_2$⟩)
multthru(⟨ 式 ⟩)

distrib 函数は可換積演算子 $*$ に対し和 $+$ を分配します.expand との違いは, 式の最上層のみで作用する点です. 又,multthru とも最上層の全ての和を展開する点でも異なります.

multthru 函数は ⟨ 式 ⟩ の部分展開を行います. 即ち, ⟨ 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. 例えば, $(x+1)^2*(z+1)*(y+1)$ の場合, 最も左側の因子 $y+1$ で式が展開され, $(x+1)^2*(y+1)*z+(x+1)^2*(y+1)$ 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.

尚,⟨ 式$_2$⟩ には方程式を指定出来ます. 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.

尚,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, 或いは, 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
```

distrib,multthru,expand の比較

distrib((a+b)*(c+d))	⇒	a*c + a*d + b*c + b*d
expand((a+b)*(c+d))	⇒	a*c + a*d + b*c + b*d
multthru ((a+b)*(c+d))	⇒	(a + b)*c + (a + b)*d
distrib (1/((a+b)*(c+d)))	⇒	1/((a+b) * (c+d))
expand(1/((a+b)*(c+d)),1,0)	⇒	1/(a*c + a*d + b*c + b*d)
multthru(1/((a+b)*(c+d)),1,0)	⇒	1/((a+b)*(c+d)),1,0)

2.5.5 sum 函数の簡易化に関連する函数

sumcontract

sumcontract(⟨ 式 ⟩)

sumcontract 函数は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 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 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

intosum(⟨ 式 ⟩)

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事に注意して下さい.

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.

2.5.6 簡易化を行う函数

radcan 函数

radcan(⟨ 式 ⟩)

radcan 函数の引数 ⟨ 式 ⟩ は, 対数函数, 指数函数と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. 特定の変数順序に対し,CRE 表現は一意に定まります (従って,CRE 表現は式の正準表現になります). その為,radcan を用いた簡易化も一意に定まります. 但し,radcan 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(⟨ 式 ⟩ ,⟨ 規則$_1$⟩ , ···,⟨ 規則$_n$⟩)

scsimp(=Sequential Comparative SIMPlification) 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後に, もとの式が返却されます.

2.5.7　簡易化に関する補助的函数

askinteger 函数

askinteger(⟨ 式 ⟩ ,⟨ オプション引数 ⟩)
askinteger(⟨ 式 ⟩)

⟨ 式 ⟩ は任意の有効な 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 が設定されます. この函数は Maxima に蓄えられた情報から ⟨ 式 ⟩ が even,odd, 或いは integer であるかを決定しようとします.Maxima に蓄えられた情報では不十分な場合, 利用者に質問して,Maxima に情報を蓄えます.

```
(%i16) aa:1;
(%o16)                                1
(%i17) askinteger(aa);
(%o17)                               yes
(%i18) askinteger(aa,odd);
(%o18)                               yes
(%i19) askinteger(aa,even);
(%o19)                               no
(%i20) askinteger(yy);
Is  yy  an integer?

yes;
(%o20)                               yes
(%i21) askinteger(yy,odd);
Is  yy  an odd number?

no;
(%o21)                               no
(%i22) askinteger(yy,even);
(%o22)                               yes
(%i23) askinteger(zz,even);
Is  zz  an even number?
```

```
no;
(%o23)                                  no
(%i24) askinteger(zz,odd);
Is  zz  an odd number?

no;
(%o24)                                  no
(%i25) askinteger(zz,integer);
Is  zz  an integer?

yes;
(%o25)                                  yes
(%i26) askinteger(zz+yy+aa,integer);
(%o26)                                  yes
(%i27) askinteger(zz,integer);
(%o27)                                  yes
(%i28) askinteger(zz,even);
Is  zz  an even number?

yes;
(%i29) askinteger(zz*2+aa,even);
(%o29)                                  no
```

ここでの例で示す様に yy が integer であると指定すると, それから yy は integer となります. 更に,yy が odd であると宣言すれば, 自動的に even になります. 但し,zz が odd でなく,even ではないと宣言しても,askinteger(zz) は no とはならずに尋ねて来ます. ここで,yes とすれば, それまで入力した odd でも even でもない事が消去されます.askinteger は zz+yy+aa や 2*zz+aa の様な式に対しても, それ以前の入力情報から, 整数, 奇数か偶数であるかを判断します.

asksign 函数

asksign (⟨ 式 ⟩)

asksign 函数は ⟨ 式 ⟩ が, 正, 負, 或いは零であるかを決定します. この際に,Maxima に蓄えられた情報をもとに決定しようとしますが, 情報が不十分で決定出来なければ, その演繹を完遂する為に必要な質問を利用者に対して行います.

利用者の答は Maxima に記録されます.

asksign が尋ねる値は pos(正値),neg(負値),zero(零) の何れか一つです.

numerval 函数

numerval(⟨ 変数$_1$⟩ ,⟨ 式$_1$⟩ , ···,⟨ 変数$_n$⟩ ,⟨ 式$_n$⟩)

numerval 函数は ⟨ 変数$_i$⟩ を ⟨ 式$_i$⟩ の数値変数として宣言し,⟨ 式$_i$⟩ は大域変数 numer が true であれば, 任意の式に現われる変数に対して評価と代入が行われます.

2.6 リスト

2.6.1 Maximaのリスト

Maxima にはリストと呼ばれるデータ型があります. 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] の様に, 各成分をコンマ , で区切った列を大括弧 [] で括ったものです. この様に,LISP のリストの様に成分を単純に空行で区切るのではない事に注意をして下さい. 勿論,[1,2,[3,4],[4,[5]]] の様にリストを入れ子にしても構いません.

ここで Maxima は LISP で記述されたシステムの為,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. 更に,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. その為,Maxima 内部表現では演算子の部分が 0, それ以降の成分に $1, 2, \cdots$ と番号が振られた階層構造を持っています.

この事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事が多くあります. 尚,Maxima の内部表現に関しては 2.3 節を参照して下さい.

2.6.2 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 は 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 は true の場合, 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. 簡易化は式の再並び換えを行う事に注意が必要になります. その為,first(x+y) は 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.

尚, 大域変数 inflag に影響を受ける函数に以下のものがあります.

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

2.6.3　リスト処理に関連する主な函数

リストに関連する述語函数

函数	true を返す条件
atom(⟨ 変数 ⟩)	アトムの場合
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

⟨ 変数 ⟩ がアトムかリストであるかは atom 函数と listp 函数で調べる事が出来ます.atom 函数は引数がアトムであれば true, それ以外は false を返します. listp 函数は引数がリストであれば true, そうでなければ false を返します.

尚, ここでの判定では内部表現は無関係です. その為,sin (x) や x+y の様な式では各変数に値が束縛されていなければアトムでもリストにもなりません.

member 函数は二つの引数を取り,⟨ 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

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$_2$⟩ ,$\cdots$)	複数のリストの結合を実行
econs(⟨ 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

リストの長さは length を用いて調べられます. この length は LISP の同名の函数と同じ動作をしています. 但し,Maxima の任意の式は内部表現では LISP のリスト構造を持ち,length 函数はその内部表現で, 先頭の式の識別子を除いたリストの長さを返す函数です.

```
(%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) となり,length は先頭の (MPLUS SIMP) を除いたリストの長さを返しています.

reverse 函数は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. 例えば, リスト [a1,a2,a3] に対して `reverse([a1,a2,a3]);` を入力すれば [a3,a2,a1] を返却します. 但し, 各成分 ai はそのままです.

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 の様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 函数は複数のリストの結合を行います.Maxima の append 函数は LISP の append と同様の事をします.

endcons 函数は append 函数と似た函数ですが,append が先頭に追加するのに対し, endcons は後に追加します.

即ち,econs(⟨ 式$_1$⟩ ,⟨ リスト$_1$⟩) で ⟨ 式$_1$⟩ を ⟨ リスト$_1$⟩ の後に追加します. 但し,Maxima の式は内部的にはリストの為, 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
```

この様に 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 函数は ⟨ 式$_2$⟩ に含まれる ⟨ 式$_1$⟩ を式の先頭から ⟨n⟩ 個削除します. 但し, ⟨ 式$_1$⟩ が函数, 或いは単項式でなければ除去出来ません.

尚,⟨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, ⟨ 式$_1$⟩ を全て削除します.

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⟩ ,$\cdots$,⟨n_k⟩)

first 函数は ⟨ 式 ⟩ の最初の成分を返します.

これに対し,last 函数は ⟨ 式 ⟩ の最後の成分を返します.

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.inflag はデフォルト値が false の為, 内部表現に対して操作したければ, inflag:true; を実行し,inflag の値を true に変更する必要があります.

rest 函数は,⟨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.

n が負整数であれば,⟨ 式$_1$⟩ の後から ⟨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 ]
```

sublist 函数は ⟨ 述語函数 ⟩ 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.
例えば, sublist([1,2,3,4],evenp); は [2,4] を返します.
substpart 函数は ⟨ 式 ⟩ の $\langle n_1 \rangle$,⋯,$\langle n_k \rangle$ で指定した成分を $\langle x \rangle$ で置換えます. $\langle x \rangle$ は式, アトム, 演算子が指定可能です.
$\langle n_1 \rangle$,⋯,$\langle n_k \rangle$ の指定方法は, ⟨ 式 ⟩ がリストで, 第 m 番目の成分であれば m を指定します. 更に, 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] の第二成分 2 を x で置換える為に,2 を指定しています. 複合リスト [1,[2,3],4] の場合, 第二成分はリスト [2,3] なので, 第二成分を x で置換すれば [1,x,4] になります. 第二成分に含まれる 3 を x で置換えたければ, 第二成分のリストの第二成分を指定すれば良い事になります.
この事は Maxima の一般の式に対しても適応が可能です. 但し,Maxima の場合は入力した式の順番と,Maxima に入力された後の順番が異なる事がある為, 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 sin(x) で置換える操作を行っています．ここで第一成分は入力した順序とは異なって%eˆx となっている事に注意して下さい. 次に, 第二成分は有理式全体となりますが, この第二成分の構造は (割算,x+1,xˆ2+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．その為,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しまい, 有理式が xˆ2+2*x+2 で置換えられてしまいます.

2.6.4　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ものです．これは Mathematica や Maple でも採用されており, 非常に便利な函数です.Maxima では, 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為,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 での map 函数には,map,maplist,scanmap の 3 個あります. これらの函数は全て Maxima の式に函数を作用させる働きがあります. この中で,scanmap のみが一つの函数を一つの式に作用させますが,map と maplist は n 個の引数を持つ函数に n 個の式に作用させる事が出来ます.

又,map 函数と maplist 函数はリストや式の第一層に含まれる部分式に作用させる事が出来ます. 更に,scanmap 函数は式の全階層の部分式に作用させる事が可能です.

先ず,map 函数の例を示します.

```
(%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 x*y の場合, 主演算子は*, 第一層には被演算子の x と y がある為,sin は式 x*y の第一層の部分式 x と y に作用しますが, 演算子の置換を行わない為に sin(x)+sin(y) が得られます.x*y+y の場合, この式の第一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 を作用させていますが,factor(x*y+y) が (x+1)*y となる為, この式の第一層には部分式 x+1 と y があり, 主演算子が*なので, 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))
```

上の例で示す様に, 内部表現で MTIMES が MLIST の変化している事に注目して下さい.

2.6.5 map 函数族に関連する大域変数

大域変数 maperror

変数名	初期値値	概要
maperror	true	map や maplist 函数を制御します

大域変数 maperror は map 函数と maplist 函数の動作に影響します. 先ず,map 函数と maplist 函数の引数は ⟨ 函数 ⟩,⟨ 式$_1$⟩,$\cdots$,⟨ 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, それ以外の引数でも適用可能になり, 以下の動作となります.

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$_1$⟩,$\cdots$,⟨ 式$_n$⟩] に ⟨ 函数 ⟩ を作用させ,apply 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);
```

2.6.6　map 函数いろいろ

map 函数に関連する述語函数

mapatom(⟨ 式 ⟩)

mapatom 函数は,⟨ 式 ⟩ が map 函数によってアトムとして扱われる時, true を返します

map 函数

map(⟨ 函数 ⟩,⟨ 式$_1$⟩, ···,⟨ 式$_n$⟩)
maplist(⟨ 函数 ⟩,⟨ 式$_1$⟩, ⟨ 式$_2$⟩,···)

ma 函数は 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 函数は map 函数に似ています.⟨ 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 の場合,⟨ 式$_i$⟩ の主演算子で式を纏めたものが出力されるのに対して,maplist では, 主演算子の代りに Maxima のリスト演算子が置かれる事です.

```
(%i27) maplist(sin,x+y);
E(%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 ⟩)
scanmap(⟨ 函数 ⟩,⟨ 式 ⟩,bottomup)

scanmap(⟨ 函数 ⟩,⟨ 式 ⟩) の場合, 再帰的に ⟨ 函数 ⟩ を ⟨ 式 ⟩ の内部表現の頂上から適用して行きます. これは徹底した因子分解が望ましい時には特に便利です.

例えば, $(a^2+2a+1)y+x^2$ 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以下に示します.

```
((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, 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 等々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となる為, 上記の結果を得ています. この作用の様子は factor 函数の代りに'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)    ))
```

この性質がある為に式を全て展開してしまうと, 各項に factor 函数を作用させる為に, 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, 内部表現の最下層から ⟨ 函数 ⟩ を作用させます.

2.7 配列

2.7.1 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. 先ず,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_art_q 函数や 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) a2:make_art_q(10);
(%o3)        {Array:  #(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)}
(%i4) array(a3,fixnum,5);
(%o4)                                a3
(%i5) make_array(hashed,5);
(%o5)             {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事が可能です. 又,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事で行います. 他に, fillarray 函数等を用いて他の配列やリストから生成する事も可能です.

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. 先ず,make_art_q 函数で生成する配列は LISP の make-array 函数を直接用いて配列を生成するものです. その為,Maxima の配列の中では最も原始的なものとなります.

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事が可能です.

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]]
```

先ず,listarry(a1) で, 配列 a1 に 1 と 10 が設定されている事が判ります. arrayinfo(a1) で返却される 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為に, 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

make_art_q(⟨ 整数 ⟩)
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$_n$⟩)

make_art_q 函数は LISP の make-array 函数を呼出して配列を生成するだけの函数で,Maxima の配列を生成する函数の中では最も原始的な配列を生成する函数です.

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. 即ち,array 函数で生成可能な配列は 5 次以下の配列です. 例えば,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.

use_fast_arrays

変数名	初期値	概要
use_fast_arrays	false	配列の種類を制限

先ず, 大域変数 use_fast_arrays が true の場合,make_array 函数を用いて any 型の配列を生成します. 尚,make_array 函数で any 型の配列を生成する場合, 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. 初期値が設定される事を除くと,make_art_q 函数で生成される配列との違いはありません.

次に大域変数 user_fast_arrays がデフォルトの 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, 浮動小数点型と整数型配列, その他に complete 型があります.

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. 尚,functional 型の場合は第一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. その為,flonum 型の場合は初期値が 0.0,fixnum 型の場合, 初期値が 0 の配列となります. 尚, 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.

先ず,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. 尚,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為, 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. 尚, この場合は LISP の gensym 函数で生成されたシンボルが表示されます.

```
(%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]
```

但し,listarray 函数や arrayinfo 函数でこのシンボルは使えません. その為,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.

2.7.2 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 ⟩)

第一引数の ⟨ 配列 ⟩ に第二引数に指定した ⟨ 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, 第二引数は浮動小数点 (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.

第一引数の配列には第二引数の内容が先頭から順番に入れられますが, もしも, 第一引数の配列が第二引数よりも大きければ, 第二引数の最後部の元で第一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$_1$⟩,$\cdots$, ⟨ 添字$_k$⟩])

arrayapply は第一引数に ⟨ 配列 ⟩ を取り, その後に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$_1$⟩, $\cdots$,⟨ 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て行きます. 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$_2$⟩,$\cdots$)
remarray(all)

remarray 函数は函数を除去し, 占領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$_i$⟩ の内部表現で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.

2.8　行列

2.8.1　行列について

Maxima では行列を扱う事が出来ます. 但し,MATLAB の様な数値行列処理ソフトの様にリストの一種類として直接設定出来るものではありません.

一般的には,matrix 函数を用いて行列を生成します. この matrix 函数は Maxima の複数のリストから行列を生成する函数です. 構文は以下の様になっています.

行列の定義を行う matrix 函数

matrix(⟨ リスト$_1$⟩ ,$\cdots$,⟨ リスト$_n$⟩)

ここで, ⟨ リスト$_i$⟩ は行列の i 行に対応します. Maxima のリストは [1,2,3] の様に, コンマで区切った式を大括弧で括ったものです. 行列は Maxima の内部では, 先頭が matrix で開始する S 式となっており, substpart 等の函数を用いて,matrix を"["に変更すれば, リストに置換出来ます. 又, 逆にリストから行列に置換する事も可能です,

以下に行列の生成と行列からリスト, リストから行列への変換の例を示します.

```
(%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) c:substpart(matrix,b,0);
                                  [ 1  2  3 ]
(%o6)                             [         ]
                                  [ 4  5  6 ]
```

行列の和と差は演算子 + と演算子 − を各々用います. 尚, 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. 通常の行列積を行いたい場合には, 非可換積演算子を用います.

以下に matrix 函数による行列の生成と, 演算子 $+, -, *, /$ の結果を示します.

```
(%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 ]
(%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 ]
```

この様に, 演算子 $+$ と $-$ 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 $*$ は可換性を持っている為, 可換性を持たない行列の積は表現出来ないからです. その為, 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, 或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. デフォルトでは, 分配律と結合律を満す様になっています. 次に, 非可換積に対応する行列の冪乗は記号^^で記述します. 特に, 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. 尚, 非可換積の詳細に関しては,1.6.5 小節を参照して下さい.

2.8.2　行列を生成する函数

Maxima には matrix 函数の他にも行列を生成する函数を持っています.

行列の生成を行う函数

entermatrix(⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩)
ident(⟨ 整数 ⟩)
zeromatrix(⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩)
genmatrix(⟨ 配列 ⟩, ⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩ , ⟨ 整数$_3$⟩ ,⟨ 整数$_4$⟩)
ematrix(⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩ ,⟨x⟩ , ⟨ 整数$_3$⟩ ,⟨ 整数$_4$⟩)
diagmatrix(⟨ 整数$_1$⟩ ,⟨ 整数$_2$⟩)
coefmatrix([⟨ 方程式$_1$⟩ ,···], [⟨ 変数$_1$⟩ ,···])
augcoefmatrix([⟨ 方程式$_1$⟩ ,···], [⟨ 変数$_1$⟩ ,···])
echelon(⟨ 行列 ⟩)

entermatrix を実行すると,Maxima の要求に従って ⟨ 整数$_1$⟩ × ⟨ 整数$_2$⟩ 個の成分を入力して ⟨ 整数$_1$⟩ 行 ×⟨ 整数$_2$⟩ 列の行列を生成する函数です.

```
(%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 ]
```

この様に対話的に行列を生成する事が出来ますが, 実用上, 用途は限られるかもしれません.

ident 函数は, ⟨ 整数 ⟩ 次の単位正方行列, 即ち, 対角成分が全て 1 で, 他が全て 0 となる正方行列を生成します.

```
(%i6) ident(3);
                                [ 1  0  0 ]
                                [         ]
(%o6)                           [ 0  1  0 ]
                                [         ]
                                [ 0  0  1 ]
```

zeromatrix 函数は, ⟨ 整数$_1$⟩ 行 ⟨ 整数$_2$⟩ 列の零行列を返します.

```
(%i7) zeromatrix(3,2);
                                 [ 0  0 ]
                                 [      ]
(%o7)                            [ 0  0 ]
                                 [      ]
                                 [ 0  0 ]
```

genmatrix 函数を用いると, 与えられた二次元配列から行列が生成出来ます. この genmatrxi 函数は二次元配列から行列を抜き出す形になります. 抜き出し方は,⟨ 整数$_1$⟩,⟨ 整数$_2$⟩ で配列の大きさを指定します. それから,⟨ 整数$_3$⟩,⟨ 整数$_4$⟩ で行列の 1 行,1 列目の成分を指定します. すると,⟨ 整数$_1$⟩ から ⟨ 整数$_3$⟩ 迄の配列の第一成分を行とし, と ⟨ 整数$_2$⟩ から ⟨ 整数$_4$⟩ 迄の第二成分を列とする領域が ⟨ 整数$_1$⟩ – ⟨ 整数$_3$⟩ 行, ⟨ 整数$_2$⟩ – ⟨ 整数$_4$⟩ 列の行列として抜き出されます. ここで, 実際の配列が生成する行列よりも小さかった場合, 即ち,i 行 j 列の成分が未定の場合,⟨ 配列 ⟩$_{ij}$ の形式で行列の成分が表示されます. 又, 以下に示す様に配列を函数で与える事も可能です.

```
(%i3)  h[i,j]:=1/(i*j+1)
(%i4)  genmatrix(h,3,3);
                                [    1   1 ]
                                [ 1  -   - ]
                                [    2   3 ]
                                [         ]
                                [ 1  1   1 ]
(%o4)                           [ -  -   - ]
                                [ 2  3   4 ]
                                [         ]
                                [ 1  1   1 ]
                                [ -  -   - ]
                                [ 3  4   5 ]
```

ematrix 函数は ⟨ 整数$_1$⟩ 行 ⟨ 整数$_2$⟩ 列の行列で, (⟨ 整数$_3$⟩,⟨ 整数$_4$⟩) 成分のみが ⟨x⟩ となり, 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 函数は n 行 n 列で, 対角成分が x の対角行列を返します. ここで対角成分は全て ⟨ 行列 ⟩ のものです. 尚, diagmatrix(n,1) は ident(n) と同じ n 次元の単位行列を生成します.

```
(%i19) diagmatrix(3,2);
                                  [ 2  0  0 ]
                                  [         ]
(%o19)                            [ 0  2  0 ]
                                  [         ]
                                  [ 0  0  2 ]
```

coeffmatrix は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. 尚, 連立一次方程式は, 演算子=の右辺, 或いは左辺の何れかが 0 の場合, 式だけでも構いません.

```
(%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 函数は, 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$_1$⟩,$\cdots$ から構成される線形方程式系に含まれる ⟨ 変数$_1$⟩ ,$\cdots$ 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 ]
(%i27)
```

echelon 函数は, ⟨ 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様に変形するものです (上三角行列).

```
                        [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]
```

行列処理を指定する大域変数

変数名	初期値	概要
assumescalar	true	引数がスカラー値であると仮定.
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
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

assumescalar が true の場合, 値が割当てられていない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.

detout が true であれば, 逆行列を計算した時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為には doallmxops と doscmxops が 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 で与える事も出来ます.

ratmx が 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 (ratfac の設定に依存するものの) の原因になります.

scalarmatrixp が 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, この変換は, 1 行 1 列の行列は何時でもスカラに変換されます. 但し,false であれば, この変換は実行されません.

sparse が true で ratmx:true であれば,determinant は疎行列式を計算する為のルーチンを利用します.

do 一家

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

doallmxops が true であれば, 全ての行列演算子が評価されます. false であれば, 演算子を支配する個々の dot 大域変数の設定が実行されます.

domxexpt が 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. 尚,⟨ 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.

domxmxops が 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. 尚, 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.

domxnctimes が false であれば行列の非可換積が実行されます.

doscmxops が true であればスカラと行列間の演算子が実行されます.

doscmxplus が true であれば, スカラ+行列が行列値となります. この大域変数は doallmxops と独立した変数です.

行列の括弧を設定する大域変数

変数名	初期値	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

lmxchar は行列の (左) の括弧として表示する文字を設定します. 右側は rmxchar で指定します.

rmxchar は行列の (右) の括弧として表示する文字を設定します. 左側は lmxchar で指定します.

matrix_element 変数一族

変数名	初期値	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

matrix_element_add は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.

matrix_element_mult は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.

matrix_element_transpose は上記の大域変数と同様に, 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.

2.8.3 行列に関連する函数

行列に関連する述語函数

構文	true を返す条件
matrixp(⟨ 式 ⟩)	⟨ 式 ⟩ が行列の場合
diagmatrixp(⟨ 式 ⟩)	⟨ 式 ⟩ が対角行列の場合
nonscalarp(⟨ 式 ⟩)	⟨ 式 ⟩ がスカラでない場合
scalarp(⟨ 式 ⟩)	⟨ 式 ⟩ がスカラの場合

matrixp 函数は ⟨ 式 ⟩ が行列であれば true, そうでなければ false を返します.

diagmatrixp 函数は ⟨ 式 ⟩ が対角行列の場合に true を返します.

nonscalarp 函数は ⟨ 式 ⟩ が非スカラ, つまり, アトムを含み, 非スカラと宣言されたリストや行列であれば true を返しますが, スカラの場合は false を返します.

scalarp 函数が true となるのは,⟨ 式 ⟩ が数, 定数やスカラとして宣言された変数, 数, 定数, そしてその様な変数の合成で行列やリストを含まない場合です.

行列の成分に函数を作用させる函数

matrixmap(⟨ 函数 ⟩ ,⟨ 行列 ⟩)

matrixmap 函数は, 行列 ⟨m⟩ の各成分に ⟨ 函数 ⟩ を作用させる函数です. 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) ]
```

行や列の取り出しと追加を行う函数

col(⟨ 行列 ⟩ ,⟨i⟩)
row(⟨ 行列 ⟩ ,⟨i⟩)
addcol(⟨ 行列 ⟩ ,⟨ リスト$_1$⟩, ⟨ リスト$_2$⟩ ,$\cdots$,⟨ リスト$_n$⟩)
addrow(⟨ 行列 ⟩ ,⟨ リスト$_1$⟩ , ⟨ リスト$_2$⟩ ,$\cdots$,⟨ リスト$_n$⟩)
copymatrix(⟨ 行列 ⟩)
setelmx(⟨x⟩ ,⟨i⟩ ,⟨j⟩ , ⟨ 行列 ⟩)

col 函数と row 函数は, ⟨ 行列 ⟩ に対し, ⟨i⟩ で指定される列と行を各々行列の形式で返す函数です.

```
(%i17) m1;
                        [ x   0   - 3   x    0   0 ]
                        [                          ]
                        [ 0   x    0   - 3   x   0 ]
                        [                          ]
                        [ 0   0    x    0   - 3  x ]
(%o17)                  [                          ]
                        [ y  - 3   0    y    0   0 ]
                        [                          ]
                        [ 0   y   - 3   0    y   0 ]
                        [                          ]
                        [ 0   0    y   - 3   0   y ]
(%i18) row(m1,1);
(%o18)                     [ x  0  - 3  x  0  0 ]
(%i19) col(m1,1);
                                    [ x ]
                                    [   ]
                                    [ 0 ]
                                    [   ]
                                    [ 0 ]
(%o19)                              [   ]
                                    [ y ]
                                    [   ]
                                    [ 0 ]
                                    [   ]
                                    [ 0 ]
```

addcol 函数は列として,addrow 函数は行として, 複数の複数のリストや行列を ⟨ 行列 ⟩ に追加します. 尚, 追加するリストや行列は, 行列の大きさに矛盾しないものでなければなりません.

copymatrix 函数は ⟨ 行列 ⟩ の複製を行います. この命令は ⟨ 行列 ⟩ を成分毎に再生成する時だけに使います.

setelmx 函数は ⟨ 行列 ⟩ の (⟨i⟩,⟨j⟩) 成分を ⟨x⟩ で置換します.

```
(%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 ]
```

尚,setelmx 函数を使わなくても, 行列成分を直接指定して置換える事も可能です. この場合, A[i,j]:x で行列 A の (i,j) 成分を x で置換します. 但し, この場合の返却値は x になります.

小行列を生成する函数

minor(⟨ 正方行列 ⟩ ,⟨i⟩ ,⟨j⟩)
submatrix(⟨ 行$_1$⟩, $\cdots$, ⟨ 行$_m$⟩, ⟨ 行列 ⟩, ⟨ 列$_1$⟩, $\cdots$, ⟨ 列$_n$⟩)
submatrix(⟨ 行$_1$⟩, $\cdots$, ⟨ 行$_m$⟩, ⟨ 行列 ⟩)
submatrix(⟨ 行列 ⟩, ⟨ 列$_1$⟩, $\cdots$, ⟨ 列$_n$⟩)

minor 函数は与えられた ⟨ 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 函数は ⟨ 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.

転置, 上三角, 共役な行列を計算する函数

transpose(⟨ 行列 ⟩)
triangularize(⟨ 行列 ⟩)

transpose 函数は ⟨ 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 函数は ⟨ 行列 ⟩ の上三角行列形式を生成します. 尚, 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  ]
```

階数と対角和を計算する函数

rank(⟨ 行列 ⟩)
mattrace(⟨ 行列 ⟩)

rank 函数は ⟨ 行列 ⟩ の階数を求めます. 行列の階数は行列から求めた小行列式で, 零にならない小行列式で, 最大のものの大きさです. 尚,rank 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事があります.

mattrace 函数は,⟨ 行列 ⟩ が正方行列の場合, 対角和, 即ち, 行列の主対角成分の総和を計算します. この函数は,ncharpoly で利用されています. ncharpoly は Maxima の charpoly の代りに使える函数です. 尚,mattrace 函数を利用する為には, 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
```

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 函数は ⟨ 正方行列 ⟩ の余因子行列を計算します.

invert 函数は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 命令は余因子の計算で利用されるので,ratmx が false ならば, 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. 尚, 大域変数 detout が true の場合, 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 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
```

尚, 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 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 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    ]
```

行列式を計算する函数

determinant(⟨ 行列 ⟩)
newdet(⟨ 行列 ⟩)
newdet(⟨ 配列 ⟩ ,⟨ 整数 ⟩)
permanent(⟨ 行列 ⟩ ,⟨ 整数 ⟩)

determinant 函数は,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 函数は ⟨ 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. 尚 ⟨ 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,⟨ 行列 ⟩ が正方行列であればそのまま行列式を計算し,⟨ 行列 ⟩ が正方行列で無ければ, 余分な末尾の行, 或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 函数は,⟨ 行列 ⟩ の permanent を計算します. 尚 ⟨ 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.

特性多項式の生成に関連する函数

charpoly(⟨ 行列 ⟩ ,⟨ 変数 ⟩)
ncharpoly(⟨ 行列 ⟩ ,⟨ 変数 ⟩)

charpoly 函数は ⟨ 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 函数は ⟨ 変数 ⟩ に対する ⟨ 行列 ⟩ の特性多項式を計算します. これは Maxima の charpoly とは別物です.

ncharpoly 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 は var*ident[n]-a の行列式を計算している. そんな訳で ncharpoly は優れている. 例えば, 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為です. 予め load(”nchrpl”); で読込む必要があります.

2.8.4 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為には, load(eigen); を予め実行します. この load(eigen) を実行する事で, 函数が読込まれ, 以下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	初期値	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	[]	左行列
leftmatrix	[]	右行列

大域変数 hermitianmatrix が true の場合, 与えられた行列が Hermite 行列であると仮定します. その為, 大域変数 leftmatrix は rightmatrix の転置行列と複素共役な行列になります. 又,rightmatrix 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 が false の場合, 大域変数 leftmatrix と rightmatrix に行列が設定され,leftmatrix . ⟨ 行列 ⟩ . rightmatrix が対角行列で,⟨ 行列 ⟩ の固有値がその対角成分に現れるもの

となります. 但し,nondiagonalizable が true であれば, これらの行列は生成されません.

knowneigvals が true の場合, 行列の固有値は既知で, 大域変数の 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. その為,knowneigvals が true の場合は listeigvals に eigenvalues 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

knowneigvlects が true の場合, 行列の固有値に対応する固有ベクトルが既知であり, 大域変数の 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. その為,knowneigvects が true の場合は listeigvects に 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 函数は内積を表現します. 短縮名は inprod です. 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 函数は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 は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) は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一成分が固有値リスト, 第二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の為, 高次多項式に対しては, その根を見付け損なう事があります. 更に, 不正確な答を返す事もあります. しかし 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 函数を必要としません.

eigenvectors 函数は引数として一つの行列を取り, 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.

尚,algsys 函数が固有ベクトルの計算で使われています. 尚, 固有値が不正確な場合, algsys が解を生成する事が出来ない事があります. 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事を勧めます.

similarityransform 函数は ⟨ 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 函数は Gram-Schmidt による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 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.

gramschmidt 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. 尚、返ってきた結果には因子分解された整数が含まれる事があります. これは Maxima の factor 函数が gram-schmidt の処理の過程で使われた為です. こうする事で式が複雑なものになる事を回避し, 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 は ⟨ リスト ⟩ の大きさを 1 にしたリストを返します. uniteigenvalues は与えられた ⟨ 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. 出力リストの第一成分のリストには eigenvalues 函数の出力があり, 第二成分の副リストには正規化した固有ベクトルが, 第一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 函数は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.

第3章　プログラム

この章で解説する事:

- Maxima でプログラム
- 函数の定義
- データの入出力

3.1 Maximaでプログラム

Maxima には制御文として,if 文, 反復処理に do ループや go といった, 原始的な構文があります. 又,block 文を用いる事で局所変数を利用する事も出来ます.

Maxima の処理言語は C や PASCAL の様な手続型の言語に見えます. しかし,LISP 上で動作する為に, データの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.

Maxima には compile 函数等で,Maxima の処理言語で記述した函数をコンパイルする手段もあり, それによってある程度の速度向上も見込めます, それでも,Maxima の処理言語で記述したプログラムを LISP で解釈して実行する手間もある為, 直接, LISP で記述する方が速度的には有利です.

3.1.1 block 文

block 文

block(⟨ 変数リスト ⟩,⟨ 式$_1$⟩,⟨ 式$_2$⟩,$\cdots$,⟨ 式$_n$⟩)

Maxima の block は FORTRAN の subroutine,PASCAL の procedure に似たものです. 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事に使えます. 同名の大域変数が存在した場合,block 文の実行中, 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 文の局所変数の値は破棄されてしまいます.

尚, 局所変数を用いない場合には, 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. 更に, 局所変数に初期値を設定する事も可能です. この場合,[a:1,b:2] の様に局所変数に対して:による通常の割当を行います.

ここで,block 内部で用いられているものの,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. その為, 値は block 文の終了後も保持されます.

block の値は, 最後の文の値か block から 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 内の文に移動する為に使えます. 文のラベル付けは,block 内部で, アトムを文の前に置く事で対処します. 例えば,block(⟨ 式$_1$⟩,$\cdots$,⟨ 式$_n$⟩, loop,⟨ 式 ⟩,$\cdots$,go(loop),$\cdots$) の様に用います. go 函数の引数は go 函数を含む block 内部に現われるラベル名でなければなりません. go を含まない block 文中のラベルに go で移動する事は出来ません.

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 を置かなかった場合には,block 文の末尾の式の値が返却値となります.

以下に block 文を用いた函数の例を示します. 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
```

この例で示す様に, 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.

3.1.2 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 ⟩ を評価して,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, 又は false であるかが評価可能な式で, 述語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)
and	and	中置式演算子 (infix)
または	or	中置式演算子 (infix)
否定	not	前置式演算子 (prefix)

尚, 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もです. 詳細は演算子の節を参照して下さい.

これに対し,⟨ 式$_1$⟩ と ⟨ 式$_2$⟩ は任意の Maxima の式 (勿論,if 文を含んで入れ子になっても構いません) が利用可能です.

3.1.3　do 文による反復処理

Maxima では,do 文を用いて反復処理を行います.

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. それから, 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, 再度, 終了条件の判定を行う事を反復します.

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.

ここで, 最初の初期値の割当てでは, 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. 尚, 初期値が 1 の場合,: ⟨ 初期値 ⟩ や from ⟨ 初期値 ⟩ は省略可能です.

次に, 制御変数を一定の値で増加させたり, 減少させたければ,do 文の内部に, step ⟨ 増分 ⟩ を追加します. この時, ⟨ 増分 ⟩ を正実数とすれば, 通常の増分となり, 負の実数とすれば, 減少分になります. 尚, ⟨ 増分 ⟩ が 1 の場合は step 1 を省略する事が可能です

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 ⟩ とすれば,i の値に ⟨ 制御変数の式 ⟩ で計算した値が制御変数に割当てられます.

例えば, 以下の二つ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. 最初の二つでは step を用いて増分を定めています. 最後の next を用いたものでは, 函数 i+2 で制御変数の値を定めています.

これらの方法に加えて, リストを用いた制御変数の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, リストの成分には数値以外の函数や式を用いる事も可能です.

```
(%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.

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, 即ち,true か false かが判別可能な式です.unless を用いた do 文は,C 等の repeat-until 文,while を用いた do 文は while 文に相当します.

thru,unless と while を用いた例を以下に示します. 尚, 三種類ともに全て同じ反復処理, 即ち,1 から 10 迄の数を表示して終了, を実行するのです.

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);
```

尚, 通常の do 文によって返される値はアトムの done です. 函数 return を用いると, 本体の中で do から早目に抜けて必要な値を与える事に使えます. block にある do 文中の return 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 文から抜ける為に使ってはなりません.

プログラムに関連する大域変数

変数名	初期値	概要
backtrace	[]	函数リスト
dispflag	true	block 文中の函数出力を制御
prederror	true	if 文や is 函数のエラーメッセージ出力を制御
errorfun	false	エラー発生時に起動させたい函数名

backtrace は debugmode:all の時に, 入力された函数全てのリストを値として持ちます.

dispflag が false ならば,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 dispflag を false に設定します.

prederror が true であれば,if 文や is 函数で,true か false であるか述語の評価に失敗すると, 何時でもエラーメッセージが表示されます.

false であれば,unknown が代りに返されます.

errorfun に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, その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時に使えます.

block 文内部で利用する函数

go(⟨ ラベル ⟩)
return(⟨ 式 ⟩)
break(⟨ 引数 ⟩ ,···)
catch(⟨ 式$_1$⟩ ,···,⟨ 式$_n$⟩)
throw(⟨ 式 ⟩)

go 函数は block 内部で指定した block 文中のラベルに移動する事に用いられます. ラベルとしてアトムを用い, このアトムは文の前に置きます.

```
block([x],
       x:1,
  loop,x+1,
       ...,
       go(loop),
       ...)
```

go の引数は同じ block の中で現われるラベルでなければなりません. go を含む別の block にあるラベルに go を用いて移動する事は出来ません.

return 函数は block 文から引数を伴って抜ける時に用います. 場所は block 文の何処に置いても構いません.

break 函数は,⟨ 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, 変更する事が出来る様にします. (maxma-break) からは exit; を入力すれば計算が再開されます. 尚,Ctrl+a(^a) で maxima-break に何時もで対話的に入る事が出来ます.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事に用いても構いません.

catch 函数は throw と対で用います. これは非局所的回帰 (non-local return) で用いる函数で, 最も近い throw に対応する catch に行きます. その為,throw に対応する catch 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 の評価に至らなかった場合, catch 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,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. それ以外では,g は l の最初の負の要素を捉えて, それを放します.

throw 函数は ⟨ 式 ⟩ を評価し, 近くにある catch に値を投げ返します. throw は catch と一緒に使われます.

エラー処理を行う函数

errcatch(⟨ 式$_1$⟩ ,···,⟨ 式$_n$⟩)
error(⟨ 引数$_1$⟩ ,···,⟨ 引数$_n$⟩)
errormsg()

errcatch 函数は引数を一つづつ評価して, エラーが生じなければ最後の値のリストを返します., 任意の引数の評価でエラーが生じた場合,errcatch はエラーを捉えて即座に [](空のリスト) 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忽ち batch を終了させる様にすると便利です.

error 函数は引数の評価と表示を行い,Maxima のトップレベルか,errcatch 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何処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で, 残りの対象は問題を起しているものです.

errormsg 函数は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もののリストが設定されており, 最初は文字列で, 残りは問題の対象です.

ttyintfun:lambda([],errormsg(),print(”)) で利用者中断文字 (^ u) をメッセージの再表示を行う様に設定します.

3.2 Maximaで函数の定義

3.2.1 函数定義

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.
Maximaの函数定義では演算子:=を用いる方法とdefine函数を用いる方法があります.

函数の定義方法

define (⟨ 函数名 ⟩,⟨,(⟨ 引数$_1$⟩,$\cdots$,⟨ 引数$_n$⟩)),⟨ 式 ⟩)
⟨ 函数名 ⟩(⟨ 引数$_1$⟩,$\cdots$,⟨ 引数$_n$⟩) := ⟨ 函数本体 ⟩

基本的にdefineで定義する函数はblock文,if文やdo文を含まない, 一つの式のみので構成された函数を定義します.
一般的な函数の定義は演算子:=を用いた函数定義で函数を定義します. ここで, 演算子:=を用いる函数定義で, その函数本体は式をコンマで区切ったものになります.
函数の定義を行うと, 大域変数 fuctions に定義した函数が追加されます.

大域変数 functions

変数名	初期値	概要
functions	[]	利用者定義函数のリスト

大域変数 functions は利用者が定義した全ての函数名を含むリストです. 利用者が函数定義を行うと自動的に functions のリストに函数名が追加されます. functions に追加されるのは演算子:=の左辺となります.

```
(%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, 立ち上げ時には空リストであった functions に演算子:=の左辺が登録される為,f(x) が追加されています.
函数定義の演算子:=を用いた函数で, 最も簡単なものは, 函数本体が 式$_1$, 式$_2$, $\cdots$, 式$_n$ の様に複数の式を並べたものです. この場合, 函数の返却値は単純に最後の式の結果だけです. 例えば, f(x):=(1-x,2+x,2*x) で函数 f を定義した場合, この函数の結果は最後の式 2*x を計算した値になります.
但し, この函数では割当が実行されていない為に, 変数の内容の書換えは生じません.
では, f(x):=(y:x,z:2+y,2*z) と定義した函数を実行した場合はどうなるでしょうか？

```
(%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
```

この様に, 先頭の式から順番に処理されて行き, 最後の式の結果だけが返却されています. 更に, 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事に注意して下さい.

この様に,Maxima 内部で用いた変数は, 通常は大域変数として扱われます. その為, 変数の値の書換が生じます. この事を避ける為に,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, 後述の block 文の中で定義可能です.

Maxima の lambda 式を用いると無名の函数が構築出来ます. この lambda 函数の構文は最初に函数の変数を宣言し, その後に函数本体が続きます. 函数本体は基本的に Maxima の式をコンマで区切ったものになります.

lambda 函数

lambda([⟨ 変数$_1$⟩, ··· , ⟨ 変数$_n$⟩], ⟨ 函数本体 ⟩)

構文自体は LISP の lambda 式と同様の構文となっています.

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 式を利用した函数 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 が lambda 函数内部のみで値を持つ事に注意して下さい.

Maxima では引数の個数が可変な函数も定義出来ます. この場合, 最後の引数を特別な引数リストとして割当てて, 函数を定義します. 尚, 引数が少ない場合, 安易な処理を行っていると少し問題になるかもしれません.

```
(%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, []]
```

単純に式を並べる方式では, 函数の返却値は最後に処理された式の値となります. そうではなく, 函数本体に含まれる式の返却値が必要な場合には,block 文と return 文を組合せます.

定義した函数の内容は函数 dispfun や fundef で参照する事が出来ます.

定義した函数の内容を表示する函数

dispfun(⟨ 函数名$_1$⟩, $\cdots$, ⟨ 函数名$_n$⟩)
dispfun(all)
fundef(⟨ 関数名 ⟩)

利用者定義の函数 ⟨ 函数名$_1$⟩, $\cdots$, ⟨ 函数名$_n$⟩ の内容を表示します. この函数の表示では, 函数を定義した時点での函数や定数等がそのまま表示されます.

尚, 引数に all を設定すると, 大域変数 functions と arrays で与えられる函数を全て表示します.

fundef 函数は ⟨ 関数名 ⟩ に対応する函数の定義を返します. fundef は dispfun に似ていますが,fundef では 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)
```

この例で示す様に,dispfun を実行すると結果は%t ラベルに表示されていますが, fundef の方は通常の%o ラベルに表示されています. それ以外で違いはありません.

利用者定義函数を削除する函数

remfunction (⟨ 函数$_1$⟩,⟨ 函数$_2$⟩,$\cdots$)
remfunction (all)

remfunction 函数は利用者定義函数を Maxima から削除します. 利用者定義函数は大域変数 functions にその名前が保存されており,remfunction は functions に含まれている函数名の削除を行います. 尚, 引数として all が与えられた場合, 大域変数 functions に含まれる全ての利用者定義の函数が削除されます.

3.2.2 マクロの定義

Maxima ではマクロの定義も出来ます. マクロの定義では演算子::=を用います. Maxima のマクロは最終的に LISP の S 式に展開し, それから評価が行われます.

複雑なマクロを生成する為に,block 文に似た buildq 函数を用います.

buildq 函数

buildq(⟨ 変数リスト ⟩,⟨ 式 ⟩)

buildq の引数は, 実際の式の代入が実行される迄,Maxima の解釈で勝手に変換される事を防ぐ必要があります. その為に単引用符' を用います.

buildq の応用に漸化式の計算があります. ここで, 幾つかの数列を定義してみましょう.

フィボナッチ数

フィボナッチ数は, 次の漸化式を満す数です.

フィボナッチ数

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+1} &= F_n + F_{n-1} \end{aligned}$$

では,Maxima の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;
(%o12)                                  0
(%o13)                                  1
(%o14)                                  1
(%i15) F[10];
(%o15)                                 55
(%i16) F[140];
(%o16)                    810559000960235041972064086O5
```

漸化式が単純な場合は buildq を使わずに,define(F[n],F[n-1]+F[n-2]) でも問題はありません.

ルジャンドルの多項式

ルジャンドルの多項式

$$
\begin{aligned}
P_0(z) &= 1 \\
P_1(z) &= z \\
P_{n+1}(z) &= (2n-1)zP_n - (n-1)P_{n-1}
\end{aligned}
$$

```
(%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;
(%o12)                                  1
(%o13)                                  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 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. 次の例では同じ計算で,expand を省略したものの計算結果を示しています.

```
(%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;
(%o19)                                 1
(%o20)                                 z
(%i21) Pz[4];
                                2
                           5 z (3 z  - 1)
                      7 z (-------------- - 2 z)         2
                                2                3 (3 z  - 1)
                      ------------------------- - ------------
                                 3                     2
(%o21)                ----------------------------------------
                                       4
```

この例で示す様に, 式の簡易化が行われていない為に,n=4 でも必要以上に複雑な結果となっています.

このマクロに関連する大域変数として,macroexpansion があります.

大域変数 macroexpansion

変数名	初期値	概要
macroexpansion	false	マクロ展開の制御で利用

大域変数 macroexpansion は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展開が実行されます. 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, 展開する手間が不要となるので, 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為には, それなりのメモリが必要となります.

- false
  マクロが呼出される度にマクロの展開を行います
- expand
  マクロ呼出で評価されると, 以後, マクロの展開をしなくても良い様に展開内容を記憶します.

マクロの呼出では通常 grind と display も呼出します. その分, 全ての展開を記憶する為に特別にメモリーを必要とします.

- displace
  最初の呼出でマクロ展開が実行されます.expand の場合と比較して, 幾らか少ない保存領域を必要とする割に処理速度は同程度ですが, マクロ呼出が記憶されない欠点を持っています. 展開は display か grind が呼出されていれば参照出来ます.

3.2.3 apply 函数

Maxima で重要な函数の一つに 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 ⟩, ⟨ リスト ⟩)

apply 函数は ⟨ 函数 ⟩ を ⟨ リスト ⟩ に適用した結果を与えます.
例えば, apply(min,[1,5,-10.2,4,3]) は -10.2 となります.
函数の呼出しで, それらの引数が評価されておらず, それらの評価を希望する場合も apply は便利です. 例えば, filespec がリスト [test,case] であれば, apply(closefile,filespec) は closefile(test,case) と同値です.
一般的に,apply で評価させる場合, 単引用符' を函数の先頭に置いて, 函数を名詞型として,apply に評価させます. 幾つかのアトムの変数が, とある函数と同じ名前を持っていれば, 函数としてではなく, その変数値が利用される事になりす. 何故なら,apply それ自身が第二の引数と同様に第一の引数をも評価するからです.

```
(%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 を函数として定義していますが, 同時に, 変数としての neko には-128 が束縛されています. その為,apply では単引用符' がなければ変数として評価されてしまいます.

3.2.4　最適化

Maxima は LISP で記述されています.Maxima のプログラムも含めた全てのデータは内部表現として LISP の S 式になっており,LISP がこの内部表現を解釈して処理を実行しています. その為, 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為, 処理速度向上が見込める事になります.

Maxima には利用者定義函数を LISP の函数に変換する函数を幾つか持っています.

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()

translate 函数は Maxima の処理言語で記述した利用者定義の函数を LISP の函数に変換する函数です.Maxima 言語で記述された函数は, 裏の LISP で解釈されて実行されます. これを LISP の函数にすれば, 解釈する手間は省ける分, 処理の高速化が望めます.

引数は ⟨ 函数$_1$⟩, $\cdots$, ⟨ 函数$_n$⟩ の様に利用者定義函数を直接指定する方法に加えて, 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事も出来ます.

変換される函数が内部で局所変数を利用する場合,mode_declare 函数を用いて, その局所変数の型を宣言する必要があります. この場合,block 文で局所変数を宣言した直後に mode_declare 函数による局所変数の型の宣言を入れます. この宣言の方法を以下に示しておきましょう.

mode_declare の使い方

f($x1, x2, \cdots$)　:=　block([局所変数$_1$, 局所変数$_2$, $\cdots$],
mode_declare(局所変数$_1$, 型$_1$, 局所変数$_2$, 型$_2$, $\cdots$),
⟨ 函数本体 ⟩)

この mode_declare 函数の詳細に関しては,1.5.4 を参照して下さい.

次に, 函数を translate 函数で変換すると, 大域変数 savedef が false の場合, 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て, 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.

当然の事ですが, 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. 更に,translate 函数は変換する函数内部の式が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.

その為, 大域変数の simp を false に設定して変換式の簡易化を禁じる事をしてはいけません.

注意すべき事に,translate 函数を用いて LISP の函数に変換しても,Maxima と LISP の整合性の問題から以前と同じ動作をする保証はありません.

translate_file 函数は 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 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, オプションの第二の引数は生成すべき LISP ファイル名です. 第二の引数は第一引数に,trisp のデフォルト値の tr_output_file の値を第二ファイル名のデフォルト値として与えます. 例えば, 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.

更に, 生成されるものには translate 函数が出力した様々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. 第二ファイル名は常に 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為の情報が含まれています.

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, 名前も長いものが多い為, 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.

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 函数は指定した 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. 尚,compile 函数は函数名リストを返します. 尚, 引数に functions や all を指定すると利用者定義函数を全てコンパイルします.

これに対し,compile_file 函数は指定したファイルのコンパイルを行います. 先ず, 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, これを LISP 函数に translate 函数で変換し, その結果を compile 函数でコンパイルします. 変換とコンパイルに成功すると, 今度は結果を Maxima に読込みます.

compile_file は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, 元の Maxima プログラムファイル, LISP への変換ファイル, 変換に関する註釈ファイルと compile でコンパイルされたプログラムのファイルです. 尚, コンパイルに失敗すると, 返却されるリストの第四成分は false になります.

compfile 函数は ⟨ 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))))
```

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	初期値	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 が 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 が 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. その為, 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. 又,dispfun 函数で函数の定義が表示可能で, 函数の編集も出来ます.

false の場合は,functions に割当てた利用者函数名リストから該当函数名を削除します.

大域変数 translate が 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. 尚,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事に注意して下さい.

変数が mode_declare された CRE 表現の場合,rat 函数を一つ以上の引数で用いたり,ratvars 函数を使ってはなりません. 又,prederror:false は変換しません.

大域変数 transrun が 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 には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	初期値	概要
define_variable	[]	
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	機械コードへの翻訳を制御
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 が 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 が true の場合, 配列の変換を行います. 大域変数 tr_array_as_ref は,translate_fast_arrays が false の場合のみに 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シンボルとして配列が現れます.

大域変数 tr_function_call_default は 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事を保証します. 例えば, 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ったとすると, 警告を出して,apply(f,[x]) の事として函数を変換します.

尚, デフォルト設定で何等の警告メッセージが無ければ,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事を意味します.

tr_numer が 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.

tr_optimize_max_loop は考えられる形式で 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為で, 非中断の最適化属性です.

tr_semicompile が 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 コンパイラで機械コードに翻訳されたものにはなりません.

tr_warn_fexpr は, 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.

tr_warn_meval は函数 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.

tr_warn_mode は, 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.

tr_warn_undeclared が未宣言の変数に関する警告を tty に送るべき時を決めます.

tr_warn_undefined_variable は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	初期値	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

tr_bound_function_applyp が true の場合, 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. 例えば, g(f,x):=f(x+1) の様な場合です.

tr_file_tty_messagesp は,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 tty に送るかどうかを決めます. false であれば, ファイルの translate 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, メッセージは tty に送られ, UNLISP ファイルにも挿入されます.

tr_float_can_branch_complex は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,x が float(浮動小数点型) であったとしても,acos(x) は any 型となります.false にしている時は,x が float 型で, その時に限って,acos(x) は float 型となります.

tr_warn_bad_function_calls は, 変換時に不適切な宣言が行われた為に, 函数の呼出しが生じた場合に警告します.

3.2.5　函数定義に関連する函数

funmake 函数

funmake(⟨ 関数 ⟩, [⟨ 引数$_1$⟩, $\cdots$, ⟨ 引数$_n$⟩])

funmake 函数は ⟨ 関数 ⟩ で指定した函数を呼出して評価を行う事はしません. 単純に,⟨ 関数 ⟩ (⟨ 引数$_1$⟩, $\cdots$, ⟨ 引数$_n$⟩) を返すだけです.

```
(%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)
```

関数に指定したアトムに値が束縛されている場合, そのアトムは内部で評価される為, エラーになります. この場合は, 単引用符' を先頭に付けて名詞型とすれば問題ありません.

tr_warnings_get 函数は translate 函数による変換中に translate 函数が出力する警告のリストを表示します.

declare_translated 函数

declare_translated(⟨ 函数$_1$⟩, $\cdots$, ⟨ 函数$_n$⟩)

declare_translated 函数は, 引数の函数が既に変換されている事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, 或いは compile 函数でコンパイル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, 又, 未定義のものであるかを知る事は translate 函数にとって重要な事です.

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.

局所変数を定義する函数

local(⟨ 局所変数$_1$⟩, $\cdots$, ⟨ 局所変数$_n$⟩)

local 函数は, この函数が利用される文中で, ⟨ 局所変数$_1$⟩, $\cdots$, ⟨ 局所変数$_n$⟩ を全ての属性に対して局所的なものにします.local は block 文, 函数定義の本体,lambda 式, 又は ev 函数でのみ一度だけ使えます. この local 函数は文脈からも独立しています.

3.3　データ入出力について

3.3.1　データの入出力

この節では Maxima のデータ入力と出力について述べます. Maxima は Common LISP で記述されている為, 基本的に入出力は LISP の入出力函数を用いたものです. 特に Maxima は表示されている形式と内部形式が別物の為, 結果や式を保存したり, 逆に保存したものを読込む際には, 注意が必要になります.

ファイルに画面と同じ出力を行いたければ,writefile を用います. この writefile は LISP の dribble 函数を用いたもので, 入力と出力をそのまま指定したファイルに保存します. 但し,writefile で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 と appendfile 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.

但し, これらのファイルは実質的に記録ファイルであって,Maxima でそのまま再利用は出来ません. 再利用可能なファイルを生成するのは,save と stringout です. ここで,save は Maxima の内部表現を保存する函数で,load や loadfile を用いて Maxima に読込みます. これに対して,stringout や grind は内部形式ではなく, Maxima の入力に対応する通常の形式でデータの保存を行います.

Maxima で C の scan 命令に似たものに,read と readonly 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ます. ここで, 行末には; か$を付けます. すると,read の場合は式を Maxima で評価し, readonly の場合は評価せずにそのまま受け取ります.

この様に, 単純なファイル操作に限定されるとは言え, 必要なものは一応揃っており, 足りない部分は LISP で補う事になります.

ファイル処理に関連する大域変数

変数名	初期値	概要
batchkill	false	以前のバッチファイルの影響を無効にする
batcount	0	式番号を記録
file_search_Maxima		.mac ファイル読込ディレクトリ
file_search_LISP		.lisp ファイル読込ディレクトリ
file_search_demo		.dem ファイル読込ディレクトリ
file_string_print	true	ファイル名の表示を決める
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

batchkill が true の場合, バッチファイルを読み込む際に kill(all) と reset() が自動的に実行されるので, 以前のバッチファイルの影響が全て無効になります. batchkill が他のアトムであれば,batchkill の値で kill が実行されます.

batcount にはファイルからのバッチ処理された最後の式の番号を設定します. batcon(batchcount-1) は以前の処理から, 最新の batch 処理された式の処理結果を保存します.

file_search_maxima,file_search_lisp と file_search_demo は load や他の函数で, ファイルの読込を行う際に検索されるディレクトリのリストです.

file_search_maxima は Maxima のプログラム (末尾が,mac と mc) 向け, file_search_lisp は LISP のプログラム (末尾が,fas,lisp,lsp) 向け, file_search_demo が LISP のプログラム (末尾が,dem,dm1,dm2, dm3,dmt) 向けとなっており, 各々が Maxima のリスト形式となっています. 尚, これらの値は, src/init-cl.lisp で設定されています.

file_string_print は true であれば, ファイル名は文字列, false であれば, リストとして出力されます.

loadprint は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. loadprint が取る値は,true,loadfile,autoload,false の四種類あり, 各々, 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 命令が用いられた時のみに表示されます.
- autoload の場合は, ファイルが自動的に読込まれた時のみに表示されます,
- false の場合, メッセージが決して表示されません.

packagefile は save や translate を用いてパッケージ (ファイル) を作成する時に, packagefile:true と設定すると, ファイルが読み込まれる時点で必要な場所を除いた情報が Maxima の大域変数, 例えば values,functions に追加される事が避けられます.

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, 利用者の側では得られません. これは名前の衝突の問題を解決するものでは無い事に注意して下さい. 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事に注意して下さい. 尚, 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.

3.3.2 ファイル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 ⟩)
batcon(⟨ 引数 ⟩)

batch 函数は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. ファイルはパスを含まない場合,file_search_Maxima に含まれるディレクトリ上を検索し, 存在した場合には読込みと実行をします.

ファイルの内容は基本的に Maxima での入力行と同じもので, 行末には; か$を置きます. 又,%と %th を用いて入力と出力を指定する事も出来ます. 尚, 空行,Tab や改行コードは無視されます.

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, 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. 但し, 利用者はどの時点でも Cntrl-g を押せば, この処理を止められます.

batchload 函数は指定されたファイルのバッチ処理を行います.batch との違いは, batchload ではファイルに記述された式の入力や出力表示等を行わない事です.

batcon 函数は中断されたファイルのバッチ処理を再開します.

読込を行う函数

load(⟨ ファイル名 ⟩)
loadfile(⟨ ファイル名 ⟩)
read(⟨ 文字列$_1$, $\cdots$⟩)
readonly (⟨ 文字列$_1$⟩ ,$\cdots$)

load 函数は文字列や, 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 file_search_Maxima,file_search_lisp や file_search_demo といった大域変数に保存されているディレクトリを検索し, 指定された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事を見付けると,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 処理を実行する事を意味しています).

他のファイルの読込を行う Maxima 命令に,loadfile,batch と demo があります.loadfile は save で書込んだファイルに対して動作し,batch と demo は strignout で書込まれたり, テキストエディタで命令のリストとして生成されたファイル向けです.

loadfile 函数は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 に戻す事に使えます.

ここで, パスの指定はオペレーティングシステムのパスの指定に方法に従います. 例えば,unix の場合は/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" となります.

尚.save 函数で保存したファイルを loadfile で読込むと,Maxima は初期化されるので注意が必要です.

read 函数は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 と表示された後に、`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 に評価されて, 結局,a に `2*x` が割当てられています.

readonly 函数は引数を全て表示し, それから式を読込みます. 基本的には read と同様ですが,read と違うのは, 読込んだ式を評価しない事です.

```
(%i46) a:readonly("mikeneko");
mikeneko
diff(x^2+1,x);
                                      2
(%o46)                          diff(x  + 1, x)
```

stringout 函数

stringout(⟨ ファイル名 ⟩ ,⟨ 式$_1$⟩ , ⟨ 式$_2$⟩ ,···)
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)

stringout 函数は指定したファイルに Maxima が読込める書式で出力します. 直接,⟨ 式$_1$⟩ ,⟨ 式$_2$⟩,··· と式を並べると, 各式が順番にファイルに書込まれます.

引数に,[⟨m⟩ ,⟨n⟩] と指定すると入力行の m 行から n 行がファイルの書込まれます.

これらに対し,input を指定すると入力行全てが書込まれます. functions を指定すると大域変数 functions に記載された利用者定義の関数が全て保存されます.

同様に 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書込まれます.

尚, この stringout 函数は writefile を実行中に利用する事も可能です.

大域変数の grind が true であれば,stringout は文字列では無く,grind と同じ書式で出力します.

書込みを行う函数

appendfile(⟨ ファイル名 ⟩)
writefile(⟨ ファイル名 ⟩)
with_stdout(⟨ ファイル名 ⟩,⟨ 式$_1$⟩,···, ⟨ 式$_n$⟩)
closefile()

appendfile 函数は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,writefile 函数は上書きをしてしまいますが,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事が違います. 尚,writefile 函数と同様に指定したファイルは closefile() 函数で閉じます.

writefile 函数は書込み用に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 への入力と出力処理は全て指定したファイルに記録されます. その為, このファイルをそのまま Maxima に再度読込ませたりする事は出来ません.

ファイル名の指定は文字列で行います. 文字列ではなく,ABCD の様に二重引用符無しで指定すると,Maxima はアトムの内部表現で用いる$を頭に付けたファイル名, 即ち, この例では$ABCD でファイルを生成します. 尚, この writefile の実体は LISP の dribble 函数です.

ファイルを閉じる場合は closefile() を用います.

以下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
```

この様に,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 函数は, 指定されたファイルを開き,⟨ 式$_1$⟩,···, ⟨ 式$_n$⟩ の評価と書込みを行います. 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ファイルに送られ, 端末側には常に false が返されます.

closefile 函数は closefile() で appendfile や writefile 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

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 函数は指定したファイルに, 指定した式や関数等の値を書込みます. 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.

⟨ 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,functions,labels 等を指定する事も可能です. values,functions で利用者が設定した変数値や定義関数を全て保存します. 又,labels を指定すると, 入出力行の内容が全て保存します.

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 の様々な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大きなものとなるので注意が必要になります. save 函数の返却値は保存先のファイル名となります.

```
(%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
```

save 函数で保存したファイルは loadfile 函数で Maxima に再び読込む事が出来ます. 但し,loadfile を実行すると, 以前の Maxima 自体を初期化し,save 函数を実行した時点にまで戻してしまう効果があるので, 注意が必要になります.

以下の例では最初に loadfile 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事と, 二度目に loadfile を実行するとラベルが (%i8) から (%i5) に戻っている事に注意して下さい.

```
(%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)
```

尚,save ファイルの内容は,LISP の S 式そのものとなります. 要するに,LISP 上で動く Maxima の為のデータファイルになります. 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, その後には Maxima の諸設定が保存されます. その為, 以下に示す例は 1+2+3 から 4 行の入力だけでしたが,save 函数で保存したファイル (test) は 256 行に及びます. これは内部形式で記述するとどうしても長くなる側面もありますが, 実際は, 入出力以外に様々な設定 (大域変数の値等) も保存されている為です.

```
;;; -*- 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

その為,Maxima 内部でデータがどの様に処理されているかを見る場合には重宝するでしょう. 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, 例えば,stringout や grind を用いた方が総合的な使い勝手自体は良いでしょう.

その他の函数

filename_merge(⟨ 文字列$_1$⟩ ,⟨ 文字列$_2$⟩)
file_search(⟨ ファイル名 ⟩)
file_type(⟨ ファイル名 ⟩)

filename_merge 函数は二つの ⟨ 文字列$_1$⟩ と ⟨ 文字列$_2$⟩ の結合を行います. 内部的には, 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様にしか見えません.

基本的には Maxima の各種命令でファイルの検索を行う際にパス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 函数は指定したファイルを file_search_lisp,file_search_maxima と file_search_demo に含まれるディレクトリ上で検索し, 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.

file_type 函数は指定したファイルの属性を返します. 但し, ファイル名の末尾で判断する函数なので, 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 の処理言語で記述されたファイルになります.

第4章　Maximaのシステム回りの函数

この章で解説する事:

- システムの初期化
- 処理の中断
- ラベルの参照
- システムの状態を調べる
- 外部プログラムの起動
- 式の変換を行う函数
- システムに関連する大域変数
- Maxima の終了

4.1 システムの初期化

4.1.1 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-init.mac を自動的に読込みます.

maxima-init.mac 内部には Maxima の函数や大域変数の設定が記述可能です. 更に,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.

以下に非常に簡単な例を示します.

```
/*-*-MAXIMA-*-*/
showtime:all;
put(surfg, d_chain_bisection,root_finder)$
put(surfg, 0.0000000001,epsilon)$
put(surfg, 20000,iterations)$
put(surfg, 500,width)$
put(surfg, 500,height)$
put(surf, yes,do_background)$
put(surf, 5,background_red)$
put(surf, 5,background_green)$
put(surf, 5,background_blue)$
put(surf, 0.14,rot_x)$
put(surf, -0.3,rot_y)$
setup_autoload("surfplot.mc",surfplot)$
load("fox.mc")$
```

Maxima が読込むファイルにて, 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事を示しています. その後に, 大域変数の設定や, 属性の設定等を行っています.

ここで,setup_autoload 函数や load 函数を用いたファイルの読込について簡単に解説しておきましょう. この setup_autoload 函数の構文を以下に示しておきます.

setup_autoload 函数

setup_autoload ($\langle$ ファイル $\rangle, \langle$ 函数$_1\rangle, \cdots, \langle$ 函数$_n\rangle$)

この setup_autoload 函数は, 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. ここで,setup_autoload 函数は引数として配列函数を扱えない事に注意して下さい.

この例では, ファイル surfplot.mc に含まれている surfplot 函数が呼出された時に, 未定義であれば, ファイル surfplot.mc を読込む設定となっています.

load 函数の場合は予め指定されたファイルの内容を全て Maxima に読込みます. この場合, ファイルの定義, 未定義とは無関係に読込を実行します.

この maxima-init.mac ファイルを利用する事で,Maxima の起動時の環境が容易に変更出来ます.

4.1.2 セッションの初期化

Maxima では,reset 函数を用いて Maxima 起動時の初期状態に戻す事や kill 函数を用いて, 処理で蓄えたラベルや大域変数等を破棄する事が可能です.

reset 函数

reset()

この rest 函数は引数無しで, reset(); と入力します.reset 函数を用いると, 入出力の基数は 10 に, 係数環は $\mathbb{Z}$ に, ラベルと大域変数 fpprec は Maxima の初期状態に戻されます. 但し, 完全に Maxima を起動した状態に戻る訳ではなく, 利用者が定義した大域変数とその値, 以前のラベルに保存されている値を削除しません. 尚, ラベルのカウンタが初期化されている為, ラベルに保存されている値は, 処理を進める内に上書きされて消去されます.

kill 函数

kill(⟨ 引数$_1$⟩, ⟨ 引数$_2$⟩, $\cdots$)
kill(⟨ 整数 ⟩)
kill([⟨ 整数$_1$⟩,⟨ 整数$_2$⟩])
kill(all)
kill(allbut(⟨ 引数$_1$⟩, ⟨ 引数$_2$⟩, $\cdots$))

kill 函数は reset 函数とは異なり, 必ず引数を取ります.kill 函数は引数によって削除する対象が異なります.

- ⟨ 引数$_i$⟩ が変数 (単配列要素を含みます), 函数, 又は配列の場合, 指定された項目は, その属性の全てと一緒に Maxima から消去されます.
- ⟨ 引数$_i$⟩ が labels であれば, 古くなった全ての入出力行, 中間行が消去されます.
- ⟨ 引数$_i$⟩ が clabels であれば, 入力行だけが消去されます.
- ⟨ 引数$_i$⟩ が elabels であれば, 中間行のみが消去されます.
- ⟨ 引数$_i$⟩ が dlabels であれば, 出力行のみが消去されます.
- ⟨ 引数$_i$⟩ が任意の他の情報のリスト, 即ち, 大域変数 infolists に含まれる名前であれば, そのクラスとその属性全ての項目が削除されます.

infolists に含まれる values や variables に対応して,kill(values) や kill(variables) で同じ式を指定するラベルが消去される迄, メモリの占有領域を解放しない事に注意して下さい. 例えば, 大きな式

が%i10 行の変数 x に対して割り当てられていた場合, 占有された保存領域を解放する為には,kill(x) と kill(%o10) も実行しなければなりません.

kill(all) で入出力行, 中間全てを削除します. これに対し,kill(allbut(···) は,allbut で指定したものを除外して kill(all) を実行します. 但し,allbut で指定する引数は,functions 等の大域変数 infolists に含まれる項目ではなく, より具体的なものです.

kill 函数は与えられた引数から全ての属性を削除する為,kill(values) は大域変数 values に割当てられたリストの全ての項目に関連する属性を削除しますが, それに対して remove 函数群 (remvalue,remfunction,remarray,remrule) は指定した属性のみを削除します. ここで,remove 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 函数は指定した項目が存在しない場合でも, 常に done を出力します.

collapse 函数

collapse (⟨ 式 ⟩)
collapse ([⟨ 式$_1$⟩, ··· , ⟨ 式$_n$⟩])
collapse (*listarray*('⟨ 配列 ⟩))

全ての共通 (つまり, 等しい) 部分式を共有する (つまり, 同じセルを用いる) 事で引数を潰し, 領域を節約します. 因に,collapse 函数は optimize 函数でも用いられています. save で保存したファイルを読込んだ後に collapse 函数を呼出すと良いでしょう. 又, 配列 a に対し,collapse(listarray('a)) とする事で, 配列の成分を潰す事も出来ます.

4.2 処理の中断

4.2.1 中断の制御文字

Maxima の計算が異常に長い場合, 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 ^ 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 レベルに戻るので, 通常は Ctrl+C を用います.

```
(%i11) factor(213749812794387098237 4);
Maxima encountered a Lisp error:

 EXT:GC: User break

Automatically continuing.
To reenable the Lisp debugger set *debugger-hook* to nil.
(%i12)
```

この例では,Ctrl+C で因数分解を中断させています.

4.2.2 break 函数による中断

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も可能です.

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 に設定し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 から抜ける為には 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後に Maxima-break に入っています.Maxima-break で出力されるプロンプトはデフォルトで_となっています.Maxima-break から抜ける為には,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.

4.3 ラベルの参照

Maxima は入力を入力ラベル%i に, 計算結果を出力ラベル%o に各々保存しています. これらの値の参照は, 入力の場合は%i⟨ ラベル番号 ⟩ で, 出力の場合は%o⟨ ラベル番号 ⟩ で行います.

最新の結果は%で参照出来ます. これに対し, 最新の入力は_で参照します.

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 は%th(6) の様に用い,6 個前の結果を参照します. 更に,%i7 や%o8 とすれば,(%i7) 行で入力した式や,(%o8) に表示された結果を参照する事が出来ます. 但し,_はその様な使い方が出来ません.

入力と出力ラベルは kill(labels) で全て削除する事が可能です. これを実行すると, 入力と出力ラベルに割当てられた値は消去され, 各ラベルのカウンタも 1 に戻されます, その為, 入力は (%i1), 出力も (%o1) から開始する事になります.

```
(%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, 様々な処理を行い, それらを%や_で確認し, 最後に kill(labels) でラベル (%i や%o) の内容を全て消去しています. ラベルの消去を行った為に, kill(labels) を入力した (%i109) から,(%o1) を経て (%i1) に初期化されている事に注目して下さい.

この他にも, ラベルの処理に関連する大域変数を Maxima は持っています.

ラベルに関連する大域変数

変数名	初期値	概要
%		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
nolabels	false	入力値と計算結果をラベルへの束縛を制御

大域変数%%は大域変数%と同じ意味を持ちます. 但し, 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です.

```
(%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
```

この例は大域変数%と大域変数%%の違いを説明するものです. 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る事はありません. これに対し, 大域変数%%は block 文内部の局所変数で, その値も block 文内部のみで更新されます. その為,block 文が終了すると, 値は消去されます.

大域変数 inchar は Maxima の入力行ラベルで用いられる文字です. デフォルトでは入力行ラベルは (%i1) の様に大域変数 inchar で指定した%i の後に番号が続きます.

大域変数 outchar は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 には中間表示に於ける式の前に置かれる文字を指定します.

大域変数 linenum には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 は 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(slow) 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. 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.

大域変数 nolabels が true の場合, 入力値と計算結果をラベルに束縛しません. 即ち,%等で結果や入力を参照出来なくします. この様にする事で,batch 処理の空き領域を増す為に,kill(labels) を実行しなくて済みます.

Maxima にはラベルの処理に関連する函数として,%th0 函数と labels 函数を持っています.

ラベル処理に関連する函数

%th(⟨ 正整数 ⟩)
labels(⟨ シンボル ⟩)

%th 函数は ⟨ 正整数 ⟩ 番前の計算結果を取出します. 即ち,%th$\langle i \rangle$ を含む式の入力ラベルが%$\langle j \rangle$ であれば,%th$\langle i \rangle$ は%i$\langle j - i \rangle$ の結果, 即ち,%o$\langle j - i \rangle$ の値となります.

この%th は batch ファイル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 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事になるからです.

label 函数は, シンボルを引数として取り, そのシンボルに対応するリストを返します. ここで, シンボルとしては入力ラベル (デフォルトで%i), 出力ラベル (%o) 等のラベルが一般的ですが,labels 函数は引数として任意のシンボルを取ります. 入出力等のラベルを設定する大域変数 inchar,outchar や linechar を再設定すれば,labels 函数はそのラベルに対応するリストを返しますが, ラベルの最初の文字は labels に与えた引数の最初の文字に適合します.

4.4　結果の表示

Maxima では入力行に; を付けると,Maxima が評価した値が表示されます. 数値の四則演算はデフォルトの場合, 直ちに処理されて結果が表示されます. 更に Maxima は結果を二次元的に表示するのがデフォルトとなっています.

```
(%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良いものですが, 式が長くなると非常に判り難いものになります. そこで, 表示を 1 行で済む様に指定が行える大域変数 display2d があります. この変数の値がデフォルトの true であれば, 二次元的な表示を行い,false を指定すると, 結果を入力可能な書式で表示します.

```
(%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為, 積分記号の様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.

4.4.1 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	初期値	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
display_format_internal	false	内部表現に対応した表示切替
%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
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
ttyoff	false	出力の停止制御

大域変数 ibase は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. 尚, 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 は絶対値を描く際に用いる文字を指定します. 尚, 絶対値は精々一行の高さしかありません.

大域変数 display2d が false であれば, 結果表示が二次元的書式ではなく, 一行に収まる様に表示します. 長い式や複雑な式, 表示に余裕が無い場合は有効です.

大域変数 display_format_internal が 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. 但し, 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るとも言えます.

内部書式との比較

入力	part	inpart
$a-b;$	$a-b$	$a+(-1)b$
$a/b;$	$\frac{a}{b}$	ab^{-1}
$sqrt(x);$	$sqrt(x)$	$x^{1/2}$
$x*4/3;$	$\frac{4x}{3}$	$\frac{4}{3}x$

%edispflag は自然底%e の冪表示を定める大域変数です. デフォルトの false の場合, %e の負の冪は負の冪のまま冪表示されます. 例えば,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 の冪の商の形式で表示されます. 即ち,%eˆ (-x) は 1/%eˆ x の形式で表示されます.

```
(%i2) exp(-x);
                                        - x
(%o2)                                  %e
(%i3) %edispflag:true;
(%o3)                                  true
(%i4) exp(-x);
                                         1
(%o4)                                   ---
                                          x
                                        %e
```

大域変数 exptdispflag が true であれば,Maxima は負の冪を持った項を分数式で表示します. 例えば,xˆ (-1) は 1/x となります.

大域変数 pfeformat が 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. 例えば, 入力が b/4 であれば,1/4*b と表示されます. 但し,a/b の様に,a,b の両方が不定元であれば, 通常の表示になります.

大域変数 stardisp が false の場合, 可換積演算子は出力では省略されています. true であれば, 可換積演算子を表示します.

大域変数 linel は一行に表示される文字数を設定します. 変更は何時でも可能です.

大域変数 ttyoff が 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. 但し, エラー表示は行います. 尚,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.

4.4.2 式の表示を行う函数

式の表示を行う函数

display(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, $\cdots$)
disp(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, $\cdots$)
ldisplay(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, $\cdots$
ldisp(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, $\cdots$
print(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, $\cdots$)
grind(⟨ 式 ⟩)

display 函数式を表示します. 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, アトム, 添字された変数や函数呼出しになります.

disp 函数は display 函数にに似ていますが引数の値のみを表示します.

ldisplay 函数と ldisp 函数は各々,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が異なります.

```
(%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 函数は,⟨ 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$_i$⟩ に含まれるアトムや函数の前に単引用符' が置かれていたり, 文字列 (全体を二重引用符で括ったもの) の場合

は, 評価を行わずに, そのまま表示を行います.

grind 函数は,⟨ 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て表示する函数

- dispterms (⟨ 式 ⟩)
- reveal (⟨ 式 ⟩, ⟨ 深度 ⟩)
- tcl_output (⟨ リスト ⟩, ⟨ 添字 ⟩, ⟨ 飛幅 ⟩)
- tcl_output (⟨ リスト ⟩, ⟨ 添字 ⟩)

dispterms 函数は与式を内部表現に合せて, 第一層の部分式を順次表示します.

reveal 函数は ⟨ 整数 ⟩ で指定された各々の成分の長さで ⟨ 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.

```
(%i10) aa:integrate(1/(x^3+2),x)$
(%i11) aa;
                                                 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 函数は,⟨ 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, 飛幅の初期値は 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, 飛幅から外れた全ての要素が表示されます.

4.4.3 後戻し表示を行う函数

xmaxima の様なグラフィカルな GUI 端末を持たなかった時代では特に重宝されたと思われる機能に, 以前処理した結果の表示を再度表示する機能があります. これは playback 函数を用いて指定した入力とその処理結果を再表示させるもので, その際に再計算を行うものではありません.

後戻し表示の函数 playback

playback (⟨ 整数 ⟩)
playback ([⟨ 整数$_1$⟩, ⟨ 整数$_2$⟩])
playback ([⟨ 整数 ⟩])
playback ()
playback(input)
playback(slow)
playback(time)
playback(grind)

playback 函数は入力と出力行の後戻し表示を行います. 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 を各々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$_2$⟩ 迄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.

その他の引数に,input,slow,time,grind があります.

先ず,input の場合は入力行を再表示行します.

slow の場合は 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. 又,Enter キー以外のキーが入力されると playback 函数を終了します. 尚, 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.

time の場合には計算時間が式と同様に表示されます.

更に,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.

string であれば, 全ての入力行の再表示を文字列として返します.

grind の場合は,Maxima で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.

尚,playback 函数による行の再表示は, playback([2,5],10,time,grind) の様に複数の引数を組合せても構いません.

4.4.4 エラー表示

エラー表示に関連する大域変数

変数名	初期値	概要
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_size はエラーメッセージの大きさを制御します. 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 はエラーメッセージで,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 error_syms リストから取られて初期値は 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後, 例えば, "the function foo doesn't like errexp1 as input."であれば, errexp1; と利用者が入力すると, その式を見る事が出来ます.

error_syms は, 必要なら別の文字列を設定しても構いません.

4.5 記号?

Maxima の記号?は変った性質を持っています. これは函数としては, 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, 特殊な記号としては, 裏の LISP に?の直後の文字列を流し込む働きを行います.

?函数

?	? ⟨ 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 ⟩	S 式を評価

函数?は Maxima で二つの意味を持ちます. 一つは, ヘルプの表示で用います. この場合,?との間に空行や 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.

```
(%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?の意味は, 引数の間に空白を空けずに利用した場合,?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. この場合の?引数は,LISP に評価させる函数を記述しますが,LISP の S 式ではなく.Maxima の函数風に表記しなければなりません. 更に, その LISP の函数に与える引数も,Maxima の表の表記にする必要があります. 例えば,Maxima の変数 x は, 内部では $x と表記されています. その x に割当てられた式の car を取りたい場合, ?car(x); と入力し, ?(car $x) とはしません.LISP の函数で式の評価をする場合,:lisp 函数もありますが, こちらは LISP の S 式を引数に取りますが, その返却値は函数?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)
```

4.6 システムの状態を調べる

4.6.1 大域変数 infolists

大域変数 infolists

変数名	概要
infolists	Maxima の情報リストの全ての名前を含むリスト

Maxima には大域変数 infolists があります. この infolists には以下の大域変数が含まれています.

- labels
  代入された全ての i,o と t 等のラベル付けられた行
- values
  代入された全てのアトム. 利用者変数で Maxima のものではない.
- オプションや大域変数 (:,::, や関数による割当が設定されたもの)
- functions
  全ての利用者定義関数 ($f(x) := \cdots$ で設定されたもの)
- arrays
  宣言されたものと非宣言の配列 (:,::, や::=で設定されたもの)
- macros
  利用者定義された任意のマクロ
- myoptions
  利用者によって再設定された全てのオプション (それらがデフォルト値に再設定されていようが無関係に)
- rules
  利用者定義の並び照合と簡易化の規則 (tellsimp, tellsimpafter,defmatch や defrule で設定されたもの)
- aliases
  利用者定義の別名を持つアトム (alias,ordergreat, orderless 関数や declare で名詞型 (noun) として宣言されたもの)
- dependencies
  関数の依存関係を持つアトム (depends や gradef 関数で設定されたもの)
- gradefs
  利用者定義の微分を持つ関数 (gradef 関数で設定されたもの)

- props
  上で言及された他の任意の属性, 例えば,atvalues,matchdeclares 等の declare 関数で指定された属性を持つアトム.

- let_rule_packages
  全ての利用者定義のリストで, 規則パッケージに特別な default_let_rule_package を追加するもの. (default_let_rule_package は規則パッケージの名前で, 利用者によって明示的に設定されなかった時に用いられます)

この infolists に含まれる項目を参照して, 情報の蒐集やデータや属性の変更や削除を行う函数が多くあります. 又, 利用者もこの変数名を直接入力する事で,Maxima に含まれる対象を調べる事が出来ます.

4.6.2 status 函数

status 函数

status (⟨ 引数 ⟩)

status 函数は利用中の Maxima に関する様々な情報を与えられた ⟨ 引数 ⟩ に従って返却します.

- time
  計算で消費した時間.

- day
  その週の日

- date
  年, 月, 日々のリスト

- daytime
  時間, 分, 秒のリスト

- runtime
  現時点の Maxima で集計された cpu 時間に原子”milliseconds”をかけたもの.

- realtime
  利用者が Maxima を立ち上げてから経過した実際の時間 (秒単位).

- gctime
  計算で費やしたゴミ集め (garbage collection) 時間

- totalgctime
  Maxima でゴミ集めに費やした総時間.

- freecore
  アドレス空間を使い果す前に拡張可能な Maxima のコアブロックの数 (1 ブロックは 1024 語).250*blocks(mc にて得られる最大値) から値を減じ, Maxima が使い上げた中心核のブロック数を知せます (固定ファイル無しの Maxima は大体 191 ブロックで開始します).

- feature
  システム機能のリストを返す. 現在,mc に対するリストを以下に示します.
  Maxima,noldmsg,maclisp,pdp10,bignum,fasload,hunk,funarg, roman,newio,sfa,paging,mc と its.
  これらの任意の機能は status(feature,···) の第二引数として与えても構いません.

4.6.3　room 函数

room 函数

room ()
room (true)
room (false)

room 函数は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.

4.6.4　処理時間に関連する函数

処理時間表示の函数

time ($\langle \%o_1 \rangle, \langle \%o_2 \rangle, \cdots$)

time 函数は $\langle \%o_i \rangle$ の計算で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. 尚, 大域変数数 showtime を true にすると, 各%o-行 (結果表示行) で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	初期値	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 は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 が 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. showtime:all; とすれば,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. 尚,time=には計算時間のみが含まれ, 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事が難しい為, 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. その為に, 稀に time=よりも gctime の方が大きくなる事があります.

4.7 外部プログラムの起動

Maxima では Maxima 外部のプログラムを起動する事が可能です. この場合,system 函数を用います.

system 函数

system(⟨ 命令 ⟩)

Maxima 外部の命令を実行します. オプションを持つ命令を実行したければ, 命令全体を文字列として system 函数に引渡します. 例えば, ls -a を実行したければ, system(”ls -a”); と入力します.

尚,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. 例えば,surf を用いたければ, system(”surf&”); の様に&を使います. もし,&が無ければ,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為, Maxima による処理は停止したままです.

4.8　式の変換を行う函数

Maxima では出力式を TeX,FORTRAN の書式に変換出来ます.

tex 函数

tex(⟨ 式 ⟩)
tex(⟨ 式 ⟩,⟨ ファイル名 ⟩)
tex(⟨ ラベル行 ⟩,⟨ ファイル名 ⟩)

tex 函数は, 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. 尚, 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. 尚, 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.

fortran 函数

fortran(⟨ 式 ⟩)

fortran 函数は与式を FORTRAN の構文に変換します. 一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足りなくなると, :,;,<,=,>,?,@が順番で振られ, それから再度 1 から 9 を繰返します.

尚,fortran 函数は Maxima の式を FORTRAN の書式に合せます. 例えば, 式中の演算子^は**で置換し, 複素数 a + b%i は (a,b) に置換えます.

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様に変換します.

尚, 与式に fortran 函数で解釈出来ないものがあった場合,grind 函数を用いて式の表示を行います.

大域変数 fortindent は左側の空白を制御します. デフォルトの 0 で, 通常の空白行 (6 空白行) となります.

大域変数 fortspaces が true であれば,fortran は 80 列で出力を行います. 尚,fortran 函数は常に返却値は done です.

```
(%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
```

便利な函数

alias(⟨ 新名称$_1$⟩, ⟨ 旧名称$_1$⟩, ⟨ 新名称$_2$⟩, ⟨ 旧名称⟩, ···)
apropos(⟨ 文字列 ⟩)

alias 函数は (利用者, 又はシステム) 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.

apropos 函数は ⟨ 文字列 ⟩ を含む Maxima の函数, 大域変数のリストを返します, 例えば, `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, 残りの名前を探す事が可能になります.

4.9 システムに関連する大域変数

システムに関連する大域変数

変数名	初期値	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 は alias,ordergreat,orderless 函数やアトムを名詞型と宣言する事によって設定された別名を持つアトムのリストです.

debugmode が true の場合, エラーが生じた時や false で中断モードに入った時は何時でも 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 を調べる事が出来ます.

myoptions は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.

optionset が 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時に便利です.

4.10　Maxima の終了

quit 函数

quit()

Maxima の終了は quit 函数を用います. この函数は引数を必要としませんが, 必ず quit() と入力し, 後の小括弧 () を忘れないで下さい. 尚,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.

第5章　Maximaの函数

この章で解説する事:

- 三角函数
- 指数函数と対数函数
- 代数方程式
- 極限
- 微分
- 積分
- 常微分方程式

5.1　三角函数

5.1.1　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, 即ち,$\cos^2(x)+\sin^2(x)=1$ や $\cos(2x)=2\cos^2(x)-1$ の様なものは予め 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事が出来ます.

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

cos	余弦函数
cosh	双曲線余弦函数
cot	余接函数
coth	双曲線余接函数
csc	余割函数
csch	双曲線余割函数
sec	正割函数
sech	双曲線正割函数
sin	正弦函数
sinh	双曲線正弦函数
tan	正接函数
tanh	双曲線正接函数

逆三角函数と逆双曲函数

函数名	概要
acos	逆余弦函数
acosh	逆双曲線余弦函数
acot	逆余接函数
acoth	逆双曲線余接函数
acsc	逆余割函数
acsch	逆双曲線余割函数
asec	逆正割函数
asech	逆双曲線正割函数
asin	逆正弦函数
asinh	逆双曲線正弦函数
atan	逆正接函数
atan2	逆正接函数
atanh	逆双曲線正接函数

尚, 逆正接函数には atan 函数 と atan2 函数の二種類がありますが,atan2 函数は, `atan2(y,x)` の様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様々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事も可能です. この場合, 大域変数 trigexpand を true にした状態で, 角の整数倍には, 大域変数 trigexpandtimes, 角の和に対しては大域変数 trigexpandplus を各々true に設定すると, 角の和と積の公式を用いて, 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 $x + \frac{\sin(5x)}{\cos(x)}$ を大域変数 trigexpand を true にした状態で展開を行う事を意味します. この表記は Maxima のトップレベルだけで利用可能です. 尚,ev 函数の詳細に関しては, 評価の節を参照して下さい.

三角函数の半角公式に関しては, 大域変数 halfangles を true にすると式の自動展開を実行します. この変数は大域変数 trigexpand の影響は受けません.

これらの変数とば別に, 三角函数の引数の declare 函数による宣言に従って, 三角函数を含む式の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, 最初に変数 i を整数,a を偶数,b を奇数として宣言しています. すると,Maxima は三角函数の引数を評価し, 式を自動的に変換しています.

三角函数に付随する函数は, 大域変数の trigexpand,trigreduce と trigsign を参照して下さい. 二つの share パッケージは Maxima に組み込みの簡易化の規則の trig と atrig を拡張します.

三角函数に関連する大域変数

変数名	初期値	概要
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 halfangles が true の場合, 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 は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 が設定されている時に, 引数の和を含む $\sin(x+y)$ の様な三角関数の自動展開が, 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 は trigexpand 函数の積規則を制御します. 大域変数 trigexpand が true に設定されている時に, 三角函数の引数が整数, 或いは有理数倍の式に対し, 三角函数の自動展開が実行されます.

triginverses は三角函数, 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.

trigsign が true であれば三角函数に対し負の引数の自動簡易化を行います. 例えば,trigsin が true の時に限り,`sin(-x)` は`-sin(x)` に変換されます.

5.1.2 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 函数は ⟨ 式 ⟩ に含まれる三角函数や双曲函数に対し, 倍角公式等を適用して式の展開を実行します. 最良の結果を得る為に, 予め expand 函数等で ∠式 ⟩ を展開しておきましょう.

簡易化の利用者制御を拡張する為, 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.sin と cos の全体の自動展開が必要ならば, 大域変数 trigexpand を true に設定しておきます.

trigreduce 函数は ⟨ 変数 ⟩ の積を持つ三角函数と双曲 sin 函数と cos 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事も試みます. 尚, ⟨ 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 函数は tan,sec 等を含む ⟨ 式 ⟩ の簡易化の為に, 恒等式 $\sin^2(x)+\cos^2(x)=1$ と $\cosh^2(x)-\sinh^2(x)=1$ を使って, sin,cos,sinh,cosh へと変換します.trigreduce と組合せる事で, より良い簡易化が得られます.

```
(%i6)  trigreduce(trigsimp(-sin(x)^2+3*cos(x)^2+x));
                              cos(2 x)   1
(%o6)                      4 (-------- + -) + x - 1
                                 2       2
(%i7)  trigsimp(trigreduce(-sin(x)^2+3*cos(x)^2+x));
(%o7)                          2 cos(2 x) + x + 1
```

trigrat 函数は 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 による線形結合とします.trigrat の計算結果は簡易化された sin と cos を含む有理式となります.

```
(%i10) trigexpand((cos(2*x+%pi*4/3*y+%pi/2)+sin(2*y+x))/(cos(x)+sin(y)));
                        4 %pi y                   4 %pi y
(%o10) (- cos(2 x) sin(-------) - sin(2 x) cos(-------) + cos(x) sin(2 y)
                          3                         3
                                     + sin(x) cos(2 y))/(sin(y) + cos(x))
```

5.1.3　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. 尚, このパッケージを利用する為には予め load(atrig1); を実行する必要があります.

5.2 指数函数と対数函数

5.2.1 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 があります. 指数函数 exp は Maxima 内部では, 自然底%e の冪として表現されています.

これらの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.

対数函数に関連する大域変数

変数名	初期値	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

先ず,%e_to_numlog が true であれば,r を有理数,x を式とすると, $e^{r\log x}$ が x^r に簡易化されます. 尚,radcan 函数も, この変換を行います.

大域変数 logabs が true の場合,integrate(1/x,x) の様に計算結果に log 函数が結果に含まれる場合, log 函数が log(abs($\cdots$)) で置換えられます. 但し,logabs が false であれば log($\cdots$) の項を持つものになります. 尚, 定積分の場合は,logabs:true の設定が利用されます. これは, 不定積分の両端点での評価が必要となる事が多い為です.

大域変数 logarc が true であれば, 逆三角函数, 逆双曲函数を対数函数の書式に変換します. logarc(⟨式⟩) はこの大域変数の設定なしで, 特定の式に対し,ev を用いた式の再評価を行います.

大域変数 logcoeffp は logcontract を用いた時に潰される係数の制御に利用する述語函数名を一つ設定します. 函数名は評価を行わない様に, 名詞型にします,log 函数から sqrt を生成したい場合, 係数が 1/2 の様な有理数でなければなりません. 従って, 述語函数は,log 函数の係数が整数であるか, 有理数の何れかとすれば良いでしょう. これを具体的に記述すれば, 以下の様に設定すれば十分です.

```
logconcoeffp:'logconfun
logconfun(m):=featurep(m,integer) or ratnump(m)
```

ここで, logcontract(1/2*log(x)); と入力されると, logcontract 函数は, 評価を実行する前に, logconcoeffp に設定された評価函数に与式に含まれる log 函数の係数 $\frac{1}{2}$ を引き渡します. 尚,false の場合は featurep(1/2,integer) で評価します. 述語函数の結果が true であれば, この例では, $\log\left(x^{\frac{1}{2}}\right)$ を評価します. false の場合はそのままの式を返します.

大域変数 logexpand が true の場合, 自動的に $log(a^b)$ を $blog(a)$ に変換します. ここで,all の場合, 自動的に $log(ab)$ は $log(a)+log(b)$ に変換されます.super の場合, $a=1$ でない有理数 a/b に

対し, $log(a/b)$ は $log(a) - log(b)$ に変換されます. 尚, 整数 b に対し, $log(1/b)$ は logexpand とは無関係に, 常に簡易化されます.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 が true の場合, 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 が true の場合,log の負の浮動小数引数は log 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 が false の場合,log を含む%e の冪乗の自動簡易化は実行されません.

5.2.2 対数函数に関連する函数

対数函数を潰す函数

logcontract(〈 式 〉)

logcontract 函数は log を含む式を簡易化で潰すものです. 即ち,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 〉 を再帰的に調べて, 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. 例えば, declare(n,integer); を実行して, logcontract(2*log(x)+4*n*log(y)+8); を実行すれば, 第二式の 4+n+log(y) の係数 4*n は述語函数 featurep(coeff,integer) を満す為, $\log\left(x^2y^{4n}\right)+8$ 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,m は integer として未宣言の為, 簡易化が途中で停止しています.

分枝を定める函数

plog(⟨x⟩)

plog 函数は複素数値の自然対数函数の主分枝を $-\pi < carg(x) \leq \pi$ とします.

極座標形式に変換する函数

polarform(⟨ 式 ⟩)

polarform 函数は与えられた ⟨ 式 ⟩ を r*%eˆ (%i*theta) の形に変換します. 尚, 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 $\%e^{\%i\%pi}$ をかけたものが返されます.

```
(%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)
```

5.3 代数方程式

5.3.1 Maxima の方程式

Maxima の方程式は演算子 = の両側に演算子 = を持たない式を配置した式です. 即ち, x^ 2+2*x+1=0 の様な形式になります. 但し, 右辺が 0 の場合は, 演算子 = を省略しても構いません.

尚, ここで演算子 = の左右の式は lhs 函数と rhs 函数を使って取り出す事が出来ます.

lhs と rhs

lhs(⟨ 方程式 ⟩)
rhs(⟨ 方程式 ⟩)

```
(%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, lhs(eq1) で方程式の左側の x^2+2*x+1, rhs(eq1) で方程式右側の y^2 を各々取り出しています. 尚, これらの lhs 函数と rhs 函数は演算子=に対してのみ使える函数です. 他の二項演算子 (例えば, > 等) に対しては使えません.

Maxima で扱える方程式としては,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, 微分'diff を含む方程式や積分'integrate を含む方程式もあります. 尚, 微分を含む方程式は, 常微分方程式の節で詳細を述べる為, ここでは, 微分と積分を含まない代数方程式を中心に述べます.

Maxima で連立方程式を扱う場合, $[eq_1,\cdots,eq_n]$ の様に, 複数の方程式で構成した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 から構成される方程式のリストを eq2 に割当てています. 連立方程式をリストで表現する為, 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ます.

Maxima には与えられた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. 又, 厳密解を求める函数として,linsolve 函数と solve 函数, 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 函数があります.

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区分出来ます.

先ず, 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出来ます. 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 函数を用いて可能であれば厳密解, 近似解が計算可能な場合は, 近似解を計算出来ます. 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.

Maxima で計算した結果を自動的に変数に代入したり, 重複リストを生成させる為には, 以下の大域変数を調整する必要があります.

方程式に関連する大域変数

変数名	初期値	概要
backsubst	true	三角関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,linsolve,solve 等の出力を制御

大域変数 backsubst は三角関数化した方程式に対して代入の制御を行います. backsubst が false の場合, 方程式を三角関数化した後で, 代入を防ぎます. これは, 後代入でとてつもなく大きな式が生成される様な問題で必要となります.

大域変数 globalsolve は Maxima で方程式を解いた時に, 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.globalsolve が true の場合, 解かれた変数に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
```

globalsolve:true とした状態で, ある方程式を解いた後に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します. 例えば, 上記の例の (c106) 行の方程式以下の行で置き換えた場合には次の様になります.

```
(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)
```

尚, 重複度が 2 以上の変数が存在する場合, 自動代入は実行されない事に注意して下さい.

大域変数 multiplicities は,solve や realroots 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 の為,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 である事が判ります. 最後の例では, $x = 1$ の重複度が 2, $x = -1$ の重複度が 3 である事が判ります. 実際, 与式を factor 函数で因子分解すれば確認出来ます.

大域変数 programmode は, 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.programmode が false の場合,solve,realroots,allroots と linsolve 函数は%t ラベル (中間行ラベル) に答をラベル付けして出力します.

true の場合, これらの函数はリストの要素として答えを返します (programmode:false も使われます. 但し,backsubst が false に設定された時を除きます).

```
(%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]
```

5.3.2 1 変数多項式方程式の場合

最初に, 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 ⟩)

最初の allroots 函数は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 の場合に実数上で因子分解を行いますが, 係数に%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 は重複解を持つ時に不正確な結果を返す事があります. この場合は与式に %i をかけたものを計算すれば解決するかもしれません.

allroots は多項式方程式以外には使えません.rat 命令を実行した後に, 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. polyfactor が true であれば,allroots の結果として常に同値な式 (但し, 因子分解されたもの) が返されます.

realroots 函数は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. 尚, ⟨ 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 ⟩ を省略した場合は, 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. 解は浮動小数点ではなく, 有理数で返されます.

realroots は解の大域変数 multiplicities に重複度の情報をリストの形式で追加します. 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	初期値	概要
polyfactor	false	因子分解の有無
rootsepsilon	1.0E-7	根を含む区間

polyfactor は allroots で利用される大域変数です.polyfactor が true であれば, polyfactor に与えられた多項式が実係数多項式なら実数上で因子分解し, 係数に%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.

rootsepsilon は 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.

5.3.3 区間内の根の個数

Maxima では指定した半開区間に存在する根の個数を計算する函数 nroots 函数があります. 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各々対応する minf,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
```

5.3.4 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, 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.

ここで, 多変数の線形多項式で構成された方程式系に対しては linsolve 函数が使えます.

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. 各方程式は各々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	初期値	概要
linsolve_params	true	解に助変数を導入
linsolvewarn	true	linsolve の警告を抑制

linsolve_params が true であれば,linsolve はまた記号%ri を生成し, algsys に記載された任意の助変数を表現する為に用いられます.

false であれば, 以前の様に linsolve が動作します. 即ち, 不定方程式型に対し, 他の項の幾つかの引数に対して解きます.

linsolvewarn が 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.

多変数多項式で構成された連立方程式は,solve 函数や algsys 函数で解く事が出来ます. 但し,solve 函数がより一般的な方程式の厳密解の計算が可能ですが,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.

algsys 函数

algsys([⟨ 方程式$_1$⟩ ,⟨ 方程式$_2$⟩ ,$\cdots$], [⟨ 変数$_1$⟩ ,⟨ 変数$_2$⟩ ,$\cdots$])

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事がありますが, これらは助変数を表示する為に用いられるもので, 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]
```

この例の様に,(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. 尚,%rnum_list には algsys で使われた助変数が逐次追加されて行きます.

algsys 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 で因子分解し, 各因子から構成される部分系 $system_i$, 即ち, 方程式の集合を構築します.

2. 部分系 $system_i$ から, 方程式 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 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 eq_n

と $system_i$ の部分系 $system_i \setminus \{eq_n\}$ に含まれる方程式 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 $system_{i+1}$ は $system_i$ 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, 係数が浮動小数でなければ, 厳密解を求める為に solve を呼出します. 更に, 方程式が単変数で線型, 二次, 或いは四次の多項式であれば,solve を再び呼出します.

   係数が浮動小数点で近似されている場合, 方程式が単変数で線型, 二次又は四次の何れでもなく, 大域変数 realonly が true の場合, 実数値解を見付ける為に函数 realroots を呼出します.realonly が false の場合, 解を求める為に函数 allroots を呼出します.
   尚,algsys 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. 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. 尚, 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 や reveal 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, 例えば, 多項式 f と g の根を α_i, β_j とすると, $res(f,g,x) = a_m^n b_n^m \prod_{0\le i\le m, 0\le i\le n}(\alpha_i - \beta_j)$ となる事が知られています.

この事は 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事を意味します. 従って, f と g が多変数の場合, 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ると, f と g の共通の根が存在する場合, 終結式は零でなければなりません. こうする事で, 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事で,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.

話を簡単にする為に, 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.

$$res(f,g,x) = \det \begin{pmatrix} a & by+p \\ c & dy+q \end{pmatrix}$$

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. この事から, 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けば $y=\frac{ap-cp}{ab-bc}$ 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 に値を代入すれば, 変数 x の方程式が得られ, その方程式を解けば最終的に $x=\frac{bq-dp}{ad-bc}$ 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, 次数を落した各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. この処理を繰返す事で, 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られます. 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 を用いて近似数値解を求め, それから, 処理を逆に遡る事で, 全ての解を求める事が出来ます.algsys はこの様なアルゴリズム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	初期値	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
realonly	false	true の場合,algsys 函数は実数解のみを返却

%rnum_list は algsys 函数で方程式の解を求めた時に導入された変数%r が生成順で追加されるリストです. これは後に解に代入する時に便利です.

algexact は algsys 函数に影響を与える大域変数の一つです. true であれば,algsys は solve を呼出し,realroots 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.algexact:true とすると, 厳密解のみを保証するものではなく,algsys が最初に厳密解を計算しようと試み, 結局,all か失敗した時に近似解のみを生成します.

algepsilon は algsys 函数で利用される定数です.

realonly が true であれば,algsys は実数解, 即ち%i を持たない解のみを返します.

5.3.5　一般的な方程式

一般的な方程式に対しては solve 函数が利用出来ます. この solve 函数は, 与えられた方程式系の厳密解を返す函数です. 方程式には,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. 但し,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する様な事は行いません.

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. 即ち, 式 x^2+2*x+1 が〈式〉であれば,solve 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アトムでない式でも構いません. 尚,〈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 を解く事で対処出来ます.

具体的には以下の処理を行います.

```
(%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〉を省略出来ます. 更に,〈式〉は有理式でも良く, その上, 三角関数, 指数関数等を含んでいても構いません.

solve 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 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 が呼出されます.
- 方程式二次, 三次, 又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$_1$〉,$\cdots$,〈方程式$_n$〉], [〈変数$_1$〉,$\cdots$,〈変数$_n$〉]) の場合, 多項式の方程式系を linsolve, 或いは algsys 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 を用いる場合, 第一引数のリスト [〈方程式$_i$〉,i=1,$\cdots$,n] は解くべき方程式を表現し, 第二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, 第二の引数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 で制御出来ます. 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	初期値	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

solvedecomposes が true であれば, 多項式を解く際に,solve に polydecomp を導入します.

solveexplicit が true であれば solve に陰的な解, 即ち,f(x)=0 の形式で返す事を禁止します.

solvefactors が false であれば,solve は式の因子分解を実行しません.

solvenullwarn が true であれば, 空の方程式リストや空の変数リストで solve を呼んだ場合に警告が出ます. 例えば, solve([],[]) と入力すると警告と一緒に空リスト [] が返されます.

solveradcan が true であれば solve は radcan を用います.radcan を使うと solve は遅くなりますが, 指数や対数関数を含む問題に対処出来ます.

solvetrigwarn が false であれば,solve は方程式を解く為に逆三角関数を利用し, それによって解が失なわれる事を警告しません.

solve_inconsistent_error が true であれば,solve と linsolve は, [a+b=1,a+b=2] の様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. 尚,false であれば, 空リスト [] を返します.

breakup が false であれば,solve 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. 但し,breakup が true となるのは programmode が false の時だけです.

5.3.6 漸化式の場合

Maxima では, 漸化式を扱う事が可能です. 但し, 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 函数は ⟨ 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.

但し, 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. 即ち,funcsolve 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)
```

5.4　極限

5.4.1　極限について

Maxima では limit が使えます. 一見すると limit は代入と似たものに見えるかもしれませんが, 実際は全く異った操作です.

例えば,$\frac{\sin(x)}{x}$ の原点での値は何になるでしょうか？ 安易な代入では,$\frac{0}{0}$ となってしまって判りませんね. 因に,$\frac{0}{0}$ や $\frac{\infty}{\infty}$ は不定形と呼ばれるものです. 字面は同じでも, 安易に割ってしまっては意味がありません.

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.

$\sin(x) = \sum_{i=0}^{\infty}(-1)^i \frac{x^{2i+1}}{(2i+1)!}$ となり, これを x で割ってしまうと,$\frac{\sin(x)}{x} = 1 + x \cdot (x$ の冪級数) となります. 以上から,x を 0 に近づけると 1 になる事が判ります.

Maxima で試してみましょう.Maxima には limit 函数があり, この函数は 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]);
```

極限はこの様な計算を行いますが, この近付けるという操作には方向を考えなければなりません. 例えば,$\frac{1}{x}$ はどうでしょうか. この函数は $x > 0$ なら正で,$x < 0$ で負になっており,$x = 0$ が不連続点になっています.

limit 函数を使って,Maxima で計算させてみましょう. 猶,Maxima では数直線の右側 (プラス側) から近付ける場合, 左側 (マイナス側) から近付ける場合を各々オプションの plus や minus を用いて指定する事が可能です.

```
(%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
```

この様に右側から近付けた場合には正の無限大, 左側から近付けた場合には負の無限大となっていますね. この様に左右の極限が異なるので, limit(1/x,x,0) の結果は und となっています.

limit 函数は正の無限大を inf, 負の無限大を minf, 複素数での無限大 (無限遠) を 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 といった表記で結果を返します.

```
(%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
```

極限に関連する大域変数

変数名	初期値	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

lhospitallim は limit 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) の様な場合に無限ループに陥いる事を防ぐ為のものです.

limsubst は limit が未知の形式に代入を行う事を防ぎます. これは limit(f(n)/f(n+1),n,inf) の様な式が 1 となるバグを避ける為です. limsubst が true であれば, この様な代入が許容されます.

tlimswitch が true であれば, 極限パッケージは可能な時に taylor 展開を利用します.

limit 函数

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する事も可能です

. 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 とします. 尚, 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. この場合は zeroa が原点の左側 (-側),zerob が原点の右側 (+側) から近付ける事を意味します. この場合は,minus や plus の様な方向を指定する必要はありません.

計算手法は,Wang,p. の"Evaluation of definite integrals by symbolic manipulation"- ph.d. thesis - Mac tr-92 October 1971. を参照して下さい.

limit は特別な記号として 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. 尚,-inf と minf, -minf と inf は Maxima では意味が異なるので注意して下さい. この様に極限を含む式の計算で limit 函数が

利用可能です即ち, 上記の-inf や-minf に加え,inf-1 様な定数式に対しては,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
```

この例で示す様に,inf-1 の様な式は 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事が出来ます.limit(x^ 2+inf*x) の様な式も同様です.

tlimit 函数

tlimit(⟨ 式 ⟩ ,⟨ 変数 ⟩ ,⟨ 値 ⟩ , ⟨ 方向 ⟩)
tlimit(⟨ 式 ⟩ ,⟨ 変数 ⟩ ,⟨ 値 ⟩)
tlimit(⟨ 式 ⟩)

tlimswitch を true にした limit 関数です. この tlimit 函数は ⟨ 式 ⟩ のテイラー展開を行い. その展開式に対して極限計算を行います.

5.5 微分

5.5.1 微分に関係する函数

Maxima の式の微分では diff 函数を用います.

diff 函数

diff(⟨ 式 ⟩ ,⟨ 変数$_1$⟩ ,⟨ 階数$_1$⟩ , ···,⟨ 変数$_n$⟩ ,⟨ 階数$_n$⟩)
diff(⟨ 式 ⟩ ,⟨ 変数 ⟩)
diff(⟨ 式 ⟩)

1 階微分の場合のみ,diff(⟨ 式 ⟩ ,⟨ 変数 ⟩) の様に階数を省略しても構いませんが, 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. 即ち,⟨ 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
```

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.

微分の表示を制御する大域変数

変数名	デフォルト値	概要
derivabbrev	false	微分の表示を制御

この大域変数 derivabbrev が true の場合, 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)
```

更に, 大域変数 derivsubst で名詞型の微分項の代入制御が行えます.

名詞型の微分の代入を制御する大域変数

変数名	デフォルト値	概要
derivsubst	false	名詞型の微分を含む項の代入を制御

大域変数 derivsubst は微分を含む項の代入の制御を行います. 例えば, $\frac{d^2y}{dt^2}$ 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 は名詞形の微分を含む式に対し, この様な置換を行うかどうかを制御するものです.

大域変数 derivsubst が false の場合,susbst 函数による微分項の置換が出来ませんが, 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
```

次に, 微分の名詞型を含む式に対し, その式に含まれる 名詞型微分の最大の階数を返す函数があります.

derivdegree 函数

derivdegree(⟨ 式 ⟩ ,⟨ 従属変数 ⟩ , ⟨ 独立変数 ⟩)

derivdegree 函数は 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
```

但し,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事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事で消去されるものですが, derivdegree 函数は与式を展開する事を行わずに, 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.

5.5.2 vect パッケージ

vect パッケージはデフォルトで 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.

vect パッケージでは非可換積. を内積演算子として再定義します. その為, 非可換積が可換化されてしまうので注意が必要です. 更に, 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様にする為, 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 にする事で, スカラーとの非可換積が可換に設定されます.

vect パッケージには以下の演算子の宣言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宣言された演算子であり, 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 expandflags 変数に割当てられたリストに含まれる変数で制御されます. 但し, 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. 但し,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,vsimp 函数で実際の処理を実行させています.

expandflags

変数名	初期値	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デフォルト値として false を持ちます. 変数名の後に 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, 後に 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.

expandcrosscross は `p~(q~r)` を `(p,r)*q-(p.q)*r` で置換するかどうかを決めます.

expandcurlcurl は 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.

expandcross が true の場合,expandcrossplus と expandcrosscross が true と同じ効果があります.

二つの大域変数 expandplus と 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, 他の大域変数, expandlaplaciantodivgrad は laplace 演算子を div grad で置換えます.

尚, 簡便の為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 として宣言されます.

```
(%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
```

express 函数

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. 但し,express 函数は微分を名詞型で返す為, 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];
(%o5)                     [x1, x2, x3] ~ [y1, y2, y3]
(%i6) express(v1);
(%o6)           [x2 y3 - x3 y2, x3 y1 - x1 y3, x1 y2 - x2 y1]
```

5.6 積分

5.6.1 記号積分について

Maxima は記号積分, 数値積分の両方が行えます. 積分に関しては,Risch の積分も不完全ながら実装されています. その為, 単純に公式を当て嵌めるだけで積分の計算を行う様なシステムよりも一段と優れた処理が行えます.

尚,Maxima は初等函数 (有理式, 三角函数, 対数, 指数) と多少の拡張 (error 函数, dilogarithm) で可積分なものに限定しているので,g(x) や h(x) の様な未知函数の積分は扱いません.

記号積分を行う函数

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 ⟩)

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事も,ev 函数を利用する事で出来るので, 記号積分に関しては integrate 函数だけで済ませられます.

積分の計算で式が非常に繁雑になる場合や, 積分記号の文字による表示が不要な場合, 大域変数 display2d を false にすると, 一行で結果が表示されます. 複雑な式の計算を行う場合には非常に便利です.

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. それに対し,risch 函数は比較の演算子 ($=, >, <$) 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.

積分定数を定める大域変数

変数名	初期値	概要
integration_constant_counter	0	積分定数のカウンター

実際に, その動作を見てみましょう.

```
(%i1) integration_constant_counter;
(%o1)                                0
```

```
(%i2) integrate(x^2,x);
                                    3
                                   x
(%o2)                              --
                                   3
(%i3) integrate(x^2=0,x);
                      3                          /
                     x                           [
(%o3)                -- = integrationconstant1 + I 0 dx
                     3                           ]
                                                 /
(%i4) integration_constant_counter;
(%o4)                                1
(%i5) integrate(x^3=0,x);
                      4                          /
                     x                           [
(%o5)                -- = integrationconstant2 + I 0 dx
                     4                           ]
                                                 /
(%i6) integration_constant_counter;
(%o6)                                2
```

この様に, 演算子=を含まない式の積分では, 結果に積分定数 integrationconstant が含まれませんが, 演算子=を含む式の積分では integrationconstant が出現します. これは,integration 函数が呼出す sinint 函数内部でその様に設定されている為で, この設定は, 他では行われていません.

integrate 函数から risch 函数を用いて Risch 積分を実行させる事が可能です. この場合は ev 函数を用います.

```
(%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)
```

尚,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. 尚, 実際に定積分を実行するのは defint 函数です. 尚,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. 数値積分が必要な場合,Maxima には romberg 函数や quanc8 函数 を用います.

広義の定積分では, 正の無限大には inf, 負の無限大には minf, 複素無限大には infinity が使えます. この inf と minf に関しては, -inf や-minf が各々minf や inf と同値なものではない事に注意して下さい. Maxima の広義の積分や, その他の代入操作で inf や minf は普通に使えますが, -inf や-minf を用いると全く無意味な結果を得る事があるので注意が必要です. その為,-inf や-minf が現れる式は limit 函数で一度評価しましょう.

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, 名詞型の'integrate を利用します.

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. 尚,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.

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場合の処理が行えます.integrate は, これらの場合, 自動的に risch 函数を適用します.

```
(%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)
```

risch 函数の動作を制御する大域変数に erfflag があります.

大域変数 erfflag

変数名	デフォルト値	概要
erfflag	true	risch 函数による erf 函数の挿入を制御

この大域変数 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,risch 函数が答に erf 函数を入れる事を抑制します.

ここで,erf 函数は次の error 函数の事です.

erf 函数

erf(x)

この函数の微分 $\frac{d}{dx}\left(erf(x)\right)$ は $\frac{2e^{-x^2}}{\sqrt{\pi}}$ となります.

5.6.2　変数の変換について

Maximaでは被積分函数の変数を新しい変数で置換えて積分する事が可能です. この為に,changevar函数を用います.

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 は総和 ($\sum$) や積 ($\prod$) の添字の変更にも使えます. この場合, 添字の変更は単純なシフトだけで, 次数の高い函数には出来ません.

```
(%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
```

5.6.3　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. 但し,$\frac{1}{x^3-x^2-x+4}$ の様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出来ません.

```
(%i3) integrate(1/(x^3-x^2-x+4),x);
                              /
                              [        1
(%o3)                         I --------------- dx
                              ]  3    2
                              / x  - x  - x + 4
```

この様な有理式の記号積分に対しては, 大域変数 integrate_use_rootsof を true に設定する事で, 計算を行う事が可能です.

代数的数の利用を制御する大域変数

変数名	デフォルト値	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, 大域変数 integrate_use_rootsof を true にして,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ました. 大域変数 integrate_use_rootsof が true の場合,integrate 函数は有理式の積分で, 分母の多項式から定義される代数的数を導入して積分を実行します. この意味を上の例で説明しましょう. 先ず,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 $x^3-x^2-x+4=0$ の根を決めます. Maxima では単純に%r1 等の%r で開始する変数を導入します. しかし, このままでは判り難いので, この例ではもう少し細かく説明します. この例の場合, 三次方程式なので根は 3 個あり, それらを α,β,γ とします. すると, 与式はこれらの根を用いると,$\frac{1}{(x-\alpha)(x-\beta)(x-\gamma)}$ に変換されます. この様に分母が一次

式の有理式の積になってしまえば, 後は簡単に計算する事が可能です. 実際の処理では, 各根に記号を割当てる必要は無く, 方程式 xxx の解,%r2 といった括り方で十分です.

但し, 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事です. 又, 返却される式は名詞型の為に, 安易に微分が出来ません. 何故なら,Maxima によるこの処理は, 単純に公式に当て嵌めているだけだからです.

5.6.4 記号積分の結果検証

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為, 比較的単純な式の計算でも思わぬ結果を得る事があります. その為, 記号積分の結果は何等かの形で検算を行う事を強く薦めます. 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事です.

Maxima の 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. 又,assume 函数を使って, 条件 $0 < x < 1$ を追加した場合でも, $\sqrt{x+\frac{1}{x}-2}$ の integrate の計算は間違っています.

この様に 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事も可能な場合もあります. これは Maxima に限った話ではなく, 数式処理一般でも言える事です. 更に, 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.

何故, 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為, 式を変形していれば. 並びも勿論異なる為に照合の結果も異なり, それによって処理の流れが違う為です. 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為, この様な現象は累積誤差の事を除くとまず生じません.

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, この例の様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等の処理を実行して積分したものと結果を照合する事は, 正しい答を得る為の有効な手段です.

しかし, 実際はこれでも不十分な事があります. 例えば, $\frac{3}{5-4\cos(x)}$ の積分です.

```
(%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様に思えます. しかし, 残念ながら計算は間違っています. 何故なら被積分函数は滑かな連続函数ですが, 結果の方は不連続函数になっています. この様な代物はグラフを利用して積分した函数を描くと間違いが一目で判ります.

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. 更に, 極限操作が必要な場合には, ldefint 函数もあります. 但し. この函数は非常に曲者で,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. 従って, 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為, 注意が必要です.

参考迄に,defint 函数の処理を簡単に説明しておきます. 通常は integrate 函数は, LISP 内部では $integrate です. この函数は内部で sinint 函数を呼出し, その結果に上限と下限の値を代入しています.sinint でも risch 積分を行う rischint を呼出す事も出来ますが,ev 函数を用いて rischint を用いる様に指定する事が出来ません. この点を修正する為には defint.lisp の antideriv の修正が最低でも必要となります.

defint 函数や integrate 函数による定積分の計算で, 下限や上限に記号や式が含まれている場合, その正負を尋ねてくる事があります. この場合, 正であれば pos; , 負であれば neg; , 零であれば zero; と入力します. 更に,assume を用いて正負の指定を予め行っていれば,Maxima はこの様な質問を行ないません.

```
(%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
```

留数を計算する函数

residue(⟨ 式 ⟩, ⟨ 変数 ⟩, ⟨ 点 ⟩)

residue 函数は ⟨ 点 ⟩ 回りの ⟨ 式 ⟩ の複素平面上での留数を計算します. 尚, 留数は式の laurent 級数展開を行った時の $(\langle \text{変数} \rangle - \langle \text{点} \rangle)^{-1}$ の項の係数になります.

```
(%i5) residue(x/(x^2+1),x,%i);
```

```
(%o5)                                  -
                                       2
(%i6) residue(sin(x)/x^4,x,0);
                                        1
(%o6)                                  - -
                                        6
```

ラプラス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 函数は ⟨ 旧変数 ⟩ と ⟨ 新変数 ⟩ に対する ⟨ 式 ⟩ の Laplace 変換を計算します. 逆 laplace 変換は ilt です.

⟨ 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為に, 函数の従属性をはっきりと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 に従属しているのであれば, laplace('diff(f(x,y),x),x,s) の様に函数 f が現れる場合は何時でも f(x,y) と記述する必要があります.

尚,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 函数は ⟨ 新変数 ⟩ と ⟨ 旧変数 ⟩ に対する ⟨ 式 ⟩ の逆 Laplace 変換を計算します. ⟨ 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為には有理式の展開を予め実行しておくと良いでしょう.

laplace と ilt の両方を solve や linsolve と使うと, 単変数の微分方程式か畳込み積分方程式を解く事が出来ます.

```
(%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
```

delta 函数は Dirac の δ 函数です. 尚,laplace 函数のみが 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り当てが無い為に, "Is a positive, negative, or zero?" と Maxima が尋ねています. assume を用いて $a > 0$ と仮定した場合を以下に示します.

```
(%i39) assume(a<0);
(%o39)                               [a < 0]
(%i40) laplace(delta(t-a)*sin(b*t),t,s);
(%o40)                                 0
```

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 函数は定積分を実行する函数です. 内部的に integrate を利用しており, 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. 但し, 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 よりは安全です. 但し,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 函数は〈変数〉の〈上限〉と〈下限〉に関する〈式〉の不定積分に対し, limit 函数を用いた評価を行い,〈式〉の定積分を計算します. 尚, 上限と下限に minf と inf といった値を与える場合には注意が必要です. 又,-inf や-minf の様に符号を付けて用いると, 無意味な結果を得る事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
```

尚,ldefint は内部的で函数の極の判別を一切行わずに,integrate 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.

一応, 右極限と左極限を下限と上限で取る様にしていますが, 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, 上限と下限の大小関係を逆にして ldefint を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 では内部的に sinint 函数を用いるだけです. ところで, 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. その為, 函数をいきなり ldefint 函数を用いて積分したり, 結果の検証を省く事は薦められません.

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
```

この例で示す様に,%i19 の ldefint 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為, この様な現象が生じています. これに対し,defint と integrate では極が存在する為にエラーを返しています.

尚,ldefint には zeroa,zerob が使えます.zeroa が 0 の右極限,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
```

但し,1+'zeroa の様な使い方は出来ないので注意します.

尚,ldefint は defint と同様に積分で Risch 積分を用いる事が出来ません.

tldefint 函数は大域変数 tlimswitch が true に設定された ldefint 函数に相当します. 尚, 大域変数 tlimsiwtch が true の場合, 極限の計算で Taylor 展開を利用する事を意味します.

5.6.5 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は quanc8 函数と romberg 函数があります.

数値積分を行う函数

quanc8('⟨ 被積分函数 ⟩, 実数$_1$, 実数$_2$)
quanc8(⟨ 被積分函数 ⟩, 変数, 実数$_1$, 実数$_2$)
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$_2$⟩)

先ず,quanc8 函数は load("qq") で予め読込む必要があります. quanc8('⟨ 被積分函数 ⟩, 実数$_1$, 実数$_2$) で, 最初の引数で指定された函数の定積分を区間 実数$_1$ から 実数$_2$ で計算します. 第一引数に函数名を用いる場合, 函数名に単引用符' を付けなければなりません

quanc8(⟨ 被積分函数 ⟩, 変数, 実数$_1$, 実数$_2$) で, 函数か式 (最初の引数) の変数 (二番目の引数) で区間 実数$_1$ から 実数$_2$ で定積分を計算したものになります.

使われる手法は Newton-Cotes の 8 次多項式による求積法で, ルーチンは適応型です.

romberg 函数は romberg 積分を行う函数です.romberg 函数は quanc8 函数とは違い, Maxima に自動的に読込まれます.

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. 尚,compile 函数でコンパイルされた函数の場合は, 浮動小数点型, 即ち,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 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.

尚,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為, 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.

一般的に,quanc8 函数の方が計算速度, 精度と安定性で romberg 函数に勝っている事が多いのですが,romberg 函数の方が精度が良好な場合もあります. 以下の $\sqrt{2x-x^2}$ の積分の例では romberg 函数の方が精度で勝り, $e^{-x}\sin(x)$ の積分では計算速度で romberg 函数が勝っています.

```
(%i31) showtime:all;
Evaluation took 0.00 seconds (0.00 elapsed) using 80 bytes.
(%o31)                                all
(%i32)  romberg(sqrt(2*x-x^2),x,0,1);
Evaluation took 0.18 seconds (0.18 elapsed) using 316.305 KB.
(%o32)                        .7853897937007632
(%i33)  quanc8(sqrt(2*x-x^2),x,0,1);
```

```
Evaluation took 0.02 seconds (0.02 elapsed) using 35.602 KB.
(%o33)                          .7849358178522697
(%i34) integrate(sqrt(2*x-x^2),x,0,1);
Evaluation took 0.12 seconds (0.12 elapsed) using 212.039 KB.
                                      %pi
(%o34)                                ---
                                       4
(%i35) bfloat(%);
Evaluation took 0.00 seconds (0.00 elapsed) using 808 bytes.
(%o35)                        7.853981633974483B-1
(%i36) bfloat(integrate(sqrt(2*x-x^2),x,0,1));
Evaluation took 0.12 seconds (0.12 elapsed) using 212.875 KB.
(%o36)                        7.853981633974483B-1
(%i37) romberg(exp(-x)*sin(x),x,0.,1.);
Evaluation took 0.01 seconds (0.00 elapsed) using 28.227 KB.
(%o37)                          .2458370426035679
(%i38) bfloat(integrate(exp(-x)*sin(x),x,0.,1.));
Evaluation took 0.13 seconds (0.13 elapsed) using 298.562 KB.
(%o38)                        2.458370070002374B-1
```

数値積分の方が上記の極を検出し易い事もあり, 両者の結果を比較するのも検算の方法としては良いでしょう.

romberg 函数に影響を与える大域変数

変数名	初期値	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 は 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],··· とする時,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 ならば 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. その為,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時に便利です. 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, 後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.

romberg 積分命令の精度は大域変数の rombergtol と rombergit で支配されます. romberg 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. 諦める前に刻幅の romgergit を半分にして試行します.

大域変数 rombergmin は romberg 函数による函数評価の最小数を制御します. romberg 函数はその第一引数を少なくとも $2^{\mathrm{rombergmin}+2}+1$ 回評価します. これは通常の収束テストが時々悪い通り方をする周期的函数の積分に有効です.

二重積分を行う函数

dblint('⟨F⟩,'⟨R⟩,'⟨S⟩,⟨ 浮動小数点$_1$⟩,⟨ 浮動小数点$_2$⟩)

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為には予め load(dblint) で dbint 函数を読込んでおく必要があります.

先ず,dblint('F,'R,'S,浮動小数点$_1$,浮動小数点$_2$) で以下の二重積分を計算します

$$\int_{\text{浮動小数点}_1}^{\text{浮動小数点}_2}\int_{R(x)}^{S(x)} F(x,y)dydx$$

第一引数の ⟨F⟩ は x,y の 2 変数の函数, 第二引数と第三引数の ⟨R⟩ と ⟨S⟩ は各々変数 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. 更に, 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. その為, 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, 各々の大域変数が x と y の区間の分割数を定めます. 尚, 収束性を改善する為に大域変数 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.

但し, 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為, 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

変数名	初期値	概要
dblint_x	10	X 軸方向の分割数
dblint_y	10	Y 軸方向の分割数

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. それらは X,Y の区間の分割数を定め,2*dblint_x+1 点が X 方向に計算され,Y 方向は 2*dblint_y+1 点となります. これらの変数は勿論, 互いに独立して変更する事か可能です.

5.6.6 antid パッケージ

antid パッケージには antid,antidiff と nonzeroandfreeof 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 函数は与式の不定積分を行います.antid の返却値は二成分のリストで, このリスト L とする時,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 は内部で 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 は述語函数で,⟨y⟩ が零でなく, ⟨x⟩ が ⟨y⟩ に含まれない場合に true を返す函数です. この述語函数を用いて,antid や antidiff の規則を定めています.

5.7 常微分方程式

5.7.1 常微分方程式の扱い

Maxima での常微分方程式の記述は通常の方程式と同様に, 演算子=を挟んで左右に式が記述される形となりますが, 式中に名詞型の微分'diff が含まれるものです.

Maxima では常微分方程式の一般解を計算する函数に ode2 函数と desolve 函数があります. 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.

例えば, 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 が何の変数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,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.

微分方程式の初期値問題を解く場合,desolve 函数を用いる場合と ode2 函数を用いる場合で解き方の手順が異なります.

最初に,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.

atvalue 函数は函数 f が一変数の場合, atvalue(f(x),x=pt,val) の様に函数 f(x) に対する x=pt での値 val を設定します.

f が多変数の場合, atvalue(f(x,y,$\cdots$),[x1=p1,x2=p2,$\cdots$],[val1,val2,$\cdots$]) の様に境界と対応する値をリストで与えます.

では, 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
```

この例では, 常微分方程式 $\frac{dy(x)}{dx} + 2x = \sin(x)$ を初期条件 $[y(0) = 0, \frac{dy(x)}{dx}|_0 = 0]$ で解いています.

次に,ode2 函数を用いる場合では, 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
                                          x
```

この例では, 微分方程式 $x^2\frac{dy}{dx} + 3xy = \frac{\sin(x)}{x}$ を ode2 函数を用いてその一般解を求め, ic1 函数から, $x = \pi$ の場合に y が 0 となる条件で特殊解を求めています.

```
(%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
```

一般解を求める函数

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 函数は, 常微分方程式系を与えられた変数に対して解きます.

初期値は 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 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.desolve 函数では, 求める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.

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,desolve 函数は false を返します.

```
(%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 函数による評価の一つの形態です. 詳細は 1.8.1 の小節を参照して下さい. この場合は式 [%o1,%o2] の f(x) と g(x) を代入し,%o1 と%o2 に含まれる微分の名詞型を評価させ,desolve による結果の検証を行っています. この評価の結果%o6 で=の両辺の式が等しい事から解が正しく求められている事が判ります.

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, 又は二階の常微分方程式を与えます. 尚, 常微分方程式の右側 (rhs(exp)) が 0 ならば, 左側のみを与えるだけでも構いません. 第二引数には ⟨ 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, 或いは陰的な解の何れかを返します. ここで,%c は一階の方程式の定数,%k1 と%k2 は二階の方程式の定数を表記する為に用いられます.

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後に 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, そして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 又は同次元方程式, 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. 例えば,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..

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 函数は, 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為, 特殊解を求める為, 異なった二点での連立方程式を解く必要があります.

その為, ⟨x の値$_1$⟩ と ⟨y の値$_1$⟩ が一つの点での値 ⟨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を x と y とすると, ⟨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 函数は初期値問題 (ivp) と境界値問題 (bvp) を解く為の ode2 パッケージに含まれるプログラムです.⟨ 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, 後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 函数は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, 後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.

関連図書

[1] 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.

[2] 河内明夫編, 結び目理論, シュプリンガーフェラーク東京,1990.

[3] 下地貞夫, 数式処理, 基礎情報工学シリーズ, 森北出版,1991.

[4] 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.

[5] ポール・グレアム,ANSI Common Lisp, ピアソン・エデュケーション,2002.

[6] 本間龍雄, 組合せ位相幾何学, 共立出版,1980.

[7] 寺坂英孝編, 現代数学小辞典, ブルーバックス, 講談社,2005.

[8] 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.

[9] 村上順, 結び目と量子群, 数学の風景 3, 朝倉書店,2000.

[10] 日本数学会編, 数学辞典 第 3 版, 岩波書店,1987.

[11] H.Cohen, A Course in Computational Algebraic Number Theory,GTM 138, Springer-Verlag,New York-Berlin,2000.

[12] D.Cox,J.Little and D. O'Shea,Ideals, Varieties, and Algorithms, UTM,Springer-Verlag,New York-Berlin,1992.

[13] Gert-Martin Greuel, Gerhard Pfister, A Singular Introduction to Commutative Algebra, Springer-Verlag,New York-Heiderberg-Berlin,2000.

[14] D.Rolfsen, Knots and Links. Publish or Perish, Inc,1975.

[15] Hal Schenck, Computational Algebraic Geometry London Mathematical Society student texts;58,2003.

[16] Open AXIOM のサイト http://wiki.axiom-developer.org/FrontPage

[17] DERIVE のサイト http://www.derive.com

[18] GAP のサイト http://www-gap.dcs.st-and.ac.uk/

[19] Macaulay2 のサイト http://www.math.uiuc.edu/Macaulay2/

[20] Mathsoft Engineering & Education, Inc. http://www.mathcad.com/

[21] Wolfram Research Inc. http://www.wolfram.com/

[22] Waterloo Maple Inc. http://www.maplesoft.com/

[23] Maxima の SOURCEFORGE のサイト http://maxima.sourceforge.net/

[24] MuPAD のサイト http://www.mupad.de/

[25] PARI/GP のサイト http://pari.math.u-bordeaux.fr/

[26] REDUCE のサイト http://www.zib.de/Symbolik/reduce/

[27] Risa/Asir 神戸版 http://www.math.kobe-u.ac.jp/Asir/asir-ja.html

[28] OpenXM(Open message eXchange for Mathematics)
http://www.math.sci.kobe-u.ac.jp/OpenXM/index-ja.html

[29] SINGULAR のサイト http://www.singular.uni-kl.de/

[30] 株式会社シンプレックスのページ htto://www.simplex-soft.com/

[31] The MathWorks,Inc. http://www.mathworks.com/

[32] Octave WebPage http://bevo.che.wisc.edu/octave/

[33] Scilab のサイト http://www.scilab.org/

[34] 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

[35] R のサイト http://www.r-project.org

[36] Insightful Corporation のページ http://www.insightful.com/

[37] 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/

[38] dynagraph のサイト http://www.math.umbc.edu/ rouben/dynagraph

[39] KSEG のサイト http://www.mit.edu/ ibaran/kseg.html

[40] Geomview のサイト http://http.geomview.org/

[41] surf の sourceforge のサイト http://surf.sourceforge.net/

[42] XaoS のサイト http://wmi.math.u-szeged.hu/ kovzol/xaos

索 引