

Maxima の計算事例

横田博史

平成18年2月14日(火)版

目次

第 1 章	結び目の Alexander 多項式	1
1.1	結び目の概要	1
1.2	結び目の射影図	3
1.3	結び目群の Wirtinger 表示	5
1.4	群環と Fox の微分作用素	9
1.5	Maxima で遊ぶ Fox の微分子	10
1.6	Alexander 多項式	17
1.7	Alexander 多項式で結び目を分類しよう	22
1.8	結び目の連結和と Alexander 多項式	24
1.9	結び目の鏡像と Alexander 多項式	27
第 2 章	surf を使う話	29
2.1	代数曲面	29
2.2	surf の概要	29
2.3	Maxima から surf を使う方法	29
2.4	surf の設定の取込み方法	30
2.5	多項式の処理	32
2.6	Steiner のローマ曲面ギャラリー	36
2.7	SINGULAR を使ってみよう	39
2.7.1	SINGULAR 初歩	39
2.7.2	SINGULAR で surf を使ってみよう	42
2.8	Maxima で終結式を使ってみよう	44

第1章 結び目の Alexander 多項式

1.1 結び目の概要

ここでは結び目理論から結び目の Alexander 多項式の計算について簡単なお話をします。

ここで何故、結び目なのかという理由ですが、結び目理論はドイツ語では Knotten Theorie と呼びます。それにしても、Knotten!! 実に、KNOPPIX に似ていますねえ…。そこで、結び目愛好家の為に、Maxima で結び目の不変量を計算して、KNOPPIX/Math を Knotten Pics/Math と洒落込もうというのが目的です。

結び目理論の全般の話はクロウエル、フォックス [3], 本間 [5], 河内 [1], 村上 [7] 等を参照して下さい。尚、亀甲結びの結び方と言った別の方面の愛好家の方の要望には沿える内容ではありませんので、悪からず。

結び目には、蝶々結びとか、色々な紐の結び方があります。ここで、結び目のある二つの紐が与えられた時、その結び目が同じ結び方で出来ているのかどうかを判別する為にはどうすれば良いのでしょうか? 現実問題としては、紐を引いたりして同じ形に変形出来るかどうかで判別する方法がありますが、数学では、どの様に表現すべきでしょうか?

まず、結び目理論で扱う結び目は、紐の両端を繋いで輪にしています。丁度、図 1.1 に示す様に空間内部の円周になっています。

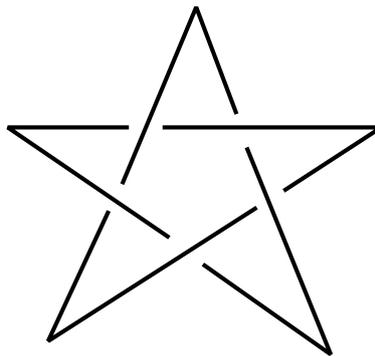


図 1.1: 星型結び目 (5_1)

こうするとどうでしょう、結び目の両端が切れたままだと、結び目の瘤の部

分を引いて行けば結び目が消えてしまいますが、両端を繋げた為に、引っ張っても結び目が解けて消えるとは限らなくなります。

この結び目は自分自身が交わる事の無い様に三次元空間 \mathbb{R}^3 に置かれています。この状態を円周 S^1 の三次元空間 \mathbb{R}^3 への埋め込みと呼びます。更に、三次元空間 \mathbb{R}^3 から結び目を取り除いた空間 $\mathbb{R}^3 - K$ の事を結び目 K の補空間と呼び、 $C(K)$ と記述します。

次に、扱う結び目に現実的な制約を入れましょう。先ず、ここで扱う結び目は、有限個の折線で近似可能な結び目に限定しましょう。この有限個の折線で近似出来る結び目の事を順な結び目と呼びます。因に、無限個の折線が必要な結び目を野性的な結び目と呼びます。この様な野性的な結び目は、力学系、特にカオスの話に出て来る事があります。

次に、結び目をゴム紐で出来たものの様に考えて、結び目を \mathbb{R}^3 内部で、紐を切ったり交点させずに変形して、互いに移り合う事の出来る結び目を同値な結び目と呼びましょう。この事を数学では、結び目 K_1 と K_2 に対し、ambient isotopy と呼ばれる図形が相互に連続的に変形して移り合える可能性を保証する写像が存在する場合を意味し、ここでは $K_1 \Leftrightarrow K_2$ と表記しましょう。この ambient isotopy という写像は、簡単に言ってしまえば、 $t = 0$ [分] の時が、結び目 K_1 で、それから結び目が自分自身が交点する事もなく、結び目の形を徐々に変形させながら、 $t = 1$ [分] の時に結び目 K_2 に移って行く映画が作られる事を意味します。

そこで、個々の結び目を特徴付けるものに何かあるのでしょうか？次の節では、結び目の射影図を考えますが、そこで現れる結び目の交点の総数もその一つです、勿論、射影図の取り方で交点の総数は色々変化します。例えば、平面 \mathbb{R}^2 に置かれた $x^2 + y^2 - 1 = 0$ で定義される円周の事を自明な結び目と呼びますが、この自明な結び目を Y 軸を中心に捻じったものを射影すれば、交点は幾らでも出来ます。しかし、結び目の射影図が持ち得る最小の交点数を考えると如何でしょうか。自明な結び目の場合は 0 個になります。一般の結び目でも、下限が 0 個なので、必ず最小限を持つ事が、選択公理によって保証されています。そこで、交点の最小数で結び目を分類する方法が考えられますね。

実際どうなるのでしょうか？先ず、交点が無い結び目は自明な結び目の他には存在しません。この場合は一つだけです。何か使えそうですね。実際、この交点で分類したものが、数学辞典 [8] の付録、公式 7 の結び糸等の文献にある結び目の表です。表を見て頂くと判りますが、交点が 3 個と 4 個のケースは一つだけですが、交点数が 5 個以上になると同じ交点数を持つ結び目が複数個存在します。その為、結び目を交点数でグループ分けは出来ますが、個々の結び目を特徴付けるには十分ではありません。残念でした。

では、どうすればよいのでしょうか。天下り的になりますが、結び目には結び目群というものがあり、この結び目群で結び目は特徴付けられます。この結び目群は結び目 K の補空間 $C(K)$ の基本群と呼ばれる群の一種の事です。既に、

三次元空間内部の結び目はその補空間で分類が可能な事が知られています.

ここで, 結び目群の計算方法は, Dehn によるものと, Wirtinger による二つの方法が代表的ですが, どちらも結び目の射影図を描いて求める方法です. ここでは, より機械的に計算が出来る Wirtinger による結び目群の表示を利用します.

この結び目群は非常に情報が沢山ありますが, そのままでは扱い難いものです. 実際, 群も自由群に関係子を入れたものになるので, 二つの群が与えられた時に, それらの群が同じものかどうか判断するのは簡単ではない事が殆どです.

そこで, もっと簡単に結び目を比較出来る方法があります. それが, 結び目群の Wirtinger 表示で得られる関係子から, Alexander 多項式を生成する方法です. この時, Fox の微分子という一風変わった演算子を用います.

この章では, Maxima の規則について詳細を述べる為に, 結び目群から Alexander 方程式を生成する方法について簡単に述べます.

1.2 結び目の射影図

最初に, 結び目の射影図というものを描きます. これは三次元空間 \mathbb{R}^3 内部の結び目を平面 \mathbb{R}^2 に射影したもので, 図 1.2 に示す要領で, 光源 O (無限遠点でも構いません) から出る灯で結び目を射影したものです.

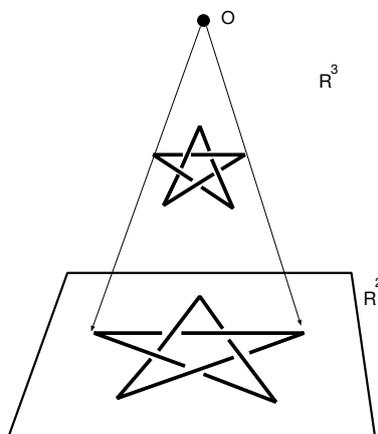


図 1.2: 結び目の射影図

ここで, 射影図上には結び目自体が交点する個所が現われる事があります. この交点する個所を交点と呼びますが, この交点を十字で描くだけでは, どちらが上にあるか判らないので, 地図で高速道路のジャンクションを描く要領で, 上の道で下の道が切断される様に描きます. 即ち, 図 1.1 の様な絵を描きます.

ところが, 図 1.1 の様に綺麗な射影図が何時も描けるとは限りません. 図 1.3 の左側に示す様に, $n(\geq 2)$ 重点や直交しない交点が出る事もあります. ところが, 図 1.3 の様な交点は順な結び目では, 有限個しか出て来ないので, それらの性格の悪い交点を図 1.3 の右側に示す要領で, 紐を局所的に動かしてしまえば除去出来てしまいます. この処理を射影図に施すと, 最終的に射影図で現われる交点は二重点だけに出来ます. この様にして, 有限個の交点で, それらが二重点のみとなった射影図の事を正則な射影図と呼びます.

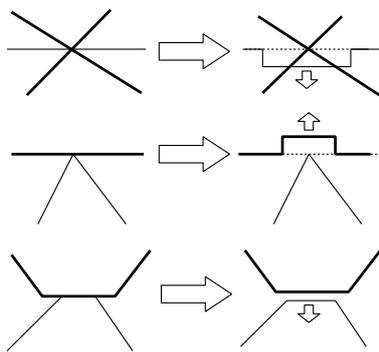


図 1.3: 変形可能な交点

ところで、同じ結び目でも、射影の方法で形が違うものに写ります。射影図に対し、図 1.4 に示す Reidemeister 移動と呼ばれる変形で結び目の射影図を変形する事も可能です。

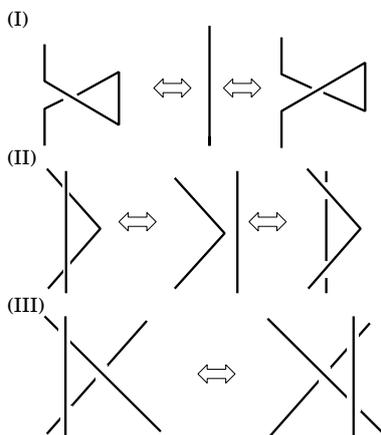


図 1.4: Reidemeister 変形

ここで、同値な結び目は Reidemeister 移動を繰替えして行う事で、相互に変形可能となる事が知られています。

1.3 結び目群の Wirtinger 表示

この節で、いよいよ結び目 K の結び目群 $G(K)$ の Wirtinger 表示を計算しましょう。

まず、綺麗な射影図が出来上がった状態で、結び目の射影図は交点で幾つの上道で分割されていますね。では、次に結び目に向きを入れて、それから各上道に適当な変数を割当てます。

図 1.5 では星型結び目 5_1 の射影図の各曲線に変数 v, w, x, y, z を割当てた様子を示しています。

まず、Wirtinger 表示による結び目群では、射影図上の曲線に割当てた変数が群の生成元となります。次に、結び目群の関係子は、各交点で図 1.6 に示す方法で、交点毎に決定されます。

因に、交点の上の $+1$ と -1 は交点の符号と呼ばれるものです。結び目の交点の符号の総和は符号和と呼ばれるもので、これも重要な結び目の不変量の一つです。

以上から、Wirtinger による結び目群の表示は、次の形になります。

$$\langle \text{上道}_1, \dots, \text{上道}_n \mid \text{交点}_1, \dots, \text{交点}_n \rangle$$

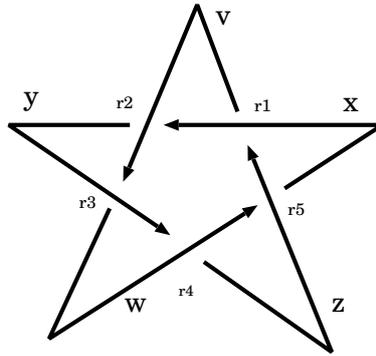


図 1.5: 星型結び目と変数

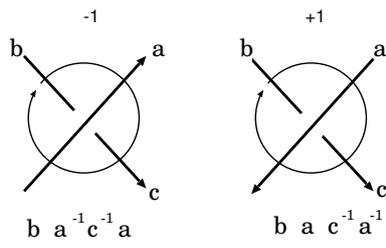


図 1.6: Wirtinger 表記

では何故, Wirtinger 表示で結び目群が表現出来たのでしょうか? この節も非常に天下りの的になりますが, その事を簡単に説明しましょう.

まず, 結び目の射影図の上道に付けた変数は図 1.7 に示す様に, $\mathbb{R}^3 - K$ 内部の点 P から出て, 結び目の上道の向きに対して右回りで上道を一回りし, 点 P に戻る閉じた道が対応します. この閉じた道は点 P に根本が固定されていますが, ゴム紐の様に伸縮自在で, 結び目を取り除いて出来た穴に邪魔されなければ, 点 P に潰れてしまう性質を持っています. ここで, 点 P を基点とする閉じた道 a_1 と a_2 が等しくなるのは, a_1 と a_2 が空間内部で連続的に互いに変形出来る場合とします.

閉じた道 $a b$ による積 ab は点 P を出て a の道を辿って, 次に b の道を辿る閉じた道と定義します. この時, 点 P から動かない道は, この積の単位元となります. 更に, 生成元 a を閉じた道とすれば, a^{-1} を逆の同じ閉じた道で, 逆方向の左回りに上道を一周する閉じた道としましょう.

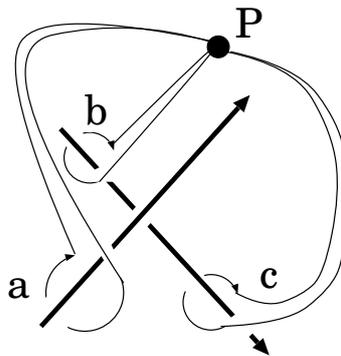


図 1.7: 結び目と閉じた道

ここで, a と a^{-1} の意味を説明するのが図 1.8 です.

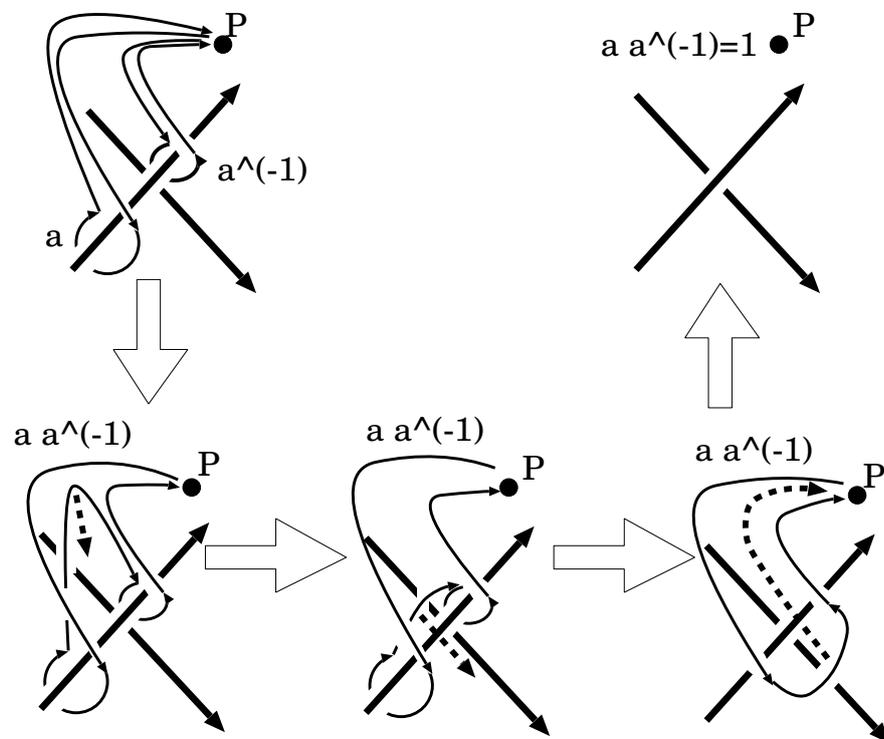


図 1.8: 閉じた道 a , a^{-1} と aa^{-1} の意味

この様に, 通常の閉じた道 a と a^{-1} は各々結び目の上道に遮られているので, この上道を越えて点 P へと潰れる事は出来ません.

ところが, 積 aa^{-1} は, 一度, 閉じた道 a を一回りしてから, 次は a^{-1} の道, つまり, a の道の逆方向に動く事を意味します. そこで, aa^{-1} は図 1.8 の中段左の様な道になります. この道を矢印に沿って動かすと, 結局, この閉道は結び目から外れてしまうので, 点 P に潰れる事になります. すると, 点 P から動かない点が単位元 1 となります. この事から, a^{-1} の意味が明確になるかと思えます.

Wirtinger 表示の関係子の意味は, この図 1.8 でやっている事と似た処理を行う事で判ります. 先ず, 図 1.9 に示す様に, これらの閉じた道を関係子の順で繋ぐと, 曲線から外れた点 P から出て点 P に戻る向き付けられた円になります. この円を少しずつ動かして行くと, 結び目の射影図の曲線に引掛かからずに点 P に潰す事が出来ます.

この様に, Wirtinger 表示の関係子は, 関係子で表現される式が単位元 1 になる事を意味しているのです.

尚, 結び目群で 1 になるという事は, 空間上で邪魔される事無しに, 基準点

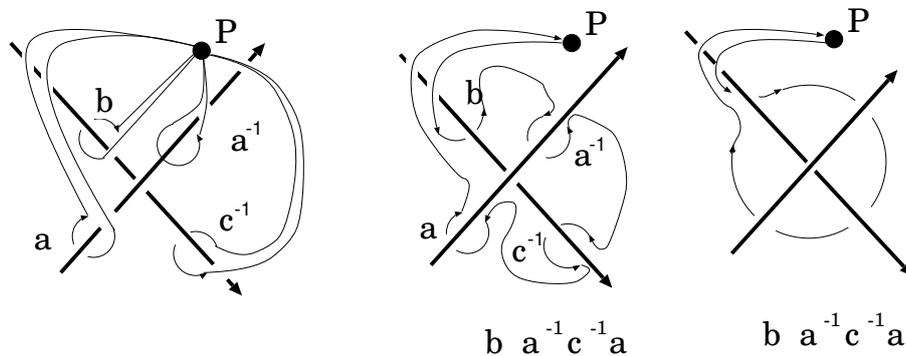


図 1.9: Wirtinger 表記の意味

P に潰れる事を意味します。ここで、 $\mathbb{R}^3 - K$ 内部では、閉じた道は、埋め込まれた円盤、即ち、自分自身が交点した変な円盤ではなく、歪んでいるだけの円盤の境界になっている事が知られています (Dehn の補題)。

この事から、自明な結び目の場合、結び目群は $\langle x \rangle$ 、即ち、 \mathbb{Z} に等しい事が判ります。

さて、Wirtinger 表示の話に戻ります。図 1.5 の星型結び目 5_1 の場合、関係子の計算は図 1.7 の関係式を参照すれば、結び目群の Wirtinger 表現として以下が得られます

— 星型結び目 5_1 の Wirtinger 表現 —

$$\langle v, w, x, y, z | xz^{-1}x^{-1}v, vx^{-1}v^{-1}y, yv^{-1}y^{-1}w, wy^{-1}w^{-1}z, zw^{-1}z^{-1}x \rangle$$

尚、この群では関係子が交点の数だけ出来ませんが、実際はこの関係子の内で一つは不要です。そこで一つを取り除いたものでも構いません。

Wirtinger 表示による結び目群は、図 1.4 に示す Reidemeister 移動の不変量となります。この事を説明する為には語の Tietze 変換を用います。詳細はクロウエル、フォックス [3] 等の結び目理論の本を参照して下さい。

1.4 群環と Fox の微分作用素

次に、結び目群 $G(K)$ の生成元を x_1, \dots, x_n とする時、これらで生成される自由群 $F = \langle x_1, \dots, x_n \rangle$ とその群環 $\mathbb{Z}F_n$ を考えましょう。群環は、群の成分を可換演算子 $+$ を使って形式的な和を取ったものになります。この群環は、積 $*$ に対しては群になり、和 $+$ を結合律や分配律を満す様に入れるので、環になります。

Fox の微分子 Δ は次の性質を持っています。

— Fox の微分子 Δ —

- $\Delta(x + y) = \Delta(x) + \Delta(y)$
- $\Delta(xy) = t(y)\Delta(x) + x\Delta(y)$

まず, Fox の微分子 Δ は $\mathbb{Z}F_n \rightarrow \mathbb{Z}F_n$ の線形写像 (\mathbb{Z} 準同形写像) となります. それから, 語の積に対しては, 何処となく普通の微分の積公式に似たものですが, 函数 $t: \mathbb{Z}F_n \rightarrow \mathbb{Z}$ で, 群 F_n の各生成元に 1 を代入する線形函数 t が入っている事に注意して下さい.

この基本性質を利用すると, Fox の微分子が次の性質を持つ事を容易に確認出来ます.

— Fox の微分子 Δ の性質 —

- $\Delta(n) = 0, n \in \mathbb{Z}$
- $\Delta(x^n) = \sum_{i=0}^{n-1} x^i \Delta(x), n \in \mathbb{N}$
- $\Delta(x^{-n}) = -\sum_{i=-n}^{-1} x^i \Delta(x), n \in \mathbb{N}$

では, Maxima で Fox の微分子を実際に表現してみましょう.

1.5 Maxima で遊ぶ Fox の微分子

Fox の微分子を Maxima で表現しましょう. その為には, Fox の微分子で利用する各種の函数を最初に定義して行きましょう.

まず, Fox 微分の積の性質で出て来た函数 t を定義しなければなりません. 尚, t は変数で使う事が多い為, ここでは函数名を $t1$ としましょう.

この函数は線形函数なので, 語に対してのみ函数を定義し, 後は declare 函数を使って, 線形性を持つ事を宣言してしまえば十分です.

— 函数 $t1$ —

```
t1(x):=block([vars:listofvars(x),n,i],
             n:length(vars),
             for i in vars do
               (x:subst(1,i,x)),
             return(x));
```

この函数では, 全ての変数に 1 を代入する為, listofvars 函数で語を構成する変数を全て取出し, subst 函数を使って変数に 1 を代入します. 尚, 実際は引数が語であれば 1 となる様に定義し, それから, declare 函数で函数 $t1$ が linear

であると宣言しておけば十分ですが,ここでは subst 関数と listofvars 関数を使いたかったので,敢て複雑にしています.

今度はこの関数 t1 の値域となる群環 $\mathbb{Z}F_n$ の Maxima での表現方法です.ここでは,群 F_n の積は Maxima の非可換積である dot 積を用い,冪はそれに対応する非可換積の冪^{^^}を使います.但し,整数と語の積では,可換積 * を用います.

従って,非可換積と可換積が混入した式を扱う事になりますが,このような式の計算を Maxima は難無く実行します.

```
(%i76) 2*x.y+3*x.y;
(%o76)                    5 (x . y)
(%i77) 2*x.y.z+3*x.y;
(%o77)                    2 (x . y . z) + 3 (x . y)
```

これで群環の演算は一応大丈夫そうですね.

では,与式が語であるかどうかを判別する述語関数を定義しましょう.与えられた式を語として判断しても良いのは以下の状態に限られます.

————— w が語となる条件 —————

- 与式がアトム w
- 与式がアトムの積 $w_1.w_2 \cdots w_n$
- 与式が整数とアトムの可換積を含む式 $n * w_1.w_2 \cdots w_n$
- 与式がアトムや語の非可換冪 $x^{\wedge m}$

これらの状況の判定は演算子で出来てしまいます.先ず,最初のアトム w の場合,演算子が含まれていませんが,述語関数 atom で十分判別出来ます.それから $n * w_1.w_2 \cdots w_n$, $w_1.w_2 \cdots w_n$ と $x^{\wedge m}$ の場合,各々の内部表現の先頭は,可換積 *, 非可換積 . と非可換積の冪^{^^}になります.

これらの事から,以下の判定条件で式を判別すれば良い事になります.

————— w が語となる判定条件 —————

- $atom(w)$ が true
- $inpart(w,0)$ が可換積 *
- $inpart(w,0)$ が非可換積 .
- $inpart(w,0)$ が非可換積の冪^{^^}

この様に,与えられた式が語であるかどうかを調べる述語関数 wordp は,最初に与式がアトムであるかどうかを調べ,アトムであれば true, そうでなけれ

ば, 与式の内部表現の先頭元を抽出し, これが非可換積積, 可換積, 冪の何れかであれば, true を返し, それ以外は false を返す函数になります.

この函数を Maxima で構築したのが, 以下の述語函数 wordp となります.

述語函数 wordp

```
wordp(w) := if atom(w) then true
           else if member(inpart(w,0),[".", "^^", "**"]) then true
           else false;
```

この word 函数では, member 函数を用いて, $\text{inpart}(w, 0)$ の値が, 非可換積, 非可換積の冪[^]か可換積^{*}の何れかになれば true を返す様になっています.

では早速, この函数を試してみましょう.

```
(%i59) wordp(w);
(%o59)                                     true
(%i60) wordp(2*w.z);
(%o60)                                     true
(%i61) wordp(w.z);
(%o61)                                     true
(%i62) wordp(w.z+2*x.y);
(%o62)                                     false
```

これで, 与式が語であるかどうかを検証出来る述語函数が出来ましたね. この述語函数の存在によって, Maxima にとっても, 何が語となるかを判断出来ます. この様な規則を定める対象を判別する述語函数が出来た段階で, 初めて規則が定義出来る様になります.

では, Fox の微分子 D_{fox} を Maxima で表現しましょう. ここでは最初に演算子の属性を宣言しておきます. この函数は, 引数の前に置くので, prefix 函数を用いれば十分です. この宣言では, 具体的に函数を記述する必要は全くありません. この宣言の次に, 今度は線形性 $D(x+y) = D(x) + D(y)$ も追加しましょう. この線形性の宣言では declare 函数を用います.

```
(%i1) prefix("D_fox:");
(%o1)                                     D_fox:
(%i2) declare("D_fox:", linear);
(%o2)                                     done
```

これで, D_{fox} の演算子としての属性と線形性が付加されました. 次に, 微分の性質を付加します. この性質は Maxima の規則として与えます.

```
(%i3) matchdeclare([_x,_y],wordp);
(%o3)
done
(%i4) defrule(Dfox_Prod,D_fox:(_x._y),
             D_fox:_x*t1(_y)+_x.D_fox:_y);
(%o4) Dfox_Prod : D_fox: (_x . _y) ->
      D_fox:_x t1(_y)+_x.D_fox:_y
```

この例では、最初に matchdeclare 関数を使って、規則を与える為の変数の定義を行います。規則は matchdeclare 関数で宣言した変数を用いて定義を行いますが、一般の式への適用では、後の述語関数を用いて、その式に適用する際に、前提とした変数の条件を満すものかどうかを Maxima は検証します。この宣言を行わなければ、Maxima は defrule 関数で用いた変数に対してのみ規則の適用を行い、一般の式に対して利用は出来なくなります。

次に,defrule 関数を用いて、規則名と、適用する式の並びと規則の適用後の変換式を宣言します。この例では、規則名を Dfox_Prod とし、その内容は、 $D_fox : (x . y)$ を $D_fox : x * t1(y) + x.D_fox : y$ で置換えるものです。こうする事で、変数 a と b が語としての属性を持っていれば、 $D_fox : (a.b)$ は規則 $Dfox_Prod$ を適用する事で、 $D_fox : a * t1(b) + a.D_fox : b$ で置換えられる事になります。

以上で非可換積に対する微分の性質が付加されました。

ここで注意しなければならない事は、非可換積に影響を与える大域変数 dotassoc をデフォルトの true のままにしていると、Maxima が $(x.y).z$ を自動的に $x.y.z$ に平坦化してしまう事です。これは内部的には自動的に $(. x y z)$ となる事を意味しますが、この規則では二項演算を前提にしているので、 $(x.y).z$ の様に平坦化される前の二項演算のままであれば処理が上手く出来ません。その為、結び目群を定義する前に、`dotassoc:false` を設定していなければなりません。

後は与式が定数や非可換冪の場合の処理を組込みましょう。先ず、与式が定数の場合、Fox の微分子は 0 を返します。ここで、環は整数環 \mathbb{Z} なので、定数は integer として仮定して構いません。従って、定数に対しては、次の規則を入れれば十分です。

```
(%i5) matchdeclare(_a,integerp);
(%o5)
done
(%i6) defrule(Dfox_const,D_fox:_a,0);
(%o6)
Dfox_const : D_fox: _a -> 0
```

次に、与式が非可換冪の場合を組込みましょう。この場合、matchdeclare で宣言する整数には正の場合と負の場合を各々宣言する必要があります。そこで、

posintp と negintp の二つの述語関数を定義します.

posintp 関数と negintp 関数

```
posintp(x) := if featurep(x,integer) then if is(x>0) then true
negintp(x) := if featurep(x,integer) then if is(x<0) then true
```

posintp と negintp 関数は基本的に同じ操作を行う関数です. 与えられた引数が整数であれば, 既に整数の属性を持っているので, featurep 関数は true を返し, 後は正負の判定を行えば良く, 一般の変数の場合, 先ず, declare 関数で整数の属性を持たせ, それから assume 関数で正負の性質を入れてあるものを扱います.

では, 関数の動作をここでも確認しておきましょう.

```
(%i11) declare(n1,integer);
(%o11)                                     done
(%i12) assume(n1>0);
(%o12)                                     [n1 > 0]
(%i13) posintp(n1);
(%o13)                                     true
(%i14) declare(n2,integer);
(%o14)                                     done
(%i15) assume(n2<0);
(%o15)                                     [n2 < 0]
(%i16) negintp(n2);
(%o16)                                     true
(%i17) negintp(n1);
(%o17)                                     false
```

これで道具が揃ったので, これらの述語関数を用いて正の冪の場合と負の冪の場合の展開規則を D_fox : に付与しましょう. 規則の与え方は, 微分としての展開規則 *Dfox_Prod* と同様です. 最初に規則を与える変数を matchdeclare 関数を用いて宣言し, その変数を用いて defrule 関数で, 規則を宣言します.

```
(%i9)matchdeclare(_ap,posintp)$
(%i10)matchdeclare(_an,negintp)$
(%i11)defrule(Dfox_PPpower, D_fox:(_x^(_ap)),
             sum(_x^(i),i,0,_ap-1).D_fox:_x);
(%o11) Dfox_PPpower : D_fox: (_x      )
             -> sum(_x      , i, 0, _ap - 1) . D_fox: _x
(%i12)defrule(Dfox_NPower, D_fox:(_x^(_an)),
```

```

-sum(_x^(i),i,_an,-1).D_fox:_x);
(%o12) Dfox_NPower : D_fox: (_x
      )
      -> - sum(_x      , i, _an, - 1) . D_fox: _x

```

これで,Fox の微分子を利用する為に必要な規則は全て揃いました. では試しに $r1=x.y.z^{(-1)}.y^{(-1)}$ に Fox の微分子を作用させた結果を計算してみましょう.

```

(%i9) dfr1:D_fox:(r1);
      <- 1>      <- 1>
(%o9)      D_fox: (x . (y . (z      . y      )))

```

入力しても別に展開されていません. 何故でしょう?Maxima では規則を定めるだけではなく, その定めた規則を式に対して適用する函数を必要とします. この様な函数には幾つかありますが, 今回は apply1 函数を用います.

必要な規則は, 語の非可換積の展開規則 Dfox_Prod と負の冪の規則 Dfox_NPower の二つです. この様に複数の規則を利用する場合, apply1 に適用する規則を順番に並べます. ここでは最初に展開規則を用い, それから負の冪の展開規則を用いるので, 以下の様に入力します.

```

(%i11) apply1(dfr1,Dfox_Prod,Dfox_NPower);
      <- 1>      <- 1>      <- 1>
(%o11) x . (y . (- z      . D_fox: z - z      . (y      . D_fox: y))
      + D_fox: y) + D_fox: x

```

確かに展開が出来ていますね.

以上の Fox の微分子の定義, 規則の宣言等を纏め, ファイル fox.mc に記入しましょう. このファイルの中にコメントを入れておくことで後で読み易く, 管理を行うのも容易になります. このファイル fox.mc の内容を以下に示しておきましょう.

```

/* MAXIMA */

```

```

/* 函数 t1

```

```

      x:結び目群の語.

```

```

Fox 微分で現われる函数. 線形性を持ち, 語を 1 に写す函数.

```

```

函数の定義では, 単純に語の変数に 1 を subst を用いて設定し,

```

```

線形性は declare を利用します.
*/

t1(x):=block([vars:listofvars(x),n,i],
             n:length(vars),
             for i in vars do
               (x:subst(1,i,x)),
             return(x));

declare(t1,linear);

/* Fox の微分子の定義.

演算子は prefix 関数を用いて, 前置式の演算子である事を宣言します.
更に, 線形性は declare 関数を用います.
*/

prefix("D_fox:");
declare("D_fox:",linear);

/* Fox の微分子の置換規則で用いる変数宣言と述語関数定義 */

/* _x と _y は語とします. */
matchdeclare([_x,_y],wordp);

/* 語の判定関数 */
wordp(w) :=
if atom(w) then true
else if member(inpart(w,0),[".", "^", "*"]) then true
else false;

/* _a は整数とします. 判定関数は Maxima 組込の integerp を用います. */
matchdeclare(_a,integerp);

/* _ap は正整数, _an は負の整数とします. ここで判定関数は,
正整数の場合は posintp, 負の整数の場合は negintp を用います.*/
matchdeclare(_ap,posintp);
matchdeclare(_an,negintp);

```

```

/* 語の判定関数 */
posintp(x):= if featurep(x,integer) then
              if is(x>0) then true;
negintp(x):= if featurep(x,integer) then
              if is(x<0) then true;

/* 積規則 Dfox_Prod */

defrule(Dfox_Prod,D_fox:(_x._y),D_fox:_x*t1(_y)+_x.D_fox:_y);

/* 定数に対する規則 Dfox_const */
defrule(Dfox_const,D_fox:_a,0);

/* 正の冪に対する規則 Dfox_PPower */
defrule(Dfox_PPower, D_fox:(_x^(_ap)),
        sum(_x^(i),i,0,_ap-1).D_fox:_x);

/* 負の冪に対する規則 Dfox_NPower */
defrule(Dfox_NPower, D_fox:(_x^(_an)),
        -sum(_x^(i),i,_an,-1).D_fox:_x);

/* 非可換積の結合律の適用を阻止. 積規則を利用する為に必要 */
dotassoc:false;

```

このファイルの演算子, 関数や規則を用いたければ, `load("fox.mc");` と入力します. 更に, Maxima の起動時に直ちに使用したければ, Maxima を起動するディレクトリに `maxima-init.mac` という名前のファイルを作り, その中に, 予め, `load("fox.mc");` を記入しておきます. すると, `maxima-init.mac` があるディレクトリ上で `maxima` を起動させると, 自動的に `maxima-init.mac` 内部が解釈され, `fox.mc` が読み込まれます.

これで Alexander 多項式を計算する為の道具が漸く揃いました.

1.6 Alexander 多項式

Alexander 多項式を計算する為には, Alexander 行列というものを計算しなければなりません. この Alexander 行列は結び目群 $G(K)$ の関係子に Fox の微分子を作用させる事で出来る行列です. 要するに, ベクトル (r_1, \dots, r_n) から Fox の微分子による Jacobian が Alexander 行列となります.

ここで、先程の節で定義した Fox の微分子は、通常の微分で言えば、 D に相当します。普通、変数 x に対する微分 $\frac{d}{dx}$ が存在しても良さそうですが、Fox の微分子も同様です。この場合、結び目群の Wirtinger 表現の生成元 x_i に対して、 $\frac{\partial}{\partial x_i}$ が定義され、他の生成元 x_j に対して、 $\text{frac}\partial x_j \partial x_i = \delta_{ij}$ を満たします。

例えば、星型結び目の場合、生成元は x, y, z, v, w の 5 個あるので、Fox の微分子は具体的には、 $\frac{\partial}{\partial x}, \dots, \frac{\partial}{\partial w}$ と生成元の数だけ存在します。

ここで Maxima 上で定義した Fox の微分子 D.fox:には、生成元の情報がありません。実はこうした方が便利なのです。例えば、星型の場合、5 個の微分子を定義し、それらを使って行列を求めるたりする事や、2 変数の関数を定義して処理を行うよりも、ここで行った様に、一つの演算子を決めて、 $\frac{\partial}{\partial x_j} x_i = \delta_{i,j}$ の関係を後で代入する方が、より処理を機械的に行えるだけでなく、余計な定義や処理が不要になるからです。

それから、Fox の微分子を作用させた後に、結び目群 $G(K)$ の Wirtinger 表示による生成元を全て変数 t で置換し、非可換積とその冪も全て可換積とその冪に置換します。この処理によって、D.fox:の Jacobian は $\mathbb{Z}[t^{-1}, t]$ の元を成分を持つ正方行列になります。この正方行列を Alexander 行列と呼びます。

結び目群 $G(K)$ の (1 次)Alexander 多項式は、この Alexander 行列の (1 次)の余因子行列の各成分の最大公約因子の事です。

尚、Alexander 多項式は Laurant 多項式環 $\mathbb{Z}[t^{-1}, t]$ のイデアルの生成元です。その為、 $t^n, n \in \mathbb{Z}$ の違いが生じます。そこで、Alexander 多項式として定数項が零にならない様に、各項の次数が正となるものを選択します。

この処理を纏めたプログラム calcAlexanderPoly を以下に示します。

```

/* MAXIMA */

/* calcAlexanderPoly

fox.mc と併用する事が前提です。

結び目 G はリストで表現します。
即ち、G=<x1, ..., xn | r1, ..., rm>を G: [[x1, ..., xn], [r1, ..., rm]]
で表現します。

calcAlexanderMatrix では、Wirtinger 表示の結び目群に対して
Alexander 行列の計算を行います。その為、関係子 r1, ..., rm の
個数 m は n か n-1 に等しくなければなりません。従って、関係子の
個数が n か n-1 に等しくなければエラーになります。
*/
calcAlexanderPoly(G):=
block(

```

```

[vars:G[1],rels:G[2],amat,Alex:[],
Ia,dfx,rdfx,rdfx,crdfx,mrow,n:length(G[1]),
m:length(G[2]),AlexanderPoly,APolyData:false,
tmp,lst:[]],
/* 結び目群の Wirtinger 表示に限定 */
if m=n or m=n-1 then
(
Ia:subst("[",matrix,ident(n)),
for i from 1 thru n do append(lst,[i]),
dfx:map("D_fox:",vars),
/* 関係子リストに演算子を作用 */
rdfx:map(lambda([x],
          apply1(D_fox:x,Dfox_Prod,
                  Dfox_PPower,
                  Dfox_NPower)),
          rels),
/* 微分に 1 か 0 を設定 */
for i from 1 thru n do
( mrow:Ia[i],
  rdfx2:rdfx,
  for j from 1 thru n do
    (
      rdfx2:map(lambda([x],
                    subst(mrow[j],dfx[j],x)),rdfx2)
    ),
/* 非可換積の冪を可換積の冪に変換 */
crdfx:map(lambda([x],
                subst("^","^^",x)),rdfx2),
/* 非可換積を可換積に変換し, 簡易化を実施 */
amat[i]:ratsimp(map(lambda([x],
                        subst("*",".",x)),crdfx))
),
/* Alexander 行列を配列として最初に構築. */
for i from 1 thru n do
  Alex:append(Alex,[amat[i]]),
/* 配列データから行列データに変換 */
Alex:substpart(matrix,Alex,0),
/* 変数を t に変換します */
for i in vars do

```

```

        Alex:subst(t,i,Alex),
Alex:ratsimp(Alex),
/* Wirtinger 表現で関係子の総数が生成元の個数 n よりも
   一つ少ない場合の処理. Maxima の組込函数では余因子
   行列の生成は正方行列に限定されます.
*/
*/
if m=n-1 then
(
  /* nx1 の零行列を n 列目に追加. */
  tmp:addcol(Alex,zeromatrix(n,1)),
  /* 小行列式を計算. */
  tmp:map(lambda([x],
                determinant(minor(tmp,x,n))),lst)
)
else
  tmp:expand(adjoint(Alex)),
  /* 余因子行列をリストに変換. */
  tmp:substpart("[",tmp,0),
  /* LGCD で Alexander 多項式を抽出 */
  AlexanderPoly:num(LGCD(map(LGCD,tmp))),
  /* Alexander 行列と多項式のリストを生成. */
  APolyData:[Alex,AlexanderPoly]
)
else
  /* エラー処理. Error! と表示するだけのシンプルなもの */
  print("Error!"),
  /* Alexander 行列を返します. */
  return(APolyData)
)$
/* リストから最大公約因子を計算します.

Lp: 多項式, 或いは, 整数で構成されるリスト.

長さが 1 の場合, その成分をそのまま返します.
長さが 2 以上の場合, 先頭の二つの成分の最大公約因子を計算し,
その最大公約因子と頭の Lp の成分を二つ取り除いたリストを合せ
たリストを用いて, 再帰的に LGCD を呼出しています.
*/
LGCD(Lp):=

```

```

block(
  [a1,n,lgcd:false],
  /* Lp がリストであるかを確認. */
  if listp(Lp) then
    (
      /* Lp の長さを求めます. */
      n:length(Lp),
      /* Lp の長さが 1 の場合, 成分をそのまま返します. */
      if n=1 then
        lgcd:first(Lp)
      else
        (
          /* Lp の先頭二つで GCD を計算. */
          a1:gcd(Lp[1],Lp[2]),
          /* LGCD を再帰的に呼出しています.
             尚,rest(Lp,2) で先頭の二つの成分を削除した
             リストを生成し,append で二つのリストを結合します.
          */
          lgcd:LGCD(append([a1],rest(Lp,2)))
        )
      )
    else
      lgcd,
      /* lgcd を返して終わります. */
      return(lgcd)
  )$

```

このファイルは calcAlexanderPoly 関数と LGCD 関数の二つで構成されています。calcAlexanderPoly 関数は計算した Alexander 行列と多項式をリストの形式で返します。内部で処理している事は、与えられた結び目群から生成元と関係子を取り出し、生成元に D_{fox} を作用させたリストを生成します。このリストへの作用は map 関数を用います。次に、生成元の個数 n から n 次の単位行列 I_a を生成します。この単位行列の i 番目の行は i 列のみが 1 となるので、 $D_{fox} : x_j \Leftrightarrow \frac{d}{dx_i} x_j = \delta_{ij}$ への変換に用います。ここで、lambda 式を用いて臨時的な関数を構築し、その関数に対して subst 関数を用いて変換を行います。

それから、非可換積を可換積、非可換積の冪を可換積の冪へと変換します。この変換では substpart 関数を用います。Maxima では演算子は式の 0 に置かれるので、0 を入れ換える様になっています。ここでも相手がリストの為、map 関

数を効果的に適用する為に,lambda 関数を使って, 臨時の関数を定義しています. それからリストの形式から行列データに substpart を用いて変換し, 各変数を全て t に置換えると Alexander 行列が出来ます.

次に Alexander 行列の余因子行列を計算しなければなりません, 余因子行列を計算する adjoint 関数や小行列を求める minor 関数も正方行列のみにしか使えません. Wirtinger 表示の場合, 変数よりも一つ関係子が少なくても問題が無い為, これでは余因子行列が計算出来ない事があります. そこで, 変数よりも関係子が一つ少ない場合には,Alexander 行列の n 列目に n 次の零ベクトルを addcol 関数を使って追加します. そして,minor 関数で Alexander 行列 $Alex$ の小行列 $Alex_{1 \leq i \leq n, n}$ を minor 関数を用いて計算し, determinant 関数を map 関数を適用する事で作用させて必要な余因子行列を計算します. すると, 多項式成分の行列/リストが出来ますが, その成分の最大公約因子を求める為に, 新たに構築した LGCD 関数を使います.

この LGCD 関数はリストに含まれる各成分の最大公約因子を計算する関数で, リストの長さが 1 であれば, リストに含まれる式をそのまま返し, 長さが 2 以上であれば, 頭の二つに gcd 関数を使って最大公約因子を求め, 頭二つを削ったリストの先頭に結果を加えたリストを使って LGCD 関数を呼出すという再帰的な手法を用いています. この再帰的な手法は LISP で顕著な特徴の一つで,Maxima でもこの様に利用出来ます. 但し, 安易な再帰的な手法は速度的な面で問題が出易い傾向があります.

そこで, 余因子行列をリストに変更し, この LGCD 関数を用いて最大公約因子を求めます. この時, 最初に map 関数で余因子行列リストの行成分の最大公約因子を計算すると, 行の最大公約因子のリストが得られます. そのリストに LGCD 関数を適用すると求める Alexander 多項式が得られるのです.

1.7 Alexander 多項式で結び目を分類しよう

星型結び目 (5₁)

先程から出ている星型結び目の Alexander 多項式を計算しましょう. 先ず, 星型結び目の結び目群の Wirtinger 表示を以下に示します.

$$\langle v, w, x, y, z | xz^{-1}x^{-1}v, vx^{-1}v^{-1}y, yv^{-1}y^{-1}w, wy^{-1}w^{-1}z, zw^{-1}z^{-1}x \rangle$$

では, 早速計算しましょう. この例題では, 予め, `load("fox.mc")` と `load("AlexanderPoly.mc")` が実行されているものとします.

```
(%i14) star: [[x,y,z,v,w],
[x.z^(-1).x^(-1).v,v.x^(-1).v^(-1).y,
y.v^(-1).y^(-1).w,
y.v^(-1).y^(-1).w,
```

```

w.y^^(-1).w^^(-1).z,
z.w^^(-1).z^^(-1).x]]$
(%i15) K5_1:calcAlexanderPoly(star)$
(%i16) K5_1[2];

(%o16)
          4   3   2
         t  - t  + t  - t + 1
(%i17)tex(K5_1[1]);

$$\begin{matrix} \frac{t-1}{t} & -1 & 0 & 0 & \frac{1}{t} \\ 0 & \frac{1}{t} & \frac{t-1}{t} & -1 & 0 \\ -1 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} \\ \frac{1}{t} & \frac{t-1}{t} & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & -1 \end{matrix}$$

(%o17)
          false

```

星型結び目の Alexander 多項式として $t^4 - t^3 + t^2 - t + 1$ が得られます, 従って, この結び目は自明な結び目と違う事が判りました.

尚, Alexander 行列は tex 関数で出力した結果を, $\text{T}_{\text{E}}\text{X}$ のソースファイルに入れると以下の様に表示されます.

$$\begin{pmatrix} \frac{t-1}{t} & -1 & 0 & 0 & \frac{1}{t} \\ 0 & \frac{1}{t} & \frac{t-1}{t} & -1 & 0 \\ -1 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} \\ \frac{1}{t} & \frac{t-1}{t} & -1 & 0 & 0 \\ 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & -1 \end{pmatrix}$$

クローバー結び目 (3_1)

クローバー結び目は三葉結び目 (trefoil) とも呼ばれ、図 1.10 に示す様に、三枚の葉がある結び目です。

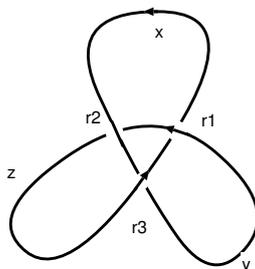


図 1.10: クローバー結び目 (3_1)

クローバー結び目の結び目群

$$\langle x, y, z \mid xyz^{-1}y^{-1}, yzx^{-1}z^{-1}, zxy^{-1}x^{-1} \rangle$$

```
(%i1) load("AlexanderPoly.mc");
(%o1) AlexanderPoly.mc
(%i2) Trefoil: [[x,y,z],
[x.y.z^(-1).y^(-1),y.z.x^(-1).z^(-1),z.x.y^(-1).x^(-1)]]$
(%i3) calcAlexanderPoly(Trefoil);
      [ 1 - t t - 1 ]
      [                ] 2
(%o3) [[ t - 1 1 - t ], t - t + 1]
      [                ]
      [ - t t - 1 1 ]
```

この calcAlexanderPoly では Alexander 行列と Alexander 多項式のリストを返します。この結果から、クローバー結び目の Alexander 多項式は $t^2 - t + 1$ である事が判ります。この様に Alexander 多項式が 1 でない為、クローバー結び目は自明な結び目と等しくない事、即ち、本当に結ばれている事が判ります。

1.8 結び目の連結和と Alexander 多項式

二つの向き付けられた結び目が与えられた時、連結和によって、新しい結び目を生成する事が出来ます。

結び目の連結和

1. 先ず, 二つの結び目 K_1 と K_2 を用意し, 各結び目に向きを入れておきます.
2. 結び目 K_i 上の点 P_i を中心とする半径 r の球 $B_i(r), i \in \{1, 2\}$ を取ります. これらの球は結び目の一部を含みますが, この際, 半径 r を十分小さく取れば, $B_i(r)$ がドーナツの様な形状にする事が出来ます.
3. これらの球 $B_i(r)$ を結び目 $K_i, i \in \{1, 2\}$ から結び目の曲線を含めて削除します.
4. 取り除いた個所で, 結び目の向きが一致する様に結び目 K_1 と K_2 を繋ぎ合せます.

この結び目を結び目 K_1 と K_2 の連結和と呼び, $K_1 \# K_2$ と表記します.

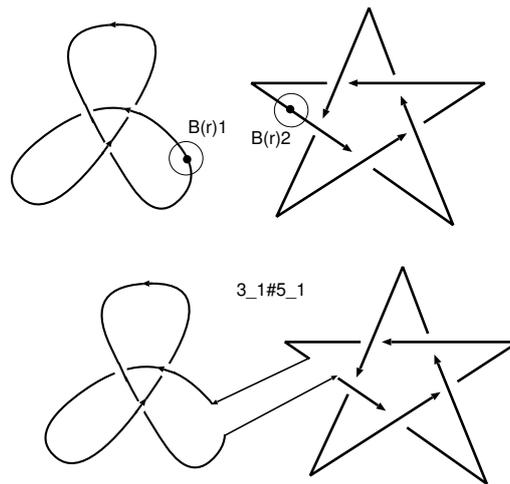


図 1.11: 結び目 3_1 と 5_1 との連結和

この時, 結び目の集合に, 連結和 $\#$ を入れる事で, 結び目の集合に半群の構造が入ります. 単位元はこの場合, 自明な結び目になります.

ここで, 3_1 と 5_1 の連結和の Alexander 多項式を計算してみましょう. この場合, $3_1 \# 5_1$ の Wirtinger 表示を直接計算しても構いませんが, 3_1 と 5_1 の Wirtinger 表示があるので, それを利用しましょう.

$3_1 \# 5_1$ の場合, 図 1.12 の様に, K_1 側の結び目群はそのままにして, K_2 側の変数名を変更しておきます. そして, 繋げる個所の生成元に関係式を追加します. 例えば, この例では, x と y_1 の道を繋げるので, $x = y_1$. 即ち, xy_1^{-1} を追加します. ここで, calcAlexanderPoly は簡易的なプログラムなので, 関係子

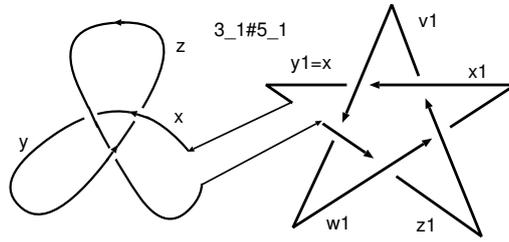


図 1.12: 結び目 3_1 と 5_1 との連結和の結び目群

を一つ抜いておく必要があります. その為, k_2 側の本来の関係子を一つ抜いて, その代わりに, xy_1^{-1} を入れておきます.

— $3_1\#5_1$ の結び目群 —

$$\left\langle \begin{array}{l} x, y, z, x_1, y_1, z_1, v_1, w_1 | xyz^{-1}y^{-1}, yzx^{-1}z^{-1}, \\ zxy^{-1}x^{-1}, v_1x_1^{-1}v_1^{-1}y_1, y_1v_1^{-1}y_1^{-1}w_1, \\ w_1y_1^{-1}w_1^{-1}z_1, xy_1^{-1} \end{array} \right\rangle$$

```
(%i61) star1:subst(x1,x,star)$
(%i62) star1:subst(y1,y,star1)$
(%i63) star1:subst(z1,z,star1)$
(%i64) star1:subst(v1,v,star1)$
(%i65) star1:subst(w1,w,star1)$
(%i66) ts:[append(Trefoil[1],star1[1]),
append(append(Trefoil[2],rest(star1[2],1)),[x.y1^(-1)])]$
(%i67) cs:calcAlexanderPoly(ts)$
(%i68) factor(cs[2]);

                2          4   3   2
(%o68)          (t  - t + 1) (t  - t  + t  - t + 1)
```

この様に連結和の Alexander 多項式は, 二つの結び目の Alexander 多項式の積になります.

これは, 何故でしょうか? 実際は非常に簡単な事で, 連結和の Alexander 行列を見ると判ります..

3₁#5₁ の Alexander 行列

$$\begin{pmatrix} 1 & -t & t-1 & 0 & 0 & 0 & 0 & 1 \\ t-1 & 1 & -t & 0 & 0 & 0 & 0 & 0 \\ -t & t-1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & \frac{1}{t} & 0 \\ 0 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & 0 \\ 0 & 0 & 0 & \frac{t-1}{t} & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{t} & \frac{t-1}{t} & -1 & 0 \end{pmatrix}$$

如何でしょうか？ クローバ結び目と星型結び目の Alexander 行列の成分が対角線上に二つのブロックで出現していますね。

この事から、与えられた結び目の Alexander 多項式が既約でなければ、この結び目は Alexander 多項式の各既約因子に対応する結び目連結和で構成されている可能性があります。

1.9 結び目の鏡像と Alexander 多項式

(途中で) ここでは結び目 K の鏡像 \bar{K} について考えましょう。結び目を図 1.13 に示す

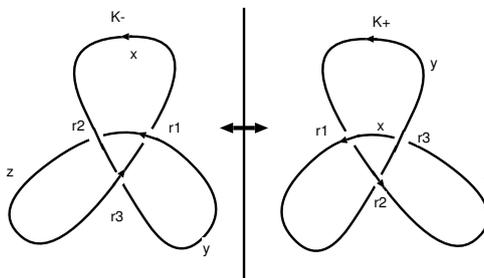


図 1.13: クローバ結び目の鏡像

ここで、 K_+ とその鏡像 K_- の違いは何でしょうか？それは交点の符号が逆になる事です。 K_+ の場合、交点の符号は全て $+1$ になりますが、 K_- では全て -1 になっています。その為、 K_- と K_+ は本質的に違う結び目の筈です。そこで、Alexander 多項式を計算してみましょう。

K_- の結び目群

$$\langle x, y, z \mid xzx^{-1}y^{-1}, yz^{-1}y^{-1}zy, yxy^{-1}z^{-1} \rangle$$

```

(%i20) TrefoilM: [[x,y,z],
[ x.z^(-1).x^(-1).y,z.y^(-1).
z^(-1).x,y.x^(-1).y^(-1).z]]$
(%i21) tm:calcAlexanderPoly(TrefoilM);
          [ t - 1   1           ]
          [ ----- -   - 1 ]
          [  t      t           ]
          [                    ]
          [  1           t - 1 ]  2
(%o21)    [[ -   - 1   ----- ], t  - t + 1]
          [  t           t     ]
          [                    ]
          [           t - 1   1   ]
          [ - 1   ----- -   ]
          [           t     t     ]

```

ところが, k_2 の Alexander 多項式を計算して見ると, クローバー結び目の場合は鏡像の区別が付きません.

この様に, Alexander 多項式は万能ではありません. 他に, 自明な結び目でないのに Alexander 多項式が 1 になるものも多く見つっています.

第2章 surfを使う話

2.1 代数曲面

Maxima で描くグラフは基本的に $y = f(x)$ や $z = h(x, y)$ の形式, 或いは助変数を用いたグラフに限定されます. その為, $x^2 + y^2 - 1 = 0$ を満す点のグラフを描きたい場合, $y = \sqrt{1 - x^2}$ と $y = -\sqrt{1 - x^2}$ のグラフを同時に描くか, $x(t) = \frac{2t}{t^2+1}$ と $y(t) = \frac{t^2-1}{t^2+1}$ を描く事になります.

そこで, 多項式の零点集合が簡単に描ける surf の登場となります.

2.2 surfの概要

surf は変数 x, y, z の多項式の零点集合を描くアプリケーションです. 単純な GUI を持っており, 直接スクリプトを編集して描画する事も容易に出来ます. C 風の簡単な処理言語を持っていますが, 複雑な処理は出来ません. 基本的に, 幾つかの基本的な設定を入力し, その設定に対して多項式の零点を描くだけで, 多少の反復処理が出来る程度です.

surf の詳細はこの本では述べません. surf のサイト <http://surf.sourceforge.net/> のオンラインマニュアル <http://surf.sourceforge.net/doc.shtml> や, 私の webpage (<http://www.bekkoame.ne.jp/ponpoko/Math/surf/SurfExamples.html>) を参照して下さい.

尚, 残念な事ですが, UNIX 環境でしか動作しないので, Windows で利用される方は, coLinux か KNOPPIX/Math で遊んで下さい.

2.3 Maxima から surf を使う方法

Maxima から surf を使う最も簡単な方法は system 関数を用いて surf を呼出す事です. 但し, 単純に呼出しても, Maxima 側の多項式は surf には引渡されません. そこで, 中間ファイルを Maxima 側で作って, それを surf に引渡せば良いのです. これは surf の起動時のオプションの -x を用います. 即ち, `surf -x <ファイル>` で指定したファイルを surf が起動時に読んで, 実行をします.

そうとなれば, Maxima で surf 用のスクリプトを生成する関数を構築すれば良いのです. 最も簡単な方法は, 多項式の変数を完全に x, y, z に限定し, 変数

が2個あれば曲線,3個ならば曲面を描く様に, 予め容易した surf の設定ファイルに曲線や曲面の方程式として多項式を引渡せば良いのです.

ここではもう少し複雑な処理を行わせましょう.

2.4 surf の設定の取込み方法

surf では, 曲線や曲面を描く際に, 方程式が満す解を求める為に, 解法の設定, 解の精度や反復の回数, そして, 絵の大きさを各々, root_finder,epsilon,iterations,width と height で行います. 具体的には, 以下の値で十分でしょう.

— 求解に関するパラメータとウィンドウの大きさ —

```
root_finder=d_chain_bisection;
epsilon=0.0000000001;
iterations=20000;
width=500; height=500;
set_size;
```

尚,root_finder には色々ありますが, 自己交差を持つ曲面, 曲面と平面との断面が綺麗に描けるのが d_chain_bisection です.

曲面の場合はこれらの設定に加えて, 以下の設定を追加します.

— 曲面の場合の設定 —

```
do_background=yes;
background_red=255;
background_green=255;
background_blue=255;
rot_x=0.14; rot_y=-0.3;
```

これらの値は,rot_x と rot_y で曲面を X 軸と Y 軸で回転させ, それ以外は背景色の指定パラメータになります.

これらの値をファイルに書込むのが面倒であれば, 大域変数にしてしまえば, 便利です. しかし, 属性を用いるともっと扱い易くなります. この場合,maxima-init.mac に以下の設定を行います.

maxima-init.mac での属性設定

```
put(surfg, "d_chain_bisection",root_finder);
put(surfg, 20000,iterations);
put(surfg, 500,width);
put(surfg, 500,height);

put(surf, "yes",do_background);
put(surf, 255,background_red);
put(surf, 255,background_green);
put(surf, 255,background_blue);
put(surf, 0.14,rot_x);
put(surf, -0.3,rot_y);
```

これで、二次元と三次元の共用の設定が属性に含まれる surfg と曲面専用の設定が含まれる surf の属性が設定されます。

又、別の方法は、これから構築する函数ファイルに上記の設定を含める事です。今回はこの方法を採用します。

初期設定の変更は再度と同じ方法で、値を変更して再設定すれば出来ます。又、属性値の取出しは、get 函数を用いて取込みます。

具体的に言うと、`put(surf,255,background_red);` と入力された場合,surf の background_red の属性値は `get(surf,background_red);` で得られます。

尚,surf と surfg の属性は 10 個あります。この属性を一々函数に書込むのは流石に面倒です。そこで,properties 函数を使って,surf と surfg の属性に何かがあるか一覧のグラフを出させ、その属性名と属性値の等式を作る方法を考えましょう。つまり、次の処理を行う事です。

```
(%i8) properties(surfg);
(%o8) [["user properties",height,width,iterations,root_finder]]
(%i9) properties(surf);
(%o9) [["user properties",rot_y,rot_x,background_blue,background_green,
background_red,do_background]]
(%i10) %[1][2]=get(surf,%[1][2]);
(%o10) rot_y = -0.3
```

この等式を配列に入れてしまえば、あとは stringout 函数で配列をファイルに書出してしまえば良いのです。こうすると、属性を減らしたり増したりするのも容易になりますね。

2.5 多項式の処理

surf で用いる数値は $1.0e-3$ は使えますが, Maxima の様に bigfloat 等の数値はありません. 余計な問題を避ける為に, ここでは多項式を expand で展開した後に, ratsimp で簡約化します. ratsimp を使うと, 多項式の係数は有理数に近似されます.

次に重要な事ですが, surf で利用可能な多項式は基本的に変数 x, y, z の多項式に限定されます. 勿論, x, y, z 以外の surf 内部での変数も扱えますが, あくまでも, 描かれるグラフは XY 平面, 或いは XYZ 空間に限定されているので意味はありません.

その為, 変数は 2 か 3 個に限定するの必要があり, x, y, z 以外の変数が使われると, それらを x, y, z で置換える操作もあった方が良いでしょう.

ここで, 多項式の変数を返す函数として, showratvars 函数があります.

例えば, `showratvars(x + y^2 + a^3);` と入力すると, `[a, x, y]` が返されます. このリストは Maxima の変数順序の小さい順で変数が左から並んでいます. Maxima の変数順序は基本的に逆辞書式, 即ち, 逆アルファベット順となるので, 変数を小さい順に並べると, アルファベット順になります. そこで, showratvars 函数で得られた変数リスト vars の左側から順番に x, y, z を対応させれば良い事になります.

具体的には vars[1] を x , vars[2] を y , vars[3] を z で置換えるので, subst 函数が使えそうです. ところが, subst 函数では, 置換えのリストを与えると左から右へと代入が行われる為,

`subst([a = x, x = y, y = z], a + x + y)` の結果は困った事に `3 * z` になってしまいます.

そこで, 置換えは二段階で行います. 函数に局所変数 surf.tmp_x, surf.tmp_y と surf.tmp_z に一旦置換え, これらを $[x, y, z]$ に最終的に置換え事で多項式を変数 x, y, z の多項式に変換します.

それから, スクリプトに曲線と曲線の場合に分けて, surf の描画命令を追加します.

これは曲線の場合は,

```
curve=多項式;  
draw_curve;
```

曲面の場合は,

```
surf=多項式;  
draw_surface;
```

を surf の設定の最後に追加すれば, 後は Maxima の system 関数を使って, surf に実行させる事が出来ます.

この考えで構築した Maxima の関数を以下に示します.

```
/* Maxima */

/* 属性の設定.*/
/* surfg の属性設定

surfg      平面曲線と空間曲面を描く際に用いる共通の設定.
root_finder 零点を計算する際の, 解法の指定.
            d_chain_bisection を用いると自己交差も綺麗に
            描きます.
iterations: 零点を計算する際の繰返しの上限を設定
width:      画像の横幅
height:     画像の高さ
*/
put(surfg, d_chain_bisection,root_finder);
put(surfg, 0.0000000001,epsilon);
put(surfg, 20000,iterations);
put(surfg, 500,width);
put(surfg, 500,height);

/* surf の属性設定

surf には空間曲面を描く際に用いる設定を入れます.
surf で色の指定は 0 から 255 までの整数で RGB を指定します.
do_background: 背景色の設定を許可 (yes)
background_red: 背景色の指定 (赤)
background_green: 背景色の指定 (緑)
background_blue: 背景色の指定 (青)
rot_x:          X 軸回りの回転角度 (rad)
rot_y:          X 軸回りの回転角度 (rad)
*/
put(surf, yes,do_background);
put(surf, 0,background_red);
put(surf, 0,background_green);
put(surf, 0,background_blue);
put(surf, 0.14,rot_x);
put(surf, -0.3,rot_y);
```

```

/* surfplot

引数は多項式. 多項式の変数は2個か3個でなければエラーに
なります. 描画は surf を用いますが, この函数では, 臨時ファイル
として surf.tmp に surf のスクリプトを書込み, system 函数で
surf -x surf.tmp を実行させます.
*/

surfplot(f):=block(
[
poly,poly0,vars,lls1:0,lls2:0,
f:ratsimp(expand(f)),tmp,
str,target,j,sl,obj,
ls1:properties(surfg),
ls2:properties(surf)
],
vars:showratvars(f),
n:length(vars),
display2d:false,
if n=2 or n=3 then
  (lls1:length(ls1[1])-1,
  for i:1 thru lls1 do
    (str:ls1[1][i+1],
    (if str=epsilon then tmp:rat(get(surfg,str))
    else tmp:get(surfg,str)),
    surf_settings[i]:str=tmp
    ),
  if n=3 then
    (lls2:length(ls2[1])-1,
    for i:1 thru lls2 do
      (str:ls2[1][i+1],
      j:i+lls1,
      surf_settings[j]:str=get(surf,str)
      ),
    /* 変数の入換を行います.*/
    poly0:subst([vars[1]=surf_tmp_x,vars[2]=surf_tmp_y,
    vars[3]=surf_tmp_z],f),
    poly:subst([surf_tmp_x=x,surf_tmp_y=y,surf_tmp_z=z],

```

```

        poly0),
    /* 曲面を描く為の設定 */
    target:surface,
    obj:draw_surface)
else
(
    /* 変数の入換を行います.*/
    poly0:subst([vars[1]=surf_tmp_x,vars[2]=surf_tmp_y],f),
    poly:subst([surf_tmp_x=x,surf_tmp_y=y],poly0),
    /* 曲線を描く為の設定 */
    target:curve,
    obj:draw_curve),
/* 配列 s1 の定義. 描画設定と曲線/曲面の方程式+描画命令を割当 */
j:l1s1+l1s2,
array(s1,j+2),
(for i:0 thru j-1 do
    s1[i]:surf_settings[i+1]),
s1[j]:target=poly,
s1[j+1]:obj,
/* 只今, 描画中... */
print("Surf is now drawing ", poly,". Please wait ...."),
/* stringout 関数で, スクリプトの式を書込みます. */
if n=2 then
    (stringout("surf.tmp",s1[0],s1[1],s1[2],s1[3],s1[4],
        s1[5],s1[6]))
else
    (stringout("surf.tmp",s1[0],s1[1],s1[2],s1[3],s1[4],
        s1[5],s1[6],s1[7],s1[8],s1[9],
        s1[10],s1[11],s1[12])),
/* system 関数で surf を起動します.XView 版の surf の場合は,
-x オプションを外すと動く様です */
system("surf -x surf.tmp")
)
else
/* 多項式が,2 変数でも 3 変数でもなければエラーを表示します. */
print("Error!")$

```

この関数を使うと,2 変数多項式や 3 変数多項式の零点のグラフが描けます. surfplot を実行すると,直ちに surf が立ち上ります. 描画は計算機の能力

と式の複雑さに依存する為、どの程度かかるかは一概に言えませんが、最近の Pentium 4 の 1GHz 程度でもあれば、30 秒もあれば描画が完了します。

描いた絵は surf の側で保存も出来ます。このような操作は描画ウィンドウ上で、マウスの右ボタンを押すと、制御用のウィンドウが出るので、そこで色々操作を行います。詳細は surf のマニュアルを参照して下さい。

surf の終了は描画ウィンドウ上でキーボードから q を入力すれば、surf を終了し、Maxima に制御が戻ります。

では、色々曲面や曲線を描いて遊んでみましょう。

2.6 Steiner のローマ曲面ギャラリー

Steiner のローマ曲面を描いてみましょう。ローマ曲面は $x^2y^2 + x^2z^2 + y^2z^2 - 17xyz = 0$ を満す点の集合で、この曲面は $f : (x, y, z) \rightarrow (xy, yz, zx)$ で球面を写したものです。この曲面は二次元の実射影空間の三次元空間への嵌め込みの一つです。二次元の射影空間は、Moebius の帯の境界となる円周に円盤を貼って出来る曲面に同相になりますが、この曲面は通常の三次元空間では実現出来ません。どうしても、自分自身が交わる点が出来てしまいます。その辺が面白い曲面です。

最初に、上記のファイル (surfplot.mc とします) を読み込みましょう。尚、Maxima の利用者定義の関数を含むファイルは load 関数で読み込みます。surfplot.mc が Maxima を起動したディレクトリ上にある場合、`load("surfplot.mc");` で Maxima に読み込みます。これでエラーも何も出なければ問題ありません。カレントディレクトリに surfplot.mc が無ければ、surfplot.mc を置いたディレクトリを直接指定します。この場合、相対パス (カレントディレクトリを基準とするパス) でも絶対パス (ルートディレクトリを基準とするパス) でもどちらでも構いません。

では、早速描いてみましょう。

`surfplot(x^2*y^2+x^2*z^2+y^2*z^2-17*x*y*z);` と入力して下さい。

すると、この関数はカレントディレクトリ上にデータファイル surf.tmp を生成し、surf に引渡し、surf が起動します。その結果、図 2.1 に示す絵が得られます。

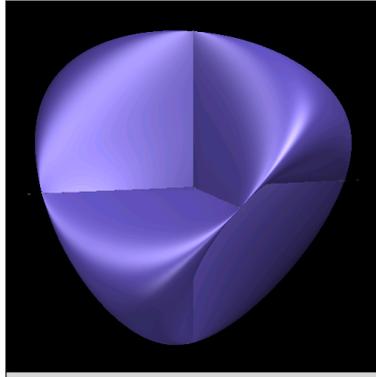


図 2.1: Steiner のローマ曲面

この絵で判る様に, ローマ曲面は X,Y,Z 軸で潰れた球面の様な形をしています. これだけでは何がどうなっているのか判りませんね. では, 曲面を平面で切った断面を見てみましょう. 平面による断面は Maxima+surf で簡単に見られます.

例えば, 平面の方程式が $z + ax + by - c = 0$ の形であれば, 変数 z に $c - ax - by$ を代入した式を描けば良いのです.

では,XY 平面 $z = 0$ による断面はどうなるのでしょうか? この場合, 断面の方程式として x^2y^2 が得られるので, 零点の集合として,X 軸と Y 軸が描かれてしまいます.

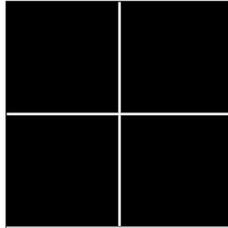
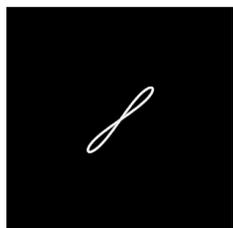


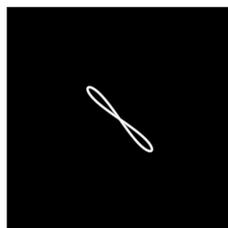
図 2.2: Steiner のローマ曲面,Z=0

以下に, z を $[-8,-2,-0.1,0,0.1,2,8]$ で描いた結果を以降に示しておきます.

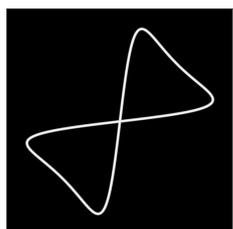
これらの断面図は単純に Z 軸を法線とする平面で描いたものですが, この曲面の面白さが出ています. $z = 0$ を除くと, 断面は 8 の字になっています. $z = 0$ に近付くにつれて, 8 の字の上下が潰れて, 交差点付近が XY 軸に貼り付き, $z = 0$ を通り過ぎると, 再び 8 の字が出現しますが, 今度は潰れた 8 の字が 45 度回転した形で出現します. この曲線で自分自身が交わる点で, 接ベクトルの次元は 2 になります.



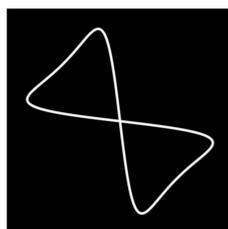
Steiner のローマ曲面, $Z=8$



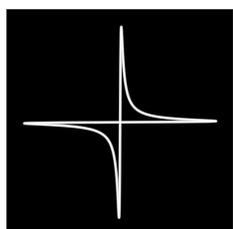
Steiner のローマ曲面, $Z=-8$



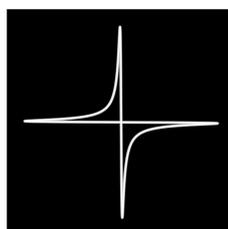
Steiner のローマ曲面, $Z=2$



Steiner のローマ曲面, $Z=-2$



Steiner のローマ曲面, $Z=0.1$



Steiner のローマ曲面, $Z=-0.1$

2.7 SINGULAR を使ってみよう

SINGULAR は可換代数、代数幾何と特異点理論に特化した計算機代数システムです。SINGULAR は C 風の処理言語があり、この処理言語で記述したライブラリで機能を拡張して行く事が出来ます。SINGULAR 自体には GUI やエディタ、更にはグラフ表示機能を持っていませんが、Emacs や Surf といった外部プログラムで代用が出来ます。

2.7.1 SINGULAR 初歩

SINGULAR の式の入力、Maxima と同様に、式の末尾にセミコロン ; を必ず付けます。セミコロンが無ければ、Maxima と同様に、SINGULAR は入力が継続していると判断して入力行の評価を行わず、プロンプトを ">" から "." にして入力待ちになります。

SINGULAR の対象には全て型があります。整数の和、差、積のみは、そのまま入力しても、結果が帰って来ますが、それ以外は最初に基礎環 (Base Ring) を定義しなければなりません。

ここで基礎環の定義は以下の様に行います。

——— 基礎環の定義方法 ———

```
ring <環の名前> = <係数体, (<変数1>, ..., <変数n>), <順序>;
```

ここで、係数体は実数 \mathbb{R} 、複素数 \mathbb{C} 、有理数 \mathbb{Q} 、整数 \mathbb{Z} 等が選べ、標数 p は 2147483629 以下の整数が指定出来ます。更に、指定可能な順序の代表的なものには、辞書式順序 (lp)、斉次辞書式順序 (Dp)、斉次逆辞書式順序 (dp) と重み付き逆辞書式順序 (wp) と重み付き辞書式順序 (Wp)、負辞書式順序 (ls)、負斉次逆辞書式順序 (ds)、負斉次辞書式順序 (Ds)、負重み付き逆辞書式順序 (ws) と負重み付き辞書式順序 (Ws) 等が選べます。この様に、Maxima と比べ変数順序は非常に充実しており、この点を利用したプログラムが書き易くなっています。

基礎環を定義すれば、その基礎環上の多項式 (poly) やイデアル (ideal)、更に商環 (qring) が扱える様になります。以下に多項式、イデアルと商環の定義方法を示しましょう。

——— 多項式、イデアル、商環の定義方法 ———

```
poly <多項式名> = <多項式>;  
ideal <イデアル名> = <多項式1>, ..., <多項式n>;  
ideal <商環名> = <イデアル>;
```

次に、商環 R/I の定義で指定するイデアル I は単純に `ideal` で定義したものも使えますが、その場合はイデアルが標準基底で無いと警告します。イデアルの標準基底は `std(<イデアル>);` で得られるので、この結果を用いませよ

う. 尚, 標準基底を計算する命令に `groebner` もありますが, `std` 命令と違って `groebner` 命令は環の順序に制約があります.

SINGULAR は入力された大文字と小文字を判別します. 一度定義した対象は, 名前を SINGULAR に入力すればその内容が表示されます. この点は Maxima と同じです.

次に各対象の定義例を示しておきましょう.

```
> ring r=0,(x,y,z),dp;
> poly unit_circle_eq=x^2+y^2-1;
> ideal unit_circle=x^2+y^2-1;
> unit_circle_eq;
x2+y2-1
> unit_circle;
unit_circle[1]=x2+y2-1
> ideal points=unit_circle_eq,y-x;
> points;
points[1]=x2+y2-1
points[2]=-x+y
> points[1];
x2+y2-1
```

この例では基礎環として $r = \mathbb{Q}[x, y, z]$ を定義し, それから多項式の `unit_circle_eq` とイデアルの `points` を定義しています.

この様に SINGULAR は Maxima と違い, 値の割当てでは `=` を使います. 対象の内容は単純に対象名を入力すれば多項式 `unit_circle_eq` の様に返されていますね. ここで, イデアルは番号付で返されるので, イデアルを生成する多項式を取り出したい時は, この例の `points[1]` の様にすれば一番目の生成元が取り出せます. 尚, 取り出した生成元の型は多項式 (poly) 型になります.

次に, 商環の定義例を示します.

```
> qring ri=std(x^2+y^2-1);
> ri;
// characteristic : 0
// number of vars : 3
//      block 1 : ordering dp
//              : names  x y z
//      block 2 : ordering C
// quotient ring from ideal
_[1]=x2+y2-1
> r;
```

```
// characteristic : 0
// number of vars : 3
//      block 1 : ordering dp
//      : names x y z
//      block 2 : ordering C
```

この例では前の基礎環 $\mathbb{Q}[x, y, z]$ 上で, 商環の定義に必要なイデアルを $std(x^2 + y^2 - 1)$; で与え, 商環 ri を $\mathbb{Q}[x, y, z]/\langle x^2 + y^2 - 1 \rangle$ で定義しています.

SINGULAR では環を複数定義する事が出来ませんが, 定義する毎にポイントが新規に定義された環に移動します. 多項式, イデアル, 写像等はポイントがある環上で定義されるので, 必要に応じてポイントを戻す必要があります. ポイントをある環に移す場合には, `setring` 命令を使います. その使い方は `setring <環の名前>;` と環の名前を直接指定するだけです.

SINGULAR では写像が扱えます. この写像の定義では名前が他の数式処理のものとは紛らわしい名前ですが, `map` 関数を用います. この場合, 対象が含まれる環を基礎環にした状態で写像の定義を行います. 以下に写像の定義例を示しましょう.

```
> ring r1=0,(x,y,z),dp;
> ring r2=0,(a,b),dp;
> map f=r1,a,b,0;
> f;
f[1]=a
f[2]=b
f[3]=0
```

この例では, 環 $r1$ と環 $r2$ を各々 $\mathbb{Q}[x, y, z], \mathbb{Z}[a, b]$ とし, 環 $r1$ から環 $r2$ への写像 $f(x, y, z) \rightarrow (a, b, 0)$ を写像の値域となる基礎環の変数に対して一つ一つ定めています. 基本的に, SINGULAR の写像の定義方法は, 基礎環の変数がどの様に写されるかを定めるだけです.

面白いのは, 定義した写像によるイデアルの逆像が計算可能な事でしょう. 例えば, 上の例の環 $r2$ のイデアル $i2$ の写像 f による逆像は `preimage(r1, f, i2);` で計算出来ます.

次の例ではイデアル $i2$ を零イデアル (0) として, 写像 f の核を計算するものです.

```
> ring r1=0,(x,y,z),dp;
> ring r2=0,(a,b),dp;
```

```

> map f=r1,a,b,0;
> ideal i2=0;
> setring r1;
> preimage(r2,f,i2);
_[1]=z

```

尚, この例では零イデアルを $ideal\ i2 = 0;$ で定義していますが, 単純に, $ideal\ i2$ としても同じ意味になります.

2.7.2 SINGULAR で surf を使ってみよう

SINGULAR はその処理言語を用いてライブラリが構築出来ます. このライブラリの読込は LIB 命令 (大文字) を用います. SINGULAR には様々なライブラリが附属していますが, 標準で附属するライブラリで, surfplot.mc の様に, SINGULAR で定義した関数を surf を用いて可視化する surf.lib があります. 実は, surfplot.mc の構成は, この surf.lib の内容を参考にしています.

この surf.lib を用いたローマ曲面の描き方の例を以下に示しておきます. 但し, Maxima と同じ事をしては面白く無いので, ここでは写像の定義と, その逆像を用いて表示する例を示しておきましょう.

```

> ring r=0,(x,y,z),dp;
> poly sp4=x2+y2+z2-16;
> ideal i1=sp4;
> map f=r,xy,yz,zx;
> ideal steiner=preimage(r,f,i1);
> steiner;
steiner[1]=x2y2+x2z2+y2z2-16xyz
> plot(steiner,"background_red=0;background_green=0;
. background_blue=0;rot_x=2;rot_y=0.5;");

```

この例では, 写像 $f : (x, y, z) \rightarrow (xy, yz, zx)$ による半径 4 の三次元球面の逆像を preimage で求め, plot に surf の背景色を設定する命令と一緒に引き渡しています. 尚, この背景色の設定が長い為に途中で改行を入れているが, SINGULAR は行末のセミコロンが未入力の為に, 行が継続中であると判断して, ピリオド. を出しています.

この様に, シュタイナーのローマ曲面は, 球面の写像 f による逆像としても得られます.

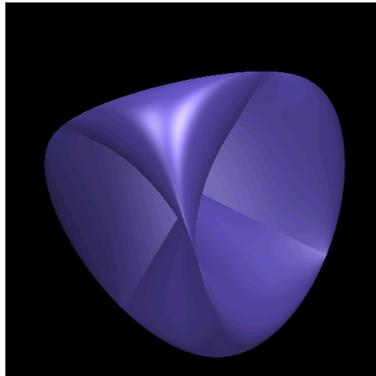


図 2.3: Steiner のローマ曲面

では, 今度は SINGULAR の方から, 曲線や曲面の新しい描き方を提案しましょう. 曲線や曲面が, 助変数表示されている場合, surf で絵を描く事は出来ません. この場合は, Maxima の描画函数の方が上手く描くかもしれません.

ところが, SINGULAR の eliminate+命令を併用して描く事が可能な場合があります. SINGULAR の eliminate 命令は不要なイデアルの変数を削除する命令です. 以下に, eliminate 命令を使って図 2.4 に示すデカルトの葉状曲線を描く例を示しましょう.

```
> ring r1=0,(x,y,t),dp;
> ring r2=0,(x,y),dp;
> map f=r1,x,y,0;
> setring r1;
> ideal i1=(1+t^3)*x-3*t,(1+t^3)*y-3*t^2;
> ideal i2=eliminate(i1,t);
> poly c2=i2[1];
> c2;
x3+y3-3xy
> setring r2;
> plot(f(c2));
```

この例では, 助変数を含めた環 $r1$ とそれを除いた環 $r2$, 更に, 環 $r1$ から環 $r2$ への写像 f を定義し, 次に, $r1$ 上でイデアル $i1$ を定義します. この葉状曲線は助変数 t を用いて $x = \frac{3t}{1+t^3}, y = \frac{3t^2}{1+t^3}$ で表現されていますが, 基礎環 $r1$ が $\mathbb{Q}[x, y, t]$ の為, イデアル $i1$ の定義で分子をかけた形にしています. 次の eliminate 命令では, 助変数 t を削ったイデアル $i1$ の表現を求めています. この eliminate 函数ではイデアルが返され, それをイデアル $i2$ と定義してい

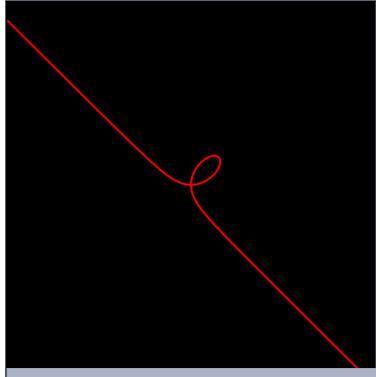


図 2.4: デカルトの葉状曲線

ます. 次に, イデアルは生成元を $c1$ に示す様に取り出せます. それから基礎環を `setring` 命令で環 $r2$ にし, 写像 f による $c1$ の像を `plot` 命令で描いた結果が図 2.4 になります. ここで環 $r1$ ではなく環 $r2$ 上で曲線を描いた理由は単純に `surf.lib` で, 環の変数の総数で曲線と曲面を判別する為, 環 $r1$ の変数が (x, y, t) の三変数の為, 曲面が描かれるからです.

2.8 Maxima で終結式を使ってみよう

ここで話を SINGULAR から Maxima に戻します. 先程の例では, パラメータ表示されたデカルトの葉状曲線を SINGULAR の `eliminate` 命令を用いて, x, y の多項式に変換しました. これと同じ事を Maxima するにはどうすればよいでしょうか?

SINGULAR の `eliminate` 関数は実際は終結式を用いています. 従って, Maxima で同じ事を行う為には, `eliminate` 関数を用いれば良いのです.

しかし, これでは幾ら何でも呆気ないので, `eliminate` 関数で利用されている多項式の終結式を使ってみましょう. Maxima では終結式の計算に `resultant` 関数が使えます. 又, `bezout` 関数を使って行列を求め, その行列の `determinant` を計算する方法もあります. ここでは色々と試してみましょう.

—— 終結式を求める resultant 関数 ——

`resultant(<多項式1>, <多項式2>, <変数>)`

`resultant` 関数は引数として, 二つの多項式と一つの変数を必要とします. ここで指定する変数は, 二つの多項式から消去したい変数になります. この終結式は, 二つの多項式を指定した変数の多項式として並び換えを行います.

Maxima では `resultant` 関数は `bezout` 関数と `determinant` 関数の合成で表

現可能です. ここではデカルトの葉状曲線を用いて順番に作業を追ってみましょう.

```
(%i1) p1:(1+t^3)*x-3*t;
(%o1) (t + 1) x - 3 t
(%i2) p2:(1+t^3)*y-3*t^2;
(%o2) (t + 1) y - 3 t
(%i3) expand(p1);
(%o3) t x + x - 3 t
(%i4) expand(p2);
(%o4) t y + y - 3 t
(%i5) m1:matrix([x,0,-3,x,0,0],
[0,x,0,-3,x,0],
[0,0,x,0,-3,x],
[y,-3,0,y,0,0],
[0,y,-3,0,y,0],
[0,0,y,-3,0,y]);
(%o5) [ x 0 - 3 x 0 0 ]
[ 0 x 0 - 3 x 0 ]
[ 0 0 x 0 - 3 x ]
[ y - 3 0 y 0 0 ]
[ 0 y - 3 0 y 0 ]
[ 0 0 y - 3 0 y ]
(%i6) expand(determinant(m1));
(%o6) - 27 y + 81 x y - 27 x
```

この例では, 最初に二つの多項式多項式 $p1 = (1 + t^3)x - 3t$ と $p2 = (1 + t^3)y - 3t^2$ を定義しています. ここで, 多項式 $p1$ と多項式 $p2$ を展開すると, 各々, $p1 = xt^3 + 0t^2 - 3t + x$ と $p2 = yt^3 - 3t^2 + 0t + y$ となります. 多項式 $p1$ と $p2$ の終結式は, この t の多項式と看做した場合の係数を並べたもの

で得られます. 両方の多項式の次数が 3 の為, 構築する行列 $m1$ は次の 6 次
の正方行列になります.

$$m1 = \begin{bmatrix} x & 0 & -3 & x & 0 & 0 \\ 0 & x & 0 & -3 & x & 0 \\ 0 & 0 & x & 0 & -3 & x \\ y & -3 & 0 & y & 0 & 0 \\ 0 & y & -3 & 0 & y & 0 \\ 0 & 0 & y & -3 & 0 & y \end{bmatrix}$$

この行列の行列式を計算したものが終結式になります.

ここで, この終結式には面白い性質があります. それは終結式と二つの多項
式の解の関係を示すものです.

一般的には, 変数 x を主変数とする多項式 f と g が以下で表現されるもの
とします.

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

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

この二つの多項式の終結式は次で表現されます.

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

ここで, 多項式 f と g の解を各々 α_i, β_j とすると, 多項式 f と g の終結式は
解 α_i, β_j を用いて以下の式に等しくなります.

$$\text{resultant}(f, g, x) = a_m^m b_n^n \prod_{1 \leq i \leq m, 1 \leq j \leq n} (\alpha_i - \beta_j)$$

この事は, 多項式 f と g が共通の解を持ては終結式が常に 0 になり, 同時
に, 終結式が 0 になるのは多項式 f と g が共通の解を持つ時に限る事を意味
します. ここでは, 零点集合を表現する多項式を計算するのが目的なので, 終
結式の零点集合と, 多項式 $p1$ と $p2$ が同時に 0 になる零点集合は一致しなけ
ればなりません. この性質を利用して, 終結式を使って助変数を排除している
のです.

尚, Maxima ではこの行列を効率良い行列で書き出す bezout 関数もあります.
この関数で, 上記の多項式 $p1$ と $p2$ の係数から構成される行列と, determinant
関数による結果を示しておきます.

```
(%i7) bezout(p1,p2,t);
          [ 0  3 x  - 3 y ]
          [                ]
(%o7)     [ 3 y  - 9  3 x ]
          [                ]
          [ - 3 x  3 y   0 ]
(%i8) expand(determinant(%));
          3                3
(%o8)     - 27 y  + 81 x y - 27 x
```

この計算を resultant 命令で一度に実行したものを以下に示します.

```
(%i5) p1:(1+t^3)*x-3*t;
          3
(%o5)     (t  + 1) x - 3 t
(%i6) p2:(1+t^3)*y-3*t^2;
          3                2
(%o6)     (t  + 1) y - 3 t
(%i7) i1:resultant(p1,p2,t);
          3                3
(%o7)     - 27 (y  - 3 x y + x )
```

これで,SINGULAR と定数項を除いて等しい多項式が得られました. ここで定数項は無視しても構いません. 実際, a を定数, $f(x_1, \dots, x_n)$ を多項式とする時, $f(x_1, \dots, x_n)$ の零点の集合と, $af(x_1, \dots, x_n)$ の零点の集合は一致するからです.

一寸, 脇道に逸れますが, 多項式 f の零点の集合を $V(f)$ と記述しましょう. すると, 多項式 $g(x_1, \dots, x_n)$ と多項式 $f(x_1, \dots, x_n)$ の積の零点集合 $V(fg)$ はどうでしょうか? これは, 多項式 f と g の零点の両方を含むので, $V(f) \cup V(g)$ になります. 又 f で割切れる多項式 h の零点集合 $V(h)$ に対しては, $V(h) \supset V(f)$ が成立します. 従って, 多項式 f で生成されるイデアル (f) を考えると, $V(f)$ がイデアル (f) に含まれる多項式の零点集合の中で最小の集合になります. このイデアルで考えてしまえば, 多項式が定数倍である事は大きな問題となりません. ですから, 多項式の零点集合を考える場合, そのイデアルを考えるのが最も手軽な手段になります.

次に, 応用で猿の腰掛と呼ばれる曲面を表示してみましよう. この曲面は助変数表示では以下の関係式を満すものです.

猿の腰掛の助変数表示

$$\begin{aligned}x - u &= 0 \\y - v &= 0 \\z - u^3 + 3uv^2 &= 0\end{aligned}$$

本当は、最後の式の変数 u, v を x と y で各々置換えてしまえば済む話なのですが、ここでは、終結式を使って猿の腰掛の変数 x, y, z の式に変換してみましょう。ここで、注意する事は、迂闊に、 $\text{resultant}(x-u, y-v, u)$ の様に、片方の式にしか存在しない変数を使ってはいけません。無意味な式が返って来るだけです。この例では、変数 u と v が含まれているのは、 $z - u^3 + 3uv^2 = 0$ のみなので、 resultant はこの多項式を中心に、例えば、 $x-u$ から開始し、 $y-v$ で終える手順になります。

```
(%i1) p1:x-u;
(%o1) x - u
(%i2) p2:y-v;
(%o2) y - v
(%i3) p3:z-u^3+3*u*v^2;
(%o3) z + 3 u v^2 - u^3
(%i4) a1:resultant(p1,p3,u);
(%o4) - z + x^3 - 3 v x^2
(%i5) a2:resultant(a1,p2,v);
(%o5) - z - 3 x y^2 + x^3
(%i6) surfplot(a2);
```

これで多項式として、 $-z - 3xy^2 + x^3$ が得られました。このグラフを surf で描いたものが図 2.5 に示す、尻尾の生えた猿に丁度良さそうな腰掛みたいな曲面が描かれます。

尚、Maxima には eliminate 関数が存在するので、実際は、

`$\text{eliminate}([x-u, y-v, z-u^3+3*u*v^2], [u, v]);$` で十分です。

試しに、動作を確認してみましょう。

```
(%i13) eliminate([x-u,y-v,z-u^3+3*u*v^2],[u,v]);
(%o13) [- z - 3 x y^2 + x^3]
(%i14) %[1];
(%o14) - z - 3 x y^2 + x^3
```

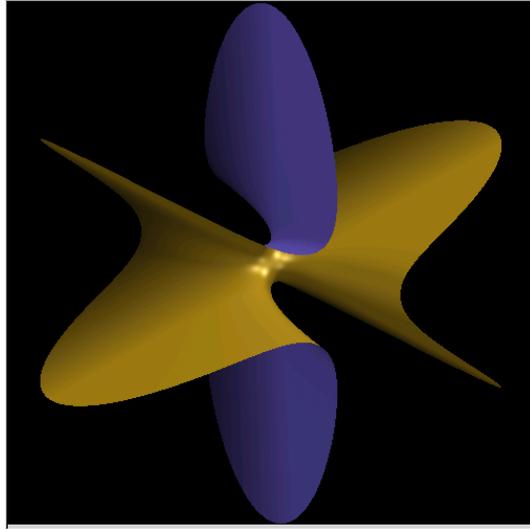


図 2.5: 猿の腰掛

(%o14) $-z - 3xy + x$

長々とした計算を行っていましたが,eliminate 函数を使えば,簡単に出来る事でした. ご苦労さまでした.

関連図書

- [1] 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.
- [2] 下地貞夫, 数式処理, 基礎情報工学シリーズ, 森北出版,1991.
- [3] 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.
- [4] ポール・グレアム,ANSI Common Lisp, ピアソン・エデュケーション,2002.
- [5] 本間龍雄, 組合せ位相幾何学, 共立出版,1980.
- [6] 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.
- [7] 村上順, 結び目と量子群, 数学の風景 3, 朝倉書店,2000.
- [8] 日本数学会編, 数学辞典 第3版, 岩波書店,1987.
- [9] D.Cox,J.Little and D. O'Shea,Ideals, Varieties, and Algorithms, UTM,Springer-Verlag,New York-Berlin,1992.
- [10] Gert-Martin Greuel, Gerhard Pfister, A Singular Introduction to Commutative Algebra, Springer-Verlag,New York-Heiderberg-Berlin,2000.
- [11] Knots and Linkx. Publish or Perish, Inc,1975.
- [12] Hal Schenck, Computational Algebraic Geometry London Mathematical Society student texts;58,2003.
- [13] Maxima の SOURCEFORGE の サイト,
<http://maxima.sourceforge.net/>