数式処理 *Bulletin of JSSAC(2012)*
Vol. 18, No. 2, pp. 139 - 150

特集: 数式処理システム 〜いま使うならこれだ〜

Maxima
– 長い歴史の中で進化し続ける数式処理システム –

横田博史*

Maxima is the one of the most old CAS. It's origin is MACSYMA from MAC-Project of MIT in 1960's. Maxima is written in Common Lisp, so it has a extensible features.

1 はじめに

Maxima は Common LISP で記述された汎用の数式処理システムである. その為に Common LISP が移植されている Microsoft Windows, Linux や Solaris 等の主要な OS 環境で動作する. Maxima は MAC Project の成果の一つである MACSYMA をその直接の起源とする. MACSYMA の開発は終了しているが, メーリングリストを中心に Maxima の保守・管理が行われ, パッケージの開発も盛んに行われている.

また, Maxima は他のアプリケーションの数式処理エンジンとしても用いられている. 例えば, 数値行列解析ツールの Euler MathToolbox や数式処理システム Sage で数式処理の中核として用いられており, Sage では Maxima を直接利用する事も可能である.

この Maxima の優れた入門書として中川義行氏による「Maxima 入門ノート」[1] を挙げておく. また, Maxima の内部表現やその処理の詳細については拙著「はじめての Maxima」[2] やその WEB 版 [3] を参照されたい.

2 フロントエンド

Maxima のフロントエンドには仮想端末を用いる **maxima**, GUI ベースの **wxMaxima** と **xMaxima** が標準的である. 特に wxMaxima は Maxima パッケージで標準のフロントエンドとなっている為に Maxima=wxMaxima と認識している人が多い. この wxMaxima は *Mathematica* 風のフロントエンドであり, 表示する数式のレンダリングも行う. 他に GNU Emacs を利用する **imaxima**, エディタの **TeXmacs** や KDE 環境で利用可能な **Cantor** があり, これらも入出力の数式のレンダリング機能を有する.

猶, ここでの紹介では仮想端末上の Maxima を用いた結果を利用する.

*ponpoko@cap.bekkoame.ne.jp

3 操作の概要

実際の Maxima の入力の様子を示しながら，基本的な操作を説明していく．

```
(%i1) 1+2;
(%o1)                                  3
```

Maxima の入力行のプロンプトの書式は ‘(%i⟨ 番号 ⟩)’ で, ⟨ 番号 ⟩ が行番号, ‘%i⟨ 番号 ⟩’ が行ラベルである．この入力に対応する出力行のプロンプトは ‘(%o⟨ 番号 ⟩)’ で, ‘%o⟨ 番号 ⟩’ が行ラベルである．Maxima の入出力はこれらのラベルに保存される為に利用者はラベルを直接指定することで入力式や結果の参照や再利用が容易に行える．ここでの例では入力ラベル “%i1” に入力式の ‘1+2’, 対応する出力ラベル “%o1” に結果の ‘3’ が保存されている．

```
(%i2) 2/3+3/5
;
                                      19
(%o2)                                 --
                                      15
(%i3) 2/3+3/5;
(%o3) 19/15
```

Maxima では入力行の末尾を示す記号としてセミコロン “;” か記号 “$” のどちらかを用いる．2/3 + 3/5 の計算例では行末にセミコロン “;” を付けなかった為に入力待ちとなっており, セミコロン “;” を入力した時点で Maxima は入力が終了したと判断し, その入力に対する計算結果を返している．セミコロン “;” が入力行末尾にある場合は処理結果を表示し, 記号 “$” が入力行末尾にあれば結果表示を行わない仕様となっている．

```
(%i4) (%o1)+(%o2)*15;
(%o4)                                  22
(%i5) quit();
```

Maxima では入出力にラベルが割り当てられる為, それらを再利用して計算処理を行う事が可能である．例えば, 式 ‘(%o1)+(%o2)*15’ の意味は, ラベル ‘(%o1)’ に表示された計算結果にラベル ‘(%o2)’ で表示された結果を 15 倍して両者の和を計算した結果を意味する．この様に Maxima ではラベルを履歴として用いた計算が可能で, 履歴を参照する大域変数や函数もある．例えば, 大域変数%に直前の結果が割当てられ．函数%th は ‘%th(n)’ で n 回前の計算結果の参照が行える．猶, 履歴の消去は ‘kill(labels)’ で行う．この kill 函数による消去でラベルに割当てた値は全て破棄され, ラベル番号は全て 1 に戻される．

Maxima の終了は ‘quit();’ で行う．この quit 函数は引数を必要としないが, ‘quit’ のうしろに小括弧 “()” は省略できない．省略すると Maxima はこの入力式を変数と判断し, その変数が束縛変数であれば割当てられた値, 自由変数であればその変数名を返却する．

オンラインマニュアルの参照は describe 函数で行い, example 函数で例題の閲覧が可能である．また, describe 函数はキーワード検索も可能である．この場合, 関連する項目の一覧を返却するので, その一覧から必要とする事項を番号で選べばオンラインマニュアルが表示される．

4 Maximaの数式

Maximaで処理可能な数式はCやFORTRANの数式とほぼ同様の書式である. 実際, 四則演算は和 "+", 差 "-", 積 "*", 商 "/" は他のプログラム言語と同様の演算子であり, 冪は記号 "^" や記号 "**" の双方が使える. また, 演算子は利用者が必要に応じて定義出来る. そして, Maximaの式を括る必要があるときは小括弧 "()" を用いる.

三角函数等の初等函数もCやFORTRANと同様の書式であり, 利用者が函数を定義する事も可能である. また, π 等の数学定数はMaximaでは記号 "%" を付けて通常の変数と区別する. 例えば, 円周率 π は '%pi', ネイピア数 e は '%e' と表記する.

Maximaのリストは '[1,2,3]' の様に演算子 "[ ]" の中に区切記号を "," で対象を列記したものとなる. リストの四則演算も可能だが, この場合は長さが同じリスト同士かスカラーとの場合に限定される. このリストを用いてMaximaの行列をmatrix函数で定義する事が可能である. 例えば, 'matrix([1,2,3],[4,5,6*x+1])' で2行3列の行列 $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6x+1 \end{pmatrix}$ が定義される. ここで行列の和と差はそれぞれ演算子 "+" と演算子 "-", 通常の行列積は演算子 "." を用いる. 猶, 演算子 "*" は成分単位の積となる.

変数への値の割当は記号 ":" を用い, 記号 ":=" は函数定義で用いる. 記号 "=" は通常の数式と同様の等値性を表現する比較の演算子となり, Cの演算子 "==" に対応する.

5 式とその評価

Maximaは大域変数simpの値が既定値のtrueであれば, 入力式を自動で評価し, その結果を返却する. ここでの自動評価は入力式中の項をMaximaの項順序に従って並び換え, それに伴う項のまとめといった簡単な簡易化, 入力式に含まれる函数項の処理である. より徹底した簡易化処理が必要であればexpand函数やev函数等に加えて文脈や規則といった仕組を適宜利用する必要がある.

Maximaの式の基本的な命令を幾つか紹介する. 先ず, 式の展開はexpand函数, 式を因子分解する場合はfactor函数, 有理式を整理する場合はratsimp函数, 三角函数や平方根が含まれる式の整理はtrigsimp函数, 指数函数や対数函数が含まれる式の整理はradcan函数を用いる.

```
(%i5) expand((1+x)^3);
                               3       2
(%o5)                         x  + 3 x  + 3 x + 1
(%i6) factor(%);
                                           3
(%o6)                                 (x + 1)
(%i7) ratsimp(-1/(x+1)+1/(x-1));
                                        2
(%o7)                                  ------
                                       2
                                      x  - 1
```

```
(%i8) trigsimp(sin(x)^2+3*cos(x)^2);
                                      2
(%o8)                            2 cos (x) + 1
(%i9) radcan(log(x*(x-1)));
(%o9)                          log(x) + log(x - 1)
```

より高度な式の評価を行える函数が ev 函数である. この ev 函数の構文は ev(〈 与式 〉,〈 条件$_1$〉,···,〈 条件$_n$〉) ' で, 与式に続く条件を逐次適用して評価を行う函数である. この ev 函数の代表的な使い方を以下に示しておく.

```
(%i9) A:sin(3*x)-cos(3*y)*z;
(%o9)                        sin(3 x) - cos(3 y) z
(%i10) ev(A,trigexpand=true,trigexpandtimes=true,eval,z=2);
                3                   2          3           2
(%o10)   - 2 (cos (y) - 3 cos(y) sin (y)) - sin (x) + 3 cos (x) sin(x)
```

この例では最初に変数 A に $(\sin(3x)\cos(3y))z$ を代入し, 三角函数の倍角公式による展開と変数 z に 2 を代入する処理を行っている. この ev 函数は Maxima の最上層（プロンプトが (%i〈 番号 〉) となっている階層）であれば, ev 函数の中身だけ（引数だけ）を記述する事が許容されている. この表記を使った例を示しておく.

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

この例では変数 B に $(\sin(2x)\cos(3y))(z+1)^3$ を代入し, 倍角公式の適用を trigexpand=true とする事で指示し, z=2 で変数 z への 2 の代入を指示している.

この ev 函数を利用すると方程式の求解後に結果を用いた計算が容易に行える. ここでは方程式の求解を行い, その結果を代入する例を示す.

```
(%i44) eq1:[x^2+y*x-1,x*y^2-x+y];
                                  2         2
(%o44)                   [x y + x  - 1, x y  + y - x]
(%i45) ans:algsys(eq1,[x,y]);
                                       sqrt(sqrt(2) + 2)
(%o45) [[x = sqrt(sqrt(2) + 2), y = - -----------------],
                                            sqrt(2)
```

```
                                 sqrt(sqrt(2) + 2)
[x = - sqrt(sqrt(2) + 2), y = -----------------],
                                       sqrt(2)
                                   sqrt(2 - sqrt(2))
[x = - sqrt(2 - sqrt(2)), y = - -----------------],
                                        sqrt(2)
                                sqrt(2 - sqrt(2))
[x = sqrt(2 - sqrt(2)), y = -----------------]]
                                       sqrt(2)
(%i46) x^2+y^2-1,ans[1]$

(%i47) trigsimp(%);
                                    3 sqrt(2) + 4
(%o47)                              -------------
                                         2
```

この例では連立方程式 $\begin{cases} x^2+yx-1=0 \\ xy^2-x+y=0 \end{cases}$ をリストを用いて表現して algsys 函数で解いている. この algsys 函数で求めた結果を ans に代入し, 'x^2+y^2-1,ans[1]' で解の最初の組を x^2+y^2-1 に代入する処理を行い, 最後に trigsimp 函数を用いて簡易化を行っている.

6 演算子

Maxima の演算子には被演算子と演算子の配置から,「**nary 型**」と「**infix 型**」の「**中置**」,「**前置 (prefix)**」,「**後置 (postfix)**」,「**matchfix 型**」と「**引数無し (nofix)**」の演算子がある. ここで中置演算子には nary 型と infix 型がある. nary 型の演算子は結合律を満す演算子であり, infix 型の演算子は結合律を満すとは限らない通常の二項演算子である. これらの演算子の違いは内部表現では明確となる. 実際, 和 "+" の内部表現の概形は '(+ a1 a2 a3)' の様に n-ary の演算子となるが, "∘" が infix 型であれば '(∘ a1 (∘ a1 a2))' の様に被演算子を二つのみ取る内部表現となる.

Maxima では利用者が必要に応じて演算子を定義することが可能である. ここで演算子の定義手順には二種類あり, 第 1 の方法は形式的な演算子を宣言する方法, 第二の方法は既存の函数を演算子として利用する方法である.

最初に infix 型の内挿表現演算子 "mike" を定義する例を示しておく.

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

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

最初の ‘infix("mike")’ で対象 “mike” を infix 型の中置演算子として宣言する. この宣言後に文字列 ‘x mike y’ は項として意味を持つ.

最初に函数を定義して, その函数を演算子として宣言する手法も次に示しておく.

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

Maxima の演算子には被演算子を引き付ける力が設定されている. 例えば, 数式 $2^3 \times 4 + 1$ は $((2^3) \times 4) + 1$ と解釈され, 決して $2^{3\times(4+1)}$ とは解釈されない. Maxima では 0 から 200 までの整数値で被演算子を演算子が引き寄せる力, 即ち, **束縛力**を表現する. この束縛力は演算子の宣言時に付与することが可能である.

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

この例では前置表現の演算子 “test:” を定義している. この場合は被演算子が左側のみに配置されるので左束縛力のみの設定となる. 最初の定義では既定値として 180 が自動的に設定され, この束縛力が和や差と比べて格段に大きいために ‘test:2+3’ を ‘(test:2)+3’ と Maxima は解釈する. 次に prefix 函数を使って束縛力を和や差よりも小さな 90 に設定した場合, ‘test:2+3-test:(2+3)’ の最初の演算子 “test:” の直後にある 2 は和の演算子 “+” に引き寄せられる為に ‘2+3’ が優先して処理されて差の演算子 “-” よりも演算子 “test:” の方が束縛力が弱い為に ‘2+3’ は差の演算子に引き寄せられる. 以上から, ‘test:2+3-test:(2+3)’ は ‘test:((2+3)-test:(2+3))’ として解釈される事になる.

7 属性

Maxima では対象に与えた属性を用いた処理が可能である. この属性は利用者が自由に与える事も可能であるが, Maxima で予め与えられた属性に対しては, その属性を持つ対象の処理を行う内部函数も準備されている. 該当する属性を持った対象の内部函数による処理の制御は大域変数を用いて行われる. Maxima に予め定義された属性は, 線形性, 可換性（対称性）, 歪対象性等の函数や演算子の性質, ev 函数による式の評価に関連するものがあり, これらの属性は declare 函数を用いて対象に付与される.

例えば, 前置型の演算子を定義して, その演算子に線形性を与える例を示す.

```
(%i2) prefix(Dx);
(%o2)                                  Dx
(%i3) declare("Dx",linear);
(%o3)                                done
(%i4) Dx(x+y);
(%o4)                             Dx y + Dx x
```

この例では prefix 函数で演算子 "Dx" が前置型である事を宣言し, 次の declare 函数で線形性 (linear) 属性を付与している.

この様な数学上の特性を表現する為に属性は使われているが, より自由な利用も可能である. 一例として Maxima の対象を TEX の式に変換する tex 函数を紹介しておく. この tex 函数は対象に予め設定された texword 属性値が存在した場合, その属性名を属性値で置換する働きがある. これによって Maxima の式は TEX の式に変換されるのである.

```
(%i27) expr1:expand((x+%pi*y)^3);
                       3  3        2    2          2      3
(%o27)              %pi  y  + 3 %pi  x y  + 3 %pi x  y + x
(%i28) tex(%);
$$\pi^3\,y^3+3\,\pi^2\,x\,y^2+3\,\pi\,x^2\,y+x^3$$
(%o28)                                false
(%i29) :lisp (get '$%pi 'texword)

\pi
(%i29) :lisp (get '$^ 'texword)

NIL
```

この例では $(x+\pi y)^3$ の展開結果を tex 函数で TEX の式に変換している. ここで '%pi' の texword 属性値を:lisp 函数を使って直接 LISP の S 式を評価させて調べると '\pi' である事が分る. そして, tex 函数に与えた式中の '%pi' は '\pi' に置換されている. しかし, 演算子 "^" は texword 属性値を持たない為, tex 函数は 'NIL' を返している.

8 文脈

文脈は一般的に与えられた命題を判断する上で必要な条件や仮定等の情報である. Maxima では仮定や条件を文脈に登録し, その文脈の上で式の解釈を行う事が可能である. Maxima の文脈は木構造を持っており, 根本の文脈が **global**, その子供で既定値の文脈が **initial** である. Maxima の文脈では子孫に相当する下位の文脈は, その上位の文脈の内容を継承する. 従って, 文脈 global に与えられた情報は, 文脈 initial でも継承されるので, 文脈 initial 上でもそのまま利用可能である. 但し, 文脈 initial のみに与えた情報は文脈 global からは参照出来ない.

文脈の切替は大域変数 context に利用する文脈名を割り当てる事で行う. また, 文脈の生成は, 文脈 global の直上の文脈を生成するのであれば newcontext 函数, 既存の文脈の直上の文脈を生成するのであれば supcontext 函数, 文脈の削除は killcontext 函数で行う.

```
(%i1) context;
(%o1)                                initial
(%i2) newcontext(test1);
(%o2)                                 test1
(%i3) newcontext(test2);
(%o3)                                 test2
(%i4) context;
(%o4)                                 test2
(%i5) supercontext(test3,test2);
(%o5)                     supercontext(test3, test1)
(%i6) context;
(%o6)                                 test3
(%i7) killcontext(test2);
(%o7)                                 done
```

ここでの例では, 文脈 test1 と test2 を文脈 initial の子文脈として newcontext 函数を使って生成し, 文脈 test3 を文脈 test1 の子文脈として supcontext 函数を使って生成している. そして, 最後に killcontext 函数で文脈 test2 を削除している.

仮定や条件の文脈への登録は assume 函数, 削除は forget 函数で行ない, 登録した内容の確認は facts 函数で行なう. このときに文脈に登録可能な論理式は二項関係を表現した式に限定される.

数学記号	Maxima の演算子/函数	例
=	equal	equal(x,y)
≠	notequal	notequal(x,y)
≥	>=	x >= y
>	>	x > y
≤	<=	x <= y
<	<	x < y

表 1: 文脈に登録可能な論理式例

内部で演算子 “<=” の演算子項と演算子 “<” の演算子項は被演算子の左右の入替えによって, 演算子 “>=” の演算子項と演算子 “>” の演算子項として文脈に登録される. 例えば, 述語 ‘x <= y’ は述語 ‘y >= x’ , 述語 ‘x < y’ は述語 ’y > x’ として置換されて登録される. また, 二つの対象の同値性と非同値性を示す論理式では演算子 “=” と演算子 “#” が文脈では使えないために, それぞれを函数 equal と函数 notequal を用いた論理式で置き換える必要がある. また, 演算子 “not” を用いた論理式は自動的に演算子 “not” を持たない同値な論理式に変換される.

文脈を使った簡単な式の判断の例を示す. 最初に文脈 test1 と文脈 test2 を生成する.

```
(%i1) newcontext(test1);
(%o1)                                test1
(%i2) supcontext(test2,test1);
(%o2)                                test2
```

ここでは文脈 initial の子文脈として文脈 test1 を newcontext 函数を用いて生成し, 文脈 test1 の子文脈として文脈 test2 を supcontext 函数を使って生成している. 次に文脈を指定して仮定を登録する例を示す.

```
(%i3) context:test1;
(%o3)                                test1
(%i4) assume(x>0);
(%o4)                               [x > 0]
(%i5) context:test2;
(%o5)                                test2
(%i6) assume(x>1);
(%o6)                               [x > 1]
```

ここででは大域変数 context に test1 を割り当てる事で文脈 test1 を利用する事を指定して assume 函数を用いて ‘x > 0’ を仮定している. 同様に文脈 test2 で ‘x > 1’ を仮定している. 次に文脈を用いた処理の様子を示す.

```
(%i7) context:test1;
(%o7)                                test1
(%i8) sqrt(x^2);
(%o8)                                 x
(%i9) sqrt((x-1)^2);
(%o9)                            abs(x - 1)
(%i10) context:test2;
(%o10)                               test2
(%i11) sqrt(x^2*(x-1)^2);
(%o11)                             (x - 1) x
```

```
(%i12) context:initial;
(%o12)                                initial
(%i13) sqrt(x^2*(x-1)^2);
(%o13)                           abs(x - 1) abs(x)
```

大域変数 context に test1 を割り当てる事で文脈 test1 上に移動し, その文脈 test1 上で ‘sqrt(x^2)’ を入力する事で $\sqrt{x^2}$ を計算させると ‘x’ が返却される. しかし, ‘sqrt((x-1)^2)’ の結果は ‘abs(x-1)’, 即ち, $|x-1|$ となる. 次に文脈 test2 で ‘sqrt(x^2*(x-1)^2)’ を計算すると文脈 test2 の上位の文脈である文脈 test1 の仮定 ‘x > 0’ と文脈 test2 の仮定 ‘x > 1’ から ’(x-1)*x’ が返される. 文脈 test1 と test2 の上位の文脈 initial には ‘x に関する仮定が一切無い為に ‘sqrt(x^2*(x-1)^2)’ の結果は ‘abs(x-1)*abs(x)’, 即ち, $|x-1||x|$ となっている事に注意されたい.

Maxima の文脈では大小関係に関して一寸した推論が可能である.

```
(%i7) newcontext(test1);
(%o7)                                  test1
(%i8) assume(a1<a2);
(%o8)                                [a2 > a1]
(%i9) supcontext(test3,test1);
(%o9)                                  test3
(%i10) assume(a2<a3);
(%o10)                               [a3 > a2]
(%i11) is(a1<a3);
(%o11)                                 true
(%i12) context:test1;
(%o12)                                 test1
(%i13) is(a1<a3);
(%o13)                               unknown
```

ここでの例では initial の子文脈として test1 と test2 を newcontext 函数で生成し, 文脈 test3 を文脈 test1 の子文脈として生成している. それから, 文脈を test1 に固定して assume 函数で ‘a1 < a2’ を登録し, 文脈 test3 に移動して ‘a2 < a3’ を登録している. ここで文脈 test3 は文脈 test1 の内容を継承する為に ‘a1 < a2’ が使え, 文脈 test3 の仮定 ‘a2 < a3’ から ‘a1 < a3’ が正しいと判断している. しかし, 文脈 test1 では ‘a1’ と ‘a3’ を繋ぐ仮定 ‘a2 < a3’ が欠落している為に ‘a1 < a3’ の判断が出来ない事に注意されたい.

9 規則

Maxima の式の評価には, ある決った式の並びに対してのみに変換処理を行う規則を設定する事が可能である. 最初に規則を定める為に用いる変数を matchdeclare 函数を使って定義し, let 函数等で規則を適用する項と適用後の式の関連付けを行う. この具体的な例を次に示しておく.

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

この例では tama という実態のない函数に対して ‘{tama(x)^2}’ を ‘{tama(2*x)+1}’ で置換する規則を定めている. 最初の matchdeclare 函数で, この規則を適用する項の変数が ‘_a’ である事を宣言しておく. 次の実際の規則の定義では規則を与える項の変数が予め matchdeclare 函数で宣言した変数を用いなければ, その規則の指定で用いた変数に限定された規則となってしまうので注意が必要である. それから let 函数を使って規則を定義する. 猶, 規則の適用は自動では行われない為, let 函数で定義した規則は letsimp 函数を使って展開する必要がある.

10 数値計算

Maxima で扱える数には, 「**整数**」, 「**有理数**」, 「**浮動小数点数**」(倍精度と多倍長の二種類) と「**複素数**」がある. 整数は ‘123’ のような ASCII 文字の “0” から “9” までの数字で構成された対象, 有理数は ‘2/3’ の様に二つの整数の商 “/” の式として表現される. 浮動小数点数は LISP の倍精度の浮動小数点数と Maxima 独自の多倍長浮動小数点数の二種類が存在する. ここで倍精度浮動小数点数の表記は ‘123.4’ の様に小数点を含む表記と ‘12.34e+1’ や ‘1.234e2’ の様な指数表記がある. これに対して多倍長の浮動小数点数は ‘12.34b+1’ や ‘1.234b+2’ といった指数を含む表記のみが許容される. 複素数はこれらの数 a, b と純虚数 ‘%i’ を用いて ‘a+b*%i’ と表記される Maxima の式として表現される.

多倍長浮動小数点数では必要に応じて利用者が浮動小数点数の精度を大域変数 fpprec で設定し, 表示桁数は大域変数 fpprintprec で設定する. 大域変数 fpprintprec が 0 であれば表示桁数は大域変数 fpprec の値に合せられる. 大域変数 fpprec の既定値は 16 と倍精度の浮動小数点数の精度であり, 大域変数 fpprintprec の値は 0 である. また, 多倍長浮動小数点数への変換は bfloat 函数を用いる. ここで多倍長精度の浮動小数点数の実例を示す.

```
(%i3) fpprec; fpprintprec;
(%o3)                                 16
(%o4)                                  0
(%i5) bfloat(1/3);
(%o5)                        3.333333333333333b-1
(%i6) fpprec:50$ bfloat(1/3);
(%o7)       3.3333333333333333333333333333333333333333333333333b-1
(%i8) fpprintprec:16$ bfloat(1/3);
(%o9)                        3.333333333333333b-1
```

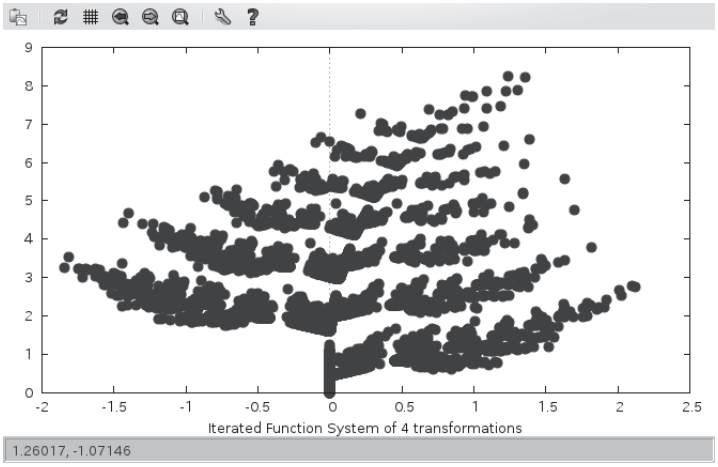
Maxima の多倍長浮動小数点数は LISP の多倍長浮動小数点数を使わずに自前で定義した独特の書式の多倍長浮動小数点数を利用する為に処理に時間がかかる欠点を持つ.

Maxima には数値行列向けの lapack パッケージがあるが, この lapack パッケージは倍精度の浮動小数点数に限定される. 又, LAPACK 自体, 単純に FORTRAN プログラムを Common LISP に変換したもので最適化が行われたものではない.

11 グラフィックスについて

Maxima はグラフ表示に外部アプリケーションを利用する. 標準の二次元描画の plot2d 函数や三次元描画の plot3d 函数では, gnuplot, openmath や Geomview が利用可能である. そして, 高機能の描画パッケージの draw パッケージでは gnuplot の利用が前提となっている. 又, dynamics パッケージを使えば力学系の二次元グラフやフラクタル図形の描画が可能である. ここでは dynamics パッケージを使って Barnsley の羊歯を描く例を示しておく.

```
(%i1) mt1:matrix([0.85,0.04],[-0.04,.85])$
(%i2) mt2:matrix([-0.15,0.28],[0.26,.24])$
(%i3) mt3:matrix([0.2,-0.26],[0.23,.22])$
(%i4) mt4:matrix([0.0,0.0],[0.0,.16])$
(%i5) p1:[0,1.6]$ p2:[0,0.44]$ p3:[0,1.6]$ p4:[0,0]$
(%i9) ifs([40,60,80,100],[mt1,mt2,mt3,mt4],[p1,p2,p3,p4],[0,0],10000);
```



参 考 文 献

[1] 中川 義行: Maxima 入門ノート 1.2.1, http://www.eonet.ne.jp/~kyo-ju/maxima.pdf, 2005.

[2] 横田 博史: はじめての Maxima, 工学社, 2006.

[3] _____: はじめての Maxima（WEB 版), http://www.bekkoame.ne.jp/~ponpoko/KNOPPIX/MaximaBook.pdf, 2012.