

Maxima 簡易マニュアル

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 10 月 27 日 (土)

MATLAB は The MathWorks の登録商標です。
Maple は Waterloo Maple Inc. の登録商標です。
Mathematica は Wolfram Research Inc. の登録商標です。
VMWare は WMWare Inc. の登録商標です。
Maxima 簡易マニュアル©(2007) 横田 博史著
この文書の内容の誤りなどによって起こった損害に対し、私は一切の責任を負いません。
この文書の誤りや御意見は以下にお知らせ下さい。連絡先:ponpoko @ cap.bekkoane.ne.jp (@を
半角に変更して下さい)

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ェアです。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ェアであり、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ェアの一つです。

この本は「はじめての Maxima」(工学社)の簡易マニュアルを基に、新たに、Maxima でのグラフィックス処理に関する章、stringproc パッケージと Maxima の改造の話を追加したものです。latxe2html による変換がどうやら上手く出来る様になったので、今度は絵付きになります。

今後、新しいパッケージの話の追加、更に、従来の例題や記述等を見直して行く予定です。

最後にお願ひになりますが、「はじめての Maxima」も宜しくお願ひします。

平成 19 年 10 月 勲武の三歳の誕生日を記念して。

横田博史

目次

第 1 章	Maxima の処理原理について	1
1.1	順序	2
1.1.1	Maxima の変数と順序	2
1.1.2	変数順序の変更	4
1.1.3	順序に関連する関数	5
1.1.4	関数を含めた順序	6
1.2	文脈の概要	8
1.3	属性の宣言と属性値	14
1.3.1	declare 関数	14
1.3.2	Maxima に用意された属性	16
1.3.3	put 関数による属性値の指定	20
1.3.4	変数宣言	21
1.3.5	atvalue 関数による函数値の設定	24
1.3.6	属性と属性値の削除	29
1.3.7	属性の表示	29
1.4	演算子	32
1.4.1	Maxima の演算子について	32
1.4.2	演算子の束縛力	33
1.4.3	演算子の型	35
1.4.4	演算子の属性を宣言する関数	36
1.4.5	数式演算子	40
1.4.6	論理演算子	43
1.4.7	割当の演算子	44
1.4.8	その他の演算子	45
1.5	規則と式の並びについて	46
1.5.1	規則の定義と適用	49
1.5.2	規則の削除	54
1.5.3	規則に関連する大域変数	54
1.6	評価	56
1.6.1	ev 関数	56
1.6.2	評価に関連する関数	64
1.7	LISP に関する関数	66
1.7.1	Maxima と LISP	66

第 2 章	Maxima のデータとその操作	69
2.1	数値	70
2.1.1	Maxima で扱える数値について	70
2.1.2	数値に関連する大域変数	70
2.1.3	Maxima の定数	71
2.1.4	数値に関連する関数	72
2.2	多項式	75
2.2.1	多項式の一般表現	75
2.2.2	多項式の CRE 表現	77
2.2.3	一般表現と CRE 表現への変換を行う関数	78
2.2.4	有理式の CRE 表現	79
2.2.5	係数体について	84
2.2.6	tellrat 関数	85
2.2.7	Horner 表記	89
2.2.8	多項式に関する関数	90
2.3	式について	102
2.3.1	変数や文字列の内部表現	102
2.3.2	二項演算の内部表現	103
2.3.3	割当の内部表現	104
2.3.4	Maxima の関数の内部表現	104
2.3.5	配列とリストの内部表現	105
2.3.6	Maxima の制御文の内部表現	105
2.3.7	表示式と内部表現	106
2.3.8	変数	108
2.3.9	部分式に分解する関数	111
2.3.10	部分式を扱う関数	114
2.3.11	総和と積	116
2.3.12	式の最適化	121
2.4	代入操作	123
2.4.1	通常の代入関数	123
2.4.2	substpart 関数と substinpart 関数	125
2.5	式の展開と簡易化	128
2.5.1	自動展開を行う大域変数	128
2.5.2	簡易化に関連する関数	130
2.5.3	指数関数の展開に関連する関数	130
2.5.4	式の展開に関連する関数	130
2.5.5	sum 関数の簡易化に関連する関数	132
2.5.6	簡易化を行う関数	133
2.5.7	簡易化に関する補助的関数	134
2.6	リスト	137

2.6.1	Maxima のリスト	137
2.6.2	リスト処理に関連する大域変数	137
2.6.3	リスト処理に関連する主な関数	138
2.6.4	map 関数族	142
2.6.5	map 関数族に関連する大域変数	143
2.6.6	map 関数いろいろ	144
2.7	配列	147
2.7.1	Maxima の配列について	147
2.7.2	配列操作に関連する関数	151
2.8	行列	152
2.8.1	行列について	152
2.8.2	行列を生成する関数	154
2.8.3	行列に関連する関数	159
2.8.4	eigen パッケージ	167
2.9	文字列	171
2.9.1	Maxima の文字列	171
2.9.2	ストリーム処理に関連する関数	173
2.9.3	文字の判別関数	174
2.9.4	文字列変換の関数	176
2.9.5	文字列操作の関数	177
2.9.6	評価関数を利用する文字列操作の関数	181
2.9.7	関連する大域変数	186
第 3 章	プログラム	187
3.1	Maxima でプログラム	188
3.1.1	block 文	188
3.1.2	if 文	189
3.1.3	do 文による反復処理	190
3.2	Maxima で関数の定義	195
3.2.1	関数定義	195
3.2.2	マクロの定義	198
3.2.3	apply 関数	201
3.2.4	最適化	202
3.2.5	関数定義に関連する関数	206
3.3	データ入出力について	208
3.3.1	データの入出力	208
3.3.2	ファイル処理に関連する関数	210

第 4 章	Maxima のシステム回りの関数	217
4.1	システムの初期化	218
4.1.1	maxima-init.mac ファイル	218
4.1.2	セッションの初期化	219
4.2	処理の中断	220
4.2.1	中断の制御文字	220
4.2.2	break 関数による中断	221
4.3	ラベルの参照	221
4.4	結果の表示	226
4.4.1	表示に関連する大域変数	227
4.4.2	式の表示を行う関数	229
4.4.3	後戻し表示を行う関数	231
4.4.4	エラー表示	232
4.5	記号?	232
4.6	システムの状態を調べる	234
4.6.1	大域変数 infolists	234
4.6.2	status 関数	235
4.6.3	room 関数	236
4.6.4	処理時間に関連する関数	236
4.7	外部プログラムの起動	237
4.8	式の変換を行う関数	238
4.9	システムに関連する大域変数	239
4.10	Maxima の終了	240
4.11	虫取りに関連する関数	241
4.11.1	trace 関数と untrace 関数	241
4.11.2	関連する大域変数	244
第 5 章	Maxima の関数	247
5.1	三角関数	248
5.1.1	三角関数一覧	248
5.1.2	三角関数に関連する関数	251
5.1.3	atrig1 パッケージ	252
5.2	指数関数と対数関数	253
5.2.1	指数関数と対数関数の概要	253
5.2.2	対数関数に関連する関数	254
5.3	代数方程式	256
5.3.1	Maxima の方程式	256
5.3.2	1 変数多項式方程式の場合	259
5.3.3	区間内の根の個数	261
5.3.4	多項式方程式の場合	261

5.3.5	一般的な方程式	264
5.3.6	漸化式の場合	266
5.4	極限	268
5.4.1	極限について	268
5.5	微分	272
5.5.1	微分に関する函数	272
5.5.2	vect パッケージ	275
5.6	積分	278
5.6.1	記号積分について	278
5.6.2	変数の変換について	281
5.6.3	有理式の記号積分	283
5.6.4	記号積分の結果検証	284
5.6.5	数値積分について	292
5.6.6	antid パッケージ	295
5.7	常微分方程式	296
5.7.1	常微分方程式の扱い	296
5.8	Maxima のグラフ表示	301
5.8.1	はじめに	301
5.8.2	表示アプリケーションについて	301
5.8.3	フロントエンド	302
5.8.4	plot2d 函数と plot3d 函数の動作	303
5.8.5	plot2d 函数	304
5.8.6	plot3d 函数	307
5.9	大域変数 plot_options	308
5.9.1	gnuplot に関連した函数	319
5.10	その他の描画函数	321
5.10.1	openplot_curves	321
5.10.2	contour_plot	323
5.10.3	Postscript に関連する函数	323
5.11	plot_option 以外の描画に関連する大域変数	325
5.12	gnuplot による描画について	326
5.12.1	maxout.gnuplot の内容について	326
5.12.2	set 命令	327
5.12.3	plot 命令による曲線の表示	328
5.12.4	splot 命令による曲面の表示	330
5.12.5	pm3d	331
5.12.6	等高線の階調の変更	336
5.12.7	等高線の調整	339
5.12.8	視点の変更	344
5.12.9	cbrange と cblabel	346

5.12.10	陰線処理	348
5.12.11	曲線と曲面の細かさの指定	352
5.12.12	描画の領域設定	353
5.12.13	ラベル表示と注釈に関連する事項	355
5.12.14	gnuplot の式	361
5.12.15	電卓としての gnuplot	365
5.12.16	プログラム言語としての gnuplot	366
5.13	plot2d 関数と plot3d 関数の活用事例	369
5.13.1	gnuplot_preamble の使い方	369
5.13.2	Maxima のバッチ処理	371
5.14	draw パッケージ	376
5.14.1	draw パッケージの概要	376
5.14.2	対象と属性について	376
5.14.3	draw 関数と draw2d 関数, draw3d 関数との関係	380
5.14.4	draw 関数	380
5.14.5	大域的な属性について	384
5.14.6	グラフの枠に関連する属性	386
5.14.7	対象について	388
5.14.8	環境の違いを調整する大域変数	397
第 6 章	Maxima の簡単な改造	399
6.1	使い勝手の向上を目指して	399
6.2	describe 関数は何処か	399
6.3	describe 関数の動作	400
6.4	ponpoko の仕様	401
6.5	大域変数の作り方	401
6.6	判別について	403
6.7	外部アプリケーションの立ち上げ方	403
6.8	完成	404
6.8.1	MS-Windows 環境の場合	405
6.9	大域変数の変更について	406

第1章 Maximaの処理原理について

この章では、Maxima の文脈、宣言と属性の設定、そして、評価の方法について述べます。これらの処理に関連する事項として、演算子と LISP に関連した関数についても簡単に述べます。

Maxima の大きな特徴の一つに、利用者が必要な述語や規則を定義し、その述語や規則を式に適用して式の評価を行うといった処理手順を踏む点です。これは単純に式を代入して関数で処理を行う場合に意識する事は然程ないでしょう。しかし、関数によっては式に含まれる成分に関する述語を付加したり、処理に適した宣言を行う事で、より円滑に処理が行える様になります。

更に、規則を上手く利用する事で作用素や代数的性質を付加する事が容易になります。この際に、作用素の定義で利用する Maxima の組込の演算子の特性を熟知していれば、より効果的な規則の適用が行えます。

1.1 順序

1.1.1 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.

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` 関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	$>_m$	<code>ordergreat</code> の第一引数	$>_m$...	$>_m$
<code>ordergreat</code> の最後の引数	$>_m$	<code>alphabetic</code> 宣言した文字 ₁	$>_m$...	$>_m$
<code>alphabetic</code> 宣言した文字 _n	$>_m$	頭文字が Z の変数	$>_m$...	$>_m$
頭文字が A の変数	$>_m$	頭文字が z の変数	$>_m$...	$>_m$
頭文字が a の変数	$>_m$	<code>orderless</code> の最後の引数	$>_m$...	$>_m$
<code>orderless</code> 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	$>_m$...	$>_m$
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 関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 $>_m$ は通常的大小関係 $>$ と同値です. 基本的に ordergreat 関数や orderless 関数で入れた順序に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 関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, $i = 1, \dots, k-1$ 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_k 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関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 x_1 から x_9 の場合, $x_9 >_m x_8 >_m \cdots >_m x_1$ となるので, 変数の並びは x_9, \dots, x_1 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 $x_1x_2^2x_8^3$ と $x_1x_2^2x_3x_9$ が与えられた場合, 二つの項を構成する変数 x_9, \dots, x_1 の順序に従って変数の並び換えを行います. その結果, $x_1x_2^2x_8^3$ は, $x_8^3x_2^2x_1$ に, $x_1x_2^2x_3x_9$ は $x_9x_3x_2^2x_1$ 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, $x_8^3x_2^2x_1$ は x_9 と x_3 が抜けているので $x_9^0x_8^3x_3^0x_2^2x_1$ になり, $x_9x_3x_2^2x_1$ は x_8 が無いので, $x_9x_8^0x_3x_2^2x_1$ 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的大小関係 $>$ 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, $x_1x_2^2x_3x_9 >_m x_1x_2^2x_8^3$ 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きます。

```
(%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。

1.1.2 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( <v1>, ..., <vn> )
orderless( <v1>, ..., <vn> )
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` 関数や `orderless` 関数で 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` 関数や `orderless` 関数で入れた変数順序は `unorder` 関数で無効にしない限り、これらの関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` 関数や `orderless` 関数で入れた変数順序で、Maxima 全体の変数順序の変更を伴う為、`ordergreat` 関数や `orderless` 関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。

1.1.3 順序に関連する関数

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関数として Maxima には `ordergreatp` 関数と `orderlessp` 関数の二つの述語関数があります。

順序を確認する述語関数

述語関数	true となる条件
<code>ordergreatp(<式₁>, <式₂>)</code>	<式 ₁ > が <式 ₂ > よりも大となる場合
<code>orderlessp(<式₁>, <式₂>)</code>	<式 ₁ > が <式 ₂ > よりも小となる場合

`ordergreatp` 関数と `orderlessp` 関数は、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関数です。尚、`ordergreatp` 関数と `orderlessp` 関数は LISP の `great` 関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つた方を大とします。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ひます。

この様に Maxima では変数の順序を `ordergreat` 関数や `orderless` 関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。SINGULAR の様な数式処理では、複数の順序を目的に応じて選択する事が出来ませんが、Maxima ではそうではありません。その点で Maxima は古風な数式処理システムです。

1.1.4 関数を含めた順序

ここまでは多項式の項順序について述べましたが、Maxima では通常が多項式に加えて \exp や \sin 等の Maxima 組込の初等関数にも順序が入ります。

この場合、同じ引数の関数の比較では関数名で $>_m$ による順序が入ります。次に、関数と他の変数では、関数に含まれる変数に主変数があれば、関数が変数よりも上位になりますが、関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、関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、利用者定義の関数の場合、一度 Maxima 側で値を解釈する為に `ordergreatp` 関数や `orderlessp` 関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;
(%o77)          neko(x) := if x < 0 then x2 else cos(x)3
(%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.

1.2 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。まず、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、このよう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。まず、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 >, ... )
forget( < 述語1 >, ... , < 述語n > )
forget( [ < 述語1 >, ... , < 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 >, ... )
deactivate( < 文脈1 >, ... )
killcontext( < 文脈1 >, ... )
newcontext( < 文脈 > )
supcontext( < 親文脈 >, < 子文脈 > )
```

まず、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 deactivate 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、どのような文脈があるのか判らなくなるかもしれません。Maxima の大域変数 context に現在、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

まず、どのような文脈が 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` 関数で表示しています。次に `context` 関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` 関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` 関数を使って生成しています。この `supcontext` 関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` 関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);
```



```

(%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` 関数で文脈 `mike` に入れています。`facts` 関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` 関数による宣言は `assume` 関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值性です。この正值性の設定では通常、`assume` 関数を用いて設定を行いますが、大域変数を用いて `assume` 関数による設定を省く事も可能です。

変数の正值性に関連する大域変数

変数名	初期値	概要
<code>assume_pos</code>	<code>false</code>	<code>assume_pos_pred</code> で指定した述語関数が <code>true</code> となる対象に対し、正值であると仮定。

まず、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` で指定した述語関数が `true` となる Maxima の式に対して、その正值性を設定します。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` 関数が `true` となる Maxima の式に対し、その正值性を設定します。尚、`assume` 関数による設定がある場合、その `assume` 関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` 関数による検査では `true` になります。ここで、大域変数 `assume_pos_pred` に `featurep` 関数による検査を行う関数を割当てます。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。

1.3 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。

1.3.1 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(<a1>, <属性1>, <a2>, <属性2>, ...)
declare(<a>, [<属性1>, ..., <属性n>])
declare([<a1>, ..., <am>], [<属性1>, ..., <属性n>])
```

declare 函数による宣言で $\langle a_i \rangle$ に $\langle \text{属性}_i \rangle$ 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 a_1 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 b_1, c_1 と d_1 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` 関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に、`declare` 関数で別個に宣言しても問題はありません。

`declare` 関数による宣言の影響は、文脈で利用される `assume` 関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` 関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` 関数は Maxima が持っている属性を与えるだけの関数ではありません。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関数の `featurep` 関数を使って調べられます。

述語関数 featurep

```
featurep(<アトム>, <属性>)
```

featurep 関数は、<アトム> が <属性> を持つかどうかを調べる関数です。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 関数で 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 の関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。

1.3.2 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 関数を使う事で、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, 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 1.6.1 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 `ans2` 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。このような場合、`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。

`alphanumeric` 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 `alphanumeric` 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

<code>declare($\langle a \rangle$,integer)</code>	$\langle a \rangle$ を整数として宣言
<code>declare($\langle a \rangle$,noninteger)</code>	$\langle a \rangle$ を非整数として宣言
<code>declare($\langle a \rangle$,even)</code>	$\langle a \rangle$ を偶数として宣言
<code>declare($\langle a \rangle$,odd)</code>	$\langle a \rangle$ を奇数として宣言
<code>declare($\langle a \rangle$,rational)</code>	$\langle a \rangle$ を有理数として宣言
<code>declare($\langle a \rangle$,irrational)</code>	$\langle a \rangle$ を無理数として宣言
<code>declare($\langle a \rangle$,real)</code>	$\langle a \rangle$ を実数として宣言
<code>declare($\langle a \rangle$,imaginary)</code>	$\langle a \rangle$ を純虚数として宣言
<code>declare($\langle a \rangle$,complex)</code>	$\langle a \rangle$ 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($\langle f \rangle$, additive)	函数 $\langle f \rangle$ の加法性
declare($\langle f \rangle$, multiplicative)	函数 $\langle f \rangle$ の乗法性
declare($\langle f \rangle$, outative)	函数 $\langle f \rangle$ と積の可換性
declare($\langle f \rangle$, linear)	函数 $\langle f \rangle$ の線形性
declare($\langle f \rangle$, commutative)	函数 $\langle f \rangle$ の対称性
declare($\langle f \rangle$, symmetric)	函数 $\langle f \rangle$ の対称性
declare($\langle f \rangle$, lassociative)	函数 $\langle f \rangle$ の左分配律
declare($\langle f \rangle$, rassociative)	函数 $\langle f \rangle$ 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, $f(x_1 + y_1, \dots)$ は $f(x_1, \dots) + f(y_1, \dots)$ 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, $f(x_1 * y_1, \dots)$ は $f(x_1, \dots) * f(y_1, \dots)$ 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, $f(a * x_1, \dots)$ は $a * f(x_1, \dots)$ 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, $f(x_1 + y_1, \dots)$ は $f(x_1, \dots) + f(y_1, \dots)$ に同値になり, 定数 a に対し, $f(a * x_1, \dots)$ は $a * f(x_1, \dots)$ 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 は歪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, $f(x, y, \dots) = -f(y, x, \dots)$ 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 は右結合律性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性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, 正值函数, 解析的函数等の性質を指定する事も可能です.

————— 関数の属性 —————

<code>declare($\langle f \rangle$,analytic)</code>	関数 $\langle f \rangle$ を解析的関数として宣言
<code>declare($\langle f \rangle$,increasing)</code>	関数 $\langle f \rangle$ を増加関数として宣言
<code>declare($\langle f \rangle$,decreasing)</code>	関数 $\langle f \rangle$ を減少関数として宣言
<code>declare($\langle f \rangle$,oddfun)</code>	関数 $\langle f \rangle$ を奇関数として宣言
<code>declare($\langle f \rangle$,evenfun)</code>	関数 $\langle f \rangle$ を偶関数として宣言
<code>declare($\langle f \rangle$,posfun)</code>	関数 $\langle f \rangle$ を正值関数として宣言
<code>declare($\langle f \rangle$,noun)</code>	関数 $\langle f \rangle$ を名詞型関数として宣言

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関数や大域変数は `ev` 関数を用いた評価で利用されます。これらの属性は全て `declare` 関数を用いて設定する事が可能です。

————— `ev` 関数で用いる `evfun, evflag` 属性 —————

<code>declare($\langle f \rangle$,evfun)</code>	関数 $\langle f \rangle$ に <code>evfun</code> 属性を付加
<code>declare($\langle a \rangle$,evflag)</code>	大域変数 $\langle a \rangle$ に <code>evflag</code> 属性を付加

詳しくは、1.6.1 節で述べますが、`ev` 関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` 関数内部でのみ `true` が設定され、又、`evfun` 属性を持つ関数を `ev` 関数の引数として与えると、`evflag` 属性を持った引数と、その関数と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関数に対して `declare` 関数を用いて宣言する事で、`ev` 関数の引数として与えられるようになります。

1.3.3 put 関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` 関数による宣言の他に、`put` 関数でアトムと属性に対応する属性値を指定し、`get` 関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関数 —————

<code>put(\langleアトム\rangle,\langle属性値\rangle,\langle属性\rangle)</code>
<code>qput(\langleアトム\rangle,\langle属性値\rangle,\langle属性\rangle)</code>
<code>get(\langleアトム\rangle,\langle属性\rangle)</code>

`put` 関数は \langle アトム \rangle に \langle 属性 \rangle で指定した \langle 属性値 \rangle 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` 関数で取出せます。

`put` 関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` 関数で確認出来ます。属性値の取出しは `get` 関数を用いて、アトムと属性を指定すると得られます。

`qput` 関数は `put` 関数に似ていますが、引数の評価を行わない点で `put` 関数と異なります。

`get` 関数は、 \langle アトム \rangle に設定された属性で、 \langle 属性 \rangle に対応する属性値を取出す関数です。属性値自体は `put` 関数で与えます。

\$RATIONAL

この例では変数 x_1 を整数型, x_2 と x_3 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 関数を用いて取出せます. この様に, mode_declare 関数は基本的に型の指定を行う関数です. mode_declare 関数で指定した変数と型は mode_identity 関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関数で, 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 関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` 関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` 関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` 関数は変数の宣言に加え、初期値と型の設定が行える関数です。但し、この関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` 関数の処理手順を以下に示しておきます。

- `mode_declare` 関数を用いて変数型を宣言します。
- `declare` 関数を用いて変数が `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。値の検証を行うには大域変数の `value_check` 属性に述語関数を割当てておく必要があります。この設定では `qput` 関数が有効です。
- 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` 関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` 関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関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関数 `ptest` を定義し、それから、`define_variable` 関数で大域変数 `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` 関数を用いて `check_value` 属性に `ptest` を与えます。この時、変数 `mike` に `15` を設定しようとするれば、`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。まず、`define_variable` 関数による宣言で、変数の型を `integer` と宣言しています。`define_variable` 関数は、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` 属性に内部関数の `assign-mode-check` 関数を割当てます。この内部関数は変数の `value_check` 属性に設定された関数を用いて変数の評価を実行する関数です。次に、この `value_check` 属性の設定で `qput` 関数を用いています。ここでの例では、大域変数 `tama` の `value_check` 属性に `ptest` 関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` 関数が、大域変数 `tama` の `check_value` 属性に割当てられた述語関数で割当てようとする値を評価します。この例では、`ptest` 関数に `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ようになります。

1.3.5 atvalue 関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` 関数等を用いる事で、函数の値の設定も属性で与える事が出来ます。

atvalue 関数

```
atvalue(<式>, <点リスト>, <境界値>)
at(<式>, <点リスト>)
```

atvalue 関数による境界値の設定では、関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 関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 $f(v_1, \dots, v_n)$ 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 関数で設定された値を printprops 関数を使って表示する際に、函数の疑似変数を表記する為に、記号@1,@2,... が用いられます。

```
(%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` 関数で設定した関数の値は `get` 関数で取出せず, 削除も `rem` 関数では出来ません.

```
(%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` 関数で関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` 関数は `atvalue` 関数と対で用います. \langle 式 \rangle を `atvalue` 関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` 関数で指定した値を入れて評価します.

部分式が `atvalue` 関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` 関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(⟨ 関数名 ⟩ (⟨ 変数1 ⟩, ..., ⟨ 変数m ⟩), ⟨ 式1 ⟩, ..., ⟨ 式n ⟩)
gradef(⟨ アトム ⟩, ⟨ 変数 ⟩, ⟨ 式 ⟩)
depends(⟨ 関数 ⟩, ⟨ 変数1 ⟩, ..., ⟨ 関数n ⟩, ⟨ 変数n ⟩)

```

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

引数が変数の総数 $m \cdot n$ 個の勾配 n よりも少ない場合, 最初の ⟨ 関数 ⟩ の i 番目の引数が参照されます。 x_i は関数定義で用いる疑似変数と同類で, 関数 ⟨ 関数 ⟩ 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 関数の引数は ⟨ 式 ⟩ が定義された関数ならばその関数が呼出されて結果が用いられます。勾配は関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 ⟨ 関数 ⟩ 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` 関数によって属性 `dependency` が付加され, 同時に大域変数 `dependencies` にも ⟨ 関数 ⟩ が追加されます。

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

`depends` 関数で, 関数の変数に対する従属を宣言します。例えば, `depends(f,x)` で関数 f が変数 x に従属する性質を付加します。尚, `depends` 関数による従属性の宣言にて, 関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 関数を実行していなければ,diff 関数を実行した時点で 0 になります. depends 関数で neko が変数 tama と mike に従属する関数と宣言した為に,微分を行っても零になりません. 最初の例では関数 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.

関数の変数に対する従属性は大域変数 dependencies に登録された関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	[]	勾配を定義された関数名リスト
dependencies	[]	従属性を宣言された関数名リスト

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

大域変数 dependencies には,depends や gradef で指定された関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 関数を用いる事で,gradefs や dependencies に関数名が蓄積されます. 尚,gradef を用いて dependencies に関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.

1.3.6 属性と属性値の削除

属性値の削除は `rem` 関数や `remove` 関数で行えます。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と一緒に削除されます。

属性や属性値を削除する関数

```
rem(<アトム>,<属性>)
remove(<アトム11nn>)
remove([<アトム1m>],[<属性1n>])
remove("<アトム>",operator)
remove(<アトム>,transfun)
remove(all,<属性>)
```

`rem` 関数は `<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` 関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11nn>)` 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` 関数で変換された LISP 関数が削除されます。この後は、Maxima 版の関数が用いられます。

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` 関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。

1.3.7 属性の表示

属性の表示に関連する関数としては、`properties` 関数、`propvars` 関数と `printprops` 関数があります。

属性の表示に関連する関数と大域変数

```
properties(<アトム>)
propvars(<属性>)
props
```

`properties` 関数は `<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` 関数でアトムに属性を指定し、`properties` 関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〉を持つアトムを生成します。例えば、`propvars(atvalue)` で atvalue 関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 関数, atvalue 関数や matchdeclares 関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,
funcsol, dimsum, ratsolve, prodshift, rform, 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, bessell, 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 の関数や変数が登録されています。

————— printprops 関数 —————

```
printprops(<アトム>, <属性>)
printprops([<アトム1>, ..., <アトムn>], <属性>)
printprops(all, <属性>)
```

`printprops` 関数は `<アトム>` と `<属性>` に対応する属性値を表示する関数です。`<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
<code>atvalue</code>	数値属性. <code>atvalue</code> 関数で与えられます。
<code>atomgrad</code>	勾配属性. <code>gradef</code> 関数で与えられます。
<code>gradef</code>	勾配属性. <code>gradef</code> 関数で与えられます。
<code>matchdeclare</code>	並びの照合変数属性. <code>matchdeclare</code> 関数で与えられます。

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)]
```



```
(%o33)                                     7
(%i34) 5 :/6;

                                     sin(5) + 5
(%o34) -----
                                     6

(%i35) 4 pochi 2;
(%o35)                                     16
```

この例では関数 (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関数とも看做せます.

1.4.2 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` 関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、 $a><c$ 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、 $a><c$ 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うまいでしょう。


```
(%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 の関数 `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関数となっているので問題ありませんが、次に LISP の `put` 関数を使って、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。

1.4.4 演算子の属性を宣言する関数

ここでは最初に演算子の属性を宣言する関数から述べましょう。Maxima では利用者が関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関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、値もきちんと関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)!);
(%o63)                                     m!
m C n := -----
n! (m - n)!
(%i64) 5 C 3;
(%o64)                                     10
```

演算子の属性を宣言する関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知れません.

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` 関数で演算子 `tama` を削除しています. ここで, `tama` の属性は `properties` 関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` 関数を使って演算子 `mike` 全てを削除しています. この様に `remove` 関数と `kill` 関数で削除する事が可能です.

尚, この例では `remove` 関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関数です.

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

演算子定義関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 `$any` に設定されます. 尚, `nary` 関数と `matchfix` 関数で宣言された演算子の被演算子型は他の関数と異なり, 左右の変数型属性 `lpos` や `rpos` ではなく, `argpos` 属性で指定し, その属性値は `$any` です.

1.4.5 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関数 `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 $x^{(n)}$ の様に $\langle \rangle$ 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
<code>dot0nscsimp</code>	<code>true</code>	<code>true</code> 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。
<code>dot0simp</code>	<code>true</code>	<code>true</code> の場合、零のスカラ項の非可換積を可換積に変換します。
<code>dot1simp</code>	<code>true</code>	<code>true</code> 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。
<code>dotassoc</code>	<code>true</code>	<code>true</code> 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。
<code>dotconstrules</code>	<code>true</code>	<code>true</code> 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 <code>dotocimo</code> , <code>dotonscsimp</code> , <code>dot1simp</code> も影響を与えます。
<code>dotdistrib</code>	<code>false</code>	<code>true</code> 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。
<code>dotexptsimp</code>	<code>true</code>	<code>true</code> 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。
<code>dotident</code>	<code>1</code>	非可換冪の 0 乗で返される値を設定します。
<code>dotscrules</code>	<code>false</code>	<code>true</code> 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 関数で 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関数です。

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 関数と呼ばれる関数で、 $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$ で定義されています。

$n!!$ は少し変わった関数で、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。この関数は、 $\prod_{i=0}^{\text{entier}(\frac{n}{2})} (n-i)$ 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 関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				

1.4.6 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

まず、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性に = 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

1.4.7 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 の関数 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 で関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関数で宣言した演算子の束縛力と同じ大きさですが、infix 関数等で右束縛力を 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。

1.4.8 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

1.5 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;` と入力してみてください。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よう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 >, ... )
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);
```

```
(%t10)          trigrule0 : tan(a) -> -----
                                     sin(a)
                                     cos(a)
```

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

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

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

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

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

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

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

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

disprule 関数を使って表示される規則は、まず、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. 実際の利用では関数 `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,置換を行う関数を必要とします. `disprule` 関数で表示される規則は `apply1,apply2,applyb1` 関数を用います.

apply 関数族

```
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` 関数族は基本的に `defrule` 関数で与えられた規則を式に適用させる函数ですが,`apply1` 関数と `apply2` 関数が式の木構造の上側から作用するのに対し,`applyb1` 関数が木構造の下側(Bottom)から作用させます.

最初の `apply1` 関数は与えられた `<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` 関数は `<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` 関数とは異なります. 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` 関数は `apply1` 関数と似ていますが,`apply1` 関数が `<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` 関数は, `<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` 関数,`apply2` 関数と `applyb1` 関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` 関数と `apply2` 関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` 関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` 関数を使って,関数 `tan` を含む式に Maxima に組込の規則 `trigrule0` を適用してみましよう. 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,関数 `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);
(%o20) sin(x)
-----
cos(x)
```

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

1.5.1 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` 関数や `let` 関数で行います.

ここでは最初に `defrule` 関数による規則の定義方法を示しましょう.

defrule 関数

```
defrule(<規則名>,<並び>,<置換>)
```

`defrule` 関数は与えられた `<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関数です.

この場合, `<規則名>` で指定された規則が `apply` 関数族の関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` 関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` 関数で宣言します. 次に, `dfx` が `dfx(a.b) → dfx(a).b+a.dfx(b)` となる規則 `chain1` を `defrule` 関数で定めます.

それから `apply1` 関数を用いて `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, その変数が満たすべき条件を定義していないからです. Maxima では変数が満たすべき述語が与えられなければ, `defrule` 関数や `let` 関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` 関数や `let` 関数で規則を定義する前に, 予め `<並び>` で用いる変数が満た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` 関数です.

matchdeclare 関数

```
matchdeclare(⟨変数⟩,⟨述語関数⟩,…)
matchdeclare([⟨変数1⟩, …,⟨変数n⟩],⟨述語関数⟩, …)
```

matchdeclare 関数は引数として ⟨変数⟩ と ⟨述語関数⟩ の変数と述語関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関数は、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関数です。述語関数として、特に指定する関数が存在しない場合は true、その他には、Maxima 組込の述語関数、lambda 関数や block 関数を含まない利用者定義関数が使えます。

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関数で、最後の引数に matchdeclare 関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 関数によって ⟨変数⟩ には matchdeclare 属性が付加されます。この属性は print-props 関数で参照する事が可能です。

matchdeclare 関数に与える変数と述語関数の対は、後で行う規則の定義で、変数が満すべき条件を述語関数を与え、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、述語関数を 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 関数で並び変数を予め定義し、それから defrule 関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` 関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` 関数で宣言した変数の可換積*や冪^を含む式に対しては, `let` 関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` 関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` 関数を用いる方が良いでしょう.

let 関数

```
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` 関数は `<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` 関数の引数の末尾に `<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. 各々の規則の組合せは任意数の `let` 関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前で参照されます.

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

letsimp 関数

```
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` 関数は `<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一結果が得られます.

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 関数と letsimp 関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 関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 関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 関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 関数と tellsimpafter 関数は各々規則に沿った置換を実行します。これらの関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 関数は、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 関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 関数、defmatch 関数、tellsimp 関数や tellsimpafter 関数で追加されたものです。

tellsimpafter 関数は tellsimp 関数と同様に <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 関数 —

```
defmatch(<関数名> , <並び> , <助変数1> , ..., <助変数n> )
```

defmatch 関数は 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関数を <関数名> で指定した名前で生成する関数です。

defmatch 関数の <並び> は変数と <助変数₁>, ..., <助変数_n> を含む式になります。ここで変数

が既に `matchdeclare` 関数で宣言されていたとしても, `defmatch` 関数の〈助変数 i 〉の引数として `defmatch` 関数に与えます.

関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` で構成された関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, 与えられた関数の線形性を調べる関数 `linear` の定義を行い, 関数 `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))` と同値な関数 `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` 関数を用いて宣言しています. この宣言の後に `defmatch` 関数で関数 `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関数 `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.

1.5.2 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, `remlet(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.

1.5.3 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
<code>maxapplydepth</code>	10000	<code>apply1</code> と <code>apply2</code> が停止する階層
<code>maxapplyheight</code>	10000	<code>applyb1</code> が停止する階層
<code>current_let_rule_package</code>	<code>default_let_rule_package</code>	利用規則パッケージ名
<code>letrat</code>	false	<code>letsimp</code> の動作に影響
<code>let_rule_packages</code>	<code>default_let_rule_package</code>	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_let_rule_package` が何時でも使えます.

大域変数 `letrat` が `false` であれば, `letsimp` 関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.

1.6 評価

1.6.1 ev 関数

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

ev 関数

$$\text{ev}(\langle \text{式} \rangle, \langle \text{引数}_1 \rangle, \dots, \langle \text{引数}_n \rangle)$$

$$\langle \text{式} \rangle, \langle \text{引数}_1 \rangle, \dots, \langle \text{引数}_n \rangle$$

ev 関数は与式 $\langle \text{式} \rangle$ 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 $\langle \text{引数}_1 \rangle, \dots, \langle \text{引数}_n \rangle$ 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、式を評価する関数 (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;
(%o3)              x . y . z
(%i4) (x.y).z,dotassoc:false;
(%o4)              (x . y) . z
(%i5) (x.y).z;
(%o5)              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 関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 関数内部で, 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, $(x \cdot y) \cdot z$ は $x \cdot y \cdot z$ に自動的に評価されます. ここで, ev 関数で評価を行う際に, 一時的に大域変数 dotassoc を false にしていません. ev 関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 属性を持つ関数である factor 関数と ratsimp 関数を使って評価しています. ここで evfun 属性を持たない関数を用いた場合, 式が評価されずにそのままの式が返されます.

更に, ev 関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 関数の性質で, 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 関数で利用可能な引数を示しましょう.

— ev 関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 属性を持つ関数	evfun 属性を持つ関数を, 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(\langle 整数 $_1$ \rangle , \langle 整数 $_2$ \rangle)	大域変数 maxposex に \langle 整数 $_1$ \rangle , maxnegex に \langle 整数 $_2$ \rangle 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 , ..., \langle x_n \rangle)	名詞型の微分で, \langle x_1 \rangle , ..., \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 , ..., \langle x_n \rangle)	ev 内部で用いる局所変数 \langle x_1 \rangle , ..., \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 関数では, 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 属性を持つ関数や, 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, infeas, %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 関数を用います. 更に, ある大域変数が evflag 属性を持つかどうかは, properties 関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 関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` 属性を取ります。`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 で二階微分を行う関数 `mike` を定義しています。この関数は `evfun` 属性を最初には持っていない為に `ev` 関数による評価が出来ません。ところが、`declare` 関数で `evfun` 属性を付加すれば、`ev` 関数による評価では `mike` が実行され、結果として二階微分が得られます。関数が `evfun` 属性を持つかどうかの判定は、`properties` 関数や、LISP の `get` 関数を用いて行えます。この点も `evflag` 属性と同様です。

`evfun` 属性を持つ関数が複数 `ev` 関数に与えられた場合、この `evfun` 関数の作用の順番は `ev` 関数の左端の `evfun` 関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` 関数の引数として、`evfun` 属性を持つ `factor` と利用者定義関数の `tst` を与えています。まずはじめに `ev` 関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` 関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` 関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;
```

```
(%o1) (x + 2)1001
```

```
(%i2) (x+2)^2/(x+1)^3,expand(2,3);
```

```
(%o2) 
$$\frac{x^2}{x^3 + 3x^2 + 3x + 1} + \frac{4x}{x^3 + 3x^2 + 3x + 1} + \frac{4}{x^3 + 3x^2 + 3x + 1}$$

```

```
(%i3) (x+2)^2/(x+1)^3,expand(2,2);
```

```
(%o3) 
$$\frac{x^2}{(x + 1)^3} + \frac{4x}{(x + 1)^3} + \frac{4}{(x + 1)^3}$$

```

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);
```

```
(%o45) 
$$\sin\left(\frac{\%pi}{10}\right)$$

```

```
(%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;
```

```
(%o48) 
$$2 e^x + .7853981633974483$$

```

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 関数に Risch 積分を実行させます。これは, integrate 関数内部で, risch が指定される場合, rischint 関数を用い, それ以外は sinit 関数を用いる様になっている為です。

derivlist は関数に似た構文を持っています。

derivlist

$$\text{derivlist}(\langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_k \rangle)$$

ev 関数は derivlist で指定した変数 $\langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_k \rangle$ 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
          ---- (y  + 2 x y  + x )
          dx dy
(%o9)
(%i10) a1,diff;
(%o10)          4 y
(%i11) a1,derivlist(x);
          d      2
          -- (2 y  + 2 x)
          dy
(%i12) a1,derivlist(y);
          d      3
          -- (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

$$\text{local}(\langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_k \rangle)$$

こちらは ev 関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 関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て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]);
(%o37) sin(sqrt(----- + 1))
                2
                %pi
                16
```

この例では, $\sin(x)$ の変数 x に 1 や $\frac{\pi}{4}$ 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て場合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` 関数や `solve` 関数等の方程式を解く関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
```

1.6.2 評価に関連する関数

評価に関連する述語関数

```
equal(<< 式1>>,<< 式2>>)
is(<< 述語 >>)
```

equal 関数は is 関数と一緒に使われ、 $\langle \text{式}_1 \rangle$ と $\langle \text{式}_2 \rangle$ が ratsimp 関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 関数は \langle 述語 \rangle 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 関数が true と返すのは, \langle 述語 \rangle 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 関数はエラーを出力を行い, false であれば unknown を返します.

————— 関数評価に関連する関数 —————

```
' $\langle$  式  $\rangle$ 
" $\langle$  式  $\rangle$ 
eval( $\langle$  式  $\rangle$ )
```

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;
(%o67)
%pi
---
4
(%i68) ''%i66;
(%o68)
1/2
2
----
2
(%i69) ''sin(test);
(%o69)
.7071067811865475
```

eval 関数は \langle 式 \rangle の評価を行います. LISP の eval 関数と同じ働きをします.

1.7 LISP に関する関数

1.7.1 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 は関数型と呼ばれるプログラム言語で、プログラムは様々な関数を定義し、それらを組合せて行く作業とも言えます。因みに、C や FORTRAN は手続き型と呼ばれます。

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` と入力してみてください。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 の関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` 関数を利用します。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 に戻りたいければ、`(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 `$a` に 1 を割当てています。それから `(to-maxima)` で Maxima に戻っています。`to_lisp` 関数の返却値は true です。最後に `a;` を入力すると、LISP で `$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、更には関数それ自体も 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 の関数名で先頭に?が付いているものが幾つか存在します. この様な関数は?を外した部分は LISP の関数で,Maxima から裏の LISP で処理させて, その結果を Maxima 側に持って来る関数です.?は通常の LISP の関数にも適応可能で?を頭に付けた関数は,Maxima 内部では?を外して,LISP の関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 の関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,?と関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.

?と似た関数に:lisp 関数があります. 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関数です.?演算子との違いは,?演算子では引数が LISP の関数で, 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関数です.

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 の関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 の関数を LISP から利用する場合, `mfuncall` 関数を用います. この場合, 引数は Maxima の関数名の頭に '\$ 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` 関数を使って 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あれば, 使い難いものです. 現実的には, Maxima 関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

この章では,Maxima の数値, 多項式, リスト, 配列, 行列, 及び文字列とそれらで構成される式について述べ, それらの直接処理を, 代入, 簡易化に分けて記述しています.

Maxima の式の内部表現に関しては式の節を参照して下さい. この式の内部表現は代入処理や式の簡易化を行う上で非常に重要で,Maxima の要の一つです.

尚, 比較的規模の大きな数値行列の処理に関しては余程の理由でも無い限り, Maxima よりも Octave 等の数値行列処理に長じたソフトを利用される事を勧めます.

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` 関数で行います。

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
<code>domain</code>	<code>real</code>	<code>[real,complex]</code>	多項式の係数環を指定
<code>float2bf</code>	<code>false</code>	<code>[true,false]</code>	<code>float</code> → <code>bigfloat</code> の際の警告の有無を指定
<code>fpprec</code>	16	正整数	<code>bigfloat</code> 型の桁数を指定
<code>fpprintprec</code>	0	正整数	<code>bigfloat</code> 型の表示桁数を指定
<code>m1pbranch</code>	<code>false</code>	<code>[ture,false]</code>	-1 の原始 n 乗根自動変換の有無を指定
<code>radexpand</code>	<code>true</code>	<code>[true,false]</code>	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ち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

<code>%e</code>	自然底 e
<code>%gamma</code>	Euler の定数
<code>%phi</code>	$\frac{1+\sqrt{5}}{2}$
<code>%pi</code>	円周率 π
<code>false</code>	Bool 代数の定数. 偽 (LISP の <code>nil</code>)
<code>true</code>	Bool 代数の定数. 真 (LISP の <code>t</code>)
<code>inf</code>	正の実無限大.
<code>minf</code>	負の実無限大.
<code>infinity</code>	複素無限大, 任意の偏角で無限大
<code>zeroa</code>	<code>0+.limit</code> 函数で利用
<code>zerob</code>	<code>0-.limit</code> 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 数値に関連する関数

数値に関連する述語関数

関数名	true となる条件
numberp	整数, 有理数, 浮動小数や可変長実数
bfloatp	bfloat 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

これらの述語関数は全て引数が一つです.

max と min

$\max(\langle \text{実数値}_1 \rangle, \langle \text{実数値}_2 \rangle, \dots)$	$\langle \text{実数値}_1 \rangle, \langle \text{実数値}_2 \rangle, \dots$ の最大値
$\min(\langle \text{実数値}_1 \rangle, \langle \text{実数値}_2 \rangle, \dots)$	$\langle \text{実数値}_1 \rangle, \langle \text{実数値}_2 \rangle, \dots$ の最小値

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

数値に関連する関数

関数名	変換前	変換後
bfloat	全ての数	→ bfloat 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 は全ての数と数値関数を bfloat 型に変換します.

isqrt 関数は引数〈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 関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 関数と entier 関数は〈数値〉が実数の場合, 〈数値〉を越えない最大の整数 n を返す関数で, 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 関数は, 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関数です。

cabs 関数, realpart 関数と imagpart 関数は各々複素数の絶対値, 実部と虚部を返す関数です。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 関数は与えられた複素数の偏角 θ を $\pi \geq \theta > -\pi$ の範囲で返す関数です。

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 由来の数値関数

?round 〈数値〉の最も近い整数
?truncate 〈数値〉の小数点以下を切り捨て

これらの数値関数は LISP の関数をそのまま利用します。その為、先頭に?が付きます。

?round 関数は 〈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 関数は 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;
(%o34)
                                     2      2
                                     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。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 $x1^2x8^2xx3$ の場合、単項式は $x1^2x3x8^2$ 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 … 項 _m)	項 _m $>_m$ … $>_m$ 項 ₁
単項式	((mtimes simp) 数値 変数 ₁ の冪 … 変数 _n の冪)	変数 _n $>_m$ … $>_m$ 変数 ₁

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;
(%o38)
          x
          -
          y
(%i39) :lisp $t3;
((MTIMES SIMP) $X ((MEXPT SIMP) $Y -1))
(%i40) t4:x^y;
(%o40)
          y
          x
(%i41) :lisp $t4;
```

(MEXPT SIMP) \$X \$Y)

Maxima では $x-y$ が $x+(-y)$, x/y が $x^{\wedge}(-y)$ 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 (Canonical 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 関数や ratsimp 関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関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 $\langle \text{係数}_1 \rangle \langle \text{変数} \rangle^{\langle \text{次数}_1 \rangle} + \langle \text{係数}_2 \rangle \langle \text{変数} \rangle^{\langle \text{次数}_2 \rangle} + \dots$ 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 ———

$(\langle \text{変数} \rangle \langle \text{次数}_1 \rangle \text{係数}_1 \langle \text{次数}_2 \rangle \text{係数}_2 \dots)$

尚, この CRE 表現では, 次数に関し, $\langle \text{次数}_1 \rangle > \dots > \langle \text{次数}_2 \rangle > \dots$ 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, $\lambda x \cdot (3x^2 + (-1)x^0)$ 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も問題はありませぬ. 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 1.1 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 xyy 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, $xxz >_m xyy$ 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えます。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

<code>rat(<式>, <変数₁>, ..., <変数_n>)</code>	一般表現	⇒	CRE 表現
<code>ratdisrep(<式>)</code>	CRE 表現	⇒	一般表現
<code>totaldisrep(<式>)</code>	CRE 表現	⇒	一般表現

先ず、最初の `rat` 函数は与えられた \langle 式 \rangle 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、`ratvars` 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));
(%o1)/R/
          y - 1
          -----
                2
          (y - x) z  + 1
(%i2) r2: rat((y-1)/((x-y)*z^2+1));
(%o2)/R/
          y - 1
          -----
                2
          (y - x) z  - 1
(%i3) r3:rat((y-1)/(-(y-z)*x^2+1));
(%o3)/R/
          y - 1
          -----
                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);
(%o4)
          y - 1
          -----
                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 関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 関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び、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);
```

```
(%o4)/R/
```

$$\frac{x - 1}{5}$$

```
(%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 関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なく、正か負の有理数となる様に拡張されており、係数はそれ自身、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、浮動小数点の有理数に近似変換される事を防ぎます。尚、浮動小数点の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
(%o32) 5000001 x
-----
5000000
```

この例で示す様に, `ratsimp` 関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);
(%o18) R/
          2
      sqrt(2) x  + 2
      -----
          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
<code>ratdenomdivide</code>	<code>true</code>	分子の項の分離を制御
<code>ratexpand</code>	<code>false</code>	CRE 表現の展開を制御
<code>ratfac</code>	<code>false</code>	CRE 表現式の因子分解を制御
<code>ratsimpexpons</code>	<code>false</code>	<code>ratsimp</code> の自動実行を制御
<code>ratwtlvl</code>	<code>false</code>	近似の際の切捨てを制御
<code>ratweights</code>	<code>[]</code>	重みのリスト
<code>rootsconmode</code>	<code>true</code>	<code>rootscontract</code> 関数を制御

大域変数 `ratdenomdivide` が `false` であれば, `ratexpand` 関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` 関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はいけません.

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` 関数が実行されます.

大域変数 `ratwtlvl` は `ratweight` 関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` 関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()` 関数でそのリストが見られます.

大域変数 `rootsconmode` は `rootscontract` 関数の挙動を定めます. 大域変数 `rootsconmode` が `false` ならば, `rootscontract` 関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
$x^{1/2} * y^{3/2}$	false	$(x * y^3)^{1/2}$
$x^{1/2} * y^{1/4}$	false	$x^{1/2} * y^{1/4}$
$x^{1/2} * y^{1/4}$	true	$(x * y^{1/2})^{1/2}$
$x^{1/2} * y^{1/3}$	true	$x^{1/2} * y^{1/3}$
$x^{1/2} * y^{1/4}$	all	$(x^2 * y)^{1/4}$
$x^{1/2} * y^{1/3}$	all	$(x^3 * y^2)^{1/6}$

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 関数にも影響を与えます。但し、mod 関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、 $3 \equiv -2 \pmod{5}$, $4 \equiv -1 \pmod{5}$ となります。実際、 $3 - (-2) = 4 - (-1) = 5 \pmod{5}$ 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

整数を処理する関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` 関数、`ratsimp` 関数等の CRE 表現の式を扱う関数に大きく影響します。その他の関数では、`gcd` 関数も影響を受けます。

`factor` 関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関数もあります。例えば、`factor` 関数の場合、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` 関数に引き渡します。

```
(%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 関数

より徹底して代数的整数を利用する場合は `tellrat` 関数を用います。この `tellrat` 関数は最小多項式や等式に `tellrat` 属性を設定するもので、`ratsimp` 関数や `ratexpand` 関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` 関数は次の写像を Maxima に組込むものです。

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

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

— tellrat 関数 —

```
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` 関数が代数的整数を 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` 関数に与えた場合、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` 関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);
(%o22) [x2 + x + 1]
(%i23) tellrat(y^3+y^2+y+1);
(%o23) [y3 + y2 + y + 1, x2 + x + 1]
(%i24) tellrat();
(%o24) [y3 + y2 + y + 1, x2 + 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 表現が扱える関数で処理を行う必要があります.

例として, tellrat(a²-2, b²=c⁴) と入力した場合を示します.

```
(%i11) tellrat(a^2-2, b^2=c^4);
(%o11) [b2 - c4, a2 - 2]
(%i12) (a+2)^4, algebraic, expand;
(%o12) a4 + 8 a3 + 24 a2 + 32 a + 16
(%i13) (a+2)^4, algebraic, expand, ratsimp;
(%o13) 48 a + 68
(%i14) (b+c)^3, algebraic, expand, ratsimp;
(%o14) 3 c5 + b c4 + c3 + 3 b c2
(%i15) (b+c)^3, expand, ratsimp;
(%o15) c3 + 3 b c2 + 3 b2 c + b3
(%i16) algebraic:true;
(%o16) true
(%i17) ratexpand((b+c)^3);
(%o17) 3 c5 + b c4 + c3 + 3 b c2
(%i18) expand((b+c)^3);
(%o18) c3 + 3 b c2 + 3 b2 c + b3
```

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 関数や ratexpand 関数等の CRE 表現を内部で用いる関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 関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 関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 関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 関数

`untellrat($\langle x \rangle$)`

tellrat 関数で設定した属性は untellrat 関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);
(%o36)          2 3 2
          [c - b , a - 2]
(%i37) tellrat();
(%o37)          2 3 2
          [c - b , a - 2]
(%i38) untellrat(a);
(%o38)          2 3
          [c - b ]
(%i39) untellrat(b);
(%o39)          2 3
          [c - b ]
(%i40) untellrat(c);
(%o40)          []
(%i41) tellrat(a^2-2,b^3=c^2);
(%o41)          3 2 2
          [b - c , a - 2]
(%i42) untellrat(c);
(%o42)          3 2 2
          [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 $y^2 + 1 = 0$ を満た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 $y^2 + 1 = 0$ を満たす代数的整数) 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 関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りません. 又, 多変数の場合, untellrat 関数は主変数 (mainvar) 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ぬ。untellrat 関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 則に基づ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 $a_n X^n + a_{n-1} X^{n-1} + \dots + a_1 X + a_0$ とすると、 $X((\dots(a_n X + a_{n-1}) + \dots) + a_1) + a_0$ 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      3      2      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(\langle 式 \rangle)	\langle 式 \rangle が有理式の場合
ratp(\langle 式 \rangle)	\langle 式 \rangle が拡張 CRE 表現の場合

ratnump 函数は \langle 式 \rangle が \langle 有理数式 \rangle であれば true, それ以外は false を返します.

ratp 函数は \langle 式 \rangle 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.

多項式の係数を取り出す函数

coeff(\langle 式 \rangle , \langle 変数 \rangle , \langle 次数 \rangle)
ratcoef(\langle 式 $_1$ \rangle , \langle 式 $_2$ \rangle , $\langle n \rangle$)
ratcoef(\langle 式 $_1$ \rangle , \langle 式 $_2$ \rangle)
bothcoef(\langle 式 \rangle , \langle 変数 \rangle)

coeff 函数は, \langle 式 \rangle に含まれる項 \langle 変数 \rangle^{\langle 次数 $\rangle}$ の係数を求めます. \langle 次数 \rangle を省略すると次数は 1 が設定されます. \langle 変数 \rangle 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りません. ここで \langle 変数 \rangle^{\langle 次数 $\rangle}$ 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があります. 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 \langle 式 $_1$ に含まれる項 \langle 式 $_2$ $\rangle^{\langle n \rangle}$ の係数を返します. $\langle n \rangle$ が 1 の場合は $\langle n \rangle$ が省略出来ます. 尚, 返却値には, \langle 式 $_2$ \rangle 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.

このよう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 \langle 式 $_2$ $\rangle^{\langle n \rangle}$ の係数を返す coef と異った答を返します.

例えば, ratcoef($(x+1)/y+x, x$) は $(y+1)/y$ を返却しますが, coeff は 1 を返します.

ratcoef(\langle 式 $_1$ \rangle , \langle 式 $_2$ \rangle , 0) は, \langle 式 $_1$ \rangle から \langle 式 $_2$ \rangle を含まない項の和を返します. その為, \langle 式 $_2$ \rangle が負の冪の項に含まれれば, ratcoef は使ってはいけません. \langle 式 $_1$ \rangle 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 \langle 式 \rangle 中の \langle 変数 \rangle 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 \langle 式 \rangle の残りとなります. 即ち, $[a, b]$ が返却値であれば, \langle 式 $\rangle = a * \langle$ 変数 $\rangle + b$ 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 函数の引数がどのような式であっても、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;
(%o30) z3 + y2 + x2
(%i31) showratvars(exp);
(%o31) [x, y, z]

```

多項式を纏める関数

```
factcomb(⟨ 式 ⟩)
fasttimes(⟨ 多項式1 ⟩, ⟨ 多項式2 ⟩n)
rootscontract(⟨ 式 ⟩)
```

factcomb 関数は ⟨ 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 関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であ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 関数を用いると $\max(n, m)^{1.585}$ のオーダーになります。

rootscontract 関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 関数は `abs` を `sqrt` に変換します。即ち、 $\text{abs}(x)*\sqrt{y}$ を $\sqrt{x^2*y}$ に変換します。

rootscontract 関数は `logcontract` 関数と似た手法で `ratsimp` 関数を用います。

有理式に関連する関数

```
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 関数は ⟨ 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 関数は ⟨ 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);
                1
(%o41)          1
(%i42) denom(1/2*x^2+1/2);
                1
(%o42)          1
(%i43) denom((x^2+1)/2);
                2
(%o43)          2
```

num 関数は有理式の分子 (NUMerator) を返します。

ratdenom 関数は〈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 関数を使いましょう。

ratnumer 関数は〈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 関数を使いましょう。

ratdiff 関数は〈有理式〉の微分を〈変数〉で行います。有理式に対しては diff 関数よりも処理が速く、計算結果は CRE 表現になります。尚、ratdiff 関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 関数を使いましょう。

終結式に関連する関数

```
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〉, …, 〈方程式n〉], [〈変数1〉, 〈変数2〉, …, 〈変数k〉])
```

bezout 関数は、〈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 関数と determinant 関数と組合せれば resultant 関数の代替になります。

resultant 関数は二つの多項式〈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。

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

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り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、

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

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

すると、

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

で終結式は計算出来ます. 尚, $\langle \text{多項式}_1 \rangle, \langle \text{多項式}_2 \rangle$ 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 $\langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_k \rangle$ を消去した $n-k$ 個の式のリストを返します. 最初の $\langle \text{変数}_1 \rangle$ は消去されて $n-1$ 個の式を生成し, $\langle \text{変数}_2 \rangle$ 以降も同様です.

$k=n$ の場合, 結果リストは k 個の $\langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_k \rangle$ 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;
                                - z + 5 y + 3 x - 1
(%o2)
(%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]
```


終結式のアルゴリズムを指定する大域変数

変数名	初期値	可能な値
resultant	subres	[subres, mod, red]

大域変数 resultant は同名の resultant 関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。

共通因子を求める関数

```
gcd(⟨ 式1 ⟩, ⟨ 式2 ⟩, ⟨ 変数1 ⟩, …)
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 ⟩, …)
content(⟨ 多項式 ⟩, ⟨ 変数1 ⟩, …, ⟨ 変数n ⟩)
```

gcd 関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 関数は多くの関数、例えば, ratsimp 関数や factor 関数等でも利用されています。

gcd 関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 関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 関数を利用
eez	eez gcd を利用
red	被約
smod	剰余
false	gcd 関数は常に 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 関数は 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 $c = a * \langle \text{多項式}_1 \rangle + b * \langle \text{多項式}_2 \rangle$ を満す多項式となります。この関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の一つを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、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、Euclid 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 関数は 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 関数は 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 関数は factorsum に似ていますが, factor の代わりに 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 関数は, 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 関数では ezgcd アルゴリズムが常用されています。

content 関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]
```

因子分解を行う関数

```
factor(〈式〉)
```

```
factor(〈式〉, 〈p〉)
```

```
factorsum(〈式〉)
```

```
sqfr(〈式〉)
```

```
factorout(〈式〉, 〈変数1〉, 〈変数2〉, …)
```

```
nthroot(〈多項式〉, 〈n〉)
```

factor 関数は $\langle \text{式} \rangle$ を整数上で既約因子に分解します. factor($\langle \text{式} \rangle, \langle p \rangle$) の場合, 最小多項式が $\langle p \rangle$ 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 関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 関数はグループ単位で $\langle \text{式} \rangle$ 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 関数は factor 関数と似ていますが, 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 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 $\gcd(a_i, a_j)=1$ を満たすものです. 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 関数は式を徹底して分解しているのに対し, sqfr 関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 関数は $\langle \text{式} \rangle$ を $f(\langle \text{変数}_1 \rangle, \langle \text{変数}_2 \rangle, \dots) * g$ 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 関数は与えられた整数係数の $\langle \text{多項式} \rangle$ が, ある整数係数の多項式を $\langle \text{正整数} \rangle$ 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.

この関数は factor 関数や sqfr 関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 関数で因子分解しない式のリスト
faceexpand	true	factor 関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 関数で因子分解しない式を登録します。尚、ratvars 関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 faceexpand は factor 関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 関数で 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 関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、式の各因子がある関数で保存されます。

剰余を計算する関数

```
divide(<多項式1>, <多項式2>, <変数1>, ..., <変数n>)
quotient(<多項式1>, <多項式2>, <変数1>, ...)
remainder(<多項式1>, <多項式2>, <変数1>, ...)
mod(<多項式>)
mod(<多項式>, <整数>)
```

divide 関数は <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 関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 関数は $\langle \text{多項式}_2 \rangle$ による $\langle \text{多項式}_1 \rangle$ の割り算の商を計算します。

remainder 関数は $\langle \text{多項式}_2 \rangle$ による $\langle \text{多項式}_1 \rangle$ の剰余を計算します。

mod 関数は $\langle \text{多項式} \rangle$ 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($\langle \text{多項式} \rangle$) 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($\langle \text{多項式} \rangle, \langle \text{整数} \rangle$) で $\langle \text{多項式} \rangle$ の $\langle \text{整数} \rangle$ 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関数

```
ratexpand( $\langle \text{式} \rangle$ )
fullratsimp( $\langle \text{式} \rangle, \langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_n \rangle$ )
fullratsimp( $\langle \text{式} \rangle_n$ )
ratsimp( $\langle \text{式} \rangle$ )
ratsimp( $\langle \text{式} \rangle, \langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_n \rangle$ )
```

ratexpand 関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 $\langle \text{式} \rangle$ の展開を行います。これは $\langle \text{式} \rangle$ 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 関数は $\langle \text{式} \rangle$ 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 関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 $\exp:(x^{(a/2)+1})^2 \cdot (x^{(a/2)-1})^2 / (x^a - 1)$ に対し, ratsimp(exp) によって $(x^{(2 \cdot a) - 2} \cdot x^{a+1}) / (x^a - 1)$ が得られ, fullratsimp(exp) を実行すれば $x^a - 1$ が得られます。

ratsimp 関数は, 非有理的関数に対して, 有理的に $\langle \text{式} \rangle$ 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 の様に非有理的関数 (例えば, $\sin x^2 + 1$ を含みますが, ratsimp で非有理的関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($\langle \text{式} \rangle, \langle \text{変数}_1 \rangle, \dots, \langle \text{変数}_n \rangle$) で, ratvars に変数の $\langle \text{変数}_1 \rangle, \dots$ 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 にどのよう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
$"a"$	$\& a$
$'a$	$((\text{mquote}) \$a)$

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
$x+y$	<code>((mplus simp) \$x \$y)</code>
$x-y$	<code>((mplus simp) \$x ((mminus simp) \$y))</code>
$x*y$	<code>((mtimes simp) \$x \$y)</code>
x/y	<code>((mquotient simp) \$x \$y)</code>
$x.y$	<code>((mnctimes simp) \$x \$y)</code>
x^2 又は $x^{**}2$	<code>((mexpt simp) \$x 2)</code>
$x^{^^}2$	<code>((mncexpt simp) \$x 2)</code>

基本的に `変数1 演算 ... 変数n` は内部では `((演算名) 変数1 ... 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えます。

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.

関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 の関数の内部表現

MaximaX の内部表現：関数

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)

関数名には基本的に\$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%が先頭に付いた関数は Maxima で自動的に解釈される事はありません。

微分を行う diff 関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、多少勝手に違います。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えます。

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 ;	$\frac{a}{b}$	$a b^{-1}$
$\text{sqrt}(x)$;	$\text{sqrt}(x)$	$x^{1/2}$
$x * 4/3$;	$\frac{4x}{3}$	$\frac{4}{3}x$
$\text{exp}(x)$	$\%e^x$	$\%e^x$

まず、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 関数と inpart 関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関数として dispform 関数があります。

dispform 関数

```
dispform(< 式 > )
dispform(< 式 > ,all)
```

dispform 関数は < 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、式に一对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;
(%o25)                                     x
                                           -
                                           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 関数は与えられた CRE 表現もこの様な外部表現に変換します。

但し、dispform(\langle 式 \rangle) では、式に含まれた関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(\langle 式 \rangle ,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、 $f(\sqrt{\frac{y}{x}})$ 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、1.1 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関数 `freeof` があります。

変数の述語関数

`freeof(<変数1>, ..., <変数n>, <式>)` <変数_i> が <式> に存在しない場合
`lfreeof([<変数1>, ..., <変数n>], <式>)` <変数_i> が <式> に存在しない場合

この `freeof` 関数は <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、関数、或いは二重引用符””で括られた演算子が扱えます。

尚、`freeof` 関数は、`sum` 関数や `product` 関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` 関数も動作は `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
<code>values</code>	<code>[]</code>	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` 関数を用いて Maxima から削除する事が出来ます。

変数値の削除を行う関数

```
remvalue(⟨変数1⟩,⟨変数2⟩,…)
remvalue (all)
```

remvalue 関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 の割当てでは変数 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り出したりする関数が幾つか用意されています.

式中の変数を扱う関数

```
args(⟨式⟩)
listofvars(⟨式⟩)
```

args 関数は ⟨式⟩ が関数の場合, 関数の引数を返します. 但し, 多項式の CRE 表現から変数を取り出す showratvars 関数とは異なり, 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);
(%o23) y^2 + 2 sin(x) y + sin^2(x)
(%i24) args(%);
(%o24) [y^2, 2 sin(x) y, sin^2(x)]
```

尚、args 関数と substpart 関数の両方は大域変数 inflag の設定に依存します。

listofvars 関数は〈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 関数が返すリストに、これらの定数が含まれます。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 関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 関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 関数や product 関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。まず, `isolate` 函数と `disolate` 函数は指定した変数を含む式と含まない式に分解して表示します。

————— 指定した変数を分離して式を表示する函数 —————

```
isolate(⟨式⟩,⟨変数⟩)
disolate(⟨式⟩⟨変数1⟩, …,⟨変数n⟩)
```

`isolate` 函数は ⟨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);

          2 a
(%t16)

          2
(%o16)    x  + %t16 x + 2 x + %t14
(%i17) isolate((1+a+x)^2,x);

          a + 1
(%t17)
```



```
(%o17) 
$$(x + \%t17)^2$$

```

disolate 関数は 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 関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 of 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 関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);
(%o17) 
$$x^2 + 2 b x + 2 a x + b^2 + 2 a b + a^2$$

(%i18) isolate_wrt_times;
(%o18) false
(%i19) exp1:expand((a+b+x)^2);
(%o19) 
$$x^2 + 2 b x + 2 a x + b^2 + 2 a b + a^2$$

(%i20) isolate_wrt_times;
(%o20) false
(%i21) isolate(exp1,x);
(%t21) 
$$b^2 + 2 a b + a^2$$

(%o21) 
$$x^2 + 2 b x + 2 a x + \%t21$$

(%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 関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関数 partition と純虚数を含む項と含まない項に分解する関数 rectform があります.

—— 部分式に分解する関数 ——

```
partition(< 式 > , < 変数 > )
rectform(< 式 > )
```

partition 関数は与えられた < 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 関数は与式を $a+b*i$ 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);
(%o12)                                3          2
                                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やすいものですが, 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]);
(%o73)
      z      2
      cos(-) + sin(x + 2 x + 1)
      w

(%i74) part(expr,[1,4]);
(%o74)
      z
      cos(-) + x
      w

(%i75) expr2:x*y*z*sin(x^2+1);
(%o75)
      2
      x sin(x + 1) y z

(%i76) inpart(expr2,[1,4]);
(%o76)
      x z

(%i77) part(expr2,[1,2]);
(%o77)
      2
      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;
(%o49) (x + 1)3
(%i50) pickapart(exp,1);
(%o50) %t483
(%i51) exp2: expand((x+1)^3);
(%o51) x3 + 3 x2 + 3 x + 1
(%i52) pickapart(exp2,1);
(%t52) x3
(%t53) 3 x2
(%t54) 3 x
(%o54) %t54 + %t53 + %t52 + 1
```

2.3.11 総和と積

— Σ と Π —

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

product 関数は ⟨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 関数はエラー出力ではなく 1 を返します。評価は sum 関数と似ていますが、総和関数 sum とは異なる点は、product の簡易化は無い事です。

— product 関数を制御する大域変数 —

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

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


```
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;
```

```
(%o38)                               1
-----
1 - x
```

この様に `simpsum` が `false` の場合, $\text{sum}(x^n, n, 0, m)$ 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. $\text{sum}(x^n, n, 0, \text{inf})$ 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, $\text{sum}(f(x), x, a, b)$ の a, b が $a > b$ の場合, $\text{sum}(f(x), x, a, b)$ を $-\text{sum}(f(x), x, b-1, a+1)$ 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` 関数同士の積を纏めて `sum` 関数の入れ子にします.

```
(%i1) sum(f(x), x, 0, m)*sum(g(x), x, 0, n);
```

```
(%o1)
          m          n
      =====
      \             \
      (> 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 \min 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` 関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` 関数内部で疑似変数を生成する度に、一つずつ増加します。

```
(%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

—— 式の最適化を行う関数 ——

```
optimize(〈式〉)
```

optimize 関数は〈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));
```

```
(%o40)      block([%1, %2], %1 : x + 1, %2 : %1 , %e2 + %13 + --  
                                                    %2
```

```
(%i41) ans1:solve( x^4+x^3+3*x-1=0,x);
```

```
(%o41) [x = -  $\frac{\sqrt{5}}{4}$  -  $\frac{\sqrt{25 - \sqrt{5}}}{4}$  -  $\frac{1}{4}$  -  $\frac{1}{4}$ ,
```

```
           2 sqrt(2) 5  
  sqrt(5)  sqrt(25 - sqrt(5))  1  
x = -  $\frac{\sqrt{5}}{4}$  +  $\frac{\sqrt{25 - \sqrt{5}}}{4}$  -  $\frac{1}{4}$ ,
```

```
           2 sqrt(2) 5  
  sqrt(sqrt(5) + 25) %i  sqrt(5)  1  
x = -  $\frac{\sqrt{5}}{4}$  +  $\frac{\sqrt{25 - \sqrt{5}}}{4}$  -  $\frac{1}{4}$ ,
```

```
           2 sqrt(2) 5  
  sqrt(sqrt(5) + 25) %i  sqrt(5)  1  
x = -  $\frac{\sqrt{5}}{4}$  +  $\frac{\sqrt{25 - \sqrt{5}}}{4}$  -  $\frac{1}{4}$ ]  
           2 sqrt(2) 5
```

```
(%i42) optimize(ans1);

(%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 関数に影響する大域変数

変数名	初期値	概要
otimprefix	%	optimize 関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 関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、式の中の変数 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 $\text{式}_a / \text{式}_b$ 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 $\text{式}_a^{1/\text{式}_b}$ 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
<code>exptsust</code>	<code>false</code>	指数函数の代入操作を制御
<code>opsubst</code>	<code>true</code>	演算子に代入する事を抑制

大域変数 `exptsust` 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、`(opsubst: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 `opsubst` 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]));
```

```
(%o64)          2
                x  + r
                0
```

ratsubst 函数

```
ratsubst (< 式1>, < 式2>, < 式3>)
```

ratsubst 函数は *lang*le 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 (radsubstflag 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、ratsubst 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いません。〈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(\langle 式 $_1$ \rangle , \langle 式 $_2$ \rangle , \langle 正整数 $_1$ \rangle , ..., \langle 正整数 $_n$ \rangle)

substinpart(\langle 式 $_1$ \rangle , \langle 式 $_2$ \rangle , \langle 正整数 \rangle , ...)

substpart 函数は \langle 式 $_2$ \rangle から part 関数の様に \langle 正整数 $_1$ \rangle , ..., \langle 正整数 $_n$ \rangle で抽出した部分式に \langle 式 $_1$ \rangle を代入します。 \langle 式 $_1$ \rangle 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。

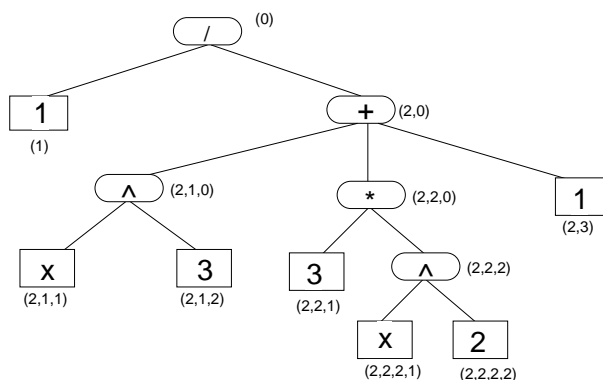


図 2.1: $\frac{1}{x^3+3x^2+1}$ の構造

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);
```

```
(%o7)
```

```

      1
-----
      3      2
x  + 3 x  + 1
```

```
(%i8) substpart(4,%,2,1,2);
```

```
(%o8)
```

```

      1
-----
      4      2
x  + 3 x  + 1
```

```
(%i9) substpart(1,%,2,2,2,2);
(%o9)

$$\frac{1}{x^4 + 3x + 1}$$

(%i10) substpart(x,%,1);
(%o10)

$$\frac{x}{x^4 + 3x + 1}$$

(%i11) substpart("^",%,0);
(%o11)

$$x^4 + 3x + 1$$

(%i12) substpart(sin(x),%,1);
(%o12)

$$\sin(x)$$

(%i13) substpart(y,%,2);
(%o13)

$$\sin(y)$$

```

substpart 関数は substpart 関数に似ていますが、substpart 関数と違い、式の内部表現に対して作用します。


```
(%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 $2.718\dots$ に変換されます. `%ex` 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` 関数とは別個に, 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
         x  + 4 x  + 6 x  + 4 x + 1
(%o42)
(%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(\langle 式 \rangle) \langle 式 \rangle が演算子を一つ持ち、簡易化函数が不明の場合

\langle 式 \rangle 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(\langle 式 \rangle)
 exponentialize(\langle 式 \rangle)

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 \langle 式 \rangle 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(\langle 式 \rangle , $\langle p \rangle$, $\langle n \rangle$)
 expand(\langle 式 \rangle)
 expan(\langle 式 \rangle , p, n)
 expandwrt(\langle 式 \rangle , \langle 変数 $_1$ \rangle , ..., \langle 変数 $_n$ \rangle)
 expandwrt_factored(\langle 式 \rangle , \langle 変数 $_1$ \rangle , ..., \langle 変数 $_n$ \rangle)

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 \langle 式 \rangle 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` 関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` 関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` 関数は、`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` 関数は `<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。この関数を使う為には予め `load(stopex);` で読込を実行します。

`expandwrt_factored` 関数は `expandwrt` 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関数

```
distrib(<式>)
multthru(<式12>)
multthru(<式>)
```

`distrib` 関数は可換積演算子 `*` 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` と最も上層の全ての和を展開する点でも異なります。

`multthru` 関数は `<式>` の部分展開を行います。即ち、`<式>` が $f_1 * f_2 * \dots * f_n$ 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12>)` 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。この関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));
```

```
(%o18)          2          2
          (x + 1) z + (x + 1)
```

```
(%i19) multthru((x+1)^2*(y+1)^2*(z+1)^2,z+1);
```

```
(%o19)          2          2          2          2          2          2
          (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);
```

```
(%o20)          2          2          2          2          2          2
          x (x + 1) (y + 1) (z + 1) + (x + 1) (y + 1) (z + 1)
```

```
(%i21) multthru((x+1)^2*(z+1)*(y+1));
```

```
(%o21)          2          2
          (x + 1) (y + 1) z + (x + 1) (y + 1)
```

```
(%i22) multthru((x+1)^2*(y+1)^2*(z+1)^2,x^2+1=0);
```

```
(%o22)          2          2          2          2          2          2
          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 関数の簡易化に関連する関数

sumcontract

```
sumcontract(<< 式 >>)
```

sumcontract 関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。sumcontract 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
      >  -- + >  --
      /      2  /      3
      ====  n  ====  n
      n = 1      n = 1

(%o18)

(%i19) sumcontract(%);
      m
      ====
      \      1  1
      >  ( -- + -- )
      /      2  3
      ====  n  n
      n = 1

(%o19)
```

———— 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、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、奇数が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せんが、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`(正值),`neg`(負値),`zero`(零) の何れか一つです。

numeval 函数

$$\text{numeval}(\langle \text{変数}_1 \rangle, \langle \text{式}_1 \rangle, \dots, \langle \text{変数}_n \rangle, \langle \text{式}_n \rangle)$$

numeval 函数は $\langle \text{変数}_i \rangle$ を $\langle \text{式}_i \rangle$ の数値変数として宣言し、 $\langle \text{式}_i \rangle$ 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, ... 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、 $\text{first}(x+y)$ 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 の影響を受ける関数

関数	概要
part	式の成分を抽出する関数
substpart	式の成分の代入を行う関数
first	リストの先頭を抽出する関数
rest	リストの残りを抽出する関数
last	リストの末尾を抽出する関数
length	リストの長さを返す関数
for-in 構文	リストを用いるループ文
map	リストに関数を作用させる関数
fullmap	リストに関数を作用させる関数
maplist	リストに関数を作用させる関数
reveal	式の置換えを行う関数
pickapart	成分を抽出する関数

2.6.3 リスト処理に関連する主な関数

リストに関連する述語関数

関数	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 関数と listp 関数で調べる事が出来ます。atom 関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 関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 関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(\langle リスト \rangle)	\langle リスト \rangle の長さを返却
copylist(\langle リスト \rangle)	\langle リスト \rangle の複製
reverse(\langle リスト \rangle)	\langle リスト \rangle の並びを逆にしたリストを返却
append(\langle リスト ₁ \rangle , \langle リスト ₂ \rangle , \dots)	複数のリストの結合を実行
econs(\langle 式 ₁ \rangle , \langle 式 ₂ \rangle)	\langle 式 ₂ \rangle を \langle 式 ₁ \rangle に追加
endcons(\langle 式 \rangle , \langle リスト \rangle)	\langle 式 \rangle を \langle リスト \rangle の後に結合
delete(\langle 式 ₁ \rangle , \langle 式 ₂ \rangle , $\langle n \rangle$)	\langle 式 ₁ \rangle を \langle 式 ₂ \rangle の先頭から $\langle n \rangle$ 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 $[a_1, a_2, a_3]$ に対して `reverse([a1, a2, a3])` を入力すれば $[a_3, a_2, a_1]$ を返却します。但し、各成分 a_i 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));
(%o45) sin(x) (y + x)
-----
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` 関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` 関数は `⟨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⟩` が関数, 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関数

```
first(⟨式⟩)
last(⟨式⟩)
rest(⟨式1⟩,⟨n⟩)
sublist(⟨リスト⟩,⟨函数⟩)
substpart(⟨x⟩,⟨式⟩,⟨n1⟩,⋯,⟨nk⟩)
```

`first` 関数は `⟨式⟩` の最初の成分を返します.

これに対し, `last` 関数は `⟨式⟩` の最後の成分を返します.

`rest` 関数, `first` 関数と `last` 関数は, 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` 関数は, `⟨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 関数は〈述語関数〉が true を返す〈リスト〉に含まれる成分を抽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 関数は〈式〉の〈 n_1 〉, …, 〈 n_k 〉で指定した成分を〈 x 〉で置換えます. 〈 x 〉は式, アトム, 演算子が指定可能です.

〈 n_1 〉, …, 〈 n_k 〉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);
(%o19)

$$\frac{x + 1}{\sin(x)} + e^x$$

(%i20) substpart(sin(x),expr,2,1);
(%o20)

$$\frac{\sin(x)}{x^2 + x + 1} + e^x$$

(%i21) substpart("+",expr,2,0);
(%o21)

$$e^x + x^2 + 2x + 2$$

```

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 `%e^x` 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`,`x^2+x+1`) となっています。Maxima の内部表現では式はリストで表現され、演算子が先頭の第 0 成分となり、以下にその引数が続く構造となっています。その為、`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 `x^2+2*x+2` で置換えられてしまいます。

2.6.4 map 関数族

`map` 関数は LISP ではお馴染みの関数で、基本的に関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、非常に便利な関数です。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` 関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` 関数のお仲間について解説します。

Maxima での `map` 関数には、`map`、`maplist`、`scanmap` の 3 個あります。これらの関数は全て Maxima の式に関数を作用させる働きがあります。この中で、`scanmap` のみが一つの関数を一つの式に作用させますが、`map` と `maplist` は `n` 個の引数を持つ関数に `n` 個の式に作用させる事が出来ます。

又、`map` 関数と `maplist` 関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` 関数は式的全階層の部分式に作用させる事が可能です。

先ず、`map` 関数の例を示します。

```
(%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なった結果になります。次の例では $\text{factor}(x*y+y)$ の結果に map 関数で \sin を作用させていますが、 $\text{factor}(x*y+y)$ 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 関数は基本的に map 関数と同様ですが、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 関数族に関連する大域変数

大域変数 `maperror`

変数名	初期値	概要
<code>maperror</code>	<code>true</code>	<code>map</code> や <code>maplist</code> 関数を制御します

大域変数 `maperror` は `map` 関数と `maplist` 関数の動作に影響します。まず、`map` 関数と `maplist` 関数の引数は $\langle \text{関数} \rangle, \langle \text{式}_1 \rangle, \dots, \langle \text{式}_n \rangle$ で、 $\langle \text{関数} \rangle$ は n 変数の関数です。ここで、大域変数 `maperror` の値が `true` の場合、各 $\langle \text{式}_i \rangle$ 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。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 $\langle \text{式}_i \rangle$ が同じ長さでなければ、最短の $\langle \text{式}_j \rangle$ を終えた時点で停止します。
2. $\langle \text{式}_i \rangle$ の主演算子が全て同じものでなければ、 $[\langle \text{式}_1 \rangle, \dots, \langle \text{式}_n \rangle]$ に $\langle \text{関数} \rangle$ 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 関数いろいろ

map 関数に関連する述語関数

```
mapatom(< 式 >)
```

`mapatom` 関数は, `< 式 >` が `map` 関数によってアトムとして扱われる時, `true` を返します

map 関数

```
map(< 関数 >, < 式1 >, ..., < 式n >)
maplist(< 関数 >, < 式1 >, < 式2 >, ...)
```

`map` 関数は n 個の式の列 `< 式1 >, ..., < 式n >` から n 個の成分を取り出し, n 個の引数を取る `< 関数 >` を作用させた結果を返します.

`maplist` 関数は `map` 関数に似ています. `< 式i >` の各成分に `< 関数 >` を作用させたリストを生成します. `< 関数 >` は関数の名前や `lambda` 式となります.

`maplist` が `map` 関数と異なる点は, `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, 主演算子の代わりに 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. 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);
(%o4)          2          2
          a  y + 2 a y + y + x
(%i5) scanmap(factor,exp);
(%o5)          2      2
          (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 函数を上の方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する為, 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);
(%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 関数に作用させる為に、入力値がそのまま返却されてしまいます。

```
(%i16) expand(exp);
```

```
(%o16)          2          2
              a y + 2 a y + y + x
```

```
(%i17) scanmap(factor,expand(exp));
```

```
(%o18)          2          2
              a y + 2 a y + y + x
```

scanmap(⟨ 関数 ⟩,⟨ 式 ⟩,bottomup) は scanmap(⟨ 関数 ⟩,⟨ 式 ⟩) とは逆に、内部表現の最下層から ⟨ 関数 ⟩ 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。まず、`array` 関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` 関数や `make_array` 関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(hash,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、この例のように直接指定する事で行います。他に、`fillarray` 関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。

配列を関数を用いて生成する場合、目的に応じて三種類の関数が使えます。まず、`make_art_q` 関数で生成する配列は LISP の `make-array` 関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` 関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` 関数は `array` 関数の様に配列名を与える必要はありません。これらの関数では共に、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関数として `listarray` 関数と `arrayinfo` 関数の二つがあります。

配列の情報を表示する関数

```
listarray(< 配列名 >)
arrayinfo(< 配列名 >)
```

最初の `listarray` 関数は配列データを表示し、`arrayinfo` 関数は配列の型や大きさを返す関数です。では、先程の例から、`listarray` 関数と `arrayinfo` 関数で配列の情報を調べてみましょう。

```
(%i6) listarray(a1);
(%o6)      [1, 10]
(%i7) arrayinfo(a1);
```

```
(%o7) [hashed, 2, [0, 3], [1, 2]]
(%i8) arrayinfo(a2);
(%o8) [declared, 1, [9]]
(%i9) arrayinfo(a3);
(%o9) [complete, 1, [5]]
```

まず, `listarray(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, 配列を生成する関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,⟨ 関数 ⟩,⟨ 型 ⟩,⟨ 整数1 ⟩,⋯,⟨ 整数n ⟩)
```

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

`array` 関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` 関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` 関数は大域変数 `use_fast_arrays` の影響を受けます.

use_fast_arrays

変数名	初期値	概要
<code>use_fast_arrays</code>	<code>false</code>	配列の種類を制限

まず, 大域変数 `use_fast_arrays` が `true` の場合,`make_array` 関数を用いて `any` 型の配列を生成します. 尚,`make_array` 関数で `any` 型の配列を生成する場合, LISP の `make-array` 関数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` 関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 `use_fast_arrays` がデフォルトの `false` であれば,`array` 関数は指定した ⟨ 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 関数で指定可能な型

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	関数型
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 関数は#####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 関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 関数を用いて中身を見ています。

make_array 関数は array 関数よりも複雑な配列が生成可能です。この make_array 関数で指定可能な〈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。

まず、any 型の場合、LISP の make-array 函数を用いて初期値 NIL の配列が生成されます。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関数

—— リストや配列の値を配列に割当てて関数 ——

```
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り出す関数 ——

```
arrayapply(<配列>, [<添字1>, …, <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関数 ——

```
rearray(<配列>, <次元1>, …, <次元n>)
```

rearray 関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。

—— 配列を削除する関数 ——

```
remarray(<配列1>, <配列2>, …)
remarray(all)
```

remarray 関数は関数を除去し, 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 関数を作用させた結果は〈関数〉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` 関数を用いて行列を生成します. この `matrix` 関数は Maxima の複数のリストから行列を生成する関数です. 構文は以下の様になっています.

————— 行列の定義を行う `matrix` 関数 —————

```
matrix(<リスト1>, ..., <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` 等の関数を用いて, `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` 関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. は見落とし易いので、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, $A \cdot B$ 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, 1.4.5 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 $\langle \text{整数}_1 \rangle \times \langle \text{整数}_2 \rangle$ 個の成分を入力して $\langle \text{整数}_1 \rangle$ 行 $\times \langle \text{整数}_2 \rangle$ 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.
```

```
(%o1) [ a 0 0 ]
      [      ]
      [ 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 関数は、〈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 関数は、〈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 関数を用いると、与えられた二次元配列から行列が生成出来ます。この genmatrix 関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関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 ]
      [ 1  -  - ]
      [ 2  3  ]
      [      ]
(%o4) [ 1  1  1 ]
      [ -  -  - ]
      [ 2  3  4 ]
      [      ]
      [ 1  1  1 ]
      [ -  -  - ]
      [ 3  4  5 ]
```

ematrix 関数は $\langle \text{整数}_1 \rangle$ 行 $\langle \text{整数}_2 \rangle$ 列の行列で、 $(\langle \text{整数}_3 \rangle, \langle \text{整数}_4 \rangle)$ 成分のみが $\langle x \rangle$ 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 関数は 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 $\langle \text{行} \rangle$ 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 ]
```

coefficientmatrix 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) coefficientmatrix([2*x-3*y-1=0,3*x+3*y+10=0],[x,y]);
      [ 2  -3 ]
(%o22) [      ]
      [ 3   3 ]
```

```
(%i23) coefficientmatrix([2*x-3*y-z,3*x+3*y+10*z],[x,y]);
      [ 2  -3 ]
(%o23) [      ]
      [ 3   3 ]
```

augcoefficientmatrix 関数は、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、 $\langle \text{方程式}_1 \rangle, \dots$ から構成される線形方程式系に含まれる $\langle \text{変数}_1 \rangle, \dots$ の係数から構築されます。coefficientmatrix 関数との違いは、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]);
```

```
(%o25)      [ 2  - 3  - z ]
            [          ]
            [ 3   3  10 z ]
```

```
(%i26) augcoefmatrix([2*x-3*y=1,3*x+3*y=10],[x,y]);
```

```
(%o26)      [ 2  - 3  - 1 ]
            [          ]
            [ 3   3  -10 ]
```

```
(%i27)
```

echelon 関数は、〈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)。

```
(%o2)      [2  1 - a  -5 b ]
            [          ]
            [a   b    c   ]
```

```
(%i3) echelon(d2);
```

```
(%o3)      [      a - 1      5 b      ]
            [1  - -----  - ----  ]
            [      2      2      ]
            [          ]
            [          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 列行列のスカラー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	行列に対する指数関数の処理
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, $\%e^{\text{matrix}([1,2],[3,4])}$ は $\text{matrix}([\%e, \%e^2], [\%e^3, \%e^4])$ となります。一般的に、この変換は $\langle \text{基底} \rangle^{\langle \text{次数} \rangle}$ の形式の変換に影響します。尚、 $\langle \text{基底} \rangle$ はスカラか定数の式であり、 $\langle \text{次数} \rangle$ 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関数を設定

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。関数名や 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。関数名や 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、作用させる関数や lambda 式をを指定します。

2.8.3 行列に関連する関数

行列に関連する述語関数	
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 関数は 〈式〉 が行列であれば true, そうでなければ false を返します。

diagmatrixp 関数は 〈式〉 が対角行列の場合に true を返します。

nonscalarp 関数は 〈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 関数が 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 $\langle m \rangle$ の各成分に \langle 函数 \rangle 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 > , ..., < リストn > )
```

```
addrow(< 行列 > , < リスト1 > , < リスト2 > , ..., < リストn > )
```

```
copymatrix(< 行列 > )
```

```
setelm(x , < i > , < j > , < 行列 > )
```

col 関数と row 関数は、〈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関数です。

```
(%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 関数は列として、addrow 関数は行として、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 関数は〈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 関数は〈行列〉の $(\langle i \rangle, \langle j \rangle)$ 成分を $\langle x \rangle$ 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 関数を使わなくても、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。

— 小行列を生成する関数 —

```
minor(〈 正方行列 〉, 〈 i 〉, 〈 j 〉)
submatrix(〈 行1 〉, …, 〈 行m 〉, 〈 行列 〉, 〈 列1 〉, …, 〈 列n 〉)
submatrix(〈 行1 〉, …, 〈 行m 〉, 〈 行列 〉)
submatrix(〈 行列 〉, 〈 列1 〉, …, 〈 列n 〉)
```

minor 関数は与えられた〈正方行列〉の $\langle i \rangle, \langle j \rangle$ 成分の小行列、つまり、〈正方行列〉から $\langle i \rangle$ 行と $\langle j \rangle$ 列を抜いた行列を返します。

submatrix 関数は〈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 関数は minor 関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 関数は、〈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。この関数は、ncharpoly で利用されています。ncharpoly は Maxima の charpoly の代りに使える関数です。尚、mattrace 関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関数

```
adjoint(〈 正方行列 〉)
invert(〈 正方行列 〉)
```

adjoint 関数は〈正方行列〉の余因子行列を計算します。

invert 関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, $\text{adjoint}(\langle \text{行列} \rangle) / \text{determinant}(\langle \text{行列} \rangle)$ 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 関数は、〈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関数 —————

```
charpoly(〈行列〉, 〈変数〉)
ncharpoly(〈行列〉, 〈変数〉)
```

charpoly 関数は〈行列〉の特性多項式 $\det(\langle \text{変数} \rangle I - \langle \text{行列} \rangle)$ を計算します。

determinant(〈行列〉 - diagsmatrix(length(〈行列〉), 〈変数〉)) と同じ結果を返します。

ncharpoly 関数は〈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 $\text{var} * \text{ident}[n] - a$ 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");
```

で読み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関数と、基底の直交化に関連する関数が含まれています。

ここで説明する関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、関数が読み込まれ、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` パッケージの関数は仮定します。その為, `knowneigvals` が `true` の場合は `listeigvals` に `eigenvalues` 関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 `listeigvlects` に保存されていると `eigen` パッケージの関数が仮定します。その為, `knowneigvlects` が `true` の場合は `listeigvlects` に `eigenvlects` 関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

内積関数

```
innerproduct( $\langle x \rangle$ ,  $\langle y \rangle$ )
inprod( $\langle x \rangle$ ,  $\langle y \rangle$ ) (innerproduct 関数の別名)
```

`innerproduct` 関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 $\cdot \langle y \rangle$ 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関数が含まれています。

行列操作の関数

```
columnvector( $\langle$  リスト  $\rangle$ )
conjugate( $\langle$  リスト  $\rangle$ )
conj( $\langle$  リスト  $\rangle$ ) (conjugate 関数の別名)
```

`columnvector` 関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.

similaritytransform 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)
uniteigenvalues(< 行列 >)
ueivects(< 行列 >)(uniteigenvalues 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 $\langle \text{リスト}_i \rangle$ は全て長さが等しくなければなりません。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 $\langle \text{リスト} \rangle$ の大きさを 1 にしたリストを返します。uniteigenvalues は与えられた $\langle \text{行列} \rangle$ 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。

uniteigenvalues 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。

2.9 文字列

2.9.1 Maxima の文字列

この節では、Maxima での文字列操作に関連する函数、主に contrib に含まれる stringproc パッケージに含まれる函数の解説を行います。

Maxima のデフォルトの文字列操作の函数は殆どありません。しかし、LISP 自体は文字列操作に関しては非常に強力な言語です。

この穴を埋めるのが contrib に含まれる stringproc パッケージです。stringproc パッケージは文字列操作を行う為の様々な函数を含んでいます。これらの函数は stringproc.lisp 内部で定義されており、ファイル名からも分かる様に LISP で記述されています。この stringproc パッケージは Maxima の文字操作の弱さを補強するものとなっています。

この stringproc パッケージで定義される函数は、大きく分けて四種類あります。一つは入出力に関連するもので、ファイルやストリームの処理を行う函数群です。次に、与えられた引数に対し、真偽値を返す函数があります。更に、与えられた数値を文字に変換する函数の様に、与えられた引数を文字や数値、或いは LISP の文字列データ等に変換する函数群、それから最後に、与えられた Maxima の文字列の結合や指定した位置の文字を取出したりする文字列操作の函数群があります。

ここで注意すべき事として、Maxima の文字列データは LISP の文字列型ではない事です。2.3.1 小節で解説した様に、Maxima の文字列データの内部表現は、文字列データの二重引用符を取り除いて、先頭に&を付けたものになります。これを具体的に確認しておきましょう。

```
%i19) neko1:"x^2+x-1";
(%o19)
                                     x^2+x-1
(%i20) neko2:x^2+x-1;
                                     2
(%o20)
                                     x  + x - 1
(%i21) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
```

```
MAXIMA> $neko1
&X^2+X-1
MAXIMA> $neko2
((MPLUS SIMP) -1 $X ((MEXPT SIMP) $X 2))
MAXIMA> (stringp $neko1)
NIL
MAXIMA> (atom $neko1)
T
```

この例では、変数 neko1 に"x^2+x-1"を割り当て、変数 neko2 には式 x^2+x-1 を割り当てています。to_lisp() で裏で動作している LISP に入り、変数 neko1 と neko2 の内部表現である \$neko1 と \$neko2 の値を表示させています。ここで、\$neko1 に束縛された値が &X^2+X-1 と先頭に&が付いて、

更に x が大文字になっている事に注意して下さい。そして, $\$neko1$ に束縛させた値は LISP の文字列型ではない事を `stringp` 関数を用いて確認しています。このデータの型は `atom` 型になります。

又, LISP との違いは Maxima には文字型のデータが無い事です。その為, この節を通して長さが 1 の Maxima の文字列の事を Maxima の文字と呼ぶ事にします。

この様に, Maxima と LISP ではデータ型が異なる為, `stringproc` パッケージで定義される関数では, Maxima と LISP との間での文字列データの変換関数や, 判別関数が多く含まれています。又, LISP にある関数を Maxima の関数に移植する傾向がある為, このパッケージで定義される大域変数は少なく, 関数も単機能なものが多いのが特徴です。

猶, Maxima にも文字列操作の関数は無い訳ではありません。例えば, `sconcat` の様に与えられた複数の Maxima の文字列を結合し, 一つの LISP の文字列を生成する関数があります。

Maxima 固有の文字列操作の関数

```
sconcat( $\langle$ Maxima の文字列 $\rangle_1, \dots, \langle$ Maxima の文字列 $\rangle_n$ )
string( $\langle$ 対象 $\rangle$ )
```

最初の `sconcat` 関数は LISP の `concatenate` 関数を Maxima に実装したもので, `macsys.lisp` 内部で定義されている関数です。この関数は与えられた複数の文字列を単純に結合するだけですが, 出力が LISP の文字列になります。

```
(%i1) sconcat("これは", "テスト", "だよ!");
(%o1)                これはテストだよ!
(%i2) neko:sconcat("This", " ", "is", " ", "a", " ", "test!");
(%o2)                This is a test!
(%i3) ?stringp(neko);
(%o4)                true
```

この例では, `sconcat` 関数の出力が LISP の文字列型である事を確認しています。

ここで, 結合した文字列も Maxima の文字列であって欲しい場合, LISP の関数を用いて内部データを料理するか, 次に解説する `string` 関数を用いるか, 或いは `stringproc` パッケージの `sconc` 関数を用いなければなりません。

ここで `string` 関数は与えられた対象を Maxima の文字列に変換する関数です。因に, この関数は `suprv1.lisp` で定義されています。

```
(%i40) string(factor(x^2+2*x+1));
(%o40)                (x+1)^2
(%i41) stringp(%);
(%o41)                true
```

この例では, Maxima の式 $x^2+2*x+1$ を `string` 関数で Maxima の文字列に変換しています。そして, 最後に `stringproc` パッケージの関数である `stringp` 関数を用いて Maxima の文字列である事を確認しています。

それでは `stringproc` パッケージに含まれる関数の解説を始めましょう。最初に入力操作に関連する関数について次の小節で解説しましょう。

2.9.2 ストリーム処理に関連する関数

stringproc パッケージには純粹に文字列の操作に関係する関数だけではなく、ストリーム操作の関数も含まれています。Maxima は LISP 上で動作している割に、ストリーム処理が全体的に弱かったのですが、stringproc パッケージを用いる事で、Maxima の処理言語だけを用いても、LISP と機能的に大差無い水準に近づけられるのです。

入出力操作

```

openw(< ファイル名 >)
opena(< ファイル名 >)
openr(< ファイル名 >)
close(< ストリーム >)
make_string_input_stream(< 文字列 >)
make_string_output_stream()
get_output_stream_string(< ストリーム >)
fposition(< ストリーム >)
fposition(< ストリーム >, < 正整数値 >)
flength(< ストリーム >)
readline(< ストリーム >)
freshline()
freshline(< ストリーム >)
newline()
newline(< ストリーム >)
printf(< ストリーム >, < Maxima の文字列 >)
printf(< ストリーム >, < Maxima の文字列 >, < オプション >)

```

openw 関数は Lisp のストリーム出力をファイルに落す為に用いる関数です。ファイルが存在しない場合には新たに生成され、既存の場合にはファイルを開きますが、ファイルはストリームを閉じた時点で書き換えられてしまいます。この関数は LISP の open 関数を利用しています。

opena 関数は openw 関数に似ていますが、既存のファイルに対して末尾にストリーム出力を追加する点で異なります。この関数も LISP の open 関数を利用しています。

openr 関数は入力ストリームとしてファイルを開く関数です。ファイルが存在しない場合はエラーになります。この関数は LISP の open 関数を利用していますが、他と違い、open 関数のオプションを設定していません。

make_string_input_stream 関数 は LISP の make-string-input-stream 関数を利用した関数です。引数の文字列から入力用のストリームを生成します。ここで引数は Maxima の文字列か LISP の文字列型でなければなりません。猶、LISP の with-input-from-string は make_string_input_stream の様に Maxima には実装されていません。

make_string_output_stream 関数 は LISP の make-string-output-stream 関数を利用した関数で、出力ストリームを開きます。猶、LISP の with-output-from-string は make_string_output_stream の様に Maxima で実装されていません。

close 関数はストリームを閉じる関数です。LISP の close 関数をそのまま利用しただけの関数です。

flength 関数は指定したストリームの成分数を返す関数です。この関数は LISP の file-length 関数をそのまま利用している関数です。その為、ファイルからの出力ストリームに対してのみ利用可能です。

fposition 関数は file-position 関数を利用した関数で、ストリーム上でポインタを移動させたり、そのポインタの位置を返したりする関数です。まず、引数がストリームだけの場合は、ストリームの頭からのポインタの位置を返します。この場合、1 がストリームの先頭であることを示します。次に、引数がストリームと正整数値が指定された場合には、正整数値で指定した箇所にポインタを移動させます。勿論、指定可能な正整数値の範囲は、1 から flength で求めたストリームの長さに限られます。

readline 関数は指定したストリームから一行づつ読取を行う関数です。

freshline 関数は LISP の fresh-line 関数を実装したものです。この関数は指定したストリーム、引数が無い場合には標準出力に改行を出力します。

newline 関数は LISP の terpri 関数を実装したものです。引数が無い場合には、標準出力に、ストリームの指定があれば、指定されたストリームに対して改行を行います。

printf 関数は Common LISP の format 関数を Maxima で利用する為の関数です。関数名が C 風なのが面白い事ですが、書式は C ではなく LISP の format 関数の書式に準じます。

2.9.3 文字の判別関数

以下で解説する判別関数は、与えられた対象が目的の対象 (Maxima の文字列等) に合致する場合に true を返し、それ以外は false を返す関数です。

判別関数

```
lcharp(<対象>)
charp(<対象>)
stringp(<対象>)
lstringp(<対象>)
constituent(<対象>)
alphanumericp(<対象>)
lowercasep(<対象>)
uppercasep(<対象>)
digitcharp(<対象>)
```

lcharp 関数は LISP の characterp 関数に引数をそのまま引渡す関数で、与えられた対象が LISP 型の文字の場合に true を返す関数です。LISP の character 関数を用いています。

charp 関数は、引数が Maxima の文字の場合に true を返す関数です。内部では引数が Maxima の文字列であることを確認し、その文字列の長さが 1 の場合を Maxima の文字としています。

lstringp 関数は LISP の stringp 関数に引数をそのまま引渡す関数で、与えられた対象が LISP 型の文字列の場合に true を返す関数です。

stringp 関数は、引数が Maxima の文字列の場合に true を返す関数です。

consituent 関数は引数が画面上で表示される文字の何れかで空行でない場合に true を返す関数です。猶、この関数の実体は [5] の P.61(原書では P.67) で定義されている contituent 関数そのもので、LISP の graphic-char-p 関数と char=関数を用いています。

alphacharp 関数は、引数が LISP の文字型であり、且つ、アルファベットの場合に true を返す関数です。この関数は alpha-char-p の Maxima への実装になります。

digitcharp 関数は数の場合のみに true を返す関数です。この関数は、内部で char-int 関数を用いて引数の文字を ASCII データの数値に変換し、その数値が 47 よりも大きく、58 よりも小さい場合に true を返します。

alphanumeric 関数は名前から判る様に、引数がアルファベットか数の場合に true を返す関数です。この関数は alphanumeric 関数の Maxima への実装に他なりません。

lowercasep 関数は引数が小文字の場合に、uppercasep 関数は引数が大文字の場合に各々 true を返す関数です。各々 .lower-case-p 関数と upper-case-p 関数の Maxima への実装になります。

次に二つの文字を比較する関数を以下に纏めておきます。

文字の比較関数

```
cequal(< 対象1>, < 対象2>)
cequalignore(< 対象1>, < 対象2>)
cless(< 対象1>, < 対象2>)
clessignore(< 対象1>, < 対象2>)
cgreaterp(< 対象1>, < 対象2>)
cgreaterpignore(< 対象1>, < 対象2>)
```

これらの関数には対応する LISP の関数があり、実質的には LISP の関数の Maxima への実装になっています。

cequal 関数は LISP の char=関数に対応する関数で、二つの Maxima の文字が等しい場合に true を返し、そうでない場合には false を返します。

cequalignore 関数は LISP の char-equal 関数に対応する関数で、二つの Maxima の文字が等しい場合に true を返し、そうでない場合には false を返します。

clessp 関数は LISP の char< 関数に対応する関数で、二つの Maxima の文字を ASCII コードで比較する関数です。基本的に第一引数の ASCII コードが第二引数の ASCII コードよりも小さい場合に true を返します。

clesspignore 関数は LISP の char-lessp 関数に対応する関数で、二つの Maxima の文字を ASCII コードで比較する関数です。基本的に第一引数の ASCII コードが第二引数の ASCII コードよりも小さい場合に true を返します。

cgreaterp 関数は LISP の char> 関数に対応する Maxima の関数で、二つの Maxima の文字を ASCII コードで比較する関数です。基本的に第一引数の ASCII コードが第二引数の ASCII コードよりも大きい場合に true を返します。

cgreaterignore 関数は LISP の char-greaterp 関数に対応する関数で、二つの Maxima の文字を ASCII コードで比較する関数です。基本的に第一引数の ASCII コードが第二引数の ASCII コード

よりも大きい場合に true を返します。

文字列に対しても同様の関数がありますが、`cequal` と `cequalignore` の文字列版のみです。

文字列の比較関数

```
sequal(<文字列1>, <文字列2>)
sequalignore(<文字列1>, <文字列2>)
```

`sequal` 関数と `sequalignore` 関数は二つの文字列が等しい場合に true を返す関数です。各々が LISP の `string=` 関数と `string-equal` 関数の Maxima への実装となっています。

2.9.4 文字列変換の関数

以下に LISP と Maxima の間での文字列の変換関数等を纏めておきます。

変換関数

```
cunlisp(<LISP の文字型 >)
sunlisp(<LISP の文字列 >)
lstring(<Maxima の文字列 >)
cint(<Maxima の文字 >)
ascii(<Maxima の整数値 >)
```

`cunlisp` 関数は LISP の文字型データを Maxima の文字列型のデータに変換する関数です。これに対し、`sunlisp` 関数は LISP の文字列型データを Maxima の文字列型データに変換する関数になります。

```
(%i43) to_lisp();
Type (to-maxima) to restart, ($quit) to quit Maxima.
```

```
MAXIMA> (setf $neko1 #\a)
#\a
MAXIMA> (setf $neko2 "aBcD")
"aBcD"
MAXIMA> (to-maxima)
Returning to Maxima
(%o43)                                     true
(%i44) cunlisp(neko1);
(%o44)                                     a
(%i45) sunlisp(neko2);
(%o45)                                     aBcD
(%i46) lstringp(neko1);
(%o46)                                     false
```

```
(%i47) lstringp(neko2);
(%o47) true
```

`lstring` 関数は Maxima の文字列を LISP の文字型に変換する関数です。この関数は単純に Maxima 内部で Maxima の文字列を LISP の文字列に変換する `l-string` 関数に Maxima の文字列を引渡すだけの関数です。

`cint` 関数は Maxima の長さが 1 の文字列を ASCII コードの数値に変換する関数です。LISP の `char-int` 関数を実装したものです。これに対し、`ascii` 関数は `cint` 関数の逆操作を行う関数で、与えられた整数値から ASCII コードの数値に対応する Maxima の文字型データを返す関数です。

```
(%i33) map(cint, ["a", "1"]);
(%o33) [97, 49]
(%i34) map(ascii, [40, 87]);
(%o34) [(, w]
```

2.9.5 文字列操作の関数

文字操作の関数

```
scopy(⟨ 文字列 ⟩)
smake(⟨ 整数値 ⟩, ⟨ 文字列 ⟩)
charat(⟨ 文字列 ⟩, ⟨ 整数値 ⟩)
charlist(⟨ 文字列 ⟩)
slength(⟨ 文字列 ⟩)
parsetoken(⟨ 文字列 ⟩)
sconc(⟨ 文字列1 ⟩, ., s⟨ 文字列n ⟩) sposition(⟨ 文字 ⟩, ⟨ 文字列 ⟩)
sreverse(⟨ 文字列 ⟩)
strim(⟨ 文字列1 ⟩, ⟨ 文字列2 ⟩) striml(⟨ 文字列1 ⟩, ⟨ 文字列2 ⟩) strimr(⟨ 文字列1 ⟩, ⟨ 文字列2 ⟩)
```

`scopy` 関数は LISP の `copy-seq` を用いる関数で、与えられた Maxima の文字列を複製する関数です。

`smake` 関数は LISP の `make-string` 関数を用いる関数で、Maxima の文字を指定した整数個並べて新しい文字列を生成する関数です。例えば、`smake(3, "a")` から Maxima の文字列 `aaa` が得られます。

`charat` 関数は与えられた Maxima の文字列から指定した整数で表現される位置にある文字を取り出します。因に、文字列の先頭 (左端) が 1 で、文字列の末尾が `slength` 関数で得られる値、即ち、文字列の文字数になります。例えば、`charat("explode", 2)` を実行すると、与えた文字列 `explode` の先頭から二番目の文字 `x` が返されます。因に、この関数は LISP の `subseq` 関数の Maxima への実装になります。

`charlist` 関数は与えられた Maxima の文字列を分解してリストデータに変換する関数です。例えば、`charlist("explode")` は `[e, x, p, l, o, d, e]` に変換されます。LISP に依存すると思いますが日本語も可能です。

```
(%i53) charlist("explode");
(%o53)          [e, x, p, l, o, d, e]
(%i54) charlist("爆発");
(%o54)          [爆, 発]
```

slength 関数は与えられた Maxima の文字列の長さを返す関数です。この関数の実体は、引数を LISP の文字型に変換し、length 関数を作用させているだけです。

parsetoken 関数は与えられた文字列が数と空行、コンマ、セミコロン、タブや改行で構成されている場合に、先頭の数値部分を数値に変換する関数です。

```
(%i106) parsetoken("1;234");
(%o106)          1
(%i107) parsetoken("1,234");
(%o107)          1
(%i108) parsetoken("1 234");
(%o108)          1
(%i109) 1/parsetoken("1234");
(%o109)          ----
                  1234
```

sconc 関数は Maxima に昔からある sconcat 関数に似ていますが、出力が Maxima の文字列になる点が異なります。構文は複数の Maxima の文字列を引数とし、これらの文字列を繋いで一つの Maxima の文字列を生成します。この関数は、sconcat 関数と全く同じ定義ですが、唯一の違いが、最後に答え返す段階で、sconc 関数では Maxima の文字列に変換する関数 m-string を作用させている点です。

sposition 関数は第一引数で指定した Maxima の文字が第二引数の文字列上で何処で最初に表われているかを返す関数です。

```
(%i67) sposition("c","cCcC");
(%o67)          1
```

sreverse 関数は Maxima の文字列の順番を逆にする関数です。LISP の reverse 関数の Maxima への実装です。

```
(%i1) sreverse("うえからよんでも山本山");
(%o1)          山本山もでんよらかえう
```

sinsert 関数は指定した位置に文字列を挿入する関数です。

```
(%i18) sinsert(", ","絶景かな絶景かな",5);
(%o18)          絶景かな、絶景かな
```

strim 函数, striml 函数と strimr 函数は共に LISP の string-trim, string-left-trim, string-right-trim 函数を Maxima に実装した函数です. 共に, 第二引数 〈文字列₂〉から特定の位置にある第一引数 〈文字列₁〉を除いた文字列を返す函数です.

これらの函数は 〈文字列₂〉に指定した文字列 〈文字列₁〉複数含まれている場合, 一つだけを削除し, 全ての 〈文字列₁〉を削除する事はしません.

先ず, striml 函数は 〈文字列₁〉から開始する文字列 〈文字列₂〉に対し, 〈文字列₁〉を取り除いた部分を返します. もし, 〈文字列₂〉が 〈文字列₁〉から開始していない場合には, 〈文字列₂〉をそのまま返却します.

strimr 函数は逆に 〈文字列₁〉を末尾に持つ文字列 〈文字列₂〉に対し, 〈文字列₁〉を除いた頭の部分を返却します.

strim 函数はこれらの striml 函数と strimr 函数の両方の働きをします. 猶, `strim("abc", "123abc123")` を実行すると, 真中の abc を抜いた文字列を返すのではなく, "123abc123" をそのまま返却します.

substring 函数は与えられた Maxima の文字列から指定した位置の Maxima の文字列を取り出す函数です. この函数は LISP の subseq 函数の Maxima への実装になります.

substring 函数

```
substring(〈文字列〉, 〈正整数〉)
substring(〈文字列〉, 〈正整数1〉, 〈正整数2〉)
```

```
(%i2) substring("うえからよんでも山本山", 9);
(%o2)                山本山
(%i3) substring("うえからよんでも山本山", 3, 9);
(%o3)                からよんでも
```

split 函数

```
split(〈文字列〉)
split(〈文字列〉, 〈分離文字〉)
split(〈文字列〉, 〈分離文字〉, false)
```

split 函数は与えられた文字列を分解したリストを返す函数です. 猶, 切れ目になる文字はデフォルトでは空行 (space) ですが, 別途指定する事が可能です. 又, 末尾に false を指定した場合, 区切文字が複数存在する場合に, 空行を入れたリストを返します.

```
(%i53) split("first  third fourth", " ", false);
(%o53)                [first, , third, fourth]
(%i54) split("first  third fourth", " ");
(%o54)                [first, third, fourth]
```

simplode 関数

```
simplode(< リスト >)
simplode(< リスト > < 文字列 >)
```

simplode 関数は explode 関数の逆操作の関数で、与えられたリストから文字列を生成する関数です。

引数が Maxima のリストのみの場合、リストの成分を繋げた文字列を返します。第二引数として Maxima の文字列を指定した場合、リストの成分を繋げる際に、第二引数の文字列を間に挿入します。

```
(%i53) simplode(["12","34","56"]);
(%o53)          123456
(%i54) simplode(["12","34","56"],"*-*");
(%o54)          12*-*34*-*56
```

この例では、第二引数に何も指定しない場合と、第二引数を指定した場合の違いを示しています。第二引数を指定しない場合には、単純に文字リストの成分を結合するだけですが、第二引数として "*" を指定すると、第一引数のリストの各成分にこの文字列を挿入している事が判るかと思います。

supcase 関数は与えられた文字列に対し、指定された範囲のアルファベットを大文字に変換する関数で、後述の sdowncase の逆操作になります。

supcase 関数

```
supcase(< 文字列 >)
supcase(< 文字列 >, < 正整数値 >)
supcase(< 文字列1 >, < 正整数値1 >, < 正整数値2 >)
```

```
(%i44) supcase("this is a pen.",1,1);
(%o44)          this is a pen.
(%i45) supcase("this is a pen.",1);
(%o45)          THIS IS A PEN.
(%i46) supcase("this is a pen.");
(%o46)          THIS IS A PEN.
(%i47) supcase("this is a pen.",5);
(%o47)          this IS A PEN.
(%i48) supcase("this is a pen.",1,2);
(%o48)          This is a pen.
```

sdowncase 関数は与えられた文字列に対し、指定された範囲のアルファベットを小文字に変換する関数です。この関数は supcase の逆操作になります。

sdowncase 関数

```
sdowncase(< 文字列 >)
sdowncase(< 文字列 >, < 正整数値 >)
sdowncase(< 文字列1 >, < 正整数値1 >, < 正整数値2 >)
```

```
(%i53) sdowncase("此の猫の名前は MIKE です");
(%o53)          此の猫の名前は mike です
(%i54) sdowncase("此の猫の名前は MIKE です",9);
(%o54)          此の猫の名前は Mike です
(%i55) sdowncase("此の猫の名前は MIKE です",9,11);
(%o55)          此の猫の名前は MikE です
```

sinvertcase 関数は与えられた文字列に対し、指定された範囲でアルファベットの大小を逆に変換する関数です。

sinvertcase 関数

```
sinvertcase(< 文字列 >)
sinvertcase(< 文字列 >, < 正整数値 >)
sinvertcasee(< 文字列1 >, < 正整数値1 >, < 正整数値2 >)
```

```
(%i58) sinvertcase("此の猫の name は MIKE です");
(%o58)          此の猫の NAME は mike です
(%i59) sinvertcase("此の猫の name は MIKE です",5,6);
(%o59)          此の猫の Name は MIKE です
```

2.9.6 評価関数を利用する文字列操作の関数

tokens 関数

```
tokens(< 文字列 >)
tokens(< 文字列 >, < 評価関数 >)
```

tokens 関数は [5] の P.61 にある tokens を取り込んだものです。評価関数は省略可能ですが、この場合は constituent 関数が用いられます。この関数は与えられた文字列を分解して、Maxima のリストに変換する関数です。この分解で評価関数を用います。具体的には評価関数で True になる部分のみが Maxima のリストの成分として返されます。

```
(%i79) tokens("MIKEandTAMAand123");
(%o79)          [MIKEandTAMAand123]
```

```
(%i80) tokens("MIKEandTAMAand123",constituent);
(%o80) [MIKEandTAMAand123]
(%i81) tokens("MIKEandTAMAand123",lowercasep);
(%o81) [and, and]
(%i82) tokens("MIKEandTAMAand123",uppercasep);
(%o82) [MIKE, TAMA]
(%i83) tokens("MIKEandTAMAand123",digitcharp);
(%o83) [123]
```

この例の場合、評価関数を指定しない場合と constituent 関数を指定した時の結果は同じものになります。又、lowercasep 関数を評価関数として指定した場合、lowercasep 関数が true を返す and の部分のみが返却されています。それに対し、uppercase 関数を指定すると大文字の部分のみ、digitcharp を指定した場合には、数値の部分のみが返却されています。

ssubstfirst 関数は文字列に指定した評価関数に適合する部分文字列が存在する場合に、部分文字列の入れ替えを行う関数です。

ssubstfirst 関数

```
ssubstfirst(<文字列1>, <文字列2>, <文字列3>)
ssubstfirst(<文字列1>, <文字列2>, <文字列3>, <評価関数>)
ssubstfirst(<文字列1>, <文字列2>, <評価関数>, <正整数値>)
ssubstfirst(<文字列1>, <文字列2>, <評価関数>, <正整数値1>, <正整数値2>)
```

名前の通り、評価関数に適合する部分が複数存在する場合には、一番最初の部分が取り換えられます。又、評価関数が未指定の場合は sequal 関数が用いられます。末尾の正整数値は評価を行う為の領域の指定を行う為に用いられます。

```
(%i13) ssubstfirst("三毛","虎","虎猫");
(%o13) 三毛猫
(%i14) ssubstfirst("三毛","虎","虎は虎猫ではない");
(%o14) 三毛は虎猫ではない
(%i15) ssubstfirst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequal,5);
(%o15) 虎は虎猫ではないが、三毛キチでもない
(%i16) ssubstfirst("三毛","虎","虎は虎猫ではないが、虎キチでもない",
sequal,5,9);
(%o16) 虎は虎猫ではないが、虎キチでもない
```

猶、領域指定の整数を引数として渡す場合には必ず評価関数を記述しなければなりません。

この ssubstfirst 関数と似た関数で、評価関数に合致する部分を全て入れ替えてしまうのが ssubst 関数です。

ssubst 関数

```
ssubst(<文字列1>, <文字列2>, <文字列3>)
ssubst(<文字列1>, <文字列2>, <文字列3>, <評価関数>)
ssubst(<文字列1>, <文字列2>, <評価関数>, <正整数値>)
ssubst(<文字列1>, <文字列2>, <評価関数>, <正整数値1>, <正整数値2>)
```

この関数で評価関数を未指定の場合は equalignore 関数が用いられます。

```
(%i19) ssubst("三毛", "虎", "虎猫");
(%o19)
三毛猫
(%i20) ssubst("三毛", "虎", "虎は虎猫ではない");
(%o20)
三毛は三毛猫ではない
(%i21) ssubst("三毛", "虎", "虎は虎猫ではないが、虎キチでもない",
sequalignore)
;
(%o21)
三毛は三毛猫ではないが、三毛キチでもない
(%i22) ssubst("三毛", "虎", "虎は虎猫ではないが、虎キチでもない",
sequalignore, 5);
(%o22)
虎は虎猫ではないが、三毛キチでもない
(%i23) ssubst("三毛", "虎", "虎は虎猫ではないが、虎キチでもない",
sequalignore, 5, 9);
(%o23)
虎は虎猫ではないが、虎キチでもない
```

ssubst 関数でも領域指定の整数を引数として渡す場合には必ず評価関数を記述しなければなりません。

sremovefirst 関数は文字列に指定した評価関数に適合する部分文字列が存在する場合に、その部分文字列を削除する関数です。

sremovefirst 関数

```
sremovefirst(<文字列1>, <文字列2>)
sremovefirst(<文字列1>, <文字列2>, <評価関数>)
sremovefirst(<文字列1>, <文字列2>, <評価関数3>, <正整数値>)
sremovefirst(<文字列1>, <文字列2>, <正整数値1>, <正整数値2>)
```

名前の通り、評価関数に適合する部分が複数存在する場合、一番最初の部分が削除されます。又、評価関数が未指定の場合は sequal 関数が用いられます。末尾の正整数値は評価を行う為の領域の指定を行う為に用いられます。

```
(%i28) sremovefirst("さようなら", "さようなら-ああ, さようなら", sequal, 5, 7);
(%o28)
さようなら-ああ, さようなら
(%i29) sremovefirst("さようなら", "さようなら-ああ, さようなら");
```



```
(%o29)                -ああ, さようなら
(%i30) sremovefirst("さようなら", "さようなら-ああ, さようなら", sequal);
(%o30)                -ああ, さようなら
(%i31) sremovefirst("さようなら", "さようなら-ああ, さようなら", sequal, 3);
(%o31)                さようなら-ああ,
(%i32) sremovefirst("さようなら", "さようなら-ああ, さようなら", sequal, 5, 7);
(%o32)                さようなら-ああ, さようなら
```

猶, 領域指定の整数を引数として渡す場合には必ず評価関数を記述しなければなりません.

sremove 関数は文字列に指定した評価関数に適合する部分文字列が存在する場合に, その部分文字列を全て削除する関数です.

sremove 関数

```
sremove(< 文字列1>, < 文字列2>)
sremove(< 文字列1>, < 文字列2>, < 評価関数 >)
sremove(< 文字列1>, < 文字列2>, < 評価関数 3>, < 正整数値 >)
sremove(< 文字列1>, < 文字列2>, < 正整数値1>, < 正整数値2>)
```

名前の通り, 評価関数に適合する部分が複数存在する場合, 全ての部分列が削除されます. 又, 評価関数が未指定の場合は sequalignore 関数が用いられます. 末尾の正整数値は評価を行う為の領域の指定を行う為に用いられます.

```
(%i33) sremove("さようなら", "さようなら-ああ, さようなら");
(%o33)                -ああ,
(%i34) sremovefirst("さようなら", "さようなら-ああ, さようなら", sequal, 3);
(%o34)                さようなら-ああ,
(%i35) sremovefirst("さようなら", "さようなら-ああ, さようなら", sequal, 3, 5);
(%o35)                さようなら-ああ, さようなら
```

猶, 領域指定の整数を引数として渡す場合には必ず評価関数を記述しなければなりません.

ssort 関数は与えられた文字列を指定した評価関数で大小関係を判別して並び換える関数です. 評価関数が未指定の場合は, clessp 関数が用いられます.

ssort 関数

```
ssort(< 文字列 >)
ssort(< 文字列 >, < 評価関数 >)
```

```
(%i63) ssort("bkavbhjsaAA");
(%o63)                AAaabbhjksv
(%i64) ssort("bkavbhjsaAA", cgreaterp);
```

```
(%o64)          vskjhbbaaAA
(%i65) ssort("いろはにほへと",cgreaterp);
(%o65)          ろほへはにとい
(%i66) ssort("いろはにほへと",clessp);
(%o66)          いとにはへほろ
```

評価関数を `clessp` から `cgreaterp` に変更する事で結果が逆になる事が判ります. 又, 日本語も扱えます.

`smismatch` 関数は二つの文字列を指定した評価関数を用いて比較し, 不適合が出る位置を返します. 評価関数が未指定の場合, `ssequal` 関数が用いられます.

————— `smismatch` 関数 —————

```
smismatch(< 文字列1>, < 文字列2>)
smismatch(< 文字列1>, < 文字列2>, < 評価関数 >)
```

```
(%i72) smismatch("桃も李も桃の内", "桃も李も藻も桃も");
(%o72)          5
(%i73) smismatch("1234123", "2345123", clessp);
(%o73)          5
(%i74) smismatch("1234123", "2345123", cgreaterp);
(%o74)          1
```

最初の例では評価関数を未指定の為, `ssequal` が用いられます. その為, 両者が初めて異なる藻の位置が返されています. 次の例では, 評価関数を `clessp` にした場合です. この場合 5 番目から大小関係が逆になるので 5 が返されていますが, 最後に `cgreaterp` を指定すると最初から不適合になる為, 1 が返されています.

`ssearch` 関数は第一引数の文字列が第二引数の文字列に含まれる場合に, その開始位置を返します. 又, 評価関数を用いて第二引数の文字列の部分列で, 第一引数の文字列との関係を満す箇所があれば, その開始位置を返します. 評価関数を指定する場合に限って, 評価を行う際の第二引数の文字列の開始位置や終了位置を指定する事も可能です. 又, 評価関数が未指定の場合, 内部では `sequalignore` 関数が用いられています.

————— `ssearch` 関数 —————

```
ssearch(< 文字列1>, < 文字列2>)
ssearch(< 文字列1>, < 文字列2>, < 評価関数 >)
ssearch(< 文字列1>, < 文字列2>, < 評価関数 >, < 正整数値 >)
ssearch(< 文字列1>, < 文字列2>, < 評価関数 >, < 正整数値1>, < 正整数値2>)
```

```
(%i1) ssearch("猫", "三毛猫と虎猫");
(%o1)          3
(%i2) ssearch("A", "AAAABCD", clessp);
```

```
(%o2) 5
(%i3) ssearch("A","AAAABCD",clessp,4);
(%o3) 5
(%i4) ssearch("A","AAAABCD",clessp,7,8);
(%o4) 7
(%i5) ssearch("A","AAAABCD",clessp,8);
(%o5) false
```

2.9.7 関連する大域変数

stringproc パッケージで定義されている大域変数は実はあまりありません。基本的に、stringprocs で定義されている関数で大域変数を殆ど用いない事が影響していますが、こうなる理由も基本的に LISP にある関数を Maxima 上で実現させる事を目的にした為、関数を制御する大域変数がそんなに必要ではなかった事も挙げられるでしょう。その為、関連する大域変数としては、改行、タブや空行を表現する newline,tab,space 程度しかありません。

——— 関連する大域変数 ———

newline	改行
tab	タブ
space	空行

第3章 プログラム

この章では,Maxima のプログラムに関連した事柄について解説を行います. 更に,Maxima の LISP に関連する事項と, ファイルの入出力についても述べます.

Maxima 上でのプログラムでは,Maxima の PASCAL 風の処理言語を用います. この処理言語を用いる事で, 利用者定義の Maxima の関数を構築する事も可能です.

Maxima 言語で記述された利用者定義関数は, 実行時には LISP で解釈され,LISP で処理されます. その為,LISP による解釈で速度の遅延が生じます. この為, 利用者定義関数を LISP の関数に変換する `translate` 関数や,LISP のコンパイラを利用して高速化を図る `optimize` 関数があります.

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>,...,<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, スタックに保存され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>,...,<式_n>, loop,<式>,...,go(loop),...) 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, 述語関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) と中置式演算子 (infix) は, 演算子を引数の置き方で区分したものです. 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,終了条件を満たさなければ,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て

```
<変数> : <初期値>
<変数> from <初期値>
```

この制御変数の初期値の割当て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,減少させたいければ,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関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.

制御変数の割当て方法

```
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,関数 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て

```
for i in <リスト> ...
```

この場合,リストの成分には数値以外の関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 条件を満たした時点で反復処理を終える。
 while 条件を満た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. 関数 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 関数も 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	[]	関数リスト
dispflag	true	block 文中の関数出力を制御
prederror	true	if 文や is 関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関数名

backtrace は debugmode:all の時に, 入力された関数全てのリストを値として持ちます.

dispflag が false ならば, block 文の中で呼ばれた関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 関数で, 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 が代わりに返されます.

errorfun に引数を持たない関数名が設定されていれば, エラー発生時に, その関数が実行されます. 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 文内部で利用する関数 ——

```

go(<ラベル>)
return(<式>)
break(<引数>, ...)
catch(<式1>, ..., <式n>)
throw(<式>)

```

go 関数は 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 関数は 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 関数は、 \langle 引数 \rangle 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. (maxima-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 関数は throw と対で用います. これは非局所的回帰 (non-local return) で用いる関数で、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. \langle 式 $_i$ \rangle 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 \langle 式 $_n$ \rangle 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
```

関数 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 関数は \langle 式 \rangle 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.

エラー処理を行う関数

```
errcatch( $\langle$  式 $_1$  $\rangle$ , ...,  $\langle$  式 $_n$  $\rangle$ )
error( $\langle$  引数 $_1$  $\rangle$ , ...,  $\langle$  引数 $_n$  $\rangle$ )
errmsg()
```

errcatch 関数は引数を一つずつ評価して、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. この関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 関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関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、最初のは文字列で、残りの対象は問題を起しているものです.

errmsg 関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([],errmsg(),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 で関数の定義

3.2.1 関数定義

Maxima では、予めシステムが持っている関数の他に、利用者定義の関数が扱えます。

Maxima の関数定義では演算子:=を用いる方法と define 関数を用いる方法があります。

関数の定義方法

```
define ( < 関数名 >, < ( ( < 引数1 >, …, < 引数n > ) ) >, < 式 > )
< 関数名 > ( < 引数1 >, …, < 引数n > ) := < 関数本体 >
```

基本的に define で定義する関数は block 文、if 文や do 文を含まない、一つの式のみで構成された関数を定義します。

一般的な関数の定義は演算子:=を用いた関数定義で関数を定義します。ここで、演算子:=を用いる関数定義で、その関数本体は式をコンマで区切ったものになります。

関数の定義を行うと、大域変数 functions に定義した関数が追加されます。

大域変数 functions

変数名	初期値	概要
functions	[]	利用者定義関数のリスト

大域変数 functions は利用者が定義した全ての関数名を含むリストです。利用者が関数定義を行うと自動的に functions のリストに関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、関数 $f(x)$ 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。

関数定義の演算子:=を用いた関数で、最も簡単なものは、関数本体が式₁、式₂、…、式_nの様に複数の式を並べたものです。この場合、関数の返却値は単純に最後の式の結果だけです。例えば、 $f(x):=(1-x,2+x,2*x)$ で関数 f を定義した場合、この関数の結果は最後の式 $2*x$ を計算した値になります。

但し、この関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、 $f(x):=(y:x,z:2+y,2*z)$ と定義した関数を実行した場合はどうなるのでしょうか？

```
(%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、この関数の場合、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 では関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関数が構築出来ます。この lambda 関数の構文は最初に関数の変数を宣言し、その後に関数本体が続きます。関数本体は基本的に Maxima の式をコンマで区切ったものになります。

lambda 関数

```
lambda([<変数1>, ..., <変数n>], <関数本体>)
```

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 を加える無名関数を構成し、その関数に map 関数でリスト [1,2,3] に作用させた結果と、lambda 式を利用した関数 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 関数内部で用いた疑似変数 i が lambda 関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関数も定義出来ます. 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,関数を定義します.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,関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,関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.

定義した関数の内容は関数 dispfun や fundef で参照する事が出来ます.

定義した関数の内容を表示する関数

```
dispfun(< 関数名1>, ..., < 関数名n> )
dispfun(all)
fundef( < 関数名 > )
```

利用者定義の関数 < 関数名₁>, ..., < 関数名_n> の内容を表示します.この関数の表示では,関数を定義した時点での関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 で与えられる関数を全て表示します.

fundef 関数は < 関数名 > に対応する関数の定義を返します. fundef は dispfun に似ていますが,fundef では display 関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.

————— 利用者定義関数を削除する関数 —————

```
remfunction (< 関数1>, < 関数2>, ...)  
remfunction (all)
```

remfunction 関数は利用者定義関数を Maxima から削除します. 利用者定義関数は大域変数 functions にその名前が保存されており, remfunction は functions に含まれている関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関数が削除されます.

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 関数を用います.

————— buildq 関数 —————

```
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_n := F_{n-1} + F_{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)          81055900096023504197206408605
```

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_n - (n - 1) Pz_{n-2}
          -----
                   n
(%o10) -----
                   n
(%i11) define(Pz[n],buildq([v:pnz],expand(v)));
          (2 n - 1) Pz_n - (n - 1) Pz_{n-2}
          -----
                   n
(%o11) Pz_n := expand(-----)
                   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 が grind が呼出されていれば参照出来ます.

3.2.3 apply 関数

Maxima で重要な関数の一つに apply 関数があります. この apply 関数は Maple や Mathematica にもある関数です.

— apply 関数 —

```
apply(〈関数〉,〈リスト〉)
```

apply 関数は〈関数〉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.

関数の呼出しで, 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' を関数の先頭に置いて, 関数を名詞型として, apply に評価させます. 幾つかのアトムの変数が, とある関数と同じ名前を持っていれば, 関数としてではなく, その変数値が利用される事になります. 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 を関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、関数やデータを LISP の関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 には利用者定義関数を LISP の関数に変換する関数を幾つか持っています。

LISP 関数に変換する関数

```
translate(〈関数1〉, …, 〈関数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 関数は Maxima の処理言語で記述した利用者定義の関数を LISP の関数に変換する関数です。Maxima 言語で記述された関数は、裏の LISP で解釈されて実行されます。これを LISP の関数にすれば、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〈関数₁〉, …, 〈関数_n〉の様に利用者定義関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、利用者定義関数を一度に変換する事も出来ます。

変換される関数が内部で局所変数を利用する場合、mode_declare 関数を用いて、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 関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, … ) := block([ 局所変数1, 局所変数2, … ],
                        mode_declare( 局所変数1, 型1, 局所変数2, 型2, … ),
                        〈関数本体〉)
```

この mode_declare 関数の詳細に関しては、1.3.4 を参照して下さい。

次に、関数を translate 関数で変換すると、大域変数 savedef が false の場合、変換された関数の名前は大域変数 functions に割当てられた関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関数名が追加されます。

当然の事ですが、関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 関数は変換する関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 関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 関数を用いて LISP の関数に変換しても、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 関数は Maxima 言語で記述したプログラムを含むファイルを LISP 関数のファイル

に変換する関数です。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 関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関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。

コンパイルを行う関数

```
compile ( < 関数1 >, ... , < 関数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 > )
compile( < ファイル >, < 関数1 >, ... , < 関数n > )
```

compile 関数は指定した Maxima の処理言語で記述した関数を LISP の関数に変換し、それを LISP の関数 compile を用いてコンパイルします。尚、compile 関数は関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関数を全てコンパイルします。

これに対し、compile_file 関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 関数は、これを LISP 関数に translate 関数で変換し、その結果を compile 関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 に読み込みます。

compile_file は四つ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。

compile 関数は < 関数₁ >, ... , < 関数_n > を LISP の関数に変換し、< ファイル > に書込みます。

```
(%i28) neko(x):=sin(x);
(%o28)                               neko(x) := sin(x)
(%i29) compile("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 関数と compile 関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 関数と compile 関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 関数による出力制御
savedef	true	translate 関数による変換後に元の関数を残す
translate	false	translate 関数による自動変換を制御
transrun	true	変換前の関数の実行
undeclaredwarn	compile	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 による関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, 利用者関数を translate 関数で変換しても, 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関数名リストから, 変換した関数名を削除せずに残します。又, dispfun 関数で関数の定義が表示可能で, 関数の編集も出来ます。

false の場合は, functions に割当てた利用者関数名リストから該当関数名を削除します。

大域変数 translate が true であれば, 利用者定義関数が自動的に LISP 関数に変換されます。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 関数を一つ以上の引数で用いたり, ratvars 関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 関数で変換されたものではなく, 元の Maxima の関数 (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。

undeclaredwarn の設定項目

設定	動作
false	警告を表示しません
compile	compile であれば警告します
translate	translate や translate:true であれば警告します
all	compile 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 関数でコンパイルされる利用者定義関数中の変数を宣言する特別な動作は全て無効にしなればなりません。

次に,translate 関数に影響を与える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
<code>define_variable</code>	<code>[]</code>	
<code>transcompile</code>	<code>false</code>	<code>compile</code> 関数に必要な宣言を自動生成
<code>translate_fast_arrays</code>	<code>true</code>	配列変換を制御
<code>tr_array_as_ref</code>	<code>true</code>	変換関数の配列評価を指定
<code>tr_function_call_default</code>	<code>general</code>	関数変換を制御
<code>tr_numer</code>	<code>false</code>	数値変数の型を制御
<code>tr_semicompile</code>	<code>false</code>	機械コードへの翻訳を制御
<code>tr_warn_fexpr</code>	<code>compile</code>	<code>fexpr</code> 型に対する警告制御
<code>tr_warn_meval</code>	<code>compile</code>	<code>meval</code> 型関数に対する警告制御
<code>tr_warn_mode</code>	<code>all</code>	変数型に対する警告を制御
<code>tr_warn_undeclared</code>	<code>compile</code>	未宣言変数に対する警告を制御
<code>tr_warn_undefined_variable</code>	<code>all</code>	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 関数は可能な `compile` 関数に必要な宣言を生成します.`compile` 関数は `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` 関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,変換された関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 内部関数 `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 関数と仮定します.
- `apply` の場合:`apply` 関数を用いて関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,コンパイルされた関数中で変数の割当てが正確であることを保証します.例えば,関数 $f(x)$ 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])` の事として関数を変換します.

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 関数で変換し, `compile` 関数でコンパイルした利用者定義関数には元の `maxima` 関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` 関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` 関数と `compile` 関数の出力形式は拡張されたマクロになります。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` は関数 `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` 関数に関連する大域変数を纏めておきます。

— LISP 関数への変換に関連する大域変数 —

変数名	初期値	概要
<code>tr_bound_function_apply</code>	<code>true</code>	変換関数の割当てに対し警告
<code>tr_file_tty_messagesp</code>	<code>false</code>	メッセージ出力制御
<code>tr_float_can_branch_complex</code>	<code>true</code>	逆三角関数の複素数値の制御
<code>tr_optimize_max_loop</code>	100	マクロループの回数
<code>tr_warn_bad_function_calls</code>	<code>true</code>	変換時の警告を制御

`tr_bound_function_apply` が `true` の場合、関数の引数として割当てられた変数が、関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` は逆三角関数が複素数値を返しても良いかどうかを宣言します。逆三角関数は `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、関数の呼出しが生じた場合に警告します。

3.2.5 関数定義に関連する関数

— funmake 関数 —

`funmake(〈関数〉, [〈引数1〉, …, 〈引数n〉])`

funmake 関数は \langle 関数 \rangle で指定した関数を呼出して評価を行う事はしません. 単純に, \langle 関数 \rangle (\langle 引数 $_1$ \rangle, \dots, \langle 引数 $_n$ \rangle) 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 関数は translate 関数による変換中に translate 関数が出力する警告のリストを表示します.

declare_translated 関数

```
declare_translated( $\langle$ 関数 $_1$  $\rangle, \dots, \langle$ 関数 $_n$  $\rangle$ )
```

declare_translated 関数は, 引数の関数が既に変換されている事を宣言する関数です. 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関数が translate 関数で変換された関数, 或いは compile 関数でコンパイルされた関数として呼出されるべきか, そして, どれが Maxima の関数で, 又, 未定義のものであるかを知る事は translate 関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 関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, \dots) が生成されるのは, \langle 関数 $_n$ \rangle が LISP 関数に変換されるべきものであるかを translate 関数が知らない時です.

local 関数

```
local( $\langle$ 局所変数 $_1$  $\rangle, \dots, \langle$ 局所変数 $_n$  $\rangle$ )
```

local 関数は, この関数が利用される文中で, \langle 局所変数 $_1$ \rangle, \dots, \langle 局所変数 $_n$ \rangle を全ての属性に対して局所的なものにします. local は block 文, 関数定義の本体, lambda 式, 又は ev 関数でのみ一度だけ使えます. この local 関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み込む際には、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、`writfile` を用います。この `writfile` 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、`writfile` 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。`writfile` 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 に読み込みます。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 や他の関数で、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関数

—— バッチ処理に関連する関数 ——

```
batch(⟨ ファイル名 ⟩)
batchload(⟨ ファイル名 ⟩)
batcon(⟨ 引数 ⟩)
```

batch 関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み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 関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 関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 関数は中断されたファイルのバッチ処理を再開します。

—— 読みを行う関数 ——

```
load(⟨ ファイル名 ⟩)
loadfile(⟨ ファイル名 ⟩)
read(⟨ 文字列1, … ⟩)
readonly (⟨ 文字列1 ⟩, …)
```

load 関数は文字列や、リストで表現されたファイル名の読み込みを行います。ディレクトリが指定されていないならば、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み込もうとします。

load 関数はファイルが 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)。

他のファイルの読みを行う 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 関数は指定されたファイルを読みみます。この関数は以前の Maxima の処理で save 関数で保存した値を 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 ファイルを読み込むのであれば、"/home/user/foo.mc" となります。

尚、save 関数で保存したファイルを loadfile で読み込むと、Maxima は初期化されるので注意が必要です。

read 関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(x²+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 ; 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 関数は引数を全て表示し、それから式を読みみます。基本的には read と同様ですが、read と違うのは、読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 関数

```
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 関数は指定したファイルに Maxima が読み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 関数は 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 は文字列ではなく、grind と同じ書式で出力します。

書き込みを行う関数

```

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 関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 関数との違いは、同名のファイルが存在した場合、writefile 関数は上書きをしてしまいますが、appendfile 関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 関数と同様に指定したファイルは closefile() 関数で閉じます。

writefile 関数は書き込み用にファイルを新規に開きます。writefile 関数を実行すると、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 に再度読み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 of 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 関数です。

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 関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 関数は, 指定されたファイルを開き, $\langle \text{式}_1 \rangle, \dots, \langle \text{式}_n \rangle$ 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わ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 関数は closefile() で appendfile や writefile で開かれたファイルを閉じます. closefile 関数は LISP の close 関数を使った関数です. この close 関数は開かれたストリームを閉じる関数です.

save 関数

```
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 関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.

$\langle \text{引数}_1 \rangle, \langle \text{引数}_2 \rangle, \dots$ で, 各引数の値を保存します.

$[\langle m \rangle, \langle n \rangle]$ 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くなるので注意が必要になります. save 関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;
(%o2) y2 + x2 + 1
(%i3) resultant(x-t,y-t^2,t);
(%o3) y2 - x2
(%i4) save("test",all);
(%o4) test
```

save 関数で保存したファイルは loadfile 関数で Maxima に再び読み込む事が出来ます. 但し, loadfile を実行すると, 以前の Maxima 自体を初期化し, save 関数を実行した時点にまで戻っ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 を読み込んでいますが, 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 関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 '03)
(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。

その他の関数

```
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 関数は二つの < 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関数です。

file_search 関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 関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関数なので、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

この章では Maxima のシステムやユーザーインターフェイスに関連する大域変数と函数を解説します。

関連する函数が多岐に及ぶ為、雑多なものの集まりの印象が免れないかもしれません。

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 を自動的に読み込みます。

maxima-init.mac 内部には Maxima の関数や大域変数の設定が記述可能です。更に,setup_autoload 関数や load 関数を用いて,必要なパッケージの読み込み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 が読み込むファイルにて,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 関数や load 関数を用いたファイルの読み込みについて簡単に解説しておきましょう。この setup_autoload 関数の構文を以下に示しておきます。

setup_autoload 関数

```
setup_autoload (<ファイル>, <関数1>, ..., <関数n>)
```

この setup_autoload 関数は,指定した関数が呼出された時点で関数が未定義の場合,指定したファイルの読み込みを実行します。ここで,setup_autoload 関数は引数として配列関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 関数が呼出された時に,未定義であれば,ファイル surfplot.mc を読み込む設定となっています。

load 関数の場合は予め指定されたファイルの内容を全て Maxima に読み込みます。この場合、ファイルの定義、未定義とは無関係に読み込みを実行します。

この maxima-init.mac ファイルを利用することで、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 関数を用いて Maxima 起動時の初期状態に戻す事や kill 関数を用いて、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 関数

```
reset()
```

この reset 関数は引数無しで、reset(); と入力します。reset 関数を用いると、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、ラベルのカウント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 関数

```
kill(< 引数1>, < 引数2>, ...)  
kill(< 整数 >)  
kill([< 整数1>, < 整数2>])  
kill(all)  
kill(allbut(< 引数1>, < 引数2>, ...))
```

kill 関数は reset 関数とは異なり、必ず引数を取ります。kill 関数は引数によって削除する対象が異なります。

- $\langle \text{引数}_i \rangle$ が変数 (単配列要素を含みます)、関数、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。
- $\langle \text{引数}_i \rangle$ 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。
- $\langle \text{引数}_i \rangle$ が clabels であれば、入力行だけが消去されます。
- $\langle \text{引数}_i \rangle$ が elabels であれば、中間行のみが消去されます。
- $\langle \text{引数}_i \rangle$ が dlabels であれば、出力行のみが消去されます。
- $\langle \text{引数}_i \rangle$ 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 関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 関数群 (remove, remfunction, remarray, remrule) は指定した属性のみを削除します。ここで、remove 関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 関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 関数

```
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 関数は optimize 関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 関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 レベルに戻るため、通常は Ctrl+C を使います。

```
(%i11) factor(2137498127943870982374);
```

```
Maxima encountered a Lisp error:
```

```
EXT:GC: User break
```

```
Automatically continuing.
```

```
To reenble 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 関数による中断

Maxima では break 関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 関数

```
break(< 式1>, ..., < 式n>)
```

break 関数の返却値は < 式_n> の処理結果となります。break 関数は引数の全ての式を処理した後 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 では通常の関数が利用可能ですが、ev 関数の省略型や、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、 $\int x \sin(x) dx$ 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 関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関数に%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、各ラベルのカウントも 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);
(%o102) y - x
(%i103) algsys([2*x+3*y=1],[x,y]);
(%o103) [[x = %r2, y = -  $\frac{2 \%r2 - 1}{3}$ ]]
(%i104) %;
(%o104) [[x = %r2, y = -  $\frac{2 \%r2 - 1}{3}$ ]]
(%i105) _;
(%o105) %
(%i106) %i101;
(%o106) 3
(%i107) %o101;
(%o107) 3
(%i108) %i102;
(%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 関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` 関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` 関数で Maxima-break に入った時点で表示されます。

```
(%i1) integrate(x^2-1,x);
(%o1)

$$\frac{x^3 - 3x}{3}$$

```

```
(%i2) prompt:"(;;)";
(%o2) (;;)
(%i3) playback(slow);
```

```
(%i1) integrate(%,x);
(;;)
(%o1)

$$\frac{x^3 - 3x}{3}$$

(;;)
```

```
(%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(i)` を含む式の入力ラベルが `%(j)` であれば、`%th(i)` は `%i(j - i)` の結果、即ち、`%o(j - i)` 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関数があります。基本的に Maxima はキャラクター端末しかなかった昔のシステム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
pformat	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。ここで出力は `part` 関数に対応するのではなく、`inpart` 関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}
$\text{sqrt}(x)$;	$\text{sqrt}(x)$	$x^{1/2}$
$x * 4/3$;	$\frac{4x}{3}$	$\frac{4}{3}x$

`%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 `%e^(-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、 $\exp(-x)$ は $1/\%e^x$ 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、 x^{-1} は $1/x$ となります。

大域変数 `pformat` 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` 関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 式の表示を行う関数

式の表示を行う関数

```
display(< 式1>, < 式2>, ...)  
disp(< 式1>, < 式2>, ...)  
ldisplay(< 式1>, < 式2>, ...  
ldisp(< 式1>, < 式2>, ...  
print(< 式1>, < 式2>, ...)  
grind(< 式 >)
```

display 関数は式を表示します。その左側が未評価の $\langle \text{式}_i \rangle$ で、その右側の行の中心がその式の値となります。この関数は 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、添字された変数や関数呼出しになります。

disp 関数は display 関数に似ていますが引数の値のみを表示します。

ldisplay 関数と ldisp 関数は各々、display 関数と disp 関数に似ていますが、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 関数は、 $\langle \text{式}_1 \rangle$ 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、 $\langle \text{式}_i \rangle$ に含まれるアトムや関数の前に単引用符 ' 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 関数は、〈式〉を Maxima の入力に適した形式で表示します。grind 関数の返却値は常に done です。

—— 式の内部表現に関連して表示する関数 ——

```
dispterm (〈式〉)
reveal (〈式〉, 〈深度〉)
tcl_output (〈リスト〉, 〈添字〉, 〈飛幅〉)
tcl_output (〈リスト〉, 〈添字〉)
```

dispterm 関数は与式を内部表現に合わせて、第一層の部分式を順次表示します。

reveal 関数は〈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。指数関数は expt で表現されます。

```
(%i10) aa: integrate(1/(x^3+2), x)$
```

```
(%i11) aa;
```

```
(%o11) - 
$$\frac{\log(x^2 - 2) \sqrt[3]{x + 2}}{6} - \frac{\operatorname{atan}\left(\frac{2x - 2}{\sqrt{3}}\right)}{2\sqrt{3}} + \frac{\log(x + 2)}{3}$$

```

```
(%i12) reveal(aa,1);
```

```
(%o12) sum(3)
```

```
(%i13) reveal(aa,2);
```

```
(%o13) negterm + quotient + quotient
```

```
(%i14) reveal(aa,3);
```

```
(%o14) - quotient + 
$$\frac{\operatorname{atan}}{\operatorname{product}(2)} + \frac{\log}{\operatorname{product}(2)}$$

```

```
(%i15) reveal(aa,4);
```

```
(%o15) - 
$$\frac{\log}{\operatorname{product}(2)} + \frac{\operatorname{atan}(\operatorname{quotient})}{\operatorname{expt} \sqrt{3}} + \frac{\log(\operatorname{sum}(2))}{\operatorname{expt}}$$

```

```
(%i16) reveal(aa,5);
```

$$\frac{\log(\text{sum}(3))}{6 \text{ expt}} + \frac{\text{atan}\left(\frac{\text{sum}(2)}{\text{product}(2)}\right)}{2 \sqrt{3}} + \frac{\log(x + \text{expt})}{3^2}$$

```
(%o16)
```

tcl_output 関数は、〈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 後戻し表示を行う関数

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 関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。

後戻し表示の関数 playback

```
playback (< 整数 >)
playback ([< 整数1>, < 整数2>])
playback ([< 整数 >])
playback ()
playback(input)
playback(slow)
playback(time)
playback(grind)
```

playback 関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 [\langle 整数₁ \rangle , \langle 整数₂ \rangle] の場合、 \langle 整数₁ \rangle から \langle 整数₂ \rangle 迄の全ての行を再実行します。 \langle 整数₁ $\rangle = \langle$ 整数₂ \rangle であれば、引数として [\langle 整数 \rangle] 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。

まず, input の場合は入力行を再表示行します。

slow の場合は example 関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 関数を終了します。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` 関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
<code>error_size</code>	10	エラーメッセージの大きさを制御
<code>error_syms</code>	<code>[errexpl,errexpl2,errexpl3]</code>	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 `errexpl,errexpl2,errexpl3` 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 `errexpl1` as input." であれば, `errexpl1;` 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. これは関数としては, 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.

?関数

<code>?</code>	<code>? < 事項 ></code>	<code>< 事項 ></code> に関連するオンラインマニュアルを表示
<code>?</code>	<code>? < LISP に評価させる関数 ></code>	S 式を評価

関数 `?` 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 の関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 に評価させる関数を記述しますが、LISP の S 式ではなく、Maxima の関数風に表記しなければなりません。更に、その LISP の関数に与える引数も、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 の関数で式の評価をする場合、:lisp 関数もありますが、こちらは LISP の S 式を引数に取りますが、その返却値は関数?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 ($f(x) := \dots$ 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関数が多くあります. 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 関数

status 関数

`status (< 引数 >)`

`status` 関数は利用中の `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). $250 * \text{blocks}(\text{mc})$ 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 関数

room 関数

```
room ()
room (true)
room (false)
```

room 関数は保存領域の状況を詳細な記述で出力します. この room 関数は LISP の room 関数を利用したものです.

4.6.4 処理時間に関連する関数

処理時間表示の関数

```
time (<%o1>, <%o2>, ...)
```

time 関数は $\langle \%o_i \rangle$ 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 `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み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した全ての gctime を含んでいます。その為、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 式の変換を行う関数

Maxima では出力式を TeX, FORTRAN の書式に変換出来ます。

tex 関数

```
tex(< 式 >)
tex(< 式 >, < ファイル名 >)
tex(< ラベル行 >, < ファイル名 >)
```

tex 関数は、与えられた < 式 > や < ラベル行 > を $\text{T}_{\text{E}}\text{X}$ 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 関数

```
fortran(< 式 >)
```

fortran 関数は与式を 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 used。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 を繰り返します。

尚、fortran 関数は Maxima の式を FORTRAN の書式に合せます。例えば、式中の演算子 \wedge 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て出力します。ここで、右辺が行列名であれば、fortran 関数は、行列の各成分に割当を行う様に変換します。

尚、与式に fortran 関数で解釈出来ないものがあつた場合、grind 関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 関数は常に返却値は 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2>, ...)  
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 `aliases` 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 を全て終了させたい場合は, `($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.

4.11 虫取りに関連する函数

Maxima には他のプログラム言語と同様に、虫取りの函数が幾つか用意されています。ここでの節では、虫取りに関連する函数と大域変数について解説します。猶、`mtrace.lisp` にここで解説する函数が定義されています。

4.11.1 trace 函数と untrace 函数

— trace 函数の構文 —

```
trace ( < 函数1 >, ..., < 函数n > )
trace(all)
trace()
trace_it(< 函数 >)
```

`trace` 函数に函数を指定する事で、指定した函数の動作追跡を行います。函数の指定は、`trace(integrate);` の様に一つの函数のみを指定したり、`trace(integrate,factor);` の様に複数の函数を指定する事が可能です。又、引数に `all` を指定した場合、全ての函数に対し、動作追跡を行う事になります。猶、引数を指定しない `trace()` の場合、`trace` 函数による動作追跡が設定されている函数のリストを返します。

`trace_it` 函数は `trace` 函数と同じ働きをしますが、引数は一つの函数だけです。

それでは簡単な例で確認してみましょう。

```
(%i15) trace(integrate);
(%o15) [integrate]
(%i16) integrate(2*sin(x)*cos(x),x);
1 Enter integrate [2 cos(x) sin(x), x]
      2
1 Exit integrate - cos (x)
      2
(%o16) - cos (x)
(%i17) trace();
(%o17) [integrate]
(%i18) trace(integrate,risch);
integrate is already traced.
(%o18) [risch]
(%i19) trace();
(%o19) [risch, integrate]
```

ここでの例では最初に `integrate` 函数の追跡を行う様に `trace` 函数で指定しています。この状態で、`integrate` 函数を起動すると、`integrate` 函数への入力と `integrate` 函数の出力が表示されます。次

に,trace() を実行する事で,trace 函数による追跡が指定された函数のリストを返します.

次に,integrate 函数と risch 函数を trace 函数で指定しています. この場合, integrate 函数は既に指定済みの為に警告が出ますが,risch 函数の登録は実行されます.

さて,integrate 函数ではオプションの risch を ev 函数で指定する事で, Risch 積分を行います. この場合の動作はどの様になるでしょうか. 先程の例の続きを示しておきましょう.

```
(%i25) ev(integrate(3^log(x),x),'risch);
          log(x)
1 Enter integrate [3      , x]
          log(3) log(x)
          x %e
1 Exit  integrate -----
          log(3) + 1
          log(3) log(x)
          x %e
(%o25) -----
          log(3) + 1

(%i26) integrate(3^log(x),x);
          log(x)
1 Enter integrate [3      , x]
          1
          (----- + 1) log(x)
          log(3)
          3
1 Exit  integrate -----
          1
          (----- + 1) log(3)
          log(3)
          1
          (----- + 1) log(x)
          log(3)
          3
(%o26) -----
          1
          (----- + 1) log(3)
          log(3)

(%i27) risch(3^log(x),x);
          log(x)
1 Enter risch [3      , x]
          log(3) log(x)
          x %e
```

```

1 Exit  risch -----
          log(3) + 1
          log(3) log(x)
          x %e
(%o27)  -----
          log(3) + 1

```

ここでの例では, `ev` 函数を用いる事で `risch` 積分も実行させています. 但し, ここでの例では, `risch` 函数の本体を使うものの, `Maxima` の `risch` 函数そのものは使いません. その為, `trace` 函数では `risch` 函数の事は表に出て来ません. 但し, これは随分と捻くれた使い方をしているので, 現れなくても問題はありません.

`trace` 函数と逆の指定を行うのが `untrace` 函数です. この `untrace` 函数の構文を以下に示しておきましょう.

----- untrace 函数の構文 -----

```

untrace ( < 函数1 >, ..., < 函数n > )
untrace()

```

`untrace` 函数は, `trace` 函数による動作追跡が指定された函数に対し, その動作追跡の解除を行います. 引数を指定しない場合, 無条件で全ての函数の動作追跡を解除します.

単純に `trace` 函数を函数に作用させるだけでは, その函数への入力と出力だけしか出力されませんでした. その為, より詳細な情報を得る為には, `trace` 函数に対するオプション設定が必要になります. このオプションの設定は `trace_options` 函数を用います.

----- trace_options 函数 -----

```

trace_options(< 函数 >, < オプション1 >, ..., < オプションn >)
trace_options(< 函数 >)
trace_options()

```

まず, 基本的な構文としては, 第一引数に `trace` 函数で追跡を行う函数名を指定し, その後に指定可能なオプションを記入します.

猶, オプション無しの `trace_options(< 函数 >)` で, 追跡を行う函数に設定したオプションを解除し, 初期状態に戻します.

指定可能なオプションを以下の表に纏めておきます.

指定可能なオプション

noprint	true の場合には表示を行いません.
break	true の場合には breakpoint を与えます.
lisp_print	true の場合,LISP の内部形式で表示を行います.
info	
errocatch	true の場合, エラーが捕捉されます.

猶,trace_options で設定した trace 関数のオプションの情報は get 関数を用いて取出す事が可能です.

```
(%i9) trace_options(integrate,info);
(%o9) [info]
(%i10) get('integrate,'trace_options);
(%o10) [info]
```

この例で判る様に,trace_options によって関数の trace_options 属性にオプションの値が設定されます. その為,get 関数を用いる事で, オプションの情報が入手可能となる訳です.

timer と timer_info

noprint とした場合,trace 関数による表示を無効にします. break を指定した場合,Maxima break point に入ります. この時点で, 関数内部変数の値を調べたり, 変更する事が可能です. この時, 変数値の表示は n; の様にし, 値の変更は n:128; の様に通常の Maxima の様に使います.

Maxima break 中は exit; を入力すると, 次の break point に移ります.

lisp_print は trace 関数による出力を内部の LISP 書式で行います.

info を設定した場合, 内部の処理で true->を

4.11.2 関連する大域変数

虫取りに大域変数

大域変数名	初期値	概要
setcheck	false	変数をリストで指定し, その挙動を追跡する.
setcheckbreak	false	
setval	'setval	Maxima break 時に追跡している変数の値を一時的に保全.
trace	[]	追跡を行う関数のリスト.
trace_max_indent	15	
trace_break_arg		

setcheck の初期値は false ですが, ある特定の変数がどの様に書換えられて行くかを追跡したい場合に, その追跡したい変数のリストとして指定します.

それでは実際にその効果を試してみましょう.

```

(%i2) setcheck:['x,'y];
(%o2) [x, y]
(%i3) x:128;
x SET T0 128
(%o3) 128
(%i4) y:x*2;
y SET T0 256
(%o4) 256
(%i5) x:'x;
(%o5) x
(%i6) x:10;
x SET T0 10
(%o6) 10

```

この様に, `setcheck` に指定した変数リストに含まれる変数に対して割当てが発生して時点で, `x SET T0 10` の様に表示されます.

`setcheck` は `all` で全ての変数に `true` 設定しても構いません. 但し, `x:'x` の様に `setcheck` で指定された変数がそれ自身に設定されていると, 表示されません.

`setcheckbreak` の値が `true` の場合、`setcheck` リストに収録された変数に値の設定が行われる時に `break` による処理が中断されます. この時点で `setval` 変数に設定されようとする変数の値が保全されます. 猶, `setval` 変数に保全された値を別途の再設定して, `setcheckbreak` に設定された変数の値を意図的に変更しても構いません.

大域変数 `setval` は `setcheckbreak` 時に, 追跡している変数の値を一時的に蓄える為に用いられます. この変数は Maxima Break 時にはじめて値が設定され, 利用者はその値の書換も可能です.

第5章 Maximaの函数

この章では, その他の重要な函数について解説します.

但し, Maxima が持つ全ての函数を解説するものではない為, 必要に応じて, 配布の Maxima に付属のマニュアルを参照して下さい.

5.1 三角関数

5.1.1 三角関数一覧

Maxima は沢山の三角関数を持っています。三角関数の恒等式、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関数は下記のものがあります。

三角関数と双曲関数

cos	余弦関数
cosh	双曲線余弦関数
cot	余接関数
coth	双曲線余接関数
csc	余割関数
csch	双曲線余割関数
sec	正割関数
sech	双曲線正割関数
sin	正弦関数
sinh	双曲線正弦関数
tan	正接関数
tanh	双曲線正接関数

逆三角関数と逆双曲関数

関数名	概要
acos	逆余弦関数
acosh	逆双曲線余弦関数
acot	逆余接関数
acoth	逆双曲線余接関数
acsc	逆余割関数
acsch	逆双曲線余割関数
asec	逆正割関数
asech	逆双曲線正割関数
asin	逆正弦関数
asinh	逆双曲線正弦関数
atan	逆正接関数
atan2	逆正接関数
atanh	逆双曲線正接関数

尚、逆正接関数には atan 関数 と atan2 関数の二種類がありますが、atan2 関数は、atan2(y, x) の様に引数を二つ必要とする関数で、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應する事も可能です。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)
          ----- - 10 cos(x) sin (x) + 5 cos (x) sin(x) + x
          cos(x)
(%o43)
(%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。

これらの変数とは別に、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 は三角関数の引数を評価し、式を自動的に変換しています。

三角関数に付随する関数は、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。

—— 三角関数に関連する大域変数 ——

変数名	初期値	概要
<code>halfangles</code>	<code>false</code>	半角公式の自動適用を制御
<code>trigexpandplus</code>	<code>true</code>	和公式の自動適用を制御
<code>trigexpandtimes</code>	<code>true</code>	角度の積による展開を制御
<code>triginverses</code>	<code>all</code>	逆関数との合成による簡易化を制御
<code>trigsign</code>	<code>true</code>	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` の影響を受けません。

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` 関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、三角関数の引数が整数、或いは有理数倍の式に対し、三角関数の自動展開が実行されます。

`triginverses` は三角関数、双曲関数とその逆関数との合成の簡易化を制御します。

- `all`
両方、例えば、 $\operatorname{atan}(\tan(x))$ と $\tan(\operatorname{atan}(x))$ の両方が x に簡易化されます。
- `true`
 $\operatorname{arcfunction}(\operatorname{function}(x))$ の簡易化が切り捨てられます。
- `false`
 $\operatorname{arcfunc}(\operatorname{func})$ と $\operatorname{fun}(\operatorname{arcfun}(x))$ の簡易化が切り捨てられます。

`trigsign` が `true` であれば三角関数に対し負の引数の自動簡易化を行います。例えば、`trigsign` 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関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 指数関数と対数関数

5.2.1 指数関数と対数関数の概要

Maxima には指数関数 `exp` と対数関数 `log` があります。指数関数 `exp` は Maxima 内部では、自然底 `%e` の冪として表現されています。

これらの関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。

対数関数に関連する大域変数

変数名	初期値	概要
<code>%e.to.numlog</code>	false	指数に対数を持つ冪乗の簡易化
<code>logabs</code>	false	<code>log</code> を含む不定積分の結果を制御
<code>logarc</code>	false	逆三角関数や逆双曲関数を対数関数で表現
<code>logconcoeffp</code>	false	<code>logcontract</code> で潰される係数を制御
<code>logexpand</code>	true	対数関数の積や冪の自動変換
<code>lognegint</code>	false	対数関数の引数が負の場合の処理を制御
<code>lognumner</code>	false	対数関数の浮動小数点引数の制御
<code>logsimp</code>	true	<code>log</code> を含む指数関数の冪乗の自動化を制御

まず、`%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` 関数も、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` 関数が結果に含まれる場合、`log` 関数が `log(abs(...))` で置換えられます。但し、`logabs` が false であれば `log(...)` 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、逆三角関数、逆双曲関数を対数関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 `logconcoeffp` は `logcontract` を用いた時に潰される係数の制御に利用する述語関数名を一つ設定します。関数名は評価を行わない様に、名詞型にします、`log` 関数から `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、述語関数は、`log` 関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 ratnum(m)
```

ここで、`logcontract(1/2*log(x))` と入力されると、`logcontract` 関数は、評価を実行する前に、`logconcoeffp` に設定された評価関数に与式に含まれる `log` 関数の係数 $\frac{1}{2}$ を引き渡します。尚、false の場合は `featurep(1/2,integer)` で評価します。述語関数の結果が true であれば、この例では、`log(x1/2)` 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(ab)` を `b*log(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` の負の浮動小数指数は `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 対数関数に関連する関数

対数関数を潰す関数

```
logcontract((式))
```

`logcontract` 関数は `log` を含む式を簡易化で潰すものです. 即ち, `log` 関数の係数を指数に取込み, その結果によって, `log` から `sqrt` 等の関数に変換します. 具体的には, `(式)` を再帰的に調べて, $a_1 \log(b_1) + a_2 \log(b_2) + c$ の形の部分式を $\log(\text{ratsimp}(b_1^{a_1} * b_2^{a_2})) + c$ に変換します.

```
(%i8) 2*log(x)+4*log(y)+8;
(%o8)          4 log(y) + 2 log(x) + 8
(%i9) logcontract(%);
(%o9)          2 4
              log(x y ) + 8
(%i10) declare(n,integer);
(%o10)          done
(%i11) logcontract(2*log(x)+4*n*log(y)+8);
(%o11)          2 4 n
              log(x y ) + 8
(%i12) logcontract(2*log(x)+4*m*log(y)+8);
(%o12)          4      2
              m log(y ) + log(x ) + 8
```

この `logcontract` 関数は `log` 関数の整数係数に対して影響を与える関数です. 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 $4*n$ は述語関数 `featurep(coeff,integer)` を満たす為, $\log(x^2 y^{4n}) + 8$ 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( $\langle x \rangle$ )
```

plog 函数は複素数値の自然対数函数の主分枝を $-\pi < \text{carg}(x) \leq \pi$ とします.

極座標形式に変換する函数

```
polarform( $\langle \text{式} \rangle$ )
```

polarform 函数は与えられた $\langle \text{式} \rangle$ を $r * e^{(i * \text{theta})}$ 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そのまま返され, 函数を含む場合には, その函数が負であれば $e^{i * \text{pi}}$ 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 関数と rhs 関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;
```

```
(%o17)          2          2
              x  + 2 x + 1 = y
```

```
(%i18) lhs(eq1);
```

```
(%o18)          2
              x  + 2 x + 1
```

```
(%i19) rhs(eq1);
```

```
(%o19)          2
              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 関数と rhs 関数は演算子=に対してのみ使える関数です。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 等の初等関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、`[eq1, ..., eqn]` 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];
```

```
(%o25)          2          2
              [2 x  - 5 y = 1, y  + x y + x = 4]
```

```
(%i26) eq2[1];eq2[2];
```

```
(%o26)          2
              2 x  - 5 y = 1
```

```
(%i27)
```

```
(%o27)          2
              y  + x y + x = 4
```

この例では、二つの方程式 $2x^2 - 5y = 1$ と $x + yx + y^2 = 4$ 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值的に解く函数に 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);
(%o9)          2      3
          (x - 1) (x + 1)
```

この例では、最初に方程式 $x^2 - 4x + 4 = 0$ を solve 関数を用いて解き、次に、realroots 関数を使って、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 関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 関数は %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、これらの関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 関数と realroots 関数を用いて計算出来ます。

数値解を求める関数

```
allroots(< 方程式 >)
realroots(< 多項式 >, < 許容範囲 >)
realroots(< 多項式 >)
```

最初の allroots 関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 関数は、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。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);
(%o16) 2
      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 関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);
(%o34) [x = - ----, x = ----]
      370727      370727
      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関数は、他に solve 関数があります。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 関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 関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関数 `nroots` 関数があります。この `nroots` 関数は 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関数 —————

```
nroots(<多項式> , <下限> , <上限> )
```

`nroots` 関数は `<上限>` と `<下限>` で指定された半開区間 $(\text{<下限>}, \text{<上限>}]$ 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` 関数と `realroots` 関数は、1 変数多項式の近似解を計算する関数です。これに対し、方程式の厳密解を計算出来る関数として、`linsolve` 関数、`algsys` 関数と `solve` 関数があります。

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` 関数が使えます。

————— 線形連立方程式を解く関数 —————

```
linsolve([<方程式1> , <方程式2> , ...], [<変数1> , <変数2> , ...])
```

`linsolve` 関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
<code>linsolve_params</code>	<code>true</code>	解に助変数を導入
<code>linsolvewarn</code>	<code>true</code>	<code>linsolve</code> の警告を抑制

`linsolve_params` が `true` であれば, `linsolve` はまた記号 `%r1` 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` 関数や `algsys` 関数で解く事が出来ます. 但し, `solve` 関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` 関数は, 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 関数

```
algsys([<方程式1>, <方程式2>, ...], [<変数1>, <変数2>, ...])
```

`algsys` 関数では, 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]);
```

```
(%o1) [[x = %r1, y = -  $\frac{2 \%r1 - 1}{3}$ ]]
```

```
(%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.

- 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.
- 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、 $res(f, g, x) = a_m^n b_n^m \prod_{0 \leq i \leq m, 0 \leq j \leq n} (\alpha_i - \beta_j)$ 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。

$$f : ax + by + p = 0 \quad g : cx + dy + q = 0$$

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 関数は、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。この処理を繰り返す事で、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 はこのようなアルゴリズムを採用しているのです。

algsys 関数に影響を与える大域変数

変数名	初期値	概要
%rnum_list	[]	algsys 関数で解に導入された変数リスト
algexact	false	algsys 関数が solve 関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 関数は実数解のみを返却

%rnum_list は algsys 関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 関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 が失敗した時に近似解のみを生成します。

algepsilon は algsys 関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 関数が利用出来ます。この solve 関数は、与えられた方程式系の厳密解を返す関数です。方程式には、sin 等の三角関数、指数関数や対数関数を含んだ方程式が扱えます。但し、algsys 関数と違い、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 関数

```
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 関数は代数方程式 <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 関数は解釈します。

〈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〉, ..., 〈方程式_n〉], [〈変数₁〉, ..., 〈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〉, $i=1, \dots, n$] 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 関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
solvetricwarn	true	方程式を解く際に逆三角関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。

solvetricwarn 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ではなく、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。

漸化式を解く関数

```
funcsolve(<方程式>, <g(t)>)
```

funcsolve 関数は `<方程式>` を満たす有理関数 `<g(t)>` が存在すれば有理関数のリストを返し、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)

(%o28)

$$\text{foo}(n) = \frac{n}{(n + 1)(n + 2)}$$

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のでしょうか？ 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、 $\frac{\sin(x)}{x} = 1 + x \cdot (x \text{ の冪級数})$ となります。以上から、 x を 0 に近づけると 1 になる事が判ります。

Maxima で試してみましよう。Maxima には `limit` 関数があり、この関数は `limit(⟨ 関数 ⟩, ⟨ 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]);
```

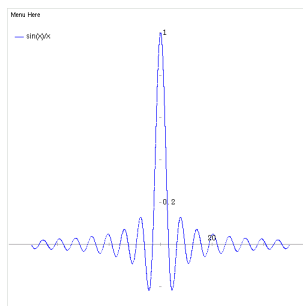
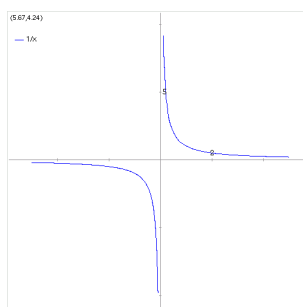


図 5.1: $\sin(x)/x$ のグラフ

極限はこの様な計算を行いますが、この近づ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。この関数は $x > 0$ 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。

```
(%i73) limit(1/x,x,0);
(%o73) und
(%i74) limit(1/x,x,0,plus);
(%o74) inf
(%i75) limit(1/x,x,0,minus);
```

図 5.2: $1/x$ のグラフ

```
(%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, $\text{limit}(1/x, x, 0)$ の結果は `und` となっています.

`limit` では正の無限大は `inf`, 負の無限大は `minf`, 複素数での無限大は `infinity`, 左右の極限が異なる場合には `und`, 未定でも有界なものには `ind` といった表記を返します.

```
(%i89) limit(1/(x^2-1), x, 1, plus);
(%o89)                                inf
(%i90) limit(1/(x^2-1), x, 1, minus);
(%o90)                                minf
(%i91) limit(1/(x^2-1), x, 1);
(%o91)                                und
(%i92) limit(sin(1/x), x, 0);
(%o92)                                ind
(%i93) limit(1/(x^2+1), x, %i);
(%o93)                                infinity
```

————— 極限に関連する大域変数 —————

変数名	初期値	概要
lhospitallim	4	limit 関数で用いられる 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 $\lim_{x \rightarrow 0} (\cot(x)/\csc(x))$ の様な場合に無限ループに陥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 $\lim_{n \rightarrow \infty} (f(n)/f(n+1))$ 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、このよう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 関数 —————

```
limit(⟨式⟩,⟨変数⟩,⟨値⟩,⟨方向⟩)
limit(⟨式⟩,⟨変数⟩,⟨値⟩)
limit(⟨式⟩)
```

limit 関数は与えられた ⟨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 関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 関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 微分に関する関数

Maxima の式の微分では diff 関数を用います。

diff 関数

```
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, $\text{diff}(\langle \text{式} \rangle, \langle \text{変数} \rangle)$ の様に階数を省略しても構いませんが, 通常は $\langle \text{変数}_i \rangle$ と $\langle \text{階数}_i \rangle$ の一組を指定して $\langle \text{式} \rangle$ の微分を行います。

$\text{diff}(\langle \text{式} \rangle)$ は全微分を与えます。即ち, $\langle \text{式} \rangle$ の各変数に対する微分と, 各変数の関数 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, 関数の名詞型 '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
<code>derivsubst</code>	<code>false</code>	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、このよう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、`subst` 関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関数があります。

————— derivdegree 関数 —————

derivdegree(〈式〉, 〈従属変数〉, 〈独立変数〉)

derivdegree 関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。この関数は多項式の次数を返す関数 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);
```

```
(%o18) 5 2
        d y d y
        --- + ---
        5 2
        dx dx
```

```
(%i19) derivdegree(%,y,x);
```

```
(%o19) 5
```

但し、derivdegree 関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);
```

```
(%o26) 5 2 5 2
        2 d y (x - 1) --- - (x - 1) (x + 1) --- + ---
        5 dx 5 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));
```

```
(%o28) 2
        d y
        ---
        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 関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関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関数が収録されています。

vect パッケージに含まれる主な関数

```
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 変数に割り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ものではありません。vect パッケージの関数 vectsimp 関数を用いて簡易化を行います。但し、vectorsimp 関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 関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 div 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 $p \sim (q \sim r)$ を $(p, r) * q - (p, q) * r$ で置換するかどうかを決めます.

expandcurlcurl は $\text{curl curl } p$ を $\text{grad div } p + \text{div grad } p$ 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);
(%o2)          2 2 2
          laplacian (x y z )
(%i3) express(e1);
(%o3)          2          2          2
          d 2 2 2 d 2 2 2 d 2 2 2
          --- (x y z ) + --- (x y z ) + --- (x y z )
          2          2          2
          dz          dy          dx
(%i4) ev(%, 'diff);
(%o4)          2 2      2 2      2 2
          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 は初等関数 (有理式, 三角関数, 対数, 指数) と多少の拡張 (error 関数, dilogarithm) で可積分なものに限定しているので, $g(x)$ や $h(x)$ の様な未知関数の積分は扱いません.

記号積分を行う関数

```
integrate( < 式 >, < 変数 > )
integrate( < 方程式 >, < 変数 > )
integrate( [ < 式1 >, ..., < 式n > ], < 変数 > )
integrate( [ < 方程式1 >, ..., < 方程式n > ], < 変数 > )
integrate( < 行列 >, < 変数 >, < 下限 >, < 上限 > )
risch( < 式 >, < 変数 > )
```

Maxima で記号積分を行う関数に, integrate 関数と risch 関数の二種類があります. integrate 関数から, risch 関数の本体を呼出す事も, ev 関数を利用する事で出来るので, 記号積分に関しては integrate 関数だけで済ませられます.

積分の計算で式が非常に複雑になる場合や, 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 関数は, 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 関数は比較の演算子 (=, >, <) を含まない通常の式の積分に限定されます.

ここで, integrate 関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` 関数が呼出す `sinint` 関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` 関数から `risch` 関数を用いて Risch 積分を実行させる事が可能です。この場合は `ev` 関数を用います。

```
(%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 関数は depends 関数で設定される大域変数 dependencies の影響を受けません.

integrate 関数は定積分の計算も出来ます. この場合は defint 関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 関数です. 尚,defint 関数による定積分は, 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 関数や quanc8 関数 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 関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 関数は Risch の積分アルゴリズムから, Transcendental 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 関数は,integrate の主要部が処理出来ない入れ子状態の指数関数と対数関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 関数を適用します.

```
(%i24) risch(x^2*erf(x),x);
```

$$\frac{\pi x^3 \operatorname{erf}(x) + (\sqrt{\pi} x^2 + \sqrt{\pi}) e^{-x^2}}{3 \pi}$$

```
(%i25) diff(%,x),ratsimp;
```

$$x^2 \operatorname{erf}(x)$$

risch 関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 関数による erf 関数の挿入を制御

この大域変数 erfflag が false であれば,erf 関数が被積分関数の中に含まれていない場合,risch 関数が答に erf 関数を入れる事を抑制します.

ここで,erf 関数は次の error 関数の事です.

erf 関数

$$\operatorname{erf}(x)$$

この関数の微分 $\frac{d}{dx}(\operatorname{erf}(x))$ 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
      >    %e    a
      /              i
      ====
      i = 0
(%i4) changevar(%,i-5-n,n,i);
      inf
      ====
      \      n
      >    %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
<code>integrate_use_rootsof</code>	<code>false</code>	代数的数を用いた <code>integrate</code> 関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` 関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)
> -----
/      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` 関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。この意味を上例で説明しましょう。まず, `integrate_use_rootsof` が `true` に設定されると, Maxima の `integrate` 関数は与式の分母 $x^3 - x^2 - x + 4$ 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て必要は無く、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 関数や risch 関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一目で判ります。

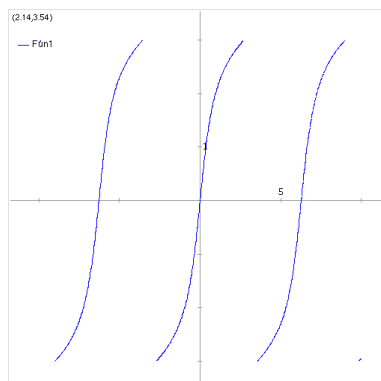


図 5.3: $2\operatorname{atan}\left(\frac{3\sin(x)}{\cos(x)+1}\right)$ のグラフ

この様に、結果のグラフを描いて見るのも、計算結果を確認する上では非常に有効な手段です。

Maxima は積分処理の為に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いません。

```
(%i24) integrate(sqrt(2*x-x^2),x,0,a);
```

```
Is a positive, negative, or zero?
```

```
pos;
```

```
(%o24) 
$$\frac{\operatorname{asin}(a-1) + (a-1)\sqrt{2a-a^2}}{2} + \frac{\pi}{4}$$

```

```
(%i25) assume(a>0 and a<1);
```

```
(%o25) [a > 0, a < 1]
```

```
(%i26) integrate(sqrt(2*x-x^2),x,0,a);
```



```
(%o106)
```

$$\frac{e^{a(2s-4)}}{(s^2-4s+5)^2}$$

```
(%i107) ilt(%,s,x);
```

```
(%o107)
```

$$x e^{2x+a} \sin(x)$$

ilt 関数は〈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 関数による処理を効率的に行うためには有理式の展開を予め実行しておくとい良いでしょう。

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;
```

```
(%o11)
```

$$\int_0^t f(t-x) \sinh(ax) dx + b f(t) = t^2$$

```
(%i12) laplace(%,t,s);
```

```
(%o12)
```

$$b \operatorname{laplace}(f(t), t, s) + \frac{a \operatorname{laplace}(f(t), t, s)}{s^2 - a^2} = \frac{t^2}{s^3}$$

```
(%i13) linsolve([%],[ 'laplace(f(t),t,s)]);
```

```
(%o13)
```

$$[\operatorname{laplace}(f(t), t, s) = \frac{2s^2 - 2a}{b s^5 + (a - a^2 b) s^3}]$$

```
(%i14) ilt(rhs(first(%)),s,t);
Is a b (a b - 1) positive, negative, or zero?
```

```
pos;
```

$$\frac{\sqrt{a b (a b - 1)} t}{2 \cosh\left(\frac{b}{a t}\right)} + \frac{a t}{a b - 1} + \frac{2}{a b - 2 a b + a}$$

```
(%o14) ----- + ----- + -----
          3 2      2          3 2      2
          a b - 2 a b + a      a b - 1      a b - 2 a b + a
```

delta 関数は Dirac の δ 関数です. 尚,laplace 関数のみが δ 関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;
```

$$\sin(a b) e^{-a s}$$

```
(%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
```

定積分を行う関数

```
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 関数は定積分を実行する関数です. 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,このよう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);
```

```
(%o7) (limit 1/x - 1)
      x -> 0+ x
```

```
(%i8) ldefint(1/x^2,x,zerob,-1);
```

```
(%o8) (limit 1/x + 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 関数は大域変数 tlimswitch が true に設定された ldefint 関数に相当します.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。この処理を行う関数には `quanc8` 関数と `romberg` 関数があります。

数値積分を行う関数

```
quanc8( '〈被積分関数〉, 実数1, 実数2 )
quanc8( 〈被積分関数〉, 変数, 実数1, 実数2 )
romberg( 〈被積分関数〉, 〈積分変数〉, 〈実数1〉, 〈実数2〉 )
romberg( 〈被積分関数〉, 〈実数1〉, 〈実数2〉 )
```

先ず、`quanc8` 関数は `load("qq")` で予め読み込む必要があります。 `quanc8('〈被積分関数〉, 実数1, 実数2)` で、最初の引数で指定された関数の定積分を区間 実数₁ から 実数₂ で計算します。第一引数に関数名を用いる場合、関数名に単引用符 ' を付けなければなりません

`quanc8(〈被積分関数〉, 変数, 実数1, 実数2)` で、関数か式 (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` 関数は `romberg` 積分を行う関数です。`romberg` 関数は `quanc8` 関数とは違い、Maxima に自動的に読み込まれます。

最初の引数は `translate` 関数で変換された関数か `compile` 関数でコンパイルされた関数でなければなりません。尚、`compile` 関数でコンパイルされた関数の場合は、浮動小数点型、即ち、`flonum` 型の関数として宣言されていなければなりません。関数が `translate` 関数で変換されたものでなければ、`romberg` 関数は `translate` 関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` 関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` 関数の 〈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` 関数は計算結果を返します。諦める前に大域変数 `rombergit` 倍の刻幅を半分にして試みます。`romberg` 関数は反復と関数評価の大域変数 `rombergabs` と `rombergmin` で制御されます。

一般的に、`quanc8` 関数の方が計算速度、精度と安定性で `romberg` 関数に勝っている事が多いのですが、`romberg` 関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` 関数の方が精度で勝り、 $e^{-x}\sin(x)$ の積分では計算速度で `romberg` 関数が勝っています。

```
(%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 関数に影響を与える大域変数

変数名	初期値	概要
rombergabs	0.0	romberg 関数の終了条件を与える変数
rombergit	11	刻幅を設定
rombergtol	1.0E-4	romberg 関数の精度
rombergmin	0	関数評価の最小回数

大域変数 rombergabs は romberg 関数の終了条件を与える大域変数の一つです。romberg 関数内部の反復処理で生成された値の列を $y[0], y[1], y[2], \dots$ とする時、 n 回目の反復処理で、 $(\text{abs}(y[n]-y[n-1]) \leq \text{rombergabs})$ か $\text{abs}(y[n]-y[n-1])/(y[n]=0.0)$ ならば 1.0、それ以外は $y[n] \geq \text{rombergtol}$ を満した時点で、romberg 関数は答を返します。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 rombergit を半分にして試行します。

大域変数 `rombergmin` は `romberg` 関数による関数評価の最小数を制御します。 `romberg` 関数はその第一引数を少なくとも $2^{\text{rombergmin}+2} + 1$ 回評価します。これは通常の収束テストが時々悪い通り方をする周期的関数の積分に有効です。

————— 二重積分を行う関数 —————

```
dblint( ' $F$ ', ' $R$ ', ' $S$ ',  $\langle$ 浮動小数点 $\rangle_1$ ,  $\langle$ 浮動小数点 $\rangle_2$  )
```

`dblint` 関数は二重積分の数値計算を行う関数で、Maxima の処理言語で記述されています。この関数を利用する為には予め `load(dblint)` で `dblint` 関数を読み込んでおく必要があります。

まず、`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

$$\int_{\text{浮動小数点}_1}^{\text{浮動小数点}_2} \int_{R(x)}^{S(x)} F(x, y) dy dx$$

第一引数の $\langle F \rangle$ は x, y の 2 変数の関数、第二引数と第三引数の $\langle R \rangle$ と $\langle S \rangle$ は各々変数 x の関数で、`dblint` 関数には関数名のみを名詞型で引渡します。更に、これらの関数は全て、`translate` 関数で LISP の関数に変換されたものか、`compile` 関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、これらの関数は全て `flonum` 型の関数でなければならず、多倍長精度は扱えません。

`dblint` 関数は $\langle R \rangle$ と $\langle S \rangle$ の両方に Simpson 則を用います。そして、`dblint` 関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` 関数は 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 軸に沿った積分を関数の値を 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関数や特異点(領域に極や分岐点)を持つ関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。

関数値は保存されない為、その関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` 関数に影響を与える大域変数 —————

変数名	初期値	概要
<code>dblint_x</code>	10	X 軸方向の分割数
<code>dblint_y</code>	10	Y 軸方向の分割数

二重積分を行う関数 `dblint` 関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、 $2 * \text{dblint}_x + 1$ 点が X 方向に計算され、 Y 方向は $2 * \text{dblint}_y + 1$ 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が可能です。

5.6.6 antid パッケージ

antid パッケージには antid, antidiff と nonzeroandfreeof という関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 関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, 関数 ⟨u(x)⟩ は未知関数でも構いません.

```
(%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 関数を用いた関数で, antid 関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
]
/
```

関数 nonzeroandfreeof は述語関数で, ⟨y⟩ が零でなく, ⟨x⟩ が ⟨y⟩ に含まれない場合に true を返す関数です. この述語関数を用いて, 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関数に ode2 関数と desolve 関数があります。ここで、desolve 関数で扱える常微分方程式は、未知関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 関数にとっては正確な書式ではありません。何故なら、関数 f と関数 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 関数の場合、常微分方程式の未知関数が何の関数であるかを下記のよう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 関数に対し、ode2 関数は $x^2 \cdot \text{diff}(y,x) + 3 \cdot y \cdot x = \sin(x)/x$ 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 関数を用いる場合と ode2 関数を用いる場合で解き方の手順が異なります。

最初に、desolve 関数を用いて初期値問題を解く場合、atvalue 関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 関数による初期値設定は desolve 関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 関数は函数 f が一変数の場合、 $\text{atvalue}(f(x),x=\text{pt},\text{val})$ の様に函数 f(x) に対する $x=\text{pt}$ の値 val を設定します。

f が多変数の場合、 $\text{atvalue}(f(x,y,\dots),[x1=p1,x2=p2,\dots],[val1,val2,\dots])$ 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 関数を用いる場合には、与えられた一般解と初期条件から特殊解を計算する bc 関数、ic1 関数や ic2 関数を併用します。

```
(%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 関数を用いてその一般解を求め、ic1 関数から、 $x = \pi$ 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;
```

$$\begin{aligned} & \frac{d^2 y}{dx^2} + y \left(\frac{dy}{dx} \right)^3 = 0 \end{aligned}$$

```
(%i92) ode2(%,y,x);
```

$$\begin{aligned} & \frac{y^3 + 6 \%k1 y}{6} = x + \%k2 \end{aligned}$$

```
(%i93) ratsimp(ic2(%o92,x=0,y=0,'diff(y,x)=2));
```

$$\begin{aligned} & - \frac{2 y^3 - 3 y}{6} = x \end{aligned}$$

```
(%i94) bc2(%o92,x=0,y=1,x=1,y=3);
```

$$\begin{aligned} & \frac{y^3 - 10 y}{6} = x - \frac{3}{2} \end{aligned}$$

一般解を求める関数

```
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 関数は、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 関数を呼出す前に atvalue 関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 関数では、求める関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 関数は 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 関数による評価の一つの形態です。詳細は 1.6.1 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 関数は 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 関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..

境界値問題を解く関数

```
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 関数は、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 関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 $x=x_0$ や $y=y_0$ の様に〈対応する変数〉=〈境界値〉の書式で記述します。

ic1 関数は初期値問題 (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 $x = x_0$ や $y = y_0$ の様に〈対応する変数〉=〈境界値〉の書式になります。

ic2 関数は二階の常微分方程式の境界値問題を解く関数です。〈一般解〉は ode 関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、 $x = \langle x \text{ の値} \rangle$ での値が〈y の微分値〉となります。

5.8 Maxima のグラフ表示

5.8.1 はじめに

Maxima には目的に応じて様々な描画が可能な函数を幾つか持っています。Mathematica や Maple といった商用の数式処理システムと比較すると、流石に見劣りする面もありますが、一寸した式やデータの可視化ではそこその機能は持っています。Maxima での描画の特徴としては、Maxima で生成したデータを外部のアプリケーションに引渡して、そのアプリケーションで描画を行う点です。函数によっては、この表示用のアプリケーションを目的に応じて切り替える事が可能なものもありますが、特定のアプリケーションのみに対応したものもあります。

Maxima の標準的な描画函数は `plot2d` 函数と `plot3d` 函数です。これらの函数は各々、2次元グラフと3次元グラフを専門に描く函数です。`plot2d` 函数や `plot3d` 函数以外の函数で、機能的にこれらの函数に勝る物は `draw` パッケージのものを除くとありません。

猶、`draw` パッケージでは `gnuplot` のバージョンが 4.2 以上に限定されています。ところで、MS-Windows 版では `draw` パッケージにも対応した `gnuplot` が附属している為に、最初から利用可能ですが、その他の環境では、`gnuplot` を 4.2 以上に入れ直しておく必要があります。その為、`draw` パッケージについては次の章で詳細を述べる事とし、ここでは上記の `plot2d` 函数と `plot3d` 函数を中心に解説します。

5.8.2 表示アプリケーションについて

Maxima は描画用の特別なモジュールを持っている訳ではありません。Maxima の描画函数の動作手順は、最初に Maxima 内部で描画用のデータを生成し、それをファイルに出力して Maxima から外部のアプリケーションを呼出し、そのデータの描画を外部アプリケーションに任せて行うという手法を採用しています。

さて、`plot2d` 函数と `plot3d` 函数の大きな特徴としては、目的に応じて様々な外部アプリケーションを呼出す事が可能な点です。ここで、この `plot2d` 函数と `plot3d` 函数で利用可能な外部アプリケーションで代表的なものを以下に挙げておきましょう。

— `plot2d` 函数や `plot3d` 函数で利用可能なアプリケーション —

<code>gnuplot</code>	汎用 (デフォルト)
<code>openmath</code>	汎用
<code>Geomview</code>	3次元グラフ描画専用

Maxima-5.13.0 のデフォルトの描画アプリケーションは `gnuplot` です。但し、以前のデフォルトは `openmath` でした。`gnuplot` と `openmath` を単純に比較すると、`gnuplot` の方が遥かに高機能ですが、Maxima 側からグラフを一寸描く程度の処理であれば大差の無い処理が出来ます。寧ろ、`openmath` の方が3次元グラフィックスが綺麗かもしれません。更に、`openmath` は Maxima のソースファイルに附属している事もあって、Maxima が利用可能な環境であれば、`openmath` も使えるという長所もあります。

最後の Geomview は 3 次元グラフ専用ですが、非常に高品質のグラフを表示する事が可能なだけでなく、Euclid 空間以外の空間内部でのグラフの表示も可能です。但し、Geomview の動作環境は基本的に UNIX 環境に限定されます。

5.8.3 フロントエンド

plot2d 函数や plot3d 函数にはこれといって特殊なフロントエンドがある訳ではありません。基本的に描画用のデータファイルを出力した後は、呼出したアプリケーションに全てを任せる為です。但し、Maxima-5.12.0 から導入された新しい方式を利用すれば、外部アプリケーションを gnuplot に指定した場合に限って Maxima から再描画の実行といった処理が行える様になっています。

猶、openmath が以前のデフォルトだった事もあって、Maxima のフロントエンドの一つの xmaxima の入力副ウィンドウ内にグラフを表示させる事が可能です。

5.8.4 plot2d 関数と plot3d 関数の動作

最初に plot2d 関数と plot3d によるグラフ表示について簡単に説明しましょう。これらの関数を用いてグラフの表示を行う場合、Maxima 内部で指定された外部アプリケーション向けのグラフデータを最初に生成します。外部アプリケーションの指定は大域変数 `plot_options` の `plot_format` に対して行います。又、データファイルの置かれる場所は UNIX 環境ではホームディレクトリ上、MS-Windows 環境の場合は、Documents and Settings フォルダの中にあるログインユーザー名のフォルダになります。

ここで、データファイルの内容は外部アプリケーションによって詳細は異なりますが、基本的に曲線や曲面を構成する点の座標値を示す数値データを中心に構成されたもので、Maxima で plot2d 関数や plot3d 関数に入力した式そのものが含まれている訳ではありません。

それから Maxima は外部アプリケーションに生成したファイルを引渡し、外部アプリケーションの立ち上げを行います。すると外部アプリケーションが実際の描画を行います。この様に、グラフデータファイルを引渡ししまうと後は外部アプリケーション任せになります。

外部アプリケーションの切り替えは前述の様に、`plot_format` に値を設定する事で行いますが、大域変数 `plot_options` に含まれる `run_viewer` の設定で外部アプリケーションを起動させずに、外部アプリケーション向けのデータのみを生成させる事も可能です。

さて、ここで外部アプリケーションに引渡されるファイルの名前は、外部アプリケーション毎に異なります。具体的には、`openmath` で `maxout.openmath`、そして、`Geomview` では `maxout.geomview`、`gnuplot` の場合は、`maxout.gnuplot` と `maxout.gnuplot_pipes` と `maxout` の後に `plot_format` の値に対応するアプリケーション名が付いたものです。

ここで `gnuplot` には `gnuplot` と `gnuplot_pipes` の二種類の設定可能な `plot_format` の値があります。最初の `gnuplot` を選択すると、一旦、グラフを表示させた場合、Maxima 側から設定の変更や再描画は行えず、マウス操作によるグラフの拡大や縮小、3次元グラフの回転といった操作が行なえません。それに対して `gnuplot_pipes` の場合、Maxima 側からの設定の変更や再描画やマウスによる画像操作が可能になります。

ここでマウスによる画像の直接操作では `gnuplot` の `mouse` を `on` にする事で行っています。逆に `mouse` を `off` にしてウィンドウの直接操作を停止したければ、グラフを表示しているウィンドウ上で、キーボードから直接 `m` と入力します。そして、`off` から `on` にする場合も同様に `m` と入力します。猶、`gnuplot` を立ち上げて処理を行っている場合は、`gnuplot` の命令入力ウィンドウから、`set mouse` を実行すると `on` に、`uset mouse` を入力すると `off` に出来ます。猶、MS-Windows 版固有の機能として、`mouse` を `off` にした状態からメニューを呼出して、表示画像をクリップボードに保存する事が可能です。その為には、`mouse` を `off` にした状態で右マウスボタンをクリックします。するとメニューが現われるので、そのメニューから `Copy to Clipboard` を選択すると画像がクリップボードに貼り付けられます。

次に `maxout.gnuplot_pipes` ファイルと `maxout.gnuplot` ファイルの場合の違いを解説しておきましょう。先ず、`maxout.gnuplot_pipes` には描画する為の数値データのみが格納されており、描画命令や設定等の `gnuplot` の命令文は Lisp のストリームを用いて `gnuplot` に送り込まれます。

それに対し、`maxout.gnuplot` ファイルの内容は、`maxout.gnuplot_pipes` に含まれている曲面や曲線のデータに加え、`gnuplot` の描画命令や設定を含む命令文がその先頭に入っています。その為、別

途立ち上げた gnuplot から `load 'maxout.gnuplot'` で読み込めば, Maxima の時と同じグラフが表示されます. 更に, gnuplot で mouse を on にしておけば, マウスによる画像の回転や拡大が可能になります.

猶, MS-Windows のみは gnuplot ではなく wgnuplot を利用します. 但し, ここでは gnuplot と wgnuplot を区別せずに, gnuplot と呼んでいます. 猶, この wgnuplot は MS-Windows 版のバイナリに同梱されているので別途入手する必要はありません. この wgnuplot は Maxima のフォルダ (通常は Program Files フォルダにある筈です) の bin フォルダにあります. 因に, この wgnuplot は独立して動かす事も可能です. 但し, 最初に立ち上げた時点では, フォントが潰れて読めない状態なので, gnuplot のウィンドウで右マウスボタンをクリックし, Choose Font を選択して適当なフォントに設定し直せば読める様になります.

5.8.5 plot2d 関数

plot2d 関数は平面曲線の描画を行う関数です. 以下にその構文を示しておきます.

— plot2d の構文 —

```
plot2d (< 式 >, < 定義域 >, < オプション1 >, ..., < オプションn >)
plot2d (< 式 >, < 定義域 >, < 値域 >, < オプション1 >, ..., < オプションn >)
plot2d (< 媒介変数式 >, < オプション1 >, ..., < オプションn >)
plot2d (< 媒介変数式 >)
plot2d (< 離散式 >, < オプション1 >, ..., < オプションn >)
plot2d (< 離散式 >)
plot2d ([[ 式1 ], ..., [ 式n ]], < 定義域 >, < 値域 >, < オプションn >)
plot2d ([[ 式1 ], ..., [ 式n ]], < 定義域 >, < 値域 >)
plot2d ([[ 式1 ], ..., [ 式n ]], < 定義域 >)
```

plot2d 関数は二次元グラフの表示を行います. plot2d 関数で描ける式は, 特に指定しない場合, 1 変数を含むの Maxima の式ですが, 利用者が記述した一変数の Maxima の関数, 複素値関数で実部が 1 実数変数を含む式の実部, 関数を媒介変数を用いて記述したもの, そして, 離散的なデータの表示が可能です.

先ず, 通常 of Maxima の式を描画する場合, その変数の定義域が必要になります. 定義域は [x 変数], \langle 下限 \rangle , \langle 上限 \rangle] で構成されたリストです. 又, グラフの縦方向の表示範囲を指定する値域もこれと似たリストですが, ここでの変数は y に固定されている為, [y , \langle 下限 \rangle , \langle 上限 \rangle] になります. 但し, 離散式に対する値域の指定は無効になります.

複素関数の実部を表示する場合は大域変数 plot_options の plot_realpart の項目を true に予めしておくか, `[plot_realpart, true]` をオプションとして引渡します. 猶, 大域変数 plot_options の詳細については後の小節で詳細を述べます.

次に, 媒介変数式の書式は以下の二通りの書式が許容されます.

—— 媒介変数式の書式 ——

```
[parametric, <X 座標>, <Y 座標>, <オプション1>, ..., <オプションn>]
[parametric, <X 座標>, <Y 座標>]
```

まず, X と Y 座標は変数 t を唯一の変数として持つ式でなければなりません. 因に, この書式で媒介変数は t に固定されています. その為, 半径 1 の円を描く場合, `[parametric,cos(t),sin(t)]` と表記します. オプションは通常の plot2d 関数のオプションが入れられます. 猶, 媒介変数 t の定義域は, 媒介変数式の中に入れても, plot2d 関数で描く式の定義域として与えても構いません. 図 5.4 に `plot2d([parametric,cos(2*%pi*t/6),sin(2*%pi*t/6)])` の結果を示しておきましょう.

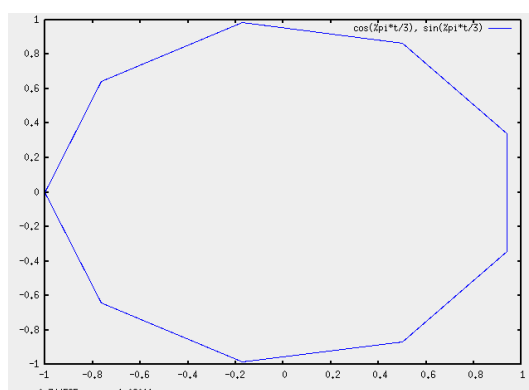


図 5.4: `plot2d([parametric,cos(2*%pi*t/6),sin(2*%pi*t/6)])` の結果

猶, t の定義域を省略する事も可能です. これは大域変数 plot_options に予め t の定義域が `[t,-3,3]` で設定されている為です. ここで t の定義域を変更したい場合は, 大域変数 plot_options の項目 t の定義域する方法と plot2d 関数で定義域を指定する方法があります. 又, 媒介変数式でグラフの粗さが目立つ場合には, 大域変数 plot_options の項目の nticks を 10 よりも大きな値, 例えば, 100 にするか, 或いは, `[nticks,100]` の様に plot2d 関数のオプションとして引渡します.

plot2d 関数は Maxima の式だけではなく, 点列の描画も可能です. 点列の場合は, 媒介変数式と似た書式になります. この場合, 二つの書式が可能です.

—— 離散式の書式 ——

```
[discrete, <X 成分リスト>, <Y 成分リスト>]
[discrete, [ [<X 成分1>, <Y 成分1>], ..., [<X 成分n>, <Y 成分n>]]]
```

まず, 双方の書式共に discrete で開始し, それから, X 成分のリストとそれに対応する Y 成分のリストか, X 成分と Y 成分の対, $[x_i, y_i]$ で構成されるリストの二種類になります.

plot2d 関数は一度に複数の式のグラフを表示する事も可能です.

この場合, 表示する複数の式を 1 つのリストで与えます. ここで, リストに含まれる媒介変数式や離散式以外の Maxima の式は, 定義域が全て同じものでなければなりません.

媒介変数式を含む場合, その媒介変数式毎に定義域やその他のオプションが設定可能です. 図 5.5

に媒介変数式を含む式リストの処理結果を示します。

```
plot2d([x^3+2, [parametric, cos(t), cos(t)*sin(t)]],
[x, -3, 3], [y, -2, 4], [t, -5, 5], [nticks, 100])
```

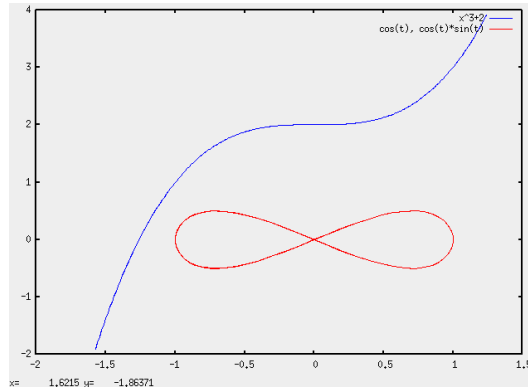


図 5.5: 媒介変数式と通常の式の混在

又、複数の媒介変数式がある場合には、媒介変数式内部にオプションを持たせて媒介変数式毎に設定を行う事が可能です。

```
plot2d([[parametric, cos(t), sin(t), [t, -%pi, %pi], [nticks, 20]],
[parametric, 2*cos(t), sin(-t), [t, -%pi/2, %pi/4], [nticks, 5]]]);
```

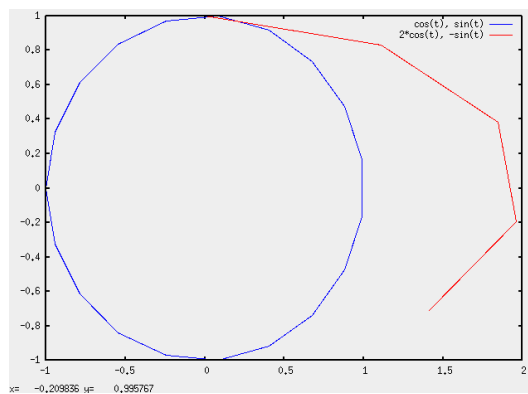


図 5.6: 複数の媒介変数式

この例では、最初の半径 1 の円周の描画を 20 個の点で描画しますが、その次の楕円の描画では、5 点で描画しています。ここでは、nticks の位置に注意して下さい。即ち、各々を媒介変数式の中に記述しているので、その影響は個々の媒介変数式にのみ影響されます。

では以下の様に変更するとどうなるでしょうか？

```
plot2d([[parametric,cos(t),sin(t),[t,-%pi,%pi],[nticks,20]],
[parametric,2*cos(t),sin(-t),[t,-%pi/2,%pi/4],[nticks,5]]],
[nticks,40]);
```

この場合、媒介変数式内部の設定の方が優先される為に、外の `[nticks,40]` は無視され、結果として図 5.6 と同じ絵になります。

ところで、`plot_format` で `openmath` を指定する場合、複数の媒介変数式の中に定義域を記述した時に描画が上手く出来ない事があります。その為、`plot2d` 函数を用いる場合は、`plot_format` をデフォルトの `gnuplot` にしたままの方が無難でしょう。

さて、`plot2d` 函数のオプションは大域変数 `plot_options` を構成するリストでもあります。この大域変数 `plot_options` の諸項目リストを設定する事で、グラフのタイトルやラベル、 X 、 Y 軸のラベル等の細かなグラフの指定が行えます。特に、`gnuplot` を用いる場合、即ち、`plot_format` が `gnuplot`、或いはオプションとして `[plot_format,gnuplot]` を指定した場合、`plot_options` の `gnuplot_preamble` を上手く使う事で、`gnuplot` の命令文を実行させる事が可能です。この大域変数 `plot_options` には多くの説明すべき事項がある為、次の節で詳細を述べる事とします。

5.8.6 plot3d 函数

Maxima では $f(x,y)$ の形式の 2 変数の式の描画が可能です。更に、一変数複素数値函数の実部 $f(x+iy)$ のみの表示も可能です。

但し、 $x^2+y^2+z^3=1$ の様な式のグラフはそのままの形式では描けません。`plot3d` 函数の構文は基本的に `plot2d` 函数の構文を 3 次元にそのまま拡張したものになります。

— plot3d の構文 —

```
plot3d (< 式 >, < 定義域1 >, < 定義域2 >, < オプション1 >, ..., < オプションn >)
plot3d (< 式 >, < 定義域1 >, < 定義域2 >)
plot3d ([< 式1 >, < 式2 >, < 式3 >], < 定義域1 >, < 定義域2 >, < オプション1 >, ..., < オプションn >)
plot3d ([< 式1 >, < 式2 >, < 式3 >], < 定義域1 >, < 定義域2 >)
plot3d ([< 式1 >, < 式2 >, < 式3 >], < 定義域1 >, < 定義域2 >, < 定義域3 >, < オプション1 >, ..., < オプションn >, [transform_xy, < 函数 >])
```

`plot3d` 函数は `plot2d` 函数と似た構文を持っていますが、`plot2d` 函数とは違い、複数の曲面を同時に描く事は出来ません。又、`gnuplot` で曲面を描く場合、`pm3d` の設定を行うと曲面を張りますが、無設定の場合はワイヤフレーム表示になります。詳細は、大域変数 `plot_options` の `gnuplot_pm3d` の項目を参照して下さい。

`plot3d` 函数は三次元空間内の曲面だけではなく、空間曲線も描けます。ここで空間曲線を描く場合は、 x, y, z 成分の各成分を、`< 定義域1 >` か `< 定義域2 >` の変数として記述します。

極座標系の場合も空間曲線を描く方法に似ていますが、この場合には、大域変数 `plot_options` の項目 `transform_xy` に座標変換を行う函数を指定する必要があります。ここで通常のデカルト座標が

ら極座標への変換は単純に `polar_to_xy` を指定するだけで済みます。但し、利用者独自の座標変換関数の指定が必要であれば、`make_transform` 関数を併用する必要があります。この詳細は、大域変数 `plot_options` の `transform_xy` の個所で述べます。

5.9 大域変数 `plot_options`

Maxima はグラフ表示に `gnuplot`, `openmath` 等の外部アプリケーションを用います。アプリケーションに様々な設定を行いたい場合にはどうすればよいでしょうか？先程の実例に示した様に二つの方法がありました。まず、通常の描画を行う `plot2d` 関数や `plot3d` 関数を用いる場合にオプションとして引渡す方法、そして、セッションを通じて設定しておきたい場合に用いる、`set_plot_option` 関数による大域変数 `plot_options` の内容変更を行う方法です。

この小節では、大域変数 `plot_options` の解説を行いましょう。

まず最初に `plot_options` の値を見てみましょう。これは直接 `plot_options;` と入力すれば見る事が出来ます。例えば、Maxima-5.10.0 で入力した例を以下に示しておきましょう。

```
(%i8) plot_options;
(%o8) [[x, - 1.75555970201398E+305, 1.75555970201398E+305],
[y, - 1.75555970201398E+305, 1.75555970201398E+305], [t, - 3, 3],
[grid, 30, 30], [transform_xy, false], [run_viewer, true],
[plot_format, gnuplot_pipes], [gnuplot_term, default],
[gnuplot_out_file, false], [nticks, 10], [adapt_depth, 10],
[gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]], [gnuplot_curve_styles,
[with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,
with lines 6, with lines 7]], [gnuplot_default_term_command,
set term x11 font "Helvetica,16"], [gnuplot_dumb_term_command,
set term dumb 79 22], [gnuplot_ps_term_command,
set size 1.5, 1.5;set term postscript eps enhanced color solid 24],
[gnuplot_pipes_term, x11], [plot_realpart, false]]
```

ここで示す様に大域変数 `plot_options` は二成分リストから構成されたリストです。この大域変数を構成する二成分リストは `[plot_format,gnuplot]` の様に先頭が `plot_options` の項目で、その後に項目の値が続きます。例えば、`[plot_format,gnuplot]` は外部アプリケーションが `gnuplot` であり、生成するファイルも `gnuplot` のデータファイルになる事を示しています。

この例では、`plot_options` と入力して全ての設定を表示しています。これでは流石に見難いので、`get_plot_option` 関数を用いると、大域変数 `plot_options` の必要な設定値だけを調べる事が出来ます。ここで、`get_plot_option` 関数は1つの項目だけを引数とし、項目とそれに対する値の二成分リストを返す関数です。

ここで大域変数 `plot_options` の各項目に対応する値を変更したい場合、代入の演算子 `:` を使って、このリストを一々全て記述する事はあまり頭の良い方法とは言えません。指定した項目を変更する目的の為、`set_plot_option` 関数が用意されています。この `set_plot_option` 関数の引数として、項目と

その値で構成されるリストを引渡します。但し、set_plot_option 関数に指定可能な項目は一件に限られます。これらの関数の構文を以下に纏めておきましょう。

大域変数 plot_options に関連する関数

```
set_plot_options( [ <項目>, <値> ] )
get_plot_option( <項目> )
```

では、これらの関数の実例を示しておきましょう。この実例では、plot_format の値を get_plot_option 関数で取出し、この項目の値を set_plot_option 関数で openmath に変更させるというものです。

```
(%i9) get_plot_option(plot_format);
(%o9) [plot_format, gnuplot_pipes]
(%i10) set_plot_option([plot_format,openmath]);
(%o10) [[x, - 1.75555970201398E+305, 1.75555970201398E+305],
[y, - 1.75555970201398E+305, 1.75555970201398E+305], [t, - 3, 3],
[grid, 30, 30], [transform_xy, false], [run_viewer, true],
[plot_format, openmath], [gnuplot_term, default], [gnuplot_out_file, false],
[nticks, 10], [adapt_depth, 10], [gnuplot_pm3d, false], [gnuplot_preamble, ],
[gnuplot_curve_titles, [default]], [gnuplot_curve_styles,
[with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,
with lines 6, with lines 7]], [gnuplot_default_term_command,
set term x11 font "Helvetica,16"], [gnuplot_dumb_term_command,
set term dumb 79 22], [gnuplot_ps_term_command,
set size 1.5, 1.5;set term postscript eps enhanced color solid 24],
[gnuplot_pipes_term, x11], [plot_realpart, false]]
(%i11) get_plot_option(plot_format);
(%o11) [plot_format, openmath]
```

この例等で示した様に、大域変数 plot_options には色々な項目があります。ここでは、大域変数 plot_options の各項目について解説しましょう。そこで、外部アプリケーションに依存しない表示設定に関する項目を以下に示しましょう。

大域変数 `plot_options` の一般的な項目

項目	初期値	概要
<code>plot_format</code>	<code>[plot_format, gnuplot]</code>	グラフ表示アプリケーションを設定
<code>run_viewer</code>	<code>[run_viewer, true]</code>	<code>true</code> の場合に <code>plot_format</code> で指定したアプリケーションを起動
<code>colour_z</code>	<code>[colour_z, false]</code>	カラーの PS ファイルを出力するかどうかを指定するフラグ
<code>view_direction</code>	<code>[view_direction, 1, 1, 1]</code>	<code>plot_format</code> が <code>ps</code> の場合に三次元グラフの視点を指定
<code>transform_xy</code>	<code>[transform_xy, false]</code>	三次元グラフ表示で座標変換を行うかどうかを指定するフラグ
<code>grid</code>	<code>[grid 30, 30]</code>	三次元グラフの解像度を指定
<code>nticks</code>	<code>[nticks, 10]</code>	二次元グラフの解像度を指定
<code>logx</code>	<code>[logx, false]</code>	二次元グラフでの X 座標の対数目盛フラグ
<code>logy</code>	<code>[logy, false]</code>	二次元グラフでの Y 座標の対数目盛フラグ
<code>plot_realpart</code>	<code>[plot_realpart, false]</code>	複素函数実部の表示フラグ
<code>adapt_depth</code>	<code>[adapt_depth, 10]</code>	
<code>x</code>	<code>[x,-a,a]</code>	a はシステムによって異なる浮動小数点. 表示可能な領域の目安
<code>y</code>	<code>[y,-a,a]</code>	a はシステムによって異なる浮動小数点. 表示可能な領域の目安
<code>t</code>	<code>[t,-3,3]</code>	助変数表示に於ける助変数の定義域

Maxima は現在,gnuplot をデフォルトの描画用のアプリケーションとしています. その為,gnuplot を制御する項目が沢山存在します.

gnuplot に関連する plot_options の項目

項目	初期値	概要
gnuplot_pm3d	[gnuplot_pm3d, true]	gnuplot の PM3D(曲面に面を張るかかどうか) のオプション
gnuplot_curve_titles	[gnuplot_curve_titles,[default]]	描画する曲線の表題を設定.
gnuplot_curve_styles	[gnuplot_curve_styles,[with lines 3, with lines 1, with lines 2, with lines 5, with lines 4,with lines 6, with lines 7]]	曲線の様式を指定
gnuplot_out_file	[gnuplot_out_file, false]	gnuplot の画像出力ファイルを指定.
gnuplot_term	[gnuplot_term, default]	gnuplot の term を設定し, 対応するデータを出力する.
gnuplot_default_term_command	[gnuplot_default_term_command, set term x11 font "Helvetica,16"]	gnuplot で実行する term に関連する命令文を記述する
gnuplot_dumb_term_command	[gnuplot_dumb_term_command, set term dumb 79 22]	term を dumb とした場合の設定
gnuplot_ps_term_command	[gnuplot_ps_term_command, set size 1.5, 1.5; set term postscript eps enhanced color solid 24]	PS ファイルに追加する命令
gnuplot_pipes_term	x11	gnuplot の端末を指定
gnuplot_preamble	[gnuplot_preamble,]	gnuplot に引渡す諸設定を文字列で指定

この plot_options の各成分の値は set_plot_options 関数で設定する事も可能ですが, plot2d や plot3d 関数のオプションとして引渡す事も可能です. 又, Maxima-5.12.0 から gnuplot_default_term_command にフォントを Helvetica とし, サイズを 16 ポイントとする命令が初期値として入っています. 更に, plot_options では, 一般的なグラフ表示の為の設定の他に, gnuplot 専用の設定もあります. その場合には, gnuplot_xxx の様に頭に gnuplot が付くので分かり易いかと思います.

それでは各成分の意味について解説しましょう.

plot_format

plot_format で Maxima がグラフ表示で用いる外部アプリケーションを指定します. 指定可能な値は gnuplot_pipes, gnuplot, openmath, geomview, ps と zic です.

Maxima-5.12.0 から plot_format の初期値として gnuplot が指定されています. 描画に用いる外部アプリケーションは基本的にここでの値と同じ名前のアプリケーションになりますが, ps を指定した場合だけは ghostview がグラフ表示用の外部アプリケーションとして設定されます.

ここで gnuplot, openmath と geomview については今迄色々解説しているので問題は無いと思います. mgnuplot は Tcl/Tk で記述したフロントエンドを gnuplot に被せたものです. こちらは三次元空間内の曲面をスライダーを使って回転させる事が出来ませんが, 基本的に gnuplot を操作する plot_option の項目は反映されません. 最後の zic は Izic を外部アプリケーションとするものです. この Izic 自体は Tcl/Tk でフロントエンドを記述した古いアプリケーションの為, 実際に使われる事は無いでしょう.

Maxima には openmath も同梱されているので, 指定すれば使えますが, ps(ghostview), geomview, mgnuplot

に Izic になると、各々のアプリケーションが利用可能な環境でなければ使えません。

ここで Maxima の `plot2d` 関数と `plot3d` 関数は、`plot_format` で指定したアプリケーション向けのデータファイルを UNIX 環境の場合は、利用者のホームディレクトリに生成します。それから `run_viewer` の値が `true` であれば、そのファイルを指定した外部アプリケーションに引渡し、この値が `false` の場合、外部アプリケーションの立上げを行いません。

猶、ここで生成されるデータファイルの名前は `maxout.gnuplot` の様に `maxout.` の後に `plot_format` の値が付いたファイル名になります。又、続けて描画を行う場合、データファイルをそのまま上書きします。

このファイルは各アプリケーションから開く事も可能です。初期値の `gnuplot` の場合、データの他に描画命令やオプションの諸設定も記述されたファイルとなっているので、単純に `gnuplot` から `load 'maxout.gnuplot'` で読み込むと、グラフの表示を行います。この時、マウスが ON になっていればマウスで直接三次元画像を把持して、回転や拡大が行えます。もしも、マウスが on でなければ、`set mouse` で on に出来ます。

`mgnuplot` の場合は、`gnuplot` 向けの曲線、或いは曲面のデータのみが含まれたファイルになります。その為、`plot2d` 関数の場合は `plot 'maxout.mgnuplot'`、`plot3d` 関数の場合、`splot 'maxout.mgnuplot'` を実行すれば描画を行います。

colour_z

`colour_z` は、カラーの PostScript ファイルを出力するかどうかを指定する項目です。`true` の場合にカラーの PostScript ファイル出力を行い、`false` の場合は白黒の PostScript ファイルを出力します。この `colour_z` は `plot_format` として `ps` が指定された場合にのみ有効です。

view_direction

`view_direction` は視点の位置を指定します。この指定は `plot_format` が `ps` の場合の時だけ有効です。

transform_xy と make_transform 関数

`transform_xy` は通常のデカルト座標系から別の座標への変換を指定します。ここで設定する値は、通常のデカルト座標系を用いる場合に初期値の `false`、極座標系を用いる場合には `polar_to_xy` を設定します。この `transform_xy` に利用者定義の座標変換を利用する事も可能です。この場合は、`make_tranform` 関数を用いて指定しなければなりません。

以下に `make_transform` 関数の構文を示しておきます。

make_transform 関数の構文

```
make_transform(<変数リスト>, <fx>, <fy>, <fz>)
```

ここで f_x, f_y, f_z は各々デカルト座標系の X, Y, Z 成分となる `<変数リスト>` に含まれる変数の関数です。例えば、通常の円筒座標であれば、`make_transform([r, th, z], r*cos(th), r*sin(th), z)` の様にします。実際に利用する場合は `make_transform` 関数を用いた利用者定義の関数を生成し、それを `transform_xy` に設定するか、`make_transform` 関数をそのまま設定するか何れかが使えます。但

し、利用者定義の変換函数を用いる場合には `polar_to_xy` の様に函数名だけを指定する事は出来ず、変数も一緒に書かなければなりません。

これだけでは分かり難いので以下に実例を示しておきます。

```
(%i6) neko(r,th,z):=make_transform([r,th,z],
r*cos(%pi*th/180),r*sin(%pi*th/180),z);

(%o6) neko(r, th, z) := make_transform([r, th, z], r cos(-----),
                                     180
                                     %pi th
                                     r sin(-----), z)
                                     180

(%i7) plot3d(r*th^2, [r,1,2], [th,0,360], [gnuplot_pm3d,true],
[transform_xy,neko(r,th,z)]);
```

この例では、`plot3d` 函数のオプションとして `[transform_xy,neko(r,th,z)]` を引渡していますが、このオプションを `[transform_,make_transform([r,th,z], r*cos(%pi*th/180),r*sin(%pi*th/180),z)]` としても同じ結果になります。ここで定義した函数は、通常の `polar_to_xy` では角度がラジアンになるのに対し、こちらは度を用いる様にしている点です。

ここで注意点として、`[transform_xy,neko]` の様に利用者定義の変換函数に対し、その函数名だけを記述する事は出来ません。必ず、`[transform_xy,neko(r,th,z)]` の様に変数も含めて記述しなければなりません。但し、`polar_to_xy` に関しては、`[transform_xy,polar_to_xy]` と変数を省略して表記出来ます。これは、`polar_to_xy` の実体が `plot.lisp` 内部で定義された LISP の函数の為だからです。

では、上記の実行結果を以下に示しておきましょう。

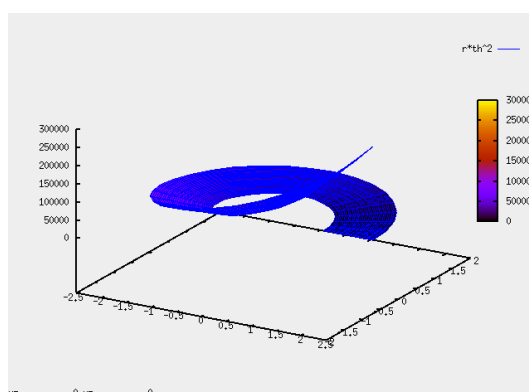


図 5.7: `transform_xy` と `make_transform` の組合せ

nticks

`nticks` は二次元グラフの解像度になります。初期値は 10 です。描いた曲線が粗い場合に、この

nticks の値を大きくすると良いでしょう。特に媒介変数式の表示では初期値を予め大きく指定しなければ綺麗な曲線は描けないでしょう。

grid

grid は三次元グラフの解像度で、初期値は [30,30] です。grid の値は X 方向と Y 方向の解像度の対で記述し、[grid,50] の様に記述する事は出来ません。必ず、[grid,50,50] の様に記述します。描いた曲線が粗い場合、この grid に大き目の整数値を成分に持つリストを設定します。

x と y

x と y に設定される値は Maxima の土台にある Common Lisp 処理系の (/ most-positive-double-float 1024) の処理結果となります。この値は Common Lisp で扱える数値の限界を示すものの為、この値を越える数値は Maxima では扱えないので、当然、グラフ表示も出来ません。猶、グラフ表示可能な数値の上限は Maxima で利用可能なグラフ表示アプリケーション毎に異なる為、ここでの設定以下であっても表示が出来ない場合もあります。例えば、gnuplot では、 \sqrt{x} のグラフをこの x の値以下であれば表示が可能ですが、openmath では 10^{203} よりも大きな数値になると表示が出来なくなります。

t

これは函数の媒介変数式で利用する変数 t の定義域を定めます。Maxima では媒介変数式の変数は t に固定されています。この定義域を定めておけば、グラフ表示で媒介変数の定義域を省略する事が可能になります。

xlog と ylog

xlog と ylog は各々 X 軸と Y 軸の目盛を対数目盛にします。この設定は二次元グラフの場合に有効で、三次元グラフの場合には無効です。

gnuplot_pm3d

gnuplot_pm3d は gnuplot の pm3d を有効にします。初期値は false の為、gnuplot で描く曲面はワイヤフレーム表示となっていますが、true を設定した場合、maxout.gnuplot に set pm3d を書込みます。又、必要に応じて、b,s,t といった文字で構成された文字列も与えられます。これらの文字にはそれ相当の意味がありますが、この詳細については gnuplot の pm3d の 5.12.5 節で改めて解説します。

実例を以下に示しておきましょう。

pm3d の例

```
plot3d(sin(x*y), [x, -3, 3], [y, -3, 3], [gnuplot_pm3d, true], [grid, 50, 40])
```

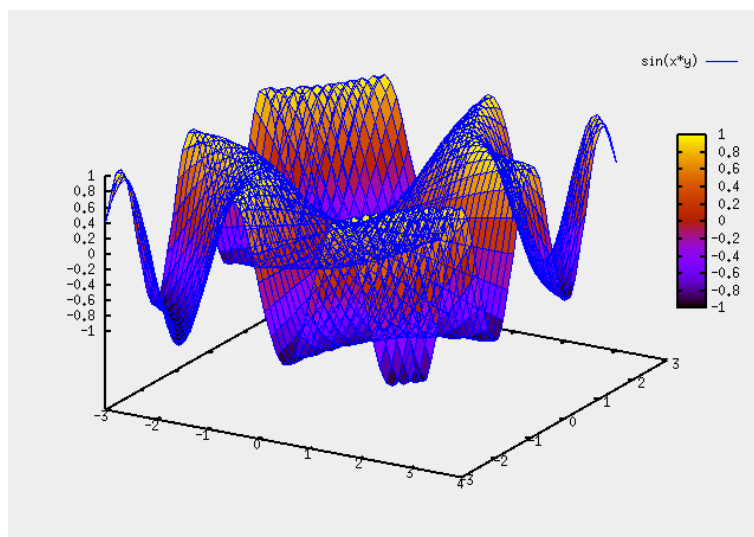


図 5.8: pm3d の例

しかし,gnuplot ではより高度な表示が可能です.set pm3d にしても, `set pm3d at bs` とする事で,最初に底面への曲面の投影を行い,それから曲面を描くといった指定が行えます.このような細かい設定は,gnuplot_preamble を用いればより詳細に行えますが,gnuplot_pm3d に引渡す事も可能です.例えば,底面への投影を意味する文字 b と曲面の描画を意味する s を組合せた文字の列 bs を Maxima の文字列にせず,[gnuplot_pm3d,bs] の様にして引渡せます.以下に例を示しておきましょう.

gnuplot_pm3d の設定例

```
plot3d(sin(x*y), [x, -3, 3], [y, -3, 3],
[gnuplot_pm3d, bs], [gnuplot_preamble, "unset surf"], [grid, 50, 40])
```

この例では,gnuplot の pm3d のオプションとして bs の他に,gnuplot_preamble を用いて,gnuplot に unset surf を実行させます.この設定は,曲面上の網目を消す効果があります.この式を実行して得られた図を図 5.9 に示しておきます.

猶, 図 5.9 は次の様に gnuplot_preamble をだけを用いても描けます.

gnuplot_preamble を使った例

```
plot3d(sin(x*y), [x, -3, 3], [y, -3, 3],
[gnuplot_preamble, "set pm3d at bs;unset surf"], [grid, 50, 40])
```

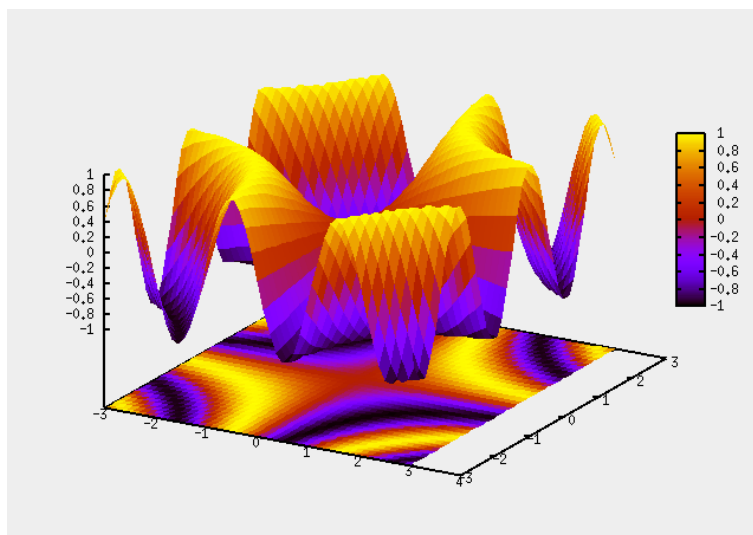


図 5.9: `gnuplot_preamble` を使った例

`gnuplot_term`

`gnuplot_term` は `gnuplot` の出力端末の設定を行います。これは指定した端末に対応したデータを出力するもので、`gnuplot` 内部では `set term` による端末の設定が行われます。ここで設定可能な端末の型には、`x` 端末に対応する X11, MacOS の `aqua` と `gnuplot` の画像を表示する際に必要な端末の情報や、`postscript` や `gif` といった画像データ等と非常に多い為、ここでは一々説明しません。この端末の詳細は参考文献 [14] か、或いは `gnuplot` 上で `? term` と入力すれば、オンラインマニュアルと `term` に設定可能な値の一覧が表示されるので、そちらを参照して下さい。

猶、`gnuplot_term` に `'default'` 以外の値を設定した場合に Maxima で描画を行うと、`gnuplot` に引渡すデータファイル (`maxout.gnuplot`, 或いは `maxout.gnuplot_pipes`) に加えて、別途、グラフ画像のデータファイルが出力されます。この場合、`maxplot.<端末名>` といった名前のファイルになります。例えば、`terminal` として `tgif` を設定した場合で解説しましょう。

この時、グラフ出力時に、`set term tgif` と `set out '/home/yokota/maxplot.tgif/'` (ここで `/home/yokota/` がホームディレクトリです) が `gnuplot` に設定されます。そして、同時にグラフ出力ファイルとして、`maxplot.tgif` が生成され、画像データが書込まれます。猶、`maxplot.tgif` を `maxplot.obj` に変更して `tgif` に読込ませた結果が図 5.10 に示すものです。

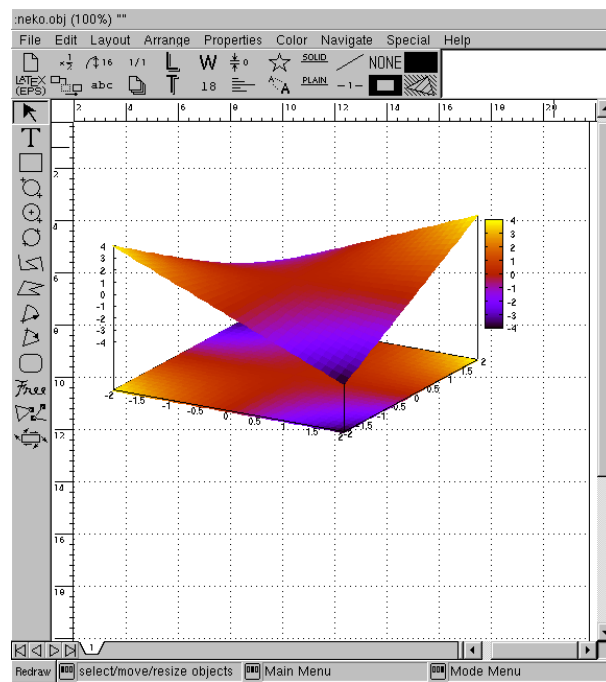


図 5.10: term を tgif にした結果 (maxplot.tgif) を tgif で表示

gnuplot_dumb_term_command

gnuplot_dumb_term_command は gnuplot の term として dumb を選択した時の設定が行えます。以下に実例を示しておきます。

```
(%i86) plot2d([sin(x),cos(x)], [x,0,10], [gnuplot_term,dumb],
[gnuplot_dumb_term_command,["set term dumb 70 20"]]);
```

```

1 ***-----$$$$$$-----+-----*****-----$$$$$-----++
+ *** $$  $$          +          *** ** $$$sin(x) $$$$$$ +
0.8 ++ *$$  $          **      *$$ cos(x)$*****++
0.6 ++ $$*  $          **      $$*  $$  ++
| $$ **  $          *      $$ **  $$ |
0.4 ++$$  **  $          *      $$ **  $  ++
0.2 +$$  *  $$  $          **      $  **  $  ++
|$  *  $$  $          **      $  *  $$ |
0 $+  **  $          **      $$  *  $$ ++
|  **  $          *      $$  **  $$ |
-0.2 ++  *  $$  *      $$  **  **  $++
-0.4 ++  *  $$  **      $          **  $+$
|  **  $$ **      $          *      $$
-0.6 ++  **  $$*  $$  $          *      ++
-0.8 ++  **  *$$  $$  $          **  ++
+  +  ***  ***+$$  $$  +          +  **  **
-1 ++-----+-----*****+-----$$$$$-----+-----*****+
0 2 4 6 8 10
```

Output file "/home/yokota/maxplot.dumb".
(%o86)

この例では,dumb 端末を 70 列 20 行として表示しています。猶,term を dumb にしなければ, この設定は無効です。

gnuplot_curve_titles

gnuplot_curve_titles には描く曲線や曲面の表題をリストで追加します。この時, 構文は gnuplot の plot 命令や splot 命令に引渡す表題の設定に準じます。例えば, 二曲線に A1,A2 と設定したい場合には, gnuplot_curve_titles,["title 'A1'", "title 'A2'"] とします。

gnuplot_curve_styles

gnuplot_curve_styles に描画する線分の書式を設定します。これは gnuplot の with 命令の文を文字列として引渡す為に用います。例えば,[gnuplot_curve_styles,["with lines 7", "with lines 2"]] の様に行います。因に,maxout.gnuplot 内部では,gnuplot の描画命令 plot のオプションとして引渡されます。その為,, 後述の gnuplot_preamble で曲線のスタイルを幾ら設定しても, こちらの設定が優先されるので注意が必要です。

gnuplot_pipes_term

gnuplot_pipes_term で gnuplot が用いる端末 (terminal に相当) を指定します。この変数は,Maxima-

5.12.0 より導入されています。猶、MS-Windows 環境でも初期値として x11 が指定されています。通常、この値を変更する必要は無いでしょう。

gnuplot_preamble

gnuplot_preamble は gnuplot の制御文を、gnuplot に直接引渡す為の仕組みで、gnuplot の制御文の間には ; を入れた Maxima の文字列を指定します。この preamble の設定は plot_option で gnuplot に関連する設定と、gnuplot の数値データの間にかかれます。

preamble の内容がどのように書込まれるかを plot_format を gnuplot に設定した状態で以下の描画をさせて、maxout.gnuplot の内容を調べてみましょう。

```
(%i4) nekoneko:"set title 'mike';set xlabel 'X';\
set ylabel 'Y';set zlabel 'height';";
(%o4) set title 'mike';set xlabel 'X';set ylabel 'Y';set zlabel 'height';
(%i5) plot3d(sin(x*y),[x,0,10],[y,0,10],
[gnuplot_pm3d,true],[gnuplot_preamble,nekoneko]);
```

では、以下に maxout.gnuplot の頭の部分を示しておきます。

gnuplot_preamble を反映した maxout.gnuplot のヘッダ部分

```
set pm3d

set title 'mike';set xlabel 'X';set ylabel 'Y';set zlabel 'height';
plot '-' title 'sin(x*y)' with lines 3
0.0      0.0      0.0
0.3333333333333333      0.0      0.0
0.6666666666666666      0.0      0.0
1.       0.0      0.0
```

この様に gnuplot の plot 命令の直前に gnuplot_preamble の内容が書込まれている事が分りますね。この事を利用すれば gnuplot の諸設定を行う事が可能になるのです。

5.9.1 gnuplot に関連した関数

gnuplot に関連した関数

```
gnuplot_start()
gnuplot_restart( )
gnuplot_close()
gnuplot_pipes
gnuplot_reset()
gnuplot_replot(< オプション >)
```


`gnuplot_start` 函数は引数を必要としない函数です。MS-Windows でこの函数を実行すると `gnuplot` が立ち上がります。

`gnuplot_restart` は内部的に `gnuplot_close` を実行し、それから `gnuplot_start` を実行するだけの函数です。その為、グラフを表示している場合には、グラフのウィンドウが消滅し、MS-Windows であれば、`gnuplot` のウィンドウが現われます。

`gnuplot_reset` 函数は引数を必要としない函数です。この函数は `gnuplot` で設定した諸設定を初期化してしまう函数です。この函数をグラフ表示中に実行すると、それ迄のマウス操作による変更は全て無効になり、最初の設定のグラフが表示れます。

`gnuplot_replot` 函数はその名前のとおり、`gnuplot` で再描画を行う函数です。 `gnuplot_reset();` を最初に実行し、 `gnuplot_replot();` を実行すれば、それまでのマウスによる操作は無効になり、最初に描画したグラフが再描画されます。

5.10 その他の描画函数

Maxima の描画函数には上述の `plot2d` 函数や `plot3d` 函数の他に、幾つかの描画函数があります。とは言え、どちらかと言えば補助的な函数が多く、それも `openmath` 専用や PostScript への出力のみといった機能や出力が限定されたものが殆どです。

最初に、比較的汎用性がある `openplot_curves` 函数の解説をしましょう。

5.10.1 `openplot_curves`

`openplot_plot` 函数は大域変数 `plot_options` の `plot_format` とは無関係に `openmath` を用いて与えられた点列の描画を行う函数です。とは言え、オプションを変更する事で、実数値一変数函数の描画も可能です。

————— `openplot_curves` —————

```
openplot_curves ([[ <点列リスト1>, ..., <点列リストn> ]])
openplot_curves ([[ オプション1 ], <点列リスト1>, ..., ..., <オプションn>, <点列リストn> ])
openplot_curves (<点列リスト>)
openplot_curves (<オプション>, <点列リスト>)
openplot_curves (<xfun を含むオプション>)
```

ここでの点列リストの書式は、 $[x_1, y_1, \dots, x_n, y_n]$ が $[[x_1, y_1], \dots, [x_n, y_n]]$ の二種類に限定されます。`openplot_curves` のオプションは `plot_options` とは無関係で、`openmath` のメニューからも設定可能な事項になります。又、点列リストの直前にあるオプションがその点列の表示で用いられます。`openplot_curves` の設定項目を以下に示しておきます。

————— `openplot_curves` の設定項目 —————

項目	初期値	概要
<code>xfun</code>	なし	指定した変数 <code>x</code> を持つ式を指定
<code>color</code>	<code>blue</code> から開始	曲線の色を指定
<code>plotpoints</code>	0	直線や曲線上に点を打つ為のフラグ。初期値は 0
<code>linecolors</code>	<code>blue</code>	直線/曲線の色を指定。
<code>pointsize</code>	0	点の大きさを指定
<code>nolines</code>	0	点列を繋ぐ線分表示の為のフラグ
<code>bargraph</code>	0	棒グラフ表示への切替の為のフラグ
<code>xaxislabel</code>	無指定	X 軸のラベルを指定
<code>yaxislabel</code>	無指定	Y 軸のラベルを指定

`openplot_curves` のオプションには独特の書式があります。これは `plot_options` に似たもので、`{ 項目 値 }` の書式の文字列を空行で区切って、全体を引用符で括ったリストになります。例えば、各点を表示して、その点の大きさを 6 にする場合には、`["{plotpoints 1} {pointsize 6}"]` をオプションとして与えます。

次の例では、折線毎に点の大きさと、折線と点の色を変更し、各軸にラベルを設定しています。

```
openplot_curves([[{"plotpoints 1} {pointsize 6} {color red}"],
[1,2,3,4,5,1],
["{pointsize 10} {xlabel time} {ylabel neko} {color black}"],
[10,9,9,2,4,2]])
```

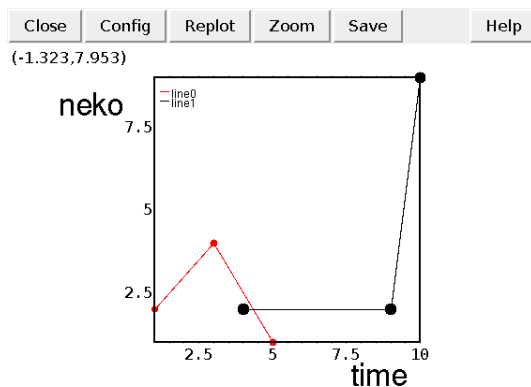


図 5.11: openplot_curves のオプション例

又,xfun を指定する事で,Maxima の式をグラフに追加する事も可能です。先程の例に正弦函数を追加した例を以下に示しておきましょう。

```
openplot_curves([[{"plotpoints 1} {pointsize 6} {color red}"],
[1,2,3,4,5,1],
["{pointsize 10} {xlabel time} {ylabel neko} {color black}"],
[10,9,9,2,4,2],
["{xfun sin(x)} {color green} {plotpoints 1} {pointsize 1}"]]])
```

ここで注意する事は、描画する式の変数は x に限定される事です。又、函数の描画だけを行いたければ、点列リストを持たないオプションのみの引数を与えます。但し、openplot_curves 函数で式の定義域を与える事は出来ない様です。

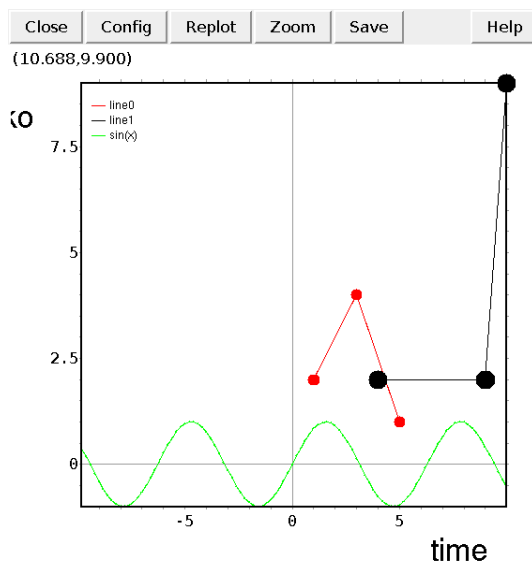


図 5.12: openplot_curves のオプション例 (その 2)

5.10.2 contour_plot

contour_plot

```
contour_plot(<式>)
contour_plot(<式>, <オプション>)
```

contour_plot は与えられた 2 変数実数値関数の等高線を描画する関数です。オプションは plot_format と gnuplot_preamble に限定されます。従って、グラフに様々な細工を行いたければ、plot_format は gnuplot でなければなりません。

5.10.3 Postscript に関連する関数

Postscript に関連する関数

```
plot2d_ps(<式>, <定義域>)
viewps( <PostScript ファイル名> )
viewps()
psdraw_curve (<点列リスト>)
psdraw_points (<点列リスト>)
pscom( <PostScript 命令文> )
closeps()
```

plot2d_ps 関数は plot2d 関数に似た構文を持ちますが、デカルト座標系での関数の表示のみが可能です。具体的には関数と定義域の二つの引数のみを取ります。この関数はカレントディレクトリ上に maxout.ps ファイルを生成し、内部で viewps 関数を呼出して、maxout.ps ファイルの表示を行います。

viewps 関数は単純に ghostview に PostScript ファイルを引渡す関数です。ファイル名を指定しない場合には、ホームディレクトリ上の maxout.ps ファイルの表示を行います。猶、ghostview 命令を持たない環境では、この関数は使えません。

psdraw_curve は点列リストのグラフを PostScript 形式で出力します。ここで点列リストの書式は $[x_1, y_1, x_2, y_2, \dots, x_n, y_n]$ の書式と $[[x_1, y_1], [x_2, y_2], \dots, [x_n, y_n]]$ の書式の何れかになります。psdraw_curve 関数を実行する事でホームディレクトリ上の maxout.ps へのストリームを開きます。このストリームは `closeps()` で閉じられます。

psdraw_points 関数は点列リストのグラフを PostScript 形式のファイルで出力します。この関数は psdraw_curve 関数に似ていますが、この関数の引数となる点列リストの書式は $[[x_1, y_1], [x_2, y_2], \dots, [x_n, y_n]]$ のみに限定されます。psdraw_curve 関数と同様に psdraw_points 関数を実行するとホームディレクトリ上のファイル maxout.ps へのストリームが開かれます。又、`closeps()` を実行すると、ストリームが閉じられて、maxout.ps ファイルが生成されます。

pscom 関数は与えられた PostScript の命令文をホームディレクトリ上の maxout.ps へのストリームに出力する関数です。

psdraw_curve 関数、psdraw_points 関数と pscom 関数はどちらから言えば原始的な関数で、これらは全てホームディレクトリ上の maxout.ps へのストリームを開き、そのストリームに対して出力を行う関数です。そして、closeps() 関数を実行する事で、ストリームが閉じられて、maxout.ps ファイルが生成されます。

因に、ストリームが開いているかどうかは、大域変数 pstream で調べられます。ストリームが開いていない場合に、pstream を入力すると結果として false が返り、開いている場合には、その情報を返します。

実際に、pstream の働きを見ておきましょう。

```
(%i32) psdraw_curve([[1,2],[3,3],[5,1]]);
(%o32)
false
(%i33) pstream;
(%o33) #<OUTPUT BUFFERED FILE-STREAM CHARACTER /home/yokota/maxout.ps>
(%i34) closeps();
(%o34)
true
(%i35) pstream;
(%o35)
false
```

この例では、最初に psdraw_curve 関数を実行する事で、ホームディレクトリ (ここでは /home/yokota) 上のファイル maxout.ps へのストリームを開きます。この事は、pstream を見ると、maxout.ps へのストリームが開いている事が分ります。それから `closeps()` を実行した後に、pstream の値を見ると false になっているので、この事からストリームが閉じている事が分ります。

5.11 plot_option 以外の描画に関連する大域変数

Maxima の描画関数で最も重要な大域変数としては `plot_options` がありました。Maxima にはこの大域変数の他にも多くの大域変数があります。但し、これらの変数はどちらかと言えば、Maxima の描画関数で内部的に用いる事を想定したものが多く、用途も特殊なものが多いのが実情です。その為、ここでは一覧表に纏めて解説しておきます。

因に、以下で解説する変数は Maxima のソースファイル `plot.lisp` の中を見れば色々ある事が分ります。

plot_option 以外の描画に関連する大域変数

変数名	初期値	概要
<code>in_netmath</code>	<code>false</code>	<code>plot_format</code> の値が <code>openmath</code> の場合に限り、 <code>max-out.openmath</code> の内容を画面に表示
<code>show_openplot</code>	<code>false</code>	<code>plot_format</code> が <code>openmath</code> の場合は <code>true</code> 、それ以外は <code>false</code>
<code>viewps_command</code>	<code>(ghostview " a")</code>	<code>plot_format</code> を <code>ps</code> にした場合の、PS ファイルの表示命令を設定
<code>ps_scale</code>	<code>[72,72]</code>	PS ファイルの解像度
<code>ps_translate</code>	<code>[0,0]</code>	原点の移動に利用
<code>window_size</code>	<code>[612.0,792]</code>	PS ファイルのウィンドウの大きさを指定

5.12 gnuplot による描画について

この章は Maxima を使う上で必要な gnuplot の事項のみを記述します。猶、gnuplot 単体の解説ではなく、Maxima から見た gnuplot の解説になります。この際に、maxout.gnuplot ファイルから gnuplot の事を開始します。

gnuplot 全般の使用方法や例題を知りたければ、「使いこなす GNUOPlot」 [14] 等の gnuplot の解説本や gnuplot に付属するマニュアル等の文献を参照して下さい。

5.12.1 maxout.gnuplot の内容について

maxout.gnuplot ファイルは plot_format を gnuplot に指定した場合に生成されるファイルで、UNIX 環境であればホームディレクトリ上、MS-Windows 環境であれば Documents and Settings 以下の個人フォルダに生成されます。

このファイルの先頭には数値データを描画する為の gnuplot の命令文や様々な設定文が置かれ、ファイルの末尾に描画される式の数値データが記述されています。その為、別途 gnuplot を起動させて `load "maxout.gnuplot"` と gnuplot の入力行に入力すると、gnuplot 単体で Maxima から処理したのと同じグラフが表示出来ます。

この事は、Maxima 側で行ったグラフの諸設定がどの様に maxout.gnuplot に反映されているかを観察すれば、gnuplot がどの様に処理を行っているかを理解する事が容易になるとも言えるのです。そこで、手始めに plot2d 関数による maxout.gnuplot の構造を簡単に解説しておきましょう。

plot2d 関数による maxout.gnuplot

```

set log x      /* gnuplot_logx が true の場合に記述 */
set log y      /* gnuplot_logy が true の場合に記述 */

<gnuplot_preamble の内容>
plot '-' <gnuplot_curve_titles の内容> <gnuplot_curve_styles の内容>
<実データ>

```

ここで示す様に gnuplot_logx と gnuplot_logy の設定は plot 命令の前に記述されます。更に、plot 命令の直前の行に、gnuplot_preamble の内容が書込まれます。そして、plot 命令の直後の行に描画するデータが置かれるのです。ここで、`plot '-'` とありますが、ここでの `'-'` で次の行にある数値データを表示する事を gnuplot に指示しています。それから、曲線の表題や曲線の型の指定が並んでいます。

さて、今度は plot3d 関数による maxout.gnuplot の構造を示しておきましょう。plot3d 関数で生成した maxout.gnuplot も plot2d 関数と似通った書式になります。

plot3d 函数による maxout.gnuplot

```
set pm3d          /* gnuplot_pm3d が true の場合に記述 */

<gnuplot_preamble の内容>
splot '-' <gnuplot_curve_titles の内容> <gnuplot_curve_styles の内容>
<実データ>
```

今度は gnuplot_pm3d に対応する `set pm3d` がファイルの先頭に置かれていますね. その他は plot2d 函数で生成した maxout.gnuplot と似ていますが, 描画命令が plot ではなく splot 命令になっている事に注意しておいて下さい.

これらのファイルの内容を眺めるだけでも, plot 命令や splot 命令が gnuplot の描画命令である事が何となく分るかと思います. これらの描画命令の前に, set 命令から開始して, xlog, ylog や pm3d が続く命令文があります. これらは gnuplot で様々な設定を行う命令文で, 先ず, set 命令で様々な設定を行います. そして設定内容は show 命令で確認が行え, unset 命令で設定内容を無効にする事が出来ます.

さて, Maxima から単純に plot2d 函数や plot3d 函数を用いて maxout.gnuplot を生成しても, `set pm3d at bs` の様な gnuplot の命令文を埋込む事は出来ません. gnuplot から直接描画する場合は, この様な命令文を入れて replot 命令で再描画すれば良いのですが, Maxima から操作したければ, plot_format を gnuplot_pipes にした場合のみ有効になります. 又, MS-Windows 版では, gnuplot_pipes が選べない為, この身軽な操作は出来ません.

その為, より高度なグラフ表示を行う場合, 直接 gnuplot の命令文が書き込める gnuplot_preamble に色々と設定すると結構楽に処理が行えます. この事から, gnuplot を利用する場合, gnuplot_preamble の使いこなしが非常に重要である事が理解出来るかと思います.

それでは, 先程の maxout.gnuplot に表われた命令の意味を簡単に紹介する事にしましょう.

5.12.2 set 命令

先ず gnuplot で様々な設定を行う上で必要な set 命令について簡単に説明しておきましょう.

この set 命令は前述の様に gnuplot の様々な設定を行う命令です. set 命令で設定した内容は show 命令を使って確認する事が可能です. 更に, set 命令による設定を無効にする場合は unset 命令を用います. これらの命令は非常に重要な命令です. 以下に, set 命令, show 命令と unset 命令の基本的な構文を示しておきます.

set 命令 show 命令, unset 命令

```
set <項目>          <項目> を有効にする
set <項目> <設定内容> <項目> に <設定内容> を設定する
show <項目>         <項目> の内容を表示
unset <項目>        <項目> を無効にする
```


ここで示した様に `set` 命令には単純に項目を指定する場合と、その項目に対して値を指定する場合の二種類があります。この具体的な例は Maxima の諸設定と絡めて詳細を述べる事とします。

5.12.3 plot 命令による曲線の表示

先程の `maxout.gnuplot` の例で示した様に、曲線を描く場合には `plot` 命令を用います。この `plot` 命令は変数 x の式や 2 次元の数値データを描画する事が可能です。以下に `gnuplot` での `plot` 命令の構文を示しておきます。

plot 命令の基本的な構文

```
plot <表示範囲> <式> title <文字列> with <曲線の様式>
plot <式> axes <軸の設定> title <文字列> with <曲線の様式>
plot <式> title <文字列>
plot <式>
plot <データファイル名> title <文字列> with lang 曲線の様式
plot <データファイル名> title <文字列>
plot <データファイル名>
```

式の表示範囲は複数の式を表示する場合でも、一つの `plot` 関数に対し、一つだけが設定可能です。ここで表示範囲は `plot [-3:3] [-2:2] cos(x)` の様に定義域と値域の順番で MATLAB 風のベクトル表記で記述します。ここで `plot [-3:3]` の様に `[-3:3]` だけを設定した場合、`[-3:3]` が式の定義域となり、グラフの値域は式全体が収まる様に `gnuplot` 側で自動的に調整します。次に `gnuplot` の式には `gnuplot` で表記可能な数式、或いは表示すべきデータが含まれているデータファイル名や記号 `'-'` を設定します。

ここで `plot` 命令に与える数式は基本的に変数 x の `gnuplot` の式に限定されます。複数の式を同時に表示させる場合、式とその式に対する設定を一組として、それらをコマンドで区切ったものを与えます。例えば、正弦函数と余弦函数を同時に表示したい場合、最も簡単な表示の場合は `plot sin(x), cos(x)`、曲線の名前や曲線の様式を指定する場合、以下の様に入力します。

```
plot sin(x) title 'Sin' with lines 1, cos(x) title 'Cos' with points 5
```

データファイル名は引用符、或いは二重引用符で括られた文字列として与えます。猶、Maxima で `gnuplot` で描画させるファイル名やラベルの指定の為に文字列を `gnuplot` の命令文に組込む際に注意する事として、Maxima の文字列が二重引用符に限定される為、ファイル名やラベル等の `gnuplot` の命令文中の文字列は単引用符で括ったものを用いなければならない事です。例えば、曲線の表題を `test` にしたければ、Maxima の大域変数 `plot_options` の項目 `gnuplot_curve_titles` への設定を `[gnuplot_curve_titles, "title 'test'"]` で行います。即ち、二重引用符で括った場合、二重引用符内部の文字の列に二重引用符が出現すると、文字列がそこで一旦途切れてしまいますが、単引用符の場合は途切れずに `gnuplot` に引渡される為です。

あるファイルに `plot` 命令とデータ本体を記載する場合、`plot` 命令で指定するファイル名の箇所を `'-'` と記述すれば、`plot` 命令の次の行から開始するデータ列の読みを行い、ファイルの EOF を検出

した時点で描画を行います。この方法が maxout.gnuplot で採用されています。

gnuplot の式とデータファイルの内容を混在して表示する事も可能です。この場合、引数として、数式とデータファイルをコンマで区切ったものを第一引数として引き渡します。例えば、先程の正弦函数と余弦函数に加え、データファイル test1 のデータを表示させたい場合、`plot sin(x),cos(x),'test1'` の様に入力します。

曲線の名前は title の後に文字列を指定します。これは gnuplot の右上に表示される凡例 (key) で用いられる曲線の名前です。Maxima では大域変数 plot_options の項目 gnuplot_curve_titles の内容がここに記入されます。因にこの凡例を調整する場合には、`set key` を用いて行います。

ここで、Maxima から gnuplot を利用する場合、この gnuplot_curve_titles の内容が標準入力を示す記号 '-' の直後に置かれます。そこで、この事を利用した阿漕な使い方としては、別のグラフデータを指定したり、gnuplot の式を入れる事が挙げられます。この事を早速確認してみましょう。

```
plot2d(1/x, [x, 1.0e-8, 1.0e-5],
[gnuplot_curve_titles, "title 'Maxima', 1/x title 'GNUPlot' "]);
```

この例では、 $\frac{1}{x}$ のグラフを描きますが、一方は Maxima で描いたもので、もう一方は gnuplot の式として gnuplot に計算させて描画するものです。その結果を図 5.13 に示しておきます。

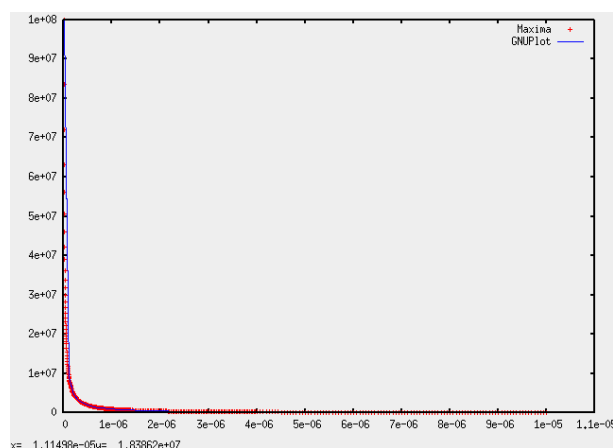


図 5.13: gnuplot_curve_titles の応用で gnuplot の式も描く

ここで、gnuplot の式は `[gnuplot_curve_titles, "title 'Maxima', 1/x title 'GNUPlot' "]` で与えている事に注目して下さい。この処理で生成された maxout.gnuplot のヘッダの部分に以下に示しておきますが、上手く埋め込まれている事が分ります。

maxout.gnuplot の様子

```
plot '-' title 'Maxima',1/x title 'GNUPlot' with lines 3
1.00000000E-8 100000000.
1.195117187500000100E-8 83673802.90897205
1.390234375000000400000000E-8 71930317.5049171
1.585351562500000600000000E-8 63077491.68411973
1.780468750000000500000000E-8 56164984.642386995
1.975585937500000700000000E-8 50617894.216510125
2.170703125000000600000000E-8 46068022.3141983
2.365820312500000500000000E-8 42268637.001568556
2.560937500000000600000000E-8 39048200.12202562
---- 以下略 ----
```

最後に、曲線の書式は with の後に記入します。Maxima では `gnuplot_curve_styles` に with から開始する文字列を設定しておけば、`maxout.gnuplot` にその文字列が記入されます。通常の実線で色を赤にする為には、`with lines 1` の様に設定します。

5.12.4 splot 命令による曲面の表示

`splot` 命令は数値データや変数 x, y の式で表現された曲面を描く命令です。基本的な構文は `plot` 命令と同様で、`plot` を単純に 3 次元に拡張した面を持ちます。

splot 命令の基本的な構文

```
splot <式> title <文字列> with langle 曲面の様式>
splot <式> title <文字列>
splot <式>
splot <データファイル名> title <文字列> with langle 曲面の様式>
splot <データファイル名> title <文字列>
splot <データファイル名>
```

この `splot` 命令でも `plot` 命令と同様に、曲面の表題 (title で指定)、そして曲面の様式 (実際はワイヤースケッチを構成する線分の様式に対応) の指定が行えます。ファイル名や文字列に関しては、`plot` 命令と違いはありません。又、`gnuplot` で扱える式は変数 x, y の式でなければなりません。

猶、曲面の様式には、`lines`, `point`, `linespoints`, `dots`, `impulses` があります。この曲面の様式は曲面上に表示される網目に対するもので、後述の `pm3d` に関連する項目に影響を及ぼしません。

`gnuplot` の 3.8 版より `pm3d` が標準になった事で、`pm3d` の様々な設定を色々行う必要が出てくるでしょう。次に、この `pm3d` の基本的な設定について述べる事とします。

5.12.5 pm3d

gnuplot では pm3d を有効にする事によって, 面を貼った曲面表示が行えます. この pm3d は従来は gnuplot のパッチとして配布されていましたが, 3.8 以降から標準で付属しています.

pm3d の設定

```
set pm3d
set pm3d at b
set pm3d at s
set pm3d at t
set pm3d map
unset pm3d
show pm3d
```

最初の `set pm3d` で pm3d を有効にします. 因に, Maxima の `[gnuplot_pm3d, true]` がこの設定に対応します. Maxima の大域変数 `plot_options` には, その他に pm3d に関する項目はありませんが, 他にも細かな指定が出来ます. 例えば, 面を張った曲面の表示だけではなく, 曲面と底面や上部への投影と一緒に表示したりする事が可能です. この投影の設定は pm3d の設定で, `at` の後の文字列で指定を行います.

次の `set pm3d at b` で曲面を底面に投影します. 後述の `map` に似ていますが, こちらは斜め上から曲面と投影の両方を眺める形になります. この指定だけでならば曲面はワイヤーステイク表示になります.

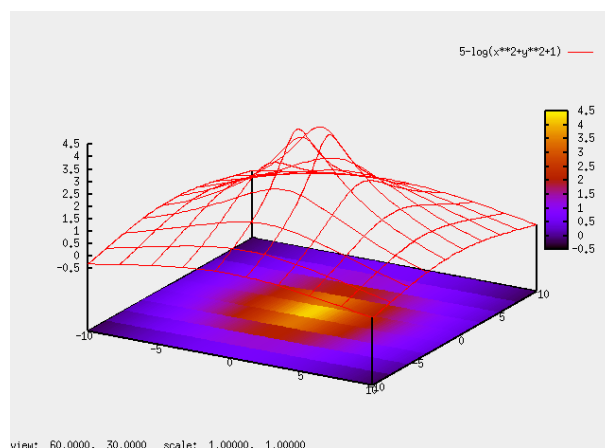


図 5.14: set pm3d at b の場合

これに対し, `set pm3d at s` を実行すると, 今度は曲面に面を張り, この面には連続的に等高線で変化する色彩が付けられた曲面が表示されます. この場合, 曲面の底面への射影は実行されません. 又, gnuplot で `hidden3d` を有効にしたり, `surface` を無効にしていなければ, 曲面とワイヤーステイクが同時に表示されています.

gnuplot では底面だけに射影が行える訳ではありません. `set pm3d at t` とする事で, 曲面の等高

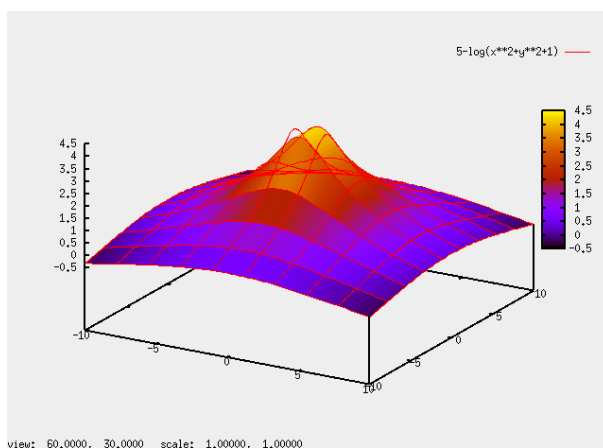


図 5.15: set pm3d at s の場合

線表示の射影を今度は上面に行います. この場合, 射影が上側に行われる事を除いて, b を指定した場合との違いはありません (勿論, 射影が上に出る違いはあります...).

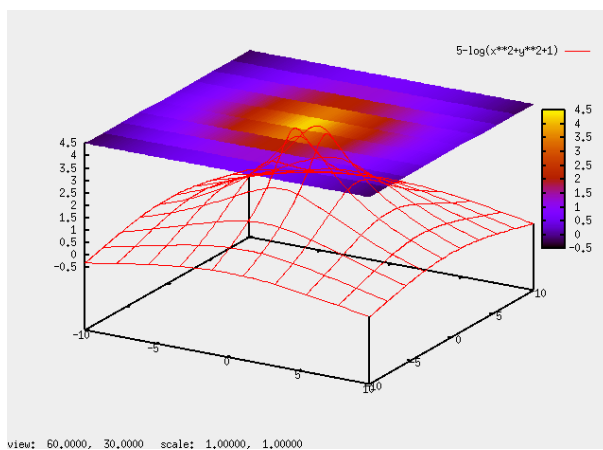


図 5.16: set pm3d at t の場合

この様に上や下側への射影を行うと, 射影には綺麗な等高線図が得られます. では, この等高線図だけが欲しい場合はどうすれば良いでしょうか? それは `set pm3d map` で得られます. 次の図 5.17 に示しておきましょう.

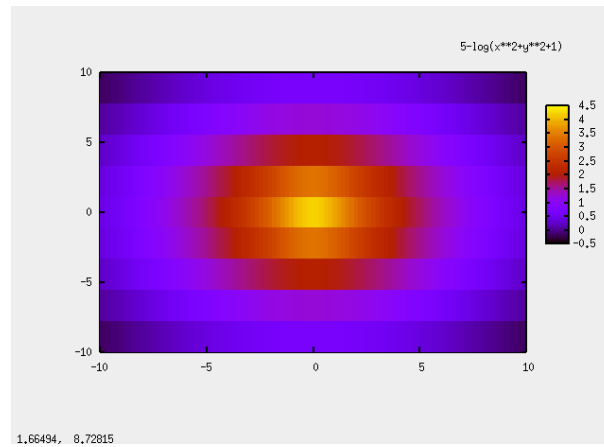


図 5.17: set pm3d map の場合

これらの b,s,t の指定を組合せる事も gnuplot では可能です。指定の方法は単純に、`set pm3d at bs` の様に at の後に並べるだけです。では gnuplot で `splot x*y` の描画を、`set pm3d at bs` を行った場合を図 5.18 に示し `set pm3d at sb` を行った場合の結果を図 5.19 に示しておきましょう。

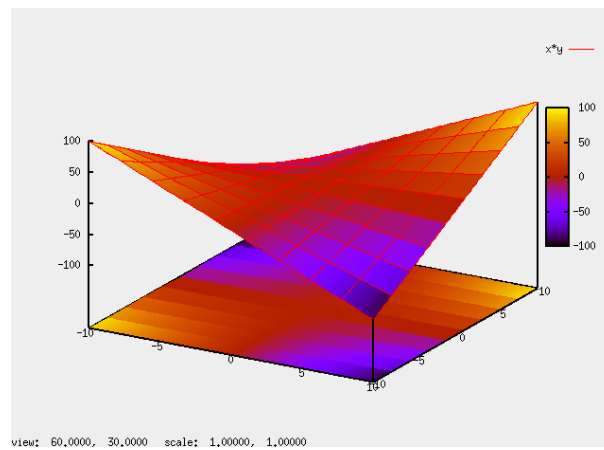


図 5.18: set pm3d at bs の場合

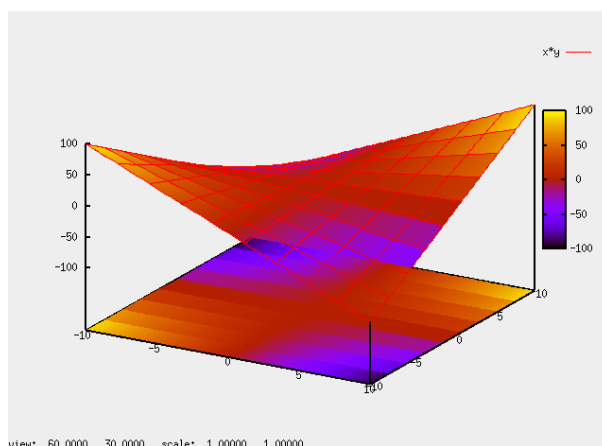


図 5.19: set pm3d at sb の場合

図 5.18 と図 5.19 の違いが判ったでしょうか? 先ず, 図 5.19 では, 底部の絵の一部が上にある筈の曲面に被さっているのに対し, 図 5.18 では逆に底部の絵を曲面が覆い被さっていますね. これは, at の後が bs の場合は最初に底面 (b) を描いてから曲面 (s) を描く為に, 曲面が底面を覆う事になりますが, sb の場合は, 曲面 (s) から描いて次に底面 (b) を描く為に底面が曲面を覆ってしまう事になるのです.

この様に, at の後の文字列は左側から実行される為, 射影を描いた時に, 上にある側が覆い被されている場合は, この設定の順序を間違えていないか確認すると良いでしょう.

猶, 底面や上面への曲面の投影図の位置を調整する事も可能です. この場合, ticslevel で数値を指定して行います.

投影面の位置の変更

```
set ticslevel < 数値 >
show ticslevel
```

ここでの数値は $\frac{\text{投影面の } Z \text{ 座標} - \text{曲面の最高値}}{\text{曲面の最低値} - \text{曲面の最高値}}$ に等しくなります. 則ち, 0 を指定した場合, 投影面は底辺側で, -1 を指定すると投影面は頂部側に配置される事になります.

これを確認する為、以下に $1 - \log(x^2 + y^2)$ のグラフを描いた例を示しておきます。

ここで図 5.20 では `set pm3d at bs` として曲面が投影面を覆う様にし、図 5.21 では `set pm3d at sb` として投影面が曲面を覆う様に設定しています。

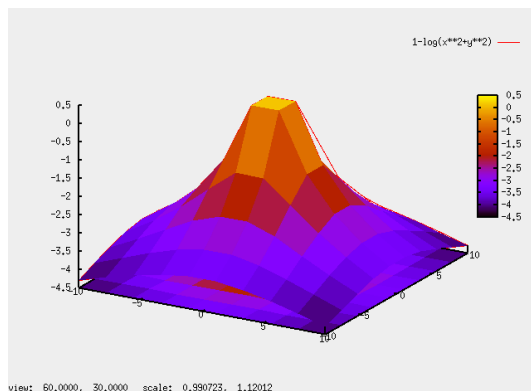


図 5.20: ticslevel が 0 の場合

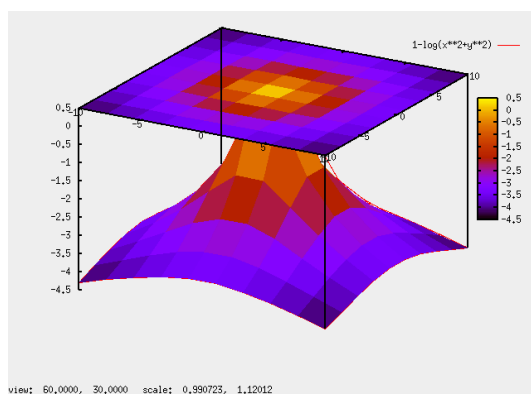


図 5.21: ticslevel が-1 の場合

5.12.6 等高線の階調の変更

pm3d を有効にしていると, 曲面は等高線状に色付けられています. ここで, この階調は変更が可能です. 階調の変更は palette に以下の様な値を設定する事で行います.

palette の変更

```
set palette color
set palette color positive
set palette color nagative
set palette gray
set palette gray positive
set palette gray nagative
set palette rgbformulae < 数値1>, < 数値2>, < 数値3>
show palette
```

では例として以下の命令文を実行した得られたグラフを使って, palette の指定を変更してみましょう.

```
set hidden3d;
set isosample 50;
splot 10-log(sqrt(x**2+y**2+1));
```

この命令文では, hidden3d を設定する事で曲面上の網目を非表示にし, isosample を 50 とする事で X 軸, Y 軸側の分割を各々 50 にして曲面の解像度を上げています. 因に, gnuplot では描画した曲線や曲面の設定を変更しただけであれば, replot 命令を実行するだけで再描画を行います.

この状態から set palette color positive を実行させて再描画したものが図 5.22 に示すグラフになります. この設定が通常の等高線の階調になります.

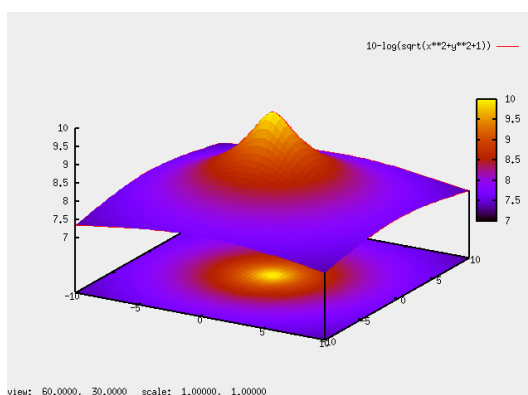


図 5.22: set palette color positive;replot

この図に示す様に, 明るい色の方が Z 座標が高くなります. もしも, 階調を逆にしたければ, 今度は

color の後に negative を指定して palette の再設定を行い、それから描画を行えば良いのです。図 5.23 に階調を逆に設定した場合の再描画の様子を示しています。

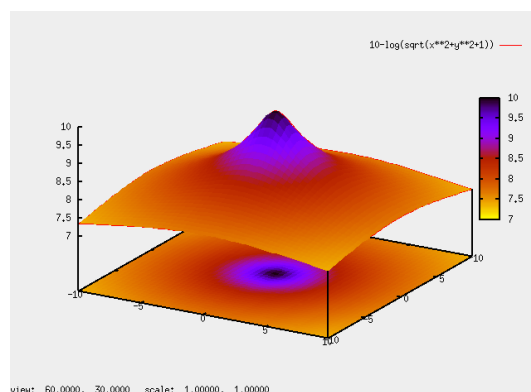


図 5.23: set palette color negative;replot

階調には通常の color の他に白黒の gray もあります。gray を指定した時の様子を以下の図 5.24 に示しておきます。

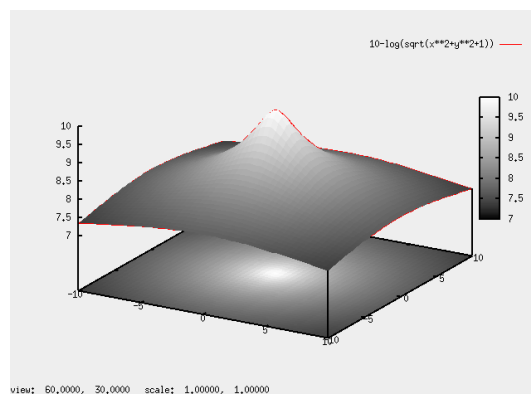


図 5.24: set palette gray positive;replot

gray の場合も, 逆の階調にする場合に gray の後に negative を指定して, palette を設定するだけで済みます,

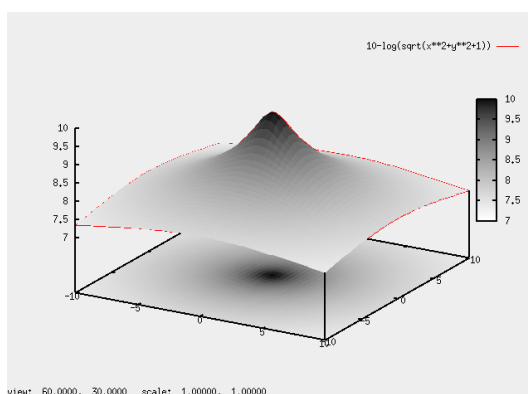


図 5.25: set palette gray negative;replot

この palette は他に色々指定が可能です. 例えば, rgbformulae を使って次の図 5.26 に示す様な階調も得られます.

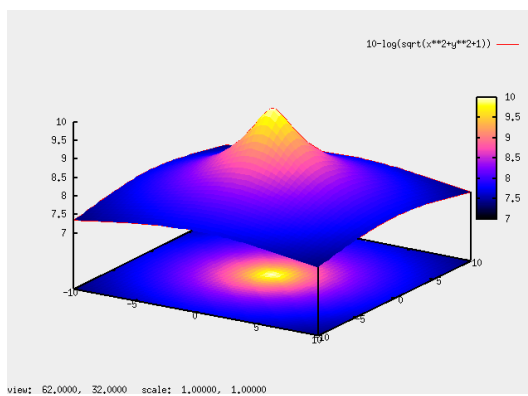


図 5.26: set palette rgbformulae 30,31,32;replot

5.12.7 等高線の調整

さて, 等高線の表示自体は `pm3d` を有効にしなくても表示されます. この場合は本当に線だけになります, `pm3d` と共存する事が可能です.

まず, 等高線を表示する為には, `contour` を設定します.

等高線の設定

```
set contour base
set contour surface
set contour both
set contour
show contour unset contour
```

等高線は `set contour` で有効になります. 猶, この場合は等高線の射影が描かれるだけで, 曲面の側には等高線が現われません. これは `contour` に `base` を指定する `set contour base` と同じです. 次に `contour` に `surface` を指定すれば, 今度は曲面の側だけに等高線が表示されます. そして最後の `both` を指定した場合, 曲面の等高線と射影の両方が表示されます.

この事を実際に確認してみましょう. その為に, ここでは以下の設定を始めに実行してから順次, `contour` の設定を切替えて `replot` 命令を用いて再描画してみましょう.

```
set isosamples 50
set hidden3d
splot [-4:4] [-4:4] -x*y*cos(x**2+y**2)
```

`isosamples` を 50 に設定し, `splot` の XY 座標の範囲は, [-4,4] に限定しています. `hidden3d` を指定する事で, 曲面の陰線処理を行わせる様にしています.

先ず, `set contour` を指定して, `replot` を実行した結果を図 5.27 に示します.

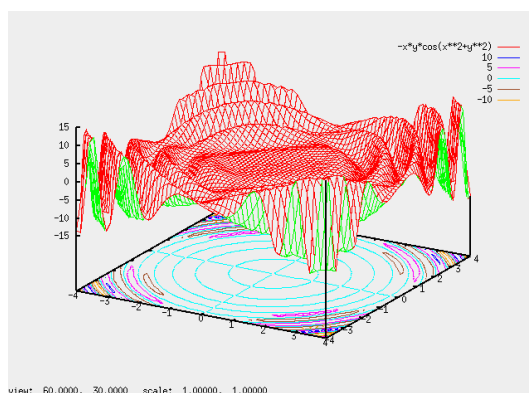


図 5.27: `set contour;replot`

この図 5.27 では等高線が投影面に描かれていますね. 次に `set contour surface` を指定して,

再描画した結果を図 5.28 に示しましょう。

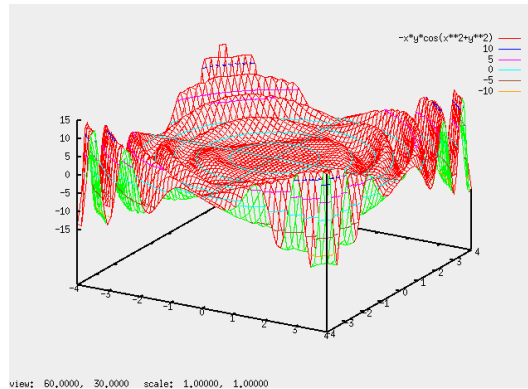


図 5.28: `set contour surface;replot`

今度は投影面が消えて、曲面に等高線が描かれています。ここで `unset surf` を実行しておけば、曲面のワイヤーステイクも消えて等高線だけが表示されます。

最後に、`set contour both` を実行して再描画した結果を図 5.29 に示します。

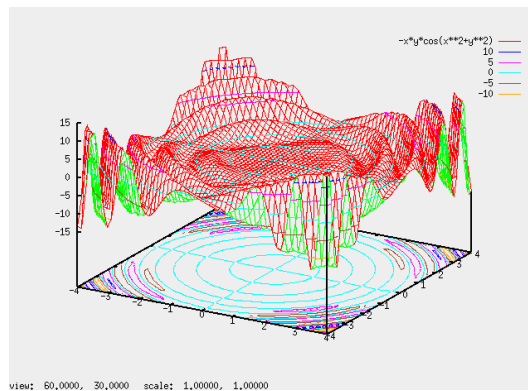


図 5.29: `set contour both;replot`

この図に示す様に、`contour` に `both` を指定する事で、今度は曲面と投影面の両方に等高線が描かれていますね。

`contour` の設定は `pm3d` の設定とも共存する事が可能です。例えば、`set pm3d at s` を指定し、`set contour both` を設定した場合のグラフの様子を見る為に、以下の入力をしてみましょう。

```
set isosamples 100
set hidden3d
set contour
set pm3d
```

```
unset surface
splot -x*y*cos(sqrt(x**2+y**2))
```

この結果が図 5.30 になります。

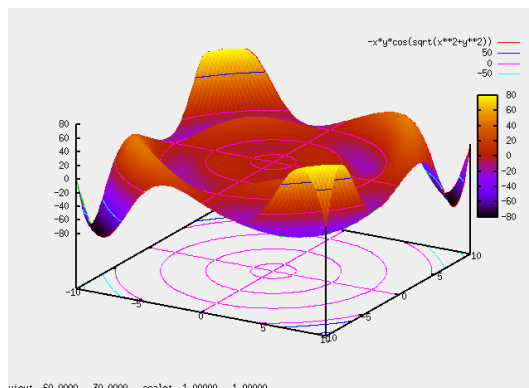


図 5.30: pm3d との共存

この例では,unset surf を実行する事で,等高線以外の曲面上の曲線を消去しています。

gnuplot では等高線の間隔の調整も行えます。この等高線の間隔等の微調整は cntrparam を設定する事で行えます。以下にその構文を示しておきましょう。

等高線の調整

```
set cntrparam linear
set cntrparam cubicspline
set cntrparam bspline
set cntrparam points <個数>
set cntrparam order <次数>
set cntrparam levels <等高線総数>
set cntrparam levels discrete <高度1>, ..., <高度n>
set cntrparam levels incremental <開始>, <増分>, <終点>
set cntrparam levels incremental <開始>, <増分>
```

先ず,linear,cubicspline,bspline は等高線の補間式の指定になります。初期値としては線形補間 linear が設定されています。cubicspline や bspline を利用する場合,points に指定する数値が補間の点数になり,order で指定する整数値が bspline の補間式の次数を定めます。これらの数値は大きい程,滑かな曲線が描ける事になります。

次に,等高線の総数は初期値が 5 となっています。この場合,最低値から最高値を 5 本の等高線で高度を等分する事になります。

この様な等高線の設定は levels で指定出来ます。level では等高線の総数を指定したり,指定した高さに等高線を描いたり,指定した間隔で定められた範囲の等高線を描く様に指定する事が可能です。

先ず, 等間隔の等高線を描く場合には, `level auto` で等高線の総数を指定します. この時, 等高線は最低点と最高点の 2 点を含めて, 指定した数だけ高度に対して等間隔に描かれます.

次に, 指定した高度の等高線だけを描かせる場合は, `levels discrete` で等高線を描く高度を指定します.

例えば, 以下の入力を行った場合, 図 5.31 に示す絵が得られます.

level discrete の例

```
set contour
set isosamples 50
set cntrparam levels discrete -20,0,20
splot x*y*sin(sqrt(x**2+y**2))
```

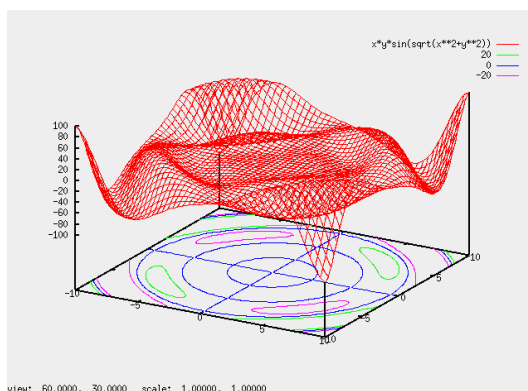


図 5.31: 指定した高さの等高線を描画

ここでの入力は, 等高線を高さ-20,0,20 の三種類描く事を

```
set cntrparam level discrete -20,0,20
```

で指定しています. その為, 図 5.31 の凡例には曲面と 3 個の等高線が出ています.

最後に, 一定の間隔で決められた高度の範囲で等高線を描画する場合は `incremental` を用います. この `incremental` の引数は始点, 増分, そして終点の順で 3 個の引数を記述します. ここで増分は始点と終点が矛盾さえないければ正でも負でも構いません.

先程の例に以下の命令を実行した結果を図 5.32 に示しておきます.

```
set cntrparam level incremental -20,2,20
replot
```

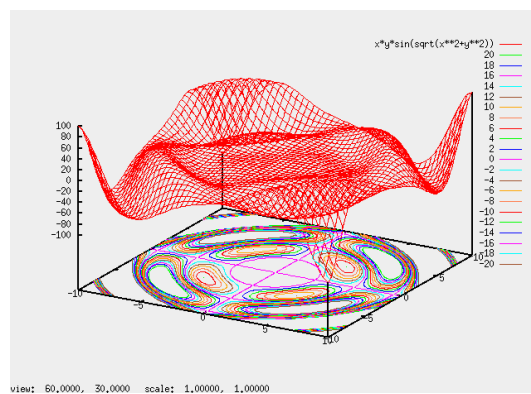


図 5.32: 指定した高さの等高線を描画

5.12.8 視点の変更

gnuplot で 3 次元グラフの視点は view で設定出来ます。その構文を以下に示しておきましょう。

view の設定

```
set view <X 軸の回転角>, <Z 軸の回転角> <拡大率>, <Z 軸方向の拡大率>
set view <X 軸の回転角>, <Z 軸の回転角>
set view map
show view
```

ここで, view の初期値は X 軸回りの回転角が 60 度, Z 軸回りが 30 度で, 各軸方向の拡大率は 1 になっています。

又, view には map があります。こちらは効果としては, `set pm3d map` と同じで, 曲面を真上から眺める形になります。これは X 軸回りと Z 軸回りの回転角が 0 度の場合に一致します。

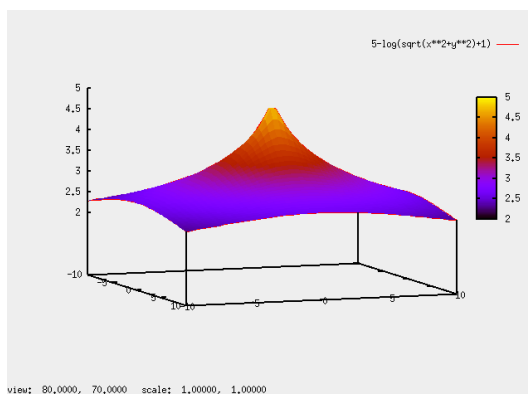


図 5.33: set view 80,70,1,1

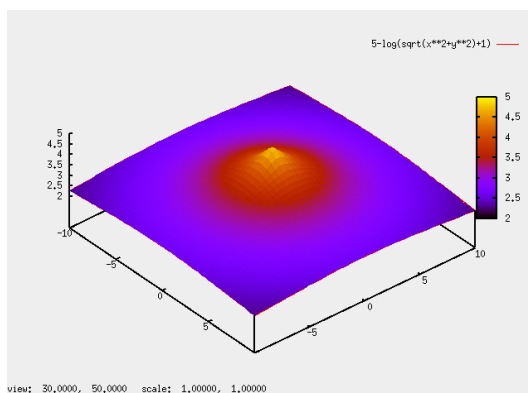


図 5.34: set view 30,50,1,1

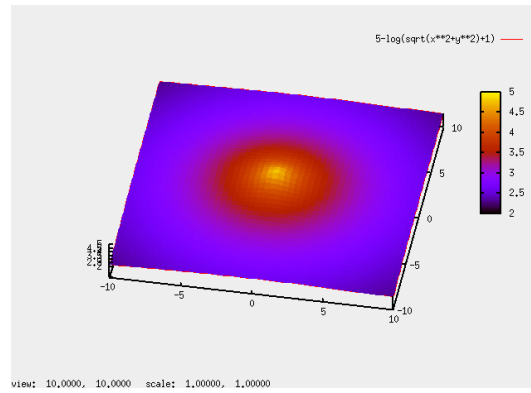


図 5.35: set view 10,10,1,1

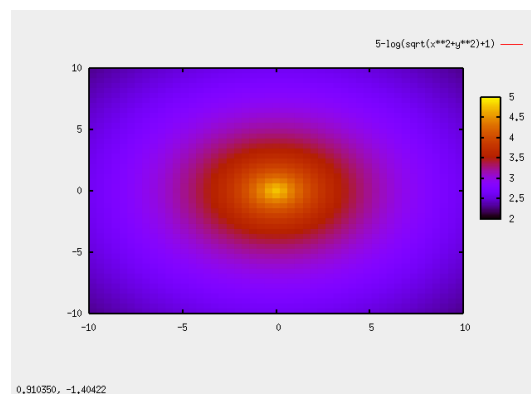


図 5.36: set view map

5.12.9 cbrange と cblabel

pm3d を有効にしていると、右側に曲面の階調を示す箱が表示されています。この階調の調整は cbrange で行いますが、その構文は xrange と同じです。まず, cbrange は曲面の高さと階調の対応関係を指定するものです。曲面の最低値と最高値が、この cbrange の下端と上端であれば、きめ細かな階調表示になりますが、この cbrange の幅が曲面の高低差と比較して大き過ぎたりすれば、曲面の階調が殆ど出ない、のっぺら坊な曲面になってしまいます。

cbrange の設定

```
set cbrange [〈下限〉 : 〈上限〉]
unset cbrange
show cbrange
```

この cbrange の設定で、階調の範囲を指定する方法は、MATLAB のベクトルの表記の様な [a:b] を用います。これは通常のリスト [a,b] とは異なるので注意が必要です。

更に、図の右側に通常出現しているこの階調ボックスにラベルを表示させる事も可能です。このラベルの表示も X 軸等の各軸にラベルを配置する xlabel 等と同様の構文になります。

cblabel の設定

```
set cblabel 〈文字列〉 〈x 座標〉, 〈y 座標〉
set cblabel 〈文字列〉
unset cblabel
show cblabel
```

ここで、〈x 座標〉, 〈y 座標〉は何も指定しない位置を基準とした文字列表示位置を指定します。ここでの座標は縦上方向を Y 軸正方向、横軸右向きを X 軸の正方向とするもので、1 で一文字分になります。この値は表示フォントやターミナル等の環境で異なります。

実例として先程の図 5.26 に対して以下の処理を行ったものを図 5.37 に示しておきます。

```
set cblabel 'Height' -13,6;  
set cbrange [8:10];  
replot
```

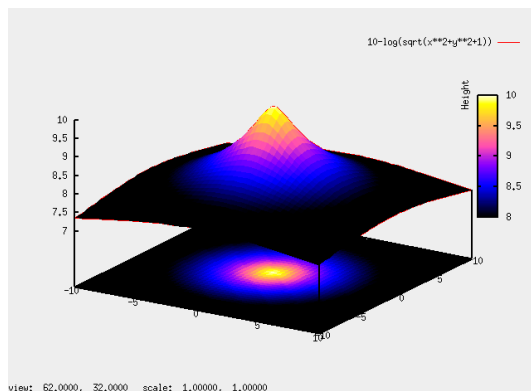


図 5.37: set cbrange [8:10];replot

ここでは cbrange を狭くした為、高さが 8 以下は真っ黒になっています。次に、以下の処理を行ったものを図 5.38 に示します。

```
set cblabel 'Height' -13,6;  
set cbrange [5:10];  
replot
```

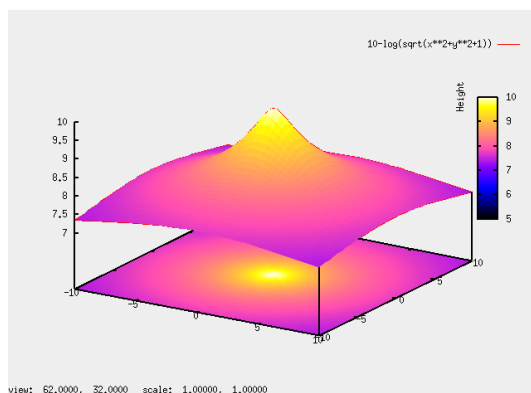


図 5.38: set cbrange [5:10];replot

今度は下に長く取った為に、曲面全体が明るくなっている事が分ります。

5.12.10 陰線処理

gnuplot で三次元グラフを行うと、通常は陰線処理を行っていません。陰線処理を行う為には、hidden3d を有効にします。因に、この hidden3d は pm3d が無効でも、有効になっていても無関係に使えます。

又、pm3d が有効の場合、unset surface を実行する事で pm3d の曲面上の網目を非表示にする事が可能です。但し、こちらは pm3d の曲面の設定を無効にする副作用があります。

ここでは set hidden3d と unset surface の違いを確認しましょう。比較で用いるグラフは以下の設定とします。

```
set pm3d at bs
set cbrange [-0.05:1]
splot 1/(x**2+y**2+1)
```

この処理結果を初期状態とし、図 5.39 にそのグラフを示しておきます。

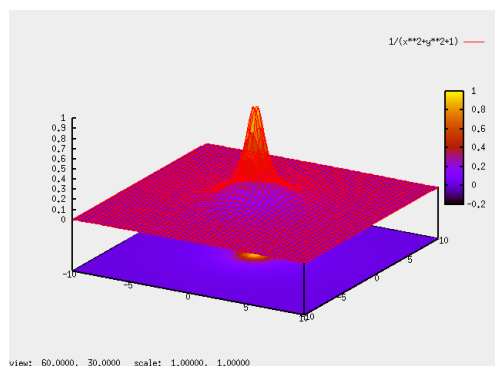


図 5.39: 初期状態

次に、set hidden3d を実行した後に replot した結果を図 5.40 に示します。

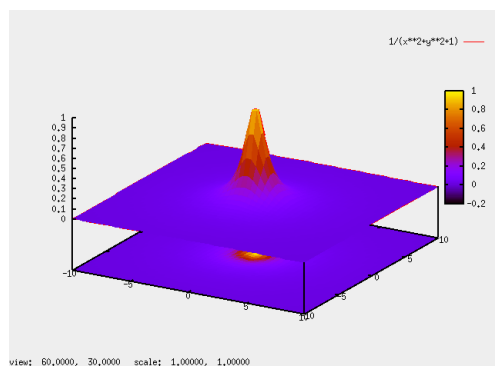


図 5.40: set hidden3d

最後に,unset surface を実行して replot した結果を図 5.41 に示しておきます.

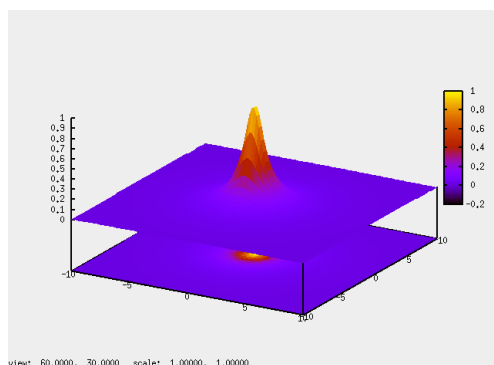


図 5.41: unset surface

hidden3d と unset surface の違いは hidden3d では稜線や境界に線分が残っていますが,unset surf では一切の線分が消えている事です.

猶,gnuplot の式を描画する場合,陰線処理を有効にした場合,解像度の問題から,極が埋没する傾向が強くなります.これは Maxima の様に高精度で計算したデータを用いる場合ではグラフの形が大きく異なるので問題が良く分ります.

実際に確認してみましょう.ここで,描く関数は $\log(x^2y^2)$ とします.

最初に gnuplot で安易に `splot log(x**2*y**2)` を実行した結果を図 5.42 に示します.こう見ると結構綺麗な曲面ですね.

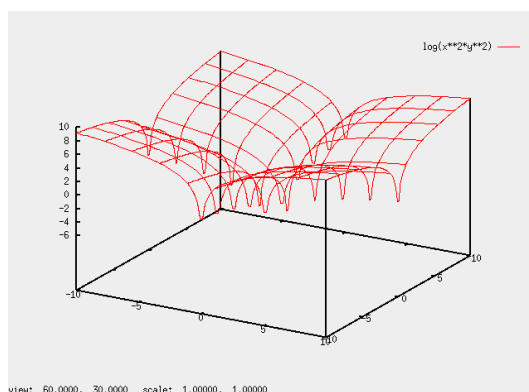


図 5.42: splot log(x**2*y**2)(その 1)

次に `set pm3d at s` で `plot log(x**2*y**2)` を実行した結果を図 5.43 に示しておきます。この図からは、面を張ったが為に、粗さが目立っています。

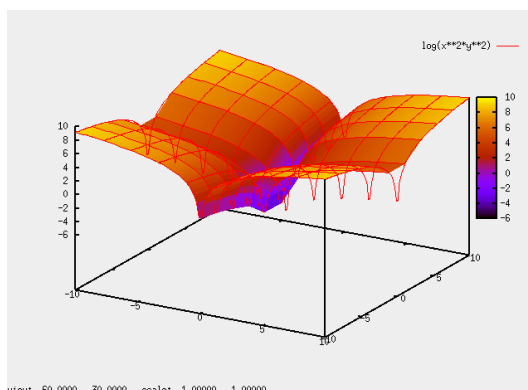


図 5.43: `plot log(x**2*y**2)`(その 2)

そこで、`set isosample 100` に変更して、より細かな分割で曲面を `replot` した結果を図 5.44 に示します。

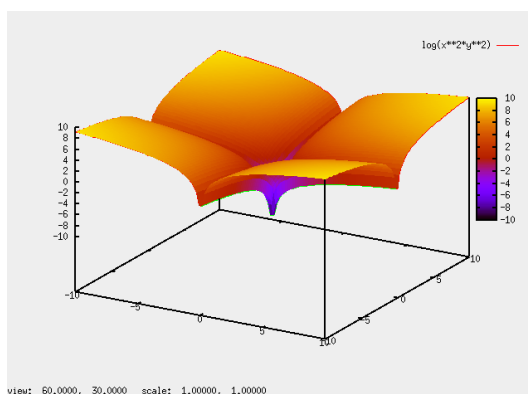


図 5.44: `plot log(x**2*y**2)`(その 3)

こう見ると、原点付近が極みたいです。

では、この関数を Maxima から表示してみましょう。gnuplot と同様の細かさを得る為に、ここでは以下の処理を Maxima に実行させます。

Maxima での描画処理

```
plot3d(realpart(log(x^2*y^2)), [x, -10, 10], [y, -10, 10],
[grid, 100, 100], [gnuplot_preamble, "set pm3d at s; unset surf"]);
```

この処理結果を図 5.45 に示しておきます。

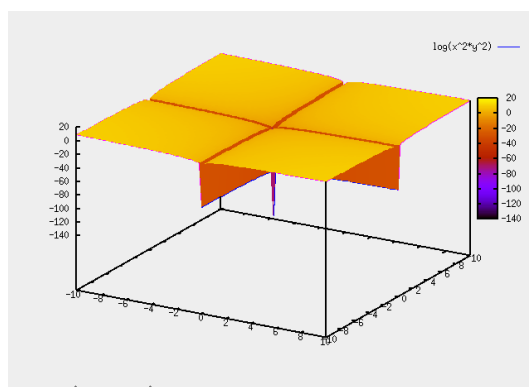


図 5.45: Maxima から $\log(x^2y^2)$ の描画

この図になると、今迄の gnuplot の式のもの比べて谷と山が遥かに深くなっています。何故、同じ gnuplot でも Maxima から描くと違った形の絵になるのでしょうか？単純にスケールの違いでしょうか？

これは、Maxima では maxout.gnuplot を生成する為に数値データを生成している為に、splot で式を記述する方法よりも良好な結果が得られているのです。

実は式を良く見れば分る簡単な事ですが、X 軸上の点と Y 軸上の点が函数の極になっています。即ち、gnuplot の表示は大人し過ぎるのです。この意味では、Maxima の処理の方がまだ雰囲気良く出ていると言えます。

さて、この式を Maxima で描画する際には、Maxima から色々質問が出ています。

————— $\log(x^2y^2)$ の Maxima での描画処理 —————

```
(%i6) plot3d(realpart(log(abs(x))+log(abs(y))), [x, -10, 10], [y, -10, 10],
[grid, 100, 100], [gnuplot_preamble, "set pm3d at s; set hidden3d;"]);
Is x zero or nonzero?

nonzero;
Is y zero or nonzero?

nonzero;
(%o6)
```

ここで Maxima が聞いている事は、 x と y が零か零でない数値であるかを尋ねているのです。この様な事を Maxima が聞いて来る時の多くは、 x や y の何れかが零になると非常に都合の悪い事が生じる可能性がある事を示唆しています。この様な質問が出た場合には、函数の定義域に関して吟味する必要があるのです。この点から、Maxima の plot.lisp で記述されている内容の方が gnuplot よりも遥かに安全であると言えます。

5.12.11 曲線と曲面の細かさの指定

gnuplot の式を表示して粗さが目立つ場合, 曲線の場合は `sample`, 曲面の場合は `isosamples` の設定を大きなものに変更します.

— sample と isosamples の設定 —

```
set isosample < 数値1>, < 数値2>
set sample < 数値1>
set isosamples < 数値1>, < 数値2>
set isosamples < 数値1>
show isosamples
```

`sample` と `isosamples` には二つの数値か一つの数値を指定します. `sample` の初期値は 100 で, `isosamples` の初期値は 10 です.

`sample` の場合, X 軸上の総点数がこの `sample` で指定した値になります. 実例として, 以下で描画した $\sin(\frac{1}{x})$ のグラフの `sample` を変更して replot したものを示します.

```
set xrange [-1:1]
plot sin(1/x)
```

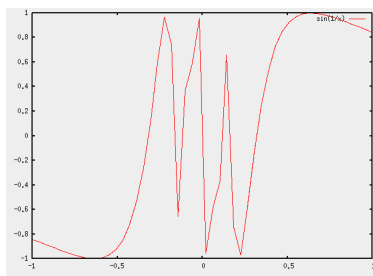


図 5.46: set sample 50 の場合

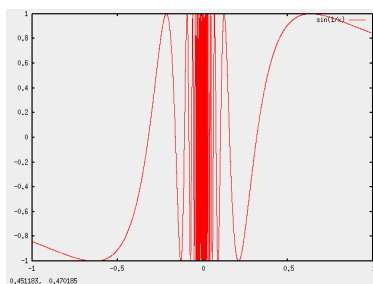


図 5.47: set sample 10000 の場合

isosamples の場合、 $\langle \text{数値}_1 \rangle, \langle \text{数値}_2 \rangle$ で X 軸と Y 軸の分割数を定め、 $\langle \text{数値}_1 \rangle$ だけが指定された場合には、X 軸も Y 軸も、この分割数になります。

但し、Maxima から曲線や曲面を描画する場合、plot2d 関数や plot3d 関数に与える Maxima の式から gnuplot の描画に必要な数値データを計算してしますが、この計算の際に大域変数 plot_options の nticks や grid に設定した値が反映されます。その為、gnuplot_preamble で sample や isosample を指定しても、この設定は gnuplot の式の描画にだけ有効な為、直接役には立ちません。とは言え、gnuplot_curves_titles を使って、gnuplot の式を埋込む場合には有効です。

5.12.12 描画の領域設定

plot 命令や splot 命令で描画する gnuplot の式に対し、その描画する範囲は xrange, yrange や zrange で指定します。

描画する領域を指定する場合

```
set xrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション1 ⟩ ⟨ オプション2 ⟩
set xrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション1 ⟩
set xrange [⟨ 下限 ⟩:⟨ 上限 ⟩]
set xrange restore
show xrange
set yrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション1 ⟩ ⟨ オプション2 ⟩
set yrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション1 ⟩
set yrange [⟨ 下限 ⟩:⟨ 上限 ⟩]
set yrange restore
show yrange
set zrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション1 ⟩ ⟨ オプション2 ⟩
set zrange [⟨ 下限 ⟩:⟨ 上限 ⟩ ⟨ オプション1 ⟩
set zrange [⟨ 下限 ⟩:⟨ 上限 ⟩]
set zrange restore
show zrange
```

cxlabel の小節でも解説しましたが、ここで領域の設定は MATLAB のベクトルの定義に似た書式となります。即ち、[-1:1] と記述し、[-1,1] でない事に注意が必要です。ここで下限の初期値は-10、上限の初期値は 10 になっています。その為、何も指定せずに plot 命令や splot 命令で描画する時は、この領域で描画が実行されます。

この領域の設定には、オプションとして軸の向きを逆にする `reverse`、元に戻す `noreverse` があります。早速、`reverse` を `plot` と `splot` で試してみましょう。

```
set xrange [0:10] reverse
plot cos(x)
```



図 5.48: `plot cos(x)`

```
set zrange [0:10] reverse
splot sqrt(x**2+y**2)
```

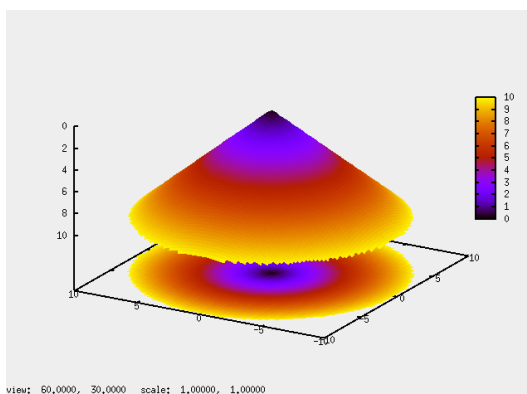


図 5.49: `splot sqrt(x**2+y**2)`

図 5.48 では X 軸の向きが逆になっていますね。又、図 5.49 の場合は、Z 軸が通常の逆になっています。

猶、領域を初期値に戻したければ、`set xrange restore` の様に区間を記述せずに `restore` を指定すれば良いのです。

5.12.13 ラベル表示と注釈に関連する事項

gnuplot の便利の良さの一つに、曲線や曲面に色々とラベルや注釈を入れる事が容易な事が挙げられるでしょう。実際、この辺の機能は非常に豊富にあり、この小冊子で把握出来る様なものではありません。その為、ここでは本当に一部の機能の解説に留めておきます。詳細は文献 [14] を参照して下さい。

では手始めに座標軸のラベルの指定について解説します。

軸のラベル指定

軸のラベルを指定する場合

```
set xlabel < 文字列 > < X 座標 >, < Y 座標 >
set xlabel < 文字列 >
show xlabel
set ylabel < 文字列 > < X 座標 >, < Y 座標 >
set ylabel < 文字列 >
show ylabel
set zlabel < 文字列 > < X 座標 >, < Y 座標 >
set zlabel < 文字列 >
show zlabel
```

先ず, xlabel, ylabel, zlabel は X 軸, Y 軸, そして Z 軸のラベルを指定します。文字列の後に入れる X, Y 座標は、通常の文字列が表示される位置を基準として、その位置から何文字分移動するかを指定します。但し、ここでの値はフォントの指定等で異なる値になります。

曲線の表題表示

複数の曲線を表示した際に, 右上側に線分と曲線の表題が並んで表示されています. gnuplot ではこれを key と呼んでおり, この key の位置, key に表示する線分の長さ等の指定が可能です.

最初に key の表示位置を設定する構文を以下に示しておきます.

—— グラフの key の位置を設定 ——

```
set key right
set key left
set key top
set key top left
set key top right
set key bottom
set key bottom left
set key bottom right
set key outside
set key below
```

これらの設定は, key のグラフ上での位置を指定するものです. 更に, key の表示自体を調整する事も可能です.

—— グラフの key を設定 ——

```
set key reverse
set key noreverse
set key samplen < 線分の長さ >
set key box linetype < 線分の様式 > linewidth < 線分の幅 >
unset key box
show key
```

ここで key に対する reverse は線分とその名称の羅列の順序を逆にするものです. noreverse は初期状態に戻します.

samplen は key で表示されている線分の長さの調整に用います.

set key box linetype は, key の線分の様式や幅を変更するものです. 最後に key を表示したくなければ, `unset key box` とします.

では、各軸のラベルと key の設定例を以下に示しておきます。

```
set xlabel 'Time -sec-'
set ylabel 'Height -mm-' 0,40
set key outside
set key reverse
set key samplen 10
plot sin(x),cos(x),sin(x)/x
```

この例では、X 軸のラベルを 'Time -sec-', Y 軸のラベルを 'Height -mm-' とし、Y 軸のラベルは 40 文字程上側に配置します。そして、key をグラフの外側に置き、この key での線分の長さを 10 とし、ここでの線分と曲線名の順番を reverse で逆にしています。

この様子を図 5.50 に示しておきます。

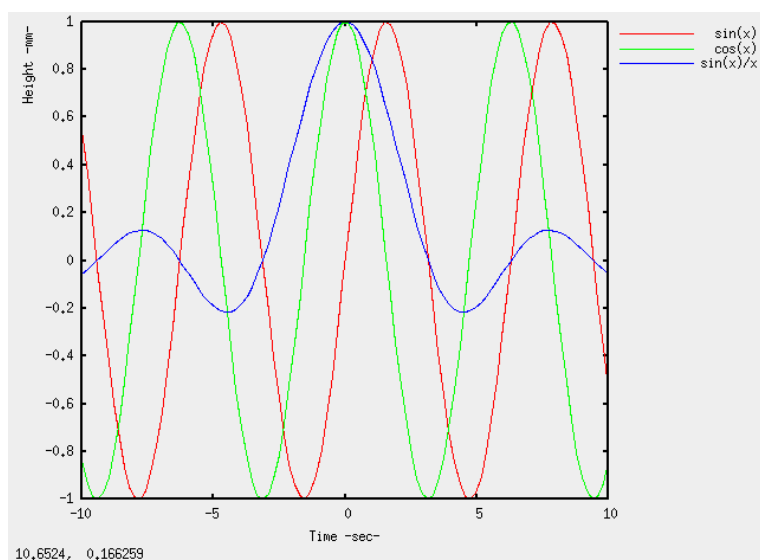


図 5.50: 各軸のラベルと key の設定例

注釈と矢印の追加

gnuplot ではグラフに注釈を入れる事が可能です. この注釈は label で設定します.

— 註釈の設定 —

```
set label < 文字列 >, < 注釈番号 > at < X 座標 >, < Y 座標 >
set label < 注釈番号 >, < 文字列 > at < X 座標 >, < Y 座標 >
set label < 文字列 >
set label < 注釈番号 > < 位置合せ >
set label < 注釈番号 > font < フォント名 >, < 大きさ >
set label < 注釈番号 > front
set label < 注釈番号 > back
set label < 注釈番号 > textcolor < 表示色 >
set label < 注釈番号 > rotate by < 角度 (deg) >
set label < 注釈番号 > norotate
set label < 注釈番号 > point pointsize < 数値 >
unset label < 注釈番号 >
unset label
show label
```

gnuplot では, set label 文を < 注釈番号 > を指定せずに実行すると, 自動的に注釈に適切な番号を割当てます. その後の註釈の回転, 内容やフォントの変更等の処理はこの注釈番号を指定して行わなければなりません. 指定しない場合には新規の註釈が生成されるだけです.

ここで注釈の座標は先程のグラフの点の位置に対応します. 猶, 文字列の左寄, 中寄, 右寄といった位置決めは各々 left, center, right を指定する事で行います.

注釈を XY 面内で回転させる事も可能です. この場合, rotate by で行いますが, by の後に指定する角度の単位は度になります.

では実例として, 以下に示す処理を gnuplot でさせてみましょう.

```
set label 1 ' Comment at Origin' at 0,0
set label 2 ' Comment at (1,1)' at 1,1
set label 1 point pointsize 6
set label 2 point pointsize 2
set label 2 rotate by 90
plot sin(x)*2
```

最初に注釈を原点と (1,1) に配置しています. それから註釈番号 1 の註釈に対して大きさが 6 の点を配置しています. 同様に, 番号 2 の註釈には大きさが 2 の点を配置します. それから番号 2 の註釈を 90 度回転させています. 最後は, plot 命令で $2 \sin x$ 関数の描画を行います.

この結果を図 5.51 に示しておきます。

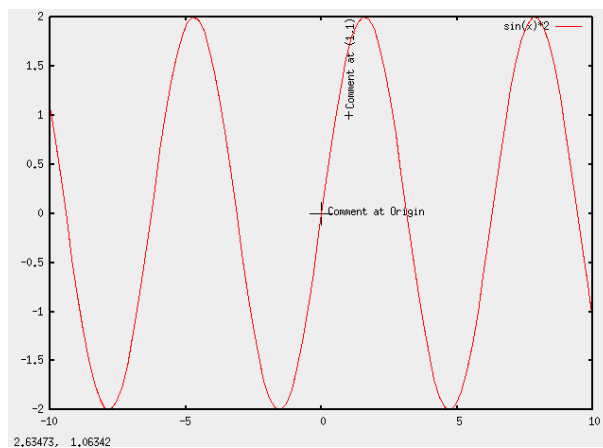


図 5.51: 注釈の設定例 (その 1)

或る注釈を非表示にしたければ、`unset label <注釈番号>` と入力する事で対処します。ここでは図 5.51 の例に対し、番号 2 の注釈を `unset label 2` を実行した後に replot した結果を図 5.52 に示しておきます。

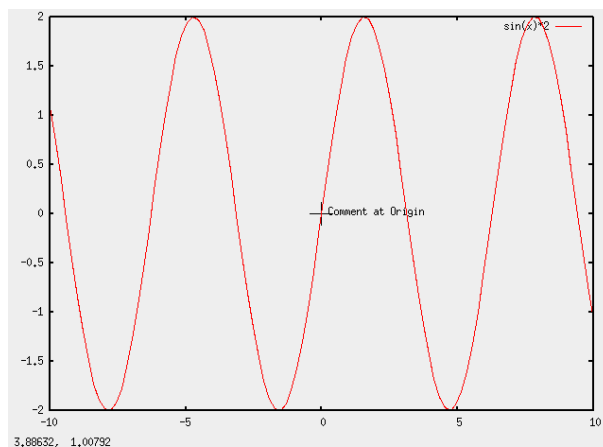


図 5.52: 注釈の設定例 (その 2)

番号 2 の注釈が実際に消えている事が分ります。

矢印の追加

gnuplot では矢印をグラフ中に追加する事も可能です. この矢印は註釈と併用する事で効果が上ります.

arrow による矢印の設定

```
set arrow <番号> from <X座標>, <Y座標> to <X座標>, <Y座標>
set arrow <番号> from <X座標>, <Y座標>
set arrow <番号> to <X座標>, <Y座標>
set arrow <番号> size <矢の長さ>, <矢の角度>
set arrow <番号> head
set arrow <番号> heads
set arrow <番号> nohead
set arrow <番号> filled
set arrow <番号> nofilled
unset arrow
show arrow
```

やっている事は label を用いた註釈の設定に似ています. 先ず, 図形としての矢印の設定は `set arrow 1 from 0,0 to 1,1` の様に始点と終点を決めるだけで生成されます. この時, 矢印の向きは from から to に向きます. 猶, from の点, 或いは to の点が原点の場合は省略しても構いません.

次に arrowsize で矢印の頭の部分を調整する事が可能です. ここでの <矢の長さ> は矢印の頭の矢の長さで, <矢の角度> は頭の矢の角度になります. この角度は度数です. filled を指定すると, 矢印の矢の部分塗り潰されます.

では実際に以下のように入力してどのようなグラフになるか確認してみましょう.

```
set arrow 1 from 0.4,-1 to 1.1,-1
set arrow 2 from 0.9,1 to 1.1,1
set arrow 3 from 1,-1 to 1,1
set arrow 1 nohead
set arrow 2 nohead
set arrow 3 size 0.1,30 filled heads
set label 1 ' Width' at 1.05,0
plot cos(6.283*x);
set xrange [0:2]
set yrange [-2:2]
plot cos(6.283*x);
set arrow 3 filled
plot cos(6.283*x);
```

ここでは最初の三行で矢印を生成しています. 次の三行で各矢印の性質を指定しています. ここで, 第三の矢印のみ両端に矢を付け, その他は棒にします. そして, 第三の矢印の矢の部分を大きく

して塗り潰してしまいます。この矢印の設定は、この様に一度に設定しなければ、最後に入力した設定だけが有効になり、他は初期化されてしまうので注意が必要です。

この結果を図 5.53 に示しておきます。

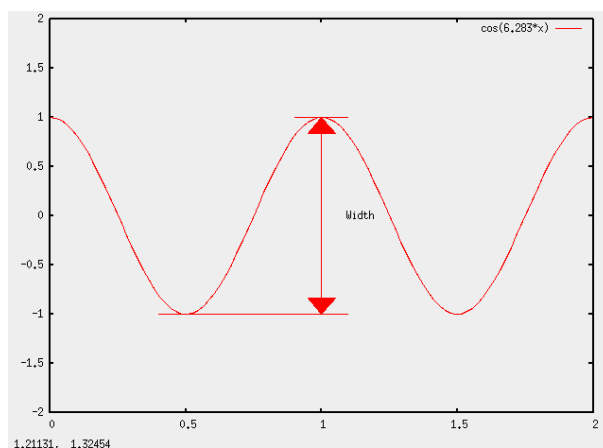


図 5.53: 矢印と註釈の設定例

5.12.14 gnuplot の式

gnuplot では FORTRAN で用いられる数式が利用可能です。その他の関数も充実していますが、ここでは簡単に gnuplot の式について述べます。

まず、式の演算子は FORTRAN の書式に準じます。gnuplot の算術演算子を以下に示しておきます。

gnuplot の算術項演算子

演算子	例	概要
+	a+b	和
-	a-b	差
*	a*b	積
**	a**b	冪
/	a/b	商
%	a%b	剰余
-	-a	負の演算子
+	+a	正の演算子
!	a!	階乗

この様に、gnuplot では一般的な算術演算子の殆どが使えます。但し、冪は Maxima 等の数式処理でよく用いられる \wedge ではなく、FORTRAN 等で用いられる $**2$ を用います。この点は Maxima と併用する場合には特に間違え易いので注意しましょう。

更に,gnuplot では様々な数学函数が使えます. 以下に gnuplot で利用可能な函数の一覧を示しておきます.

数学函数

abs	絶対値
sqrt	平方根を計算
imag	複素数値の虚部を返す函数
real	複素数値の実部を返す函数
rand	乱数函数

三角函数

cos	余弦函数
sin	正弦函数
tan	正接函数
acos	逆余弦函数
asin	逆正弦函数
atan	逆正接函数
atan2	逆正接函数
cosh	双曲余弦函数
sinh	双曲正弦函数
tanh	双曲正接函数
acosh	逆双曲余接函数
asinh	逆双曲正弦函数
atanh	逆双曲正接函数
exp	指数函数
log	自然底の対数函数
log10	底が 10 の対数函数
arg	偏角を返す函数

特殊函数

ibeta	不完全 B 函数
gamma	Γ 函数
igamma	不完全 Γ 函数
besj0	第一種のベッセル函数 $J_0(x)$
besj1	第一種のベッセル函数 $J_1(x)$
besy0	第二種のベッセル函数 $Y_0(x)$
besy1	第二種のベッセル函数 $Y_1(x)$
erf	誤差函数
erfc	相補誤差函数 ($\text{erfc}(x)=1-\text{erf}(x)$)
inverf	逆誤差函数
norm	正規分布
invnorm	逆正規分布函数
lambertw	ランベルトの W 函数

その他の函数

ceil	与えられた値を越えない整数に 1 を加えた値を返す. $\text{ceil}(x)$ は $\text{floor}(x)+1$ と同じ
floor	与えられた値を越えない整数を返す
int	小数点の切り捨てを行う函数
sgn	符号函数. 引数が負の場合は -1, 正の場合は 1 を返す

これらの算術演算子と函数の他に論理演算も可能です.

gnuplot の論理演算子

演算子	例	概要
==	a==b	等号
!=	a!=b	等しくない
>=	a>=b	以上
>	a>b	大なり
<=	a<=b	以下
<	a<b	小なり
&&	a&&b	論理積
		論理和
&	a&b	ビット単位の論理積
		ビット単位の論理和
~	~ a	補間
!	!a	否定

ここで独特なのが、ビット単位の論理積&と論理和|です。これらは与えられた整数に対し、それらを二進数で置換えて各桁で論理積や論理和を各々実行した結果を返す演算子です。

これらの演算子を活用する事で複雑な関数を gnuplot 内部で定義し, 更に, その関数を使って描画する事も可能です. この場合, 関数に与えられる式は $\sin(x)**+1$ の様な gnuplot の関数と算術演算子を用いた式, そして, 変数の範囲で関数を切り替える式の二種類が使えます.

—— 関数の定義 ——

$\langle \text{関数名} \rangle (x) = \langle \text{gnuplot の算術演算子と関数の式} \rangle$
 $\langle \text{関数名} \rangle (x) = \langle \text{条件}_1 \rangle ? \langle \text{式}_1 \rangle : \dots \langle \text{条件}_n \rangle ? \langle \text{式}_n \rangle$
 $\langle \text{関数名} \rangle (x) = \langle \text{条件}_1 \rangle ? \langle \text{式}_1 \rangle : \dots \langle \text{条件}_n \rangle ? \langle \text{式}_n \rangle : \langle \text{各条件を満たさない場合の値} \rangle$

条件は gnuplot の論理式で構成されるもので, 関数の定義域を定めるものです. 例えば, -1 から 1 迄が x^2 で, それ以外を 1 とする関数を定義したければ, 以下の様に設定します.

```
f(x) = x >= -1 && x <= 1 ? x**2 : 1
```

この様に, 指定した区間以外の値は関数の定義の最後に値を設定すれば良いのです. 更に, gnuplot 上で定義する関数の変数は x,y,z に限定されません.

猶, 特定の区間のみの値のグラフ表示をしたければ, 特定の区間だけで式を定義しておいて, 最後に, 1/0 や -1/0 の様に極を設定すればグラフ表示の場合には, 指定した区間だけが表示されます.

```
f(x) = x >= -5 && x <= 1 ? cos(x) : x >= 1 && x <= 5 ? sin(x) : \
> x >= 7 ? sqrt(x) : x <= -7 ? (x-x**2)/100 : 1/0
plot f(x)
```

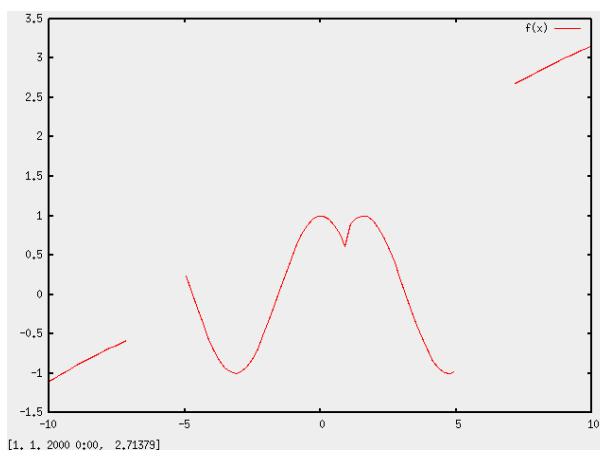


図 5.54: 不連続なグラフの描画

5.12.15 電卓としての gnuplot

この様に豊富な関数もあり、自分で関数も定義出来るのであれば、Octave 迄はいなくても、簡単な卓上計算機として使いたいものです。ところが、Maxima と Octave とは異なる事は、関数に値を入れたものを入力してもエラーが返される事でしょうか。

```
gnuplot> 1+1
~
invalid command

gnuplot> sin(0)
~
invalid command
```

この事は gnuplot の式はグラフ描画の為に用いる事が第一であって、電卓代りにする事はあまり考えていなかった為でしょう。

とは言え、前述の様に C や FORTRAN の様な式が入力可能な程の能力を持っているので、このままにしておくのは非常に勿体無いので、電卓として使う方法について簡単に述べておきましょう。

とは言っても話は非常に簡単な事で、式の結果を何等かの変数に代入して、その変数を print 命令に引渡すか、その式を直接 print 命令に引渡せば良いのです。

```
gnuplot> print 1+1
2
gnuplot> print cos(3.14)
-0.99999873172754
```

で、電卓として使う場合に、この電卓で扱えるデータには何があるでしょうか？先程の式の解説では gnuplot で扱えるデータに関して類被りしていましたが、gnuplot には論理演算を行う為の真偽値、整数値、浮動小数点、そして複素数値があります。

真偽値 gnuplot では真を 1、偽を 0 とします。この値は他の数値と混在しても構いません。この事を利用すれば if 文を用いずに式を容易に定義する事が出来ます。例えば、x が 2 なら 10 を返し、それ以外は -9 を返す関数も以下の様に定義します。

```
gnuplot> g(x) = (x==2)*10 + (x!=2)*(-9)
gnuplot> print g(2)
10
gnuplot> print g(4)
-9
```

この方法は Octave でもお馴染の手応です。この機能を使えば、gnuplot でも長々と条件分岐を書かなくても済む訳です。

整数値, 浮動小数点 整数は小数点を持たない実数で, 浮動小数点は小数点を持つ実数です. ここで整数同士の演算結果は整数になるので, 整数同士の割算を行う場合は注意が必要です.

```
gnuplot> print 4/3, 4.0/3
1 1.333333333333333
```

この例では, 最初の $4/3$ は分子, 分母共に整数の為, 整数の剰余が適用されています. 但し, 最後の $4.0/3$ では分子が浮動小数点の為, 割算の結果は浮動小数点になります. この様に, 実数式で浮動小数点を含む式の場合, 結果は浮動小数点に, 後述の複素数を含む式の場合, その結果は複素数になります.

複素数 gnuplot での複素数は大括弧 $\{\}$ を用います. 例えば, $1 + 2i$ は $\{1, 2\}$ の様に数値の対で表現します. 複素数の場合, 実部と虚部は各々が浮動小数点になります.

```
gnuplot> print 1+{1,3}+{0,3}
{2.0, 6.0}
gnuplot> print 3*{1,4}/9
{0.333333333333333, 1.333333333333333}
```

最初の例で示す様に, 入力側の数値は整数でしたが, 結果は実部と虚部が浮動小数点になっている事に注意して下さい. 又, 複素数に対しても通常の算術演算子が使えます. 猶, 実部を返す命令が `real` で, 虚部を返す命令は `imag` です.

この様に, gnuplot は電卓としても十分な機能を持っています. この電卓には履歴機能に加えて, グラフ描画機能さえも付いて来るのです. 素晴らしい事ですね.

猶, 関数の定義で特定区間の描画を行う為に, 関数の値に, $1/0$ を設定すると申しましたが, `print` 文でこの式を表示させた場合には, `undefined value` となります.

5.12.16 プログラム言語としての gnuplot

gnuplot では変数が扱え, 関数も定義出来ます. その際に Octave の様な処理も行える訳ですが, ではプログラム言語としてはどうでしょうか?

gnuplot では一応, `if` 文による条件分岐を持っていますが, `surf` で見られる様な `if-goto` による原始的な `loop` 文さえも無く, 反復処理を行う為には, `reread` 命令を用いてファイルを介して行う必要があります.

先ず, `if` 文の構文を以下に示しておきますが, 基本的に C に似た構文になっており, 文の区切はセミコロンで行います.

— if 文の構文 —

```
if (<論理式>) <文1>; ... <文n>
if (<論理式>) <文1>; ... <文n>; else <文n+1>; ... ; <文n+m>;
if (<論理式1>) <文1>; ... ; <文n>; else if (<論理式2>) <文n+1>; ...
```

gnuplot の if 文は論理式や else の後に文をセミコロンで区切って並べます。但し、複雑な処理は行えません。if 文では条件式に合致する場合に、条件式の直後から else が出現する迄の文全てを処理し、そうで無い場合には、else の直後の文から行末の文迄の全ての文を実行する仕様の為です。

その為、gnuplot の if 文は文中に if 文を入れて階層的にせずに、平面的に処理を行うのが妥当です。

又、処理言語の多くが条件分岐に加えて反復処理を行う制御文を持っています。ところが、gnuplot の場合は明示的な反復処理を行う制御文を持っていません。gnuplot では再帰的に reread 命令を用いて gnuplot の文を記述したファイルの再実行による反復処理だけしか行えません。

例えば、次の様に用います。

—— ファイル tama の内容 ——

```
set isosamples 50;
if(amp<5) splot [-3:3] [-3:3] [-1:1] \
amp*sin(x*y)*exp(-sqrt(x**2+y**2));\
print "Hit any key!";pause -1;\
amp=amp+1;reread;else print "The End";
```

—— 実行の様子 ——

```
gnuplot> set pm3d at s; set hidden3d;\
> amp=1;load "tama"
Hit any key!

Hit any key!

Hit any key!

Hit any key!

The End
gnuplot>
```

この様に、ファイルを介した再帰によって反復処理が実現出来ます。猶、ファイルを介する為、反復処理の制御を行う変数の初期化は gnuplot 側で実行しておく必要があります。その為、現時点の Maxima でアニメーションを実行させる方法は、結局、maxout.gnuplot に書き込んで、それを gnuplot に引渡す方式の為、制御変数の初期化を maxout.gnuplot に書込む方法しか無いので、plot2d や plot3d を用いてスマートに処理させる事は難しくなります。

強引に実行させるのであれば、gnuplot の load 命令と save 命令を用いて、制御変数を別途臨時ファイルに保存と再読込を行う事で実現させる事が可能です。

ファイル tama

```
load "nekonekoxx";
set isosamples 50;
if(amp<5) splot [-3:3] [-3:3] [-1:1] \
amp*sin(x*y)*exp(-sqrt(x**2+y**2));\
print "Hit any key!";pause -1;\
amp=amp+1;save var "nekonekoxx";reread;else print "The End";
```

nekonekoxx の内容

```
amp = 0
```

この例では予めファイル nekonekoxx に amp の値を設定しておきます。それから,gnuplot でファイル tama を load 命令を使って読み込みます。このファイル tama では、変数 amp の値をファイル nekonekoxx から読み込む事で gnuplot 内部に取り込み、それから amp の値が5より小であれば、グラフの表示と amp に1を加えてファイル nekonekoxx に gnuplot 内部の変数を保存して reread 命令を用いて自分自身を呼出します。amp が5以上であれば、The End と表示します。

この手法を Maxima で使えなくもありませんが、plot2d や plot3d では Maxima 側で数値データとして計算した結果を gnuplot に引渡すので、アニメーションにしたい Maxima の式を gnuplot_curve_titles 側に引渡すといった随分と込み入った方法になります。その為、割り切って処理を行う必要があります。

以上で gnuplot の簡単な使い方の解説を終えます。以降は、Maxima での使い方について幾つかの実例の解説をしましょう。

5.13 plot2d 関数と plot3d 関数の活用事例

5.13.1 gnuplot_preamble の使い方

この節では plot2d 関数と plot3d 関数で gnuplot を用いた描画例を幾つか示しておきます。

猶,gnuplot は非常に機能が高いアプリケーションで色々な事が出来ますが,本質的には plot 命令と splot 命令で描画,set 命令で諸設定,そして replot で再描画を行う仕組みになっています。

その一方で,Maxima では plot2d 関数や plot3d 関数に与えられた式から数値データを生成し,諸設定は,gnuplot の命令を単純にファイルやストリームに出力する方式を採用しています。

その為,gnuplot を Maxima 側から操作する事は,gnuplot にストリームとして送り込んだり,maxout.gnuplot に書き込んだりする gnuplot の命令文を,如何に記述するかという問題に帰着されます。

ここで,gnuplot は非常に高機能のツールである為,Maxima に準備された plot_options の項目だけで全てを網羅する事は事実上不可能です。しかし,大域変数 plot_options の項目の gnuplot_preamble に様々な gnuplot の制御文が設定可能となっているので,この特性を利用しないのは幾ら何でも勿体無い事です。但し,plot_options で設定可能な項目は各々位置が決まっている為,思い通りに利用する為にはそれなりの工夫が必要になります。その為,plot_format を gnuplot にして,maxout.gnuplot の内容を確認しながら進めて行く事は非常に有効な手段です。

さて,今迄の復習になりますが,グラフの表題を mike,X 軸と Y 軸のラベルを各々 X,Y としたい場合,Maxima でどの様に入力すれば良いかを考えましょう。

ここで,gnuplot ではグラフの表題は set title で指定し,X 軸と Y 軸のラベルの設定は,各々 set xlabel と set ylabel で行います。最終的には,これらの gnuplot の命令文をセミコロンで区切り,全体を二重引用符で括ったものを gnuplot_preamble に設定します。

この処理と描画の実行を以下に示しておきます。

```
(%i5) nekoneko:"set title 'mike';set xlabel 'X';set ylabel 'Y';";
(%o5)          set title 'mike';set xlabel 'X';set ylabel 'Y';
(%i6) plot2d(sin(x)/x,[x,0,10],[gnuplot_preamble,nekoneko]);
```

この処理の結果得られるグラフを図 5.55 に示しておきます。

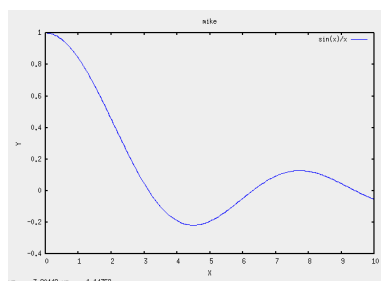


図 5.55: gnuplot_preamble で設定した結果

念の為,この例題の maxout.gnuplot の先頭部分を確認しておきましょう。

maxout.gnuplot のヘッダ部分

```

set title 'mike';set xlabel 'X';set ylabel 'Y';
plot '-' title 'sin(x)/x' with lines 3

2.441406250000E-4 0.9999999900658926
4.88281250000E-4 0.9999999602635706
---省略---
```

この様に,gnuplot_preamble の内容は gnuplot の描画命令である plot 命令や splot 命令の直前にそのまま埋込まれます。又,Maxima の式は解釈されて,数値データとして gnuplot の描画命令の直下に書込まれます。従って,plot 命令や splot 命令のオプションとして設定される曲線の表題や曲線の様子を gnuplot_preamble で設定したとしても,描画の際に,plot 命令や splot 命令のオプションが優先され,その上,式で gnuplot は描画を行わないので,sample や isosamples の設定は意味はありません。曲線の表題や曲線の様子は各々,gnuplot_curve_titles と gnuplot_curve_style に設定すべきであり,描画する曲線や曲面の細かさは予め nticks や grid で設定すべきである事が理解されるでしょう。

では,以下の様な gnuplot_curve_style への設定を行って描画させてみましょう。

```
plot2d(sin(x), [x, -1, 1], [gnuplot_curve_styles, "with impulse"])
```

この時,maxout.gnuplot の plot 命令付近は以下の様になっています。

gnuplot_curve_styles の設定

```

plot '-' title 'sin(x)' with impulse
-1.      -0.8414709848078965
-0.975   -0.8277018881672576
--- 略 ---
```

この結果は図 5.56 の様になります。

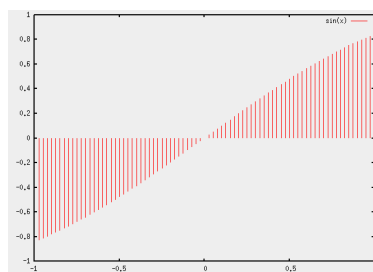


図 5.56: gnuplot_preamble で設定した結果

5.13.2 Maxima のバッチ処理

今度は,Maxima で複数のグラフを描く処理を行う為に,Maxima にバッチ処理を行わせる方法について簡単に解説しましょう.

先ず,gnuplot_preamble に記述する設定の内容は,高度なグラフ表示を行わなければならないならば,益々,膨大なものとなるでしょう.これを一々手入力する事は最初の一枚のグラフを得る為に試行錯誤する場合は仕方が無いものとしても,書式が全て決っており,その決められた書式で出力すれば済む話であるのならば,自動化を図りたいところです.

ここで Maxima にはバッチ処理機能があります.この場合,Maxima に実行させるファイルを予め準備しておきますが,このファイルの中身は通常の Maxima の式をそのまま記述したものです.このファイルを Maxima に引渡すと,Maxima がファイルの内容に従って処理を行うもので,この処理は Maxima の通常の数式や数値の処理だけではなく,画像の生成にも使えます.

手始めに簡単な入力ファイル A を記述しておきましょう.

——— ファイル A の内容 ———

```
a: (x+1)^2;
b: expand(a);
plot2d(b, [x, -1, 1]);
plot2d(b, [x, -1, 1],
[gnuplot_preamble, "set term png;set output 'test.png'"]);
```

これは非常に簡単な処理です.最初の二つの式は通常の Maxima の処理の式です.それから plot2d の処理を行なっていますが,この場合は,gnuplot でグラフを生成し,グラフ表示ウィンドウにグラフを表示してものを出力します.そして,最後の処理では,gnuplot_preamble を利用して描画を行うものです.ここで gnuplot_preamble には, `set term png` と `set output 'test.png'` といった二つの gnuplot の設定文を纏めたものを入れています.ここで,最初の命令文は gnuplot の terminal を PNG に変更するもので,次の命令文が出力先をファイル'test.png'に切替えています.即ち,最後の二つの命令文により,gnuplot はグラフを PNG データとして,ファイル test.png に書込む事になります.

ここで,gnuplot が出力可能な画像データ形式は MS-Windows か Linux 等の OS 環境によって異なります.一応,PNG であれば問題は無いかと思いますが,確認したければ,gnuplot を立ち上げて,`? term`と gnuplot に入力してみましょう.すると色々説明が表示されますが,末尾に一覧が表示される筈です.その中にある端末が利用可能なので,その一覧表にある画像データ形式を選択しましょう.

このファイルは Maxima の中から, `batch(A);` と入力する事でも処理が出来ますが,一々,Maxima を立ち上げなくても,以下の構文で Maxima にファイルを引渡す事が可能です.

——— Maxima によりバッチ処理 ———

```
maxima -b <ファイル>
maxima -batch=<ファイル>
```

猶、この処理はMS-Windowsでもコマンドプロンプト¹からの実行が可能です。因に、デスクトップに現われているMaximaのアイコンはwxMaximaのアイコンです。このwxMaximaはMaximaに被せるGUI環境に過ぎません。その為、上に示した方法はwxMaximaには使えません。この場合は、batch函数を使いますが、画像の生成を行いたければ、wxMaximaではなく、maximaを起動させた方が良いでしょう。猶、maximaをコマンドプロンプトから起動させる為には、MS-Windowの環境変数のPATHにmaximaへの経路を追加する必要があります。この設定は、コントロールパネルのシステムから詳細の中にある環境変数を選び、システムの環境変数のPathに書き込みを行います。

この場合、maximaにパスが通っている必要があります。そうでなければ、フルパスで指定する必要があります。更に、バッチファイルでtermの設定を行っている場合、Maximaに付属のgnuplotでは使えないtermが存在するので、注意が必要です。例えば、jpegは使えませんが、gifとpngは利用可能なので、これらを選択すると良いでしょう。

Maximaによるバッチ処理の計算結果は、出力先をバッチファイル内部で指定しない限り、標準出力に出力されます。重要な計算を行う場合は、`maxima -b test>test.log`の様にリダイレクトを用いると良いでしょう。

又、一々、コマンドラインに入力するのが面倒であれば、この内容を含むシェルスクリプトを記述しておくのも良いでしょう。

参考迄に`maxima -b A`を実行した様子を以下に示しておきます。

```
yokota@Zuse:~/TEST> maxima -b A
Maxima 5.10.0 http://maxima.sourceforge.net
Using Lisp CLISP 2.37 (2006-01-02)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
This is a development version of Maxima. The function bug_report()
provides bug reporting information.
(%i1) batch(A)

batching /home/yokota/TEST/A

                2
(%i2) a : (1 + x)
                2
(%o2) (x + 1)
(%i3) b : expand(a)
                2
(%o3) x + 2 x + 1
(%i4) plot2d(b, [x, - 1, 1])
(%o4)
(%i5) plot2d(b, [x, - 1, 1], [gnuplot_preamble,
set term png;set output 'test.png'])
(%o5)
```

この処理では、gnuplotのグラフも表示されています。但し、このグラフは面白味が無いので、ここでは示しません。あしからず。

次に、もう少し複雑なグラフを描いてみましょう。その為に、次の内容のファイルA2を記述しましょう。

¹所謂DOS窓

ファイル A2 の内容

```

nekoneko:"set pm3d at bs;set xrange [-2:2];\
set yrange [-2:2];set zrange [-10:20];\
set label 1 'top' at 0,0,20;\
set arrow 1 from 0,0,20 to 0,0,10;\
set arrow 1 size 5,30 filled head;\
set hidden3d;\
set output 'test1.png';set term png;";
plot3d(10*cos(x*y), [x,-2,2], [y,-2,2], [grid,50,50],
[gnuplot_preamble,nekoneko]);

```

このファイル A2 では、gnuplot_preamble に与える文字列が長くなるので、適宜、行が継続している事を示す \ を入れて改行しています。

maxima -b A2 の結果の text1.png の絵を図 5.57 に示しておきます。

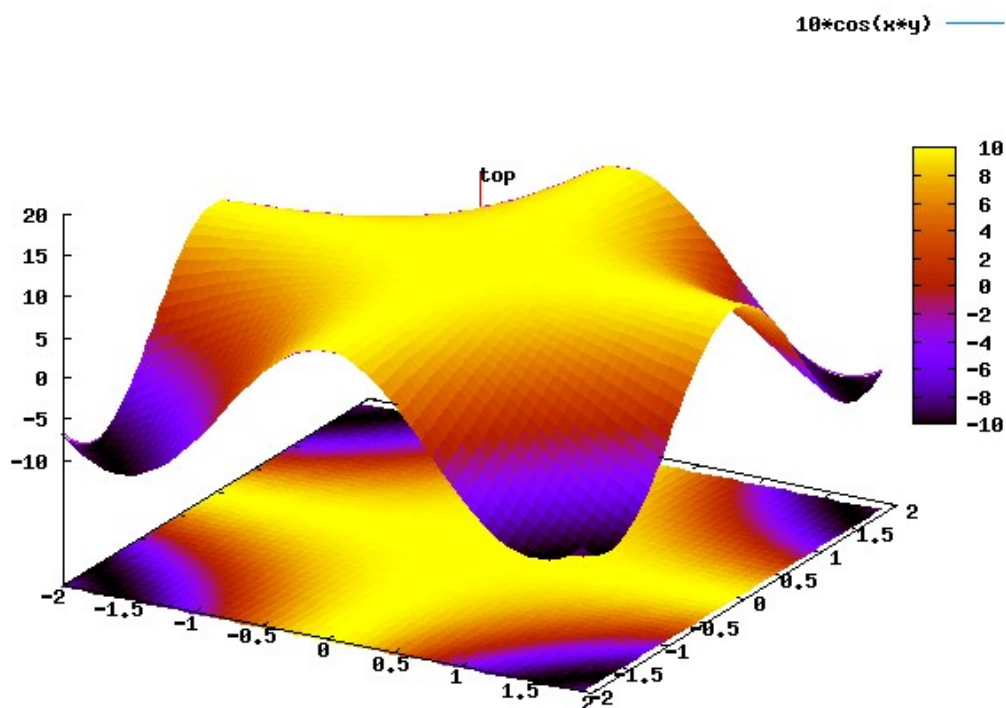


図 5.57: ファイル A2 の実行結果

ここで gnuplot で行う処理が複雑になればなる程、gnuplot_preamble の内容は必然的に長くなり

ます。画像 1 では設定 1, 画像 2 の場合に設定 2 を使う...等々とする場合, その設定を全て `gnuplot_preamble` に与える文字列として持たせるという方法は, 安易な方法ではありますが, 自動処理の事を考えた場合には, それ程, 効率的ではありません。

寧ろ, 共通の設定 1 があって, 後はケース毎に設定が追加される方が汎用性が高いでしょう。又, ケース毎に画像ファイル等の保存先のファイルを切り替える事は普通に行われる事でもあります。ところで, `gnuplot_preamble` に与えるのは一つの文字列です。そこで, Maxima で複数の文字列を纏めて一つの文字列にする操作を行えば良い事になります。Maxima には, 文字列の結合を行う処理を行う関数として, `concat` 関数があります。この関数は与えられた複数の文字列を一つの文字列に変換する関数です。

そこで今度は `concat` 関数を使ってみた例を示しましょう。

——— ファイル B1 の内容 ———

```
A1:"set pm3d at bs;";
R1:"set xrange [-10:10];\
set yrange [-10:10];set zrange [7:12];";
H1:"set hidden3d;set isosamples 50;";
O1:"set output ";
T1:"set term png;";
Ox:concat(O1,"'Fig1.png'",");");
neko:concat(A1,R1,H1,Ox,T1);
assume(x^2+y^2>0);
fxy:10-realpart(log(sqrt(x^2+y^2+1)));
plot3d(fxy,[x,-10,10],[y,-10,10],[grid,20,20],
[gnuplot_curve_titles,"title 'fxy',5*x*y/10 title 'test'"],
[gnuplot_preamble,neko]);
```

この例では, `O1` に `set output` を割当てており, これを雛形として, 出力先のファイルを `O1` に `concat` 関数で追加した文字列 `Ox` を利用しています。この方法を使えば, 条件文やバッチファイルを生成するスクリプトで, ファイル名の切替えが容易に行えます。

又, このファイルでは, `splot` に `gnuplot` の式を `gnuplot_curve_titles` を使って組込ませる様になっています。

更に, 今度は Maxima に B1 を処理させる以下に示すシェルスクリプトを記述します。

```
#!/usr/local/bin/maxima

maxima -b B1>B1.log
convert Fig1.png Batch2.eps
display Fig1.png
less B1.log
```

このシェルスクリプトでは, 最初に Maxima でファイル B1 の処理を実行します。その結果, 画像

ファイル Fig1.png が生成されますが、次に convert 命令を使って、この Fig1.png を Batch2.eps に変換して、Fig1.png を display 命令を使って閲覧します。そして最後にバッチ処理の記録ファイルの B1.log を見るというものです。

このバッチ処理で生成された画像ファイルを図 5.58 に示しておきます。

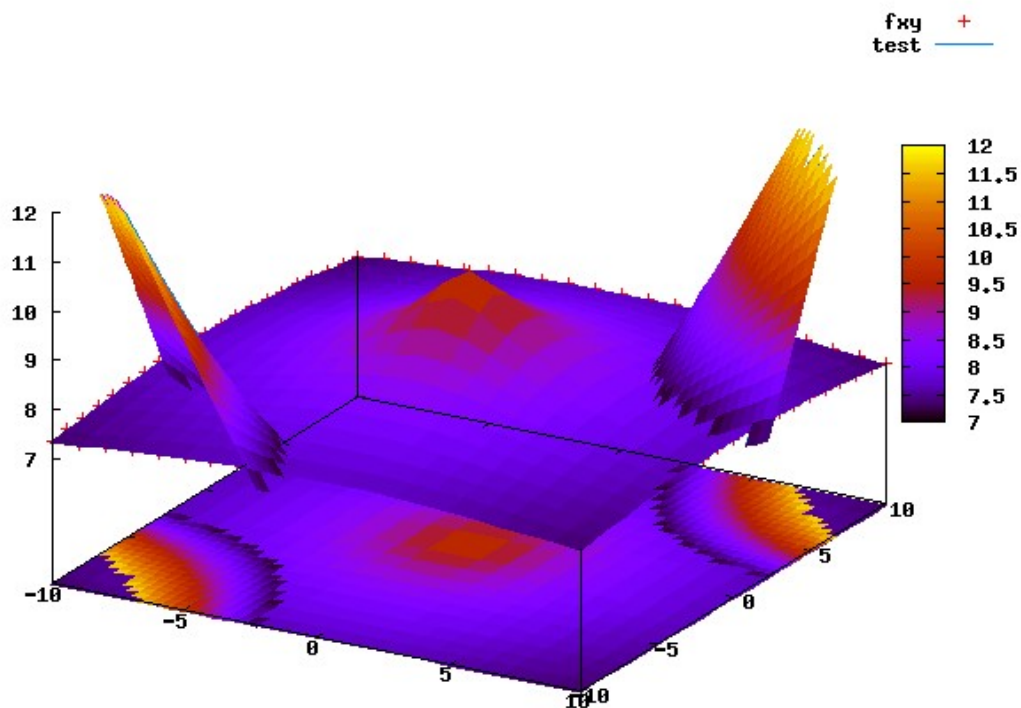


図 5.58: ファイル B1 の実行結果

Maxima のバッチ処理と gnuplot_preamble が揃えば、後は,awk 等の適切な言語があれば十分複雑な処理が行えるでしょう。

5.14 draw パッケージ

5.14.1 draw パッケージの概要

Maxima では plot2d と plot3d 関数を主要なグラフ表示関数としていましたが, gnuplot を利用する関数として, draw パッケージが新たに追加されました. この draw パッケージは従来の plot2d 関数や plot3d 関数とは異なり, 外部アプリケーションとして利用可能なものは gnuplot のみ, それも 4.2 以降のものに対応しています.

draw パッケージには描画関数として draw 関数, draw 関数を簡易にした draw2d 関数と draw3d 関数があります. これらの関数は従来の plot2d 関数や plot3d 関数と違い, 式ではなく式等から構成された対象を表示する関数です.

draw パッケージで用いられる対象は, 表示する式の型によって様々な文を用いて構築されるものです. そして, これらの対象の表示, 例えば, 線分を実線, 或いは点線で表示するといった性質を変更させる為の指定, 即ち, 属性が指定可能となっています.

draw2d 関数と draw3d 関数は各々 2 次元と 3 次元グラフを描く為の関数ですが, これらの関数の実体は draw 関数です. この draw 関数を用いる事で, 一枚のウィンドウに複数の 2 次元や 3 次元の対象のグラフを同時に表示させる事が可能となっています.

draw パッケージではグラフの出力は gnuplot に任せますが, 基本的に, Lisp のストリームを用いる事で gnuplot の制御を行う方式を採用しています. その為, 中間ファイルは plot2d 関数や plot3d 関数と同様に, MS-Windows 以外は maxout.gnuplot_pipes になります. 猶, MS-Windows 環境であれば maxout.gnuplot となり, 名前から判る様に, Maxima 側からグラフデータだけではなく, gnuplot の全ての設定文や描画命令文を書出したファイルで, ストリームを用いない方式になっています. その為, MS-Windows 側から gnuplot を操作する事は出来ません.

5.14.2 対象と属性について

draw パッケージで実際の描画を行う関数である draw, draw2d, draw3d では, plot2d 関数の様に式をそのまま描画するのではなく, 式を対象として定義し, その対象を対象を描画する方式となっています.

ここでは従来の描画関数の plot2d や plot3d と対応のつく draw2d 関数や draw3d 関数を使って簡単な例題からこれらの関数の特徴を眺めてみましょう.

先ず, $\frac{\sin(x)}{x}$ を描画する時の draw2d の最も単純な構文を以下に示しておきましょう.

```
draw2d(explicit(sin(x)/x,x,-10,10));
```

この出力を以下の図 5.59 に示します.

この例では, draw2d の中に explicit 文の中に式が含まれています. この様に, 描画するものを含む文の事を対象と呼びます. この explicit 文には構文があり, 描画する式が $\frac{\sin(x)}{x}$ で, その式の変数が x で, この変数 x の定義域が $[-10, 10]$ である事を宣言しています. これだけだと, plot2d 関数に explicit を加えて複雑にした様に見えませんが,

では次の例を考えましょう.

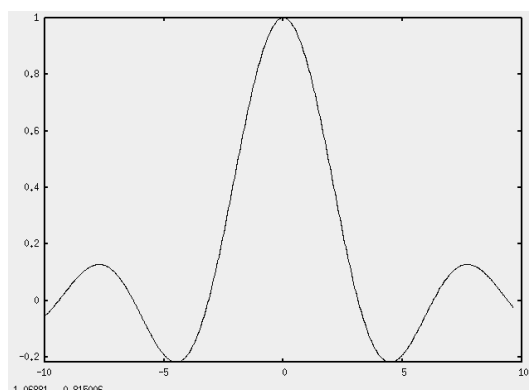


図 5.59: draw2d による単純なグラフ表示

```
draw2d(xrange=[-20,20],explicit(sin(x)/x,x,-10,10));
```

この結果を今度は図 5.60 に示しておきます。

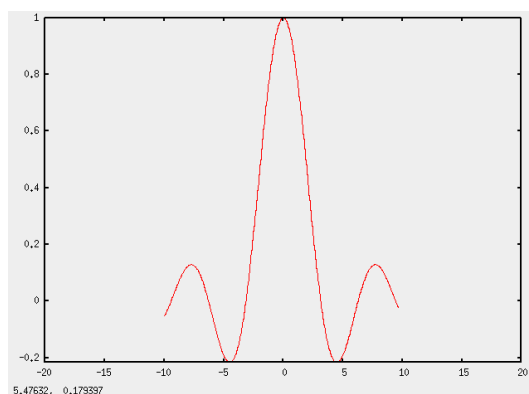


図 5.60: xrange による X 軸側の表示領域設定

この例では、先程の `explicit` 文に加えて、`xrange` というものがありますね。こちらの `xrange` は `explicit` とは構文が異なり、`xrange = <値>` の書式になっています。このような構文を持つ文をここでは属性と呼びます。属性は演算子 `=` の左辺に属性名を置き、右辺にその値を指定します。

ここでの `xrange` 属性はグラフの枠に対し、その X 軸側の領域を定める属性です。draw2d 関数や draw3d 関数を利用する場合には、`xrange` 属性は大域的な属性の様に思えますが、draw 関数を用いると升目の様に複数のグラフを表示可能で、その様に表示したグラフの内の一つのグラフに対し、その枠に影響を及ぼす為、局所的な属性になります。

ところで、`explicit` で定めたグラフの定義域が $[-10, 10]$ なのに対して、`xrange` が $[-20, 20]$ となっている事に注意して下さい。その結果、図 5.60 では、式のグラフは X 軸側が $[-10, 10]$ を定義域とする値域のみしか描かれていませんが、グラフの枠は X 軸側が $[-20, 20]$ に設定されています。

猶,xrange があるので,二次元グラフの場合は yrange,三次元の場合は zrange も当然ありますが,例に示す様に,これらのグラフの枠の属性は Maxima 側に任せる自動設定にしても構いません.

上述の例では,対象を定義する文や属性の式を draw2d 関数内部で記述していましたが,対象や属性を一々描画関数内部で色々設定してやる必要はありません. 実際,以下の様な方法でグラフを描画させる事も可能です.

```
(%i4) neko:explicit(sin(x)/x,x,-10,10);
          sin(x)
(%o4)      explicit(-----, x, - 10, 10)
          x
(%i5) X1:xrange=[-20,20];
(%o5)      xrange = [- 20, 20]
(%i6) Y1:yrange=[-0.5,1.5];
(%o6)      yrange = [- 0.5, 1.5]
(%i7) draw2d(X1,neko,Y1);
(%o7)      [gr2d(explicit)]
```

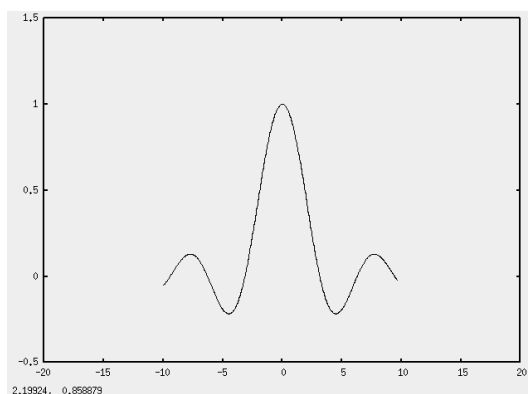


図 5.61: draw2d の表示例

この例では,explicit といった対象に対し,xrange や yrange といった属性を Maxima 上で設定し,それらを纏めて,draw2d 関数を用いて表示したものです. この方法だと,複数の対象を描く事が容易に行える事が理解出来るかと思えます.

さて,draw パッケージの属性には,全体に影響を与える大域的な属性,グラフの枠や利用者定義の特定の対象に影響を及ぼす属性の二通りの属性があります. ここで,大域的な属性に関しては,draw 関数,draw2d 関数や draw3d 関数の内部で,他の対象や属性との順番は全く問題になりません. 次に,グラフの枠といった対象で,draw パッケージの描画関数で明示的に生成しない対象の属性の順番は比較的自由です. ここで比較的自由であるとは,draw2d 関数と draw3d 関数では自由に置けますが,draw 関数の場合には緩い制約が付く為です. これに対して,曲線や曲面等の実際に描画され

る対象に影響を及ぼす属性は必ず対象の前に置かなければなりません。即ち、描画する対象とその属性は一つのグループとして扱う必要があります。これは draw パッケージの描画関数で、前にある属性を解釈して gnuplot の設定文に翻訳し、対象が現われた時点で、その対象を描画する命令と設定文を書込む為です。この方式を採用した理由としては、gnuplot の構文上の特性による影響が大きいと言えるでしょう。

では、属性の対象に対する修飾の様子を以下の例で観察してみましょう。

```
(%i28) nekoneko:points([1,2,3],[1,2,3]);
(%o28)          points([1, 2, 3], [1, 2, 3])
(%i29) draw2d(xrange=[-1,6],
              yrange=[-1,6],
              nekoneko,
              point_size=3,
              point_type=diamant,
              points_joined=true,
              line_type=dots,
              color=red,
              nekoneko);
(%o29)          [gr2d(points, points)]
```

ここで用いられている対象 `points` は平面内の点列を定義する対象で、`point_size`、`point_type`、`points_joined`、`line_type` や `color` といった属性を持っています。これらの属性に関しては後の小節で詳細を述べますが、各々、点の大きさ、点の型、点列の繋ぎ型と色に関連する属性です。

さて、上記の入力から次の絵が得られます。

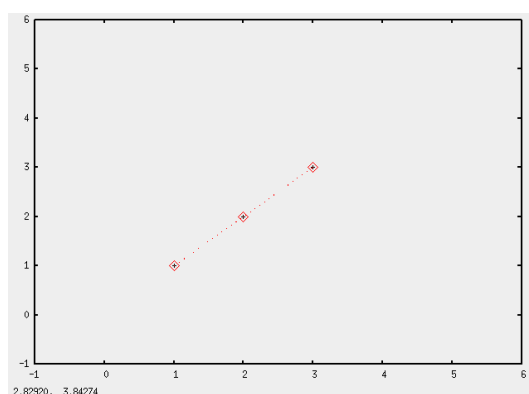


図 5.62: 点と属性の関係

この図に示す様に、最初の点 `nekoneko` は後に続く対象 `points` の属性の影響を受けていませんね。ところが、後の `nekoneko` は点の大きさが 3、点の型が `diamant`、各点を点線で結んでおり、色も赤になっています。この様に、最初の対象 `nekoneko` はデフォルトの属性のままですが、次の対象 `nekoneko` には、最初の `nekoneko` 以降に置かれた属性になっています。

5.14.3 draw 関数と draw2d 関数, draw3d 関数との関係

最初に draw2d 関数と draw3d 関数の実体は draw 関数であるとお話ししましたが、ここではもう少し詳細を述べる事にしましょう。

draw2d 関数と draw3d 関数は各々、2次元と3次元の対象を専門に描く関数です。では、2次元の対象と3次元の対象を画面を複数に分割して描く場合はどうすれば良いのでしょうか？残念ながら draw2d や draw3d では出来ません。しかし、draw 関数なら可能なのです。この理由は非常に単純な事です。実は、draw2d と draw3d は draw 関数に gr2d 文や gr3d 文を合成した書式になっています。

この様子を以下に纏めておきましょう。

draw2d, draw3d と draw との関係

$$\begin{aligned} \text{draw2d}(\text{対象}_1, \dots, \text{対象}_n) &\Rightarrow \text{draw}(\text{gr2d}(\text{対象}_1, \dots, \text{対象}_n)) \\ \text{draw3d}(\text{対象}_1, \dots, \text{対象}_n) &\Rightarrow \text{draw}(\text{gr3d}(\text{対象}_1, \dots, \text{対象}_n)) \end{aligned}$$

ここで gr2d と gr3d は各々2次元と3次元の対象を纏めたものになります。先程の例で、explicit 文や points 文を持つ対象を描かせた時に、[gr2d(explicit)] や [gr2d(points, points)] と Maxima が返していましたが、その gr2d です。draw2d の場合は、gr2d 文、draw3d の場合は gr3d 文で対象を処理した事を示している訳です。

具体的には、Maxima の内部では draw2d 関数や draw3d 関数の引数に各々(\$gr2d) や (\$gr3d) を Lisp の cons 関数で結合したものを draw 関数に引渡しています。そして、draw 関数内部では、その付加した情報を基に、make-scene-2d 関数や make-scene-3d 関数に引渡して個別のグラフ処理を行っています。

この draw2d 関数や draw3d 関数のみを用いていると、大域的な属性と局所的な属性の扱いが少し見え難くなります。例えば、draw2d 関数や draw3d 関数で xrange の様なグラフの枠を定める属性は、他の対象や属性とは独立して好きな場所に置けます。これは、draw2d では gr2d 文、draw3d で gr3d 文を用いて一つのグラフの枠が定義されているからで、その枠の内部で枠の設定を行うのは枠の中なら何処でも構わないという事に過ぎません。

ところが、draw2d や draw3d 関数では複数の gr2d 文や gr3d 文を扱う事は出来ません。これらの複数の文を処理する事が可能な関数が draw 関数で、この draw 関数を用いれば、xrange の様に一見大域的な属性が、実際は局所的な属性である事が明瞭になります。

5.14.4 draw 関数

さて、draw2d や draw3d の実体が draw 関数であると述べましたが、ここで、その詳細を述べます。

draw 関数は複数の gr2d 文や gr3d 文で構成された文の列を左から順番に表示する事が可能な関数です。この draw 関数では対象が2次元であるか3次元であるかに応じて、gr2d 文や gr3d 文を用います。但し、同じ次元の対象を一つの gr2d 文や gr3d 文で記述するか、複数の gr2d 文や gr3d 文で記述するかで結果が異なります。

最初に、正弦関数と放物線を一つの gr2d 文で記述した場合の様子を以下に示しておきます。

```
(%i6) draw(gr2d(explicit(sin(x), x, -10, 10),
```

```
explicit(x^2+1,x,-2,2));
(%o6) [gr2d(explicit, explicit)]
```

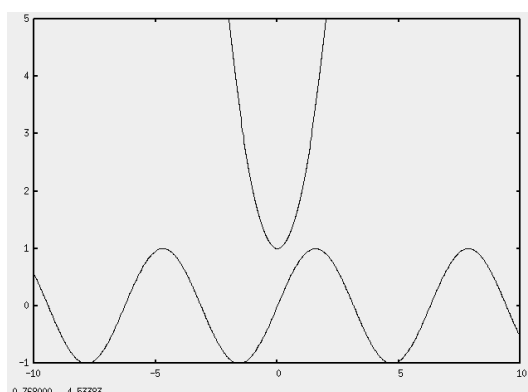


図 5.63: 一つの gr2d 文の場合

図 5.63 に示す様に、一つのグラフの中に正弦函数と放物線が記述されていますね。又、draw 函数の応答からも、一つの gr2d 文の中に explicit が二つある事が判りますね。

では次に、gr2d 文で正弦函数と放物線を別々にした場合を以下に示しましょう。

```
(%i7) draw(gr2d(explicit(sin(x),x,-10,10)),
           gr2d(explicit(x^2+1,x,-2,2)));
(%o7) [gr2d(explicit), gr2d(explicit)]
```

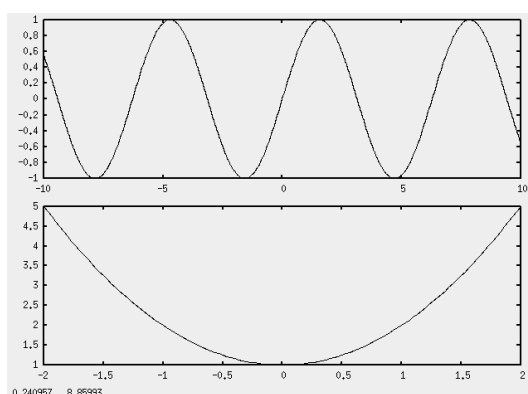


図 5.64: 二つの gr2d 文の場合

図 5.64 から判る様に、今度は別々にグラフが表示されていますね。更に、draw 文の応答からも、gr2d 文が二つある事が判りますね。ここで、最初の正弦函数のグラフが上で、放物線のグラフが下に描かれていますね。draw 函数では gr2d 文や gr3d 文で構成される列を左から順番に描きます。通常の

表示は一行にグラフを表示しますが、これを複数の列で表示させる事も可能です。この場合、大域的属性の `columns` の値を変更すれば可能です。

では早速以下の例で解説しましょう。

```
(%i15) draw(columns=1,gr2d(explicit(sin(x),x,-10,10)),
           gr3d(explicit(x*y,x,-2,2,y,-2,2)),
           gr2d(explicit(x^2-x+1,x,-2,2)));
(%o15) [gr2d(explicit), gr3d(explicit), gr2d(explicit)]
```

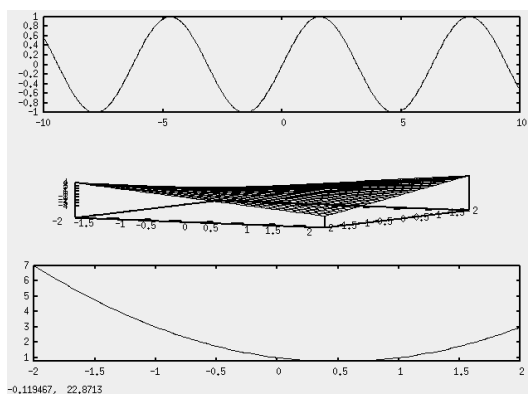


図 5.65: `columns=1` の場合

ここでの例では、最初が正弦函数、次が xy 、最後が $x^2 - x + 1$ のグラフを描かせていますが、`columns=1` の為、図 5.65 では正弦函数から順番に頭から一行に表示されていますね。では、`columns` を 3 にするとどうなるでしょうか？

```
(%i16) draw(columns=3,gr2d(explicit(sin(x),x,-10,10)),
           gr3d(explicit(x*y,x,-2,2,y,-2,2)),
           gr2d(explicit(x^2-x+1,x,-2,2)));
(%o16) [gr2d(explicit), gr3d(explicit), gr2d(explicit)]
```

図 5.66 で示す様に、3 列でグラフが左から順に表示されている事が判りますね。最後に、`columns=2` とした場合の例を示しておきましょう。

```
(%i17) draw(columns=2,gr2d(explicit(sin(x),x,-10,10)),
           gr3d(explicit(x*y,x,-2,2,y,-2,2)),
           gr2d(explicit(x^2-x+1,x,-2,2)));
(%o17) [gr2d(explicit), gr3d(explicit), gr2d(explicit)]
```

この様に、対象が 3 個の為、最初の 2 個が同じ行に表示されて、3 番目の対象が一段下に表示されていますね。猶、`mouse` が `on` に来るのは、一番最後に描画される対象のみです。

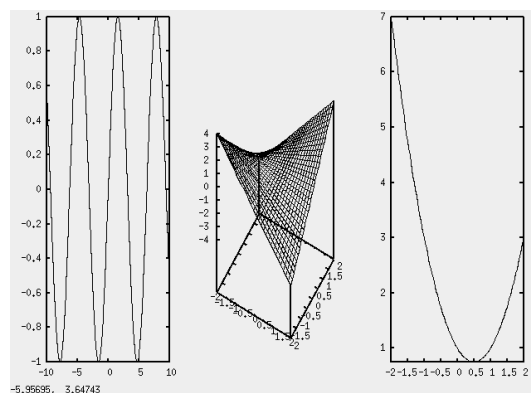


図 5.66: columns=3 の場合

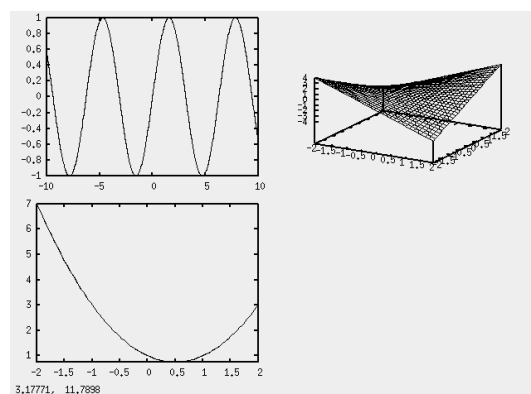


図 5.67: columns=2 の場合

この columns 文は gr2d や gr3d の外に、しかも、先頭に置いています。実際は何処にでも置ける文です。gr2d 文や gr3d 文の中に入れる事も可能です。この様に、draw 関数内部で何処にでも置ける属性の事を大域的な属性と呼びます。これに対し、個々の対象に関連する属性は、その対象が含まれている gr2d 文や gr3d 文の外に置く事は出来ません。例えば、xrange を draw 関数の中で columns の様に扱えません。

```
(%i6) draw(xrange=[-1,1],gr2d(explicit(sin(x)/x,x,-0.5,0.5)));
Unknown global option xrange
-- an error. To debug this try debugmode(true);
(%i7) draw(gr2d(explicit(sin(x)/x,x,-0.5,0.5)),xrange=[-1,1]);
Unknown global option xrange
-- an error. To debug this try debugmode(true);
```

この例で判る様に、属性 xrange は大域的な属性ではありません。この様に、局所的な属性は draw 関数で記述した際に、gr2d 文や gr3d 文の外に出る事が出来ません。

さて、以降の小節では draw パッケージで用意されている大域的な属性について述べましょう。

5.14.5 大域的な属性について

draw パッケージには以下の大域的な属性が用意されています。

大域的属性			
columns	正整数値	1	グラフの列数
terminal			screen の値
pic_width	正整数値	640	画像ファイルの幅 (画素単位)
pic_height	正整数値	480	画像ファイルの高さ (画素単位)
eps_width	正整数値	12	画像ファイルの幅
eps_height	正整数値	8	画像ファイルの高さ
file_name	文字列	maxima_out	画像の出力ファイル名

columns は draw 関数で複数の gr2d 文や gr3d 文の列を描く時に、表示方法を指定するものです。通常、columns の初期値は 1 の為、columns に値を指定しなければ、draw 関数は gr2d 文と gr3d 文の列を左から順番に縦一列に表示します。より一般的に、columns を正整数値 n 、gr2d と gr3d の列を a_1, a_2, \dots, a_m とすると、 a_1, \dots, a_n を左から順番にグラフの 1 行目に、 a_{n+1}, \dots, a_{2n} をグラフの 2 行目に描画してゆきます。猶、グラフの行数は $\frac{m}{n}$ を越える最小の整数が相当します。

猶、mouse が on になっているのは gr2d と gr3d の列を a_1, a_2, \dots, a_m とする場合、 a_m のグラフのみです。その為、グラフの座標の読取、グラフの拡大・縮小、そして、グラフの回転は a_m のグラフに対してのみ実行され、グラフの拡大・縮小、回転によってグラフの再描画が行われた場合には、 a_m のグラフのみが再表示されて、他のグラフが消えてしまう事に注意しましょう。

terminal は gnuplot の terminal に対応する属性で、グラフ出力先の端末を指定する属性です。UNIX 環境でその初期値は通常 x11 となっていますが、この変数は計算機環境で著しく異なります。猶、JPEG, PNG, GIF

や PostScript といった様々な画像形式でグラフを保存する場合、この変数を目的の画像形式に設定します。例えば、PNG 形式の画像ファイルにグラフを出力したければ、この terminal を `terminal=png` で PNG に指定します。この terminal の値は個々の環境の gnuplot で利用可能な端末に限定されます。MS-Windows に同梱されている wgnuplot では JPEG が扱えないので、MS-Word とかに利用したければ、PNG を用いると良いでしょう。又、その他にどのような値が設定可能かを知りたいければ、gnuplot を立ち上げ、命令入力行に `? terminal` と入力すれば、terminal に関するオンラインマニュアルの内容が表示され、このオンラインマニュアルの末尾に利用可能な端末の一覧が表示されます。

例として、図 5.68 に `? terminal` と入力した時に表示されるオンラインマニュアルの末尾を示しておきます。

```
This document may describe terminal types that are not available to you
because they were not configured or installed on your system. To see a list of
terminals available on a particular gnuplot installation, type 'set terminal'
with no modifiers.

Subtopics available for terminal:
aed512          aed767          aifm             bitgraph
cgm            corel             dumb            dxf
sepic          eaf              entex           epslatex
epson-180dpi   eposn-60dpi      eposn-1x800     fig
gif            gp1c             hp2623a         hp2648
hp500c         hpdl             hpdl            hp1jii
hpgl           lwagen          jpeg            kc-tek40xx
kw-tek40xx    listex          mf              aif
mp            nec-cp6         okidata         pbw
pcl5           png             pop             postscript
pslatex       pstex           pstricks        push
qms           regis           se1anar         stanc
SVG           tandy-60dpi      tek40xx         tek410x
texdraw       tgif            tkcanvas        tpic
vttek         x11             xlib
```

lines 4-26/26 (END)

図 5.68: `? terminal` の結果

gnuplot の terminal にどのような値が設定されているかを知りたいければ、今度は gnuplot で `show terminal` と入力します。すると terminal の値が表示されます。例えば、Linux 上の gnuplot で `show terminal` を入力した様子を示しておきましょう。

```
gnuplot> show terminal

terminal type is x11

gnuplot>
```

因に MS-Windows 版に付属の wgnuplot で `show terminal` を実行すると、結果として `windows color nonenhanced` が返されます。

`pic_width`, `pic_height` は EPS ファイル以外の書式でグラフをファイル出力する場合に、その画像の大きさを指定する属性値です。`pic_width` が画像の幅、`pic_height` が画像の縦方向の大きさを画素単位で指定します。

`eps_width` と `eps_height` は画像形式が PostScript や EPS の場合に設定します。

`file_name` は画像ファイルとして保存する場合、そのデータの保存先の名称です。猶、修飾子は Maxima 側でつけます。例えば、terminal の値を PNG とし、`file_name` を `tama` にしていると、画像ファイル名は `tama.png` になります。

以上が draw パッケージに含まれる大域変数です。残りの属性は、グラフの枠に関連したものや、描画される対象の見え方を指定するものになります。次の小節では局所的な属性でグラフの枠に関連するものを解説しましょう。

5.14.6 グラフの枠に関連する属性

局所的な属性の特徴としては、draw 関数で描く際に、gr2d 文や gr3d 文の外に置く事が出来ない点が挙げられます。その中で、グラフの枠に関連する属性は gr2d 文や gr3d 文の中で、自由に置く事が出来る為、大域的な属性の様に見えるものです。しかし、その影響はその属性が置かれている gr2d や gr3d 内部を越える事はありません。

先ず、グラフの枠に関連する属性で大きさと表示方法に関連する属性を以下に示しておきます。

グラフの枠の大きさと表示に関連する属性

xrange	二成分の実数リスト	自動	グラフの X 軸方向の枠を指定
yrange	二成分の実数リスト	自動	グラフの Y 軸方向の枠を指定
zrange	二成分の実数リスト	自動	グラフの Z 軸方向の枠を指定
axis_top	論理値	true	上側のグラフの枠表示を指定
axis_bottom	論理値	true	底のグラフの枠表示を指定
axis_right	論理値	true	右側のグラフの枠表示を指定
axis_left	論理値	true	左側のグラフの枠表示を指定
axis_3d	論理値	true	3 次元グラフの枠表示を指定

属性 xrange,yrange,zrange に関しては既に述べている様に、表示する領域、即ち枠の大きさを指定するものです。これらが無指定にしておくと、draw パッケージの関数側で自動的にその表示対象の大きさから指定を行います。例えば、対象 explicit や対象 explicit3d の場合、これらを定義する explicit から変数の定義域や値域を自動的に枠の領域として設定します。

属性 axis_top,axis_bottom,axis_right,axis_left は二次元グラフでの上下、左右の枠の表示を行うかどうかを指定する属性です。例えば、axis_top が false の場合、グラフの上下、左右を囲む枠の上側が表示されません。同様に、axis_bottom が false であれば下側、以降、axis_right が false なら右側、axis_left が false なら左側の枠が表示されません。

これと同様に、axis_3d が false の場合、三次元のグラフで枠が非表示になります。

グラフの目盛に関連する属性もあります。これらの属性では目盛の有無や目盛を対数表示にするかどうか、そして、グラフ全体に網目を入れるかが指定出来ます。

— グラフの軸の目盛に関する属性 —

xtics	論理値	true	X 軸に目盛を入れる
ytics	論理値	true	Y 軸に目盛を入れる
ztics	論理値	true	Z 軸に目盛を入れる
logx	論理値	false	X 軸方向を対数目盛に設定
logy	論理値	false	Y 軸方向を対数目盛に設定
logz	論理値	false	Z 軸方向を対数目盛に設定
grid	論理値	false	網目の描画

属性 `xtics`, `ytics`, `ztics` は各々、グラフの枠に目盛を入れるかどうかを指定する属性です。true の場合に目盛を入れます。猶、各軸の目盛の属性は `axis.top` 等の枠の表示に関連する属性から独立しています。即ち、枠を非表示にしている場合、軸目盛の表示が true であれば、その軸の目盛は表示されます。

又、目盛の間隔を対数にする事も可能です。この場合、属性 `logx`, `logy`, `logz` の何れか必要な軸の属性を true にします。

属性 `grid` は 2 次元グラフの網目表示を指定する為の属性です。grid に true を指定すると、二次元グラフで目盛に対応する様に網目の表示が行われます。

draw パッケージのグラフの表題や各軸のラベルに関連する属性を以下に示します。

— グラフの表題等に関連する属性 —

title	文字列	"""	グラフの表題を指定
xlabel	文字列	"""	グラフの X 軸のラベルを指定
ylabel	文字列	"""	グラフの Y 軸のラベルを指定
zlabel	文字列	"""	グラフの Z 軸のラベルを指定

ここでの属性は論理値ではなく、実際に表示する文字列になります。属性 `title` にグラフの表題を設定します。これは単純に `gnuplot` の `set title` 文に `title` の内容を引渡すだけです。

次の属性 `xlabel`, `ylabel` と `zlabel` は X, Y, Z 軸に設定するラベルを指定する属性です。これらも `gnuplot` の `set xlabel` 文等に各々の属性を引渡すだけです。

視点の変更やマウスで検出した座標値の保存に関連する属性もあります。

— 視点の変更, 検出した座標値の保存に関連する属性 —

rot_vertical	0 から 180 の間の数	60	Z 軸回りの回転角度
rot_horizontal	0 から 360 の間の数	30	水平の回転角度
xy_file	文字列	"""	データ保存用のファイル名を指定

属性 `rot_vertical` や属性 `rot_horizontal` は `gnuplot` の `set view` 文に対応する属性です。属性 `rot_vertical` は 0 から 180 の間の実数を設定する事が可能です。同様に、属性 `rot_horizontal` には 0 から 360 の間の実数が設定可能です。

属性 `xy_file` はマウスによる座標の検出を行った結果を保存する為のファイルを指定します。このマウスによる座標の検出は、グラフが表示されているウィンドウ上の任意の点をマウスの左ボタンで押した時にウィンドウの左下に座標が表示されています。この値はキーボードから `[x]` を押す事

で,xy_files に指定したファイルの保存されます。

以上で, グラフの枠に関連する属性の説明を終えます。その他の局所的な属性に関しては, 対象に密接に関連する為, 対象の解説と共に, その属性の解説も行います。

5.14.7 対象について

draw パッケージの描画関数で実際に表示されるものが対象です。draw パッケージの対象には様々なものがありますが, それは Maxima で計算した式は勿論の事として, その他に, 多角形や楕円もあります。更に, これらの対象の色や塗り潰しといった, 各対象固有の属性があります。

ところで, 対象は各々固有の構文を持った文で生成されます。ここでは文毎に生成可能な対象を纏めて解説しましょう。

explicit 文

explicit 文で定義可能な対象は, 2次元グラフの対象 explicit と 3次元グラフの対象 explicit3d です。

対象 explicit は $y = f(x)$ の型の一変数の式のグラフを表現する対象です。この対象は以下に示す構文で記述されます。

対象 explicit を表現する構文

```
explicit(<式>, <変数>, <最小値>, <最大値>)
```

この場合, 第一引数に式, 第二引数に式の変数, それから, その変数の定義域を表現する最小値と最大値を各々第三引数と第四引数に指定します。

対象 explicit には以下の属性があります。

対象 explicit の属性

adapt_depth	正整数値	10	描画助変数
nticks	正整数値	30	点数
line_width	正整数値	1	曲線の太さ
line_type	solid, 又は dots	solid	曲線の型
color	文字列	black	曲線の色
filled_func	論理値	false	塗り潰しの指定の有無
fill_color	文字列	red	
key	文字列		凡例 (legend) を記述

属性 adaptive_depth は explicit のみで用いられる属性で, Common Lisp の関数 adaptive-plot で利用される属性です。ここでの値は adaptive-plot で描画を行う際の曲線分割の最大数で, 初期値は 10 です。

属性 nticks は曲線上の点数です。曲線上の点は explicit の X 軸上の最大値と最小値を用いて, 始点を X 軸の最小値から間隔 $\delta = \frac{\text{最大値} - \text{最小値}}{\text{nticks} - 1}$ で等間隔に配置され, これらの点を線分で結ぶ事でグ

ラフが得られます。従って、複雑な曲線になればなる程、この `nticks` の値を大きくする必要がありますが、直線に違い曲線であれば小さな値にしても構いません。

属性 `line_width` は直線の太さの属性になります。初期値として 1 が設定されており、大きな整数値を設定すると曲線が太くなります。

属性 `line_types` は曲線の型を属性です。この属性が取り得る値は `solid` と `dots` の二つだけです。属性 `line_types` が `solid` の場合、`draw` パッケージの描画関数は実線を引きますが、`dots` の場合は破線になります。猶、`gnuplot` には、`solid` であれば 1、`dots` であれば 0 を引渡しています。

属性 `color` は曲線の色属性です。初期値として `black`(黒) が指定されています。猶、曲線と X 軸で囲まれた領域を塗り潰す場合には、次の属性 `fill_color` や属性 `filled_func` の設定を行う必要があります。

属性 `fill_color` は曲線と X 軸で挟まれた領域を塗り潰す色の属性です。初期値として `red`(赤) が指定されていますが、実際に、対象によって塗り潰しを行う為には、別の属性 `filled_color` を `true` に設定しておく必要があります。

属性 `filled_color` は塗り潰しの指定を行う属性です。属性 `filled_color` を `true` に設定した時に、属性 `fill_color` で指定した色で塗り潰しが行われます。猶、赤で塗り潰す場合には、属性 `filled_color` の指定をしなくても構いません。

属性 `key` は `gnuplot` の凡例 (`legend`) を設定する為の属性です。要するに、`gnuplot` のグラフで右上側にどのグラフが何のグラフに対応するかが描かれる事がありますが、これを設定する属性です。例えば、以下の様に入力した結果を図 5.69 に示しておきます。

```
(%i8) draw2d(line_type=solid,
             key="sin(x)/x",
             explicit(sin(x)/x,x,-10,10),
             line_type=dots,
             key="x^2-1",
             explicit(x^2-1,x,-3,3));
(%o8) [gr2d(explicit, explicit)]
```

この図 5.69 の右上に、どれが $\frac{\sin(x)}{x}$ のグラフで、どれが $x^2 - 1$ のグラフであるかを示す凡例が現われていますね。`key` はここの文字列を指定する属性になります。

対象 `explicit3d` は $z = f(x, y)$ の様な 2 変数の実数値関数として表現可能な式のグラフを表現します。猶、この時の式の値域は、絶対値が `1.75555970201398d+305` を越えない領域でなければなりません。

——— 対象 `explicit3d` を生成する構文 ———

```
explicit(<式>, <変数1>, <変数1の最小値>, <変数1の最大値>, <変数2>, <変数2の最小値>, <変数2の最大値>)
```

`explicit3d` は基本的に対象 `explicit` を定義する `explicit` 文に第二変数に関連する引数が追加されただけです。又、この対象 `explicit3d` の属性には以下のものがあります。

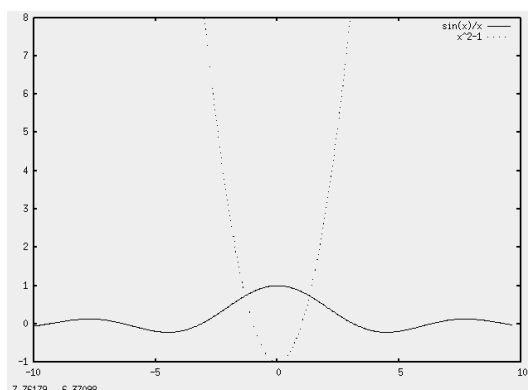


図 5.69: key による凡例

対象 explicit3d の属性

xu_grid	正整数値	30	X 軸方向の点数
yv_grid	正整数値	30	Z 軸方向の点数
contour	文字列		等高線の表示
contour_levels	正整数値	5	等高線の数
enhanced3d	文字列		
line_width	正整数値	1	曲線の太さ
line_type	solid 又は dots	solid	曲線の型
color	文字列	black	曲線色
key	文字列		汎用 (legend) を記述

属性 xu_grid と属性 yv_grid は対象 explicit の属性 nticks に相当する属性で、曲面の肌理の細やかさをここで指定する事になります。

属性 contour は等高線を描く位置を定める属性です。属性 contour が取り得る値は, surface, base, both と map です。ここで、これらの値と gnuplot の命令との対応を以下に示しておきましょう。

属性 contour の値と gnuplot の命令文との関係

surface	set contour surface;set cntrparam levels contour_levels
base	set contour base;set cntrparam levels contour_levels
both	set contour both;set cntrparam levels contour_levels
map	set contour base;unset surface; set cntrparam levels contour_levels

猶、対象 explicit の属性と同名のもは、対象 explicit の同名の属性と同じです。

implicit 文

implicit 文は対象 implicit の生成で用います。この対象 implicit は $x^2 + y^2 - 1 = 0$ の様な 2 変

数の陰関数のグラフの表示に用います。

対象 implicit の構文

```
implicit(⟨式⟩, ⟨変数1⟩, ⟨変数1の最小値⟩, ⟨変数1の最大値⟩, ⟨変数2⟩, ⟨変数2の最小値⟩, ⟨変数2の最大値⟩)
```

ここで、対象 implicit の属性を以下に示しておきます。基本的に、属性 ip_grid と属性 ip_grid_in 以外の属性は対象 explicit の同名の属性と同一です。

implicit の属性

ip_grid	二成分の正整数リスト	[50,50]	点数
ip_grid_in	二成分の正整数リスト	[5,5]	
line_width	正整数	1	曲線の太さ
line_type	solid, 又は dots	solid	曲線の型
color	文字列	black	曲線色
key	文字列		凡例に用いる文字列

属性 ip_grid は対象 implicit で一次抽出で用いる網目を定め、属性 ip_grid_in は二次抽出で用いる網目を定めます。

parametric 文

parametric 文は対象 parametric の生成で用います。ここで対象 parametric は媒介変数表示の平面曲線を描く際に用いる対象です。

対象 parametric の構文

```
parametric(⟨式x⟩, ⟨式y⟩, ⟨媒介変数⟩, ⟨媒介変数の最小値⟩, ⟨媒介変数の最大値⟩)
```

この対象 parametric の属性を以下に示しておきますが、対象 explicit の属性と同名の属性と同じ働きをする為、ここでは詳細を述べません。

parametric の属性

nticks	正整数値	30	点数
line_width	正整数値	1	曲線の太さ
line_type	solid 又は dots	solid	曲線の型
color	文字列	black	曲線の色
key	文字列		凡例 (legend) を記述

parametric3d 文

対象 parametric3d は媒介変数表示された空間内曲線の描画で用います。構文そのものは、対象 parametric を定義する parametric 文に Z 座標の媒介変数式が追加された書式になります。

— 対象 parametric3d の構文 —

```
parametric(⟨ 式x ⟩, ⟨ 式y ⟩, ⟨ 式z ⟩, ⟨ 媒介変数 ⟩, ⟨ 媒介変数の最小値 ⟩, ⟨ 媒介変数の最大値 ⟩)
```

対象 parametric3d の属性は対象 parametric と同じ属性を持っています。

— parametric3d の属性 —

nticks	正整数値	30	点数
line_width	正整数値	1	曲線の太さ
line_type	solid 又は dots	solid	曲線の型
color	文字列	black	曲線の色
key	文字列		凡例 (legend) を記述

polar 文

polar 文は極座標表示された式のグラフを表現する対象 polar の定義で用いられます。

— 対象 polar の構文 —

```
polar(⟨ 動径の長さ ⟩, ⟨ 角度 ⟩, ⟨ 最小値 ⟩, ⟨ 最大値 ⟩)
```

対象 polar の属性は parametric の属性と同じものです。

— polar の属性 —

nticks	正整数値	30	点数
line_width	正整数値	1	曲線の太さ
line_type	solid 又は dots	solid	曲線の型
color	文字列	black	曲線の色
key	文字列		凡例 (legend) を記述

parametric_surface 文

parametric_surface 文は媒介変数表示された空間曲面を表現する対象 parametric_surface を定義する際に用います。

この parametric_surface 文の構文を以下に示しておきます。

— 対象 parametric_surface の構文 —

```
parametric_surface(⟨ 式x ⟩, ⟨ 式y ⟩, ⟨ 式z ⟩, ⟨ 媒介変数1 ⟩, ⟨ 媒介変数1の最小値 ⟩, ⟨ 媒介変数1の最大値 ⟩, 媒介変数2, ⟨ 媒介変数2の最小値 ⟩, ⟨ 媒介変数2の最大値 ⟩)
```

この parametric_surface の属性を以下に纏めておきますが、対象 explicit3d に包含されるものです。

parametric_surface の属性

xu_grid	正整数値	30	X 軸方向の点数
yv_grid	正整数値	30	Z 軸方向の点数
line_width	正整数値	1	曲線の太さ
line_type	solid 又は dots	solid	曲線の型
color	文字列	black	曲線色
key	文字列		汎用 (legend) を記述

points 文

points 文は平面上の点列を表現する対象 points と空間内の点列を表現する対象 points3d を定義する為に用いられます。

points 文の構文

```
points(< 点列のリスト >)
```

対象 points を定義する場合、点列リストの書式は $[[x_1, y_1], \dots, [x_n, y_n]]$ の様に点毎に定義するか、 $[[x_1, \dots, x_n], [y_1, \dots, y_n]]$ の様に点列の X 座標と Y 座標のリストの二通りで表記する事が可能になっています。

これは対象 points3d も同様で、対象 points3d の場合、点列リストの書式として、 $[[x_1, y_1, z_1], \dots, [x_n, y_n, z_n]]$ の様に三次元空間内の点として表現するか、 $[[x_1, \dots, x_n], [y_1, \dots, y_n], [z_1, \dots, z_n]]$ の様に X, Y, Z 座標のリストで表現するか二通りが選べます。

対象 points 対象と points3d には以下の属性があります。

points と points3d の属性

point_size	正整数値		描画時の点の大きさを指定
point_type	文字列		描画時の点の型を指定
points_joined	論理値	false	描画時に点列を線分で繋ぐかどうかを指定
line_width	正整数値		描画時の点列を繋ぐ線分の太さを指定
line_type	solid 又は dots	solid	点列を結ぶ線分の型を指定
color	文字列	red	色を指定
key	文字列		凡例 (legend) を記述

polygon 文と ractangular 文

polygon 文は平面上の多角形を表現する対象 polygon の定義で用いられます。

polygon 文では頂点を構成する点列を列記するだけなので、その表記方法は対象 points と全く同じです。

対象 polygon の構文

```
polygon(< 点列のリスト >)
```

又、平面上の長方形は `rectangular` 文を使って、対象 `rectangular` で表現する事も可能です。

rectangle 文の構文

```
rectangle([x1, y1], [x2, y2])
```

ここで、`rectangular` 文の構文で出てくる点 $[x_1, y_1]$, $[x_2, y_2]$ の関係ですが、これらの点は定義する長方形の対角線上の点になります。

以下に対象 `polygon` と対象 `rectangle` の属性を以下に示しておきます。

polygon と rectangle の属性

<code>transparent</code>	論理値	<code>false</code>	透明にするかどうかを指定
<code>border</code>	論理値	<code>true</code>	描画時に境界を描くかどうかを指定
<code>fill_color</code>	文字列	<code>red</code>	塗り潰しの色を指定
<code>line_width</code>	正整数値	<code>1</code>	線分の太さを指定
<code>line_type</code>	<code>solid</code> 又は <code>dots</code>	<code>dots</code>	点列を結ぶ線分の型を指定
<code>color</code>	文字列	<code>black</code>	色を指定
<code>key</code>	文字列		凡例 (legend) を記述

`transparent` と `border` 以外は既に説明したものと同一属性の為、ここでは解説しません。先ず、`transparent` は多角形内部の塗り潰しを行うかどうかを指定するものです。次の `border` は多角形の境界を描くかどうかを指定するものです。基本的に多角形は `fill_color` が `red` で、`transparent` が `false` の為、何も指定しなければ赤で塗り潰されます。

ellipse 文

`ellipse` 文は楕円を表現する対象 `ellipse` を定義する為に用います。

`ellipse` 文の構文を以下に示しておきましょう。

ellipse 文の構文

```
ellipse(< 原点x>, < 原点y>, < 半径1>, < 半径2>, < 角度1>, < 角度2>)
```

対象 ellipse の属性

nticks	正整数値	30	総点数を指定
transparent	論理値	false	透明にするかどうかを指定
border	論理値	true	描画時に境界を描くかどうかを指定
fill_color	文字列	red	塗り潰しの色を指定
line_width	正整数値	1	線分の太さを指定
line_type	solid 又は dots	solid	点列を結ぶ線分の型を指定
color	文字列	black	色を指定
key	文字列		凡例 (legend) を記述

vector 文

vector 文は矢印を生成する為の構文です。平面上の矢印は draw パッケージで対象 vector で表現されており、3次元空間内の矢印は対象 vector3d で表現されています。

先ず、平面内の対象 vector の構文を以下に示しておきます。

2次元空間内での対象 vector の構文

```
vector([[ 基点の x 座標 ], < 基点の y 座標 ], [[ 基点からの x 成分の増分 ], < 基点からの y 成分の増分 ]])
```

対象 vector は3次元空間に置く事も可能です。この場合の構文は単純に平面の場合から、Z座標の部分を増やしただけです。

3次元空間内部での対象 vector の構文

```
vector([[ 基点の x 座標 ], < 基点の y 座標 ], < 基点の z 座標 ], [[ 基点からの x 成分の増分 ], < 基点からの y 成分の増分 ], < 基点からの Z 成分の増分 ]])
```

猶、3次元空間でも、対象 vector は2次元空間のものと同様に平面的に表示されます。

対象 vector と対象 vector3d は基点から増分で表現される向きに線分を引き、この増分側を向きにして、矢印型を向けている事になります。

対象 vector の属性を以下に纏めておきます。

vector の属性

head_angle	正整数値	45	矢印の頭の角度
head_both	論理値	false	矢印の矢の設定
head_length	正整数値	2	矢印の柄の長さ
head_type	lilled,empty,nofilled	filled	矢印の頭の矢の型
line_width	正整数値	1	線分の太さを指定
line_type	solid 又は dots	solid	点列を結ぶ線分の型を指定
color	文字列	black	色を指定
key	文字列		凡例 (legend) を記述

対象 explicit の属性と同名の属性は同じ性質を持っています. それ以外の対象 vector 独自の属性は矢印の型に密接に関連するものばかりです.

image 文

image 文は対象 image を定義する為に用いられます.

image 文の構文

```
image(<< 実数行列 >>, < $x_0$ >, < $y_0$ >, < 幅 >, < 高さ >)
image(<< [r, g, b] の行列 >>, < $x_0$ >, < $y_0$ >, < 幅 >, < 高さ >)
```

対象 image の属性

colorbox	論理値	true	指定
palette	文字列		凡例 (legend) を記述

label 文

label 文は平面グラフ中の対象 label や 3 次元グラフ中の対象 label3d を定義する為に用います. 猶, label 文では複数の対象を同時に定義する事が可能です.

対象 label を定義する構文

```
label([< 文字列1>, < X 座標値1>, < Y 座標値1>], ..., [< 文字列n>, < X 座標値n>, < Y 座標値n>])
```

対象 label3d を定義する構文

```
label([< 文字列1>, < X 座標値1>, < Y 座標値1>, < Z 座標値1>], ..., [< 文字列n>, < X 座標値n>, < Y 座標値n>, < Z 座標値n>])
```

猶, 対象 label3d に含まれる文字列は常にグラフに対して正面を向く様に表示されます. その為, グラフを回しているとならベルが裏返ったりする事はありません.

label の属性

属性名	取り得る値	初期値	概要
label_allgnment	center,left,right	center	ラベルの位置合せ
label_orientation	horizontal,vertical	horizontal	ラベルの並び

その他の関数

```
add_zeroes(< 整数 >)
```

5.14.8 環境の違いを調整する大域変数

draw パッケージは MS-Windows 環境でも UNIX 環境でも利用可能です。但し、MS-Windows 版では gnuplot ではなく、wgnuplot が同梱されていたり、ストリームを用いない方式となっている為、その他の環境との違いを吸収する際に、次の大域変数を用いています。

環境の違いを調整する大域変数

draw_pipes	true	Windows 環境の場合は false
draw_command	gnuplot	Windows 環境の場合は wgnuplot

先ず、大域変数 draw_pipes は plot2d や plot3d 函数でも採用された Lisp のストリームを用いる方式である事を示すものです。前述の様に、中間ファイルとして、maxout.gnuplot_pipes か maxout.gnuplot の何れかを用いる事が出来ますが、大域変数 draw_pipes が true であれば maxout.gnuplot_pipes を生成し、ストリームを用いて Maxima 側から gnuplot へ制御命令を送り込めます。しかし、大域変数 draw_pipes が false の場合、maxout.gnuplot ファイルを生成しますが、この中間ファイルを用いる場合、Maxima は gnuplot への描画データだけではなく、各種設定文や描画命令を含む maxout.gnuplot を生成し、そのファイルを gnuplot に引渡す方式となっています。その為、Maxima 側からは、データを生成した後の面倒は一切見ません。

UNIX 環境の場合は、true にも false にも出来ます。但し、false にすると、マウスを使った図形の拡大・縮小、三次元グラフの回転といったマウス操作が出来ません。更に、グラフ画面に `[m]` と入力しても、gnuplot の mouse が on になりません。その代わりに、別途、gnuplot を立ち上げて maxout.gnuplot ファイルを読み込めば mouse を on にすればマウスによる操作が行えます。又、グラフ画面で `[m]` と入力する事で、mouse の on/off の切り替えが行えます。

猶、UNIX 環境でこの変数の値が true の場合、マウスによるグラフの操作も、gnuplot の mouse の on/off も可能です。

この変数の具体的な利用方法としては、グラフ描画の設定文を取り込んだ maxout.gnuplot ファイルを出力する為に使えます。この maxout.gnuplot ファイルは非常に便利なファイルで、このファイルさえあれば、ファイルを編集する事で、後は gnuplot 単体で色々な処理が行える為です。

次に、draw_command は MS-Windows では wgnuplot、その他の環境では gnuplot を用いる為に用いる大域変数です。独自の gnuplot を使いたい場合に、この変数にアプリケーション名を指定しておきます。その様な特殊な大域変数の為、この変数を弄る必要はまずないでしょう。

第6章 Maximaの簡単な改造

6.1 使い勝手の向上を目指して

Maxima は非常に面白いシステムです。特に面白い事が改造が容易に出来る事です。これは LISP のエキスパート (LISPer) である必要はありません。まずは Maxima のソースファイルを活用すれば良いのです。実際、Maxima は単純に数式を処理するだけでなく、様々な入出力を行っています。その中で、グラフィックスでは外部のアプリケーションを立ち上げたりもしている程です。その為、Maxima で何かを最初から作り上げる必要性は意外に無く、似た処理を行う函数を観察して、その函数の仕組を真似たり、場合によってはその函数の複製を作ったり、挙句の果てには、その函数を乗っ取ってしまっても構わないのです。勿論、乗っ取りを行えば、他との互換性が崩れてしまうので、あまり勧められませんが、複製を作ってしまうのが一番良いのです。

さて、Maxima の Help システムはそれなりの良く出来ています。しかし、最近の数式処理システムでは HTML 等を利用するのが主流で、texinfo を用いる方法は過去の方法です。そこで、HTML や PDF が表示出来れば随分と使い易くなる筈ですね。その様な訳で、describe 函数を改造してみましょう。

6.2 describe 函数は何処か

まず、describe 函数を改造したければ、describe 函数の所在を明確にし、それがどの様なものであるかを知っておく必要があります。

describe 函数の探し方は色々ありますが、UNIX 環境であれば、Maxima のソースファイルが収録されたディレクトリ src に移動して、`grep describe *.lisp|less` を実行してみましょう。又、MS-Windows 版でも Maxima のソースファイルが付属しているので、何らかのアプリケーションを使って探すのも良いでしょう。

では以下に結果を一部示しておきましょう。


```

ifactor.lisp;;; Elliptic curve factorization method as described in the paper
init-cl.lisp:(defvar $help "type 'describe(topic);' or 'example(topic);' or '? topic'"
limit.lisp;;; I believe a large portion of this file is described in Paul
macdes.lisp:(defmspec $describe (x)
macdes.lisp:(defmspec $describe (x)
mdebug.lisp:      ;; Process "?" lines. This is either a call to describe or a
mdebug.lisp:      '(((displayinput) nil ((describe) ,line $exact)))
mdebug.lisp:      '(((displayinput) nil ((describe) ,line $inexact)))
nparsel.lisp;;; the comments in GRAM package, which describe them in reasonable detail.
nset.lisp;;; difficult to describe -- for an odd number of arguments, we favor the left side of the tree.
option.lisp:      ((eq 'describe (caar ans)) (mdescribe (decode (cadr ans))))
option.lisp:(subc $describe (c))
option.lisp:(subc $general-info () $describe $example $options $primer $propos)
option.lisp:(subc $options (c) $down $up $back $describe $exit)
option.lisp:(subc $user-aids () $primer $describe $options $example $propos $visual-aids)
rand-mt19937.lisp;;; The initialization routine is described below. Let s be the seed,
residu.lisp:      ;; described in bug 1073338.)
schat.lisp;;; I think this is described in Chapter 3 of J. Moses' thesis,
lines 1-18

```

今回、Maxima-5.13.0 を利用しているため、他の版では多少出力が違っているかもしれませんが、しかし、探し方は単純で、Maxima の関数定義では `defun` や `defmspec` 等が用いられているので、`$describe` と一緒にこれらの文字列が出ている箇所を捜せば良いのです。すると、`macdes.lisp:(defmspec $describe (x)` という行があるので、この事から `describe` 関数は `macdes.lisp` で定義されている事が判ります。

そこで今度は `macdes.lisp` の複製を作って、それを適当なエディタを用いて編集してみる事にしましょう。

6.3 describe 関数の動作

まず、`describe` 関数の定義式を以下に示しておきます。猶、Maxima-5.13.0 の `describe` 関数なので、他の版では多少異っているかもしれません。

describe 関数の定義

```

(defmspec $describe (x)
  (let
    ((topic ($sconcat (cadr x)))
      (exact-p (or (null (caddr x)) (eq (caddr x) '$exact))))
    (cl-info::*prompt-prefix* *prompt-prefix*)
    (cl-info::*prompt-suffix* *prompt-suffix*))
    (if exact-p
      (cl-info::info-exact topic)
      (cl-info::info topic))))

```

まず、`defmspec` は `maxmac.lisp` で定義された Maxima のマクロで、Common Lisp で関数の定義に用いられる `defun` の様に `$describe` 関数 (Maxima 上では `describe` 関数) を定義するものです。

この `describe` 関数を定義する S 式の構造を以下に示しておきましょう。

```

(defmspec 関数名 (変数1, ..., 変数n)
  (let ((内部変数1 値1) ... (内部変数m 値m))
    (処理1) ... (処理k)))

```

この `describe` 関数の処理を簡単に解説すると、最初に与えられた変数 `x` に束縛された Maxima の文字列に対し、`($sconcat (cadr x))` を実行し、結果を変数 `topic` に束縛させます。次に、`(caddr x)` の値

が空、或いは'\$exact に等しい場合に exact-p を T にし、それ以外は NIL にします。それから cl-info パッケージの変数*prompt-prefix*と変数*prompt-suffix*に,*prompt-prefix*と*prompt-suffix*の値を束縛させます。

次が if による分岐処理になります。ここでは exact-p の値が T であれば、cl-info パッケージの info-exact 関数に topic の値を引き渡します。NIL であれば cl-info パッケージの info 関数に topic の値を引き渡します。通常の処理では、info-exact に topic の値が引渡されて、texinfo によるマニュアルが表示される事になります。

6.4 ponpoko の仕様

さて、describe 関数の動作は大体判りました。今度はどの様に改造するかを考えてみましょう。先ず、最近のアプリケーションは困った時には、HTML ブラウザを使ってヘルプを表示する事が可能ですね。又、PDF も表示出来れば良いですね。

そこで、describe 関数を基にして作る関数の名前は、そのオリジナリティの無さから狸らしく ponpoko としましょう。又、この命令を書き込むファイル名を ponpoko.lisp としましょう。

この関数 ponpoko は、関数名を指定した場合に通常の info ファイルを用いたヘルプを表示します。そして、文字列として html を指定すると HTML のマニュアルを表示し、pdf を指定すると PDF のマニュアルを表示するものとしましょう。猶、HTML マニュアルと PDF マニュアルは私のホームページに置いてあるものを用います。¹

6.5 大域変数の作り方

ここで HTML ブラウザとしては、firefox や opera に w3m、internet explore と色々ありますね。PDF も同様で、acroread や xpdf に gv もあり、環境によって色々切り替えられるべきでしょう。

そこで、他の関数でも用いられている大域変数で、これらのブラウザを切り替える事にします。ここで Maxima 内部で大域変数を定義したい場合、defvar マクロを用いて定義します。

defvar マクロ

(defvar 変数名 値)

ここで、defvar は LISP のマクロなので、変数名は Maxima の内部表現になります。但し、値の方は変換函数を用いる事もあるので、Maxima の内部表現にする必要はありません。プログラムの特性で決めれば良いのです。

又、HTML ファイルや PDF ファイルの格納先も別途定めておく必要がありますね。

この事から、必要となる大域変数はブラウザとファイルの格納先の 4 個になります。そこで、Maxima 側の変数名を hbrowser で HTML ブラウザ、pbrowser で PDF ブラウザ、jhtmlmdir で HTML ファイルの格納先、jpdfmdir で PDF ファイルの格納先を指定する事にします。

¹<http://www.bekkoame.ne.jp/ponpoko/Math/books/ManualBook.tgz>,
<http://www.bekkoame.ne.jp/ponpoko/Math/books/ManualBook.pdf>

ここで,Maxima の変数の内部表現は,先頭に\$が付きます.その為,hbrowser は LISP 側では\$hbrowser になります.この事を留意して大域変数を定義しましょう.

ponpoko.lisp での大域変数の定義

```
(defvar $hbrowser "firefox")
(defvar $pbrowser "xpdf")
(defvar $jhtmlmdir "Maxima/ManualBook/")
(defvar $jpdfdir "Maxima/ManualBook/Book-New/")
```

ここで,\$jhtmlmdir と\$jpdfdir は私の環境の値を設定しています.この値は皆さんの環境に合わせて指定する様にして下さい.

ここで注意する事として,これらの変数には LISP の文字型のデータが束縛されている事です.注意すべき事として,Maxima の文字型は LISP の文字型では無い事が一番に挙げられます.ここで Maxima の文字列から LISP の文字列に変換する場合,stringproc パッケージに含まれている函数 lstring が使えます.

さて,次に函数の定義を行う事にしましょう.今回は describe 函数の乗っ取りが目的なので,引数から html や pdf が得られた時に,こちらの処理を挿入すれば良いのです.その為,LISP の cond 函数を用いた処理が妥当でしょう.

即ち,以下の様な函数のイメージになります.

函数 ponpoko のイメージ

```
(defmspec $ponpoko (x)
  (let
    ((topic ($sconcat (cadr x)))
      (exact-p (or (null (caddr x)) (eq (caddr x) '$exact))))
    (cl-info::*prompt-prefix* *prompt-prefix*)
    (cl-info::*prompt-suffix* *prompt-suffix*))
  (cond
    ((topic に"html"の情報がある場合)
     (hbrowser を起動しろ))
    ((topic に"pdf"の情報がある場合)
     (pbrowser を起動しろ))
    (t
     (if exact-p
         (cl-info:::info-exact topic)
         (cl-info:::info topic))))))
```

本来の describe 函数にこちらの要求を取り込んだらこの様な形になります.すると,topic がどのような書式なのか.そして,外部のアプリケーションに大域変数の値を組合せたパラメータを引き渡して起動させれば良いのかが問題になりますね.今度はこれらの課題を吟味してみましよう.

更に,HTML ファイルや PDF ファイルの名前も大域変数にしておくとうどうでしょうか? 随分と汎用性が出ますね,そこで,ブラウザで開く HTML ファイルと PDF ファイルを指定する変数名を各々hfile と pfile とし,初期値を index.html と ManualBook.pdf にしておきましょう.

これを纏めると,函数 ponpoko は完成です.

6.8 完成

以上の事を組合せて函数 ponpoko を完成させてみましょう.

函数 ponpoko

```
(defvar $hbrowser "firefox")
(defvar $pbrowser "xpdf")
(defvar $hfile "index.html")
(defvar $pfile "ManualBook.pdf")
(defvar $jhtmlmdir "Maxima/ManualBook/")
(defvar $jpdfdir "Maxima/ManualBook/")
(defmspec $ponpoko (x)
  (let
    ((topic ($sconcat (cadr x)))
      (exact-p (or (null (caddr x)) (eq (caddr x) '$exact)))
      (cl-info::*prompt-prefix* *prompt-prefix*)
      (cl-info::*prompt-suffix* *prompt-suffix*))
    (cond
      ((equal topic "html")
       ($system (concatenate 'string $hbrowser " "
                             $jhtmlmdir $hfile "&")))
      ((equal topic "pdf")
       ($system (concatenate 'string $pbrowser " "
                             $jpdfdir $pfile "&")))
      (t (if exact-p
              (cl-info::info-exact topic)
              (cl-info::info topic)))))))
```

説明は特に不必要かと思いますが,jhtmlmdir と jpdfdir は貴方の環境に合わせて修正して下さい.私の環境では,HTML ファイルをホームディレクトリ上の Maxima/ManualBook 以下に置き,PDF ファイルも同じディレクトリに置いています.

ここで,この函数の遊び方ですが,兎に角,Maxima 側から load 函数を用いて読み込んでしまえば良いのです.ponpoko.lisp をカレントディレクトリ上に置いているのであれば,ディレクトリの指定をする必要も無く,単純に `load("ponpoko.lisp");` と入力して実行すれば良いのです.これで函数 ponpoko が読み込まれます.後は,ponpoko("html") や ponpoko("pdf") を実行すれば良いのです.

さて実際の実行例を示しておきましょう。

```
(%i42) load("ponpoko.lisp");
(%o42)                                ponpoko.lisp
(%i43) ponpoko("to_lisp");

-- Function: to_lisp ()
   Enters the Lisp system under Maxima. '(to-maxima)' returns to
   Maxima.

(%o43)                                true
(%i44) ponpoko("html");
(%o44)                                0
```

読込んだ後に, `to_lisp` 関数を調べています. この場合は `describe` 関数と同じ働きをします. 次に, 引数を "html" にすると, デフォルトで `firefox` を立ち上げます. ここで, `system` 関数に送り込んだ文字列の末尾に `&` を入れている為, `Maxima` は `firefox` を立ち上げて入力プロンプトに復帰しています. ここで, `&` を末尾に入れていなければ, 起動した外部アプリケーションを終了する迄, 処理が `Maxima` に戻りません.

ここで, 関数 `ponpoko` を一々読込むのが面倒であれば, `maxima-init.mac` ファイルに予め `load("ponpoko.lisp");` を記入しておきます. 例として, `lstringproc` パッケージも読込む様にした `maxima-init.mac` を以下に示しておきます.

```
load("ponpoko.lisp");
load("stringproc.lisp");
```

このファイルは UNIX の場合にはホームディレクトリに置けば, 準備が完了です.

6.8.1 MS-Windows 環境の場合

MS-Windows 環境の場合は UNIX 環境と比べ, 幾つか注意しなければならない事があります. 先ず, フォルダ (ディレクトリ) の区切は MS-Windows では逆スラッシュ (`\`) を用いていますが, `Maxima` 内部では UNIX と同様にスラッシュ (`/`) を用います. ところで, 考えておかなければならないのは, `ponpok.lisp` と `maxima-init.mac`, そしてマニュアルを置くフォルダです.

MS-Windows を利用する方は, 基本的に `Dos` 窓を開いて処理する人よりも, `wxMaxima` 等の GUI 環境を用いるの方が大多数でしょう. その為, 利用する環境によって `maxima-init.mac` 等の置場を変更する必要があります.

先ず, 何がなんでも `DOS` 窓を利用する方の場合, `maxima-init.mac` と `ponpoko.lisp` の置場は `Maxima` による作業を行うフォルダに入れておく必要があります.

これに対し, `wxMaxima` や `XMaxima` を利用する場合は勝手が異なります. 先ず, メニューバーのアプリケーションから `Maxima` や `XMaxima` を選んで立ち上げる場合には, `C:\Documents and Settings`

フォルダにある利用者固有のフォルダに `maxima-init.mac` と `ponpoko.lisp` を入れておく必要があります。ところが、`wxMaxima` はやや勝手が異なります。`wxMaxima` の場合、通常は `wxMaxima` が置かれた場所がそのホームディレクトリとなるので、ここに `maxima-init.mac` ファイルを置かなければなりません。

勿論、`wxMaxima` を DOS 窓で `C:\Documents and Settings` フォルダにある個人フォルダに移動して立ち上げてしまえば良いのですが、そうする場合には、`wxMaxima` や `maxima` の実行ファイルのあるフォルダを環境変数 `Path` に追加しなければなりません。この様に、Windows では何かと面倒な事が生じるのです。そこで、何も考えずに出来る方法は兎に角、個人フォルダと `wxMaxima` のフォルダの二箇所に別々に置いてしまう事です。`wxMaxima` だけを利用するのであれば、`wxMaxima` のフォルダだけで十分です。

さて、この `wxMaxima` は Maxima のフォルダの直下にあります。Maxima はインストール時に特に設定を行わなければ、`C:\Program Files\Maxima-5.13.0` (利用する Maxima が 5.13.0 の場合) にある筈です。そして、`maxima-init.mac`、`ponpoko.lisp` や Maxima マニュアルのフォルダも一緒に置きます。ここで、Maxima マニュアルのフォルダは下に `ManulaBook` フォルダを置き、その中に HTML ファイル、PDF ファイルや利用する画像ファイルを置きます。

又、`firefox` とかのアプリケーションも MS-Windows の場合は `Path` が通っていない事があるので、別途、環境変数 `Path` にアプリケーションへの経路を設定しなければなりません。この環境変数 `Path` の設定はコントロールパネルのシステムをダブルクリックし、その中にある環境変数のボタンを押します。ユーザー環境変数とシステム環境を編集する為のウィンドウが出て来るので、システム環境変数から、`Path` を選んで編集ボタンを押すと、入力用のウィンドウ欄が出るので、そこに、`firefox` 等のアプリケーションが置かれたフォルダの位置を追加します。それで使える様になる筈です。

猶、UNIX 環境と MS-Windows 環境で切り替えを行う際に、Maxima では、`*autoconf-win32*` を用いて判別を行う様になっています。実際、グラフ処理を行う為に、環境によってアプリケーションの切り替えを行う必要がある `plot.lisp` では動作する環境が MS-Windows 環境であるかどうかを、`(string= *autoconf-win32* "true")` を用いて判別しています。実際、この S 式を評価して `true` になる場合を MS-Windows 環境、そうでなければ UNIX 環境として処理を行っています。

6.9 大域変数の変更について

ところで、`ponpoko.lisp` で定義した大域変数の変更は少し面倒です。何故なら、Maxima の文字列と LISP の文字列は別物だからです。そこで、この変換に `stringproc` パッケージに付属する `lstring` 関数を用います。例えば、`hbrowser` を初期値の `firefox` から `opera` に変更してみましょう。

```
(%i1) load("stringproc.lisp");
(%o1) /usr/local/share/maxima/5.13.0/share/contrib/stringproc/stringproc.lisp
(%i2) hbrowser;
(%o2) firefox
(%i3) hbrowser:lstring("opera");
(%o3) opera
(%i4) ponpoko("html");
```

(%o4)

0

この方法は ponpoko.lisp で定義した大域変数全てで使えます。猶、注意する事として、ディレクトリを指定する場合、末尾に必ず”/”を入れ忘れないで下さい。これが無いと、ファイル名と変に繋がってしまうので、エラーになる筈です。

さて、如何でしょうか？ブラウザを別個に立ち上げる手間が省けただけですね。今後、必要な機能としては、指定した項目が載っている HTML ファイルを開くといった操作が必要になりますね。これはとても大変な事で、安易な方法を目指すこの章の範囲を越えてしまいそうなので、以降の作業は皆さんに投げとおきます。

関連図書

- [1] J. リヒター-ゲバート/U.H. コルテンカンブ著, 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アラーク東京,1990.
- [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] 高木貞治, 復刻版 近世数学史 数学雑談, 共立出版,1997.
- [7] 田中尚夫, 選択公理. 遊星社,1987.
- [8] 広瀬健 横田一正, ゲーデルの世界 -完全性定理と不完全性定理-, 海鳴社,1985.
- [9] 本間龍雄, 組合せ位相幾何学, 共立出版,1980.
- [10] 寺坂英孝編, 現代数学小辞典, ブルーボックス, 講談社,2005.
- [11] 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.
- [12] 村上順, 結び目と量子群, 数学の風景 3, 朝倉書店,2000.
- [13] 日本数学会編, 数学辞典 第3版, 岩波書店,1987.
- [14] 矢吹道郎, 大竹敢, 使いこなす GNUPLOT(改訂新版), テクノプレス,2001.
- [15] H.Cohen, A Course in Computational Algebraic Number Theory,GTM 138, Springer-Verlag,New York-Berlin,2000.
- [16] D.Cox,J.Little and D. O'Shea,Ideals, Varieties, and Algorithms, UTM,Springer-Verlag,New York-Berlin,1992.
- [17] Gert-Martin Greuel, Gerhard Pfister, A Singular Introduction to Commutative Algebra, Springer-Verlag,New York-Heidelberg-Berlin,2000.
- [18] D.Rolfen, Knots and Links. Publish or Perish, Inc,1975.
- [19] Hal Schenck, Computational Algebraic Geometry London Mathematical Society student texts;58,2003.

- [20] Open AXIOM のサイト <http://wiki.axiom-developer.org/FrontPage>
- [21] DERIVE のサイト <http://www.derive.com>
- [22] GAP のサイト <http://www-gap.dcs.st-and.ac.uk/>
- [23] Macaulay2 のサイト <http://www.math.uiuc.edu/Macaulay2/>
- [24] Mathsoft Engineering & Education, Inc. <http://www.mathcad.com/>
- [25] Wolfram Research Inc. <http://www.wolfram.com/>
- [26] Waterloo Maple Inc. <http://www.maplesoft.com/>
- [27] Maxima の SOURCEFORGE のサイト <http://maxima.sourceforge.net/>
- [28] MuPAD のサイト <http://www.mupad.de/>
- [29] PARI/GP のサイト <http://pari.math.u-bordeaux.fr/>
- [30] REDUCE のサイト <http://www.zib.de/Symbolik/reduce/>
- [31] Risa/Asir 神戸版 <http://www.math.kobe-u.ac.jp/Asir/asir-ja.html>
- [32] OpenXM(Open message eXchange for Mathematics)
<http://www.math.sci.kobe-u.ac.jp/OpenXM/index-ja.html>
- [33] SINGULAR のサイト <http://www.singular.uni-kl.de/>
- [34] 株式会社シンプレックスのページ <http://www.simplex-soft.com/>
- [35] The MathWorks,Inc. <http://www.mathworks.com/>
- [36] Octave WebPage <http://bevo.che.wisc.edu/octave/>
- [37] Scilab のサイト <http://www.scilab.org/>
- [38] 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>
- [39] R のサイト <http://www.r-project.org>
- [40] Insightful Corporation のページ <http://www.insightful.com/>
- [41] 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/>
- [42] dynagraph のサイト <http://www.math.umbc.edu/rouben/dynagraph>
- [43] KSEG のサイト <http://www.mit.edu/ibaran/kseg.html>

[44] Geomview のサイト <http://http.geomview.org/>

[45] surf の sourceforge のサイト <http://surf.sourceforge.net/>

[46] XaoS のサイト <http://wmi.math.u-szeged.hu/~kovzol/xaos>

索引

逆引

三角函数

- 三角函数を含む式の展開を行いたい, 249
- 三角函数を含む式を簡単にしたい, 251

式

- Maxima に式の自動展開をさせたい, 128
- 式の因子分解をしたい, 98
- 式を展開したい, 130

常微分方程式

- 微分方程式の書き方, 296
- 常微分方程式の簡単な解き方, 297

数値

- Maxima の数学定数, 71
- 精度を変更したい, 70

積分

- 数値積分を行いたい, 292
- 代数的数を使って積分したい, 283
- 変数変換をしたい, 281
- ラプラス変換を使いたい, 287
- 留数の計算をしたい, 287
- 定積分を行いたい, 289

微分

- 式の微分をしたい, 272

評価

- 式を手軽に展開したい, 61
- 式を手軽に評価したい, 61
- 方程式の解を手軽に他の式に入れたい, 63

描画

- gnuplot の使い方を知りたい, 326
- plot2d 函数の使い方, 304
- plot3d 函数の使い方, 307

方程式

- 決った範囲内で実数解だけを求めたい, 261
- 実数解だけを求めたい, 260
- 自動的に方程式の解を変数に入れたい, 257
- 連立方程式を解きたい, 262

演算子

- A
- and, 43
- D
- do, 45
- E
- else, 45
- elseif, 45
- F
- for, 45
- from, 45
- I
- if, 45
- if 文で利用可能な演算子, 189
- in, 190
- N
- next, 45, 190
- not, 43
- O
- or, 43
- S
- step, 45, 190
- T
- then, 45
- thru, 45, 191
- U
- unless, 45, 191

W

while, 45, 191

記号

<, 43

<=, 43

>, 43

>=, 43

*, 40

**, 40

+, 40

-, 40

., 40

/, 40

;, 44

::, 44

::=, 44

:=, 44, 195

=, 43, 256

#, 43

^, 40

^^, 40

型

A

any, 21, 23, 35

any_check, 21

B

big, 21

bigfloat, 70

bignum, 21, 70

bool, 21

boolean, 21

C

clause, 35

complex, 21

E

expr, 35

F

fixnum, 21, 70

fixp, 21

float, 21, 70

floatnum, 21

flonum, 21

I

integer, 21

L

list, 21

listp, 21

N

none, 21

number, 21

R

rat, 21

rational, 21

real, 21

Maxima 函数

B

batch, 371

C

closeps, 323

coutou_plot, 323

G

gnuplot_close, 319

gnuplot_pipes, 319

gnuplot_replot, 319

gnuplot_reset, 319

gnuplot_restart, 319

gnuplot_start, 319

O

openplot_curves, 321

P

plot2d, 304

plot2d_ps, 323

plot3d, 307

pscom, 323

psdraw_curve, 323

psdraw_points, 323

V

viewps, 323

函数

:lisp, 4, 67

?round, 74

?truncate, 74

A

acos, 248
 acosh, 248
 acot, 248
 acoth, 248
 acsc, 248
 acsch, 248
 activate, 9
 addcol, 160
 addrow, 160
 adjoint, 164
 algsys, 262
 alias, 239
 allroots, 259
 alphanumericp, 174
 and, 8
 antid, 295
 antidiff, 295
 append, 139
 appendfile, 212
 apply, 201
 apply1, 48
 apply2, 48
 approps, 239
 args, 109
 array, 148
 arrayapply, 151
 arrayinfo, 147
 ascii, 176
 asec, 248
 asech, 248
 asin, 248
 asinh, 248
 askinteger, 134
 asksin, 135
 assume, 8, 10
 at, 25
 atan, 248

atan2, 248

atanh, 248

atom, 138

atomgrad, 27

atvalue, 25

augcoefmatrix, 154

B

batch, 210

batchload, 210

batcon, 210

bc2, 300

bezout, 94

bfloat, 72

bfloatp, 72

bigfloat, 70

block, 45, 57, 188

bothcoef, 90

break, 192, 221

buildq, 198

C

cabs, 72

carg, 72

catch, 192

cequal, 175

cequalignore, 175

cgreaterp, 175

cgreaterpignore, 175

changevar, 281

charat, 177

charlist, 177

charp, 174

charpoly, 167

cint, 176

cless, 175

clessignore, 175

close, 174

closefile, 212

coeff, 90

coefmatrix, 154

col, 160

- collapse, 220
 - columvector, 168
 - combine, 93
 - compile, 203
 - compile_file, 203
 - conj, 168
 - conjugate, 168
 - constantp, 72
 - constituent, 174
 - content, 96
 - copylist, 139
 - copymatrix, 160
 - cos, 248
 - cosh, 248
 - cot, 248
 - coth, 248
 - csc, 248
 - csch, 248
- D
- dblint, 294
 - deactivate, 9
 - declare, 2, 11, 14, 58
 - declare_translated, 207
 - define, 195
 - define_variable, 18, 21
 - define_variable 関数の処理手順, 23
 - defint, 289
 - defmatch, 52
 - defrule, 49
 - delete, 139
 - delta, 287
 - demoivre, 130
 - denom, 93
 - depends, 27, 280
 - derivdegree, 274
 - derivlist(ev 関数の引数), 58, 62
 - desolve, 298
 - determinant, 166
 - detout(ev 関数の引数), 58
 - diagmatrix, 154
 - diagmatrixp, 159
 - diff, 272
 - diff(ev 関数の引数), 58
 - digitcharp, 174
 - dipslay, 229
 - disp, 229
 - dispform, 40, 106
 - dispfun, 197
 - displate, 111
 - disprule, 46
 - dispterm, 230
 - distrib, 131
 - divide, 100
 - do, 45, 190
 - ldisplay, 229
- E
- derivlist, 62
 - echelon, 154
 - econs, 139
 - eigenvalues, 169
 - eigenvectors, 169
 - eivals, 169
 - eivects, 169
 - eliminate, 94
 - ematrix, 154
 - endcons, 139
 - entmatrix, 154
 - entier, 72
 - equal, 64
 - erf, 280
 - errcatch, 193
 - error, 193
 - errmsg, 193
 - ev, 17, 56
 - eval, 65
 - eval(ev 関数の引数), 58
 - evenp, 72
 - ev 関数で利用可能な引数, 58
 - expand, 60, 130

- expand(ev 関数の引数), 58, 60
- exponentialize, 130
- express, 277
- ezgcd, 96
- F
 - factcomb, 93
 - factor, 98
 - factorout, 98
 - factorsum, 98
 - facts, 8, 15
 - facttimes, 93
 - feature, 15
 - featurep, 11, 16
 - Ffortran, 238
 - file_search, 215
 - file_type, 215
 - filename, 215
 - fillarray, 151
 - first, 140
 - fix, 72
 - flength, 174
 - floatnump, 72
 - forget, 8
 - fposition, 174
 - freeof, 108
 - freshline, 174
 - fullratsimp, 101
 - fullratsubst, 124
 - funcsolve, 266
 - fundef, 197
 - funmake, 206
- G
 - Γ 関数, 42
 - gcd, 96
 - gcde, 96
 - gcdex, 96
 - gcfactor, 96
 - genmatrix, 154
 - get, 20
 - gfactor, 96
 - gfactorsum, 96
 - go, 192
 - gradef, 27
 - gramschmidt, 170
 - grind, 229
 - gschmidt, 170
- H
 - hipow, 91, 274
 - horner, 89
- I
 - ic1, 300
 - ic2, 300
 - ident, 154
 - if, 45, 189
 - ilt, 287
 - imagpart, 72
 - infix, 37
 - inflag の影響を受ける関数, 138
 - innerproduct, 168
 - inpart, 114, 123, 125
 - inprod, 168
 - integerp, 72
 - integrate, 19, 278
 - intosum, 133
 - invert, 164
 - is, 64
 - isolate, 111
 - isqrt, 72
- K
 - kill, 219, 221
 - killcontext, 9
- L
 - labels, 225
 - lambda, 57, 196
 - laplace, 287
 - last, 140
 - lcharp, 174
 - ldefint, 289
 - ldisp, 229
 - length, 139

- let, 51
- letrat, 51
- letrules, 46
- letsimp, 51
- lfreeof, 108
- lhs, 256
- limit, 19, 270, 280
- linsolve, 261
- listarray, 147
- listofvars, 109
- listp, 138
- load, 210
- loadfile, 210
- local, 207
- local(ev 関数の引数), 58, 62
- logconrctat, 254
- lopow, 91
- lowercasep, 174
- lratsubst, 124
- lstring, 402, 406
- lstringp, 174
- l
- lstring, 176
- M
- kill, 38
- mainvar, 88
- make_array, 148
- make_art.q, 148
- make_fransform, 312
- make_string_input_stream, 173
- make_string_output_stream, 173
- map, 144
- mapatom, 144
- matchdeclare, 50
- matchfix, 37
- matrix, 152
- matrixmap, 160
- matrixp, 159
- mattrace, 163
- max, 72
- member, 138
- mfuncall, 68
- min, 72
- minor, 162
- mod, 100
- mode_declare, 21, 202
- mode_identity, 21
- multthru, 131
- N
- nary, 37
- ncharpoly, 167
- newcontext, 9
- newdet, 166
- newline, 174
- noeval(ev 関数の引数), 58
- nofix, 37
- nonscalarp, 159
- nonzeroandfreeof, 295
- not, 8
- nouns(ev 関数の引数), 58
- nroots, 261
- nterms, 91
- nthroot, 98
- num, 93
- numberp, 72
- numer(ev 関数の引数), 58
- numerval, 136
- O
- oddp, 72
- ode2, 298
- opena, 173
- openr, 173
- openw, 173
- optimize, 121
- or, 8
- ordergear, 4
- ordergreatp, 5
- orderless, 4
- orderlessp, 5
- print, 229

P

parmanent, 166
parsetoken, 177
part, 114, 123, 125
partition, 113
pickapart, 114
playback, 231
ploarform, 255
plog, 255
postfix, 37
powers, 91
prefix, 37
printf, 174
printprops, 31, 50
product, 19, 116
properties, 20, 29, 58
propvars, 29
put, 20

Q

qput, 20, 23
quanc8, 280, 292
quit, 240
quotient, 100

R

radcan, 133
random, 72
rank, 163
rat, 78
ratcoef, 90
ratdenom, 93
ratdiff, 93
ratdisrep, 78
ratexpand, 101
ratnumer, 93
ratnump, 90
ratp, 90
ratsimp, 64, 101
ratsubst, 124
ratvars, 78, 92
ratweight, 92

read, 210
readline, 174
readonly, 210
realpart, 72
realroots, 259
rearray, 151
rectform, 113
rem, 29
remainder, 100
remarray, 151
remfunction, 198
remlet, 54
remove, 29, 38
remrule, 54
remvalue, 109
reset, 219
residue, 287
rest, 140
resultant, 94
return, 192
reveal, 230
reverse, 139
rhs, 256
risch, 278
risch(ev 函数の引数), 58, 62
romberg, 280, 292
room, 236
rootscontract, 93
row, 160

S

cunlisp, 176
save, 213
scalarp, 159
scanmap, 145
sconc, 177
sconcat, 172
scopy, 177
scsimp, 134
sdowncase, 181
sec, 248

sech, 248
setelmx, 160
setup_autoload, 218
showvars, 92
similaritytransform, 169
simplode, 180
simtran, 169
sin, 248
sinh, 248
sinvertcase, 181
slength, 177
smake, 177
smismatch, 185
solve, 264
split, 179
sposition, 177
sqfr, 98
sqrt, 72
sremove, 184
sremovefirst, 183
sreverse, 177
ssearch, 185
ssort, 184
status, 235
strim, 177
striml, 177
strimr, 177
string, 172
stringout, 211
stringp, 174
sublis, 125
sublist, 140
submatrix, 162
subst, 123
substinpart, 126
substpart, 126, 140
substring, 179
sum, 19, 116
sumcontract, 132
sunlisp, 176

supcase, 180
supcontext, 9
subst, 183
substfirst, 182
system, 237

s

sequal, 176
sequalignore, 176

T

tan, 248
tanh, 248
tcl_output, 230
tellrat, 85
tellsimp, 52
tex, 238
throw, 192
tldefint, 289
tlimit, 271
to-maxima, 66
to_lisp, 66
tokens, 181
totaldisrep, 78
translate, 29
transplate, 202
transpose, 163
triangularize, 163

U

ueivects, 170
uniteigenvectors, 170
unitvector, 170
unknown, 130
unorder, 4
untrllrat, 87
uppercasep, 174
use_fast_arrays, 148
uvect, 170

W

with_stdout, 212
writefile, 212

Z

zeromatrix, 154

記号

%ith, 225

記号

$>_m$, 2

A_m , 2

' , 65

" , 65

?, 67, 232

%i, 221

%o, 221

! , 42

!! , 42

属性

A

additive, 19

algebraic, 35

alphabetic, 2, 16, 108

analytic, 20

argpos, 35

assign, 23, 24

assign-mode-check, 23

atomgrad, 31

atvalue, 31

C

commutative, 19

complex, 18

constant, 16

D

decreasing, 20

dependency, 27

E

english, 35

even, 18

evenfun, 20

evflag, 20, 58

evflag 属性をデフォルトで持つ大域変数, 58

evfun, 20, 57–59

evfun 関数の作用の順番, 60

evfun 属性を持つ関数, 59

G

gradef, 27, 31

I

imaginary, 18

increasing, 20

integer, 18

irrational, 18

L

lassociative, 19

linear, 19

logical, 35

lpos, 35

M

mainvar, 16

matchdeclare, 31, 50

mode, 21

mode_check_errorp, 22

mode_check_warnp, 22

mode_checkp, 22

multiplicative, 19

N

noninteger, 18

nonscalar, 16

noun, 20

O

odd, 18

oddfun, 20

outative, 19

P

pos, 35

posfun, 20

R

rassociative, 19

rational, 18

real, 18

rpos, 35

S

scalar, 16

special, 16, 23

- symmetric, 19
- T
 - transfun, 29
- U
 - untyped, 35
- V
 - value_check, 23
- 大域变数
 - %, 223
 - %e_to_numlog, 253
 - %edispflag, 227
 - %num_list, 264
 - %%, 223
 - %emode, 128
 - %enumer, 61, 128
- A
 - absboxchar, 227
 - algebraic, 84
 - algepsilon, 264
 - algexact, 264
 - aliases, 239
 - arrays, 150
 - assume_pos, 12
 - assumescalar, 157
- B
 - backsubst, 257
 - backtrace, 192
 - batchkill, 209
 - batcount, 209
 - berlefact, 100
 - breakup, 266
- C
 - cauchysum, 117
 - compgrind, 204
 - context, 9
 - contexts, 9
 - current_let_rule_, 54
- D
 - dblnt_x, 294
 - dblnt_y, 294
- debugmode, 239
- demoivre, 128
- dependencies, 27, 28, 280
- derivabbrev, 272
- derivsubst, 273
- detout, 157
- dispflag, 192
- display2d, 226, 227, 278
- display_format_internal, 227
- doallmxops, 158
- domain, 70
- domxexpt, 158
- domxmxops, 158
- domxnct, 158
- dontfactor, 100
- doscmxops, 158
- doscmxplus, 158
- dot0nscsimp, 41
- dot0simp, 41
- dot1simp, 41
- dotassoc, 41
- dotconstrules, 41
- dotdistrib, 41
- dotexptsimp, 41, 275
- dotident, 41
- dotsassoc, 275
- dotscrules, 41, 275
- E
 - erfflag, 280
 - error_size, 232
 - error_syms, 232
 - errorfun, 192
 - expon, 60, 128
 - expop, 60, 128
 - exptdispflag, 227
 - exptisolate, 112
 - exptsubst, 123
- F
 - faceexpand, 100
 - factorflag, 100

- file_search_demo, 209
- file_search_LISP, 209
- file_search_Maxima, 209
- file_string_print, 209
- float2bf, 70
- fppintprec, 70
- fpprec, 70
- functions, 195
- G
- gcd, 96
- genindex, 117
- gensumnum, 117
- globalsolve, 257
- gradefs, 27, 28
- H
- halfangles, 250
- hermitianmatrix, 167
- I
- ibase, 227
- in_netmath, 325
- inchar, 223
- inflag, 110, 126, 137
- infolists, 234
- integrate_use_rootsof, 283
- integration_constant_counter, 278
- intfaclim, 100
- isolate, 112
- K
- keepfloat, 81
- knowneigvals, 167
- knowneigvects, 167
- L
- lasttime, 237
- leftmatrix, 167
- let_rule_packages, 54
- letrat, 54
- lhospitallim, 270
- ligarc, 253
- limsubst, 270
- linechar, 223
- linel, 227
- linenum, 223
- linsolve_params, 262
- linsolvewarn, 262
- listarith, 137
- listconstvars, 110
- listdummyvars, 110
- listeigvals, 167
- listeigvects, 167
- lmxchar, 158
- loadprint, 209
- logabs, 253
- logconcoeffp, 253
- logexpand, 253
- lognegint, 253
- lognumer, 253
- logsimp, 253
- M
- m1pbranch, 70
- macroexpansion, 200
- maperror, 143
- matrix_element_add, 159
- matrix_element_mult, 159
- matrix_element_transpose, 159
- maxapplydepth, 54
- maxapplyheight, 54
- maxnegex, 60, 130
- maxpogex, 130
- maxposex, 60
- modulus, 84
- multiplicities, 257
- myoptions, 239
- N
- newfac, 100
- nolabels, 223
- nondiagonalizable, 167
- O
- obase, 227
- opsubst, 123
- optimprefix, 122

- optionset, 239
- outchar, 223
- P
- packagefile, 209
- partswitch, 115
- pfeformat, 227
- piece, 115
- plot_options, 308
- ~ \mathcal{O} colour_z, 312
- ~ \mathcal{O} gnuplot_curve_styles, 318
- ~ \mathcal{O} gnuplot_curve_titles, 318
- ~ \mathcal{O} gnuplot_dumb_term_command, 318
- ~ \mathcal{O} gnuplot_pipe_term, 318
- ~ \mathcal{O} gnuplot_pm3d, 314
- ~ \mathcal{O} gnuplot_preambles, 319
- ~ \mathcal{O} gnuplot_term, 316
- ~ \mathcal{O} grid, 314
- ~ \mathcal{O} nticks, 313
- ~ \mathcal{O} plot_format, 311
- ~ \mathcal{O} t, 314
- ~ \mathcal{O} transform_xy, 312
- ~ \mathcal{O} view_direction, 312
- ~ \mathcal{O} x, 314
- ~ \mathcal{O} xlog, 314
- ~ \mathcal{O} y, 314
- ~ \mathcal{O} ylog, 314
- polyfactor, 260
- prederror, 65, 192
- prodhack, 116
- prompt, 221, 223, 231
- props, 29
- ps_scale, 325
- ps_translate, 325
- R
- radexpand, 70
- radsubstflag, 124
- rataigdenom, 81
- ratdenomdivide, 83
- ratepsilon, 81
- ratexpand, 83
- ratfac, 83
- ratmx, 157
- ratprint, 81
- ratsimpexpons, 83
- ratweights, 83
- ratwtlvl, 83
- readonly, 264
- resultant, 96
- rightmatrix, 167
- rmxchar, 158
- rombergabs, 293
- rombergit, 293
- rombergmin, 293
- rombergtol, 293
- rootsconmode, 83
- rootsepsilon, 260
- rpgrammode, 257
- S
- savedef, 204
- savefactors, 100
- scalarmatrix, 157
- show_openplot, 325
- showtime, 237
- simpsum, 117
- solve_inconsistent_error, 266
- solvedecomposes, 266
- solveexplicit, 266
- solvefactors, 266
- solvenullwarn, 266
- solveradcan, 266
- solvetricwarn, 266
- sparse, 157
- stardisp, 227
- sublis_apply_lambda, 125
- sumexpand, 117
- sumhack, 117
- T
- tlimswitch, 270, 291
- tr_array_as_ref, 205
- tr_bound_function_applyp, 206

tr_file_tty_messagesp, 206
 tr_float_can_branch_complex, 206
 tr_function_call_default, 205
 tr_numer, 205
 tr_optimize_max_loop, 206
 tr_semicompile, 205
 tr_warn_bad_function_calls, 206
 tr_warn_fexpr, 205
 tr_warn_meval, 205
 tr_warn_mode, 205
 tr_warn_undeclared, 205
 tr_warn_undefined_variable, 205
 transcomile, 205
 translate, 204
 translate_fast_arrays, 205
 transrun, 204
 trigexpandplus, 250
 trigexpandtimes, 250
 triginverses, 250
 trigsign, 250
 ttyoff, 227
 U
 undeclaredwarn, 204
 undeclearewarn の設定項目, 204
 V
 values, 108
 W
 window_size, 325

定数

%e, 61, 71
 %gamma, 71
 %phi, 71
 %pi, 71
 false, 71
 inf, 71, 280
 infinity, 71, 280
 minf, 71
 true, 71
 zeroa, 71
 zerob, 71

文脈

global, 8
 initial, 8

draw パッケージの関数

A
 add_zeros, 397
 D
 draw, 380
 ~ と draw2d, draw3d との関係, 380
 draw2d, 380
 draw3d, 380

draw パッケージのグラフの属性

A
 axis_3d, 386
 axis_bottom, 386
 axis_left, 386
 axis_right, 386
 axis_top, 386
 C
 columns, 382, 384
 E
 eps_height, 384
 eps_width, 384
 F
 file_name, 384
 G
 grid, 387
 L
 logx, 387
 logy, 387
 logz, 387
 P
 pic_height, 384
 pic_width, 384
 R
 rot_horizontal, 387
 rot_vertical, 387
 T
 terminal, 384
 title, 387

- X
 - x_range, 386
 - xtics, 387
 - xy_file, 387
- x
 - xlabel, 387
- Y
 - y_range, 386
 - ylabel, 387
 - ytics, 387
- Z
 - z_range, 386
 - zlabel, 387
 - Ztics, 387
- draw パッケージの大域変数
 - D
 - draw_command, 397
 - draw_pipes, 397
- draw パッケージの対象
 - E
 - ellipse, 394
 - ~ 構文, 394
 - ~ の属性, 395
 - ~ の border, 395
 - ~ の color, 395
 - ~ の fill_color, 395
 - ~ の key, 395
 - ~ の line_type, 395
 - ~ の line_width, 395
 - ~ の nticks, 395
 - ~ の transparent, 395
 - ~ 文, 394
 - explicit, 388
 - ~ の構文, 388
 - ~ の属性, 388
 - ~ の adapt_depth, 388
 - ~ の color, 388
 - ~ の fill_color, 388
 - ~ の filled_func, 388
 - ~ の key, 388
 - ~ の line_type, 388
 - ~ の line_width, 388
 - ~ の nticks, 388
 - ~ 文, 388
 - explicit3d, 389
 - ~ の構文, 389
 - ~ の属性, 390
 - ~ の color, 390
 - ~ の contour, 390
 - ~ の contour_levels, 390
 - ~ の enhance3d, 390
 - ~ の key, 390
 - ~ の line_type, 390
 - ~ の line_width, 390
 - ~ の xu_grid, 390
 - ~ の yv_grid, 390
 - G
 - gr2d, 380
 - gr3d, 380
 - I
 - image, 396
 - ~ 構文, 396
 - ~ の属性, 396
 - ~ の colorbox, 396
 - ~ の palette, 396
 - ~ 文, 396
 - implicit, 390
 - ~ の構文, 391
 - ~ の属性, 391
 - ~ の color, 391
 - ~ の ip_grid, 391
 - ~ の ip_grid_in, 391
 - ~ の key, 391
 - ~ の line_type, 391
 - ~ の line_width, 391
 - ~ 文, 390
 - L
 - label, 396
 - ~ 構文, 396
 - ~ の属性, 397

~ の label_allgment, 397
 ~ の label_orientation, 397
 ~ 文, 396
 label3, 396
 ~ 構文, 396

P

parametric, 391
 ~ の 構文, 391
 ~ の 属性, 391
 ~ の color, 391
 ~ の key, 391
 ~ の line_type, 391
 ~ の line_width, 391
 ~ の nticks, 391
 ~ 文, 391
 parametric3d, 391
 ~ の 構文, 392
 ~ の 属性, 392
 ~ の color, 392
 ~ の line_width, 392
 ~ の key, 392
 ~ の line_type, 392
 ~ の nticks, 392
 ~ 文, 391
 parametric_surface, 392
 ~ 構文, 392
 ~ の 属性, 393
 ~ の color, 393
 ~ の key, 393
 ~ の line_type, 393
 ~ の line_width, 393
 ~ の xu_grid, 393
 ~ の yv_grid, 393
 ~ 文, 392
 points, 393
 points3d, 393
 ~ の color, 393
 ~ の key, 393
 ~ の line_type, 393
 ~ の line_width, 393

~ の point_ljoined, 393
 ~ の point_size, 393
 ~ の point_type, 393
 points と points3d の 属性, 393
 ~ 構文, 393
 ~ 文, 393
 polar, 392
 ~ の 構文, 392
 ~ の 属性, 392
 ~ の color, 392
 ~ の key, 392
 ~ の line_type, 392
 ~ の line_width, 392
 ~ の nticks, 392
 ~ 文, 392
 polygon, 393
 ~ の 属性, 394
 ~ の border, 394
 ~ の color, 394
 ~ の fill_color, 394
 ~ の key, 394
 ~ の line_type, 394
 ~ の line_width, 394
 ~ の transparent, 394
 ~ の 構文, 394
 ~ 文, 393

R

~ 文, 393
 rectangle, 394
 ~ の 構文, 394

V

vector, 395
 ~ の 構文 (2d) , 395
 ~ の 構文 (3d) , 395
 ~ の 属性, 396
 ~ の color, 396
 ~ の head_angle, 396
 ~ の head_both, 396
 ~ の head_length, 396
 ~ の head_type, 396

~の key, 396
 ~の line_type, 396
 ~の line_width, 396
 ~文, 395
 vector3d, 395

え

演算子

- ~の型, 35
- ~の属性, 32
- ~の束縛力, 33
- ~の束縛力 (bp), 33
- ~の左束縛力 (lbp), 33
- ~の右束縛力 (rbp), 33
- 前置表現の~, 32
- 内挿表現の~, 32
- 後置表現の~, 32
- 無引数の~, 32

か

解の自動代入, 257

き

規則, 46

- ~の削除, 54
- ~の定義, 49
- ~の表示, 46

こ

項順序, 3

し

辞書式順序, 3

(Maxima の) 次数リスト, 3

述語, 8

主変数, 16

主変数 (mainvar), 3

そ

属性, 14

- ~値, 14
- ~値の削除, 29
- ~を追加, 15
- アトム属性, 16
- 関数属性, 20
- 数値属性, 18

ち

重複度のリスト, 258

な

並び方 (パターン), 48

並びの照合 (パターンマッチング), 48

ふ

文脈, 8

- ~の切替, 9
- ~の生成, 9
- ~を無効にする, 9
- ~を有効にする, 9
- ~の切替, 11
- 親文脈, 9
- ~の生成, 9
- ~の削除, 9

へ

変数, 2

- ~順序, 2

ほ

方程式, 256

も

文字, 2

れ

連立方程式, 256

GNUPLOT

A

- arrow, 360

C

- cbrange, 346
- clabel, 346
- cntrparam, 341
- contour, 339
- base, 339
- both, 339
- surfacee, 339

H

- hidden3d, 348

I

- isosamples, 352

K

- key, 356
- L
 - label, 358
- P
 - palette, 336
 - gray, 337
 - negative, 337
 - positive, 336
 - rgbformulae, 338
 - plot, 328
 - pm3d, 331
- dots, 330
- impulse, 330
- lines, 330
- linespoints, 330
- point, 330
- S
 - sample, 352
 - set, 327
 - show, 327
 - splot, 330
 - surface, 348
- T
 - ticslevel, 334
- U
 - unset, 327
- V
 - view, 344
- X
 - xrange, 353
- Y
 - yrange, 353
- Z
 - zrange, 353
- GNUPLOT 記号
 - '', 328
- LISP
 - G
 - great, 3, 5
- T
 - trigexpand, 251
 - trigrat, 251
 - trigreduce, 251
 - trigsimp, 251
- アプリケーション
 - G
 - Geomview, 301
 - gnuplot, 301
 - O
 - openmath, 301
 - S
 - SINGULAR, 6
- え
 - S 式
 - S 式, 66
- か
 - 可換積, 40
- し
 - CRE 表現, 77
- せ
 - 全微分, 272
- た
 - 多項式と単項式の一般表現, 76
 - 単変数多項式の CRE 表現, 77
- は
 - 媒介変数式, 305
- ひ
 - 非可換積, 40
- む
 - 無平方, 99
 - 無平方因子分解, 99