Maximaマニュアル改訂版

目 次

第1章　順序

1.1　順序について

変数の間に入れる順序というものがあります. この変数の順序は, A よりも B の方が大きい (先に置かれる) といった順番の事です. 多変数多項式の処理では, この順序が非常に重要な位置を占めます.

例えば,x+y と y+x は同じ式でしょうか？人間にとっては, 当然の事でも, 計算機にとっては違います. この式が, この順番の配列として入力されていれば, 全く違うデータになりますね. しかし, 演算子+の両端にある被演算子を入れ換えてやれば y+x は x+y になりますね. そこで, 計算機に, 演算子+に対し, 被演算子の入れ換え操作で同じデータになれば同じ式と判断すると教えてやる方法があります.

しかし,x+y の例では 2 項でしかなかったので問題は簡単ですが, 比較する式 a1 と b1 の項が 100 個になるとこの検証は大変な事になります.a1 の順序を固定して, b1 の項の入れ換えで a1 の並びに到達する迄に最悪の場合,100!個の処理が必要になります. ところが, 変数の間に順番を入れてしまえば, 両方の式を与えられた順番に沿って置換してしまえば良いので, 計算量も少なくて済みますし, 他の函数で順序の入れ換えを一々行う必要も特殊な場合を除くと必要が無くなります. 更に重要な事に, 式が項の入力の順番に関係無く一意に決まる事です. ではどの様に順番を入れるかと言えば, 一番参考になるのは辞書の並びです. a と z だと,a の方が z よりも先に,abuse と abnormal でば abuse の方が先にありますね. 先にあるという事を, 大きいと言い換えたものが辞書式順序です. 他にも様々な順序がありますが,Maxima では辞書を後から見る逆辞書式順序がデフォルトで採用されています.

この順序関係を上手く利用しているのが,Maxima で用いられている多項式の CRE 表現です.Maxima の命令で頭に rat の付いた函数が幾つかありますが, それらは内部で CRE 表現を用いています. その為, 一般的表現よりも迅速な処理が可能となっています.

1.2　色々な順序

ここでは多項式の項に入れる順序 (項順序) について解説します. Maxima では逆辞書式順序を用いているだけで, 他は実装されていませんが, 必要があれば Maxima を改造して入れるのも楽しいのではないでしょうか.

先ず, 多項式が一変数の場合,x^k と多項式の次数 k は一対一で対応します. 即ち, 一変数多項式だけを考えていれば, 項に順序を入れたければ, 次数の大小関係だけを見れば良い事になります.

n 変数の多項式の場合は, 最初の節で述べた様に, 同じ一次の項でも, $x_1, \cdots, x_n$ と n 個存在する為に, これらの一次の項に対しても順序を入れる必要があり, それからより一般の $x_1^{i_1} \cdots x_n^{i_n}$ についても考察する必要があります.

そこで, 変数にある順序 (例えばアルファベット順) を入れて項の変数順序を並び替えた項 $X_1^{i_1} \cdots X_n^{i_n}$ を考えると, この項とその次数リスト $(i_1, \cdots, i_n)$ は一対一に対応します.

例えば,x,y,z の 3 変数多項式の世界 ($K[x, y, z]$) では, 変数間の順番をアルファベット順で並べ,yz の様に変数が抜けている場合, 抜けている変数の次数を 0 とすると, 項は $x^{i_1} y^{i_2} z^{i3}$ の形になります. これから長さ 3 の次数のリスト (i_1, i_2, i_3) が得られ, このリストと項は一対一に対応します.

では,$x^2y^2z(= (2, 2, 1))$ と $xy^2z^3(= (1, 2, 3))$ の間の順序はどの様に入れれば良いでしょうか. 単純に x の多項式と見れば,x^2y^2z の方が大きいとも言えますが,$x = y = z$ とすると,xy^2z^3 の方が次数が大きくなります. この様に多変数多項式の場合, 項の順序には色々な考え方があります.

項順序として代表的なものに, 辞書式順序, 斉次辞書式順序, 逆辞書式順序, 逆斉次辞書式順序, これらの順序に, 変数の重みを加味したもの等があります. ここでは, 基本的な辞書式順序, 斉次辞書式順序, 逆辞書式順序と斉次逆辞書式順序について簡単に説明しましょう. 尚, 変数の並びは $x_1, \cdots x_n$ とし,$x_1^{a_1}, \cdots x_n^{a_n}$ の次数リストを $a = (a_1, \cdots, a_n)$ $x_1^{b_1}, \cdots x_n^{b_n}$ の次数リストを $b = (b_1, \cdots, b_n)$ とします. 更に,$|a| = a_1 + \cdots + a_n$ で次数リスト a の次数の総和を表記します.

辞書式順序

$$a > b \Leftrightarrow a_1 = b_1, \cdots, a_i = b_i, a_{i+1} > b_{i+1}$$

この関係で定められる順序を辞書式順序と呼びます. この順序を示す記号として,$>_{lex}$ が使われます.

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

例えば,x^2y^2z と xy^2z^3 を各々表現する整数の列 (2,2,1) と (1,2,3) に対しては, 先頭の 2 と 1 を比較した段階で,$2 > 1$ より, $(2, 2, 1) >_{lex} (1, 2, 3)$, 即ち,$x^2y^2z >_{lex} xy^2z^3$ となります.

斉次辞書式順序

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

この関係で定められる順序を斉次辞書式順序と呼びます. この順序を示す記号として,$>_{glex}$ が使われます.

この斉次辞書式順序の場合, 総次数で最初に項を比較し, 総次数が一致すれば, 今度は項を辞書式順序で比較する二段方式となっています.

例えば,x^2y^2z と xy^2z^3 を各々表現する整数の列 (2,2,1) と (1,2,3) に対しては,$|x^2y^2z| = 5$ と $|xy^2z^3| = 6$ となるので, $(1,2,3) >_{glex} (2,2,1)$, 即ち,$xy^2z^3 >_{glex} x^2y^2z$ となり, 辞書式順序とは逆の結果になります.

逆辞書式順序

$$a > b \Leftrightarrow a_n = b_n, \cdots, a_i = b_i, a_{i-1} < b_{i-1}$$

この関係で定められる順序を逆辞書式順序と呼びます. この順序を示す記号として,$>_{revlex}$ が使われます.

この逆辞書式順序では辞書を逆から見た順序と言えます. x^2y^2z と xy^2z^3 の場合、各々の項を表現する整数の列 (2,2,1) と (1,2,3) に対しては, 後の 1 と 3 を比較した段階で,$1 > 3$ から $(2,2,1) >_{revlex} (1,2,3)$, 即ち,$x^2y^2z >_{revlex} xy^2z^3$ が得られます.

斉次逆辞書式順序

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

この関係で定められる順序を斉次逆辞書式順序と呼びます. この順序を示す記号として,$>_{grevlex}$ が使われます.

この順序は項の総次数で最初に項を比較し, 総次数が等しければ, 逆辞書式順序で項の順序を決める二段方式のものです.

x^2y^2z と xy^2z^3 の場合、総次数が各々,5 と 6 になるので,$5 > 6$ から $xy^2z^3 >_{grevlex} x^2y^2z$ となり, 逆辞書式順序とは逆の結果になります.

1.3 Maximaでの順序

先程の節では簡単に項順序について解説しました. では, 肝心の Maxima の変数や項順序はどの様に入っているのでしょうか？

Maxima での変数順序を $>_m$ とすると, この順序は以下の様なものになっています.

Maxima の変数順序

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

アルファベットで開始する変数に関しては, デフォルトの状態で逆辞書順で並んでいます. アルファベットで開始する変数は頭文字が小文字よりも大文字のものが大で, 名前の長いものの方が大になります.

又, 定数, スカラーや主変数の宣言は,declare 函数を用いて, 変数に対して, constant,scalar,mainvar を各々宣言する事で行います.

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

```
(%i13) ordergreat(c,b);
(%o13)                                done
(%i14) ordergreat(b,z);
Reordering is not allowed.
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i15) unorder();
(%o15)                               [b, c]
```

この例で示す様に,$c > b$ という順序を入れましたが, その次に,$b > z$ 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() で ordergreat(c,b) で入れた変数順序を解除しています.

これは ordergreat や orderless で入れる順序によって,Maxima 全体の変数順序の変更を伴なうので,ordergreat や orderless 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の方が色々追加を行なって矛盾した順序が出来上る恐れがない分, すっきりしています.

これに対し,Maxima の項順序は, 先ず, 変数には辞書の逆の順番で順序が入っていますが, この変数順序に対し, 項順序自体は辞書式の順序となっています.

そこで今度は Maxima で変数順序や項順序がどの様に入っているか, 実際に調べてみましょう. 二つの与えられた式の順序を調べる函数として,Maxima には ordergreatp と orderlessp の二つがあります.ordergreatp(a,b) は a の順序が b よりも上の場合に true が返ります.

```
(%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
```

先ず, 変数順序について関しては,abc の方が a よりも辞書では後の頁に或る事で判る様に abc の方が a よりも大になります. 同様に ax の方が abc よりも大になります.

項順序では,z の方が y よりも大きい為, 次数とは無関係に z の方が $y\hat{2}$ 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,$z^2 * x * y^2$ と $z^2 * x * t^3$ の場合は, 頭から比較して,y の方が t よりも大である為, $z^2 * x * y^2$ の方が $z^2 * x * t^3$ よりも大になっています.

この様に,Maxima では変数の順序は ordergreat や orderless で多少の設定が可能ですが, 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すが,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と呼べます.

ここでは多項式の項順序について述べましたが,Maxima では, 通常の多項式に加えて exp や sin 等の初等関数にも同様の順序が入ります. 基本的に, 変数順序では, mainvar よりも初等関数が大となります.

利用者定義の函数に関しては, 一度 Maxima 側で値を解釈する為に,ordergreatp や orderlessp の値は, その状況に応じて変化します.

```
(%i77) neko(x):=if x<0 then x^2 else cos(x)^3;
                                                 2         3
(%o77)             neko(x) := if x < 0 then x  else cos (x)
(%i78) assume(p0>0);
(%o78)                               [p0 > 0]
(%i79) ordergreatp(cos(p0),neko(p0));
(%o79)                                 false
(%i80) assume(p1<0);
(%o80)                               [p1 < 0]
(%i81) ordergreatp(cos(p1),neko(p1));
(%o81)                                 true
(%i82) ordergreatp('neko(x),atan(x));
(%o82)                                 true
(%i83) ordergreatp(neko(x),atan(x));
Maxima was unable to evaluate the predicate:
x < 0
#0: neko(x=x)
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i84)
```

この例で示す様に, 利用者函数に単引用符を付けなければ,Maxima で解釈が実行され, その結果で ordergreatp 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は名詞型の場合, 引数も含めた函数名で, 逆アルファベット順で比較が実行されます.

1.4 順序に関連する函数

ordergreat

ordergreat ($\langle v_1\rangle, \cdots, \langle v_n\rangle$)

指定した変数 $\langle v_1\rangle, \cdots, \langle v_n\rangle$ に順序を入れます. この順序は左から順に,$v_1 > v_2 > \cdots > v_n$ とし, 変数 v_n の下に, ordergreat の引数に含めていない変数が来る様に全体に順序を入れます.

ordergreatp

ordergreatp ($\langle$ 式$_1\rangle, \langle$ 式$_2\rangle$)

ordergreat 函数で設定された順序で,$\langle$ 式$_2\rangle$ が $\langle$ 式$_1\rangle$ よりも順序が小さければ true を返し, そうでなければ false を返します.

orderless

orderless ($\langle v_1\rangle, \cdots, \langle v_n\rangle$)

ordergreat の逆で順序を入れます. 即ち, 変数 $\langle v_1\rangle, \cdots, \langle v_n\rangle$ に対し, $v_1 < v_2 < ... < v_n$ で順序を入れ, 引数に含めていない変数よりも $\langle v_n\rangle$ が下となる様に全体に順序を入れます.

orderlessp

orderlessp ($\langle$ 式$_1\rangle, \langle$ 式$_2\rangle$)

orderless 命令によって設定された順序で $\langle$ 式$_1\rangle$ が $\langle$ 式$_2\rangle$ よりも小さければ true を返します.

unorder

unorder()

ordergreat や orderless 函数で定めた順序を無効にします.

第2章　規則

2.1　規則と式の並びについて

MAXIMA では数式を扱う際に様々な規則に従って処理を進めます. 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には, 大域変数の rules に規則名のリストが設定されています. より詳しく知りたい場合は disprule(⟨ 規則名 ⟩) で規則を表示させます. ここで引数に all を指定すると rules に含まれる規則全てが表示されます.

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

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

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

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

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

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

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

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

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

disprule 函数を使って表示される規則の詳細では, 先ず, 左端が規則名で, ->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,trigrule0 は $\tan x$ を $\frac{\sin x}{\cos x}$ で置換える規則です.

ここで,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. 更に, 実際の利用では函数 tan は式の中にあって, 引数も x の様な単純なものではありません. 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行います. この処理を並び照合 (パターンマッチング) と呼びます. この様な処理を行う函数として apply1 や apply2 等の函数があります.

例えば, 函数 tan(x) の規則 trigrule0 は引数 x はそのままで, 単純に sin(x)/cos(x) で置換える事を意味します. その為,apply1 で函数 tan を含む式に trigrule0 を適用すると, 式の中の tan を見付けると,trigrule0 を適用して sin(x)/cos(x) で置換えてしまいます.

```
(%i19) tan(x);
(%o19)                              tan(x)
(%i20) apply1(tan(x),trigrule0);
                                    sin(x)
(%o20)                              ------
(%i21) apply1(tan(a1*x+y+b1),trigrule0);
                              sin(y + a1 x + b1)
(%o21)                        ------------------
                              cos(y + a1 x + b1)
                                   cos(x)
```

この規則は利用者が定義する事も可能です. 規則の設定は defrule 等で行えます. この時, 変数の条件に応じて規則の適用を制限したい事があります. 例えば,x^2 の平方根は,x が実数で, 負の数でなければ x で置換える規則がこれに当ります. この場合, 並び変数に対する述語函数があれば,let 函数を用いて規則を導入する事が出来ます. この let 函数の構文は, let(⟨ 項 ⟩, ⟨ 式 ⟩, ⟨ 述語 ⟩, ⟨ 引数$_1$⟩, $\cdots$, ⟨ 引数$_n$⟩)

で与えられます. ここで,⟨ 述語 ⟩ の後の ⟨ 引数$_1$⟩, $\cdots$, ⟨ 引数$_n$⟩ は述語函数の引数で, この中に, 並び変数も含まれます.

この節ではユーザ定義のパターン照合と簡易化の規則 (tellsimp,tellsimpafter, defmatch 又は defrule で設定したもの) について述べます.

この設定によって,MAXIMA 本体の簡易化に影響を与えたり,apply1 や apply2 を使う際に, この規則が表で適用される事があります.

2.2　関連する大域変数

maxapplyheight

デフォルト値:[10000]

apply1,apply2 や applyb1 が停止する与式の最高度となります. 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 よりも低い個所に作用し, それよりも高ければ作用しません. 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.

default_let_rule_package

デフォルト値:[default_let_rule_package]

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

letrat

デフォルト値:[false]

false であれば,letsimp は式の分子と分母を各々別に簡易化し, 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,letrat を true に設定すべきです. すると, 分子, 分母の商は要求の通りに簡易化されます.

let_rule_packages

デフォルト値:[default_let_rule_]package]

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.

尚,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.

2.3 規則に関連する函数

apply1

apply1(⟨ 式 ⟩, ⟨ 規則$_1$⟩, · · · , ⟨ 規則$_n$⟩)

apply1 は与えられた ⟨ 式 ⟩ に対し, 最初の ⟨ 規則$_1$⟩ を作用させます. この時,⟨ 式 ⟩ に含まれる全ての部分式に対して, ⟨ 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$_2$⟩ を同様に適用させます. 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.

```
                                sec (c) + tan(b)
(%i118) apply1(tan(a)/(sec(c)+tan(b)),trigrule1,trigrule2);
                                    sin(a)
(%o118)                     ------------------------
                                 1      sin(b)
                            cos(a) (------ + ------)
                                   cos(c)   cos(b)
```

apply2

apply2(⟨ 式 ⟩, ⟨ 規則$_1$⟩, · · · , ⟨ 規則$_n$⟩)

⟨ 規則$_1$⟩ が ⟨ 式 ⟩ の部分式で失敗すると, ⟨ 規則$_2$⟩ が適用させられる事で apply1 と異なります. 全ての部分式で失敗した時に限って, 全ての規則の集合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ると, その同じ部分式が ⟨ 規則$_1$⟩ で再実行されます. ここで,maxapplydepth は apply1 と apply2 が実行される最高度となります.

applyb1

applyb1(⟨ 式 ⟩, ⟨ 規則$_1$⟩, · · · , ⟨ 規則$_n$⟩)

apply1 と似ていいますが,apply1 が ⟨ 式 ⟩ の階層構造の上から下 (全体 → 部分式 → · · ·) であるのに対し, applyb1 は ⟨ 式 ⟩ の最下層にある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.

letsimp

letsimp(⟨ 式 ⟩)

letsimp(⟨ 式 ⟩, ⟨ 規則パッケージ名 ⟩)

letsimp(⟨ 式 ⟩, ⟨ 規則パッケージ名$_1$⟩, · · · , ⟨ 規則パッケージ名$_n$⟩)

⟨ 式 ⟩ が規則パッケージに含まれる規則の適用により, 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, デフォルトで rule_pkg_name が用いられます.

defmatch

defmatch(⟨ プログラム ⟩, ⟨ 並び ⟩, ⟨ 助変数$_1$⟩, · · · , ⟨ 助変数$_n$⟩)

n+1 個の引数を持ち, 特定の並びに適合するかどうか式を検査する函数で, 名前が ⟨ プログラム ⟩ のものを生成します.⟨ 並び ⟩ は変数と助変数を含む式になります. 変数が以前の matchdeclare 函数の中で暗示的に与えられていたとしても,⟨ 助変数$_i$⟩ は引数としてはっきりと defmatch に与えられます.

函数の最初の引数は並びに対して照合する式で, 他の n 個の引数は式中の実際の値, 並びの中で変数として置かれるものです.defmatch 内の助変数は FORTRAN の SUBROUTINE でのダミー変数に似ています.

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

次の例では, 与えられた函数の線形性を調べる函数 linear を定義し, 実際に実行する例です.

```
(%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 を定義していますが, 次で, 式 $3*z+(y+1)*z+y^2$ が変数 z の一次式であるか検証していますが,false が返却されています. これは最初の a と b に関して何等の情報も無い為, 単純に a と 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 である判断した為に `[x = z]` が返却されています.

そこで,a と b の情報を追加します. この例では, 最初に is(e#0 and freeof(x,e)) と同値な函数 nonzeroandfreeof を定義し, それから変数 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 を用いて宣言しています. この宣言の後に defmatch で函数 linear を再定義します. この再定義を行わないと, a,b に関する情報は更新されません. これによって, 変数 a と b が x と独立したもので, x に対して線形であれば対応するもののリストを返す函数 linear が定義されます.

次に,`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.

defrule

defrule (⟨ 規則名 ⟩,⟨ 並び ⟩,⟨ 置換 ⟩)

与えられた ⟨ 並び ⟩ に対し,⟨ 置換 ⟩ を行う規則の定義とその名付けを行います. この場合,⟨ 規則名 ⟩ で指定された規則が apply 函数の一つによって式に適用されると,⟨ 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えられます.

⟨ 並び ⟩ で値が割当てられた ⟨ 置換 ⟩ 中の全ての変数は, 簡易化された ⟨ 置換 ⟩ の中の値が割当てられます. 規則自体は並びの照合と置換操作による式の変換函数として扱う事が可能です. この函数は並びの照合に失敗すると元の式を返却します.

```
(%i9) defrule(hiyo,tama(x+y),tama(x)+tama(y)*x);
(%o9)              hiyo : tama(y + x) -> x tama(y) + tama(x)
(%i10) tama(x+y);
(%o10)                            tama(y + x)
(%i11) apply1(tama(x+y),hiyo);
(%o11)                         x tama(y) + tama(x)
```

disprule

disprule (⟨ 規則$_1$⟩,⟨ 規則$_2$⟩,$\cdots$)

disprule(all)

defrule,tellsimp,tellsimpafter によって与えられた規則, 規則名や, defmatch によって定義された並びを名前込みで表示します.

disprule(all) は全ての規則を表示します.

```
(%i12) disprule(hiyo);
(%o12)              hiyo : tama(y + x) -> x tama(y) + tama(x)
(%i13) disprule(all);
                                              sin(a)
(%t13)                   trigrule0 : tan(a) -> ------
                                              cos(a)

                                              sin(a)
(%t14)                   trigrule1 : tan(a) -> ------
                                              cos(a)

                                                1
(%t15)                   trigrule2 : sec(a) -> ------
                                              cos(a)

                                                1
(%t16)                   trigrule3 : csc(a) -> ------
```

```
                                          sin(a)

                                          cos(a)
(%t17)                 trigrule4 : cot(a) -> ------
                                          sin(a)

                                           sinh(a)
(%t18)                htrigrule1 : tanh(a) -> -------
                                           cosh(a)

                                             1
(%t19)                htrigrule2 : sech(a) -> -------
                                           cosh(a)

                                             1
(%t20)                htrigrule3 : csch(a) -> -------
                                           sinh(a)

                                           cosh(a)
(%t21)                htrigrule4 : coth(a) -> -------
                                           sinh(a)

(%t22)           hiyo : tama(y + x) -> x tama(y) + tama(x)

(%o22)                              done
```

let

let (⟨ 項 ⟩,⟨ 式 ⟩)

let (⟨ 項 ⟩,⟨ 式 ⟩,⟨ 述語 ⟩,⟨ 変数$_1$⟩,$\cdots$,⟨ 変数$_n$⟩)

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

letsimp で ⟨ 述語 ⟩ が true の場合に ⟨ 項 ⟩ を ⟨ 式 ⟩ で置換する置換規則を定義します. 尚,⟨ 述語 ⟩ と ⟨ 変数 ⟩ を省く事も可能ですが, この場合,⟨ 項 ⟩ に含まれる変数が matchdeclare によって true と宣言されている必要があります.

let の引数の末尾に ⟨ パッケージ名 ⟩ を追加する事で, この置換規則をパッケージに追加します.

1. letsimp が文字として検索するアトムで, 以前,letsimp を呼んだ事がなければ, matchdeclare 函数をアトム と述語を繋げる為に用います. この場合,letsimp はアトムで表現された述語を満す, 任意の積の項に対して適合します.

2. sin(x),n!,f(x,y) 等の様な核.matchdeclare が核の引数と述語を繋げる為に用いられる迄, 上述

の様に letsimp は綴でアトムと一致するものを探します. 正の冪乗に対する項は, 少なくとも,letsimp された式中に冪乗を持つ項のみに適合します. 一方で, 負の冪乗に対する項は少なくとも, 負の冪乗を持つ項にのみ適合します. 積中の負の冪乗の場合, 大域変数 letrat は true に設定されていなければなりません. 何故なら, 述語が引数のリストが続く let 函数の中の含まれていれば, 従属的な適合 (つまり, 述語が省略されていたとしても受容されるもの) は,argi' を argi に適合する値とした場合,predname(arg1',...,argn') が true に評価される時に限って受容されます.argi は ⟨ 積 ⟩ の中に現われる任意のアトム名か任意の核の引数で構いません.⟨ 積 ⟩ からの任意のアトムや引数は ⟨ 有理式 ⟩ 中に現われると妥当な置換が行われます.

これらの置換函数は一度に幾つかの規則の組合せを用いて作用させられます. 各々の規則の組合わせは任意数の let で操作された任意の数の規則を含む事が可能で, 利用者が与えた名前で参照されます.

letrules

letrules()

引数を取らない函数で, 現行の規則の組合わせで規則を表示します. letrules(⟨ 規則名 ⟩) は名前が ⟨ 規則名 ⟩ の組合せ規則を表示します. 現在の規則の組み合わせは current_let_rule_package の値であり, その規則の初期値は default_let_rule_]package です.

letsimp

letsimp(⟨ 式 ⟩)

letsimp(⟨ 式 ⟩, ⟨ パッケージ名$_1$⟩, $\cdots$)

規則を適用して ⟨ 式 ⟩ が変化しなくなるまで, 規則を適応し続けます. ⟨ パッケージ名 ⟩ を省略した場合は, パッケージとし, current_let_rule_package 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,package_1,package_2)` を実行すると, 最初に `letsimp(expr,package_1)` が実行され, それから, `letsimp(%,packgae_2)` が実行されます. この場合,current_let_rule_package は切替えられません.

matchdeclare

matchdeclare (⟨ 並び変数 ⟩, ⟨ 述語 ⟩, $\cdots$)

並びの照合を行う際に, 変数の適用に制限を加える為に, 並び変数にと述語を結び付けます. 即ち,⟨ 述語 ⟩ が false でない式にだけ, ⟨ 並び変数 ⟩ が適合する様にします.

例えば,matchdeclare(q,freeof(x,%e)) が実行された場合,q は x と%e を含まない全ての式に適合します. これは let 等で規則を設定する際に. 式に q を用いると, 述語が成立する場合に,letsimp 等によって規則が適用される事になります.

remlet

remlet(⟨ 項 ⟩)

remlet(⟨ 項 ⟩, ⟨ パッケージ名 ⟩)

remlet(all)

remlet()

let 函数で定義された最新の代入規則 ⟨ 項 >→< 式 ⟩ を削除します.⟨ パッケージ名 ⟩ が与えられると, 規則パッケージ名から規則が削除されます.

remlet() と 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.

規則パッケージの名前が, 例えば,relmet(all,⟨ パッケージ名 ⟩) で与えられていれば, 指定された規則パッケージも削除されます. 代入が (最初に) 同じ生成物を用いて変更されるものであれば remlet が呼出される必要は無く, 単に let 函数と新しい代入と述語名を持った同じ生成物を用いる代入の再定義を行います.

remlet(生成物) が呼出されると本来の置換規則が復活します.

remrule

remrule (⟨ 函数 ⟩, ⟨ 規則 ⟩)

remrule (all)

⟨ 規則 ⟩ で指名された規則を defrule,defmatch,tellsimp や tellsimpafter で設定された ⟨ 函数 ⟩ から削除します.

引数が all の場合, 全ての規則が削除されます.

tellsimp

tellsimp(⟨ 並び ⟩, ⟨ 置換 ⟩)

tellsimpafter に類似していますが, 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される様にします.tellsimp は簡易化が実行される前に式を改変する事が重要な時に用いられます. 例えば, 簡易化が式について何かを知っているが, それが返すものが期待したものではない場合です.

簡易化が式の主演算子について何かを知っていても, 十分な簡易化でなければ, 恐らく tellsimpafter を使う事になるでしょう. 並びは和, 積, 単変数や数値であってはなりません. 規則は名前のリストで名前は簡易化の規則を持ち, それらは defrule,defmatch,tellsimp や tellsimpafter で追加されたものです.

tellsimpafter

tellsimpafter (⟨ 並び ⟩, ⟨ 置換 ⟩)

⟨ 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 の簡易化が組込みの簡易化規則の適用後に用いるものです.⟨ 並び ⟩ には単変数や数を除く任意の式が設定出来ます.

第3章　数値

3.1　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. 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 (fixnum) と可変桁の整数 (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.float 型から bigfloat への変換は函数 bigfloat 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. 但し,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 $x^2 - 4 * x + 13 = 0$ の根は 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, 虚部は imagpart 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.

大域変数 domain は函数等の動作に影響を与えます.domain はデフォルトでは real となっています. 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,complex にすると Maxima が処理する世界は複素数の世界となる事を意味します.

3.2　数値に関連する大域変数

domain

デフォルト値:[real]

Maxima の多項式や函数で扱う係数や変数の領域を定めます. デフォルトの real は全てが実数で処理されます. complex を指定すると, 複素数上で全てが処理されます.

domain を complex,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.

float2bf

デフォルト値:[false]

浮動小数点数を bigfloat 型に変換する際に警告を出すかどうかを決定します.

fpprec

デフォルト値:[16] 桁.

bigfloat 型の精度 (桁数) を指定します.

fpprintprec

デフォルト値:[0] 桁.

bigfloat 型の数値を (実際よりも) 小さな桁数で表示する際の表示桁数を指定します.

m1pbranch

デフォルト値:[false]

大域変数 domain と一緒に利用し,-1 の原始 n 乗根への自動変換を制御します.

大域変数 domain が real の場合,mlpbranch の設定に関係無く, (-1)^(1/n) は n が奇数であれば-1 に自動変換されますが, それ以外はそのままです.

大域変数 domain が complex で,m1pbranch が true の場合のみ, (-1)^(1/n) は-1 の原始 n 乗根に自動的に変換されます.

radexpand

デフォルト値:[true]

true の場合, 因子の n 乗根の積で n 乗のものを根号の外に出します.　例えば,sqrt(16*x^2) は radexpand が true の場合に限って 4*x となります.

sqrtdispflag

デフォルト値:[true]

false ならば,sqrt は 1/2 乗の形式で表示されます.

```
(%i38) sqrtdispflag;
(%o38)                                true
(%i39) sqrt(x);
```

```
(%o39)                                sqrt(x)
(%i40) sqrtdispflag:false;
(%o40)                                 false
(%i41) sqrt(x);
                                        1/2
(%o41)                                 x
```

3.3　数値に関連する函数

bfloat

bfloat(⟨ 数値 ⟩)

全ての数と数値函数を bigfloat 型に変換します. 大域変数 fpprec を n に設定すると, bigfloat 型の数値精度は n 桁になります.

尚, 大域変数 float2bf が false の場合, 浮動小数点数を bigfloat 型の数値に変換する時に計算精度が落るとの警告メッセージが表示されます.

bfloatp

bfloatp(⟨ 式 ⟩)

⟨ 式 ⟩ が bigfloat 型の数値であれば true, それ以外は false を返します.

cabs

cabs(⟨ 式 ⟩)

複素数の絶対値を返します.

entier と fix

fix(⟨ 数値 ⟩)

entier(⟨ 数値 ⟩)

fix と entirer はこれと同じ物で,⟨ 数値 ⟩ が実数の場合,⟨ 数値 ⟩ を越えない最大の整数 n を返します.

```
(%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
```

この例で示す様に, 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.

floatnump

floatnump(⟨ 式 ⟩)

⟨ 式 ⟩ が浮動小数点数であれば true, それ以外は false となります.

max と min

max(⟨ 数値$_1$⟩, ⟨ 数値$_2$⟩, $\cdots$)

min(⟨ 数値$_1$⟩, ⟨ 数値$_2$⟩, $\cdots$)

max は複数の実数引数を取り, それらの最大値を返します. それに対して,min は複数の実数引数を取り, それらの最小値を返します.

evenp と oddp

evenp(⟨ 数値 ⟩)

oddp(⟨ 数値 ⟩)

evenp は引数が偶数ならば true, その他は false を返します. oddp は引数が奇数であれば true を返し, それ以外は false を返します.

integerp

integerp (⟨ 式 ⟩)

⟨ 式 ⟩ が整数ならば true, それ以外は false を返します.

numberp

numberp (⟨ 式 ⟩)

⟨ 式 ⟩ が整数, 有理数, 浮動小数や可変長実数であれば true, それ以外は false となります.

realpart と imagpart

realpart(⟨ 式 ⟩)

imagpart(⟨ 式 ⟩)

⟨ 式 ⟩ を展開し,%i を含まない部分を実部,%i の係数を虚部とすると, realpart で ⟨ 式 ⟩ の実部を返します.imagpart で ⟨ 式 ⟩ の虚部を返します.

```
(%i34) realpart(log(2+%i));
                                  log(5)
(%o34)                            ------
                                    2
(%i35) imagpart(log(2+%i));
                                       1
(%o35)                            atan(-)
                                       2
(%i36) realpart((x+%i)^2);
                                   2
(%o36)                            x  - 1
```

```
(%i37) imagpart((x+%i)^2);
(%o37)                                2 x
```

random　random(⟨ 数値 ⟩)
　引数として ⟨ 数値 ⟩ を一つ取り,0 から ⟨ 数値 ⟩-1 の間の整数の乱数を返します.

isqrt
　isqrt(⟨ 整数 ⟩)
　⟨ 整数 ⟩ に対して, そ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
```

　この函数は, 以下の函数と同等の働きをします.

```
myisqrt(x) := block(local(a),
                    a : fix(bfloat(sqrt(abs(x)))),
                    return(a));
```

sqrt
　sqrt(⟨ 式 ⟩)
　⟨ 式 ⟩ の平方根を返します. 尚,Maxima 内部では x(1/2) で表現されています.

?round
　?round(⟨ 数値 ⟩,&オプション)
　⟨ 数値 ⟩ を最も近い整数に丸めます.⟨ 数値 ⟩ は float 型で,bigfloat 型ではありません.

?truncate
　?truncate(⟨ 数値 ⟩,&オプション)
　float 型の数値を引数とし, 小数点以下を切り捨てます.

第4章　Maximaの定数

4.1　定数について

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.

%e
指数の自然底 e を Maxima では%e で表記します. 尚,e^x は%e^x や exp(x) で表現します.

%pi
円周率 π を Maxima では%pi と表記します.

false
bool 代数の定数. 偽 (LISP の verb+nil+) を表現します.

true
bool 代数の定数. 真 (LISP の t) を表現します.

inf
正の実無限大.

minf
負の実無限大.

infinity
複素無限大, 任意の偏角で無限の大きさを持ちます.

zeroa と zerob　limit で用いる定数で,zeroa が正の微小量と zerob が負の微小量を意味します.

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

尚,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.

第5章　リスト

5.1　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, 多くの数式処理でも採用されています.

Maxima のリストは `[1, 2, 7, x+y]` の様に, 各成分をコンマ, で区切った列を大括弧 `[]` で括ったものです. この様に,LISP のリストの様に, 成分を単純に空行で区切るのではない事に注意をして下さい. 勿論,`[1,2,[3,4],[4,[5]]]` の様に, リストをリストの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なので,Maxima の式やプログラムも前節で紹介した様に内部的には LISP のリストで表現されています. 尚,Maxima の演算子が前置式でなくても, 内部的には前置表現になっている為, Maxima 内部表現では演算子の部分が 0, それ以降の成分に $1, 2, \cdots$ と番号が振られた階層構造を持っています.

5.2　リスト処理に関連する大域変数

listarith

デフォルト値:[true].

false であれば, 算術演算子によるリストの評価が抑制されます.

inflag

デフォルト値:[false].

true であれば, 成分を取り出す函数は与えられた式の内部表現に対して処理を行います. 簡易化は式の再並び換えを行う事に注意が必要になります. その為,`first(xy)+`は inflag が true であれば x となりますが,inflag が false ならば y となります.inflag の設定によって影響を受ける函数は, part,substpart,first,rest,
last,length,for-in 構文,map,fullmap,maplist,reveal と pickapart です.

5.3 リスト処理に関連する主な函数

atom と listp

atom(⟨ 変数 ⟩)

listp(⟨ 変数 ⟩)

⟨ 変数 ⟩ がアトムであるかリストであるかは atom と listp で各々調べる事が出来ます.atom は引数がアトムであれば true を返し, それ以外は false を返します. listp 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 の様な式は, 各変数に値が束縛されていなければアトムでもリストにもなりません.

member 函数

member(⟨ 式$_1$⟩,⟨ 式$_2$⟩)

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.⟨ 式$_2$⟩ はリストでも構いません.

```
(%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
```

length

length(⟨ リスト ⟩)

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ます. 要するに, 一番上の階層のリストの個数を返します. 従って, リストの中のリストはそれを一つとして数えられます.

copylist

リストの複製を行います

reverse

reverse(⟨ リスト ⟩)

リストの並びを逆にします. 例えば, リスト [a1,a2,a3] に対して
reverse([a1,a2,a3]) で作用させると,[a3,a2,a1] が返されます. 但し, 各成分 (各 ai) 自体はそのままです. この函数もリストではなく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

append(⟨ リスト$_1$⟩,⟨ リスト$_2$⟩,$\cdots$)

複数のリストの結合を行います. この append は LISP の append と同様の事をします.

endcons

endcons(⟨ 式$_1$⟩,⟨ リスト$_1$⟩)

append と似た函数ですが,append が先頭に追加するのに対し,endcons は後に追加します. 即ち,econs(⟨ 式$_1$⟩,⟨ リスト$_1$⟩) で ⟨ 式$_1$⟩ を ⟨ リスト$_1$⟩ の後に追加します. 但し,Maxima の式は内部的にはリストの為, econs(⟨ 式$_1$⟩,⟨ 式$_2$⟩) とすると ⟨ 式$_2$⟩ に ⟨ 式$_1$⟩ が追加されます.

```
(%i43) endcons(x+y,[1,2,3,4]);
(%o43)                          [1, 2, 3, 4, y + x]
(%i44) endcons(x+y,sin(x)+cos(y));
(%o44)                        cos(y) + y + sin(x) + x
(%i45) endcons(x+y,sin(x)/cos(y));
                                 sin(x) (y + x)
(%o45)                           --------------
                                      cos(y)
(%i46) endcons(x+y,sin(x)*cos(y));
(%o46)                         sin(x) (y + x) cos(y)
(%i47) endcons(x+y,sin(x)-cos(y));
(%o47)                       - cos(y) + y + sin(x) + x
```

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

delete(⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨n⟩)

⟨ 式$_2$⟩ に含まれる ⟨ 式$_1$⟩ を頭から ⟨n⟩ 個削除します. 但し,⟨ 式$_1$⟩ が函数, 或いは単項式でなければ除去出来ません. 尚,⟨n⟩ はオプションで, 指定しない場合と ⟨ 式$_2$⟩ に含まれる ⟨ 式$_1$⟩ が ⟨n⟩ 個よりも少ない場合, ⟨ 式$_1$⟩ を全て削除します.

rest

rest(⟨ 式$_1$⟩,⟨n⟩)

⟨n⟩ が正整数であれば,⟨ 式$_1$⟩ の頭から n 個の成分を除いた式を返し,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 ]
```

first

first(⟨ 式 ⟩)

⟨ 式 ⟩ の最初の成分を返します.

last

last(⟨ 式 ⟩)

⟨ 式 ⟩ の最後の成分を返します. 尚,rest,first と last に関しては, 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 はデフォルトで 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 に変更します.

sublist

sublist (⟨ リスト ⟩,⟨ 函数 ⟩)

⟨函数⟩がtrue を返す⟨ リスト ⟩に含まれる成分を抜出したリストを返します. 例えば, sublist([1,2,3,4],evenp); は [2,4] を返します.

substpart

substpart(⟨x⟩,⟨ 式 ⟩,⟨n_1⟩,$\cdots$,⟨n_k⟩)

⟨ 式 ⟩ の ⟨n_1⟩,$\cdots$,⟨n_k⟩ で指定された成分を ⟨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⟩,$\cdots$,⟨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] の第二の成分 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 で置換すると [1,x,4] となります. 第二成分に含まれる 3 を x で置換えたければ, 第二成分のリストの第二成分を指定すれば良い事になります.

この事は Maxima の一般の式に対しても適応が可能です. 但し,Maxima の場合は入力した式の順番と,Maxima に入力された後の順番が異なる事がある為, 注意が必要になります.

```
(%i16) expr:(x+1)/(x^2+x+1)+exp(x);
                                  x     x + 1
(%o16)                           %e  + ----------
                                         2
                                        x  + x + 1
(%i17) substpart(sin(x),expr,1);
                                           x + 1
(%o17)                          sin(x) + ----------
                                           2
                                          x  + x + 1
(%i18) substpart(sin(x),expr,2);
                                              x
(%o18)                              sin(x) + %e
(%i19) substpart(sin(x),expr,2,2);
                                    x + 1      x
(%o19)                              ------ + %e
                                    sin(x)
(%i20) substpart(sin(x),expr,2,1);
                                    sin(x)       x
(%o20)                             ---------- + %e
                                   2
                                  x  + x + 1
(%i21) substpart("+",expr,2,0);
                                    x    2
(%o21)                             %e  + x  + 2 x + 2
```

この例では式 expr の成分を sin(x) で置換える事を行っています. ここで第一成分は入力した順序とは異なり $\%e^x$ 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, 最後に 2,0 で演算子となります. 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.

第6章　文字列

6.1　Maximaの文字列

文字列は配列の一種です. その為, 文字列やMaximaで定義したオブジェクト名も配列になります. この文字列から指定した位置の文字を返す函数にgetcharがあります. この函数は getchar(文字列, 位置) で指定した位置の文字を返します.

getchar

getchar(⟨ 文字列 ⟩, ⟨ 整数 ⟩)

与えられた文字列の ⟨ 整数 ⟩ で指定された位置の文字を取出します.

concat

concat(⟨ 引数$_1$⟩, $\cdots$, ⟨ 引数$_n$⟩)

引数を評価し, その値を結合したものを文字列として返します. 尚, 引数に割当てられた値はアトムでなければなりません.

```
(%i138) a1:128;
(%o138)                                128
(%i139) concat(1,a1,2)+1;
(%o139)                              11282 + 1
(%i140) :lisp $%o139;
((MPLUS SIMP) 1 &11282)
```

sconcat

sconcat(⟨ 引数$_1$⟩, $\cdots$, ⟨ 引数$_n$⟩)

引数を評価て文字列に結合します.concat とは違い, 引数に割当てられた値がアトムである必要はありません. 結果は Common LISP の文字列となります.

```
(%i140) eq1:x^2+1=0;
                                    2
(%o140)                            x  + 1 = 0
(%i141) sconcat(eq1,"<->","x=[-%i,%i]");
(%o141)                      x^2+1 = 0<->x=[-%i,%i]
(%i142)
(%i142) :lisp $%o141;
x^2+1 = 0<->x=[-%i,%i]
```

string

string (⟨ 式 ⟩)

⟨ 式 ⟩ を文字列に変換します.

```
(%i123) eq1:x^2+1=0;
                                    2
(%o123)                            x  + 1 = 0
(%i124) str1:string(eq1);
(%o124)                             x^2+1 = 0
```

```
(%i125) :lisp $eq1;
((MEQUAL SIMP) ((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2)) 0)
(%i125) :lisp $str1;
&X^2+1 = 0
```

第7章　配列

7.1　Maximaの配列について

Maxima はリストの他に配列が扱えます.

Maxima で配列を生成する場合, 幾つかの方法があります. 先ず,array 函数で配列を宣言して具体的な値を割当てて行く方法と, 配列を宣言せずに, 直接値を割当てて行く方法を説明しましょう.

```
(%i30) array(a,2,2);
(%o30)                                a
(%i31) a[0,0]:2;
(%o31)                                2
(%i32) a[2,2]:4;
(%o32)                                4
(%i33) listarray(a);
(%o33) [2, #####, #####, #####, #####, #####, #####, #####, 4]
(%i34) b[0,0]:128;
(%o34)                               128
(%i35) b[0,123]:128;
(%o35)                               128
(%i36) listarray(b);
(%o36)                          [128, 128]
(%i37) b[0,123];
(%o37)                               128
(%i38) arrayinfo(a);
(%o38)                    [declared, 2, [2, 2]]
(%i39) arrayinfo(b);
(%o39)                [hashed, 2, [0, 0], [0, 123]]
```

この例では,array を用いて配列 a を定義し, その次には成分を直接指定する事で配列 b を生成しています. 但し, 内部的には array を用いて生成した配列は LISP の配列で, その大きさは変更出来ませんが, 配列 b の様に成分を指定する事で生成された配列は LISP のハッシュ表であり, その大きさは動的に変更可能となります.

実際,listarray で配列の内容,arrayinfo で配列の属性と内容を見る事が可能です. 最初の listarray では配列の中身が表示れます.a の場合は未設定の個所に#####が表示されていますが, 配列 b では,

値の設定されている個所のみが表示されています. 更に,arrayinfo で表示すると, 配列 a では先頭に `declared` と表示されています. これに対して, 配列 b では先頭に `hashed` と表示されており, これは配列 b が LISP のハッシュ表から生成されている事が分ります.

この様に Maxima の配列には二つの型がありますが,make_array を用いれば, 配列データの型を指定して配列の生成が行えます.

ここで,make_array の構文は以下の様になります.

make_array(型, 次元$_1$, 次元$_2$, $\cdots$, 次元$_n$)

型には`'any`,`'flonum`,`'fixnum`,`'hashed` や`'functional` が使えます.array 命令との違いは, 生成された配列が函数配列オブジェクトになる点です.array よりも良い点は, 引数に配列の名前を持たないので, 一時的に利用し易い事です. これは,y:make_array($\cdots$) とすると,y はメモリの或る領域を占有するオブジェクトを指定しますが, ここで, y:false; とすると,y は配列を指定せず, 配列の方はゴミ収集で回収されるからです.

重要な事ですが, ここでの次元は array 命令のものと異なったものになります. 何故なら, それらには 0 から i-1, つまり,10 次元は 0 から 9 までの元がある事を意味するからです.

y:make_array('functional,'f,'hashed,1); と入力すると,make_array の第二の配列は函数で, 配列の元の計算の為に呼出すものになります. そして, 残りの引数は再帰的に make_array に渡されて, 配列函数オブジェクト向けのメモリを生成します.

7.2 配列に関連する変数

bashindices

allsym

デフォルト値:[true]

true の場合, 全ての添字付けられた対象は, それらの全ての共変と反変添字の対称性を持つと仮定されます.

false であれば, 添字に関してどの様な対称性も仮定されません. 但し, 微分の添字は常に対称性を持っています.

bashindices(⟨ 式 ⟩)

各和と積に単一の添字を与える事で ⟨ 式 ⟩ を変換します. 和や積で作用している時,changevar でより大きな精度が与えられます. 添字の形式は j⟨ 番号 ⟩ となります. ここで ⟨ 番号 ⟩ は gensumnum を参照して決められ, その gensumnum は利用者が変更可能です. 例えば, gensumnum:0 で再設定出来ます.

genindex

デフォルト値:[i]

計算で必要な時に和の変数を生成する為に使われるアルファベットの前置詞です.

gensumnum

デフォルト値:[0]

数値の修飾子で総和の次の変数を生成する為に用いられます. false に設定されていれば, その添字は数値では無い添字で genindex だけで構成されます.

use_fast_arrays

デフォルト値:false

true であれば, 唯二つの種類の配列のみが認識されます.

1. art-q 配列 (Common LISP で t) は整数で添字された次元を持つものですが, 項として任意の LISP や Maxima のオブジェクトが保持出来ます. この配列の構成には, `a:make_array` を用います.

2. ハッシュ表配列では,c の配列とは意味合いが異り, この場合, 配列の添字は整数値である必要はありません. 例えば,b が art-q 配列, リストや行列でもない状態で `b[x+1]:y^2` とすると, ハッシュ表配列として b が生成されます. ここで,b が配列やリストでも行列の内の一つで,x:1 の様に x に予め整数が割当てられてなければ,`x+1` は art-q 配列, リストや行列では無効な添字です. その為, この様な配列の割当はエラーになります. その添字 (LISP では鍵として知られています) には任意のオブジェクトが使えます. 尚, これは古い macsyma のハッシュ表による配列とは互換性がありませんが,cons の実行は保ちます.

尚, 記号値で配列を保管する利点は, 函数の局所変数に関する通常の取り決めが配列にも同様に作用する事が挙げられます. ハッシュ表型は MACSYMA の古い型の hashar よりも少なく cons を用いており効率的です. 変換されてコンパイルされたコードで十分な振舞いを得たければ,translate_fast_arrays を true に設定して下さい.

7.3 配列に関する函数

arrayapply

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.

arrayinfo

arrayinfo(⟨ 配列 ⟩)

引数として一つの ⟨ 配列 ⟩ を取り, その配列の情報をリスト形式で表示する函数です. 返却され情報は ⟨ 配列 ⟩ が array を用いて生成されたものであれば, 先頭に `declared`, 次に配列の次元とその大きさが表示されるが, データ型がハッシュ表であれば, 先頭が,`hashed`, 次に配列の次元, 最後に数値が設定されている個所の添字が表示されます. これは LISP の配列とハッシュ表の違いに関係するものです.

arrays

arrays

arrays は引数を必要としない函数で, arrays; と入力すると,Maxima で利用者が定義した全ての配列名のリストを返します. 尚, 具体的な情報は arrayinfo や arraylist で調べられます,

fillarray

fillarray(⟨ 配列 ⟩, ⟨ 配列かリスト ⟩)

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.

rearray

rearray(⟨ 配列 ⟩, ⟨ 次元$_1$⟩, ··· , ⟨ 次元$_n$⟩)

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.

remarray

remarray(⟨ 配列$_1$⟩, ⟨ 配列$_2$⟩, ···)

指定した配列の除去を用います.remarray を用いると, 配列とその配列に関連した函数を除去し, 占領されていた保存領域を解放します. 引数が all であれば全ての配列が除去されます.

ハッシュ表による配列で値を再定義したい場合, この函数を使わなければなりません.

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 を作用させた結果は ⟨ 函数 ⟩ を作用させた各成分を主演算子で繋げたものとなります.

7.4　map 函数族

map 函数は LISP ではお馴染の函数で, 基本的に函数をリストに作用させるものです. これは Mathematica や Maple でも採用されており, 非常に便利な函数でもあります.Maxima では, 式は内部的にリスト構造を持っていますが, 表にはそれが現われていない為,map 函数の動作が判り難い側面もあります.

ここでは, 内部形式と絡めて,map 函数のお仲間について解説します.

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

ここで,Maxima の式の内部表現は, 先頭に演算子や函数が置かれる前置式表現となっています. 例えば,$x + 2z^2 + \sin z + a$ は
`(+ x (* 2 (^ z 2)) (sin (+ z a)))` の様になります. これに階層を入れると次の様になります.

```
0| (+                                          )
1|    x  (*               )  (sin            )
2|          2  (^     )           (+     )
3|               z 2                z a
```

最上部に主演算子とも言える+が置かれ, 第一層にはアトムの x,2, 式 z$\hat{}$2 と式 sin(z+a), 第二層には, 式 z$\hat{}$2 と式 z+a があり, 最下層にはアトムの z$\hat{}$2 のアトム z,2 と z+a のアトム z,a があります.

この様に,Maxima の全ての式には内部表現による階層があり,map 函数族はこの階層構造を反映した処理を行います.

先ず,map 函数と maplist 函数はこの第一層に含まれる部分式に作用しますが, scanmap 函数は各階層全ての式に対して作用します.

最初に map 函数の例を示します.

```
(%i34) map(sin,x*y);
(%o34)                          sin(x) sin(y)
(%i35) map(sin,x*y+y);
(%o35)                        sin(x y) + sin(y)
(%i36) map(sin,factor(x*y+y));
(%o36)                        sin(x + 1) sin(y)
(%i37) map(lambda([x,y],x*y),x+y,w+z);
(%o37)                           y z + w x
```

最初の x*y の場合, 主演算子は*で, 第一層には x と y がある為,sin は第一層の x と y に作用し, 演算子の置換えを行わない為, sin(x)+sin(y) が得られます.x*y+y の場合, 第一層には x*y と w がある為, 今度は sin(x*y)+sin(y) となります. ところが, 同値な式でも, 内部表現が異なると異った結果になります. 次の例では factor(x*y+y) の結果に map 函数で sin を作用させていますが,factor(x*y+y)

は (x+1)*y となるので, 第一層には (x+1) と y があり, 主演算子が*なので,sin(x+1)*sin(y) となります.

次の maplist 函数は基本的に map 函数と同様ですが, 主演算子をリストに置換えてしまいます.

```
(%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.

map 函数と maplist 函数の動作には大域変数の maperror が影響を与えます. maperror はデフォルト値が true で,map 函数や maplist 函数の動作に影響を与えます.

先ず,map 函数と maplist 函数の引数は ⟨ 函数 ⟩, ⟨ 式$_1$⟩, $\cdots$, ⟨ 式$_n$⟩ となります. ここで,⟨ 函数 ⟩ は n 変数の函数です. 基本的に各 ⟨ 式$_i$⟩ の主演算子は同じもので, 成分の個数も同じ個数ですが,maperror が false の場合は, それ以外の引数でも適用可能になり, 以下の動作となります.

1. 全ての ⟨ 式$_i$⟩ が同じ長さでなければ, 最短の ⟨ 式$_j$⟩ を終えた時点で停止します.
2. ⟨ 式$_i$⟩ の主演算子が全て同じものでなければ,[⟨ 式$_1$⟩, $\cdots$, ⟨ 式$_n$⟩] に ⟨ 函数 ⟩ を作用させ,apply と同じ動作になります.

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);
```

7.4.1 map 函数族に関連する大域変数

maperror

デフォルト値:[true]

この変数は map 函数や maplist 函数の動作に影響を与えます.

7.4.2 map 函数概要

map

map(⟨ 函数 ⟩,⟨ 式$_1$⟩,···,⟨ 式$_n$⟩)

n 個の式の列 ⟨ 式$_1$⟩,···,⟨ 式$_n$⟩ から n 個の成分を取出し,n 個の引数を取る ⟨ 函数 ⟩ を作用させた結果を返します.

mapatom

mapatom (⟨ 式 ⟩)

⟨ 式 ⟩ が map 函数によってアトムとして扱われる時,true となります. mapatom はアトム, 整数, 有理数と添字された変数です.

maplist

maplist(⟨ 函数 ⟩,⟨ 式$_1$⟩,⟨ 式$_2$⟩,···)

⟨ 式$_i$⟩ の各成分に ⟨ 函数 ⟩ を作用させたリストを生成します.⟨ 函数 ⟩ は函数の名前や lambda 式となります.

maplist は map(⟨ 函数 ⟩,⟨ 式$_1$⟩,⟨ 式$_2$⟩,···) と異なるのは,map の場合,⟨ 式$_i$⟩ の主演算子で式を纏めますが,maplist が主演算子の代りに Maxima のリストが来る事です.

```
(%i27) maplist(sin,x+y);
(%o27)                          [sin(y), sin(x)]
(%i28) map(sin,x+y);

(%o28)                           sin(y) + sin(x)
(%i29) maplist(lambda([x,y],x*y),x+y,w+z);
(%o29)                             [y z, w x]
(%i30) map(lambda([x,y],x*y),x+y,w+z);
(%o302)                              y z + w x
```

scanmap

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,$(a^2+2*a+1)*y+x^2$ を factor で因子分解しても, そのまま元の式が返却されるだけですが,scanmap を使って factor を式に作用させると,$a^2+2*a+1$ の部分が因子分解された結果が返されます.

```
(%i3) exp:(a^2+2*a+1)*y + x^2$
(%i4) factor(exp);
                              2                 2
(%o4)                        a  y + 2 a y + y + x
(%i5) scanmap(factor,exp);
                                      2      2
(%o5)                           (a + 1)  y + x
```

scanmap 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 は factor を上の階層から各部分式に対して作用させます. この式の場合, 最初に x$\hat{2}$ と (a$\hat{2}$+2*a+1)*y に factor を作用させ, その次に x$\hat{2}$ の x と 2,(a$\hat{2}$+2*a+1) と y 等々と下の階層の成分に作用します. その結果,a$\hat{2}$+2*a+1(=(+ 1 (* 2 a) (^ a 2))) の展開以外はそのままとなる為, 上記の結果を得ています. この作用の様子は factor の代りに'f を与えると判り易くなります.

```
(%i18) scanmap('f,exp);
                  f(2)                                          f(2)
(%o18) (f(f(f(f(a)    ) + f(f(2) f(a)) + f(1)) f(y)) + f(f(x)    ))
```

この性質がある為に式を全て展開してしまうと, 各項に factor を作用させる為に, 入力値がそのまま返却されてしまいます.

```
(%i16) expand(exp);
                              2                 2
(%o16)                       a  y + 2 a y + y + x
(%i17) scanmap(factor,expand(exp));
                              2                 2
(%o18)                       a  y + 2 a y + y + x
```

scanmap((⟨ 函数 ⟩,⟨ 式 ⟩,bottomup) は scanmap((⟨ 函数 ⟩,⟨ 式 ⟩) とは逆に, 内部表現の最下層から ⟨ 函数 ⟩ を作用させます.

第8章　行列と線形代数について

8.1　行列とベクトル

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なく,matrix命令を用いて行列を宣言して行く方式になります.

この matrix 命令は matrxi(行$_1$, 行$_2$, $\cdots$) の様に行列の行ベクトルをリストの形式でコンマで区切った列を引数とします. 行列の和と差は通常の演算子+と演算子-を各々用います. 尚, 積演算子*と商演算子/は行列の成分同士の積と商となり, 所謂, 行列や逆行列との積にはなりません.

```
(%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 ]
```

その為, 通常の積演算子*の事をここでは可換積と呼びます. 通常の行列の積演算子は dot 積と呼び, 記号は dot(.) で表現します. 例えば, 行列 A と B に対しては, `A . B` で行列の積を定めます. この行列の積に関しては可換性はありません. この dot 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.

又, 行列の冪乗は記号^^で記述します. ここで, 行列 A の逆行列が存在する場合,`A^^(-1)` が行列 A の逆行列になります.

ベクトルは `share/vector/`にベクトル解析パッケージが含まれています. ベクトル解析パッケージの読込は load(vect) で行います. ベクトル解析パッケージを使えば内積と外積, 勾配, 発散, 回転や laplace 演算子を含む記号式の纏めと簡易化が行える様になります. これらの演算子の和や積に対する分配は, 利用者指定で制御出来ます. 尚, スカラー値や場のベクトルの微分も可能です.

このパッケージは次の命令を含んでいます.

vectorsimp,scalefactors,express,potential と vectorpotential.

ここで注意する事に,vect パッケージの読込みを行うと dot 演算子が行列の非可換積では無く, 可換演算子であると宣言してしまう事です.

8.2 関連する大域変数

detout

デフォルト値:[false]

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.

doallmxops

デフォルト値:[true]

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

デフォルト値:[true]

true の場合,%eˆ matrix([1,2],[3,4]) は matrix([%e,%eˆ 2], [%eˆ 3,%eˆ 4]) となります. 一般的に, この変換は<基底>ˆ<冪>の形式の変換に影響します. 尚,⟨ 基底 ⟩はスカラか定数の式であり,⟨ 冪 ⟩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

デフォルト値:[true]

true であれば, 行列と行列間の演算子や行列とリストの間の演算子が実行されます. この大域変数が false なら, これらの演算は実行されません. 尚, この大域変数はスカラーと行列との間の演算には影響を与えません.

domxnctimes

デフォルト値:[false]

false であれば行列の非可換積が実行されます.

dontfactor

デフォルト値:[]

dontfactor に因子分解を行わない変数リストを設定します. この変数リストを設定すると,CRE 形式を構成する上で仮定された変数順序で, ここで指定した変数よりも小さなものに対しても, 因子分解は実行されません.

doscmxops

デフォルト値:[false]

true であればスカラと行列間の演算子が実行されます.

doscmxplus
　デフォルト値:[false]
　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.

lmxchar
　デフォルト値: [[]
　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
　デフォルト値:[]]
　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_add
　デフォルト値:[+]
　matrix_element_add は行列同士の和を計算する演算子を設定します. 函数名や lambda 式であっても構いません.

matrix_element_mult
　デフォルト値:[*]
　matrix_element_mult
は行列の成分同士の積を計算する演算子を設定します. 函数名や lambda 式であっても構いません.

matrix_element_transpose
　デフォルト値:[false]
　別の便利な設定は transpose と nonscalars; であり, 函数名や lambda 式であっても良い. こうする事で様々な代数的構造が扱える.

ratmx
　デフォルト値:[false]
　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 形式で行われて逆行列の結果も CRE 形式となります. 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) の原因となるかもしれせん.

"[" と "]"　[と] は Maxima でリストの区切り記号として使う文字です.

scalarmatrixp
　デフォルト値:[true]
　true であれば, 二つの行列の dot 積の計算で得られた 1x1 行列はスカラーに変換されます. もし,all に設定されていれば, この変換は 1x1 行列は何時でもスカラーに実行されます.

但し,false に設定されていれば, この変換は実行されません.

sparse

デフォルト値:[false]

この変数が 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.

vect_cross

デフォルト値:[false]

rue であれば `diff(x~y,y)` が使えます. ここで,~は share;vect で定義されている (但し,vect_cross が true に設定されていれば).

8.3　関連する函数

addcol と addrow

addcol (⟨ 行列 ⟩, ⟨ リスト$_1$⟩, ⟨ リスト$_2$⟩, $\cdots$, ⟨ リスト$_n$⟩)
addrow (⟨ 行列 ⟩, ⟨ リスト$_1$⟩, ⟨ リスト$_2$⟩, $\cdots$, ⟨ リスト$_n$⟩)
addcol は列として,addrow 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 m の大きさと矛盾しないものでなければなりません.

adjoint

adjoint (⟨ 行列 ⟩)
⟨ 行列 ⟩ の余因子行列を計算します.

augcoefmatrix

augcoefmatrix([⟨ 方程式$_1$⟩, $\cdots$],[⟨ 変数$_1$⟩, $\cdots$])
与えられた方程式と指定した変数のリストから, 係数行列を生成します. 行列は,⟨ 方程式$_1$⟩, $\cdots$ から構成される線形方程式系に含まれる ⟨ 変数$_1$⟩, $\cdots$ の係数から構築されるものです.
この係数行列には各方程式の定数項が列として付け加えられています.

charpoly

charpoly (⟨ 行列 ⟩, ⟨ 変数 ⟩)
⟨ 行列 ⟩ の特性多項式 $det(<$ 変数 $>$ I$-$ $<$ 行列 $>)$ を計算します. Maxima の $determinant(<$ 行列 $> - diagmatrix(length(<$ 行列 $>), \langle$ 変数 $\rangle))$ と同じ結果を返します.

coefmatrix

coefmatrix ([⟨ 方程式$_1$⟩, $\cdots$], [⟨ 変数$_1$⟩, $\cdots$]) 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.

col

col (⟨ 行列 ⟩, ⟨i⟩)
⟨ 行列 ⟩ の ⟨i⟩ 番目の列を返します.

columnvector

columnvector (⟨ リスト ⟩))
eigen パッケージの函数なので, 利用する為には予め load(eigen) を実行します.
columnvector はリストを引数とし, 引数リストを成分とする列ベクトルを返します.

conjugate

conjugate (⟨ リスト ⟩)
eigen パッケージの函数です.conjugate は引数の複素共役を返します.

copymatrix

copymatrix (⟨ 行列 ⟩)

⟨ 行列 ⟩ の複製を行います. この命令は m を成分毎に再生成する時だけに使われます. 行列の複製では setmelmx を使うと便利です.

determinant

determinant(⟨ 行列 ⟩)

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 と設定した場合に使えます.

diagmatrix

diagmatrix(⟨n⟩, ⟨ 行列 ⟩)

n 行 n 列の対角行列を返します. ここで対角成分は全て ⟨ 行列 ⟩ のものです. 尚,diagmatrix(n,1) は ident(n) と同じ n 次元の単位行列を生成します.

echelon

echelon(⟨ 行列 ⟩)

⟨ 行列 ⟩ の echelon 形式を生成します. これは初等的な行操作で各々の行の最初の非零元を 1, その元を含む列に対してはその元を含む行よりも下の成分を全て零となる様に変形するものです (上三角行列).

```
                       [2  1 - a  -5 b ]
(d2)                   [               ]
                       [a    b      c  ]

(c3) echelon(d2);
                [      a - 1        5 b      ]
                [1   - -----       - ---      ]
                [        2            2       ]
(d3)            [                             ]
                [                2 c + 5 a b  ]
                [0     1         ------------ ]
                [                       2     ]
                [                2 b + a  - a ]
```

eigenvalues

eigenvalues (⟨ 行列 ⟩)

share ディレクトリ上のパッケージです.eigenvalues と eigenvectors と関連する行列の計算を行う函数が含まれています.

eigenvalues は引数に一つの行列を取り, リストを返します. このリストの最初の副リストは固有値リストで, その他の副リストは固有値の順番に対応した重複度のリストとなります. 尚,Maxima の函数 solve が, 行列の特性多項式の根の計算で利用されていいます.solve は多項式の根を見付け損なう事がありますが, conjugate,innerproduct,univector,columnvector と gramschmidt 以外は使えず, これらだけが固有値が判らなくても使えます. 幾つかの状況で,solve は非常にいい加減な固有値を生成する事があります.

処理を続ける前に答を簡易化しても構いません. この為の予測があり, それらは以下で説明されています.(これは solve が実数と予測されるが, それ程明確でない実数式を返す場合に生じます.)

eigenvalue 函数は Maxima から直接使える. この他の函数を利用する為には eigen パッケージの読み込みが必要となり, 事前に eigenvalues を実行するか, load(”eigen”) を実行しましょう.

eigenvectors

eigenvectors (⟨ 行列 ⟩)

引数として行列を取り, リストを返します. 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. この函数は Maxima から直接使えますが, 以下の大域変数の設定を有効にしたければ,eigen パッケージの読込が必要となります.

- nondiagonalizable[false] は,eigenvectors 命令の実行後に, 行列が対角化不可能か否かによって true か false のどちらかが設定されます.
- hermitianmatrix が true ならば,Gram-Schmidt のアルゴリズムを用いて行列を対角化し,Hermite 行列の固有ベクトルを削減します.
- knowneigvals が,true に設定されていれば, 行列の固有値は既知であって, 大域変数の listeigvals に保存されていると eigen パッケージが仮定します. その為,listeigvals に eigenvalues 函数の出力と同じリストが設定されている必要があります. 尚,algsys 函数が固有ベクトルの計算で使われています. 固有値が不確かな場合,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.

ematrix

ematrix ($\langle m\rangle, \langle n\rangle, \langle x\rangle, \langle i\rangle, \langle j\rangle$)

$\langle m\rangle$ 行 $\langle n\rangle$ 列の行列で $\langle i\rangle$ 行 $\langle j\rangle$ 列成分のみが $\langle x\rangle$, 他が全て零となる行列を生成します.

entermatrix

entermatrix ($\langle m\rangle, \langle n\rangle$)

Maxima の要求に従って $\langle m\rangle \times \langle n\rangle$ 個の成分を入力して行列を生成します.

```
(c1) entermatrix(3,3);
is the matrix  1. diagonal  2. symmetric  3. antisymmetric
 4. general

answer 1, 2, 3 or 4
1;
row 1 column 1:  a;
row 2 column 2:  b;
row 3 column 3:  c;
matrix entered.
                                [ a  0  0 ]
                                [         ]
(d1)                            [ 0  b  0 ]
                                [         ]
                                [ 0  0  c ]
```

genmatrix

genmatrix(⟨ 配列 ⟩, ⟨i_2⟩, ⟨j_2⟩, ⟨i_1⟩, ⟨j_1⟩)

配列から行列を生成します. ここで配列 array(⟨i_1⟩, ⟨j_1⟩) は最初 (左側上) の元 array(⟨i_2⟩, ⟨j_2⟩) が残り (左下) の元となります. ⟨j_1⟩ = ⟨i_1⟩ であれば,⟨j_1⟩ は削除されます. ⟨j_1⟩ = ⟨i_1⟩ = 1 ならば, ⟨i_1⟩ と ⟨j_1⟩ の両方が省略される事もあります.

配列の元が不足した場合には, 記号的な元が使われます.

```
(c1) h[i,j]:=1/(i+j-1)$
(c2) genmatrix(h,3,3);
                           [   1  1]
                           [1  -  -]
                           [   2  3]
                           [       ]
                           [1  1  1]
(d2)                       [-  -  -]
                           [2  3  4]
                           [       ]
                           [1  1  1]
                           [-  -  -]
                           [3  4  5]
```

gramschmidt

gramschmidt([⟨ リスト$_1$⟩, $\cdots$, ⟨ リスト$_n$⟩])

eigen パッケージに含まれる函数です. 予め load(eigen) を実行して利用します. gramschmidt は引数にリストの列で構成されるリストを取ります. ここで ⟨ リスト$_i$⟩ は全て長さが等しくなければなりませんが, 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.

hach

hach (a,b,m,n,l)

Hacijan の線型プログラミングアルゴリズムの実装. 予め load(kach) で読込む必要があります.

ident

ident (⟨n⟩)

⟨n⟩ 行 ⟨n⟩ 列の単位行列を生成します.

innerproduct

innerproduct (⟨x⟩, ⟨y⟩)

eigen パッケージに含まれる函数です. 使う為には予め load(eigen) を実行します. innerproduct は同じ長さの二つのリスト ⟨x⟩ と ⟨y⟩ を引数として取り, (⟨x⟩ の複素共役) . ⟨y⟩ で定義されています. ここで,dot 演算子は通常のベクトルの内積演算子と同じものです.

invert

invert (⟨ 行列 ⟩)

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.

detout フラグが true であれば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.

行列式で割られた方が望ましい場合,xthru(%) を併用したり, expand(adjoint(⟨ 行列 ⟩))/expand(determinant(⟨ 行列 ⟩)) で計算すると良いでしょう.

matrix

matrix (⟨ 行$_1$⟩, $\cdots$, ⟨ 行$_n$⟩)

指定した行を持つ行列を定義します.

matrixmap

matrixmap (⟨ 函数 ⟩, ⟨ 行列 ⟩)

行列 ⟨m⟩ の各成分に ⟨ 函数 ⟩ を作用させます.

matrixp

matrixp (⟨ 式 ⟩)

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

mattrace

mattrace (⟨ 行列 ⟩) ⟨ 行列 ⟩ が正方行列の場合, 対角和 (行列の主対角成分の総和) を計算します. これは ncharpoly で利用されます.ncharpoly は Maxima の charpoly の代りに用いられています.load(”nchrpl”) を実行する必要があります.

minor

minor (⟨ 行列 ⟩, ⟨i⟩, ⟨j⟩)

⟨ 行列 ⟩ の ⟨i⟩, ⟨j⟩ 成分の小行列, つまり,⟨ 行列 ⟩ から ⟨i⟩ 行と ⟨j⟩ 列を抜いた小行列を計算します.

ncexpt

ncexpt(⟨ 行列$_1$⟩, ⟨ 行列$_2$⟩)

⟨ 行列$_1$⟩ ^^ ⟨ 行列$_2$⟩ を (非可換) 指数式で表示する際に, 大き過ぎれば
ncexpt(⟨ 行列$_1$⟩, ⟨ 行列$_2$⟩) が用いられます.

ncharpoly

ncharpoly (⟨ 行列 ⟩, ⟨ 変数 ⟩)

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

ncharpoly では与えられた行列の冪乗の対角和を計算しますが, 対角和は特性多項式の根の冪乗の総和に等しいものです. これらの諸量から根の対称式の計算が可能ですが, それらは特性多項式の係数です.charpoly は var*ident[n]-a の行列式を計算している. そんな訳で ncharpoly は優れている. 例えば, 整数成分の非常に大きな行列の場合は算術的に多項式の計算を避ける為です.

予め load(”nchrpl”): で読込む必要があります.

newdet

newdet (⟨m⟩, ⟨n⟩)

⟨m⟩ 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.⟨m⟩ は行列か配列の何れかで,⟨n⟩ は行列の大きさになります, これは ⟨m⟩ が行列であればオプションになります.

nonscalarp

nonscalarp (⟨ 式 ⟩)

⟨ 式 ⟩ が非スカラ, つまり, 原子を含み, 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.

permanent

permanent (⟨ 行列 ⟩, ⟨n⟩)

⟨ 行列 ⟩ の permanent を計算する.permanent は行列式に似ていますが, 符号の変化のないものです.

rank

rank (⟨ 行列 ⟩)

⟨ 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.

row

row (⟨ 行列 ⟩, ⟨i⟩)

⟨ 行列 ⟩ の ⟨i⟩ 番目の行を出力します.

scalarp

scalarp (⟨ 式 ⟩)

true となるのは,⟨ 式 ⟩ が数, 定数やスカラー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.

setelmx

setelmx (⟨x⟩, ⟨i⟩, ⟨j⟩, ⟨ 行列 ⟩

⟨ 行列 ⟩ の ⟨i⟩ 行 ⟨j⟩ 列成分を ⟨x⟩ で置換えます. 置換えられた行列が返却されます. 尚, 直接行列の成分を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.

similaritytransform

similaritytransform (⟨ 行列 ⟩)

eigen パッケージに含まれる函数です. 予め, load(eigen) で読込む必要があります.

⟨ 行列 ⟩ を引数とし,uniteigenvectors 命令の出力結果のリストを返します.

大域変数 nondiagonalizable が false であれば, 二つの行列 leftmatrix と rightmatrix が生成されます. この leftmatrix と rightmatrix は, leftmatrix . ⟨ 行列 ⟩ . rightmatrix が対角行列となり,⟨ 行列 ⟩ の固有値が対角成分に現れるものです.

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

大域変数 hermitianmatrix が true であれば,leftmatrix は rightmatrix の複素共役の転置となります. その他で,leftmatrix は rightmatrix の逆行列になります. 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.

submatrix

submatrix (⟨ 行$_1$⟩, ⋯, ⟨ 行$_m$⟩, ⟨ 列$_1$⟩, ⋯, ⟨ 列$_n$⟩)

⟨ 行$_i$⟩ 行と ⟨ 列$_j$⟩ 列が削除された新しい行列を生成すします.

transpose

transpose (⟨ 行列 ⟩) ⟨ 行列 ⟩ の転置行列を生成します.

triangularize

triangularize (⟨ 行列 ⟩)

⟨ 行列 ⟩ の上三角行列形式を生成します. 但し, 行列が正方行列である必要はありません.

uniteigenvectors

uniteigenvectors (⟨ 行列 ⟩)

eigen パッケージの函数. 利用する為には予め load(eigen) を実行します. uniteigenvalues は与えられた ⟨ 行列 ⟩ の固有値と固有ベクトルで構成されたリストを返します. 出力リストの第一成分のリストには eigenvalues 函数の出力があり, 第二成分の副リストには正規化した固有ベクトルが, 第一成分のリストの固有値に対応する順番で並んでいます.

eigenvectors 函数の詳細で述べた大域変数はここでも同様の影響を与えます.

unitvector

unitvector (⟨ リスト ⟩)

eigen パッケージに含まれる函数. 予め load(eigen) を実行して読込む必要があります.

unitvector は ⟨ リスト ⟩ を正規化したリスト, 即ち, その大きさを 1 にしたリストを返します.

vectorsimp

vectorsimp (⟨ ベクトル ⟩)

この函数は和の簡易化で様々な大域変数の設定と共に用います. ここで式は以下の大域変数の設定に関連するものです.

`expandall, expanddot, expanddotplus, expandcross, expandcrossplus, expandcrosscross, expandgrad, expandgradplus, expandgradprod, expanddiv, expanddivplus, expanddivprod, expandcurl, expandcurlplus, expandcurlcurl, expandlaplacian, expandlaplacianplus, expandlaplacianprod.`

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.

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 の付く大域変数はあらゆる種類の積演算に対する被演算子への分配性に関連するものです.

expandcrosscross は p (q r) を (p,r)*q-(p.q)*r で置換するかどうかを決めます.

expandcurlcurl は curl curl p を grad div p + div grad p で置換するかどうかを決定します.

expandcross が true の場合,expandcrossplus と expandcrosscross が true と同じ効果があります.

二つの大域変数 expandplus と expandprod は似た名前の大域変数に true に設定した場合と同効果です. これらが,true であれば, 他の大域変数, expandlaplaciantodivgrad は laplace 演算子を div grad で置換えます.

尚, 簡便の為にこれら全ての大域変数は evflag として宣言されています.

zeromatrix

zeromatrix ($\langle m \rangle, \langle n \rangle$)

引数として整数値の $\langle m \rangle$,$\langle n \rangle$ を取って $\langle m \rangle$ 行 $\langle n \rangle$ 列の零行列を返します.

第9章　多項式について

9.1　多項式の内部表現

Maxima の多項式は通常の C や FORTRAN で記述する x^2+3*x*z+4 の様な書式で入力され, 表示される結果も同様です. 但し, 表示の方は入力式そのままではありません. 多少の簡易化を行ない, 係数の内部的順序に沿って項の並替えを行います.

```
(%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
```

この例では多項式 x+y+z と (1+2)*x+3*y+(2+1-2)*z-z の処理を示したものです. 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) が返却されています. (MPLUS\ SIMP) でこの式が Maxima の和である事を示し, 後には項が並んでいます. この項の並びは,x+y+z と入力しても,z+x+y と入力しても同じです. これは Maxima には変数に対して順序を入れており, その順序に沿って与えられた多項式の各項を並替えているからです.

多項式 (1+2)*x+3*y+(2+1-2)*z-z を入力すると,Maxima は自動的に簡易化を行なっている事が判ります. 但し, この簡易化は項の係数が数値の場合に限定されます.

以下,x+y,x-y,x*y,x/y,x^y についても:lisp で内部表現を調べてみましょう.

```
(%i33) t0:x+y;
(%o34)                                y + x
```

```
(%i34) :lisp $t0;
((MPLUS SIMP) $X $Y)
(%i35) t1:x-y;
(%o35)                                x - y
(%i35) :lisp $t1;
((MPLUS SIMP) $X ((MTIMES SIMP) -1 $Y))
(%i36) t2:x*y;
(%o36)                                 x y
(%i37) :lisp $t2;
((MTIMES SIMP) $X $Y)
(%i38) t2:x/y;
                                        x
(%o38)                                  -
                                        y
(%i39) :lisp $t3;
((MTIMES SIMP) $X ((MEXPT SIMP) $Y -1))
(%i40) t4:x^y;
                                        y
(%o40)                                 x
(%i41) :lisp $t4;
((MEXPT SIMP) $X $Y)
```

この様に Maxima では x-y は x+(-y),x/y は x^ (-y) で置換えている事が判ります. Maxima ではこの様に演算子を被演算子の前に置いた前置表現を一般形式と呼んでいます. これに対し, 更に内部表現を簡潔にした正準有理式表現 (Cannonical Rational Expressions, 略して CRE 表現) もあります.

CRE 表現は factor や ratsimp 等々の函数で内部的に利用されるもので, 利用者はこの表現をあまり意識する必要はありません. この CRE 表現は本質的に展開された多項式や有理式函数に適したリストによる表現の一つです.

尚, 多項式のリストによる表現には連想リスト (alist) による表現もありますが, CRE 表現はより簡潔な (小括弧の少い) 表現となります.

先ず, 多項式 $3x^2-1$ で説明しましょう. この場合,$3x^2+(-1)x^0$ と同値です. これを λ 式風に考えれば, $\lambda\ x\dot{3}x^2+(-1)x^0$ となりますね. ここで, 係数と次数の対をから構成されるリストは ((2 3) (0 -1)) となります. これだけでは変数が判らないので, 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. ここで, 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. この様に, 単変数の多項式に対する CRE 表現は $\boxed{(\langle\ 変数\ \rangle\ \langle\ 次数_1\rangle\ 係数_1\ \langle\ 次数_2\rangle\ 係数_2\ \cdots)}$ で与えられるもので, この表現と単変数多項式は一対一に対応します.

多変数多項式の場合も同様に表現出来ます. 但し,1 変数の場合の様な平坦なリストで CRE 表現は表現されず,CRE 表現のリストによる複合リストとなります. 多変数の場合, 重要な事は変数の間に順序を入れる事です. これは辞書で見られる様な順番です.Maxima の場合は逆辞書式と呼ばれ

る順序がデフォルトで入っています. この順序ではアルファベットの z が一番大きく,a が一番小さいというアルファベットの逆の順序で順番が付けられています. ここで, 変数が二文字以上の場合, 最初に先頭の文字を比較します. そこで, 先頭の文字が等しければ次の文字で比較します. 途中で大小関係が付くと, その順序で変数に順番が入ります. 例えば,xxz と xxy の場合は頭の二つは x ですが, 最後が z と y で比較する為,z>y であれば,xxz>xxy となります.

さて, 多変数多項式の CRE 表現に戻りますが, この場合は, 再帰的な考えで処理します. 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 の多項式とすると $(2y+1)x-3$ 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. 今度は,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 表現に置換えて, (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. その為, 二つの CRE 表現が与えられた時に, それらの変数順序が異なれば, 処理は非常に厄介な事になるのが判るかと思います. その為,CRE 表現を用いる函数に関しては, 後の事も考えて主変数の設定を行う必要があります.

又,CRE 表現の変数には, 通常の算術演算子 (+,-,*,/) や整数冪 (^) を持たないものを与えます. その為,y^ 2 の様な式は変数に使えませんが,log(x) や cos(x+1) の様な函数は使えます.

ここで, 変数順序は ratvars 函数で指定出来ます.ratvars で指定しなかった場合,Maxima は逆アルファベット順の順序を入れています.

有理式は多項式の分数ですが, 有理式の CRE 表現の表現は, 多項式の分母と分子に共通因子が無く, 分母の筆頭項 (leading term) の係数を正にしたものとなります.

以下に実例を示しましょう.

```
(%i1) r1: rat((y-1)/((y-x)*z^2+1));
                                    y - 1
(%o1)/R/                          --------------
                                          2
                                  (y - x) z  + 1
(%i2) r2: rat((y-1)/((x-y)*z^2+1));
                                     y - 1
(%o2)/R/                         - --------------
                                           2
                                   (y - x) z  - 1
(%i3) r3:rat((y-1)/(-(y-z)*x^2+1));
                                     y - 1
(%o3)/R/                          ---------------
                                   2      2
                                  x  z - x  y + 1
(%i4) :lisp $r3;
((MRAT SIMP ($X $Y $Z) (X13180 Y13181 Z13182)) (Y13181 1 1 0 -1)
Z13182 1 (X13180 2 1) 0 (Y13181 1 (X13180 2 -1) 0 1))

(%i4) t3:(y-1)/(-(y-z)*x^2+1);
                                    y - 1
(%o4)                             --------------
                                   2
                                  x  (z - y) + 1
(%i5) :lisp $t3;
((MTIMES SIMP)((MPLUS SIMP) -1 $Y)
 ((MEXPT SIMP)
  ((MPLUS SIMP) 1
   ((MTIMES SIMP)((MEXPT SIMP) $X 2)((MPLUS SIMP)
    ((MTIMES SIMP) -1 $Y) $Z)))
  -1))
(%i5)
```

この例では順序が逆アルファベット順の為, 変数の順序は $z > y > x$ となります. その為,z が主変数となって式は 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,$y > x$ となるので $-(y-x)z^2$ 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 yz^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, その後で変数が X,Y,Z であり,X,Y,Z の順序が, 整数 13180,13181 と 1382 で与えられている事を示しています. この式では Z の値が最大の為,Z が主変数となります. 尚,showratvars はこの変数リストを返却する函数です. 次に,(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, 重み 13181 が付いた変数 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, 式は変数の順序に従って読む必要があります. この例では $Z > Y > X$ の順なので,Z があれば, Z の多項式として表現し,Z が無くて Y があれば Y の多項式,Z と Y が無ければ X の多項式として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, この様に, 非常に長いものとなっている事が判ります.

尚,CRE 表現で分母が整数の場合 (CRE 表現では, 浮動小数点は有理数で近似されます),CRE 表現の内部表現は少し変化します.

```
(%i4) r1:rat((x-1)/5);
                                      x - 1
(%o4)/R/                              -----
                                        5
(%i5) :lisp $r1;
((MRAT SIMP ($X) (X13157)) (X13157 1 1 0 -1) . 5)
```

この例れ示す様に, 分子は x-1 ですが, 分子と分母の間に. 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. を入れています. これは CRE 表現の生成で cons が用いられている事を示しています.

拡張 CRE 表現は taylor 級数を表現する為に用いられています. 有理式の表記は正の整数では無く, 正か負の有理数となる様に拡張されており, 係数はそれ自身, 多項式と云うよりは, 上で記述した様に有理式となっています. これらは内部的に再帰的な多項式形式によって表現され, 多項式形式は CRE 表現に類似していますが, より一般化したものです. 尚, 切り捨てられる次数の様な情報も追加されています.

式を表示する際に, 拡張 CRE 表現の場合は記号/T/が式の行ラベルに続きます.

```
(%i1) t1:taylor(exp(x),x,0,5);
                                2    3    4    5
                               x    x    x    x
(%o1)/T/              1 + x + -- + -- + -- + --- + . . .
                               2    6    24   120
(%i2) :lisp $t1;
((MRAT SIMP (((MEXPT SIMP) $%E $X) $X) (%e^x13162 X13163)
  (($X ((5 . 1)) 0 NIL X13163 . 2)) TRUNC)
 PS (X13163 . 2) ((5 . 1)) ((0 . 1) 1 . 1)
                 ((1 . 1) 1 . 1) ((2 . 1) 1 . 2)
 ((3 . 1) 1 . 6) ((4 . 1) 1 . 24) ((5 . 1) 1 . 120))
```

雰囲気は CRE 表現に近いものですが, 係数と冪のリストの書式が cons で結合されたリストであり, 他に, リストの第一成分に Taylor 展開の様々な情報が追加されている事が判ります.

9.2 多項式に関する大域変数

algebraic

デフォルト値:[false]

代数的整数の簡易化を行う場合,true にしなければなりません.

尚, 代数的整数は最高次数の係数が 1 となる整数係数多項式の解となる数の事です. これに対し, 代数的数もあります. こちらは有理数係数多項式の解になる数の事です. 例えば, 純虚数 i や sqrt(2) の様な数は代数的整数になります. 又, 代数的整数と有理数は代数的数になります.

facexpand

デフォルト値:[true]

factor 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.

factorflag

デフォルト値:[false]

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.

berlefact

デフォルト値:[true]

false であれば,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.

intfaclim

デフォルト値:[1000]

大きな整数の因子分解を行う時に試す最大の約数.false を指定した場合 (利用者が factor を明示的に呼び出す場合) や, 整数が fixnum の場合 (つまり, 一つの機械語長に適合) に, 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.

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.

keepfloat

デフォルト値:[false]

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

modulus

デフォルト値:[false]

正素数 p が設定されていれば, 全ての有理函数での演算は p の剰余で計算されます (即ち, 多項式環の係数体は z_p となります). この場合,p/2 よりも大きな全ての整数の事は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=-2(mod

5),4=-1(mod 5) となる (実際,3-(-2)=4-(-1)=5(mod 5) となるので, 絶対値では {0,1,2} だけを考えれば良い事になります. その為, その分計算を簡単に行う事が可能になります.

modulus を再設定した時点で式が既に CRE 表現となっていると, 正しい結果を得る為には, 式に再び rat を作用させる必要があります. 即ち, exp:rat(ratdisprep(exp)) を実行しなければならない事に注意して下さい.

尚,modulus に素数でない正整数を設定しようとすれば, 設定は実行されるものの警告が出力されます. 尚, 素数でない正整数を設定する事はあまり意味がある事ではありません. 例えば,modulus に 4 を設定すると $(2x+1)(2x+1)$ は 1 となり,x と x^2 の項が消えてしまいます.

newfac

デフォルト値:[false]

true であれば,factor は新しい因子分解ルーチンを用います.

ratalgdenom

デフォルト値:[true]

true であれば, 分母の有理化を許容します. これを実行する為には, CRE 表現を利用しなければなりません.

ratdenomdivide

デフォルト値:[true]

false であれば,ratexpand 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.

ratepsilon

デフォルト値:[2.0e-8]

有理数に変換する際に用いられる許容範囲.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, keepfloat を true に設定します.

```
(%i30) ratepsilon;
(%o30)                              2.0e-8
(%i31) ratsimp((1+2.0e-8)*x);

rat replaced 1.00000002 by 1//1 = 1.0
(%o31)                                 x
(%i32) ratsimp((1+2.0e-7)*x);

rat replaced 1.0000002 by 5000001//5000000 = 1.0000002
                                   5000001 x
(%o32)                             ---------
                                    5000000
```

この例で示す様に,ratsimp を作用させた場合に,ratepsilon よりも小さな数が無視されて, 浮動小数点が有理数に変換されている事が判ります。

ratexpand

デフォルト値:[false]

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,false であれば, それらは再帰的な形式に置かれます.

ratfac

デフォルト値:[false]

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 パッケージを実際に呼ばずに, 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` となりますが, 各部分の因子は互いに素とは限りません.

ctensr(component tensor manipulation) パッケージでは,ratfac が true なら, ricci,einsten,riemann と weyl テンソルとスカラ曲率が自動的に因子分解されます. 但し, これはテンソルの成分が幾つかの項で構成されている事が判っている場合に対してのみ設定すべきです.

ratfac と ratweight の手法は互換性が無いので, 両者を同時に使っていけません.

ratprint

デフォルト値:[true]

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.

ratsimpexpons

デフォルト値:[false]

true であれば, 簡易化中に式の冪に対し, 自動的に ratsimp が実行されます.

ratweights

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

```
kill(...,ratweights)
```

と

```
save(...,ratweights);
```

は両方共に動作する.

ratweyl

デフォルト値:[]

weyl の共役テンソル成分の簡易化を制御する大域変数の一つです.true であれば成分は有理的に簡易化されます.facrat が true であれば結果は因子分解されたものになります.

ratwtlvl

デフォルト値:[false]

ratweight 函数を用いた式を纏める際の有理式 (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.

resultant

デフォルト値:[subres]

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
hline 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.

savefactors

デフォルト値:[false]

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

9.3　多項式に関する函数

bezout

bezout(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数 ⟩)

⟨ 多項式$_1$⟩ と ⟨ 多項式$_2$⟩ を ⟨ 変数 ⟩ を主変数とした時の係数で構成される行列を返します. 行列の大きさは, ⟨ 多項式$_1$⟩ と ⟨ 多項式$_p$⟩2 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.

この行列の determinant が ⟨ 多項式$_1$⟩ と ⟨ 多項式$_2$⟩ の ⟨ 変数 ⟩ による終結式になります. この bezout 函数は determinant 函数と組合せると resultant 函数の代替になります.

bothcoef

bothcoef(⟨ 式 ⟩, ⟨ 変数 ⟩)

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

coeff

coeff(⟨ 式 ⟩,⟨ 変数 ⟩,⟨ 次数 ⟩)

⟨ 式 ⟩ に含まれる ⟨ 変数 ⟩$^{⟨\text{次数}⟩}$ の項の係数を求めます.⟨ 次数 ⟩ を省略すると次数は 1 が設定されます. ⟨ 変数 ⟩ はアトムか真の部分式です. 具体的には,x,sin(x),a[i+1],x+y 等です.

尚, 真の部分式の場合,(x+y) が式の中に現れていなければならなりません. ここで ⟨ 変数 ⟩$^{⟨\text{次数}⟩}$ の項を正確に求める為には, 式の展開や因子分解が必要な場合もあります. 何故なら,coeff 函数では自動的に式の展開や因子分解が実行されれないからです.

```
(%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
```

combine

combine(⟨ 式 ⟩)

⟨ 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.

content

content(⟨ 多項式 ⟩,⟨ 変数$_1$ >,$\cdots$,< 変数$_n$⟩)

二成分のリストを返します. このリストは,⟨ 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をリストの第一成分とし, 第二成分は第一成分で多項式を割ったもの (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]
```

denom

denom(⟨ 有理式 ⟩)

⟨ 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 が返されます.

```
(%i40) denom((x^2+1)/(y^2+1)/2);
                                       2
(%o40)                              2 (y  + 1)
(%i41) denom(x^2+1);
(%o41)                                 1
(%i42) denom(1/2*x^2+1/2);
(%o42)                                 1
```

```
(%i43) denom((x^2+1)/2);
(%o43)                                 2
```

divide

divide(⟨ 多項式$_1$⟩,⟨ 多項式$_2$⟩,⟨ 変数$_1$ >,$\cdots$,< 変数$_n$⟩)

⟨ 多項式$_2$⟩ による ⟨ 多項式$_1$⟩ の商と剰余を計算します. 各多項式は ⟨ 変数$_n$⟩ を主変数とし, その他の変数は ratvars 函数に現れるものとします. 結果はリストで返却され, 第一成分が商, 第二成分が剰余となります.

```
(c1) divide(x+y,x-y,x);
(d1)                          [1, 2 y]
(c2) divide(x+y,x-y);
(d2)                        [ - 1, 2 x]
```

eliminate

eliminate([⟨ 方程式$_1$⟩,⟨ 方程式$_2$ >,$\cdots$,< 方程式$_n$ >],[< 変数$_1$⟩,⟨ 変数$_2$ >,$\cdots$,< 変数$_k$⟩])

方程式 (又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. k 個の ⟨ 変数$_1$ >,$\cdots$,< 変数$_k$⟩ が消去された n-k 個の式のリストを返します. 最初の ⟨ 変数$_1$⟩ は消去されて n-1 個の式を生成し,⟨ 変数$_2$⟩ 以降も同様です.k=n の場合, 結果リストは k 個の ⟨ 変数$_1$ >,$\cdots$,< 変数$_k$⟩ を持たない一つの式となります. それから,solve が最後の変数に対する終結式を解く為に呼出されます.

```
(%i1)  exp1:2*x^2+y*x+z;
                                          2
(%o1)                           z + x y + 2 x
(%i2) exp2:3*x+5*y-z-1;
(%o2)                         - z + 5 y + 3 x - 1
(%i3)  exp3:z^2+x-y^2+5;
                                  2    2
(%o3)                             z  - y  + x + 5
(%i4) eliminate([exp3,exp2,exp1],[y,z]);
            8         7         6          5
(%o4) [7425 x  - 1170 x  + 1299 x  + 12076 x
                                  4         3         2
                        + 22887 x  - 5154 x  - 1291 x

                        + 7688 x + 15376]

(%i5)  eliminate([x+y=2,2*x+3*y-5=0],[x,y]);
```

```
(%o5)                                [1]
(%i6)  eliminate([x+y=2,2*x+3*y-5=0],[x]);
(%o6)                               [y - 1]
(%i7) eliminate([x+y=2,2*x+3*y+5=0],[x]);
(%o7)                               [y + 9]
(%i8) eliminate([x+y=2,2*x+3*y+5=0],[x,y]);
(%o8)                                [- 9]
```

ezgcd

ezgcd(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, $\cdots$)

リストを返します. リストの成分は, 先ず第一成分が全ての多項式の GCD (最大公約因子) となり, 残りの元が GCD で割った値になります. この ezgcd では ezgcd アルゴリズムが常用されています.

factcomb

factcomb (⟨ 式 ⟩)

⟨ 式 ⟩ 中に現われる階乗の係数を階乗それ自体で置換して纏めます. 例えば,(n+1)*n!を (n+1)! にする事です. ここで,sumsplitfact[true] が false に設定されていれば,minfactorial が factcomb の後に適用されます.

factor

factor(⟨ 式 ⟩)

factor(⟨ 式 ⟩, ⟨p⟩)

⟨ 式 ⟩ を整数上で既約因子に分解します.

factor(⟨ 式 ⟩, ⟨p⟩) の場合, 最小多項式が ⟨p⟩ となる整数体上で式の因子分解を行います. 但し, 大域変数の設定によって動作が異なります.

- 大域変数 factorflag が false の場合, 有理式の整数部分の分解を抑制します. 大域変数 dontfactor に変数リスト (デフォルト値は空) を設定し, その変数に対して因子分解を行わない様にしても構いません. dontfactor リストに含まれた変数よりも (CRE 表現に仮定された変数の順序を用いて) 小さな任意の変数に対しても因子分解は実行されません.

- 大域変数 savefactors が true の場合, 式の因子を幾つかの同じ因子を含む式の因子分解の処理速度を向上させる為, 函数で保存します.

- 大域変数 berlefact が false の場合,kronecker 因子分解アルゴリズムが使われ, そうでなければ, 標準の berlekamp アルゴリズムが使われます.

- intfaclim は大きな整数の引数分解を行う時に試みられる最大の約数となります. false(この場合は factor を明示的に呼び出した場合) に設定した場合, 整数が fixnum(1 機械語長に適合するものです. 尚, それ以外の数は bignum と呼ばれます) の場合には, 整数の因数分解を試み

ます.intfaclim の設定では, 内部に対し factor の呼出に使われます.intfaclim を Maxima が大きな整数の因数分解に長時間を費さない様に再設定しても構いません.

- 大域変数 newfac が true の場合, 新しい因子分解ルーチンを使います.

factorout

factorout(⟨ 式 ⟩, ⟨ 変数$_1$⟩, ⟨ 変数$_2$⟩, $\cdots$)

⟨ 式 ⟩ を $f(\langle$ 変数$_1\rangle, \langle$ 変数$_2\rangle, \cdots) * g$ の形式の項の和に書換えます. ここで,g は factorout の引数の各変数を含まない式の積で,f は因子分解されたものになります.

factorsum

factorsum(⟨ 式 ⟩)

グループ単位で ⟨ 式 ⟩ の因子分解を試みます. このグループの項はそれらの和が因子分解可能なものです.`expand((x+y)^2+(z+w)^2)` の結果は復元可能ですが,`expand((x+1)^2+(x+y)^2)` の結果は共通項が存在する為に復元出来ません.

```
(c1) (x+1)*((u+v)^2+a*(w+z)^2),expand;
      2      2                            2      2
(d1) a x z  + a z  + 2 a w x z + 2 a w z + a w  x + v  x
                     2        2    2           2
        + 2 u v x + u  x + a w  + v  + 2 u v + u
(c2) factorsum(%);
                                   2          2
(d2)                 (x + 1) (a (z + w)  + (v + u) )
```

fasttimes

fasttimes(⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩)

多項式の積に対する特殊なアルゴリズムを用いて,⟨ 多項式$_1$⟩ と ⟨ 多項式$_2$⟩ の積を計算します. これらの多項式は, 多変数で各次数に対して係数が 0 でない項が多く, 両者共に殆ど同じ大きされなければ効果があまり出ません.

n と m を多項式の次数とすると, 古典的な積では n*m のオーダーで計算を行いますが, fasttimes を用いると max(n,m)**1.585 のオーダーとなります.

fullratsimp

fullratsimp(⟨ 式 ⟩)

⟨ 式 ⟩ に非有理式が含まれていれば, 普通, 簡易化した結果を返す際に, やや非力な非有理的 (一般的) 簡易化に続いて ratsimp を呼出します. 時には, その様な呼出しが一回以上必要であるかもしれません. fullratsimp は, この操作を簡易にしたものです.

fullratsimp は非有理的簡易化に続けて ratsimp を式に変化が生じなくなる迄適用します. 例えば, 式 `exp:(x^(a/2)+1)^2*(x^(a/2)-1)^2/(x^a-1)` に対しては,ratsimp(exp) により `(x^(2*a)-2*x^a+1)/(x^a-1)` が得られ,fullratsimp(exp) を実行すると `x^a-1` が得られます.

fullratsimp(⟨ 式 ⟩,⟨ 変数$_1$⟩,···,⟨ 変数$_n$⟩)
ratsimp と rat と同様に一つ又はそれ以上の引数を取ります.

gcd

gcd (⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨ 変数$_1$⟩,···)
p1 と p2 の最大公約数を計算します. 大域変数 gcd で適用されるアルゴリズムを決めます. 大域変数 gcd に ez,eez,subres,red や spmod を設定する事で,ezgcd, 新しい eez gcd, 副終結式 prs, 縮約や剰余と云ったアルゴリズムが各々選択されます.
gcd:false であれば,gcd(⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨ 変数$_1$⟩) は全ての ⟨ 変数 ⟩ に対して常に 1 を返します. 多くの函数 (例えば,ratsimp,factor, 等) は gcd を裏側で計算しています.
同次多項式に対しては,gcd:subres を用いる事を推奨します.
代数的整数の場合, 例えば,`gcd(x^2-2*sqrt(2)*x+2,x-sqrt(2))` の gcd を計算する為には algebraic が true でなければならず,gcd が ez であってはなりません.
subres は新しいアルゴリズムで red 設定を用いている場合, それを subres に変更すべきです
尚, 大域変数 gcd が false であれば, 式が CRE 表現に変換すると, 最大公約因子を取らない様になります. これは gcd が必要とされない場合には計算の高速化に繋がる事があります.

gcdex

gcdex(⟨ 多項式$_1$⟩,⟨ 多項式$_2$⟩,⟨ 変数 ⟩)
gcdex(⟨ 多項式$_1$⟩,⟨ 多項式$_2$⟩)
3 個の多項式を成分とするリスト [a,b,c] を返します. この多項式 c は引数の ⟨ 多項式$_1$⟩ と ⟨ 多項式$_2$⟩ の最大公約因子 (gcd) で, 多項式 a,b は $c = a * ⟨$ 多項式$_1$⟩ $+$ $b * ⟨$ 多項式$_2$⟩ を満すものです.
ここでのアルゴリズムはユークリッドの互除法に基くものです.
尚, 多項式が単変数の場合は ⟨ 変数 ⟩ を指定する必要はありませんが, 多変数の場合, 多項式を ⟨ 変数 ⟩ で指定した単変数の多項式と看做して gcd を計算します.
何故, 多変数多項式が相手の場合に, 変数の指定を行う必要があるかと言えば, 多変数多項式の場合は, 二つの多項式の最大公約因子が存在するとは限らないからです.
数学的には, 最大公約因子は二つの多項式が生成するイデアルの生成元となります. 通常, 多項式の係数が実数や複素数で, 考えている多項式が単変数のみ, 即ち, 多項式環 $k[x]$ であれば, 任意のイデアルは単項イデアル, 即ち, 一つの多項式だけで生成されるので, この場合は最大公約因子が必ず存在します.
その為, 多変数の場合には二つの多項式の主変数として変数を一つ選択する必要があります. 但し, その選択が妥当でなければ,gcdex は適当な答を返すだけです.

```
(%i16) gcdex(x^2+1,x^3+4);
                              2
                             x  + 4 x - 1  x + 4
(%o16)/R/                 [- ------------, -----, 1]
                                  17        17

(%i18) gcdex(x*(y+1),y^2-1,x);
```

```
                                        1
(%o18)/R/                         [0, ------, 1]
                                      2
                                     y  - 1
(%i19) gcdex(x*(y+1),y^2-1,y);
(%o19)/R/                         [1, 0, x y + x]
```

ここで, 最後の多変数の例で,`gcdex(x*(y+1),y^2-1,x)` の結果で, gcd として 1 を返している事に注意して下さい. この場合, 多項式環 k(y)[x] で処理を行っているので, 共通の因子として期待される y+1 にはなりません. ここで,k(y)[x] は x を主変数とした x と y の多項式環, つまり,x の多項式で, その係数が k 上の y の多項式となるものとして,x と y の k[x,y] 多項式を見直したものです. 一般的に可換環 k が ufd であれば,k[x] も ufd になる事が知られています. その為, ユークリッド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 ではなくにならず,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

gcfactor(⟨*Gauss* 整数 ⟩)

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

gfactor(⟨ 多項式 ⟩)

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

gfactorsum (⟨ 多項式 ⟩)

factorsum に似ていますが,factor の代りに gfactor が適用されます.

```
(%i58) gfactorsum(x^2+1);
(%o58)                          (x - %i) (x + %i)
(%i59) factor(x^2+1);
                                      2
(%o59)                               x  + 1
```

hipow

hipow (⟨ 多項式 ⟩, ⟨ 変数 ⟩)

⟨ 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
```

尚,hipow の逆の低い次数を検出する函数は lopow です.

lopow

lopow (⟨ 多項式 ⟩, ⟨ 変数 ⟩)

⟨ 多項式 ⟩ の部分式 ⟨ 変数 ⟩ の次数で, 明示的に現われものの中で最も低い次数を返します.

mod

mod (⟨ 多項式 ⟩) mod (⟨ 多項式 ⟩, ⟨ 整数 ⟩)

⟨ 多項式 ⟩ を大域変数 modulus で指定した値に対する剰余を計算します. 尚, 大域変数 modulus はデフォルトでは false の為,mod(⟨ 多項式 ⟩) を実行するとエラーになります.

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

num

num(⟨ 有理式 ⟩)

有理式の分子 (NUMerator) を返します.

quotient

quotient (⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数$_1$⟩, $\cdots$)

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

rat

rat(⟨ 式 ⟩,⟨ 変数$_1$ >,$\cdots$,< 変数$_n$⟩)

⟨ 式 ⟩ を CRE 表現に展開して変換を行い, 全ての項を共通の分母で纏めて, 浮動小数点を許容範囲 ratepsilon[2.0e-8] 以内で有理数に変換し, 分子と分母の最大公約因子を除去します. 指定されていれば, 変数は ratvars にて ⟨ 変数$_1$ >,$\cdots$,< 変数$_n$⟩ の順で順序付けられます. rat は+,-,*,/と冪乗の他の函数を一般的には簡易化しません.cer 形式に於けるアトムは一般形式のそれらと異なります.rat(x)-x は 0 と異なる内部表現の rat(0) で計算されます.

この rat 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.

```
(c1) ((x-2*y)**4/(x**2-4*y**2)**2+1)*(y+a)*(2*y+x)
       /(4*y**2+x**2);
                                          4
                                  (x - 2 y)
              (y + a) (2 y + x) (---------- + 1)
                                    2     2 2
                                   (x  - 4 y )
(d1)          -----------------------------------
                                 2    2
                               4 y  + x
(c2) rat(%,y,a,x);
                                2 a + 2 y
(d2)/R/                         ---------
                                 x + 2 y
```

ratcoef

ratcoef(⟨ 式$_1$⟩,⟨ 式$_2$⟩,⟨n⟩)

ratcoef(⟨ 式$_1$⟩,⟨ 式$_2$⟩)

与式の ⟨ 式$_2$⟩$^{\langle n\rangle}$ の係数を返します.⟨n⟩ が 1 の場合は ⟨n⟩ が省略出来ます. 尚, 返却値には,⟨ 式$_2$⟩ に含まれる変数を函数の変数としても含まないものです.

この様な係数が存在しない場合は零を返します.ratcoef は展開等を行って式を簡易化するので, 単純に ⟨ 式$_2$⟩$^{\langle n\rangle}$ の係数を返す coef と異った答を返します.

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

ratcoef(⟨ 式$_1$⟩,⟨ 式$_2$⟩,0) は, ⟨ 式$_1$⟩ から ⟨ 式$_2$⟩ を含まない項の和を返します. その為,⟨ 式$_2$⟩ が負の冪の項に含まれれば,ratcoef は使ってはいけません.⟨ 式$_1$⟩ が有理的に簡易化されていれば, 係数は予期した様に現れないかもしれません.

```
(c1) s:a*x+b*x+5$
(c2) ratcoef(s,a+b);
(d2)                x
```

ratdenom

ratdenom (⟨ 有理式 ⟩)

⟨ 有理式 ⟩ の分母を計算します.¡有理式¿が一般形式で, 一般式の結果を必要とするのであれば,dem 函数を用いるべきです.

ratdiff

ratdiff(⟨ 有理式 ⟩,⟨ 変数 ⟩)

⟨ 有理式 ⟩ の微分を ⟨ 変数 ⟩ で行います. 有理式に対しては diff よりも処理が速く, 計算結果は CRE 表現になります. 尚,ratdiff 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 diff を使いましょう.

```
(c1) (4*x**3+10*x-11)/(x**5+5);
                                      3
                                   4 x  + 10 x - 11
(d1)                               ----------------
                                        5
                                       x
(c2) modulus:3$
(c3) mod(d1);
                                    2
                                   x  + x - 1
(d3)                         --------------------
                              4    3    2
                             x  + x  + x  + x + 1
(c4) ratdiff(d1,x);
                             5    4    3
                            x  - x  - x  + x - 1
(d4)                 ------------------------------
                      8    7    5    4    3
                     x  - x  + x  - x  + x  - x + 1
```

ratdisrep

ratdisrep (⟨ 式 ⟩)

CRE 表現から一般形式に引数を変換します. 伝染を止めたり, 非有理的な文脈 (context) の中で有理函数を利用する場合, これは時々便利です. 殆どの CRE 函数は CRE と非 CRE 式に対して作用するが, 答は異った形式を取ります.ratdisrep が引数に対して非 CRE 表現で与えられれば, 引数を変更しないで返します.

ratexpand

ratexpand (⟨ 式 ⟩)

和の積や指数の和を掛け, 共通の分子で因子を纏め, 分子と分母の共通約数を通分し, 分子を分母によって割られた項へと分割して ⟨ 式 ⟩ の展開を行う. これは ⟨ 式 ⟩ を CRE 表現に変換し, それからを一般形式に戻して達成している.

ratexpand に影響を与える大域変数には ratexpand,ratdenomdivide,keepfloat があります.

```
(c1) ratexpand((2*x-3*y)**3);
                3         2       2        3
(d1)      - 27 y  + 54 x y  - 36 x  y + 8 x
(c2) (x-1)/(x+1)**2+1/(x-1);
                              x - 1       1
(d2)                          -------- + -----
                                      2   x - 1
                              (x + 1)
(c3) expand(d2);
                          x              1           1
(d3)                -------------- - ------------ + -----
                     2              2             x - 1
                    x  + 2 x + 1   x  + 2 x + 1
(c4) ratexpand(d2);
                             2
                          2 x                 2
(d4)                  ---------------- + ----------------
                       3    2             3    2
                      x  + x  - x - 1    x  + x  - x - 1
```

ratnumer

ratnumer(⟨ 有理式 ⟩)

⟨ 有理式 ⟩ の分子を取り出します. 一般形式の¡有理式¿に対して, CRE 表現の結果が不要であれば,num 函数を使います.

ratnump

ratnump (⟨ 式 ⟩)

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

ratp

ratp (⟨ 式 ⟩)

⟨ 式 ⟩ が CRE か拡張 CRE 表現であれば true, それ以外は false となります.

ratsimp

ratsimp(⟨ 式 ⟩)

ratsimp(⟨ 式 ⟩, ⟨ 変数$_1$, $\cdots$, < 変数$_n$⟩)

非有理的函数に対して, 有理的に ⟨ 式 ⟩ とその部分式の全てを引数も含めて (ratexpand の様に) 簡易化します. 結果は二つの多項式の商として, 再帰的な形式で返されます. 即ち, 主変数の係数は他の変数の多項式となっており, その係数もまた変数の順序に沿って, 主変数の次に順序の高い変数の多項式の係数と纏められています. 変数は ratexpand の様に非有理的函数 (例えば,sin(x**2+1)) を含みますが, ratsimp で非有理的函数に対する引数は有理的に簡易化されます. ratsimp は ratexpand に影響を与える幾つかの大域変数の影響を受ける事に注意して下さい.

尚,ratsimp(⟨ 式 ⟩, ⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩) で,ratvars に変数の ⟨ 変数$_1$⟩, $\cdots$ を設定した場合と同様に, この変数の並びの順序で有理的簡易化を行います.

```
(c1) sin(x/(x^2+x))=%e^((log(x)+1)**2-log(x)**2);
                                              2          2
                     x          (log(x) + 1)  - log (x)
(d1)          sin(------) = %e
                   2
                  x  + x
(c2) ratsimp(%);
                                  1          2
(d2)                        sin(-----) = %e x
                                x + 1
(c3) ((x-1)**(3/2)-(x+1)*sqrt(x-1))/sqrt((x-1)*(x+1));
                        3/2
                (x - 1)    - sqrt(x - 1) (x + 1)
(d3)            ---------------------------------
                    sqrt(x - 1) sqrt(x + 1)
(c4) ratsimp(%);
                                 2
(d4)                       - -----------
                             sqrt(x + 1)
(c5)  x**(a+1/a),ratsimpexpons:true;
                   2
                  a  + 1
                  ------
                    a
(d5)              x
```

ratvars

ratvars (⟨ 変数$_1$ >, $\cdots$, < 変数$_n$⟩)

n 個の変数の引数でリストを構成します.ratvars 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, それを有理式の主変数とします. 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 の引数は変数, 或いは非有理的函数 (sin(x) の様な函数) のいず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

ratweight (⟨ 変数$_1$⟩, ⟨ 重み$_1$ >, $\cdots$, < 変数$_n$⟩, ⟨ 重み$_n$⟩)

⟨ 重み$_i$⟩ を ⟨ 変数$_i$⟩ に割当てます. その重みが大域変数 raywtlvl の値 [デフォルト値は false で, 切捨てなしを意味する] を超過した時に, 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*v1**2*v2 の重みは 2*w1+w2 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 と ratweight の手法は互換性がないので, 両方同時に使えません.

```
(c5) ratweight(a,1,b,1);
(d5)                          [[b, 1], [a, 1]]
(c6) exp1:rat(a+b+1)$
(c7) %**2;
                      2                   2
(d7)/R/              b  + (2 a + 2) b + a  + 2 a + 1
(c8) ratwtlvl:1$
(c9) exp1**2;
(d9)/R/                        2 b + 2 a + 1
```

remainder

remainder (⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数$_1$⟩, $\cdots$)

⟨ 多項式$_2$⟩ による ⟨ 多項式$_1$⟩ の割算による剰余を計算します.

resultant

resultant (⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩, ⟨ 変数 ⟩)

二つの ⟨ 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.

尚, 終結式は ⟨ 多項式$_1$⟩ と ⟨ 多項式$_2$⟩ が共通の定数の因子を持つ場合に限って零になる多項式です. ⟨ 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. 尚,⟨ 多項式$_1$⟩, ⟨ 多項式$_2$⟩ が因子分解可能であれば resultant を呼び出す前に factor を呼出すと良いでしょう.

showratvars

showratvars (⟨ 式 ⟩) ⟨ 式 ⟩ の ratvars(CRE 変数) のリストを返します.

```
(%i30) exp:x^2+y^2+z^3;
                                  3    2    2
(%o30)                           z  + y  + x
(%i31) showratvars(exp);
(%o31)                             [x, y, z]
```

sqfr

sqfr (⟨ 式 ⟩)

factor と似ていますが, 多項式因子は無平方 (square-free) となります. ここで無平方であるとは, 多項式の因子 fi とその微分 dfi が共通の零点を持たない事を言います. 多項式式 a の無平方因子分解は $\pi_{i=1}^{n} a_i^i$ で, 各 a_i に関しては,$i \neq j$ の場合,gcd(a_i, a_j)=1 となるものを呼びます.Maxima の sqfr は与えられた式に対し, この無平方分解を計算します.

では, 無平方分解 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 は式を徹底して分解しているのに対し,sqfr は程々で止めている事です.

この様に, 多項式を無平方因子分解は通常の因子分解程の手間をかけません. 更に, 有理多項式の積分計算では, 無平方な因子に分母を分解して, 各因子を分母に持つ式に変形して積分を行うアルゴリズムもあります. この無平方分解は幅広く利用されています.

tellrat

tellrat (⟨ 多項式 ⟩)

Maximaが知っている代数的整数環に整数係数の多項式の解 (代数的整数数) を追加します.Maximaには最初に%i と全て整数といった根があります. tellrat(x) は有理函数に含まれる x に 0 を代入する事を意味します.

untellrat 函数は, これは tellrat による写像の核 (有理函数に含まれる x に 0 を代入する操作の事) を取り,tellrat の特性を除去するものです. tellrat を多変数多項式で実行する場合, 例えば,`tellat(x^2-y^2);x^2` に対して `y^2` を代入するかどうか等々の曖昧さが生じる. 例えば利用者が `tellrat(y^2=x^2)` として `x^2` で `y^2` を置き換えたければ, システムは特別の順序を採用する.tellrat と untellrat の両方は任意の数の因子を取る事が可能であり,tellrat() で現在の代入のリストを返します.

tellrat が多項式を被約する時, 零因子で分母の有理化を試みない様に注意します. 例えば,`tellrat(w^3-1)$ algebra` は零による割算になる. このエラーは `ratalgedenom:false` と設定する事で回避出来ます.

大域変数 algebraic は代数的整数の簡易化で効果を発揮する為に true に設定していなければなりません. 又,tellrat を実行したからといって, いきなり全ての入力が tellrat で設定した性質で評価される訳ではありません.tellrat で設定した性質を考慮した評価を行う為には,algebraic:true とするか,ev 函数と algebraic オプション付で式を評価する必要があります.

```
(%i1) tellrat(x^2+1);
                                    2
(%o1)                              [x  + 1]
(%i2) rat(x^2+1);
                                    2
(%o2)/R/                           x  + 1
(%i3) ratsimp(x^2+1);
                                    2
(%o3)                              x  + 1
(%i4) ev(rat(x^2+1),algebraic);
(%o4)/R/                             0
(%i5) ev(ratsimp(x^2+1),algebraic);
(%o5)                                0
(%i6) ev(ratsimp(x^3+1),algebraic);
(%o6)                              1 - x
(%i7) algebraic:true;
(%o7)                              true
(%i8) ratsimp(x^3+1);
(%o8)                              1 - x
(%i9) ratsimp(x^2+1);
(%o9)                                0
(%i10) rat(x^2+1);
(%o10)/R/                            0
```

totaldisrep

totaldisrep (⟨式⟩)

CRE 表現から一般形式へ⟨式⟩の全ての部分式を変換します.⟨式⟩が CRE 表現であれば,ratdisrep と同じ事になりますが, そうでなければ,totaldisprep がそれを"totally disrep"すれば,ratdisrep は⟨式⟩を変更せずに返す事はありません. これは ratdisprep を実行している式, 例えば, 方程式, リスト, 行列等で幾つかの CRE 表現の部分式を持つものに便利です.

untellrat

untellrat (⟨x⟩)

tellrat で指定した tellrat 属性を取り除きます.

本来,tellrat でやっている事は,Maxima に Maxima の式を用いて同値関係を導入する事です. つまり, 元々の集合から,tellrat による同値関係を入れた集合への包含写像から誘導される自然な写像を考えており,tellrat で指定した式がその写像の核を生成しています.

例えば,tellrat(x^ 2+1) を実行すると, 包含写像から誘導される自然な写像の核は x^2+1 で生成される多項式環のイデアルになります. ここで x の整数係数多項式 f に対して ev(f(x),algebraic) を実行すると $R[x]/\langle x^2+1\rangle$ で評価が行なわれます

この処理を例で示しましょう.

```
(%i11) tellrat(x^2+1);
                                    2
(%o11)                             [x  + 1]
(%i12)  ev(rat(x^3+1),algebraic);
(%o12)/R/                           - x + 1
(%i13) untellrat(x);
(%o13)                               []
(%i14) ev(rat(x^3+1),algebraic);
                                    3
(%o14)/R/                           x  + 1
```

上の例では, 最初に変数 x に対し,tellrat(x^ 2+1) で x が代数的整数で $x^2+1=0$ を満す事を設定しています, その為,tellrat では x は純虚数として扱われます. ところが untellrat(x) で, この x の性質を除去しています. この untellrat の後には,tellrat で入れた x の評価が実行されません.

次に, 複数の変数に対して tellrat と untelrat を実行した例を示します.

```
(%i21)  tellrat(x^2+1,y^2+1);
                                 2       2
(%o21)                          [y  + 1, x  + 1]
```

```
(%i22) ev(rat(x^3+1+y^3+y),algebraic);
(%o22)/R/                              - x + 1
(%i23)  untellrat(y);
                                         2
(%o23)                                 [x  + 1]
(%i24)  ev(rat(x^3+1+y^3+y),algebraic);
                                    3
(%o24)/R/                          y  + y - x + 1
```

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

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

尚,tellrat で入力可能な多項式は主変数に関して monic(係数が 1) なものでなければならなりません. 又, 多変数の場合,untellrat は主変数 (naimvar) 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*y1 に関しては主変数が z で係数が y1 となる為にエラーになります. 但し,2*x+y+z+1 の様に主変数 z が monic でありさえすれば問題はありません.untellrat は主変数のみに使える事も上の例から分ります.

主変数はデフォルトでは与えられた多項式の変数の中で, 逆アルファベット順で最も大きなものです. この主変数は declare を用いて任意のアトムを主変数とする事が可能です.

第10章　級数について

10.1　級数データの扱い

$c^{\inf}$ – 可微分函数に対して,taylor や powerseries で級数に変換が可能です. taylor と powerseries の違いは,taylor が表示桁数を指定して, 級数に表示するのに対し,powerseries はより一般的な級数として計算します.

以下に sin を原点付近で taylor と powerseries を使って級数展開してみましょう. 尚,taylor で x の次数が 10 迄に制限しています.

```
(%i1) ts:taylor(sin(x),x,0,10);
                           3    5      7       9
                          x    x      x       x
(%o1)/T/             x - -- + --- - ---- + ------ + . . .
                          6    120   5040   362880
(%i2)  ps:powerseries(sin(x),x,0);
                           inf
                           ====         i1  2 i1 + 1
                           \       (- 1)   x
(%o2)                       >      -----------------
                           /          (2 i1 + 1)!
                           ====
                           i1 = 0
```

この様に,taylor では x の冪の列となっていますが,powerseries では多項式の和として表現されています. この二つのデータが内部表現, 即ち, LISP の側のデータがどの様に記述されているか調べてみましょう. このデータを見る簡単な方法は,LISP の側からデータを見る事です. この為に,:lisp 関数を使います.:lisp 関数の引数として,LISP の S 式を与えると,Maxima はその結果を出力ラベルを付けずにそのまま表示します. Maxima のアトムの先頭に$を付けたものが,LISP 側のアトムに対応するので, ts のデータを見たければ,:lisp 関数の引数として$ts を与えれば良いのです.

```
(%i3) :lisp $ts;
((MRAT SIMP (((%SIN SIMP) $X) $X) (sin(x)13183 X13184)
  (($X ((10 . 1)) 0 NIL X13184 . 2)) TRUNC)
 PS (X13184 . 2) ((10 . 1)) ((1 . 1) 1 . 1)
    ((3 . 1) -1 . 6) ((5 . 1) 1 . 120)
 ((7 . 1) -1 . 5040) ((9 . 1) 1 . 362880))
(%i3) :lisp $ps;
((%SUM SIMP)
 ((MTIMES SIMP) ((MEXPT SIMP) -1 $I1)
  ((MEXPT SIMP) ((MFACTORIAL SIMP)
                ((MPLUS SIMP) 1 ((MTIMES SIMP) 2 $I1))) -1)
  ((MEXPT SIMP) $X ((MPLUS SIMP) 1
                ((MTIMES SIMP RATSIMP) 2 $I1))))
 $I1 0 $INF)
```

この様に,taylor と powerseries による結果の内部表現を見ると, 全く異なったデータになっている事が判ります. 先ず,taylor は多項式の CRE 表現を拡張した表現となっていますが,powerseries は多項式の一般表現を利用したものになっています.

この taylor や powerseries 函数とは逆に与えられた級数を函数に戻す函数として,nusum,revert や revert2 があります. 但し,numsum は漸化式,revert と revert2 は多項式近似以上のものではなく, 元の函数の復元が行えるとは限りません.

10.2 級数展開に関連する大域変数

cauchysum

デフォルト値:[false]

二つの無限和 (inf を上限として持つ和) の積で sumexpand が true,cauchysum が true に設定されていれば,cauchy 積が通常の積の代りに用いられます.cauchy 積では, 内部和の添字が他と独立して変化する添字ではなく, 外側の添字変数の函数となります. 例えば,`sum(f(i),i,0,inf)*sum(g(j),j,0,inf)` は `sum(sum(f(i)*g(j-i),i,0,j),j,0,inf)` となります.

maxtayorder

デフォルト値:[true]

true であれば, 切詰められた taylor 展開の代数操作で taylor が正確と確証される迄の項を保持する.

niceindicespref

デフォルト値:[i,j,k,l,m,n]

niceindices が和や積で添数を探す為に用いるリスト. これは利用者が niceindices が如何に良い添数を見付けるか, その順序を設定する事が出来ます. 例えば,`niceindicespref:[q,r,s,r,index]` の様にします.

ある特定の和で, 添数としてこれらを何れも用いる事が出来ない事を niceindices が見付けた場合, 最初のものを基準に用います. 更に, リストの添字を使い果すと,q0, その次に q1 等々と試みます.

powerdisp

デフォルト値:[false]

true であれば, 逆の順序で項の和が表示されます. この場合は, 多項式も切り詰められた冪級数の様に, 最低次数の項が最初に表示されます.

psexpand

デフォルト値:[false]

true であれば, 拡張有理函数式は全体的に展開される (ratexpand もこの原因となる).false であれば, 多変数式が丁度有理函数パッケージに含まれるものであるかの様に表示される.

psexpand:multi であれば, 同じ総次数で項が互いに纏められる.

taylordepth

デフォルト値:[3]

零でない項がまだ存在すれば,pow*2n̂ に達するまで taylor は g(var) の展開の次数を倍にして行く. ここで n は変数 taylordepth[3] の値である.

taylor_logexpand

デフォルト値:[true]

taylor 級数中の対数函数の展開を制御します.true であれば, 全ての log は全面的に展開され, 対数函数の同一性に関連する零の判定問題は, この展開の過程の邪魔にはなりませんが, この手法は分枝の情報を無視するので, 数学的に常に正しいとは限りません.

taylor_logexpand が false に設定されていれば,log の展開は形式的冪級数を得る必要がある場合に限られます.

taylor_order_coefficients

デフォルト値:[true]

式中の係数の順序を制御します. デフォルト値で taylor 級数の係数は正規順序で並べられます.

taylor_truncate_polynomials

デフォルト値:[true]

false であれば,taylor に入力される多項式は無限精度を持つものとして考えられます. そうでなければ, 切捨水準を基準として切捨てられます

verbose

デフォルト値:[false]

true であれば,powerseries の実行状況を処理に沿って表示して行きます.

10.3 級数展開に関連する函数

deftaylor

deftaylor(func,exp)

任意の 1 変数函数 func に対し,0 の近傍に於ける)aylor 級数として exp を定めます. exp はその変数の多項式や sum 函数を用いたものでも構いません.deftaylor で与えられた情報を表示する為に powerseries(f(x),x,0) が使えます.

```
(%i1) deftaylor (f(x), x^2 + sum(x^i/(2^i*i!^2), i, 4, inf));
(%o1)                          [f]
(%i2) powerseries (f(x), x, 0);
                       inf
                       ====       i1
                       \         x          2
(%o2)                   >      -------- + x
                       /        i1    2
                       ====    2   i1!
                       i1 = 4
(%i3) taylor (exp (sqrt (f(x))), x, 0, 4);
                       2          3           4
                      x    3073 x    12817 x
(%o3)/t/     1 + x + -- + ------- + -------- + . . .
                      2     18432      307200
```

niceindices

niceindices (⟨ 式 ⟩)

⟨ 式 ⟩ を取り, 和と積の全ての添数を変更して幾らかより判り易いものに変更する. 添数として i が使え,i が内部式で用いられていれば, ちゃんとした添数を見つけるまで j,k,l,m,n,i0,i1,i2,i3,i4,··· の順で添字を振って行きます.

nusum

nusum (⟨ 式 ⟩, ⟨ 変数 ⟩, ⟨*low*⟩, ⟨*high*⟩)

R.W.Gosper による決定手順を用いた ⟨ 変数 ⟩ に対する ⟨ 式 ⟩ の不定和を計算します.⟨ 式 ⟩ と潜在的な解は n 次の冪乗, 階乗, 二項係数, そして有理函数の積として表現可能なものでなければなりません. 尚, ここでは定和と不定和という言葉を定積分と不定積分の類似として用いています. 不定和を取るとは, 変数の複数の区間, 例えば 0 から inf へと云ったものではなく, その上での和に対する近い公式を与える事を意味します. その為, 二項級数の一般的な部分和に対し, 公式が存在しないので,nusum は使えません.

pade

pade(⟨*taylor* 級数 ⟩, ⟨ 分子の上限の次数 ⟩, ⟨ 分母の上限の次数 ⟩)

有理函数のリストを返します.

この有理函数は taylor 展開で分子と分母の次数の和が taylor 級数展開の切捨て水準以下のものとなっています. 即ち, 返却される有理函数はあくまでも与えられた taylor 展開の有理函数による最上の近似で, 指定した次数の上限を満すものになります.

最初の引数は単変数の taylor 級数, 第二と第三の引数は正整数であり, 近似有理函数の分子と分母の次数の上限を定めます.

尚,pade の最初の引数は laurent 級数でも構いません. この場合, 総次数が冪級数の長さ以下の全ての有理函数が返されます. 尚, ここでの

⟨ 総次数 >=< 分子の次数 > + < 分母の次数 ⟩ 冪級数の長さ=切捨ての水準+1-min(0, 級数の次数)

となります.

```
(c15) ff:taylor(1+x+x^2+x^3,x,0,3);
                                      2    3
(d15)/t/                     1 + x + x  + x  + . . .
(c16) pade(ff,1,1);
                                        1
(d16)                                [- -----]
                                      x - 1
(c1) ff:taylor(-(83787*x^10-45552*x^9-187296*x^8
                 +387072*x^7+86016*x^6-1507328*x^5
                 +1966080*x^4+4194304*x^3-25165824*x^2
                 +67108864*x-134217728)
       /134217728,x,0,10);
(c25) pade(ff,4,4);
(d25) []
```

この冪級数表現では分子/分母の次数が 4 の有理函数を持たない. 一般的に, 未知係数を解く為に十分なだけの分子と分母の次数が, 少なくとも足し合わせて冪級数の次数に到達するものを持っていなければならない.

```
(c26) pade(ff,5,5);
(d26) [-(520256329*x^5-96719020632*x^4-489651410240*x^3
           -1619100813312*x^2 -2176885157888*x-2386516803584)
         /(47041365435*x^5+381702613848*x^4+1360678489152*x^3
                +2856700692480*x^2
                +3370143559680*x+2386516803584)]
```

powerseries

powerseries (⟨式⟩,⟨変数⟩,⟨点⟩) ⟨点⟩(無限の場合は inf) の近傍で,⟨変数⟩に対する⟨式⟩の一般的な級数展開を生成します. 尚,powerseries で式を展開出来なければ,taylor 函数で級数の最初の幾つかの項を求められるかもしれません.

尚,verbose を true にしていると, 以下の様な進行状況が表示されます.

```
(c1) verbose:true$
(c2) powerseries(log(sin(x)/x),x,0);
can't expand
                                 log(sin(x))
so we'll try again after applying the rule:
                                       d
                                    / -- (sin(x))
                                    [ dx
                        log(sin(x)) = i ----------- dx
                                    ]   sin(x)
                                    /
in the first simplification we have returned:
                               /
                               [
                               i cot(x) dx - log(x)
                               ]
                               /
                     inf
                     ====        i1  2 i1             2 i1
                     \      (- 1)   2     bern(2 i1) x
                      >     ------------------------------
                     /                i1 (2 i1)!
                     ====
                     i1 = 1
(d2)                 ------------------------------------
                                    2
```

revert と revert2

revert (⟨式⟩,⟨変数⟩)

revert2(⟨式⟩,⟨変数⟩,⟨次数⟩)

taylor 級数から⟨式⟩を多項式で復元します.⟨変数⟩は taylor 展開で用いた変数で,0 の近傍での taylor 展開に対してのみに使えます.

revert では taylor 級数の表示されている項を最高次数とする多項式に纏めてしまいますが,revert2 では表示されていなくても, 指定した ⟨ 次数 ⟩ を越えない多項式に纏めます.

これら函数を利用したければ, 予め load(revert) を実行しておく必要があります.

```
(%i39) load("revert");
(%o39)      /usr/local/share/Maxima/5.9.2/share/calculus/revert.mac
(%i40) t1:taylor(sin(x),x,0,5);
                                    3    5
                                   x    x
(%o40)/t/                     x - -- + --- + . . .
                                   6    120
(%i41) revert(t1,x);
                                  5       3

                                9 x  + 20 x  + 120 x
(%o41)/R/                       --------------------
                                        120
(%i42) revert2(t1,x,10);
                                 9      7      5    3
                           779 x    2 x    3 x    x
(%o42)                     ------ + ---- + ---- + -- + x
                           25920     45     40    6
```

taylor

taylor (⟨ 式 ⟩, ⟨ 変数 ⟩, ⟨ 点 ⟩, ⟨ 次数 ⟩)

⟨ 式 ⟩ を ⟨ 点 ⟩ の近傍で ⟨ 変数 ⟩ の taylor 級数 (必要であれば laurent 級数) として展開します. (⟨ 式 ⟩−⟨ 点 ⟩)$^{⟨\text{次数}⟩}$ までの項が生成されます. ここで,⟨ 式 ⟩ が f(⟨ 変数 ⟩)/g(⟨ 変数 ⟩) の型で,g(⟨ 変数 ⟩) に ⟨ 次数 ⟩ に達する項が無い場合,taylor は g(⟨ 変数 ⟩) を次数 2 ∗ ⟨ 次数 ⟩ まで展開しようとします.

非零の項がまだ存在し, 変数 taylordepth の値が n であれば, (⟨ 次数 ⟩ ∗ 2)^n に達する迄,g(⟨ 変数 ⟩) の展開の次数を倍にします.

taylor(⟨ 式 ⟩, [⟨ 変数$_1$⟩, ⟨ 点$_1$⟩, ⟨ 次数$_1$⟩], [⟨ 変数$_2$⟩, ⟨ 点$_2$⟩, ⟨ 次数$_2$⟩], ···) は ⟨ 点$_i$⟩ の近傍で ⟨ 変数$_i$⟩ の ⟨ 次数$_i$⟩ で切り捨てられた冪級数を返します.

taylor(⟨ 式 ⟩, [⟨ 変数$_1$⟩, ⟨ 変数$_2$⟩, ···], ⟨ 点 ⟩, ⟨ 次数 ⟩).

ここで,⟨ 点 ⟩ と ⟨ 次数 ⟩ は変数のリストに対応するリストで置換えても構いません. つまり, 各リストの n 番目の項目は互いに関連するものとなります.

taylor(⟨ 式 ⟩, [⟨ 変数 ⟩, ⟨ 点 ⟩, ⟨ 次数 ⟩, ⟨*asymp*⟩]) は (⟨ 変数 ⟩ − ⟨ 点 ⟩) の負の冪による式の展開を与えます. 最も高い次数の項は (⟨ 変数 ⟩ − ⟨ 点 ⟩)$^{-⟨\text{次数}⟩}$ となります. asymp は構文指定で変更が効きません. 尚, 展開の制御は大域変数の taylor_logexpand が効力を持ちます.

- maxtayorder が true であれば,(切捨てられた)taylor 級数の代数操作中に, taylor は正確にな

る様に可能な限りの多くの項を保持しようとします.

- psexpand が true であれば, 展開した有理函数式を広く展開したものを表示します.(ratexpand もこの処理を行います).false であれば, 多変数式が有理函数パッケージ内部で表示されます.
- psexpand:multi であれば, 全次数が n の変数の項で互いにグループ化されます.

taylorinfo

taylorinfo (⟨ 式 ⟩)

⟨ 式 ⟩ が taylor 級数で無ければ false を返します. taylor 級数であれば,taylor 展開の項を記述するリストのリストが返されます.

```
(%i10) taylor((1-y)/(1-x),x,0,4,[y,a,inf]);

(%o10)/T/ 1 - a - (y - a) + (1 - a - (y - a)) x
                                                2
                          + (1 - a - (y - a)) x
                                                3
                          + (1 - a - (y - a)) x
                                                4
                          + (1 - a - (y - a)) x  + . . .
(%i11) taylorinfo(%o10);
(%o11)                     [[y, a, inf], [x, 0, 4]]
```

taylorp

taylorp(⟨ 式 ⟩)

述語函数であり,⟨ 式 ⟩ が taylor 級数表現である時に限って true を返します.

taylor_simplifier

taylor_simplifier

一引数の函数で, その引数は taylor が冪級数の係数の簡易化で用いられます.

taytorat

taytorat (⟨ 式 ⟩)

⟨ 式 ⟩ を taylor 形式から CRE 表現に変換します. つまり,rat(ratdisprep(exp)) に似ていますが処理はより速いものになります.

```
(%i1) exp2:taylor(sin(x),x,0,10);
                           3    5      7       9
                          x    x      x       x
(%o1)/T/             x - -- + --- - ---- + ------ + . . .
```

```
                          6    120   5040   362880
(%i2)  exp3:taytorat(exp2);
                   9       7         5          3
                  x  - 72 x  + 3024 x  - 60480 x  + 362880 x
(%o2)/R/          ----------------------------------------
                                   362880
(%i3) :lisp $exp2;
((MRAT SIMP (((%SIN SIMP) $X) $X) (sin(x)13183 X13184)
  (($X ((10 . 1)) 0 NIL X13184 . 2)) TRUNC)
 PS (X13184 . 2) ((10 . 1)) ((1 . 1) 1 . 1)
                ((3 . 1) -1 . 6) ((5 . 1) 1 . 120)
                ((7 . 1) -1 . 5040) ((9 . 1) 1 . 362880))
(%i3) :lisp $exp3;
((MRAT SIMP (((%SIN SIMP) $X) $X) (sin(x)13183 X13184))
 (X13184 9 1 7 -72 5 3024 3 -60480 1 362880) . 362880)
```

この例で示す様に, ラベルが/T/から/R/に変化している事に注意して下さい. 又, 内部表現も,CRE 表現に似た Taylor 級数展開から,X の多項式の CRE 表現に変換されている事にも注目して下さい.

trunc

trunc (⟨ 式 ⟩)

一般表現の ⟨ 式 ⟩ を切詰めた taylor 級数であるかの様に表示します.

```
(c155) exp1:x^2+x+1;
                                  2
(d155)                           x  + x + 1
(c156) exp2:trunc(x^2+x+1);
                                      2
(d156)                        1 + x + x  + . . .
(c157) is(exp1=exp2);
(d157)                              true
```

この例で示す様に, 表示は一見異なりますが,exp1=exp2 となり, データ自体に変化はありません.

unsum

unsum (⟨ 函数 ⟩, ⟨n⟩)

第一後退差分 ⟨ 函数 ⟩(⟨n⟩) − ⟨ 函数 ⟩(⟨n⟩ − 1) を計算します.

```
(c1) g(p):=p*4^n/binomial(2*n,n);
                                             n
                                          p 4
```

```
(d1)                        g(p) := ----------------
                                    binomial(2 n, n)
(c2) g(n^4);
                                      4  n
                                     n  4
(d2)                               ----------------
                                   binomial(2 n, n)
(c3) nusum(d2,n,0,n);
                          4        3       2              n
          2 (n + 1) (63 n  + 112 n  + 18 n  - 22 n + 3) 4      2
(d3)      ------------------------------------------------ - ------
                        693 binomial(2 n, n)                 3 11 7
(c4) unsum(%,n);
                                      4  n
                                     n  4
(d4)                               ----------------
                                   binomial(2 n, n)
```

(平成 17 年 12 月 13 日 (火))

第11章　式

11.1　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, 上記の 1+2+3 を入力すると 6 が返されますが,a:sin(x) の場合は x にどの様な値が割当てられているか, 或いは関連する大域変数の設定によって結果が異なります.

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

MAXIMA では表に表示されている式と内部の式は大きく異なります. 式の内部表現は LISP の前置式表現を全面に出したものです. 先ず,MAXIMA で変数 a や文字列"a"等の対照表を以下に示します.

表 11.1: maxima の内部表現：変数

a	$a
?a	a
"a"	& a
'a	((mquote) $a)

変数に関しては, 内部では先頭に$を付けたものになります. これに対して, MAXIMA で?が先頭に付く場合, 内部では?を外したもので置換えられます. これを利用して,LISP の函数を MAXIMA から実行させる為に, 先頭に?を付ける訳です.

では, 演算どの様になるのでしょうか？因に,LISP の函数は S 式の先頭に置かれます. 例えば,1+2 は LISP では (+ 1 2) となります.MAXIMA も内部で, これと同じ事を行っています.

表 tbl:internalops は MAXIMA の演算がどの様に内部で表現されているかを示すものです.

表 tbl:internalops に示す様に,MAXIMA の演算は,x+y の様に二つの被演算子の間にあった演算子が, 先頭に移され, その後に被演算子が並んでいますね. この様に全てが前置表現へと変換されています.

その上,x-y から x+(-y) として表現されています. これは単純に, 後の処理が楽になる為です.

表 11.2: maxima の内部表現 : 数値演算

x+y	((mplus simp) $x $y)
x-y	((mplus simp) $x ((mminus simp) $y)
x*y	((mtimes simp) $x $y)
x/y	((mquotient simp) $x $y)
x.y	((mnctimes simp) $x $y)
x^2 又は x**2	((mexpt simp) $x 2)
x^^2	((mncexpt simp) $x 2)

更に函数名は全て 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.

これは表 11.3 に示す論理演算子についても同様で, 基本的に MAXIMA の式は前置表現に変換されています.

表 11.3: MAXIMA の内部表現 : 論理演算

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)

割当の演算子は表 11.4 に示す表現です.

表 11.4: maxima の内部表現 : 割当

a:b	((msetq simp) $a $b)
a::b	((mset simp) $a $b)
a(x):f	((mdefine simp) (($a) $x) $f)

この割当で,`a:b` が LISP の `(setq a b)` に似た構造になっている事が分りますね. 函数定義も LISP の `(defun a(x) f)` に似た並びで内部表現されています.

MAXIMA の函数の内部表現を表 11.5 に示します.

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

表 11.5: maxima の内部表現：函数

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 $x 1)
integrate(a,b,c,d)	(($integrates simp) $a $b $c $d)
ìntecrate(aib,c,d)	((%integrate simp) $a $b $c $d)

函数の例で, 一階の微分を行う場合は,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.

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

MAXIMA の配列やリストの内部表現を表 11.6 に示します.

表 11.6: maxima の内部表現：函数

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)

最後に MAXIMA の制御文の内部表現を表 11.7 に示します.

表 11.7: maxima の内部表現：制御文

if a then b	((mcond simp) $a $b t $false)
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)

この様に,MAXIMA の式は LISP の式でもあります. その為, 部分式は LISP の S 式の部分 S 式となります. 式を操作する函数で, 部分式の位置を指定するものが幾つかあります. これらの函数は

先頭にある演算子の処理は別としても, 基本的に式を階層化して考えており, この内部表現に何等かの影響を受けたものとなっています.

11.2 大域変数

exptisolate

デフォルト値:[false]

true であれば,isolate(⟨ 式 ⟩,⟨ 変数 ⟩) を実行する際に,⟨ 変数 ⟩ に含まれる (%e の様な) アトムの指数項についても調べます.

isolate_wrt_times

デフォルト値:[false]

isolate 函数の動作に影響を与えます.

isolate_wrt_times が false の場合,isolate 函数は指定した変数を含まない項と含む項に分けて表示を行います.

true の場合, 更に式を分解し, 指定した変数を含む項も, 指定した積を除く項と指定した変数の項の積に分解して表示を行います.

```
(%i17) eq1:expand((a+b+x)^2);
                     2                   2           2
(%o17)              x  + 2 b x + 2 a x + b  + 2 a b + a
(%i18) isolate_wrt_times;
(%o18)                              false
(%i19) exp1:expand((a+b+x)^2);
                     2                   2           2
(%o19)              x  + 2 b x + 2 a x + b  + 2 a b + a
(%i20) isolate_wrt_times;
(%o20)                              false
(%i21) isolate(exp1,x);

                                2           2
(%t21)                         b  + 2 a b + a

                          2
(%o21)                   x  + 2 b x + 2 a x + %t21
(%i22) isolate_wrt_times:true;
(%o22)                              true
(%i23) isolate(exp1,x);

(%t23)                               2 a


(%t24)                               2 b
```

```
                              2
(%o24)                       x  + %t24 x + %t23 x + %t21
```

listconstvars

デフォルト値:[false]

listofvars 函数の動作に影響を与えます.

true であれば, 定数として宣言した変数と MAXIMA の数学定数である%e,%pi,%i が式に含まれていると,listofvars はこれらの定数も変数として加えたリストを返します.

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]
```

listdummyvars

デフォルト値:[true]

false であれば, 式の疑似変数は listofvars で返されるリストの中に含まれません. 尚, 疑似変数は sum 函数や product 函数等の添字や極限変数や定積分変として利用される変数です.

optimprefix

デフォルト値:[%]

optimize 函数で生成された記号に使用される前置詞です.

partswitch

デフォルト値:[false]

true であれば,part 函数で選択した部分式が式中に存在しなければ end が返されます.

false の場合はエラーメッセージが返されます.

11.3 式に関連する函数

args

args (⟨ 式 ⟩)

⟨ 式 ⟩ が函数の場合は函数の引数を返します. 一般の式の場合, 内部表現で第 0 階層を [で置換えたもの, 即ち, substpart("[",⟨ 式 ⟩,0) と同じ働きをします.

尚, この args と substpart の両方は inflag の設定に依存します.

disolate

disolate (⟨ 式 ⟩⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩)

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る事が出来ます.

例えば, 重積分での変数変換や, 積分変数を二つかそれ以上を巻込んだ変数変換で便利です. この函数を利用する為には予め load(disol) で読込む必要があります.

dispform

dispform (⟨ 式 ⟩)

dispform(⟨ 式 ⟩, *all*)

⟨ 式 ⟩ の外部表現を返します.MAXIMA の式には表示される外部表現と内部処理で用いられる内部表現があります.dispform では返却される値は外部表現となる為, 外部表現を扱う part と組合せると便利です.

dispform(⟨ 式 ⟩, *all*) で式全体を外部形式に変換します. この場合, 函数の引数として与えられた式も含めて処理が行われる為, part 函数の結果も異なる事があります. 例えば,exp:cos(sqrt(x)) の場合, freeof(sqrt,exp) と freeof(sqrt,dispform(exp)) は true となりますが, freeof(sqrt,dispform(exp,all)) は false となり, 函数の引数内部の式に対しても freeof が使える様になります.

freeof

freeof($\langle x_1 \rangle, \cdots, \langle x_n \rangle$, ⟨ 式 ⟩)

$\langle x_i \rangle$ 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.$\langle x_i \rangle$ はアトム, 添字付けられた名前, 函数, 或いは二重引用符””で括られた演算子となります.

尚, 函数内部で利用される疑似変数を $\langle x_i \rangle$ に指定した場合, その疑似変数が ⟨ 式 ⟩ に含まれていても true を返します.

inpart

inpart(⟨ 式 ⟩, ⟨ 整数$_1$⟩, $\cdots$, ⟨ 整数$_k$⟩)

part と似ていますが,part が MAXIMA に表示された ⟨ 式 ⟩ に対して作用しますが,inpart の場合は ⟨ 式 ⟩ の内部表現に対して直接作用するものです.

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. 和や積, 単一演算子 (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を払う必要があります.

isolate

isolate (⟨ 式 ⟩, ⟨ 変数 ⟩)

isolate 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_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.

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.

尚,isolate は式の展開を行いません.

```
(%i12) isolate_wrt_times:false;
(%o12)                                false
(%i13) exp1:expand((1+a+x)^2);
                          2                   2
(%o13)                   x  + 2 a x + 2 x + a  + 2 a + 1
(%i14) isolate(exp1,x);

                                    2
(%t14)                             a  + 2 a + 1

                              2
(%o14)                       x  + 2 a x + 2 x + %t14
(%i15) isolate_wrt_times:true;
(%o15)                                true
(%i16) isolate(exp1,x);

(%t16)                                 2 a

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

(%t17)                                a + 1

                                                2
(%o17)                               (x + %t17)
```

listofvars

listofvars (〈式〉)

〈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 が返すリストに, これらの定数が含められます.

但し, デフォルトの false の場合, 定数を除外したリストを返却します.

multthru

multthru (〈式〉)

multthru(⟨式$_1$⟩,⟨式$_2$⟩)

multthru (⟨式⟩) で⟨式⟩の式の部分展開を行います. つまり,⟨式⟩が $f_1 * f_2 * \cdots * f_n$ の形式で, 各因子の中で, 冪乗でない⟨式⟩中で最も左側の因子を f_i とすると, ⟨式⟩の f_i 以外の因子と f_i の項との積の和に分解します. 例えば,(x+1)$\hat{}$2*(z+1)*(y+1) の場合, 最も左側の因子 y+1 で式が展開され, (x+1)$\hat{}$2*(y+1)*z+(x+1)$\hat{}$2*(y+1) となります.

multthru(⟨式$_1$⟩,⟨式$_2$⟩) の場合, ⟨式$_2$⟩ の各項を⟨式$_1$⟩倍にします. つまり,multthru(⟨式$_1$⟩ * ⟨式$_2$⟩) と同値です.

尚,⟨式$_2$⟩には方程式を指定出来ます. この場合, 演算子=の二つの被演算子に⟨式$_1$⟩との積が返されます.

尚,multthru は冪乗された和の展開は行いません. この函数は和に対する可換, 或いは非可換積の分配に関して最も速いものです.

```
(%i18) multthru((x+1)^2*(z+1));
                                    2           2
(%o18)                        (x + 1)  z + (x + 1)
(%i19) multthru((x+1)^2*(y+1)^2*(z+1)^2,z+1);
                  2        2          2          2        2        2
(%o19)     (x + 1)  (y + 1)  z (z + 1)  + (x + 1)  (y + 1)  (z + 1)
(%i20) multthru((x+1)^2*(y+1)^2*(z+1)^2,x+1);
                    2        2        2          2        2        2
(%o20)     x (x + 1)  (y + 1)  (z + 1)  + (x + 1)  (y + 1)  (z + 1)

(%i21) multthru((x+1)^2*(z+1)*(y+1));
                             2                   2
(%o21)               (x + 1)  (y + 1) z + (x + 1)  (y + 1)
(%i22) multthru((x+1)^2*(y+1)^2*(z+1)^2,x^2+1=0);
         2        2        2        2          2        2        2
(%o22)  x  (x + 1)  (y + 1)  (z + 1)  + (x + 1)  (y + 1)  (z + 1)  = 0
```

nterms

nterms (⟨式⟩)

⟨式⟩を展開した時の項数を返却します. 但し, 函数の引数に関しては, その引数が幾つ項を持とうが, ただ一つの項として数えられる事に注意して下さい.

```
(%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
```

optimize

optimize (⟨式⟩)

⟨式⟩をより効率的に計算出来る MAXIMA の式を返却します. optimize は式の展開を行うものではありませんが, 式に含まれる共通部分式を内部変数で置換えて, 効率的に計算出来る様な式に変換します. この際に, block 文が用いられます. 但し, 共通部分式が無い場合は,⟨式⟩をそのまま返却します.

```
(%i40) optimize((x+1)^3+1/(x+1)^2+exp((x+1)^2));
                                                    2    %2     3   1
(%o40)      block([%1, %2], %1 : x + 1, %2 : %1 , %e   + %1  + --)
                                                                 %2

(%i41) ans1:solve( x^4+x^3+3*x-1=0,x);
              sqrt(5)   sqrt(25 - sqrt(5))   1
(%o41) [x = - ------- - ------------------ - -,
                 4                   1/4     4
                           2 sqrt(2) 5
      sqrt(5)   sqrt(25 - sqrt(5))   1
x = - ------- + ------------------ - -,
         4                   1/4     4
                  2 sqrt(2) 5
      sqrt(sqrt(5) + 25) %i   sqrt(5)   1
x = - --------------------- + ------- - -,
                    1/4          4      4
         2 sqrt(2) 5
    sqrt(sqrt(5) + 25) %i   sqrt(5)   1
x = --------------------- + ------- - -]
                   1/4          4      4
       2 sqrt(2) 5
(%i42) optimize(ans1);
                                                        1           1
(%o42) block([%1, %2, %3, %4, %5, %6, %7], %1 : -------, %2 : ----,
                                                    sqrt(2)        1/4
                                                                  5
                                           %3       %3
%3 : sqrt(5), %4 : sqrt(25 - %3), %5 : - --, %6 : --, %7 : sqrt(%3 + 25),
```

```
                                          4         4
          %1 %2 %4   1            %1 %2 %4   1         %1 %2 %7 %i         1
[x = %5 - -------- - -, x = %5 + -------- - -, x = - ------------ + %6 - -,
             2       4               2       4              2               4
    %1 %2 %7 %i         1
x = ----------- + %6 - -])
         2              4
```

part

part(⟨ 式 ⟩, ⟨n_1⟩, ⋯, ⟨n_k⟩)

⟨ 式 ⟩ の部分式を抽出する函数です. 取出す部分式は ⟨n_1⟩, ⋯, ⟨n_k⟩ で指定されます. この part 函数は MAXIMA の表示された式から部分式を取出すもので, 内部表現から部分式を取出す函数は inpart になります.

部分式の取出し方は, 最初に ⟨ 式 ⟩ から ⟨n_1⟩ 番目の部分式を取出します. 今度は取出した部分式から ⟨n_2⟩ 番目の成分を取出します. 以降同様に ⟨$n_k - 1$⟩ 番目の部分式から ⟨n_k⟩ 番目の成分を取出し, この部分式を結果として返します.

```
(%i15) part((x+1)^3+2,1);
                                        3
(%o15)                                (x + 1)
(%i16) part((x+1)^3+2,1,1);
(%o16)                                  x + 1
(%i17) part((x+1)^3+2,1,1,1);
(%o17)                                    x
```

この part の動作を見て判る様に,MAXIMA の式には階層構造が入っています. これは MAXIMA のデータがリストで結局表現される為でもあります. その事もあって,part は MAXIMA のリストや行列でも, その成分の取出しに使えます.

part 函数の最後の引数が添字のリストであれば, 幾つかの部分式が引掛り, 各々がそのリストの添数に関連したものとなります. 例えば,part(x+y+z,[1,3]) は z+x となります.

大域変数 piece には 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 の場合, 式に指定した成分が存在しない時に end を返します.false の場合は, エラーメッセージを返します.

partition

partition (⟨ 式 ⟩, ⟨ 変数 ⟩)

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]]
```

pickapart

pickapart (⟨ 式 ⟩,⟨ 整数 ⟩)

⟨ 整数 ⟩ で指定された式の階層に含まれる全ての部分式を%t ラベルに割当て, ラベルを用いた式に ⟨ 式 ⟩ を変換します. 階層指定は part 函数と同様で, 表示された形式に対して指定を行います. pickapart 函数は大きな式を扱う際に,part 函数を使わずに部分式に変数を自動的に割当てる事にも使えます.

```
(%i49) exp:(x+1)^3;
                                        3
(%o49)                           (x + 1)
(%i50) pickapart(exp,1);
                                       3
(%o50)                           %t48
(%i51) exp2:expand((x+1)^3);
                             3      2
(%o51)                      x  + 3 x  + 3 x + 1
(%i52) pickapart(exp2,1);

                                       3
(%t52)                                 x



                                     2
(%t53)                             3 x



(%t54)                               3 x
```

```
(%o54)                        %t54 + %t53 + %t52 + 1
```

piece

piece()

part 函数を用いた時に選ばれた最後の式を保ちます. 函数の実行中に設定されて, その函数自身の中で参照する事が出来ます.

powers

powers (⟨ 式 ⟩,⟨ 変数 ⟩)

⟨ 式 ⟩ に現われる ⟨ 変数 ⟩ の次数リストを返します. 利用前に予め load(powers) で読込を行う必要があります.

product

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

⟨ 添字変数 ⟩ の ⟨ 下限 ⟩ から ⟨ 上限 ⟩ 迄の ⟨ 式 ⟩ の値の積を与えます. 評価は sum 函数と似ています. 積の簡易化はこの時点では使えません.

⟨ 上限 ⟩ が ⟨ 下限 ⟩ より小になると空の積となりますが, この場合,product はエラー出力ではなく 1 を返します.

rectform

rectform (⟨ 式 ⟩)

`a+b*%i` の形式で式を返します. ⟨ 式 ⟩ が複素数の場合,a と b は実数となりますが, そうでない場合には,`%i` を持たない部分と`%i` で括られた式に分解し, 内部の項順序に従って出力される為, 正確に `a+b*%i` の形式にはなりません.

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

remvalue

remvalue (⟨ 変数$_1$⟩,⟨ 変数$_2$⟩,$\cdots$)

remvalue (all)

指定した利用者変数の値をシステムから削除します. ここで, 利用者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.

sum

sum (⟨ 式 ⟩,⟨ 添字変数 ⟩,⟨ 下限 ⟩,⟨ 上限 ⟩)

⟨ 添字変数 ⟩ を ⟨ 下限 ⟩ から ⟨ 上限 ⟩ 迄の値の ⟨ 式 ⟩ の和を取ります. ⟨ 上限 ⟩ と ⟨ 下限 ⟩ が整数で異なっていれば, 和の各々の項は評価されて互いに加えられます.sumpsum が true の場合, その結果も簡易化されます.

簡易化は閉形式の生成が出来る場合があります.simpsum が false, 或いは, 'sum が用いられた場合, 値は sum の名詞型で, 数学の Σ 表記で表示されます. 尚,⟨ 上限 ⟩ が ⟨ 下限 ⟩ よりも小さい場合には, 所謂,"空の総和 (empty sum)"になり,sum はエラー出力ではなく 0 を返します.sum は微分, 和, 差や積が可能で, それらの自動的簡易化も行われます.

cauchysum が true であれば, 和同士の掛け算で通常の積ではなく Cauchy 積が利用されます.Cauchy 積では内部和の添字が独立に変化するのではなく, 外部添数の函数となります.

大域変数の genindex は和の次の変数を生成するのに利用されるアルファベットの前置詞です.

gensumnum は和の次の変数生成に用いられる数値的な修飾子です.false が設定されていれば, 添字は数値の修飾子を持たない genindex のみで構成されます.

大域変数の simpsum はデフォルトで false が設定されています. 以下に simpsum の値による違いを示します.

```
(c18) simpsum;
(d18)                                  false
(c19) sum(x^n,n,0,m);
                                    m
                                   ====
                                   \n
(d19)                               >x
                                   /
                                   ====
                                 n = 0
(c20) simpsum:true;
(d20)                                  true
(c21) sum(x^n,n,0,m);
                                   m + 1
                                  x      - 1
(d21)                             ----------
                                    x - 1
(c22) sum(x^n,n,0,inf);
is  abs(x) - 1  positive, negative, or zero?

negative;
                                      1
(d22)                                 -----
                                      1 - x
```

```
(c23)

(c24) sum(x^n,n,0,inf);
is  abs(x) - 1  positive, negative, or zero?

zero;
(d24)                              undefined
(c25) sum(x^n,n,0,inf);
is  abs(x) - 1  positive, negative, or zero?

positive;
(d25)                                inf
```

この様に simpsum が false の場合,sum(xn̂,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. 又,sum(xn̂,n,0,inf) とすると,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.

indices

indices(⟨ 式 ⟩)

二つの元のリストを返します. 最初のものは ⟨ 式 ⟩ で利用されていない添数のリスト (これらは一度だけ現われる) で, 二番目のものは ⟨ 式 ⟩ の無効添数のリスト (これらは丁度二度現れる) となります.

平成 17 年 12 月 4 日 (日)

第12章　函数定義

12.1　MAXIMA での函数の定義

MAXIMA では, 予めシステムが持っている函数の他に, 利用者定義による函数を扱えます. 定義函数では define 函数を用いる方法もありますが, 代表的な方法は:=演算子を用いる方法です.

:=演算子で函数を定義する場合, 演算子の左側に函数名と引数を記述し, 右側に処理の内容を記述します. 例えば, f(x):=sin(x) と入力すると, 一変数函数 f が定義され, 入力値を x とすれば, 出力は sin(x) となります. 又, 函数の定義を行うと, 大域変数 fuctions に定義した関数名が追加されます.

```
(%i1) functions;
(%o1)                                []
(%i2) f(x):=sin(x);
(%o2)                          f(x) := sin(x)
(%i3) f(10);
(%o3)                              sin(10)
(%i4) functions;
(%o4)                              [f(x)]
```

簡単な函数は,f(x):=(式$_1$, 式$_2$, $\cdots$, 式$_n$) の様に複数の式を並べて, 全体を小括弧で括ったものです. この函数は単純に最後の式の結果だけを返します. 例えば, f(x):=(1-x,2+x,2*x) で函数 f を定義した場合, この函数の結果は 2*x を計算したものになります.

但し, この函数では割当が実行されていない為に, 変数の内容の書換えは生じません. では, f(x):=(y:x,z:2+y,2*z) で定義した場合はどうなるでしょうか？

```
\(%i1)  f(x):=(y:x,z:2+y,2*z);
(%o1)                    f(x) := (y : x, z : 2 + y, 2 z)
(%i2)  f(x);
(%o2)                                2 (x + 2)
(%i3) y;
(%o3)                                   x
(%i4) z;
(%o4)                                 x + 2
(%i5)  f(2);
(%o5)                                   8
(%i6) x;
(%o6)                                   x
(%i7) y;
(%o7)                                   2
(%i8) z;
(%o8)                                   4
```

この様に, 先頭の式から順番に処理されて行き, 最後の式の結果だけが返却されています. この函数の場合, 内部で用いた変数 y と z の内容が書換えられている事に注意して下さい.

MAXIMA では, 函数内部だけで利用可能な変数, 即ち, 局所函数が使えます. この場合, 後述の block 文を用いて函数を定義します.

MAXIMA の lambda 式を用いると無名の函数が構築出来ます. この lambda 函数の構文は以下の様に最初に函数の変数を宣言し, その後に函数本体が続きます. 函数本体は基本的に MAXIMA の式をコンマで区切ったものになります.

lambda([⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩], ⟨ 函数本体 ⟩)

構文自体は,LISP の lambda 式と同様の構文となっています.

次の例では `lambda([i],2*i+1)` で引数 i に 1 を加える無名函数を map 函数でリスト [1,2,3] に作用させた結果と,lambda 式を利用した函数 neko を示しています.

```
(%i58)  map(lambda([i],2*i+1),[1,2,3]);
(%o58)                              [3, 5, 7]
(%i59)  neko(x):=map(lambda([i],sin(2*i+1)),x);
(%o59)           neko(x) := map(lambda([i], sin(2 i + 1)), x)
(%i60)  neko([1,2,3,4,5]);
(%o60)             [sin(3), sin(5), sin(7), sin(9), sin(11)]
(%i61) i;
(%o61)                                  i
```

MAXIMA では引数の個数が可変な函数も定義出来ます. この場合, 最後の引数を特別な引数リス

トとして割当てて函数を, 定義します. 尚, 引数が少ない場合, 安易な処理を行っていると少し問題になるかもしれません.

```
(%i41) f([u]):=u;
(%o41)                              f([u]) := u
(%i42) f(1,2,3,4,5);
(%o42)                            [1, 2, 3, 4, 5]
(%i43) f(a,b,[u]):=[a,b,u];
(%o43)                       f(a, b, [u]) := [a, b, u]
(%i44) f(1,2,3,4,5,6);
(%o44)                         [1, 2, [3, 4, 5, 6]]
(%i45) f(1,2);
(%o45)                              [1, 2, []]
```

函数本体に含まれる式からの返却値が必要であれば,block 文と return 文を組合せます.

この block 文の構造は,block 内部本体に変数リスト, 処理を行う式をコンマで区切った式の列が記述されています :

```
block($\langle 変　数　リ　ス　ト\rangle ,\langle 式_1\rangle ,\langle 式
_2\rangle ,\cdots,\langle 式_n\rangle $)
```

ここで変数リストは block 文内部で用いる局所変数のリストです. 局所変数を用いない場合に空リスト [] を表記しても, 局所変数リスト自体を省略しても構いません. 各局所変数に初期値を設定する事も可能で, 例えば, 局所変数として a,b,c を用い,a に初期値 2,c に空リストを設定する場合は [a:2,b,c:[]] とします. 因に, この局所変数への値の束縛は動的束縛と呼ばれるものになります.

次に,return 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.

定義した函数の内容は函数 dispfun や fundef で参照する事が出来ます.

```
(%i72) dispfun(neko);
(%t72)       neko(x) := block([a : 2, k], k : sin(x) a, return(k))

(%o72)                                done
(%i73) fundef(neko);
(%o73)       neko(x) := block([a : 2, k], k : sin(x) a, return(k))
```

dispfun と fundef は機能的には殆ど違いはありません. 但し,dispfun の場合は, 函数を%t ラベルに表示するのに対して,fundef は%o ラベルに出力する事です.

12.2 マクロの定義

MAXIMA ではマクロの定義も出来ます.MACRO の定義では演算子::=を用います. MAXIMA のマクロは最終的に LISP の S 式に展開し, それから評価が行われます.

複雑なマクロを生成する為に,block 文に似た buildq 函数を用います. 構文は, buildq(⟨ 変数リスト ⟩,⟨ 式 ⟩) となります.

buildq の引数は, 実際の式の代入が実行される迄,MAXIMA の解釈による変換を防ぐ必要があります. この為に単引用符' を用います. 函数 splice は buildq でリストの生成を行う為に用います.buildq との組合せで引数リストの作成が簡単になります,

```
mprint([x])::= buildq([u : x],
  if (debuglevel > 3) then print(splice(u)));
```

呼出しが以下であれば,

```
mprint("matrix is ",mat,"with length",length(mat))
```

これは次の行と同値です.

```
if debuglevel > 3
  then print("matrix is ",mat,"with length",
              length(mat))
```

次の非自明な例は変数の値とそれらの名前を表示するものです.

```
mshow(a,b,c)
```

は値をプログラムの中で行として表示する事が出来る様に

```
print('a,"=",a,",",'b,"=",b,", and",'c,"=",c)
```

となる.

```
(c101) foo(x,y,z):=mshow(x,y,z);
(c102) foo(1,2,3);
x = 1 , y = 2 , and z = 3
```

実際の mshow の定義は以下の通りである.buildq がどうやって'u が, その変数名を与える様に引用された構造を組み立てるかに注意して下さい. マクロに含まれる result は, そのマクロに対して代入と評価されるコードの一片である事に注意して下さい.

```
mshow ([l]) ::= block ([ans:[], n:length(l)],
    for i:1 thru n do
        (ans: append (ans, buildq ([u: l[i]], ['u, "=", u])),
        if i < n then
```

```
            ans: append (ans, if i < n-1 then [","] else [", and"])),
    buildq ([u:ans], print (splice(u))));
```

splice は代数的操作に引数を与えます.

```
(c108) buildq([a:'[b,c,d]],+splice(a));
(d108) d+c+b
```

代入後にのみ簡易化がどのよう行なわれるかを注意して下さい. 最初の場合では,splice に対して適応されるのは+で, 第二の場合は*です. そこで, 論理的に splice(a)+sploce(a) は 2*splice で置換えられそうですが, 実際は, どの様な簡易化も buildq では実行されません.

splice が代数で何を実行するかを理解する為には,a+b+c の様な式が MAXIMA 内部では前置式表現, 即ち, 演算子を前に配置したリスト (+ a b c) の様な表現となっており, これは積についても同様な事を理解している必要があります.

```
(c114) buildq([a:'[b,c,d]],+splice(a));
(d114) d+c+b


(c111) buildq([a:'[b,c,d]],splice(a)+splice(a));
(d111) 2*d+2*c+2*b
```

しかし,

```
(c112) buildq([a:'[b,c,d]],2*splice(a));
(d112) 2*b*c*d
```

最後に,buildq は再帰的な函数構築で計り知れない価値があります. ここで, ある微分方程式を級数を用いた方法で解くプログラムがあり, その方程式で次の再帰的な関係を構築する必要があったとします.

```
f[n]:=(-((n^2-2*n+1)*f[n-1]+f[n-2]+f[n-3])/(n^2-n))
```

そして, プログラムの内部で直接これを実行しなければならないとする.

ここで,expand を実際に付け足したければ,

```
f[n]:=expand((-((n^2-2*n+1)*f[n-1]+f[n-2]+f[n-3])
  /(n^2-n)));
```

としますが, このプログラムをどの様にして構築すれは良いのしょうか. 先ず expand が動作するのが, その函数が動作する各時点で, その以前でない事を希望すれば, 下記の様に記述すると希望通りの処理を行います.

```
kill(f),
val:(-((n^2-2*n+1)*f[n-1]+f[n-2]+f[n-3])/(n^2-n)),
define(f[n],buildq([u:val],expand(u))),
```

これは便利です. 実際,expand 付きで実行すると, 次の結果を得ます.

```
(c28) f[6];
(d28) -aa1/8-13*aa0/180
```

ところが, 簡易化しないままだと第 6 項でさえもこうなってしまうでしょう.

```
(c25) f[6];
(d25) (5*(-4*(-3*(-2*(aa1+aa0)+aa1+aa0)/2
            -(aa1+aa0)/2+aa1)
        /3
        -(-2*(aa1+aa0)+aa1+aa0)/6+(-aa1-aa0)/2)
      /4
      +(-3*(-2*(aa1+aa0)+aa1+aa0)/2
       -(aa1+aa0)/2+aa1)
       /12-(2*(aa1+aa0)-aa1-aa0)/6)
      /30
```

この様に各段階で簡易化を実行しなければ, 途端に即座に複雑になってしまいます. そこで, 簡易化が定義の一部でなければなりませんね. この様に buildq はその様な定義を行うのに便利です.

上記の f を決める為には勿論 f[0],f[1],f[2] が必要になります. ここでは上記の例に沿って MAXIMA に式を入れて行きます. ここで, 尚,f[0]:0,f[1]:aa0,f[2]:aa1 としましょう.

```
(c1) val:(-((n^2-2*n+1)*f[n-1]+f[n-2]+f[n-3])/(n^2-n));
                       2
                  - (n  - 2 n + 1) f      - f      - f
                                    n - 1    n - 2    n - 3
(d1)              ----------------------------------------
                                   2
                                  n  - n
(c2) define(f[n],buildq([u:val],expand(u)));
                            2
                       - (n  - 2 n + 1) f      - f      - f
                                         n - 1    n - 2    n - 3
(d2)        f  := expand(----------------------------------------)
             n                            2
                                         n  - n
(c3) f[0]:0;f[1]:aa0;f[2]:aa1;
(d3)                                  0
(c4)
(d4)                                 aa0
(c5)
```

```
(d5)                                  aa1
(c6) f[6];
                                  3 aa1   aa0
(d6)                              ----- + ---
                                   10     40
```

この様に, 簡易化された答が返却されます. それに対し,define 内部の buildq で expand を抜いた場合を確認してみましょう.

```
(c1)  val:(-((n^2-2*n+1)*f[n-1]+f[n-2]+f[n-3])/(n^2-n));
                      2
                  - (n  - 2 n + 1) f      - f      - f
                                    n - 1    n - 2    n - 3
(d1)              ----------------------------------------
                                   2
                                  n  - n
(c2) define(f[n],buildq([u:val],u));
                         2
                     - (n  - 2 n + 1) f      - f      - f
                                       n - 1    n - 2    n - 3
(d2)           f  := ----------------------------------------
                n                      2
                                      n  - n
(c3) f[0]:0;f[1]:aa0;f[2]:aa1;
(d3)                                  0
(c4)
(d4)                                 aa0
(c5)
(d5)                                 aa1
(c6) f[6];
                             3 (- 4 aa1 - aa0)
                 4 (- aa1 - ----------------- - aa0)
                                   2                  - 4 aa1 - aa0
       5 (- aa1 - ---------------------------------- - ------------)
                                   3                        6
(d6) (- ------------------------------------------------------------
                                   4

                                   3 (- 4 aa1 - aa0)
                          - aa1 - ----------------- - aa0
```

```
                                2                 - 4 aa1 - aa0
                        - ------------------------------ - -------------)/30
                                12                        6
```

この様に展開されない為に複雑なままです.

12.3　最適化

MAXIMA は LISP で記述されています. その為, 全てのデータは MAXIMA の内部表現を用いており,LISP がこの内部表現を解釈しています. その為, 函数を LISP の形式に変換してしまえば, 内部表現の解釈の手間が省ける為に, 速度の向上が見込めます.translate 函数は,MAXIMA の利用者定義函数を LISP の函数に変換する函数です. 更に,compile を用いると LISP のコンパイル機能を用いる為に一層の速度向上が見込めます.

12.4 函数定義に関連する大域変数

compgrind
デフォルト値:[false]
true であれば,compile による函数定義の出力が整形表示されます.

functions
デフォルト値:[]
利用者が定義した全ての函数名を含むリストです.

macroexpansion
デフォルト値:[false]
マクロの効率性に影響を与える機能を制御します.

- false を設定すると, マクロが呼出されるたびに, マクロの展開を行います
- expand を設定すると, 最初の呼出で評価されると, 後で呼出す時には展開をしなくても良い様に, この展開が内部的に記憶され, その後の呼出を早くします. マクロの呼出は grind と display を通常実行しますが, 全ての展開を覚えておくには, 一層のメモリが必要となります.
- displace を設定すると, 最初に或る特定の呼出が評価され, マクロ展開は呼出に対して代入されます.expand に設定された時よりも, 幾らか少ない保存領域を必要とする割には, 処理速度は同程度で, その上, マクロの呼出が記憶されない欠点を持っています. 展開は display か grind が呼出されていれば参照出来ます.

mode_checkp
デフォルト値:[true]
true の場合,mode_declare は定値変数のモードを検査します.

mode_check_errorp
デフォルト値:[false]
true であれば,mode_declare はエラーを呼びます.

mode_check_warnp
デフォルト値:[true]
true の場合, モードエラーが記載されます.

savedef
デフォルト値:[true]
true であれば, 利用者函数を translate 函数で変換しても, 元の maxima のプログラムを残します. その為, 大域変数 functions に割当てられた利用者函数名リストから, 変換した函数名を削除せずに残します. 又,dispfun 函数で函数の定義が表示可能で, 函数の編集も出来ます.
false の場合は,functions に割当てた利用者函数名リストから該当函数名を削除します.

transcompile

デフォルト値:[false]

true であれば,translate 函数は可能な compile 命令に必要な宣言を生成します. compile 命令は transcompile:true を用います.

translate

デフォルト値:[false]

true であれば, 利用者定義函数が自動的に LISP 函数に変換されます. 尚,MAXIMA と LISP の整合性の問題から, 変換される前と同じ動作をするとは限らない事に注意して下さい.

変数が mode_declare された CRE 表現の場合,rat 函数を一つ以上の引数で用いたり,ratvars 函数を使ってはなりません. 又,prederror:false は変換しません.

transrun

デフォルト値:[true]

false であれば,translate 函数で変換されたものではなく, 元の maxima の函数 (それらが存在していれば) が実行されます

tr_array_as_ref

デフォルト値: [true]

true であれば, 実行コードは配列として変数の値を用います.

tr_bound_function_applyp

デフォルト値:[true]

函数として用いようとする変数に値が設定されていれば警告を出します.

tr_filek_tty_messagesp

デフォルト値:[false]

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 ttu に送るかどうかを決めます.

false(デフォルト値) 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

デフォルト値:[true]

逆三角函数が複素数値を返しても良いかどうかを宣言します. 逆三角函数は sqrt,log, acos 等です. 例えば,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_function_call_default

デフォルト値:[general]

false の場合, 中断して meval を呼出す事を意味します.

expr の場合は, 引数固定の LISP 函数を仮定する事を意味します.

general の場合,mexprs と mlexprs に対しては利用者定義函数を良くしますが, macros に対してはそうではありません. 尚, 変数束縛が compile 函数でコンパイルされた MAXIMA の利用者定義函数の中で正しい事を保証し, 函数 f(x) を変換する時に,f が束縛変数であれば,apply(f,[x]) を意味していると仮定し, 適切な警告と共にその様に変換します. これを無効にする必要性はありません.

デフォルト設定で, 何等の警告メッセージが無ければ,translate 函数で変換されたり,compile でコンパイルされた利用者定義函数には元の maxima 函数と完全な互換性がある事を意味します.

tr_gen_tags

デフォルト値:[false]

true であれば,translate_file はテキストエディタで用いる tags ファイルを生成します.

tr_numer

デフォルト値:[false]

true であれば, 数の属性はそれらによって与えられたアトムに対して用いられます. 例えば,%pi.

tr_optimize_max_loop

デフォルト値 [100]

考えられる形式で translate 函数でのマクロ展開と最適化工程ループの最大回数を定めます. これは macro 展開エラーを捉える為で, 非中断の最適化属性です.

tr_semicompile

デフォルト値:[false]

true であれば,translate_file 函数と compile 函数の出力形式は拡張されたマクロになりますが,LISP コンパイラで機械コードに翻訳されたものではありません.

tr_state_vars

デフォルト値:

```
[transcompile, tr\_{}semicompile,
tr\_{}warn\_{}undeclared, tr\_{}warn\_{}meval,
tr\_{}warn\_{}fexpr, tr\_{}warn\_{}mode,
tr\_{}warn\_{}undefined\_{}variable,
tr\_{}function\_{}call\_{}default,
 tr\_{}array\_{}as\_{}ref,tr\_{}numer]
```

変換された出力形式に影響を与える大域変数のリストです. この情報は, 変換の虫取りを行う時に便利です. 変換されたものと与えられた状況で生成されなければならない物を比較する事で, 虫を追跡する事が可能になります.

tr_warn_bad_function_calls

デフォルト値:[true]

変換時に不適切な宣言が行われた為, 函数の呼出しが生じた場合に警告します.

tr_warn_fexpr

デフォルト値:[compfile]

任意の fexpr が与えられていれば警告します.fexpr 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

デフォルト値:[compfile]

函数 meval が呼び出されると警告します. もし,meval が呼出されると, 変換の問題点を指定します.

tr_warn_mode

デフォルト値:[all]

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

デフォルト値:[compile]

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

デフォルト値: [all]

未宣言の大域変数があれば警告します.

tr_windy

デフォルト値:[true]

助けになる註釈とプログラムのヒントを生成します.

undeclaredwarn

デフォルト値:[compfile]

四種類の設定項目があります.

設定	動作
false	警告を表示しません
compfile	compfile であれば警告します
translate	translate や translate:true であれば警告します
all	compfile や translate であれば警告します

mode_declare(⟨ 変数 ⟩,any) を実行して ⟨ 変数 ⟩ が一般の MAXIMA の変数である事を宣言します. 即ち,float, 又は fixnum である事に限定されません.compile 函数でコンパイルされる利用者定義函数中の変数を宣言する特別な動作は全て無効にしなければなりません.

12.5 函数定義に関連する函数

apply

apply (⟨ 函数 ⟩, ⟨ リスト ⟩)

⟨ 函数 ⟩f を ⟨ リスト ⟩ に適用した結果を与えます.⟨ 函数 ⟩ を適用する前に, その引数を計算したい時に便利です.

例えば,apply(min,[1,5,-10.2,4,3]) は-10.2 となります.

函数の呼出しで, それらの引数が評価されておらず, それらの評価を希望する場合も apply は便利です. 例えば,filespec がリスト [test,case] であれば, apply(closefile,filespec) は closefile(test,case) と同値です. 一般的に,apply で評価させる為には, 単引用符' を最初の引数の先頭に置いて, 名詞型として apply に評価させるべきです. 幾つかのアトムの変数が, とある函数と同じ名前を持っていれば, 函数としてではなく, その変数値が利用される事になりす. 何故なら,apply それ自身が第二の引数と同様に第一の引数をも評価するからです.

```
(c17) apply('expand,[(x+1)^2]);
                                 2
(d17)                           x  + 2 x + 1
(c18) apply('apply,['expand,[(x+1)^2]]);
                                 2
(d18)                           x  + 2 x + 1
(c19)
```

この様にする理由は, 以下の馬鹿馬鹿しい例から明確になるでしょう.

```
(c30) neko(x):=2*x;
(d30)                          neko(x) := 2 x
(c31) neko:-128;
(d31)                               - 128
(c32) neko(10);
(d32)                                20
(c33) apply('neko,[20]);
(d33)                                40
(c34) appply(neko,[20]);
(d34)                         appply(- 128, [20])
```

この例では,neko を函数として定義していますが, 同時に, 変数としての neko には-128 が束縛されています. その為,apply では単引用符' がなければ変数として評価されてしまいます.

bindtest

bindtest (⟨ 変数 ⟩)

⟨ 変数 ⟩ が無束縛のままであれば, エラーを返します.

block

block ([⟨ 局所変数$_1$⟩, $\cdots$, ⟨ 局所変数$_k$⟩], ⟨ 文$_1$⟩, $\cdots$, ⟨ 文$_j$⟩)

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. 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ても構いません. 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為に用いるものです. 同名の (大域) 変数が存在した場合, block 文の実行中, その変数はスタックに保存されるので, その間は参照出来ません.block 文が終了した時点で, スタックに保存さていた値はもとに戻され, block 文の局所変数の値は失なわれてしまいます.

尚,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 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います. 尚, 文のラベル付けは,block でアトムを文の前に置く事で対処します. 例えば,block([x],x:1,loop,x:x+1,$\cdots$,go(loop)) の様にします. go の引数は 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ない.block は通常, 函数定義の右側に現われるが, 他の場所に置く事も同様に可能です.

break

break (⟨ 引数 ⟩, $\cdots$) ⟨ 引数 ⟩ の評価と表示を行い,(maxima-break) にて利用者がその環境を調べ, 変更する事が出来る様にします.(maxma-break) からは exit; を入力すれば計算が再開されます. 尚,Cntrol-a(â) で maxima-break に何時もで対話的に入る事が出来ます.Cntrl-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.

buildq

マクロの定義で用います.

catch

catch (⟨ 式$_1$⟩, $\cdots$, ⟨ 式$_n$⟩)

throw と対で用います.

この非局所的回帰 (non-local return) は, 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. ここで,⟨ 式$_i$⟩ の評価が何らの throw の評価に至らなかった場合,catch の値は最後の引数 ⟨ 式$_n$⟩ の値となります.

```
(c1) g(l):=catch(map(lambda([x],
       if x<0 then throw(x) else f(x)),l));
(c2) g([1,2,3,7]);
(d2)                     [f(1), f(2), f(3), f(7)]
(c3) g([1,2,-3,7]);
(d3)                                - 3
```

函数 g は,l が非負の数のみであれば l の各要素に対する f のリストを返します. それ以外では,g は l の最初の負の要素を捉え (catch) て, それを放擲 (throw) します.

compfile

compfile (⟨ ファイル ⟩,⟨ 函数$_1$⟩,···,⟨ 函数$_n$⟩)

MAXIMA の指定した函数の ⟨ 函数$_1$⟩,···,⟨ 函数$_n$⟩ を LISP の函数に変換し, ⟨ ファイル ⟩ に書込みます.

```
(%i28) neko(x):=sin(x);
(%o28)                          neko(x) := sin(x)
(%i29) compfile("mike",neko);

Translating neko
(%o29)                          /home/yokota/mike


(PROGN (DEFPROP $NEKO T TRANSLATED) (ADD2LNC '$NEKO $PROPS)
 (DEFMTRFUN ($NEKO $ANY MDEFINE NIL NIL) ($X) (DECLARE (SPECIAL $X))
  (SIMPLIFY (LIST '(%SIN) $X))))
```

compile

compile (⟨ 函数$_1$⟩,···,⟨ 函数$_n$⟩) compile (functions) compile (all)

指定した MAXIMA の処理言語で記述した函数を LISP の函数に変換し, それを LISP の函数 COMPILE を用いてコンパイルします. 尚,compile 函数は函数名リストを返します.

引数に functions や all を指定すると利用者定義函数を全てコンパイルします.

尚, 大域変数 functions には利用者定義函数の名前がリストの成分として保持されています.

define

define (⟨ 函数 ⟩(⟨ 引数$_1$⟩,···,⟨ 引数$_n$⟩),⟨ 本体 ⟩)

函数を定義します. この函数は引数として ⟨ 引数$_1$⟩,···,⟨ 引数$_n$⟩ を持ち, 函数内部の文は ⟨ 本体 ⟩ に記述します. この函数定義は ⟨ 函数 ⟩(⟨ 引数$_1$⟩,···):= "⟨ 本体 ⟩ と同値です.

define_variable

define_variable (⟨ 変数名 ⟩,⟨ デフォルト値 ⟩,⟨ 型 ⟩,⟨ オプションの文書 ⟩)

MAXIMA 環境に大域変数を導入する函数です. 主にパッケージで利用する環境変数の設定等で用いられます.

⟨ デフォルト値 ⟩ は定義した大域変数の初期値となりますが, この値は次の ⟨ 型 ⟩ に適合するものでなければなりません.

⟨ 型 ⟩ には boolean,fixnum(16 ビット整数),number(多倍長整数), rational(有理数) と float(浮動小数点), 或いは, これらの内のいずれかを意味する any を設定します.

⟨ オプションの文書 ⟩ は省略可能な引数で, 文字列を設定します. これは,translate_file が文書文字列を含むパッケージに用いられた時, マニュアル, 用例ファイル, 例えば,describe 向けに適した書式の文書文字列を含む lisp のファイル出力となります.

any 以外のモードで define_variable とされた任意の変数で value_check 属性を与える事が可能です. この属性は, 利用者がその変数に設定しようとする値に関して呼出された 1 引数の函数となります.

ここで, 単純な例で define_variable の動作を説明しましょう. 先ず, define_variable(foo,true,boolean); と入力すると, 以下の処理が逐次実行されます.

1. mode_declare(foo,boolean) を実行し, 変数 foo が Boolean である事を宣言します.
2. 変数 foo に変数に値が束縛されていなければ, foo:true を実行し, 変数 foo に true を割当てます.
3. declare(foo,special) を実行し, 変数 foo が special であると宣言します.
4. 間違った変数型の値を設定する事が決してない様に, 属性を割当てます. ここでは,foo を boolean として定義しているので, foo:44 を実行すると, その結果はエラーになります.

dispfun

dispfun (⟨ 函数名$_1$⟩, $\cdots$, ⟨ 函数名$_n$⟩) dispfun(all)

利用者定義の函数である ⟨ 函数名$_1$⟩, $\cdots$, ⟨ 函数名$_n$⟩) を表示します. この函数の表示では, 函数を定義した時点での函数や定数等をそのまま表示します.

引数に all を設定すると, 大域変数 functions と arrays で与えられる函数を全て表示します.

fundef

fundef(⟨ 関数名 ⟩)

⟨ 関数名 ⟩) に対応する函数の定義を返します. fundef は dispfun に似ていますが,fundef では display 函数を呼出さない点で異なります.

```
(%i9) neko(x):=sin(x)*exp(x);
(%o9)                    neko(x) := sin(x) exp(x)
(%i10) dispfun(neko);
(%t10)                   neko(x) := sin(x) exp(x)

(%o10)                                done
(%i11) fundef(neko);
(%o11)                   neko(x) := sin(x) exp(x)
```

この例で示す様に,dispfun を実行すると結果は%t ラベルに表示されていますが,fundef の方は通常の%o ラベルに表示されています.

funmake

funmake (⟨ 関数 ⟩, [⟨ 引数$_1$⟩, ··· , ⟨ 引数$_n$⟩])

⟨ 関数 ⟩ で指定した函数を呼出して評価を行う事はしません. 単純に,⟨ 関数 > (< 引数$_1$⟩, ··· , ⟨ 引数$_n$⟩) を返すだけです.

```
(%i2)  funmake(f,[x,y,z]);
(%o2)                              f(x, y, z)
(%i3)  funmake(neko,[x,y,z]);
(%o3)                            neko(x, y, z)
(%i4) funmake(expand,[128,"うちのタマ知りませんか？"]);
(%o4)                       expand(128, うちのタマ知りませんか？)
(%i5) funmake(a,[1,2,3]);
(%o5)                              a(1, 2, 3)
(%i6)  a:10;
(%o6)                                 10
(%i7)  funmake(a,[1,2,3]);
Bad first argument to ‘funmake’: 10
 -- an error.  Quitting.  To debug this try debugmode(true);
(%i8)  funmake('a,[1,2,3]);
(%o8)                              a(1, 2, 3)
```

関数に指定したアトムに値が束縛されている場合, そのアトムは内部で評価される為, エラーになります. この場合は, 単引用符' を先頭に付けて名詞型とすれば問題ありません.

local

local (⟨ 局所変数$_1$⟩, ··· , ⟨ 局所変数$_n$⟩)

この函数の函数が利用される文中で,⟨ 局所変数$_1$⟩, ··· , ⟨ 局所変数$_n$⟩ を全ての属性に対して局所的なものにします.local は block 文, 函数定義の本体,lambda 式, 又は ev 函数でのみ一度だけ使えます. この local 函数は文脈からも独立しています.

mode_declare

mode_declare(⟨ 変数$_1$⟩, ⟨ 変数型$_1$⟩, ··· , ⟨ 変数$_n$⟩, ⟨ 変数型$_n$⟩)

modedeclare と同義です. この mode_declare は translate 函数や compile 函数で LISP 函数に変換したりコンパイルする利用者定義の函数内部で局所変数の宣言の後で, これらの変数型宣言で用います.

引数は ⟨ 変数$_i$⟩ とその変数に対応する ⟨ 変数型$_i$⟩ の組で, 変数型は boolean,fixnum(機械長整数),number(整数),rational(有理数) や float(浮動小数点), 或いは, これらの何れからの値が取れる any の何れか一つを指定します.

又,⟨ 変数$_i$⟩ が配列で, 参照される配列の全ての要素が値を持っているとすると, 配列の範囲を最初に宣言する場合は, array(⟨y_i⟩, ⟨ 次元$_1$⟩, ⟨ 次元$_2$⟩, ···) ではなく, array(⟨y_i⟩, *complete*, ⟨ 次元$_1$⟩, ⟨ 次元$_2$⟩, ···)

を用いる必要があります.

ここで,配列の全ての要素が fixnum(float) の場合,complete の代りに fixnum(float) を用います. 配列の全ての要素が同じ型の数値の場合, 例えば m であれば, mode_declare(completearray(yi),m)) を効率的な変換の為に使わなければなりません. 更に, 配列を用いた数値は予想される配列の大きさを宣言する事でより速く動作させる事が可能です.

mode_declare(completearray(a[10,10]),float) は,10x10 の浮動小数点の配列向けです. 加えて, 函数結果を function(f1,f2,...) を引数として宣言しても構いません. 尚,f1,f2,... は函数名です.

mode_declare([function(f1,f2,...),x], fixnum,q,completearray(q),float) で,x と f1,f2,... によって返される値が機械長の整数である事と,q が浮動点小数配列である事を宣言しています.

mode_identity

mode_identity (⟨ 引数$_1$⟩,⟨ 引数$_2$⟩)

mode_declare と用いられる特殊な形式で, 宣言すべきマクロ, 例えば,flonum 型のリストのリストや, 他のデータの混合物になります.

⟨引数$_1$⟩はプリミティブの値の型名で, mode_declare に与えられるものです. 即ち,[float,fixnum,number,list,any] の何れかになります.⟨ 引数$_2$⟩ は式で, 評価されて,mode_identity の値として返されます. しかし, 返された値が ⟨ 引数$_1$⟩ で宣言されたモードに合致しないものであれば, エラーや警告が出力されます.

重要な事は,maxima から lisp への変換で決められた式の型が ⟨ 引数$_1$⟩ として与えられ,⟨ 引数$_2$⟩ に続く全てのものから独立している事です. その為, x:3.3; mode_identity(fixnum,x); と入力すると, 警告が出て,mode_identity(flonum,x) は 3.3 を返します.

これは色々な使い方があり, 例えば,first(l) が数 (number) を返す事が判っていれば,mode_identity(number,first(l)) と書いても良いでしょう. しかし, より効率的な方法は, 新しいプリミティブ,

```
firstnumb(x)::=buildq([x],mode\_{}identity(number,x));
```

を定義して, 数のリストの最初の元を取る時は何時も firstnumb を用いる事です.

translate

translate (⟨ 函数$_1$⟩,$\cdots$,⟨ 函数$_n$⟩) translate (functions) translate (all)

translate は MAXIMA の処理言語で記述した利用者定義の函数を LISP の函数に変換する函数です.MAXIMA 言語で記述された函数は, 裏の LISP で解釈されて, 実行されます. これを LISP の函数にすれば, 解釈する手間は省ける分, 処理の高速化が望めます.

引数は,⟨ 函数$_1$⟩,$\cdots$,⟨ 函数$_n$⟩ の様に利用者定義函数を直接指定する方法, に加えて, 引数に all や functions を指定して, 利用者定義函数を一度に変換する事も出来ます. 尚, 大域変数 functions は利用者定義函数の名前が全て登録されるリストです,

変換される函数には, 内部で局所変数を利用する場合には特に,mode_declare で局所変数の型を宣言する必要があります. これは, より効率的な LISP 函数を生成する為には必要な事です.

例えば,block 文を用いた函数を定義する場合, 局所変数を宣言した直後に, mode_declare を入れて, 局所変数の型を宣言します.

```
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,...),\langle 函数本体>)
```

尚, 函数を translate で変換すると, 大域変数 savedef が false の場合, 変換された函数の名前は大域変数 functions に割当てられた函数名リストから削除されて, 今度は大域変数 props に割当てられたリストに名前が追加されます.

当然の事ですが, 函数は虫取りが完遂されるまで変換すべきではありません. 更に,translate 函数は, 変換する函数内部の式はきちんと簡易化されていると仮定しています. そうでなければ, 最適化されていない LISP 函数が生成され, 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 に設定して変換されるべき式の簡易化を禁じる事をしてはいけません.

注意すべき事に,translate 函数を用いて LISP の函数に変換しても,MAXIMA と LISP の整合性の問題から, 以前と同じ動作をする保証はありません.

translate_file

translate_file (⟨ ファイル ⟩) translate_file (⟨ ファイル ⟩, ⟨*LISP* ファイル ⟩)

MAXIMA 言語で記述したプログラムを含むファイルを LISP 函数のファイルに変換します.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変換器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る. 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多く, 名前も長いものが多い為, apropos(tr_) を実行すると,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のも良いでしょう.

translate_file("foo.mc"), loadfile("foo.lisp") は batch("foo.mc") にある制限 (例えば," や

tr_warnings_get

tr_warnings_get ()

変換中に変換器が出力する警告のリストを表示します.

compile_file

compile_file (⟨ ファイル ⟩)

compile_file (⟨ ファイル ⟩, ⟨ コンパイルされたファイル ⟩)

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 ⟩)

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, これを LISP 函数に変換し, その結果を compile 函数でコンパイルします. 変換とコンパイルに成功すると, 今度は結果を MAXIMA に読込みます.

compile_file は四個のファイル名のリストを返します. このリストに含まれるファイル名は, 元の MAXIMA プログラムファイル, LISP への変換ファイル, 変換に関する註釈ファイルと compile で

コンパイルされたプログラムのファイルです. 尚, コンパイルに失敗すると, 返却されるリストの第四成分は false になります.

declare_translated

declare_translated (⟨ 函数$_1$⟩, $\cdots$, ⟨ 函数$_n$⟩)

MAXIMA のプログラムファイルを LISP に変換する際に, そのファイル中のどの函数が translate 函数で変換された函数, 或いは compile 函数でコンパイルされた函数として呼出されるべきか, そして, どれが MAXIMA の函数で, 又, 未定義のものであるかを知る事は translate 函数にとって重要な事です.

ファイルの先頭にこの宣言を置くと, ある記号がたとえ LISP 函数の値を持っていなかったとしても, 呼出された時にそれを持つ事を教えます. (mfunction-call fn arg1 arg2..) が生成されるのは,⟨ 函数$_n$⟩ が LISP 函数に変換されるべきものであるかを translate 函数が知らない時です.

第13章　プログラム

13.1　Maximaでのプログラム

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, 原始的な構文もあります. 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 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理です.

13.2　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 ⟨ 条件 ⟩ then ⟨ 式$_1$⟩ else ⟨ 式$_2$⟩

if 文は ⟨ 条件 ⟩ が true でならば ⟨ 式$_1$⟩,false ならば ⟨ 式$_2$⟩ を実行します. ⟨ 式$_1$⟩ と ⟨ 式$_2$⟩ は任意の Maxima の式 (勿論,if 文の入れ子も含んでいても構いません) で,⟨ 条件 ⟩ は true か false であるかが評価出来る式や関係と論理的演算子で構成されたものです.

if 文で利用可能な論理演算子を以下の表 13.1 に示しておきます.

表 13.1: if 文で利用可能な論理演算子

演算子	記号	種類
大きい	>	中置式表現演算子 (infix)
等しい	= equal	中置式表現演算子 (infix)
等しくない	#	中置式表現演算子 (infix)
小さい	<	中置式表現演算子 (infix)
以上	>=	中置式表現演算子 (infix)
以下	<=	中置式表現演算子 (infix)
and	and	中置式表現演算子 (infix)
または	or	中置式表現演算子 (infix)
否定	not	前置式表現演算子 (prefix)

前置式演算子 (prefix) と中置子演算子 (infix) は, 演算子を引数の置き方で区分したもです. 詳細は演算子の章を参照して下さい.

13.3　do 文による反復処理

do 文は反復処理で利用します.
この do 文には三種類の変種があり, 各々は終了条件が異なります.

1. for ⟨ 変数 ⟩ : ⟨ 初期値 ⟩ step ⟨ 増分 ⟩
   thru ⟨ 境界値 ⟩ do ⟨ 本体 ⟩
2. for ⟨ 変数 ⟩ : ⟨ 初期値 ⟩ step ⟨ 増分 ⟩
   while ⟨ 終了条件 ⟩ do ⟨ 本体 ⟩
3. for ⟨ 変数 ⟩ : ⟨ 初期値 ⟩ step ⟨ 増分 ⟩
   unless ⟨ 終了条件 ⟩ do ⟨ 本体 ⟩

(step は終了条件や境界の後に置いても構いません.)

do 文の実行は最初に初期値を ⟨ 変数 ⟩ に割当てます. この ⟨ 変数 ⟩ の事を制御変数と呼びます. この制御変数は局所的なもので,do 文の中だけで効力を持ちます.
do 文は初期値を制御変数に割当てると, 以下の手順で反復処理を行います.

1. do 文が終了するのは以下の場合です.
   (a) 制御変数が thru によって指定された境界値を越えた場合.
   (b) unless 条件が true になった場合.
   (c) while 条件が false になった場合.
2. 本体が評価されます.
3. 増分が制御変数に加えられます.

⟨ 終了条件 ⟩ が満される迄,(1) から (3) までの処理が繰返し実行されます. ⟨ 終了条件 ⟩ は幾つも与えても良く, その場合, それらの内のどれか一つが条件を満した時点で do 文が終了します.
do 文の ⟨ 初期値 ⟩,⟨ 増分 ⟩,⟨ 境界値 ⟩ は, 終了条件さえ thru,unless,while で判別出来るのであれば, どの様な式でも構いません. 更に,⟨ 増分 ⟩ が 1 の場合は,step 1 を省略しても構いません. 又,⟨ 本体 ⟩ にも特に制約はありません.
その為, 以下の例は全て同じ結果となります.

1.
```
for i:1 step 2 thru 10 do
      print("sin(%pi/6*",i,")=",float(sin(i*%pi/6)));
```

2. `for i:1 step 2 while i<10 do`
   `print("sin(%pi/6*",i,")=",float(sin(i*%pi/6)));`

3. `for i:1 step 2 unless i>=10 do`
   `print("sin(%pi/6*",i,")=",float(sin(i*%pi/6)));`

⟨境界値⟩,⟨増分⟩と⟨終了条件⟩は do 文の各時点で評価される事に注意して下さい. これらの処理を行うだけで膨大な計算を引き起すものや, 結果が⟨本体⟩の実行中に変化しないものならば,do 文に対してより効果的な値を制御変数に設定して,do 文で用いるのがより効率的です.

通常の do 文によって返される値はアトムの done です. 函数 return を用いると, 本体の中で do から早目に抜けて必要な値を与える事に使えます. block にある do 文中の return は do 文を出るだけで,block 全体から出る訳ではありません. 同様に go 函数も block 中の do 文から抜ける為に使ってはなりません.

13.3.1 do 文の追加形式

制御変数に対して何時も加える量の代りに, 各々の反復で別の手法に代えたい事もあるかもしれません. この場合,next 式を step の代りに使えます. これで制御変数にはループを通して各時点で式を評価した値が設定されます.

```
(c1)  for count:2 next 3*count thru 20
         do display(count)$
                     count = 2
                     count = 6
                     count = 18
```

for⟨変数⟩:⟨値⟩ ··· *do* ···構文の代りに, for⟨変数⟩*from*⟨値⟩ ··· *do* ···が使える. from⟨値⟩を step や next の値や終了条件の後に置く事も出来ます. from⟨値⟩が省略されると 1 が初期値として用いられます.

時には反復処理の実行で, 制御変数が実際に使われていないものに興味を持つかもしれません. そこで, 終了条件のみに情報の初期化と更新を, 次の貧弱な初期推定値を用いて 5 の平方根を計算する次の例の様に省略する事も出来ます.

```
(c1) x:1000
(c2)  thru 10 while x#0.0 do x:.5*(x+5.0/x)$
(c3) x;
(d3)                2.236068
```

終了条件でさえもすっかり省略し, 不定値のまま本体を評価し続ける do 本体を与えても構いません. この場合,return 函数を do 文の実行を中断する為に使う必要があります.

```
(c1) newton(f,guess):=
```

```
  block([numer,y],
        local(df),
        numer:true,
        define(df(x),diff(f(x),x)),
        do (y:df(guess),
            if y=0.0 then error("derivative at:",guess," is zero."),
            guess:guess-f(guess)/y,
            if abs(f(guess))<5.0e-6 then return(guess)))$
(c2) sqr(x):=x^2-5.0$
(c3) newton(sqr,1000);
(d3)                    2.236068
```

return が実行された時,guess の現行の値が do 文の値として返される事に注意して下さい.block から抜けると do 文の値は block の値として返されます. 何故なら, do は block の中で最後の文だからです.

do 文のもう一方の形式が Maxima で利用出来ます.

構文は

for ⟨ 変数 ⟩ in ⟨ リスト ⟩ [⟨ 終了条件判定 ⟩] do ⟨ 本体 ⟩

リストの成分は任意の式であり, 本体の各反復では変数に続けて割り当てられます. オプションの ⟨ 終了条件判定 ⟩ は do 文の実行を終了させる為に用いる式で, この ⟨ 終了条件判定 ⟩ に当て嵌まらない場合は,⟨ リスト ⟩ を消耗した場合, ⟨ 本体 ⟩ で return が実行された場合に終了します.

尚,⟨ リスト ⟩ は任意のアトムでない式や列でも構いません.

```
(%i8) for f in [sin,cos,tan] do print(f(1),"=",float(f(1)));
sin(1) = .8414709848078965
cos(1) = .5403023058681398
tan(1) = 1.557407724654902
(%o8)                                done
```

13.4 関連する大域変数

backtrace

デフォルト値:[]

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

dispflag

デフォルト値:[true]

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

デフォルト値:[true]

true であれば,if 文や is 函数で,true か false であるか述語の評価に失敗すると, 何時でもエラーメッセージが表示されます.

false であれば,unknown が代りに返されます.

errorfun

デフォルト値 [false]

引数を持たない函数名が設定されていれば, エラー発生時に, その函数が実行されます. この設定は batch ファイルで, エラーが生じた場合に Maxima を終了したり, 端末からログアウトしたい時に使えます.

errcatch

errcatch(⟨ 式$_1$⟩, $\cdots$, ⟨ 式$_n$⟩)

引数を一つづつ評価して, エラーが生じなければ最後の値のリストを返します., 任意の引数の評価でエラーが生じた場合,errcatch はエラーを捉え (catches) て即座に [](空のリスト) を返します. この函数はエラーが生じていると疑われる batch ファイルで, エラーを捉えなければ忽ち batch を終了させる様にすると便利です.

error

error(⟨ 引数$_1$⟩, $\cdots$, ⟨ 引数$_n$⟩.) その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,control-$\hat{}$が入力出来ない場所なら何処でも函数を中断させられるので便利です.

大域変数 error にはエラーを記述するリストが設定されており, 最初のものは文字列で, 残りの対象は問題を起しているものです.

errormsg

errormsg()

最新のエラーメッセージを再表示します. 変数 error にはエラーを記したもののリストが設定されており, 最初は文字列で, 残りは問題の対象です.

ttyintfun:lambda([],errormsg(),print(”)) で利用者中断文字 (û) をメッセージの再表示を行う様に設定します.

for 文　反復処理にて用いられます. 詳細は do 文を参照して下さい.

go

go (⟨ ラベル ⟩)

block 内部で,go の引数で指定した block の文に制御を移動するのに用いられます. 文をタグで指定する為,block の中の他の文の様にアトムを文前に置きます.

```
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

return (⟨ 式 ⟩)

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.

throw

throw (⟨ 式 ⟩)

⟨ 式 ⟩ を評価し, 近くにある catch に値を投げ返します.throw は catch と一緒に使われます.

lispdebugmode

lispdebugmode(), debugprintmode(),debug()

利用者に対し, システムプログラマーが用いる虫取り機能を使える様にします. これらのツールは強力で, しかも, 幾つかの操作方法が通常の MACSYMA の階層と違っていますが, それらの利用はとても直接的と感じるでしょう.[幾つかの表示は遅い端末では冗長かもしれませんが, これを制御する大域変数がります]. これらの命令は変換された Maxima のコードを虫取りしなければならない利用者向けに設計されています.

第14章　文脈

14.1　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. 式の計算や簡易化ではこの文脈を利用する事で実行して行きます. 例えば, 非常に簡単な例ですが, 4 の平方根は 2 になりますが,x^2 はどうでしょうか？x は正か負か, ひょっとすると複素数かもしれません. この様に $\sqrt{x^2}$ を考えるだけでも,x に関して様々な情報が必要になります. 先ず, 実数であれば`|x|`となりますね. 更に,$x \leq 0$ であれば答えは x になります.

この処理では,

- x は実数である
- $x \leq 0$ である

と言った x に対する仮定を用いています. 文脈とはこの様な問題を考える上での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. この場合, 文脈 global がその最上位になります. この文脈を上手く利用しながら問題を解いて行きます. これらの文脈を有効や無効にする事が `activate` と `deactivate` で各々行えます.

どの様な文脈が Maxima に存在するかは `contexts;` と入力すれば, 利用可能な文脈のリストが表示されます. 尚,Maxima を立ち上げた時点では, `initial` と `global` の二種類の文脈があります.

新しい文脈の生成は `newcontext(文脈名);` で行います. 又, 既存の文脈の親文脈を生成する場合は `supcontext` で, `supcontext(新規生成する親文脈, 既存の文脈);` とします. 文脈の削除は `killcontext` で行いますが, 現在利用中の文脈, 或いはその文脈が有効 (activate) であれば削除出来ません.

ここで,Maxima に仮定を教える函数は `assume` です. 教えた仮定を忘れさせるのが `forget` で, ある事項に関して, どの様な仮定を Maxima に教えたかを調べたければ,`facts` を用います.

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` で表示し, それから `context` で最初に用いている文脈が `initial` である事を示しています. 次に `newcontext` を使って新しい文脈 mike を生成しています. それから mike の親文脈となる neko を `supcontext` を使って生成しています. この `supcontext` は子としては既存の文脈を指定しなければなりません.

次に,assumle を用いて式を簡易化する例と文脈の切り替えの効果を見ましょう.

```
(%i6) assume(y>0);
(%o6)                                [y > 0]
(%i7) assume(x>0,z<0);
(%o7)                           [x > 0, z < 0]
(%i8) facts();
(%o8)                        [y > 0, x > 0, 0 > z]
(%i9) sqrt(x^2);
(%o9)                                  x
(%i10) context:initial;
(%o10)                               initial
(%i11) sqrt(x^2);
(%o11)                               abs(x)
(%i12) facts();
(%o12)                                 []
(%i13) activate(neko);
(%o13)                                done
(%i14) context;
(%o14)                               initial
(%i15) sqrt(x^2);
(%o15)                                 x
(%i16) facts();
(%o16)                                 []
(%i17) deactivate(neko);
(%o17)                                done
(%i18) sqrt(x^2);
(%o18)                               abs(x)
(%i19) killcontext(neko);
(%o19)                                done
(%i20) contexts;
(%o20)                       [mike, initial, global]
```

先程の文脈の定義に続いて, 文脈 neko で assum を使って変数 x,y,z に対して仮定を設定します. 利用している文脈で設定した仮定全体は facts(); で表示が出来ます. 文脈 neko で abs(x^2) を実行すると x>0 より x が返されます. ここで文脈の切り替えは変数 context に文脈を代入する事で行います. ここでは文脈を neko から inital に変更します. ここで,sqrt(x^2) を実行すると abs(x) が返されます. 何故なら,facts(); から,initial には何も仮定が無い事が分ります. ここで, 文脈 neko の仮定を文脈 initial で利用したければ, actvate(neko); を実行します. その結果,sqrt(x^2) の結果が x になります. 但し,initial の内容には変化がありません. 不要になった文脈は deactivate す

るか,killcontext で削除してしまいます.

他に,features があります. これは組込の特性リストです. 述語が features のリストに含まれる特性を持たせたければ, declare を用います. 例えば birthday を整数として宣言して, birthday に整数としての特性を持たせたければ, declare(birthday,integer); とします. 逆に述語 p が特性 q を持つかどうかを調べる場合には featurep(p,q); で調べます. 尚,declare による影響は,assume とは違い大域的ですが,featurep では declare を用いた文脈上でしか使えません.

```
(%i1) newcontext("mike");
(%o1)                                mike
(%i2) supcontext("neko");
(%o2)                                neko
(%i3) context:mike;
(%o3)                                mike
(%i4) declare(bb,lassociative);
(%o4)                                done
(%i5) featurep(bb,lassociative);
(%o5)                                true
(%i6) facts();
(%o6)                      [kind(bb, lassociative)]
(%i7) bb(bb(a,b),bb(c,d));
(%o7)                       bb(bb(bb(a, b), c), d)
(%i8) context:initial;
(%o8)                              initial
(%i8) bb(bb(a,b),bb(c,d));
(%o9)                      bb(bb(bb(a, b), c), d)
(%i10) aa(aa(a,b),aa(c,d));
(%o10)                      aa(aa(a, b), aa(c, d))
(%i11) facts();
(%o11)                               []
(%i12) featurep(bb,lassociative);
(%o12)                               false
(%i13) killcontext("mike");
(%o13)                               done
(%i14) contexts;
(%o14)                      [neko, initial, global]
(%i15) bb(bb(a,b),bb(c,d));
(%o15)                      bb(bb(bb(a, b), c), d)
```

14.2　文脈に関連する変数

initial
Maxima を立ち上げた時点で利用されるデフォルトの文脈.

global
Maxima の文脈全体の親文脈.

features
組込の属性リスト. 述語に属性を指定する場合は declare を用います. 属性には以下のものがあります:

integer	整数
noninteger	非整数
even	偶数
odd	奇数
rational	有理数
irrational	非有理数
real	実数
imaginary	純虚数
complex	複素数
analytic	解析的函数
inCREasing	増加函数
deCREasing	減少函数
oddfun	奇函数
evenfun	偶函数
posfun	正値函数
commutative	(演算が) 可換
lassociative	左結合的
rassociative	右結合的
symmetric	対称
antisymmetric	歪対称

述語がある属性を持つかどうかは,featurep で調べられます.

assumescalar
デフォルト値:true
引数がスカラー値であると仮定します.

assume_pos
デフォルト値:false.

assume_pos_pred
デフォルト値:false.

14.3　文脈に関連する函数

activate

activate(⟨ 文脈$_1$⟩, ···)

指定した文脈を有効にします. 文脈を無効にしたければ,deactivate を用います.

deactivate

deactivate(⟨ 文脈$_1$⟩, ···)

指定した文脈を無効にします. 逆は activate です.

killcontext

killcontext(文脈$_1$, ···)

文脈を削除します.

newcontext

newcontext(⟨ 文脈 ⟩)

新しい文脈を生成します.

supcontext

supcontext(⟨ 親文脈 ⟩, ⟨ 子文脈 ⟩)

既存の文脈の親となる文脈を生成します.

assume

assum(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, ···)

様々な仮定を設定します. 引数は Maxima の論理式です.assume で設定した事項は facts() で見る事が出来ます.

constantp

constantp(⟨ 式 ⟩)

⟨ 式 ⟩ が定数であれば true, そうで無ければ false を返します.

nonscalar

アトムを dot 演算子に対するリストや行列と同じ挙動にします.

第15章　属性

15.1　属性の宣言と属性値

Maxima では変数や函数に様々な属性を付加する事が出来ます. 更に, 属性に対応する値を設定する事も出来ます.

属性の宣言は declare で行います. これに対し, 属性値の設定は put 函数で行います.

属性値を取出す場合は,get 函数でアトムと属性を指定します. アトムに指定した属性とその値の削除は rem や remove 函数で行います.

15.2　declare 函数

Maxima では変数や函数に様々な属性を付加する事が出来ます. 演算子の infix の様に函数を用いて特定の属性を追加する方法もありますが, Maxima では, それに加えて declare 函数を用いて属性を追加する事が可能です.

declare($\langle a_1 \rangle, \langle f_1 \rangle, \langle a_2 > \rangle, \langle f_2 \rangle, \cdots$)

declare は $\langle a_i \rangle$ に様々な属性 $\langle f_i \rangle$ を指定する函数です. ここで,$\langle a_i \rangle$ と $\langle f_i \rangle$ はアトムやリストでも構いません.

この場合, アトムはリストに含まれる全ての属性を持ちます. 尚,declare は assume と異なり, その影響は特定の文脈上に留まらずに大域的です. その一方で,`facts()` に設定される内容は declare を行った文脈上で設定されています.

Maxima が現在認識し, 利用可能な変数に与えられる属性を以下の表 15.1 に示しておきます.

函数の属性を表 15.2 に示します.

featurep 函数で declare 函数を使って属性が定義されているかどうかを調べる事が可能です.

featurep(⟨ 対象 ⟩,⟨ 属性 ⟩)

で ⟨ 対象 ⟩ が ⟨ 属性 ⟩ を持つと declare で宣言されているかどうか判ります.

additive

declare(f,additive) で函数 f の加法性を宣言します. f が 1 変数函数の場合, 変数内部の和に対して f が分配されます. 例えば, f(x+y) は f(x)+f(y) に簡易化されます. f が多変数函数の場合, 加法性は第一番目の引数に対してのみ設定されます. 例えば,f(h(x)+g(x),x) は f(h(x),x)+f(g(x),x) に簡易

表 15.1: 変数の属性

属性値	概要
even	偶数
odd	奇数
integer	整数
rational	有理数
irrational	非有理数
real	実数
imaginary	虚数
complex	複素数

表 15.2: 函数の属性

属性値	概要
increasing	単調増加函数,$x > y \to f(x) > f(y)$
decreasing	単調減少函数,$x > y \to f(x) > f(y)$
oddfun	奇函数,f(-x)=-f(x)
evenfun	偶函数,f(-x)=f(x)
linear	線形性,$f(ax+by,\cdots) = af(x,\cdots) + bf(y,\cdots)$
hline commutative(symmetric	可換性,f(x,y)=f(y,z)
antisymmetric	歪対称,f(x,y)=-f(y,x)
lassociative	左結合律
rassociative	右結合律

化されます. 但し,f を `sum(x[i],i,i0,i1)` の形式の式に対して作用させる場合, この簡易化は実行されません.

multiplicative

declare(f,multiplicative) で函数 f に乗法性を与えます. ここで,f が単変数であれば,f を積に作用させる時は何時でも f は積に対して分配されます. 即ち,f(x*y) は f(x)*f(y) に簡易化されます.

f が多変数函数の場合, この乗法性は f の最初の引数に対して定義されます. 例えば,f が 2 変数の函数の場合,f(g(x)*h(x),x) は f(g(x),x)*f(h(x),x) に簡易化されます.

尚, この簡易化は f を `product(x[i],i,` 下限`,` 上限`)` といった形式の式に作用させる場合には生じません.

linear

declare(f,linear) で函数 f の第一変数に対し, その線形性を宣言します. 単変数函数 f が linear であると宣言されている場合,a が定数であれば, f(x+y) は f(x)+f(y) に,f(a*x) は a*f(x) に展開されます.

多変数函数 f の場合は最初の変数に対してのみ線形性が付加されます. linear は additive+outative に相当します.

commutative
declare(h,commutative) で函数 h が引数に対して可換函数となります. これは symmetric と同値です.

symmetric
declare(f,symmetric) で函数 f が対称函数である事を宣言します. これは commutative と同値です. 尚, 引数を入れ換えて符号が反転する函数は属性 antisymmetric を持っています.

antisymmetric
declare(h,antisymmetric) で, 函数 h は歪対称として簡易化されます. 例えば,h(x,z,y) は-h(x,y,z) に簡易化されます. 即ち,symmetric や commutative を宣言して得られた結果に, 引数同士を互いに n 回交換して $(-1)^n$ 倍したものになります.

lassociative
declare(g,lassociative) で函数 g の左分配律性を宣言します. 即ち,g(g(a,b),g(c,d)) は g(g(g(a,b),c),d) に簡易化されます.

rassociative
declare(g,rassociative) で函数 g が右分配律を満す函数である事を宣言します. 即ち,g(g(a,b),g(c,d)) は g(a,g(b,g(c,d))) に簡易化されます.

outative
declare(f,outative) で定数の積に対する f との可換性を宣言します. 即ち,f が単変数の場合,f を定数を含む積に作用させると, 定数は f の外に出されます. 例えば,a を定数とすると,f(a*x) は a*f(x) に簡易化されます. 但し, 非アトムの因子は外に出されません.
f が多変数函数であれば,outativity は`'sum` や`'integrate` の様に定義されます. つまり,f(a*g(x),x) は,a が変数 x を含まなければ a*f(g(x),x) に簡易化されます.
尚, デフォルトで`'sum`,`'integrate` と`'limit` が属性 outative を持つと宣言されています.

scalar
declare(f,scalar) で f がスカラーであると宣言します.

nonscalar
declare(f,nonscalar) で f がスカラーではないと宣言します.

constant
declare(a,constant) でアトム a を定数として宣言します.

noun

declare(f,noun) で函数 f を自動的に評価されない名詞型函数として宣言します.

posfun

declare(f,posfun) で函数 f が正値函数 (positive function) であると宣言します. この場合,is(f(x) 0) の結果は true です.

mainvar

declare(x,mainvar) で, 変数 x を主変数にします. この場合,Maxima 内部の順序で,mainvar が変数の最高位になっています. この mainvar を用いて多項式等の内部表現が決定される為, 式の表示だけではなく, 様々な函数を用いた処理も異なる事があります.

```
(%i22) expand((x+y)^4);
                       4       3     2  2     3     4
(%o22)                 y  + 4 x y  + 6 x  y  + 4 x  y + x
(%i23) (declare(x,mainvar),expand((x+y)^4));
                       4       3     2  2     3     4
(%o23)                 x  + 4 y x  + 6 y  x  + 4 y  x + y
```

二つの式の計算を行う際に,mainvar を一方の式で宣言しておきながら, もう一方で宣言していない場合, 内部表現が各々異なる可能性がある為, ev 函数による簡易化が必要となる場合もあります.

alphabetic

maxima のアルファベット, 初期では a から z 迄の小文字と大文字に%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,alphabetic で宣言を行います.

例えば,declare(" ",alphabetic) で宣言すると,new value が名前として使える様になります.

15.3 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 による宣言の他に,put 函数でアトムと属性に対応する属性値を指定し, get 函数でアトムと属性を指定して対応する属性値を取出す事が出来ます.

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

以下の例では put でアトムに属性を指定し,properties でアトムに指定したキーワードに何があるかを確認し, 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
```

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

函数の値の設定も実際は属性を用いて行っています.Maxima では atvalue 函数で指定した点での値を設定します.

具体的には,atvalue(⟨ 式 ⟩,⟨ リスト ⟩,⟨ 境界値 ⟩) で与えます. この函数で値を指定すると,atvalue 属性に座標と函数値が対応付けられ,⟨ 函数 ⟩ の properties リストに atvalue が追加されます.

但し,atvalue の値は get 函数で取出せず, 削除も rem 函数では行えません.

```
(%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]]
```

但し, 属性値は他のものと同様に printprops 函数で, 函数名と属性を指定する事で表示されます.

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

```
(%i22) printprops(all,atvalue);
                                    f(0) = 0

                                    g(0) = 1

                                    g(1) = 2

(%o22)                                done
```

get

get (⟨ アトム ⟩,⟨ キーワード ⟩)

put 函数で ⟨ アトム ⟩ に ⟨ キーワード ⟩ に対して与えた属性を取出します.

props

Maxima が持つ属性の一覧のリスト.infolists で明示的に言及されたものの他の任意の属性, 例えば,atvalue,matchdeclares 等々で,declare 関数で指定された属性と同様のものです.

properties

properties (⟨ アトム ⟩)

⟨ 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.

propvars

propvars (⟨ 属性 ⟩)

⟨ 属性 ⟩ リストのアトムのリストを生成し,prop によって指定された属性を持ちます. その為,propvars(atvalue) は atvalue を持つアトムのリストを生成します.

put

put (⟨ アトム ⟩,⟨ 属性値 ⟩,⟨ 属性 ⟩)

⟨ アトム ⟩ に ⟨ 属性 ⟩ で指定した ⟨ 属性値 ⟩ を加えます. これは利用者がアトムに任意の属性を与える事が可能になります.

尚,put 函数で与えた属性は get 函数で取出せます.

qput

qput (⟨ アトム ⟩,⟨ 属性値 ⟩,⟨ 属性 ⟩)

put に似ていますが, その引数は評価されません.

printprops

printprops(⟨ アトム ⟩,⟨ 属性 ⟩)

printprops([⟨ アトム$_1$⟩,$\cdots$,⟨ アトム$_n$⟩],⟨ 属性 ⟩)

printprops(all,⟨ 属性 ⟩)

⟨ アトム ⟩ と ⟨ 属性 ⟩ に対応する属性値を表示します. ⟨ アトム ⟩ のリストも指定可能ですが,⟨ 属性 ⟩ は一つだけです.

アトムに all を指定すると, 指定した属性を持つ全てのアトムや函数の値が表示されます.

rem

rem (⟨ アトム ⟩,⟨ 属性 ⟩)

⟨ アトム ⟩ から ⟨ 属性 ⟩ で指定された属性と属性値を削除します.

remove

remove(⟨ アトム$_1$⟩,⟨ 属性$_1$⟩,$\cdots$,⟨ アトム$_n$⟩,⟨ 属性$_n$⟩)

remove([⟨ アトム$_1$⟩,$\cdots$,⟨ アトム$_n$⟩], [⟨ 属性$_1$⟩,$\cdots$,⟨ 属性$_n$⟩])

remove (”⟨ アトム ⟩”,operator)

remove (⟨ アトム ⟩,transfun)

remove (all,⟨ 属性 ⟩)

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,function, mode_declare でも構いません.

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

ここで指定するアトムと属性は各々が対応するリストでも構いません. 属性が transfun の場合,translate 函数で変換された LISP 函数が削除されます. この後は,Maxima 版の函数が用いられます.

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.

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から, その属性が削除されます.

remove 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.

atvalue

atvalue (⟨ 式 ⟩,⟨ リスト ⟩,⟨ 境界値 ⟩)

利用者が ⟨ リスト ⟩ で指定した点での ⟨ 境界値 ⟩ を ⟨ 式 ⟩ に割当てます.

⟨ 式 ⟩ は函数 f(v1,v2,...) か, 導函数 diff(f(v1,v2,...),vi,ni,vj,ni,...) (ni は vi による微分の階数) で, その引数の中ではっきりと現れるものでなければなりません.

等式のリストが境界を定めます.⟨ リスト ⟩ は方程式のリスト, 或いは上述の様に単一の方程式 vi=expr でも構いません.atvalue が表示された時に, 記号@@1,@@2,... は函数変数 v1,v2,.... を表現する為に用いられます.

at

at (⟨ 式 ⟩,⟨ リスト ⟩)

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

部分式が atvalue 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 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.

gradef

gradef($\langle f(x_1,\cdots,x_m)\rangle,\langle g_1\rangle,\cdots,\langle g_n\rangle$)

gradef($\langle f\rangle,\langle x\rangle,$⟨ 式 ⟩)

函数 $\langle f\rangle$ の n 個の引数に対する微分を, $\frac{d<f>}{dx_i} =< g_i\rangle$ で定めます.

もしも変数の総数数 mn 個の勾配 n よりも小さい場合, 最初の $\langle f\rangle$ の i 番目の引数が参照されます.x_i は函数定義で用いるダミー変数と同類で, 函数 $\langle f\rangle$ 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 の引数は $\langle g \rangle$ が定義された函数ならば, その函数が呼出され, 結果が用いられます. 勾配は函数が第一階微分を除いて正確に判らない場合で, より高階の微分を得たい時に必要となります.

gradef($\langle f \rangle, \langle x \rangle, \langle$ 式 $\rangle$) は $\langle x \rangle$ による $\langle$ 函数 $\rangle$ の微分が $\langle$ 式 $\rangle$ となる事を宣言します. この時に,depends($\langle f \rangle, \langle x \rangle$) が実行されます. この depends によって属性 dependency が付加されます.

gradef は Maxima の既に定義された函数の微分を再定義する事にも使えます. 例えば,gradef(sin(x),sqrt(1-sin(x)^2)) の様に出来ます. 但し, 添字された函数に gradef は使えません.

尚,gradef で勾配を定義すると, 大域変数 gradefs にその函数の名前が蓄えられます.

使えます.

depends

depends ($\langle$ 函数$_1\rangle, \langle$ 変数$_1\rangle, \cdots, \langle$ 函数$_n\rangle, \langle$ 変数$_n\rangle$)

函数の変数に対する従属性 (dependency) を宣言します. 即ち,depends(f,x) で f は x の値に従属する性質を付加します.

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

函数の変数に対する従属性は dependencies に登録された函数の情報から調べる事が出来ます.

次の例では,f と g が x と y に,r と s が u,v と w に依存し,u が t に従属する事を diff に報せる為に, depends([f,g],[x,y],[r,s],[u,v,w],u,t) としています

```
(%i11) dependencies;
```

```
(%o11)                                []
(%i12) depends([f,g],[x,y],[r,s],[u,v,w],u,t);
(%o12)         [f(x, y), g(x, y), r(u, v, w), s(u, v, w), u(t)]
(%i13) dependencies;
(%o13)         [f(x, y), g(x, y), r(u, v, w), s(u, v, w), u(t)]
(%i14) diff(r.s,u);
                                dr           ds
(%o14)                          -- . s + r . --
                                du           du
(%i15) diff(r.s,t);
                                 dr du       ds du
(%o15)                       s . -- -- + r . -- --
                                 du dt       du dt
```

この例では,depends を実行する事で,dependencies に depends で従属性を宣言した函数が変数と一緒にリストに加えられている事に注目して下さい. 又, 微分では合成函数の微分も出来ます.

depends で設定した従属性は,remove 函数を使って削除する事が出来ます. 上の例の (%i11) で宣言した r の u に対する従属性を削除したければ, remove(r,dependency) と入力します.

```
(%i16)  remove(r,dependency);
(%o16)                             done
(%i17) ''%i15;
                                     ds du
(%o17)                           r . -- --
                                     du dt
```

diff は dependencies に設定された情報を用います. 積分,laplace 変換等での引数は, integrate(f(x),x) の様に, 従属性を函数内部ではっきりと与えなければなりません.

nounify

nounify (⟨ 函数 ⟩)

⟨ 函数 ⟩ の名詞型を返します. 動詞函数を名詞型であるかの様に参照したい場合に必要です. 幾つかの動詞型函数は, ある引数に対して評価が出来なかった場合, それらの名詞型を返す事に注意して下さい. 函数の呼出で単引用符' を頭に置いた場合も, その名詞型か返されます.

verbify

verbify (⟨ 函数 ⟩)

⟨ 函数 ⟩ を動詞型で返します.

第16章　Maximaによる評価

16.1　代入と評価に関連する大域変数

evflag
　デフォルト値:[]
　ev 函数が認知しているもののリストが設定されます. 例えば,ev(%,numer) で ev を呼出し, 指定されていれば ev の実行時で, 項目に true が設定されます.
　ここで,evflag 属性を持つ大域変数を以下に示します.

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

evfun
　デフォルト値:[]
　このリストに函数名が含まれていれば,ev 函数がその函数に対して適用されます.evfun の初期値としては,

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

exptsubst
　デフォルト値:[false]
　true であれば%e(a*x) の%e の y による置換操作が可能になります.

opsubst
　デフォルト値:[true]
　false であれば,subst 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:false,subst(x^ 2,r,r+r[0]))` を実行すると, r+r[0] の左側の r に代入されますが,r[0] の r には代入されません.

```
(c1) subst(a,x+y,x+(x+y)^2+y);
                                    2
(d1)                       y + x + a
(c2) subst(-%i,%i,a+b*%i);
(d2)                               a - %i b
```

(注：c2 の方法は共役複素数を求める一つの方法です).

prederror
デフォルト値:[false]

sublis_apply_lambda
デフォルト値:[true]
sublis を用いた後の簡易化で,lambda による代入を行うかどうか, 作用させるべき物を得る為に ev を実行しなければならないかどうかを制御します.true であれば, この操作の遂行を意味します.

16.2 代入と評価に関連する函数

'

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. この場合,'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.

''

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 に作用させて動詞型で返します.

```
(%i65) test:2*%pi;
(%o65)                                2 %pi
(%i66) sin(test);
(%o66)                                  0
(%i67) test:%pi/4;
                                       %pi
(%o67)                                 ---
                                        4
(%i68) ''%i66;
                                       1/2
                                      2
(%o68)                                ----
                                       2

(%i69) ''sin(test);
(%o69)                          .7071067811865475
```

equal

equal(⟨式$_1$⟩,⟨式$_2$⟩)

is 函数と一緒に使われ,⟨式$_1$⟩ と ⟨式$_2$⟩ が (ratsimp で指定された) 全ての可能な変数値に対して等しい (又は等しくない) 場合, 又, その時に限って true(又は 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) と同値です.

```
(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

is(⟨ 述語 ⟩)

⟨ 述語 ⟩ が,Maxima の文脈や宣言等に含まれている事象に適合するかどうかを判定します.is が true と返すのは,⟨ 述語 ⟩ に含まれる変数に関して, 全ての値で述語が true となる場合で, そうでない場合は false を返します. それ以外は prederror の設定に依存します.

is は prederror が true の場合はエラーを出力を行い,falase であれば unknown を返します.

eval

eval(⟨ 式 ⟩)

⟨ 式 ⟩ の評価を行います.LISP の `eval` 函数と同じ働きをします.

ev

ev(⟨ 式 ⟩, ⟨ 引数$_1$⟩, $\cdots$, ⟨ 引数$_n$⟩)

Maxima の最も強力で高機能の命令の一つです.⟨ 引数 ⟩ で指定した環境で ⟨ 式 ⟩ を評価します. 評価は次の手順で実行されます.

1. 最初に以下の様に設定された ⟨ 引数$_i$⟩ を探索し, 環境が設定されます.

   - simp
     与式を大域変数 simp の設定とは無関係で簡易化を行います. 大域変数 simp は, 値が false の場合は簡易化を禁じる大域変数です.
   - noeval
     ev の評価 (以下の (4) を見よ) を中断します. これは他の大域変数と組合せたり, 与式が再評価されずに再簡易化が行われるので便利です.
   - expand
     与式の展開を行います. 尚,expand(m,n) で maxposex と maxnegex の値に m,n を各々設定して展開を行います.
   - detout

     与式で計算した逆行列に対し, その行列式を逆行列の外に置いたままにして各要素を割らないでおきます.
   - diff

     与式内部の指定された全ての微分を実行します.

- derivlist(⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩)
  指定した変数に対し, 微分を実行します.
- float
  非整数の有理数を浮動点小数に変換します.
- numer

  数値変数を持つ幾つかの数学函数 (指数函数を含む) は浮動点小数で評価され, 与式中の変数で, 数値を割当てられたものは, 割当てられた値で置換えます. 又, 大域変数 float も入ります.
- pred

  述語 (true か false で評価されるべき式) が評価されます.
- eval

  exp の特別な後評価が生じる (以下の段階 (5) を見よ).
- e が evflag として宣言されたアトムであれば, 与式の評価中は e を true とします.
- v:式 (または代りに v=式)

  与式の評価中に, 式の値が v に束縛されます.v が Maxima のオプションであれば, 与式の評価の間, その値が用いられる事に注意して下さい. ev に対して一つ以上の引数がこの型であれば, 並行して束縛が実行されます. もし,v がアトムでなければ, 束縛ではなく代入が実行されます.
- infeval
  無限評価モードに入ります. この場合,ev は与式の変化がなくなる迄, 繰返し式を評価します. 変数, ここでは x としますが, このモードで評価されるのを防ぐ為に, 単に ev に対する引数として x='x を含めます. 勿論,ev(x,x=x+1,infeval) の様な式は無限ループを生成します.
- evfun

  函数が evfun を持つ函数として宣言されていれば, その函数を与式に適用します. 任意の別の函数名 (例えば,sum) が与式中に現われると, これらの名前を動詞型として評価し, 与式に現われる函数 (f(引数) とします) を与式評価の為に ev の引数として f(引数):=本体を与えて局所的に定義しても構いません.

上で言及されていないアトム, 添字された変数, 又は添字された式が引数として与えられていれば, それらが評価されて, その結果が方程式や割当てであれば, 指定された束縛や代入が実行されます. 結果がリストであれば, そのリストの成分は, それらが ev に対して与えられた追加の引数であるかの様に扱われます. この為, 方程式のリスト (例えば,[x=1,y=a$\hat{2}$]) や solve

で返されるものの様な方程式の名前のリスト (例えば,[e1,e2]. ここで,e1 と e2 は方程式である) で与えても構いません.

ev 函数の引数は, 左から右への順番で処理される方程式の代入を除いて, 任意の順序で与えられます. そして,ev(exp,ratsimp,realpart) の様に合成された evfunc は,realpart(ratsimp(exp)) として処理されます.

大域変数 simp,numer,float と pred は block 文の中で局所的に, 或いは Maxima のトップレベルで大域的にそれらが再設定される迄, 効果を持ち続ける様に設定していても構いません.

与式が CRE 形式であれば,ev は結果を CRE 形式で返しますが, 大域変数の numer と float 両方共 true にはなりません.

2. 段階 (1) の間, 引数か, もしも値が方程式であれば, ある引数の変数中で方程式の左側に現れる添字されていない変数のリストが作られます.

   与式中の変数 (配列函数に無関係の添字された変数と添字されていない変数の両方) は, それらの大域変数値で置換えられますが, このリストに現れるものは除外されます. 通常, 与式はラベルであるか (以下の (c2) の様に) %であり,ev がそれに作用しても良い様に, この段階は単純にラベルによって名付けられた式を見出します.

3. 任意の下添字が引数で指定されていれば, それらは直ちに実行されます.

4. 結果の式は (引数の内の一つが noeval で無い限り) その様に再評価され, それから引数に従って簡易化されます. 任意の函数の与式での呼出しは, その中の変数が評価された後に実行されます. それ故,ev(f(x)) が f(ev(x)) の様に振舞う事に注意して下さい.

5. 引数のうち一つが eval であれば,(3) と (4) の段階を繰り返す.

```
(c1) sin(x)+cos(y)+(w+1)^2+'diff(sin(w),w);
                          d                   2
(d1)  cos(y) + sin(x) + -- sin(w) + (w + 1)
                          dw
(c2) ev(%,sin,expand,diff,x=2,y=1);
                     2
(d2)       cos(w) + w  + 2 w + cos(1) + 1.90929742
```

ev 函数は,Maxima のトップレベルでは,ev() 無しにその引数を何処で入力しても構いません. つまり, 次の様に簡単に書いても良いのです.

ev 函数の別の表記方法

exp, arg1, ...,argn.

これは他の式, つまり, 函数や block 等での成分として記述する事は出来ません.

```
(c4) x+y,x:a+y,y:2;
(d4)                  y + a + 2
```

```
(並行した束縛の過程に注意せよ)
(c5) 2*x-3*y=3$
(c6) -3*x+2*y=-4$
(c7) solve([d5,d6]);
solution
                                  1
(e7)                       y =  - -
                                  5
                            6
(e8)                    x = -
                            5
(d8)                [e7, e8]
(c9) d6,d8;
(d9)                - 4 =  - 4
(c10) x+1/x > gamma(1/2);
                     1
(d10)           x + - > sqrt(%pi)
                     x
(c11) %,numer,x=1/2;
(d11)             2.5 > 1.7724539
(c12) %,pred;
(d12)                    true
```

第17章　代入操作

17.1　はじめに

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.

17.2　関連する大域変数

opsubst

デフォルト値:[true]

false であれば, 函数 subst が式中の演算子にも代入する事を防ぎます.

```
(%i63) subst(x^2,r,r+r[0]);
                                    2     2
(%o63)                             x  + (x )
                                                0
(%i64)  (opsubst:false,subst(x^2,r,r+r[0]));
                                     2
(%o64)                              x  + r
                                           0
```

この様に r[0] に安易な代入が行われていない事に注目して下さい.

17.3　関連する函数

fullratsubst

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

ratsubst 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 は lratsubst 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

lratsubst(⟨ リスト ⟩,⟨ 多項式 ⟩)

subst(方程式のリスト,式) と似ていますが,rstsubst が subst の代りに使われる点で異なります.lratsubst の最初の因子は方程式か方程式のリストで,subs から得られる書式と同一のものなければなりません. 代入は方程式のリストで与えられた順序 (リストの左から右) 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
```

ratsubst

ratsubst (a, b, c)

c に含まれる b に a を代入します.b は和, 積, 冪等であっても構いません.subst が代入を行う個所で,ratsubst 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 は z+a を返します.

radsubstflag が true であれば,ratsubst が x に u を sqrt(x) として代入する事を許容します.

```
(c1)  radsubstflag:false;

(d1)                                false
(c2)  ratsubst(u,sqrt(x),x);

(d2)                                  x
(c3) radsubstflag:true;

(d3)                                true
(c4)  ratsubst(u,sqrt(x),x);

                                      2
(d4)                                 u
```

この様に ratsubstflag が true でないと sqrt(x) が x に代入されません.

sublis

sublis (⟨ リスト ⟩,⟨ 式 ⟩)

⟨ 式 ⟩ に ⟨ リスト ⟩ で指定した複数の代入を並行して行ないます. ⟨ リスト ⟩ には,a=b の形で式を記述します. 演算子=の左辺の a が ⟨ 式 ⟩ に含まれるアトムや函数名を指定し, 右辺の b に置換える値や式を設定します.

```
(%i23) sublis([sin=cos,x=2*theta+1],sin(x-1)^2);
                                    2
(%o23)                           cos (2 theta)
(%i24) sublis([sin=cos,cos=sin],cos(x)^2+sin(x+1)^3);
                                3             2
(%o24)                       cos (x + 1) + sin (x)
```

尚,`sublis([sin=cos,cos=sin],cos(x)^2+sin(x+1)^3)` の様な入れ換えの指定では,⟨ リスト ⟩ に含まれる式の代入を順番に行うのではなく同時に行う為,cos と sin が入れ換えられている事に注意して下さい.

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

subst

subst (a, b, c)

c の中の b を a で置き換える. ここで b は原子か,c に完全に含まれる部分式でなければならない. 例えば,x+y+z は 2*(x+y+z)/w に完全に含まれる部分式であるが,x+y はそうではない.b がこれらの特徴を持たなければ, 時には substpart か ratsubst (以下を参照せよ) を使っても良い. 他に b が e/f の様な形式であれば subst(a*f,e,c) が使え, 同様に b が e**(1/f) の形式であれば,subst(a**f,e,c) が使える.

この subst 命令は `x^-y` で `x^y` を認識するので, subst(a,sqrt(x),1/sqrt(x)) は 1/a となる.a と b はまた"で括られた式の演算子や関数名でも良い. 微分形式での独立変数に代入を行う場合,at 関数 (以下を見よ) を利用すべきである.

注意: subst は substitute の別名である.subst(eq1,exp) や subst([eq1,...,eqk],exp) は他の可能な代入形式である. 各 eqi は指定した代入が実行されるべき等式である. 各々の等式でその右側が式 exp の左側に代入される.

substinpart

substinpart (x, exp, n1, ...) substpart に似ているが,exp の内部式に対して作用する.

```
(c1) x.'diff(f(x),x,2);
                                    2
```

```
                                  d
(d1)                         x . (--- f(x))
                                   2
                                  dx
(c2) substinpart(d**2,%,2);
                                     2
(d2)                              x . d
(c3) substinpart(f1,f[1](x+1),0);
(d3)                            f1(x + 1)
```

追加情報

part 関数の最後の引数が添字のリストであれば, 幾つかの部分式が取り出され, 各々はそのリストの添字に関連するものである. それ故,part(x+y+z,[1,3]) は z+x となる.

piece は part 関数を用いた時に選ばれた最後の式の値を保つ. その関数の実行中に設定され, それ故に, その関数でそれ自体を以下に見る様に参照しても良い.

partswitch[false] が true に設定されていれば, 存在しない式の成分を選択した場合, end が返され, それ以外ではエラーメッセージが与えられる.

```
(c1)  27*y**3+54*x*y**2+36*x**2*y+y+8*x**3+x+1;
           3         2       2            3
(d1)    27 y  + 54 x y  + 36 x  y + y + 8 x  + x + 1
(c2)  part(d1,2,[1,3]);
                 2
(d2)          54 y
(c3)  sqrt(piece/54);
                y
(c4)  substpart(factor(piece),d1,[1,2,3,5]);
                          3
(d4)           (3 y + 2 x) + y + x + 1
(c5) 1/x+y/x-1/z;
                                1   y   1
(d5)                           - - + - + -
                                z   x   x
(c6) substpart(xthru(piece),%,[2,3]);
                               y + 1   1
(d6)                           ----- - -
                                 x     z
```

又, オプションの inflag を true に設定して part/substpart を呼出す事は, inpart/substinpart を呼出す事と同じである.

substpart

substpart (x, exp, n1, ..., nk)

part 関数の様に引数の残り (訳者注：n1,...,nk の事) で抜き出した部分式に x を代入する.exp の新しい値を返す.x は exp の演算子として代入されるべきものでも良い. 場合によっては”で括られていなければならない (例えば,substpart(”+”,a*b,0); -¿ b + a).

```
(c1) 1/(x**2+2);
                                    1
(d1)                              ------
                                   2
                                  x  + 2
(c2) substpart(3/2,%,2,1,2);
                                    1
(d2)                              --------
                                   3/2
                                  x    + 2
(c3) a*x+f(b,y);
(d3)                           a x + f(b, y)
(c4) substpart("+",%,1,0);
(d4)                          x + f(b, y) + a
```

(c2) の代入では 1/(x**2+2) の内部形式を利用し, その内部の 2,1,2 で指定される要素を入れ換えている. 内部形式は lisp の s 式による表現 (/ 1 (+ (** x 2) 2) に似たものである (実際は演算子が例えば,"+"が (mplus) と原子のリストになっている程度で本質的な構造は同じ). 実例での 2,1,2 の意味はこの内部形式の階層構造に関連する.

先頭の 2 が内部形式のリストの 2 番目の元を意味するが, 内部形式で先頭が演算子となるので実際は 1 を加えたものが対応する. 従って,3 番目の成分 (+ (** x 2) 2) となる. 次の 1 は同様に (** x 2),2 で, このリストの 2 番目の引数 2 が対応し, それを 3/2 に変更する為, 最終的に 1/(x**(3/2)+2) となる. ここで 0 が演算子となるので,0 を指定すれば一番上の"+"が対応する.

```
(c15) a:1/(x**2+2);
                                    1
(d15)                             ------
                                   2
                                  x  + 2
```

```
(c16)  substpart("*",a,0);

                                    2
(d16)                              x  + 2
(c17)  substpart("*",a,2,0);

                                     1
(d17)                                ----
                                       2
                                    2 x
```

又, この階層を利用すれば, 成分の入れ替えも可能である.

```
(c22) substpart("sin(x)+cos(y)+2",a,2,1);

                                       1
(d22)                  ------------------
                       sin(x)+cos(y)+2 + 2
```

この例では,x^2 を sin(x)+cos(y) で置き換えている.

オプションの inflag を true にし,part/substpart を用いる事は inpart/substinpart を直接利用する事と同じである.

平成 17 年 12 月 6 日 (火)

第18章　簡易化について

18.1　MAXIMAでの式の簡易化

MAXIMA の式の簡易化には, 大域函数変数の指定による MAXIMA の自動簡易化があります.

先ず, 変数 demoivre を true に設定すると,%eˆ (a+b*%i) の b が実数であれば, %eˆ a*(cos(b)+%i*sin(b)) と自動的に展開されます. %emode が true であれば,%eˆ (%pi*%i*x) の x が整数, 或いは分母が 2,3,4,6 の有理数であれば,cos(%pi*x)+%i*sin(%pi*x), その他の数値であれば,%eˆ (%pi*%i*y), ここで y=x-2k,—y—<1 と変換されます.

MAXIMA では多項式を入力しても,(x+1)ˆ 2 や (x+1)*(y-2) の様な式は自動的に展開されません. 但し, 多項式の冪に関しては, 大域変数の expop や expon の値を変更する事で, 自動的に展開させる事が出来ます. これらの変数はデフォルト値が 0 に設定されている為,(x+1)ˆ 0 の様に冪が零の場合は自動的に 1 に変換されます.expop を例えば 4 に変更すると, 冪の次数が 0 以上,4 以下であれば MAXIMA は冪を自動的に展開します. 又,expon を 4 にすると, 負の冪の次数の絶対値が 0 以上, 4 以下であれば自動的に冪を展開します. 以下に簡単な例を示しましょう.

```
(%i38) expon:4;
(%o38)                                4
(%i39) (x+1)^(-3);
                                      1
(%o39)                      ------------------
                             3      2
                            x  + 3 x  + 3 x + 1
(%i40) (x+1)^(-5);
                                     1
(%o40)                           --------
                                        5
                                 (x + 1)

(%i41) expop:4;
(%o41)                                4
(%i42) (x+1)^4;
                          4      3      2
(%o42)                   x  + 4 x  + 6 x  + 4 x + 1
(%i43) (x+1)^5;
```

```
                                        5
(%o43)                             (x + 1)
```

18.2 簡易化に関連する大域変数

demoivre
デフォルト値:[false]
true ならば,%eˆ (a+b*%i) の b に%i が含まれていなければ, %eˆ a*(cos(b)+%i*sin(b)) となって,a と b は展開されません.

%emode
デフォルト値:[true]
true であれば,%eˆ (%pi*%i*x) が次の様に簡易化されます.

- x が整数, 或いは 1/2,1/3,1/4 や 1/6 と整数の積であれば, cos(%pi*x)+%i*sin(%pi*x) となります.
- その他の数値の場合,%eˆ (%pi*%i*y) となります. ここで y は x-2*k,k は abs(y) < 1 となる整数です.

%emode が false の場合,%eˆ (%pi*%i*x) の特殊な簡易化は何も実行されません.

%enumer
デフォルト値:[false]
true であれば,%e は 2.718... に変換されます.%eˆ X の指数が整数の場合に限り, この変換が行なわれます.

expon
デフォルト値:[0]
expon は expand 函数とは別個に,MAXIMA が自動的に展開する式に含まれる負の冪の次数を定めます.

expop
expop[0]
自動的に展開される正の最も高い次数. 負の冪に対しては expon.

maxnegex
デフォルト値:[1000]
maxnegex は expand 函数で展開される絶対値が最大となる負の冪の次数です. 正の冪の最大次数は maxposex です.

maxposex
デフォルト値:[1000] maxposex は expand 函数で展開される最大の正の冪の次数です. 負の冪の最大次数は maxnegex です.

prodhack

デフォルト値:[false]

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.

18.3 簡易化に関連する函数

apply_nouns

apply_nouns (⟨ 式 ⟩)

式中の名詞型を処理します. 例えば, exp:'diff(x^ 2/2,x);apply_nouns(exp); は x になります. これは apply_nouns(exp) ではなく ev(exp,nouns) を用いた場合と同じ結果になりますが,apply_nouns を用いた方が速くてメモリ消費もより少ない長所があります. また,ev で問題が生じる恐れのある translate 等で変換された MAXIMA のプログラムに対しても使えます.

名詞形式の演算子に対し,apply で関連する規則を適用させるのは, 呼出された apply_nouns であって,ev_nouns ではない事に注意しましょう.

paragraphdemoivre

demoivre(⟨ 式 ⟩)

demoivre 変数の設定や ev による式の再評価なしで変換を行います.

exponentialize

exponentialize(⟨ 式 ⟩)

⟨ 式 ⟩ を指数函数形式に変換します.

sumcontract

sumcontract(⟨ 式 ⟩)

上限と下限の差が定数となる加法の全ての総和を結合します. 結果は, 各総和の集合に対して, 全ての適切な外の項を加えて一つの総和にしたものを含む式になります.suncontract は全ての互換な総和を結合し, 可能であれば, 総和の一つから添字の一つを用います.sumcontract を実行する前に intosum(⟨ 式 ⟩r) の実行が必要かもしれません.

askinteger

askinteger (⟨ 式 ⟩)

askinteger (⟨ 式 ⟩,⟨ オプション引数 ⟩)

⟨ 式 ⟩ は任意の有効な MAXIMA の式で,⟨ オプション引数 ⟩ は even(偶数), odd(奇数),integer(整数) の何れか一つで, 省略された場合は内部で integer が設定されます. この函数は MAXIMA に蓄えられた情報から ⟨ 式 ⟩ が even,odd, 或いは integer であるかを決定しようとします.MAXIMA に蓄えられた情報では不十分な場合, 利用者に質問して,MAXIMA に情報を蓄えます.

```
(%i16) aa:1;
(%o16)                                1
(%i17) askinteger(aa);
(%o17)                               yes
(%i18) askinteger(aa,odd);
(%o18)                               yes
(%i19) askinteger(aa,even);
```

```
(%o19)                                  no
(%i20) askinteger(yy);
Is  yy  an integer?

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

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

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

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

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

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

ここでの例で示す様に yy が integer であると指定すると, それから yy は integer となります. 更

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

asksign

asksign (⟨ 式 ⟩)

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

利用者の答は MAXIMA に記録されます.

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

expand

expand (⟨ 式 ⟩)

expand (⟨ 式 ⟩, ⟨p⟩, ⟨n⟩)

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, 乗法 (可換と非可換の両方) を ⟨ 式 ⟩ の全ての階層で加法に対して分配します.

尚, 多項式に対しては, より効率的なアルゴリズムを用いる ratexpand を通常用いるべきです.

大域変数の maxnegex と maxposex は MAXIMA が展開する式の負と正の冪の次数の最大値を設定します.

expand(⟨ 式 ⟩,p,n) の場合,p を maxposex,n を maxnegex に各々対応し, この条件で ⟨ 式 ⟩ の展開を行います.

expandwrt

expandwrt (⟨ 式 ⟩, ⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩)

⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩ に対し, ⟨ 式 ⟩ を展開します.⟨ 変数$_i$⟩ を含む全ての積は明示的に現れます. 返される形式は ⟨ 変数$_i$⟩ を持つ式の和の積を持たないものとなります.⟨ 変数$_i$⟩ は変数, 演算子や式でも構いません.

デフォルトで分母は展開されませんが, 大域変数の expandwrt_denom でこれは制御が出来ます. この函数を使う為には予め load(stopex) で読込を実行します.

expandwrt_factored

expandwrt_factored (⟨ 式 >⟩, ⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩)

expndwrt に似ているが, 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.

intosum

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.

幾つかの場合では intosum の前に scanmap(multthru,⟨ 式 ⟩) が必要かもしれません.

numerval

numerval (⟨ 変数$_1$⟩, ⟨ 式$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩, ⟨ 式$_n$⟩)

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

radcan

radcan (⟨ 式 ⟩)

⟨ 式 ⟩ は, 対数函数, 指数函数と冪乗根を含んでいても構いません. ⟨ 式 ⟩ をある変数順序に対する CRE 表現に変換し, 簡易化を行います. 特定の変数順序に対し,CRE 表現は一意に定まります (従って,CRE 表現は式の正準表現になります). その為,radcan を用いた簡易化も一意に定まります. 但し,radcan は時間を多く消費します. これは因子分解と指数の部分分数展開を基本とした簡易化の為, 式の成分の間の関係を探索する為です.

scsimp

scsimp (⟨ 式 ⟩, ⟨ 規則$_1$⟩, $\cdots$, ⟨ 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.

unknown

unknown (⟨ 式 ⟩)

⟨ 式 ⟩ が一つの演算子を持つ場合, 函数が組込みの簡易化函数が分らない場合に true を返します.

第19章　三角函数

Maxima は沢山の三角函数を持っています. 三角函数の恒等式, 即ち,$cos(x)^2 + sin(x)^2 = 1$ や $cos(2 * x) = 2 * cos(x)^2 - 1$ の様なものは予め Maxima に組込まれていますが, システムの並び照合機能を使う事で, 多くの恒等式を規則として利用者が付加する事が出来ます.

Maxima で予め定義された三角函数は下記のものがあります.

acos, acosh, acot, acoth, acsc, acsch,asec,asech,asin,asinh, atan, atanh, cos, cosh, cot, coth,csc, csch, sec, sech, sin, sinh,tanl,tanh.

三角函数に付随する函数は, 大域変数の trigexpand,trigreduce と trigsign を参照して下さい. 二つの share パッケージは Maxima に組み込みの簡易化の規則の trig と atrig を拡張します.

19.1　三角函数に関連する大域変数

halfangles
　デフォルト値:[false]
　true の場合,$\frac{\theta}{2}$ に対して簡易化が実行されます.

trigexpandplus
　デフォルト値:[true]
　trigexpand での和の規則を制御します. つまり,trigexpand 命令が使われるか, 大域変数 trigexpand が true に設定されている時に, 和 (例えば,sin(x+y)) の展開が trigexpandplus が true の場合に限って実行されます.

trigexpandtimes
　デフォルト値:[true]
　trigexpand で積の規則を制御します. つまり,trigexpand 命令が使われるか,trigexpand スイッチが true に設定されている時に, 積 (例えば,sin(2*x)) の展開が trigexpandtimes が true の場合に限って実行されます.

triginverses
　デフォルト値:[all]
　三角函数, 双曲函数とその逆函数との合成の簡易化を制御します.
　all の場合, 両方, 例えば,atan(tan(x)) と tan(atan(x)) が x に簡易化されます.
　true の場合,arcfunction(function(x)) の簡易化が切り捨てられます.
　false であれば arcfunc(func)) と fun(arcfun(x)) の簡易化が切り捨てられます.

trigsign
　デフォルト値:[true]
　true であれば三角函数に対し負の引数の簡易化を許容します.
　例えば,trigsin が true の時に限り,sin(-x) は-sin(x) となります.

19.2　函数

acos
　逆余弦函数

acosh
　逆双曲線余弦函数!

acot
　逆余接函数

acoth
逆双曲線余接函数

acsc
逆余割函数

acsch
逆双曲線余割函数

asec
逆正割函数

asech
逆双曲線正割函数

asin
逆正弦函数

asinh
逆双曲線正弦函数

atan
逆正接函数

atan2
atan2 (y,x)
区間 (-%pi,%pi) の間で atan(y/x) を計算します.

atanh
逆双曲線正接函数

atrig1
atrig1 には逆三角函数に対する幾つかの追加の簡易化の規則を含みます. 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.

cos
余弦函数

cosh
　双曲線余弦函数

cot
　余接函数

coth
　双曲線余接函数

csc
　余割函数

csch
　双曲線余割函数

sec
　正割函数

sech
　双曲線正割函数

sin
　正弦函数

sinh
　双曲線正弦函数

tan
　正接函数

tanh
　双曲線正接函数

trigexpand
　trigexpand (< 式 >)
　< 式 > の中で生じる角度の和と角度の倍を持つ三角函数と双曲函数の展開を実行します. 最良の結果を得る為に, 予め < 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度の和と角度の積の展開を行います.sin と cos の全体の展開を直ちに得る為には, trigexpand:true; とします.

```
(c1) x+sin(3*x)/sin(x),trigexpand=true,expand;
                              2           2
(d1)                     - sin (x) + 3 cos (x) + x
(c2) trigexpand(sin(10*x+y));
(d2)               cos(10 x) sin(y) + sin(10 x) cos(y)
```

trigreduce

trigreduce (⟨ 式 ⟩,⟨ 变数 ⟩)

⟨ 变数 ⟩ 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,⟨ 变数 ⟩ が省略されると,⟨ 式 ⟩ の全ての変数が利用されます.

```
(c4) trigreduce(-sin(x)^2+3*cos(x)^2+x);
(d4)                        2 cos(2 x) + x + 1
```

三角函数簡易化ルーチンは幾つかの単純な場合で宣言された情報を用います.

変数に関する宣言は次の様に使われる,

例えば,

```
(c5) declare(j, integer, e, even, o, odd)$
(c6) sin(x + (e + 1/2)*\%pi)$
(d6)                         cos(x)
(c7) sin(x + (o + 1/2) \%pi);
(d7)                        - cos(x)
```

trigsimp

trigsimp (⟨ 式 ⟩)

tan,sec 等を含む ⟨ 式 ⟩ の簡易化の為に, 恒等式 $sin(x)^2+cos(x)^2=1$ と $cosh(x)^2-sinh(x)^2=1$ を使って,sin,cos,sinh,cosh へと変換し, その結果に trigreduce を用いればより進んだ簡易化が得られる様にします.

trigrat

trigrat (⟨ 三角函数を含む式 ⟩)

三角函数式の疑線型形式の正規簡易化を与えます.

trigexp は幾つかの sin,cos や tan の有理函数であり, それらの引数は幾つかの変数 (又は核) と %pi/n(n は整数) の整数係数の線型結合となっています.

結果は簡易化された sin と cos の線型な分子と分母を持つ分数となります. trigrat は可能であれば常に線形化します (d.lazard が記述した).

```
(c1) trigrat(sin(3*a)/sin(a+\%pi/3));

(d1)                    sqrt(3) sin(2 a) + cos(2 a) - 1


(c4)   c:\%pi/3-a-b;

                                          \%pi
(d4)                             - b - a + ---
                                           3

(c5)   bc:sin(a)*sin(3*c)/sin(a+b);

                              sin(a) sin(3 b + 3 a)
(d5)                          ---------------------
                                   sin(b + a)

(c6)   ba:bc,c=a,a=c$

(c7)   ac2:ba^2+bc^2-2*bc*ba*cos(b);

        2       2
     sin (a) sin (3 b + 3 a)
(d7) -----------------------
               2
            sin (b + a)

                                        \%pi
  2 sin(a) sin(3 a) cos(b) sin(b + a - ---) sin(3 b + 3 a)
                                         3
- ------------------------------------------------------
                           \%pi
                  sin(a - ---) sin(b + a)
                           3
```

```
      2         2         \%pi
   sin (3 a) sin (b + a - ---)
                          3
 + --------------------------
             2     \%pi
          sin (a - ---)
                    3
```

```
(c9)   trigrat(ac2);
totaltime= 65866 msec.  gctime= 7716 msec.

(d9)
- (sqrt(3) sin(4 b + 4 a) - cos(4 b + 4 a)

- 2 sqrt(3) sin(4 b + 2 a)

+ 2 cos(4 b + 2 a) - 2 sqrt(3) sin(2 b + 4 a) + 2 cos(2 b + 4 a)

+ 4 sqrt(3) sin(2 b + 2 a) - 8 cos(2 b + 2 a) - 4 cos(2 b - 2 a)

+ sqrt(3) sin(4 b) - cos(4 b) - 2 sqrt(3) sin(2 b) + 10 cos(2 b)

+ sqrt(3) sin(4 a) - cos(4 a) - 2 sqrt(3) sin(2 a) + 10 cos(2 a)

   - 9)/4

(* 訳者注：

pentium!!! 600mhz での trigrat(ac2) の結果：

gcl 上の maxima:   0.51[sec]
clisp 上の maxima: 0.90[sec]

*)
```

第20章　対数函数

20.1　Maximaでの対数函数

20.2　対数函数に関連する大域変数

%e_to_numlog

%e_to_numlog:[false]

true であれば,r を有理数,x を式とすると,%eˆ (r*log(x)) が xˆ r に簡易化されます. 尚,radcan 命令もこの変換を行います.

logabs

デフォルト値:[false]

logabs が true の場合,integrate(1/x,x) の様に log が結果に含まれる不定積分の結果は,log(abs(···)) の項を持つものとなります. 但し,logabs が false であれば log(*cdots*) の項を持つものになります.

尚, 定積分の場合は,logabs:true の設定が利用されます. これは, 不定積分の両端点での評価が必要となる事が多い為です.

logarc

デフォルト値: [false]

true であれば, 逆円函数 (逆三角函数), 逆双曲函数を対数函数の形式に変換します.logarc(⟨ 式 ⟩) はこの大域変数の設定無しで, 特定の式に対し, ev を用いた式の再評価を行います.

logconcoeffp

デフォルト値:[false]

logcontract を用いた時に潰される係数を制御します. ここでは一つの引数の函数の名前を設定します. 例えば,sqrt を生成したい場合には,

```
logconcoeffp:'logconfun$
logconfun(m):=featurep(m,integer) or ratnump(m)$
```

とすると, logcontract(1/2*log(x)); から log(sqrt(x)) が得られます.

logexpand

デフォルト値:[true]

true の場合, 自動的に log(a^ b) を b*log(a) に変換します.　all の場合, 自動的に log(a*b) 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) となります (整数 b に対して log(1/b) は常に簡易化される). false の場合はこれらの簡易化は全て実行されません.

lognegint

デフォルト値: [false]

true の場合, 正整数に対し,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.

lognumer

デフォルト値: [false]

true の場合,log の負の浮動小数引数は log に渡される前に, 常にその絶対値に変換されます.

number が又 true であれば,log の負の整数引数もその絶対値に変換されます.

logsimp

デフォルト値: [true]

false の場合,log を含む%e の冪乗の自動簡易化が実行されません.

20.3　対数函数に関連する函数

exp

exp (x) 指数函数. 内部的には%e^ x として表現されています.

log

log (x)

自然対数函数

logcontract 命令は log を含む式を簡易化で”潰す”ものです.

logcontract

logcontract (⟨ 式 ⟩)

再帰的に ⟨ 式 ⟩ を調べ,a1*log(b1) + a2*log(b2) + c の形式の部分式を log(ratsimp(b1^ a1 * b2,a2)) + c に変換します.

```
(c1) 2*(a*log(x) + 2*a*log(y))$
(c2) logcontract(%);
                                      2  4
(d3)                             a log(x  y )
```

declare(n,integer); を実行していれば,logcontract(2*a*n*log(x)); は a*log(x^ (2*n)) とります. この方法で潰される係数は, 上の例で示す様に 2 と n で,featurep(coeff,integer) を満すものです.

利用者は潰される係数を,1 引数の既に入力した函数の名前に対し, 大域変数の logconcoeffp を設定する事で制御が出来ます.

つまり,sqrt を生成したい場合,

```
logconcoeffp:'logconfun$
logconfun(m):=featurep(m,integer) or ratnump(m)$
```

とすれば, logcontract(1/2*log(x)); は log(sqrt(x)) となります.

plog

plog (⟨x⟩)

複素数値の自然対数函数の主分枝を $-\pi < carg(x) <= \pi$ とします.

polarform

polarform (⟨ 式 ⟩)

与えられた ⟨ 式 ⟩ を r*%e^ (%i*theta) の形に変換します. 尚, 多項式が実数係数多項式の場合は入力のままで返され, 函数を含む場合には, その函数が負であれば%e^{%i %pi}をかけたものが返されます.

```
(%i31) polarform((1+%i)^3);
                                        3 %i %pi
                                        --------
                                           4
(%o31)                        2 sqrt(2) %e
(%i32) polarform(x^2+1);
                                     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)
```

20.4　特殊関数について

%j

%j [index](expr) 第 1 種の bessel 関数 (specint での)

%k [index](expr)

第 2 種の bessel 関数 (specint での)

ode2 では定数.

gamalg

dirac の Γ 行列代数プログラム,n 次元の Γ 行列の対角和や演算を行う. loadfile("gam"); を実行すれば macsyma で使える. 限定されたマニュアルは share;gam usage に含まれており,rpintfile(gam,usage,share); を実行すると表示される.

specint

超幾何特殊関数パッケージ hypgeo はまだ開発中である. 現時点では laplace 変換の他に特殊関数やそれらの組み合わせの 0 から inf 迄の積分がある程度である. 因子 exp(-p*var) は明示的に宣言されていなければならない.

構文は以下の通り：specint(exp(-p*var)*expr,var); ここで,var は積分の変数で,expr は特殊関数を含む任意の式 (但し, 危険は承知の上で利用の事) でも良い.

特殊関数の表記は次の通り：

`%j[index](expr)`	第 1 種 bessel 関数
`%k[index](expr)`	第 2 種 bessel 関数
`%i[ ](  )`	変形 bessel 関数
`%he[ ]( )`	エルミート多項式
`%p[ ]( )`	ルジャンドル関数
`%q[ ]( )`	第 2 種ルジャンドル関数
`hstruve[ ]( )`	struve の h 関数
`lstruve[ ]( )`	" の l 関数
`%f[ ]([],[],expr)`	超幾何関数
`gamma()`	Γ 関数
`gammagreek()`	
`gammaincomplete()`	
`slommel`	
`%m[]()`	第 1 種の whittaker 関数
`%w[]()`	第 2 種の whittaker 関数

何が出来るかを確認する為には,demo(hypgeo,demo,share1); を実行せよ.

20.5 特殊関数に関する諸定義

20.5.1 大域変数

poislim

初期値:[5]

三角関数の引数にて, 係数の領域を定める. 初期値の 5 は内部的に $[-2^(5-1)+1, 2^(5-1)]$, 即ち,[-15,16] に関連するが, これを $[-2^(n-1)+1, 2^(n-1)]$ に設定可能である.

gammalim

デフォルト値:[1000000]

整数と有理数引数に対するΓ関数の簡易化を制御する. 引数の絶対値が gammalim よりも大きくなければ, 簡易化が行われる. factlim スイッチは整数引数の gamma の結果の簡易化も同様に制御する事を挙げておく.

20.5.2 函数

airy

airy ($\langle x \rangle$)

実引数 x の airy 関数 ai を返す. ファイル share1;airy fasl に airy 関数 ai(x),bi(x) とそれらの微分 dai(x),dbi(x) を評価するルーチンが含まれている.ai と bi は airy 方程式 diff(y(x),x,2)-x*y(x)=0 を満すものである. 詳細は share;airy usage を読む事.

asymp

share1 ディレクトリにインストールされている feynman 図の漸近的な振舞いを見付けるプログラムのテスト版. より詳細な情報は share1;asymp usage を見よ (漸近的解析関数に関しては asympa を見よ).

asympa

asympa

漸近的解析 - ファイル share1;asympa ¿に漸近的解析向けの簡易化関数があり, そこには広く複素解析や数値解析で用いられている big-0 や little-o 関数も含まれている.batch(”asympa.mc”); を実行せよ (feynman 図の漸近的振舞いに関しては asymp を見よ).

bessel

bessel ($\langle z \rangle, \langle a \rangle$)

複素数 z と実数 a¿0 に対する bessel 関数 j の値を返す. 又, 配列 besselarray は besselarray[i]=j[i+a-entier(a)](z) となる様に設定されている.

beta

beta ($\langle x \rangle, \langle y \rangle$) gamma(x)*gamma(y)/gamma(x+y) と同じである.

gamma

gamma (⟨x⟩) Γ 関数. 正整数に対しては gamma(i)=(i-1)!である.euler-macsheroni 定数に関しては%gamma を見よ. 又,makegamma 関数も見よ.gammalim[1000000](この項参照) はΓ 関数の簡易化を制御する.

intopois

intopois (⟨a⟩) a を poisson の符号化 (* 訳者注; 原文:poisson encoding,poisson 過程の列？*) に変換する.

makefact

makefact (⟨ 式 ⟩) ⟨ 式 ⟩ の中の二項係数, Γ と β 関数を階乗で置き換える.

makegamma

makegamma (⟨ 式 ⟩) ⟨ 式 ⟩ 内の二項係数, 階乗, β 関数をΓ 関数で置き換える.

numfactor

numfactor (⟨ 式 ⟩) 積の ⟨ 式 ⟩ から数値の因数を取り出す. ここで,⟨ 式 ⟩ は単一項でなければならない. 和の中の全ての項の gcd が必要であれば,content 関数が使える.

```
(c1) gamma(7/2);
(d1)                15 sqrt(%pi)
                    ------------
                         8
(c2) numfactor(%)
                     15
(d2)                 --
                      8
```

outofpois

outofpois (⟨a⟩)

⟨a⟩ を poisson の符号化から一般表現に変換する.a が poisson の形式で無い場合は変換される, つまり,outofpois(intopois(a)) の結果の様なものになる. この関数は sin や cos の様な特定の型の冪乗の和に対し, その様な正規な簡易化を行ったものとなる.

poisdiff

poisdiff (⟨a⟩, ⟨b⟩) ⟨a⟩ を ⟨b⟩ で微分する.⟨b⟩ は三角関数の引数か, 係数の中だけに表われるものでなければならない.

poisexpt

poisexpt ($\langle a\rangle, \langle b\rangle$) $\langle b\rangle$ (正の整数) は機能的に intopois(ab̂) と同一である.

poisint

poisint ($\langle a\rangle, \langle b\rangle$)

(poisdiff と似た) 制限された積分を行う.$\langle b\rangle$ の非周期的項は $\langle b\rangle$ が三角関数の引数に含まれていれば落される.

poismap

poismap ($\langle$ 級数 $\rangle, \langle sinfn\rangle, < cosfn\rangle$) 与えられた poisson 列の sine 項と cosine 項に関数 $\langle sinfn\rangle$ と $\langle cosfn\rangle$ を写す. $\langle sinfn\rangle$ と $\langle cosfn\rangle$ は二つの引数を持つ関数で, 数列表現で係数と三角関数項が交互に現われるものである.

poisplus

poisplus ($\langle a\rangle, \langle b\rangle$) intopois(a+b) と機能的には同一である.

poissimp

poissimp ($\langle a\rangle$)

一般表現 $\langle a\rangle$ を poisson 列に変換する.

特殊記号 poisson

記号 /p/の後ろに poisson 列表現の行ラベルが続く.

poissubst

poissubst ($\langle a\rangle, \langle b\rangle, \langle c\rangle$) $\langle c\rangle$ の中の $\langle b\rangle$ に $\langle a\rangle$ を代入する.$\langle c\rangle$ は poisson 列である.

1. $\langle b\rangle$ が変数 u,v,w,x,y, 又は z であれば,$\langle a\rangle$ はこれらの変数の線型な式 (例えば,6*u+4*v) でなければならない.

2. $\langle b\rangle$ がこれらの変数以外の場合,$\langle a\rangle$ はこれらの変数を含まず, 更に,sin や cos を含まないものでなければならない.

poissubst(a,b,c,d,n) は代入の特殊な種類で,a と b に対して上の (1) の種類で作用するが, ここで d は poisson 列で,a+d を c の中の b に代入した結果となる様に,cos(d) と sin(d) を n 次まで展開する. この手法では d が小さな助変数の項で展開されている. 例えば,poissubst(u,v,cos(v),e,3) は $cos(u)*(1-e^2/2)-sin(u)*(e-e^3/6)$ を返す.

poistimes

poistimes (a, b)

機能的に intopois(a*b) と同一である.

poistrim

poistrim () 指定された関数名で,poisson 積の間に (利用者が定義していれば) 実行される.6 個の引数を持つ述語関数であり, その引数は項にある u,v,..,z の係数である.(項の係数に対し)poistrim が true となる項は積の間, 消去される.

printpois

printpois (a) poisson 列を読める形式で出力する. 一般的には outofpois と使い, 必要があれば, a を最初に poisson 符号に変換する.

psi

psi (x) log(gamma(x)) の導関数. 現時点で macsyma は psi に対し, 数値的な評価を行う機能を持っていない.psi[n](x) の情報に関しては,polygamma を見よ.

第21章　極限の計算

21.1　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 $sinx$ の原点周りの級数展開を考えると判り易くなります.

$sin(x) = \sum_{i=0}(-1)^i \frac{x^{2i+1}}{(2i+1)!}$ となり, これを x で割ってしまうと,$\frac{sinx}{x} = 1 + x$(冪級数) となります. 以上から,x を 0 に近づけると 1 になる事が判ります.

Maxima で試してみましょう.Maxima には limit 函数があり, この函数は limit(⟨ 函数 ⟩,⟨ 変数 ⟩,⟨ 値 ⟩) で極限の計算が行えます.

```
(%i39) limit(sin(x)/x,x,0);
(%o39)                                1
(%i40) plot2d(sin(x)/x,[x,-50,50]);
```

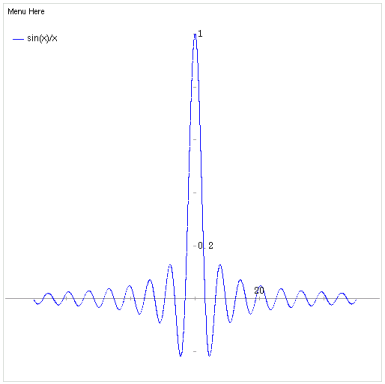

図 21.1: sin(x)/x のグラフ

極限はこの様な計算を行いますが, この近付けるという操作には方向を考えなければなりません. 例えば,$\frac{1}{x}$ はどうでしょうか. この函数は $x > 0$ なら正で,$x < 0$ で負になっており,$x = 0$ が不連続点になっています.

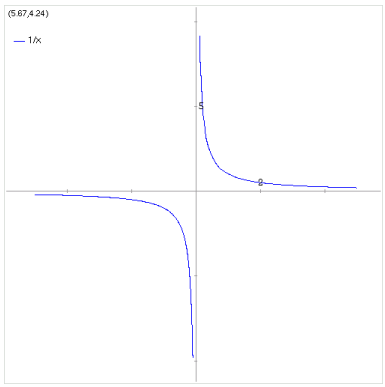

図 21.2: 1/x のグラフ

```
(%i73) limit(1/x,x,0);
(%o73)                                 und
(%i74) limit(1/x,x,0,plus);
(%o74)                                 inf
(%i75) limit(1/x,x,0,minus);
(%o75)                                minf
```

この様に右側から近付けた場合には正の無限大, 左側から近付けた場合には負の無限大となっていますね. この様に左右の極限が異なるので, limit(1/x,x,0) の結果は und となっています.

limit 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 infity, 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
```

21.2　極限に関連する大域変数

lhospitallim

デフォルト値:[4]

limit で用いられる l'Hospital 則の適用回数の最大値. これは limit(cot(x)/csc(x),x,0) の様な場合に無限ループに陥いる事を防ぐ為のものです.

limsubst

デフォルト値:[false]

limsubst は limit が未知の形式に代入を行う事を防ぎます. これは limit(f(n)/f(n+1),n,inf) の様なもので,1 となるバグを避ける為です. limsubst が true であれば, この様な代入が許容されます.

tlimswitch

デフォルト値:[false]

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.

21.3 極限に関する函数

limit

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

limit 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 ⟩ 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, 省略しても構いません. 通常, 省略した場合は両側極限が計算されます. 尚, 原点の極限計算であれば 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. その為,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.

limit は inf-1 の様な定数式の簡易化で利用される事が多い為, 引数が唯一, 例えば,limit(inf-1) の場合でも limit を使っても構いません.

tlimit

tlimit (⟨< 式 ⟩,⟨ 変数 ⟩,⟨ 値 ⟩) tlimit (⟨< 式 ⟩,⟨ 変数 ⟩,⟨ 値 ⟩,⟨ 方向 ⟩)

tlimswitch を true にした limit 関数です.

第22章　微分

22.1　微分に関連する大域変数

dependencies

デフォルト値:[]

依存性を持つアトムのリストです. 依存性は depends, 或いは gradef 函数で設定出来ます.

derivabbrev

デフォルト値:[false]

true であれば, 微分は下添字として表示されます. この大域変数は単純に微分の表示にのみに影響します.

```
(c1) diff(f(x),x);

                                    d
(d1)                                -- (f(x))
                                    dx

(c2) derivabbrev:true;

(d2)                                 true
(c3) diff(f(x),x);

(d3)                                 f(x)
                                          x
```

derivsubst

デフォルト値:[false]

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 で置換えれば,$\frac{dx}{dt}$ となります. derivsubst は名詞形の微分を含む式に対し, この様な置換を行うかどうかを制御するものです.

false の場合,susbst 函数による置換は出来ませんが, true の場合は許容します.

```
(%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
```

gradefs

デフォルト値:[]

gradef 函数で勾配を与えた函数のリストです.

22.1.1 微分に関係する函数

antidiff と antid

antidiff($\langle g\rangle, \langle x\rangle, \langle u(x)\rangle$) antid ($\langle g\rangle, \langle x\rangle, \langle u(x)\rangle$)

antidiff は任意の未割当函数とその導函数を含む式の積分を評価します. 例えば,antidiff(g,x,u(x)) とすると,g は函数 u(x) とその微分を含む式の積分が求めるものになります.

x による antidiff を利用する為には load(antid) を予め実行しておきます.

函数 nonzeroandfreeof と linear も antid と同様に antid は antidiff と同様ですが, 二つの成分のリストを返す点で異なます. 又, そのリストの最初の成分は式を積分したもので, 二番目の成分は残りの微分不能な成分になります.

```
(%i1) load ("antid")$
(%i2) expr: exp (z(x)) * diff (z(x), x) * y(x);
                              z(x)  d
(%o2)                 y(x) %e     (-- (z(x)))
                                    dx
(%i3) a1: antid (expr, x, z(x));
                        z(x)      z(x)  d
```

```
(%o3)          [y(x) %e    , - %e     (-- (y(x)))]
                                       dx
(%i4) a2: antidiff (expr, x, z(x));
                              /
                     z(x)     [   z(x)  d
(%o4)         y(x) %e     - i %e     (-- (y(x))) dx
                              ]         dx
                              /
(%i5) a2 - (first (a1) + 'integrate (second (a1), x));
(%o5)                              0
(%i6) antid (expr, x, y(x));
                               z(x)  d
(%o6)             [0, y(x) %e     (-- (z(x)))]
                                    dx
(%i7) antidiff (expr, x, y(x));
                  /
                  [        z(x)  d
(%o7)             i y(x) %e     (-- (z(x))) dx
                  ]              dx
                  /
```

cartan

微分形式の外積は Elie Cartan により発展した微分幾何学の基本的な道具で, 偏微分方程式論でも重要な適用事例を持っています. 現行の Maxima での実装は F.B.Estabrook と W.H.Wahlquist によるものです. プログラムは自己説明的なもので,batch("cartan"); を実行すれば使えます.

delta

delta (t)

Dirac のデルタ函数です. なお,laplace 函数のみがデルタ函数を認識しています.

```
(%i38) laplace(delta(t-a)*sin(b*t),t,s);
Is  a  positive, negative, or zero?

pos;
                                          - a s
(%o38)                          sin(a b) %e
```

この例では変数 a に関して何らの仮定や割り当てが無い為に, "Is a positive, negative, or zero?" と Maxima が尋ねています. assume を用いて a⟨0 と仮定した場合を以下に示します.

```
(%i39) assume(a<0);
```

```
(%o39)                              [a < 0]
(%i40) laplace(delta(t-a)*sin(b*t),t,s);
(%o40)                                0
```

depends

depends (⟨ 函数リスト$_1$⟩, ⟨ 変数リスト$_1$⟩, ··· , ⟨ 函数リスト$_n$⟩, ⟨ 変数リスト$_n$⟩)
diff で用いる変数に函数の従属性を宣言します.

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

函数の変数に対する従属性は deoendencies に登録された函数の情報から調べる事が出来ます.

次の例では,f と g が x と y に,r と s が u,v と w に依存し,u が t に従属する事を diff に報せる為に,depends([f,g],[x,y],[r,s],[u,v,w],u,t) としています

```
(%i11) dependencies;
(%o11)                               []
(%i12) depends([f,g],[x,y],[r,s],[u,v,w],u,t);
(%o12)         [f(x, y), g(x, y), r(u, v, w), s(u, v, w), u(t)]
(%i13) dependencies;
(%o13)         [f(x, y), g(x, y), r(u, v, w), s(u, v, w), u(t)]
(%i14) diff(r.s,u);
```

```
                                dr           ds
(%o14)                          -- . s + r . --
                                du           du
(%i15) diff(r.s,t);
                                 dr du       ds du
(%o15)                       s . -- -- + r . -- --
                                 du dt       du dt
```

この例では,depends を実行する事で,dependencies に depends で従属性を宣言した函数が変数と一緒にリストに加えられている事に注目して下さい. 又, 微分では合成函数の微分も出来ます.

depends で設定した従属性は,remove 函数を使って削除する事が出来ます. 上の例の (%i11) で宣言した r の u に対する従属性を削除したければ, remove(r,dependency) と入力します.

```
(%i16)  remove(r,dependency);
(%o16)                              done
(%i17) ''%i15;
                                    ds du
(%o17)                           r . -- --
                                    du dt
```

diff は dependencies に設定された情報を用います. 積分,laplace 変換等での引数は, integrate(f(x),x) の様に, 従属性を函数内部ではっきりと与えなければなりません.

derivdegree

derivdegree (⟨ 式 ⟩, ⟨ 従属変数 ⟩, ⟨ 独立変数 ⟩)

⟨ 式 ⟩ 中の ⟨ 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.

```
(%i47)  'diff(y,x,2)+'diff(y,z,3)*2+'diff(y,x)*x^2$
(%i48) derivdegree(%,y,x);
(%o48)                                2
```

この様に名詞型の入力に対し, 最高階数を探します. 尚, 微分を名詞型で入力していなければ評価された式で返される為, 希望する値が返ってこないので注意が必要です.

derivlist

derivlist (⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_k$⟩)

ev 命令内部で,derivlist で指定した変数に対してのみ微分を行います.

diff

diff (⟨ 式 ⟩, ⟨ 変数$_1$⟩, ⟨ 階数$_1$⟩, ··· , ⟨ 変数$_n$⟩, ⟨ 階数$_n$⟩)

diff は各 ⟨ 変数$_i$⟩ で各々⟨ 階数$_i$⟩ の ⟨ 式 ⟩ の微分を行います. 1 変数による 1 階微分の場合は,diff(⟨ 式 ⟩, ⟨ 変数 ⟩) の書式で構いません.

函数の名詞型が要求される場合 (例えば, 微分方程式を記述する時), 'diff を用いなければなりません. この場合, デフォルトの表示は二次元的 (プリティプリント) 書式になります. ここで,deriveabbrev が true であれば, 微分は添字で表示されます.

diff(⟨ 式 ⟩) は全微分を与えます. 即ち,⟨ 式 ⟩ の各変数に対する微分と, 各変数の函数の del との積の和になります. 例えば, diff(sin(x*y)); を実行すると x cos(x y) del(y) + y cos(x y) del(x) が返されます.

```
(%i49) diff(exp(f(x)),x,2);
                              2
                    f(x)  d                f(x)  d          2
(%o49)             %e     (--- (f(x))) + %e     (-- (f(x)))
                             2                    dx
                            dx
(%i50)  derivabbrev:true$
(%i51) 'integrate(f(x,y),y,g(x),h(x));
                                h(x)
                               /
                               [
(%o51)                         I     f(x, y) dy
                               ]
                               /
                                g(x)
(%i52) diff(%,x);
             h(x)
            /
            [
(%o52)      I     f(x, y)  dy + f(x, h(x)) h(x)  - f(x, g(x)) g(x)
            ]            x                     x                  x
            /
             g(x)
```

テンソルパッケージ向けには, 以下の改変が含まれています.

1. ⟨ 式 ⟩ の添字された任意のオブジェクトの微分は追加引数として加えられた ⟨ 変数$_i$⟩ を持ちます. 全ての微分の添字は蓄えられます.
2. ⟨ 変数$_i$⟩ は 1 から大域変数 dimension[デフォルト値:4] までの整数が指定可能です. これでリスト coordinates の i 番目の要素で微分が実行され, その coordinates は座標系の名前のリス

ト, 例えば,[x,y,z,t] の様に, 設定されていなければなりません.coordinates にアトムの変数が設定されていれば, 変数は vi によって添字され, 微分の変数として利用されます. これは座標系の名前か,$x[1], x[2], \cdots$ の様に添字された名前の配列が使えます.coordinates が指定された値を持っていなければ, 変数は上述の 1) として扱われます.

dscalar

dscalar (⟨ 函数 ⟩)

スカラ値函数に対し, スカラの d'Alembert の演算子を作用させます.

express

express (⟨ 式 ⟩)

偏微分の項の名詞型の微分を展開します.express は grad,div,curl や laplacian といった演算子を認識します.express は他に外積 も認識します. 但し,express で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.

この函数を利用する為には予め load("vect") を実行します.

```
(%i1) load(vect);
(%o1)         /usr/local/share/maxima/5.9.2/share/vector/vect.mac
(%i2) e1:laplacian(x^2*y^2*z^2);
                                       2  2  2
(%o2)                        laplacian (x  y  z )
(%i3) express(e1);
               2                2                2
              d     2  2  2    d     2  2  2    d     2  2  2
(%o3)         --- (x  y  z ) + --- (x  y  z ) + --- (x  y  z )
                2                2                2
              dz               dy               dx
(%i4) ev(%,'diff);
                             2  2      2  2      2  2
(%o4)                     2 y  z  + 2 x  z  + 2 x  y
(%i5) v1:[x1,x2,x3]~[y1,y2,y3];
(%o5)                     [x1, x2, x3] ~ [y1, y2, y3]
(%i6) express(v1);
(%o6)           [x2 y3 - x3 y2, x3 y1 - x1 y3, x1 y2 - x2 y1]
```

gradef

gradef($\langle f(x_1, \cdots, x_m)\rangle, \langle g_1\rangle, \cdots, \langle g_n\rangle$)

gradef($\langle f\rangle, \langle x\rangle, \langle$ 式 $\rangle$)

函数 $\langle f\rangle$ の n 個の引数に対する微分を, $\frac{d<f>}{dx_i} =< g_i\rangle$ で定めます.

もしも変数の総数数 mn 個の勾配 n よりも小さい場合, 最初の $\langle f \rangle$ の i 番目の引数が参照されます.x_i は函数定義で用いるダミー変数と同類で, 函数 $\langle f \rangle$ 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 の引数は $\langle g \rangle$ が定義された函数ならば, その函数が呼出され, 結果が用いられます. 勾配は函数が第一階微分を除いて正確に判らない場合で, より高階の微分を得たい時に必要となります.

gradef($\langle f \rangle$, $\langle x \rangle$, ⟨ 式 ⟩) は $\langle x \rangle$ による ⟨ 函数 ⟩ の微分が ⟨ 式 ⟩ となる事を宣言します. この際に, 自動的に depends(⟨ 函数 ⟩, $\langle x \rangle$) が実行されます.

gradef 函数は Maxima の既に定義された函数の微分を再定義する事にも使えます. 例えば,gradef(sin(x),sqrt(1-sin(x)^ 2)) の様に出来ます. 但し, 添字された函数に gradef 函数は使えません.

gradef で勾配を定義すると, 大域変数 gradefs にその函数の名前が蓄えられます.

gradef は函数に対する属性を設定する函数の為, 値は printprops で表示出来ます. この場合, 属性は gradef となります. 又, アトムに対して宣言した場合には, 属性は atomgrad となります.

ilt

ilt (⟨ 式 ⟩, ⟨ 旧変数 ⟩, ⟨ 新変数 ⟩)

⟨ 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 の計算を効率的に行う為には有理式の展開を予め実行しておくと良いでしょう.

laplace と ilt の両方を solve や linsolve と使うと, 単変数の微分方程式か畳込み積分方程式を解く事が出来ます.

```
(%i11) 'integrate(sinh(a*x)*f(t-x),x,0,t)+b*f(t)=t^2;
                     t
                     /
                     [                                    2
(%o11)               I  f(t - x) sinh(a x) dx + b f(t) = t
                     ]
                     /
                      0
(%i12) laplace(%,t,s);
                                    a laplace(f(t), t, s)   2
(%o12)        b laplace(f(t), t, s) + --------------------- = --
                                          2    2           3
                                         s  - a           s
(%i13)  linsolve([%],['laplace(f(t),t,s)]);
                                           2      2
                                        2 s  - 2 a
(%o13)           [laplace(f(t), t, s) = --------------------]
                                           5         2     3
                                      b s  + (a - a  b) s
(%i14) ilt(rhs(first(%)),s,t);
```

```
Is  a b (a b - 1)  positive, negative, or zero?

pos;
                  sqrt(a b (a b - 1)) t
          2 cosh(----------------------)       2
                             b              a t             2
(%o14)  - ---------------------------- + ------- + ------------------
                 3  2      2              a b - 1    3  2      2
                a  b  - 2 a  b + a                  a  b  - 2 a  b + a
```

laplace

laplace (⟨式⟩,⟨旧変数⟩,⟨新変数⟩)

⟨旧変数⟩と⟨新変数⟩に対する⟨式⟩の Laplace 変換を計算します. 逆 laplace 変換は ilt です.

⟨式⟩は多項式に函数 exp,log,sin,cos,sinh,cosh と erf 函数を含むもの, atvalue の従属変数が使われている定数係数の線形微分方程式でも構いません.

初期条件は零で指定されていなければならないので, 他の一般解の何処かに押込める境界条件があれば, その境界条件に対して一般解を求めて値を代入して定数消去が出来ます.

⟨式⟩にはは畳込み (convolution integral) を含んでいても構いません. laplace が適切に動作する為に, 函数の従属性をはっきりと表示していなければなりません. つまり, 函数 f が x と y に従属しているのであれば, laplace('diff(f(x,y),x),x,s) の様に f が現れる場合は何時でも f(x,y) と記述する必要があります.

尚,laplace は depends 命令で設定される dependecies の影響を受けません.

```
(%i106)  laplace(%e^(2*t+a)*sin(t)*t,t,s);
                                    a
                                 %e  (2 s - 4)
(%o106)                          ---------------
                                   2           2
                                  (s  - 4 s + 5)

(%i107) ilt(%,s,x);
                                    2 x + a
(%o107)                        x %e        sin(x)
```

undiff

undiff (⟨式⟩)

微分が実行されれば, 添字されたオブジェクトの全ての微分を除いた⟨式⟩と同値な式で, その微分の引数が添字付けられたものを生成します. 即ち k,diff 函数の名詞型で置き換えたものに相当します.

微分され, 添字付けられた式をある函数定義で置き換え,ev(...,diff) を指定して微分を実行したい場合には便利です.

第23章　積分

23.1　Maximaでの積分について

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.

但し. 積分の計算は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で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 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 integration 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, 数式処理一般で記号積分の結果は面倒でも確認した方が良いのです.

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. 並びも勿論異なるので, 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. これが数値の場合は, 式の並びは無関係で, 函数の値だけしか見ない為, 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, 式を展開したり,factor で因子分解する等で, 式を変形させて積分したもので結果を照合するのも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
```

この様に一見すると正しい結果が出ている様ですね. でも間違っています. 何故なら被積分函数は滑かな連続函数ですが, 結果の方は不連続な函数です.

この様な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.

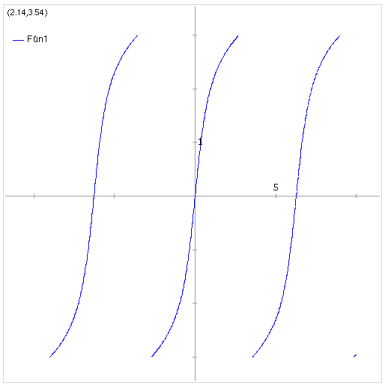

図 23.1: $2atan(\frac{3\sin x}{\cos x+1})$ のグラフ

グラフを描いて見るのも非常に有効な手段です.

Maxima は積分処理の為の函数を幾つか持っています. その中でも,integrate 函数は最も使われるものです.antid は指定していない関数の操作 (そしてその微分も勿論の事) を行えます. この integrate で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. 但し. この函数は非常に曲者で,lntegreta 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.

ここで,defint の処理を簡単に説明しておきます. 通常は integrate(lisp 内部では$integrate) で用いられている sinint 函数を内部で呼出し, 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 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 や integrate による定積分の計算で, 下限や上限に記号や式が含まれている場合, その正負を尋ねてくる事があります. この場合, 正であれば pos; , 負であれば neg; , 零であれば zero; と入

力します. 更に,assume を用いて正負の指定を予め行っていれば,Maxima はこの様な質問を行ないません.

```
(%i24) integrate(sqrt(2*x-x^2),x,0,a);
Is  a  positive, negative, or zero?

pos;
                                                  2
                  asin(a - 1) + (a - 1) sqrt(2 a - a )   %pi
(%o24)            ------------------------------------ + ---
                                  2                       4
(%i25) assume(a>0 and a<1);
(%o25)                          [a > 0, a < 1]
(%i26) integrate(sqrt(2*x-x^2),x,0,a);
                                    2
                  (a - 1) sqrt(2 a - a ) - asin(1 - a)   %pi
(%o26)            ------------------------------------ + ---
                                  2                       4
```

数値計算の手法を用いる場合には,romberg の他に quanc8 があります. quanc8 の方が計算速度,精度と安定性で romberg に勝っている事が多いのですが, romberg 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 が精度で勝り,$\exp -x\sin x$ の積分では計算速度で romberg が勝っています.

```
(%i30) load(qq);
(%o30)        /usr/local/share/maxima/5.9.2/share/numeric/qq.lisp
(%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.

全般的に 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) の様な未知の関数の積分は扱いません.

23.2 大域変数

erfflag

デフォルト値:[true]

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

integration_constant_counter

方程式の不定積分を行うと, 式に導入される積分定数 (integrationconstant) が導入されます. この積分定数は順番が付いており, この順番が integration_constant_counter に記録されています.

```
(%i1) integration_constant_counter;
(%o1)                                0
(%i2) integrate(x=0,x);
                      2                          /
                     x                           [
(%o2)                -- = integrationconstant1 + I 0 dx
                     2                           ]
                                                 /
(%i3) integration_constant_counter;
(%o3)                                1
(%i4) integrate(exp(x)=0,x);
                                                  /
                       x                          [
(%o4)                 %e  = integrationconstant2 + I 0 dx
                                                  ]
                                                  /
(%i5) integration_constant_counter;
(%o5)                                2
```

rombergabss

デフォルト値:[0.0]

romberg によって生成された値の列を y[0],y[1],y[2] 等とすると, romberg は n 回の反復の後に,(大雑把に言えば)(abs(y[n]-y[n-1])≤rombergabs か abs(y[n]-y[n-1])/(y[n]=0.0 であれば 1.0, それ以外は y[n]≥rombergtol).

true であれば,0.0b0 を返します. その為,rombergabs が 0.0 であれば相対誤差の検証が得られます. 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し. 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.

integral(exp(-x),x,0,50) を数値的に 10000000 分の 1 の相対精度で計算したいとします.n をカウンターとして, どれだけの関数評価が必要とされたかを見られる様に次の関数を定義します.

```
(%i7) f(x):=(mode_declare(n,integer,x,float),n:n+1,exp(-x))$
(%i8) translate(f)$

Warning-> n is an undefined global variable.
(%i9) block([rombergtol:1.e-6,romberabs:0.],n:0,romberg(f,0,50));
(%o9)                          1.000000000488271
(%i10) n;
(%o10)                                 257
```

この例では,n=257 となっているので, 函数評価は 257 回となります. それから, 先ず最初に integral(exp(-x),x,0,10) を実行して rombergabs を 1.e-6*(この部分積分) に設定するという風に積分を実行します.

```
(%i11) block([rombergtol:1.e-6,rombergabs:0.,sum:0.],
  n:0,sum:romberg(f,0,10),rombergabs:sum*rombergtol,rombergtol:0.,
      sum+romberg(f,10,50));
(%o11)                         1.000000001234793
(%i12) n;
(%o12)                                 130
```

二番目の手法は二倍近く俊敏 (257 対 130) な事になります.

rombergit

デフォルト値:[11]

romberg 積分命令の精度は大域変数の rombergtol と rombergit で支配されます. romberg は, 隣り合った近似解での相対差が rombergtol よりも小さければ値を返します. 諦める前に刻幅の romgergit を半分にして試行します.

rombergmin

デフォルト値:[0]

romberg による関数評価の最小数を制御します.romberg はその第一引数を少なくとも $2\hat{}$(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時に, 周期的関数の積分に対して便利です.

rombergtol

デフォルト値:[1.e-4]

romberg 積分命令の精度は大域変数 rombergtol と rombergit が支配します．romberg が結果を返すのは, 隣り合った近似解の相対差が rombergtol 以下になった時です. 尚, 諦める前に刻幅の rombergit を半分にして試します.

23.3 積分に関連する函数

changevar

changevar (⟨ 式 ⟩, ⟨$f(x,y)$⟩, ⟨y⟩, ⟨x⟩) ⟨ 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.

```
(%i1) 'integrate(%e^sqrt(a*y),y,0,4);
                                4
                               /
                               [    sqrt(a y)
(%o1)                          I  %e          dy
                               ]
                               /
                                0
(%i2) changevar(%,y-z^2/a,z,y);
Is  a  positive, negative, or zero?

pos;
                              0
                             /
                             [                abs(z)
                           2 I            z %e       dz
                             ]
                             /
                              - 2 sqrt(a)
(%o2)                  - ----------------------------
                                       a
(%i3) ev(%,'risch);
Is  a  positive, negative, or zero?

pos;
                                    2 sqrt(a)     2 sqrt(a)
                  2 (- 2 sqrt(a) %e          + %e          - 1)
(%o3)           - ------------------------------------------
                                      a
```

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, 次数の高い関数には出来ません.

```
(%i43) sum(a[i]*x^(i-2),i,0,inf);
```

```
                                inf
                                ====
                                \          i - 2
(%o43)                           >    a  x
                                /      i
                                ====
                                i = 0
(%i44) changevar(%,i-2-n,n,i);
                               inf
                               ====
                               \                n
(%o44)                          >      a       x
                               /        n + 2
                               ====
                               n = - 2
```

dblint

dblint ('f,'r,'s,a,b)

Maxima のトップレベルで記述されたもので, 機械語コードに変換・翻訳された二重積分函数です. このパッケージの利用は load(dblint); を用います. 計算で x と y の両方向で simpson 則を用います.

関数 f(x,y) は二変数関数で translate で変換されたものか,compile で翻訳されたものでなければなりません. 更に,r(x) と s(x) も各々が 1 変数関数で変換か翻訳された関数で,a と b は浮動点小数でなければなりません. この函数は二つの大域変数を持ち, それらは x と y の区間の分割数:dblint_x,dblint_y を決めます. その両方の初期値は 10 で, 他の整数値 (2*dblint_x+1 点が x 方向に計算され,y 方向は 2*dblint_y+1 となります) と独立して変更する事か可能です.

この函数は x 軸を分割し,x の各値に対して最初に r(x) と s(x) を計算します. そして, r(x) と s(x) の間の y 軸を分割し,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. しかし, 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 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, 様々な y 積分で異なった刻幅を持つ切捨てによって重大な誤差が発生するかもしれません.

領域での収束性を改善する為に dblint_x と dblint_y を増す事も可能であるが, 計算時間が増大します. 関数値は保存されず, その関数が非常に多くの時間を費すものであれば, 何かを変更すると再計算で待つ必要があります. 関数 f,r と s の両方は dblint を呼出す為に, 最初に translate で変換されたものか compile で翻訳されたものでなければなりません. これは多くの場合で翻訳されたコードで最大の速度向上を目指した結果です

defint

defint (⟨ 式 ⟩,⟨ 変数 ⟩,⟨ 下限 ⟩,⟨ 上限 ⟩)

定積分を実行する函数です. 内部的に integrate を利用しており, 原始函数を求めると, 単純に ⟨ 上限 ⟩ と ⟨ 下限 ⟩ の値を代入したものの差を取るだけです.

この処理では区間 (⟨ 下限 ⟩,⟨ 上限 ⟩) の ⟨ 式 ⟩ の極の判定を一切行わないので注意が必要です. その為, 数値的な計算手法を用いたければ defint よりも romberg を利用して下さい.

erf

erf (x)

error 関数. この関数の微分は $2exp(-x^2)/sqrt(\pi)$ になります.

integrate

integrate (⟨ 式 ⟩,⟨ 変数 ⟩)

integrate(⟨ 式 ⟩,⟨ 変数 ⟩,⟨ 下限 ⟩,⟨ 上限 ⟩)

⟨ 変数 ⟩ に対する ⟨ 式 ⟩ の積分を行います. 原始函数の計算が出来ない場合, 名詞型の積分式を返します.

integrate では三段階の大きな処理の流れがあります.

1. integrate は被積分関数が f(g(x))*diff(g(x),x) の形式であるかを, 幾つかの部分式 (例えば, 上の場合は g(x)) の微分で被積分関数が割れるかどうか検査します. 割切れると,f を積分表で探し,f の積分で g(x) を x の代りに代入します. ここで微分を取る際に勾配を用います. 未知関数が被積分関数に現われれば, この段階で消去されていなれければなりません. そうでなければ,integrate は被積分関数の名詞型を返します.

2. integrate は被積分関数が特殊な手法, 例えば, 三角関数の代入と云った特定の手法が使える形式に適合するかどうかを試みます.

3. 最初の二段階が失敗すると risch のアルゴリズムが利用されます. 関数の相互関係は integrate が正確に動作する様に厳密に表現されている必要があります.

integrate で risch の積分を強制的に行わせる事が可能である. これは以下の様に ev 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)
```

integrate は depends 函数で設定される大域変数 dependencies の影響を受けません.

integrate は定積分の計算も可能です. この場合は defint と同じ引数を取ります. 即ち,integrate(⟨式⟩,⟨変数⟩,⟨下限⟩,⟨上限⟩) は ⟨変数⟩ に対する ⟨下限⟩ から ⟨上限⟩ までの ⟨式⟩ の定積分を行います. この計算では, integrate(⟨式⟩,⟨変数⟩) で原始函数を求めて ⟨上限⟩ と ⟨下限⟩ の値を求めた原始函数に代入して, 単純にその差を取るという荒っぽい方法です. その為, 定積分を行う区間内に極が存在しても, その検証は一切行われません. この定積分では, 最初の integrate(⟨式⟩,⟨変数⟩) の計算に失敗すると名詞型を返します.

広義の定積分では, 正の無限大に関して inf を使い, 負の無限大には minf が使えます. この inf と minf に関しては,-inf や-minf は各々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.

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) が望ましいければ, 名詞型の'integrate を利用します. この際に, 積分形式の文字による表示が不要であれば, 大域変数 display2d を false にすると, 一行で表示されます.

Maxima で積分が出来ない事が, 直ちに常にその積分が閉形式で存在しない事を意味しません. 以下に示す例で integrate は名詞型を返すが, その原始関数を簡単に見つける事が可能です. 例えば,$x^3+x+1=0$ の根で被積分関数を $\frac{1}{7}((x-\alpha)(x-\beta)(x-\gamma))$ の形式に書換えると計算が出来ます. ここで α,β,γ は方程式の根になります.Maxima はこの同値な形式を積分を行いますが, その積分は非常に複雑なものです.

ldefint

ldefint (⟨式⟩,⟨変数⟩,⟨下限⟩,⟨上限⟩)

⟨変数⟩ の ⟨上限⟩ と ⟨下限⟩ に関する ⟨式⟩ の不定積分に対し,limit を用いた評価を行い,⟨式⟩ の定積分を計算します.

`minf` と `inf` の扱いは注意しなければならない. 特に`,-inf` や`-minf` の様に符号を付けて用いると本来の結果と同値でも無意味な結果を得るので注意が必要である.

```
(c37) ldefint(exp(-x)*sin(x),x,0,-minf);
                      minf                  minf
                     %e     sin(- minf)    %e     cos(- minf)   1
(d37)              - ------------------ - ------------------ + -
                            2                     2            2
(c38) ldefint(exp(-x)*sin(x),x,0,inf);
                                      1
(d38)                                 -
                                      2
(c39) ldefint(exp(-x)*sin(x),x,minf,0);
```

```
                                  - x            - x
                    limit     - %e    sin(x) - %e    cos(x)
                    x -> minf                                 1
(d39)             - ------------------------------------- - -
                                        2                     2
(c40) ldefint(exp(-x)*sin(x),x,-inf,0);
                          inf              inf
                        %e    sin(inf)   %e    cos(inf)   1
(d40)                 - -------------- + -------------- - -
                              2                2          2
```

尚,ldefint は内部的には極の判別を一切行わずに, 記号積分した結果に上限と下限を limit で代入するだけである. 一応, 右極限と左極限を下限と上限で取る様にしているが, 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関数の場合, 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がある.

尚,defint や integrate で定積分を行う場合は区間内での極限の判別も行っています. 而し,ldefint では sinint 関数を用いて機械的に記号積分を行うだけである. その為, 関数をいきなり ldefint を用いて積分したり, 結果の検証を省く事は薦められない.

以下に安易な例を示す.

```
(c1) ldefint(1/x^2,x,-1,1);
(d1)                                  - 2
(c2) ldefint(1/x^2,x,0,1);
                                        1
(d2)                             (limit  -) - 1
                                 x -> 0 x
(c3) ldefint(1/x^2,x,-1,0);
                                          1
(d3)                           - (limit  -) - 1
                                   x -> 0 x
(c4) d2+d3;
(d4)                                  - 2
(c5) defint(1/x^2,x,-1,1);
integral is divergent
 -- an error.  quitting.  to debug this try debugmode(true);)
(c6) integrate(1/x^2,x,-1,1);
```

```
integral is divergent
 -- an error.  quitting.  to debug this try debugmode(true);)
```

この例で示す様に,d1 の ldefint の結果は-2 となっています. これは maxima では 1/x^2 をパターンマッチングで安易に記号積分し, 区間の上限と下限の極限を取っている為である. 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. この様に,ldefint を用いる場合には被積分関数の連続性に関して defint や integrate 以上に注意が必要となる.

尚,ldefint には zeroa,zerob が使える. 各々が 0 の右極限と 0 の左極限を現わす.

```
(c10) ldefint(1/x^2,x,zeroa,1);
                                          1
(d10)                             (limit   -) - 1
                                   x -> 0+ x
(c11) ldefint(1/x^2,x,zerob,-1);
                                          1
(d11)                             (limit   -) + 1
                                   x -> 0- x
```

但し,1+'zeroa の様な使い方は出来ないので注意します.

尚,ldefint は defint と同様に積分で risch 積分を用いる事が出来ない. 但し,defint と同様の修正を加えれば,ev を用いて risch 積分を用いる事が可能である.

potential

potential (⟨ 勾配 ⟩)
大域変数を用いる計算.

```
potentialzeroloc[0]
```

これは nonlist かその形式でなければならない.

```
[indeterminatej=expressionj, indeterminatek=expressionk, ...]
```

前者は後者の全ての右手側に対する非リスト式に同値であり, 指定された右手側は定積分の下限として用いられる. 積分の成功はそれらの変数と順序に依存します. potentialzeroloc はデフォルト値として 0 が設定されています.

qq

これは load("qq"); で読み込まれます.3 か 4 個の引数を取る事が可能な関数 quan8 を含みます.

3 引数版では quanc8(' 関数名,lo,hi) にて最初の引数で指定された関数の積分を区間 lo から hi で計算します. 第一引数は' 関数名の様に関数名に引用符' を付けなければなりません

4 引数版では quanc8($\langle f(x)$ または x の式 $\rangle, x, lo, hi$) にて, 関数か式 (最初の引数) の変数 (二番目の引数) で区間 lo から hi で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. 区間の分割に時間を費すのは, 大域変数 quanc8_]relerr(デフォルト値=1.0e-8) と quanc8_abserr(デフォルト値=1.0e-8) で指定されたエラー条件に達した時のみで, エラー条件は以下の二つで与えられます.

- 相対エラーテスト:
  —integral(関数)-計算値—¡quanc8_relerr*—integral(関数)—
- 絶対値エラーテスト :
  —integral(関数)-計算値—¡quanc8_abserr.

quanc8

quanc8 ('function name,lo,hi)

適応型積分器で share1;qq fasl にて利用可能.demo と usage ファイルがある. この手法は newton-cotes 8-panel 求積法を用い, そして, 関数名 quanc8 は 3 か 4 個の引数のものが利用可能であります. 絶対と相対エラーチェックが用いられています. 利用には load("qq"); を実行します.

residue

residue (exp, var, val)

変数 var の値 val に対する式 exp の複素平面上での留数を計算します. 尚, 留数は exp の laurent 級数展開に於ける (var-val)^(-1) の係数です.

```
(c1) residue(s/(s**2+a**2),s,a*%i);
                          1
(d1)                      -
                          2
(c2) residue(sin(a*x)/x**4,x,0);
                          3
                         a
(d2)                   - --
                         6
```

risch

risch (exp, var)

risch アルゴリズムの transcendential case を用いて var に対する exp の積分を行うものです (risch アルゴリズムで代数的な場合は実装されていません). これは現在,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を行う.integrate は, これらの場合が与えられると自動的に risch を適用します.

erfflag[true]
false であれば, 開始すべき導関数が無い場合に risch が erf 関数を答の中に導入する事を避ける.

```
(c1) risch(x^2*erf(x),x);
            2     2
          - x     x             3           2
        %e     (%e   sqrt(%pi) x  erf(x) + x  + 1)
(d1)   ----------------------------------------
                      3 sqrt(%pi)
(c2) diff(%,x),ratsimp;
                                 2
(d2)                            x  erf(x)
```

integrate で risch 積分を強制的に利用するには ev を用います:
`ev(integrate(3^log(x),x),'risch);`

尚,defint 等ではこの指定が出来ない為,defint.lisp で定義されている関数 antideriv の修正が必要になります.

romberg
romberg(⟨ 被積分関数 ⟩,⟨ 積分変数 ⟩,⟨ 下限 ⟩,⟨ 上限 ⟩);
romberg(⟨ 被積分関数 ⟩,⟨ 下限 ⟩,⟨ 上限 ⟩);
romberg 積分.romberg を利用する為に任意のファイルを読込む必要は無く, 自動的に読込まれます.

例:

```
romberg(sin(y),y,1,%pi);
        time= 39 msec.          1.5403023
f(x):=1/(x^5+x+1);
romberg(f(x),x,1.5,0);
        time= 162 msec.         - 0.75293843
```

例 :

```
f(x):=(mode_declare([function(f),x],float),1/(x^5+x+1));
translate(f);
romberg(f,1.5,0);
        time= 13 msec.          - 0.75293843
```

最初の引数は translate 関数で変換された関数か,compile 関数で翻訳された関数でなければならない (compile 関数で翻訳されていれば,flonum を返す為に宣言されていなければならない). 最初の引数が未だに変換されたものでなければ, romberg はそれを translate で変換せずにエラーを返す. 積分の精度は大域変数 rombergtol(デフォルト値 1.e-4) と rombergit(デフォルト値 11) で操作される. もし, 続く近似で相対誤差が rombergtol よりも小さければ romberg は結果を返す. 諦める前に romgergit 倍の刻幅を半分にして試みる.romberg が実行する反復と関数評価の数は romgergabs と rombergmin で制御される.

romberg は再帰的に呼び出されていても良く, それ故, 二重, 三重積分が実行可能です.

例：

```
integrate(integrate(x*y/(x+y),y,0,x/2),x,1,3);
                          13/3 (2 log(2/3) + 1)
%,numer;
                          0.81930233
define_variable(x,0.0,float,"global variable in function f")$
f(y):=(mode_declare(y,float), x*y/(x+y) )$
g(x):=romberg('f,0,x/2)$
romberg(g,1,3);
                           0.8193023
```

この方法の長所は関数 f が他の目的, 例えば, プロットの為に使える事です. 短所は関数 f とその自由変数 x の両方に対する名前を考慮しなければならない事です.

即ち, 大域変数無しならば：

```
g1(x):=(mode_declare(x,float), romberg(x*y/(x+y),y,0,x/2))$
romberg(g1,1,3);
                              0.8193023
```

となります.

ここでの長所は簡潔さにある.

```
q(a,b):=romberg(romberg(x*y/(x+y),y,0,x/2),x,a,b)$
q(1,3);
                              0.8193023
```

この方法はより簡潔なもので, 変数は romberg の文脈に含まれる為に宣言される必要がありません. 残念な事に, 多重積分での romberg の利用には非常に大きな短所があります. これは多重積分を表現する事で幾何学的情報が欠落する為に膨大な特別な計算が必要とされるので, この方法は信頼出来ません. 大域変数 rombergtol と rombergit を正確に理解して使わなければなりません.

尚,romberg の〈 上限 〉と〈 下限 〉では, 内部計算で倍精度の浮動小数点を用いているので, 多倍長精度 (bigfloat) で変換した浮動小数点は使えません.

tldefint

tldefint(exp,var,ll,ul)

tlimswitch が true に設定されている ldefint.

第24章　代数方程式

24.1　Maximaでの代数方程式について

方程式は演算子=を挟んで左右の=を持たない式が配置された式です. 例えば,x^2+2*x+1=0の様な形になります. 尚,=の左右の式は函数 lhs と rhs で取り出す事が出来ます.

```
(%i17) eq1:x^2+2*x+1=y^2;
                                 2              2
(%o17)                          x  + 2 x + 1 = y
(%i18) lhs(eq1);
                                   2
(%o18)                            x  + 2 x + 1
(%i19) rhs(eq1);
                                        2
(%o19)                                  y
```

この例では方程式として x^2+2*x+1=y^2 を eq1 に割当てており, lhs(eq1) で方程式の左側の x^2+2*x+1,rhseq1 で方程式右側の y^2 を各々取り出しています.

尚,lhs と rhs は演算子=に対してのみ使える函数です. 他の二項演算子 (例えば,¿等) には使えません.

Maxima では連立方程式も扱えます. ここで連立方程式は$[eq_1, \cdots, eq_n]$の様に, 複数の方程式で構成されたリストで表現します.

```
(%i25) eq2:[2*x^2-5*y=1,x+y*x+y^2=4];
                           2             2
(%o25)                [2 x  - 5 y = 1, y  + x y + x = 4]
(%i26) eq2[1];eq2[2];
                                    2
(%o26)                             2 x  - 5 y = 1
(%i27)
                                   2
(%o27)                            y  + x y + x = 4
```

この例では, 二つの方程式 $2 * x^2 - 5 * y = 1$ と $x + y * x + y^2 = 4$ から構成される方程式のリストを eq2 に割当てています. 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います.

Maxima では与えられた方程式を, 厳密解, 近似解の二通りで解く事が出来ます. 厳密解は数学的に厳密な解, 近似解は文字通りの真の解に対する近似です. 例えば, 方程式を $x^2-2=0$ とすれば,$x=\pm\sqrt{2}$ が厳密解, $x=\pm 1.4142$ がその近似解となります.

尚, 重複解を持つ方程式を解くと, 大域変数 multiplicities に重複度をリストとして設定する函数が幾つかあります.

algsys では factor で与えられた方程式を因子分割し, 終結式を用いて不要な多項式を減らして解を求めようとします.

24.2　方程式に関連する大域変数

%rnum_list

デフォルト値:[]

変数%r が algsys 函数で解に導入された場合, それらは%rnum_list に生成された順番で追加されます. これは後に解に代入する時に便利です. concat('%r,j) を実行するよりも, このリストを使う事を薦めます.

algexact

デフォルト値:[false]

algsys 函数に影響を与える大域変数の一つです. true であれば,algsys は solve を呼出し,realroots を常に利用します. false であれば, 終結式が単変数でない場合と quadratic か biquadratic な場合のみ solve の呼出しを行います.

algexact:true は厳密解のみを保証するものではなく,algsys が最初に厳密解を計算しようと試み, 結局,all か失敗した時に近似解のみを生成します.

algepsilon

デフォルト値:[$10\hat{}8$]

algsys 函数で利用される定数です.

backsubst

デフォルト値:[true]

false ならば, 方程式を三角関数化した後の代入を防ぎます. 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るでしょう.

breakup

デフォルト値:[true]

false であれば,solve はデフォルト値の幾つかの共通部分式で構成されたものとしてはなく, 一つの式として三次又は二次の方程式の解の表示を行います. 但し,breakup が true となるのは programmode が false の時だけです.

dispflag

デフォルト値:[true]

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.

globalsolve

デフォルト値:[false]

true の場合, 解かれた変数に解の値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
```

尚,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)
```

linsolvewarn

デフォルト値:[true]

false であれば,dependent equations eliminated(従属方程式は消去された) というメッセージ出力が抑制されます.

insolve_params

デフォルト値:[true]

true であれば,linsolve はまた記号%ri を生成し,algsys に記載された任意の助変数を表現する為に用いられます.

false であれば, 以前の様に linsolve が動作します. 即ち, 不定方程式型に対し, 他の項の幾つかの引数に対して解きます.

multiplicities

デフォルト値:[not_set_yet]

solve や realroots で返される個々の解の重複度リストが設定されます.

```
(%i3) multiplicities;
(%o3)                                not_set_yet
(%i4) realroots(x^2-4=0,1.0e-5);
(%o4)                             [x = - 2, x = 2]
(%i5) multiplicities;
                                [1, 1]
(%o5)
(%i6) realroots(x^3-4*x^5=0);
                                      1      1
(%o6)                         [x = - -, x = -, x = 0]
                                      2      2
(%i7) multiplicities;
(%o7)                                [1, 1, 3]
```

polyfactor

デフォルト値:[false]

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.

programmode

デフォルト値:[true]

false であれば,solve,realroots,allroots と linsolve が,%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 であれば,solve 等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]
```

realonly
デフォルト値:[false]
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.

rootsepsilon
デフォルト値:[1.0e-7]
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.

rootsmode
デフォルト値:[true]
大域変数 rootsconmode は,true,false,all の値が設定可能で, rootscontract に影響を与えます.
rootsconmode が false ならば,rootscontract は wrt 有理数次数の分母が同じ次数のものだけを約分します.
true の場合, 次数の分母が割切れるもの同士のみ約分します. 例えば, (x^(1/6)*y^(1/8) の場合, 次数の分母の 8 と 6 に注目すると,8 は 6 の倍数ではない為,rootscontract でこの項を纏める事は出来ません. これに対し,(x^(1/4)*y^(1/8) の場合は, 分母の 8 は 4 で割切れる為, rootscontract は (x^2*y)^(1/8) を返します.
all の場合,rootscontract は次数の分母の LCM を取って纏めます.

ieqnprint
デフォルト値:[true]
ieqn 函数が返した結果の挙動を制御します.ieqnprint が false であれば, ieqn 関数によって返されたリストは以下の形式となります.

```
[solution, technique used, nterms, flag]
```

ここで, 解が完全 (exact) であれば flag の個所はありません. そうでなければ, 不完全や非閉形式解に各々対応する approximate か incomolete という語になります. 級数による手法を用いると,nterms

は取られた項の数 (これは, エラーがより多くの項の生成を妨げた場合,ieqn に対して与えられた n よりも小さくなる) を与えます.

rootsconmode
デフォルト値:[true]
rootscontract 命令の挙動を定めます.

solvedecomposes
デフォルト値:[true]
true であれば, 多項式を解く際に,solve に polydecomp を導入します.

solveexplicit
デフォルト値:[false]
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
デフォルト値:[true]
false であれば,solve は式の因子分解を実行しません.

solvenullwarn
デフォルト値:[true]
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 [] が返されます.

solveradcan
デフォルト値:[false]
true であれば solve は radcan を用います.radcan を使うと solve は遅くなりますが, 指数や対数関数を含む問題に対処出来ます.

solvetrigwarn
デフォルト値:[true]
false であれば,solve は方程式を解く為に逆三角関数を利用し, それによって解が失なわれる事を警告しません.

solve_inconsistent_error
デフォルト値:[true]
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とエラーを表示します.
false であれば, 空リスト [] を返します.

24.3 代数方程式に関連する函数

algsys

algsys ([⟨ 方程式$_1$⟩, ⟨ 方程式$_2$⟩, ···], [⟨ 変数$_1$⟩, ⟨ 変数$_2$⟩, ...])

algsys は多項式方程式のリストや連立多項式方程式を, 与えられた変数リストに対して解きます. 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. これらは助変数を表示する為に用いられるもので, 助変数は大域変数%rnum_list に蓄えられています.

```
(%i1) algsys([2*x+3*y=1],[x,y]);
                                          2 %r1 - 1
(%o1)                      [[x = %r1, y = - ---------]]
                                             3
(%i2) %rnum_list;
(%o2)                                  [%r1]
```

この例の様に,(連立) 方程式の階数が足りない場合には, 助変数を補って与えられた方程式を解きます. 尚,%rnum_list には algsys で使われた助変数が逐次追加されて行きます.

algsys は以下の手順で方程式を解き, 必要であれば再帰的に処理を行います.

1. 最初に方程式を factor で因子分解し, 各因子から構成される部分系 $system_i$, 即ち, 方程式の集合を構築します.

2. 各部分系 $system_i$ から, 方程式 eqn を取出し, 変数 var を選択します. ここで変数 var は非零で最小次数を持つものから選びます. それから方程式 eqn と部分系 $system_i - eqn$ に含まれる方程式 eqn_j を変数 var の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 $system_{i+1}$ がより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, 浮動小数による近似がなければ, 厳密解を求める為に solve を呼出します.

   方程式が単変数で線型, 二次, 或いは四次の多項式で近似されていれば, solve が再び呼出されます.

   近似されている場合や方程式が単変数で線型, 二次又は四次の何れでもなく, realonly が true の場合, 実数値解を見付ける為に関数 realroots を呼出します. realonly が false の場合, 解を求める為に allroots が呼び出されます.

   尚,algsys が要求以下の精度解を生成した場合、algepsilon の値をより小さな値に変更しても構いません.algexact が true であれば,solve が常に呼び出されます.

4. (3) の段階で得られた解を以前の段階に挿入し, 解の計算過程は (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 を使えば, 大きくて複雑な式が出来ます. この場合,pickapart や reveal を解の計算に使えます.

実数値解であるにもかかわらず, 解の中に純虚数%i が入っている事があります.

allroots

allroots(⟨ 方程式 ⟩)

単変数の実数係数多項式に対して実数解と複素解全てを計算します.

実多項式には Jenkins(Algorithm 493,toms,vol. 1, (1975),p.178), 複素多項式には,Jenkins と Traub のアルゴリズム (Algorithm 419,comm. acm,vol.15,(1972), p. 97) が用いられています.

polyfactor が true の時, 与えられた多項式が実係数であれば実数上で因子分解を行いますが, 係数に%I が含まれていれば複素数上で因子分解を行います.

```
(%i14) allroots(%i*x^2+1=0);
(%o14) [x = .7071067811865475 %i + .7071067811865475,
        x = - .7071067811865475 %i - .7071067811865475]
(%i15) polyfactor:true;
(%o15)                                true
(%i16) allroots(x^2+1=0);
                                     2
(%o16)                              x  + 1.0
(%i17) allroots(%i*x^2+1=0);
(%o17) %i (x - .7071067811865475 %i - .7071067811865475)
          (x + .7071067811865475 %i + .7071067811865475)
```

allroots は重複解を持つ時に不正確な結果を返す事があります. この場合は与式に%i をかけたものを計算すれば解決する場合かもしれません.

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.

funcsolve

funcsolve(⟨ 方程式 ⟩, ⟨$g(t)$⟩)

方程式を満たす有理函数 ⟨$g(t)$⟩ が存在すれば有理函数のリストを返し, 存在しない場合は空リストを返します. 但し, 方程式は ⟨$g(t)$⟩ と ⟨$g(t+1)$⟩ の一次線形多項式でなければなりません. 即ち,funcsolve は一次線形結合の漸化式に対して利用可能です. 但し, この函数の出来はまだ良くありません.

```
(%i28)  funcsolve((n+1)*foo(n)-(n+3)*foo(n+1)/(n+1) =
(n-1)/(n+2),foo(n));

dependent equations eliminated:  (4 3)
                                              n
(%o28)                        foo(n) = ---------------
                                       (n + 1) (n + 2)
```

ieqn

ieqn(⟨ 積分方程式 ⟩,⟨ 未知函数 ⟩,⟨ 手法 ⟩,⟨n⟩,⟨$guess$⟩)

積分方程式を解く函数です. 尚, この函数は保存領域の解放を行う為, kill(labels) が含まれています.kill(labels) を実行すると, 入力 (%i ラベルに保存) や計算結果 (%o ラベルに保存) は全て失われてしまいます. その為, 積分方程式パッケージを読込む前に, 保持しておきたい式に名前を付けておかなければなりません.

ここで,⟨ 手法 ⟩ には,

- first
  解を見付ける第一の方法を試みる.
- all
  全ての適応可能な方法を試みる.

の二種類があります.

⟨n⟩ と ⟨$guess$⟩ は neumann や firstkindseries への予測値の初期値になります.

⟨n⟩ は taylor,neumann,firstkindseries や fedseries(微分手法に対する繰返しの最大深度でもある) から取る冪の最大次数です.

第二から第五の助変数に対する初期値は,unk:p(x) です. ここで p は第一の関数で, Maxima の未知関数となる被積分関数に現れるものです. そして,x は第 2 種の方程式の場合, 積分の外側で現われるものです.

p の第一の出現に対応する引数として現われるか, 第 1 種の方程式で積分変数を除いたその他の変数となります.

x を探す事に失敗すると, 利用者は独立変数を与える事を要求されます.

tech:first; n: 1; guess:none

これで neumann と firstkindseries を初期予測値として f(x) を用います.

lhs と rhs

lhs(⟨ 方程式 ⟩)

rhs(⟨ 方程式 ⟩)

lhs は ⟨ 方程式 ⟩ の演算子=の左側を返し,rhs は右側の式を返します.

linsolve

linsolve ([⟨ 方程式$_1$⟩, ⟨ 方程式$_2$⟩, ···], [⟨ 変数$_1$ > . < 変数$_2$⟩, ···])

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が, 方程式であっても構いません.

尚,globalsolve が true であれば,solve で解かれた変数に連立方程式の集合の解が設定されます.

```
(c1) x+z=y$
(c2) 2*a*x-y=2*a**2$
(c3) y-2*z=2$
(c4) linsolve([d1,d2,d3],[x,y,z]),globalsolve:true;
solution
(e4)                          x : a + 1
(e5)                           y : 2 a
(e6)                          z : a - 1
(d6)                        [e4, e5, e6]
```

nroots

nroots (⟨ 多項式 ⟩, ⟨ 下限 ⟩, ⟨ 上限 ⟩)

⟨ 上限 ⟩ と ⟨ 下限 ⟩ で指定された半開区間 (⟨ 下限 ⟩, ⟨ 上限 ⟩] 内部に存在する単変数多項式の実数根の個数を返します.

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. 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
```

nthroot

nthroot (⟨ 多項式 ⟩, ⟨n⟩)

⟨ 多項式 ⟩ は整数係数の多項式,⟨n⟩ を正整数, 返すのは整数上の多項式で, この多項式の ⟨n⟩ 乗が ⟨ 多項式 ⟩ となるか, そうで無ければ ⟨ 多項式 ⟩ が完全な第 ⟨n⟩ の冪乗ではないというエラーメッセージが表示されます.

このルーチンは factor や sqfr よりも遥かに速い特徴を持っています.

```
(%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
```

realroots

realroots (⟨ 多項式 ⟩)

realroots (⟨ 多項式 ⟩, ⟨ 許容範囲 ⟩)

⟨ 多項式 ⟩ は実単変数多項式で, その全ての実根を ⟨ 許容範囲 ⟩ で指定する許容範囲内で求めます.

ここで,⟨ 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 ⟩ は希望する精度を達成する為に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, rootsepsilon に設定した値が使われます.

realroots は解の重複度リスト multiplicities を設定します.

```
(%i34) realroots(x^2-2=0,1.0e-5);
                                  370727      370727
(%o34)                    [x = - ------, x = ------]
                                  262144      262144
(%i35) float(sqrt(2)-rhs(%o34[2]));
(%o35)                         2.289179735770474E-6
```

この例では多項式ではなく, 方程式 $x^2 - 2 = 0$ を精度 1.0e-5 以内で求めています. 解は浮動小数点ではなく, 有理数の形で返されます.

rootscontract

rootscontract (⟨ 式 ⟩)

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)` の様に纏めたものに変換します.

rootsconmode が false ならば,rootscontract は wrt 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*y3^)^(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	(x2^*y)^(1/4)
x^(1/2)*y^(1/3)	all	(x3^*y2^)^(1/6)

radexpand が true で domain が real であれば,rootscontract は abs を sqrt に変換します. 即ち,`abs(x)*sqrt(y)` を `sqrt(x^2*y)` に変換します.

rootscontract は logcontract と似た手法で ratsimp を用います.

solve

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$_n$⟩])

代数方程式 ⟨ 式 ⟩ を ⟨ 変数 ⟩ に対して解き, 方程式の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 ⟩ が関数 f(x) の多項式であれば, 最初に f(x) に対して解き, その結果を c とすれば, 方程式 f(x)=c を解く事で対処出来ます.

```
(%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.

- 式 exp が変数 var のの線形結合であれば,var に対して自明に解けます.
- 式 exp が $a var^n + b$ 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.
- 式 exp が変数 var の線形結合ではなく, 式 exp に含まれる var の各次数の gcd (n とします) が次数を割切る場合には, 根の重複度リスト multiplicities に n が追加されます. そして,solve は var^n で exp を割った結果に対して再び呼出されます.
- exp 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.

- 式 exp が二次, 三次, 又は四次の多項式方程式の場合, 解の公式を必要があれば用います.

solve([⟨ 方程式$_1$⟩, $\cdots$, ⟨ 方程式$_n$⟩], [⟨ 変数$_1$⟩, $\cdots$, ⟨ 変数$_n$⟩]) の場合, 多項式の方程式系を linsolve, 或いは algsys を用いて解き, その変数で解のリストを返します.

linsolve を用いる場合, 第一引数のリスト (⟨ 方程式$_i$⟩, $i = 1, \cdots, n$) は解くべき方程式を表現し, 第二の引数リストは求めるべき未知変数のリストになりますが, 方程式中の変数の総数が方程式数と等しい場合, 第二の引数リストは省略しても構いません.

与えられた方程式が十分でない場合,inconsistent と云うメッセージを表示します. これは大域変数 solve_inconsistent_error で制御出来ます. 単一解が存在しない場合は singular と表示されます.

第25章　微分方程式

25.1　Maximaでの微分方程式の扱い

Maximaでの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が含まれています. 更に, 未知函数がどの様な変数に依存するものかを明示的に記述しなければなりません. 例えば, 次の書式は正確な書式ではありません. 何故なら, fとgが何の変数であるか判りません.

```
'diff(f,x,2)=sin(x)+'diff(g,x);
'diff(f,x)+x^2-f=2*'diff(g,x,2);
```

Maximaの微分方程式は次の様に何の函数であるかをきちんと書か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);
```

常微分方程式に初期値を与えたければ, 普通の函数と同様にatvalue命令を用います. 尚,atvalueで初期値を設定する場合は,desolveやodeを用いて微分方程式を処理する前に初期値の設定を行わなければなりません. atvalueは atvalue(f(x),x=pt,val) で, 函数f(x)に対し,x=ptで値valを設定します. 尚,fが多変数の場合は atvalue(f(x,y,···),[x1=p1,x2=p2,···],[val1,val2,···]) の様に境界と対応する値をリストで与えます.

```
(%i88)  x^2*'diff(y,x) + 3*y*x = sin(x)/x;
                              2 dy           sin(x)
(%o88)                       x  -- + 3 x y = ------
                                dx             x
(%i89)  ode2(%,y,x);
                                    %c - cos(x)
(%o89)                         y = -----------
                                         3
                                        x
```

```
(%i90)  ic1(%o88,x=%pi,y=0);
                                      cos(x) + 1
(%o90)                           y = - ----------
                                           3
                                          x
(%i91) 'diff(y,x,2) + y*'diff(y,x)^3 = 0;
                                2
                               d y       dy 3
(%o91)                         --- + y (--)  = 0
                                 2       dx
                               dx
(%i92) ode2(%,y,x);
                             3
                            y  + 6 %k1 y
(%o92)                      ------------ = x + %k2
                                6
(%i93) ratsimp(ic2(%o92,x=0,y=0,'diff(y,x)=2));
                                   3
                                  2 y  - 3 y
(%o93)                            - ---------- = x
                                       6
(%i94) bc2(%o92,x=0,y=1,x=1,y=3);
                                3
                               y  - 10 y       3
(%o94)                         --------- = x - -
                                   6           2
```

25.2 微分方程式に関連する函数

bc2

bc2(< 一般解 >,< x の値$_1$ >,< y の値$_1$ >,< x の値$_2$ >,< y の値$_2$ >)

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 等で計算した微分方程式の一般解です. 二階の方程式の一般解では定数が二つ現われる為, 特殊解を求める為, 異なった二点での連立方程式を解く必要があります. その為, < x の値$_1$ > と < y の値$_1$ > が一つの点での値,< x の値$_2$ > と < y の値$_2$ > がもう一つの別の点での値を定めます. ここで, 値の与え方は, 一般解の変数を x と y とすると,< x の値$_1$ > と < y の値$_1$ > を x=x0 や y=y0 の様に,< 対応する変数 >=< 境界値 > の書式で記述します.

尚, load(ode2) でプログラムの読込を行う必要があります.

desolve

desolve ([< 方程式$_1$ >,$\cdots$,< 方程式$_n$ >],[< 変数$_1$ >,$\cdots$,< 変数$_n$ >])

微分方程式 < 方程式$_i$ > とその従属変数 < 変数$_1$ >,$\cdots$ < 変数$_n$ > を指定します. ここで, 関数と変数の関連性ははっきと指定しなければなりません. 又, 初期条件は desolve を呼出す前に,altvalue 函数で与え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 は false を返します.

尚,load(ode2) でプログラムの読込を行う必要があります.

```
(c11) 'diff(f(x),x)='diff(g(x),x)+sin(x);
                        d         d
(d11)                   -- f(x) = -- g(x) + sin(x)
                        dx        dx
(c12) 'diff(g(x),x,2)='diff(f(x),x)-cos(x);
                         2
                        d          d
(d12)                   --- g(x) = -- f(x) - cos(x)
                          2        dx
                        dx
(c13) atvalue('diff(g(x),x),x=0,a);
(d13)                                 a
(c14) atvalue(f(x),x=0,1);
(d14)                                 1
(c15) desolve([d11,d12],[f(x),g(x)]);
                x                             x
(d16) [f(x)=a %e  - a+1, g(x) = cos(x) + a %e  - a + g(0) - 1]

/* 検証 */

(c17) [d11,d12],d16,diff;
                    x       x      x                x
(d17)         [a %e  = a %e , a %e  - cos(x) = a %e  - cos(x)]
```

ic1

ic1 (< 一般解 >,< x の値$_1$ >,< y の値$_1$ >)

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 ode 等で計算した微分方程式の一般解となります. 後の二つが境界条件を与えます. 一般解の変数を x,y とすると,< x の値$_1$ > と < y の値$_1$ > は x=x0 や y=y0 の様に,< 対応する変数 >=< 境界値 > の書式になります.

尚,load(ode2) でプログラムの読込を行う必要があります.

ic2

ic1 (< 一般解 >, < x の値$_1$ >, < y の値$_1$ >)

一階の微分方程式の境界値問題を解く函数です.< 一般解 > は ode 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$_1$ > と < y の値$_1$ > は `x=x0` や `y=y0` の様に,< 対応する変数 >=< 境界値 > の書式になります.

尚, load(ode2) でプログラムの読込を行う必要があります.

ode2

ode2 (< 常微分方程式 >, < 従属変数 >, < 独立変数 >)

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 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..

第26章　グラフ表示

in_netmath
[false]
nil でなければ,plot2d は openplot 関数に対して適切なグラフ表示の出力を行う.

plot_options
このリストの要素はグラフ表示に関するデフォルト値である. それらは set_plot_option を用いて変更しても良い.

```
[x, - 3, 3]
[y, - 3, 3]
```

は各々x と y の領域である. [transform_xy, false] にて,false でなければ,

```
make_transform([x,y,z], [f1(x,y,z),f2(x,y,z),f3(x,y,z)])
```

の出力は 3 次元から 3 次元への変換を構成し, これがグラフに適用される. 円筒極座標 (polar_xy) は次の様に与えられる.

```
make_transform([r,th,z],[r*cos(th),r*sin(th),z])
```

- [run_viewer,true] で,false でなければ可視化ソフトウエアがデータの出力を行っていない事を意味する.
- [grid,30,30] は x の領域を 30 個の区間とし,y の領域も同様である事を意味する.
- [colour_z,false] は plot_format ps で色付を適用する.
- [plot_format,zic] は plot3d 向けで, 現在,zic,gnuplot,ps と geomview のデータ書式がサポートされている.

  これらの書式に対しては高品位でパブリックドメインの可視化ツールがある. それらは openmath,izic,gnuplot,ghostview と geomview である.

  openmath viewer はこの配布に含まれており,tcl/tk で構築されている. 実行ファイルは maxima/bin/omplotdata である. この可視化ツールは拡大, 並行移動と (三次元であれば) 回転が

行える. この書式は,netmath グラフ表示を行う為に,netmath でも用いられているものである. (http://www.ma.utexas.edu/users/wfs/netmath.html を参照せよ)

geomview はミネソタ大学の geometry center で開発されたもので, http://www.geom.umn.edu/software/down で入手可能で, anonymous ftp 先は ftp://ftp.geom.umn.edu/pub/software/geomview/ である. 現時点では izic 程綺麗なものでは無いが, 多体, 多光源の優れたサポートを持つ.

gnuplot は ghostview と同様にそこらじゅうにある.gnuplot に tcl のインターフェイスを持ち, マウスを用いたグラフの回転や拡大が行える mgnuplot を提供している.

izic は zennon.inria.fr から ftp によって利用可能である. これは美しいカラーの guraud shading と非常に表示の速い wireframe 表示機能を持ち, X Window System で動作する.

`geomview` の最新版は `http://www.geomview.org/`より入手可能. このマニュアルの原文で指定された `url` は現在でも存在するが, ここでは旧版のみが入手可能である.
又,`izic` に関しては最新の `tcl/tk` に適合したものでは無く, その上, 表示色数も `256` 色である. その為,`izic` を無理にでもインストールする価値はあまり無い.

26.1　描画関数

openplot_curves

openplot_curves (list &rest-options)

```
[[x1,y1,x2,y2,...],[u1,v1,u2,v2,...],..]
```
や
```
 [[[x1,y1],[x2,y2],...],.. ]
```

の様な曲線リストを取り, それらのグラフ表示を行う.xgraph_curves に似ているが, open plot ルーチンを用いる. 追加の記号の引数は”xrange -3 4”の様に与えます. 次の二つの曲線のグラフでは, 大きな点で, 最初のものにはラベル jim を付け, 第二のものには jane を付けて表示します.

```
openplot_curves([["@{plotpoints 1@} @{pointsize 6@} @{label jim@}
      @{text @{xaxislabel @{joe is nice@}@}"] ,
      [1,2,3,4,5,6,7,8],
      ["@{label jane@} @{color pink @} "], [3,1,4,2,5,7]]);
```

他の特別なキーワードは xfun,color,plotpoints,linecolort,pointsize, nolines, bargraph, labelposition, xaxislabel と yaxislabel がある.

plot2d

plot2d (expr,range,...,options,..)
plot2d ([expr1,expr2,..,exprn],xrange,...,options,..)
plot2d (parametric_expr)
plot2d ([..,expr,..,parametric_expr,..],xrange,...,options)

expr は y 軸について 1 変数の関数としてグラフ表示される式である. range は [var,min,max] の形式であり,expr は var に対してグラフ表示される式である. 二番目の plot2d 関数の形式で, 式のリストは表示する為に与えても良い.y 方向での切り捨てはデフォルト値の y の領域に対して行われる. これは option や set_plot_option を用いて指定が行える.

```
plot2d(sin(x),[x,-5,5]);
plot2d(sec(x),[x,-2,2],[y,-20,20],[nticks,200]);
```

更に,expr に助変数式を用いても良い:

parametric_expr は [parametric, xexpr, yexpr, trange, ..options] の形式の maxima のリストであり, ここで xexpr と yexpr は値域 trange の最初の元 var の 1 変数関数となる. グラフ表示は [xexpr, yexpr] の対で,trange での var の変化に沿った軌跡である. 次の例では円を描き, それから, 星型が得られる様に幾つかの点だけで描き, 最後に円と一緒に x の通常の関数を描く.

```
(c1) plot2d([parametric,cos(t),sin(t),[t,-%pi*2,%pi*2]]);
(c2) plot2d([parametric,cos(t),sin(t),[t,-%pi*2,%pi*2],
        [nticks,8]]);
(c3) plot2d([x^3+2,[parametric,cos(t),sin(t),[t,-5,5]]],
        [x,-3,3]);
```

xgraph_curves

xgraph_curves(list)

xgraph を用いてリストで与えられた' 点集合' のリストをグラフ表示する.

点集合の形式は

`[x0,y0,x1,y1,x2,y2,...]` または
`[[x0,y0],[x1,y1],....]`

とする. 点集合はラベルや他の情報を持つ記号を含んでいても良い.

```
xgraph_curves([pt_set1,pt_set2,pt_set3]);
```

は,3 つの点集合のグラフを 3 個の曲線として描く.

```
pt_set:append(["nolines: true","largepixels: true"],
        [x0,y0,x1,y1,...])
```

は点集合 [とその部分] を構築し, 点との間には線分が無い大きな点を用いる. 指定可能なオプションに関しては,xgraph の man ページを見よ.

```
pt_set:append([concat("\"","x^2+y")],[x0,y0,x1,y1,...])
```

はこの特定の点集合に対して,"label" を"x^ 2+y" とする. 頭の" は xgraph にラベルである事を指定するものである.

```
pt_set:append([concat("titletext: sample data")],[x0,...])
```

でグラフ表示の表題を”maxima plot”の代りに”sample data”とする. .2 unit の幅の棒グラフを作り, 異なった棒グラフを二つ表示する為には :

```
xgraph_curves(
       [append(["bargraph: true","nolines: true","barwidth: .2"],
               create_list([i-.2,i^2],i,1,3)),
        append(["bargraph: true","nolines: true","barwidth: .2"],
           create_list([i+.2,.7*i^2],i,1,3))
]);
```

テンポラリファイルとして@filexgraph-out が使われる.

plot3d

plot3d (expr,xrange,yrange,...,options,..)
plot3d ([expr1,expr2,expr3],xrange,yrange,...,options,..)

```
plot3d(2^(-u^2+v^2),[u,-5,5],[v,-7,7]);
```

で, 変数 u と v の領域を各々[-5,5] と [-7,7],u を x 軸,v を y 軸として, $z = 2^(-u^2+v^2)$ のグラフを表示する.

二番目の引数のパターンの例は

```
plot3d([cos(x)*(3+y*cos(x/2)),sin(x)*(3+y*cos(x/2)),y*sin(x/2)],
   [x,-%pi,%pi],[y,-1,1],['grid,50,15])
```

で, これはメビウスの輪を表示し,plot3d の最初の引数で与えられた 3 個の式でパラメータ付けられている. 最後のオプションの ['grid,50,15] は x と y 方向の長方形の刻み数を与える.

```
/* real part of z ^ 1/3 */
 plot3d(r^.33*cos(th/3),[r,0,1],[th,0,6*%pi],
     ['grid,12,80],['plot_format,ps],
     ['transform_xy,polar_to_xy],['view_direction,1,1,1.4],
     ['colour_z,true])
```

ここで, 視点方向は射影の方向を指す. 無限遠からこれを行うが, 視点方向から原点への直線は並行になる. 他の可視化ツールがインタラクティブなオブジェクトの回転を許容する為, これは現在,'ps' グラフ表示形式でのみ利用される.

- メビウスの輪の別の例:

```
plot3d([cos(x)*(3+y*cos(x/2)),
          sin(x)*(3+y*cos(x/2)),y*sin(x/2)],
           [x,-%pi,%pi],[y,-1,1],['grid,50,15]);
```

- クラインの壷:

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

- トーラス:

```
plot3d([cos(y)*(10.0+6*cos(x)),
           sin(y)*(10.0+6*cos(x)),
           -6*sin(x)], [x,0,2*%pi],[y,0,2*%pi],
            ['grid,40,40])
```

gnuplot にも出力する事が可能である：

```
plot3d(2^(x^2-y^2),[x,-1,1],[y,-2,2],[plot_format,gnuplot])
```

時には式の描写を行う関数を定義する必要があるかもしれない.plot3d に渡される全ての引数は plot3d に渡される前に評価されるので, 希望する通りの作業を行う式を構築する事は難しい事もあるが, 関数を構築する事で遥かに簡単なものとなる.

```
m:matrix([1,2,3,4],[1,2,3,2],[1,2,3,4],[1,2,3,3])$
f(x,y):=float(m[?round(x),?round(y)]);
plot3d(f,[x,1,4],[y,1,4],['grid,4,4]);
```

plot2d_ps

plot2d_ps (expr,range)

range に対する expr のグラフ表示を行うポストスクリプト命令の列を pstream に書き出す.expr は 1 変数の式でなければならない.range は expr の表示で用いる [変数, 最小, 最大] の書式のものである.closeps を参照せよ.

closeps

closeps ()

これば表示命令の列の終りに通常呼び出されるべきものである. 現行の出力ストリーム pstream を閉じて nil を設定する.pstream が開いていれば,pstream を閉じる為, グラフ表示の最初に呼んでも良い.pstream に対して書き込む全ての命令は必要であれば pstream を開く.closeps は他のグラフ表示命令と分ける. 二つの領域でグラフ表示を行いたい場合や幾つかのグラフ表示の追加を行いたい場合とそのストリームを開いたままにしておかなければならない為である.

set_plot_option

set_plot_option (option)

option は plot_options リストの要素の一つの形式である.

```
set\_{}plot\_{}option([grid,30,40])
```

は plot3d のデフォルト値で刻み数を変更する. 記号 grid が値を持っていれば, ここで示す様に, それに引用符をつけなければならない：

```
set_plot_option([’grid,30,40])
```

これで値が代入されない.

psdraw_curve

psdraw_curve (ptlist)

plist に含まれる点を結ぶ曲線を描く. 後者は [x0,y0,x1,y1,...] か [[x0,y0],[x1,y1],...] の形式である.

関数 join は x のリストと y のリストを取り, それらを互いに繋ぎ合せる. psdraw_curve は単により原始的な関数 pscurve を呼び出すだけである.

定義は次のとおり：

```
(defun $psdraw_curve (lis)
  (p "newpath")
  ($pscurve lis)
  (p "stroke"))
```

?draw2d もリストを生成する為に用いても良い.

```
points1:?draw2d(1/x,[.05,10],.03)
```

pscom

pscom (com)

com はポストスクリプトファイルに挿入される.

例えば,

```
pscom("4.5 72 mul 5.5 72 mul translate  14 14 scale");
```

第27章　システム

27.1　ラベルの参照

Maximaは入力と計算した結果を各々%iと%oラベルに保存しています. ここで,%は%oの中で最も数字の大きなものを指します. その為,%はMaximaの処理が進むにつれて更新されます.

入力ラベル%iと出力ラベル%oに割当てられた値を参照する函数に, %,_と%thがあります.%は最新の結果を表示し, 最新の入力を返すのが_です. 又,%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,_にはその様な使い方は出来ません.

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て, 各ラベルのカウンタも1に戻されます, その為, 入力は再度(%i1)から開始する事になります.

```
(%i101) 1+2;
(%o101)                                3
(%i102) resultant(x-t,y-t^2,t);
                                      2
(%o102)                            y - x
(%i103) algsys([2*x+3*y=1],[x,y]);
                                        2 %r2 - 1
(%o103)                [[x = %r2, y = - ---------]]
                                           3
(%i104) %;
                                        2 %r2 - 1
(%o104)                [[x = %r2, y = - ---------]]
                                           3
(%i105) _;
(%o105)                                %
(%i106) %i101;
(%o106)                                3
(%i107) %o101;
(%o107)                                3
```

```
(%i108) %i102;
                                                2
(%o108)                    resultant(x - t, y - t , t)
(%i109) kill(labels);
(%o0)                                  done
(%i1)
```

この例では, 様々な処理を行い, それらを%や_で確認し, 最後に kill(labels) でラベル (%i や%o) の内容を全て消去しています. ラベルの消去を行った為に, kill(labels) を入力した (%i109) から,(%o1) を経て (%i1) に初期化されている事に注目して下さい.

27.2　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. 数値の四則演算は実行された後の値がデフォルトで表示されます. ここで, 結果表示に関しては,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, 結果を一次元で表示します.

```
(%i4) display2d;
(%o4)                                  true
```

```
(%i5) 'integrate(f(x),x);
                                  /
                                  [
(%o5)                             I f(x) dx
                                  ]
                                  /
(%i6) expand((x+1)^3);
                              3      2
(%o6)                        x  + 3 x  + 3 x + 1
(%i7) display2d:false;
(%o7) false
(%i8) 'integrate(f(x),x);
(%o8) 'integrate(f(x),x)
(%i9) expand((x+1)^3);
(%o9) x^3+3*x^2+3*x+1
```

Maxima にはこの他にも特殊な表示を行う函数があります. 基本的に Maxima はキャラクター端末しかなかった昔のシステムでの利用を前提としていた為, 積分記号の様に文字を使って数式を表示する等の, ある意味では涙ぐましい努力の跡があります. しかし, 近年の 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.

27.3　ラベルに関連する大域変数

%

Maxima で処理されたもので, 最新の結果を指定します.

%%

この値は Maxima-break の間に処理された最新の値です. 通常の入力では意味を持ちませんが, 例えば,block 文の中で,(n-1) 行目の文の値を n 行目の文で参照する場合には便利です.

例えば, f(n):=block(integrate(xn̂,x),subst(3,x,%%)-subst(2,x,%%)) と f(n):=block([%%],%%:integrate(xn̂,x),s は同値です.

inchar

デフォルト値:[%i]

Maxima の入力行ラベルで用いられる文字. 例えば, デフォルトでは i の為, 入力行ラベルは (%i1) の様に%の後に inchar で指定した i が続いています.

outchar

デフォルト値:[%o]

出力式の名前の先頭に付けられるアルファベットを指定します.

linechar

デフォルト値:[%t]

中間表示される際, 式の名前の前に置かれる文字を指定します.

linenum

その時点での入力行番号が割当てられています.

```
(%i17) linenum;
(%o17)                                17
```

nolabels

デフォルト値:[false]

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,%i や%o 等で入出力の参照が出来なくします. 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.

prompt

デフォルト値:[_]

これは demo 函数のプロンプト記号を指定するもので, playback(slow) mode と (Maxima-break) があります.

27.4 表示に関連する大域変数

ibase

デフォルト値:[10]

入力数値の基数.

obase

デフォルト値:[10]

表示の際に用いる数値の基数.

absboxchar

デフォルト値:[!]

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.

cursordisp

デフォルト値:[true]

true であれば式が論理的列として描かれます. これはカーソルの移動が可能なコンソールでのみ動作します.false であれば, 式は行から行へと単純に表示されます.cursordisp は writefile が有効であれば false になります.

display2d

デフォルト値:[true]

false であれば, 結果表示が二次元的な書式ではなく, 一行に収まる様に表示されます. 長い式や複雑な式, 表示に余裕が無い場合には特に便利です.

display_format_internal

デフォルト値:[false]

true が設定されていれば, 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. 従って, 内部表現そのもので表示するものではありません. ここでの出力は part 函数に対応するものではなく, inpart 函数に準じたものになるとも言えます.

入力	表示	内部表現
a-b;	a - b	a + (- 1) b
a/b;	$\frac{a}{b}$	ab^{-1}
sqrt(x);	sqrt(x)	$x^{1/2}$
x*4/3;	$\frac{4x}{3}$	$\frac{4}{3}x$

%edispflag

デフォルト値:[false]

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,%e^ (-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
```

exptdispflag

デフォルト値:[true]

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 で表示されます.

lasttime

直前に入力した式の計算時間をミリ秒単位で `time` と `gctime` の組合せを成分とするリストです.

linel

デフォルト値:[79]

一行に表示される文字数を設定します. 何時でも変更可能ですが, 長過ぎたり, 短か過ぎたりすれば, 実用的ではないでしょう.

pfeformat

デフォルト値:[false]

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, 通常のプリティプリントで表示されます.

showtime

デフォルト値:[false]

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 は計算処理に於けるメモリのゴミ収集 (gc) に費した時間も零でなければ表示します. この時間は time=の時間表示に含まれています. 尚,time=には計算時間のみが含まれ, 中間表示時間やファイルを読込む時間は含まれておらず,gc への反応性に分けて認識する事が難しい為, 表示される gctime には計算の実行中に費やした全ての gctime を含んでいます. それ故, 稀に time=よりも gctime の方が大きくなるかもしれません.

stardisp

デフォルト値:[false]

デフォルトの 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.

ttyoff

デフォルト値:[false] 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 を開いたファイルに対しても, 同じ出力となります.

27.5　エラー表示に関連する大域変数

error_size

デフォルト値:[10]

エラーメッセージの長さを制御します.

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. この大域変数のデフォルト値は利用者のが置換えても構いません.

error_syms

デフォルト値:[errexp1,errexp2,errexp3]

エラーメッセージで,error_size よりも大きな式は文字列に置換され, その文字列には式が設定されています. 文字列は error_syms リストから取られて初期値は errexp1,errexp2,errexp3 となっています. エラーメッセージが表示された後, 例えば, "the function foo doesn't like errexp1 as input." であれば,errexp1; と利用者が入力すると, その式を見る事が出来ます. 必要であれば, error_syms に別の文字列を設定しても構いません.

27.6 利用者の環境設定に関連する大域変数

myoptions

デフォルト値:[]

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

デフォルト値:[false]

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.

27.7　デバッグに関連する大域変数

debugmode

デフォルト値:[false]

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.

ttyintfun

デフォルト値:[false]

ttyinnum に設定された中断文字が入力された時点で, 動作する函数を制御します. この機能を利用する為には,ttyintfun(デフォルト値は false で, この機能が使われていない事を意味します) を引数を持たない函数として設定します.

ここで,ttyinnum をデフォルトの 21 のままにしていれば,û (Cntrl-u) が入力された時に, この函数が動作します.

例えば,for 文のループで増分が i で for 文を動かしている間に i の値を簡単に確認したければ次の様に行なえます：

```
ttyinnum:21$ ttyintfun:printi$ printi():=print(i)$
```

すると,ˆ u を打ち込めば何時でも変数 i の検査が行える様になります.

ttyintnum

デフォルト値:[21](control+u(ˆ u) のアスキーコード値に対応)

これはどの文字が中断文字になるかを制御します.ˆ u は記憶を補助する変数 (mnemoric value) として選ばれています. 他の利用者はˆ u を何か他の事で利用している訳でも無い限り,ttyintnum を再設定すべきではありません.

values

デフォルト値:[]

全ての値が束縛されたアトム, 即ち, 利用者変数で,Maxima のオプションや大域変数では無いもので,:,::や函数等で値を束縛されたものを含むリストです.

```
(%i29) values;
(%o29)                                []
(%i30) a:1;
(%o30)                                 1
(%i31) values;
(%o31)                                [a]
```

27.8 ラベル処理に関連する函数

%th

%th(⟨ 正整数 ⟩)

⟨ 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, 結果が指定出来るからです.

labels

labels(⟨ 文字 ⟩)

文字%i,%o に%t を引数として取り, 全ての%i-ラベル,%o-ラベルや%t-ラベルのリストを各々生成します, もし,solve で沢山の%o-ラベルを生成した場合は first(rest(labels(%o))) とすれば, 最新の%o-ラベルが何であるかが判ります.

labels は引数として任意の記号名を取り,inchar,outchar や linechar を再設定すれば, ラベルの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. 変数の labels はデフォルト値が無設定の c,d と e 行の値が設定されたもののリストとなります.

27.9　式の表示に関連する函数

disp

disp(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, $\cdots$)

display に似ていますが, 引数の値のみを表示します.

dispcon

dispcon (tensor1,tensor2,...)

dispcon (all)

defcon に対して与えられた tensori の縮約属性を表示します. dispcon(all) は定義されている全ての縮約属性を表示します.

display

display(⟨ 式$_1$⟩, $\cdots$, ⟨ 式$_n$⟩)

⟨ 式$_1$⟩, $\cdots$, ⟨ 式$_n$⟩ を表示します. その左側が未評価の式で, その右側の行の中心がその式の値となる. この函数は block や for 文で, 中途結果の表示を行うのに便利です. display の引数は通常, アトム, 添字された変数や函数の呼出しです.

dispterms

dispterms (⟨ 式 ⟩)

引数の成分を一つづつ, 一つの成分を表示すると下に次の成分を表示して行きます. つまり, 最初に ⟨ 式 ⟩ の演算子が表示され, 和の各項, 積の因子, より一般の式の成分が分離されて表示されます. これは,⟨ 式 ⟩ が表示にとても大きい時には便利です.

例えば,p1,p2,$\cdots$ がとても大きな式ならば, 表示プログラムは p1+p2+$\cdots$ を一度に表示しようとして, 保存領域を使い果してしまうかもしれません. しかし,dispterms(p1+p2+$\cdots$) は p1 を最初に表示すれば, その下に p2 を表示等々となります.dispterms を利用しない時, もし, 指数式が a^ b の形式で表示するにはとても長い場合,expt(a,b)(又は,a^ (-b) の場合は ncexpt(a,b)) で表示されます.

expt

expt (a,b)

指数式が a^ b で表示されるのに余りにも大き過ぎる場合は expt(a,b), a^ (-b) の場合は ncexpt(a,b) と表示します.

grind

grind (⟨ 引数 ⟩)

⟨ 引数 ⟩ を string 函数よりも, より読み易い書式で表示します. 値として%o-行を返します.

ldisp

ldisp(⟨ 式$_1$⟩, ⟨ 式$_2$⟩, $\cdots$)

disp に似ていますが, 中間ラベルを生成する点で異なります.

```
(%i1) ldisp(x^2+y);
                                      2
(%t1)                               y + x

(%o1)                                [%t1]
```

ldisplay

ldisplay(⟨ 式$_1$⟩,⟨ 式$_2$⟩,$\cdots$)

display に似ていますが, 中間ラベルを生成する点が異なります.

nostring

nostring(⟨ 引数 ⟩)

後戻しを行っている時に, 全ての入力行を string で文字列にする代りに, その表示を行います.⟨ 引数 ⟩ が grind であれば, その表示はより読み易い書式となる.

尚,playback([5,10],20,time,slow) の様に, 任意の数のオプションを入れても構いません.

print

print(⟨ 式$_1$⟩,⟨ 式$_2$⟩,$\cdots$)

⟨ 式$_1$⟩ がら順番に評価を行い, その結果を表示します. ここで,⟨ 式$_i$⟩ に含まれるアトムや函数の前に単引用符' が置かれていたり, 文字列 (全体を二重引用符で括ったもの) の場合は, 評価を行わずに, そのままで表示を行います.

tcl_output

tcl_output (⟨ リスト ⟩,⟨ 添字 > & < オプション : 飛幅 ⟩) ⟨ 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.

```
tcl_output([x1,y1,x2,y2,x3,y3],1) --> @{x1 x2 x3 @}
tcl_output([x1,y1,x2,y2,x3,y3],2) --> @{y1 y2 y3 @}
tcl_output([1,2,3,4,5,6],1,3) --> @{1 4@}
tcl_output([1,2,3,4,5,6],2,3) --> @{2 5@}
```

reveal

reveal (⟨ 式 ⟩,⟨ 深度 ⟩)

⟨ 深度 ⟩ は整数値で k, 指定された各々の成分の長さで ⟨ 式 ⟩ を表示します. 和は sum(n), 積は product(n) として表示されます.

ここで n は和や積の成分の数になります. 指数函数は expt で表現されます.

```
(%i10) aa:integrate(1/(x^3+2),x)$
(%i11) aa;
                                                 1/3
                                          2 x - 2
                                    atan(------------)
              2    1/3      2/3          1/3                      1/3
         log(x  - 2    x + 2   )        2    sqrt(3)    log(x + 2   )
(%o11)  - ----------------------- + ------------------ + -------------
                    2/3                2/3                     2/3
                 6 2                   2    sqrt(3)         3 2
(%i12) reveal(aa,1);
(%o12)                              sum(3)
(%i13) reveal(aa,2);
(%o13)                    negterm + quotient + quotient
(%i14) reveal(aa,3);
                                     atan         log
(%o14)               - quotient + ---------- + ----------
                                  product(2)   product(2)
(%i15) reveal(aa,4);
                       log       atan(quotient)   log(sum(2))
(%o15)            - ---------- + -------------- + -----------
                    product(2)     expt sqrt        3 expt
(%i16) reveal(aa,5);
                                      sum(2)
                                 atan(----------)
                 log(sum(3))        product(2)    log(x + expt)
(%o16)         - ----------- + ---------------- + -------------
                   6 expt         2/3                   2/3
                                 2    sqrt(3)        3 2
```

show

show (⟨式⟩)

⟨式⟩を添字された対象として表示します. その際, 共変添字を下添字, 反変添字を上添字として表示します. 微分添字は下添字として表示され, コンマで共変添字が分離されています.

tex

tex(⟨式⟩)

tex(⟨式⟩,⟨ファイル名⟩)

tex(⟨ ラベル行 ⟩,⟨ ファイル名 ⟩)

与えられた ⟨ 式 ⟩ や ⟨ ラベル行 ⟩ を TEX の書式に変換します.⟨ ファイル名 ⟩ を指定すると, 出力結果は指定ファイルに保存されます. 尚, 指定ファイルが既存の場合, 結果はそのファイルの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.

```
(c1) integrate(1/(1+x^3),x);

                                      2 x - 1
                          2        atan(-------)
                   log(x  - x + 1)      sqrt(3)    log(x + 1)
(d1)             - --------------- + ------------- + ----------
                          6            sqrt(3)          3
(c2) tex(d1);
$$-@{@{\log \left(x^@{2@}-x+1\right)@}\over@{6@}@}+@{@{\arctan @{@{2\>x-1@}\ov
 \sqrt@{3@}@}@}@}\over@{\sqrt@{3@}@}@}+@{@{\log \left(x+1\right)@}\over@{3@}@}

(d2)                                  (d1)
(c6) tex(integrate(sin(x),x));

$$-\cos x$$
(d6)                                  false
(c7) tex(d1,"/tmp/jo.tex");

(d7)                                  (d1)
```

box

box(⟨ 式 ⟩)

box(⟨ 式 ⟩,⟨ ラベル ⟩)

⟨ 式 ⟩ を文字で囲んで返します. この箱は式の一部でもあります. 尚, box(⟨ 式 ⟩,⟨ ラベル ⟩) とすると ⟨ 式 ⟩ を文字で囲み, 上に ⟨ ラベル ⟩ を表示します. 但し,⟨ ラベル ⟩ が長ければ, 表示の際に切捨てられてしまいます.

尚, 大域変数 boxchar に box,dpart や lpart 函数で与えられた文字列を囲む際に使う文字を割当てます.

```
(c1) box(" this is a pen. ");

                              """""""""""""""""""
(d1)                          " this is a pen. "
                              """""""""""""""""""
```

```
(c30) box(" this is a pen. ","maxima");

                              "maxima"""""""""""
(d30)                         " this is a pen. "
                              """""""""""""""""""
(c31) boxchar:x;

(d31)                                  x
(c32) box(" this is a pen. ","maxima");

                              "maxima"xxxxxxxxxx
(d32)                         x this is a pen. x
                              xxxxxxxxxxxxxxxxxx
```

dpart

dpart(⟨ 式 ⟩, ⟨ 整数$_1$⟩, ··· , ⟨ 整数$_k$⟩)

⟨ 式 ⟩ の内部表現から, ⟨ 整数$_1$⟩, ··· , ⟨ 整数$_k$⟩ で指定される部分式を文字で囲んで ⟨ 式 ⟩ 全体を表示します. この囲まれる部分 ⟨ 式 ⟩ は part 函数で抜出す部分式になります.

```
(c1) dpart(x+y/z**2,1,2,1);
                        y
(d1)                   ---- + x
                         2
                       *****
                       * z *
                       *****
```

lpart

lpart (⟨ ラベル ⟩, ⟨ 式 ⟩, ⟨n_1⟩, ··· , ⟨n_k⟩)

dpart に似ていますが, ラベル付けられた箱を用います. ラベル付けられた箱は dpart で生成したものに似ているますが, 上の行に名前がある点で異なります.

rembox

rembox (⟨ 式 ⟩, ⟨ 引数 ⟩)

⟨ 式 ⟩ から ⟨ 引数 ⟩ に沿って box を削除します. ⟨ 引数 ⟩ が unlabeled, 全てのラベル付けされていない box が削除されます.⟨ 引数 ⟩ があるラベルの名前であれば, そのラベルを伴う box だけが削除されます.⟨ 引数 ⟩ が省略されると, 全ての box が削除されてしまいます.

27.10 システムに関連する函数

alias

alias(⟨新名称$_1$⟩, ⟨旧名称$_1$⟩, ⋯, ⟨新名称$_n$⟩, ⟨旧名称$_n$⟩)

(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.

debug,debugprintmode,LISPdebugmode

これらの函数は引数を必要としません. debug() の様に使います, LISPdebugmode(); debugprintmode(); と debug(); はシステムプログラマーが使う虫取り機能を利用者が使える様にします. これらのツールは強力ですが, 幾つかの取決めが通常の Maxima のものと異っていて, それらの利用はとても直感的に感じられるかもしれません. これらの命令は Maxima 言語で構築した関数が上手く動作する様に虫取りしなければならない利用者向けに設計されています.

kill

kill(⟨引数$_1$⟩, ⟨引数$_2$⟩, ⋯)

Maxima から指定した引数を消去します.

- argi が変数 (単配列要素を含みます), 函数, 又は配列の場合, 指定された項目は, その属性の全てと一緒に Maxima の中核から消去されます.
- argi=labels であれば, 古くなった全ての入力, 中間行, 出力行 (他の名付けられていない項目を除きます) が消去されます.
- argi=clabels であれば, 入力行だけが消去されます.
- argi=elabels であれば, 中間の t-行のみが消去されます.
- argi=dlabels であれば, 出力行のみが消去されます.
- argi が任意の他の情報のリスト (要素は Maxima 変数の infolists のもの) の名前であれば, そのクラス (とその属性) の全ての項目が削除されます.
- argi=all であれば, 以前定義された全ての情報リストに関する全ての項目が labels と同様に削除されます.
- argi=数値 (n とする) であれば, 最後の n 行 (つまり, 最後の n 個の行) が消去されます.
- argi が [m,n] の形式であれば,m と n の間に含まれる行番号の行が全て削除されます.

kill(values) や kill(variabkes) 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 c7 行の 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(d7) も実行しなければなりません.

kill(allbut(name1,...,namek)は kill(all) を, その名前を allbut で指定したものを除外して実行します (注意:namei は,u,v,f,g の様な名前を意味し, functions の様な infolist ではありません).kill は与

えられた引数から全ての属性を削除するので,kill(values) は values リストの全ての項目に関連する全ての属性を削除しますが, それに対して remove 函数群 (remvalue, remfunction,remarray,remrule) は指定した属性を削除します.

更の, 後者はリストの名前か指定した引数が存在しなければ false を出力しますが, kill は指定した項目がたとえ存在しなくても常に `done` を出力します.

式の削除は, あまりにも多くの fasl ファイルが読み込まれたり, メモリの割当ての水準が高くなり過ぎたかの何れかの理由により,`no core -fasload` という文句が出る問題への対処にはなりません. この場合は, 削除すべきものが無いのです. 式の削除は単に幾つかの領域を空にするだけで, より小さくする事ではありません.

remfunction

remfunction ($\langle$ 函数$_1\rangle, \langle$ 函数$_2\rangle, \cdots$)

利用者の定義函数を Maxima から削除します. 利用者定義の函数は大域変数 functions にその名前が保存されており,remfunction はこの functions に含まれている函数の削除を行います. 尚,all が唯一の引数であれば functions に含まれる全ての利用者定義の函数が削除されます.

reset

reset ()

全ての Maxima の初期化を行います. この場合, 全ての大域変数の値の殆どがデフォルト値に戻されますが, 一部,linel の様に割当によってのみ変更可能な表示に関連する大域変数は Maxima の計算機能として考えられている為, この初期化されるものには含まれません.

sstatus

sstatus ($\langle$ 属性 $\rangle, \langle$ パッケージ $\rangle$)

状態を設定する事を意味します.status(feature,hack_package) が true を返す様に sstatus(feature,hack_package) を用いる事が出来ます. これはパッケージを書く場合, それらが読み込まれた機能の記録を保つのに便利である.sstatus(feature,hack_package) はその名の通り, 状態を設定する事を意味します.

sstatus(feature,hack_package) が true を返す様に status(feature,hack_package) を用いる. これはパッケージを書く際に, それらが読み込んだ機能の履歴を保存するのに便利である

playback

playback(arg)

入力と出力行の後戻し (play-back) を実行します. arg=n(整数) であれば, 最近の n 個の式 (ci,di と ei を各々1 で数える) が後戻しされ,arg が省略されていれば, 全ての行となります.arg=input であれば, 入力行が後戻しされ,arg=[m,n] であれば,m から n 迄の間に含まれる全ての行が後戻しされます. もし,m=n であれば,[m] だけで引数としては十分です. arg=slow であれば, デモ (その対極としては"速い"batch) の様に遅く後戻しを実行します. これは有用な式を取り出す目的で,save や stringout と連携して第二の保存ファイルを生成する時に便利です.

もし,arg=time であれば, 計算時間が式と同様に表示される. もし,arg=gctime か totaltime であれば,showtime:all; を用いたのと同じ様に計算時間の完全な詳細が表示される.arg=string であれ

ば, 全ての入力行を文字列として返し (string 函数を見よ), それらを表示すると云うよりは後戻しをする.arg=grind であれば,”grind” モードに (入力行の進行に対して) 切り替わる (grind を見よ). playback([5,10],20,time,slow) の中のオプションの様に任意の数を含んで良い.

system

system(⟨ 命令 ⟩)

オペレーティングシステムの命令等の 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”); を入力します.

quit

quit ()

Maxima を停止させます. 入力は引数無しで quit(); 或いは quit()$と入力します. 単に quit だけでは意味がありません.

尚,Maxima を一時的に停止させるのであれば,Cntrl-C(Ĉ) を入力します.

to_lisp

collapse

collapse (⟨ 式 ⟩)

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, 領域を節約する (collapse は optimize 命令で用いられるサブルーチンである). それ故,fassave を使う前や save で保存したファイルを読み込んだ後に collapse を呼出す事は便利である.

collapse([expr1,...,exprn]) を用いて, 幾つかの式を一緒に潰す事も可能である. 同様に,collapse(listarray('a)) とする事で, 配列の成分を潰す事も出来る.

? 函数や変数の前置詞として, 函数や変数が LISP の表記であって,Maxima の表記では無い事を識別する. 二つの疑問符??は Maxima の現行の Maxima の命令行の内容を空にします.

(平成 17 年 12 月 1 日 (木))

第28章　LISPに関連する函数

28.1　MaximaとLISP

Maxima は Common LISP と呼ばれる LISP の一方言で記述されています. LISP は函数型と呼ばれるプログラム言語で, プログラムは様々な函数を定義し, それらを組合せて行く作業とも言えます. 因みに,C や FORTRAN は手続型と呼ばれます.

Maxima はこの LISP の上で動作する環境ですが,Maxima 自体は PASCAL 風の処理言語を持っており, 構文的にも LISP を意識する事は単純な利用では殆どありません.

但し,Maxima で酷いエラーを出すと LISP のデバッガに落ちる事があります. LISP のデバッガからの抜け方は,Maxima を実装した LISP によって微妙に異なりますが,CLISP の場合は`:q`と入力してみて下さい. すると Maxima に戻ります.

LISP の特徴は言語仕様が非常に柔軟な点です.LISP にはアトムと呼ばれる変数があり, それらを空行で区切って小括弧 () で括ったリストと呼ばれるデータが, 最も基本的なデータとなります. このリストはリストのリストといったものも許容します. このアトムとリスト等で構成されたデータを S 式と呼びます. 因に,LISP のプログラム自体も S 式です. その為,LISP の函数でプログラムを操作する事も容易に行えます.

その為,Maxima のプログラムでは足りない所を LISP で代用する事も多くあります. 又,Maxima から LISP を直接利用する事も可能です. この場合は Maxima の to_LISP 函数を利用します.Maxima で`to_lisp();`と入力すると, 裏方の LISP が表に出ます. これで LISP を使って遊べます. この状態から Maxima に戻りたければ, `(to-maxima)`と入力します. すると通常の Maxima に戻ります.

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

Maxima> (setq $a '1)
1
Maxima> (to-maxima)
returning to Maxima
(%o1)                                true
(%i2) a;
(%o2)                                  1
```

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

この例で注目して頂きたい事は,Maxima で表示されているものと LISP 側で見たものと様子が違う事です. 即ち,Maxima で扱うデータ, 更には函数それ自体も LISP 側では別の表記方法があります. この LISP 側での表現を単純に内部表現と呼んでいます. この内部表現を通常の処理で気にする事は殆どありませんが, 細かな処理を行う必要が出た時点で初めて意識する事になります.

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

尚,?と函数の間に空行を入れると Maxima のオンラインマニュアルを呼出そうとするので, 注意が必要です.

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

```
(%i26) a:x+y+z;
(%o26)                                z + y + x
(%i27) :lisp $a;
((MPLUS SIMP) $X $Y $Z)
(%i27) :lisp (car $a)
(MPLUS SIMP)
(%i27) ?car(a);
(%o27)                               ("+", simp)
```

上記の例では変数 a に x+y+z を割当てていますが,$a が変数 a の内部変数名となります. :lisp $a; でこの変数に割当てた値を参照していますが, 返却値は内部表現そのものです. この様な変数の参照は?では出来ません.

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

この様に,?は入力と出力に Maxima が介在する為に, 入出力をする度に Maxima の評価を受ける事になりますが,:lisp の場合は直接操作となります. :lisp は Maxima で内部表現を確認する必要がある場合に特に便利な函数です.

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

各種データの内部表現については各々の章で必要があれば解説を行います.

28.2　LISP に関連する函数

?

?⟨*LISP* の函数 ⟩

演算子?を LISP の函数の頭に付ける事で,Maxima から直接 LISP の函数が利用可能になります. 函数の記述方法は Maxima 風に行い, 引数も Maxima 側で見えている変数名 (内部表現では通常アトムの先頭に$が付いています) を与えます. 又, 返却値も Maxima の解釈を経たものとなり, 内部表現そのものが返却される訳ではありません.

尚,?と LISP の函数の間には空行を入れてはなりません. 空行を入れた場合, Maxima は describe(引数) と解釈して, 引数に関連したオンラインマニュアルを起動しようとします. その為, オンラインマニュアルが起動してしまうか, 或いは結果として false が返却される事になります.

:lisp

:lisp ⟨*S* 式 ⟩

:lisp の後に空行を入れて LISP の S 式を続けると,LISP 側で S 式を評価し, その結果を Maxima に返します. 尚, 結果は%o ラベルには蓄えられず, 内部表現を処理したものが生のまま返されます.

to_lisp,to-maxima と continue

to_lisp()

(to-maxima)

(continue)

to_lisp は Maxima から LISP に移行する為の函数で, 引数は必要ありません. to_lisp で制御は Maxima から LISP に移されます. この時点でプロンプトが `Maxima>`に変化します.

LISP から Maxima に戻る函数は (to-maxima) です. この函数も引数は不要です. この函数を実行すると,Maxima は to_lisp() を入力した入力行の結果として true を返し, セッションを再開します. 又,(continue) も LISP から Maxima 戻る際に使えます. 動作は (to-maxima) と同様です.

28.3 ヘルプについて

Maxima にはオンラインヘルプがあります. オンラインヘルプを参照する為には, DESCRIBE か? を使います. 使い方は,describe("inte"), 或いは,? inte とすると inte を含む事項が以下の様に表示されます.

```
(%i37) ? inte;

 0: (maxima.info)Introduction to Elliptic Functions and Integrals.
 1: Definitions for Elliptic Integrals.
 2: Integration.
 3: Introduction to Integration.
 4: Definitions for Integration.
 5: INTERRUPTS.
 6: ASKINTEGER :Definitions for Simplification.
 7: DISPLAY_FORMAT_INTERNAL :Definitions for Input and Output.
 8: INTEGERP :Definitions for Miscellaneous Options.
 9: INTEGRATE :Definitions for Integration.
 10: INTEGRATE_USE_ROOTSOF :Definitions for Integration.
 11: INTEGRATION_CONSTANT_COUNTER :Definitions for Integration.
 12: INTERPOLATE :Definitions for Numerical.
Enter space-separated numbers, ALL or NONE:
```

後は番号を入力するか,all を入力します.

他に, 函数のデモを実行する DEMO 命令と例題を実行する EXAMPLE 命令があります. どちらも各命令の実行後に中断し, スペースキーを押すと次の例題を実行します. 尚, デモファイルの拡張子は .dem です.

28.4 LISP と Maxima

Maxima の全ては lisp で記述されています. その為, 関数と変数の名前の変換があります. lisp の階層で$で始まる全ての記号は Macsyma の階層では$を外して読まれます. 例えば, 二つの lisp の関数 TRANSLATE と$TRANSLATE があります.Maxima の階層で TRANSLATE(FOO); と入力すると, 内部で呼出される関数は$translate 関数になります. もう一方の関数を呼出す為には, ?を前に置きます. 尚,?の後には空行を入れてはいけません. LISP 命令を直接入力したければ,:lisp 命令を使って,

```
(%i1) :lisp (foo 1 2)
```

の様にするか,to_lisp(); を使って直接 LISP を表に出す方法, 最後に, debug break に入る為に Ctrl-c を使う方法があります. それから,$%o2 の評価を行えば, ラベル%o2 行の値を内部 LISP の書式で見られます. 更に,:q と入力するとトップレベルに戻ります. もし,to_lisp(); で Maxima を抜けていれば,LISP プロンプトに続けて (run) か (to-maxima) と入力します.

Macsyma の階層で呼出せる LISP の関数を書きたければ, それらの名前は$ で始まる様にしなければなりません.LISP の階層で入力された全ての記号は|$odeSolve|の様にでもしない限り, 大文字で自動的に読込まれます. もし, 記号が既に読込まれていた場合や, 最初の読込んだ時点で, 大文字だけの同名の記号がまだ存在しなければ,Maxima はその記号を大文字と小文字が混在したものとして解釈します.

```
(C1) Integrate;
(D1) INTEGRATE
(C2) Integ;
(D2) Integ
```

記号 Integrate は Maxima のプリミティブの為に, その大文字が既に存在しています. しかし,INTEG はまだ存在していないので,Integ は Maxima に受け入れられます, これは幾らか曖昧に見えるかもしれませんが,Maxima のプリミティブが大文字であろうが小文字であろうが, 古い Maxima のプログラムが動作するのを保ちたい為です. このシステムの長所は,LISP の階層で入力しても, 即座に Maxima のキーワードか関数であるかが参照可能な事でしょう.

LISP の階層で Maxima の書式を入力をする為には,#$マクロを用いて構いません.

```
(setq $foo #$[x,y]$)
```

これは次に様に入力するのと同じ効果を持ちます.

```
(%i1)FOO:[X,Y];
```

例外は VALUES リストに foo が現れない事です.Macsyma の表示書式で foo を見る為には次の様に入力しなければなりません :

```
(displa $foo)
```

この文書では,Macsyma の記号を参照する時に,Macsyma の階層で入力するのと全く同様に$を省略します. これは,LISP の記号を参照する時には混乱の原因となる. この場合, 通常の LISP 記号に対しては小文字を用い,Macsyma の記号に対しては大文字を用います. 例えば,$list には LIST, 表示名が”list”の LISP の記号には list とします.

Maxima 言語で定義された関数は通常の LISP 関数ではないので, それらを呼出す為に mfuncall 函数を使わなければなりません.

例えば :

```
(%o2)                          FOO(X, Y) := X + Y + 3
```

が LISP の階層では

```
MAXIMA> (mfuncall '$foo 4 5)
12
MAXIMA>
```

となります.

幾つかの LISP 関数が Maxima パッケージに潜んでいます. 何故なら, それらを Maxima で利用しても, システム関数の定義と互換性が無いからです. 例えば,typep は MacLisp と Common LISP での挙動は異なります. もし,Maxima のパッケージで zeta lisp の typep を参照したければ,global:typep(又は Common Lisp では cl:typep) を用いなければなりません.

つまり,

```
(macsyma:typep '(1 2)) ==> 'list
(lisp:typep '(1 2))==> error (lisp:type-of '(1 2))==> 'cons
```

となります.

どの記号が裏に潜んでいるかを参照する為には,"src/maxima-package.lisp" を参照するか,LISP 階層でパッケージの describe 函数を実行しましょう.

28.5 ごみ集め

記号処理は膨大なゴミの生成を行う傾向があり, この効率的な実装は, ある種のプログラムの完遂にとって非常に重要な事になります. mprotext システムコールが利用可能な UNIX システム (SunOS 4.0 や BSD の幾つかを含む) 上での GCL では多重層のゴミ集め機能が利用可能です. これは最近書き込まれたページの回収に限定されます.ALLOCATE や GBC に関する GCL の文書を参照して下さい. LISP 階層で (setq si::*notify-gbc* t) を実行すれば, どの領域がより多くの空領域が必要とするのかを決定する手助けになるでしょう.

第29章　ファイルの利用

29.1　Maxima[でのデータ入出力について

ここではMaxima[へのデータ入力と出力について述べます. Maxima[はCommon LISPで記述されている為, 基本的に入出力はLISPの入出力函数を用いたものです. 特にMaxima[は表示されている形式と内部形式が別物の為, 結果や式を保存したり, 逆に保存したものを読込む際には, 注意が必要になります.

ファイルに画面と同じ出力を行いたければ,writefileを用います. このwritefileはLISPのdribble函数を用いたもので, 入力と出力をそのまま指定したファイルに保存します. 但し,writefileでは指定したファイルを新規に生成するので, 単純に既存のファイルに記録したければ,appendfile函数を用います.writefileとappendfileで開いたファイルを閉じる場合はclosefile()で開いたファイルを閉じます.

但し, これらのファイルは実質的に記録ファイルであって,Maxima[でそのまま再利用は出来ません. 再利用可能なファイルを生成するのは,saveとstringoutです. ここで,saveはMaxima[の内部表現を保存する函数で,loadやloadfileを用いてMaxima[に読込みます. これに対して,stringoutやgrindは内部形式ではなく, Maxima[の入力に対応する通常の形式でデータの保存を行います.

Maxima[でCのscan命令に似たものに,readとreadonlyがあります. これらの函数は, 引数として与えた文字列を全て同一行に表示し, キーボードからの入力を待ちます. 利用者は通常のMaxima[の入力と同様に式を入力します. ここで, 行末には; か$を付けます. すると,readの場合は式をMaxima[で評価し, readonlyの場合は評価せずにそのまま受け取ります.

この様に, 単純なファイル操作に限定されるとは言え, 必要なものは一応揃っており, 足りない部分はLISPで補う事になります.

ウインドウのスクロールダウンで入力行の後戻し (playback) が可能で, 現在の作業を失なう事もありません. これは函数 e を入力する事で実行が可能です. 開始すべき行数を数値引数で尋ねると, それ以外は40行遡る事となります.

29.2　ファイル処理に関連する大域変数

batchkill

デフォルト値:[false]

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

デフォルト値:[0]

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, file_search_demo

load や他の函数で, ファイルの読込を行う際に検索されるディレクトリのリスト. 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

デフォルト値:[true].

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

デフォルト値:[false]

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

デフォルト値:[false]

save,fassave, 或いは translate を用いてパッケージ (ファイル) を作成する場合, packagefile:true$と設定して, 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[の情報リスト (例えば values,functions) に追加される事を避けたいかもしれない.

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.

with_stdout(file,stmt1,stmt2,...) ファイルを開き,stm1,stm2,... の評価を行います. 標準出力への任意の表示は端末の代りにファイルに送られ, 端末側には常に false が返されます.

```
mygnuplot(f,var,range,number_ticks):=
 block([numer:true],
 with_stdout("/tmp/gnu",
   for x:range[1] thru range[2] step
                (range[2]-range[1])/number_ticks
       do (print(x,at(f,var=x)))),
 system("echo \"set data style lines; set title '",
        f,"' ;plot '/tmp/gnu'
;pause 10 \" | gnuplot"));

(c8) with_stdout("/home/wfs/joe",
      n:10,
      for i:8 thru n
        do(print("factorial(",i,") gives ",i!)));
(d8)                                false
(c9) system("cat /home/wfs/joe");
factorial( 8 ) gives  40320
factorial( 9 ) gives  362880
factorial( 10 ) gives  3628800
(d9)                                  0
```

29.3　ファイル処理に関連する函数

appendfile

appendfile(⟨ ファイル名 ⟩)

指定したファイルに Maxima[の入出力の追加を行います.writefile との違いは, 同名のファイルが存在した場合,writefile では上書きをしてしまいますが, appendfile 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 と同様に指定したファイルは closefile() で閉じます.

batch

batch(⟨ ファイル名 ⟩)

指定されたファイルに含まれる Maxima[の命令行を逐次評価します. ファイルはパスを含まない場合,file_search_Maxima[に含まれるディレクトリ上を検索し, 存在した場合には読込みと実行をします.

ファイルの内容は基本的に Maxima[での入力行と同じもので, 行末には; か$を置きます. 又,%と %th を用いて入力と出力を指定する事も出来ます. 尚, 空行,Tab や改行コードは無視されます.

batch 処理ファイルは通常のテキストエディタで編集する事も出来ますし, Maxima[の stringout 函数で出力したものも使えます.

深刻なエラーが生じた場合, ファイル末端に達した場合にのみ, 利用者に制御が戻されます. 但し, 利用者はどの時点でも Cntrl-g を押せば, この処理を止められます.

batchload

batchload (⟨ ファイル名 ⟩) 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

batcon (⟨ 引数 ⟩) 中断されたファイルのバッチ処理を再開します.

closefile

closefile()

appendfile や writefile で開かれたファイルを閉じます. closefile は LISP の close を使った函数です. この close は開かれたストリームを閉じる函数になります.

filename_merge

filename_merge (⟨ 文字列$_1$⟩, ⟨ 文字列$_2$⟩)

文字列$_1$ と 文字列$_2$ の結合を行います. 内部的には, 先頭に#Pを文字列の先頭に付けた対象を生成しますが,Maxima[上では, 単純に文字列を繋ぎ合せた様にしか見えません.

基本的には Maxima[の各種命令でファイルの検索を行う際にパス指定のあるファイル名を生成する際に用いられる函数です.

file_search

file_search(⟨ ファイル名 ⟩)

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

file_type (⟨ ファイル名 ⟩)

指定したファイルの属性を返します. 但し, ファイル名の末尾で判断する函数で, 返却する値も,fasl,LISP や Maxima[を返します.

尚,fasl はコンパイルされた LISP ファイルです.

load

load (⟨ ファイル名 ⟩)

文字列や, リストで表現されたファイル名の読込みを行います. ディレクトリが指定されていなければ, 最初にカレントディレクトリ, それから file_search_Maxima[,file_search_LISP や file_search_demo といった大域変数に保存されているディレクトリを検索し, 指定されたファイルを読込もうとします.

load はファイルが batch 処理に対応している事を見付けると,batchload を用います (これは, 黙って端末に出力やラベルを出力せずにファイルの batch 処理を実行する事を意味しています).

他のファイルの読込を行う Maxima[命令に,loadfile,batch と demo があります.loadfile は save で書込んだファイルに対して動作し,batch と demo は strignout で書込まれたり, テキストエディタで命令のリストとして生成されたファイル向けです.

loadfile

loadfile (⟨ ファイル名 ⟩)

指定されたファイルを読込みます. この函数は以前の Maxima[の処理で save 函数で保存した値を Maxima[に戻す事に使えます.

ここで, パスの指定はオペレーティングシステムのパスの指定に方法に従います. 例えば,unix の場合は/home/user ディレクトリにある foo.mc ファイルを読込むのであれば,"/home/user/foo.mc" となります.

尚.save 函数で保存したファイルを loadfile で読込むと,Maxima[は初期化されるので注意が必要です.

read

read (⟨ 文字列$_1$, $\cdots$ ¿ この函数は画面上に全ての引数を表示して入力を待ちます. 利用者が式を入力すると, 入力した式は Maxima[に引渡されて評価が行なわれます.

```
(%i45) a:read("mikeneko");
mikeneko
diff(x^2+1,x);
(%o45)                                2 x
```

この例では,mikeneko と表示された後に、`diff(x^2+1,x);` を入力しています. ここでの入力でも通常の入力と同様に行末に; か$が必要です. この例では, 入力した値が Maxima[に評価されて, 結局,a に `2*x` が割当てられています.

readonly

readonly (⟨ 文字列$_1$⟩, ···)

引数を全て表示して, それから式を読込みます. 基本的には read と同様ですが, read と違うのは, 読込んだ式を評価しない事です.

```
(%i46) a:readonly("mikeneko");
mikeneko
diff(x^2+1,x);
                                      2
(%o46)                          diff(x  + 1, x)
```

save

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)

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. 尚, 保存した値は削除されずに Maxima[本体にも残っています.

⟨ 引数$_1$⟩,⟨ 引数$_2$⟩, ··· で, 各引数の値を保存します.

[⟨m⟩, ⟨n⟩] で m 番目の入力行から n 番目の入力行の内容を保存します.

引数に大域変数 values,functions,labels 等を指定する事も可能です. values,functions で利用者が設定した変数値や定義関数を全て保存します. 又,labels を指定すると, 入出力行の内容が全て保存します.

最後に, 引数に all を指定すれば,Maxima[の内容をファイルに保存しますこの場合は, 入力や計算結果だけではなく,Maxima[の様々な設定も一緒に保存されるので, 処理した内容以上にファイルが大きなものとなるので注意が必要になります. save 函数の返却値は保存先のファイル名となります.

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

```
(%i4) save("test",all);
(%o4)                                test
```

save 函数で保存したファイルは loadfile 函数で Maxima[に再び読込む事が出来ます. 但し,loadfile を実行すると, 以前の Maxima[自体を初期化し,save 函数を実行した時点にまで戻してしまう効果があるので, 注意が必要になります.

以下の例では最初に loadfile でファイル test を読込んでいますが, 行ラベルは上の save で保存する場合と同じものになっている事と, 二度目に loadfile を実行するとラベルが (%i8) から (%i5) に戻っている事に注意して下さい.

```
(%i1) loadfile("test");
(%o4)                                test
(%i5) %i2;
                                      2    2
(%o5)                          a1 : y  + x  + 1
(%i6) %i1;
(%o6)                                  6
(%i7) %o3;
                                         2
(%o7)                                y - x
(%i8) loadfile("test");
(%o4)                                test
(%i5)
```

尚,save ファイルの内容は,LISP の S 式そのものとなります. 要するに,LISP 上で動く Maxima[の為のデータファイルになります. ファイルの先頭側に実行内容の内部形式が記述されますが, その後には Maxima[の諸設定が保存されます. その為, 以下に示す例は 1+2+3 から 4 行の入力だけでしたが,save 函数で保存したファイル (test) は 256 行に及びます. これは内部形式で記述するとどうしても長くなる側面もありますが, 実際は, 入出力以外に様々な設定 (大域変数の値等) も保存されている為です.

```
;;; -*- Mode: LISP; package:Maxima[; syntax:common-LISP; -*-
(in-package "Maxima[")
(DSKSETQ $%I1 '((MPLUS) 1 2 3))
(ADDLABEL '$%I1)
(DSKSETQ $%O1 6)
(ADDLABEL '$%O1)
(DSKSETQ $%I2 '((MSETQ) $A1 ((MPLUS) ((MEXPT) $X 2) ((MEXPT) $Y 2) 1)))
(ADDLABEL '$%I2)
(DSKSETQ $%O2 '((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 2)))
```

```
(ADDLABEL '$%O2)
(DSKSETQ $%I3
 '(($RESULTANT) ((MPLUS) $X ((MMINUS) $T))
   ((MPLUS) $Y ((MMINUS) ((MEXPT) $T 2))) $T))
(ADDLABEL '$%I3)
(DSKSETQ $%O3
 '((MPLUS SIMP) ((MTIMES SIMP) -1 ((MEXPT SIMP RATSIMP) $X 2)) $Y))
(ADDLABEL '$%O3)
(DSKSETQ $%I4 '(($SAVE) &TEST $ALL))
(ADDLABEL '$%I4)
(DSKSETQ $A1 '((MPLUS SIMP) 1 ((MEXPT SIMP) $X 2) ((MEXPT SIMP) $Y 2)))
(ADD2LNC '$A1 $VALUES)
以下略
```

その為,Maxima[内部でデータがどの様に処理されているかを見る場合には重宝するでしょう. とは言え, 作業を一旦中断し, 中断した個所から再度処理を行う必要がなければ,save 以外の命令, 例えば,stringout や grind を用いた方が総合的な使い勝手自体は良いでしょう.

stringout

stringout(⟨ ファイル名 ⟩,⟨ 式$_1$⟩,⟨ 式$_2$⟩,$cdots$)

stringout(⟨ ファイル名 ⟩,[⟨m⟩,⟨n⟩])

stringout(⟨ ファイル名 ⟩,input)

stringout(⟨ ファイル名 ⟩,functions)

stringout(⟨ ファイル名 ⟩,values)

指定したファイルに Maxima[が読込める書式で出力します. 直接,⟨ 式$_1$⟩,⟨ 式$_2$⟩,$cdots$ と式を並べると, 各式が順番にファイルに書込まれます.

引数に,[⟨m⟩,⟨n⟩] と指定すると入力行の m 行から n 行がファイルの書込まれます.

これらに対し,input を指定すると入力行全てが書込まれます. functions を指定すると大域変数 functions に記載された利用者定義の関数が全て保存されます.

同様に values を指定すると, 大域変数 values に記載された利用者定義の変数の値が書込まれます.

尚, この stringout 函数は writefile を実行中に利用する事も可能です.

大域変数の grind が true であれば,stringout は文字列では無く,grind と同じ書式で出力します.

writefile

writefile(⟨ ファイル名 ⟩)

書込み用にファイルを新規に開きます.writefile を実行すると, それ以降の Maxima[への入力と出力処理は全て指定したファイルに記録されます. その為, このファイルをそのまま Maxima[に再度読込ませたりする事は出来ません. ファイル名の指定は文字列で行います. 文字列ではなく,ABCD の様に二重引用符無しで指定すると,Maxima[はアトムの内部表現で用いる$を頭に付けたファイル名 (この例では$ABCD) でファイルを生成します. 尚, この writefile の実体は LISP の dribble 函数です.

ファイルを閉じる場合は closefile() を用います.
以下に簡単な例を示します.

```
(%i1) writefile("test1");
(%o1)           #<OUTPUT BUFFERED FILE-STREAM CHARACTER test1>
(%i2) 1+2+3;
(%o2)                                  6
(%i3) diff(sin(x)*x+2,x);
(%o3)                          sin(x) + x cos(x)
(%i4) closefile();
(%o4)        #<CLOSED OUTPUT BUFFERED FILE-STREAM CHARACTER test1>
```

上記の writefile で生成したファイル test1 の内容を以下に示します.

```
;; Dribble of #<IO TERMINAL-STREAM> started 2005-11-17 06:31:16
(%o1)           #<OUTPUT BUFFERED FILE-STREAM CHARACTER test1>
(%i2) 1+2+3;
(%o2)                                  6
(%i3) diff(sin(x)*x+2,x);
(%o3)                          sin(x) + x cos(x)
(%i4) closefile();
;; Dribble of #<IO TERMINAL-STREAM> finished 2005-11-17 06:31:40
```

この様に,Maxima[の画面入出力そのままが保存されています. この writefile 函数を記録ファイルの生成に利用すれば, 下手なフロントエンドも不要になります.

第30章　Maximaの実行環境について

30.1　Maximaの初期化

Maximaを起動する際に,Maximaをカスタマイズする為のファイルmaxima-init.macが自動的に読込まれます. 尚, このファイルはMaximaを起動するディレクトリ上に置く必要があります.

maxima-init.macにはMaximaの函数や大域変数の設定が記述可能です. 特に,setup_autoload函数を用いると, 必要なパッケージの読込が自動的に行えます.

以下に非常に簡単な例を示します.

```
/*-*-Maxima-*-*/
setup_autoload("vect");
showtime:all;
testdata:"nekoneko";
```

先ず, 註釈は/* */で入れます. この例では頭に,/*-*-Maxima-*-*/ と置いてMaxima言語のファイルである事を示していますが, わざわざ書込む必要性はありません. 次のsetup_autoloadは初期化ファイルの中で函数を自動的に読込む際に使います. その後には大域変数showtimeの設定によるMaximaの環境設定を行っています. この初期化ファイルには利用者独自の物を書込めます. この例ではアトムtestdataに文字列”nekoneko”を割当てています.

30.2　中断の方法

計算が異常に長い場合, 間違って計算を実行させた場合に,Maximaを一旦中断する必要があります.Maximaの計算を中断したければ, 通常は制御文字^c(Cntrl-c)を使います.Maximaは^z(Cntrl-z)が入力されても中断しますが, この場合はMaxima を出てUNIXのshellレベルに戻るので, 通常はCntrl-cで計算を中断してbreak loopに入ります. 尚,:tと入力すればMaximaの最上層に再び戻ります.

30.3　函数

room

room ()

room (true)

room (false)

保存領域の状況を詳細な記述で出力して Maxima の管理に蓄える. これは lisp の room 函数を利用したものです.

room(false) - 非常に詳細な記述を出力するか, 大半は同様の情報を含んでいます.

setup_autoload

setup_autoload (⟨ ファイル ⟩, ⟨ 函数$_1$⟩, $\cdots$, ⟨ 函数$_n$⟩)

⟨ ファイル ⟩ , 関数名,"funci" で,"funci" の呼出しを行い,"funci" が定義されていなければ, "file" によるファイルの指定は自動的に load によって読み込まれなければならず, その file には"funci" の定義を含んでいなければならない事を指図する (これは呼出しによる過程, 例えば,Maxima で integrate が様々なファイルを読み込む原因となる). Maxima でファイルを扱う他の命令での様に setup_autoload の引数は評価されません.

setup_autoload は配列関数に対しては動作しません.

status

status (⟨ 引数 ⟩)

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.

使える引数と結果は:

- time
  計算で消費した時間.
- day
  その週の日
- date
  年, 月, 日々のリスト
- daytime
  時間, 分, 秒のリスト
- runtime
  現時点の Maxima で集計された cpu 時間に原子"milliseconds" をかけたもの.
- realtime
  利用者が Maxima を立ち上げてから経過した実際の時間 (秒単位).
- gctime
  計算で費やしたゴミ集め (garbage collection) 時間
- totalgctime
  Maxima でゴミ集めに費やした総時間.
- freecore
  アドレス空間を使い果す前に拡張可能な Maxima のコアブロックの数 (1 ブロックは 1024

語).250*blocks(mc にて得られる最大値) から値を減じ, Maxima が使い上げた中心核のブロック数を知せます (固定ファイル無しの Maxima は大体 191 ブロックで開始します).

- feature
  システム機能のリストを返す. 現在,mc に対するリストを以下に示します.

  Maxima,noldmsg,maclisp,pdp10,bignum,fasload,hunk,funarg, roman,newio,sfa,paging,mc と its.
  これらの任意の機能は status(feature,···) の第二引数として与えても構いません.

time

time ($\langle$%$o_1\rangle$,$\langle$%$o_2\rangle$,···) $\langle$%$o_i\rangle$ 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 (注：変数 showtime, デフォルト値:[false] 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]
```