REDUCE

ユーザーズマニュアル

バージョン 3.7

Anthony C. Hearn
Santa Monica, CA, USA

Email: hearn@rand.org

1999 年 1 月

目 次

第I部

REDUCE3.7 ユーザーズマニュアル

概要

この文書は数式処理システム REDUCE について説明してある。このシステムは次のような機能を持っている:

1. 多項式や有理式の展開と整理、
2. 多様な形式での項の置き換えならびにパターンマッチ、
3. 自動的ならびにユーザの指定による式の簡約の制御、
4. 記号行列の計算、
5. 任意精度の整数、実数計算、
6. 新しい関数を定義して、プログラムの構文を拡張する機能、
7. 数式の微分や不定積分の計算、
8. 多項式の因数分解、
9. 代数方程式や超越方程式の求解、
10. 式を多様な形式で出力する機能、
11. 入力された数式から、数値計算用のプログラムを出力する機能、
12. 高エネルギー物理学で使われるディラック行列の計算。

謝辞

このバージョンのマニュアルを作成するにあたり多くの人が時間を割いて、以前のバージョンを改善するための意見を寄せてくれ、また新しい章についての原稿を作成してくれた。特に、Gerry Rayna はこのマニュアルの前半の原稿の下書きを作成してくれた。その他、特に John Fitch, Martin Griss, Stan Kameny, Jed Marti, Herbert Melenk, Don Morrison, Arthur Norman, Eberhard Schrüfer , Larry Seward と Walter Tietze からは、多くの寄与をうけた。最後に Richard Hitt は REDUCE のバージョン 3.3 のマニュアルを TEX 型式に書き直してくれた。これはこのマニュアルを LATEX 型式で書くときに大変参考になった。

第1章　はじめに

REDUCE は、非常に複雑な代数計算を正確に実行するシステムである。このシステムは、多項式を展開した形式や因数分解した形式など多様な形式で扱うことができます。また、数式の一部を取り出すことが出来るし、また微分計算や不定積分の計算も可能である。しかし、この章では簡単な例だけに話をとどめて置きます。

その他、配列 (array) の使い方、関数やオペレータを定義する方法、高エネルギー物理学の計算、ファイルの使い方、入力の修正の方法等についても省略します。また計算の過程を制御したり、出力形式、数値の型等を変えるために用意してある多くのオプションについても、その詳細には触れないでおきます。

REDUCE は対話的に使われるように設計されており、ユーザは数式を入力し、次の計算に進む前に今計算させた結果をディスプレイで確認することが出来ます。ユーザは計算した結果から判断して、次にどの様な計算を行うかを決めることができます。対話的な利用をサポートしていないような計算機システム上で利用する場合や、時間のかかる計算だが計算途中でユーザの判断を必要とせず決まった手順で行える場合には、計算手順をファイルに作成して置き、REDUCE をバッチ処理モードで実行させることができます。

ここでは、対話的な利用法についてのみ説明します。なぜならこれが REDUCE の機能をもっともよく概観出来るからです。

REDUCE は起動されると、最初に次の様なメッセージを出力します:

```
REDUCE 3.7, 15-Jan-99 ...
```

これは REDUCE のバージョンとシステムがリリースされた日付を表しています。実際に出力されるバージョンと日付はそれぞれのシステムによってこれとは異なるかもしれません。

それから、次のような入力促進文字列を出力して、ユーザからの入力を待ちます。

```
1:
```

この状態で、 REDUCE のコマンドを入力することが出来ます。入力の最後には式の終わりを意味するセミコロンを付けます。例えば:

```
(x+y+z)^2;
```

入力の最後には別の文字 (普通、キーボードの 改行 キー) を入力する必要があります。REDUCE は入力された式を読み込み、計算した結果を出力します。

```
 2                    2           2
X  + 2*X*Y + 2*X*Z + Y  + 2*Y*Z + Z
```

この簡単な例から、REDUCE が数式をどのように扱うのか見てみることにしましょう。まず、REDUCE は変数と数 (定数) を扱うことができます。計算では、変数の値はそれ自身になっています。数式の計算は、高等学校等で学ぶような普通の数学の規則に従って計算されます。上の例で気が付くのは数式が展開された形で出力されていることです。REDUCE は通常、入力された式を可能な限り展開し、同じ項をまとめ、変数に関してある順番で並べ替えをおこない、その結果を出力します。しかし、展開するかどうかや変数の順番、出力の形式等はユーザが変更することが可能で、これを行うための宣言文等が用意されています。

他にこの例で気が付くのは、入力で小文字を使っているのに、出力が大文字になっていることです。実際、REDUCE では入力は大文字でも小文字でも可能で、小文字で入力した場合には大文字に変換されて読み込みます。ただし、システムによっては小文字が標準になっていて、大文字で入力しても小文字に変換するシステムもあります。このマニュアルでは、入力は小文字で表し、出力は大文字で表すことにします。しかし、文中に現れる一文字の変数等については見やすさを考えて大文字で表しています。

最後に、入力の促進文字として数字が使われており、この番号を使って前に行った計算結果を参照することが出来るようになっています。

ほかの例をやってみましょう。次のように入力してみてください。

```
for i:= 1:40 product i;
```

結果は 40!(40 の階乗) で、

```
815915283247897734345611269596115894272000000000
```

となります。同じ結果は、次のように入力しても得られます。

```
factorial 40;
```

代数計算では、近似値ではなく正確な結果が必要になってくるので、整数の計算が上の例のように任意の桁数で計算できることは非常に重要です。さらに、上記の例で繰り返しを表すのに使われている FOR 文のように、REDUCE はユーザがプログラムを作成する時に便利な機能を多く用意しています。

REDUCE に用意された多くのオプションの中には REDUCE で操作する数値の形式を、任意に指定した精度 (例えば 100 桁) まで計算するような多倍長浮動小数点数モードに切り替えるオプションもあります。

多くの場合に、前に計算した結果を後の計算で使うことが必要になります。これを行う一つの方法として、適当な変数に結果を保存しておくことができます。例えば、

```
u := (x+y+z)^2;
```

として計算結果を変数 U に代入して置き、後の計算で U という変数を使うと、その値は上式の右辺の値が使われます。

また、REDUCE で行った全ての計算の結果は、WS (作業領域 (WorkSpace) の省略です) という名前の変数に保存されています。直後の計算でこの変数を使うと、その変数には直前の計算結果が値として代入されています。

例えば、上での例を実行した直後に、

```
df(ws,x);
```

と入力すると、(x+y+z)^2 という数式を変数 X に関して微分した結果を返します。またこの入力の代わりに、

```
int(ws,y);
```

と入力すると、今度は同じ数式を Y に関して積分した結果を返します。

REDUCE ではまた記号行列の操作ができます。例えば、

```
matrix m(2,2);
```

は M を 2 行 2 列の行列として宣言します。そして、

```
m := mat((a,b),(c,d));
```

は、行列の各要素の値が代入されます。M を含む式は行列としての計算が行われます。つまり、1/m は逆行列を、2*m - u*m^2 は行列と通常の式との積及び行列と行列の積を、そして det(m) では行列 M の行列式 (determinant) が計算されます。

REDUCE は多様な置き換えの機能を持っています。このシステムには初等関数 (三角関数や指数関数、対数関数等) についての知識は入っていますが、これらの関数に関するよく知られた性質や恒等式は自動的には使われません。例えば、三角関数の積を自動的に和に直すことはしません。しかし、ユーザがこの変換を行いたいなら、例えば次のように入力すればできます。

```
(sin(a+b)+cos(a+b))*(sin(a-b)-cos(a-b))
    where cos(~x)*cos(~y) = (cos(x+y)+cos(x-y))/2,
          cos(~x)*sin(~y) = (sin(x+y)-sin(x-y))/2,
          sin(~x)*sin(~y) = (cos(x-y)-cos(x+y))/2;
```

ここで変数 X と Y の前に付けられたチィルダ (~) の文字は、この変換ルールでの変数 X と Y は任意の変数もしくは任意の式を表していることを意味しています。つまり、関数 sin や cos の引数がどんな変数や式であっても、その引数を X や Y と思って右辺の形に変形せよということを意味しています。この計算を実行した結果は、

```
-(COS(2*A) + SIN(2*B))
```

となります。

その他、よく使われるシステムの機能として、数式を FORTRAN 形式で出力する機能があります。これは数多く用意されている REDUCE の出力形式の一つでもあります。これで出力した結果は、FORTRAN 言語による数値計算のプログラムとして使うことができます。この機能は、数値計算で使う代数式を生成するのに利用することができます。

例えば、次の実行文は

```
on fort;
df(log(x)*(sin(x)+cos(x))/sqrt(x),x,2);
```

次のような出力を生成します。

```
 ANS=(-4.*LOG(X)*COS(X)*X**2-4.*LOG(X)*COS(X)*X+3.*
 . LOG(X)*COS(X)-4.*LOG(X)*SIN(X)*X**2+4.*LOG(X)*
 . SIN(X)*X+3.*LOG(X)*SIN(X)+8.*COS(X)*X-8.*COS(X)-8.
 . *SIN(X)*X-8.*SIN(X))/(4.*SQRT(X)*X**2)
```

これまでは REDUCE の代数モード (algebraic モード) での代数式の操作について説明してきました。REDUCE は Standard Lisp という Lisp 言語の一つで書かれています。この REDUCE の基礎となっている Lisp 言語でのプログラムを実行する、記号モード (symbolic モード) というモードも用意されています。このモードでは Lisp の構文に基づいて処理が行われます。例えば、

```
symbolic car '(a);
```

また二つのモード間でデータのやり取りを行うこともできます。

これまでの簡単な説明で、REDUCE はどの様なシステムであるか分かってもらえたと思います。REDUCE の持っている全ての機能については、2 章以下に説明してあります。もし、チュートリアルが必要であれば、R. Stoutemyer による七つのレッスンプログラムが役に立つでしょう。これは、REDUCE と一緒に配布されています。

第2章　プログラムの構造

REDUCE のプログラム は、計算機によって順に評価される関数やコマンドの列からなっています。コマンドは、宣言文、実行文と数式から構成されています。これらの要素は、数字の列、変数、オペレータ、文字列、予約語と句切り記号 (コンマや括弧等) からなっています。これらはすべて標準文字で表されます。

2.1　REDUCE の標準文字集合

REDUCE のシンボルを構成している標準文字は次のものです:

1. a から z までの 26 文字
2. 0 から 9 までの数字 10 文字
3. 特殊文字 _ ! ” $ % ’ ( ) * + , - . / : ; < > = { } < 空白 >

文字列中や感嘆符をつけた文字以外は入力された文字が大文字でも小文字でも同じ文字として扱われます。ALPHA, Alpha や alpha はすべて同じ名前を表しています。使用している Lisp 処理系により、入力された文字は内部で大文字もしくは小文字に変換されます。多くの処理系ではこのような変換を行なわないようにできます。これについては、使用しているシステムの説明書を参照してください。このマニュアルでは、使用しているシステムに依存しないよう、使用する文字としては上記の標準文字集合に入っている文字だけを使うことにします。

2.2　数値

REDUCE で使える数値の型としては、いくつかの型が使用できます。**整数 (integer)** は小数点の付かない数字の列からなるもので、先頭には符号を付けてもかまいません。例えば:

```
-2, 5396, +32
```

原則として、たいていのインプリメントで整数の桁数に実際上の制限はありません。(しかし、この点については、使用しているシステムの使用説明書で確認して下さい。) 例えば、2^{2000} の値を計算させると、603 桁の数値が得られます。これは通常の表示で約 9 行になります。もっとも、このような多倍長での計算は時間がかかるということに注意して置いた方がよいでしょう。

整数でない数値は通常二つの整数の商として表されます。これは**有理数 (rational)** と呼ばれています。

REDUCE では計算を浮動小数点数を使った近似計算 (任意に指定した精度で) で行うように指定することもできます。これらの数は**実数 (real)** と呼ばれています。実数を入力するのには次の三つの方法があります。

1. 先頭に符号の付いた、もしくは符号の付かない数字の列で、途中もしくは最後に小数点をつけたもの。
2. 1 と同じで、その後に指数部を指定したもの。これは、文字 E に続いて符号付きまたは符号の付かない整数で表す。
3. 最初に符号の付いた、もしくは符号の付かない数字の列で、その後に文字 E に続く指数部を指定したもの。

例えば、 `32.  +32.0 0.32E2` や `320.E-1 320E-1` 等はすべて実数 32.0 を表します。

浮動小数点数の出力は `SCIENTIFIC_NOTATION` 宣言で制御されます。通常は 1000000 以下の数値は `12345.6` のように固定小数点形式で出力されます。これ以上の数値は `1.234567E+5` のように指数形式 (科学技術計算形式) で出力されます。同様に、10^{-12} 以下の数値は指数形式で出力されます。この出力を変更するには、`SCIENTIFIC_NOTATION` を次の二つの方法のいずれかで行います。`SCIENTIFIC_NOTATION` m; ただし m は正の整数、では m 桁以上の数値または小数点以下 m 以上零である数値は指数形式で出力されるようになります。`SCIENTIFIC_NOTATION` $\{m,n\}$, ただし m と n は両方とも正の整数、では m 桁以上の数値もしくは小数点以下 n 個以上零であるような数値は指数形式で出力されるようになります。

注意: 数字の先頭に小数点をつけたものは記号モードでの `CONS`(.) オペレータと混同されるので許されません。`.5 -.23 +.12` は許されず、代わりに `0.5 -0.23 +0.12` としなくてはいけません。

2.3 識別子

REDUCE における識別子 (名前) は最初アルファベットで始まり、以後アルファベットもしくは数字の列から構成されます。識別子を構成する文字の最大数はインプリメントによって変わってきますが、ほとんどのインプリメントでは少なくとも 24 文字以上使うことができます。また、識別子の文字中に限っては下線文字 (_) も文字として使うことができます。例えば、

```
a az p1 q23p  a_very_long_variable
```

は全て識別子になります。しかし、

```
_a
```

は識別子にはなりません。

数字で始まる英数字の列は、数値と変数との積として解釈されます。例えば、`2ab3c` は `2*ab3c` として解釈されます。注意しないといけないのは数字の直後の文字が E の時です。このような場合にはシステムは実数として解釈しようとします。例えば、`2E12` は実数の $2.0 * 10^{12}$ です。`2e3c`

は 2000.0*C となります。また 2ebc では文字 E の直後の文字が数字 (これは指数部を表す数値と解釈されます) でないのでエラーとなります。

特殊文字、例えば –, * や空白文字、を識別子名の中に使うこともできます。ただしそのような識別子を入力するときには特殊文字の前に感嘆符 (!) をつけないといけません。例えば、

```
light!-years    d!*!*n          good! morning
!$sign          !5goldrings
```

注意: システムで使用している識別子には、特殊文字 (特に* や =) を含んだ名前がよく使われています。もし、ユーザが自分で使うための識別子の名前として、システムで使用している識別子の名前と同じ名前を使うと重大なエラーを起こすかも知れません。したがって、ユーザはこのような名前を使わないように注意しないといけません。

識別子は変数名やラベル名、配列名、オペレータ名、関数名として使われます。

制約事項

次節に予約語のリストを挙げて置きます。これらは識別子としては使えません。識別子の中には空白があってはいけません。また識別子は行を越えて続けることはできません。

2.4 変数

全ての変数 は識別子として名前が付けられており、特別な型が与えられています。通常のユーザはこの型について気にする必要はありません。通常の変数にはスカラー型と呼ばれる既定の型が付けられます。この型の変数はその値として通常の代数式を代入することができます。また、変数に値が代入されていなければ (未定義の変数)、その変数名自身を値として持ちます。

予約変数

いくつかの変数名 は REDUCE では特別な意味を持っており、ユーザが勝手には使えないものがあります。これらの変数には次のものがあります:

E　自然対数の底を表す。 log(e) が式の中に現れるとこれは 1 に簡約される。もし ROUNDED フラグがオンであれば、E は設定された桁数の浮動小数点数での近似値で表される。

I　虚数単位。 -1 の平方根を表す。i^2 は-1 で置き換えられる。(これは、シンボル I がトップレベルで使われたときであり、関数等の仮引数に現れたときや、ローカル変数や for 文で for i:= ... のように使われたときには単なる変数として使われます。)

INFINITY　極限 (limit 演算) やべき級数の計算等で無限大数 ∞ を表す。しかし、現在のところ数学でいうところの無限大数とは異なり、例えば $\infty + \infty = \infty$ ではなく infinity + infinity は 2*infinity となる。

NIL　REDUCE (代数モードのみ) では 0 を表す。したがって、NIL は変数としては使えない。

PI　円周率 π を表す。 ROUNDED スイッチがオンであれば、浮動小数点数での近似値で置き直される。

T　記号モードで論理値の真を表す。関数等の仮引数名やローカル変数名として使えない。

これら以外の予約変数名、例えば LOW_POW のように別の章で説明されているもの、については付録 A にリストを載せてあります。

これらの予約変数を別の目的で使ったり、不用意に使った場合にはエラーを起こすかも知れません。

これら以外に、 REDUCE の内部で使われている変数で、同様な使用上の制限があるものがあります。これらの変数名には通常星印 (*) が使われています。したがって、初心者がこのような変数名を使うことは進められません。このような変数名には、例えば漸近コマンドのパッケージで使われる K!*という変数名があります。

ある名前は REDUCE で予約されており、一般に使えないものがあります。これらの変数名のリストは “予約識別子”の節に書いてあります。このような、使用上の制限のある名前のリストは非常に多いのでここに全てを書き表すことはできません。読者はこのような名前のリストから自分が誤って使ってしまう可能性のあるような名前のリストを各自作成し、プログラムを作成する時にそれを参照するようにした方がよいでしょう。

2.5 文字列

文字列 は WRITE 文でメッセージ等を出力する時や、ファイル名を指定するとき等で用いられます。文字列はダブルクォート (") で囲まれた任意の文字の列からなります。例えば、

```
"A String".
```

文字列中の小文字は大文字には変換されません。

文字列 "" は空の文字列を表します。文字列の中にダブルクォートを使いたいときには、ダブルクォートを前に付けます。つまり"a""b" は文字列 a"b を、そして """" は文字列 "を表します。

2.6 注釈

プログラム 中に人間に対する注釈 (REDUCE はこの注釈を無視する) として文章を挿入して置くことができます。この注釈を書く方法として二つあります。

1. COMMENT で始まり、文の句切り記号 (; または $) までの文章は注釈として無視されます。この注釈は空白が置けるような場所にはいつでも置くことができます。(END と >> は注釈の句切り記号としては解釈されません。)

2. 文字 % から行の終わりまでは何が現れても無視されます。この注釈は行の最後に挿入することができます。文の終端記号は意味を持ちません。もし、同じ行に実行文がある場合には%記号の前に終端記号を忘れないようにしないといけません。複数の行を注釈文にするときには各行の先頭に%記号を付けないといけません。

2.7 演算子 (オペレータ)

REDUCE での演算子 (オペレータ) は名前と型で特徴付けられています。型には内挿演算子 (infix) と前置演算子 (prefix) の二つの型があります。演算子はまったく抽象的な演算子 (シンボルのみが定義されているだけで、演算の性質がまったく未知の演算子) というのも定義できます。また、特定の引数に対する演算の結果を、代入文 (:=による) もしくは単純な LET 文によって指定したもの、ある条件を満足するような引数に対する結果をもっと一般的な LET 文で定義したもの、それから、プロシジャ宣言文によって、演算子の性格を完全に定義したもの、というようにいろいろなレベルで定義することが可能です。

内挿演算子は互いに優先順位が定義されています。そして、演算子はその引数の間にいれます。例えば、

```
a + b - c          (空白はなくてもよい)
x<y and y=z        (AND の前後には一つ以上空白が必要)
```

演算子と変数の間および演算子と演算子の間には自由に空白をいれてもかまいません。ただし演算子の名前が文字で表されているときには (例えば AND のように)、変数 (例えば Y) との間や同じ様な演算子との間に、空白を一つ以上いれなくてはいけません。

前置演算子は引数の前に付けられます。通常の数学関数のように引数は括弧でくくり、それぞれの引数はコンマで区切ります。

```
cos(u)
df(x^2,x)
q(v+w)
```

括弧が合わなかったり、内挿演算子と間違ってグルーピングされているときなどはエラーになります。もし、引数が一つの演算子でその引数が一個のシンボルの場合や前置演算子で始まっているような場合には、括弧を省略することができます。

```
cos y              cos(y) を意味する
cos (-y)           括弧が必要
log cos y          log(cos(y)) を意味する
log cos (a+b)      log(cos(a+b)) を意味する
```

しかし

```
cos a*b            (cos a)*b を意味する
cos -y             はエラー。変数 "COS" 引く変数 y と解釈される。
```

単項の前置演算子はどんな内挿演算子 (単項の内挿演算子も含めて) よりも高い優先順位を持っています。つまり、REDUCE は `cos y + 3` は `cos(y + 3)` ではなく、常に `(cos y) + 3` と解釈します。

内挿演算子はまた、入力するときに前置形式で使うこともできます。例えば、`+(a,b,c)` と入力することができます。出力では、このような式は常に内挿形式で (つまり、`a + b + c` のように) 出力されます。

多くの前置演算子がすでにシステムに組み込まれています。ユーザは新しい演算子を定義することもできます。組み込みの演算子については節を改めて説明します。

組み込みの内挿演算子

次の内挿演算子 はシステムに組み込まれています。これらはすべて内部でプロシジャとして定義されています。

```
<内挿演算子>   ::=  where|:=|or|and|member|memq|=|
                    neq|eq|>=|>|<=|<|+|-|*|/|^|**|.
```

これらの演算子は次のように分類されます。

```
<代入演算子>   ::=  :=
<論理演算子>   ::=  or|and|member|memq
<関係演算子>   ::=  =|neq|eq|>=|>|<=|<
<置換演算子>   ::=  where
<代数演算子>   ::=  +|-|*|/|^|**
<構成演算子>   ::=  .
```

`MEMQ` と `EQ` は REDUCE の代数モードでは使えません。これらについては記号モードの節で説明します。`WHERE` は置き換えの項で説明します。

REDUCE の前の版では *not* が内挿演算子として定義されていました。これは、新しい版では通常の前置演算子になっており、*null* 演算子で置き換えることが可能です。

REDUCE で使っている中間言語との互換性から、特殊文字で表されている内挿演算子は文字で表される別名をもっています。これらの識別子は対応する特殊文字の代わりにこの別名を入力で使うこともできます。それぞれの別名は次のようになっています。

```
:=        setq          (代入演算子)
=         equal
>=        geq
>         greaterp
<=        leq
<         lessp
+         plus
-         difference  (もし単項演算のときには minus)
```

*	times	
/	quotient	(もし単項演算のときには recip)
^	expt	(べき乗演算)
**	expt	(べき乗演算、^と同じ)
.	cons	

注意: NEQ は *not equal* の意味です。これに対する特殊文字による演算子は有りません。

以上の演算子 は 2 項演算子です。ただし、NOT は単項演算子です。また、+ や * は n 項演算子で任意の数の引数を取ることができます。- や / は単項演算子としても使えます。そのときには /は逆数を取る演算子となります。例えば、/2 は 1/2 と同じ意味になります。全ての 2 項演算子は左結合の規則に従って解釈されます。つまり、a/b/c は (a/b)/c として解釈されます。ただし、:=と . は右結合で、例えば、a:=b:=c は a:=(b:=c) と解釈されます。ALGOL や PASCAL 言語とは異なり、^は左結合です。つまり、 a^b^c は (a^b)^c と解釈されます。

演算子 <, <=, >, >= は数値の比較にのみ使用できます。一般の数式に対する関係演算は定義されていません。

演算の順序を変更するために括弧を使うことができます。もし、括弧がない場合には、演算はこの節の最初に挙げた<内挿演算子>の表での順序で優先順位が決まります。後の方が順位が高くなります。つまり、WHERE が一番順位が低く、.(ドット演算子) が一番順位が高くなります。

第3章　数式

REDUCE の数式はいくつかの型に分けられ、数値、変数、演算子、括弧とコンマからなります。最も一般的な型は次のものです。

3.1　スカラー式

スカラー式は数値と通常の “スカラー” 変数 (識別子)、添字付きの配列名、演算子や関数名に引数をつけたものや実行文と 代数演算子 + - * / ^ から成る式です。例えば、

```
x
x^3 - 2*y/(2*z^2 - df(x,z))
(p^2 + m^2)^(1/2)*log (y/m)
a(5) + b(i,q)
```

べき乗の記号 (^) の代わりにシンボル ** を使うこともできます。(これはシステムによって変わっているかも知れません。)

文 (通常括弧で囲まれて) をスカラー式の一部に使うことができます。例えば、

```
w + (c:=x+y) + z .
```

数式の代数値が必要になったとき、REDUCE は次のようにしてその値を決定します。

変数や引数の指定された演算子は最後に代入された値を持ちます。もし、それらに対して何も代入されていなければ、それ自身を値として持ちます。しかし、配列の要素については同じく最後に代入された値を持ちますが、もし一度も代入されていなければゼロとして扱われます。

関数は実引数に対して値が評価されます。

数式を評価していくときには通常の数学の規則が適用されます。しかしながら、数値計算の場合とは違って数式計算の場合には、計算に曖昧さが残ります。この計算過程は、“簡約 (simplification)” と呼ばれています。つまり、REDUCE において代数式を計算するということは通常、数式を最も簡単な形に変形した結果を出力するからです。

このような簡約を行うときに、ユーザが指定できる多くのオプションがあります。もし、ユーザが何も指定しないときには、既定の方法が選ばれます。既定の状態では、数式を展開し、同類項をまとめ、項を並べ変え、LET 文等で指定された規則にしたがって演算子やプロシジャ、関数等の評価を行います。多くの場合ではこのような計算で十分です。

計算のやり方を制御するオプションの使い方についてはまた別の節で詳しく説明します。例えば、もし計算の中で実数 (浮動小数点数) が出てきた場合、システムはふつうこの数を有理数に変

換します。もしユーザが実数のままで計算をしたいならば on rounded; というコマンドを入力することでできます。係数体として他にどの様な数体が使えるかについてはまた別の節で説明します。

もし、計算の途中で定義されていない演算を行おうとした場合 (例えば、ゼロでの割り算等) や、関数の引数の数が合わないというような場合には、それぞれに対するエラーの表示が出ます。

3.2 整数式

整数式とは 数式中の定数値や変数の値によって、計算した結果、整数値になるような数式のことをいいます。

例:

```
2,      37 * 999,       (x + 3)^2 - x^2 - 6*x
```

は明らかに整数値を取ります。また

```
j + k - 2 * j^2
```

は J と K が整数値を取るとき、または “変数と数値の小数部分が打ち消し合う” 場合には、整数値になります。例えば

```
k - 7/3 - j - 2/3 + 2*j^2.
```

は J,K が整数値であれば、整数式になります。

3.3 論理式

論理式 は真偽値を取ります。REDUCE の代数モードでは、論理式は次のような構文を取ります。

<数式> <関係演算子> <数式>、

<論理演算子> (<引数>)

または

<論理式> <論理演算子> <論理式>.

式の演算順序を変えるために括弧を使うことができます。

内挿演算子の所で述べた論理演算子と関係演算子に加えて、次のような論理演算子が定義されています:

EVENP(U) 数値 U が偶数かどうかを判定する;

FIXP(U) 数式 U が整数かどうかを判定する;

FREEOF(U,V) 数式 U がカーネル V を含まないかどうかを判定する;

NUMBERP(U) U が数値かどうかを判定する;

ORDP(U,V) U が順序として V より前かどうか判定する。この順序は式の構造または識別子の内部順序に基づく順序によって決まる;

PRIMEP(U) もし U が素数であれば真.

例:

```
j<1
x>0  or  x=-2
numberp x
fixp x and evenp x
numberp x and x neq 0
```

論理式は IF, FOR, WHILE と UNTIL 文でのみ使うことができ、通常の式の中に使うことや、変数に代入することはできません。

注意: 記号モード (シンボリックモード) に詳しい人は、これらの演算子のうちには記号モードでは異なった動作をするものがあることに注意してください。例えば、NUMBERP は記号モードでは引数が整数、もしくは実数の場合にのみ真を返します。

二つ以上の論理式が AND 演算子で結合されている場合、各論理式は**偽**である論理式が出てくるまで最初から順に評価されます。途中の論理式が偽になった場合、それ以後の論理式は評価されず、全体の論理式の値は偽になります。従って、

```
numberp x and numberp y and x>y
```

では x>y は X と Y が共に整数の場合のみ評価されます。

同じように、OR 演算子で結合された論理式は、**真**のものがでてくるまで順に評価されます。

注意: 論理式および論理値が要求されるような所では、数式の 0 は偽と解釈されます。それ以外の全ての数式は真として解釈されます。従って、代数モードで論理式の中に直接使えるような関数は、例えば値として 0 か 1 を返すような関数として、定義することができます。

```
procedure polynomialp(u,x);
   if den(u)=1 and deg(u,x)>=1 then 1 else 0;
```

このように定義した関数は、論理式の中で次のように使うことができます。

```
if polynomialp(q,z) and not polynomialp(q,y) then ...
```

さらに、返すべき値を定義していない関数等 (ブロック文で RETURN 文のないもの) は、論理値としては偽の値を持ちます。

3.4 等式

等式 とは特別な型の数式で次の様な構文を取ります。

```
<左辺式> = <右辺式>.
```

等式は論理式としても使われますが、それ以外にも関数の引数 (例えば、SOLVE 演算子) や関数の返す値として使われることがあります。

普通、右辺式は評価されますが、左辺式は評価されません。もし両辺の式を評価して欲しければ、EVALLHSEQP スイッチをオンにしなければなりません。

等式を操作するために、二つの演算子 LHS と RHS が用意されています。これらの演算子は等式の左辺値と右辺値をそれぞれ取り出します。例えば、

```
lhs(a+b=c) -> a+b
```

```
rhs(a+b=c) -> c
```

となります。

3.5 数式としての実行文

実行文の中には文の値として、ある種類の代数式もしくは数値を持ち、従って式の一部に数式として使えるものがあります。例えば、代入文は文の値として代入した数式そのものを返します。従って、

```
2 * (x := a+b)
```

は 2*(a+b) になり、“副作用” として数式 a+b が変数 X に代入されます。

```
y := 2 * (x := a+b);
```

を実行すると、X には a+b が、Y には 2*(a+b) がそれぞれ代入されます。

Proper 文の節では、様々な数式として式の一部に使用することができる実行文について説明しています。

第4章　リスト

リスト は要素の列からなるもので、要素はそれ自身リストであっても構いません。リストは中括弧で囲まれ、要素間はコンマで区切られています。リストの例:

```
{a,b,c}

{1,a-b,c=d}

{{a},{{b,c},d},e}.
```

空リストは次の様に表されます。

```
{}.
```

4.1　リスト演算

いくつかの演算子は結果としてリストを返します。また、ユーザは中括弧とコンマを使って新しいリストを作ることができます。それとは別のやり方として、LIST 演算子を使って LIST を作ることもできます。リストに対する演算子として重要なものに MAP と SELLECT があります。これらの演算子に対する詳しい説明は, MAP, SELECT および FOR コマンドの章を見て下さい。ASSIST パッケージの中にも関連する項目があります。

このようなリストを操作するために、いくつかの演算子が用意されています。例えば、`PART(<リスト>,n)` はリストの n 番目の要素を取り出します。`LENGTH` はリストの長さ (要素の数) を返します。リストに特有の演算子もあります。これらの演算子は Lisp 言語のリスト操作関数に対応しています。それらの演算子は次のものです。

4.1.1　LIST

LIST 演算子は中括弧を使うのと同じことを行ないます。LIST は、任意個の引数を取り、その引数からなるリストを返します。この演算子は、演算子名を引数に渡す必要がある場合に有用です。例えば、

```
list(a,list(list(b,c),d),e);        ->  {{a},{{b,c},d},e}
```

4.1.2 FIRST

この関数は リストの先頭の要素を返します。もし、引数がリストでなかったり、空リストの場合にはエラーになります。

4.1.3 SECOND

SECOND はリストの第二要素を返します。もし引数がリストでなかったり、第二要素を持たない場合にはエラーになります。

4.1.4 THIRD

この関数は リストの第三要素を返します。もし引数がリストでなかったり、第三要素を持たなかった場合にはエラーになります。

4.1.5 REST

REST は引数のリストから先頭の要素を取り除いた残りのリストを返します。もし、引数がリストでなかったり、空リストの場合にはエラーになります。

4.1.6 . (Cons) 演算子

この演算子はリストの先頭に要素 (式) を追加 (“コンス”) します。例えば、

```
a . {b,c}      ->   {a,b,c}.
```

4.1.7 APPEND

この演算子は第一引数のリストに第二引数のリストを加えた新しいリストを返します。

例:

```
append({a,b},{c,d})      ->      {a,b,c,d}
append({{a,b}},{c,d})    ->      {{a,b},c,d}.
```

4.1.8 REVERSE

REVERSE 演算子は引数のリストから、各要素を逆に並べた新しいリストを作ります。逆に並べられるのは一番上位の要素についてのみで、要素がまたリストになっている場合、その要素の中身は変更されません。例えば、

```
reverse({a,b,c})        ->     {c,b,a}
reverse({{a,b,c},d})    ->     {d,{a,b,c}}.
```

4.1.9 リストを引数とする演算子

リスト操作として定義されたもの以外の演算子に引数としてリストを与えた場合、結果はリストの各要素に演算を行ったリストが返されます。例えば、log{a,b,c} を評価すると、{LOG(A), LOG(B), LOG(C)} が返されます。

このような演算の分配を禁止するには二つの方法があります。一つは、LISTARGS スイッチをオンにすることです。こうすれば、このような演算の分配はすべて禁止されます。もう一つの方法は、指定した演算子に LISTARGP と宣言することにより、特定の演算子に対してのみ分配を禁止する方法です。例えば、listargp log; と宣言すると、log{a,b,c}; は LOG({A,B,C}) となります。

引数を二つ以上取るような演算子 (関数) に対しては、このような演算の分配は起こりません。

4.1.10 注意と例

幾つかのリスト演算、例えば *member* や *delete* は ASSIST パッケージをロードした後でないと利用できません。

注意すべきことは、CONS への第二引数にリスト以外のもの (LISP でのドット対) を指定することはできません。この場合、”リストではない”というエラーを起こします。

```
a := 17 . 4;

***** 17 4 invalid as list
(***** 17 4 はリストでない)
```

また、スカラー変数の初期値は空リストではありません。次の例のように、必要なら明示的に空リストに設定する必要があります。

例:

```
load_package assist;

procedure lotto (n,m);
 begin scalar list_1_n, luckies, hit;
    list_1_n := {};
    luckies := {};
    for k:=1:n do list_1_n := k . list_1_n;
    for k:=1:m do
      << hit := part(list_1_n,random(n-k+1) + 1);
         list_1_n := delete(hit,list_1_n);
```

```
          luckies := hit . luckies >>;
     return luckies;
  end;                              % In Germany, try lotto (49,6);
```

別の例: 多変数多項式に対して、リストで与えた変数名に関する全ての係数を求める。

```
procedure allcoeffs(q,lis); % q : polynomial, lis: list of vars
   allcoeffs1 (list q,lis);

procedure allcoeffs1(q,lis);
  if lis={} then q else
    allcoeffs1(foreach qq in q join coeff(qq,first lis),rest lis);
```

第5章　文

文 は予約語と数式からなっており、次の構文を取ります。

```
<文> ::= <式>|<Proper 文>
```

REDUCE のプログラムは文の後に終端記号を付けたコマンドの列からなっています。

```
<終端記号> ::= ;|$
```

プログラムをどのように複数の行に分割するのかはまったく自由で、複数の文を一行に書いても、また一つの文を複数の行に分割しても構いません。もしプログラムが対話的に動いているときには; または$で終わっている文を入力しても、改行キーが押されるまでは実行されません。この改行キーはシステムによって変わってきます。しかし、普通 改行 (RETURN) キーが、これに相当しています。

もし文が Proper 文であれば対応する動作が実行されます。

Proper 文の性格に応じて、実行した結果もしくは応答が出力されるかどうかが決まります。応答 (計算結果ではなく演算子を実行したときに出力されるメッセージ) は終端記号として何を使用したかに関係なく出力されます。

もし文が式であるなら、それは評価されます。さらに、終端記号がセミコロンであれば結果が出力されます。終端記号がドル記号のときには結果は出力されません。評価した結果として得られる数式の大きさを前もって見積ることは通常不可能なので、出力形式の詳細をユーザが明示的に指定することはできません。しかしながら、多くの種類の出力形式を指定する為のコマンドが使用可能で、異なった形式での出力が可能となっています。これらの出力形式の宣言文に付いては 8.3.3 節で説明されています。

次の節では REDUCE での Proper 文の型について記述しています。

5.1　代入文

代入文の構文は次の通りです。

```
<代入文> ::= <数式> := <数式>
```

左辺の <数式> は通常、変数名、オペレータ名に引数のリストを指定したものまたは配列名で添字を整数値で指定したものになっています。例えば、

```
a1 := b + c
```

```
h(l,m) := x-2*y          (ただし h はオペレータ名)
k(3,5) := x-2*y          (ただし k は 2 次元の配列名)
```

もっと一般的な a + b := c のような代入 も許されます[1] 。このような代入文での振舞いに付いては 10.2.5 節で説明されています。

代入文では右辺の式は評価されます。もし左辺が変数であれば、右辺を評価した値が変数に代入されます。左辺がオペレータや配列であれば、オペレータや配列の引数のみが評価されます。これ以上の評価は行われず、結果の式に右辺を評価した値が代入されます。例えば、A は一次元の配列であるとすると、a(1+1) := b は B の値が配列の要素 a(2) に代入されます。

もし、代入文で、終端記号としてセミコロンが使われ、コマンドとして実行された場合 (つまり、グループ文やプロシジャ文等で使われるような場合以外の時) には、左辺の式が出力され、続いて ":=" が出力され、それから右辺の値が出力されます。

複合代入文を使うことができます。

```
<数式> := ... := <数式> := <数式>
```

この形式では、一番最後の<数式>を除くそれぞれの<数式> に最後の数式を評価した値が代入されます。終端記号としてセミコロンを使うと、最後を除く各数式に ":=" を付けたものが出力され、その後ろに最後の数式を評価した値が出力されます。

5.1.1 SET 文

場合によっては、左辺を評価した結果の値に右辺を代入したいことがあります。これには SET 文を使います。この文の構文は、次の通りです。

```
SET(<数式>,<数式>);
```

例えば、

```
j := 23;
set(mkid(a,j),x);
```

を実行すると変数 A23 に X の値が代入されます。

---

[1] a+b := c では、実際には+ b は右辺に移項されて、a に c-b が代入されます。ただし、代入文としての値は、右辺式の c になっています。したがって、x+y := z+w := c; は、x+y := (z+w := c); と解釈され、x には c-y が、z には c-w が代入されます。

代入文は LET 文とよく似ています。ただ、代入文では左辺は評価されず、右辺のみが評価されます。これに対して、LET 文では両辺とも評価されないそのままの式に対して代入が行われます。

代入文の左辺が評価されないのは、左辺式が変数の場合のみで、関数や式が現れたときには、関数の引数部分は評価されます。例えば、

```
x+y := a+b;
x+y := c;
```

では、代入文の左辺は plus(x,y) と関数 PLUS で表されています。a,b,c,x,y すべて未定義の変数とすると、plus(x,y) は評価しても変わりません。従って、最初の代入文では、x に a+b-y が代入されます。しかし、二番目の代入文では、左辺の x+y を評価した値が a+b-y+y となるので、a に c-b が代入されます。

5.2 グループ文

グループ文は REDUCE の構文上一つの文しか書けない所に、複数の文を書きたいときに使います。グループ文は一つ以上の文を << と >> で囲ったもので、文同士はセミコロンかドル記号で区切ります。このときどちらを使っても同じです。各文は順に実行されます。

この文の例は IF 文等の説明の項で挙げてあります。そのような文では << ... >> が非常によく使われます。

もし、グループ文を構成している最後の文が値を持てば、その値がグループ文の値になります。注意しておかなければいけないことは、グループ文の値が必要なときには、グループ文の最後の文の終わりにはセミコロンやドル記号を付けてはいけないということです。もし、余分な終端記号を付けると、グループ文の値が NIL または 0 になってしまいます。

5.3 条件文

条件文 は次の構文を取ります。

```
<条件文> ::=
   IF <論理式> THEN <文> [ELSE <文>]
```

論理式を評価した結果が真であれば最初の (then 以下の)<文>が実行されます。もし偽であれば二番目の (else 以下の)<文>が実行されます。

例:

```
if x=5 then a:=b+c else d:=e+f

if x=5 and numberp y
   then <<ff:=q1; a:=b+c>>
   else <<ff:=q2; d:=e+f>>
```

ここで、文としてグループ文を使っています。
条件文は右結合です。つまり、

```
IF <a> THEN <b> ELSE IF <c> THEN <d> ELSE <e>
```

は次と同じ意味に解釈されます。

```
IF <a> THEN <b> ELSE (IF <c> THEN <d> ELSE <e>)
```

さらに、

```
IF <a> THEN IF <b> THEN <c> ELSE <d>
```

は

```
IF <a> THEN (IF <b> THEN <c> ELSE <d>).
```

と解釈されます。条件文の実行文に代入文等の副作用を持つような文が含まれず、条件文の返す値のみが使われるような場合、これは条件式と呼ばれます。IF 文での論理式の値によってどの実行文が実行されたかに応じて、実際に実行された文の値がこの条件式の値になります。もし、実行された文が値を持たない文の場合は、条件式の値は (どの様に使われるかによって) 値をもたないか、もしくは 0 を値として持つか、いずれかになります。

例:

```
a:=if x<5 then 123 else 456;
b:=u + v^(if numberp z then 10*z  else 1) + w;
```

もし条件文の値が必要ではなく、また論理式の値が偽の場合には何もしなくてもよいのであれば ELSE 以後は省略できます。

```
if x=5 then a:=b+c;
```

注: 3.3 節で説明したように、論理式でスカラー式や数値が使われたとき、また変数名が書かれているような場合、その式の値が 0 でない限り常に真と解釈されます。

5.4 FOR 文

FOR 文はプログラムでループを記述するのに使われます。この文の構文を次に示します。

$$\texttt{FOR}\left\{\begin{array}{l}\langle\text{変数}\rangle\ \texttt{:=}\ \langle\text{初期値}\rangle\ \left\{\begin{array}{c}\texttt{STEP}\ \langle\text{増分値}\rangle\ \texttt{UNTIL}\\ \texttt{:}\end{array}\right\}\langle\text{終値}\rangle \\ \texttt{EACH}\ \langle\text{変数}\rangle\left\{\begin{array}{l}\texttt{IN}\\ \texttt{ON}\end{array}\right\}\ \langle\text{リスト}\rangle\end{array}\right\}\langle\text{動作}\rangle\ \langle\text{式}\rangle$$

ここで、

$$\langle\text{動作}\rangle\ \texttt{::= do|product|sum|collect|join.}$$

FOR 文での代入形式はループを指定された初期値から初めて、終値に達するまで繰り返すことを指定します。もし、ここの数式を評価した結果が数値でなければエラーになります。

FOR 文の FOR EACH 形式はリストで与えられた各要素に対して繰り返しを実行します。もし、<リスト>がリストでなければエラーになります。

<動作>が DO の時には <式> は単に評価されるだけでその値は捨てられます。FOR 文の値はこの時 0 になります。COLLECT の場合にはそれぞれの<式>を評価した結果を要素とするリストが返されます。JOIN の場合には、 <式>を評価した結果がそれ自身リストで、各々のリストを足し合わせたリストが FOR 文の値になります。

最後に、PRODUCT と SUM の場合にはそれぞれ<式>を評価した値の積または和を求めます。

いずれの場合においても、<式>は代数式として評価され、その中に現れる<変数>はそのときのループ変数の値で置き換えられます。もし<動作>が DO のときには、評価するだけでそれ以外は何

もしません。それ以外の場合には、<動作> は二項演算子として結果を構成し、FOR 文の値として返されます。ループの値は最初 SUM の場合には 0、PRDUCT の場合には 1、それ以外の場合には空リストに初期設定されます。終了条件はループの最初に調べられます。ループ変数の初期値が終了条件を満たしている場合、もしくは FOR EACH 文の場合で<リスト>が空リストの場合、<式> は一度も実行されることなく FOR 文を終了します。

例:

1. A, B が配列として定義されているものとして、次の例を実行すると A(5) から A(10) には 5^2 から 10^2 がそれぞれ代入されます。また、B(5) から B(10) には三乗した値が代入されます。

```
for i := 5 step 1 until 10 do <<a(i):=i^2; b(i):=i^3>>
```

2. 増分が 1 の場合というのは多く使用されるので、

```
STEP 1 UNTIL
```

はコロンで代用できるようになっています。従って、上の例は、

```
for i := 5:10 do <<a(i):=i^2; b(i):=i^3>>
```

と書くことができます。

3. 次の例では C には 1、3、5、7、9 の自乗の和が、そして D には x*(x+1)*(x+2)* (x+3)*(x+4) が代入されます。

```
c := for j:=1 step 2 until 9 sum j^2;
d := for k:=0 step 1 until 4 product (x+k);
```

4. 次の例ではリスト {a,b,c} 中の各要素を自乗したものを要素とするリストが返されます。

```
for each x in {a,b,c} collect x^2;
```

5. 次の例では、リスト {a,b,c} の各要素を自乗した値を要素とするリストをリストにして返します。(つまり、{{A^2},{B^2},{C^2}}) を返します。)

```
for each x in {a,b,c} collect {x^2};
```

6. 次の例ではやはり要素を自乗したものを要素とするリストを返します。JOIN はそれぞれのリストを結合して (join)、一つのリストにします。

```
for each x in {a,b,c} join {x^2};
```

FOR 文での制御変数はまったく新しい変数で、FOR 文の外で使われている同じ名前を持つ変数とは違うものになります。言い替えれば、for i:= ... はシステムで定義している変数 i(虚数単位 $i^2 = -1$) 自身は変更しません。(しかし、この FOR 文の中では、変数 i は虚数単位とは違うものと解釈されます。) さらに、代数モードでは、制御変数の値が代入されるのは<式>中に明示的に現れている変数に対してのみ行われます。式を評価した結果として現れる変数 (制御変数と同じ名前の変数) に対しては置き換えは行われません。例えば、

```
b := a; for a := 1:2 do write b;
```

では、1 と 2 が出力されるのではなく、A が二度出力されます。

5.5 WHILE ...DO 文

FOR ...DO 文は繰り返す回数が前もって分かっている場合の繰り返しを記述するのに使われます。繰り返しの条件がもっと複雑になってくると WHILE ...DO がよく使われます。WHILE 文の構文は、

```
WHILE <論理式> DO <文>
```

で、<論理式>の値が真である間、<文>が実行されます。WHILE ...DO は DO の後には一つの文 (実行文) のみが書けます。もし、複数の文を DO 以降に書きたいときには、以下の例のように << と >> で囲ったグループ文を使うか、それとも BEGIN ...END のように BEGIN と END で囲ったブロック文を使います。

WHILE 文の<論理式> は DO 以下の文を実行する前に毎回テストされます。論理式の値が真の間 DO 以下の文が実行されます。偽になると DO 以下の文は実行されず、WHILE 文は終了します。最初のテストで偽になると DO 以下は一度も実行されずに終わることになります。テストする式は最初に適当な値に初期設定しておかなくてはいけません。

例:

数列の各項の和を求める問題を考えます。項を一つずつ加えて、項の値が 1/1000 より小さくなるとそこで打ち切るものとします。第一項は 1 で、二項目以降の項の値は前項の値の 3 分の 1 と自乗した値の 3 分の 1 の和で与えられるとする。プログラムは、

```
ex:=0; term:=1;
while num(term - 1/1000) >= 0  do
        <<ex := ex+term; term:=(term + term^2)/3>>;
ex;
```

と書けます。ここでは TERM の値が 1/1000 に等しいかより大きい間、その値を EX に加え、次項の値 TERM の値を計算しています。TERM の値が 1/1000 未満になると WHILE のテストは偽になり、TERM の値はもう足されません。

5.6 REPEAT 文

REPEAT 文 は WHILE 文と同じ様に繰り返しを記述するのに使われます。この文の構文は、

```
REPEAT <文> UNTIL <論理式>
```

で、<論理式>の値が偽である間<文>が実行されます。(PASCAL の利用者への注: REPEAT と UNTIL との間には一つの文 (通常グループ文が使われる) しか書けません。)

REPEAT 文は WHILE 文とは次の二つの点で異なっています。

1. テストは<文>を実行した後で行われる。従って、<文>は少なくとも一度は必ず実行されます。

2. テストは継続するかを調べるのではなく、終了するかどうかを判定します。つまり WHILE 文とは判定条件が逆になっています。

例として、WHILE 文の所で使った例題を REPEAT 文で書き直してみると次のようなプログラムになります。

```
ex:=0; term:=1;
repeat <<ex := ex+term; term := (term + term^2)/3>>
   until num(term - 1/1000) < 0;
ex;
```

結果は前の例とまったく同じ結果が得られます。

5.7 複合文

多くの場合、計算は一連の計算ステップを順次行っていくことで行われます。これは、グループ文を使ってできます。しかし、あるステップでの中間結果が最後の結果を求めるときに必要な場合とか、WHILE 文 や REPEAT 文 では記述できないような繰り返し計算を行いたい場合等、グループ文では不十分な場合があります。このような場合に、文を BEGIN と END で囲った ブロック文もしくは複合文を使うことができます。このようなブロック文はグループ文が使われている所でその代わりとして使うこともできます。しかし逆に、ブロック文で記述されたプログラムをグループ文で直接書き直すのは不可能な場合があります。

もし、中間結果をとっておく必要があれば、ローカル変数 (局所変数) を使うことができます。**ローカル**というのは、それらの変数の値がブロック文が終了した時点で消されてしまうことを意味しています。従って、ブロック文の外で同じ名前の変数を使っても、変数の使い方の衝突は起こりません。ローカル変数は BEGIN 直後の SCALAR 宣言で作られます。

```
scalar a,b,c,z;
```

便利のため、SCALAR 宣言を分けて、

```
scalar a,b,c;
scalar z;
```

と書くこともできます。SCALAR の代わりに、同様な宣言、INTEGER か REAL を使うこともできます。現在の REDUCE では、INTEGER として宣言された変数には整数値が入っているものとして扱われ、初期値には 0 が代入されます。REAL 変数は、現在のところ SCALAR 宣言された変数とまったく同じ扱いになっています。

注意: INTEGER, REAL や SCALAR 宣言は BEGIN の直後に書かなくてはなりません。もしこのような宣言をブロックの中で、他の文 (ARRAY 宣言や OPERATOR 宣言を含みます。これらは大域的な宣言です。) の後に書いた場合にはエラーを起こします。SCALAR と宣言された変数は代数モードでは 0、記号モードでは NIL、に初期設定されます。

ブロック文の中で、ローカル変数として宣言されていない変数はすべてこのブロックを含むブロックもしくは、このブロックを呼び出している関数を定義しているブロックの変数が参照されます。特に、このブロックを含むどのブロックでもローカル変数として宣言されていない変数は大域変数として処理され、この変数への代入はブロックの終了したあとも有効になります。

SCALAR 宣言に続いて、任意の数だけ、実行すべき文を順に句切り記号 (; または$) (どちらを使ってもまったく同じことです。) で区切って書きます。プログラムの見やすさの観点からは、セミコロンを使った方がよいでしょう。

ブロック文の一番最後の文、END の直前の文、の最後には終端記号を付ける必要はありません。BEGIN と END は丁度ブロック文を囲む括弧と考えることができます。一番最後の文は RETURN とその後に変数もしくは式を書いたものあっても構いません。この時、この変数または式の値がこのブロック文の値として返されます。RETURN を省略したり、RETURN の後になにも書かなかったときは、値として 0 、記号モードでは NIL、が返されます。END の後には終端記号を付けないといけません。

例:

与えられた N(整数値) に対して、次のブロックを実行すると、変数を X とする N 次の Legendre 多項式が求められます。

```
begin scalar seed,deriv,top,fact;
   seed:=1/(y^2 - 2*x*y +1)^(1/2);
   deriv:=df(seed,y,n);
   top:=sub(y=0,deriv);
   fact:=for i:=1:n product i;
   return top/fact
end;
```

5.7.1 GOTO 文

BEGIN ...END ブロック中ではもっと複雑な文を書くこともできます。ブロック文を構成する各文は代入文だけではなく、ほとんど全ての種類の文やコマンドを書くことができます。例えば、条件文、WHILE 文、REPEAT 文等を使うことで、もっと込み入ったブロックを書くことができます。

もし、これらの構造だけでは不十分ならば、ラベル と GO TO を複合文で使うことができます。また、RETURN を END の直前ではなく、ブロックの途中に入れることもできます。続く節でこれらの詳細に付いて説明します。

多くの読者には次の例—N に設定された値に対して N!(N の階乗) を求めるプログラム—で十分理解できるでしょう。

例:

```
begin scalar m;
    m:=1;
 l: if n=0 then return m;
    m:=m*n;
```

```
        n:=n-1;
        go to l
    end;
```

5.7.2 ラベル

BEGIN ...END 複合文中では、文にラベルを付けることができます。そして、GO TO 文によりこれらのラベルを付けた文へジャンプすることができます。複合文中で最上位の文にのみラベルを付けることができます。グループ文 (<< ...>>)、IF 文、WHILE 文等の中で使われている文に付けることは出来ません。

ラベルと GO TO 文は次の構文を取ります。

```
<go to 文> ::= GO TO <ラベル> | GOTO <ラベル>
<ラベル> ::= <ラベル>
<ラベル付きの文> ::= <ラベル>:<文>
```

文の名前等はラベル名に使うことはできません。

GO TO は無条件にジャンプを行いますが、条件文 (IF 文) 中で

```
if x>5 then go to abcd;
```

の様に使うことで条件付きのジャンプが行えます。

GO TO によるジャンプは、GO TO が使われた同じブロック文中の文にのみ行えます。つまり、ブロック文をその中に含んでいるブロック文で、内側のブロック文から GO TO で外側のブロック文中の文にジャンプすることはできません。しかし、グループ文 (あるブロック文にの中にある) の中から外へ GO TO でジャンプすることはできます。

5.7.3 RETURN 文

複合文 BEGIN ...END の値は普通 0、記号モードでは NIL、になっています。複合文中で RETURN 文を実行することで 0 以外の RETURN 文で指定した値を返すことができます。RETURN を実行すると、複合文中の残りの文は実行されません。

例:

```
return x+y;
return m;
return;
```

RETURN 文中の式は括弧で囲う必要はありません。この最後の例は、return 0 または return nil (返された値が使われる時には 0、使われないときには NIL を指定したのと) と同じことを意味します。

RETURN はブロックの階層 を一レベルだけ抜け出します。この点に関しては、一般のユーザは理解しがたいと思われるので、注意すべき点を列記しておきます。

1. RETURN は複合文の中の最上位レベルでのみ使うことができます。つまり、BEGIN ...END で囲われた文の一つとして使うことができます。
2. RETURN は複合文の中に含まれた、グループ文 (<< ...>>) の一番上位のレベルの文に使うことができる。この場合、RETURN はグループ文から抜け出すと同時に、複合文 (ブロック文) からも抜け出します。
3. RETURN は複合文の中にある IF 文の実行文の中に現れてもよい。

注意: 今の所、FOR , WHILE もしくは REPEAT 文を途中で中断するような手段は用意されていません。特に、RETURN をこのような文中で使うとエラーを起こします。例えば、

```
begin scalar y;
   y := for i:=0:99 do if a(i)=x then return b(i);
   ...
```

は、エラーを起こします。

第6章　コマンドと宣言

コマンド はシステムへ何らかの操作を行わせる命令です。あるコマンドは命令を実行した結果が目に見える形で (入力や出力を呼び出す場合などのように) 出力されます。それ以外のコマンドは宣言 (declaration) と呼ばれるもので、オプションを設定したり、変数の型等を定義したり、また関数 (プロシジャ) を定義します。コマンドは文に終端記号を付けたものとして定義されています。

```
<コマンド> ::= <文> <終端記号>
<終端記号> ::= ;|$
```

REDUCE のコマンドと宣言の一部は以下の節で説明します。

6.1　配列宣言

REDUCE の配列 (array) 宣言は FORTRAN の dimension 文と似ています。例えば、

```
array a(10),b(2,3,4);
```

配列の添字は 0 から宣言文で宣言した値まで取ります。配列の要素は FORTRAN のように、例えば `A(2)` で参照します。

配列の範囲を指定するときに式を使うこともできます。このとき、式の値は正の整数でなければいけません。例えば、`X` が 10 で `Y` の値が 7 のとき、`array c(5*x+y)` は `array c(57)` と同じ結果になります。

もし配列の添字が範囲外を指しているときにはエラーを起こします。一次元の配列を宣言する場合、範囲を整数もしくは変数で指定するときには、括弧を省略することができます。

```
array a 10, c 57;
```

オペレータ `LENGTH` の引数として配列名を与えると、その配列のサイズをリスト形式で返します。

配列の要素は配列の宣言時に 0 に初期設定されます。言い替えれば、配列の要素は**即時評価**の性質を持っており、それ自身を表すことはありません。もしこのようなことが行いたければ、オペレータを代わりに使います。

配列宣言はプログラムのどこに書いても構いません。一度配列と宣言された名前は、`CLEAR` 文で定義を取り消さない限り、変数名、オペレータ名や関数名として使うことはできません。しかし、配列として再定義することは可能で、そのときに配列のサイズを変更することもできます。配列名と同じ名前を、関数の仮引数名や、ローカル変数名として使うことはできます。しかし、このような使い方はユーザが変数の混同を起こし易くなるので、あまりしない方がよいでしょう。

一度宣言された配列は大域変数として扱われ、プログラムのどこからでも参照することができます。つまり、多くの言語で使われる配列とは異なって、ブロック文中で宣言された配列はブロック文の範囲をこえて参照でき、ブロック文の実行が終了しても残っています。配列の定義を削除するには CLEAR を使います。

6.2 モード宣言

ON 宣言と OFF 宣言はシステムで用意されているオプションを操作するのに使います。それらのオプションについては**スイッチ** *(switch)* 名で表されます。ON と OFF はスイッチ名のリストを引数として取り、それらのスイッチをそれぞれオンやオフにします。

```
on time;
```

は各コマンドを実行したときに、そのコマンドの実行に要した CPU 時間を表示します。(表示されるのは前に CPU 時間を表示してから、またはこのセッションを開始してから、現在のコマンドを実行し終わるまでに掛かった CPU 時間です。) その他、対話的に実行しているときに有用なスイッチに DEMO スイッチ があります。このスイッチがオンになっていると、ファイルから入力しているときそれぞれのコマンド (ただしコメント文は除く) を実行した後で停止して、ユーザが端末から 改行 キーを入力するまで次のコマンドの実行を休止します。これを使えば、デモ用のプログラムを作成するときに、各コマンドを一つずつ順に実行するように設定できます。

ほとんどの宣言文がそうであるように、ON や OFF コマンドの引数は、コンマで区切ることによって任意の数の引数を指定できます。例えば、

```
on time,demo;
```

は、CPU 時間を表示させると同時にデモ用の表示をオンにします。

多くのスイッチに対しては、ON や OFF コマンドは直ちに実行が完了します。しかし、スイッチによっては必要なパッケージをロードするというような動作を伴うものがあり、完了までに時間がかかることもあります。

ON や OFF でスイッチとして定義されていない引数を与えると診断メッセージが出力されます。例えば、DEMO を入力するつもりで間違って、

```
on demq;
```

と入力すると、

```
***** DEMQ not defined as switch.
(***** DEMQ はスイッチとして定義されていません。)
```

というメッセージが出力されます。

6.3 END

識別子 END には二つの用途があります。

1) ブロック文 BEGIN ...END で使われる。これについてはすでに複合文の所で説明しました。

2) IN コマンド でファイルからコマンドを入力するとき、ファイルの最後に END; を入れておく必要があります。これについては IN コマンドの説明のところを参照してください。このように END を使うときには、前の END(関数定義の終わりの END など) の直後にはおけません。;END; のように使うのがよいでしょう。

6.4 BYE コマンド

コマンド BYE;(または QUIT;) は、現在開いているファイルをすべて閉じて、REDUCE の実行を終了し、REDUCE を呼び出したプログラム (普通はオペレーティングシステム) に戻ります。このとき、今までの計算結果は通常壊されてしまいます。

6.5 SHOWTIME コマンド

SHOWTIME コマンドは以前にこのコマンドを使ったときから、または一番最初に使ったときにはセッションの開始から、使用した CPU 時間を表示します。時間は通常ミリ秒で表されます。表示される時間に入出力に要した時間が含まれているかどうかはシステムに依存します。

6.6 DEFINE コマンド

DEFINE コマンドは、任意の識別子もしくはよく使われるような式に別名を付けるため使用します。DEFINE の引数は、

```
<識別子> = <数値>|<識別子>|<オペレータ>|
           <予約語>|<式>
```

をリストとして並べたものです。

例:

```
define be==,x=y+z;
```

とすればこれ以後、 BE は等号記号として解釈され、X は式 y+z として解釈されます。この変換は入力時に行われます。従って、同じ名前に対して定義されている LET 文等の置き換え規則に先だって実行されることになります。この変換は REDUCE のセッションが終わるまで有効です。

識別子 ALGEBRAIC と SYMBOLIC は特別な性質を持っており、そのため DEFINE で別名を付けることはできません。例えば、もし ALG を ALGEBRAIC の別名として使いたいときには、

```
put('alg,'newnam,'algebraic);
```

のように入力しなければいけません。

第7章　組み込みの前置演算子

この章ではREDUCEに組み込まれている前置演算子の中で、他の章では説明していないものについて説明しています。これらの内のいくつかはシステムでプロシジャとして完全に定義されていますが、そのほかの演算子は抽象的なオペレータとしてその演算の性質の一部のみが定義されています。

以下の節での説明は、多くの場合オペレータ名とその引数を示しています。引数は、その名前と引数として許される型とを示し、それぞれ引数の名前は大文字で、型のクラス名は小文字で表しています。引数の内で、通常指定しなくて省略可能な引数は { ... } のように中括弧で囲って表しています。任意の数の引数をとる演算子については、引数を大括弧 ([ ]) で囲って表しています。この引数は複数個並べることができます。また鍵括弧 (< >) で囲まれた引数はオプションです。

7.1　数値演算

REDUCEは多くの数値計算用の言語と同じように、数値計算に使われる関数を持っています。数値の引数に対しては、このような関数は期待される値を返します。しかしながら、数値以外の式を引数として与えることも可能です。この場合、可能な限り簡単な形に変形した結果を返します。この時には、返される式中にはもとのオペレータが含まれていることがあります。これらのオペレータには以下のようなものがあります。

7.1.1　ABS

ABS は引数が数値であればその引数の絶対値を返し、引数が数値でなければ、数値係数の絶対値をABS演算子の外に出した式を返します。例えば、

```
abs(-3/4)      ->  3/4
abs(2a)        ->  2*ABS(A)
abs(i)         ->  1
abs(-x)        ->  ABS(X)
```

7.1.2　CEILING

この演算子は数値の引数が与えられたときには、その数よりも大きな最小の整数を返します。数値でない場合には、入力した式がそのまま返されます。例えば、

```
ceiling(-5/4) ->  -1
```

```
ceiling(-a)   ->  CEILING(-A)
```

7.1.3 CONJ

これは、数値の引数に対しては複素共役数を返します。数値でない式に対しては、REPART と IMPART 演算子を使った形で表した複素共役式を返します。 例えば、

```
conj(1+i)     -> 1-I
conj(a+i*b)   -> REPART(A) - REPART(B)*I
                 - IMPART(A)*I - IMPART(B)
```

7.1.4 FACTORIAL

FACTORIAL は引数が負でない整数であればその階乗を求めます。それ以外の引数を与えたときは、入力した式がそのまま返されます。例えば、

```
factorial(5)  ->  120
factorial(a)  ->  FACTORIAL(A)
```

7.1.5 FIX

この演算子は引数が数値であれば、その整数部分を返します。数値以外の引数に対しては、入力したそのままの式が返されます。例えば、

```
fix(-5/4)   ->  -1
fix(a)      ->  FIX(A)
```

7.1.6 FLOOR

この演算子は、数値に対しては、与えられた数値よりも小さい最大の整数を求めます。数値でない場合には、そのままの形が返されます。例えば、

```
floor(-5/4)   ->  -2
floor(a)      ->  FLOOR(A)
```

7.1.7 IMPART

これは与えられた式の虚数部分を求めます。数値でない式に対しては REPART 演算子と IMPART 演算子を使って表した結果が返されます。 例えば、

```
impart(1+i)   -> 1
impart(a+i*b) -> REPART(B) + IMPART(A)
```

7.1.8 MAX/MIN

MAX と MIN は任意の数の引数を取り、それらの引数の最大値または最小値を計算します。もし引数の中に数値でないものがあれば、簡約された結果が返されます。例えば、

```
max(2,-3,4,5) ->  5
min(2,-2)     ->  -2
max(a,2,3)    ->  MAX(A,3)
min(x)        ->  X
```

MAX または MIN で引数として空リスト (0 個の引数) を与えた場合、0 が返されます。

7.1.9 NEXTPRIME

NEXTPRIME は与えられた整数よりも大きな最初の素数を確率論的な算法によって求めます。もし引数が整数でなければエラーになります。例えば、

```
nextprime(5)       ->  7
nextprime(-2)      ->  2
nextprime(-7)      -> -5
nextprime 1000000 -> 1000003
```

となりますが、nextprime(a) と入力すると型が間違っているというエラーを起こします。

7.1.10 RANDOM

random(n) は $0 \leq r < n$ の範囲の乱数 r を返します。代数モードの場合、引数 n が正の整数でなければエラーとなります。また記号モードの場合には正の数値でなければ、エラーとなります。例えば、

```
random(5)          ->    3
random(1000)       ->  191
```

しかし、random(a) はエラーとなります。

7.1.11 RANDOM_NEW_SEED

random_new_seed(n) は関数 RANDOM が生成する乱数を、整数 n で決まる乱数列に再設定します。これは、それ以前の RANDOM の呼び出しに依存しないある決まった疑似乱数列が必要な場合

や、例えば現在の時刻の様に変化する数値を与えることによって毎回異なった乱数列を生成したいという目的に使用することができます。REDUCE が起動したときには、random_new_seed(1) が実行された状態になっています。

もし、引数が正の整数でなければエラーを起こします。

7.1.12 REPART

これは式の実数部分を計算します。数値以外の式が与えられたときには、REPART 演算子と IMPART 演算子を使った形で表されます。例えば、

```
repart(1+i)   -> 1
repart(a+i*b) -> REPART(A) - IMPART(B)
```

7.1.13 ROUND

これは与えられた数値の小数以下を四捨五入した結果を返します。もし数値以外の式を与えたときは、入力した式そのままの形を返します。例えば、

```
round(-5/4)   ->  -1
round(a)      ->  ROUND(A)
```

7.1.14 SIGN

SIGN は引数の符号を求めます。もしこれが可能で有れば 1,0,-1 のいずれかを返します。符号が不明な場合には、簡約した結果を返します。例えば、

```
sign(-5)      ->  -1
sign(-a^2*b)  ->  -SIGN(B)
```

COMPLEX スイッチがオフの場合には、偶数べき乗の式は正であるとして処理されます[1] 。

7.2 数学関数

REDUCE には次のような数学関数が定義されています。

```
ACOS ACOSH ACOT ACOTH ACSC ACSCH ASEC ASECH ASIN ASINH
ATAN ATANH ATAN2 COS COSH COT COTH CSC CSCH DILOG EI EXP
HYPOT LN LOG LOGB LOG10 SEC SECH SIN SINH SQRT TAN TANH
```

[1] sign(a**2) ==> 1

LOG は自然対数 (LN と同じ関数) で、LOGB は二つの引数をとり第二引数で指定した底に対する対数を計算します。

これらの関数については、微分規則も知っています。

REDUCE は、これらの関数に関する様々な基本関係式や性質を知っています。例えば、次のようなものです。

```
cos(-x) = cos(x)              sin(-x) = - sin (x)
cos(n*pi) = (-1)^n            sin(n*pi) = 0
log(e)  = 1                   e^(i*pi/2) = i
log(1)  = 0                   e^(i*pi) = -1
log(e^x) = x                  e^(3*i*pi/2) = -i
```

これらの恒等式は別にして、REDUCE ではルール規則の形式で初等関数に関する多くの簡約規則が定義されています。これらの規則の中身を知りたい場合には SHOWRULES 演算子を使います。例えば、

```
SHOWRULES tan;
```

```
{tan(~n*arbint(~i)*pi + ~(~ x)) => tan(x) when fixp(n),

 tan(~x)

  => trigquot(sin(x),cos(x)) when knowledge_about(sin,x,tan),


     ~x + ~(~ k)*pi              x                              k    1
 tan(----------------) =>  - cot(---) when x freeof pi and abs(---)=---,
           ~d                    d                              d    2

     ~(~ w) + ~(~ k)*pi           w + remainder(k,d)*pi
 tan(--------------------) => tan(----------------------)
           ~(~ d)                           d

                                k                        k
  when w freeof pi and ratnump(---) and fixp(k) and abs(---)>=1,
                                d                        d


 tan(atan(~x)) => x,

                              2
 df(tan(~x),~x) => 1 + tan(x) }
```

三角関数を含んだ式のより進んだ簡単化については、TRIGSIMP パッケージのマニュアルを参照して下さい。上のリストで示した以外の特殊関数については SPECFN パッケージで定義されています。

利用者はこれらの関数を含む式に対する簡約規則を LET コマンドを使って定義することができます。

積の形の引数に対する対数関数をそれぞれの因子の対数の和に書き換えること、またはその逆に対数の和を一つの対数に書き換えることが必要な場合があります。両方の変換を同時に行った場合、REDUCE の元来の考えである、評価の同一性が満たされなくなります。従って、REDUCE では二つのスイッチ EXPANDLOGS と COMBINELOGS を使ってこれらの変換のどちらを行うかを制御するようにしています。このスイッチは通常両方ともオフです。例えば、LOG(X*Y) を展開して log の和で表したいときには次のようにすれば良い。

```
ON EXPANDLOGS; LOG(X*Y);
```

また逆にこの和を一つの log にしたい時には次のようにします。

```
ON COMBINELOGS; LOG(X) + LOG(Y);
```

現在の所、両方のスイッチを同時にオンにすることもできます。これは無限ループを引き起こす様に思われますが、実際に起こることは (現在のインプリメントでは) 式は単に一方の形式から他方の形式に変換されるだけです。しかし次のリリースでは変更されるかも知れないので、ユーザはこのような動作を期待したプログラムを作らないようにしてください。

現在の REDUCE は無理式の簡約に対しては貧弱な機能しか持ちません。特に、整数以外の数もしくは数式をべきに持つ式に対しては、出力時には簡約された形で表されていても内部では簡約されていません。例えば、次の式は

```
x^(1/3)*x^(1/6);
```

次のように出力されます。

```
SQRT(X)
```

しかし内部形式としては二つのべき乗の積として表されています。

この式から sqrt(x) を引いても零になりません。この場合次のような奇妙な結果が得られます。

```
SQRT(X) - SQRT(X)
```

このようなべき乗の項を簡約することは COMBINEEXPT スイッチをオンにしないと行われません。

平方根は SQRT で入力することも出来るし、また分数べきの形で^(1/2) を使って入力することもできます。出力時には簡約できない平方根は通常分数べきの形ではなく、SQRT 演算子の形表示されます。通常の設定では平方根の引数は最初に簡約され、平方因子が平方根の外にくくり出されます。それ以外の式は平方根の形で残されます。

従って、式

```
sqrt(-8a^2*b)
```

は次のようになります。

```
2*a*sqrt(-2*b).
```

注意して置くことは、このような簡約は、もし A が負の値を持つ場合には問題を起こします。もし式の平方および偶数べき乗が常に正であることが重要な場合には、PRECISE スイッチをオンにしておきます。(通常オンになっています。) このスイッチは、無理式から取り出した因子を絶対値で表現するように設定します。PRECISE をオンすると、上の例の結果は次のようになります。

```
2*abs(a)*sqrt(-2*b).
```

関数

```
ACOS ACOSH ACOT ACOTH ACSC ACSCH ASEC ASECH ASIN ASINH
ATAN ATANH ATAN2 COS COSH COT COTH CSC CSCH EXP HYPOT
LN LOG LOGB LOG10 SEC SECH SIN SINH SQRT TAN TANH
```

に対する近似値をユーザが指定した精度で計算することができます。また、数値に対するべき乗 (実数べき) も計算することができます。

もし、 ROUNDED スイッチとともに COMPLEX スイッチをオンにするとこれらの関数の複素数に対する値を計算することができます。例えば、on rounded,complex; と入力した後では、

```
2.3^(5.6i)    ->   -0.0480793490914 - 0.998843519372*I
cos(2+3i)     ->   -4.18962569097 - 9.10922789376*I
```

となります。

7.3 微分演算

微分計算は DF 演算子を使って計算できます。DF は式を指定された変数に関して偏微分を行った結果を返します。この演算子の構文は、

```
DF(EXPRN:代数式 [,VAR:カーネル<,NUM:整数>]):代数式.
```

第一引数は微分しようとする式で、第二引数以後は微分を行う変数およびその変数について何回微分をとるかを整数で指定します。

整数<NUM>はもし 1 であれば省略することができます。例えば、

$$\begin{aligned}
&\texttt{df(y,x)} && = \partial y/\partial x \\
&\texttt{df(y,x,2)} && = \partial^2 y/\partial x^2 \\
&\texttt{df(y,x1,2,x2,x3,2)} && = \partial^5 y/\partial x_1^2 \partial x_2 \partial x_3^2.
\end{aligned}$$

df(y,x) の計算は次のように行われます。まず、Y と X が評価されます。X には値が代入されておらず、従って、その値は X 自身になっているとします。Y を評価した値のうち、X を含んでいる項についてそれぞれ通常の微分規則に従って計算されます。Y が変数 X とは異なる変数 Z を含んでいる場合、Z を X で微分した結果は 0 となります。ただし、Z があらかじめ DEPEND 宣言によって変数 X に依存していると宣言されているときには、Z を X で微分した結果は df(z,x) と表されます。

7.3.1 微分規則の追加

LET 文を使ってユーザが微分規則を定義することができます。一般的な形として、

```
FOR ALL <変数 1>,...,<変数 n>
     LET DF(<オペレータ><変数リスト>,<変数 i>)=<式>
```

の形を取ります。ここで、<変数リスト> ::= (<変数 1>,...,<変数 n>) で、<変数 1>,...,<変数 n> は <オペレータ>に対する仮の変数引数の名前です。

内挿演算子に対しても同様な形式でその微分を定義することができます。

例:

```
for all x let df(tan x,x)= 1 + tan(x)^2;
```

(これは、REDUCE がそのソースの中でどのように関数 tan の微分を定義しているかを示しています。)

```
for all x,y let df(f(x,y),x)=2*f(x,y),
                df(f(x,y),y)=x*f(x,y);
```

微分規則がどんな引数に対しても適用されるように、考えているオペレータの全ての仮引数を FOR ALL コマンドで任意の式とマッチするように宣言されていることに注意してください。もし、与えられた引数に対する微分規則が定義されていなければ、微分した結果の式は、DF 演算子で表されます。例えば、F の第二引数に関する微分規則が定義されていないとき、df(f(x,z),z) を計算した結果は入力した式とまったく同じ式が返されます。(ここで明らかに f(x,z) は Z の関数になっているので、DEPEND 宣言は行う必要はありません。)

一度オペレータに対して微分規則を定義すると、これ以後同じオペレータに微分規則を定義するときには引数は同じ長さでなければいけません。さもないと次のようなエラーメッセージが出力されます。

```
Incompatible DF rule argument length for <オペレータ>
```
(<オペレータ>に対する DF 規則の引数の長さが一致していません。)

7.4 積分計算

INT オペレータは REDUCE で不定積分を計算するのに使います。このオペレータは Risch-Norman の算法とパターンマッチを使って積分を求めます。このオペレータの構文は、次の通り

です。

INT(EXPRN:代数式,VAR:カーネル):代数式.

このオペレータは多項式、log 関数、指数関数や tan,atan を含む式の不定積分を計算します。積分定数はつけられません。もし、不定積分が求められなかった場合には次のいずれかの形で結果が返されます。

1. 入力された式 INT(...,...) をそのまま返す。
2. 入力した式とは違う、INT を含んだ式を返す。(残念にも、入力された式よりも複雑な結果になってしまうことがしばしばあります。)

有理式は分母が因数分解できる場合には積分できます。それ以外の一般の式に対しては、不定積分が誤差関数、dilog 関数や三角関数等で表されるかどうかを調べます。この場合、不定積分が存在している場合でも、解が求められないことがあります。

例:

```
int(log(x),x) ->  X*(LOG(X) - 1),
int(e^x,x)    ->  E**X.
```

プログラムは第二引数が変数であるかどうかを調べています。もし変数でなければエラーを起こします。

注意: もし、int 演算子に対して 4 つの引数を与えると、REDUCE は定積分パッケージ (DEFINT) を呼び出します。このパッケージでは、三番めと四番目の引数は積分の下限と上限をそれぞれ表します。詳しいことは DEFINT パッケージのマニュアルを参照して下さい。

7.4.1 オプション

TRINT スイッチ をオンにすると計算のアルゴリズムをトレースします。これは、大量の出力を出します。また出力の形式が通常の数式の形式ではないので、一般のユーザには分かりにくいものになっています。通常はオフになっています。

もし、 FAILHARD スイッチがオンになっていると不定積分が求められないときにはエラーとして処理します。FAILHARD は通常オフになっています。

NOLNR スイッチ は積分が求められない場合、積分演算が線形作用素であることを使って展開した形で表します。これは通常オフになっています。

7.4.2 より進んだ使用法

もし、被積分関数が EXP, ERF, TAN, ATAN, LOG, DILOG 以外の関数を含んでいる場合、その被積分項を微分して既知の関数で表されるかどうかを調べ、これから積分を求められるかどうか調べます。しかし、この場合には正しい結果が得られるとは保証できません。もし、被積分関数がこれらの関数とは代数的に独立であることが分かっている場合には、例えば、

```
flag('(trilog),'transcendental);
```

のように、超越関数 (transendental function) として宣言して置きます。このようにすれば、この関数は積分を決定するときに、これらの関数を含んだ項は代数的に独立 (これらの関数を含んだ式が零に等しくなるのは、関数の係数が零に等しい場合に限る) として扱われます。このときには、この代数的独立性が正しいことを、ユーザが確認して置かなければなりません。さもなければ、正しい結果が得られません。

標準の積分パッケージでは代数拡大を扱うことはできません。従って、平方根等を含む式の積分は計算できません。ただし、ユーザによって作成されたライブラリには、平方根を含んだ式の積分が扱えるパッケージがあります (ALGINT)。また、定積分を計算するパッケージ DEFINT. も含まれています。

7.4.3 参考文献

A. C. Norman & P. M. A. Moore, "Implementing the New Risch Algorithm", Proc. 4th International Symposium on Advanced Comp. Methods in Theor. Phys., CNRS, Marseilles, 1977.

S. J. Harrington, "A New Symbolic Integration System in Reduce", Comp. Journ. 22 (1979) 2.

A. C. Norman & J. H. Davenport, "Symbolic Integration — The Dust Settles?", Proc. EUROSAM 79, Lecture Notes in Computer Science 72, Springer-Verlag, Berlin Heidelberg New York (1979) 398-407.

7.5 LENGTH 演算子

LENGTH は数々のオブジェクトに対する長さを計算します。長さの意味は対象によって変わってきます。代数式の場合、"長さ" は展開された形で表したときの項の数となります。

例:

```
length(a+b)     ->  2
length(2)       ->  1.
```

その他、LENGTH 演算子で計算できるものとして配列や行列の長さ (次元) があります。これらの正確な意味についてはそれぞれの項を見てください。

7.6 MAP 演算子

MAP 演算子は行列やリスト、演算子の引数のような複合構造体の全ての要素に対してある演算を行ないます。演算として行なえるものは、単項演算や演算子、一つの自由変数を持つ代数式です。

この演算子は次のように使われます。

MAP(U:関数,V:object)

ここで、object はリストや行列、演算子式等です。関数 は次にあげるいずれかです。

1. 一つの引数を取る関数。object の各要素に対して関数が評価されます。
2. 一つの自由変数（変数名の前にティルダを付けたもの）を持つ代数式。⟨*object*⟩の各要素に対して、それを自由変数に代入した式が評価されます。
3. ⟨*var* => *rep*⟩の形の置き換え規則、ここで var は変数（添字式を持たないカーネル）で、rep は var を含んだ式。式 rep 中の変数 var に object の各要素を代入した値が評価されます。変数 var にはディルダを付けても構いません。

一つ以上の自由変数を含む場合には、関数としては置き換え規則の形式のもののみが指定可能です。

例:

```
map(abs,{1,-2,a,-a})  ->  {1,2,ABS(A),ABS(A)}
map(int(~w,x), mat((x^2,x^5),(x^4,x^5))) ->

      [  3     6 ]
      [ x     x  ]
      [----  ----]
      [ 3     6  ]
      [          ]
      [  5     6 ]
      [ x     x  ]
      [----  ----]
      [ 5     6  ]

map(~w*6, x^2/3 = y^3/2 -1) -> 2*X^2=3*(Y^3-2)
```

MAP を入れ子になった式に対しても行なえます。しかしながら、MAP 演算子は識別子や数値の様な構造体でないものに対しては演算できません。

7.7 MKID 演算子

多くの応用で、識別子を作り出すことができれば便利なことがあります。たいていの場合、二つの要素から名前を作れば十分です。MKID 演算子はこのような目的で作られました。構文は、次の通りです。

MKID(U:識別子,V:識別子|非負の整数):識別子

例えば、

```
mkid(a,3)      -> A3
mkid(apple,s)  -> APPLES
```

しかし、mkid(a+b,2) のような使い方はエラーになります。

SET 演算子を使うことで、MKID で作り出した識別子に値を代入することができます。例えば、

```
set(mkid(a,3),3);
```

では変数 A3 に値 2 を代入します。

7.8 部分分数

PF(<exp>,<var>) は式<exp>を主変数<var>に関する部分分数に分解し、それをリストとして返します。PF は完全な部分分数分解を計算しますが、算法はあまりきれいではありませんし、複雑な式に対してはかなり計算に時間が掛かります。因数分解と拡張されたユークリッドの算法を使っています。

例: 2/((x+1)^2* (x+2)) がワーク領域に入っているものとして、pf(ws,x); と入力すると次のような結果が得られます。

```
     2      - 2         2
{-------,-------,--------------} .
  X + 2   X + 1    2
                  X  + 2*X + 1
```

もし、因数分解された形で表示したければoff exp; と入力してください。つまり、2/((x+1)^2* (x+2)) がワーク領域に入っているものとして、off exp; pf(ws,x); と入力すると、次の結果が得られます。

```
     2      - 2       2
{-------,-------,----------} .
  X + 2   X + 1          2
                  (X + 1)
```

これらの項を一緒にするには、FOR EACH ...SUM 形式を使えばできます。従って、上の例題のすぐ後で、for each j in ws sum j; と入力すると次のような結果が得られます。

```
        2
------------------
                2
 (X + 2)*(X + 1)
```

また、リスト演算子を使えば必要な項を取り出すことができます。

7.9 SELECT 演算子

SELECT 演算子はリストまたは n-引数の演算子から条件に一致する要素を選び出します。使用法は次のとおりです。

SELECT(U:関数,V:list)

関数 は次にあげるいずれかです。

1. 一つの引数を取る関数。object の各要素に対して関数が評価されます。
2. 一つの自由変数（変数名の前にティルダを付けたもの）を持つ代数式。$\langle object \rangle$の各要素に対して、それを自由変数に代入した式が評価されます。
3. $\langle var => rep \rangle$の形の置き換え規則、ここで var は変数（添字式を持たないカーネル）で、rep は var を含んだ式。式 rep 中の変数 var に object の各要素を代入した値が評価されます。変数 var にはディルダを付けても構いません。

一つ以上の自由変数を含む場合には、関数としては置き換え規則の形式のもののみが指定可能です。

関数 を評価した結果は、REDUCE の通常の方法に従って論理値として解釈されます。結果が真であるような要素のみを集め、リストもしくはもともとの演算子の形式で返します。

例:

```
select( ~w>0 , {1,-1,2,-3,3}) -> {1,2,3}
select(evenp deg(~w,y),part((x+y)^5,0):=list)
       -> {X^5 ,10*X^3*Y^2 ,5*X*Y^4}
select(evenp deg(~w,x),2x^2+3x^3+4x^4) -> 4X^4 + 2X^2
```

7.10 SOLVE 演算子

SOLVE は一つまたは連立の代数方程式の解をもとめます。構文は

SOLVE(EXPRN:代数式 [,VAR:カーネル|,VARLIST:カーネルのリスト]):リスト。

EXPRN は<式>または { <式 1>,<式 2>, ... } の形をとります。それぞれの式は代数方程式もしくは式 (等式の両辺の差式) です。第二引数は、カーネルもしくはカーネルのリストで、方程式の未知変数を表します。この引数は、もし式に現れる異なる定数でないカーネルの数が未知変数の数と等しいときには省略しても構いません。このときには、式に現れるカーネルは未知変数と仮定されます。

一つの方程式に対して、SOLVE は与えられた方程式に因数分解、DECOMPOSITION を取る操作や、LOG, SIN, COS, ^, ACOS, ASIN の逆関数を取ったり線形、二次、三次、四次方程式、二項因子等を繰り返し使って解を求めます。解に指数関数や対数関数が含まれている場合には、しばしば、結果が Lambert の W 関数を使って表されます。この関数は、特殊関数パッケージに（部分的に）インプリメントされています。

連立一次方程式については、Bareiss の方法を使った多段の消去法を使って解を求めます。もし CRAMER スイッチがオンの場合には、Cramer の方法によって解が求められます。方程式が大きくまた密な場合以外には、Bareiss の方法の方が通常はより効率的です。

非線形方程式は Gröbner 基底パッケージを使って解くことができます。しかし、このような計算には時間がかかることを注意しておきます。

例:

```
solve(log(sin(x+3))^5 = 8,x);
solve(a*log(sin(x+3))^5 - b, sin(x+3));
solve({a*x+y=3,y=-2},{x,y});
```

SOLVE は求めた解のリストを返します。

もし、未知変数の数が一つの場合、それぞれの解は未知変数に関する式になっています。もし完全な解が求められた場合には、"未知変数=解" のリストの形で結果が得られます。また solve パッケージが解を求めることができなかったときには、未知変数に関する方程式を返します。もし、解が複数個存在した場合には、それぞれの解をリストにして返します。例えば、

```
solve(x^2=1,x);         -> {X=-1,X=1}

solve(x^7-x^6+x^2=1,x)
                           6
          -> {X=ROOT_OF(X_  + X_ + 1,X_,TAG_1),X=1}


solve({x+3y=7,y-x=1},{x,y}) -> {{X=1,Y=2}}.
```

TAG 引数が個々の解を区別する為に用いられます。解の重複度は大域変数 ROOT_MULTIPLICITIES に保存されています。 この変数の値は、SOLVE が返す解のそれぞれについて重複度がリストの形で代入されています。 例えば、

```
solve(x^2=2x-1,x); root_multiplicities;
```

は、結果として、

```
{X=1}

{2}
```

を出力します。

もし、重複度を明示的に出力したければ、MULTIPLICITIES スイッチをオンにすればよい。例えば、

```
on multiplicities; solve(x^2=2x-1,x);
```

と入力すれば、

```
{X=1,X=1}
```

という結果が得られます。

7.10.1 不可解の場合の扱い

SOLVE パッケージはもし解が求められなかった場合、演算子 ROOT_OF を使って未知数に対する方程式の形で解を表します。例えば、次の式は

```
solve(cos(x) + log(x),x);
```

は次のような結果を返します。

```
{X=ROOT_OF(COS(X_) + LOG(X_),X_,TAG_1)}.
```

ROOT_OF 演算子で表される式は要素数が未知のリストになっています。(なぜならば方程式の解の個数は一般には不明だからです。) もしこのような式に含まれるパラメータに対する置き換えが行われた結果、閉じた形の解が得られることが起こり得ます。このようなことが起こると、ROOT_OF 演算子は ONE_OF 演算子に書き直されます。 このとき、このような解を通常 SOLVE が返す解の形に変換した方が望ましい場合があります。これを行うには EXPAND_CASES 演算子を使います。

次の例はこれらの機能の使い方を示しています。

```
solve(-a*x^3+a*x^2+x^4-x^3-4*x^2+4,x);
                2     3
{X=ROOT_OF(A*X_  - X_  + 4*X_ + 4,X_,TAG_2),X=1}

sub(a=-1,ws);

{X=ONE_OF({2,-1,-2},TAG_2),X=1}

expand_cases ws;

{X=2,X=-1,X=-2,X=1}
```

7.10.2 三次及び四次方程式の解

三次や四次方程式の解はしばしば非常に複雑な形になることがあります。このため、FULLROOTS スイッチを用意してあります。 このスイッチがオフの場合には閉じた形の解ではなく ROOT_OF 演算子を使った形で解を表します。FULLROOTS スイッチは通常オフになっています。

最後に、三次や四次方程式の解は三角関数を使っても表されるように変更しました。これは TRIGFORM スイッチで制御できます。このスイッチは通常オンです。

次の例はこれらの機能を表しています。

```
let xx = solve(x^3+x+1,x);

xx;
             3
{X=ROOT_OF(X_  + X_ + 1,X_,TAG_3)}

on fullroots;

xx;

                              - SQRT(31)*I
                     ATAN(----------------)
                              3*SQRT(3)
{X=(I*(SQRT(3)*SIN(-----------------------)
                                3

                       - SQRT(31)*I
                ATAN(---------------)
                        3*SQRT(3)
        - COS(-----------------------)))/SQRT(3),
                          3

                                - SQRT(31)*I
                         ATAN(---------------)
                                 3*SQRT(3)
 X=( - I*(SQRT(3)*SIN(-----------------------)
                                   3

                           - SQRT(31)*I
                    ATAN(---------------)
                            3*SQRT(3)
           + COS(-----------------------)))/SQRT(
                              3


      3),

                   - SQRT(31)*I
            ATAN(---------------)
                    3*SQRT(3)
    2*COS(-----------------------)*I
                      3
 X=-----------------------------------}
                SQRT(3)
```

```
off trigform;

xx;
                              2/3
{X=( - (SQRT(31) - 3*SQRT(3))   *SQRT(3)*I

                              2/3    2/3
     - (SQRT(31) - 3*SQRT(3))    - 2   *SQRT(3)*I

        2/3                             1/3  1/3
     + 2   )/(2*(SQRT(31) - 3*SQRT(3))   *6


               1/6
              *3   ),

                          2/3
 X=((SQRT(31) - 3*SQRT(3))   *SQRT(3)*I

                              2/3    2/3
     - (SQRT(31) - 3*SQRT(3))    + 2   *SQRT(3)*I

        2/3                             1/3  1/3
     + 2   )/(2*(SQRT(31) - 3*SQRT(3))   *6

               1/6
              *3   ),

                           2/3    2/3
     (SQRT(31) - 3*SQRT(3))    - 2
 X=-------------------------------------}
                          1/3  1/3  1/6
    (SQRT(31) - 3*SQRT(3))   *6   *3
```

7.10.3 その他のオプション

SOLVESINGULAR スイッチがオンの時 (通常オンになっています) には x+y=0,2x+2y=0 のような縮退した方程式に対しては、適当な任意定数を持ち込んで解を表します。自明な特異方程式 0=0、もしくは逆が唯一でないような関数を含む方程式を解くと、それぞれ任意の複素数や整数を表す ARBCOMPLEX(j) または ARBINT(j), (j=1,2,...) という新しい未知変数を使って解を表します。分岐の枝を自動的に選択させるためには off allbranch; を行っておけばよい。 新たな未知変数を持ち込ませない為には、OFF ARBVARS を実行すれば良い。こうすれば、解には新たな未知変数は

含まれず、もともとの変数名を使って表されます。OFF SOLVESINGULAR とすると、自明な特異方程式の解は結果から取り除かれます。

関数の逆関数を定義するには、例えば、

```
put('sinh,'inverse,'asinh);
put('asinh,'inverse,'sinh);
```

として、これに対する簡約規則、例えば、

```
for all x let sinh(asinh(x))=x, asinh(sinh(x))=x;
```

を定義しておけばよい。ただし逆が唯一でないような関数については、SOLVE モジュール中で^(ベキ乗)、 SIN や COS といった関数に対して行われているのと同じ処理を行う必要があります。

ASIN と ACOS の引数は実数部分の絶対値が 1 を越えてはいけないことを調べてはいません。同じように、LOG の引数は虚数部分の絶対値が π を越えないことを調べていません。しかし、LET 文を使えばこれらのチェック (また数値でない引数に対するユーザの問い合わせ等の処理も含めて) を行うようにプログラムするの事は可能です。

7.10.4 パラメータと変数の依存関係

SOLVE の第二引数で与えられる変数リストは、方程式の解の構造に大きな影響を与えます。この変数リストにない未知数は完全な自由パラメータとして処理されます。つまり、

```
solve({x=2*z,z=2*y},{z});
```

では、結果は空リストになります。なぜなら、任意の x と y に対してこの二つの式を満足するような関数 $z = z(x, y)$ は存在しないからです。このような場合、変数 requirements は方程式系のパラメータに対する拘束条件を与えます:

```
requirements;

{x - 4*y}
```

方程式に含まれるパラメータを、拘束条件をゼロにするように設定したときにのみ無くなるような矛盾が存在する場合、方程式の形式解が存在しなくなります。線形システムに対しては、拘束条件の集合は完全です: つまり、拘束条件を満たすパラメータの組み合わせに対して、方程式は可解となります。上の例で言えば、$x = 4$ を代入すると、方程式は無矛盾になります。非線形な方程式に対しては、ただ一つの拘束条件しか発見できません。もし、方程式系が一つ以上の拘束条件を持っているような場合には、一つづつ順に見つけ出して行かなければなりません[2] 。拘束条件の集合は、また、パラメータ間の依存関係を表しています。x と y の内の一方が自由パラメータ

[2] 線形と非線形の矛盾した方程式系で取り扱いが異なってくるのは、拘束条件は方程式の形式解を求めるときの副作用として導かれる、というアルゴリズムによるものです。例えば、$solve(\{x = a, x = b, y = c, y = d\}, \{x, y\}$ では拘束条件として $\{a - b, c - d\}$ が得られますが、$solve(\{x^2 = a, x^2 = b, y^2 = c, y^2 = d\}, \{x, y\}$ の場合には、$\{a - b\}$ が得られます。

であり、方程式の形式解は、solve への変数リストとしてこの変数を追加することにより、計算できます。制約条件の集合は唯一には決まりません。別の集合の取り方も可能な場合があります。

パラメータを持った方程式系は、形式解を持つことがあります。例えば、

```
solve({x=a*z+1,0=b*z-y},{z,x});

    y     a*y + b
{{z=---,x=---------}}
    b        b
```

は、パラメータのどんな値に対しても解になっているわけではありません。変数 assumptions は、制約条件の集合を示します。これらの条件がいずれもゼロでない場合にのみ、得られた解は方程式の解となっています。制約条件が一つでもゼロになる場合は、この形式解では扱えない、特殊ケースになっています。上の例では、制約条件の値は、

```
assumptions;

{b}
```

となります。つまり形式解は、$b = 0$ の場合を除いて成立します。明らかに、制約条件がゼロになる場合、方程式は特異型になります。制約条件の集合は、線形および非線形の場合共に完全です。

SOLVE は、(期待される) 計算時間を短縮するために、計算における変数の順序を変更します。このような順序の変更は、varopt スイッチで制御することができます。このスイッチは通常オンです。オフにすると、指定された変数の順序、もしくは変数のリストを指定しない場合にはシステムで決まっているカーネルの順序に基づいて計算が行われます。このスイッチの効果については、次の例で示されます。

```
s:= {y^3+3x=0,x^2+y^2=1};

solve(s,{y,x});

                6       2
{{y=root_of(y_  + 9*y_  - 9,y_),

         3
      - y
  x=-------}}
       3

off varopt; solve(s,{y,x});

                6       4        2
{{x=root_of(x_  - 3*x_  + 12*x_  - 1,x_),
```

```
          4      2
     x*( - x  + 2*x  - 10)
y=-----------------------}}
              3
```

最初の例では、 solve は x に関する次数の方が高いので、変数リスト $(y_i, x(y_i))$ で計算します。通常、このような並べ替えによって、Gröbner 基底の計算が早くなります。二番目の例では、引数で指定した変数の順序 $(x_i, y(x_i))$ で、計算が行われます。方程式が一つ以上の自由変数を含んでいるような場合、このような変数の順序の制御は特に重要になってきます。スイッチ varopt をオフにする代わりに、depend 宣言を使って、変数間の部分的な依存関係を宣言することができます。solve は、変数の依存関係に従い、従属している変数が後になるように、変数リストの並べ替えを行います。

```
on varopt;
s:={a^3+b,b^2+c}$
solve(s,{a,b,c});

                          3       6
{{a=arbcomplex(1),b= - a ,c= - a }}

depend a,c; depend b,c; solve(s,{a,b,c});

{{c=arbcomplex(2),

              6
  a=root_of(a_  + c,a_),

        3
  b= - a }}
```

この例では、solve は変数 c は、a と b の後になるように強制されます。しかし、a と b との順序については強制されません。

7.11 偶演算子と奇演算子

演算子は EVEN、もしくは ODD 宣言によって、その第一引数に関して**偶**もしくは**奇**であると宣言することができます。 このように宣言された演算子を含む式は、その演算子の第一引数が負の場合には、偶もしくは奇関数として変換が行われます。それ以外の引数については影響ありません。さらに、例えばもし F が奇関数として宣言されているとすると、f(0) は F が NONZERO 宣言によって、**非負**と宣言されていない限り、0 に置換されます。例えば、次の宣言

```
even f1; odd f2;
```

は次のことを意味します。

```
f1(-a)    ->    F1(A)
f2(-a)    ->   -F2(A)
f1(-a,-b) ->    F1(A,-B)
f2(0)     ->    0.
```

上の例での最後の置き換えを禁止するには `nonzero f2;` と宣言して置けばよい。

7.12 線形演算子

演算子は、その第一引数が第二引数のべき乗の項に関して線形な演算子であると宣言することができます。もし、演算子 F を線形演算子として宣言すると、任意の和に対する演算 F は F の和の形に展開されます。第二引数のべき乗でない係数は F の外に出されます。F は少なくとも二つの引数を持つものでなければなりません。さらに、第二引数は式ではなく識別子 (もしくはもっと一般にカーネル) でなくてはなりません。

例:

もし、F が線形として宣言されているものとすると、

```
                              5
f(a*x^5+b*x+c,x) ->  F(X ,X)*A + F(X,X)*B + F(1,X)*C
```

もっと正確にいうと、第二引数の変数およびそのべき乗の項だけではなく DEPEND でその変数に依存関係があると宣言された変数およびこれらの変数に関する式 (例えば `cos(sin(x))` 等) は演算子 F の外に出されません。

例えば、オペレータ F と G を線形演算子として宣言するには、linear を使って、次のように入力します。

```
linear f,g;
```

第一引数で与えられた式が第二引数の変数に関して依存する項と独立な項に分けられます。残りの引数については関与しません。評価は次の規則に従って行われます。

```
f(0) -> 0
f(-y,x) -> -F(Y,X)
f(y+z,x) -> F(Y,X)+F(Z,X)
f(y*z,x) -> Z*F(Y,X)        Z が X と独立なとき
f(y/z,x) -> F(Y,X)/Z        Z が X と独立なとき
```

要約すれば、Y が未知変数 X に “依存する” のは次のいずれかが満たされるときです。

1. Y は X を変数として含む式であるとき。例えば: `cos(sin(x))`

2. 式 Y に含まれる変数のいずれかが DEPNED によって X の関数であると宣言されているとき。

このような線形演算子の使用例は、Fox, J.A. と A. C. Hearn による論文 “Analytic Computation of Some Integrals in Fourth Order Quantum Electrodynamics” Journ. Comp. Phys. 14 (1974) 301-317 にあります。この論文には 4 次の量子電磁力学に現れるある種の数式の定積分を計算する完全なプログラムが載っています。

7.13 非可換演算子

演算子は NONCOM と宣言することにより、積に関して非可換な演算子として定義できます。

例:

noncom u,v; と宣言した後では数式 u(x)*u(y)-u(y)*u(x) および u(x)*v(y)-v(y)*u(x) は評価しても入力したそのままの形の式が返されます。U, V が積に関して可換ではないので、これらの式は 0 には簡約されません。

変数ではなく演算子 (上の例では U と V) に非可換演算の性質を与えることができることに注意してください。

LET 文でこれらの演算子に関する演算規則を定義することができます。特に、ORDP を使うことでこれらの演算子間の順序を決めることができます。

例:

規則

```
for all x,y such that x neq y and ordp(x,y)
   let u(x)*u(y)= u(y)*u(x)+comm(x,y);
```

は、任意の X と Y に対する u(x) と u(y) の交換則を定義しています。関数 ordp(x,x) は常に真となるので、定義の中で、最初に X と Y が等しくないことを調べて除外して置かないと、例えば u(x)*u(x) の計算で無限ループに入ってしまいます。

7.14 対称演算子と反対称演算子

SYMMETRIC 宣言は、演算子がその引数について対称な性質を持っていると宣言します。例えば、

```
symmetric u,v;
```

は、U と V がその引数に関して対称であると宣言します。REDUCE はこれらの演算子については内部の順序にもとずいて引数の順序を交換します。この順序はユーザは KORDER コマンドを使って変更することができます。

例えば、 u(x,v(1,2)) は u(v(2,1),x) となります。数値に対しては大きな数値ほど先頭に、数式に対しては、複雑な式ほど前にきます。

同じように、ANTISYMMETRIC は、演算子が引数に関して反対称であると宣言します。例えば、

```
antisymmetric l,m;
```

は演算子 L と M がその引数に関して反対称の性質を持つと宣言します。引数は内部の順序で並べかえられ、もし奇数回の交換を行ったときには演算の値は負になります。

例えば、 l(x,m(1,2)) は L と M について一回の交換が行われるので -l(-m(2,1),x) となります。l(x,x) のような式は 0 になります。

7.15 前置演算子の定義

ユーザは OPERATOR 宣言を使って新しい前置演算子を定義することができます。例えば、

```
operator h,g1,arctan;
```

は H, G1 および ARCTAN を前置演算子としてシステムに追加します。

これにより、h(w), h(x,y,z), g1(p+q), arctan(u/v) 等を式として使うことができます。しかし、これだけではこれらの演算子に対する意味や性質はなにも与えられていません。同じ演算子名に対して、それを 0-引数、1-引数、2-引数、...,n-引数の演算子としても使えます。

これらの演算子に対して計算規則を与えるには、LET 文を使うことができます。また、演算子に対してプロシジャとしての定義を与えることもできます。

もし対話モードで使っている場合、ユーザが識別子に対して演算子として宣言するのを忘れて演算子の意味で使ったときには、演算子として定義するかどうか問い合わせてきます。また、対話モードでなければ、演算子として宣言されているものとして処理されます。また、同じ識別子に対して、二度以上 OPERATOR 宣言を行った場合には警告メッセージが出力されます。

演算子として宣言されたものは大域的なものとして処理され、プログラムのどこからでも参照することができます。言い替えれば、ブロック (または関数) の中で宣言された演算子であっても、一度宣言された演算子はそのブロック (または関数) をこえて参照することができます。またブロック (関数) の実行が終わった後も定義は残ります。(演算子としての宣言を取り消すためには CLEAR を使います。)

7.16 内挿演算子の定義

INFIX および PRECEDENCE を使うことで、新しい内挿演算子を定義することができます。例えば、

```
infix mm;
precedence mm,-;
```

宣言 infix mm; を行うことによって、識別子 MM を内挿演算子として使うことができるようになります。

```
a mm b      は mm(a,b) の代わりとして入力できます。
```

宣言 precedence mm,-; は MM は演算子の優先順位として – 演算子の直後に位置するものとして定義します。つまり、– より強く、*よりも弱い優先順位が与えられます。従って、

```
a - b mm c - d     は     a - (b mm c) - d,
```

と解釈されますが、

```
a * b mm c * d     は     (a * b) mm (c * d).
```

となります。

内挿演算子や前置演算子は宣言されただけでは、如何なる意味も与えられず、具体的な計算規則等を LET 文や関数宣言を使って与えてやる必要があります。

ここで定義した内挿演算子はすべて二項演算子になることに注意してください。例えば、

```
a mm b mm c     は     (a mm b) mm c.
```

と解釈されます。

7.17 変数間の依存関係の設定/削除

REDUCE には、微分演算や線形演算子などのように、変数やカーネル間の依存関係を元に計算を行う機能を持っています。このような依存関係は DEPEND コマンドで表すことができます。DEPNED は任意の数の引数を取り、最初の引数で与えた変数が、二番目以降の変数に依存している (関数関係にある) ことをシステムに知らせます。例えば、

```
depend x,y,z;
```

は X は Y と Z の関数であることを宣言します。

```
depend z,cos(x),y;
```

は、同じように、Z が COS(X) と Y に依存していることを宣言します。

DEPEND コマンドで宣言された依存関係 (関数関係) は NODEPEND で取り消すことができます。このコマンドの引数は DEPEND と同じです。例えば、上記のように宣言されているときに、

```
nodepend z,cos(x);
```

は Z はもはや COS(X) には依存していないことを宣言します。しかし、Y に依存しているという宣言は有効なままです。

第8章　式の表示と構造化

この章では、ユーザが代数式の部分を取り出したり、式の構造を色々な形で出力する方法について説明します。また、REDUCE がどの様に設計されているかについての知識を与え、これによってユーザがシステムの構造について理解する助けを与えます。

8.1　カーネル

REDUCE はシステムのそれぞれの演算子に評価 (または簡約) を行う関数が対応しており、その関数によって数式が内部の正準形式[1] に変換されるようになっています。この内部形式は、もとの数式とはほとんど似つかぬものになっています。これについては、Hearn, A. C. による “REDUCE 2: A System and Language for Algebraic Manipulation,” Proc. of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, New York (1971) 128-133 に詳しく解説されています。

評価関数はその引数を二つの異なった方法の内のいずれかで変換します。最初のやり方は、数式をシステムで用意された他の演算子で書き直すことにより、元々の演算子が結果の式にまったく現れなくしてしまうことです。例えば+、* や/といった演算子に対応する評価関数はこのような処理を行います。なぜならば、正準形式にはこれらの演算子は現れないからです。また行列式を求める演算子 DET も同じです。この演算子は適当な行列式を計算し、結果の式には DET 演算子は現れません。もう一つのやり方では、評価した結果に元々の演算子が完全には評価されず残ったままの形で返されます。例えば、COS という演算子を考えると、COS を指数関数で書き直すような規則が適用されていない限り、COS を含んだ式を計算した結果にはやはり COS(X) の様な項が含まれます。このような、評価した後の式に現れるような演算子または関数はカーネルと呼ばれ、変数と同じように計算機の中では同一性[2] が保たれています。これにより、カーネルは変数として、後でこの演算子に関する置き換えの規則が定義されない限り、変数と同じ扱いで計算が行われます。

評価した結果に自明でない引数を持つ演算子を含んでいる場合、その引数の形は評価する時点

[1] 訳注:数式の表現には正準 (canonical) 表現、正規 (normal) 表現があります。二つの数式が同じ値の時には必ずまったく同じ形で表されるとき、この表現を正準表現といいます。同じ形になるとは限らないが、引算を行った結果は 0 になる (つまり、値が 0 である数式は必ず 0 と表されるとき) 正規表現といいます。例えば $sin^2(x) + cos^2(x) - 1$ の値は 0 ですが、このままの形では 0 とはなっていないので正規表現ではありません。

[2] REDUCE というより REDUCE の基となっている Lisp 言語では二つの変数の値が同じであると言うのには二つの意味があります。まったく同じ実体を指している場合と、出力した結果が同じ形になるという場合です。この二つは Lisp の関数 EQ と EQUAL で区別されます。通常 Lisp で代入 (例えば y:=x;) を行うと、変数 x に入っているデータのコピーが作られるのではなく、x と y で同じ実体を共有することになります。これに対して、数値に対する四則演算を行ったり、CONS や APPEND 等の演算を行った場合には新しく実体が作られます。二つの変数が同じ実体を指しているかどうかは EQ で判定できます。これに対して、値が同じかどうかは EQUAL で判定します。識別子 (ATOM) に対しては常に同一性が保たれています。つまり、関数 EQ で同じかどうかが判定できます。EQ で判定する方が早いので REDUCE で変数の比較を行うときには EQ が使われています。また変数の順序も実体を基にして決めています。

でのシステムの状態によって変わってきます。これらの引数は通常展開した形で表され、どの項も因数分解や共通項をくくり出す様な操作は行われていません。例えば、式 cos(2*x+2*y) は普通このままの形で返されます。通常式 2*x+2*y をトップレベルで評価した時には、係数の共通因子や変数の様な自明な因子はくくり出され、2*(x+y) と出力されます。もし演算子等の引数がこれと同じ形式で出力された方が望ましいときには、INSTR(“内部形式 (internal structure)” の略を意味する) スイッチをオンにすればよい。

カーネルに含まれる演算子が、その引数を並べ替えてもよい演算子である場合、システムはこれらの引数を内部での変数の固有の順序で整理します。いくつかのコマンドは引数としてカーネルを取ることができます。この場合、ユーザはシステムがこれらの引数に対してどんな内部順序を与えるかを指定する方法はありません。この困難を解決するために、数式として**カーネル形式**を定義しておき、評価するときにカーネルに変換されるようにしています。

カーネルの例:

```
a
cos(x*y)
log(sin(x))
```

しかし、

```
a*b
(a+b)^4
```

はカーネルではない。

カーネルは一般化された変数として使われ、変数に付随する代数的な性質はほとんどカーネルにも付随しています。

8.2 数式のワークスペース

以前行った計算の結果を保存して、後で取り出すためにいくつかの機構が用意されています。最も簡単な方法は、直前に行った結果を取り出します。もし、代数式の代入が行われたり、トップレベル (複合文のなかや関数定義の中ではなく) で評価が行われたりしたときに、評価の結果は変数 WS に保存され、ワークスペースとして参照できます。(もっと正確には、数式は変数 WS に代入され、後でさらに操作を行うため使うことができる。)

例:

数式 (x+y)^2 をトップレベルで評価し、続いてその式を Y で微分したいとすると、単に

```
df(ws,y);
```

とすれば望んでいた結果が得られる。

もし、ワークスペースにある式を後で使うために変数か式に代入しておきたいときは、SAVEAS 文が使えます。これの構文は

```
SAVEAS <式>
```

例えば、上の例で微分を行った後では、ワークスペースは 2*x+2*y を保持しています。この式を変数 Z に代入しておきたければ、

```
saveas z;
```

と入力すればよい。もし、ユーザが式に含まれる変数のいくつかを任意変数として代入しておきたければ、FOR ALL コマンドが使えます。

例:

```
for all x saveas h(x);
```

これにより、h(x) は 2*Y+2*Z となる。

最後の結果だけではなくそれ以外の結果を参照する方法については REDUCE の対話的使用法の節に記述してあります。

8.3 数式の出力

REDUCE は計算によって得られた数式の出力の仕方には非常に多くの柔軟性をもたせています。明示的に出力する形式を指定する文は用意されていません。これは、代数計算に於いては多くの場合出力される式の長さ等を前もって知ることはできず、フォーマット文を使って出力を行うようなことがあまり有用でないからです。その代わりに、REDUCE はユーザにモードを切り替えるためのスイッチを用意しており、これによって簡単に望む形式で出力できるようになっています。

最も極端な場合は出力を完全に行わないようにするスイッチです。OUTPUT スイッチをオフにすることで出力は抑制されます。これは通常オンになっています。このスイッチはファイルから大量に読み込む場合とか清書プログラムを使って “きれいな” 出力を得たい場合等で出力を制限したい場合に役に立ちます。

通常は計算した結果が必要で、どのようにしたら出力形式を適切に指定できるかを知る必要があります。前に言ったように、数式は通常展開した形で行幅一杯に出力されます。適当な出力宣言を使うことでこれを変更できます。まず最初に、出力行の長さを変更するオペレータを見てみましょう。

8.3.1 LINELENGTH 演算子

この演算子の構文は

```
LINELENGTH(NUM:整数):整数
```

で、出力の一行の長さを整数値 NUM に設定します。これは、以前の行の長さを返します。(従って、この演算子が返す値を覚えておけば後で元の値に戻すことができます。)

8.3.2 出力宣言

出力形式を制御するスイッチや宣言文について説明します。注意しておかなければならないのは、大きな数式をいろいろに変更した形式で出力するために変換をおこなうには、多くの計算時間と記憶容量を使うということです。もしユーザが出力の要する時間を速くしたいと思えば、PRI スイッチをオフにすることです。そうすれば出力はシステムで用意した形式 (これは数式の内部形式を反映した形式) で出力され、以下に説明するスイッチ等は効かなくなります。PRI は通常オンになっています。

PRI がオンであれば、次のような出力宣言やスイッチが使えます。

ORDER 宣言

ORDER 宣言 は出力時の変数の順序を決めるのに使います。構文は、

```
order v1,...vn;
```

で、vi はカーネルです。つまり、

```
order x,y,z;
```

は変数 X を含む項は Y の前に出力され、Y は Z の前に、そしてこの三つの変数はその他の変数全てよりも前に出力されます。order nil; は出力する順序をシステムで決めた標準に戻します。変数の順序は、引き続いて ORDER で宣言することで変更できます。後の ORDER 宣言で宣言された変数は以前に ORDER 宣言された変数よりも順序は後になります。従って、

```
order x,y,z;
order y,x;
```

は、Z の順序を Y や X よりも前にします。標準の順序は辞書式順序になっています。

FACTOR 宣言

この宣言は変数またはカーネルのリストを引数に取ります。FACTOR は因数分解を行うコマンド (これには FACTORIZE または FACTOR スイッチを使ってください。) ではなく、区切って出力させるためのコマンドです。宣言された変数に対するべき乗の項はまとめられ、べき乗とその係数の和との積の形で出力されます。

指定された前置演算子を含む式は FACTOR の引数に演算子名を指定することで factor が適用されます。例えば、

```
factor x,cos,sin(x);
```

は X と SIN(x) の全てのべき乗項それにすべての COS の関数は factor が適用されます。

FACTOR は引数の順序には何の影響も与えません。もしそうしたければ、ORDER 宣言を併用する必要があります。

宣言 remfac v1,...,vn; は変数 v1 から vn までの変数に対する factor の適用を取り除きます。

8.3.3 出力制御用スイッチ

これまで述べた宣言文に加えて、出力の形式を変更するためのスイッチがあります。これらのスイッチは宣言文 ON と OFF によって切り替えることができます。例をあげて、これらのスイッチの使い方を説明します。例えば、次の式を出力するものとします。

```
x^2*(y^2+2*y)+x*(y^2+z)/(2*a)
```

有効なスイッチには次のものがあります。

ALLFAC スイッチ

このスイッチは数式の全体、または括弧で囲まれた部分式中で単純な共通因子を捜し、これを括弧の外に出した形で出力します。ALLFAC がオフのときには

```
     2  2         2           2
(2*X *Y *A + 4*X *Y*A + X*Y  + X*Z)/(2*A)
```

と出力されますが、ALLFAC をオンにすると、

```
          2                 2
X*(2*X*Y *A + 4*X*Y*A + Y  + Z)/(2*A)
```

となります。ALLFAC は通常、そして特に断わっていない限り以下の例においても、オンになっています。

DIV スイッチ

このスイッチをオンにすると、数式の分母に現れる単純な因子による分子の割り算が行われます。従って、結果の出力には有理数や変数に関する負べきの項が現れます。DIV がオンのときには、

```
       2                 2  (-1)         (-1)
X*(X*Y  + 2*X*Y + 1/2*Y *A     + 1/2*A     *Z)
```

のように出力されます。DIV は通常オフになっています。

LIST スイッチ

このスイッチは数式の各項を一つの行に出力します。LIST がオンの時には次のように出力されます。

```
          2
X*(2*X*Y *A

    + 4*X*Y*A

       2
    + Y

    + Z)/(2*A)
```

LIST は通常オフになっています。

NOSPLIT スイッチ

通常二行以上に渡って数式を出力するときには、丁度区切りのよいところで行を分けるようにしています。これは計算時間のかかる仕事なので、もし数式のどこで切っても構わないのであれば NOSPLIT スイッチをオフにすることで出力を速くすることができます。このスイッチは通常オンになっています。

RAT スイッチ

このスイッチは変数が FACTOR によって宣言されているときにのみ役に立ちます。このスイッチをオンにすると、factor が適用された各項はそれぞれ分母で割った形で出力されます。`factor x;` と宣言されているものとし、RAT がオフのときには、

```
    2                    2
(2*X *Y*A*(Y + 2) + X*(Y  + Z))/(2*A)
```

と出力されます。RAT をオンにすると次のように出力されます。

```
 2                  2
X *Y*(Y + 2) + X*(Y  + Z)/(2*A)
```

RAT は通常オフです。

次に、X に factor が適用されているとして、DIV と RAT を共にオンにすると、結果は次のようになります。

```
 2                      (-1)   2
X *Y*(Y + 2) + 1/2*X*A    *(Y  + Z)
```

最後に、RAT がオンで ALLFAC がオフのときは元の形で出力されます。

```
 2   2              2
X *(Y  + 2*Y) + X*(Y  + Z)/(2*A)
```

RATPRI スイッチ

もし、数式の分子と分母が一行に書けるような式の時には、二次元の形で式が出力されます。つまり、分子と分母が別の行に出力され、間に点線の行が出力されます。例えば、(a+b)/2 は次のように出力されます。

```
A + B
-----
  2
```

このスイッチをオフにすることでこのような出力をやめて一行に出力します。

REVPRI スイッチ

通常数式の各項は次数の高い項から順に出力されます。ある場合 (例えば、べき級数を出力したい場合) には逆の順序で出力した方がよい場合があります。REVPRI スイッチをオンにすればこのような逆の順序での出力を行います。例えば、y*(x+1)^2+(y+3)^2 は

```
 2              2
X *Y + 2*X*Y + Y  + 7*Y + 9
```

と出力されますが、REVPRI がオンでは次のようになります。

```
           2            2
9 + 7*Y + Y  + 2*X*Y + X *Y
```

8.3.4 WRITE コマンド

簡単な場合には、出力を行うには特別なコマンドは必要ありません。なぜなら REDUCE は、入力の最後に終端記号としてセミコロンを使うと、評価した式を自動的に出力するようになっているからです。しかし、出力を行うコマンドが必要な場合があります。

つまり、FOR, WHILE や REPEAT 文では、場合によってはループを回る毎に何等かの出力を行った方が望ましいときがあります。

また、プロシジャ中で中間結果を出力したい場合や、特にファイル等に出力している時に結果に注釈をつけて出力したい場合があります。

WRITE コマンドは WRITE で始まり、後に項目をコンマで区切って指定します。項目としては、

1. 数式 (変数や定数の場合を含む)。式は評価された結果が出力される。
2. 代入文。式の:=演算子の右辺は評価され左辺に代入される。左辺のシンボルが出力され、":=" が書かれ、続いて右辺の評価した値が出力される。(代入文を実行したときとほとんど同じものが出力される。違うのは、例えば FOR 文中で使われているような場合に左辺の式中に FOR 文の制御変数が現れたときには、左辺の変数はそのときの制御変数の値で置き換えられます。)
3. 前後をダブルクォートで囲まれた任意の文字列。例えば"string"。

一つの WRITE 文中に指定された項目は一行に書かれます。(もし、出力が長いときには複数行に渡って書かれます。) 文字列はそのまま出力されます。WRITE コマンド自身は値を返しません。

WRITE コマンドの実行が終わると改行されます。従って、WRITE "";(空の文字列を出力させる) では空行が出力されます。

例:

1. もし、 A は X+5, B はそれ自身, C は 123, M は配列そして Q=3 とすると、

```
write m(q):=a," ",b/c," THANK YOU";
```

   は M(3) の値を x+5 にして、

```
M(Q) := X + 5 B/123 THANK YOU
```

   と出力します。5 と文字 B の間そして 3 と文字 T の間の空白は文字列中に空白を入れたので出力されています。

2. 1 から 20 までの整数の自乗の表を出力するには次のようにすればよい。

```
for i:=1:20 do write i," ",i^2;
```

3. 1 から 20 までの整数の自乗の表を出力し、同時にその値を配列 A の各要素に代入するには

```
for i:=1:20 do <<a(i):=i^2; write i," ",a(i)>>;
```

   とすればよい。この例では 2 列の数値が出力されます。もし、

```
for i:=1:20 do write i," ",a(i):=i^2;
```

   とすると、各行は A(i) := が出力されます。

4. 次のもう少し複雑な例は、P. Sconzo、A. R. LeSchack と R. Tobey によって最初に報告された有名な f と g 級数の計算を行います。"Symbolic Computation of f and g Series by Computer", Astronomical Journal 70 (May 1965).

```
x1:= -sig*(mu+2*eps)$
x2:= eps-2*sig^2$
x3:= -3*mu*sig$
f:= 1$
g:= 0$
for i:= 1 step 1 until 10 do begin
   f1:= -mu*g + x1*df(f,eps) + x2*df(f,sig) + x3*df(f,mu);
   write "f(",i,") := ",f1;
   g1:= f + x1*df(g,eps) + x2*df(g,sig) + x3*df(g,mu);
   write "g(",i,") := ",g1;
   f:=f1$
   g:=g1$
  end;
```

出力の一部分を書くと

```
                ... <前の出力> ...

                              2
F(4) := MU*(3*EPS - 15*SIG  + MU)

G(4) := 6*SIG*MU

                                      2
F(5) := 15*SIG*MU*( - 3*EPS + 7*SIG  - MU)

                              2
G(5) := MU*(9*EPS - 45*SIG  + MU)

                ... <続く> ...
```

8.3.5 零の抑制

零代入 (つまり<式> := 0 の形の代入文) が出力されるのは場合によっては煩わしい時があります。特に巨大な配列で要素の大部分が 0 の場合などです。このような代入文を出力させたくなければ、NERO スイッチをオンにすれば出力が抑制されます。

8.3.6 FORTRAN 形式での出力

もちろん REDUCE は数式中の変数等に数値を与えることで、式の数値計算を行うことができます。これには、他のところで指摘したように、ユーザは ROUNDED スイッチをオンにして実数計算を行うよう宣言しなくてはなりません。REDUCE における数値計算は、コンパイルして計算さ

れるのではなく逐次解釈されながら計算が行われるので、計算速度は余り速くありません。従って、数式計算を行った後で、大量の数値計算を行う必要があれば、FORTRAN 言語等の数値計算用の言語を使ってこれらの計算を行うことを勧めます。この目的の為に、REDUCE は結果の数式を FORTRAN 形式でファイルに出力する機能を持っています。

まず、FORT スイッチをオンにすると、システムは数式を FORTRAN 形式で出力します。式は 7 カラムから始まります。もし、式が一行を越えるような場合は、続く行の最初に 5 個の空白が出力された後、継続行の印 (.) が出力され、式の残りが出力されます。式が長く、継続行があらかじめ決められた限度 (変数 CARD_NO によって設定される) を越える場合は式はそこで一度終了され、新しい式が始まります。もし、式が変数への代入文であればその変数名への代入文が生成されます。それ以外の場合は、変数名 ANS への代入文が出力されます。もし、識別子や数値が FORTRAN の制限を越えている場合はエラーが起こります。

またプログラムの他の部分については WRITE コマンドを使って直接出力することができます。

例:

次の REDUCE 文

```
on fort;
out "forfil";
write "C     this is a fortran program";
write " 1    format(e13.5)";
write "      u=1.23";
write "      v=2.17";
write "      w=5.2";
x:=(u+v+w)^11;
write "C     it was foolish to expand this expression";
write "      print 1,x";
write "      end";
shut "forfil";
off fort;
```

はファイル forfil に次のような内容の出力を書き出します。

```
c this is a fortran program
 1    format(e13.5)
      u=1.23
      v=2.17
      w=5.2
      ans1=1320.*u**3*v*w**7+165.*u**3*w**8+55.*u**2*v**9+495.*u
     . **2*v**8*w+1980.*u**2*v**7*w**2+4620.*u**2*v**6*w**3+
     . 6930.*u**2*v**5*w**4+6930.*u**2*v**4*w**5+4620.*u**2*v**3*
     . w**6+1980.*u**2*v**2*w**7+495.*u**2*v*w**8+55.*u**2*w**9+
     . 11.*u*v**10+110.*u*v**9*w+495.*u*v**8*w**2+1320.*u*v**7*w
     . **3+2310.*u*v**6*w**4+2772.*u*v**5*w**5+2310.*u*v**4*w**6
     . +1320.*u*v**3*w**7+495.*u*v**2*w**8+110.*u*v*w**9+11.*u*w
```

```
     . **10+v**11+11.*v**10*w+55.*v**9*w**2+165.*v**8*w**3+330.*
     . v**7*w**4+462.*v**6*w**5+462.*v**5*w**6+330.*v**4*w**7+
     . 165.*v**3*w**8+55.*v**2*w**9+11.*v*w**10+w**11
      x=u**11+11.*u**10*v+11.*u**10*w+55.*u**9*v**2+110.*u**9*v*
     . w+55.*u**9*w**2+165.*u**8*v**3+495.*u**8*v**2*w+495.*u**8
     . *v*w**2+165.*u**8*w**3+330.*u**7*v**4+1320.*u**7*v**3*w+
     . 1980.*u**7*v**2*w**2+1320.*u**7*v*w**3+330.*u**7*w**4+462.
     . *u**6*v**5+2310.*u**6*v**4*w+4620.*u**6*v**3*w**2+4620.*u
     . **6*v**2*w**3+2310.*u**6*v*w**4+462.*u**6*w**5+462.*u**5*
     . v**6+2772.*u**5*v**5*w+6930.*u**5*v**4*w**2+9240.*u**5*v
     . **3*w**3+6930.*u**5*v**2*w**4+2772.*u**5*v*w**5+462.*u**5
     . *w**6+330.*u**4*v**7+2310.*u**4*v**6*w+6930.*u**4*v**5*w
     . **2+11550.*u**4*v**4*w**3+11550.*u**4*v**3*w**4+6930.*u**
     . 4*v**2*w**5+2310.*u**4*v*w**6+330.*u**4*w**7+165.*u**3*v
     . **8+1320.*u**3*v**7*w+4620.*u**3*v**6*w**2+9240.*u**3*v**
     . 5*w**3+11550.*u**3*v**4*w**4+9240.*u**3*v**3*w**5+4620.*u
     . **3*v**2*w**6+ans1
c     it was foolish to expand this expression
      print 1,x
      end
```

もし、WRITE 文の引数が行に渡って出力されるような式を含んでいる場合は、出力された結果のファイルを編集して修正する必要があります。WRITE 文の出力ルーチンは引数を順次書き出していきます。このとき継続行を出力するメカニズムは代入を行うための仮の変数名に続いて式を出力してしまいます。

最後に、REDUCE の**リスト**は FORTRAN に対応するものがありません。従って、リストを FORT がオンの時に出力しようとすると次の様な FORTRAN の注釈文が出力されます。

```
c ***** invalid fortran construct (list) not printed
```

(C ***** (LIST) は FORTRAN 文として出力できません)

FORTRAN 出力オプション

標準の FORTRAN 出力の形式を変更するいくつかの方法が用意されています。

数式は継続行の制限を満たすために複数の部分に分割されます。この制限値は次の代入文を実行することで変更できます。

```
card_no := <整数>;
```

ここで <整数> は継続行として許される最大行数です。CARD_NO の値は最初 20 になっています。

出力行の幅 (一行の長さ) は同じように次の代入文で変更できます。

```
fort_width := <整数>;
```

これは行の最大の長さを<整数>にします。FORTRAN 出力の幅は最初 70 になっています。

REDUCE は数式中の整数係数に小数点をつけます。(従って、例えば 4 は 4. と出力される。) これを行わないようにするには PERIOD スイッチをオフにすればよい。

FORTRAN のプログラムは通常小文字で出力されます。もし、大文字での出力の方が良ければ、FORTUPPER スイッチをオンにして下さい。

最後に、代入文でない式や、式を分割したときにつけられる変数名 ANS を別の名前にしたければ VARNAME 演算子で変更できます。これは、識別子名を引数に取り、以後 ANS の代わりにその識別子名を使うように設定します。VARNAME の値はその引数そのものになります。

FORTRAN やその他の言語への変換機能は SCOPE や GENTRAN パッケージ を使っても得られます。

8.3.7 式のファイルへの保存

式を後の計算で入力するためにファイルに出力して置きたいことがあります。ファイルを開いたり、閉じたりするコマンドについては別のところで説明してあります。しかし、出力したファイルをもう一度 REDUCE から読み込もうとする場合、通常の出力の様に “自然 (natural)” 形式で書かれていると入力できません。出力を入力可能な形式で出力するためには NAT スイッチをオフにします。これをオフにすると式の最後にはドル記号が出力されます。

例:

次のコマンド列を

```
off nat; out "out"; x := (y+z)^2; write "end";
shut "out"; on nat;
```

を実行するとファイル out には次のように出力されます。

```
X := Y**2 + 2*Y*Z + Z**2$
END$
```

8.3.8 式の構造の表示

結果の式が複雑な場合、答えの式の構造の骨格を表示されると便利なことがあります。STRUCTR 演算子は引数として与えられた式の構造を表示します。構文は次の通りです。

```
STRUCTR(EXPRN:代数式 [,ID1:識別子 [,ID2:識別子]]);
```

式の構造が木形式で出力されます。部分式には別の名前がつけられて表示されます。もし、ID1 を省略すると文字列 ANS で始まる名前がつけられます。この標準の名前は VARNAME 演算子で変更することができます。引数 ID1 を指定した時、これが配列名であれば、各部分式には配列の要素名がつけられます。それ以外の場合には、ID1 で指定された文字列で始まる名前がつけられます。(ID2 については後で説明します。)

式 EXPRN はスカラー式か行列が許されます。それ以外の場合はエラーになります。

例:

いまワークスペースには式 ((A+B)^2+C)^3+D が入っているものとします。STRUCTR WS; は (スイッチ EXP がオフの時には) 次の出力を出します。

```
ANS3

   where

                  3
      ANS3 := ANS2  + D

                  2
      ANS2 := ANS1  + C

      ANS1 := A + B
```

この操作の後でもワークスペースの値は変わりません。これは STRUCTR 演算子が通常は値を返さないからです。(もし STRUCTR が式の一部で使われたときには、その値は 0 となります。) さらに、部分式は表示されるだけで、保存されません。もし後になって、部分式を表示された名前で参照したければ、SAVESTRUCR スイッチをオンにしておきます。この場合、STRUCTR は式と部分式をリストにして返します。従って、SAVESTRUCTR がオンで上の例の STRUCTR WS; を実行すると、

```
                3              2
{ANS3,ANS3=ANS2  + D,ANS2=ANS1  + C,ANS1=A + B}
```

が返されます。このリストから、PART 演算子を使って必要な部分を取り出すことができます。例えば、上の例で ANS2 の式を取り出したければ、次の様に入力すればよいのです。

```
part(ws,3,2);
```

もし、 FORT スイッチがオンの場合、出力は逆順に行われます。つまり、部分式は必ずその値が定義された後で参照するように出力されます。省略可能な二番目の引数 ID2 は、この様に使われたとき、実際の式の名前を指定するために使われます。

例:

M は 2×1 の行列で、要素の値として ((a+b)^2 + c)^3 + d と (a + b)*(c + d) がそれぞれ入っているとします。さらに V は配列として宣言されているとします。このとき、EXP がオフで FORT がオンであれば、structr(2*m,v,k); と入力すると次のような結果が得られます。

```
V(1)=A+B
V(2)=V(1)**2+C
V(3)=V(2)**3+D
V(4)=C+D
K(1,1)=2.*V(3)
K(2,1)=2.*V(1)*V(4)
```

8.4 変数の内部順序の変更

変数 (またはカーネル) の内部順序は、計算を行うときに計算時間や使用する記憶容量に大きな影響を与えます。普通 REDUCE はある特別な順序を使っており、これは REDUCE を起動したときによって変わってきます。しかし、ユーザが変数の順序を指定することもできるようになっています。宣言 KORDER は次の構文を取ります。

```
korder v1,...,vn;
```

ここで、 Vi はカーネルです。 この宣言文により、Vi はこの順序で内部順序が指定されます。ここで指定されていない変数については Vi よりも順位が低いとされ、それら相互の順位については標準の順序がつけられます。このように KORDER を設定した後で、別の KORDER 宣言を実行すると、以前に指定した順位を取り消し、新たな順位が設定されます。KORDER NIL; は標準の順序に戻します。

ORDER 宣言 (これは単に出力するときの変数の順序を設定するだけです。) とは違って、KORDER は計算時間に大きな影響を与えます。この順序が計算時間に大きく影響するような微妙な問題では、ユーザは変数の順序についていくつかの実験を行って、問題に合った最もよい順序を決めることができます。

8.5 代数式の一部を取り出す

数式の計算において数式の一部を取り出すとか、または一部を別の式で置き直す必要がしばしばあります。REDUCE ではこのような目的のためいくつかの演算子が用意されています。この節ではこのような演算子について説明します。多項式や有理式のある部分を取り出す (主項や分子、分母等) 演算子についてはまた別の章で説明してあります。

8.5.1 COEFF 演算子

構文は次の通りです。

```
COEFF(EXPRN:多項式,VAR:カーネル):リスト
```

COEFF は EXPRN を変数 VAR の各べき乗の項の係数を取り出し、それをリストにして返します。最初の要素は VAR に関して 0 次の項になっています。

通常 EXPRN が VAR に関して多項式でなければエラーを起こします。しかし、RATARG スイッチがオンであれば、式の分母が VAR に依存しているかどうかは調べられず、すべて係数の一部であるとして処理されます。

例:

```
coeff((y^2+z)^3/z,y);
```

は次の結果を返します。

```
  2
{Z ,0,3*Z,0,3,0,1/Z}.
```

しかし、

```
coeff((y^2+z)^3/z,y);
```

はもし RATARG がオフの状態ではエラーとなり、RATARG がオンであれば

```
  3        2
{Z /Y,0,3*Z /Y,0,3*Z/Y,0,1/Y}
```

となります。

COEFF の返す結果のリストの長さは VAR の最高べきの次数に 1 を加えた値になります。上の例では 7 になります。また、変数 HIGH_POW に最高次の次数が、変数 LOW_POW には最低次の次数が代入されます。もし、EXPRN が VAR に関して 0 次であれば HIGH_POW と LOW_POW は共に 0 になります。

8.5.2 COEFFN 演算子

COEFFN 演算子は、COEFF が全部の項の係数を取り出すのに対して、多項式の指定した項の係数を取り出します。COEFFN は次の構文を取ります。

```
COEFFN(EXPRN:多項式,VAR:カーネル,N:整数)
```

これは、変数 VAR について 多項式 EXPRN の n 次の係数を返します。

8.5.3 PART 演算子

構文は

```
PART(EXPRN:代数式 [,INTEXP:整数])
```

この演算子は、評価の時点でのスイッチの設定をすべて考慮して、数式の出力**もしくはその文が評価され、出力が行われたものとして**された形から指定された部分を取り出します。ユーザはこの演算子を効果的に使うためには、REDUCE が数式を前置演算子 (REDUCE は数式をすべて前値形式で表しています) を使ってどの様な形で表しているのかについて、ある程度知っている必要があります。さらに、この演算子が使われるときには PRI スイッチはオンになっていないといけません。PRI スイッチがオンの場合には、数式は出力すべき形式に変換した式を一度前値形式で表します。ところが PRI がオフの場合ではこの変換に要する計算時間と記憶領域を節約するため、内部形式で表されている式の構造を直接たどって出力しています。ところが、PART 演算子は式が前置形式に変換されているものとして処理しています。そのため、PRI スイッチがオンでなければ、PART 演算子はうまく機能せず間違った結果を返してしまいます。

評価は順次整数のリストについて再帰的に行われます。つまり、

```
PART(<数式>,<整数 11>,<整数 2>)
    ->  PART(PART(<数式>,<整数 1>),<整数 2>)
```

などです。そして、

```
PART(<数式>) ->  <数式>.
```

となります。INTEXP は評価した結果が整数であるような式ならどんな式でも構いません。もし、値が 0 であれば演算子の名前が返されます。最後に、もし値が負の整数であれば式の最初からではなく最後から数えた項が返されます。

例えば、もし数式 a+b が A+B と出力された場合、(つまり変数の順序がアルファベット順の場合) には

```
part(a+b,2)  ->   B
part(a+b,-1) ->   B
```

そして、

```
part(a+b,0)  ->  PLUS
```

となります。演算子 ARGLENGTH が最上位の演算子の引数の数を求めるのに使えます。この ARGLENGTH 演算子は式が演算子をまったく含んでいない場合には、−1 を返します。例えば、

```
arglength(a+b+c) ->  3
arglength(f())   ->  0
arglength(a)     ->  -1
```

8.5.4 部分式への代入

PART コマンドは与えられた式の部分式への代入を行うこともできます。この場合、PART 演算子は代入文の左辺に現れ、右辺には変更すべき式を指定することになります。

例えば、通常の REDUCE のスイッチの状態では、

```
xx := a+b;
part(xx,2) := c;   ->  A+C
part(c+d,0) := -;   -> C-D
```

となります

注意しておくべきことは、上記の例での変数 xx の値はこの代入の後でも変化しません。さらに、配列や行列の要素等のような**即時評価**の性質を持っているような式とは異なり、part(xx,2) や part(c+d,0) の値も変化しません[3] 。

[3] このような代入文では part の第一引数で指定した式 (または変数の値) は変わりません。代入文の値として変更された式が返されます。従って、変更された値を使うには z:=part(a+b,0) := -; のように別の変数に代入しなくてはいけません。

第9章　多項式と有理式

計算機の数式処理システムの計算の多くは多項式や有理式に関する計算です。この章では、これらの計算に使えるようなスイッチや演算子について説明します。システムには一般の代数式に使えるような演算子 (DF や INT のような) もあります、これらについては別の章で説明しています。演算子に関して、その演算子の計算を行う前に、その引数が先に評価されます。さらに、演算は決まった型の引数に対してのみ行われ、そのときのシステムのスイッチの設定やモードの下では、与えられた引数を演算子が要求している型に変換できないときには、型が合っていないというエラーを起こします。例えば、引数としてカーネルが要求されているところで、a/2(A に対してなにも変換規則が定義されていないとして) が使われたときには

```
 A/2 invalid as kernel
(A/2 はカーネルではない)
```

というエラーメッセージが出力されます。

式の一部を取り出すというような演算を別にして、この章で説明する多項式や有理式に対する操作やスイッチの設定のあるものは、計算に必要なメモリ量や計算時間に大きな影響を与えるものがあります。ユーザは与えられた計算に対してこの章で説明しているスイッチ等を実験してみて、どの様にすれば最適な条件で計算が行えるかを調べてみる必要があります。

演算子 LENGTH は引数として与えられた式の分子の項の数を計算します。例えば、

```
length ((a+b+c)^3/(c+d));
```

は値 10 を返します。分母の式の項数を求めるには、DEN 演算子を使って有理式の分母を取り出してから LENGTH を呼ばないといけません。

```
length den ((a+b+c)^3/(c+d));
```

その他、現在用意されている演算、スイッチおよび引数の型や値のモードについて以下の節で説明します。

9.1　式の展開の制御

スイッチ EXP は式の展開を制御します。もし、このスイッチがオフの場合、式のべき乗や積は展開されません。しかし、この場合、結果として得られる式は正規形式 (normal form) にはなっていますが、正準形式 (canonical form) にはなっていないことに注意しなくてはいけません。つまり、0 になるべき式は 0 に簡約されますが、二つの等価な式を簡約した結果が同じ形になることは保証されません。

例: EXP がオンの時、二つの式

```
(a+b)*(a+2*b)
```

と

```
a^2+3*a*b+2*b^2
```

は共に後者の形に簡約されます。EXP がオフの場合には、これら二つの式は完全な因数分解を行うオプション ALLFAC が指定されていない限り、このままの形で扱われます。EXP は通常オンです。

多項式を引数とする演算子のなかには EXP スイッチがオフのときでは結果が違うものがあります。exp がオフの場合、しばしば項数は 1 になってしまいます。例えば、EXP がオフの場合、

```
length((a+b+c)^3/(c+d));
```

は値として 1 を返します。

9.2 多項式の因子分解

REDUCE は一変数や多変数の整係数多項式の因数分解を計算することができます。この計算を実行するパッケージは Cambridge 大学の Dr.Arthur C. Norman と Ms. P. Mary Ann Moore が作成したもので、P. M. A. Moore と A. C. Norman による論文 “Implementing a Polynomial Factorization and GCD Package”, Proc. SYMSAC ’81, ACM (New York) (1981), 109-116 に記述されています。

この機能を使う最も簡単な方法は、スイッチ FACTOR をオンにすることです。全ての式は出力するときに因数分解した形で出力されます。例えば、FACTOR がオンのとき、数式 A^2-B^2 は (A+B)*(A-B) と出力されます。.

明示的に与えられた数式を因数分解することもできます。FACTORIZE 演算子は次の構文を取ります。

```
FACTORIZE(EXPRN:多項式 [,INTEXP:素数]):リスト ,
```

オプションについては後で説明します。例えば、円分多項式 $x^{105}-1$ の全ての因子を求めたければ、

```
factorize(x^105-1);
```

と入力すればよいのです。結果は、因子と冪乗のペアを要素とするリストの形で出力されます。この例では、数値因子は含まれていないので、結果には X の多項式のみが出てきます。因子の数を計算したければ、分解した結果に対して、LENGTH を使えばできます。もし、数値因子が含まれているときには、例えば $(12 * x^2 - 12)$、結果には、数値因子が最初に現れます。しかし、この数値因子については素因数分解されていません。もし、素因数が必要でしたら、IFACTOR スイッチを使ってください。確率論的な算法によって素因数分解した結果が得られます。例えば、

```
on ifactor; factorize(12x^2-12);
```

とすれば、

```
{{2,2},{3,1},{X - 1,1},{X + 1,1}}.
```

が出力されます。

もし FACTORIZE の第一引数が整数の場合、IFACTOR スイッチに関係なくその整数の素因数分解が得られます。

IFACTOR スイッチは FACTORIZE の結果に対してのみ有効で、FACTOR スイッチをオンにしたときの出力には効果がありません。

因子の出力される順序はシステムに依存しており、ユーザはどういう順序で出力されるかを期待してはいけません。(ただし、数値因子は常に最初に現れます。) また、二つの因子の符号を変えても、積の符号は変わりませんが、REDUCE はときとしてこのような変形を行うことがあります。

因数分解のルーチンはまず最初に多変数の問題を一変数に変換し、これを小さな素数を法とする体上で分解します。このとき評価を行う点と使用する素数については乱数を用いて決定しています。このため、同じ問題に対しても計算の詳細については毎回異なってきます。もし、あらかじめどのような素数を選んだらよいかが分かっている場合には、FACTORIZE の第二引数で指定することができます。多項式の判別式を割り切るような素数を選んでしまうとエラーになりますが、ユーザが素数を指定した場合にはこの条件はチェックされません。従って、十分注意して使う必要があります。

整数体上だけではなく、他の係数体上での因数分解を行うこともできます。どのような係数体が使えるのかは、考えている数体の説明の項を参照してください。例えば、次の例では x^4+1 を 7 を法とする有限体上で因数分解します。

```
setmod 7;
on modular;
factorize(x^4+1);
```

因数分解パッケージはその計算をトレースする機能があります。一番簡単な使い方は、REDUCE のコマンド on trfac; を実行することです。これで、以後因数分解が呼ばれると計算の過程、多変数から一変数多項式への変換、素数の選択および因数の構成等を出力します。システムを調整するとか、バグを捜し出すときの為に、もっと深いレベルのトレースを行うこともできます。例えば、TRALLFAC スイッチ は全てのトレース情報を出力します。on timings; とすると、因数を構成していく各ステップでどれだけの計算時間を要しているか、どのステップで一番時間がかかっているかという情報が出力されます。その他、トレース情報を選択する方法には、

```
symbolic set!-trace!-factor(<数値>,<ファイル名>);
```

ここで、数値として 1, 2, 3 や 100, 101,... を指定すると、それぞれの計算で何を行っているかという情報がファイルに出力されます。数値は再起呼び出しの深さを主に指定しています。ファイル名として NIL を指定すると、ファイルではなく端末に出力されます。トレースが終わったら、ファイルを

```
symbolic close!-trace!-files();
```

として閉じて置きます。**注意:** MCD をオフにした状態で因数分解を使用するとエラーを起こします。

9.3 共通因子の除去

REDUCE の機能には、有理式の共通因子を取り除く機能がオプションとして含まれています。GCD スイッチをオンにすると分子と分母の共通因子を計算して除去します。(GCD は通常オフになっています。)

しかしながら、数値因子や単項因子についてはなにも指定しなくても自動的にチェックされます。

もし GCD がオンで EXP がオフの場合には、式中の無平方因子がチェックされます。またこのとき、係数の無平方因子もテストされ、くくり出されます。(これらの説明については Anthony C. Hearn による "Non-Modular Computation of Polynomial GCDs Using Trial Division", Proc. EUROSAM 79, published as Lecture Notes on Comp. Science, Springer-Verlag, Berlin, No 72 (1979) 227-239 を参照してください。)

例: EXP がオフで GCD がオンの場合、多項式 a*c+a*d+b*c+b*d は (A+B)*(C+D) となります。

普通、GCD の計算は上の論文に書かれているやり方で計算されます。しかし、REDUCE ではこの方法以外に、EZGCD 算法 (拡張されたザッセンハウスの算法) として知られている方法で計算することもできます。これはモジュラー計算を使った方法で、GCD スイッチに加えて EZGCD スイッチをオンにすることで EZGCD 算法を使った GCD 計算を行うように設定できます。

通常の問題に対しては EZGCD 算法を使ったほうが速く、場合によっては何十倍も速いこともあります。従って、このパッケージを使うだけのメモリがあれば ON EZGCD を使うことを強く勧めます。

EZGCD に関する説明は J. Moses and D.Y.Y. Yun, "The EZ GCD Algorithm", Proc. ACM 1973, ACM, New York (1973) 159-166 を見てください。

注意: EZGCD パッケージのプログラムの一部は因数分解のパッケージと共通に使われているので、因数分解のトレース機能を使うことで、トレース情報を得ることができます。

9.3.1 多項式の最大公約数

GCD は多項式の最大公約数を計算します。構文は、

```
GCD(EXPRN1:多項式,EXPRN2:多項式):多項式,
```

例:

```
gcd(x^2+2*x+1,x^2+3*x+2) ->  X+1
gcd(2*x^2-2*y^2,4*x+4*y) ->  2*X+2*Y
gcd(x^2+y^2,x-y)         ->  1.
```

9.4 最小公倍数

最大公約数の計算は、特に巨大な有理式を扱っている場合など、非常に時間がかかります。しかし、共通因子がくくり出せるのは、二つの有理式の和を求めるときに、それらの分母に共通因子を持っているような場合です。分母は分子に比べると多くの場合規則的な振舞いをするので、分母を計算するときには、完全な GCD を求めることで計算時間やメモリを節約することができます。分子については多くの場合、簡単なテストですませられます。つまり、計算の各ステップで分母についての最小公倍数を求めて置く方がよいのです。LCM スイッチはこのような目的のため用意されており、通常はオンになっています。

また、演算子としての LCM もあります。

LCM(EXPRN1:多項式,EXPRN2:多項式):多項式,

は二つの多項式 EXPRN1 と EXPRN2 の最小公倍数を計算します。

例:

```
lcm(x^2+2*x+1,x^2+3*x+2) ->  X**3 + 4*X**2 + 5*X + 2
lcm(2*x^2-2*y^2,4*x+4*y) ->  4*(X**2 - Y**2)
```

9.5 通分の制御

二つの有理式の和を求めると、REDUCE は通常分母を共通 (通分) にします。しかし、もしこの通分を行いたくなければ、スイッチ MCD をオフにしましょう。このスイッチは最大公約数の計算を行わせたくない場合や、有理式の高次微分係数を求める場合に特に役に立ちます。

注意: もし MCD がオフの場合、結果が正準形式や正規形式になることは保証できません。つまり、0 に等しい式が 0 に簡約されなくなります。このオプションは計算途中での式の膨張を抑えたいときに最も役にたちます。

MCD は通常オンになっています。

9.6 REMAINDER 演算子

この演算子の構文は次の通りです。

REMAINDER(EXPRN1:多項式,EXPRN2:多項式):多項式.

この演算子は EXPRN1 を EXPRN2 で割った余りを計算します。結果は、内部の変数順序に基づく本当の剰余で、擬剰余ではありません。多項式の擬除余 と擬除算 は polydiv パッケージを読み込むことで実行できます。詳細については、このパッケージのマニュアルを参照して下さい。

例:

```
remainder((x+y)*(x+2*y),x+3*y) ->  2*Y**2
remainder(2*x+y,2)             ->  Y
```

注意*:* 通常、余りは整数の範囲で計算されます。もし、整数以外のドメインで余りを計算したい時は、そのように宣言しておく必要があります。

例*:*

```
remainder(x^2-2,x+sqrt(2)); -> X^2 - 2
load_package arnum;
defpoly sqrt2**2-2;
remainder(x^2-2,x+sqrt2); -> 0
```

9.7 RESULTANT 演算子

これは次の構文を取ります。

RESULTANT(EXPRN1:多項式,EXPRN2:多項式,VAR:カーネル):多項式

これは与えられた二つの多項式 (係数は任意の domain が可能) の指定された変数に関する終結式 (Resultant) を計算します。これは Sylvester 行列の行列式で、二つの多項式から与えられた変数を消去して得られる式になります。もし与えられた多項式が共通因子を持たなければ終結式は 0 になります。

Bezout スイッチは、終結式の計算を制御します。通常はオフです。この場合、部分終結式アルゴリズムを使って計算されます。もし、Bezout スイッチがオンであれば、Bezout 行列を使って終結式が計算されます。しかし、この場合係数体としては多項式のみが許されています。

Resultant の符号は R.Loose の “Computing in Algebraic Extensions” in “Computer Algebra — Symbolic and Algebraic Computation”, Second Ed., Edited by B. Buchberger, G.E. Collins and R. Loos, Springer-Verlag, 1983 に従っています。つまり、A と B は X に依存していないとすると、

$$\mathrm{resultant}(p(x),q(x),x) = (-1)^{\deg(p)*\deg(q)} * \mathrm{resultant}(q,p,x)$$

$$\mathrm{resultant}(a,p(x),x) = a^{\deg(p)}$$

$$\mathrm{resultant}(a,b,x) = 1$$

となっています。

例:

```
                                    2
resultant(x/r*u+y,u*y,u)   ->  - y
```

代数拡大上での計算:

```
load arnum;
defpoly sqrt2**2 - 2;

resultant(x + sqrt2,sqrt2 * x +1,x) -> -1
```

有限体上での計算:

```
setmod 17;
on modular;

resultant(2x+1,3x+4,x)    -> 5
```

9.8 DECOMPOSE 演算子

DECOMPOSE 演算子は多変数多項式から、合成すれば元の多項式が得られるような一つの式と方程式のリストを返します。構文は、次の通りです。

```
DECOMPOSE(EXPRN:多項式):リスト
```

例えば、

```
decompose(x^8-88*x^7+2924*x^6-43912*x^5+263431*x^4-
               218900*x^3+65690*x^2-7700*x+234)
                 2                  2            2
         -> {U  + 35*U + 234, U=V  + 10*V, V=X  - 22*X}
                                    2
decompose(u^2+v^2+2u*v+1)  -> {W  + 1, W=U + V}
```

この分解は因数分解とは異なって、一意ではないことに注意してください。

9.9 INTERPOL 演算子

構文は:

```
INTERPOL(<値のリスト>,<変数>,<点のリスト>);
```

ここで<値のリスト> と<点のリスト>は同じ長さのリストで、<変数> は代数式 (カーネルを指定するのが望ましい) です。

INTERPOL は与えられた点で与えられた値を取る多項式を計算します。これは、与えられた変数に関する<値のリスト>の長さから一を引いた次数の多項式になります。与えられた点 p と値 v から $f(p) = v$ を満たす多項式として唯一に決まります。

Aitken-Neville の補間アルゴリズムを使って計算しており、丸め誤差や悪条件の問題に対しても安定した結果が得られることを保証しています。

9.10 多項式や有理式の一部を取り出す

この節での演算子は多項式や有理式の一部を取り出す操作を行います。式の構造を再配置するのに要する時間を考えなければ、これらの演算はほとんど時間がかかりません。

この節での演算子で第二引数にカーネル<VAR>を取るものは、第一引数で与えた式が<VAR>に関する多項式でなければエラーになります。係数は<VAR>に依存しない有理式であってもかまいません。しかし、RATARG スイッチがオンの時には、分母が<VAR>に依存しているかどうか調べず、分母はすべて係数の一部として処理されます。

9.10.1 DEG 演算子

この演算子は次の構文を取ります。

DEG(EXPRN:多項式,VAR:カーネル):整数.

これは多項式 EXPRN の変数 VAR を主変数としたときの、主項の次数を求めます。もし、変数 VAR が式 EXPRN に現れなければ 0 を返します。

例:

```
deg((a+b)*(c+2*d)^2,a) ->  1
deg((a+b)*(c+2*d)^2,d) ->  2
deg((a+b)*(c+2*d)^2,e) ->  0
```

もし、RATARG がオンであると、

```
deg((a+b)^3/a,a)       ->  3
```

となります。この場合には、分母の A は係数として処理されてしまうからです。RATARG がオフの場合には、このような場合にはエラーを起こします。

9.10.2 DEN 演算子

これは次の構文を取ります。

DEN(EXPRN:有理式):多項式

これは有理式 EXPRN の分母を取り出します。もし、EXPRN が多項式の場合は 1 を返します。

例:

```
den(x/y^2)   ->  Y**2
den(100/6)   ->  3
       [100/6 はまず 50/3 と簡約されるので分母は 3 になります]
den(a/4+b/6) ->  12
den(a+b)     ->  1
```

9.10.3 LCOF 演算子

LCOF は次の構文を取ります。

LCOF(EXPRN:多項式,VAR:カーネル):多項式

これは、多項式 EXPRN の変数 VAR に関する主項の係数 (主係数) を返します。もし EXPRN が変数 VAR を含んでいない場合は EXPRN がそのまま返されます。0 が返されます。

例:

```
lcof((a+b)*(c+2*d)^2,a) ->  C**2+4*C*D+4*D**2
lcof((a+b)*(c+2*d)^2,d) ->  4*(A+B)
lcof((a+b)*(c+2*d),e)   ->  A*C+2*A*D+B*C+2*B*D
```

9.10.4 LPOWER 演算子

LPOWER は次の構文を取ります。

LPOWER(EXPRN:多項式,VAR:カーネル):多項式

LPOWER は EXPRN の変数 VAR に関する主べきを返します。もし、EXPRN が変数 VAR に依存していない場合には、1 が返されます。

例:

```
lpower((a+b)*(c+2*d)^2,a) ->  A
lpower((a+b)*(c+2*d)^2,d) ->  D**2
lpower((a+b)*(c+2*d),e)   ->  1
```

9.10.5 LTERM 演算子

構文は、次の通りです。

LTERM(EXPRN:多項式,VAR:変数):多項式

LTERM は EXPRN の変数 VAR に関する主項を返します。もし、EXPRN が VAR を含んでいない場合は EXPRN を返します。

例:

```
lterm((a+b)*(c+2*d)^2,a) ->  A*(C**2+4*C*D+4*D**2)
lterm((a+b)*(c+2*d)^2,d) ->  4*D**2*(A+B)
lterm((a+b)*(c+2*d),e)   ->  A*C+2*A*D+B*C+2*B*D
```

互換性への注意: 以前の版の REDUCE では、LTERM はもし EXPRN が VAR に依存していない場合には 0 を返していました。現在では、EXPRN は常に LTERM(EXPRN,VAR) + REDUCE(EXPRN,VAR) に等しくなるように動作します。

9.10.6 MAINVAR 演算子

構文は、次の通りです。

MAINVAR(EXPRN:多項式):式

この演算子は多項式 EXPRN の (内部での多項式の表現に基づいた) 主変数を返します。もし、EXPRN が数値であれば 0 を返します。

例:

A は B、C や D よりも順序が高いと仮定する。

```
mainvar((a+b)*(c+2*d)^2) ->  A
mainvar(2)               ->  0
```

9.10.7 NUM 演算子

構文は、次の通りです。

NUM(EXPRN:有理式):多項式

この演算子は有理式 EXPRN の分子を返します。もし、EXPRN が多項式の場合は、EXPRN 自身を返します。

例:

```
num(x/y^2)  ->  X
num(100/6)   ->  50
num(a/4+b/6) ->  3*A+2*B
num(a+b)     ->  A+B
```

9.10.8 REDUCT 演算子

構文は、

REDUCT(EXPRN:多項式,VAR:カーネル):多項式

これは、EXPRN の変数 VAR に関する主項を除いた残りの多項式を返します。もし、EXPRN が VAR の関数でない場合は、0 を返します。

例:

```
reduct((a+b)*(c+2*d),a) ->  B*(C + 2*D)
reduct((a+b)*(c+2*d),a) ->  B*(C + 2*D)
reduct((a+b)*(c+2*d),e) ->  0
```

互換性への注意: 以前の版の REDUCE(3.5 版) では、REDUCT はもし EXPRN が変数 VAR に依存しない場合は EXPRN を返していました。現在の版では、EXPRN は常に LTERM(EXPRN,VAR) + REDUCT(EXPRN,VAR) に等しくなります。

9.11 多項式の係数演算

REDUCE は多項式の係数として色々な数体 (ドメイン) を選択することができます。通常は整数演算が行われますが、別の節で説明したように実数係数を扱うこともできます。有理数係数の多項式は、整数を分母とする整数係数の多項式としても扱えます。ON DIV とすることにより、有理数係数の多項式として出力することができます。REDUCE はこれら以外の係数オプションを持っており、この節ではこれらの係数について説明します。これらの係数モードは表駆動形式で動くようになっているため、新しいモードを追加するのは容易にできます。どのように実現されているのかについては R.J. Bradford, A.C. Hearn, J.A. Padget と E. Schrüfer による "Enlarging the REDUCE Domain of Computation," Proc. of SYMSAC '86, ACM, New York (1986), 100–106 にでています。

9.11.1 有理数係数多項式

有理数を分子と分母が整数であるような有理式として扱うのではなく直接多項式の係数として有理数を使うことができます。これは、RATIONAL スイッチをオンにすることでできます。

例: RATIONAL がオフの場合、入力された式 a/2 は分子が A で分母が 2 であるような有理式に変換されます。RATIONAL がオンのときには、入力された式は分子が 1/2*A で分母が 1 であるような有理式 (つまり多項式) になります。RATIONAL がオフの時には、入力された式は多項式ではなく有理式になってしまいます。

9.11.2 実数係数多項式

ROUNDED スイッチは多項式の係数として任意桁の実数を使えるようにします。実数の有効桁数は演算子 PRECISION で設定します。例えば、precision 50; とすると 50 桁の有効桁で計算されます。既定の桁数はシステムによって変わってきます。precision 0; と入力すれば、現在の有効桁数が分かります。実数モードでは、分母はモニック (最高次数の係数が 1) の多項式になるように分子の多項式の係数が調整されます。

例: ROUNDED がオンの状態で、a/2 という式を入力すると、分子が 0.5*A で分母が 1 の有理式に変換されます。

REDUCE は内部では、現在設定されている精度 PRECISION が機械で用意されている浮動小数点数の演算の精度以下であれば、それを使って計算します。より高い精度で計算する場合や扱う数値が浮動小数点数では表せなくなったときには**多倍長実数 (bigfloat)** を使って計算を行います。内部での計算精度は丸め誤差を防ぐため表示される桁数よりも二桁だけ多く取られます。多倍長実数は実数を小数部分と指数部分にわけ、それぞれを (任意精度) 整数で表します。これは、ハードウェアの浮動小数点数よりも多くの場合、より高精度な表現になっています。しかし、数値計算を行うときに効率的な方法とは言えません。もし、ハードウェアでの演算を使って計算を行ったのでは問題が起きるような場合、ROUNDBF スイッチをオンにすることで計算精度に関係なく多倍長実数を使って計算を行うように強制できます。非常に希な場合、このシステムがこのスイッチをオンにすることがあります。この時、ユーザに次の様なメッセージが出力されます。

```
 ROUNDBF turned on to increase accuracy
(精度を上げるために ROUNDBF をオンにしました。)
```

実数は通常現在の有効桁数までの桁数がすべて出力されます。しかし、もしユーザがもっと少ない桁数だけ表示して欲しい場合、表示する桁数を PRINT_PRECISION コマンドで指定できます。例えば、print_precision 5; と入力すると、現在の計算桁数に関わらず、数値は最大五桁までしか出力されません。

通常、ROUNDED スイッチがオンであれば、REDUCE は 1.0 を整数の 1 に変換します。これの変換を行なって欲しくない場合には、NONCONVERT スイッチをオンにして下さい。

多倍長実数は出力するときには、見やすさから五桁ずつ空白をいれて出力されます。もしこの機能が必要でなければ、BFSPACE スイッチをオフにしてください。

多倍長実数の演算に関するより詳しい説明については T. Sasaki による “Manual for Arbitrary Precision Real Arithmetic System in REDUCE”, Department of Computer Science, University of Utah, Technical Note No. TR-8 (1979) を参照してください。

実数を入力するとき、通常は入力を行っているときの有効精度で打ち切られます。もし、入力された数値を打ち切らず桁数全てを取り込みたいときには、ADJPREC (これは *adjust precision* の省略) をオンにしてください。もし ADJPREC がオンであると、整数や実数を入力したとき、自動的に精度が合うように PRECISION を増加させます。また、同時に精度を上げたことをユーザに通知します。

もし、 ROUNDED がオンであれば、有理数は実数に変換されます。有理数は有理数のままで計算を行って欲しければ、ROUNDALL をオフすればよいのです。このスイッチは通常オンです。

実数計算の結果は通常実数になります。しかし、二つの例外があります。それは、もし結果が計算の精度内で 0 または 1 に等しいときには整数の結果が返される事です。

9.11.3 モジュラー係数多項式

REDUCE は係数が与えられた数を法として計算されるような多項式を扱う機能を持っています。これを使うには、二つのコマンドを使います。SETMOD <整数> は素数の法を設定するのに、また ON MODULAR はモジュラー計算を有効にするために使います。例えば、setmod 3; と on modular; を入力した後では、多項式 (a+2*b)^3 は A^3+2*B^3 となります。

SETMOD の引数はモジュラー計算ではなく通常の整数計算の意味で行われます。従って、

```
setmod 3; on modular; setmod 7;
```

では正しく法として 7 が設定されます。

モジュラー数は通常 $[0, p-1]$ の範囲内の整数として表されます。ただし p は現在設定されているモジュローです。場合によっては、特に負の数が現れるような場合、$[-p/2+1, p/2]$ (もっと正確には $[-floor((p-1)/2), ceiling((p-1)/2)]$) の範囲内での整数で表す対称な表現の方が望ましいことがあります。CENTERED_MOD スイッチによって出力時に対称な表現を選択することができます。

ここで注意しておくことは、モジュラー計算は多項式の係数についてのみ行われるということです。現在の REDUCE では法が素数であることのチェックを行っておらず (p が素数のときは、x^{p-1} は p を法として 1 に簡約される)、指数を法について簡約することはできません。また法に関して素でない整数での割り算は “Invalid modular division (不正なモジュラー計算での割り算)” というエラーを起こします。

9.11.4 複素数係数多項式

REDUCE は変数 i の自乗が -1 に等しいとして処理できますが、これだけでは i を含む式を簡約する場合や、このような式を複素数体上での因数分解を計算するには十分ではありません。例えば、標準の状態では、

```
factorize(a^2+1);
```

は

```
{{A**2+1,1}}
```

という結果を返し、また

```
(a^2+b^2)/(a+i*b)
```

はこれ以上簡約されません。しかし、もしスイッチ COMPLEX をオンにすると、完全な複素数体上での計算が行われます。つまり、上の因数分解では、次の様な結果が得られます。

```
{{A - I,1},{A + I,1}}
```

そして、割り算では A-I*B に簡約されます。

COMPLEX スイッチは ROUNDED と一緒に使うことで、浮動小数点数の複素数が使えるようになります。

有理式の分母から複素数を取り除くのに、複素共役が使われます。COMPLEX がオフの時にこれを行うには RATIONALIZE をオンにしないといけません。

第10章　置換コマンド

REDUCE で重要なコマンドは変数や数式に対する置き換え規則を定義するコマンドです。このような置換規則の定義は、前置演算子 SUB、LET コマンドや規則集合を使って行われます。

10.1　SUB 演算子

構文は次の通りです。

SUB(<置換リスト>,EXPRN1:代数式):代数式

ここで <置換リスト> とは

VAR:カーネル=EXPRN:代数式

のような方程式を一つ以上並べたリストかまたは評価した結果がこのようなリストとなるような変数です。

SUB 演算子は数式 EXPRN1 中に現れる変数 VAR をすべて数式 EXPRN で置き直した結果の数式を返します。まず EXPRN1 が現在有効な規則を使って簡約された後、置き換えが行われます。そして最後に置き換えられた式が再評価されます。もし複数の変数に関する置き換え規則が与えられたときは、数式 EXPRN1 を再帰的に辿って行き、VAR を見つけるとそれを EXPRN で置き直していきます。式 EXPRN 自身に含まれる VAR は置き換えません。自明な場合として、SUB(EXPRN1) は EXPRN1 自身を返します。

例:

```
                                  2              2
sub({x=a+y,y=y+1},x^2+y^2) -> A  + 2*A*Y + 2*Y  + 2*Y + 1
```

そして、同じく s := {x=a+y,y=y+1} の時、

```
                                  2              2
sub(s,x^2+y^2)             -> A  + 2*A*Y + 2*Y  + 2*Y + 1
```

ここで、大域的な代入 x:=a+y 等は行われないことに注意して置きます。EXPRN1 は任意の代数式で、置き換えが意味を持つような式ならどんな式でも構いません。(例えば、スカラー式、リストや行列等) もし、EXPRN か EXPRN1 が置き換え過程が意味を持たない式の場合にはエラーを起こします。

次の例のように置換リストの前後の中括弧は省略しても構いません。

```
                                    2                2
sub(x=a+y,y=y+1,x^2+y^2)   -> A   + 2*A*Y + 2*Y   + 2*Y + 1
```

10.2 LET 規則

SUB による置き換えとは異なり、LET 規則で定義された置換規則は大域的で、また規則が別の規則で置き換えられるか CLEAR によって取り消されるまで有効です。

最も単純な LET 文の構文は、次の形です。

```
LET <置換リスト>
```

ここで <置換リスト> はコンマで区切られた規則のリストで、それぞれの規則は次の形を取ります。

```
<変数> = <式>
```

または

```
<前置演算子>(<引数>,...,<引数>) = <式>
```

または

```
<引数> <内挿演算子>,..., <引数> = <式>
```

例えば、

```
let {x = y^2,
     h(u,v) = u - v,
     cos(pi/3) = 1/2,
     a*b = c,
     l+m = n,
     w^3 = 2*z - 3,
     z^10 = 0}
```

リストを表す中括弧は、もし優先されていれば、省略できます。また、上の規則は七つの別々の LET 文で書いてもかまいません。

この LET 文が入力されたら以後、X は評価されると常に Y もしくは、X が評価された時点での Y の値の自乗となります。これは、この LET 文が実行されたとき、Y の値が Y 以外の値を持っていたとしてもそのようになります。(これに対して、代入文 x:=y^2 は代入が行われた時点での Y の値の自乗を X に代入します。この二つの結果はまったく違ったもの[1] になります。)

規則 let a*b=c は A と B が共に与えられた式の因子に含まれていれば、その式中の A と B の積が C に置き換えられます。例えば、a^5*b^7*w は c^5b^2*w になります。

[1] 訳注:例えばこの LET もしくは代入の後 Y の値を変更したような場合、X の値は二つの場合で異なってきます。但し、代入を実行した時点で Y の値が Y 自身の場合には同じ結果になります。

l+m に対する規則については、l+m が N に置き換えられるのではなく、普通 L が n-m に置き換えられます。このとき、M が n-l に置き換えられることはありません。このような場合の規則についてもっと詳しいことは 10.2.5 節で説明してあります。

w^3 に関する規則は W の 3 次以上のべきの項について適応されます

規則の最後の例 let z^10=0 は Z の 10 次以上の項を 0 にします。このような宣言はうまく使うと計算を効率よく実行することができます。(詳しくは 10.4 節を参照してください。)

このような LET 文中に新しい演算子を使った場合、もし規則がファイルから読み込まれているときには、自動的に OPERATOR として宣言されたものとして取り扱います。対話的に使っているときに入力されたときには DECLARE ... OPERATOR?(... を演算子として宣言しますか?) と聞いてきます。もし宣言してよければ Y、打ち間違い等で宣言したくない場合には N と入力して、改行 キーを押してください。

上にあげた例では、置換規則は与えられた規則と完全に一致する式に対して適用され、任意変数を含んだ適用は行われません。例えば、次のコマンド

```
let h(u,v) = u - v;
```

は h(u,v) を U - V に置き換えます。しかし、h(u,z) もしくは 演算子 H に U,V 以外の変数をつけたものは変換しません。

これらの単純な LET は代入文:=と同じ論理レベルにあります。x:=p+q は let x=y^2 として以前に定義された規則を消去します。また、逆に LET 文は以前に行われた代入を取り消します。

注意: 再帰的な規則、例えば、

```
let x = x + 1;
```

はエラーを起こします。なぜなら、X の評価は無限の評価を引き起こすからです。

```
x -> x + 1 -> (x + 1) + 1 -> ((x + 1) + 1) + 1 -> ...
```

同じように次のような二つの置換規則

```
let l = m + n, n = l + r;
```

も同じエラーを起こします。このような規則を定義した後 X もしくは L や M の値を評価しようとすると

```
X improperly defined in terms of itself
(X はそれ自身で不適当に定義されています)
```

というようなエラーが出力されます。

配列や行列の要素は**即その値が評価される**という性質を持っているため、LET 文の左辺に現れた場合、その要素に対する置換規則が定義されるのではなく、その要素が現在持っている値に対する置換規則が定義されます。例えば、

```
array a(5);
a(2) := b;
let a(2) = c;
```

では B が C に置き換えられます。a(2) は変わりません。

最後に、もし LET 文中のある式 (FOR ALL や SUCH THAT を含むような一般的な規則をも含めて) でエラーを起こした場合、それ以後の規則は定義されません。

10.2.1 FOR ALL ... LET

演算子の引数がどんな式であっても、それに対する置換規則を定義したい場合、FOR ALL 宣言が使えます。このコマンドの構文は

```
FOR ALL <変数>,...,<変数>
        <LET 文> <終端記号>
```

たとえば、

```
for all x,y let h(x,y) = x-y;
for all x let k(x,y) = x^y;
```

最初の例では h(a,b) は A-B と、h(u+v,u+w) は V-W となります。もし、演算子 H が二変数ではなくそれより少ないまたは多い数の引数で使われた場合は上記の例の LET は適用されません。またエラーも起こりません。

二つ目の例では k(a,y) は a^y になりますが、k(a,z) はそのままで変化しません。なぜなら、FOR ALL Y ... とは言っていないからです。

例での FOR ALL 文中での自由変数 X や Y はどんな変数名を使っても構いません。この自由変数の名前は LET 文が実際に適応されるときには影響ありません。しかし、どの様な名前を使ったのかは覚えておく必要があります。もし定義した LET 規則を後で取り消したいときには CLEAR コマンドで規則と取り消すときには、LET 文で使ったのと同じ自由変数を使わないといけないからです。

LET 文で定義する置換規則としてはもっと複雑な式を使うことができます。これについては後の一般式に対する置換の節で説明します。ほとんどの場合に、置換規則は受け入れられ、システムによってある一貫性をもって適用されます。しかし、もし左辺式が定数または自由変数だけになるような規則 (例えば、let 2=3 や for all x let x=2) の場合、システムはこのような置換規則を処理できないのでエラーメッセージ

```
Substitution for ... not allowed
(... に対する置換は許されていません。)
```

が出力されます。FOR ALL で指定された全ての変数は等号記号 (=) をつけて表されます。例えば上の例では自由変数 X は=X と表示されます。もし、FOR ALL 文での自由変数が LET 文での等式の両辺に現れない場合にはエラーになります。

10.2.2 FOR ALL ... SUCH THAT ... LET

もし置換を演算子中の変数やその他の式中での変数が与えられた一つの値の時だけではなく、かといって任意の値の場合でもない場合、条件付きの FOR ALL ...LET 文が使えます。

例:

```
for all x such that numberp x and x<0 let h(x)=0;
```

は h(-5) を評価すると 0 にしますが、H の引数が正の数値または数値でない場合、その形を変化させません。SUCH THAT キーワードのあとには任意の論理式を書くことができます。

10.2.3 代入や置換規則の削除

ユーザは CLEAR コマンドにより、代入や置換規則を削除することができます。これは次の構文を取ります。

```
CLEAR <式>,...,<式><終端記号>
```

例えば、

```
clear x, h(x,y);
```

配列や行列の要素は**直ちに値が評価される**性質を持っているため CLEAR コマンドによっては消去できません。例えば、A は配列名であるとすると、要素 a(3) の値を消去するには

```
a(3) := 0;
```

としなくてならず、

```
clear a(3);
```

ではできません。

しかし、配列 (または行列)A 全体を削除するには clear a; で出来ます。これは、単に配列 A の全ての要素を 0 にすることとは違います。実際、A が配列であることや、そのサイズ等がすべて削除されるので、A を別の配列として定義し直すことや、別のもの、例えば演算子として定義することができます。

もっと一般的な型の LET 宣言も CLEAR を使って取り消すことができます。単に LET の代わりに CLEAR を使い、LET 文で指定した式をそのまま書きます。ただし、等号記号 (=) 及び式の右辺はつけません。FOR ALL 文での自由変数は同じ名前の変数を使わなくてはなりません。また SUCH THAT で指定した条件の論理式は LET 文で指定したのとまったく同じ形[2] で書かなくてはなりません。(空白を挿入するのは構いません。)

例: LET 規則

[2] 規則を取り除くのにはパターンマッチを使って定義されている規則を捜し出して取り除いています。このため、まったく同じ形式で指定しないと取り除くべき規則が見つけられず、削除されません。

```
for all x such that numberp x and x<0 let h(x)=0;
```

を削除するには

```
for all x such that numberp x and x<0 clear h(x);
```

と入力すればよい。

10.2.4 重複した LET 規則

LET 規則を削除するには CLEAR 文を使う以外にも方法はあります。最初と同じもの、ただし右辺式は違う、新しい LET 文を定義すると古い定義は置き変わります。置き換えは新しい置換規則が前に定義した規則と互いに衝突している場合にも起こります。例えば、x^4 に対する規則は x^5 に対する規則を無効にして置き変わります。しかし、ユーザが同じ式に適応されるような複数の LET 規則を定義するときには注意しなくてはなりません。REDUCE がどの規則を適応するか、またどのような順序で適応するかは分かりません。新しい LET 規則を定義する前に古い規則を CLEAR で削除すべきです。

10.2.5 一般式に対する置換

これまで説明してきた規則は単純な式への置換規則のみでした。しかし、REDUCE が使っている置換のやり方は非常に一般的な方法を取っており、もっと複雑な式に対する置換規則を扱うことができます。

REDUCE が使っている一般的な置換の方法については Hearn, A. C. による “REDUCE, A User-Oriented Interactive System for Algebraic Simplification,” Interactive Systems for Experimental Applied Mathematics, (edited by M. Klerer and J. Reinfelds), Academic Press, New York (1968), 79-90, および同じく Hearn. A. C. の “The Problem of Substitution”, Proc. 1968 Summer Institute on Symbolic Mathematical Computation, IBM Programming Laboratory Report FSC 69-0312 (1969) で議論されています。

そこで議論されている理由から REDUCE は一般的なパターンマッチの方法は取っていません。しかし、現在のシステムは上記の論文で議論されている方法よりもっと洗練された方法を使っています。それで LET 文の引数として現れる規則には次のような構文のものが許されています。

```
<置換式> = <式>
```

ここで、<置換式>としては任意の意味のある規則であればどの様な式でも構いません。しかし、この規則の意味は<置換式>の形によって変わってきます。置換が行われる規則は完全に一貫性を持っていますが、この規則は置換をできるだけ効率よく実行するための実際的な必要性から少し複雑になっています。次にあげる規則は多くの場合でどの様に処理されるかを説明しています。

まず最初に、<置換式>は部分的に評価され、同類項をまとめ、識別子 (そしてカーネル) について整理されます。しかし、配列や行列の要素の様な**直ちに値が評価される**性質をもつ式を除いて、式のどの部分についても置き換え等は行われません。配列や行列の要素は実際に代入されて

いる値と置き換えられます。注意しておくべきことは、変数等の順序はシステムで決まっており、KORDER コマンド等を使っても変更出来ないことです。それぞれの場合については次のように処理されます。

1. もし、整理した規則の左辺が識別子であるか演算子またはべき乗の形であれば、これ以上変換されることなくこのままの形で置換規則が表に追加されます。
2. 左辺の最上位の式に積の演算子 (*) が現れたときには、定数項は右辺に移行されます。このように変換した結果の式が表に追加されます。例えば、

   ```
   let 2*x*y=3
   ```

   は

   ```
   let x*y=3/2
   ```

   と解釈され、従って x*y に対する置換規則が定義されます。この規則が適用されると x*y は 3/2 になります。しかし、X や Y のみの項は変わりません。
3. 左辺の最上位に演算子+, -または / が現れたときは、最初の項以外はすべて右辺に移行されます。従って、次の規則

   ```
   let l+m=n, x/2=y, a-b=c
   ```

   は

   ```
   let l=n-m, x=2*y, a=c+b.
   ```

   となります。

この場合に起こりうる問題の一つは、自由変数を含む式が右辺に移行されて左辺式が自由変数を含まなくなることによってエラーを起こすことです。例えば、

```
for all x,y let f(x)+f(y)=x*y
```

は次のように解釈され

```
for all x,y let f(x)=x*y-f(y)
```

結果として式の両辺が自由変数 Y を含まなくなります。

置換規則の左辺式に現れる配列や行列の要素が評価されてしまうことは、時として混乱をもたらします。例えば次のような場合を考えてみます。

```
array a(5); let x+a(2)=3; let a(3)=4;
```

最初の規則の左辺は X になり、二番目の規則は 0 になります。それで、最初の規則は X に対する置き換え規則で、二番目はエラーを起こします。

規則の集合からどの様な順序でそれぞれの規則が適用されるかについてはシステムの簡約処理についての詳しい知識を持たないと簡単には理解できません。また、システムのバージョンによっても変わるかも知れません。従ってユーザは規則の適用順序は不定であると仮定して、それに基づいてプログラムを作るべきです。

置換が行われた後、評価されている式はさらに置換規則が適用できるかどうか調べられます。これは、どの規則も適用されなくなるまで繰り返されます。

他の所で触れたように、置換規則が積の形になっている場合、式が割り切れる場合に置換が行われます。例えば次の規則

```
let a^2*c = 3*z;
```

は a^2*c*x を 3*Z*X に、また a^2*c^2 は 3*Z*C に置き換えます。もし積の形が規則で指定したのと比べてべき乗の部分がまったく同じ場合のみ置換したければ、MATCH コマンドを使います。

例えば、

```
match a^2*c = 3*z;
```

は a^2*c*x を 3*Z*X に置き換えますが、a^2*c^2 は置き換えません。MATCH は LET 文と同じように、FOR ALL をつけて使うことができます。

この節で説明した置換規則は CLEAR コマンドで削除することができます。このとき、もし必要ならば、定義を行ったときとまったく同じ FOR ALL 文を使わなくてはいけません。例えば、

```
for all x clear log(e^x),e^log(x),cos(w*t+theta(x));
```

ここでの自由変数の名前は、定義を行ったのと同じ名前を使う必要があります。

10.3 ルールリスト

ルールリストは SUB や LET とは違った方法で置換規則を定義します。実際、これを使うことで LET 文の機能はすべて行うことができ、また SUB の様に一時的な置換を行うこともできるので、それら両方の方法の持つ機能を提供します。しかし、この方法は比較的新しく導入されたもので、多くの REDUCE のプログラムは古い SUB や LET を使っています。

ルールリストは**規則**のリストで、次の構文を取ります。

```
<式> => <式> (WHEN <論理式>)
```

例えば、

```
{cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
 cos(~n*pi)      => (-1)^n when remainder(n,2)=0}
```

変数の前につけられたティルダはその変数が規則に対して**自由変数**、つまり LET 文での FOR ALL と宣言された変数に対応する、としてマークします。入力するときには、各規則で最初に現れた自由変数にこのようにマークします。さもなければ一貫性のない規則が定義されてしまいます。例えば、次の規則リスト

```
{cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
 cos(x)^2        => (1+cos(2x))/2}
```

は、余弦関数 (cosine) の積を和に変換する規則を定義しようとしていますが、これは正しくありません。なぜなら二番目の規則は引数が X の時のみ適応され、それ以外の引数に対しては置換が適用されないからです。同じ規則中に現れた同じ自由変数に対してティルダをつけても構いませんが、これはオプションです。(内部では、このような規則はすべて実際にマークされた変数と共に記録されています。) WHEN 句はオプションで、LET 文での SUCH THAT 句と同じように、規則を適用する時の条件を指定します。

規則リストは名前をつけることができます。例えば、

```
trig1 := {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
          cos(~x)*sin(~y) => (sin(x+y)-sin(x-y))/2,
          sin(~x)*sin(~y) => (cos(x-y)-cos(x+y))/2,
          cos(~x)^2       => (1+cos(2*x))/2,
          sin(~x)^2       => (1-cos(2*x))/2};
```

このように名前付けられた規則リストは必要なら中身を調べることができます。例えば、trig1; と入力するとリストの内容が出力されます。

規則リストは二つの使い道があります。一つは、LET コマンドで大域的に置換規則を定義するのに用います。例えば、

```
let trig1;
```

とすると、置換規則が大域的に定義され、次のように CLEARRULES を実行して規則を削除するまで有効になります。

```
clearrules trig1;
```

CLEARRULES は次の構文を取ります。

```
CLEARRULES <規則リスト>|<規則リストの名前>(,...) .
```

規則リストの二つ目の使い方は、WHERE 句で使うことにより、一時的な置換を行うのに使います。例えば、

```
cos(a)*cos(b+c)
   where {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2};
```

もしくは

```
cos(a)*sin(b) where trig1;
```

WHERE 句をつけた式の構文は次の通りです。

```
<式>
    WHERE <規則>|<規則リスト>(,<規則>|<規則リスト> ...)
```

従って上の例は次のようにも書けます。

```
cos(a)*cos(b+c)
   where cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2;
```

この構文では、WHERE 句で指定した規則のリストは WHERE 句の左の式にのみ適応され、この式以外では意味を持ちません。特に、この規則は以前に行われた WHERE や LET 文には影響を与えません。例えば、

```
let a=2;
a where a=>4;
a;
```

と入力すると、次のような出力が得られます。

```
4

2
```

WHERE は他の内挿演算子よりも低い優先度が与えられていますが、ELSE、THEN、DO や REPEAT 等よりも高い優先度を持っています。従って、例えば次の式

```
if a=2 then 3 else a+2 where a=3
```

は

```
if a=2 then 3 else (a+2 where a=3)
```

と解釈されます。

WHERE は記号モードでは 16.4 節で説明するように式の新しい変数を導入するのに用いられます。しかし、記号モードでの WHERE の使用は代数モードとは異なる意味を持ちます。従って、WHERE 文は二つのモード間で共有することはできません。

互換性への注意: ネットワークライブラリで配布された古いバージョンとの互換性を維持するため、現在の所置換を表す記号=>の代わりに等号= を使ってもよいようになっています。しかし、これは将来のバージョンでは使えなくなります。置換規則では記号=>を、また LET 文では等号記号を使うようにしてください。

ルールリストのより高度な機能

ルールリストのより高度な機能を使うことで、これまで議論してきたよりももっと複雑な規則をより簡単に定義することが可能となります。

これらの機能としては、以下のものがあります。

- 自由演算子
- //(ダブルスラッシュ) 演算子
- ~ (ダブルティルダ) 変数

パターンの左辺式に現れる**自由演算子**は、同じ個数の引数を取る任意の演算子にマッチします。自由演算子は、自由変数同じような書式で使われます。例えば、積に対する微分規則は以下のように表すことができます。

```
operator diff, !~f, !~g;

prule := {diff(~f(~x) * ~g(~x),x) =>
             diff(f(x),x) * g(x) + diff(g(x),x) * f(x)};

let prule;

diff(sin(z)*cos(z),z);

         cos(z)*diff(sin(z),z) + diff(cos(z),z)*sin(z)
```

// **演算子**は商を表す / の代わりに使用することで、分数式とマッチするように指定します[3] 。例えば、Gamma 関数の変換規則の定義中で、次のように規則を定義することができます。

```
gammarule :=
   {gamma(~z)//(~c*gamma(~zz))  => gamma(z)/(c*gamma(zz-1)*zz)
                  when fixp(zz -z) and (zz -z) >0,
    gamma(~z)//gamma(~zz) => gamma(z)/(gamma(zz-1)*zz)
                  when fixp(zz -z) and (zz -z) >0};

let gammarule;

gamma(z)/gamma(z+3);


          1
----------------------
  3      2
 z  + 6*z  + 11*z + 6
```

[3] 通常の変換規則では、a/b => c のようなパターンは a => b*c と解釈されて実行される。

この例では、要求される規則が適応されるためには、二つの規則を定義しておく必要があります。これは、 **(ダブルティルダ) 変数**を使うことで、次の様にもう少し簡単に書くことができます。

```
GGrule :=  {
   gamma(~z)//(~~c*gamma(~zz))  => gamma(z)/(c*gamma(zz-1)*zz)
    when fixp(zz -z) and (zz -z) >0};
```

一般的に、ダブルディルダ変数は使われている演算に関して中性的な要素に束縛されます。

規則	実際の引数	束縛
~z + ~~y	x	z=x; y=0
~z + ~~y	x+3	z=x; y=3 または z=3; y=x
~z * ~~y	x	z=x; y=1
~z * ~~y	x*3	z=x; y=3 または z=3; y=x
~z / ~~y	x	z=x; y=1
~z / ~~y	x/3	z=x; y=3

注意：分母に 変数が含まれているようなパターンは許されません。また、 変数がゼロになる場合を適切に処理していない場合再帰エラーを引き起こすことがあります。

```
let f(~~a * ~x,x)  => a * f(x,x) when freeof (a,x);

f(z,z);

***** f(z,z) improperly defined in terms of itself

(***** f(z,z) はそれ自身で不適切に定義されています。)

% BUT:

let ff(~~a * ~x,x)
       => a * ff(x,x) when freeof (a,x) and a neq 1;

ff(z,z);
                  ff(z,z)

ff(3*z,z);
                  3*ff(z,z)
```

演算子に関連した規則の出力

The operator SHOWRULES 演算子は識別子を引数としてとり、その識別子に関連した演算子に関するルールリストを返します。例えば、

```
showrules log;

{LOG(E) => 1,

 LOG(1) => 0,

      ~X
 LOG(E  ) => ~X,

                     1
 DF(LOG(~X),~X) => ----}
                     ~X
```

このようにして得られたリストに対してさらに操作を行なうことが可能です。例えば、rhs first ws; と入力すると、1 という値が返されます。ある種の演算子はこのような形式では表すことのできない特別な性質を持っている、ということに注意して下さい。例えば、奇関数であるとか、(procedure) 関数として定義されている内容とかは showrules では出力されません。

規則の適応順序

もし置換規則が重複しているような場合、どの様な順序で規則が適応されるかは重要な問題です。一般的には、この順序を正確に言うことは非常に難しい問題です。従って、一番よい方法は任意の順序で規則が適応されるものと仮定し、それに基づいてプログラムを作ることです。しかし、もしたった一つの演算子のみが関係している場合には、この演算子に対する置換規則の適応順序は次の様に決まってきます。

1. 少なくとも一つ以上自由変数を含んでいるような規則はまったく自由変数を含まない規則よりも先に適用される。
2. 最も最近に LET 文で定義された規則が先に適用される。
3. 一つの LET 文中に複数の規則がある場合には、並べられた順に規則が適用される。
4. 規則リスト中で、一つ以上の自由変数を含む規則は、まったく自由変数を含まない規則よりも先に、リスト中の順序で適用される。言い替えれば、リストの最初の要素が先に適用される。
5. 最初の項目と首尾一貫した形で、規則リスト中で自由変数を含まない規則は、一つ以上含む規則が適用された後に適用される。

例: 次の規則リストはガンマ関数の計算を行う。

```
operator gamma,gamma_error;
gamma_rules :=
{gamma(~x)=>sqrt(pi)/2 when x=1/2,
 gamma(~n)=>factorial(n-1) when fixp n and n>0,
 gamma(~n)=>gamma_error(n) when fixp n,
 gamma(~x)=>(x-1)*gamma(x-1) when fixp(2*x) and x>1,
 gamma(~x)=>gamma(x+1)/x when fixp(2*x)};
```

ここで、規則は値が分かっている場合もしくは定義から値が求められない場合を順に並べてあります。つまり、引数が正の整数の場合はそれ以前の規則で処理されているので、エラーを起こすのは引数が負の整数の場合のみです。そして、最後の規則が適用されるのは引数が $1/2$ の負の奇整数倍の場合だけです。上の規則の内で最初の規則は次の様にも書けます。

```
gamma(1/2) => sqrt(pi)/2,
```

しかし、このように書くとこの規則が適用されるのは一番最後になってしまうので、上の規則中で一番最後の規則では $x = 1/2$ の場合を WHEN 部分で除いて置かなければいけません。

演算子に関した変換規則の表示

演算子 SHOWRULES は 1 個の識別子を引数にとり、その演算子に関した変換規則のリストを返します。例えば、

```
showrules log;

{LOG(E) => 1,

 LOG(1) => 0,

      ~X
 LOG(E  ) => ~X,

                    1
 DF(LOG(~X),~X) => ----}
                    ~X
```

このようにして求めた規則リストはリストとして扱うことが可能です。例えば、`rhs first ws;` の値は 1 になります。演算子には、このような規則リストとしては表せない性質、例えば奇関数という性質やまたその演算子がプロシジャとして定義されているというような性質、を持っているものがあることに注意してください。

10.4 漸近コマンド

値が十分小さいと分かっている変数を含む多項式の展開を計算する場合、不用な計算を行わないようにその変数のあるべき乗以上の項は捨ててしまうことが望ましいことがしばしば起こります。このような時に LET 文を使うことができます。例えば、X に関して 7 次までの項が必要な場合、次のコマンド

```
let x^8 = 0;
```

を実行すると式中の x^8 以上の項は消されます。

注意: この特別な簡約は REDUCE では他の LET 文による置換とは違った方法で行われます。つまり、完全に計算を行った後で高次の項が削除されるのではなく、計算の途中に現れた高次項はすべて削除されます。従って、上の規則が有効であれば、x^10/x^5 分子が 0 に簡約されるので、0 になってしまいます。同じように、x^20/x^10 の計算では、分子と分母が共に 0 になってしまうので、0 での割り算を行ったというエラーを起こします。

上で説明した方法は、それぞれ異なった小ささを持つような複数の変数を含んでいる式の計算を行うのには使えません。このような場合、それぞれの変数に漸近的な重みをつけ、その重みの和に応じて、項を削除するかどうかを決定することが必要になってきます。これを行うために二つのコマンドが用意されています。WEIGHT コマンドは次のような式のリストを引数に取ります。

```
<カーネル> = <数値>
```

ここで <数値> は正の整数値 (整数式は許されません。) を指定します。このコマンドは<カーネル>の重みとして<数値>を対応させます。代数計算において、各項の重みの和があるレベル以上になっているかテストされます。もし、レベル以上の項があると、その項は削除されます。項の重みは、因子である変数の重みにその変数の重複度を掛けた値の和として計算されます。

重みのレベルは最初 1 になっています。これは次のコマンドによって設定できます。

```
wtlevel <数値>;
```

これは <数値> をシステムの新しいレベルとして設定します。<数値> は評価した結果が正の整数にならないといけません。WTLEVEL は引数として、NIL を取ることができます。この場合、現在の重みのレベルが返されます。

第11章　ファイル操作コマンド

多くの応用で、前もって用意して置いたREDUCEのファイルをシステムに読み込んだり、結果をファイルに出力したいことがあります。REDUCEはこのような目的のため、五個のコマンドを用意しています。それらは、IN、 OUT、SHUT、LOADとLOAD_PACKAGEです。 最初の三つのコマンドについてはここで説明します。LOADとLOAD_PACKAGEについては18.2節で説明します。

11.1　INコマンド

このコマンドは引数としてファイル名のリストを取り、システムにそのファイル (これらのファイルにはREDUCEの文やコマンドを含んでいます。) から入力するように指示します。ファイル名は識別子か文字列で指定します。実際のファイル名の形式はシステムに依存しており、また多くの場合、サイトによっても変わってきます。従って、実際の操作については使っているシステムの説明書を参照してください。例えば、

```
in f1,"ggg.rr.s";
```

は最初にF1を読み、続いてggg.rr.sを読みます。INコマンドの終端記号としてセミコロンを使うと現在読んでいるファイルの内容を端末もしくは現在出力しているファイルに書き出します。ドル記号 ($) をつけると、入力ファイルの内容は表示されません。セミコロンを使った場合でも、off echo; コマンドを入力ファイル中で使うことによって、出力を止めさせることができます。

INで読み込みを行っているファイルの最後には;END;を書いておかなくてはいけません。セミコロンを二ついれてください! これは例えば間違ってファイル中にBEGINをENDより多くいれた場合などに起こる誤りの箇所を見つけにくいエラーが起こるのを防ぐためです。また、これはファイル制御の管理が行われ、システムの効率を改善します。もし、ENDがなければ “End-of-file read(ファイルの終わりまで読んだ)” というエラーを起こします。

11.2　OUTコマンド

このコマンドは引数としてファイル名を一つだけ取ります。以後出力されたものはすべてこのファイルに書き出されます。これは別のOUTコマンドを実行することにより出力ファイルが変更されるか、もしくはSHUTコマンドによってファイルが閉じられるまで続きます。出力ファイルは幾つでも開くことはできますが、出力は一時に一つのファイルにしか書き出されません。ファイルが出力ファイルとしてオープンされた後、SHUTされるまで、新しい出力はこのファイルに追加されます。最初に出力を行うとき、または一度SHUTによってファイルが閉じられた後再びOUTコマンドで出力用にオープンした場合には、ファイルに入っていた古い内容は消されます。

出力ファイルを閉じないで出力を端末に切り替えるには、端末を意味する T というファイル名を使います。例えば、ファイル OFILE に出力するには、out ofile; とすればよいし、out t; とすると端末に出力します。

ファイルに書き出される出力は端末に表示されるのと同じ形式です。例えば、x^2 は二行に渡って書かれます。下の行には X が、そしてその上の行には 2 が出力されます。もし、ファイルに書き出したものを後で REDUCE で読み込もうとするにはこのような形式での出力は適当ではありません。このような場合には、最初に NAT スイッチをオフにしておきます。

例: ファイル ABCD に IN コマンドで読めるような形式で数式 XYZ の値を書き出すには

```
off echo$       % ファイルから入力しているときにはこれが
                % 必要です。
off nat$        % 出力を IN で読み込める形式で出力させる。
                % 式の最後には$が書かれます。
out abcd$       % 新しいファイルに書き出す。
linelength 72$ % 出力の行の幅を 72 文字にする。
xyz:=xyz;       % 出力を "XYZ :=" に続いて XYZ の値を書くよう
                % にする。
write ";end"$   % ファイルの最後を意味する ;end; を書く。
shut abcd$      % ファイル ABCD を閉じて、出力を端末に切り
                % 替える。
on nat$         % 通常の出力形式に戻す。
```

11.3 SHUT コマンド

このコマンドは引数として、以前 OUT コマンドで開いたファイル名のリストと引数として取り、それらのファイルを閉じます。多くのシステムでは、REDUCE のセッションが終了する前に、開いているファイルはすべて閉じておかなければいけません。さもなければファイルに出力したものを失ってしまうかも知れません。もしファイルが SHUT により閉じられ、その後同じファイル名に対して OUT コマンドで出力を行うと、新しい出力が書き出される前にファイルの中身は消去されます。

もし現在出力中のファイルが閉じられると以後の出力は端末に切り替えられます。OUT や IN で開いたファイル名以外のファイル名を SHUT で閉じようとするとエラーを起こします。

第12章　対話的使用のためのコマンド

REDUCE は対話的に使われる様に設計されていますが、バッチ的処理もしくは入力コマンドをファイルから入力することによりバックグラウンドジョブとして走らせることもできます。しかし、対話的使用とバッチ処理とでは基本的な違いがあります。対話的使用では、もしシステムが計算を行っている途中で例えば型宣言が行われていないといった不都合を発見した場合、計算を中断してユーザに訂正を求めます。しかしバッチ処理では不都合を発見する度に計算を中断してまた最初からやり直しをするのは実際的ではありません。従って、このような場合にはシステムは最も適当と思われるような仮定を行い計算を続けます。

またエラーを起こしたときの処理にも違いがあります。対話的処理ではユーザが誤りを訂正することができ、計算を続けることができますが、バッチ処理では、エラーを起こすとそれ以後の計算でのエラーを引き起こし、また無駄に計算時間を消費してしまうかもしれません。従って通常これ以後の計算を行わず、入力の残りは構文の検査を行うだけで読み飛ばします。このような場合には “Continuing with parsing only(構文テストのみを続行します)” というメッセージを出力してユーザに知らせます。もし、ERRCONT、 スイッチがオンの場合には、計算途中でエラーを起こしても以後の計算を続けます。

もし構文エラーを起こした場合、システムはエラーを発見した場所に三つのドル記号$$$をつけて表示します。対話的処理では、ED コマンドを使って修正するかまたは再度入力し直すことができます。対話的処理で構文エラー以外のエラーが起こったときは、エラーが起こったとき実行していたコマンドは保存され、後で RETRY コマンドで再実行させることができます。

12.1　以前の結果の参照

REDUCE の実行中に以前に行った計算の結果を参照したいことがしばしば起こります。この目的のため、REDUCE は対話処理した入力と結果の全ての履歴 (history) を保存しています。これらの履歴は REDUCE が対話的に使っているときに出力する数字で参照することができます。入力した式を参照したいときには、コマンドの番号 n を使って INPUT(n) でできます。結果の式は、WS(n) で参照できます。WS は直前に行った計算の結果を参照します。例えば、コマンド 1 は INT(X-1,X); でコマンド 7 の結果が X-1 であるとき、

```
2*input(1)-ws(7)^2;
```

は-1 となります。この時には積分計算は再度行われますが、

```
2*ws(1)-ws(7)^2;
```

と入力した場合は、同じ結果を積分を再度計算することなく得ることができます。

オペレータ DISPLAY は以前に入力したコマンドを表示します。引数が正の整数 n の場合、n さかのぼった入力コマンドを表示します。引数が ALL(実際は任意の整数以外の式) の場合には、以前の入力をすべて表示します。

12.2 対話的編集

ユーザが端末から入力した REDUCE のコマンドは対話的に修正することができます。また、ユーザが定義した関数定義を修正することもできます。トップレベルで、コマンド ED(n) で以前の入力を編集できます。ここで n はコマンドの番号です。ED;(つまり引数無しの場合) には直前の入力を編集します。

ED を呼び出した後、次のようなコマンドを使って修正することができます。

B	ポインターを最初の位置に戻す
C<文字>	次の文字を < **文字** > と置き直す
D	次の文字を消す
E	編集を終了して、編集したテキストを入力として読み直す。
F<文字>	次に < 文字 > が現れるところまでポインターを移す。
I<文字列><エスケープ文字>	ポインターの前に < **文字列** > 挿入する。
K<文字>	< 文字 > までの全ての文字を消す。
P	現在のポインターより後を表示する。
Q	編集を中断してエラーとして処理する。
S<文字列><エスケープ文字>	ポインターの直後から最初に現れる < **文字列** > を捜し、その直前にポインターを移す。
<空白> または X	右に一文字ポインターを動かす。

上の表はエディターにコマンドとして疑問符と改行を入力すると表示されます。エディターは鍵括弧 (>) を入力促進文字列として使います。コマンドは一行にまとめて入力しても構いません。コマンドは改行キーを押すと実行されます。

コマンド x := a+1; を x := a+2; に修正して、実行するには次の編集コマンドを使えばよい。

```
f1c2e<return>.
```

対話的なエディターはユーザが定義した関数 (コンパイルされていない場合) を修正する場合にも使えます。これには次のように入力すればよい。

```
editdef <識別子>;
```

ただしここで<識別子> は関数名を表しています。そうすると関数定義が出力され、エディターに入ります。修正した後、エディターを終了すれば関数定義が書き直されます。

ある REDUCE のシステムでは、最新のウィンドウシステムによる入力の編集システムが組み込まれているものがある。これが可能かどうかについてはそれぞれのシステムの説明書を参照し

てください。もしこれが可能で有れば、それを使う方が、ED や EDITDEF を使って編集するよりもずっと使いやすいでしょう。

12.3 対話的なファイルの制御

もし、入力がファイルから読み込まれているときには、バッチ処理として扱われます。この場合にもしユーザが対話的な反応をして欲しければ `on int;` というコマンドをファイル中に書いて置けばよい。同じようにシステムからいちいち問い合わせがあるのが面倒であれば、`off int;` というコマンドを実行すればよい。INT の設定に関わらず、ファイルから入力したコマンドは履歴には保存されず、ED コマンドで編集することはできません。しかし、多くの REDUCE の移植では外部のエディターを呼び出してこのような編集を行えるようになっています。これについては使用しているシステムの説明書を参照してください。

REDUCE で対話的にファイルを操作するためのコマンドが二つ用意されています。PAUSE; はファイルの好きな場所に挿入しておくことができます。入力中にこのコマンドに合うと REDUCE は CONT? と表示して計算を中断します。もしユーザが Y(yes の略) と答えるとファイルの中断したところから計算を続けます。N(NO の略) と答えると REDUCE は以後端末からコマンドを入力して実行します。ユーザはいつでもコマンド `cont;` を入力することで制御をファイルに戻し、PAUSE コマンドで中断した箇所からファイルの残りを実行することができます。トップレベルでのユーザ端末から `pause;` と入力しても何の効果もありません。

第13章　行列計算

REDUCE の強力な機能の一つとして、行列計算の機能があります。このような計算を行える様に構文を拡張するため、MAT 演算子を導入しています。また次に述べるような変数および式の型を導入しています。

13.1　MAT 演算子

この前置演算子は $n \times m$ 行列を表すために使われます。MAT は n 個の引数を取り、それらは行列の行と解釈されます。それぞれは m 個の要素からなるリストでそれらは行の各要素と解釈されます。例えば、次の行列

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

は `mat((a,b,c),(d,e,f))` と書かれます。

一行からなる行列、

$$\begin{pmatrix} x \\ y \end{pmatrix}$$

は `mat((x),(y))` となります。内側の括弧は次のような一列の行列、

$$\begin{pmatrix} x & y \end{pmatrix}$$

(これは `mat((x,y))` と表される)、と区別する為に必要です。

13.2　MATRIX 変数

MATRIX 宣言は識別子を matrix 変数として宣言します。行列の大きさは宣言文で指定するか、その変数に代入を行ったときに自動的に決定されます。例えば、

```
matrix x(2,1),y(3,4),z;
```

は X を 2×1 行列 (コラム行列)、Y は 3×4 行列そして Z は後で大きさが決められる行列としてそれぞれ宣言します。

行列宣言はプログラムのどこに表れても構いません。一度シンボルが行列として宣言されると、そのシンボルは配列名、演算子名、関数名や通常の変数としては使えません。しかし、配列として再定義することは可能で、そのときにサイズを変更することもできます。注意しておくことは、行列として宣言された変数は**大域的**に宣言された変数として使われ、プログラムのどこからでも

参照することが出来ることです。つまり、ブロック (または関数) 内で定義された行列はそのブロック (または関数) の外からでも参照することができ、またそのブロック (または関数) の実行が終わった後でも定義は残っています。(行列宣言と取り消すには CLEAR を使います。) 行列の要素は配列と同じように指定します。例えば、x(1,1) は行列 X の最初の要素を指します。サイズが定義されていない行列の要素を参照するとエラーを起こします。サイズが宣言されている行列の要素は最初 0 に設定されています。行列の要素は**直ちに値が評価される**性質を持っており、それ自身がその値を表すことはありません。もしこのようなことをやりたいときには行列の要素として演算子を代入すればよい。例えば、次のようにすればよい。

```
matrix m; operator x;  m := mat((x(1,1),x(1,2));
```

13.3 行列の式

行列の演算については次の様に定義される通常の演算規則が適用されます。

```
<行列の式> ::=
        MAT<行列の記述>|<行列変数>|
        <スカラー式>*<行列の式>|
        <行列の式>*<行列の式>|
        <行列の式>+<行列の式>|
        <行列の式>^<整数>|
        <行列の式>/<行列の式>
```

行列の和と積は、それぞれの行列のサイズが合っていないといけません。そうでないとエラーを起こします。同じくべき乗は正方行列に対してのみ計算できます。負のべき乗は逆行列のべき乗として計算されます。a/b は a*b^(-1) として計算されます。

例:

X と Y が 2×2 の行列として宣言されているならば、次の式を評価したものは行列になります。

```
y
y^2*x-3*y^(-2)*x
y + mat((1,a),(b,c))/2
```

二つの行列の商を計算するのには普通 Bareiss による 2-ステップの消去法が用いられます。Cramer の方法を用いた別の計算法を使うこともできます。通常この方法は、行列のサイズが大きくかつ密な行列の場合でなければ効率は悪くなります。しかしながら、この二つの方法を比べた明確な統計データは取っていません。Cramer の方法を使うには、CRAMER スイッチをオンにします。

13.4 行列に関する演算子

演算子 LENGTH を行列に作用させるとその行列の行及び列の長さをリストとして返します。これ以外のよく使われる行列に関する演算子については、次の節で説明しています。LINALG と NORMFORM パッケージも参照して下さい。

13.4.1 DET 演算子

構文は次の通りです。

```
DET(EXPRN:行列):代数式
```

演算子 DET は行列の行列式を計算します。例えば、

```
det(y^2)
```

は行列 Y の自乗の行列の行列式を返します。また、

```
det mat((a,b,c),(d,e,f),(g,h,j));
```

は行列

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & j \end{pmatrix}$$

の行列式を返します。

行列式は **直ちに値が評価される**性質を持っています。つまり、次の文

```
let det mat((a,b),(c,d)) = 2;
```

は行列式そのものに対する規則を定義するのではなく、行列式の値 (つまり今の場合では $ad-bc$) に対する規則を定義します。

```
let a*d - b*c = 2;
```

と同じことをしていることになります。

13.4.2 MATEIGEN 演算子

構文は次の通りです。

```
MATEIGEN(EXPRN:行列,ID):リスト.
```

MATEIGEN は行列の固有値方程式を解いて、対応する固有値を求めます。変数 ID は固有値を表す変数名として使われます特性方程式の無平方分解が計算されます。結果は三つの要素を持つリストのリストで、最初の要素は特性方程式の無平方因子で、次はその重複度、最後は対応する固有ベクトル ($n \times 1$ 行列で表されている) を表しています。もし無平方分解された場合、リストの各先頭の要素の積を求めると、これは最小多項式になっています。縮退 (つまり一つの固有値に対して複数の固有ベクトルが存在する場合) が起こったときには固有ベクトルの表式には一つ以上の任意変数が含まれています。結果から、それぞれの部分を抜きだしてくるには、リスト操作の演算子を使います。

例: 次のコマンド

```
mateigen(mat((2,-1,1),(0,1,1),(-1,1,1)),eta);
```

は次のような出力を返します。

```
{{ETA - 1,2,

  [ARBCOMPLEX(1)]
  [             ]
  [ARBCOMPLEX(1)]
  [             ]
  [      0      ]

  },

 {ETA - 2,1,

  [      0      ]
  [             ]
  [ARBCOMPLEX(2)]
  [             ]
  [ARBCOMPLEX(2)]

  }}
```

13.4.3 TP 演算子

構文は次の通りです。

```
TP(EXPRN:行列):行列
```

この演算子は引数の行列の転置行列 (行と列を入れ換えた行列) を求めます。

13.4.4 トレース演算子

構文は次の通りです。

```
TRACE(EXPRN:行列):代数式
```

演算子 TRACE は正方行列のトレースを返します。

13.4.5 行列の余因子

構文は次の通りです。

COFACTOR(EXPRN:行列,ROW:整数,COLUMN:整数):代数式

演算子 COFACTOR は行列 EXPRN の行 ROW と列 COLUMN の要素に関する余因子を求めます。もし、ROW や COLUMN が整数でない場合や EXPRN が正方行列でない場合にはエラーを起こします。

13.4.6 NULLSPACE 演算子

構文は次の通りです。

```
NULLSPACE(EXPRN:行列の式):リスト
```

NULLSPACE は行列 A に対して、方程式 $Ax = 0$ を満たす一次独立なベクトルのリスト (基底) を返します。この基底は最大限、上半成分が分離できるように選ばれます。

b := nullspace a を行った後で、式 length b は A のヌルスペースの次元 を与えます。そして、second length a - length b は A のランクを与えます。行列のランクは次に説明する RANK 演算子を使って直接求めることもできます。

例: 次のコマンド

```
nullspace mat((1,2,3,4),(5,6,7,8));
```

は次のような結果を返します。

```
{
 [ 1  ]
 [    ]
 [ 0  ]
 [    ]
 [ - 3]
 [    ]
 [ 2  ]
 ,
 [ 0  ]
 [    ]
 [ 1  ]
 [    ]
 [ - 2]
 [    ]
 [ 1  ]
 }
```

REDUCE の行列形式に加えて NULLSPACE はその引数としてリストを要素とするリストを渡すこともできます。これは一列の行列と解釈されます。もし、この形式の引数が渡されたとき、結果のベクトルは同じようにリストで表されます。このような入力の構文は NULLSPACE を古典的な線形代数以外の応用で使われるために用意されています。

13.4.7 RANK 演算子

構文は次の通りです。

```
RANK(EXPRN:行列の式):整数
```

RANK は引数のランクを計算します。これは NULLSPACE と同様に引数として通常の行列かリストを要素とするリストを取ります。もし引数としてリストが渡された場合は、これは一列の行列または方程式の集合として解釈されます。

例:

```
rank mat((a,b,c),(d,e,f));
```

は値 2 を返します。

13.5 行列の代入

行列の式は代入文の右辺に現れることもできます。代入文の左辺 (これは変数でないといけませんが) は行列として宣言されていなくとも構いません。右辺の行列のサイズの行列として自動的に宣言され、その後右辺の値がその変数に代入されます。

線形方程式の解を求めるのに、このような代入文を使うことが出来ます。例えば、次のような方程式

```
a11*x(1) + a12*x(2) = y1
a21*x(1) + a22*x(2) = y2
```

の解を求めるには、次のように書けばよい。

```
x := 1/mat((a11,a12),(a21,a22))*mat((y1),(y2));
```

13.6 行列要素の評価

一旦行列の要素に値が代入されるとその値は普通の配列要素と同じ形式で参照できます。例えば、y(2,1) は行列 Y の 2 行 1 列目の要素を指します。

第14章　関数

同じ様な計算をパラメーターを変えて行いたい場合や、定義した演算子に完全な計算の手続きを定義したい場合があります。REDUCE はこのような目的の為に関数を定義する方法を提供しています。この一般的な構文は次の通りです。

```
[<関数の型>] PROCEDURE <関数名>[<変数リスト>];<文>;
```

ここで、

```
<変数リスト> ::= (<変数>,...,<変数>)
```

で、これについては次の節でもっと詳しく説明します。

REDUCE の代数モードでは<関数の型>は通常 ALGEBRAIC なので省略できます。関数の型としては INTEGER や REAL というのも使えます。INTEGER 型では関数の値は整数であるとされます。現在のシステムでは REAL 型としても何も起こらず、無視されます。しかし、将来もっと完全な型のチェックが行われるようになると、変わってくるかも知れません。ユーザは実数型の関数にのみ REAL 宣言をつけるようにしてください。変数リストが空の場合は省略できます。

ユーザが定義した関数は演算子として自動的に宣言されます。

ユーザが REDUCE のソースプログラムに容易にアクセスできるように、システムで定義している関数をユーザが再定義しないようには保護していません。もし、すでに存在している関数を再定義してしまうと、次のようなメッセージが出力されます。

```
*** <procedure name> REDEFINED
(*** <関数名> は再定義されました)
```

もしこのような出力が出た場合、ユーザが自分で定義した関数を再定義したのではないならば、システムが使っている名前と同じ名前の関数名を使ってしまった可能性があります。自分で使う関数の名前を変更した方が良いでしょう。また、すでにシステム内部の関数を書き直してしまっており、以後の動作がおかしくなるかも知れません。新しく REDUCE を起動し直して計算を行なった方がよいでしょう。

関数はすべてトップレベルで定義しないといけません。これらは大域的な関数として、プログラムのどこからでも参照できます。関数の中で、関数を定義しようとするとエラーを起こします。

14.1　関数頭部

それぞれの関数宣言には PROCEDURE(場合によっては前に ALGEBRAIC が付けられる) に続いて定義しようとしている関数の名前、および関数への仮引数のリスト (関数本体へパラメーターを渡

すために使われます) が並びます。これには三つの場合があります。

1. 引数も持たない場合。関数名の後に終端記号 (セミコロンかドル記号) を付けます。

```
procedure abc;
```

   このような関数を呼び出すときには `abc()` のように空の括弧を付ける必要があります。

2. 引数が一つの関数。関数名の後に括弧で囲って変数名を書くかまたは一つ以上の空白をおいて変数名を書きます。

```
procedure abc(x);
```

   または

```
procedure abc x;
```

3. 二つ以上の引数をとる関数。引数を括弧で囲み、引数同士はコンマで区切ります。

```
procedure abc(x,y,z);
```

最後の例では、`abc(u,p*q,123)` としてこの関数を呼び出すと、関数本体を実行するときには `U` の値が `X` の値として、`p*q` の値が `Y` の値として、123 が `Z` の値としてそれぞれ設定されます。関数の実行が終了した時、`X`、`Y`、`Z` の値は関数を実行する前の値に戻されます。また、関数本体で `X`、`Y` や `Z` の値を変更しても、`U`、`P`、`Q` そしてもちろん 123 の値は変わりません。(これは、値呼び出し (call by value) と呼ばれています。) 別の所で説明するように、この値呼び出しによる変数の値の保護をすり抜ける方法はあります。

14.2 関数本体

関数の頭部の終わりの区切りに続いて関数の動作もしくは関数の値を計算する文が一つ続きます。文の最後には終端記号をつけます。それがセミコロンであれば、定義された関数名が出力されます。ドル記号を使うと何も出力されません。

もし、計算したい結果がある式で与えられているならば、関数本体には関数頭部で指定した変数を使ってその式を書けばよい。

簡単な例:

`f(x)` は `(x+5)*(x+6)/(x+7)` という関数として定義したければ次のように入力すればよい。

```
procedure f x; (x+5)*(x+6)/(x+7);
```

これで、`f(10)` は 240/17 に、 `f(a-6)` は `A*(A-1)/(A+1)` になります。

もっと複雑な例:

関数 p(n,x) が正の整数 N に対して n 次のルジャンドル多項式を与える関数として定義したいとする。これは数学の公式

$$p_n(x) = \frac{1}{n!}\,\frac{d^n}{dy^n}\,\frac{1}{(y^2-2xy+1)^{\frac{1}{2}}}\bigg|_{y=0}$$

から、n 次のルジャンドル多項式 $p_n(x)$ は変数 x と y を含むある式を y に関して n 回微分を取り、その結果で y を 0 にした式を n の階乗で割れば得られます。

この公式は REDUCE で次のように簡単に書くことができます。

```
procedure p(n,x);
   sub(y=0,df(1/(y^2-2*x*y+1)^(1/2),y,n))
       /(for i:=1:n product i);
```

この定義を入力すれば、式

```
2p(2,w);
```

は次のようになります。

```
   2
3*W  - 1
```

もし、計算したい関数が一つの式で表されているのではなくいくつかの計算を経て求められるようなものであれば、グループ文や複合文を使うことができます。

例:

上のルジャンドル多項式の計算は上の例で示したように一つの式で表す代わりに複数の文から計算を行うように書き直すと、次のようになります。

```
procedure p(n,x);
  begin scalar seed,deriv,top,fact;
       seed:=1/(y^2 - 2*x*y +1)^(1/2);
       deriv:=df(seed,y,n);
       top:=sub(y=0,deriv);
       fact:=for i:=1:n product i;
       return top/fact
  end;
```

関数は再帰的に定義することもできます。つまり、関数の本体は定義しようとしている関数それ自身やこの関数を使って定義されている他の関数を使うことができます。例えば、ルジャンドル多項式を再帰的に定義すると次のようになります。

```
procedure p(n,x);
   if n<0 then rederr "Invalid argument to P(N,X)"
    else if n=0 then 1
    else if n=1 then x
    else ((2*n-1)*x*p(n-1,x)-(n-1)*p(n-2,x))/n;
```

ここで使っている REDERR という演算子は代数モードでの関数やブロック内の計算を実行中に簡単なエラー処理と計算の中断を行います。この演算子は引数として文字列を取ります。

少し注意しておくと、ここで例としてあげた p(n,x) の定義では、p(n-1,x) と p(n-2,x) を求めて、それから計算しています。これでは、低い次数の多項式を何回も計算することになるので、効率はよくありません。より効率的に計算を行うには、配列に計算した結果を保存しておき、再帰的に計算するときにすでに計算されている次数の多項式に対しては保存しておいた結果を使うように書き直した方がよいでしょう。このように定義を書き直すことは読者の練習問題として残して置きます。

14.3 関数内での LET 文

LET 文を関数本体で代入文の代わりに使うことで、値呼び出し (call by value) による保護をバイパスすることができます。もし、X が関数の仮引数もしくは局所変数として、関数本体で変数 X に対する代入の代わりに LET 文を使って、

```
let x = 123;
```

のようにしたとすると、変数 X ではなく、X に入っている値に対する LET 規則が定義されます。これは、ブロック文中での局所変数に対しても同じことが起こります。もし、X の値が変数ではなくもっと一般の式の場合、その式が LET 文の左辺に入った、LET 規則が定義されます。例えば、X の値が p*q とすると、let p*q = 123 を実行したのと同じ結果になります。

14.4 関数としての LET 規則

LET 文は関数を定義するもう一つの方法を与えます。

```
procedure abc(x,y,z); <関数本体>;
```

の代わりに次のように書くことができます。

```
for all x,y,z let abc(x,y,z) = <関数本体>;
```

この二つの定義方法ではいくつかの違いがあります。

もし関数本体中に仮引数への代入文があると、例えば、

```
x := 123;
```

では、PROCEDURE による定義では、関数本体へは実引数の値がコピーされて渡され、関数本体で値を変更しても、実引数の値は変わりません。

それに対して LET 文を使った場合では、実引数の値が変更されます。従って、ABC が LET 文を使って定義されていると、abc(u,v,w) を計算した後では、U の値は 123 になってしまいます。つ

まり、通常の PROCEDURE 文による定義では、関数への引数は値渡し (call by value) になっているのに対して、LET 文を使った定義では番地渡し (call by address) になっています。

例: 階乗を計算する関数 FACTORIAL を関数として定義すると次のようになります。

```
procedure factorial n;
   begin scalar m;
   m:=1;
l1: if n = 0 then return m;
   m := m*s;
   n := n - 1;
   go to l1
 end;
```

これを LET 文で定義してみると、次のようになります。

```
for all n let factorial n =
            begin scalar m,s;
            m:=1; s:=n;
        l1: if s=0 then return m;
            m:=m*s;
            s:=s-1;
            go to l1
        end;
```

ここで、新しい変数 S を宣言して、最初に N の値を S に代入していることに注意してください。PROCEDURE を使って定義した中には n := n - 1; という代入文がありました。これをそのまま LET 文で書き直したとき、もし factorial(5) を求めようとすると、N は 5 を値とする変数ではなく数値の 5 そのものを表し、REDUCE は $5 := 5 - 1$ という代入を実行しようとしてエラーを起こします。

もし PQR が引数の無い関数とすると

```
procedure pqr;
   <関数本体>;
```

となりますが、LET 文を使うともっと簡単に、

```
let pqr = <関数本体>;
```

で定義できます。この後の方法で定義した**関数**PQR を使うのには単に PQR とするだけでよく、通常の関数の様に PQR() と括弧をつける必要はありません。

この二つの方法を組み合わせて、PQR は通常の PROCEDURE 文を使って定義しておき、LET 文で、

```
let pqr = pqr();
```

とすることで、pqr() の代わりに pqr で関数が呼び出せるようにできます。

LET 文で定義する方法では、普通に PROCEDURE 文で定義した関数ではできない、不完全な関数というのが定義できます。例えば、

```
for all x such that numberp x let uvw(x)=<関数本体>;
```

とすれば、関数 UVW は引数が数値の場合には<関数本体> で計算した値が返されますが、それ以外の引数、例えば z や p+q、を与えると、入力したそのままの形で uvw(z) や uvw(p+q) が返されます。

14.5 REMEMBER 文

次のように

```
REMEMBER (PROCNAME:関数名);
```

関数に対して REMEMBER オプションを指定することにより、関数計算中の中間結果を、再帰呼出によるものを含めて全て、保存することができます。引き続いて行なわれる、同じ関数に対する計算においては、以前に保存された値が使われ、これにより計算回数 (あるいは計算の複雑さ) を減少させることができます。もちろん、この評価モードは余分な記憶容量を必要とします。さらに、このようにして評価される関数については副作用が無い関数でなければなりません。

次の二つの例は、remember 文が効果を発揮する良く知られた関数の例です。

```
procedure H(n);       % Hofstadter 関数
 if numberp n then
 << cnn := cnn +1;    % 呼出回数の計算
 if n < 3 then 1 else H(n-H(n-1))+H(n-H(n-2))>>;

remember h;

> << cnn := 0; H(100); cnn>>;

100

% H はちょうど 100 回だけ計算が行なわれる。

procedure A(m,n);     % Ackermann 関数

 if m=0 then n+1 else
  if n=0 then A(m-1,1) else
  A(m-1,A(m,n-1));

remember a;

A(3,3);
```

第15章　ユーザによるパッケージ

REDUCE システムには利用者によって作成された数多くのパッケージが、ユーザへのサービスとして含まれています。これらのパッケージに関する質問はそれぞれのパッケージの作者に直接行ってください。

これらのパッケージは、インストール時にコンパイルされています。しかし、その多くは使用するときにはパッケージのロードを行わなくてはなりません。(いくつかのパッケージについては実行時に自動的にロードされます。そのようなパッケージについては以下の説明の所で明示してあります。) また、利用説明書を読んでロード操作が実際に必要かどうかについて書かれていないか調べてください。

パッケージをロードするには LOAD_PACKAGE コマンドを使います。このコマンドは引数として一つまたは複数のロードするパッケージ名のリストを取ります。例えば、

```
load_package algint;
```

しかしながら、このコマンドの構文はインプリメントによって違うかも知れません。

ほとんどのパッケージは別に説明書とテストファイルがついています。(ただし、以下の説明で説明書がついていないと記述してあるものについてはこのような説明書はついていません。) これらのファイルはソースファイルと共に REDUCE の配布に含まれています。それぞれのパッケージの使い方についてはそれらの文書を参照してください。

現在の版の REDUCE に含まれているパッケージは以下の通りです:

15.1　ALGINT: 平方根を含む式の積分

このパッケージは REDUCE の不定積分パッケージに追加して、被積分項に平方根を含む場合で、かつ積分が平方根を含む式で表せるようなクラスの不定積分を求めることができます。これは J.H.Davenport による論文 “On the Integration of Algebraic Functions”, LNCS 102, Springer Verlag, 1981 に記述されている算法をプログラムしたものです。もっと詳しいことはこの論文とプログラムのソースを参照してください。

ALGINT パッケージが LOAD_PACKAGE コマンドでロードされると、例えば次のような不定積分が求められるようになります。

```
int(sqrt(x+sqrt(x**2+1))/x,x);
```

もし、後になってこのパッケージの機能を使わずに計算を行いたい場合は ALGINT スイッチをオフにしてください。このスイッチはこのパッケージがロードされたときにオンになります。

通常の積分パッケージが用意しているスイッチ (例えば TRINT) 等はこのパッケージでもサポートしています。さらにスイッチ TRA オンであれば、代数関数の積分のトレースを出力します。

このパッケージにはこれ以上の文書はついていません。

作者: James H. Davenport.

15.2 APPLYSYM: 微分方程式の無限小対称性

このパッケージでは、APPLYSYM, QUASILINPDE と DETRAFO の各プログラムを提供しています。APPLYSYM は、微分方程式に無限小対称性を適応します。QUASILINPDE は、対称性の計算を行います。DETRAFO は、相似変数の計算を行います。

作者: Thomas Wolf.

15.3 ARNUM: 代数的数

このパッケージは REDUCE の計算において、多項式の係数として代数的数が使えるように拡張します。また代数的数を表すために中間式を導入したり、分解体を求めたり、代数的数体上での因数分解や最大公約数計算が行えます。

作者: Eberhard Schrüfer.

15.4 ASSIST: 色々な応用に便利な機能

ASSIST には REDUCE を使うときに有用な色々な種類のツールが多数含まれています。

作者: Hubert Caprasse.

15.5 AVECTOR: ベクトル代数とベクトル解析

このパッケージは REDUCE でベクトル代数とベクトル解析の計算がスカラー演算と同じ書き方でできるようにします。基本的な代数計算やベクトルの微分、積分計算は入っています。スカラー積やベクトル積、要素の操作、スカラー関数 (例えば cosine 関数) をベクトルに作用させる等の計算ができます。

このパッケージの説明書は LaTeX 形式で書かれています。

作者: David Harper.

15.6 BOOLEAN: ブール代数

このパッケージには、ブール代数の計算に必要な関数が定義されています。データは代数式と内挿演算子 **and**, **or**, **implies**, **equiv** と前置単項演算子 **not** からなります。**Boolean** は、これらの演算子から構成される式を簡単化します。また、同値性 (equivalence) や部分式 (subset property) 等のチェックを行います。

作者: Herbert Melenk.

15.7 CALI: 可換代数のパッケージ

このパッケージには、イデアルや束の Gröbner 代数に関係した可喚代数の計算のアルゴリズムを含んでいます。これは、syzygies の計算も出来るような Gröbner のアルゴリズムをインプリメントしています。このインプリメントでは、行列の列で表される生成子による自由束の部分束に対しても適応可能です。

作者: Hans-Gert Graebe

15.8 CAMAL: 天体力学での計算

これは、天体力学用の数式処理システム CAMAL のフーリエ変換のパッケージを REDUCE にインプリメントしたものです。

作者: John P. Fitch

15.9 CHANGEVR: 微分方程式の独立変数の変換

このパッケージは、微分方程式の独立変数を変換する機能を与えます。これは、基本的には鎖則の応用です。

作者: G. Üçoluk

15.10 COMPACT: 式の最適化

COMPACT は制約条件の元で、式を最適化します。COMPACT は制約条件を与えられた式に適用して、最小の項数で表される式に変換します。例えば、次の式は、

```
compact(s*(1-sin x^2)+c*(1-cos x^2)+sin x^2+cos x^2,
        {cos x^2+sin x^2=1});
```

次のような結果を返します

```
        2           2
SIN(X) *C + COS(X) *S + 1
```

作者: Anthony C. Hearn.

15.11 CONTFR: 連分数展開

このパッケージは、(分母の上限を指定することにより) ユーザが指定した精度で実数の連分数による近似および有理数による近似を求めます。

このパッケージを使うためには、**misc** パッケージをロードしておく必要があります。

演算子 `continued_fraction(r, <size>)` で実数の連分数近似を求めることが出来ます。この演算子は近似する実数と精度 (これはオプションで指定しなくともよい) の一つ又は二つの引数を取ります。結果は有理数近似と同じ値を連分数展開の項からなるリストで表したもの (`t0 +1/(t1 + 1/(t2 + ...))` の形式で) の二つの要素をリストの形で返します。第二引数⟨*size*⟩は結果の有理数の分母の上限を決めます。もし省略された場合には近似は現在設定されている精度まで行われます。例えば:

```
continued_fraction pi;   ->

                1146408
              {---------,{3,7,15,1,292,1,1,1,2,1}}
                364913

continued_fraction(pi,100);      ->

                22
              {----,{3,7}}
                7
```

このパッケージにはこれ以上の説明書はついていません。

作者: Herbert Melenk.

15.12 CRACK: 偏微分または常微分の過剰決定方程式の解

CRACK は偏微分または常微分方程式の過剰決定系を解きます。CRACK を使った例題のプログラム (常微分/偏微分方程式の対称性の計算、第一積分、等価なラグランジェ形式や常微分方程式の素因子分解) が付いています。APPLYSYM パッケージを使うことによって、対称性の計算に応用することも可能です。

作者: Andreas Brand, Thomas Wolf

15.13 CVIT: ディラックのガンマ行列のトレース計算

このパッケージは、ディラックのガンマ行列を計算する別の方法として、Cvitanovich によるガンマ行列を 3-j シンボルとして扱うやり方で計算します。

作者: V.Ilyin, A.Kryukov, A.Rodionov, A.Taranov

15.14 DEFINT: 定積分

与えられた区間での定積分を求めるパッケージです。このパッケージでは、Meijer の G 関数や区間積分と言った革新的な方法等、いくつかのテクニックを使っています。

作者: Kerry Gaskell, Stanley M. Kameny, Winfried Neun.

15.15 DESIR: 確定および不確定特異点の近傍での線形同次微分方程式の解

線形同次微分方程式の形式解を、ゼロ（確定特異点もしくは不確定特異点あるいは正則点）近傍での有理数係数体上で指定された次数まで求めます。

このパッケージの説明はテキスト形式で書かれています。

作者: C. Dicrescenzo, F. Richard-Jung, E. Tournier

15.16 DFPART: 一般関数の微分

形式関数の全微分または偏微分係数を求めます。このような計算は微分方程式やべき級数展開を扱う上で有用です。

作者: Herbert Melenk.

15.17 DUMMY: ダミー変数を使った式の正準型

ダミー変数を使った式の正準型を求めます。これによって、多項式の簡約化が完全な形で行えます。未知変数は可能な限り最小の条件を課せられた関数として表されます。このため、このパッケージは広い応用範囲を持っています。

作者: Alain Dresse.

15.18 EXCALC: 微分幾何

EXCALC は最近の微分幾何学の計算に詳しい人のために設計されています。今のところでは、スカラー値を取る外形式、ベクトルとそれらの間での演算や添字のついたスカラーでない値の形式の計算が可能です。微分方程式の研究や一般相対論や関連する分野の研究、任意に与えられたフレーム上でのテンソルのラプラシァンの計算等に役に立ちます。

作者: Eberhard Schrüfer.

15.19 FIDE: 偏微分方程式の有限要素法

偏微分方程式を有限要素法で数値的に解く場合の処理を自動化します。偏微分方程式を数値的に解く場合に有限要素法がよく用いられますが、このような計算は、特に複雑なシステムの場合、非常に面倒な処理が必要になります。このパッケージは REDUCE による数式処理と FORTRAN による数値計算を結びつけて、偏微分方程式を解くためのプログラム作成を自動的に行えるようにします。

このパッケージの説明はテキスト形式で書かれています。

作者: Richard Liska

15.20 FPS: 形式べき級数の計算

与えられた関数を Laurant-Puiseux 級数に展開します。

作者: Wolfram Koepf and Winfried Neun.

15.21 GENTRAN: コード生成

GENTRAN は自動的なコード変換と生成を行います。これは REDUCE のプログラムを数値計算プログラムに変換します。対話的なコマンドとテンプレートを使うことによって、FORTRAN, RATFOR, PASCAL や C 言語の完全なプログラムを生成します。長大な式は自動的に部分式に分割されます。そして、ファイル操作コマンドは入出力チャンネルのスタックを扱うことにより、生成したコードを複数のファイルに出力することを可能にし、またコード生成を再帰的に行うことを可能にします。

作者: Barbara L. Gates.

15.22 GNUPLOT: 関数や曲面の表示

このパッケージは、よく知られた GNUPLOT パッケージとのインターフェースを与えます。2 次元や 3 次元での曲面を色々な出力装置に出力することが出来ます。これには、X 端末、パソコ

ンの画面、ポストスクリプトや LaTeX プリンター用のファイルに出力することが出来ます:

```
plot(z=x*sin x,x=(0 .. 10));
plot(1/(x**2+y**2),x=(-0.5 .. 0.5),y=(-0.5 .. 0.5),hidden3d);
```

注意:GNUPLOT パッケージは REDUCE の全ての版でサポートされているわけではありません。

作者: Herbert Melenk.

15.23 GROEBNER: Gröbner 基底

GROEBNER は、Buchberger の算法や多項式イデアルや束に対する方法によって Gröbner 基底の計算を行うパッケージです。これは様々な係数体上で、また違った変数や項の順序の下で計算可能です。Gröbner 基底は可換代数での様々な目的、例えば変数の消去、無理式から多項式形式への変換、次元の計算や方程式の解法等に利用できます。このパッケージの機能は、SOLVE 演算子が内部での計算に使用しています。

作者: Herbert Melenk, H.M. Möller and Winfried Neun.

15.24 IDEALS: 多項式イデアルの演算

Gröbner パッケージを使って多項式イデアルの基本的な演算をインプリメントしています。計算時間を節約するために、途中での Gröbner 基底の計算結果は内部で保存され、再計算による計算時間の浪費を防いでいます。

作者: Herbert Melenk

15.25 INEQ: 不等式の解法

演算子 **ineq_solve** は不等式、もしくは複数の不等式からなる不等式系の解を求めます。

作者: Herbert Melenk.

15.26 INVBASE: 包含基底の計算

包含 (Involutive) 基底は多変数多項式に関する問題、例えば、多項式系の解を求めることや、多項式イデアルを解析するといった問題を扱うための新しい道具です。多項式イデアルの縮退基底とは、冗長な Gröbner 基底の特別な形に他なりません。縮退基底を構成することにより、多項式系を解くことは単なる線形代数の問題に帰着されます。

作者: A.Yu. Zharkov and Yu.A. Blinkov.

15.27 LAPLACE: ラプラス及び逆ラプラス変換

ラプラス変換およびその逆変換を求めます。

このパッケージの説明はテキスト形式で書かれています。

作者: C. Kazasov, M. Spiridonova, V. Tomov

15.28 LIE: 実 n-次元 Lie 代数

LIE は実 n-次元 Lie 代数の分類を行うための関数を与えます。このパッケージは **liendmc1** と **lie1234** の二つのモジュールから構成されています。**liendmcl** モジュールの関数を使うことによって、次元 1 の導来代数 $L^{(1)}$ を持つ実 n-次元 Lie 代数 L の分類が計算できます。

作者: Carsten and Franziska Schöbel

15.29 LIMITS: 極限値

LIMITS は極限値を求めるパッケージです。これは Ian Cohen と John P. Fitch による仕事に基づいてプログラムされたもので、計算可能な極と特異点を除いて連続な関数の極限値を求めることができます。クリティカルな点 (その点の周りでは関数の値は、その点の周りで展開した級数の定数項になっている) での極限値の計算には TPS パッケージが使われています。クリティカルな場合にはロピタルの定理を使っています。制限された無限大数の演算も可能な限り使っています。

このパッケージでは LIMIT 演算子を定義しています。この演算子の構文は次の通りです。

```
LIMIT(EXPRN:代数式,VAR:カーネル,LIMPOINT:代数式):代数式
```

例えば、

```
limit(x*sin(1/x),x,infinity)   -> 1
limit(sin x/x^2,x,0)           -> INFINITY
```

方向に依存する極限演算子 LIMIT!+ と LIMIT!- も定義されています。

このパッケージは自動的にロードされます。

作者: Stanley L. Kameny.

15.30 LINALG: 線形代数

線形代数の計算で必要な関数を定義しています。

作者: Matt Rebbeck.

15.31 MODSR: 合同演算による方程式の解の計算

このパッケージは、合同演算による式の解を求める関数 (M_SOLVE) と多項式方程式系の解を求める関数 (M_ROOTS) を定義しています。法は素数でなくとも構いません。M_SOLVE を使うには、setmod コマンドであらかじめ法を設定しておく必要があります。M_ROOTS は第二引数に計算の法を指定します。例えば:

```
on modular; setmod 8;
m_solve(2x=4);             ->  {{X=2},{X=6}}
m_solve({x^2-y^3=3});
    ->  {{X=0,Y=5}, {X=2,Y=1}, {X=4,Y=5}, {X=6,Y=1}}
m_solve({x=2,x^2-y^3=3}); ->  {{X=2,Y=1}}
off modular;
m_roots(x^2-1,8);          ->  {1,3,5,7}
m_roots(x^3-x,7);          ->  {0,1,6}
```

このパッケージにはこれ以上の説明は有りません。

作者: Herbert Melenk.

15.32 NCPOLY: 非可換な多項式イデアル

Lie ブラケットで定義された非可換代数の演算を行います。このパッケージでは、REDUCE の **noncom** 機能を使って多項式の演算を行っています。交換規則は、Lie ブラケットから自動的に計算されます。

作者: Herbert Melenk and Joachim Apel.

15.33 NORMFORM: 行列の正準型

このパッケージでは、行列標準型を求める以下の関数から構成されています。

- smithex_int
- smithex
- frobenius
- ratjordan
- jordansymbolic
- jordan.

作者: Matt Rebbeck.

15.34 NUMERIC: 数値計算

このパッケージでは数値計算の基本的な算法をインプリメントしてあります。それらは:

- ニュートン法による代数方程式の解

```
num_solve({sin x=cos y, x + y = 1},{x=1,y=2})
```

- 常微分方程式の数値解

```
num_odesolve(df(y,x)=y,y=1,x=(0 .. 1), iterations=5)
```

- 区間内での関数値の最小、最大

```
bounds(sin x+x,x=(1 .. 2));
```

- 関数の最小値 (Fletcher Reeves の最沈降下法)

```
num_min(sin(x)+x/5, x);
```

- チェビシェフ近似

```
chebyshev_fit(sin x/x,x=(1 .. 3),5);
```

- 数値積分

```
num_int(sin x,x=(0 .. pi));
```

作者: Herbert Melenk.

15.35 ODESOLVE: 常微分方程式

ODESOLVE パッケージは常微分方程式の解を求めます。今のところは、きわめて制限された方程式しか解けません。代数式または方程式で表された単一のスカラー方程式、簡単な型の一階の微分方程式、定数係数の線形微分方程式系やオイラー型の微分方程式が扱えます。これらこのパッケージで解くことができる方程式の型は、Lie 対称性の技巧が何等有効な情報を与えてくれないような方程式になっています。例えば、次の式を解くと、

```
depend(y,x);
odesolve(df(y,x)=x**2+e**x,y,x);
```

このようになります。

```
       X                     3
   3*E  + 3*ARBCONST(1) + X
{Y=--------------------------}
                3
```

作者: Malcolm A.H. MacCallum.

その他寄与した者: Francis Wright, Alan Barnes.

15.36 ORTHOVEC: ベクトル演算

ORTHOVEC はスカラーとベクトルを操作するための REDUCE の関数の集まりです。演算は和、差、ドット積と外積、割り算、モジュラス、div, grad, curl , ラプラシャン、微分、積分やテイラー級数展開等ができます。

作者: James W. Eastwood.

15.37 PHYSOP: 量子力学でのオペレータ計算

量子力学における演算子を含む式の演算を行いたい理論物理学者の要求に応えるよう作成しました。主として、演算子を含む式の交換式の計算や、演算子を要素とする行列の演算を行います。

作者: Mathias Warns

15.38 PM: REDUCE のパターンマッチャー

PM は、SMP や Mathematica にあるパターンマッチャーの機能を実現しています。これは、Kevin McIsaac, “Pattern Matching Algebraic Identities”, SIGSAM Bulletin, 19 (1985), 4-13 によるマッチングのアルゴリズムに基づいています。

このパッケージの説明はテキスト形式で書かれています。

作者: Kevin McIsaac

15.39 RANDPOLY: ランダム多項式の生成

Maple のランダム多項式生成のプログラムに基づいて作成したもので、それに乱数生成と仮名の関数生成の機能をつけ加えています。

作者: Francis J. Wright.

15.40 REACTEQN: 化学反応方程式

化学反応式から質量反応の法則に対応する微分方程式を導出します。

このパッケージの説明はテキスト形式で書かれています。

作者: Herbert Melenk

15.41 RESET: REDUCE を初期状態にリセットする

RESETREDUCE コマンドは、REDUCE の実行の履歴や代入された変数の消去、ルールや配列の消去を行って、最初に起動した状態に戻します。これは完全に最初の状態に戻すのではありません。しかし、大部分の状態は初期の状態に戻します。特に代入された変数の消去を行うことで、メモリを不要に消費することがなくなります。この関数は比較的容易に対話的に処理するように変更することが可能です。そのように修正を加えることによって、ある選択した部分に対して消去を実行するように書き直すことが出来るでしょう。

このパッケージにはこれ以上の説明は付いていません。

作者: John Fitch.

15.42 RESIDUE: 留数

このパッケージでは、任意の式の留数の計算が出来ます。

作者: Wolfram Koepf.

15.43 RLFI: REDUCE から LaTeX 形式への変換

REDUCE の出力する数式を LaTeX 形式で出力するようにします。数式でよく使われる記号、上付きの添字、下付きの添字、フォントの選択、ギリシャ文字、割算記号、積分や総和記号、微分記号等が使えます。

作者: Richard Liska

15.44 ROOTS: 高次方程式の数値解

このパッケージは実数係数または複素数係数の一変数多項式の零点の一部または全部をユーザが指定した精度で求めます。

これは独立したパッケージとしても、また ROUNDED がオンの時、SOLVE 演算子から呼び出されるように設計されています。例えば、次の計算

```
on rounded,complex;
solve(x**3+x+5,x);
```

は以下の結果を返します。

```
{X= - 1.51598,X=0.75799 + 1.65035*I,X=0.75799 - 1.65035*I}
```

このパッケージは自動的にロードされます。

作者: Stanley L. Kameny.

15.45 RSOLVE: 有理/整式の解法

一変数の有理式の解を合同演算を使って高速に求めます。ここで使っているアルゴリズムは、R. Loos (1983): Computing rational zeros of integral polynomials by p-adic expansion, *SIAM J. Computing*, **12**, 286–293 に記述されています。

作者: Francis J. Wright.

15.46 SCOPE: REDUCE のソースコードの最適化

SCOPE は与えられた数式を最適化された式に変換します。これは REDUCE の Proper な代入文中の共通 (部分) 式をヒューリスティックな方法で探索し、結果を代入文の列で表して出力します。出力には GENTRAN パッケージが使われています。

このパッケージの説明は troff 形式で書かれています。

作者: J.A. van Hulzen.

15.47 SETS: 集合演算

リストを集合と見なした、もしくは識別子で表された集合に対して、代数モードでの集合演算を行います。

作者: Francis J. Wright.

15.48 SPDE: 偏微分方程式の対称群

SPDE パッケージは与えられた偏微分方程式系に対して、Lie 対称、もしくは点対称による対称群を決定するのに使われる関数の集合を決定します。多くの場合、決定方程式は自動的に解かれます。自動的に解けない場合には、ユーザは解を求めるために必要な情報を与える必要があります。
作者: Fritz Schwarz.

15.49 SPECFN: 特殊関数

この特殊関数のパッケージは、取り扱いが容易なように SPECFN と SPECFN2 の二つの部分に分かれています。最初のは一般的な関数で、二番目のは特殊な関数を扱っています。説明は “doc” ディレクトリ中のファイル specfn.tex にかかれています。また、例題が “examples” ディレクトリ中に specfn.tst と specfmor.tst という名前で入っています。

SPECFN パッケージはいくつかのよく使われる特殊関数の代数操作や数値計算を行えるようにしています。これには以下の関数が含まれています。

- ベルヌーイ数とオイラー数;
- スターリング数;
- 二項係数;
- Pochhammer の記号;
- Gamma 関数;
- Psi 関数とその微分;
- リーマンの Zeta 関数;
- ベッセル関数 J と 一種および二種の Y 関数;
- 変形されたベッセル関数 I と K;
- ハンケル関数 H1 と H2;
- クマーの超幾何関数 M と U;
- Beta 関数、Struve, Lommel および Whittaker 関数;
- Airy 関数;
- 指数積分、正弦および余弦積分;
- 双曲正弦および双曲余弦積分;
- フレネル積分と誤差関数;
- Dilog 関数;
- エルミート多項式;
- ヤコビ多項式;
- ルジャンドル多項式;
- 球と Solid 調和関数;
- ラゲール多項式;
- チェビシェフ多項式;
- ゲーゲンバウアー多項式;
- オイラー多項式;
- ベルヌーイ多項式.
- Jacobi の楕円関数と積分;
- 3j シンボル、6j シンボルおよびクレブッシュゴルドン係数;

作者: Chris Cannam それに Winfried Neun, Herbert Melenk, Victor Adamchik, Francis Wright 達の寄与による

15.50 SPECFN2: 特別な特殊関数

このパッケージは一般化された超幾何関数と Meijer の G 関数に関する代数的操作を与えます。“doc” ディレクトリに meijerg.tex と ghyper.tex という名前で説明書が入っています。一般化された超幾何関数は特殊関数で、また Meijer の G 関数は特殊関数または一般化された超幾何関数に簡約します。

テストファイルが “xmpl” ディレクトリに meijerg.tst と ghyper.tst という名前で入れてあります。
作者: Victor Adamchik それに主として Winfried Neun による改訂.

15.51 SUM: 級数の和

このパッケージは Gosper による級数の和を求める算法をプログラムしてあります。このパッケージでは演算子 SUM と PROD が定義してあります。SUM は与えられた式の不定和分を、また PROD は式の積を求めます。

このパッケージは自動的にロードされます。

このパッケージの説明書は LaTeX 形式で書かれています。

作者: Fujio Kako.

15.52 SYMMETRY: 対称行列の演算

群の対称を持つ行列の対称基底とブロック対角型を求めます。このパッケージは dihedral 群のような小さな有限群の線形表現の理論を使っています。

作者: Karin Gatermann

15.53 TAYLOR: テイラー級数

このパッケージは一変数もしくは多変数のテイラー級数展開を求めます。またテイラー級数の効率的な演算 (和、差、積および商) や代数演算や超越関数への応用もできます。

作者: Rainer Schöpf.

15.54 TPS: べき級数

このパッケージは一変数に関する形式的ローラン級数展開を扱います。級数は REDUCE のドメインを使ってインプリメントされています。従って REDUCE の他の要素と同じように級数の和、積、微分等の計算が可能です。遅延評価機構を使っており、級数の各項が出力やその他のべき級数の項の値を計算するのに値が使われるなど、本当にその値が必要になったとき初めて計算

が行われるようになっています。級数は無限級数として処理されており、有限項で打ち切ったときに起こるエラー (例えば結果の式で必要な項の計算に前の計算で打ち切られた項の値を必要とする等) は、このパッケージを使えば起こりません。

作者: Alan Barnes and Julian Padget.

15.55 TRI: REDUCE から TeX 形式への変換

REDUCE の Lisp でかかれたパッケージで、REDUCE の数式を TEX 形式で出力します。三つの出力形式が可能です。一つは改行なしでの出力で、もう一つは改行を付けた出力、最後の一つは改行と段落を付けた出力です。

作者: Werner Antweiler

15.56 TRIGSIMP: 三角関数および双曲線関数の簡約と因子分解

三角関数や双曲線関数を含む式の簡約や因子分解を行います。これには、trigsimp , trigfactorize と triggcd の三つの有用な道具が用意されています。最初の関数は簡約を行います。二番目の関数は因子分解を行います。また、三番目の関数は二つの三角関数や双曲線関数を含む式の最大公約因子を求めます。

作者: Wolfram Koepf.

15.57 WU: 多項式方程式系に対する Wu の算法

Wu Wen-tsun, Institude of Systems Science, Academia Sinica, Beijing による “A Zero Structure Theorem for Polynomial-Equations-Solving” にある Wu のアルゴリズムのインプリメントです。

作者: Russell Bradford

15.58 XCOLOR: 場の理論における色因子

非アーベルゲージ場理論における色因子の計算を Cvitanovich によるアルゴリズムによって行います。

このパッケージの説明はふつうのテキスト形式です。

作者: A. Kryukov.

15.59 XIDEAL: 外積代数における Gröbner 基底の計算

XIDEAL は左イデアルのメンバーシップ問題を解くことによって Gröbner 基底 (Gröbner 左イデアル基底もしくは GLIB) を求めます。Graded イデアル (各式が次数に関して同次である)、に対しては左イデアルと右イデアルの違いはなくなります。さらに、もし生成式がすべて同次で有れば non-graded と graded Gröbner 基底は同じものになります。このような場合、Gröbner 基底の計算を、ある最大次数で打ち切ることによって計算時間を節約することが出来ます。

作者: David Hartley.

15.60 ZEILBERG: 不定和分および定和分

このパッケージでは、Gosper と Zeilberger による数列の和を求めるアルゴリズムをインプリメントしています。彼らの超幾何級数に対するアルゴリズムは、指数関数の積、階乗、ガンマ関数、二項係数やシフトした階乗の比で表される様な級数に対しても求められるように拡張されています。

作者: Gregor Stölting and Wolfram Koepf.

15.61 ZTRANS: Z-変換

数列の Z-変換をインプリメントしています。これは、離散化したラプラス変換に対応する変換です。

作者: Wolfram Koepf and Lisa Temme.

第16章　記号モード

システムのレベルではREDUCEは*Standard Lisp*と呼ばれるLisp言語で書かれています。この言語はJ. Marti, Hearn, A. C., Griss, M. L. and Griss, C., “Standard LISP Report” SIGPLAN Notices, ACM, New York, 14, No 10 (1979) 48-68に記述されています。ここでは読者はこの文献については詳しいものと仮定しています。またLispについてもある程度知っているものとします。例えば、Lisp 1.5のプログラマーズマニュアル (McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I., “LISP 1.5 Programmer’s Manual”, M.I.T. Press, 1965) もしくはこの章の最後にあげる本を参照してください。あまりこれらの文献に詳しくない人にとって、この章の内容は少し理解しにくいかも知れません。

REDUCEは代数計算を行うためで設計されましたが、そのソース言語はLispに似た記号計算を行うのに必要なだけの一般性を持っている。この一般性を実現するために、代数モードと記号モードの二つのモードを提供している。記号モードに入るには`symbolic;`(または`lisp;`) と打ち込めばよい。また代数モードに戻るには`algebraic;`と打ち込めばよい。二つのモードで評価の行われ方が異なっているので、ユーザはもし奇妙なエラーが起こったときにはどのモードになっているかを調べた方がよいでしょう。モードは次のように入力することで知ることができます。

```
eval_mode;
```

現在のモードが`ALGEBRAIC`もしくは`SYMBOLIC`と出力されます。

式の値をどちらのモードでも扱えるような互換性のある方法で受け渡せば、計算をどちらのモードでも、また計算のどのレベルでも進めることができます。これには、対象となる式の前にモードを指定する`ALGEBRAIC`もしくは`SYMBOLIC`をつけるだけでよい。もしトップレベルでの式にモード名を指定すると、その式を計算するときに、一時的にモードが変更され、評価が行われます。

例えば、現在のモードが`ALGEBRAIC`(代数モード) のとき

```
symbolic car '(a);
x+y;
```

は、最初の式は記号モードで評価され、二番目の式は代数モードで行われる。二番目に行った結果のみが式のワークスペースを変更する。また、

```
x + symbolic car '(12);
```

は代数式`X+12`を返します。

`SYMBOLIC` (そして同じように`ALGEBRAIC`) の使い方は他の演算子と同じです。つまり、上の例では意味が明らかなので括弧が省略されています。しかし、使い方によっては次のように括弧が必要です。

```
symbolic(x := 'a);
```

括弧を省略して、

```
symbolic x := 'a;
```

とすると、これは

```
symbolic(x) := 'a;
```

と解釈されるので、正しくありません。利用者にとって便利なように、**最初**の引数が quote(式の前に記号' が付いているものは Lisp では評価せずそのままの式を表す) されている演算子はすべて現在のモードに関係なく、記号モードで評価されます。従って、最初の例は次のように書いても構いません。

```
car '(a);
```

はっきり断わっていない限り、ほとんどの REDUCE の代数モードでの構文は記号モードでも使えます。しかし、いくつかの違いがあります。まず、式の評価は Lisp の式として行われます。二番目には、代入文は違った扱いを受けます。これについてはあとで少しふれます。三番目には、ローカル変数と配列は 0 ではなく NIL に初期設定されます。(実際は、代数モードでも INTEGER と宣言された変数以外は NIL に初期設定されます。しかし、代数の評価関数は NIL を 0 として扱います。) 最後に、関数の定義は Standard Lisp の規則に従います。

最初に、基本的な構文の拡張に付いて説明しておきます。これは記号モードで使用するために用意されています。

16.1 記号モードでの内挿演算子

記号モードで使える 3 つの二項の内挿演算子があります。これらは . (CONS), EQ と MEMQ です。これらの演算子の優先度は別の節で説明してあります。

16.2 記号モードでの式

これはスカラー変数と演算子より構成され、Lisp のメタ言語の通常の規則に従います。

例:

```
x
car u . reverse v
simp (u+v^2)
```

16.3 QUOTE された式

記号モードでの評価では、全ての変数は値を持っていることを要求されます。従って REDUCE では Lisp 言語の QUOTE 関数と同じ様な、QUOTE された式というものを導入する必要があります。これは、クウォート記号' を使って表されています。例えば、

```
'a        は Lisp の S-式 (quote a) を表す
'(a b c) は Lisp の S-式 (quote (a b c)) を表す。
```

しかし、文字列は定数で、記号モードではそれ自身が値となっています。従って、文字列"A String" を出力するためには、次のように書けばよい。

```
prin2 "A String";
```

QUOTE された式中では、識別子の構文規則は REDUCE と同じです。(A !. B) は三つの要素 A,. と B からなるリストですが、(A . B) は A と B のドット対になります。

16.4 ラムダ式

ラムダ (LAMBDA) 式は記号モードで Lisp の LAMBDA 式を構成するのに用います。これは代数モードでは使いません。

構文は次の通りです。

```
<LAMBDA 式> ::=
        LAMBDA <変数リスト><終端記号><文>
```

ここで

```
<変数リスト> ::= (<変数>,...,<変数>)
```

例えば、

```
lambda (x,y); car x . cdr y;
```

は Lisp の LABMDA 式

```
(lambda (x y) (cons (car x) (cdr y)))
```

と同じです。変数リストを囲む括弧は、もし省略した方が見やすければ、省略しても構いません。

LAMBDA 式は記号モードで前置演算子が使われる所や、予約語 FUNCTION の引数に使うことができます。

LAMBDA 式が同じ式を計算することを避けるために、局所変数を導入する目的で使われるときには、代わりに WHERE 文を使うことができます。例えば、次の式

```
(lambda (x,y); list(car x,cdr x,car y,cdr y))
    (reverse u,reverse v)
```

は、次のように書くこともできます。

```
{car x,cdr x,car y,cdr y} where x=reverse u,y=reverse v
```

LAMBDA 構文よりも WHERE 構文の方が自然なので、できるだけこれを使ってください。

16.5 記号モードでの代入文

記号モードでは、代入文の左辺が変数名であれば右辺の値が関数 SETQ よって代入されます。もし左辺が式の場合、それは配列の要素でなければいけません。さもなければエラーになります。例えば x:=y は (SETQ X Y) と変換されます。また a(3) := 3 は A が少なくともサイズが 4 以上の一次元配列として宣言されていなければエラーになります。

16.6 FOR EACH 文

FOR EACH 文は FOR 文の一種で、与えられたリストの要素に関する繰り返しを記述します。この構文は次の通りです。

```
FOR EACH ID:識別子 {IN|ON} LST:リスト
    {DO|COLLECT|JOIN|PRODUCT|SUM} EXPRN:S 式
```

代数モードで使われた場合、キーワード IN が使われたときにはリストの各要素について、また ON が使われた場合は与えられたリストから先頭の要素を順次取り除いていったリストについて繰り返しが行われます。記号モードでは、それぞれの FOR EACH 文は以下に示す Lisp のマップ関数に変換されます。

	DO	COLLECT	JOIN
IN	MAPC	MAPCAR	MAPCAN
ON	MAP	MAPLIST	MAPCON

例: 与えられたリスト (a b c) の各要素をリストに変換したものを要素とするリストを作るには次のようにします。

```
for each x in '(a b c) collect list x;
```

16.7 記号モードでの関数

Standard Lisp レポートに記述されている関数はすべて記号モードで使うことができます。新しく記号モードでの関数を定義することもできます。例えば、Lisp の関数 ASSOC は次のようにして定義することができます。

```
symbolic procedure assoc(u,v);
   if null v then nil
    else if u = caar v then car v
    else assoc(u, cdr v);
```

現在のモードが記号モードであれば、上の定義で先頭の SYMBOLIC は省略できます。MACRO 型 の関数を定義するには、キーワード PROCEDURE の前に MACRO という関数の型宣言を付ければよい。(通常の関数は EXPR 型の関数として定義されます。 これは Standard Lisp リポートにあるように PROCEDURE の前に EXPR という関数の型宣言をつけて定義することもできます。) 例えば、MACRO CONSCONS は次のようにして定義することができます。

```
symbolic macro procedure conscons l;
   expand(cdr l,'cons);
```

SMACRO という型の別のマクロ関数を定義することもできます。これについては Standard Lisp リポートに説明してあります。Standard Lisp リポートには FEXPR という関数型も定義されています。 しかし、この型の関数を効率よく実装するのはむずかしく、しかも多くの場合マクロで置き換えることが可能です。現在のところ、REDUCE では FEXPR 型の関数は使われていません。

16.8 REDUCE 入力と等価な S-式

REDUCE で入力したものと等価な S-式を得るには、DEFN (定義を意味する definition の略) スイッチをオンにすればよい。システムは入力した式を Lisp に変換した結果を表示します。しかし、入力した式は実行されません。通常の動作に戻すには DEFN をオフにします。

16.9 代数モードとの通信

REDUCE の代数計算の機能を使うユーザが記号モードについて知ることで、一つには代数モードだけではできないようなもっと広い範囲のテクニックが使えるようになります。例えば、基本システムのソースコードで定義されている関数の一部を使うとか、プログラムをもっと効率的にするために代数モードで書いたプログラムを改良したい場合には、REDUCE のソースが書かれているプログラミング言語について詳しく知っている必要があります。さらに REDUCE が内部でどのように処理を行うかについて知っておく必要があります。基本的に、REDUCE は数式を二つの形式で扱います。一つは前置形式 (Prefix form) で、これは通常の Lisp 形式で数式を表したものです。もう一つは正準形式 (Canonical form) と呼ばれるもので、まったく異なった構文を使います。

これらの詳細を理解すれば、ユーザが出くわす最も微妙な問題は、記号モードと代数モードで数式をやり取りする為には、どのように数式を作れば良いのか、またどのように関数を定義すれば良いのかということです。この節の目的は、このことについての基本的な考え方を説明することです。

数式を代数モードで計算し、その結果を記号モードでの計算に使いたい場合、またはその反対のことを行いたい場合、一番簡単な方法はその数式を変数に代入し、その変数を通して両モード

間で式をやり取りすることです。これを行う為に SHARE 宣言があります。SHARE は識別子のリストを引数に取り、それらの変数の値が両方のモードで使えるように印をつけます。この宣言はどちらのモードで行っても構いません。

例えば、

```
share x,y;
```

は変数 X と Y に対して、その値が両方のモードで参照できるようにします。

もしすでに代数モードで変数としてなんらかの値が代入されている変数名に SHARE 宣言を行うと、その変数に代入されている値は記号モードでも参照できるようになります。

16.9.1 代数モードの値を記号モードに渡す方法

もし代数モードでの式の一部を記号モードで計算したい場合、単に、その式を SHARE 宣言された変数に代数モードで代入すればよい。例えば、(a+b)^2 という数式を記号モードで操作したい時には代数モードで次のようにすればよい。

```
x := (a+b)^2;
```

ただし、ここで変数 X は SHARE 変数として宣言されているものとする。続いて、記号モードに移り、次のように入力すると、

```
x;
```

その値が前置形式で次のように出力されます。

```
(*SQ <標準有理式> T)
```

この特別な形式は、代数モードでの計算ではほとんどの場合、関数から関数へ前置形式で値を渡すように設計されています。しかし、関数内部での計算で使われる標準形式から本当の前置形式へ、また前置形式から標準形式への変換を毎回行うのは無駄であり、計算時間をよけいに必要とするので、できる限りこのような変換を行わないで済ませるようにしている。*SQ は代数モードでの演算において、この式は前置形式として扱っているが、実際は標準有理式 (Standard Quotient) になっていることを表すために使っています。第二引数 (T または NIL) は処理するときに再度評価する必要があるかどうかを示している。(これはすでに評価済みであるかどうかのフラグである。)

従って、記号モードで標準有理式を取り出すには変数の値の CADR を取る必要がある。つまり、

```
z := cadr x;
```

とすれば、変数 Z に数式 (a+b)^2 の標準有理式が入ります。

一度この式を取り出したら、後は好きなように操作することができます。このような式から必要な部分を取り出す選択子 (Selector) やこれらの部分から式を構成する構成子 (Constructor) が用意されています。現在定義されているこのような演算子は次のものです。

REDUCE の選択子

DENR	標準有理式の分子
LC	多項式の主係数
LDEG	多項式の主項の次数
LPOW	多項式の主べき
LT	多項式の主項
MVAR	多項式の主変数
NUMR	標準有理式の分母
PDEG	べきの次数
RED	多項式から主項を除いた式 (reductum)
TC	項の係数
TDEG	項の次数
TPOW	項のべき

REDUCE の構成子

.+	多項式と項の和
./	二つの多項式の商
.*	べきと係数の積から項を作る
.^	変数をべきにする

訳注: 標準有理式等の定義は次のようになっています。

標準有理数	:=	(分子 . 分母)
分子	:=	標準多項式
分母	:=	標準多項式
標準多項式	:=	NIL(零)| 数値 | (標準項 . 標準多項式)
標準項	:=	(標準べき . 係数)
係数	:=	標準多項式
標準べき	:=	(カーネル . 指数)
指数	:=	正の整数

例えば、上の例で標準有理式の分母を取り出すには、次の様に入力すればよい。

```
numr z;
```

また、分母の主項を取り出すには次のようにすればよい。

```
lt numr z;
```

これらのデータ構造間の変換を行う関数も定義されています。現在用意されているのは次のような関数です:

!*A2F	代数式を標準形式に変換する。結果が有理式の時にはエラーになる;
!*A2K	代数式をカーネルに変換する。もしカーネルでなければエラー;
!*F2A	標準形式を代数式に変換する;
!*F2Q	標準多項式を標準有理式に変換する;
!*K2F	カーネルを標準多項式に変換する;
!*K2Q	カーネルを標準有理式に変換する;
!*P2F	標準べきを標準多項式に変換する;
!*P2Q	標準べきを標準有理式に変換する;
!*Q2F	標準有理式を標準多項式に変換する。もし有理式の分母が 1 でなければエラーになる;
!*Q2K	標準有理式をカーネルに変換する。もし結果がカーネルでなければエラーになる;
!*T2F	標準項を標準多項式に変換する;
!*T2Q	標準項を標準有理式に変換する。

16.9.2 記号モードの値を代数モードに渡す方法

SHARE 宣言された変数を使って記号モードから代数モードへ値を渡す場合、注意するべきことはその値を前置形式で渡さなければならないということだけです。例えば、(a+b)^2 を渡すときには (expt (plus a b) 2) とするか、それとも上で説明したように (*sq <標準有理式> t) の様にするかいずれかの形式で渡します。もし、標準多項式の一部分を扱っているときには、このような形式にはなっていません。この場合には次のようにします。

1. もし、標準有理式であれば、関数 PREPSQ を呼びます。この関数は標準有理式から前置形式に変換します。もしくは、関数 MK!*SQ を呼び出すこともできます。これは引数を (*SQ <標準有理式> T) の形式に変換します。これによって、数式を本当の前置形式に変換することを避けることができます。

2. もし標準多項式であれば、関数 PREPF を呼びます。これは標準多項式を前置形式に変換します。または標準多項式をいったん標準有理式に変換して、関数 MK!*SQ を呼び出して渡すこともできます。

3. もし標準多項式の一部分の場合にはこれから標準多項式を構成し、それからステップ 2 を行います。前に説明した変換関数がこの目的に使うことができます。例えば、

   (a) Z が項であれば、 !*T2F Z は標準多項式になります。
   (b) Z が標準べきであれば、!*P2F Z は標準多項式です。
   (c) Z が変数であれば、直接これを渡すことができます。

例えば、数式 (a+b)^2 の主項を代数モードに渡したければ、次のようにすれば良い。

```
y:= mk!*sq !*t2q lt numr z;
```

ここで、変数 Y は SHARE 宣言されているものとする。これで代数モードに戻れば、変数 Y には主項が入っており、これを通常の代数モードの計算で使うことができます。

16.9.3 完全なプログラムの例

次は上記の例の計算を行う完全なプログラムです。計算の最後では得られた $(a+b)^2$ の主項の自乗を求めています。

```
share x,y;                  % 変数 X と Y を{\tt SHARE}と宣言する
x := (a+b)^2;               % (a+b)^を X に保存する
symbolic;                   % 記号モードに移る
z := cadr x;                % Z に標準有理式を代入する
lt numr z;                  % Z の分母の主項を表示する
y := mk!*sq !*t2q numr z; % Y に主項を前置形式で代入する
algebraic;                  % 代数モードに戻る
y^2;                        % (a+b)^2 の主項の自乗を計算する
```

16.9.4 両モード間のやり取りを行う関数

記号モードで、代数モードのデータを処理する関数を定義するには、その関数を記号モードで OPERATOR として宣言する必要があります。つまり、

```
symbolic operator leadterm;
```

とすれば、手続き LEADTERM を代数モードでの演算子として宣言します。代数モードで OPERATOR として宣言すること別の意味になるので、ここでは記号モードで宣言しなくてはなりません。このように宣言された手続きの値は前置形式でなくてはいけません。

代数モードでの評価ルーチンはこのような関数へは前置形式で渡します。従って、引数で渡された式を標準有理式として扱いたい場合には、最初に関数 SIMP!*を使って変換しておく必要があります。この関数は前置形式からその式を評価した結果を標準有理式に変換して返します。

例えば、与えられた数式の主項を取り出す関数 LEADTERM は次のように定義されます。

```
symbolic operator leadterm; % LEADTERM を記号モードでの関数で
                            % あり、代数モードから参照できる
                            % ものとして宣言します。

symbolic procedure leadterm u; % LEADTERM を定義する.
   mk!*sq !*t2q lt numr simp!* u;
```

この関数は LTERM 演算子とは異なった結果を返します。ここで定義した関数はシステムの選んだ主変数に関する主項を取り出すのに対して、LTERM は第二引数で指定した変数に関する主項を返します。

最後に、もし記号モードで代数モードの評価ルーチンを使いたければ、関数 REVAL を使うことができます。これは前置形式で表した式を評価します。結果は Lisp の真の前置形式で返されます。

16.10 Rlisp '88

Rlisp '88 は REDUCE を記述するのに使われている Rlisp 言語の拡張版です。この言語については Marti, J.B., "RLISP '88: An Evolutionary Approach to Program Design and Reuse", World Scientific, Singapore (1993) に詳しく説明されています。Rlisp '88 はこれまでの Rlisp に比べて以下の機能が追加されています:

1. ループを構成する `for`、`repeat` や `while` の構文が拡張されています。
2. バッククオートの機能が追加されています。
3. アクティブな注釈が使えます。
4. 名前 [添え字] の形式のベクトルが使えます。
5. 単純構造 (simple structure) が使えます。
6. レコード型が使えます。

さらに、Rlisp '88 では "–" も識別子を構成する文字として使われます。つまり `A-B` は、Rlisp のように `A` 引く `B` を表すのではなく、名前となります。つまり、Rlisp88 では、`A` 引く `B` を表すには`-`の前後に空白をいれる必要があります。Rlisp との互換性を保つために、この表式は記号モードでのプログラムでのみ使用することとします。

Rlisp '88 を使うには、`on rlisp88;` と入力します。これは、これ以後の記号モードでのプログラムの解釈を Rlisp '88 の文法のもとで、そしてその拡張機能を使って、行えるようにします。この状態では、代数モードに切り替えることはできません。また、式の前に "algebraic" を指定することもできません。しかし Rlisp '88 で記述された記号モードでのプログラムは、rlisp88 パッケージをロードすることで実行することができます。また、Rlisp '88 は多くの機能が拡張されているため、いずれ Rlisp のプログラムは Rlisp '88 に移行していくものと期待しています。通常の Rlisp もしくは代数モードに戻るには `off rlisp88;` と入力します。

16.11 参考文献

LISP に関する本はたくさんあります。ここにはその内のいくつかを挙げておきます。

Allen, J.R., "The Anatomy of LISP", McGraw Hill, New York, 1978.

McCarthy J., P.W. Abrahams, J. Edwards, T.P. Hart and M.I. Levin, "LISP 1.5 Programmer's Manual", M.I.T. Press, 1965.

Touretzky, D.S, "LISP: A Gentle Introduction to Symbolic Computation", Harper & Row, New York, 1984.

Winston, P.H. and Horn, B.K.P., "LISP", Addison-Wesley, 1981.

第17章　高エネルギー物理学の計算

REDUCE は高エネルギー物理学の計算を行うためのコマンドを用意しています。基本的な構文を拡張して、違ったデータ構造を操作できるようにしています。

17.1　高エネルギー物理学の演算子

まずこのような計算で必要になる三つの新しい演算子を説明します。

17.1.1　. (Cons) 演算子

構文は次の通りです。

```
(EXPRN1:ベクトル式) . (EXPRN2:ベクトル式):代数式
```

二項演算子 . は通常リストの先頭に要素を追加する演算子として使われますが、同じ演算子を高エネルギー物理学のパッケージでは 4 次元ローレンツベクトルのスカラー積 (内積) として使います。このとき、二番目の引数はベクトル式でなければなりません。この意味で使われるとき、この演算子はしばしば “ドット” 演算子と呼ばれます。現在のシステムではインデックスを操作するルーチンは、ローレンツの 4 次元ベクトルが使われているものと仮定しています。しかし、それ以外の場合を扱えるようにルーチンを書き直すことは可能です。

ベクトルの要素は単位ベクトルを使って表せます。EO が単位ベクトル (1,0,0,0) を表しているとき、(p.eo) は 4 次元ベクトル P の第 0 要素を表します。ここで使っているメトリックや記法は Bjorken と Drell による “Relativistic Quantum Mechanics” (McGraw-Hill, New York, 1965) に合わせてあります。同じように P の任意要素は (p.u) と表されます。もしベクトルの要素に関する縮約 (contruction) が必要なら、INDEX 宣言を使います。例えば、

```
index u;
```

は U をインデックスとして宣言し、

```
p.u * q.u
```

を簡約して次のようにします。

```
P.Q
```

メトリックテンソル $g^{\mu\nu}$ は (u.v) と表されます。もし、U と V の縮約を計算する必要があればそれらをインデックスとして宣言すればよい。

もし式のインデックスが合っていなければエラーを起こします。

もし後になって特定のベクトルからインデックス宣言を取り消したければ REMIND 宣言を使います。remind v1...vn; は V1 から Vn までの変数からインデックスフラグを削除します。しかし、これらの変数に対するベクトルとしての宣言は取り消されません。

17.1.2 ガンマ行列に対する G 演算子

構文は次の通りです。

```
G(ID:識別子 [,EXPRN:ベクトル式])
        :ガンマ行列の式
```

G は n 項演算子でローレンツの四次元ベクトルと縮約された γ 行列の積を表します。γ 行列はファイマンダィアグラムでフェルミ粒子に対応しています。もし、一つ以上の粒子が関係しているときには、それぞれの粒子を表すために (独立なスピン空間に作用する) 異なった γ 行列が必要になります。このような機能をもたせるため、G の第一引数は粒子を表す (数値でない) 識別子とし、それぞれの粒子を区別するために用いられます。

従って、

```
g(l1,p) * g(l2,q)
```

は L1 で特定されるフェルミ粒子に対応した γ.p と別の L2 に対応する γ.q の積を表します。ここで、p や q はローレンツの四次元ベクトルです。同じ粒子に関する γ 行列の積は縮約された形式で書かれています。

従って、

```
g(l1,p1,p2,...,p3) = g(l1,p1)*g(l1,p2)*...*g(l1,p3) .
```

ベクトル A は演算子 G の引数に現れたときには特別な γ 行列である γ^5 を表すものとして予約されています。

g(l,a) = フェルミオン線 L に対応した行列 γ^5

g(l,p,a) = フェルミオン線 L に対応した行列 γ.p $\times\gamma^5$

(粒子 L に対応した) γ^μ は g(l,u) と書かれます。ここで、 U はもし U に関する縮約が必要な場合はインデックスとして宣言されているものとしています。

γ 行列を含む全ての演算に於いて、Bjorken と Drell の記法を使うものと仮定しています。

17.1.3 EPS 演算子

構文は、

```
EPS(EXPRN1:ベクトル式,...,EXPRN4:ベクトル式)
      :ベクトル式
```

EPS は四つの引数を取り、4 階の完全反対称テンソルとローレンツの 4 次元ベクトルとの縮約を表します。つまり、

$$\epsilon_{ijkl} = \begin{cases} +1 & \text{もし } i,j,k,l \text{ が 0,1,2,3 の偶置換の時} \\ -1 & \text{奇置換の時} \\ 0 & \text{それ以外の時} \end{cases}$$

例えば $\epsilon_{ij\mu\nu}p_\mu q_\nu$ のような縮約は、I と J がインデックスとして宣言されているとして、eps(i,j,p,q) と書かれます。

17.2 ベクトル変数

G 演算子の粒子を表す識別子を除くと、この節で説明する変数はすべてベクトルです。ここで使われる変数は VECTOR 宣言でベクトル型として宣言しておく必要があります。例えば、

```
vector  p1,p2;
```

は P1 と P2 がベクトルであると宣言します。インデックスとして宣言された変数や、質量が与えられている変数は自動的にベクトル型と宣言されます。

17.3 追加された式の型

高エネルギー物理学での計算を行うために、二つの型が式の型として追加されています。

17.3.1 ベクトル式

これは通常のベクトルの演算規則に従います。つまり、ベクトルとスカラー式や数値との積はまたベクトルになります。またベクトル同士の和や差はベクトルです。これ以外の演算はエラーになります。さらに、対話モードで計算中にベクトルであるべき所に何も定義されていない変数が現れると、その変数をベクトルとして宣言するかどうか問い合わせてきます。バッチモードでは自動的にベクトルとして宣言され、その旨のメッセージが出力されます。

例:

P と Q がベクトルとして宣言されているとすると、次の式はベクトルになります。

```
p
2*q/3
2*x*y*p - p.q*q/(3*q.q)
```

しかし、 p*q や p/q は許されません。

17.3.2 ディラック式

これは γ 行列を含む式をいいます。γ 行列は 4×4 行列で表されており、積や和、差はまたディラック式になります。またディラック式とスカラー式との積はディラック式になります。ディラック変数というのはありません。ディラック式に γ 行列の付かないスカラー式が現れたときには、4×4 の単位行列が掛かっているものとして扱われます。例えば、`g(l,p) + m` は `g(l,p) + m*<4x4` の単位行列>と解釈されます。ディラック式の積は普通の行列の積として計算され、非可換な演算です。

17.4 トレースの計算

ディラック式が評価されると、展開された式の各項に含まれる γ 行列の積のトレースの 1/4 を計算します。トレースの 1/4 を取るのは、例えばスカラー式 `M` と、`M*<4x4` の単位行列>の意味での `M` との混同を避けるためです。式中に現れるインデックスに対する縮約は行われます。もし対応しないインデックスがあるとエラーになります。

トレースを計算する方法は、このシステムが作成された時点での最良の方法を使っています。例えば、トレースの積でのインデックスに関する縮約を求める Chisholm の方法に加えて、REDUCE には Kahane による γ 行列の積におけるインデックスの縮約を求める方法を使っています。これらの方法に付いては Chisholm, J. S. R. による論文 Il Nuovo Cimento X, 30, 426 (1963) と Kahane, J. の論文 Journal Math. Phys. 9, 1732 (1968) に述べられています。

`NOSPUR` と宣言された粒子に対してはトレースの計算が行われません。例えば、

```
nospur l1,l2;
```

では `L1` と `L2` を含む γ 行列に付いてはトレースを取りません。しかし、一つ以上の粒子を含む計算では場合によってカタストロッフィック・エラー

```
This NOSPUR option not implemented
(この NOSPUR オプションは備えられていません)
```

を起こすことがあります。 (理由も一緒に表示されます!) もし、このようなことが起これば知らせてください。

`NOSPUR` 宣言された粒子を含む γ 行列のトレースを計算したい場合、その粒子に対する `SPUR` 宣言を改めて行なって下さい。

CVIT パッケージには別の方法がありますので参照して下さい。.

17.5 質量宣言

計算において粒子に “質量” を与えたい場合があります。これは、`LET` 文で、

```
let p.p= m^2;
```

としてもできます。しかし、別の方法として MASS と MSHELL の二つのコマンドが用意されており、これを使って定義することができます。MASS は次の形の方程式のリストを引数として取ります。

```
<ベクトル変数> = <スカラー変数>
```

例えば、

```
mass p1=m, q1=mu;
```

このコマンドはスカラー変数をベクトル変数の質量に対応させます。これで、

```
mshell <ベクトル変数>,...,<ベクトル変数>;
```

と入力すれば質量がそれぞれの引数に対応付けられ、次のような置換規則が定義されます。

```
<ベクトル変数>.<ベクトル変数> = <質量>^2
```

もし、変数が MASS 宣言されていなければエラーを起こします。

17.6 例

ここでは高エネルギー物理学での計算の例として、Bjorken と Drell によるコンプトン散乱断面積の計算をやってみます。計算は次の式のトレースを求めればよい。

$$\frac{\alpha^2}{2}\left(\frac{k'}{k}\right)^2\left(\frac{\gamma.p_f+m}{2m}\right)\left(\frac{\gamma.e'\gamma.e\gamma.k_i}{2k.p_i}+\frac{\gamma.e\gamma.e'\gamma.k_f}{2k'.p_i}\right)\left(\frac{\gamma.p_i+m}{2m}\right)$$
$$\left(\frac{\gamma.k_i\gamma.e\gamma.e'}{2k.p_i}+\frac{\gamma.k_f\gamma.e'\gamma.e}{2k'.p_i}\right)$$

ここで k_i と k_f は入射光子と散乱光子の四次元運動量を、(分極ベクトル e 、 e' と実験室系でのエネルギー k 、 k') そして p_i, p_f は入射電子と散乱電子の四次元運動量を表しています。

全体の因子 $\frac{\alpha^2}{2m^2}\left(\frac{k'}{k}\right)^2$ を除けば、次のトレースの 1/4 を求めればよい。

$$(\gamma.p_f+m)\left(\frac{\gamma.e'\gamma.e\gamma.k_i}{2k.p_i}+\frac{\gamma.e\gamma.e'\gamma.k_f}{2k'.p_i}\right)(\gamma.p_i+m)$$
$$\left(\frac{\gamma.k_i\gamma.e\gamma.e'}{2k.p_i}+\frac{\gamma.k_f\gamma.e'\gamma.e}{2k'.p_i}\right)$$

適当な代入 (p_i を P1、p_f を PF、k_i を KI そして k_f を KF と表し) を行うと、プログラムは次のようになります。

```
on div; % これにより結果は Bjorken と Drell によるものと同じ形
        % になります。
mass ki= 0, kf= 0, p1= m, pf= m; vector e,ep;
```

```
% e をベクトルとして使用すると、自然対数の底としての性質は
% 失われます。
mshell ki,kf,p1,pf;
let p1.e= 0, p1.ep= 0, p1.pf= m^2+ki.kf, p1.ki= m*k,p1.kf=
    m*kp, pf.e= -kf.e, pf.ep= ki.ep, pf.ki= m*kp, pf.kf=
    m*k, ki.e= 0, ki.kf= m*(k-kp), kf.ep= 0, e.e= -1,
    ep.ep=-1;
for all p let gp(p)= g(l,p)+m;
comment これは入力を省略するために定義しています;
gp(pf)*(g(l,ep,e,ki)/(2*ki.p1) + g(l,e,ep,kf)/(2*kf.p1))
  * gp(p1)*(g(l,ki,e,ep)/(2*ki.p1) + g(l,kf,ep,e)/
    (2*kf.p1))$
write "The Compton cxn is",ws;
```

(ここで、PI ではなく P1 という変数名を使ったのは、PI は円周率を表す変数として予約されているからです。)

このプログラムは次のような結果を返します。

```
                            (-1)        (-1)            2
The Compton cxn is 1/2*K*KP     + 1/2*K    *KP + 2*E.EP  - 1
```

17.7 四次元以外への拡張

これまでの議論では通常の四次元での QED 計算に限定していました。しかし、多くの場合、プログラムは任意の次元数で動くように作られています。コマンド

```
vecdim <式>;
```

は計算の次元を設定します。次元は記号でも数値でも構いません。しかし、EPS 演算子と γ^5 シンボル (A) は四次元以外では定義されていないので、使うとエラーを起こします。

第18章　REDUCEと Rlispのユーティリティ

REDUCE とそれを記述している言語である Rlisp は、プログラムするときに便利なユーティリティを多く持っています。次に説明するのは REDUCE の大部分のインプリメントで利用できます。

18.1　Lispコンパイラ

多くのバージョンの REDUCE は Lisp コンパイラを持っている。これは必要に応じて自動的にロードされます。このようなコンパイラを持っているかどうかについてはシステムの説明書を参照してください。コンパイラを起動するにはスイッチ COMP をオンにします。以後、関数を定義するとその関数はコンパイルされます。使われているコンパイラは Griss-Hearn [1] のコンパイラを元にして作られたものです。いくつかのスイッチが用意されています。ただし、全てのコンパイラでこれらの全てのスイッチが使えるとは限りません。それらのスイッチは次のものです。

PLAP　オンにすると、コンパイラが生成したマクロをプリントする。

PGWD　オンにすると、マクロから生成された機械語をプリントする。

PWRDS　オンにすると、
<関数> COMPILED, <words> WORDS, <words> LEFT
のような統計メッセージをプリントします。最初の数字はコンパイルされた関数がどれだけの機械語領域を使っているかを示し、二番目の数字は残りの機械語領域の大きさを示す。

18.2　FASLコード生成

ほとんどの REDUCE のバージョンで、Lisp,RLisp や REDUCE のコマンドやプログラムからファーストロードバージョンを作ることができるようになっています。RLisp または REDUCE で次のように入力します。

```
faslout <ファイル名>;
<コマンドの列もしくは IN コマンド>
faslend;
```

[1] M. L. Griss and A. C. Hearn, “A Portable LISP Compiler”, SOFTWARE — Practice and Experience 11 (1981) 541-605。

このようなファイルをロードするには、LOAD コマンドを使います。例えば、load foo; または load foo, bah; のように入力します。

このような方法で、Fast-loading バージョンのファイルが作られます。あるインプリメントでは、拡張子が異なったソースファイルと同じ名前のファイルが作られます。例えば、PSL 版では、b (バイナリーを表す) という拡張子を持ったファイルが作られます。また、CSL 版では、fast-loading コードが、ある決まった一つのファイルに追加されます。このファイルには、すべてのそのようなコードが保存されます。CSL 版には、このファイルを管理する関数が用意されています。詳しくは CSL 版のマニュアルを参照してください。

このような Fast-loading ファイルの構築を行うとき、ある種のコマンドはこの過程で直ちに実行されなければならないことに注意して下さい。例えば、マクロは展開され、また属性リストに対する操作は実行されます。これらについての詳細については、REDUCE のソースを参照して下さい。

よけいな出力を避けるために、入力した式の最後はセミコロンではなくドル記号で終わるようにします。LOAD コマンドでロードしたときにはどちらを使ったかには関係なく出力は行われません。

もし、Fast-loading ファイルを作成してからソースファイルを変更した場合には、再度上記の手順を繰り返して Fast-loading ファイルを更新しておくことを忘れないようにして下さい。また Fast-loading ファイルを作成するときに読み込まれたプログラムは今使っている REDUCE の環境には取り込まれず、変換された結果がファイルに出力されるだけです。ただし、マクロ関数の定義やプロパティリストを操作するような関数等は実行された結果が残っています。従って、作成したファイルを使うときには、LOAD コマンドを使って出力した Fast-loading ファイルを読み込む必要があります。

もし、ロードされるファイルが完全なパッケージになっているときには、LOAD コマンドよりも LOAD_PACKAGE コマンドを使った方がよい。構文は二つとも同じですが、LOAD_PACKAGE はどのパッケージがすでに読み込まれたかといった情報を管理しており、これはシステムを正常に動かすために必要だからです。

18.3 Lisp の相互参照作成プログラム

CREF は Standard Lisp のプログラムから、次のような情報を生成します。

1. 次のような “要約” :
   (a) 処理されたファイルのリスト;
   (b) “関数の入口” のリスト (使われていない関数または関数内からのみ使われている関数);
   (c) 未定義の関数のリスト (使われているがファイル中では定義されていない関数のリスト);
   (d) 大域的な変数として使われているが GLOBAL もしくは FLUID と宣言されていない変数のリスト;
   (e) GLOBAL と宣言されているが FLUID と宣言された変数として使われている (つまり関数内で束縛された変数として使われている) 変数のリスト;

(f) FLUID 変数で関数内に束縛されておらず、従って GLOBAL と宣言しても構わない変数のリスト;

(g) 全ての GLOBAL 変数のリスト;

(h) 全ての FLUID 変数のリスト;

(i) 全ての関数のリスト.

2. それぞれの大域変数についての “大域変数の使用” 表:

(a) FLUID または GLOBAL と宣言されたのち、その変数を使用している関数のリスト;

(b) 宣言せずその変数を使用している関数のリスト;

(c) 変数を関数内で束縛している関数のリスト;

(d) SETQ で値を変更している関数のリスト。

3. それぞれの関数についての “関数の使用” 表:

(a) どこで定義されているか;

(b) この関数を呼び出している関数のリスト;

(c) この関数が使っている関数のリスト;

(d) 使っている大域変数のリスト。

このプログラムはそれぞれの関数が正しい引数で使われているかも調べて、もし引数の数が合わなければ診断情報を出力します。

出力は関数名の最初の 7 文字に対して、そのアルファベット順に並べて出力されます。

18.3.1 制限事項

REDUCE の代数モードでの関数は記号モードの関数として扱われます。そのため代数モードから記号モードで定義された関数を呼び出すために使われている AEVAL 等の関数がリスト中に現れます。

18.3.2 使用法

相互参照プログラムを起動するには CREF スイッチを使います。on cref とすると、相互参照プログラムがロードされ、実行されます。これ以後 off cref が入力されるまで、入力もしくは IN コマンドで読み込まれた定義から相互参照リストが作成されます。例えば、ファイル tst.red 中の関数の相互参照リストをファイル tst.crf に出力したければ、次のように入力すればよい。

```
out "tst.crf";
on cref;
in "tst.red"$
off cref;
shut "tst.crf";
```

一つ以上のファイルを処理したければ IN コマンドを off cref を実行する前に追加するか、IN コマンドを複数のファイルを入力するように変更すればよい。

18.3.3 オプション

フラグ NOLIST をつけた関数は調べられず、出力にも現れない。初期状態では Standard Lisp の関数はすべてそのフラグがつけられている。(実際はこれらの関数名のリストは変数 NOLIST!*に入れてある。もしこれらの関数についての情報も必要であれば load cref を実行して CREF のプログラムをロードした後、この変数の値を NIL にすればよい)。

また、フラグ EXPAND がつけられているか、FORCE スイッチがオンでフラグ NOEXPAND がついていない場合、マクロ関数は相互参照プログラムに渡される前に展開されます。従ってこれらのフラグを操作することで、どの関数を展開し、どれを展開せずにそのまま渡すかを変更することができます。

18.4 REDUCE の清書

REDUCE はプログラムを標準の形式で出力する機能を持っています。この機能はスイッチ PRET で起動されます。PRET は通常オフになっています。

システムは代数モードで入力されたものを記号モードでのプログラムに変換します。清書プログラムは書き出す前に、プログラムが代数モードでの式として解釈できるかどうか調べます。多くの場合これでうまくいき、代数モードでのプログラムとして出力されます。しかし時には記号モードでのプログラムがそのまま出力されることがあります。この場合、入力した式と意味的には同じプログラムですが、記号モードに変換された、入力された式とはほとんど似ていない式が出力されてしまいます。

ファイル全体を清書したければ off output,msg; とすれば (多分) 清書されたプログラムだけが出力されるでしょう。DEFN とは違って、PRET がオンの状態では入力した式は評価されます。

18.5 S-式の清書

REDUCE は Lisp の S-式を清書するプログラムも持っています。これを行う関数は PRETTYPRINT です。この関数は引数で与えられた Lisp のプログラムを、等価な清書した形で書き出します。

スイッチ DEFN を使うことで、ユーザは REDUCE のプログラムを Lisp プログラムに変換したものを、出力することが出来ます。これは、REDUCE もしくは Rlisp の構文から Lisp へ変換するのに便利です。off msg; は警告メッセージが出力されるのを抑制します。

注意: もし DEFN がオンであると入力したものは評価されません。

第19章　REDUCEの保守

REDUCE は、その機能の追加やそれぞれの機能の強化が現在も引き続いて行われている。バグが発見されたときのその修正や、複雑な機能をより簡単にすること等が行われています。ユーザがこのような拡張機能が容易に入手できるように、*REDUCE* **ネットワークライブラリ**が設立されています。インターネットの計算機ネットワークを通じて、電子メールでこのような資料が入手できるようになっています。

文書やソースファイル、ユーティリテイファイルの他に REDUCE を参照している論文のリストもあります。これには約 800 の論文が含まれています。このライブラリの使い方は電子メールのアドレスを持っている登録された REDUCE のユーザに送ってあります。もし完全なライブラリのリストが欲しければ、`reduce-netlib@rand.org` 宛に *send index* または *help* と一行書いたメールを送ってください。最新の REDUCE 情報を得るには同じように一行に *send info-package* と書いて送ると、折り返し返送されます。またデモファイルは *send demonstration* と書いて送ると得られます。もし紙に打ち出した REDUCE の情報や文献表が欲しければ

REDUCE securetary
RAND, 1700 Main Street, P.O. Box
2138, Santa Monica, CA 90407-2138 (*reduce@rand.org*)

から入手できます。ネットワークライブラリは別のアドレスから入手することもできます。このマニュアルを書いている現在では、*reduce-netlib@can.nl* と *reduce-netlib@pi.cc.u-tokyo.ac.jp* からも同じ情報が入手できます。さらに、

REDUCE の WWW サーバもある。この URL は

`http://www.rrz.uni-koeln.de/REDUCE/`

である。ここには REDUCE に関する一般的な情報の他、ネットワークライブラリ、デモ版、REDUCE のプログラミングの例、マニュアル、REDUCE のオンラインヘルプへのポインターがある。

最後に、REDUCE の電子フォーラムが同じネットワークからアクセスできます。これは REDUCE のユーザが質問をしたり、REDUCE について議論する場を与えています。ネットワークライブラリの追加や修正、新しい REDUCE のリリース等の情報はこのフォーラムを通してアナウンスされます。電子メールにアクセスすることのできるユーザは、このフォーラムの会員として登録することでこれらの情報が入手できます。これには *reduce-forum-request@rand.org* にフォーラムに加わりたいという内容のメールを送ってください。

付 録A　予約語

ここには通常 REDUCE で予約されている識別子のリストを挙げています。これには、コマンド名、演算子名およびスイッチの名前が含まれています。しかし、特定のシステムで予約されている単語は含まれていません。

コマンド　ALGEBRAIC ANTISYMMETRIC ARRAY BYE CLEAR CLEARRULES COMMENT CONT DECOMPOSE DEFINE DEPEND DISPLAY ED EDITDEF END FACTOR FOR FORALL FOREACH GO GOTO IF IN INDEX INFIX INPUT INTEGER KORDER LET LINEAR LISP MASS MATCH MATRIX MSHELL NODEPEND NONCOM NOSPUR OFF ON OPERATOR ORDER OUT PAUSE PRECEDENCE PRINT_PRECISION PROCEDURE QUIT REAL REMFAC REMIND RETRY RETURN SAVEAS SCALAR SETMOD SHARE SHOWTIME SHUT SPUR SYMBOLIC SYMMETRIC VECDIM VECTOR WEIGHT WRITE WTLEVEL

論理演算子　EVENP FIXP FREEOF NUMBERP ORDP PRIMEP

内挿演算子　& := = >= > <= < => + * / ^ ** . WHERE SETQ OR AND NOT MEMBER MEMQ EQUAL NEQ EQ GEQ GREATERP LEQ LESSP PLUS DIFFERENCE MINUS TIMES QUOTIENT EXPT CONS

数学関数

ABS ACOS ACOSD ACOSH ACOT ACOTD ACOTH ACSC ACSCD ACSCH ASEC ASECD ASECH ASIN ASIND ASINH ATAN ATAND ATANH ATAN2 ATAN2D CBRT COS COSD COSH COT COTD COTH CSC CSCD CSCH EXP FACTORIAL FIX FLOOR HYPOT LN LOG LOGB LOG10 NEXTPRIME ROUND SEC SECD SECH SIN SIND SINH SQRT TAN TAND TANH

前置演算子

APPEND ARGLENGTH CEILING COEFF COEFFN COFACTOR CONJ DEG DEN DET DF DILOG EPS ERF EXPINT FACTORIZE FIRST GCD G IMPART INT INTERPOL LCM LCOF LENGTH LHS LINELENGTH LTERM MAINVAR MAT MATEIGEN MAX MIN MKID NULLSPACE NUM PART PF PRECISION RANK REDERR REDUCT REMAINDER REPART REST RESULTANT REVERSE RHS SECOND SET SOLVE STRUCTR SUB SUM THIRD TP TRACE VARNAME

予約語

E I INFINITY K!* NIL PI T

スイッチ

ADJPREC ALGINT ALLBRANCH ALLFAC BFSPACE COMP COMPLEX CRAMER CREF DEFN DEMO DIV ECHO ERRCONT EVALLHSEQP EXP EZGCD FACTOR FORT GCD IFACTOR INT INTSTR LCM LIST LISTARGS MCD MODULAR MSG MULTIPLICITIES NAT NERO NOSPLIT OUTPUT PERIOD PGWD PLAP PRET PRI PWRDS RAISE RAT RATARG RATIONAL RATIONALIZE RATPRI REVPRI ROUNDALL ROUNDBF ROUNDED SAVESTRUCTR SOLVESINGULAR TIME TRA TRFAC TRINT

その他の予約語

BEGIN DO EXPR FASLOUT FEXPR FLAGOP INPUT LAMBDA LISP LOAD MACRO PRODUCT REPEAT SMACRO SUM WHILE WS

付 録B　CSL版のREDUCE

CSL(Codemist Standard Lisp) の上で動いている REDUCE を起動するときの構文は次の通りです。

```
reduce [オプション]
```

オプションは **-X** ＜**引数**＞ の形で与えます。ここで、X はコマンドで、＜引数＞ はコマンドに対する引数です。コマンドによっては、引数を取らないものもあります。コマンドは大文字でも小文字でもどちらで指定しても構いません。また、引数を指定する場合、コマンドとの間に空白を入れても、入れなくともどちらでも構いません。

ここで、オプションとその意味は次の通りです。

- -- 出力ファイル名

  このオプションは、出力を＜出力ファイル名＞に出力します。これは、コマンドラインでの出力のリダイレクト “> 出力ファイル” と同じ働きをするものです。ウインドウ環境 (特に、Windows NT) でこのリダイレクト機能が働かないために用意しています。このオプションはウインドウ環境のみで使用できます。

- -B
  このオプションは特殊なオプションで、これを指定すると (batchp) 関数の返す値を通常とは反対にします。バッチモードでの動作をテストするためのものです。

- -C

  このオプションは、著作権に関する注意を表示します。

- -D 名前=値

  Lisp の変数**名前** に **値**を代入する。

- -F 数値
  数値 は 1000 以上 65536 以下の数値で、これを指定すると通常の端末アプリケーションとしてではなくサーバとして起動し、指定された番号のポートからコマンドを受け取ります。-F- を指定するとポート番号としては規定値 (1206) が使用されます。

- -G

  デバッグ用のオプションで、!*backtrace の値を真 (true) に設定します。これによって、プログラムが errorset 関数を呼び出している場合、エラーを起こしたときに適当なメッセージが出力されます。

- -I ファイル名

  CSL が起動するときのイメージファイル名を指定する。実行時にロードされるパッケージはイメージファイル中に一緒に置かれています。このオプションを複数回指定して複数のファイルを指定することが出来ます。-I-を指定したときは、実行モジュールに付随した、標準のファイルを指定したことになります。

- -K 整数

  使用するメモリのサイズをキロバイト単位で指定する。(CSL 版の REDUCE では計算で必要なメモリは、システムで用意できる限度まで自動的に追加されていきます。従って、通常はこの値を指定する必要はありません。この値を指定したときには、最初にここで指定したサイズのメモリが最初に確保され、後で追加されることはありません。)

- -L ファイル名

  全ての出力を指定されたファイルにも書き出す。これはセッション中で spool "<logfile>" を実行するのと同じことです。

- -M n:l:h
  これは MEMORY_TRACE が有効な状態で作成されたシステムに対するオプションです。l から h までの範囲のメモリに n 回アクセスがあると割り込みが発生します。

- -O ファイル名

  コンパイルした結果のモジュール、複写したモジュールの出力および preserve 関数を実行して、イメージファイルを書き出すときの出力ファイル名を指定します。

- -P

  これはプロファイル用のオプションとして予約されています。

- -Q

  REDUCE で **off echo;** と入力したときと同様に、入力した文字のエコーを中止する。

- -R 数値

  乱数発生ルーチンの初期値を与える。 -R 0 とすると (これが既定値)、初期値は現在の日時から設定されます。-R nnn,mmm と指定することで、64 ビットの初期値を指定することができます。

- -S

  変数 !*plap の値を T に設定する。コンパイラは生成したバイトコードを表示します。

- -T モジュール名

  指定したモジュールの作成時刻を表示します。

- -U シンボル名

  指定された名前のシンボルを **NIL** に設定する。

- -V

  REDUCE で **on echo;** を実行したのと同様に、エコーを再開する。

- -W

  Archimedes システムでは、“-w” はウインドウ版の CSL を起動するために使用しています。それ以外の Window システムに対しては、将来より詳細なコントロールを行なわせるためのオプションとして予約しています。このオプションは、ウインドウシステムでのみ使用できます。

- -X

  これは “文書化されていない” オプションで、システムの保守を行う人の為に設けてあります。このオプションを設定すると、セグメント違反によるエラーをトラップすることを禁止します。これによって、エラーの解析が容易になります。このオプションは、ソースコードを持っている人以外には役に立たないでしょう。

- -Y

  通常、イメージファイルを読み込んだ時、restart 関数が定義されていると自動的にこの関数が実行されます。”-Y”オプションは、この機能を停止させ、Lisp の読み込み–評価–出力ループに制御を渡します。このオプションは、例えば、イメージファイルを作成したのだが、それが何らかの理由で壊れており正常に起動しない場合、原因を調べるために Lisp モードで起動させたい時に使用します。

- -Z

  CSL を、初期化の為のイメージファイルを取り込まない、裸の状態 (cold start) で起動します。システムを構成する場合には、裸の Lisp を起動する必要があります。このような時に使用します。

第II部

ユーザパッケージ

第1章　ALGINT: 平方根を含む式の積分

James H. Davenport
School of Mathematical Sciences
University of Bath
Bath BA2 7AY
England
e–mail: *jhd@maths.bath.ac.uk*

このパッケージでは、REDUCE の不定積分パッケージに追加して、被積分項に平方根を含み、かつ積分が平方根を含む式で表せるようなクラスの不定積分を求めることができます。これは J.H.Davenport による論文 “On the Integration of Algebraic Functions”, LNCS 102, Springer Verlag, 1981 に記述されている算法をプログラムしたものです。もっと詳しいことについては、この論文とプログラムのソースを参照してください。

ALGINT パッケージが LOAD_PACKAGE コマンドでロードされると、例えば次のような不定積分が求められるようになります。

```
int(sqrt(x+sqrt(x**2+1)/x,x);
```

もし、後になってこのパッケージの機能を使わずに計算を行いたい場合は、ALGINT スイッチをオフにしてください。このスイッチはこのパッケージがロードされたときにオンになります。

通常の積分パッケージが用意しているスイッチ (例えば TRINT) 等は、このパッケージでもサポートしています。さらにスイッチ TRA オンであれば、代数関数の積分のトレースを出力します。

このパッケージにはこれ以上の文書はついていません。

参考文献

Davenport, 1981 Davenport, J.H., ”On the Integration of Algebraic Functions”, LNCS 102, Springer 1981.

第2章　ARNUM:代数的数

Eberhard Schrüfer

Institute SCAI.Alg
German National Research Center for Information Technology (GMD)
Schloss Birlinghoven
D-53754 Sankt Augustin
Germany

Email: *schruefer@gmd.de*

代数的数は、ある基礎体上の既約多項式の解として定義されます。例えば、代数的数である i(虚数単位) は、多項式 i^2+1 によって定義されています。代数的数 a の計算は、定義式を法とする多項式の計算とみなせます。

代数的数 a に対して与えられた定義式:

$$a^n \ + \ p_{n-1}a^{n-1} \ + \ ... \ + \ p_0$$

より、a から作れるすべての代数的数は、次の形を取ります。

$$r_{n-1}a^{n-1} \ + \ r_{n-2}a^{n-2} \ + \ ... \ + \ r_0$$

ただし、r_j は、基礎体 (例えば、有理数) の要素である。

加算は、次式によって定義される。

$$\begin{aligned}(r_{n-1}a^{n-1} \ + \ r_{n-2}a^{n-2} \ + \ ...) \ + \ (s_{n-1}a^{n-1} \ + \ s_{n-2}a^{n-2} \ + \ ...) \ = \\ (r_{n-1}+s_{n-1})a^{n-1} \ + \ (r_{n-2}+s_{n-2})a^{n-2} \ + \ ...\end{aligned}$$

二つの代数的数の掛け算は、通常の多項式の積を行った結果を、定義多項式を法として簡約した式として定義される。

$$\begin{aligned}(r_{n-1}a^{n-1}+r_{n-2}a^{n-2}+...) \ \times \ (s_{n-1}a^{n-1}+s_{n-2}a^{n-2}+...) = \\ r_{n-1}s^{n-1}a^{2n-2}+... \ \textbf{modulo} \ a^n \ + \ p_{n-1}a^{n-1} \ + \ ... \ + \ p_0 \\ = \quad q_{n-1}a^{n-1} \ + \ q_{n-2}a^{n-2} \ + \ ...\end{aligned}$$

二つの代数的数 r と s の商 q はまた代数的数になります。

$\frac{r}{s} = q$ または $r = qs$.

後の式を具体的に書き表すと、次のようになります。

$$\begin{aligned}
&(r_{n-1}a^{n-1} + r_{n-2}a^{n-2} + \ldots) \\
&\quad = \quad (q_{n-1}a^{n-1} + q_{n-2}a^{n-2} + \ldots) \times (s_{n-1}a^{n-1} + s_{n-2}a^{n-2} + \ldots) \\
&\qquad \mathbf{modulo}(a^n + p_{n-1}a^{n-1} + \ldots) \\
&\quad = \quad (t_{n-1}a^{n-1} + t_{n-2}a^{n-2} + \ldots)
\end{aligned}$$

ここで t_i は q_j の一次の多項式で表されます。この両辺の a のべき乗の係数を等しいと置くことにより、商の係数 q_j に対する線形方程式系が得られます。

以上の結果から、代数的数に対する四則演算が計算できます。

代数的数を REDUCE にインプリメントするに当たって、定義多項式は一般には有理数係数を含む多項式を許すようにしています。代数的数は、REDUCE 内部では前に`:ar:`というタグを付けた有理数体上の一変数多項式として表しています。

< **代数的数** > ::=
`:ar:` . < **有理数体上の一変数多項式** >

< **有理数体上の一変数多項式** > ::=
< **変数** > .** <*ldeg*> .* < **有理数** > .+ <*reductum*>

<*ldeg*> ::= 整数

< **有理数** > ::=
`:rn:` . < **整数の分子** > . < **整数の分母** > : integer

<*reductum*> ::= < **有理数体上の一変数多項式** > : < **有理数** > : nil

代数的数を定義するには`defpoly`文を使います。 代数的数、例えば $\sqrt{2}$、を定義するには次のように入力します。

```
defpoly sqrt2**2 - 2;
```

例:

```
defpoly sqrt2**2-2;

1/(sqrt2+1);

sqrt2 - 1

(x**2+2*sqrt2*x+2)/(x+sqrt2);
```

```
x + sqrt2

on gcd;

(x**3+(sqrt2-2)*x**2-(2*sqrt2+3)*x-3*sqrt2)/(x**2-2);

  2
(x  - 2*x - 3)/(x - sqrt2)

off gcd;

sqrt(x**2-2*sqrt2*x*y+2*y**2);

abs(x - sqrt2*y)
```

いままでは、ただ一つの代数的数のみを扱っていましたが、実際には複数の代数的数を同時に扱う必要が出てきます。このような場合の処理法として二つの方法があります。一つは定義多項式を多変数の場合に拡張する方法 [17] で、もう一つの方法は、複数の代数的数をただ一つの定義多項式で表される一つの代数的数で表してしまうことです。このパッケージでは後者 (原始表現 primitive representation) の方法を取っています。

原始要素 (primitive element) を求める方法は Trager [45] によって与えられたのと同じ方法を使っています。複数の代数的数を定義するには、`defpoly` 文の引数としてそれぞれの定義多項式をリストとして与えます。全ての代数的数は原始要素で表されます。

例:

```
defpoly sqrt2**2-2,cbrt5**3-5;

*** defining polynomial for primitive element:
(*** 原始要素の定義多項式は:)

  6       4        3        2
a1  - 6*a1  - 10*a1  + 12*a1  - 60*a1 + 17

sqrt2;

          5             4              3              2
48/1187*a1  + 45/1187*a1  - 320/1187*a1  - 780/1187*a1  +


735/1187*a1 - 1820/1187

sqrt2**2;

2
```

Trager の算法を使った、代数的数体上での因数分解もできます。

例:

```
defpoly a**2-5;

on factor;

x**2 + x - 1;

(x + (1/2*a + 1/2))*(x - (1/2*a - 1/2))
```

関数 split_field によって、与えられた多項式を一次因子に分解する原始要素の満たす最小の多項式を計算できます。算法は Trager によるもので、原始要素の計算を繰り返して求めています。

例:

```
split_field(x**3-3*x+7);

*** Splitting field is generated by:
(*** 分解体は次式で生成されます:)

  6        4        2
a2  - 18*a2  + 81*a2  + 1215



         4          2
{1/126*a2  - 5/42*a2  - 1/2*a2 + 2/7,


           4          2
 - (1/63*a2  - 5/21*a2  + 4/7),


        4          2
1/126*a2  - 5/42*a2  + 1/2*a2 + 2/7}


for each j in ws product (x-j);

 3
x  - 3*x + 7
```

より完全な説明が文献 [11] にあります。

第3章　ASSIST: 種々の有用なツール

Hubert Caprasse
Département d'Astronomie et d'Astrophysique
Institut de Physique, B–5, Sart Tilman
B–4000 LIEGE 1, Belgium
e–mail: *caprasse@vm1.ulg.ac.be*

ASSIST パッケージは REDUCE をいろいろな計算方法に適用させる数多くの一般的用途の関数を提供しています。この節での計算例は全て ASSIST パッケージをロードしていることが必要です。

3.1　スイッチのコントロール

二つの関数 SWITCHES, SWITCHORG は引数を持ちません。

SWITCHES は有理関数を扱う場合に最もしばしば使われるスイッチ、EXP, DIV, MCD, GCD, ALLFAC, INTSTR, RAT, RATIONAL, FACTOR、の状態を表示します。分散した多項式の扱い方を制御するスイッチ DISTRIBUTE もまた含まれています。(節 3.8 を参照のこと)。

SWITCHORG はほとんど全てのスイッチを最初に REDUCE を起動した時の状態に状態に戻します。(RESET, 43 章も参照して下さい。)

新しいスイッチ DISTRIBUTE は多項式を分散した形式に変換する機能を提供します。

3.2　リスト構造の操作

リストを操作する関数は新しい構造 BAG が扱えるように拡張されています。

i. MKLIST は要素が 0 である、長さ n のリストを生成する。また、リスト l に対して、要素 0 を追加してその長さを n に拡張します。:

```
MKLIST n;          %% n は整数
MKLIST(l,n);       %% l はリストで、n は整数
```

ii. 長さ n のサブリストのリストで p 要素は 0 で、$n-p$ 要素は 1 のものを生成する。

```
SEQUENCES 2; ==> {{0,0},{0,1},{1,0},{1,1}}
```

関数 KERNLIST はカーネルのプリフィックスをリストのプリフィックスに変換する。コピーされたものが出力される。

```
KERNLIST (<kernel>); ==> {<kernel arguments>}
```

DELETE はリスト中で第一引数に一致する最初の要素を削除する。REMOVE は整数で指定された番号の要素を削除する。DELETE_ALL は第一引数に一致する全ての要素を削除する。DELPAIR はペアを要素とするリストに対して、ペアの先頭が第一引数に一致する最初の要素を削除する。

```
DELETE(x,{a,b,x,f,x}); ==> {a,b,f,x}
REMOVE({a,b,x,f,x},3); ==> {a,b,f,x}
DELETE_ALL(x,{a,b,x,f,x}); ==> {a,b,f}
DELPAIR(a,{{a,1},{b,2},{c,3}}; ==> {{b,2},{c,3}}
```

iv. 関数 ELMULT は第一引数に一致する要素の数を計算する。関数 FREQUENCY はリストの全ての要素に対して、各々が何回現れるかをペアにしたもののリストを返す。

```
ELMULT(x,{a,b,x,f,x}) ==> 2
FREQUENCY({a,b,c,a}); ==> {{a,2},{b,1},{c,1}}
```

v. 関数 INSERT は与えられたリストの指定された位置に要素を挿入する。関数 INSERT_KEEP_ORDER および MERGE_LIST は第三引数で指定された関数で順序付けて挿入を行なう。MERGE_LIST はリストを合併する。

```
ll:={1,2,3}$
INSERT(x,ll,3); ==> {1,2,x,3}
INSERT_KEEP_ORDER(5,ll,lessp); ==> {1,2,3,5}
MERGE_LIST(ll,ll,lessp); ==> {1,1,2,2,3,3}
```

vi. 関数 FIRST と REST に対応して、LAST と BELAST を用意している。LAST 最後の要素を、また BELAST は最後の要素を取り除いたリストを返す。CONS, (.), POSITION, DEPTH, PAIR, APPENDN, REPFIRST, REPLAST の関数がある。文字 ".(ドット)" にはちょっと注意が必要です。これは、幾つかの異なった操作を表しています。

1. リストの先頭に要素を追加する。これは、CONS 関数です。

   ```
   4 . {a,b}; ==> {4,a,b}
   ```

2. リストの後に付けた場合は、PART 演算子として作用する。

   ```
   {a,b,c}.2; ==> b
   ```

3. 4–次元ベクトルに適用した場合、HEPHYS パッケージでの演算として作用する。(REDUCE ユーザーズマニュアルの 17 章を参照)

POSITION はリスト中での要素の位置を返す。DEPTH は、リストの各要素の深さが同じであれば、その深さをかえす。要素によって深さが異なっている場合はその旨のメッセージが返される。PAIR は二つのリストから、各々のリストの要素をペアにしたリストを返す。この

ようなリストは連想リストあるいは ALIST と呼ばれる。APPENDN は任意の数のリストを引数にとり、それらのリストを結合したリストを返す。REPFIRST はリストの先頭要素を入れ換える。REPREST はリストの先頭以外の要素をいれかえる。

```
ll:={{a,b}}$
ll1:=ll.1;                  ==> {a,b}
ll.0;                       ==> list
0 . ll;                     ==> {0,{a,b}}
DEPTH ll;                   ==> 2
PAIR(ll1,ll1);              ==> {{a,a},{b,b}}
REPFIRST{new,ll);           ==> {new}
ll3:=APPENDN(ll1,ll1,ll1); ==> {a,b,a,b,a,b}
POSITION(b,ll3);            ==> 2
REPREST(new,ll3);           ==> {a,new}
```

vii. 関数 ASFIRST, ASLAST, ASREST, ASFLIST, ASSLIST, RESTASLIST は ALIST あるいはリストを要素とするリストに対する関数です。ASFIRST はリスト中で、最初の要素が第一引数に等しいペアを返す。ASLAST はリスト中で、後ろの要素が第一引数に等しいペアを返す。ASREST は第一引数としてリストを取り、第二引数のリスト中の要素のなかで、そのサブリストが第一引数に等しいペアを返す。RESTASLIST は第一引数で指定された**キーのリスト**を元にして、リスト中で最初の要素がキーに一致するペアの後ろの要素をすべてリストにして返す。ASFLIST はリストの要素であるペアの内で最初の要素が第一引数に等しい全てのペアを返す。ASSLIST はリストの要素であるペアの内で後ろの要素が第一引数に等しい全てのペアを返す。

```
lp:={{a,1},{b,2},{c,3}}$
ASFIRST(a,lp);              ==> {a,1}
ASLAST(1,lp);               ==> {a,1}
ASREST({1},lp);             ==> {a,1}
RESTASLIST({a,b},lp);       ==> {{1},{2}}
lpp:=APPEND(lp,lp)$
ASFLIST(a,lpp);             ==> {{a,1},{a,1}}
ASSLIST(1,lpp);             ==> {{a,1},{a,1}}
```

3.3 Bag 構造と関連する関数

REDUCE のリスト構造は非常に便利です。しかし、リストは環としての演算を百合指定内ため、多項式の未知変数としては許されません。また、多くの関数の引数としては許されていません。

BAG 型はリストとオペレータの中間の構造です。オペレータのようにカーネルとして利用でき、またリストのように複合した構造を持ちます。

定義:

bag は次の性質を持つオブジェクトです。

1. 原子的な前置詞 (envelope) と内容から構成されています。
2. 内容はリストと同様にそれを変更することができます。
3. envelope に属性を持たすことができます。例えば、NONCOM あるいは SYMMETRIC 等として宣言することが可能です。

利用可能な関数:

i. 標準の bag envelop としてBAG が定義されています。これは予約語です。LIST あるいはブール値関数としての名前以外の識別子に対して、PUTBAG を使って bag envelope として宣言することができます。特に、任意の演算子を bag として宣言することができます。識別子がすでに関数として宣言されていない場合、そしてその場合のみ、PUTBAG は前置演算子としての属性を識別子に対して付加します。

```
PUTBAG id1,id2,....idn;
```

はid1,.....,idn を bag envelopes として宣言します。同様に、コマンド

```
CLEARBAG id1,...idn;
```

は、id1,...,idn に対する bag 宣言を取り消します。

ii. ブール値関数 BAGP は bag 型であるかを判定します。

```
aa:=bag(x,y,z)$
if BAGP aa then "ok";     ==> ok
```

iii. 上で定義したほとんどのリスト操作関数は bag に対しても使えます。更に、後で説明する集合に対して定義された関数もまた利用可能です。(節 3.4 を参照のこと)。しかしながら、envelope を保存する必要から、少し動作が異なります。

```
PUTBAG op;            ==> T
aa:=op(x,y,z)$
FIRST op(x,y,z);      ==> op(x)
REST op(x,y,z);       ==> op(y,z)
BELAST op(x,y,z);     ==> op(x,y)
APPEND(aa,aa);        ==> op(x,y,z,x,y,z)
LENGTH aa;            ==> 3
DEPTH aa;             ==> 1
```

もし、異なった envelope を持つ二つの bag を結合した場合、結果の bag の envelpe には最初の引数のものが使われます。関数 LENGTH は演算子が依存している実際の変数の数を返します。関数 FIRST, SECOND, LAST および BELAST では envelope の名前は保存されます。

iv. KERNLIST は bag をリストに変換します。LISTBAG はリストを bag 型に変換します。

```
LISTBAG(<list>,<id>); ==> <id>(<arg_list>)
```

識別子<id> は自動的に bag の envelope として宣言されます。

BAGLISTP は引数が bag あるいはリストであれば t を返します。ABAGLISTP はリストを要素とするリストであるかあるいは bag を要素とする bag の場合に t を返します。

3.4 集合とその関数

通常のリスプでの集合演算と同様の機能を代数モードで提供します。

i. 関数 MKSET は、重複した要素を取り除くことによって、リストあるいは bag を集合に変換します。

```
MKSET({1,a,a1});       ==> {1,a}
MKSET bag(1,a,a1);     ==> bag(1,a)
```

SETP は引数が集合であるかを判定します。

ii. 標準的な関数 UNION, INTERSECT, DIFFSET および SYMDIFF があります。

3.5 一般目的のユーティリティ

i. 関数 MKIDNEW, DELLASTDIGIT, DETIDNUM, LIST_TO_IDS は識別子に関する処理を行ないます。MKIDNEW は MKID の変形です。

MKIDNEW は 0 もしくは 1 個の引数をとります。これは、まだ使われたことの無い識別子を作ります。

```
MKIDNEW(); ==> g0001
MKIDNEW(a); ==> ag0002
```

DELLASTDIGIT は整数を引数にとります。これは、引数の数から最後の数字を取り除きます。

```
DELLASTDIGIT 45; ==> 4
```

DETIDNUM は、識別子名から文字の後に続く数値を取り出します。これは、識別子の集合 $a_1, \ldots, a_n$ に対する繰り返しを処理する場合に便利です。

```
DETIDNUM a23; ==> 23
```

LIST_to_IDS は関数 MKID をリストに拡張したもので、要素のアトムを結合した識別子名を作成します。最初の要素は数字であってはいけません。

```
LIST_TO_IDS {a,1,id,10}; ==> a1id10
```

関数 ODDP は奇整数を判定します。

関数 FOLLOWLINE は、関数 PRIN2 を使って出力を行なう場合、出力の清書を行なうのに便利です。

```
        <<prin2 2; prin2 5>>$
25
        <<prin2 2; followline(3); prin2 5>>$
2
   5
```

関数 RANDOMLIST は正の乱数からなるリストを生成します。

```
        RANDOMLIST(10,5); ==> {2,1,3,9,6}
```

MKRANDTABL は、乱数の表を作成します。第一引数は配列の次元で、一次元あるいは二次元の配列のいずれかです。乱数の基数は整数あるいは小数点数です。小数点数を指定する場合は、スイッチ rounded をオンにしておく必要があります。最後の引数は、結果を格納する配列の名前として使われます。

```
        MKRANDTABL({3,4},10,ar); ==>
               *** array ar redefined
                     {3,4}
```

COMBNUM(n,p) は、n 個から p 個を取り出す組合せの数を求めます。

PERMUTATIONS(n) は、n 個のオブジェクトから、全ての順列生成します。CYCLICPERMLIST は、循環列を生成します。これらの関数については、引数は bag 型でも構いません。

```
        PERMUTATIONS {1,2} ==> {{1,2},{2,1}}
        CYCLICPERMLIST {1,2,3} ==>
                {{1,2,3},{2,3,1},{3,1,2}}
```

COMBINATIONS は、n 個のオブジェクトから p 個を取り出す、組合せ全てのリストを求めます。

```
        COMBINATIONS({1,2,3},2) ==> {{2,3},{1,3},{1,2}}
```

REMSYM は、REDUCE コマンド symmetric あるいは antisymmetric による宣言を取り消します。

SYMMETRIZE は対称式を生成する強力な関数です。これは三つの引数をとります。第一引数はリスト (あるいはリストを要素とするリスト) で、第二引数はカーネルで、第三引数は順列を生成する関数名です。第二引数で指定される名前の演算子の引数として、順列関数で生成された引数を持つ演算子の対称な和を計算します。このパッケージでは、PERMUTATIONS および CYCLICPERMLIST が使われるでしょう。

```
        ll:={a,b,c}$
        SYMMETRIZE(ll,op,cyclicpermlist); ==>
                OP(A,B,C) + OP(B,C,A) + OP(C,A,B)
        SYMMETRIZE(list ll,op,cyclicpermlist); ==>
                OP({A,B,C}) + OP({B,C,A}) + OP({C,A,B})
```

```
op(a,b,c):=a*b*c$
SYMMETRIZE(ll,op,cyclicpermlist); ==>
         OP(A,B,C) + OP(B,C,A) + OP(C,A,B)
for all x let op(x,a,b)=sin(x*a*b);
SYMMETRIZE(ll,op,cyclicpermlist); ==>
          OP(B,C,A) + SIN(A*B*C) + OP(A,B,C)
```

関数 SORTNUMLIST および SORTLIST はリストを並べ変えます。これらの関数は、*bubblesort* および *quicksort* をそれぞれ使って整列します。

SORTNUMLIST は、数字を要素とするリストに対して、整列したリストを返します。

SORTLIST は、上記の関数の拡張です。指定された関数に基づいて、リストの要素を並べ変えます。

```
l:={1,3,4,0}$  SORTNUMLIST l;      ==> {0,1,3,4}
ll:={1,a,tt,z}$ SORTLIST(ll,ordp); ==> {a,z,tt,1}
```

これらの関数は、リストを壊してしまうので、カーネルや bag 型に作用させるのは危険です。もし必要なら、最初に KERNLIST を使うことをお進めします。

EXTREMUM は、関数 MIN や MAX を任意の順序に対しても使えるように拡張したものです。指定した関数に関して、一番最初に来る要素を求めます。

```
EXTREMUM(ll,ordp); ==> 1
```

iii. 依存関係を特定するための関数として四つの関数が定義されています。FUNCVAR は、任意の式から、その式が依存している変数のリストを返します。

```
FUNCVAR(e+pi+sin(log(y)); ==> {y}
```

DEPATOM は、アトムに対してそれと依存関係にあると宣言された変数のリストを返します。

```
DEPEND a,x,y;
DEPATOM a;       ==> {x,y}
```

関数 EXPLICIT および IMPLICIT は、明示的なあるいは暗黙の依存関係を作ります。

```
depend a,x; depend x,y,z;
EXPLICIT a;      ==> a(x(y,z))
IMPLICIT ws;     ==> a
```

これらの関数は、依存している変数名やその特性を知りたい場合に有用です。

KORDERLIST は、現在宣言されているカーネルの順序を返します。

```
KORDER x,y,z;
KORDERLIST;      ==> (x,y,z)
```

iv. 関数 SIMPLIFY は下向きに簡約した結果を返します。これは SYMMETRIZE と共に使うと便利です。また、EXCALC パッケージ (16 章) のある出力を簡約する時にも使えます。

```
l:=op(x,y,z)$
op(x,y,z):=x*y*z$
SYMMETRIZE(l,op,cyclicpermlist); ==>
                        op(x,y,z)+op(y,z,x)+op(z,x,y)
SIMPLIFY ws;       ==> op(y,z,x)+op(z,x,y)+x*y*z
```

v. リストに対するフィルター関数。

CHECKPROLIST はリストの要素がある指定した性質を持っているか判定します。第二引数には、ブール値関数 (FIXP NUMBERP ...) あるいは順序付けの関数 (ORDP) を指定します。

EXTRACTLIST はリストから、第二引数で指定した述語関数が真となる要素を取り出します。

```
l:={1,a,b,"st")$
EXTRACTLIST(l,fixp);    ==> {1}
EXTRACTLIST(l,stringp); ==> {st}
```

3.6 属性とフラグ

代数モードでのフラグや属性リストをサポートする関数です。

i. フラグ 関数 PUTFLAG の第一引数は、識別子あるいは識別子のリストで、第二引数はフラグの名前です。第三引数は真に対しては T、偽に対しては 0 を指定します。第三引数が T であれば、フラグを立てます。0 であればフラグを削除します。

```
PUTFLAG(z1,flag_name,t);        ==> flag_name
PUTFLAG({z1,z2},flag1_name,t); ==> t
PUTFLAG(z2,flag1_name,0);       ==>
```

DISPLAYFLAG は識別子に対して、現在設定されているフラグのリストを出力します。

```
DISPLAYFLAG z1;    ==> {flag_name,flag1_name}
DISPLAYFLAG z2;    ==> {}
```

CLEARFLAG は識別子 $id_1, \ldots, id_n$ に対して全てのフラグを削除します。

ii. 属性 PUTPROP は四つの引数をとります。第二引数は属性の指示子で、第三引数は任意の式です。第四引数が T であれば、属性を付けます。0 であれば指定された属性を削除します。

```
PUTPROP(z1,property,x^2,t); ==> z1
```

より一般的な使用法は、

```
PUTPROP(LIST(idp1,idp2,..),<propname>,<value>,T);
```

です。DISPLAYPROP は指定された識別子と指示子に関して、設定されている属性値を取り出します。

```
                                                    2
DISPLAYPROP(z1,property); ==> {property,x  }
```

CLEARPROP は指定された識別子に設定されている属性を全て削除します。

3.7 制御関数

利用者による環境の制御を改善する関数が用意されています。

i. 最初の

```
ALATOMP x;     x は任意.
ALKERNP x;     x は任意.
DEPVARP(x,v); x は任意.
               (v はアトムあるいはカーネル)
```

ALATOMP は、x が評価された後で、整数あるいは識別子になる場合 T を返します。

ALKERNP は、x が評価された後で、カーネルである場合に T を返します。

DEPVARP は、x が v に依存している場合に T を返します。

以上の関数に合わせて、PRECP とオペレータ関数を宣言することによりこれらの値の判定を容易にします。

NORDP は条件式の中では、本質的に ORDP の否定と同じです。しかしながら、場合によって ORDP の否定がが使用できないところででも使用することができます。

ii. 次の関数は、利用者が変更した REDUCE の環境を解析したり、元に戻したりするのに役立ちます。

SHOW は識別子に対してその値や型を見るのに使います。SUPPRESS は、識別子に対して選択して元に戻したり、全てを元に戻します。ユーザによって入力されて定義されり、ファイルから読み込まれた定義は無視するようになっています。

```
SHOW (SUPPRESS) all
SHOW (SUPPRESS) scalars
SHOW (SUPPRESS) lists
SHOW (SUPPRESS) saveids    (for saved expressions)
SHOW (SUPPRESS) matrices
SHOW (SUPPRESS) arrays
SHOW (SUPPRESS) vectors
                (contains vector, index and tvector)
SHOW (SUPPRESS) forms
```

オプション all は SHOW に対しては最も便利です。しかし、特に何時間にも渡って計算を実行した後では、結果を得るのに時間がかかるかもしれません。REDUCE を最初に起動した状態では、SHOW を all オプションを付けて実行すると以下の結果が得られます。

```
SHOW all;          ==> scalars are: NIL
                       arrays are: NIL
                       lists are: NIL
                       matrices are: NIL
                       vectors are: NIL
                       forms are: NIL
```

幾つかのオプションを覚えておくと便利です。まったく新しく REDUCE のセッションを開始したあとでは、

```
a:=b:=1$
SHOW scalars;      ==>  scalars are: (A B)
SUPPRESS scalars; ==> t
SHOW scalars;      ==>  scalars are: NIL
```

のような結果が得られます。

iii. CLEAR 関数はOPERATORS や FUNCTIONS に関して完全な消去を行ないません。次の二つの関数は、PUTFLAG や PUTPROP によって利用者が変更したフラグや属性も考慮して、もっと完全な消去を行ないます。

```
CLEARFUNCTIONS a1,a2, ... , an $
```

は名前 a1,a2, ... ,an で定義されている関数を消去します。CLEAROP はオペレータを消去します。lose フラグで保護されている関数以外はすべてこの機能で関数定義を消去することができてしまうので、これらの機能を使用する場合は十分に注意が必要です。

3.8 多項式の操作

このモジュールは、標準商形式を扱うためのユーティリティ関数と、多項式を操作するための幾つかの新しい機能を提供する。

i. 二つの関数 ALG_TO_SYMB と SYMB_TO_ALG は代数モードでの標準商形式とリスプでの前置形式との変換を行なう。

```
algebraic procedure ecrase x;
lisp symb_to_alg flattens1 alg_to_symb algebraic x;

 symbolic procedure flattens1 x;
 % ll; ==> ((A B) ((C D) E))
```

```
% flattens1 ll; (A B C D E)
  if atom x then list x else
  if cdr x then
      append(flattens1 car x, flattens1 cdr x)
    else flattens1 car x;
```

は次のよう処理を行なう。

```
ll:={a,{b,{c},d,e},{{{z}}}}$
ECRASE ll; ==> {A, B, C, D, E, Z}
```

ii. LEADTERM および REDEXPR は、記号モードでの関数 LT と RED と同じ機能を代数モードで提供する。多項式の、**主項** と**残り** をそれぞれ返します。これらの関数は有理式に対しても機能します。主変数は、現在の変数の順序にしたがって決定されます。

```
pol:=x+y+z$
LEADTERM pol; ==> x
korder y,x,z;
LEADTERM pol; ==> y
REDEXPR pol; ==> x + z
```

標準の状態では、多変数多項式は再帰的に表現されています。このような表現では、関数 LEADTERM は真の単項式にはならず、主変数の係数 (これはまた、主変数以外の変数に対する多項式の可能性がある) を返します。真の単項式を扱いたい場合がありますが、このような場合には、**分散形式**で多項式を扱う必要があります。このような形式は GROEBNER (22 章) パッケージで提供されています。このような**分散形式**の多項式を操作するために新しいスイッチ DISTRIBUTE とを用意しています。また、新しい関数 DISTRIBUTE が定義されています。この関数は多項式を分散形式に変換します。このスイッチをオンにすれば、LEADTERM は真の単項式を返します。

MONOM は多項式を単項式のリストの形に変換します。これは、スイッチ DISTRIBUTE の設定に関わらず、正しい結果を与えます。

SPLITTERMS は MONOM とは二つのリストからなるリストを返します。最初のリストは、正の項のみのリストで、二番目は負の項からなるリストです。

SPLITPLUSMINUS は多項式の正部分と負部分に分解したリストを返します。

iii. LOWESTDEG は、多項式と未知変数に対して、未知変数に対して最低次数を返します。DIVPOL は二つの多項式に対して、商と余りを返します。

3.9 超越関数の操作

関数 TRIGREDUCE および TRIGEXPAND それと同様な双曲線関数に対する関数 HYPREDUCE および HYPEXPAND は、関数の和公式あるいは積公式を使って変換を行ないます。

```
aa:=sin(x+y)$
TRIGEXPAND aa; ==> SIN(X)*COS(Y) + SIN(Y)*COS(X)
TRIGREDUCE ws; ==> SIN(Y + X)
```

三角関数や双曲線関数の式がSIN (SINH) と COS (COSH) に対して対称な場合、TRIG(HYP)REDUCE を適用させると大幅に簡約される可能性があります。しかしながら、極端に非対称な式に対しては、TRIG(HYP)EXPAND を行なって、TRIG(HYP)REDUCE を作用させると、結果として得られる式はより対称的ではあるが複雑な式になります。

```
aa:=(sin(x)^2+cos(x)^2)^3$
TRIGREDUCE aa; ==> 1
bb:=1+sin(x)^3$
TRIGREDUCE bb; ==>
        - SIN(3*X) + 3*SIN(X) + 4
       ---------------------------
                   4

 TRIGEXPAND ws; ==>
        3                   2
  SIN(X)  - 3*SIN(X)*COS(X)  + 3*SIN(X) + 4
  ----------------------------------------
                     4
```

TRIGSIMP パッケージ (59 章) も参照のこと.

3.10 リストと配列間の変換

LIST_TO_ARRAY は三つの引数をとる。第一引数はリストで、第二引数は配列の次元を示す整数値である。第三引数は識別子名で、生成する配列名となる。もし、次元がリストの長さと整合していなければエラーを起こす。

ARRAY_TO_LIST は引数として与えられた配列をリストに変換する。

3.11 n–次元ベクトルの操作

ユークリッド空間は深さ 1 のリストあるいは Bag で表すことができる。要素はリストではなく、Bag である。このようなベクトルに対して、和、差、スカラー積を計算する関数が用意されている。もし、空間次元が 3 であれば、クロスおよびミックスド積を行なう関数が用意されている。

```
l:={1,2,3}$
ll:=list(a,b,c)$
SUMVECT(l,ll);    ==> {A + 1,B + 2,C + 3}
MINVECT(l,ll);    ==> { - A + 1, - B + 2, - C + 3}
```

```
SCALVECT(l,ll);   ==> A + 2*B + 3*C
CROSSVECT(l,ll);  ==> { - 3*B + 2*C,3*A - C, - 2*A + B}
MPVECT(l,ll,l);   ==> 0
```

3.12 グラスマン演算子の操作

グラスマン変数は、積に対して結合則と分配則が成り立つが、反可換である。

PUTGRASS は n 項のコマンドで、識別子に対してグラスマンカーネルとしての属性を付ける。REMGRASS はこの属性を取り除く。

GRASSP は、グラスマンカーネルであるかを判定するブール値関数。

GRASSPARITY は単項式にたいしてそのパリティを計算する。もし、引数が単純なグラスマンカーネルであれば 1 になる。

GHOSTFACTOR は単項式を二つ引数にとる。これは、

```
(-1)**(GRASSPARITY u * GRASSPARITY v)
```

に等しい。

次にこれらの関数の例を示す。

```
PUTGRASS eta;
if GRASSP eta(1) then "Grassmann kernel"; ==>
                                              Grassmann kernel
aa:=eta(1)*eta(2)-eta(2)*eta(1);       ==>
                      AA :=  - ETA(2)*ETA(1) + ETA(1)*ETA(2)
GRASSPARITY eta(1);                    ==> 1
GRASSPARITY (eta(1)*eta(2));           ==> 0
GHOSTFACTOR(eta(1),eta(2));            ==> -1
grasskernel:=
  {eta(~x)*eta(~y) => -eta y * eta x when nordp(x,y),
  (~x)*(~x) => 0 when grassp x}$
exp:=eta(1)^2$
exp where grasskernel;                 ==> 0
aa where grasskernel;                  ==>  - 2*ETA(2)*ETA(1)
```

3.13 行列の操作

行列を操作する機能が追加されています。

i. UNITMAT は単位行列を作成する。

```
UNITMAT M1(n1), M2(n2), .....Mi(ni) ;
```

ここで、M1,...Mi は行列名。 n1, n2, ..., ni は整数。

MKIDM は MKID の拡張で、行列名を作る。例えば、u と u1 が行列とすると、

```
matrix u(2,2);$  unitmat u1(2)$
u1; ==>
        [1  0]
        [    ]
        [0  1]
```

```
mkidm(u,1); ==>
        [1  0]
        [    ]
        [0  1]
```

注意: MKIDM(V,1) はたとえ行列 V1 が存在したとしても、V が行列として宣言されていない場合にはエラーを起こします。

COERCEMAT は第一引数で与えられた行列をリストのリストに変換する。

```
COERCEMAT(U,id)
```

id が list であれば、リストを要素とするリストに変換する。それ以外の場合は、Bag を要素とする Bag に変換する。

BAGLMAT はこの逆を行なう。

```
BAGLMAT(bgl,U)
```

bgl は行列U に変換される。もしU がすでに行列として宣言されていればこの変換は行なわれない。

ii. 関数 SUBMAT, MATEXTR, MATEXTC は行列の一部を取り出します。

```
SUBMAT(U,nr,nc)
```

U は行列名、nr と nc は行と列の番号。U から行 nr 列 nc を取り除いた余因子行列を求める。もし、行番号あるいは列番号が 0 であれば、行あるいは列は削除されない。

MATEXTR と MATEXTC は行あるいは列を取り出し、リストあるいは Bag として出力する。

```
MATEXTR(U,VN,nr)
MATEXTC(U,VN,nc)
```

U は行列, VN は “ベクトル名”, nr と nc は整数. もし VN が list であればリスト形式で出力される。それ以外は Bag 形式で出力される。

iii. 行列を操作する関数: MATSUBR, MATSUBC, HCONCMAT, VCONCMAT, TPMAT, HERMAT.

MATSUBR あるいは MATSUBC は行あるいは列を引き算する。

```
MATSUBR(U,bgl,nr)
MATSUBC(U,bgl,nc)
```

U は行列, nr と nc は整数. bgl はリストあるいは Bag ベクトル。このサイズは行列の次元と整合していなければならない。

HCONCMAT あるいは VCONCMAT は二つの行列を結合する。

```
HCONCMAT(U,V)
VCONCMAT(U,V)
```

HCONCMAT は水平方向に、VCONCMAT は垂直方向に結合する。

TPMAT は行列のテンソル積を計算する。

```
TPMAT(U,V) あるいは U TPMAT V
```

HERMAT は行列のエルミート共役を計算する。

```
HERMAT(U,HU)
```

iv. SETELMAT および GETELMAT は二つ整数を引数にとる。SETELMAT は要素(i,j) をリセットし、GETELMAT は(i,j) で示される要素を取り出す。関数中で行列を扱う場合に有用である。

第4章　AVECTOR:ベクトルパッケージ

Dr. David Harper
Astronomy Unit
Queen Mary and Westfield College
University of London
Mile End Road
London E1 4NS
England
Electronic mail: *adh@star.qmw.ac.uk*

このパッケージでは、ベクトルの演算 (代数計算、微積、ベクトル積や内積) 等の計算ができます。ベクトル解析の計算では、直交座標以外に円筒座標や球座標での計算ができます。 また、これらの座標以外に、利用者が定義した座標系での計算もできます。

このパッケージを使うには、最初に

```
load avector;
```

と、ロードコマンドによって AVECTOR パッケージをロードしておかないといけません。変数名に対してそれをベクトルとして宣言するには `VEC` コマンドで、例えば、

```
vec a,b,c;
```

のようにします。このように入力すると、以後 `A,B,C` はベクトルとして扱われます。

ベクトルとして宣言しただけでは初期値は与えられていません。値を代入するには、$AVEC(x, y, z)$ という関数を使います。例えば、

```
a := avec(a1,a2,a3);
```

とすると、ベクトル **A** の各要素には、$A1, A2, A3$ が代入されます。

4.1 ベクトル代数

スカラーとベクトルとの四則演算が+,-,*,/で行なえます。(以下では、s,s1,s2,s3,... はスカラーを、また v,v1,v2,v3,... はベクトルを表すものとします。) 例えば、

```
V := V1 + V2 - V3;    和と差
V := S1*3*V1;         スカラー積
V := V1/S;            スカラーによる割り算
V := -V1;             Negation
```

ベクトルの積については、内積と外積 (ベクトル積) がありますが、これはそれぞれ DOT と CROSS というオペレーターで行ないます。これらのオペレーターはスカラー積や商よりも高い演算順序を持っています。例えば、

```
V := V1 CROSS V2;          外積
S := V1 DOT V2;            内積
V := V1 CROSS V2 + V3;
V := (V1 CROSS V2) + V3;
```

この最後の二つの例は同じ結果を与えます。ベクトルの束は、VMOD オペレーターで計算できます。例えば、

```
S := VMOD V;
V1 := V/(VMOD V);
```

とすると、V1 には、単位ベクトルが代入されます。

また、ベクトルの微分や積分を、DF や INT オペレーターで計算できます。 DF または INT オペレーターは、ベクトルの各要素について、微分もしくは積分した結果を返します。

例えば、

```
V := AVEC(X**2, SIN(X), Y);

DF(V,X);
```

とすると、

$$(2 * X, COS(X), 0)$$

と出力されます。また、

```
INT(V,X);
```

では、

$$(X**3/3, -COS(X), Y*X)$$

となります。

4.2 ベクトル解析

ベクトル解析のオペレーターとして、DIV,GRAD と CURL が使えます。またスカラー場及びベクトル場に対するラプラス演算子 DELSQ が使えます。これらは、次のようなベクトル解析の計算を行います。

`V := GRAD S;`	グラジェント
`S := DIV V;`	ベクトル場の発散
`V := CURL V1;`	ベクトル場のローテーション
`S := DELSQ S1;`	スカラー場のラプラシャン
`V := DELSQ V1;`	ベクトル場のラプラシャン

例題として、

$$\text{curl curlA} = \text{grad divA} - \nabla^2\text{A}$$

が成り立つことを示すには、

```
load avector;
vec a;
a := avec(ax,ay,az);
depend ax,x,y,z;
depend ay,x,y,z;
depend az,x,y,z;
curl curl a - grad div a - delsq a;
```

を計算してみれば、結果がゼロになることで示せます。ここで、DEPEND を使ってベクトル A の成分 ax, ay, az が x, y, z の関数であることを REDUCE に教えています。

これらの演算は、任意の曲座標系のベクトルに対して使うことができます。最初の状態では (X,Y,Z)-座標系で、スケール因子はすべて 1 となっています。これを例えば、(R,THETA,PHI)-の極座標系に変えるには、

```
COORDINATES R,THETA,PHI;
SCALEFACTORS(1,R,R*SIN(THETA));
```

と入力すればできます。 このように定義した座標系に名前を付けて、例えば

```
PUTCSYSTEM 'spherical;
```

と宣言しておくと、以後

```
GETCSYSTEM 'spherical;
```

と入力することで、spherical 座標系での計算が行なえます。システムには、すでにカルタン座標系 (CARTESIAN、直交座標)、球座標系 (SPHERICAL) と円筒座標系 (CYLINDRICAL) という座標系が定義されていますので、例えば、球座標系での計算を行なうには、

```
GETCSYSTEM 'SPHERICAL;
```

と入力すれば出来ます。通常は、(X,Y,Z) のカルタン座標系になっています。 この状態に戻すには、

```
GETCSYSTEM 'CARTESIAN;
```

と入力すればよい。

4.3 体積積分と線積分

体積積分と線積分を計算するための関数が用意されています。スカラーとベクトルの定積分を計算するには、DEFINT という関数を使います。例えば、$\sin^2(x)$ を 0 から 2π まで積分するには、

```
DEFINT(sin(x)**2,x,0,2*pi);
```

と入力すれば計算できます。

体積積分を計算する関数は VOLINT です。この関数の構文は

```
VOLINT(式, 下限ベクトル, 上限ベクトル);
```

です。積分の上限と下限はベクトルで与えます。例えば、球の体積を計算するには、極座標系で 1 を $0 \leq r \leq RR,\ 0 \leq \theta \leq \pi, 0 \leq \phi \leq 2\pi$ の範囲で積分すれば得られます。これには、次のように入力します。

```
VLB := AVEC(0,0,0);            % r,theta,phi の下限;
VUB := AVEC(RR,PI,2*PI);       % 上限;
VOLINTORDER := AVEC(0,1,2);    % 積分の順序;
VOLINT(1,VLB,VUB);
```

ここで、VOLINTORDER という関数で積分を実行する順序を指定しています。上の例では r, θ, ϕ の順で積分することを指定しています。

線積分は LINEINT という関数で計算できます。この関数のシンタックスは、

```
LINEINT(ベクトル式, 経路, 変数名);
DEFLINENINT(ベクトル式, 経路, 変数名, 下限, 上限);
```

です。経路は積分の経路を指定するベクトル式です。例えば、球座標系で一定の傾き lat を持った曲線を積分するには、(0,0,1) をカーブ $(0, lat, \phi)$ に添って積分すればよい。つまり、

```
DEFLINEINT(AVEC(0,0,1),AVEC(0,LAT,PHI),PHI,0,2*PI);
```

で求められます。

第5章　BOOLEAN: ブール代数式の計算

H. Melenk

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Heilbronner Strasse 10
D–10711 Berlin – Wilmersdorf
Federal Republic of Germany

E–mail: *melenk@sc.zib–berlin.de*

5.1　初めに

Boolean パッケージは、ブール代数の計算を行います。データは、代数式 (“原子 (atomic parts)” もしくは “枝 (leafs)”) を、内挿ブール演算子 **and**, **or**, **implies**, **equiv**　と、単項の前置演算子 **not** を組み合わせたものからなっています。 **Boolean** では、これらの演算子から構成される式を 簡約したり、同値かどうかをテストしたり、サブセットになっているかを調べたりすることが可能です。

5.2　ブール式の入力

ブール代数式と、REDUCE の条件式 (例えば、**if** 文等) を区別するため、それぞれのブール代数式には、前に **boolean** を付けます。これ以外の式はブール代数式として入力されません。**boolean** の第一引数は、任意のブール代数式です。この式には、別のブール代数式への参照を含んでいても構いません。例えば、

```
boolean (a and b or c);
q := boolean(a and b implies c);
boolean(q or not c);
```

演算の優先順位から、括弧を付ける必要があります。ブール式の枝もしくはアトムは、式において前に **boolean** タグが付いていない部分です。これらは、ブール式を評価するときには、定数であるとして扱われます。二つの前もって定義された定数があります。

- **true**, **t** または **1**
- **false**, **nil** または **0**

これらは、ブール定数を表します。出力の中では、**1** と **0** の値のみが使われます。

通常、**boolean** 式は加法標準型に変換されます。これは、それぞれの項が、枝もしくは枝の前に **not** が付いた要素を **and** でつなげたもので、各項を **or** で結合した形式です。要素が一つの項や、一つの項からなる式では、**and** や **or** は省略されます。変換した結果は、やはり前に **boolean** のタグを付けた式として返します。ブール定数の **0** と **1** だけは、このようなタグが付けられずに返されます。

出力では、 **and** と **or** はそれぞれ、/\ と \/ で表されます。

```
boolean(true and false);    ->   0
boolean(a or not(b and c)); -> boolean(not(b) \/ not(c) \/ a)
boolean(a equiv not c);     -> boolean(not(a)/\c \/ a/\not(c))
```

5.3 標準型

標準型として、加法標準型 (**disjunctive** normal form) が通常使われます。これに対して、乗法標準型 (**conjucive** normal form) を使うこともできます。これには、**boolean** の第二引数として、**and** キーワードを指定します。

```
boolean (a or b implies c);
                    ->
     boolean(not(a)/\not(b) \/ c)

boolean (a or b implies c, and);
                    ->
     boolean((not(a) \/ c)/\(not(b) \/ c))
```

結果は、完全に簡約された加法標準型もしくは乗法標準型で、すべての冗長な要素は次の規則によって取り除かれます。

$a \wedge b \vee \neg a \wedge b \longleftrightarrow b$

$a \vee b \wedge \neg a \vee b \longleftrightarrow b$

内部では、完全な標準型が中間式として計算されます。この形では、各々の項はすべての “枝” 式を、各々一つ含んでいます。この形式での標準型を返したい場合には、キーワード **full** をつけ加えて下さい。

```
boolean (a or b implies c, full);
                    ->
boolean(a/\b/\c \/ a/\not(b)/\c \/ not(a)/\b/\c \/ not(a)/\not(b)/\c

         \/ not(a)/\not(b)/\not(c))
```

キーワード **full** と **and** を同時に指定することもできます。

5.4 ブール式の評価

もし、ブール式の枝が代数式であれば、環境が変わる度に (例えば変数の値が変化したような場合) testbool 演算子を使って、式を 再評価する事が出来ます。この演算子は、すべての枝式を REDUCE の論理式の形で計算し直します。可能な限り、各項は、ブール値で表されます。表せない項は、そのまま変わらずに残されます。結果の式は、最小な形に簡約されます。完全に簡約されてしまうと、結果は **1**(真) または **0**(偽) のいずれかの結果が得られます。

次の例では、枝は数値の大なり記号で構成されています。ここで、> を式中で使うためには、まずオペレータとして宣言しておく必要があります。エラーメッセージが出力されますが、これは意味がありません。

```
operator >;
fm:=boolean(x>v or not (u>v));
        ->
    fm := boolean(not(u>v) \/ x>v)

v:=10$

testbool fm;

   ***** u - 10 invalid as number
  (***** u - 10 は数値ではありません)
   ***** x - 10 invalid as number
  (***** x - 10 は数値ではありません)

        ->
   boolean(not(u>10) \/ x>10)

x:=3$
testbool fm;

   ***** u - 10 invalid as number
  (***** u - 10 は数値ではありません)

        ->
   boolean(not(u>10))

x:=17$
```

```
testbool fm;

   ***** u - 10 invalid as number
  (***** u - 10 は数値ではありません)

        ->
    1
```

第6章　CALI: 可換代数演算

Hans-Gert Gräbe
Institut für Informatik, Universität Leipzig
Augustusplatz 10 – 11
04109 Leipzig, Germany
e–mail: *graebe@informatik.uni-leipzig.de*

このパッケージでは、可換代数に対するアルゴリズムが含まれています。これはイデアルと束についての Gröbner アルゴリズムと密接に関連しています。一番主要な点は、syzygy の計算ができるように変更された新しい Gröbner のアルゴリズムをインプリメントしていることです。このアルゴリズムは生成元が行列の列であるような自由束の部分束に対しても適用可能です。

CALI の主な機能は

- 環、イデアルおよび束の定義
- Gröbner 基底ならびに局所標準基底の計算
- syzygy, resolutions および (graded) Betti 数の計算
- Hilbert 級数、重複度、独立集合および次元の計算
- 正規形式と正規表現の計算
- 和、積、共通部分、商、stable 商、除去イデアル等の計算
- イデアルや束の既約性のテスト、ラディカルの計算、既約分解等
- Gröbner 基底のより進んだ応用 (blowup, associated graded ring, analytic spread, symmetric algebra, monomial curves 等)
- 項順序の FGLM 変換、点集合のアフィンおよび射影イデアル等、ゼロ次元イデアルへの線形代数の応用。
- 因数分解と Gröbner アルゴリズムを組み合わせた多項式方程式系の分解、三角化、拡張された Gröbner 分解の別の方法

もっと詳しいマニュアルがべつにあります。

第7章　CAMAL: 天体力学の計算

J. P. Fitch
School of Mathematical Sciences, University of Bath
BATH BA2 7AY, England
e–mail: *jpff@maths.bath.ac.uk*

CAMAL パッケージは 1970 年代の天体力学の計算、特にケンブリッジ代数システム、が行なうフーリエ級数の演算機能を提供します。

7.1　フーリエ級数の演算

7.1.1　HARMONIC

天体力学では多項式の変数と角変数を区別します。角変数は HARMONIC 関数で前もって宣言しておく必要があります。

```
harmonic theta, phi;
```

7.1.2　FOURIER

関数 FOURIER はその引数をフーリエ級数に変換します。引数の式には調和変数に関する**正弦**や**余弦**関数の項の線形和が含まれていても構いません。

```
fourier sin(theta)
```

フーリエ式に対しては通常の REDUCE のやり方で加算、引算、乗算および微分演算が可能です。乗算の場合には、角部分の積の線形化が自動的に行なわれます。

三つの関数があります。これらは通常の限定された調和微分、調和積分それに調和代入を行なうものです。

7.1.3　HDIFF と HINT

フーリエ式を角変数に関して微分あるいは積分する。積分における全てのセキュラー項は無視されます。

```
load_package camal;
harmonic u;
bige := fourier (sin(u) + cos(2*u));
aa := fourier 1+hdiff(bige,u);
ff := hint(aa*aa*fourier cc,u);
```

7.1.4 HSUB

角度に角度とフーリエ式の和を代入し、ある指定した次数までの展開式をもとめる次の演算は調和代入と呼ばれている。関数は五つの引数をとります。基礎方程式、置き換えようとする角、置換式の角度部分、置換式のフーリエ部分それに展開する次数です。

```
harmonic u,v,w,x,y,z;
xx:=hsub(fourier((1-d*d)*cos(u)),u,u-v+w-x-y+z,yy,n);
```

7.2 簡単な例題

次のプログラムはケプラー方程式を解いて、フーリエ級数で n 次までの解を求める。

```
bige := fourier 0;
for k:=1:n do <<
  wtlevel k;
  bige:=fourier e * hsub(fourier(sin u), u, u, bige, k);
>>;
write "Kepler Eqn solution:", bige$
```

第8章　CHANGEVR: 微分方程式の独立変数の変換

G. Üçoluk
Department of Physics,
Middle East Technical University
Ankara, Turkey
e–mail: *ucoluk@trmetu.bitnet*

関数 CHANGEVAR は (少なくとも) 四つの引数をとります。

- **第一引数**
  微分方程式の従属変数のリスト。もし一つの従属変数しかなければ、リストではなく変数そのものを与えても良い。
- **第二引数**
  新しい独立変数のリストもしくは変数が一つであれば変数そのもの。
- **第三および第四引数 等** *etc.*
  次の形式の方程式。

  古い変数 = 新しい変数の式

  左辺はカーネル出なければならない。これは、古い変数を新しい変数で置き換える変換式を定義する。
- **最後の引数**
  微分方程式のリスト。一つであればリストではなく式そのものを与えても良い。

もし、CHANGEVAR の最後の引数がリストであれば、結果もまたリストになります。

入力されたものの逆ヤコビ行列を表示することができます。このためには、DISPJACOBIAN スイッチをオンにして下さい。

8.1　例: 二次元のラプラス方程式

直交座標系での二次元のラプラス方程式は次のとうりです。

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

これを極座標系に変換することを考えます。変数変換の式は、

$$x = r\cos\theta, \qquad y = r\sin\theta$$

となります。CHANGEVAR でこの問題を計算するには次のように入力します。

```
CHANGEVAR({u},{r,theta},{x=r*cos theta,y=r*sin theta},
                        {df(u(x,y),x,2)+df(u(x,y),y,2)} );
```

ここで最初と最後の引数は要素が一つだけなので中括弧は付けなくても構いません。また、第三引数の中括弧もつかなくとも構いません。(リストではなく、変換式を第三、第四引数として与えることができます。) 第二引数の中括弧は、いまの例の場合には必要で、省略することはできません。従って、上の例は次のように入力することができます。

```
CHANGEVAR(u,{r,theta},x=r*cos theta,y=r*sin theta,
                      df(u(x,y),x,2)+df(u(x,y),y,2) );
```

u(x,y) は紙と鉛筆を使って計算する時の様に u(r,theta) になります。u(r,theta) は変換された従属変数を表します。

第9章　COMPACT: 関係式による式の簡約

Anthony C. Hearn
RAND Corp.
Santa Monica CA 90407-2138
Email: *hearn@rand.org*

COMPACT パッケージは与えられた関係式のもとで多項式を簡約します。このパッケージでは COMPACT 演算子を定義しています。この演算子の構文は次の通りです。

COMPACT(< **式** >, < **リスト** >):< **式** >

< **式** > は任意の代数式で、< **リスト** > は関係式または等式のリストです。例えば、

```
compact(x**2+y**3*x-5y,{x+y-z,x-y-z1});
compact(sin(x)**10*cos(x)**3+sin(x)**8*cos(x)**5,
        {cos(x)**2+sin(x)**2=1});
let y = {cos(x)**2+sin(x)**2-1};
compact(sin(x)**10*cos(x)**3+sin(x)**8*cos(x)**5,y);
```

COMPACT は関係式を与えられた式に適用し、結果として最も項数の少なくなるように式を変形します。ここで使われている方法の概略は次の通りです。

1. 関係式はそれぞれ分子、分母に独立に適用されます。つまり、多項式をそれぞれ簡約し結果を出します。
2. 簡約は順次行われます。つまり、多項式は一度にはただ一つの関係式に関して簡約が行われます。
3. 多項式中の変数については順序の入れ替えを行い、関係式に現れる変数 (カーネル) が最も低い順位になるように並べ変えます。
4. 残っているカーネルに関する係数 (これは関係式に現れる変数のみを含んでいる) は関係式で簡約されます。
5. 多項式の商/余りの計算を行います。もし結果の余りがもとの式よりも項数が少なければ、これをもとの式と入れ換えます。
6. 残された式は、関係式を使って、“最隣接” 法を使って簡約します。

巡回セールスマン問題に適用した場合と同様に、最隣接法では簡約の問題に対して最適な結果が得られることは保証できません。実際、多くの場合でこの方法は最適な結果と比べて二倍以内の結果が得られます。しかし、特別な例に対してはもっと悪い結果を与えます。

それ以外に、最適な結果を得ることが出来なくなる原因として、与えられた式を関係式で逐次簡約を行っていくことが上げられます。例えば、次の例では

```
compact((a+b+c)*(a-b-c)*(-a+b-c)*(-a-b+c),
        {x1=a+b+c,x2=a-b-c,x3=-a+b-c,x4=-a-b+c})
```

式は $x_1x_2x_3x_4$ が一番簡単な結果となるはずですが、与えられた関係式のどの一つを取ってみても、展開された式 $a^4 - 2a^2b^2 - 2a^2c^2 + b^4 - 2b^2c^2 + c^4$ をより簡単な式に変形しません。従って、得られる最終結果は最適な結果とはなりません。

第10章　CONTFR: 連分数近似

Herbert Melenk

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin–Dahlem, Germany

e–mail: *melenk@zib.de*

連分数の計算は MISC パッケージでサポートされており、使う前にはこのパッケージをロードする必要があります。

CONTINUED_FRACTION は一つの引数をとる関数です。これは、2 つの要素をもつリストを返します。最初の要素は有理数で近似した値で、二番目の要素は同じ値を連分数で表した時の係数のリストです。

```
continued_fraction sqrt 37;

          1555849
        {---------,{6,12,12,12,12,12}}
          255780
```

近似の精度は PRECISION 演算子で変更するか、もしくは第二引数に結果の有理数の分母の最大値を指定することで変えることができます。

```
load_package misc;

precision 6;

        12

continued_fraction pi;

          355
        {-----,{3,7,15,1}}
          113
```

```
precision 12;

        6

continued_fraction pi;

          1146408
        {---------,{3,7,15,1,292,1,1,1,2,1}}
          364913

continued_fraction(pi,100);

          22
        {----,{3,7}}
          7
```

第11章　CVIT: ディラックのγ行列のトレース計算

V. Ilyin, A. Kryukov, A. Rodionov and A. Taranov

Institute for Nuclear Physics
Moscow State University
Moscow, 119899 Russia

このパッケージでは、REDUCE 高エネルギー物理学のパッケージの機能を与えます。Γ-行列に基ずく計算ではなく、クリフォード代数に基ずいて計算を行ないます。これは Cvitanovic によって記述されているように、Γ-行列を 3-j シンボルとして扱います。

関数は、通常のパッケージに含まれているものと同一の計算を行なうようになっています。計算を制御するために四つのスイッチが用意されています。

11.1　CVIT

もし、オンであれば Kennedy-Cvitanovic のアルゴリズム等を使って計算を行ないます。

11.2　CVITOP

オンにすると Fierz による最適化を行ないます。通常はオフです。

11.3　CVITBTR

バブルとトライアングルの因数分解のスイッチです。通常はオンです。

11.4　CVITRACE

CVIT パッケージの内部の動作をトレースするスイッチです。通常はオフになっています。

```
index j1,j2,j3,;
```

```
vecdim n$

g(l,j1,j2,j2,j1);


 2
n


g(l,j1,j2)*g(l1,j3,j1,j2,j3);


 2
n

g(l,j1,j2)*g(l1,j3,j1,j3,j2);


n*( - n + 2)
```

第12章　DEFINT: 定積分

Kerry Gaskell

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Heilbronner Strasse 10
D–10711 Berlin – Wilmersdorf
Federal Republic of Germany

E–mail: *neun@zib-berlin.de*[1]

July 1994

12.1　序論

この文書は REDUCE の定積分のパッケージについて記述しています。このパッケージは、いくつかの特殊関数を含む多くの関数の定積分を求めます。Stan Kameny による周回積分のプログラムもこのパッケージに含まれていますが、ここでは記述していません。ここで記述している積分の方法は通常の方法とは大分違って、まず被積分関数をまず Meijer G 関数で表し、(Meijer G 関数の定義は [6] を見て下さい。) その積分を次の Meijer G 関数の積分に関する公式

$$\int_0^\infty x^{\alpha-1} G_{uv}^{st}\left(\sigma x \;\middle|\; \begin{matrix}(c_u)\\(d_v)\end{matrix}\right) G_{pq}^{mn}\left(\omega x^{l/k} \;\middle|\; \begin{matrix}(a_p)\\(b_q)\end{matrix}\right) dx = k G_{kl}^{ij}\left(\xi \;\middle|\; \begin{matrix}(g_k)\\(h_l)\end{matrix}\right) \tag{12.1}$$

を使って計算します。

結果として得られる Meijer G 関数を、直接もしくは超幾何関数の簡約を行うことによって、最終的な結果を出します。この理論に関してもっと詳しいことは [2] を参照して下さい。

12.2　0 から ∞ までの区間での定積分

例えば、次の積分の計算をさせてみます。

$$\int_0^\infty x^{-1} e^{-x} \sin(x)\, dx$$

パターンマッチによって、Meijer G 関数に変換し、それを上の式 (1) に代入すると次式が得られます。

[1] この定積分のパッケージは ZIB での一年間の滞在中に作成された。コメントや問題点等に関することは neun@sc.zib-berlin.de の Winfried Nuen に送って下さい。

$$\sqrt{\pi}\int_0^\infty x^{-1}G_{01}^{10}\left(x\;\middle|\;\begin{matrix}\cdot\\0\end{matrix}\right)G_{02}^{10}\left(\frac{x^2}{4}\;\middle|\;\begin{matrix}\cdot\,\cdot\\\frac{1}{2}\;0\end{matrix}\right)dx$$

積分の正当性を調べます。今の例の様に、大丈夫であれば、公式を使うことにより、次のような Meijer G 関数で表した結果が得られます。

$$G_{22}^{12}\left(1\;\middle|\;\begin{matrix}\frac{1}{2}\;1\\\frac{1}{2}\;0\end{matrix}\right)$$

これは、超幾何関数で次式のように表されます。

$${}_2F_1(\tfrac{1}{2},1;\tfrac{3}{2};-1)$$

これから、積分結果として、次の正しい結果を得ることが出来ます。

$$\frac{\pi}{4}$$

上の (1) 式で、二番目の Meijer G 関数を、自明な Meijer G 関数で置き換えた式も正しいので、単一の Meijer G 関数の積分に使うことが出来ます。

文献 [6]. には、多数の特別な Meijer G 関数の形のリストが載っています。

12.3 その他の範囲での積分

前の節の説明は、ゼロから無限大までの区間での定積分に限定されていましたが、ゼロから任意の点、さらには任意の区間での定積分の計算に拡張することが出来ます。一つの方法は、ヘビサイド関数を使った次の公式を利用することです。

$$\int_0^\infty x^2e^{-x}H(1-x)\,dx=\int_0^1 x^2e^{-x}dx$$

他には、やはり通常の不定積分を経由して計算するのではなく、 Meijer G 関数を使った次の公式を利用する方法があります。

$$\int_0^y x^{\alpha-1}G_{pq}^{mn}\left(\sigma x\;\middle|\;\begin{matrix}(a_u)\\(b_v)\end{matrix}\right)dx=y^\alpha\,G_{p+1\,q+1}^{m\,n+1}\left(\sigma y\;\middle|\;\begin{matrix}(a_1..a_n,1-\alpha,a_{n+1}..a_p)\\(b_1..b_m,-\alpha,b_{m+1}..b_q)\end{matrix}\right)\quad(2)$$

もっと詳しい理論については [2] を参照して下さい。

例えば、次の積分を計算することを考えてみましょう。

$$\int_0^y\sin(2\sqrt{x})\,dx$$

まず、 Meijer G 関数をパターンマッチによって見つけだし、(2) 式に代入すると、次の式が得られます。

$$\int_0^y G_{02}^{10}\left(x \;\middle|\; \begin{matrix}\cdot & \cdot \\ \frac{1}{2} & 0\end{matrix}\right) dx$$

これは、次のようになります。

$$y\, G_{13}^{11}\left(y \;\middle|\; \begin{matrix}0 \\ \frac{1}{2} & -1 & 0\end{matrix}\right) dx$$

これから、次の結果が得られます。

$$\frac{\sqrt{\pi}\, J_{3/2}(2\sqrt{y})\, y}{y^{1/4}}$$

12.4 定積分パッケージの使い方

このパッケージを使うには、まず最初に次のコマンドを使ってパッケージをロードします。

```
load_package defint;
```

定積分の計算は、int コマンドを使って行います。この構文は、

```
INT(EXPRN:代数式,VAR:カーネル,LOW:代数式,UP:代数式)
    :代数式.
```

で、ここで、LOW と UP は積分の下限と上限で、EXPRN を変数 VAR に関して定積分した結果を返します。

12.4.1 例

$$\int_0^\infty e^{-x} dx$$

```
 int(e^(-x),x,0,infinity);

 1
```

$$\int_0^\infty x \sin(1/x)\, dx$$

```
 int(x*sin(1/x),x,0,infinity);

           1
INT(X*SIN(---),X,0,INFINITY)
           X
```

$$\int_0^\infty x^2 \cos(x)\, e^{-2x} dx$$

```
int(x^2*cos(x)*e^(-2*x),x,0,infinity);

  4
-----
 125
```

$$\int_0^\infty x e^{-1/2x} H(1-x)\, dx = \int_0^1 x e^{-1/2x} dx$$

```
int(x*e^(-1/2x)*Heaviside(1-x),x,0,infinity);

 2*(2*SQRT(E) - 3)
-------------------
      SQRT(E)
```

$$\int_0^1 x\, \log(1+x)\, dx$$

```
int(x*log(1+x),x,0,1);

 1
---
 4
```

$$\int_0^y \cos(2x)\, dx$$

```
int(cos(2x),x,y,2y);

 SIN(4*Y) - SIN(2*Y)
---------------------
          2
```

12.5 積分変換

定積分パッケージを使って次のような積分変換を計算することが出来ます。

- ラプラス変換

- ハンケル変換
- Y-変換
- K-変換
- StruveH 変換
- フーリエ正弦変換
- フーリエ余弦変換

12.5.1 ラプラス変換

ラプラス変換

$$f(s) = \mathcal{L}\ \{\mathrm{F(t)}\} = \int_0^\infty e^{-st} F(t)\, dt$$

は、laplace_transform コマンドで計算できます。

これはパラメータとして、

- 被積分関数
- 積分の変数

を必要とします。

例えば、

$$\mathcal{L}\ \{e^{-at}\}$$

は、

```
laplace_transform(e^(-a*x),x);
```

と入力します。結果は、

$$\frac{1}{s+a}$$

となります。

12.5.2 ハンケル変換

ハンケル変換

$$f(\omega) = \int_0^\infty F(t)\, J_\nu(2\sqrt{\omega t})\, dt$$

は、hankel_transform コマンドで計算できます。 例えば、

```
hankel_transform(f(x),x);
```

このコマンドの使い方は、laplace_transform と同じです。

12.5.3 Y-変換

$$f(\omega) = \int_0^\infty F(t)\, Y_\nu(2\sqrt{\omega t})\, dt$$

で定義される Y-変換は、Y_transform コマンドで計算できます。 例えば、

```
Y_transform(f(x),x);
```

このコマンドの使い方は、laplace_transform と同じです。

12.5.4 K-変換

K-変換

$$f(\omega) = \int_0^\infty F(t)\, K_\nu(2\sqrt{\omega t})\, dt$$

は、K_transform コマンドで計算できます。

```
K_transform(f(x),x);
```

このコマンドの使い方は、laplace_transform と同じです。

12.5.5 StruveH 変換

StruveH 変換

$$f(\omega) = \int_0^\infty F(t)\, \mathrm{StruveH}(\nu, 2\sqrt{\omega t})\, dt$$

は、struveh_transform コマンドで計算できます。

```
struveh_transform(f(x),x);
```

このコマンドの使い方は、laplace_transform と同じです。

12.5.6 フーリエ正弦変換

フーリエ正弦変換

$$f(s) = \int_0^\infty F(t)\ \sin(st)\, dt$$

は、fourier_sin コマンドで計算できます。

```
fourier_sin(f(x),x);
```

このコマンドの使い方は、laplace_transform と同じです。

12.5.7 フーリエ余弦変換

フーリエ余弦変換

$$f(s) = \int_0^\infty F(t)\ \cos(st)\, dt$$

は、fourier_cos コマンドで計算できます。

```
fourier_cos(f(x),x);
```

このコマンドの使い方は、laplace_transform と同じです。

12.6 Meijer G 関数の定義の追加

与えられた関数の、Meijer G 関数による表現は、Meijer G 関数の表をパターンマッチを行って探すことによって、得られます。このリストは、十分大きなものですが、決して完全なものではありません。このため、利用者はこの表に新たな定義を追加することが必要になってくることが出てくるでしょう。この表への追加は、次のような行を追加することによって、行うことが出来ます。

```
defint_choose(f(~x),~var => f1(n,x);

symbolic putv(mellin!-transforms!*,n,'
              (() (m n p q) (ai) (bj) (C) (var)));
```

ここで、$f(x)$ は新しい関数で、$i = 1 \cdots p$, $j = 1 \cdots q$、C は定数、var は変数で n はインデックスナンバーです。

例えば、$\cos(x)$ という関数を考えると、

この **Meijer G** 表現は

$$\sqrt{\pi}\, G_{02}^{10}\left(\frac{x^2}{4}\;\middle|\;\begin{matrix}\cdot & \cdot\\ 0 & \frac{1}{2}\end{matrix}\right) dx$$

となる。積分パッケージでの内部表現では

```
defint_choose(cos(~x),~var)   => f1(3,x);
```

です。ここで、*3* はインデックスナンバーで、次式の公式に対応しています。

```
symbolic putv(mellin!-transforms!*,3,'
              (() (1 0 0 2) () (nil (quotient 1 2))
              (sqrt pi) (quotient (expt x 2) 4)));
```

また、$J_n(x)$ 関数の例で見てみると。

これの *Meijer G* 表現 は、

$$G_{02}^{10}\left(\frac{x^2}{4}\;\middle|\;\begin{matrix}\cdot & \cdot\\ \frac{n}{2} & \frac{-n}{2}\end{matrix}\right) dx$$

で、、定積分パッケージの内部での表現は次のようになります。

```
defint_choose(besselj(~n,~x),~var) => f1(50,x,n);

symbolic putv(mellin!-transforms!*,50,'
              ((n) (1 0 0 2) () ((quotient n 2)
                                 (minus quotient n 2)) 1
                                 (quotient (expt x 2) 4)));
```

12.7 print_conditions 関数

積分変換の正当性は次のコマンドを使って調べることが出来ます。

```
print_conditions().
```

例えば、次のようなラプラス変換を計算した後で、

```
laplace_transform(x^k,x);
```

print_conditions コマンドを使うと、

```
repart(sum(ai) - sum(bj)) + 1/2 (q + 1 - p)>(q - p) repart(s)
```

```
and ( - min(repart(bj))<repart(s))<1 - max(repart(ai))

or mod(arg(eta))=pi*delta

or ( - min(repart(bj))<repart(s))<1 - max(repart(ai))

or mod(arg(eta))<pi*delta
```

のように出力されます。ここで、

$$
\begin{aligned}
&delta = s + t - \tfrac{u-v}{2} \\
&eta = 1 - \alpha(v-u) - \mu - \rho \\
&\mu = \textstyle\sum_{j=1}^{q} b_j - \sum_{i=1}^{p} a_i + \tfrac{p-q}{2} + 1 \\
&\rho = \textstyle\sum_{j=1}^{v} dj - \sum_{i=1}^{u} c_i + \tfrac{u-v}{2} + 1 \\
&s, t, u, v, p, q, \alpha \text{ は式 (12.1) と同じ}
\end{aligned}
$$

12.8 謝辞

Victor Adamchik は REDUCE の定積分のパッケージを作成した。これは、このパッケージを作成する上で大いに参考になった。

第13章　DESIR: 線形同次微分方程式

C. Dicrescenzo, F. Richard–Jung, E. Tournier

Groupe de Calcul Formel de Grenoble
laboratoire TIM3
France

e–mail: *dicresc@afp.imag.fr*

このソフトウェアは **Q** 上の多項式を係数に持つ同次常微分方程式のゼロ近傍 (確定もしくは不確定特異点、正則点) での形式解を求めます。

このソフトウェアは、DELIRE 関数によって、あるいは DESIR を使って対話的に実行することができます。

DESIR 関数は引数を持ちません。これを使うことによって、データをあらかじめ処理しておくこと無く DELIRE を呼び出すことができます。この手続きは、さらに入力された方程式の変換を行なえます。これによって、方程式がゼロでない特異点を持ち、右辺式やパラメータが多項式であるように変形できます。

```
delire(x,k,grille,lcoeff,param)
```

このソフトウェアは **Q** 上の多項式を係数に持つ同次常微分方程式のゼロや特異点近傍での形式解を求めます。x は変数で、k は希望する項の数です。これは polysol に現れる各々の x_t の形式級数解 $a_0 + a_1 x_t + a_2 x_t^2 + \ldots + a_n x_t^n + \ldots$ において最初の $k+1$ 個の係数 $a_0, a_1, \ldots a_k$ が計算されます。微分演算子の係数は x^{grille} の多項式です。一般的には grille は 1 です。引数 lcoeff は微分演算子の係数のリストで、微分の増加する順に並べたリストです。param はパラメータのリストです。この関数は、一般解のリストを返します。

```
lcoeff:={1,x,x,x**6};
```

```
                          6
        lcoeff := {1,x,x,x }
```

```
param:={};
```

```
        param := {}
```

```
sol:=delire(x,4,1,lcoeff,param);

                   4       3        2
                 xt  - 4*xt  + 12*xt  - 24*xt + 24
sol := {{{{0,1,-----------------------------------,1},{
                                12

             }}},

                              4                3
        {{{0,1,(6*log(xt)*xt  - 18*log(xt)*xt

                               2
               + 36*log(xt)*xt  - 36*log(xt)*xt

                     4       3
               - 5*xt  + 9*xt  - 36*xt + 36)/36,0},{}

          }},

              1
        {{{-------,1,
              4
           4*xt

                 4       3        2
            361*xt  + 4*xt  + 12*xt  + 24*xt + 24
           ---------------------------------------,10},
                             24

          {}}}}
```

第14章　DFPART: 形式的関数の微分

Herbert Melenk

Konrad–Zuse–Zentrum
für Informationstechnik Berlin
Heilbronner Strasse 10
D–10711 Berlin Wilmersdorf
Federal Republic of Germany

Email: *melenk@sc.zib–berlin.de*

パッケージ DFPART は、形式的関数の常微分および偏微分の計算を支援します。そのような計算は、微分方程式あるいはべき級数展開において有用です。

14.1　形式的な関数

形式的な関数は数学の関数を表わすシンボルです。形式的な関数に関する最小の情報はその引き数の数です。プログラミングを促進するために、またよりよい出力を得るために、については、このパッケージは、形式的な関数の引き数は $f(x,y)$, $q(rho,phi)$ のような規定の名前を持っていると仮定します。形式的な関数は次のプロトタイプ形式によって宣言されます。

GENERIC_FUNCTION $fname(arg_1, arg_2 \cdots arg_n)$;

ここで、$fname$ は関数の (新しい) 名前で、arg_i はその形式的な引数です。以後、$fname$ を"形式的関数"、$arg_1, arg_2 \cdots, arg_n$ を"形式引数" そして $fname(arg_1, arg_2, \cdots, arg_n)$ を"形式的フォーム"と呼ぶ. 例:

```
generic_function f(x,y);
generic_function g(z);
```

この宣言により、REDUCE は以下の項目が定義されます。

- 形式的な偏微分係数 $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ $\frac{\partial g}{\partial z}$ および高次の微係数が存在する。f や g の他の変数に関する微分はゼロと仮定される。
- $f()$, $g()$ のような式は、$f(x,y)$, $g(z)$ の省略であると解釈される。
- $f(u,v)$ のような式は、$sub(x=u, y=v, f(x,y))$ の省略であると仮定される。
- 全微分 $\frac{df(u,v)}{dw}$ は $\frac{\partial f}{\partial x}\frac{du}{dw} + \frac{\partial f}{\partial y}\frac{dv}{dw}$ と計算される。

14.2 偏導関数

関数 DFP は偏導関数を表わします:

DFP$(expr, dfarg_1, dfarg_2 \cdots dfarg_n)$;

ここで $expr$ は関数式で、$dfarg_i$ は微分の変数です。例:

```
dfp(f(),{x,y});
```

これは、 $\frac{\partial^2 f}{\partial x \partial y}$ を意味し、

```
dfp(f(u,v),{x,y});
```

$\frac{\partial^2 f}{\partial x \partial y}(u,v)$ を意味します。DF オペレーターとの互換性のため、微分変数をリストの形に入力する必要はありません。代わりに、DF 演算子と同じの構文を使用して、変数の後に反復回数を指定することができます。このおうな形式で入力された場合、内部では、上記の形式に変換されます。

式 $expr$ は引き数を持つ、あるいは引き数のない形式的関数であっても、またこれらの関数の代数式であっても構いません。第二のケースでは、各々の微分式を簡約するために通常の微分規則が適用されます。

スイッチ NAT がオンの場合、形式的関数の偏導関数は通常の添字形式で出力されます。例えば、$\frac{\partial^2 f}{\partial x \partial y}$ に対しては f_{xy} と、また $\frac{\partial^2 f}{\partial x \partial y}(u,v)$ は $f_{xy}(u,v)$ と出力されます。したがって、可能な場合は常に、単一文字を引数として使用するべきです。例:

```
generic_function f(x,y);
generic_function g(y);
dfp(f(),x,2);

F
 XX

dfp(f()*g(),x,2);

F  *G()
 XX

dfp(f()*g(),x,y);

F  *G() + F *G
 XY        X  Y
```

偏微分と全微分の違いは次の例によって示されます。

```
generic_function h(x);
dfp(f(x,h(x))*g(h(x)),x);

F (X,H(X))*G(H(X))
 X

df(f(x,h(x))*g(h(x)),x);

F (X,H(X))*G(H(X)) + F (X,H(X))*H (X)*G(H(X))
 X                   Y          X

 + G (H(X))*H (X)*F(X,H(X))
    Y        X
```

導関数に対する関係式 $\frac{dq}{dx} = f(x,q)$ によるテイラー級数展開の計算。

```
load_package taylor;
operator q;
let df(q(~x),x) => f(x,q(x));
taylor(q(x0+h),h,0,3);

                       F (X0,Q(X0)) + F (X0,Q(X0))*F(X0,Q(X0))
                        X              Y                          2
Q(X0) + F(X0,Q(X0))*H + ----------------------------------------*H
                                       2

+ (F  (X0,Q(X0)) + F  (X0,Q(X0))*F(X0,Q(X0))
    XX              XY

    + F (X0,Q(X0))*F (X0,Q(X0)) + F  (X0,Q(X0))*F(X0,Q(X0))
       X            Y              YX

                                2              2                3
    + F  (X0,Q(X0))*F(X0,Q(X0))  + F (X0,Q(X0)) *F(X0,Q(X0)))/6*H
       YY                         Y

      4
 + O(H )
```

通常、偏導関数に関しては非可換であると仮定されます。

```
dfp(f(),x,y)-dfp(f(),y,x);
```

```
F   - F
 XY    YX
```

しかしながら、形式的関数に対する宣言 DFP_COMMUTE を使用することにより、偏微分が可換であることを宣言できます。これは、形式的関数あるいは引数を指定した形式的関数を指定します。このような関数に対しては、微分の変数は形式的変数の順に整理されます。

```
generic_function q(x,y);
dfp_commute q(x,y);
dfp(q(),{x,y,y}) + dfp(q(),{y,x,y}) + dfp(q(),{y,y,x});

3*Q
   XYY
```

もし、微分の一部のみが可換である場合には、通常の REDUCE ルールを使用して宣言する必要があります。微分変数はリストとして書く必要があることに注意してください。

14.3 代入

もし、形式的フォームあるいは DFP 式が、代入に現れた場合、次のステップが実行されます:

1. 引数に対する置換えが行なわれます。もし引数リストが空なら、置換えは形式的引数に対して行なわれます。もしこれらが変わったなら、結果の式が新しい実際の引数として使われます。もし、形式的関数それ自身が置換えによって変化しなければこのプロセスはここで終了します。
2. もし関数名あるいは形式的関数フォームが置換えの左辺式に現れる場合、対応する右辺式で置き換えられます。
3. 新しいフォームは、偏微分変数のリストに従って部分的に微分されます。
4. 実パラメータは対応する形式的変数と置き換えられます。この置き換えは名前によって行なわれます。

例:

```
generic_function f(x,y);
sub(y=10,f());

F(X,10)

sub(y=10,dfp(f(),x,2));
```

```
F  (X,10)
 XX

sub(y=10,dfp(f(y,y),x,2));

F  (10,10)
 XX

sub(f=x**3*y**3,dfp(f(),x,2));

     3
6*X*Y

generic_function ff(y,z);
sub(f=ff,f(a,b));

FF(B,Z)
```

ファイル *dfpart.tst* は、Runge-Kutta による微分方程式の解法のための係数方程式を計算するための完全な応用を含む、より多くの例を含んでいます。

第15章　DUMMY: ダミー変数を含む式の正準表現

Alain Dresse

Université Libre de Bruxelles
Boulevard du Triomphe, CP 210/01
B–1050 BRUXELLES

E–mail: *adresse@ulb.ac.be* and

Hubert Caprasse

Université de Liège
Institut de Physique
Allée du 6 août
B–4000 LIEGE

E–mail: *hubert.caprasse@ulg.ac.be*

15.1　序論

計算機代数でダミー変数をうまく使用するのは、簡約の問題があるので非常に難しいことです。しかし、大きな式をそのままの形で保持したままで、その式を効率よく操作することは非常に重要なことです。このパッケージにある関数は、ダミー変数を含んだ式の正準表現を求めます。これによって、多項式表現の簡約が行えます。未知数 (式) は、可能な限り最小の条件を課した、一般的な演算子として表されます。このパッケージは広い範囲の計算に対して計算の効率化に役立ちます。

2節では、式を書く方法について説明します。また、宣言によってダミー変数と自由変数を区別する方法について説明します。

3節では、演算子の代数的性質および演算子としての性質を宣言する色々なやり方について説明します。`canonical` 関数の使い方について説明してあります。

4節では、インストールの方法について説明しています。

15.2 ダミー変数と自由変数

任意の n に対する次の式、

$$\sum_{a=1}^{n} f(a)$$

は、ダミー変数を使って、次のように書くことが出来ます。

$$f(a)$$

ここで、a は **ダミー**変数です。もし、上の式が、

$$\sum_{b=1}^{n} f(b)$$

と書かれたとき、ここで b もダミーの添字で、明らかに二つの式は同じなので、

$$f(a) - f(b); \to 0$$

となります。ダミー変数を宣言するには、二つの方法があります。

- i.

  ```
  dummy_base <idp>;
  ```

  ここで、idp は任意の未使用の変数名です。

- ii.

  ```
  dummy_names <d>,<dp>,<dpp> ....;
  ```

最初の宣言では、idp1,···, idpn はすべてダミー変数になります。つまり、“idxxx” という名前の変数は全てダミー変数になります。ここで、xxx は数字です。二番目の形式では、指定された変数がダミー変数となります。例えば、

```
dummy_base dv; ==> dv

        % ダミー変数は、dv1, dv2, dv3, ...

dummy_names i,j,k; ==> t

         % ダミー変数名は i,j,k.
```

と宣言したとき、次の様な式

```
op(dv1)*sin(dv2)*abs(x)*op(i)^3*op(dv2)$
```

が使えます。注意すべき点は、ダミーの添字は一度しか現れない (これはテンソル計算に限らない) か、式の中では繰り返して現れても構わないということです。

15.3 CANONICAL 演算子と使用法

全てのダミー変数は演算子の引数です。これらの演算子は、まったく一般的なもので、代数式の要素であっても、テンソル、スピノル、グラスマン変数等々であっても構いません。通常、これらは**可換**で、何も対称的な性質は持っていないと仮定されます。いくつかのコマンドを使うことによって、これらの性質に関する設定が可能です。

まず、NONCOM, SYMMETRIC そして ANTISYMMETRIC という宣言を演算子に対して使うことができます。また、次のように反可換であると宣言することができます。

```
anticom ao1, ao2;
```

最後に、SYMTREE 宣言によって、部分的な対称性を指定することができます。例えば、リーマンテンソルに対する宣言は次のようになります。

```
symtree (r, {!+, {!-, 1, 2}, {!-, 3, 4}});
```

ここで、それぞれのリストの先頭の !*, !+ と !- は、それぞれリスト中で指定された番号の添字に対して、対称性を持たない、対称である、反対称であることを表しています。ここで、添字は名前ではなく、引数として現れる番号で指定することに注意して下さい。1 は r の一番目の引数で、2 は二番目の引数、... を表します。この例では、r は、添字の組 1,2 と 3,4 の交換に対して対称であり、1 と 2 および 3 と 4 の交換に対しては反対称な演算子となります。

この宣言は、添字がダミーであるか、それともフリーであるかに関係なく有効であることに注意して下さい。このように設定して、CANONICAL を多項式に適応することによって、正準形式に変換できます。これによって、完全な簡約ができることが保証されています。

このパッケージには、二つの制限があります。まず、非可換な演算子は反可換な演算子と可換であると仮定しています。また、制約条件 (例えばビアンキの恒等式) が存在する場合の簡約はできません。

15.4 インストールの方法

DUMMY パッケージを使うためには、ASSIST パッケージの 2.2 版をあらかじめロードしておかなければなりません。このパッケージは、REDUCE のライブラリに含まれています。ソースファイルを持っているならば、MKFASL コマンドで、

```
mkfasl assist;
```

それに、

```
mkfasl dummy;
```

を実行して下さい。[1] これで、使えるようになります。例えば、DUMMY のテストファイルを実行してみて下さい。

[1] 通常、これらのパッケージは、コンパイルした形で提供されます。自分で MKFASL コマンドを使ってコンパイルする必要はありません。

第16章　EXCALC: 微分形式の計算

Eberhard Schrüfer
Institute SCAI.Alg
German National Research Center for Information Technology (GMD)
Schloss Birlinghoven
D-53754 Sankt Augustin
Germany
Email: *schruefer@gmd.de*

謝辞

このプログラムはこの数年に渡って開発されてきました。Anthony Hearn 博士に対して、彼がこの仕事に対して興味をいだいてくれたこと、そして私が 1984 年から 1985 年にかけて RAND 社を訪問したときに暖かく迎えてくれ、またサポートしてくれたことに対して深く感謝の意を表したい。Hertz-Stiftung 氏はこの訪問をサポートしてくれた。また Cologne 大学の F. W. Hehl 博士と ColegeDublin 大学の J. D. McCrea 博士はこの仕事に対して多くの示唆を与えてくれ、またこのプログラムをテストしてくれた。

16.1　はじめに

EXCALC は現代解析幾何の計算に詳しい人には容易に使用できるように設計しています。構文はできるだけ標準的な教科書に近づけています。従って数式処理のプログラムを作成するのにはあまり難しくありません。また入力も通常手で書いているのと同じ形になるようにしています。例えば、

```
f*x^y + u _| (y^z^x)
```

は外積と内積を含んだ式として解釈されます。プログラムは現在、スカラー値の外積、ベクトルおよびそれらの間での演算、非スカラー値の形式 (添字形式) を扱うことが出来ます。これによって、微分方程式や一般相対論や場の理論に置ける計算、任意に与えられたフレームでのテンソル場のラプラシアンの計算等を行うことが出来ます。このような計算は多くの領域で使用されているので、このプログラムは物理学や数学の多くの分野での問題に対して、利用することが出来ます。

このプログラムは REDUCE に完全に適合するように作成されているので、計算を行う場合に REDUCE の全ての機能を利用することが出来ます。このような計算にあまり詳しくない人にとっ

ても、このプログラムを操作していくことによって、計算方法等を学習していくことが出来るでしょう。

これは第二版の最終版です。もっと拡張された微分幾何のパッケージ (これには完全な記号添字の簡約やテンソル、写像、バンドル等が含まれます) を現在開発中です。

このパッケージに対するコメント等は直接作者に送って下さい。このプログラムを使用して得られた成果を発表するときには、この文書を引用して下さい。また論文のコピーを上の住所に送っていただけるとありがたいです。

16.2 宣言

外微分形式やベクトルは宣言文で定義されます。このような宣言はプログラムのどこに現れても構いませんが、少なくとも使用する前には行わなくてはなりません。宣言されていないものは、定数として扱われます。従って、零形式もまた宣言しておく必要があります。

外微分形式は PFORM によって、

PFORM < **宣言** $_1$>, < **宣言** $_2$>, ...;

で宣言されます。ここで、

< **宣言** > ::= < **名前** > | < **名前のリスト** >=< **数値** > | < **識別子** > |
< **式** >
< **名前** > ::= < **識別子** > | < **識別子** >(< **引数** >)

です。

例えば、

```
pform u=k,v=4,f=0,w=dim-1;
```

は U を次数 K の外微分形式、V を次数 4 の形式、F は次数 0 の形式 (関数) に、そして W は次数 DIM -1 の形式として宣言します。

もし、外微分形式が添字を持つ場合、宣言は次のようになります。

```
pform curv(a,b)=2,chris(a,b)=1;
```

添字の名前は何でも構いません。

入力を簡略するため、同じ次数の形式はグループにしてリストの形で入力しても構いません。

```
pform {x,y,z}=0,{rho(k,l),u,v(k)}=1;
```

ベクトルの宣言も同様です。TVECTOR コマンドは名前のリストを引数に取ります。

TVECTOR < **名前** $_1$>, < **名前** $_2$>, ...;

例えば、X をベクトルに、そして COMM を添字を二つ取るベクトルとして宣言するには、次のように入力します。

```
tvector x,comm(a,b);
```

もしすでに宣言されている名前に対して再度宣言を行った場合には、以前の宣言は取り消され、新しい宣言が有効になります。

記号や式の外微分の次数は関数 EXDEGREE で得られます。

EXDEGREE < **式** >;

例:

```
exdegree(u + 3*chris(k,-k));

1
```

16.3 外積

外微分形式同士の外積は演算子^ (ウェッジ) で行います。因子は外積の交換関係によって REDUCE の通常の順序に並べ替えられます。

例:

```
pform u=1,v=1,w=k;

u^v;

U^V

v^u;

- U^V

u^u;

0

w^u^v;

      K
( - 1) *U^V^W

(3*u-a*w)^(w+5*v)^u;

A*(5*U^V^W - U^W^W)
```

考えている空間の次元については、SPACEDIM を使って宣言することが出来ます。

SPACEDIM < **数値** > | < **識別子** >;

もし、結果として得られる形式の次数が空間の次元よりも大きくなればそれは 0 になります。

```
spacedim 4;

pform u=2,v=3;

u^v;

0
```

16.4 偏微分

偏微分は演算子@で表されます。これは REDUCE の DF 演算子と同じ機能を持っています。

例:

```
@(sin x,x);

COS(X)

@(f,x);

0
```

FDOMAIN は識別子を指定した変数に関する関数であると宣言します。 次の例では、F は変数 X と Y に関する関数であり、また H は X の関数であると宣言しています。

```
fdomain f=f(x,y),h=h(x);
```

F や H に偏微分作用素@を作用させると、

```
@(x*f,x);

F + X*@  F
       X

@(h,y);

0
```

という結果が得られます。

偏微分の記号@は、また一つの引数をとる演算子としても使うことができます。この場合、これは接ベクトルの自然な基底の要素を表します。

例

```
a*@ x + b*@ y;

A*@  + B*@
   X      Y
```

16.5 外微分

外微分形式の外微分は演算子 d で行います。 例えば、

```
pform x=0,y=k,z=m;

d(x * y);

X*d Y + d X^Y

d(r*y);

R*d Y

d(x*y^z);

      K
( - 1) *X*Y^d Z  + X*d Y^Z + d X^Y^Z
```

この d 演算子の展開は NOXPND D で抑制することができます。

```
noxpnd d;

d(y^z);

d(Y^Z)
```

展開のスイッチがオフの場合には、外積に対する正規表現を得るために、左側に現れる D 演算子は右にシフトされます。

```
d y ^ z;

        K
- ( - 1) *Y^d Z + d(Y^Z)
```

XPND D コマンドを実行すると再び展開が行われるようになります。

FDOMAIN コマンドによって定義された関数は偏微分を使って展開されます。

```
pform x=0,y=0,z=0,f=0;

fdomain f=f(x,y);

d f;

@  F*d X + @  F*d Y
 X          Y
```

このような関数の引数が、また関数になっている場合など、他の変数に対して依存関係にあれば、微分のチェインルールに従って計算が行われます。

```
fdomain y=y(z);

d f;

@  F*d X + @  F*@  Y*d Z
 X          Y    Z
```

偏微分による展開を行わせないようにするには、NOXPND @ を、また展開を行わせるには XPND @ を実行します。

この外微分演算子は二度作用させると結果は零になります。また、結果の外微分形式の次数が次元よりも大きくなると零になります。

```
d d x;

0

pform u=k;
spacedim k;

d u;

0
```

16.6 内積

ベクトルと外微分形式との内積は記号 _| で表します。これは多くの教科書で使われている書き方です。もし外微分形式が外積の形になっている場合には、その全ての因子に対して内積が計算されます。

例

```
pform x=0,y=k,z=m;

tvector u,v;

u _| (x*y^z);

          K
X*(( - 1) *Y^U _| Z + U _| Y^Z)
```

一つの外微分形式に内積を続けて作用させた場合には、結果の式は変数の順序によって整理されます。例えば、

```
(u+x*v) _| (u _| (3*z));

- 3*U _| V _| Z
```

基底要素の双対はシステムにより処理されます。すなわち、

```
pform {x,y}=0;

(a*@ x+b*@(y)) _| (3*d x-d y);

3*A - B
```

16.7 リー微分

リー微分は内挿演算子 |_ で表され、ベクトルと外微分形式もしくはベクトル同士で演算を行います。微分形式に対してベクトルでリー微分を行うと、結果は内積と外微分で表されます。例えば、

```
pform z=k;

tvector u;

u |_ z;

U _| d Z + d(U _| Z)
```

ベクトル同士のリー微分は、ベクトルは反交換関係によって整理されます。

例:

```
tvector u,v;

v |_ u;

- U |_ V

pform x=0,y=0;

(x*u) |_ (y*v);

- U*Y*V _| d X + V*X*U _| d Y + X*Y*U |_ V
```

16.8 ホッジの * 演算子

ホッジの * 双対演算子は K 次の外微分形式を N-K 次の外微分形式に写像します。ここで、N は考えている空間の次元です。この演算子を二階作用させると元の外微分形式と因子倍された同じ外微分形式が得られます。次の例はこの因子がどの様に選ばれるかを示しています。

```
spacedim n;
pform x=k;

# # x;

         2
       (K  + K*N)
( - 1)           *X*SGN
```

この例での未知数 SGN は計量の行列式の符号を表しています。これは利用者が値を設定するか、もしくは計量を特定 (COFRAME 演算子により) することによって自動的に設定されます。g が計量を表す行列とすると、$g/|g|$ に設定されます。もし、ホッジの*演算子が最大次数の外微分形式の左に現れているときには、次式に従って*演算子は右に移されます。

```
pform {x,y}=k;

# x ^ y;

         2
       (K  + K*N)
( - 1)           *X^# Y
```

もし coframe が定義されていれば、更に簡約が行われます。

16.9 変分

関数 VARDF は外微分形式 (ラグランジェ場) に関するラグランジェアンの変分 (variation) を返します。変数 BNDEQ!*に境界に沿って積分すると零になる式が保存されます。

構文:

VARDF(< **ラグランジェの** n **形式** >,< **外微分形式** >)

例:

```
spacedim 4;

pform l=4,a=1,j=3;

l:=-1/2*d a ^ # d a - a^# j$  %電磁場のラグランジェアン

vardf(l,a);

- (# J + d # d A)             %マックスウェルの方程式

bndeq!*;

- 'A^# d A                    %境界での方程式
```

制約事項:

現在の実装 (インプリメント) では、ラグランジェアンは場と d,#, @の演算子で構成されたものに限られます。添字の付いた量に関する変分は計算できません。

対称演算子 (ベクトル場) に付随する保存量の計算のため、NOETHER 関数が定義されています。この構文は、次の通りです。

NOETHER(< **ラグランジェの** n **形式** >,< **場** >,< **対称性の生成子** >)

例:

```
pform l=4,a=1,f=2;

spacedim 4;

l:= -1/2*d a^#d a;   %自由マックスウェル場;

tvector x(k);        %未定義の生成演算子;

noether(l,a,x(-k));

( - 2*d(X _|A)^# d A - (X _|d A)^# d A + d A^(X _|# d A))/2
         K               K                      K
```

上の式は、もし X を位置を動かす演算子とすると、マックスウェル場のエネルギー運動量を表します。

16.10 インデックスの扱い

外微分形式やベクトルは添字を持つことができます。入力では、添字は引数の形で与えます。正の引数は上付きの添字で、負の引数は下付きの添字を表します。出力では、NAT スイッチがオンのときには、添字付きの量は二次元的に表示されます。添字は識別子かまたは数値で与えます。

例:

```
pform om(k,l)=m,e(k)=1;

e(k)^e(-l);

 K
E ^E
    L

om(4,-2);

  4
OM
   2
```

現在の版では、添字の範囲が設定されたときのみ完全な簡約が行われます。この制約は速やかに取り除かれることものと期待しています。もし添字の範囲 (添字の取る値の範囲) が指定されれば、与えられた式は全ての添字の可能な値に対して計算が行われ、その和が計算されます。

例:

```
indexrange t,r,ph,z;

pform e(k)=1,s(k,l)=2;

w := e(k)*e(-k);

         T       R        PH       Z
W := E *E  + E *E  + E  *E   + E *E
      T       R       PH        Z


s(k,l):=e(k)^e(l);
```

```
 T T
S    := 0

 R T       T  R
S    := - E ^E

 PH T       T  PH
S     := - E ^E

  .
  .
  .
```

もし評価しようとする式が代入文でない場合、式の値はNSという仮の名前の添字付きの変数に対する代入文の形で出力されます。しかし、これは出力が行われるだけであり、変数NSへの代入は実際には行われません。

```
e(k)^e(l);

  T T
NS    := 0

  R T       T  R
NS    := - E ^E
  .
  .
  .
```

変数NSの添字の位置はシステムによって決められるため、式の順序の入れ替えが行われる可能性があるので、一意には決められません。式を表示するときには、大域変数への代入文を使って行った方がよい。

添字の範囲は各々の変数に対して指定することが出来ます。例えば、次の宣言

```
indexrange {k,l}={x,y,z},{u,v,w}={1,2};
```

は、変数k,lに対してはその範囲がx,y,zで、変数u,v,wに対しては1,2の範囲の値を取ることを指定します。ここで宣言されていない添字変数の値の範囲はこれらの範囲を合わせたもの全てになります。

上の例でのINDEXRANGE文を実行した後では、添字式の計算は次のようになります。

```
pform w n=0;
```

```
w(k)*w(-k);

    X       Y       Z
W *W  + W *W  + W *W
 X       Y       Z

w(u)*w(-u);

    1       2
W *W  + W *W
 1       2

w(r)*w(-r);

    1       2       X       Y       Z
W *W  + W *W  + W *W  + W *W  + W *W
 1       2       X       Y       Z
```

ある場合には、指定した添字変数に関する展開もしくは全ての変数に関する展開を行なわせたくないときがあります。この時には次のコマンド、

NOSUM < **名前** $_1$>, ...;

とスイッチ NOSUM が用意されています。NOSUM コマンドは指定した添字変数に関する展開を中止します。再開したいときには RENOSUM コマンドを使います。NOSUM スイッチをオンにすると全ての変数に関する展開を中止します。

添字付きの量に対して対称性を定義することが INDEX_SYMMETRIES コマンドでできます。これは次のように使います。

```
index_symmetries u(k,l,m,n): symmetric     in {k,l},{m,n}
                             antisymmetric in {{k,l},{m,n}},
                 g(k,l),h(k,l): symmetric;
```

この例では、u は最初の二つおよび最後の二つに関しては対称で、また指定された添字 $\{k,l\}$ と $\{m,n\}$ の組合せに関しては反対称であると宣言しています。また、g や h の例のように、添字に関して完全に対称もしくは反対称であれば、添字を指定する必要はありません。

もし可能であれば、このコマンドを使って対称性に関する設定を行うことで必要なメモリや計算時間を大幅に節約することができます。

以前の版での symmetric や antisymmetric コマンドはもはや何の効果もありません。

16.11 計量

EXCALC では計量構造は、基底の 1-形式 (coframe) の組と計量を指定して定義します。

構文:

COFRAME < **識別子** >< $(index_1)$ >=< **式** $_1$ >,
< **識別子** >< $(index_2)$ >=< **式** $_2$ >,
.
.
.
< **識別子** >< $(index_n)$ >=< **式** $_n$ >
WITH METRIC < **名前** >=< **式** >;

この文は同時に空間の次元と添字の範囲を設定します。もし計量がユークリッドであれば、WITH METRIC 以下は省略することが出来ます。また擬ユークリッド計量の場合には、 WITH SIGNATURE < **対角要素** > で計量を設定することができます。計量構造を計量テンソルと基底 1-形式とに分離する仕方はまったく任意に行えます。例えば、直交 frame と座標 frame の極大も含まれます。

例:

```
coframe e r=d r, e(ph)=r*d ph
  with metric g=e(r)*e(r)+e(ph)*e(ph);     %極 cofame

coframe e(r)=d r,e(ph)=r*d(ph);            %上と同じ

coframe o(t)=d t, o x=d x
  with signature -1,1;                     %ローレンツ coframe

coframe b(xi)=d xi, b(eta)=d eta           %光円錐 coframe
  with metric w=-1/2*(b(xi)*b(eta)+b(eta)*b(xi));

coframe e r=d r, e ph=d ph                 %極座標基底
  with metric g=e r*e r+r**2*e ph*e ph;
```

計量のそれぞれの要素は単に添字を付けて呼び出すことによって求められます。共変計量の行列式 は変数 DETM!*に保存されています。計量の添字は上または下に動かす必要はありません。システムはこれを自動的に行います。つまり、添字付きの量にどの様な添字の位置が与えられていようとも、単に希望する添字を指定するのみでどの様な添字の位置に対しても望みの値が取り出せます。

例:

```
coframe e t=d t,e x=d x,e y=d y
```

```
 with signature -1,1,1;

pform f(k,l)=0;

antisymmetric f;

f(-t,-x):=ex$ f(-x,-y):=b$  f(-t,-y):=0$
on nero;

f(k,-l):=f(k,-l);

 X
F    := - EX
   T

 T
F    := - EX
   X

 Y
F    := - B
   X

 X
F    := B
   Y
```

座標関数の微分を含んでいる任意の式は、基底 1-形式の式に変換できます。また、基底 1-形式の外微分の計算も行えます。

例:球座標

```
coframe e(r)=d(r), e(th)=r*d(th), e(ph)=r*sin(th)*d(ph);

d r^d th;

  R  TH
(E ^E  )/R

d(e(th));

  R  TH
(E ^E  )/R
```

```
pform f=0;

fdomain f=f(r,th,ph);

factor e;

on rat;

d f;        %球座標での F の傾き;

 R           TH              PH
E *@  F + (E  *@   F)/R + (E  *@   F)/(R*SIN(TH))
    R           TH              PH
```

COFRAME コマンドで定義された frame に対して双対な frame は **FRAME** コマンドで 導入することができます。

FRAME < **識別子** >;

このコマンドによって双対性を取り扱うことが出来ます。また座標関数の接ベクトルは frame の基底ベクトルで置き換えられます。

例:

```
coframe b r=d r,b ph=r*d ph,e z=d z; %円筒 coframe;

frame x;

on nero;

x(-k) _| b(l);

    R
NS    := 1
  R

     PH
NS      := 1
  PH

    Z
NS    := 1
  Z

x(-k) |_ x(-l);        %双対 frame の交換子;
```

```
NS     := X  /R
  PH R     PH


NS     := ( - X  )/R  %つまり、これは座標の基底では無い;
  R PH         PH
```

便利に使えるように、DISPLAYFRAME; というコマンドを入力したときにはいつでも現在の frame が表示されます。

ホッジの*双対演算子を coframe の基底要素に対して作用させると、現在の計量をもとに、双対要素を計算して返します。

完全反対称 Levi-Cevita テンソル EPS も求められます。EPS の共変位置での添字偶置換の値は+1になっています。

16.12 リーマン接続

RIEMANNCONX コマンドは接続 1-形式を計算するために用意されています。 値は RIEMANNCONX によって指定された名前に値が保存されます。このコマンドは接続の計算に、基底 1-形式の微分と内積を使って求めるより、はるかに簡単です。

例:$S(2)$ での接続 1-形式と曲率 2-形式の計算

```
coframe e th=r*d th,e ph=r*sin(th)*d ph;

riemannconx om;

om(k,-l);                    %接続形式の表示;

  TH
NS     := 0
     TH

  PH         PH
NS     := (E  *COS(TH))/(SIN(TH)*R)
     TH

  TH            PH
NS     := ( - E  *COS(TH))/(SIN(TH)*R)
     PH
```

```
  PH
NS      := 0
     PH

pform curv(k,l)=2;


curv(k,-l):=d om(k,-l) + om(k,-m)^om(m-l); %曲率形式

    TH
CURV      := 0
       TH

    PH            TH  PH   2
CURV      := ( - E  ^E  )/R          %明らかにこれは半径 R の球になる。
       TH

    TH         TH  PH   2
CURV      := (E  ^E  )/R
       PH

    PH
CURV      := 0
       PH
```

16.13 順序と構造

外微分形式もしくベクトルの順序は FORDER コマンドで変更することができます。式中で、FORDER で最初に現れた名前が二番目以降の名前よりも先にくるように順序付けられます。FORDER で指定されない名前は一番最後になります。この順序付けは単に出力だけではなく、内部の形式もこの順序に従っています。従って、この宣言文はプログラムの実行時間や使用するメモリに影響を与えます。REMFORDER は引数で与えられた名前に関して、その順序を元々の順序に戻します。

式の部分式に名前をつけることで、構造を持った形に変形することができます。これは KEEP コマンドで行えます。 このコマンドの構文は次の通りです。

KEEP < **名前** $_1$>=< **式** $_1$>,< **名前** $_2$>=< **式** $_2$>, ...

このコマンドを実行すると、< **名前** > はその定義を使うことなく簡約が行われます。システムは式の中に < **名前** > ができるだけ現れるよう順序を入れ換えます。

例:

```
pform x=0,y=0,z=0,f=0,j=3;

keep j=d x^d y^d z;
```

```
j;

J

d j;

0

j^d x;

0

fdomain f=f(x);

d f^d y^d z;

@  F*J
 X
```

KEEP の機能は現在の所かなり制限されており、KEEP の右辺は外積の形でなければなりません。

16.14 演算子とコマンドの要約

表 16.1 では、EXCALC のコマンドとその説明がある頁を要約しています。

^	外積	249
@	偏微分	250
@	接ベクトル	251
#	ホッジの * 演算子	254
_|	内積	252
|_	リー微分	253
COFRAME	COFRAME の宣言	259
d	外微分	251
DISPLAYFRAME	フレームの表示	262
EPS	Levi-Civita テンソル	262
EXDEGREE	外積の次数の計算	249
FDOMAIN	依存関係の宣言	250
FORDER	順序の宣言	263
FRAME	coframe に双対な frame の宣言	261
INDEXRANGE	インデックスの宣言	256
INDEX_SYMMETRIES	添字の対称性の宣言	258
KEEP	構造化コマンド	263
METRIC	計量 (メトリック) の設定	259
NOETHER	Noether 流の計算	255
NOSUM	添字に関する和の展開を行わない	258
NOXPND d	d に関する積規則を適用しない	251
NOXPND @	偏微分を展開しない	252
PFORM	外微分形式の宣言	248
REMFORDER	順序の取り消し	263
RENOSUM	添字に関する和を展開する	258
RIEMANNCONX	リーマン接続の計算	262
SIGNATURE	COFRAME 文で擬ユークリッド計量を設定する	259
SPACEDIM	空間の次元を設定する	249
TVECTOR	ベクトルの宣言	248
VARDF	変分	255
XPND d	d に関する積規則を適用する (既定値)	252
XPND @	偏微分を展開する (既定値)	252

表 16.1: EXCALC のコマンドの要約

第17章　FIDE: 偏微分方程式の有限要素法

Richard Liska

Faculty of Nuclear Science and Physical Engineering
Technical University of Prague
Brehova 7, 115 19 Prague 1, Czech Republic

e–mail: *tjerl@aci.cvut.cz*

FIDE パッケージは偏微分方程式を数値計算する場合の手続きを、有限要素法を生成することによって自動化します。この手続きでは、数式処理システムが処理できるような幾つかのルーチン処理があります。例えば、微分方程式を別の座標系に変換することや、差分化を行なうこと、差分スキームの解析、数値計算プログラムの生成などです。FIDE パッケージは次のモジュールから成ります。

EXPRES 偏微分方程式系を任意の直交座標系に変換する。

IIMET 偏微分方程式系を積分-補間法によって差分化する。

APPROX 差分スキームの次数を決定する。

CHARPOL これはフーリエの安定性解析に必要な、増幅行列や差分スキームの特性方程式を計算する。

HURWP von Neumann の安定条件を調べる為、多項式の根の場所を決定する。

LINBAND 差分スキームでよく現れる帯行列による線形代数方程式を解く為の FORTRAN プログラムを生成する。

もっと詳しいことは、このパッケージに付いている文書と例を見て下さい。このパッケージの有用性については、次の簡単な例から読みとれるでしょう。

```
off exp;

factor diff;

on rat,eqfu;

% どのインデックスが座標であるかを宣言する。
coordinates x,t into j,m;
```

```
% x 座標については均一な格子を使う。
grid uniform,x;

% 関数の座標依存を宣言する。
dependence eta(t,x),v(t,x),eps(t,x),p(t,x);

% p を既知の関数として宣言する。
given p;

same eta,v,p;

iim a, eta,diff(eta,t)-eta*diff(v,x)=0,
    v,diff(v,t)+eta/ro*diff(p,x)=0,
    eps,diff(eps,t)+eta*p/ro*diff(v,x)=0;


*****************************
*****      Program      *****          IIMET Ver 1.1.2
*****************************

      Partial Differential Equations
      ==============================

diff(eta,t) - diff(v,x)*eta    =    0

 diff(p,x)*eta
--------------- + diff(v,t)    =    0
      ro

               diff(v,x)*eta*p
diff(eps,t) + -----------------    =    0
                     ro


 Backtracking needed in grid optimalization
0 interpolations are needed in x coordinate
  Equation for eta variable is integrated in half grid point
  Equation for v variable is integrated in half grid point
  Equation for eps variable is integrated in half grid point
0 interpolations are needed in t coordinate
  Equation for eta variable is integrated in half grid point
  Equation for v variable is integrated in half grid point
  Equation for eps variable is integrated in half grid point
```

Equations after Discretization Using IIM :
==========

```
(4*(eta(j,m + 1) - eta(j,m) - eta(j + 1,m)
     + eta(j + 1,m + 1))*hx - (
    (eta(j + 1,m + 1) + eta(j,m + 1))
    *(v(j + 1,m + 1) - v(j,m + 1))
     + (eta(j + 1,m) + eta(j,m))*(v(j + 1,m) - v(j,m)))
 *(ht(m + 1) + ht(m)))/(4*(ht(m + 1) + ht(m))*hx)   =   0

(4*(v(j,m + 1) - v(j,m) - v(j + 1,m) + v(j + 1,m + 1))*hx*ro
  + ((eta(j + 1,m + 1) + eta(j,m + 1))
     *(p(j + 1,m + 1) - p(j,m + 1))
      + (eta(j + 1,m) + eta(j,m))*(p(j + 1,m) - p(j,m)))
 *(ht(m + 1) + ht(m)))/(4*(ht(m + 1) + ht(m))*hx*ro)   =   0

(4*(eps(j,m + 1) - eps(j,m) - eps(j + 1,m)
     + eps(j + 1,m + 1))*hx*ro + ((
       eta(j + 1,m + 1)*p(j + 1,m + 1)
        + eta(j,m + 1)*p(j,m + 1))
    *(v(j + 1,m + 1) - v(j,m + 1)) +
    (eta(j + 1,m)*p(j + 1,m) + eta(j,m)*p(j,m))
    *(v(j + 1,m) - v(j,m)))*(ht(m + 1) + ht(m)))/(4
   *(ht(m + 1) + ht(m))*hx*ro)   =   0
```

```
clear a;

clearsame;

cleargiven;
```

第18章　FPS: 形式巾級数の計算

Wolfram Koepf
ZIB Berlin
Email: *Koepf@ZIB.de*
Present REDUCE form by
Winfried Neun
ZIB Berlin
Email: *Neun@ZIB.de*

このパッケージはある型の関数を展開して、対応する Laurent-Puiseux 級数を

$$\sum_{k=0}^{\infty} a_k (x - x_0)^{k/n+s}$$

の形で求めます。ここで、s は 'シフト数', n は 'Puiseux 数', そして x_0 は '展開点' です。 次にあげるタイプがサポートされている。

- **'有理型' 関数**、 これは有理式もしくはある次数の有理式の微係数を持つ関数。
- **'超幾何型' 関数** 、これはある整数値 m に対して a_{k+m}/a_k が有理式になる関数。
- **'指数型' 関数**、 これは定数係数の線形同次方程式を満たすような関数。

`FPS(f,x,x0)` は `f` を変数 `x` に関して点 `x0` で形式巾級数に展開します。これは、形式的ローラン級数 (負の巾) および Puiseux 級数 (分数巾) に対しても計算できます。もし第三引数が省略された場合には `x0 := 0` が仮定されます。

例: `FPS(asin(x)^2,x)` を計算すると以下のようになります。

```
         2*k  2*k             2  2
        x   *2   *factorial(k) *x
infsum(----------------------------,k,0,infinity)
        factorial(2*k + 1)*(k + 1)
```

もし可能であれば、出力は factorial を使って与えられます。場合によっては、Pochhammer 記号 `pochhammer(a,k)`:= $a(a+1)\cdots(a+k-1)$ が使われます。

`SimpleDE(f,x)` は f に対する多項式係数の線形同次方程式を決定します。変数 y を使っているので、これを別の目的で使わないようにして下さい。`factor df;` と設定しておく方が出力がきれいな形になります。

例: SimpleDE(asin(x)^2,x) を実行すると以下の出力が得られます。

```
             2
df(y,x,3)*(x  - 1) + 3*df(y,x,2)*x + df(y,x)
```

f に対する微分方程式の探索の深さは、変数 fps_search_depth; で制御されます。higher values for fps_search_depth の値を大きくすれば、解を見つける可能性が高くなります。しかし複雑度は増加します。通常は fps_search_depth の値は 5 になっています。例えば、FPS(sin(x^(1/3)),x) または SimpleDE(sin(x^(1/3)),x) を計算するには fps_search_depth:=6 にする必要があります。

FPS パッケージの出力はスイッチ tracefps で変更できます。スイッチ tracefps をオンにすると種々の計算の中間結果を出力します。

第19章　GENTRAN:FORTRANもしくはC言語への変換

Barbara L. Gates
RAND
Santa Monica CA 90407-2138 USA
Updated for REDUCE 3.4 by
Michael C. Dewar
The University of Bath
Email: *miked@nag.co.uk*

19.1　GENTRANによるFORTRANプログラムの作成

FORTスイッチによる出力では、式しか変換できませんが、プログラムも込めて変換するGENTRANというパッケージがあります。これを使うとREDUCEのプログラムから、FORTRANやRATFOR、C言語、PASCALへの変換が可能です。

これは、最初に

```
load gentran;
```

と入力して、必要なパッケージをロードします。以後は、例えば、

```
gentran f:=2*x**2-5*x+6;
```

と入力すると、これをFORTRANに変換した式、

```
F = 2*X**2-5*X+6
```

が出力されます。GENTRANコマンド中では右辺の式が評価されずにそのままの形で変換されます。例えば、

```
f:=2*x**2-5*x+6 $
gentran q:=f/df(f,x)$
```

としても、

```
Q=F/DF(F,X)
```

のように変換されるだけです。Q に F/DF(F,X) を計算した値を代入したい場合には eval という関数を使って

```
gentran q:=eval(f/df(f,x))$
```

とするか、それとも

```
gentran q:=:f/df(f,x)$
```

のように ":=" の代わりに ":=:" を使います。この右の ":" は右辺を評価するという GENTRAN への指示を示します。

このようにすると、

```
Q=(2.0*X**2-(5.0*X)+6.0)/(4.0*X-5.0)
```

に変換されます。

同じように、

```
v(1)=1
v(2)=2
.
.
```

のような式を出力したい場合には、

```
for i:=1:5
   gentran
      v(i):=i$
```

とすると、左辺の v の添字が評価されずに (また右辺の i も評価されない)

```
 V(I)=I
 V(I)=I
  .
  .
  .
```

となってしまいます。この場合には、代入文の両辺を評価しないといけません。上の代入文で ":=" を "::=:" に変えてやることで、求める出力が得られます。

GENTRAN で変換した結果をファイルに出力したい場合には、GENTRANOUT コマンドで

```
gentranout "test.fort";
```

の様に出力するファイル名を指定します。また、変換する言語は標準では FORTRAN になっています。 これを、例えば C 言語に変えたい場合には

```
gentranlang!* := 'c;
```

とします。 言語としては、FORTRAN,RATFOR,C,PASCAL が指定できます。

GENTRAN コマンドでは、式以外の REDUCE の文や関数定義を変換することもできます。例えば

```
gentran for i:=1:n do v(i):=0$

gentran k:=for i:=1:10 sum v(i)$
```

と入力すると

```
      DO 25001 I=1,N
       V(I)=0.0
25001  CONTINUE
      K=0
      DO 25002 I=1,10
       K=K+V(I)
25002  CONTINUE
```

の様に変換した結果が出力されます。

また FAC という REDUCE の関数を変換するには、

```
gentran
   procedure fac(n);
      begin
         declare <<fac : function; fac,n: integer>>;
         f:=for i:=1:n product i;
         declare f,i:integer;
         return f;
      end$
```

と入力します。 ここで DECLARE というのは REDUCE のコマンドではなく、GENTRAN が変換するときに FORTRAN の型宣言文を生成するためのコマンドです。結果は、

```
      INTEGER FUNCTION FAC(N)
      INTEGER N,F,I
      F=1
      DO 25003 I=1,N
      F=F*I
25003 CONTINUE
      FAC=F
      RETURN
      END
```

のように出力されます。

また、 ある決まった SUBROUTINE を作成する場合に、テンプレート・ファイルを作成しておき、それを使って FORTRAN のプログラムを生成することができます。例えば、 与えられた行列の逆行列を計算するサブルーチン INV を生成するときに、ファイル “inv.tem” を次のように作成し、

```
subroutine inv(m,minv)
   begin;
      gentran
      <<
         declare m(eval(n),eval(n)),
         minv(eval(n),eval(n)) : real*8;
         minv :=: 1/mm
      >>$
      ;end;

   return
   end
;end;
```

inv.tem テンプレートファイル

REDUCE のコマンドとして

```
n := 3;
matrix mm(n,n);
operator m;
for i:=1:n do
     for j:=1:n do mm(i,j) := m(i,j);
gentranout "inv.fort";
gentranin "inv.tem";
```

と入力すると、ファイル “inv.fort” に 3 行列の逆行列を計算するサブルーチンが生成されます。REDUCE のコマンド行で、n の値を変えることによって、他のサイズの行列に対するサブルーチンを生成することができます。

```
      SUBROUTINE INV(M,MINV)
      REAL*8 M(3,3),MINV(3,3)
      MINV(1,1)=(M(3,3)*M(2,2)-(M(3,2)*M(2,3)))/(M(3,3)*M(2,2)*M(1,1)
     . -(M(3,3)*M(2,1)*M(1,2))-(M(3,2)*M(2,3)*M(1,1))+M(3,2)*M(2,1)
     . *M(1,3)+
                .
                .
                .
      RETURN
      END
```

3 行 3 列行列の逆行列を求める Fortran サブルーチン

19.2 C プログラムの作成

GENTRAN パッケージをロードして、

```
load_package 'gentran;
gentranlang!* := 'c;
```

と入力することにより C 言語のソースプログラムを生成することができます。例えば、

```
gentran procedure fac(n);
begin
   declare <<fac:function; fac,n:integer>>;
   if n < 2 then return 1
   else return n*fac(n-1);
end;
```

と入力すると

```
int fac(n)
int n;
{
    if(n < 2.0)
        return 1.0;
    else
        return n*fac(n-1);
}
```

のような C のプログラムが出力されます。

第20章　GHYPER:一般化された超幾何関数の簡約

Victor S. Adamchik

Wolfram Research Inc.
former address :
Byelorussian University, Minsk, Byelorussia

Present REDUCE form by

Winfried Neun

ZIB Berlin

Email: *Neun@sc.ZIB-Berlin.de*

ここではREDUCEの ghyper パッケージについて説明しています。このパッケージは一般化された超幾何関数を多項式、初等関数もしくは特殊関数またはより簡単な超幾何関数への簡約を行います。従って、このパッケージはREDUCE の特殊関数のパッケージと共に使われるべきです。

20.1　はじめに

(一般化された) 超幾何関数

$$ {}_pF_q\left(\left.\begin{matrix}a_1,\ldots,a_p\\ b_1,\ldots,b_q\end{matrix}\right|z\right) $$

はテキストブックの特殊関数の項、例えば [6] で定義されています。多くのよく知られた関数がこのクラスに属しています。例えば、指数関数、対数関数、三角関数やベッセル関数等です。[36] にはこのような関数の和や基本的な性質それに応用についての記述があります。

数百の特殊な点での値が [6] に載っています。

20.2　HYPERGEOMETRIC コマンド

HYPERGEOMETRIC 演算子は三つの引数を取ります。最初の二つはパラメータリストで、最後は関数の引数です。パラメータリストは空リストであっても構いません。例えば、

```
hypergeometric ({},{},z);
```

```
 Z
E

hypergeometric ({1/2,1},{3/2},-x^2);

 ATAN(X)
---------
    X
```

20.3 HYPERGEOMETRIC 演算子の拡張

数百の特別な点での一般化された超幾何関数の値が、文献で得られます。この全てのものが HYPERGEOMETIRC 演算子で処理できるとは限りません。しかし、REDUCE の LET 文を使って、特別な点で簡約規則を定義することが出来ます。例えば、

```
let {hypergeometric({1/2,1/2},{3/2},-(~x)^2) => asinh(x)/x};
```

第21章　GNUPLOT: GNUPLOTとのインターフェース

Herbert Melenk
Konrad–Zuse–Zentrum für Informationstechnik Berlin
E–mail: *Melenk@zib.de*

21.1　始めに

GUNPLOT システムは、式やデータで与えられた曲線や曲面を標準的なグラフィック装置に出力します。GNUPLOT は `X-windows` や `SUN tools`, `VGA screen`, `postscript`, `pic` TEX といった非常に多くの出力装置をサポートしています。REDUCE の GNUPLOT パッケージを使うことにより、REDUCE から GNUPLOT の機能が使えるようになり、対話的に曲線や曲面を表示したり、図形を紙に印刷したり出来るようになります。

この版では GNUPLOT3.2 をサポートしています。多くのシステムでは、GNUPLOT のバイナリーも一緒に配布されています。しかし、これは最小限のセットのみです。GNUPLOT のすべての機能を利用するには、またあなたが使っている REDUCE に GNUPLOT のバイナリーが添付されていない場合には、以下のサーバから GNUPLOT のファイルを取ってきて下さい。

- dartmouth.edu (129.170.16.4)
- monu1.cc.monash.edu.au (130.194.1.101)
- irisa.irisa.fr (131.254.2.3)

UNIX では、REDUCE は環境変数 *gnuplot* を調べます。この環境変数が設定されていれば、REDUCE は、このディレクトリの GNUPLOT を起動します。設定されていなければ、$*reduce/plot* のディレクトリ中に存在するものと仮定します。

21.2　互換性

以前の REDUCE の GNUPLOT パッケージと比べて、現在の GNUPLOT パッケージでは、データ点の計算は REDUCE 内部で実行し、それを GNUPLOT にデータファイルの形で渡すように変更されています。これによって、GNUPLOT パッケージの使い方が単純になり、互換性を損ねることや、制限等が少なくなっています。

21.3 GNUPLOT のロード

もし、あなたの計算機で動いている REDUCE の GNUPLOT パッケージが自動的にロードされるようになっていたならば、`PLOT(...)` と入力するだけで、グラフが表示されます。そうでなければ、最初に一度、

```
load_package gnuplot;
```

を実行して下さい。

21.4 PLOT コマンド

The `PLOT(...)` コマンドは、いくつかのパラメータを取ります。

- プロットする関数、次のいずれかの形式の関数が可能です。
  - $u * \sin(u) * *2$ のような変数が一つの式
  - 変数が二つの式、例えば、$u * \sin(u) * *2 + \sqrt{(}v)$
  - 左辺がシンボルで、右辺が一つまたは二つの変数を含む式、例えば、$dome = 1/(x * *2 + y * *2)$
  - 曲線を表す、二次元または三次元の点のリスト。例えば、$\{0,0\},\{0,1\},\{1,1\}\}$
  - 曲線の組を表す、二次元または三次元の点のリストのリスト。例えば、$\{\{0,0\},\{0,1\},\{1,1\}\}\{0,0\},\{0,1\},\{1,1\}\}\}$
- 変数の動く範囲、これらは $variable = (lower\_bound\,..\,upper\_bound)$ の形式で、$lower\_bound$ と、$upper\_bound$ は必ず整数でなければなりません。もし、範囲が指定されない場合、既定値として独立変数の範囲は $(-10\,..\,10)$ に、また従属変数の範囲は GNUPLOT が扱える最大値 (ほとんどの機械では IEEE の倍精度実数を、DOS では単精度実数を使っています) に設定されます。
- GNUPLOT へのオプション。これは、$hidden3d$ のようなキーワードや、$term = pictex$ のような式があります。タイトルやラベルのようなテキストはダブルクオートで囲っておく必要があります。

数値とドットの間には空白を入れる必要があります。さもなければ、REDUCE は誤って解釈してしまいます。

もし、関数が式の形で入力された場合、左辺は、主に従属変数の座標軸のラベルとして使われます。

二次元グラフの場合、一つ以上の関数を一枚の絵に描くことが出来ます。しかし、これらの関数の独立変数はすべて同じでなければなりません。関数の内、一つは点の集合または点集合のリストで与えることができます。通常、すべての関数と、点集合は、線で描かれます。もし、関数と点集合が一枚に描かれた場合には、関数は実線で、点集合は点で描かれます。

関数式は小数点数(*rounded*)モードで計算されます。これは、自動的に行われるので、`on rounded;` を実行しておく必要はありません。

例:

```
plot(cos x);
plot(s=sin phi,phi=(-3 .. 3));
plot(sin phi,cos phi,phi=(-3 .. 3));
plot (cos sqrt(x**2 + y**2),x=(-3 .. 3),y=(-3 .. 3),hidden3d);
plot {{0,0},{0,1},{1,1},{0,0},{1,0},{0,1},{0.5,1.5},{1,1},{1,0}};

 % parametric: screw

on rounded;
w:=for j:=1:200 collect {1/j*sin j,1/j*cos j,j/200}$
plot w;

 % parametric: globe
dd:=pi/15$
w:=for u:=dd step dd until pi-dd collect
    for v:=0 step dd until 2pi collect
      {sin(u)*cos(v), sin(u)*sin(v), cos(u)}$
plot w;
```

次の GNUPLOT のオプションをサポートしています。

- $title = name$: 表題 (文字列) をグラフの上に書く。
- axes labels: $xlabel$ = "$text1$", $ylabel$ = "$text2$", そして、曲面に対しては、 $zlabel$ = "$text3$"。もし、省略された場合、座標軸には、式の独立変数名および従属変数名がそれぞれ付けられます。ここで、座標軸名、$x$label, $y$label それに $z$label は普通に使われる名前の取り方に従っています。つまり、二次元の場合、x は水平方向の軸を表し、y は垂直方向の軸を表しています。また、三次元の場合、z は直角方向の軸を表します。

```
plot(1,X,(4*X**2-1)/2,(X*(12*X**2-5))/3,
     x=(-1 .. 1), ylabel="L(x,n)",
     title="Legendre Polynomials");
```

- $terminal = name$: 出力をデバイス $name$ に切り替えます。 最初の設定は、出力は端末に出すようになっています。プリンター等、別デバイスに出力させるするには、GNUPLOT のマニュアルをよく読んで下さい。GNUPLOT の set terminal コマンドに引数を指定する必要があれば、引数も含めて、REDUCE の文字列として指定して下さい。
- $size$ = "s_x, s_y": (ウインドゥではなく) グラフを縮小/拡大する。s_x, s_y は、それぞれ x, y 方向の縮小率を表します。既定値は、$s_x = 1, s_y = 1$ です。拡大率を 1.0 以上に取ると、大きすぎることがしばしば起こります。

```
plot(1/(x**2+y**2),x=(0.1 .. 5),
        y=(0.1 .. 5), size="0.7,1");
```

- *view* = "r_x, r_z": 三次元の作図で、(x,z) 平面での、視点の設定を行います。値は、角度で (degree) で与えます。既定値は、 $r_x = 60, r_z = 30$.

```
plot(1/(x**2+y**2),x=(0.1 .. 5),
        y=(0.1 .. 5), view="30,130");
```

- *contour* 対 *nocontour*: 三次元グラフで、等高線を追加して 書く。 (既定値は、*nocontour* です)
- $surface$ 対 $nosurface$: 三次元グラフで、表面を描くか、描かないか。(既定値は $surface$ です)
- *hidden3d*: 三次元グラフを描くときに、陰線処理を行う。(隠れた線は描かない)

21.5 メッシュの生成

通常、関数はあらかじめ決められた点、範囲を *plot_xmesh* と、*plot_ymesh* に分割した点、での関数値を計算して描画します。この値は、最初 20 に設定されています。これらの変数 *plot_xmesh* , *plot_ymesh* は、シェアー変数であり、この変数に値を直接代入することによって、メッシュサイズを変更することが出来ます。これらの値は、plot コマンドから直接代入することは出来ません。また、名前の *plot_xmesh* , *plot_ymesh* は、システムに組み込んであるので、自分が使っている式での変数名として、x, y 以外の名前を使っていたとしても、変更できません。

二次元グラフの描画では、曲線があまり急峻であると、特に特異点を持っているような場合、その部分がなめらかになるように細かく描かれます。しかし、特に複雑な式に対して、このようなリファインは計算時間を浪費します。`off plotrefine;` として、これを止めることが出来ます。この時には、特異点でグラフは中断されます。

三次元の場合には、曲面は固定した格子点上でしか計算されず、このようなリファインは行われません。特異点を持つ場合には、近傍の値を調べ、それが計算可能であれば、その値で打ち切った値を代わりに使います。計算できなければ、値はゼロとして描画されます。

三次元曲面を描画するとき, 、陰線消去を行うことを指定することが出来ます。 Gnuplot 3.2 の"hidden3d"のエラーのため、座標軸のラベルは正しく描かれません。そのため、ラベルは描かないようになっています。点集合で関数が定義された場合には、陰線消去は出来ません。

21.6 GNUPLOT の操作

`PLOTRESET;` コマンドは、現在の GNUPLOT の出力画面を消去します。次に`PLOT` を呼び出すと、新しい画面が開きます。

GNUPLOT が、出力パイプ (UNIX や Windows の場合) によって直接起動された場合、GNUPLOT のデータ転送にエラーが生じた場合、GNUPLOT は終了してしまいます。REDUCE は、

パイプが壊れていることを検出できないので、ユーザがPLOTRESET; を実行して、関数をリセットして下さい。REDUCE は、新しいグラフ出力を生成します。

DOS Windows 3.1 のもとでの GNUPLOT は、テキスト画面とグラフ画面の二つの画面を表示します。もし、テキスト画面を見たくないのでしたら、その画面をアイコン化し、グラフ画面のシステムメニューの “update wgnuplot.ini” をクリックして下さい。このようにすると、現在の画面の設定 (グラフ画面の大きさも含めて) が保存されます。REDUCE を終了する前に、PLOTRESET; を実行してグラフ画面を終了することを忘れないようにして下さい。もし、グラフ画面を終了し忘れて、REDUCE を終了してしまった時には、GNUPLOT のテキスト画面のコマンドボタンを使って下さい。

21.7 GNUPLOT のコマンド列の保存

もし生成した GNUPLOT コマンドの列を一回以上使いたい場合 (例えば、出版物のための絵を作成する場合等) には、次のようにして下さい。

$ON\ TRPLOT, PLOTKEEP$;

$TRPLOT$ は、GNUPLOT コマンド列を REDUCE の標準出力にも出力することを指定します。通常、データファイルは GNUPLOT を呼び出した後に消去されます。しかし、$PLOTKEEP$ をオンにしておくと、このファイルは消去されません。

21.8 GNUPLOT を直接呼び出す

GNUPLOT は多くの機能を持っており、これまでに述べてきた関数やパラメータだけでは、そのすべてを利用することは出来ません。このため、REDUCE から、GNUPLOT を直接呼び出すことが可能なようにしています。GNUPLOT の機能やパラメータについては、GNUPLOT のマニュアルを参照して下さい。REDUCE からの、一般的な呼び出し方は、

$gnuplot(< cmd >, < p_1 >, < p_2 > \cdots)$

です。ここで、$< cmd >$ は、コマンド名で、$< p_1 >, < p_2 > \cdots$ はパラメータです。パラメータは、REDUCE によって評価され、その結果が GNUPLOT に渡されます。通常、作図はコマンドの列から構成され、それらは REDUCE もしくはオペーティングシステムによってバッファされます。バッファリングを止め、コマンドを実行させるために、$plotshow$; を実行して下さい。例えば、

```
gnuplot(set,polar);
gnuplot(set,noparametric);
gnuplot(plot,x*sin x);
plotshow;
```

このような使い方をする場合、変数名や関数名に対する GNUPLOT の制限に注意して下さい。

第22章　GROEBNER: Gröbner基底の計算

H. Melenk & W. Neun
Konrad–Zuse–Zentrum
für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin–Dahlem
Germany
Email: *melenk@ZIB.DE*
and
H.M. Möller
FB Mathematik
Universität Dortmund
D–44221 Dortmund
Germany
Email: *Michael.Moeller@Math.Uni-Dortmund.DE*

Gröbner 基底は多変数多項式に関連した問題、例えば代数方程式の解を求めるとか、多項式イデアルを求めるような問題を解く場合に非常に有用な道具である。Gröbner 基底の定義やこれに関する応用については、例えば、論文 [13] を参照してください。また例については文献 [10], や [14] またこのパッケージのテストプログラムにあります。

Groebner パッケージは Buchberger の算法を用いて Gröbner 基底を計算します。これは色々な係数体上で、また異なった変数や項の順序で基底の計算ができます。

現在のバージョンは R. Gebauer, A.C. Hearn, H. Kredel と H. M. Möller によって書かれた以前のプログラムの一部を使っています。現在のバージョンでインプリメントされている算法については [23] と [26] に解説されています。

REDUCE 3.3 での Gröbner パッケージとの非互換性:

- 以前の版とは異なり、Gröbner 基底に現れる多項式には分数係数は現れません。分数形式での多項式は、それぞれの多項式をその主係数で割るか、`ON RATIONAL` と設定することで得られます。
- `RATIONAL` がオフであれば、`GREDUCE` と `PREDUCE` は入力された式をそのまま扱うのではなく、分数係数が現れないように、適当に整数倍した多項式で簡約を行います (“擬簡約”)。
- 項の順序を指定するモードを整理して、文献で使われる名前と一致するようにしました。

INVLEX → LEX,
INVTOTALDEGREE → GRADLEX,
TOTALDEGREE → REVGRADLEX

互換性から、古い名前 (LEX を除く) も使えるようになっています。

22.1 背景

22.1.1 変数、ドメインと多項式

Gröbner パッケージでの関数は方程式や多項式を扱います。方程式は内部では右辺式を左辺に移項することにより、多項式として扱われます。

計算は係数のドメイン D 上の変数 $x_1, \ldots, x_n$ に関する多項式環

$$D[x_1, \ldots, x_n],$$

で行われます。ここで D は体もしくは零要素を持たない環です。変数の集合 $x_1, \ldots, x_n$ はオプションのパラメータで指定することもできますし、入力された式から自動的に選択させることもできます。

REDUCE のカーネル “変数” として扱われます。例えば、

```
x y z22 sin(alpha) cos(alpha) c(1,2,3) c(1,3,2) farina4711
```

ドメイン D は REDUCE のドメインに変数の集合に現れないカーネルを追加したものになります。つまり D は変数の集合に現れないカーネルの多項式を要素とする体もしくは環になっています。もし、変数の集合が入力された式から自動的に選択された場合には、D は計算される時点での REDUCE の係数ドメインに一致します。変数リストに現れないカーネルは “パラメーター” として扱われます。例えば、

$a*x+(a-b)*y**2$ では $\{x, y\}$ は “変数として”
そして a と b は “パラメーター” として扱う。

現在の版の Buchberger の算法は二つのモードを持っています。体モードと環モードです。もし D が環であれば環モードが使われ、それ以外の場合は体モードが使われます。通常体モードでの計算は全ての係数が数値で REDUCE の計算モードが体 (例えば有理数やモジュラー数) の場合に行われます。一般的に言って環モードで計算する方が速い (両方のモードで計算可能な場合) ので、入力された式に分数を含んでいるような場合でも REDUCE のドメインを特に選択していないときには、環モードが使われます。入力された式は適当な定数倍され、整係数多項式に変換されます。

22.1.2 項の順序

Gröbner 基底の理論では、多項式の各項には順序が付いているものとして扱います。現在のパッケージではいくつかの順序で計算することができます。それらの順序には次にあげる順序が含まれています:

lex, gradlex, revgradlex

全ての順序は変数間の順序に基づいています。それぞれの変数の対 (a, b) に対して、例えば、"$a \gg b$" のような、順序が定義されていなければいけません。ここで記号 $\gg$ は変数間の数値の大小関係を表しているのではなく、"a" が "b" より先に置かれる、もしくは "a" の方が "b" より複雑である、ということを意味しています。

変数の列はこの順序を指定しています。従って、

$$\{x_1, x_2, x_3\}$$

は同時に変数間の順序

$$x_1 \gg x_2 \gg x_3$$

をも指定しています。

LEX 順序で項 (変数のべきを掛けたもの) を整列すると、順序の高い変数含む項が先に来ます。また同じ変数を含んでいる場合は指数べきが大きい方が先に来ます。GRADLEX 順序の場合は、まず項に含まれる全ての変数の指数部の和で比較が行われ、それが同じ値の項に付いては LEX 順序で並べられます。REVGRADLEX 順序では、まず項に含まれる変数の指数部の和 (全次数) で比較されます。全次数が同じ項に関しては、LEX 順序が逆に適応されます。この順序は Buchberger によって最初に使われました。

変数 $\{x, y, z\}$ での例:

LEX:

$x*y**3$	$\gg$	$y**48$	(変数の順序が先)
$x**4*y**2$	$\gg$	$x**3*y**10$	(最初の変数の次数が高い)

GRADLEX:

$y**3*z**4$	$\gg$	$x**3*y**3$	(全次数が高い)
$x*z$	$\gg$	$y**2$	(全次数が同じ)

REVGRADLEX:

$y**3*z**4$	$\gg$	$x**3*y**3$	(全次数が高い)
$x*z$	$\ll$	$y**2$	(全次数が同じ 従って LEX の逆順になる)

項の順序の記述は [33] と同じです。この記述では項の指数部分のみを考え、現れない変数については 0 とした整数のベクトルで表します。

$(e) = (e_1, \ldots, e_n)$ は次の式を表す $x_1**e_1\ x_2**e_2 \cdots x_n**e_n$.
$\deg(e)$ は (e) の要素の和
$(e) \gg (l) \Longleftrightarrow (e) - (l) \gg (0) = (0, \ldots, 0)$

LEX:

$$(e) \gg_{lex} (0) \implies e_k > 0 \text{ と } e_j = 0 \text{ for } j = 1, \ldots, k-1$$

GRADLEX:

$$(e) \gg_{gradlex} (0) \implies \deg(e) > 0 \text{ または } (e) \gg_{lex} (0)$$

REVGRADLEX:

$$(e) \gg_{revgradlex} (0) \implies \deg(e) > 0 \text{ または } (e) \ll_{lex} (0)$$

LEX 順序は通常の REDUCE で KORDER で変数の順序を指定したときの項の順序と同じです。

Gröbner パッケージの通常の順序は LEX 順序です。

項の順序モードに関してこれ以上触れることはこのマニュアルの範囲を越えています。これについては文献 [14] を参照してください。

変数リストは、TORDER 宣言で宣言されます。もしこの宣言がない場合、または空リストを宣言した場合には、変数は与えられた式から自動的に選択されます。また変数の順序は REDUCE の通常の順序に従います。この場合には、KORDER による順序の設定により、順序を変更することができます。

Gröbner 基底の計算の結果は、計算が行われたときに有効であった項の順序モードと変数の順序に対してのみ代数的に正しい基底になっています。以前に計算した結果を、さらに Gröbner パッケージの演算子を使って引き続いて操作を行おうとする場合には、注意しておかなくてはいけません。変数の順序を明示的に指定していない時には、正しい結果が得られないかも知れません。このような計算を行なう場合には、TORDER 宣言で、明確に変数リストと順序を指定しておく必要があります。一度、TORDER 宣言で定義された変数および順序は、再び TORDER で宣言し直すまで有効です。GVARS 演算子を使うことによって、与えられた多項式の組から、必要な変数のリストを作ることができます。

22.1.3 Buchberger の算法

このパッケージが使っている Buchberger の算法は Gebauer/Möller [26] に基づいています。アルゴリズムの拡張に関しては [35] に述べられています。

22.2 パッケージのロード

Gröbner パッケージをロードするには次のコマンドを使います。

```
load groebner;
```

このコマンドはインプリメントによって変わっているかも知れません。

このパッケージは多くの演算子や計算を制御するためのスイッチを含んでいます。これらについては次の節で説明します。

22.3 基本的な演算子

22.3.1 項の順序モード

TORDER (*vl*,*m*,[$p_1, p_2, \ldots$]); ここで、*vl* は変数リスト (もし、変数名を明示的に指定しないなら、空リスト) で、*m* は項の順序モードの名前で LEX、GRADLEX もしくは REVGRADLEX(もしくは他のインプリメントされているモード名) のいずれかを指定します。そして、[$p_1, p_2, \ldots$] は項順序モードに対する追加のパラメータです (基本モードの場合には必要ありません)。

TORDER は変数の集合と項の順序モードを設定します。何も指定しないときは LEX(辞書式順序) が使われます。以前に設定された変数の集合と順序モードがリストの形式で返されます。この返された値は、後になって TORDER の引数としてそのまま渡すことができます。

もし、変数リストが空リストであれば、または TORDER で項順序モードを設定しなければ、変数の集合は自動的に作成されます。

GVARS ($\{exp_1, exp_2, \ldots, exp_n\}$);

ここで $\{exp_1, exp_2, \ldots, exp_n\}$ は数式もしくは方程式のリストです。

GVARS は 数式 $\{exp_1, exp_2, \ldots, exp_n\}$ からカーネルを取り出します。これは Gröbner 基底の計算において変数として扱われます。これは、例えば、TORDER での宣言等に使われます。

22.3.2 GROEBNER: Gröbner 基底の計算

GROEBNER $\{exp_1, exp_2, \ldots, exp_m\}$;

ここで $\{exp_1, exp_2, \ldots, exp_m\}$ は数式もしくは方程式のリストです。

GROEBNER は与えられた数式の Gröbner 基底を TORDER で設定されたモードに基づいて計算します。

Gröbner 基底 {1} は、入力された多項式から生成されるイデアルが多項式環全体になるということを意味します。また、これは入力された多項式系が共通零点を持たないことを意味します。

副作用として、計算に使われた変数のリストが変数 GVARSLAST に保存されます。変数が入力した式から自動的に取り出された場合や最適化を行ったために変数の順序が変更された場合、計算された Gröbner 基底を他の計算 (例えば PREDUCE 演算子等) に使うときには、必ず TORDER で変数のリストと項順序モードを改めて指定しなければなりません。基底は計算が行なわれたのと同じ変数集合と項順序に対してのみ “Gröbner” 基底の性質を持ちます。

例

```
torder({},lex)$
groebner{3*x**2*y + 2*x*y + y + 9*x**2 + 5*x - 3,
2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x + 3,
x**3*y + x**2*y + 3*x**3 + 2*x**2 };
```

```
             2
    {8*x - 2*y  + 5*y + 3,

        3      2
     2*y  - 3*y  - 16*y + 21}
```

この例ではシステムの通常の順序 $\{x, y\}$ を使って計算しています。別の順序を使えば異なった基底が得られます。

```
torder({y,x},lex)$
groebner{3*x**2*y + 2*x*y + y + 9*x**2 + 5*x - 3,
2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x + 3,
x**3*y + x**2*y + 3*x**3 + 2*x**2 };

             2
    {2*y + 2*x  - 3*x - 6,

        3      2
     2*x  - 5*x  - 5*x}
```

また別の項の順序モードを指定した場合も、異なった Gröbner 基底が得られます。

```
torder({x,y},revgradlex)$
groebner{3*x**2*y + 2*x*y + y + 9*x**2 + 5*x - 3,
2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x + 3,
x**3*y + x**2*y + 3*x**3 + 2*x**2 };

   2
{2*y  - 5*y - 8*x - 3,

 y*x - y + x + 3,

   2
 2*x  + 2*y - 3*x - 6}
```

GROEBNER の計算は次のスイッチによって制御することができます。

GROEBOPT – オンにした場合、変数の順序は計算時間が少なくなるように最適化されます。これに使われる算法は [10] に記述されています。最終的な変数の順序は GVARSLAST に保存されます。

DEPEND 宣言によって依存関係が宣言されている変数に関しては、その順序は最適化されません。例えば

$$depend\ a,\ x,\ y;$$

とすると変数 a が x や y よりも先にくることを保証します。従って、GROEBOPT スイッチをある特定の変数を消去[1] したいような場合でも使うことができます。

通常は GROEBOPT はオフになっており、指定した変数の順序に基づいて計算を実行します。

GROEBFULLREDUCTION – もしオフにすると、GROEBNER で行われる簡約は純粋に主項のみに限られます。もし残りの項についても簡約が必要なら、別個に行う必要があります。

通常 GROEBFULLREDUCTION はオンになっています。

GLTBASIS – もしオンであれば、得られた基底から主項部分が取り出され、変数 GLTB に保存されます。

GLTERMS – もし、$\{exp_1, \ldots, exp_m\}$ がパラメータ (変数リストに含まれないシンボル) を持っている場合、シェアー変数 GLTERMS には、計算中にゼロでないと仮定された式のリストが代入されます。計算された Gröbner 基底は、ここに含まれた全ての式がゼロでないパラメータ領域でのみ、正しい基底になっています。

次のスイッチは GROEBNER の出力を制御します。通常は全てのスイッチはオフで、何も出力されません。

GROEBSTAT –計算の要約が出力されます。これには、計算時間、生成された中間の H–多項式の数、および判定条件に合った場合の数等が含まれています。

TRGROEB – これは GROEBSTAT の出力に加えて、中間の H–多項式も出力されます。

TRGROEBS – TRGROEB に加えて中間で生成された S–多項式も出力します。

TRGROEB1 – 内部で生成している対リストが変更されたとき出力されます

22.3.3 GZERODIM?: 零次元イデアルのテスト

GZERODIM!? *bas*

ここで *bas* は現在設定されている順序モードと指定された変数に対する Gröbner 基底を渡します。結果は、もし *bas* が多項式のイデアルの共通零点の次元が 1 次元以上であれば NIL を返されます。もしイデアルが零次元であれば、つまりイデアルに属する多項式の共通零点が有限個の場合、結果は重複度も含めた零点の数 k が返されます。

22.3.4 GDIMENSION, GINDEPENDENT_SETS: 次元と独立変数の計算

次の演算子は、任意の項順序に対する Gröbner 基底 *bas* のイデアルの次元と独立変数の集合を求めます。

Gdimension *bas*

[1] 項の消去は順序として先の変数をまず消去するように行われます。従って、この例での順序では可能であれば変数 a がまず消去されます。

Gindependent_sets *bas*

Gindependent_sets は、イデアルに対してその基底での自由パラメータに対応する、最大左独立変数の集合を求めます。これは、変数リストの部分集合になっています。結果は、これらの集合のリストです。ゼロ次元イデアルに対してはこれは空集合になります。

GDimension は、イデアルの次元を計算します。これは、独立変数の集合の大きさになります。

ここでは、"Kredel-Weispfenning" のアルゴリズム (see [32] を [8] にあるように、一般の順序に拡張したものを使って計算しています。

22.3.5 Gröbner 基底の変換

GLEXCONVERT: 辞書式順序での基底への変換

GLEXCONVERT $(\{exp_1, \ldots, exp_m\}\,[, \{var_1 \ldots, var_n\}]\,[, MAXDEG = mx]$
$[, NEWVARS = \{nv_1, \ldots, nv_k\}])$

ここで $\{exp_1, \ldots, exp_m\}$ は、変数 $\{var_1, \ldots, var_n\}$ と現在有効である項の順序モードでの Gröbner 基底で、mx は整数、$\{nv_1, \ldots, nv_k\}$ は基底に含まれる変数の部分集合です。

GLEXCONVERT は、零次元イデアル (有限個の孤立解を持つイデアル) を任意の項順序から辞書式順序での基底に変換します。GLEXCONVERT の引数に渡す Gröbner 基底は現在有効な項順序モードでの基底になっていなければいけません。

NEWVARS は新しい変数とその順序を指定します。もし省略されれば、元々の変数とその順序が使われます。これに元々の変数リストの一部のみを指定すると、部分的なイデアル基底が評価されます。一変数多項式の計算では、NEWVARS は要素が一つのリストでなけれなりません。

MAXDEG は次数の上限です。もしこの上限に達するとエラーを表示して計算を中止します。

もしイデアルが零次元でなければ警告が出力されます。

GLEXCONVERT は Faugère, Gianni, Lazard と Mora [23] による FLGM 算法を使っています。通常、Gröbner 基底の計算は、まず全次数順序での基底を求め、それを辞書式順序での基底に変換した方が、直接辞書式順序で求めるよりも効率的です。さらに、GLEXCONVERT は、辞書式順序での基底を別の変数の順序での基底に変換するために使うこともできます。また一変数多項式での計算もサポートしています。このような多項式が存在する場合には零次元でない場合にも使うことができます。

例

```
torder({{w,p,z,t,s,b},gradlex)

g  :=  groebner  { f1 := 45*p + 35*s -165*b -36,
       35*p + 40*z + 25*t - 27*s, 15*w + 25*p*s +30*z -18*t
      -165*b**2, -9*w + 15*p*t  + 20*z*s,
```

```
        w*p + 2*z*t - 11*b**3, 99*w - 11*s*b +3*b**2,
        b**2 + 33/50*b + 2673/10000};

  g := {60000*w + 9500*b + 3969,

      1800*p - 3100*b - 1377,

      18000*z + 24500*b + 10287,

      750*t - 1850*b + 81,

      200*s - 500*b - 9,
             2
      10000*b  + 6600*b + 2673}

   glexconvert(g,{w,p,z,t,s,b},maxdeg=5,newvars={w});

               2
    100000000*w  + 2780000*w + 416421

   glexconvert(g,{w,p,z,t,s,b},maxdeg=5,newvars={p});

         2
    6000*p  - 2360*p + 3051
```

GROEBNER_WALK: 全次数順序から辞書式順序への変換

アルゴリズム *groebner_walk* は全次数順序もしくはただ一つの重みベクトル (例えば $\{1,1,1,\ldots\}$) からなる重み付け順序で計算された任意の多項式系の基底を同じ変数順序での *lex* 順序による基底に変換する。この計算は、対応する単項イデアルの Gröbner 基底の列を計算し、おのおのについて元の系に持ち上げることによって行なっている。このアルゴリズムについては、[4],[5],[3],[15] にもっと一般的に記述されている。*lex* での Gröbner 基底が直接計算できなかった時のみ、*groebner_walk* を呼ぶべきである。*groebner_walk* の計算は、割算を含んだ計算等、幾つかのオーバーヘッドを含んでいる。

groebner_walk g

g は多項式イデアルの基底で、*gradlex* もしくはある一つの要素に関して *weighted* 順序 (ゼロでないただ一つの重みベクトル) で計算されたものである。結果は、イデアルの次数によらず (ゼロでないイデアルに対しても) *lex* での基底 (もしこれが計算可能であれば) である。変数の順序を定義するためには、前もって演算子 *torder* を呼び出しておく必要がある。*groebner_walk* を *groebopt* がオンの状態で呼び出さないで下さい。変数の順序は *torder* への呼び出しから変更なしで取り出されます。変数 *gvarslast* は設定されない。

22.3.6 GROEBNERF: Gröbner 基底の因数分解

背景

もし Gröbner 基底を計算する目的が、方程式系を解くとか、多項式系の共通零点を求めるような場合には、因数分解を使う Buchberger の算法が使えます。これは理論的には単純な考えに基づいています。つまり、もし多項式 p が二つの (もしくはそれ以上の) 多項式の積として $p = f * g$ と表されるとすると、p が零になるには f か g のいずれかが零になるときに限られます。つまり、p の零点は f の零点と g の零点の和集合になります。従って Gröbner 基底の計算において、ある多項式がこのように分解できる場合は、求める基底はそれぞれの因子を使って計算した基底から求めることができます。このような計算の結果は (部分的な)Gröbner 基底のリストです。元々の方程式の解は、重複度を除けば、それぞれの解の和集合になります。

GROEBNERF 演算子

GROEBNERF 演算子の構文は GROEBNER と同じです。

$$GROEBNERF(\{exp_1, exp_2, \ldots, exp_m\}[, \{\}, \{nz_1, \ldots nz_k\});$$

ここで $\{exp_1, exp_2, \ldots, exp_m\}$ は数式または方程式のリストで、$\{nz_1, nz_2, \ldots, nz_n\}$ はオプションの非ゼロと分かっている多項式のリストです。

GROEBNERF は多項式を因数分解し、それぞれの因子に対して基底を計算します。結果は部分的な Gröbner 基底のリストです。もし因数分解ができない場合や一つの場合を除いて他の全てが自明な基底 {1} になる場合、結果はただ一つの基底になります。それ以外の場合は、結果は多項式のリストのリストです。もし、解が存在しなければその結果は {{1}} になります。重複度 (2 次以上の因子を持つとか、同じ基底が二つ以上現れる場合) は、計算時間を早めるために、可能な限り早く取り除かれます。因数分解はいくつかのスイッチで制御することができます。

副作用として、変数のリストは GVARSLAST に保存されます。また、GLTBASIS がオンであれば基底の主係数のリストが生成され変数 GLTB に保存されます。

GROEBNERF の三番目の引数は、ゼロでない多項式のリストを与えます。もし、計算中、与えた多項式がゼロになるような分岐が現れた場合、その分岐は無視されます。これを指定することによって、計算時間を大幅に節約することができます。この時、第二引数として空リストを指定しておく必要があります。

```
torder({x,y},lex)$
groebnerf { 3*x**2*y + 2*x*y + y + 9*x**2 + 5*x = 3,
            2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x = -3,
             x**3*y + x**2*y + 3*x**3 + 2*x**2 \};
```

```
    {{y - 3,x},

                    2
 {2*y + 2*x - 1,2*x  - 5*x - 5}}
```

この結果から、解の方程式は直ちに得られます。

GROEBNER で使われている以下のスイッチは同じように GROEBNERF でも使用することができます。

GROEBOPT
GLTBASIS
GROEBFULLREDUCTION
GROEBSTAT
TRGROEB
TRGROEBS
TRGROEB1

GROEBNERF での追加のスイッチ:

GROEBRES – オンの場合、特定の条件が満たされれば、終結式の計算が行われます。(二変数の H–多項式が続いている場合。) これによってしばしば計算時間を短縮することができます。

通常は GREBRES はオフになっています。

TRGROEBR – 全ての中間に生成された基底は出力されます。

通常は TRGROEBR はオフになっています。

GROEBMONFAC GROEBRESMAX GROEBRESTRICTION
これらの変数については次の節で説明します。

単項因子の抑制

GROEBNERF での因数分解は、次の スイッチや変数により制御されます。変数 GROEBMONFAC は“単項因子” の取り扱いと関係しています。単項因子とは変数のべきの積が多項式の因数として現れることで、例えば $x**2*y$ は多項式 $x**3*y-2*x**2*y**2$ の単項因子です。単項因子を持っているということは、解の中に “$x=0$ または $y=0$” のようなものがある重複度で含まれていることを意味します。GROEBNERF はこのような単項因子については、その重複度を変数 GROEBMONFAC で指定された値まで下げます。この変数の値は通常 1 になっています。(つまり、単項因子は残されますが、重複している因子は一つを除いて他はすべて取り除かれます。) GROEBNONFAC := 0 とすると、単項因子はすべて取り除かれます。

例:

Ebert/Deuflhard/Jaeger (1981) [20]) で扱われている、ピリジンに対する化学反応方程式から得られる方程式系:

$$
\begin{aligned}
f1 &:= -1*A + p9*B;\\
f2 &:= p1*A - p2*B - p3*C*B + p7*D - p9*B + p10*D*F;\\
f3 &:= p2*B - p3*B*C - 2*p4*C*C - p6*C + p8*E\\
&\quad + p10*D*F + 2*p11*E*F;\\
f4 &:= p3*B*C - p5*D - p7*D - p10*D*F;\\
f5 &:= p4*C*C + p5*D - p8*E - p11*E*F;\\
f6 &:= p3*B*C + p4*C*C + p6*C - p10*D*F - p11*E*F;\\
f7 &:= p6*C + p7*D + p8*E;\\
polys &:= \{f1, f2, f3, f4, f5, f6, f7\}\$\ vars := \{A, B, C, D, E, F\}\$\\
groebmonfac &:= 1; \%\text{単項因子は重複を除いて一つの因子だけにする}\\
res &:= groebnerf\ (polys,vars);
\end{aligned}
$$

```
RES := {{A,E,B,D,C},

    {A, - E*P8 - C*P6,B,F*P6*P11 + C*P4*P8 + P6*P8,D}}
```

```
% 上の結果は二つの基底が得られます。この二つの基底で
% A,B, と D は共通に含まれており、強制的に零にします。
```

groebmonfac := 0; % 単項因子はすべて取り除く

$$res := groebnerf\ (polys,vars);$$

```
res := {{1}};

% この構成では解は持ちません。
% (変数の値が零になる解しかない。)
```

結果の式の制限

変数 GROEBRESMAX は結果の個数を制御します。通常この変数の値は 300 になっています。もし、部分的な結果の個数が GROEBRESMAX の値を越えると、計算はそこで中止されます。

解空間への制限

ある応用に対しては、与えられた方程式の全ての解ではなく、その一部のみが必要な場合があります。例えば、非負もしくは正の解しか必要でないという場合があります。このような制限をつけて計算を行うことにより、不要な解の分岐を調べる必要がないので、計算時間が大幅に節約できます。

正値: もし、ある一つの多項式が正のゼロ点を持たないのであれば、全体の解は、正の解しか持ちません。Buchberger のアルゴリズムは、もし必要であれば、多項式の係数が全て同じ符号を

持つかどうかを調べます。例えば、$13*x+15*y*z$ が x, y と z の非負の値に対してゼロになるのは、$x=0$ であり、$y=0$ または $z=0$ となる場合だけです。これは、“制限による因数分解”です。多項式、$13*x+15*y*z+20$ は非負の変数の値に対してゼロにはなりません。

ゼロ点: もし、イデアルに含まれる多項式の中に絶対値を取るものがあった時、イデアルは、原点を共通解としては持ちません。

シェアー変数

GROEBRESTRICTION

に設定することにより、GROEBNERF 演算子にどのような制約条件を付けるのかを指定できます。

GROEBRESTRICTION:=NONEGATIVE;
非負の実数解のみを求める。

GROEBRESTRICTION:=POSITIVE;
零でない非負の解のみを求める。

GROEBRESTRICTION:=ZEROPOINT;
解として、$\{0, 0, \ldots, 0\}$ を含むようなものだけを求める。

もし GROEBNERF はこの制約条件に反するような多項式を見つけると、その選択肢に対する計算を中止します。

22.3.7 GREDUCE, PREDUCE: 多項式の簡約

背景

与えられた多項式の集合 “B” を法として多項式 “p” を簡約することは、Buchberger の算法によって具体化された簡約算法で行えます。この方法は、体上の多項式の場合に次のようになります。

```
loop1:                                  % 頭項消去
     if(B の中に多項式 b が存在して、
        p の主項が b の主項の倍数になっているとき)
     do
        p := p - lt(p)/lt(b) * b        %主項を消去)
       (この loop1 を消去が可能な間繰り返す);
loop2:                                  % 引き続く項を消去
     p の各項 s に対して
         do
             if(B 中に多項式 b が存在して
                s が b の主項の倍数になっているとき)
                do
                     p := p - s/lt(b) * b  (項 s を消去)
       (この loop2 を消去が可能な間繰り返す);
```

もし係数が体ではなく、零因子を持たない環の要素の場合には、上の算法で割り算を行っている部分が実行できない場合が出てきます。しかし、体の場合 p が q に簡約される場合は任意の数 c に対して $c*p$ は $c*q$ に簡約されます。これを使うと、環の場合において適当な数 c を選んで、$c*p$ の簡約を行うことによって、上の算法での割り算が実行できるようにできます。この簡約の結果は p の擬簡約と呼ばれます。

Gröbner 基底による簡約

$$GREDUCE(exp, \{exp_1, exp_2, \ldots, exp_m\}]);$$

ここで exp は数式で、$\{exp_1, exp_2, \ldots, exp_m\}$ は任意の数の式もしくは方程式のリストです。

GREDUCE はまず式のリスト $\{exp_1, \ldots, exp_n\}$ を Gröbner 基底に変換します。その後、与えられた式をこの基底を法として簡約します。もし、与えられた式のリストが両立しない場合にはエラーになります。結果は与えられた式を簡約した結果が返されます。この GREDUCE 演算子の副作用として、GROEBNER 演算子の場合と同様に変数 GVARSLAST に変数のリストを保存します。

任意の多項式系に関する簡約

$$PREDUCE(exp, \{exp_1, exp_2, \ldots, exp_m\});$$

ここで exp は数式で、$\{exp_1, exp_2, \ldots, exp_m\}$ は任意の数の式もしくは方程式のリストです。

PREDUCE は与えられた式のリスト $\{exp_1, \ldots, exp_m\}$ を法として与えられた式を簡約します。もし、このリストが Gröbner 基底になっていれば、結果はだだ一つに決まります。しかし、Gröbner 基底でなければ、結果は簡約するステップの実行順序によって異なります。PREDUCE はリスト $\{exp_1, exp_2, \ldots, exp_m\}$ が変数の順序に関して Gröbner 基底になっているかどうかは調べません。もし、リストがそれ以前に違った変数の順序の元で求めた Gröbner 基底になっているのであれば、変数とその順序を引数として与えなければいけません。

例 (Gröbner 基底に対する PREDUCE の例)

```
torder({x,y},lex);
gb:=groebner{3*x**2*y + 2*x*y + y + 9*x**2 + 5*x - 3,
             2*x**3*y - x*y - y + 6*x**3 - 2*x**2 - 3*x + 3,
             x**3*y + x**2*y + 3*x**3 + 2*x**2}$
preduce (5*y**2 + 2*x**2*y + 5/2*x*y + 3/2*y
          + 8*x**2 + 3/2*x - 9/2, gb);

    2
   y
```

簡約の道筋

ある場合には GREDUCE や PREDUCE で簡約された結果ではなく、簡約された式に到達するまでの手続きが知りたい場合があります。もし、GROEBPROT スイッチがオンになっていると、GREDUCE や PREDUCE は変数 GREOBPROTFILE に簡約するプロセスを保存します。この変数の値は、方程式のリストで、変数 "candidate" で簡約しようとする式を表しています。多項式系は "$poly1$", "$poly2$", ... で表されています。これらの方程式を代入文として考えると、これを順次実行していくと、与えられた方程式から簡約された結果を出すようなプログラムとして見ることができます。係数のドメインが環の場合には、擬簡約を行うために式全体が何倍かされます。

例

on groebprot $
preduce $(5 * y * *2 + 2 * x * *2 * y + 5/2 * x * y + 3/2 * y + 8 * x * *2$
$+3/2 * x - 9/2, gb)$;

```
 2
y
```

groebprotfile;

```
              2         2                     2
{candidate=4*x *y + 16*x  + 5*x*y + 3*x + 10*y  + 3*y - 9,

          2
 poly1=8*x - 2*y  + 5*y + 3,

          3      2
 poly2=2*y  - 3*y  - 16*y + 21,
 candidate=2*candidate,
 candidate= - x*y*poly1 + candidate,
 candidate= - 4*x*poly1 + candidate,
 candidate=4*candidate,

              3
 candidate= - y *poly1 + candidate,
 candidate=2*candidate,

                2
 candidate= - 3*y *poly1 + candidate,
 candidate=13*y*poly1 + candidate,
 candidate=candidate + 6*poly1,

                2
 candidate= - 2*y *poly2 + candidate,
 candidate= - y*poly2 + candidate,
```

```
    candidate=candidate + 6*poly2}
```

これは次のことを意味しています。

$$\begin{aligned}16(5y^2+2x^2y+\frac{5}{2}xy+\frac{3}{2}y+8x^2+\frac{3}{2}x-\frac{9}{2}) = \\ (-8xy-32x-2y^3-3y^2+13y+6)\mathrm{POLY1} \\ +(-2y^2-2y+6)\mathrm{POLY2}\ \ +y^2.\end{aligned}$$

22.3.8 GROEBNERT と PREDUCET によるトレース

与えられた多項式系 $\{f_1,\ldots,f_k\}$ とその Gröbner 基底 $\{g_1,\ldots,g_l\}$ に対して、次の式を満たすような要素が多項式であるような行列 C_{ij} と D_{ji} が存在します。

$$f_i=\sum_j C_{ij}g_j \ \text{ と } \ g_j=\sum_i D_{ji}f_i$$

このような関係式の具体的な形が必要な場合があります。Buchberger [13] には、このような計算の例として線形方程式系の場合の例が記述されています。このような関係式を求める標準的な方法としては、Gröbner 基底を計算させ、そのときの計算の各ステップをすべて記録しておくことです。現在のパッケージでは余因子 (cofactor) を使って計算します。これは入力した式にそれぞれ名前を付けておき、式に対する計算と同時に同じ操作を名前に対しても実行します。このようにすることで、最終結果には簡約された結果と同時に、入力された式からその結果に到達するまでの代数演算の式が得られます。

二つの互いに補完的な演算子 GROEBNERT と PREDUCET があります。これらは機能的には演算子 GROEBNER と PREDUCE に対応しています。しかし、引数の式のリストには左辺が識別子で右辺が多項式である方程式のリストを指定しなければいけません。方程式の左辺の識別子は右辺式の名前として扱われます。結果は GROEBNERT については方程式のリストで、PREDUCET に対しては方程式が返されます。これらの方程式で、左辺は計算した値で、右辺は同じ値を入力した式の識別子名で書いた式になっています。

例

楕円 "$p1$" と直線 "$p2$" の Gröbner 基底を計算することを考える。$p2$ はそのまま基底の要素になっており、基底の第二要素は結果で示されるように $p1$ と $p2$ の組合せとして表される。

```
gb1:=groebnert {p1=2*x**2+4*y**2-100,p2=2*x-y+1};

gb1 := {2*x - y + 1=p2,
           2
        9*y  - 2*y - 199= - 2*x*p2 - y*p2 + 2*p1 + p2}
```

例

多項式 x**2 を前の例での Gröbner 基底に関して簡約を行い、同時に簡約の公式を求める。これを行うには、$GB1$ から基底の多項式を取り出し、それに名前を付け (ここでは $G1$ と $G2$ とする)、PREDUCET を呼びます。ここで簡約したい多項式を Q という名前にし、結果の右辺式にはこの名前で表されるようにします。もし、多項式の名前を省略すると、右辺式にはその式の値が使われます。

```
gb2 := for k := 1:length gb1 collect
      mkid(g,k) = lhs part(gb1,k)$
preducet (q=x**2,gb2);

- 16*y + 208= - 18*x*g1 - 9*y*g1 + 36*q + 9*g1 - g2
```

この結果は次のことを表しています。

$$x^2 = (\frac{1}{2}x + \frac{1}{4}y - \frac{1}{4})g1 + \frac{1}{36}g2 + (-\frac{4}{9}y + \frac{52}{9}).$$

例

もしイデアルに含まれている多項式を簡約すると、結果として左辺が零である式が得られます。

```
preducet(q=2*x**2+4*y**2-100,gb2);

0= - 2*x*g1 - y*g1 + 2*q + g1 - g2
```

これは次の式を表しています。

$$q = (x + \frac{1}{2}y - \frac{1}{2})g1 + \frac{1}{2}g2.$$

これらの演算子によって、行列 C_{ij} や D_{ij} の値が陰型式で (式の係数として) 得られます。f_i を $\{g_j\}$ に関して GROEBNERT 演算子を作用させることにより D_{ij} の値が、そして PREDUCET 演算子で多項式 f_i を $\{g_j\}$ に関して簡約することにより C_{ij} の値が求められます。後の場合には、左辺は 0 で右辺が f_i に関して線形な式が得られます。

もし $\{1\}$ が Gröbner 基底であれば、GROEBNERT 演算子は入力多項式の組合せから 1 を構成する証明を返します。

注意: トレースから求める方法以外の方法と比べると、これらの演算子は計算時間がかかります。従って、これらの演算子は小さいサイズの問題にのみ使うことができます。

22.3.9 束に対する Gröbner

与えられた多項式環、例えば、$R = Z[x_1 \cdots x_k]$ と整数 $n > 1$ に対して、R を要素とする n 次元のベクトルは、加算 (要素ごとの加算) と R の要素との乗算に対して 束 (*module*) をなす。

与えられた有限イデアルの部分束に対して、Gröbner 基底を計算することができます。さらに、GROEBNER パッケージの中で、$GROEBNERF$ と $GROESOLVE$ の演算を除いた以外の機能を使うことができます。

ベクトルは、束の単位ベクトルを表す補助変数を使って表現されています。例えば、v_1, v_2, v_3 を使って、束の要素 $[x_1^2, 0, x_1 - x_2]$ は、$x_1^2 v_1 + x_1 v_3 - x_2 v_3$ を表されます。v_1, v_2, v_3 を単位ベクトルとして設定するには、シェアー変数 $GMODULE$ に補助変数の集合を代入することでできます。次のようにします。

```
gmodule := {v1,v2,v3};
```

この設定を行なった後では、これらの変数から作られる単項式は全て、ベクトル空間上で代数的に独立であると考えられます。一度、$GMODULE$ を設定すれば、補助変数は変数リストの最後に自動的に (まだ含まれていない場合には) 追加されます。

例:

```
torder({x,y,v1,v2,v3},lex)$
gmodule := {v1,v2,v3}$
g:=groebner{x^2*v1 + y*v2,x*y*v1 - v3,2y*v1 + y*v3};

g := {x *v1 + y*v2,

            2
     x*v3 + y *v2,

      3
     y *v2 - 2*v3,

     2*y*v1 + y*v3}

preduce((x+y)^3*v1,g);

           1   3         2
 - x*y*v2 - ---*y *v3 - 3*y *v2 + 3*y*v3
           2
```

多くの場合、束の基底の計算には、全次数順序を使うのが一番適当です。しかし、補助変数について辞書式順序を設定して計算すると、興味深い結果が得られます。例えば、

```
p1:=(x-1)*(x^2-x+3)$  p2:=(x-1)*(x^2+x-5)$
gmodule := {v1,v2,v3};
torder({v1,x,v2,v3},lex)$
gb:=groebner {p1*v1+v2,p2*v1+v3};
```

```
gb := {30*v1*x - 30*v1 + x*v2 - x*v3 + 5*v2 - 3*v3,

        2       2
       x *v2 - x *v3 + x*v2 + x*v3 - 5*v2 - 3*v3}

   g:=coeffn(first gb,v1,1);

g := 30*(x - 1)

   c1:=coeffn(first gb,v2,1);

c1 := x + 5

   c2:=coeffn(first gb,v3,1);

c2 :=  - x - 3

   c1*p1 + c2*p2;

30*(x - 1)
```

ここで、二つの多項式は、ベクトル $[p_1, 1, 0]$ と $[p_2, 0, 1]$ として入力されています。項順序として、最初の次元を最大にし、それ以外の要素を最小に取ると、古典的な余因子計算はちょうど拡張されたユークリッドのアルゴリズムで行なわれることになります。したがって、結果として得られる基底の内、最初の多項式の v_1 に関する係数は、p_1 と p_2 の最大公約多項式になります。また、v_2 と v_3 の係数は、$gcd(p_1, p_2) = c_1p_1 + c_2p_2$ の関係式で与えられる、p_1 と p_2 の余因子 c_1 と c_2 になります。

22.3.10 GROEBNERM: モジュールに対する Gröbner 基底

r-対の多項式 $(p_1, \ldots, p_r)$ に対して和を要素ごとの和とし、任意の多項式との積を $p*(p_1, \ldots, p_r) := (p * p_1, \ldots, p * p_r)$ で定義します。このような r-対の多項式 $P_1 := (p_{11}, \ldots, p_{1r}), \ldots, P_m := (p_{m1}, \ldots, p_{mr})$ に対して多項式のモジュール

$$M := \{g_1 * P_1 + \cdots + g_m * P_m \mid g_1, \ldots, g_m \text{ は多項式 }\}$$

は任意の項順序と任意の重みに対して Gröbner 基底を持ちます。 GROEBNERM 演算子はモジュールの Gröbner 基底を計算します。構文は次の通りです。

$$\text{GROEBNERM } (\{expr_1, \ldots, expr_m\}[, \{var_1, \ldots, var_n\}[, \{w_1, \ldots, w_r\}])$$

ここで

> $\{expr_1, \ldots, expr_m\}$ は r-対の多項式のリストで、$\{var_1, \ldots, var_n\}$ はオプションの変数リスト、そして $\{w_1, \ldots, w_r\}$ は 1000 以下の整数のリストです。

ここで *expr* はまた GROEBNERT 演算子の場合のように名前を付けた、次のような式でも構いません。

$$p = \{pol_1, \ldots, pol_r\}$$

この場合には Gröbner 基底の計算のトレースが保存されます。もし重み付けられた順序で計算している場合には、$\{w_1, \ldots, w_r\}$ はベクトル空間での各要素の重みを指定します。もしこれが省略されたときは全ての要素が同じ重み 1 であるとされます。

例

$$\begin{aligned} B \;:=\; & \{\{1,0,0,0,0\},\\ & \{0,0,0,0,-1\},\\ & \{0,0,0,1,0\},\\ & \{0,0,-1,0,x^2\}\}\$ \end{aligned}$$

$$\begin{aligned} D \;:=\; & \{\{-x,y,1,0,0\},\\ & \{0,5x^2,y,1,0\},\\ & \{0,0,4x^2,5y,y\},\\ & \{0,5,0,x^2,y\}\}; \end{aligned}$$

groebnerM (append(b, d) , $\{x, y\}$, $\{4, 4, 5, 6, 7\}$);

```
        2
{{0,5,0,x ,y},

           2
 {0,0,-1,0,x },

 { - x,y,1,0,0},

 {1,0,0,0,0},

 {0,1,0,0,0},

 {0,0,1,0,0},

 {0,0,0,1,0},

 {0,0,0,0,-1}}
```

$$\begin{aligned} \text{Btagged} \;:=\; \{t1 &= \{1,0,0,0,0\},\\ t2 &= \{0,0,0,0,-1\},\\ t3 &= \{0,0,0,1,0\},\\ t4 &= \{0,0,-1,0,x^2\}\}\$ \end{aligned}$$

groebnerM (append(btagged,d) , $\{x, y\}$, $\{4, 4, 5, 6, 7\}$);

```
         2
{{0,5,0,x ,y},

            2
 {0,0,-1,0,x }=t4,

 { - x,y,1,0,0},

 {1,0,0,0,0}=t1,

                  2
               - x *t3 + y*t2
 {0,1,0,0,0}=-----------------,
                     5

                 2
 {0,0,1,0,0}= - x *t2 - t4,

 {0,0,0,1,0}=t3,

 {0,0,0,0,-1}=t2}
```

22.3.11 その他の順序

基本的な項の順序の他に、特別な目的に使われる順序があります。

変数のグループ分け

多項式系で、変数とパラメーターとを分けて考えたい場合がしばしばあります。これは辞書式順序 (LEX) による Gröbner 基底で行えます。しかし、辞書式順序では必要以上に変数を分割してしまうので、計算が困難なことがあります。ここで示す順序では変数を二つもしくはそれ以上のグループに分け、それぞれのグループ内では標準の順序が使われ、グループ全体はそれらの全次数順序が適応されます。Gröbner 基底の計算では、可能であれば、最初のグループの要素から消去されます。TORDER はグループを指定する為の追加の引数を指定できます。

TORDER (vl,gradlexgradlex, n)
TORDER (vl,gradlexrevgradlex,n)
TORDER (vl,lexgradlex, n)
TORDER (vl,lexrevgradlex, n)

ここで整数 n は最初のグループの変数の数を指定します。例えば、

lexgradlex, 3 for $\{x_1, x_2, x_3, x_4, x_5\}$:
$x_1^{i_1} \ldots x_5^{i_5} \gg x_1^{j_1} \ldots x_5^{j_5}$
が成立するのは

$$(i_1, i_2, i_3) \gg_{lex} (j_1, j_2, j_3)$$
または
$$(i_1, i_2, i_3) = (j_1, j_2, j_3)$$
かつ
$$(i_4, i_5) \gg_{gradlex} (j_4, j_5)$$

が成り立つときである。

注意しておくことは、ここでの順序として LEX を指定することは無意味なので指定できません。

重み付けられた順序

$$TORDER \quad (vl, \text{weighted}, \{n_1, n_2, n_3 \ldots\}) \; ;$$

は重み付けられた順序を定義します。ここで、それぞれの変数の指数部は重みを掛けて加えられます。もし重みの指定が実際の変数の数よりも少なければ、残りの変数に付いては重み 1 として処理されます。

Graded 順序

文

$$TORDER \quad (vl, \text{graded}, \{n_1, n_2, n_3 \ldots\}, order_2) \; ;$$

は、階層付けられた順序を定義します。つまり、指数は最初に重みとの積が計算されます。もし、変数よりも少ない重みがあれば、重み 1 が加えられます。もし、重み付けられた次数の計算ができなければ、$order_2$ で指定された項順序が残りの引数に対して使われます。*Graded* 順序は、最初 *dd_groebner* 演算子で使う目的で、設計されました。

Matrix 順序

$$TORDER \quad (vl, \text{matrix}, m) \; ;$$

ここで、m は整数を要素とする行列で、行の長さは変数の数に一致します。各単項式の指数はベクトルを成す。二つの項の比較は、まず指数ベクトルに m を掛け、結果として得られるベクトル同士を辞書式順序で比較します。単位行列を指定した場合には、*LEX* 順序を指定したのと同じことです。また、最初の行は全て 1 で、その後に単位行列が並んだものが、*GRADLEX* 順序を指定することになります。

行列 m は、列よりも行の方が長くなければなりません。正方行列でなければ、冗長な行を含んでいることになります。また、フルランクでなければなりません。そして、各列のゼロでない最初の要素は正でなければなりません。

matrix 順序は非常に一般的な順序を定義できますが、その反面、項を比較するのに非常に計算時間がかかります。このため、使っている REDUCE が LISP コンパイラーの機能を持ってい

れば、*matrix* 順序をコンパイル出来るようになっています。*COMP* スイッチをオンにすれば、*matrix* 順序を定義した時に、比較関数をコンパイルします。また、自分で、

```
torder_compile(<n>,<m>);
```

として、コンパイルすることも出来ます。ここで、$< n >$ は識別子の名前で、$< m >$ は、項順序の行列です。*torder_compile* は、行列を LISP のプログラムに変換し、*COMP* がオンであればコンパイルします。これ以後、*torder* 宣言の中で、項順序として $< n >$ という名前を使うことが出来ます。

22.3.12 Graded 同次方程式の Gröbner 基底

項順序 *graded*, *gradlex*, *revgradlex* または *weighted* での、多項式の同次方程式系に対する Gröbner 基底は、計算で現れる S 多項式の次数を制限して計算することが可能です。

dd_groebner ($d1$,$d2$,$\{p_1, p_2, \ldots\}$);

ここで、 $d1$ は非負の整数で、$d2$ は $> d1$ である整数かもしくは “無限大 (infinity)” です。主項の最小公倍数 (lcm) の grade が $d1$ から $d2$ の間にある多項式の組合せについてのみ、S 多項式の計算を行ないます。数学的な背景については文献 [8] を参照して下さい。

grated または *weighted* 順序に対して、最初、重みベクトルが grade の計算に使われます。それ以外の順序に対しては、項の全次数が使われます。

22.4 イデアル分解と方程式系の解法

基本的な Gröbner 演算に基づき、Gröbner パッケージにはイデアルの分解を求めるとか方程式系の解を求めるための演算子が用意されています。

22.4.1 LEX 型の順序で求めた Gröbner 基底による方程式系の解法

GROESOLVE: 多項式系の解

GROESOLVE 演算子は GROEBNERF により辞書式順序で Gröbner 基底を求め、イデアル分解の技巧を使って簡単な方程式に分解します。もし代数的に可能であれば、問題は一変数の多項式に帰着されます。これは SOLVE 演算子を使って解くことができます。もし ROUNDED スイッチがオンであれば、一変数多項式の根は数値的な近似解が求められます。

$$GROESOLVE(\{exp_1, exp_2, \ldots, exp_m\}[, \{var_1, var_2, \ldots, var_n\}]);$$

ここで $\{exp_1, exp_2, \ldots, exp_m\}$ は任意個の式もしくは方程式で、$\{var_1, var_2, \ldots, var_n\}$ はオプションの変数リストです。

結果は部分集合のリストです。この部分集合は与えられた方程式の解を含んでいます。もし有限個の解しか持たなければ、それぞれの部分集合は三角型の式 $\{exp_1, exp_2, \ldots, exp_n\}$, の集合に

なっています。ここで exp_1 は変数 var_1 にのみ依存する式で、exp_2 は変数 var_1 と var_2 にのみ依存する式、...、です。最後の exp_n は $var_1, \ldots, var_n$ に依存する式になっています。これは、順次解いていくことができます。もし無限個の解を持つような場合は、いくつかの部分集合は変数の個数よりも少ない数の方程式しか含まれません。このような場合、変数の内のいくつかを “自由パラメーター”として扱うことにより、同じように三角型で表しています。

例:直線と円との交点。

$$GROESOLVE(\{x**2 - y**2 - a, p*x + q*y + s\}, \{x, y\});$$

```
                 2      2    2            2    2
 {{x=(sqrt( - a*p  + a*q  + s )*q - p*s)/(p  - q ),
                   2      2    2            2    2
   y= - (sqrt( - a*p  + a*q  + s )*p - q*s)/(p  - q )},
                   2      2    2            2    2
  {x= - (sqrt( - a*p  + a*q  + s )*q + p*s)/(p  - q ),
                 2      2    2            2    2
   y=(sqrt( - a*p  + a*q  + s )*p + q*s)/(p  - q )}}
```

GROEPOSTPROC: Gröbner 基底の後処理

多くの場合、一般的な Gröbner 基底の計算を行うのは困難である。もし、LEX 順序での Gröbner 基底が求められていれば、方程式の解は GROEPOSTPROC によって求めることができます。実際、GROESOLVE は機能的には GROEBNERF により Gröbner 基底を求め、得られた部分基底それぞれに GREOPOSTPROC を使って解を求めるのと同じ計算を行っています。

$$GROEPOSTPROC(\{exp_1, exp_2, \ldots, exp_m\}[, \{var_1, var_2, \ldots, var_n\}]);$$

ここで $\{exp_1, exp_2, \ldots, exp_m\}$ は任意個の数式のリストで、$\{var_1, var_2, \ldots, var_n\}$ はオプションの変数リストです。数式は与えられた変数に付いて辞書式順序で求めた Gröbner 基底でなければいけません。計算の順序モードは LEX モードでなければいけません。

結果は GROESOLVE と同じです。

```
groepostproc({x3**2 + x3 + x2 - 1,
              x2*x3 + x1*x3 + x3 + x1*x2 + x1 + 2,
              x2**2 + 2*x2 - 1,
              x1**2 - 2},{x3,x2,x1});

{{x3= - sqrt(2),

  x2=sqrt(2) - 1,

  x1=sqrt(2)},
```

```
{x3=sqrt(2),

 x2= - (sqrt(2) + 1),

 x1= - sqrt(2)},

     sqrt(4*sqrt(2) + 9) - 1
{x3=-------------------------,
                2

 x2= - (sqrt(2) + 1),

 x1=sqrt(2)},

      - (sqrt(4*sqrt(2) + 9) + 1)
{x3=-----------------------------,
                  2

 x2= - (sqrt(2) + 1),

 x1=sqrt(2)},

     sqrt( - 4*sqrt(2) + 9) - 1
{x3=---------------------------,
                 2

 x2=sqrt(2) - 1,

 x1= - sqrt(2)},


      - (sqrt( - 4*sqrt(2) + 9) + 1)
{x3=---------------------------------,
                   2

 x2=sqrt(2) - 1,

 x1= - sqrt(2)}}
```

IDEALQUOTIENT: イデアルと数式の商

I をイデアル、 f を変数が同じである多項式とする。このとき商は次のように定義されます。

$$I : f = \{p \mid p * f \text{ は } I \text{ の要素}\}.$$

イデアルの商 $I : f$ は I を含んでおり、多項式環全体の部分集合になっている。つまり {1} に含まれます。ここで $I : f = \{1\}$ が成り立つことと、f が I に含まれていることは同値です。そのほかの特別な場合として、f が I の共通零点で零にならない場合には $I : f = I$ が成り立ちます。しかしながら、van der Waerden によって導入された “一般零点”の概念を説明するのはこのマニュアルの目的を越えています。GROESOLVE や GROEPOSTPREC の計算は重複したイデアル商の計算に基づいています。

もし I が基底によって与えられ、f が与えられた式とすると、商は次の様に計算できます。

$$IDEALQUOTIENT(\{exp_1, \ldots, exp_m\}, exp);$$

ここで $\{exp_1$,exp_2,...,$exp_m\}$ は任意の数の数式もしくは方程式です。

IDEALQUOTIENT は基底 $\{exp_1, exp_2, \ldots, exp_m\}$ からなるイデアル I と exp の商を計算します。$\{exp_1, exp_2, \ldots, exp_m\}$ は Gröbner 基底である必要はありません。

22.4.2 Gröbner 基底に対する演算

問題によっては辞書式順序での Gröbner 基底を求めることが不可能だけれども、全次数順序での基底は計算できる場合があります。このとき、Hilbert 多項式は解空間の次元に関する情報を与えます。また、解が有限次元であれば一変数多項式系を計算することができます。与えられた方程式の解は、全ての一変数多項式の解の cross 積に含まれています。

HILBERTPOLYNOMIAL: イデアルの Hilbert 多項式

この算法は Joachim Hollman, Royal Institute of Technology, Stockholm (private communication) によるものです。

$$HILBERTPOLYNOMIAL(\{exp_1, \ldots, exp_m\})\ ;$$

ここで $\{exp_1, exp_2, \ldots, exp_m\}$ は任意個の数式もしくは方程式のリストです。

HILBERTPOLYNOMIAL は基底 $\{exp_1, exp_2, \ldots, exp_m\}$ からなるイデアルに対して、指定された変数もしくは与えられた式から抜きだした変数に関する Hilbert 多項式を求めます。項の順序は GRADLEX や REVGRADLEX 順序のような、次数に関して両立する順序でなければいけません。項の順序モードは有効であり、$\{exp_1, exp_2, \ldots, exp_m\}$ はこの順序に関して Gröbner 基底になっていなければなりません。Hilbert 多項式は方程式系 $\{exp_1, exp_2, \ldots, exp_m\}$ の解の数に関する情報を与えます。もし、Hilbert 多項式が整数であれば方程式系は離散的な零点しか持たず、重複度も含めた解の個数に一致します。それ以外の場合には、Hilbert 多項式の次数は解空間の次元に一致します。

もし Hilbert 多項式が定数でなければ、変数 “X”(x が元々の変数リスト $\{var_1, var_2, \ldots, var_n\}$ に含まれているか否かには関係なくこの変数名が使われます。) の多項式として表されます。十分大きな “X”の値に対するこの多項式の値は、次数が $\leq X$ である多項式全体の線形空間の次元から次数 $\leq X$ であるイデアルに属する多項式全体の次元を引いた差に等しい。

注意: イデアルの零点の数と Hilbert 多項式の零点の数は、Gröbner 基底の主項にのみ依存しています。従って、引き続いて Hilbert 多項式の計算を行おうとする場合は、Gröbner 基底の計算を ON GLTBASIS の状態で行い、GLTB の値 (もしくは GROEBNERF を使った場合はその要素) を HILERTPOLYNOMIAL に与えるべきです。こうすることで大きな基底を扱っているような場合に、計算時間を大幅に節約することができます。

22.5 “手計算”による計算

次の演算子が、分布表現で表した多項式の計算を行うために用意されています。これらの演算子を使うことによって、Gröbner 基底のような計算を順を追って逐次計算することができます。単項式や多項式に対する演算には通常の REDUCE の演算が使われます。

22.5.1 多項式を分布表現に変換

$$GSORT(p);$$

ここで p は多項式もしくは多項式のリストです。

もし p が一つの多項式の場合は、結果は p を現在設定されている変数と項の順序モードに基づく分布表現に変換します。p がリストであれば、各要素が分布表現に変換され、主項の項順序に基づいて並べ換えたものが返されます。零多項式は結果から取り除かれます。

```
torder({alpha,beta,gamma},lex);
dip := gsort(gamma*(alpha-1)**2*(beta+1)**2);
```

```
            2     2                2
dip := alpha *beta *gamma + 2*alpha *beta*gamma

       2                    2
+ alpha *gamma - 2*alpha*beta *gamma - 4*alpha*beta*gamma

                       2
 - 2*alpha*gamma + beta *gamma + 2*beta*gamma + gamma
```

22.5.2 多項式を主項と残りの項への分解

$$GSPLIT(p);$$

ここで p は多項式です。

GSPLIT は多項式 p を分布表現に変換し、主項と残りの項に分解します。結果は二つの要素、主項と残りの項からなるリストです。

```
gslit(dip);

       2     2
 {alpha *beta *gamma,

         2                    2                       2
  2*alpha *beta*gamma + alpha *gamma - 2*alpha*beta *gamma

                      2
  - 4*alpha*beta*gamma - 2*alpha*gamma + beta *gamma


  + 2*beta*gamma + gamma}
```

22.5.3 Buchberger の S-多項式の計算

$$GSPOLY(p1, p2);$$

ここで p_1 と p_2 は多項式です。

GSPOLY は p_1 と p_2 から S-多項式を計算します。

DAVENPORT ET AL. [16] から取った例:

```
torder({x,y,z},lex)$
%  与えられた多項式系
g1  :=  x**3*y*z - x*z**2;
g2  :=  x*y**2*z - x*y*z;
g3  :=  x**2*y**2 - z;$


 % 最初の S-多項式
g4  :=  gspoly(g2,g3);$

    2        2
 g4 := x *y*z - z

 % 次の S-多項式
```

```
p :=  gspoly(g2,g4); $

      2          2
p := x *y*z - y*z

% そして簡約を行う。ここでは g4 の一つだけだが
% PREDUCE には引数として
% リストを渡す必要がある。

g5  :=  preduce(p,{g4});

            2    2
g5 :=  - y*z  + z

% 最後の S-多項式

g6  :=  gspoly(g4,g5);

       2  2    3
g6 := x *z  - z

% 得られた基底を並べ換える

gsort{g2,g3,g4,g5,g6};

  2  2
{x *y  - z,

  2        2
 x *y*z - z ,

  2  2    3
 x *z  - z ,

    2
 x*y *z - x*y*z,

       2    2
  - y*z  + z }
```

第23章　IDEALS: 多項式イデアル

Herbert Melenk
Konrad-Zuse-Zentrum für Informationstechnik
Heilbronner Str. 10
D1000 Berlin 31
Federal Republic of Germany
E-mail: *melenk@sc.zib-berlin.de*

May 1992

23.1　序論

このパッケージでは、REDUCE の Gröbner パッケージを使って、多項式イデアルに対する基本的な演算を行います。計算時間を節約するため、中間的な Gröbner 規定の計算結果は保存され、不必要な再計算はさけるようにしています。

23.2　初期化

計算を始める前に、*I_setting* を呼び出して変数の集合を宣言しておかなければなりません。例えば、計算を多項式環 $Q[x,y,z]$ で行う場合には、

```
I_setting(x,y,z);
```

と入力します。以後 *I_setting* を呼び出す度に、別の変数集合上での計算が可能となります。このとき、同時に内部で記憶している計算結果はすべて消去されます。

23.3　基底

イデアルは、基底 (多項式の集合) として、前に I のタグを付けて表します。例えば、

```
u := I(x*z-y**2, x**3-y*z);
```

また、基底を入力するときには多項式のリストを代わりに使うこともできます。しかし、計算結果はすべて上の形式で表されます。関数 *ideal2list* を使って、イデアル基底を通常の REDUCE のリスト形式に変換することが出来ます。

23.3.1 演算

REDUCE の構文上の制約から、特別な演算子をイデアルの演算に使わざるをえません:

`.+`	イデアルの和（内挿演算子）
`.*`	イデアルの積（内挿演算子）
`.:`	イデアルの商（内挿演算子）
`./`	イデアルの除余（内挿演算子）
`.=`	イデアルの同値（内挿演算子）
`subset`	イデアルの部分集合 (内挿演算子)
`intersection`	イデアルの共通部分 (前置演算子, 二項)
`member`	イデアルのメンバーシップ (内挿演算子: 多項式とイデアル)
`gb`	イデアルの Groebner 基底 (前置演算子, 単項)
`ideal2list`	イデアル基底を多項式のリストに変換 (前置演算子, 単項)

例:

```
I(x+y,x^2) .* I(x-z);

   2                       2    2
I(X  + X*Y - X*Z - Y*Z,X*Y  - Y *Z)
```

イデアルの同値性を判定する関数は結果として 1 (真の場合) または 0(偽の場合) を返します。これは、REDUCE の *if* 文の条件式等に直接使用することが出来ます。

演算 $sum, product, quotient, intersection$ の結果は、現在の設定と項順序に基づく Gröbner 基底で表されたイデアルです。項の順序は、Gröbner 基底のパッケージにある *torder* 演算子で変更することが出来ます。イデアルの同値性は REDUCE の等号では調べられないことに注意して下さい。

```
I(x,y)  = I(y,x)        これは間違い
I(x,y) .= I(y,x)        これが正しい
```

23.4 算法

REDUCE の Gröbner 基底パッケージの関数 *groebner*, *preduce* と *idealquotient* は、基本的な算法をサポートしています:

$GB(Iu_1, u_2...) \rightarrow groebner(\{u_1, u_2...\}, \{x, ...\})$

$p \in I_1 \rightarrow p = 0 \ mod \ I_1$

$I_1 : I(p) \to (I_1 \bigcap I(p))/p$ 要素ごとに

イデアルのパッケージは次の演算をインプリメントしています:

$I(u_1, u_2...) + I(v_1, v_2...) \to GB(I(u_1, u_2..., v_1, v_2...))$

$I(u_1, u_2...) * I(v_1, v_2...) \to GB(I(u_1 * v_1, u_1 * v2, ..., u_2 * v_1, u_2 * v_2...))$

$I_1 \bigcap I_2 \to Q[x, ...] \bigcap GB_{lex}(t * I_1 + (1 - t) * I_2, \{t, x, ..\})$

$I_1 : I(p_1, p_2, ...) \to I_1 : I(p_1) \bigcap I_1 : I(p_2) \bigcap ...$

$I_1 = I_2 \to GB(I_1) = GB(I_2)$

$I_1 \subseteq I_2 \to \ u_i \in I_2 \ \forall \ u_i \in I_1 = I(u_1, u_2...)$

23.5 例

例についてはファイル *ideals.tst* を参照して下さい。

第24章　INEQ:不等式の解法

Herbert Melenk

Konrad-Zuse-Zentrum fuer Informationstechnik
Heilbronner Str. 10, D10711 Berlin – Wilmersdorf
Germany
E-mail: *melenk@zib-berlin.de*

このパッケージは、演算子 **ineq_solve** を定義しています。これは単一もしくは複数の不等式からなる方程式の解を求めます。次のようなシステムの解法に使用できます。[1]

- 係数としてパラメータを含まない、数値係数のみのシステム,
- 線形不等式で、<= – >= の不等号のみ含むもの。Fourier と Motzkin の方法[2] を適応する。
- <=, >=, > または < 演算子を含んだ一変数不等式。左辺式及び右辺式は、多項式もしくは有理数。

構文:

```
INEQ_SOLVE(<expr> [,<vl>])
```

ここで <expr> は、不等式または不等式や等式のリストで、<vl> は、変数名もしくは変数名のリストです。変数名は指定しなくとも構いません。もし指定しなければ、<expr> から未知変数のリストが自動的に作成されます。多変数の場合、指定した変数リストは消去する順序も決めます。一番最後の変数に対して、最も限定された解が求められます。

入力された式が、現在インプリメントされているアルゴリズムで処理できなければ、エラーメッセージが出力されます。

結果はリストで、これが空であれば解を持たないことを表します。それ以外の場合は、結果は許される変数の範囲を区間で示します。

最も制限された変数は結果のリスト中で最初のもので、それぞれの式は先行する変数のみを含んでいます。区間はまた **max** もしくは **min** を含んだ式で表されることもあります。代数的数は浮動小数点数による近似値で表されます。

---

[1] 線形の最適化問題に対しては **linalg** パッケージ中の **simplex** 関数を使って下さい。
[2] G.B. Dantzig による *Linear Programming and Extensions* で記述されている

例:

```
ineq_solve({(2*x^2+x-1)/(x-1) >=  (x+1/2)^2, x>0});

{x=(0 .. 0.326583),x=(1 .. 2.56777)}

 reg:=
 {a + b - c>=0, a - b + c>=0, - a + b + c>=0, 0>=0, 2>=0,
  2*c - 2>=0, a - b + c>=0, a + b - c>=0, - a + b + c - 2>=0,
  2>=0, 0>=0, 2*b - 2>=0, k + 1>=0, - a - b - c + k>=0,
   - a - b - c + k + 2>=0, - 2*b + k>=0,
   - 2*c + k>=0, a + b + c - k>=0,
  2*b + 2*c - k - 2>=0, a + b + c - k>=0}$

ineq_solve (reg,{k,a,b,c});

{c=(1 .. infinity),

 b=(1 .. infinity),

 a=(max( - b + c,b - c) .. b + c - 2),

 k=a + b + c}
```

第25章　INVBASE: 包含的基底

A.Yu.Zharkov, Yu.A.Blinkov

Saratov University
Astrakhanskaya 83
410071 Saratov, Russia

e–mail: *postmaster@scnit.saratov.su*

包含的 (involutive) 基底は、多変数多項式と関連した問題、例えば多項式方程式の系を解くとか多項式イデアルを解析するという問題、を解決するための新しい道具です。多項式イデアルの包含的基底とは、冗長な Gröbner 基底の特別なものです。包含的基底を構成することによって、多項式システムを解く問題は単純な線形代数に帰着させます。

INVBASE パッケージは Buchberger のアルゴリズムの代用としても見ることができます。

25.1　基本的な演算子

25.1.1　項順序

利用可能な項順序は `REVGRADLEX`, `GRADLEX` および `LEX` です。これらのモードの意味は GROEBNER パッケージと同じです。

全ての順序は変数間の順序に基づいています。各々の変数の組合せに対して関係 gg が定義されている必要があります。項順序と変数の順序は演算子 `INVTORDER` $mode$,$\{x_1, ..., x_n\}$ で設定されます。ここで、$mode$ は項順序のいずれかです。$\{x_1, ..., x_n\}$ は変数のリストで、同時に変数の順序について $x_1 \gg \ldots \gg x_n$ を意味しています。

25.1.2　Involutive 基底の計算

多項式 $\{p_1, ..., p_m\}$ で生成されるイデアルの包含的基底を計算するためには、次のコマンドを入力します。

`INVBASE` $\{p_1, ..., p_m\}$

ここで、p_i は `INVTORDER` 演算子に現れた変数に関する多項式です。もし、`INVTORDER` で指定されていない変数が p_i に現れた場合、それらはパラメータとして扱われます。`INVTORDER` が省略された場合には、p_i に現れる全ての識別子は変数として扱われその順序は通常の REDUCE の順序に従います。

多項式 p_i の係数は整数もしくは有理数 (もしくは、パラメトリックな場合には多項式や有理式) です。素数を法とした演算も行なえます。これは、REDUCE のコマンドで

```
ON MODULAR; SETMOD p;
```

と入力します。ここで、p は素数です。関数 INVBASE の値は与えられてたイデアルの包含的基底である整数多項式 $\{g_1, ..., g_n\}$ です。

```
INVTORDER REVGRADLEX, {x,y,z};

g:= INVBASE {4*x**2 + x*y**2 - z + 1/4,
             2*x + y**2*z + 1/2,
             x**2*z - 1/2*x - y**2};

             3          2      3        2
g := {8*x*y*z  - 2*x*y*z  + 4*y  - 4*y*z  + 16*x*y + 17*y*z - 4*y,

         4        2        2               2
      8*y  - 8*x*z  - 256*y  + 2*x*z + 64*z  - 96*x + 20*z - 9,

         3
      2*y *z + 4*x*y + y,

           3        2      2      2
      8*x*z  - 2*x*z  + 4*y  - 4*z  + 16*x + 17*z - 4,

            3      3               2
       - 4*y*z  - 8*y  + 6*x*y*z + y*z  - 36*x*y - 8*y,

          2       2      2
      4*x*y  + 32*y  - 8*z  + 12*x - 2*z + 1,

         2
      2*y *z + 4*x + 1,

           3      2           2
       - 4*z  - 8*y  + 6*x*z + z  - 36*x - 8,

         2       2      2
      8*x  - 16*y  + 4*z  - 6*x - z}
```

辞書式順序の Gröbner 基底に変換するには、以下のように入力します。

```
h := INVLEX g;
```

```
                      6        5         4           3
h := {3976*x + 37104*z  - 600*z  + 2111*z  + 122062*z

                 2
       + 232833*z  - 680336*z + 288814,

            2          6         5         4           3
      1988*y  - 76752*z  + 1272*z  - 4197*z  - 251555*z

                 2
       - 481837*z  + 1407741*z - 595666,

          7      6    5       4       3        2
      16*z  - 8*z  + z  + 52*z  + 75*z  - 342*z  + 266*z

       - 60}
```

第26章　LAPLACE: ラプラス変換と逆ラプラス変換

C. Kazasov, M. Spiridonova, V. Tomov
Sofia, Bulgaria

このパッケージを使う上でのいくつかのヒントを書いておきます。

構文:

```
LAPLACE(<exp>,<var-s>,<var-t>)

INVLAP(<exp>,<var-s>,<var-t>)
```

ここで <exp> は変換しようとしている式で、<var-s> は元の変数 (通常 <exp> はこの変数の関数となっています。) で、<var-t> は変換後の変数名です。もし <vat-t> を省略すると `lp!&` もしくは `il!&` という名前の変数名をそれぞれ使います。

次のスイッチを使って変換を制御できます。

- lmon: もし、オンであれば sin, cos, sinh および cosh はラプラス変換した後の式中では指数関数で表されます。
- lhyp: もしオンであれば、 式 e**(x) は 逆ラプラス変換した後の式中では、双曲線関数 sinh と cosh で表されます。
- ltrig: もしオンであれば式 e**(x) は逆ラプラス変換した後の式中では三角関数 sin と cos で表されます。

ラプラス変換または逆ラプラス変換の規則を、ルールやルールセットで定義することによって拡張することが出来ます。これらのルール中では、変換前の変数名は自由変数で、変換後の変数名は `LAPLACE` に対しては `il!&`、`INVLAP` に対しては `lp!&` でなければなりません。そして、三番目の引数は省略しなくてはなりません。微分の変換規則についてもこのような形式で入れないといけません。

例:

```
  let {laplace(log(~x),x) => -log(gam * il!&)/il!&,
       invlap(log(gam * ~x)/x,x) => -log(lp!&)};

operator f;
let{
  laplace(df(f(~x),x),x) => il!&*laplace(f(x),x) - sub(x=0,f(x)),

  laplace(df(f(~x),x,~n),x) => il!&**n*laplace(f(x),x) -
    for i:=n-1 step -1 until 0 sum
      sub(x=0, df(f(x),x,n-1-i)) * il!&**i
        when fixp n,

  laplace(f(~x),x) = f(il!&)
  };
```

関数に関する注意:

- DELTA と GAMMA 関数は定義されています。
- ONE はステップ関数です。
- INTL は、パラメトライズされた積分関数です。

```
intl(<expr>,<var>,0,<obj.var>)
```

は、式 <expr> を変数 <var> に関して、0 から <obj.var> まで定積分したものを表します。つまり、`intl(2*y**2,y,0,x)` は x の関数です。

さらに詳しく知りたい人は LAPLACE.TST を見て下さい。

参考文献

Christomir Kazasov, *Laplace Transformations in REDUCE 3*, Proc. Eurocal '87, Lecture Notes in Comp. Sci., Springer-Verlag (1987) 132-133.

第27章　LIE: Lie代数の分類

Carsten and Franziska Schöbel
The Leipzig University, Computer Science Department
Augustusplatz 10/11,
O-7010 Leipzig, Germany
e–mail: *cschoeb@aix550.informatik.uni-leipzig.de*

LIE は実 n 次元 Lie 代数の分類のための関数を提供します。これは二つのモジュール **liendmc1** と **lie1234** から構成されています。

27.1　liendmc1

このモジュールに含まれる関数によって、次元 1 の派生した代数 $L^{(1)}$ を持つ実 n 次元 Lie 代数 L の分類が行なえます。L は基底 $\{X_1, \ldots, X_n\}$ のもとでの構造定数 c_{ij}^k で定義されていなければなりません。($[X_i, X_j] = c_{ij}^k X_k$)

ユーザは配列 LIENSTRUCIN(n, n, n) を定義しなければいけません。ここで n は Lie 代数 L の次元です。構造定数 LIENSTRUCIN(i, j, k):=c_{ij}^k for $i < j$ を与えれば、関数 `LIENDIMCOM1` を呼び出すことができます。これの構文は

```
LIENDIMCOM1(<数>).
```

です。<数> は次元 n です。この関数は実線形変換を行なうことにより L の構造を簡単化します。返される値は、次のようなリストです。

```
(i) {LIE_ALGEBRA(2),COMMUTATIVE(n-2)} または
(ii) {HEISENBERG(k),COMMUTATIVE(n-k)}
```

ここで $3 \leq k \leq n$ で、 k は奇数です。

返されたリストは `LIE_LIST` に保存されています。行列 `LIENTRANS` は与えられてた基底 $\{X_1, \ldots, X_n\}$ から標準基底 $\{Y_1, \ldots, Y_n\}$: $Y_j = (\text{LIENTRANS})_j^k X_k$ への変換行列を与えます。

27.2　lie1234

このモジュールは、次元 $n := \dim L = 1, 2, 3, 4$ である実 Lie 代数 L の分類を行ないます。L は同じく構造定数 c_{ij}^k を使って、基底 $\{X_1, \ldots, X_n\}$ と $[X_i, X_j] = c_{ij}^k X_k$ で定義されます。配列

LIESTRIN(n,n,n) の値を、$i<j$ に対して LIESTRIN(i,j,k):=c_{ij}^k と定義しておく必要があります。LIECLASS の呼び出し方は次のとおりです。

```
LIECLASS(<数>).
```

<数> は Lie 代数 L の次元です。この関数は L の交換関係を中心や派生代数の次元、ユニモジュラー性といった不変関係を使って順次簡約していきます。返される値は次の形式です。

```
{LIEALG(n),COMTAB(m)},
```

ここで m は標準形式 (基底 $\{Y_1,\ldots,Y_n\}$) の数に対応しています。

ここで返された値は LIE_CLASS に保存されています。基底 $\{X_1,\ldots,X_n\}$ から標準形式の基底 $\{Y_1,\ldots,Y_n\}$ への線形変換は行列 LIEMAT: $Y_j=(\mathrm{LIEMAT})_j^k X_k$ で与えられます。

第28章　LIMITS:極限値

Stanley L. Kameny

Email: *stan%valley.uucp@rand.org*

LIMITS は Ian Cohen と John P. Fitch による仕事を基に作成されたプログラムで、有限個の極と特異点を除いて連続な関数の極限値を求めます。クリティカルでない点での極限値は TPS パッケージを使って、その点の周りでの展開から求めています。クリティカルな点では l'Hôpital の方法を使って計算します。

28.1　両極限

LIMIT(EXPRN:**代数式**, VAR:**カーネル**, LIMPOINT:**代数式**):**代数式**

これは通常の極限値を求める演算子で、式 EXPRN で変数 VAR を LIMIPOINT に近づけたときの極限値を返します。

28.2　片側極限

LIMIT!+(EXPRN:**代数式**, VAR:**カーネル**, LIMPOINT:**代数式**):**代数式**
LIMIT!-(EXPRN:**代数式**, VAR:**カーネル**, LIMPOINT:**代数式**):**代数式**

もし LIMPOINT に近づけたときの方向によって極限値が異なる場合には、LIMIT!+と LIMIT!-演算子が使えます。これは次のように定義されます:

LIMIT!+ (EXP,VAR,LIMPOINT)　→　LIMIT(sub(VAR=VAR+ϵ^2,EXP) ,ϵ,0)
LIMIT!- (EXP,VAR,LIMPOINT)　→　LIMIT(sub(VAR=VAR-ϵ^2,EXP) ,ϵ,0)

28.3　診断用の関数

LIMIT0(EXPRN:**代数式**, VAR:**カーネル**, LIMPOINT:**代数式**):**代数式**

この関数は LIMIT とほとんど同じです。しかし、LOG の項については極限を取るときにまとめ

ることはしません。従って、いくつかの LOG 項で除去可能な特異点を持っているような場合、エラーを起こします。

LIMIT1(EXPRN:**代数式**, VAR:**カーネル**, LIMPOINT:**代数式**):**代数式**

この関数は TPS パッケージのみを使って極限を計算します。極限を取る点が特異点である場合にはエラーを起こします。

LIMIT2(TOP:**代数式**,
BOT:**代数式**,
VAR:**カーネル**,
LIMPOINT:**代数式**):**代数式**

この関数は商 (TOP/BOT) に対して l'Hôpital の方法を適用します。

第29章　LINALG: 線形代数パッケージ

Matt Rebbeck
Konrad-Zuse-Zentrum für Informationstechnik Berlin

29.1　序論

このパッケージでは、線形代数の世界で有用な関数を多く定義しています。これらの関数については、3節で詳しく説明してあります。これらの関数は次の四つのセクションに分類されます。

Walter Tietze (ZIB) 氏による寄与が含まれています。

29.1.1　基本的な行列操作の関数

29.1.2　生成子

行列を生成する関数には以下のものがあります。

29.1.3 応用計算

別に NORMFORM[1] のパッケージがあり、次のような行列の正準型を計算することができます。

smithex, smithex_int, frobenius, ratjordan, jordansymbolic, jordan.

29.1.4 述語関数

例への注意:

本章での例で、行列 $\mathcal{A}$ は次のような行列を表します。

$$\mathcal{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

記号

この章では $\mathcal{I}$ は、単位行列を表し、$\mathcal{A}^T$ は行列 $\mathcal{A}$ の転置を表しています。

29.2 線形代数

今までに REDUCE の行列演算機能を一度も使ったことがない人のために少し説明を加えておきます。

行列の作成

行列を作成するには、次のようにします:

```
mat1 := mat((a,b,c),(d,e,f),(g,h,i));
```

は次の行列を作ります。

$$mat1 := \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

要素を取り出す

行列の (i, j) 番目の要素を取り出すには次のように入力します。

```
mat1(i,j);
```

線形代数パッケージをロードする

このパッケージは次のようにしてロードします。

```
load_package linalg;
```

29.3 利用可能な関数

29.3.1 add_columns, add_rows

`add_columns(`$\mathcal{A}$`,c1,c2,expr);`

$\mathcal{A}$:- 行列
c1,c2 :- 正の整数.
expr :- 式.

説明:

`add_columns` は行列 $\mathcal{A}$ の列 c2 を expr $*$ column($\mathcal{A}$,c1) + column($\mathcal{A}$,c2) で置き換えます。

`add_rows` は同様な計算を $\mathcal{A}$ の行に対して行います。

例:

$$\texttt{add\_columns}(\mathcal{A}, 1, 2, x) = \begin{pmatrix} 1 & x+2 & 3 \\ 4 & 4*x+5 & 6 \\ 7 & 7*x+8 & 9 \end{pmatrix}$$

$$\texttt{add\_rows}(\mathcal{A}, 2, 3, 5) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 27 & 33 & 39 \end{pmatrix}$$

関係する関数:

add_to_columns, add_to_rows, mult_columns, mult_rows.

29.3.2 add_rows

add_columns を見よ

29.3.3 add_to_columns, add_to_rows

add_to_columns($\mathcal{A}$,column_list,expr);

$\mathcal{A}$:- 行列.
column_list :- 正の整数もしくは正の整数のリスト.
expr :- 式.

説明:

add_to_columns は行列 $\mathcal{A}$ の column_list で指定した各列に式の値を加えます。

add_to_rows は同様のことを $\mathcal{A}$ の列に対して行います。

例:

$$\texttt{add\_to\_columns}(\mathcal{A},\{1,2\},10) = \begin{pmatrix} 11 & 12 & 3 \\ 14 & 15 & 6 \\ 17 & 18 & 9 \end{pmatrix}$$

$$\texttt{add\_to\_rows}(\mathcal{A},2,-x) = \begin{pmatrix} 1 & 2 & 3 \\ -x+4 & -x+5 & -x+6 \\ 7 & 8 & 9 \end{pmatrix}$$

関係する関数:

add_columns, add_rows, mult_rows, mult_columns.

29.3.4 add_to_rows

add_to_columns を見よ。

29.3.5 augment_columns, stack_rows

augment_columns($\mathcal{A}$,column_list);

$\mathcal{A}$:- 行列.
column_list :- 正の整数もしくは正の整数のリスト.

説明:

augment_columns は行列 $\mathcal{A}$ の column_list で指定された列を抜き出し、行列を作る。

stack_rows は同様の計算を行列 $\mathcal{A}$ に対して行う。

例:

$$\texttt{augment\_columns}(\mathcal{A},\{1,2\}) \quad = \quad \begin{pmatrix} 1 & 2 \\ 4 & 5 \\ 7 & 8 \end{pmatrix}$$

$$\texttt{stack\_rows}(\mathcal{A},\{1,3\}) \quad = \quad \begin{pmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{pmatrix}$$

関係する関数:

get_columns, get_rows, sub_matrix.

29.3.6 band_matrix

band_matrix(expr_list,square_size);

expr_list :- 式もしくは、奇数個の式のリスト.

square_size :- 正の整数.

説明:

band_matrix は square_size で指定されたサイズの正方行列を作る。対角要素は、式のリストの中央の要素の値になる。左対角成分 (sub diagonals) は、リストの中央から左の要素の値になる。右対角成分 (super diagonals) は、リストの中央から右の要素の値になる。

例:

$$\texttt{band\_matrix}(\{x,y,z\},6) \quad = \quad \begin{pmatrix} y & z & 0 & 0 & 0 & 0 \\ x & y & z & 0 & 0 & 0 \\ 0 & x & y & z & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \\ 0 & 0 & 0 & 0 & x & y \end{pmatrix}$$

関係する関数:

diagonal.

29.3.7 block_matrix

block_matrix(r,c,matrix_list);

r,c :- 正の整数.

matrix_list :- 行列のリスト.

説明:

block_matrix は $r \times c$ 行列を作る。行列の要素は、与えられた行列のリストから行方向に埋め込まれる。

例:

$$\mathcal{B} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathcal{C} = \begin{pmatrix} 5 \\ 5 \end{pmatrix}, \quad \mathcal{D} = \begin{pmatrix} 22 & 33 \\ 44 & 55 \end{pmatrix}$$

$$\texttt{block\_matrix}(2, 3, \{\mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{C}, \mathcal{B}\}) \quad = \quad \begin{pmatrix} 1 & 0 & 5 & 22 & 33 \\ 0 & 1 & 5 & 44 & 55 \\ 22 & 33 & 5 & 1 & 0 \\ 44 & 55 & 5 & 0 & 1 \end{pmatrix}$$

29.3.8 char_matrix

char_matrix($\mathcal{A}, \lambda$);

$\mathcal{A}$:- 正方行列.
λ :- シンボルまたは代数式.

説明:

char_matrix は行列 $\mathcal{A}$ の特性行列 $\mathcal{C}$ を作る。

これは、$\mathcal{C} = \lambda * \mathcal{I} - \mathcal{A}$ を満たすような行列のことです。

例:

$$\texttt{char\_matrix}(\mathcal{A}, x) \quad = \quad \begin{pmatrix} x-1 & -2 & -3 \\ -4 & x-5 & -6 \\ -7 & -8 & x-9 \end{pmatrix}$$

関係する関数:

char_poly.

29.3.9 char_poly

char_poly($\mathcal{A}, \lambda$);

$\mathcal{A}$:- 正方行列.
λ :- シンボルまたは代数式.

説明:

char_poly は、$\mathcal{A}$ の特性多項式を求める。

これは、$\lambda * \mathcal{I} - \mathcal{A}$ の行列式で与えられる。

例:

char_poly(A,x) $= x^3 - 15 * x^2 - 18 * x$

関係する関数:

char_matrix.

29.3.10 cholesky

cholesky($\mathcal{A}$);

$\mathcal{A}$:- 正定値行列で、要素は数値のみであるような行列。

説明:

cholesky は行列 $\mathcal{A}$ のコレスキー分解を求める。

これは、{$\mathcal{L}$,$\mathcal{U}$} を返します。ここで、$\mathcal{L}$ は下三角行列で、$\mathcal{U}$ は上三角行列です。これらの行列に対して、$\mathcal{A} = \mathcal{L}\mathcal{U}$, そして $\mathcal{U} = \mathcal{L}^T$ が成立する。

例:

$$\mathcal{F} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$\text{cholesky}(\mathcal{F}) = \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 1 & \sqrt{2} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 \\ 0 & \sqrt{2} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \right\}$$

関係する関数:

lu_decom.

29.3.11 coeff_matrix

coeff_matrix({lin_eqn$_1$,lin_eqn$_2$, ...,lin_eqn$_n$}); [1]

lin_eqn$_1$,lin_eqn$_2$, ...,lin_eqn$_n$:- 線形方程式、これらは、式 = 数値 の形式であるかそれとも単に式のみで与える。

説明:

coeff_matrix は、線形方程式系から、係数行列 $\mathcal{C}$ を作る。

これは、$\mathcal{C}\mathcal{X} = \mathcal{B}$ となるような行列 $\{\mathcal{C}, \mathcal{X}, \mathcal{B}\}$ を返す。

[1] {} を書くのが煩わしければ、省略しても構いません。

例:

coeff_matrix($\{x+y+4*z=10, y+x-z=20, x+y+4\}$) =

$$\left\{\begin{pmatrix} 4 & 1 & 1 \\ -1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \begin{pmatrix} z \\ y \\ x \end{pmatrix}, \begin{pmatrix} 10 \\ 20 \\ -4 \end{pmatrix}\right\}$$

29.3.12 column_dim, row_dim

column_dim($\mathcal{A}$);

$\mathcal{A}$:- 行列.

説明:

column_dim は行列 $\mathcal{A}$ の列の次元を求める。

row_dim は、行列 $\mathcal{A}$ の行の次元を求める。

例:

column_dim($\mathcal{A}$) = 3

29.3.13 companion

companion(poly,x);

poly :- モニックな変数 x に関する一変数多項式.
x :- 変数.

説明:

companion は多項式の同伴行列 (Companion matrix) $\mathcal{C}$ を求める。

これは、次元 n の正方行列で、n は多項式の x に関する次数に等しい。

行列 $\mathcal{C}$ の要素の値は:

$\mathcal{C}$(i,n) = -coeffn(poly,x,i-1) for i = 1 ... n,
$\mathcal{C}$(i,i-1) = 1 for i = 2 ... n
それ以外は 0.

例:

$$\text{companion}(x^4+17*x^3-9*x^2+11, x) = \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$$

関係する関数:

find_companion.

29.3.14 copy_into

copy_into($\mathcal{A}$,$\mathcal{B}$,r,c);

$\mathcal{A}$,$\mathcal{B}$:- 行列.

r,c :- 正の整数.

説明:

copy_into は行列 $\mathcal{A}$ を行列 $\mathcal{B}$ に、$\mathcal{A}$(1,1) は $\mathcal{B}$(r,c) に写されるように複写する。

例:

$$\mathcal{G} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\texttt{copy\_into}(\mathcal{A},\mathcal{G},1,2) = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 4 & 5 & 6 \\ 0 & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

関係する関数:

augment_columns, extend, matrix_augment, matrix_stack, stack_rows, sub_matrix.

29.3.15 diagonal

diagonal({mat$_1$,mat$_2$, ...,mat$_n$});[2]

mat$_1$,mat$_2$, ...,mat$_n$:- それぞれ式もしくは正方行列.

説明:

diagonal は入力された式もしくは行列を対角要素に持つような行列を作る。

例:

$$\mathcal{H} = \begin{pmatrix} 66 & 77 \\ 88 & 99 \end{pmatrix}$$

$$\texttt{diagonal}(\{\mathcal{A},x,\mathcal{H}\}) = \begin{pmatrix} 1 & 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 & 0 \\ 7 & 8 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 66 & 77 \\ 0 & 0 & 0 & 0 & 88 & 99 \end{pmatrix}$$

関係する関数:

jordan_block.

[2] {} を書くのが煩わしければ、省略しても構いません。

29.3.16 extend

extend($\mathcal{A}$,r,c,expr);

$\mathcal{A}$:- 行列.
r,c :- 正の整数.
expr :- 代数式もしくはシンボル.

説明:

extend は行列 $\mathcal{A}$ を $r \times c$ に拡張した行列を返す。拡張された要素の値は expr になる。

例:

$$\texttt{extend}(\mathcal{A}, 1, 2, x) = \begin{pmatrix} 1 & 2 & 3 & x & x \\ 4 & 5 & 6 & x & x \\ 7 & 8 & 9 & x & x \\ x & x & x & x & x \end{pmatrix}$$

関係する関数:

copy_into, matrix_augment, matrix_stack, remove_columns, remove_rows.

29.3.17 find_companion

find_companion($\mathcal{A}$,x);

$\mathcal{A}$:- 行列 matrix.
x :- 変数.

説明:

find_companion は、与えられた 同伴行列 (companion matrix) に対する同伴多項式を求める。

例:

$$\mathcal{C} = \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$$

$\texttt{find\_companion}(\mathcal{C}, x) = x^4 + 17 * x^3 - 9 * x^2 + 11$

関係する関数:

companion.

29.3.18 get_columns, get_rows

get_columns($\mathcal{A}$,column_list);

$\mathcal{A}$:- 行列.
c :- 正の整数または正の整数のリスト.

説明:

get_columns は行列 $\mathcal{A}$ から column_list で指定された列を取り除き、結果を列行列のリストの形式で返す。

get_rows は同様の計算を行列 $\mathcal{A}$ の行に対して行う。

例:

$$\texttt{get\_columns}(\mathcal{A},\{1,3\}) \;=\; \left\{\begin{pmatrix}1\\4\\7\end{pmatrix},\begin{pmatrix}3\\6\\9\end{pmatrix}\right\}$$

$$\texttt{get\_rows}(\mathcal{A},2) \;=\; \left\{\begin{pmatrix}4 & 5 & 6\end{pmatrix}\right\}$$

関係する関数:

augment_columns, stack_rows, sub_matrix.

29.3.19 get_rows

get_columns の項を参照のこと

29.3.20 gram_schmidt

gram_schmidt({vec$_1$,vec$_2$, ...,vec$_n$}); [3]

vec$_1$,vec$_2$, ...,vec$_n$:- 線形独立なベクトル。各ベクトルは {1,0,0} のようにリストの形式で書くこと。

説明:

gram_schmidt は入力されたベクトルに対して gram_schmidt の直交化を行う。

直交化された正規化ベクトルのリストを返す。

例:

gram_schmidt({{1,0,0},{1,1,0},{1,1,1}}) = {{1,0,0},{0,1,0},{0,0,1}}

gram_schmidt({{1,2},{3,4}}) = $\{\{\frac{1}{\sqrt{5}},\frac{2}{\sqrt{5}}\},\{\frac{2*\sqrt{5}}{5},\frac{-\sqrt{5}}{5}\}\}$

29.3.21 hermitian_tp

hermitian_tp($\mathcal{A}$);

$\mathcal{A}$:- 行列.

[3] {} を書くのが煩わしければ、省略しても構いません。

説明:

hermitian_tp は行列 $\mathcal{A}$ のエルミート共役を返す。

これは、 (i, j) 要素が行列 $\mathcal{A}$ の (j, i) 要素の複素共役となっている行列である。

例:

$$\mathcal{J} = \begin{pmatrix} i+1 & i+2 & i+3 \\ 4 & 5 & 2 \\ 1 & i & 0 \end{pmatrix}$$

$$\texttt{hermitian\_tp}(\mathcal{J}) \quad = \quad \begin{pmatrix} -i+1 & 4 & 1 \\ -i+2 & 5 & -i \\ -i+3 & 2 & 0 \end{pmatrix}$$

関係する関数:

tp[4] .

29.3.22 hessian

```
hessian(expr,variable_list);
```

expr :- 式.
variable_list :- 変数もしくは変数のリスト.

説明:

hessian は、variable_list に挙げた変数に関する expr のヘシアン (Hessian) を求める。

これは、$n \times n$ 行列、ただし n は変数の数でその (i, j) 要素は df(expr, variable_list(i), variable_list(j)) である。

例:

$$\texttt{hessian}(x*y*z+x^2, \{w,x,y,z\}) \quad = \quad \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & z & y \\ 0 & z & 0 & x \\ 0 & y & x & 0 \end{pmatrix}$$

関係する関数:

df[5] .

29.3.23 hilbert

```
hilbert(square_size,expr);
```

[4] 行列の転置演算については REDUCE のマニュアルを参照のこと.
[5] 微分演算については REDUCE のマニュアルを参照のこと

square_size :- 正の整数.
expr :- 代数式.

説明:

hilbert は square_size の次元のヒルベルト行列を求める。

これは、(i, j) 要素の値が 1/(i+j-expr) であるような対称行列である。

例:

$$\texttt{hilbert}(3, y+x) = \begin{pmatrix} \frac{-1}{x+y-2} & \frac{-1}{x+y-3} & \frac{-1}{x+y-4} \\ \frac{-1}{x+y-3} & \frac{-1}{x+y-4} & \frac{-1}{x+y-5} \\ \frac{-1}{x+y-4} & \frac{-1}{x+y-5} & \frac{-1}{x+y-6} \end{pmatrix}$$

29.3.24 jacobian

jacobian(expr_list,variable_list);

expr_list :- 代数式もしくは代数式のリスト.

variable_list :- 一変数もしくは変数のリスト.

説明:

jacobian は、expr_list の variable_list で指定された変数 (または変数リスト) に関するヤコビアンを求める。

これは、(i, j) 要素の値が df(expr_list(i), variable_list(j)) である行列である。

この行列は、サイズが $n \times m$ である。ここで n は変数の数で、m は与えられた式の数である。

例:

$\texttt{jacobian}(\{x^4, x*y^2, x*y*z^3\},\{w,x,y,z\}) =$

$$\begin{pmatrix} 0 & 4*x^3 & 0 & 0 \\ 0 & y^2 & 2*x*y & 0 \\ 0 & y*z^3 & x*z^3 & 3*x*y*z^2 \end{pmatrix}$$

関係する関数:

hessian, df[6] .

29.3.25 jordan_block

jordan_block(expr,square_size);

expr :- 代数式もしくはシンボル.
square_size :- 正の整数.

[6] 微分演算については REDUCE のマニュアルを参照のこと

説明:

jordan_block は次元が square_size であるジョルダン行列 $\mathcal{J}$ を求める。

行列 $\mathcal{J}$ の要素の値は：$\mathcal{J}$(i,i) = expr for i=1 ... n, $\mathcal{J}$(i,i+1) = 1 for i=1 ... n-1, それ以外はすべて 0.

例:

$$\texttt{jordan\_block(x,5)} = \begin{pmatrix} x & 1 & 0 & 0 & 0 \\ 0 & x & 1 & 0 & 0 \\ 0 & 0 & x & 1 & 0 \\ 0 & 0 & 0 & x & 1 \\ 0 & 0 & 0 & 0 & x \end{pmatrix}$$

関係する関数:

diagonal, companion.

29.3.26 lu_decom

lu_decom($\mathcal{A}$);

$\mathcal{A}$:- 数値係数または複素数係数の行列 .

説明:

lu_decom は $\mathcal{A}$ の LU 分解を求める。

ただし、$\mathcal{L}$ は下三角行列で、かつ $\mathcal{U}$ は上三角行列であり、これらの行列は $\mathcal{A} = \mathcal{LU}$ を満たす。

注意:

ここで使っているアルゴリズムは、計算の際に $\mathcal{A}$ の行を入れ替えることがあります。このため、$\mathcal{LU}$ は $\mathcal{A}$ とはならず、$\mathcal{A}$ の行を入れ替えた行列と等しくなることが起こります。このため、lu_decom は {$\mathcal{L}$,$\mathcal{U}$,vec} を返すようにしています。convert($\mathcal{A}$,vec) として、行列 $\mathcal{A}$ を行の変換を行うことによって、実際に分解された行列を求めることができます。つまり、$\mathcal{LU}$ = convert($\mathcal{A}$,vec) となります。

例:

$$\mathcal{K} = \begin{pmatrix} 1 & 3 & 5 \\ -4 & 3 & 7 \\ 8 & 6 & 4 \end{pmatrix}$$

$$\texttt{lu} := \texttt{lu\_decom}(\mathcal{K}) = \left\{ \begin{pmatrix} 8 & 0 & 0 \\ -4 & 6 & 0 \\ 1 & 2.25 & 1.1251 \end{pmatrix}, \begin{pmatrix} 1 & 0.75 & 0.5 \\ 0 & 1 & 1.5 \\ 0 & 0 & 1 \end{pmatrix}, [\,3\,2\,3\,] \right\}$$

$$\texttt{first lu * second lu} = \begin{pmatrix} 8 & 6 & 4 \\ -4 & 3 & 7 \\ 1 & 3 & 5 \end{pmatrix}$$

$$\texttt{convert}(\mathcal{K},\texttt{third lu}) = \begin{pmatrix} 8 & 6 & 4 \\ -4 & 3 & 7 \\ 1 & 3 & 5 \end{pmatrix}$$

$$\mathcal{P} = \begin{pmatrix} i+1 & i+2 & i+3 \\ 4 & 5 & 2 \\ 1 & i & 0 \end{pmatrix}$$

$$\texttt{lu} := \texttt{lu\_decom}(\mathcal{P}) = \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 4 & -4*i+5 & 0 \\ i+1 & 3 & 0.41463*i+2.26829 \end{pmatrix}, \begin{pmatrix} 1 & i & 0 \\ 0 & 1 & 0.19512*i+0.24390 \\ 0 & 0 & 1 \end{pmatrix}, [\ 3\ 2\ 3\ ] \right\}$$

$$\texttt{first lu * second lu} = \begin{pmatrix} 1 & i & 0 \\ 4 & 5 & 2 \\ i+1 & i+2 & i+3 \end{pmatrix}$$

$$\texttt{convert(P,third lu)} = \begin{pmatrix} 1 & i & 0 \\ 4 & 5 & 2 \\ i+1 & i+2 & i+3 \end{pmatrix}$$

関係する関数:

cholesky.

29.3.27 make_identity

```
make_identity(square_size);
```

square_size :- 正の整数.

説明:

make_identity は、次元が square_size である恒等行列を作成する。

例:

$$\texttt{make\_identity}(4) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

関係する関数:

diagonal.

29.3.28 matrix_augment, matrix_stack

matrix_augment({mat$_1$,mat$_2$, ...,mat$_n$});[7]

mat$_1$,mat$_2$, ...,mat$_n$:- 行列.

説明:

matrix_augment は matrix_list で指定した行列を横に並べて作った行列を返す。

matrix_stack は同様に matrix_list で指定した行列を縦に並べた行列を作る。

例:

$$\texttt{matrix\_augment}(\{\mathcal{A},\mathcal{A}\}) = \begin{pmatrix} 1 & 2 & 3 & 1 & 2 & 3 \\ 4 & 4 & 6 & 4 & 5 & 6 \\ 7 & 8 & 9 & 7 & 8 & 9 \end{pmatrix}$$

$$\texttt{matrix\_stack}(\{\mathcal{A},\mathcal{A}\}) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

関係する関数:

augment_columns, stack_rows, sub_matrix.

29.3.29 matrixp

matrixp(test_input);

test_input :- 任意.

説明:

matrixp は、引数が行列であれば t を、それ以外は nil を返す。

例:

matrixp($\mathcal{A}$) = t

matrixp(doodlesackbanana) = nil

関係する関数:

squarep, symmetricp.

[7] {} を書くのが煩わしければ、省略しても構いません。

29.3.30 matrix_stack

matrix_augment を見よ。

29.3.31 minor

minor($\mathcal{A}$,r,c);

$\mathcal{A}$:- 行列.
r,c :- 正の整数.

説明:

minor は $\mathcal{A}$ の (r,c) 要素に対する余因子行列を求める。これは、行列 $\mathcal{A}$ から、r 番目の行と c 番目の列を取り除いたものである。

例:

$$\texttt{minor}(\mathcal{A},1,3) \quad = \quad \begin{pmatrix} 4 & 5 \\ 7 & 8 \end{pmatrix}$$

関係する関数:

remove_columns, remove_rows.

29.3.32 mult_columns, mult_rows

mult_columns($\mathcal{A}$,column_list,expr);

$\mathcal{A}$:- 行列.
column_list :- 正の整数または正の整数のリスト.
expr :- 代数式.

説明:

mult_columns は、行列 $\mathcal{A}$ で、column_list で指定された列を expr 倍した行列のコピーを返す。

mult_rows は同様の計算を行に対して行う。

例:

$$\texttt{mult\_columns}(\mathcal{A},\{1,3\},x) \quad = \quad \begin{pmatrix} x & 2 & 3*x \\ 4*x & 5 & 6*x \\ 7*x & 8 & 9*x \end{pmatrix}$$

$$\texttt{mult\_rows}(\mathcal{A},2,10) \quad = \quad \begin{pmatrix} 1 & 2 & 3 \\ 40 & 50 & 60 \\ 7 & 8 & 9 \end{pmatrix}$$

関係する関数:

add_to_columns, add_to_rows.

29.3.33 mult_rows

mult_columns を見よ。

29.3.34 pivot

pivot($\mathcal{A}$,r,c);

$\mathcal{A}$:- 行列.
r,c :- $\mathcal{A}$(r,c) が 0 でない正の整数.

説明:

pivot は、行列 $\mathcal{A}$ を (r,c) 要素をピボットとして、消去を行う。

つまり、r 番目の行の倍数を行列の他のすべての行に加えることで、c 番目の列で (r, c) 以外の要素をすべて 0 にする。

例:

$$\texttt{pivot}(\mathcal{A},2,3) = \begin{pmatrix} -1 & -0.5 & 0 \\ 4 & 5 & 6 \\ 1 & 0.5 & 0 \end{pmatrix}$$

関係する関数:

rows_pivot.

29.3.35 pseudo_inverse

pseudo_inverse($\mathcal{A}$);

$\mathcal{A}$:- 行列.

説明:

pseudo_inverse は行列 $\mathcal{A}$ の擬逆行列 (もしくは Moore-Penrose 逆行列) を求める。

行列 $\mathcal{A}$ の特異値分解、$\mathcal{A} = \mathcal{U}\sum\mathcal{V}^T$、から擬逆行列 $\mathcal{A}^{-1}$ は $\mathcal{A}^{-1} = \mathcal{V}^T\sum^{-1}\mathcal{U}$ で定義される。

従って、$\mathcal{A} * \texttt{pseudo\_inverse}(\mathcal{A}) = \mathcal{I}$ を満たす。

例:

$$\texttt{pseudo\_inverse}(\mathcal{A}) = \begin{pmatrix} -0.2 & 0.1 \\ -0.05 & 0.05 \\ 0.1 & 0 \\ 0.25 & -0.05 \end{pmatrix}$$

関係する関数:

svd.

29.3.36 random_matrix

`random_matrix(r,c,limit);`

r,c, limit :- 正の整数.

説明:

`random_matrix` は、$r \times c$ 行列を生成する。行列の要素は、$-\text{limit} < \text{entry} < \text{limit}$ の範囲内の値を取る。

スイッチ:

`imaginary` :- オンの時、要素の値が x+i*y ただし $-\text{limit} < \text{x,y} < \text{limit}$ である行列を作る。

`not_negative` :- オンの時、$0 <$ 要素 $<$ limit となる行列を作る。複素数値の場合には、$0 < \text{x,y} < \text{limit}$ となる。

`only_integer` :- オンの時、整数値を要素とする行列を作る。複素数値の場合には、$x + yi$ で、x と y は共に整数値となる。

`symmetric` :- オンの時、対称行列を生成する.

`upper_matrix` :- オンの時、上三角行列を生成する.

`lower_matrix` :- オンの時、下三角行列を生成する。

例:

$$\texttt{random\_matrix}(3,3,10) = \begin{pmatrix} -4.729721 & 6.987047 & 7.521383 \\ -5.224177 & 5.797709 & -4.321952 \\ -9.418455 & -9.94318 & -0.730980 \end{pmatrix}$$

```
on only_integer, not_negative, upper_matrix, imaginary;
```

$$\texttt{random\_matrix}(4,4,10) = \begin{pmatrix} 2*i+5 & 3*i+7 & 7*i+3 & 6 \\ 0 & 2*i+5 & 5*i+1 & 2*i+1 \\ 0 & 0 & 8 & i \\ 0 & 0 & 0 & 5*i+9 \end{pmatrix}$$

29.3.37 remove_columns, remove_rows

`remove_columns(`$\mathcal{A}$`,column_list);`

$\mathcal{A}$:- 行列.

column_list :- 正の整数または正の整数のリスト.

説明:

`remove_columns` は、$\mathcal{A}$ から column_list で指定された列を取り除く。

`remove_rows` は同様の演算を行に対して行う。

例:

$$\texttt{remove\_columns}(\mathcal{A}, 2) = \begin{pmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{pmatrix}$$

$$\texttt{remove\_rows}(\mathcal{A}, \{1,3\}) = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$$

関係する関数:

minor.

29.3.38 remove_rows

remove_columns を見よ。

29.3.39 row_dim

column_dim を見よ。

29.3.40 rows_pivot

rows_pivot(A,r,c,{row_list});

$\mathcal{A}$	:-	行列.
r,c	:-	$\mathcal{A}$(r,c) が 0 とならない正の整数.
row_list	:-	正の整数、または正の整数のリスト.

説明:

rows_pivot は、pivot と同様の演算を行う。ただし、ピボットは row_list で指定された行についてのみ行う。

例:

$$\mathcal{N} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

$$\texttt{rows\_pivot}(\mathcal{N}, 2, 3, \{4,5\}) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ -0.75 & 0 & 0.75 \\ -0.375 & 0 & 0.375 \end{pmatrix}$$

関係する関数:

pivot.

29.3.41 simplex

simplex(max/min, 関数,{ 線形不等式 });

max/min :- max または min (最大値か最小値かを指定する).
関数 :- 最大/最小値を求めようとする関数.
線形不等式 :- 制約条件. 各々は変数の和 *(<=, =, >=)* 数値の形式の不等式でなければなりません。.

説明:

simplex は、線形の不等式 (制約条件) のもとでの、関数の最適値 (最大もしくは最小値) を求める。

これは、{ 最適値,{ 最適値を取る変数の値 }} を返す。

このアルゴリズムでは、すべての変数はゼロまたは正の値を取ると仮定している。

例:

simplex($max, x+y, \{x >= 10, y >= 20, x+y <= 25\}$);

***** Error in simplex: Problem has no feasible solution.

(***** simplex のエラー: 問題は有効な解を持ちません。)

simplex($max, 10x+5y+5.5z, \{5x+3z <= 200, x+0.1y+0.5z <= 12,$
$0.1x+0.2y+0.3z <= 9, 30x+10y+50z <= 1500\}$);

$\{525.0, \{x = 40.0, y = 25.0, z = 0\}\}$

29.3.42 squarep

squarep($\mathcal{A}$);

$\mathcal{A}$:- 行列.

説明:

squarep は行列が正方であれば t を返し、それ以外は nil を返します。

例:

$\mathcal{L} = \begin{pmatrix} 1 & 3 & 5 \end{pmatrix}$

squarep($\mathcal{A}$) = t

squarep($\mathcal{L}$) = nil

関係する関数:

matrixp, symmetricp.

29.3.43 stack_rows

augment_columns を見よ。

29.3.44 sub_matrix

sub_matrix($\mathcal{A}$,row_list,column_list);

$\mathcal{A}$:- 行列.
row_list, column_list :- 正の整数または正の整数のリスト .

説明:

sub_matrix は、row_list で指定された行と、column_list で指定された列からなる部分行列を返します。

例:

$$\texttt{sub\_matrix}(\mathcal{A}, \{1,3\}, \{2,3\}) = \begin{pmatrix} 2 & 3 \\ 8 & 9 \end{pmatrix}$$

関係する関数:

augment_columns, stack_rows.

29.3.45 svd (特異値分解)

svd($\mathcal{A}$);

$\mathcal{A}$:- 数値を要素に持つ行列.

説明:

svd は、 $\mathcal{A}$ の特異値分解を求めます。

これは、$\{\mathcal{U}, \sum, \mathcal{V}\}$ を返します。ただし、$\mathcal{A} = \mathcal{U} \sum \mathcal{V}^T$ 及び $\sum = diag(\sigma_1, \ldots, \sigma_n)$. σ_i $i = (1 \ldots n)$ は $\mathcal{A}$ の特異値です。

ここで、n は行列 $\mathcal{A}$ の列ベクトルの次元です。

$\mathcal{A}$ の特異値とは $\mathcal{A}^T\mathcal{A}$ の固有値の負でない平方根です。

$\mathcal{U}$ と $\mathcal{V}$ は、$\mathcal{U}\mathcal{U}^T = \mathcal{V}\mathcal{V}^T = \mathcal{V}^T\mathcal{V} = \mathcal{I}_n$ を満たします。

例:

$$\mathcal{Q} = \begin{pmatrix} 1 & 3 \\ -4 & 3 \end{pmatrix}$$

$$\texttt{svd}(\mathcal{Q}) \quad = \quad \left\{ \begin{pmatrix} 0.289784 & 0.957092 \\ -0.957092 & 0.289784 \end{pmatrix}, \begin{pmatrix} 5.149162 & 0 \\ 0 & 2.913094 \end{pmatrix}, \begin{pmatrix} -0.687215 & 0.726453 \\ -0.726453 & -0.687215 \end{pmatrix} \right\}$$

29.3.46 swap_columns, swap_rows

`swap_columns(`$\mathcal{A}$`,c1,c2);`

$\mathcal{A}$:- 行列.
c1,c1 :- 正の整数.

説明:

`swap_columns` は、行列 $\mathcal{A}$ の列 c1 と列 c2 を交換します。

`swap_rows` は同様のことを、$\mathcal{A}$ の二つの行に対して行います。

例:

$$\texttt{swap\_columns}(\mathcal{A}, 2, 3) \quad = \quad \begin{pmatrix} 1 & 3 & 2 \\ 4 & 6 & 5 \\ 7 & 9 & 8 \end{pmatrix}$$

関係する関数:

`swap_entries`.

29.3.47 swap_entries

`swap_entries(`$\mathcal{A}$`,{r1,c1},{r2,c2});`

$\mathcal{A}$:- 行列.
r1,c1,r2,c2 :- 正の整数.

説明:

`swap_entries` は、$\mathcal{A}$(r1,c1) と $\mathcal{A}$(r2,c2) を交換します。

例:

$$\texttt{swap\_entries}(\mathcal{A}, \{1,1\}, \{3,3\}) \quad = \quad \begin{pmatrix} 9 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 1 \end{pmatrix}$$

関係する関数:

`swap_columns`, `swap_rows`.

29.3.48 swap_rows

swap_columns を見よ。

29.3.49 symmetricp

symmetricp($\mathcal{A}$);

$\mathcal{A}$:- 行列.

説明:

symmetricp は行列が対称であれば t を、そうでなければ nil を返す関数です。

例:

$$\mathcal{M} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

symmetricp($\mathcal{A}$) = nil

symmetricp($\mathcal{M}$) = t

関係する関数:

matrixp, squarep.

29.3.50 toeplitz

toeplitz({expr_1,expr_2, ...,expr_n}); [8]

expr_1,expr_2, ...,expr_n :- 代数式.

説明:

toeplitz は toeplitz 行列を生成します。

これは、最初の式が対角要素で、i 番目の式が (i-1) 番目の sub diagonal および super diagonal の要素となるような行列です。

この行列の次元 n は、式の数に等しくなります。

例:

$$\mathtt{toeplitz}(\{w,x,y,z\}) = \begin{pmatrix} w & x & y & z \\ x & w & x & y \\ y & x & w & x \\ z & y & x & w \end{pmatrix}$$

[8] {} を書くのが煩わしければ、省略しても構いません。

29.3.51 triang_adjoint

`triang_adjoint`($\mathcal{A}$);

$\mathcal{A}$:- 行列.

説明:

`triang_adjoint` は行列 A の三角化共役行列 $\mathcal{F}$ を Arne Storjohann によるアルゴリズムで計算します。$\mathcal{F}$ は下三角行列で、$\mathcal{F} * \mathcal{A} = \mathcal{T}$ で得られる行列 T は、i 番めの対角成分が A の i 番めの主部分行列の行列式であるような、上三角行列になります。

例:

$$\texttt{triang\_adjoint}(\mathcal{A}) = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ -3 & 6 & -3 \end{pmatrix}$$

$$\mathcal{F} * \mathcal{A} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 0 \end{pmatrix}$$

29.3.52 Vandermonde

`vandermonde`({expr_1,expr_2, ...,expr_n}); [8]

expr_1,expr_2, ...,expr_n :- 代数式

説明:

`Vandermonde` は式のリストから、Vandermonde 行列を生成します。

これは、(i, j) 要素が expr_list(i) $^{(j-1)}$ であるような正方行列で、その次元 n は式の数に等しくなります。

例:

$$\texttt{vandermonde}(\{x, 2*y, 3*z\}) = \begin{pmatrix} 1 & x & x^2 \\ 1 & 2*y & 4*y^2 \\ 1 & 3*z & 9*z^2 \end{pmatrix}$$

29.3.53 kronecker_product

`kronecker_product`(Mat_1, Mat_2)

Mat_1, Mat_2 :- 行列

説明:

`kronecker_product` は引数の Kronecker 積 (直積またはテンソル積とも呼ばれる) を返します。

例:

```
a1 := mat((1,2),(3,4),(5,6))$
a2 := mat((1,1,1),(2,z,2),(3,3,3))$
kronecker_product(a1,a2);
```

$$\begin{pmatrix} 1 & 1 & 1 & 2 & 2 & 2 \\ 2 & z & 2 & 4 & 2*z & 4 \\ 3 & 3 & 3 & 6 & 6 & 6 \\ 3 & 3 & 3 & 4 & 4 & 4 \\ 6 & 3*z & 6 & 8 & 4*z & 8 \\ 9 & 9 & 9 & 12 & 12 & 12 \\ 5 & 5 & 5 & 6 & 6 & 6 \\ 10 & 5*z & 10 & 12 & 6*z & 12 \\ 15 & 15 & 15 & 18 & 18 & 18 \end{pmatrix}$$

29.4 計算の高速化

`fast_la` スイッチをオンにすることで、 次の関数の計算速度を早くすることが出来ます。

add_columns	add_rows	augment_columns	column_dim
copy_into	make_identity	matrix_augment	matrix_stack
minor	mult_column	mult_row	pivot
remove_columns	remove_rows	rows_pivot	squarep
stack_rows	sub_matrix	swap_columns	swap_entries
swap_rows	symmetricp		

計算速度の向上は、非常に多くの回数 (数千回) の呼び出しを行うような場合以外はあまり顕著ではありません。このスイッチを使うと、エラーの検査を最小限にしか行いません。このため、不正な入力に対して不可解なエラーメッセージが出力されることがあります。使用には気を付けて下さい。

29.5 謝辞

このパッケージのアイデアの多くは Maple[3] の線形代数パッケージ [4] から得られました。

`cholesky`、`lu_decom` と `svd` のアルゴリズムは、J.H. Wilkinson と C. Reinsch[5] による著作「Linear Algebra」によるものです。

`gram_schmidt` のプログラムは Karin Gatermann による Symmetry パッケージ [6] からのものです。

第30章　LINENEQ:線形不等式

H.Melenk

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Heilbronner Strasse 10
D–10711 Berlin – Wilmersdorf
Federal Republic of Germany
E–mail: *melenk@sc.zib–berlin.de*

は定数係数の線形の不等式の解を求めるパッケージです。によって作成されています。例えば、

```
load linineq;       %パッケージをロードする;
linineq({ 5x1 - 4x2 + 13x3 - 2x4 + x5 = 20,
x1 - x2 + 5x3 - x4 + x5 = 8,
x1 + 6x2 - 7x3 + x4 + 5x5 = z,
x1>=0,x2>=0,x3>=0,x4>=0,x5>=0}, {z=min});
```

とすれば、

$$5x_1 - 4x_2 + 13x_3 - 2x_4 + x_5 = 20$$

$$x_1 - x_2 + 5x_3 - x_4 + x_5 = 8$$

$$x_1 + 6x_2 - 7x_3 + x_4 + 5x_5 = z$$

$$x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_4 \geq 0, x_5 \geq 0$$

の解でzの値が最小であるような解を求めます。結果は

$$\{X5 = 0, X4 = 0, X3 = \frac{12}{7}, X2 = \frac{4}{7}, X1 = 0, Z = -\frac{60}{7}\}$$

となります。今のプログラムでは等号を含まない$>$や$<$という不等号を含んだような不等式の解は求められません。

第31章　MODSR: 合同演算による方程式の解の計算

Herbert Melenk

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin–Dahlem, Germany

e–mail: *melenk@zib.de*

このパッケージでは (M_SOLVE) と (M_ROOTS) 演算子を定義しています。これは、モジュラー多項式やモジュラー多項式系の解を求めます。法は素数である必要はありません。M_SOLVE ではモジュラーは集合である必要があります。M_ROOTS は第二引数に法をとります。例えば、

```
on modular; setmod 8;
m_solve(2x=4);             ->  {{X=2},{X=6}}
m_solve({x^2-y^3=3});
    ->  {{X=0,Y=5}, {X=2,Y=1}, {X=4,Y=5}, {X=6,Y=1}}
m_solve({x=2,x^2-y^3=3}); ->  {{X=2,Y=1}}
off modular;
m_roots(x^2-1,8);          ->  {1,3,5,7}
m_roots(x^3-x,7);          ->  {0,1,6}
```

第32章　MRVLIMIT: 指数–対数関数の極限値

Neil Langmead
Konrad-Zuse-Zentrum für Informationstechnik (ZIB)
Takustrasse 7
D- 14195 Berlin Dahlem
Berlin, Germany

32.1　指数–対数関数の極限

このパッケージは、Dominik Gruntz, of the ETH Zürich の PhD 論文からのものです。彼は、”exp-log”関数の極限を計算する新しいアルゴリズムを開発しました。現在の REDUCE の極限パッケージでは計算できない多くの極限値が計算できるようになります。一番簡単な例は、次のような極限値の計算です。

```
load limits;

limit(x^7/e^x,x,infinity);

        7
       x
limit(----,x,infinity)
        x
       e
```

このパッケージでは、

$$f(x) = e^x * \log(\log(x))$$

$$f(x) = \frac{\log(\log(x + e^{-x}))}{e^{x^2} + \log(\log(x))}$$

$$f(x) = \log(x)^{\log(x)}$$

$$f(x) = e^{x*\log(x)}$$

といった関数を含んだ式の極限値の計算が行なえます。

32.2 Mrv_limit の例

```
mrv_limit(e^x,x,infinity);

infinity

mrv_limit(1/log(x),x,infinity);

0

b:=e^x*(e^(1/x-e^-x)-e^(1/x));


           -1        - x
      x + x      - e
b := e       *(e          - 1)


mrv_limit(b,x,infinity);


-1


                                        -1
  ex:=  - log(log(log(log(x))) + log(x))  *log(x)

                       *(log(log(x)) - log(log(log(x)) + log(x)));


            - log(x)*(log(log(x)) - log(log(log(x)) + log(x)))
ex:=     ------------------------------------------------------
                    log(log(log(log(x))) + log(x))

off mcd;

mrv_limit(ex,x,infinity);

1


(log(x+e^-x)+log(1/x))/(log(x)*e^x);
```

```
  - x       -1       -1          - x
e    *log(x)  *(log(x  ) + log(e     + x));

mrv_limit(ws,x,infinity);

0

mrv_limit((log(x)*e^-x)/e^(log(x)+e^(x^2)),x,infinity);

0
```

32.3 トレース機能

スイッチ tracelimit をオンにすることによって、*mrv_limit* 関数の計算の過程をトレースすることができます。

第33章　NCPOLY: 非可換多項式イデアル

Herbert Melenk

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin–Dahlem, Germany

e–mail: *melenk@zib.de*

Joachim Apel

Institut für Informatik, Universität Leipzig
Augustusplatz 10–11
D–04109 Leipzig, Germany

e–mail: *apel@informatik.uni–leipzig.de*

REDUCE は非可換な要素の積を計算する一般的な機構を用意している。ここで、交換関係はルール集合として明確に定義する必要がある。NCPOLY パッケージでは非可換性が Lie ブラケットで定義されるような代数計算を自動的に行なうように設定できる。このパッケージは REDUCE 基本的な多項式の演算には **noncom** 機構を使っている。交換関係の規則は Lie ブラケットから自動的に計算されます。多項式演算は**除算**や**因数分解**も含めて直接実行される。更に、**NCPOLY** は片側イデアル (左もしくは右) の計算、とくに片側 Gröbner 基底と多項式の簡約が行なえます。

33.1　設定と消去

計算の前に nc_setup を呼び出して、非可換代数の計算のための環境を設定しておく必要があります。

```
nc_setup(<vars>[,<comms>][,<dir>]);
```

ここで、$< vars >$ は変数のリストで、非可換要素を含んでいるなければなりません。

$< comms >$ は式<u>*<v> - <v>*<u>=<rh> のリストで、$< u >$ と $< v >$ は $< vars >$ の要素です。また、$< rh >$ は多項式です。

$< dir >$ は $left$ もしくは $right$ のいずれかで、左もしくは右イデアルの選択を行ないます。初期値は $left$ です。

nc_setup は $< comms >$ から、全ての単項式が与えられてた変数の順序に対応して順序付けられた代数をサポートするのに必要なルールを生成します。明確に交換子の集合でカバーされていない全ての変数のペアは可換であると仮定され、必要なルールが活性化されます。

nc_setup の第二引数はこの関数が二度めに呼び出された場合、例えば変数の順序を変更して呼び出すような場合、省略できます。この場合には、以前に指定して交換関係が再度使われます。

注意:

- 変数を **noncom** で宣言する必要はありません。**nc_setup** は必要な宣言を行ないます。
- 変数は形式的な関数の形をとる必要はありません。**nc_setup** は演算子 $nc!*$ が非可換性を保持しているにもかかわらず、内部で変数 x を `nc!*(!_x)` と表しています。
- コマンド **order** や **korder** は使えません。**nc_setup** はそれらの順序を項の正しく順序付けられて出力されるようにを設定します。

例:

```
nc_setup({KK,NN,k,n},
           {NN*n-n*NN= NN, KK*k-k*KK= KK});

NN*N;              ->   NN*N
N*NN;              ->   NN*N - NN
nc_setup({k,n,KK,NN});
NN*N - NN          ->   N*NN;
```

ここで、KK, NN, k, n は非可換な変数で、交換関係は $[NN, n] = NN$, $[KK, k] = KK$ と記述されています。

現在の項順序は交換関係と整合していなければいけません。積 $< u > * < v >$ は現在の項順序で、右辺 $< rh >$ の全ての項より前でないといけません。従って、

- $< rh >$ に含まれる $< u >$ や $< v >$ の最大次数は 1 です。
- 全次数順序では、$< rh >$ の全次数は多分 1 より大きくはない。
- elimination 次数順序 (例えば lex) では、$< rh >$ の全ての変数は $< u >$ か $< v >$ の小さい方よりも順序が下でなければなりません。
- もし $< rh >$ がどんな変数も含んでいないか、もしくは高々$< u >$ あるいは $< v >$ を含んでいる場合、任意の項順序が選択可能です。

非可換変数や、非可換な計算の結果を後に可換な演算に使用する場合、非可換演算モードを切替える必要があります。なぜならば、REDUCE は全ての演算でそのように動作するように環境を設定してしまっているからです。このような目的のために、引数を持たないコマンド nc_cleanup があります。

```
nc_cleanup;
```

これは、nc_setup によって設定された、全ての内部規則や定義を取り除きます。再度、非可換での計算モードに切替えるには、nc_setup をもう一度呼び出します。

33.2 左および右イデアル

(多項式の) 左イデアル L は次の公理で定義されています。

$u \in L, v \in L \Longrightarrow u + v \in L$

任意の多項式 k に対して、$u \in L \Longrightarrow k * u \in L$

ここで、"$*$" は非可換な積です。同様に右イデアル R は次のように定義されます。

$u \in R, v \in R \Longrightarrow u + v \in R$

任意の多項式 k に対して、$u \in R \Longrightarrow u * k \in R$

33.3 Gröbner 基底

nc_setup によって非可換な環境が設定されれば、左もしくは右多項式イデアルの基底を Gröbner 基底に変換することが関数 nc_groebner で可能になります。

```
nc_groebner(<plist>);
```

変数集合と変数の順序はこれ以前の nc_setup の呼び出しで定義されていなければなりません。Gröbner 計算における項順序は torder 宣言を使うことによって設定できます。

torder の詳しい説明は **REDUCE GROEBNER** マニュアルか、22 章を参照して下さい。.

```
2: nc_setup({k,n,NN,KK},{NN*n-n*NN=NN,KK*k-k*KK=KK},left);

3: p1 := (n-k+1)*NN - (n+1);

p1 :=  - k*nn + n*nn - n + nn - 1

4: p2 := (k+1)*KK -(n-k);

p2 := k*kk + k - n + kk

5: nc_groebner ({p1,p2});

{k*nn - n*nn + n - nn + 1,

 k*kk + k - n + kk,

 n*nn*kk - n*kk - n + nn*kk - kk - 1}
```

GROEBNER パッケージの関数は非可換の積について考慮していないので、直接使うことができないことに注意して下さい。

33.4 多項式の左および右除算

演算子 nc_divide は二つの多項式の片側商および除余を計算します。

```
nc_divide(<p1>,<p2>);
```

結果は、商と除余からなるリストです。除算は擬除算として、$< p1 >$ に必要な係数を掛けたのち、計算が実行されます。結果の $\{< q >, < r >\}$ は関係式

$< c > * < p1 >=< q > * < p2 > + < r >$ $left$ 方向に対して, そして

$< c > * < p1 >=< p2 > * < q > + < r >$ $right$ 方向に対して,

で定義されます。ここで、$< c >$ イデアルの変数を含まない式であり、$< r >$ はその主項が $< p2 >$ の主項よりも現在の項順序でひくい多項式です。

33.5 多項式の左および右簡約

与えられた多項式の集合を法として、多項式の片側除余を計算するためには、演算子 nc_preduce が利用できます。

```
nc_preduce(<polynomial>,<plist>);
```

簡約の結果は、もし $< plist >$ が片側 Gröbner 基底である時に、そしてこの時に限って唯一 (正準) になります。従って、計算は同時にイデアルのメンバーシップの判定になります。もし結果がゼロであれば、多項式はイデアルのメンバーであり、ゼロでなければメンバーでありません。

33.6 因数分解

非可換環上の多項式は通常の REDUCE の factorize コマンドでは因数分解できません。この代わりに、この節では演算子、

```
nc_factorize(<polynomial>);
```

を使わなくてはなりません。結果は、$< polynomial >$ の因数のリストです。もし、既約であればそれ自身のみのリストが返されます。

非可換な因数分解はただ一つだけではありません。可能な全ての分解を計算する演算子があります。

```
nc_factorize_all(<polynomial>);
```

結果は、$< polynomial >$ を分解する因数のリストです。もし、全く因子に分解できなければ、入力された多項式それ自身のみを持つリストが返されます。

33.7 式の出力

非可換演算では、可換部分 (係数) が因数分解されていることが望ましいことがしばしばある。演算子

```
nc_compact(<polynomial>)
```

は、係数を最も非可換性が小さい変数巾の係数を集めます。

```
load_package ncpoly;

nc_setup({n,NN},{NN*n-n*NN=NN})$
p1 := n**4 + n**2*nn + 4*n**2 + 4*n*nn + 4*nn + 4;

       4    2         2
p1 := n  + n *nn + 4*n  + 4*n*nn + 4*nn + 4

nc_compact p1;

  2     2          2
(n  + 2)  + (n + 2) *nn
```

第34章　NORMFORM: 行列の標準形の計算

Matt Rebbeck

Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Strasse 10
D–10711 Berlin – Wilmersdorf
Federal Republic of Germany

E–mail: *neun@sc.zib–berlin.de*

34.1　はじめに

相似な行列は同じトレース、行列式、特性方程式、固有値を持っている。しかし、次の行列

$$\mathcal{U} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad と \quad \mathcal{V} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

は、上の四つの値は同じであるが、相似ではない。もしそうでなければ、特異でない行列$\mathcal{N} \in M_2$ (2×2 行列全体の集合) が存在して、$\mathcal{U} = \mathcal{N}\,\mathcal{V}\,\mathcal{N}^{-1} = \mathcal{N}\,0\,\mathcal{N}^{-1} = 0$, が成立する。これは$\mathcal{U} \neq 0$ に矛盾する。

二つの行列が相似であるか否かを判定する一つの方法は、標準形に変換することです。もし、両方の行列が同じ標準形になれば、それらは相似です。[28]

NORMFORM は次のような行列の標準形を計算するパッケージです。

- smithex
- smithex_int
- frobenius
- ratjordan
- jordansymbolic
- jordan

通常計算は有理数体 Q 上で行なわれます。`smithex`, `frobenius`, `ratjordan`, `jordansymbolic` および `jordan` に体しては、もっと拡張された体の上での計算が可能です。

`frobenius`, `ratjordan` および `jordansymbolic` の標準形は、モジュラー基底の上ででも計算可能です。

NORMFORM は、T.M.L. Mulders and A.H.M. Levelt[44][43] によって書かれた Maple[22] のパッケージ Normform を書き直したものです。

34.2 smithex

スミスの標準形は次のような対角行列 $\mathcal{S}$ です。

- rank($\mathcal{A}$) = 行列 $\mathcal{S}$ のゼロでない行 (列) の数
- $0 < i \leq$ rank($\mathcal{A}$) に対して、$\mathcal{S}(i, i)$ はモニックな多項式
- $0 < i <$ rank($\mathcal{A}$) に対して、$\mathcal{S}(i, i)$ は $\mathcal{S}(i+1, i+1)$ を割り切る。
- 全ての i に対して、$\mathcal{S}(i, i)$ は、$\mathcal{A}$ の i 小行列式全ての最大公約式である。

34.2.1 関数

`smithex`($\mathcal{A}$, x) はスミスの標準形 $\mathcal{S}$ を計算します。

これは、$\{\mathcal{S}, \mathcal{P}, \mathcal{P}^{-1}\}$ を返します。ここで、$\mathcal{S}, \mathcal{P}$, および $\mathcal{P}^{-1}$ は、$\mathcal{P}\mathcal{S}\mathcal{P}^{-1} = \mathcal{A}$ を満たす行列です。

$\mathcal{A}$ は x についての一変数多項式を要素とする

34.2.2 体の拡大

計算は $\mathcal{Q}$ で行なわれます。この体を拡張するには、ARNUM パッケージを使います。

34.2.3 例

$$\mathcal{A} = \begin{pmatrix} x & x+1 \\ 0 & 3*x^2 \end{pmatrix}$$

$$\texttt{smithex}(\mathcal{A}, x) = \left\{ \begin{pmatrix} 1 & 0 \\ 0 & x^3 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 3*x^2 & 1 \end{pmatrix}, \begin{pmatrix} x & x+1 \\ -3 & -3 \end{pmatrix} \right\}$$

34.3 smithex_int

34.3.1 関数

与えられた n 行 m 列の整数値のみの行列 $\mathcal{A}$ に対して、スミスの標準形を計算します。

これは、$\{\mathcal{S}, \mathcal{P}, \mathcal{P}^{-1}\}$ を返します。ここで、$\mathcal{S}, \mathcal{P}$, および $\mathcal{P}^{-1}$ は、$\mathcal{P}\mathcal{S}\mathcal{P}^{-1} = \mathcal{A}$ を満たす行列です。

34.3.2 例

$$\mathcal{A} = \begin{pmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{pmatrix}$$

`smithex_int`$(\mathcal{A}) =$

$$\left\{ \begin{pmatrix} 3 & 0 & 0 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{pmatrix}, \begin{pmatrix} -17 & -5 & -4 \\ 64 & 19 & 15 \\ -50 & -15 & -12 \end{pmatrix}, \begin{pmatrix} 1 & -24 & 30 \\ -1 & 25 & -30 \\ 0 & -1 & 1 \end{pmatrix} \right\}$$

34.4 frobenius

フロベニウスの標準形 $\mathcal{F}$ は次の構造を持っています。

$$\mathcal{F} = \begin{pmatrix} \mathcal{C}(p_1) & & & \\ & \mathcal{C}(p_2) & & \\ & & \ddots & \\ & & & \mathcal{C}(p_k) \end{pmatrix}$$

ここで、$\mathcal{C}(p_i)$ は $p_1, p_2, \ldots, p_k$ に付随したコンパニオン行列で、$i = 1 \ldots k-1$ に対して、p_i は p_{i+1} を割り切ります。

このようにして定義されたフロベニウスの標準形は唯一に決まります。(つまり、p_i が p_{i+1} を割り切るように定義すると)

34.4.1 関数

`frobenius`$(\mathcal{A})$ は行列 $\mathcal{A}$ のフロベニウスの標準形 $\mathcal{F}$ を計算します。

これは、$\{\mathcal{F}, \mathcal{P}, \mathcal{P}^{-1}\}$ を返します。ここで、$\mathcal{F}, \mathcal{P}$, および $\mathcal{P}^{-1}$ は、$\mathcal{P}\mathcal{F}\mathcal{P}^{-1} = \mathcal{A}$ を満たす行列です。

$\mathcal{A}$ は正方行列です。

34.4.2 体の拡大

計算は $\mathcal{Q}$ で行なわれます。この体を拡張するには、ARNUM パッケージを使います。

34.4.3 合同演算

frobenius は素数を法とした合同演算の元ででも、計算可能です。

34.4.4 例

$$\mathcal{A} = \begin{pmatrix} \frac{-x^2+y^2+y}{y} & \frac{-x^2+x+y^2-y}{y} \\ \frac{-x^2-x+y^2+y}{y} & \frac{-x^2+x+y^2-y}{y} \end{pmatrix}$$

frobenius($\mathcal{A}$) =

$$\left\{ \begin{pmatrix} 0 & \frac{x*(x^2-x-y^2+y)}{y} \\ 1 & \frac{-2*x^2+x+2*y^2}{y} \end{pmatrix}, \begin{pmatrix} 1 & \frac{-x^2+y^2+y}{y} \\ 0 & \frac{-x^2-x+y^2+y}{y} \end{pmatrix}, \begin{pmatrix} 1 & \frac{-x^2+y^2+y}{x^2+x-y^2-y} \\ 0 & \frac{-y}{x^2+x-y^2-y} \end{pmatrix} \right\}$$

34.5 ratjordan

有理ジョルダン標準形 $\mathcal{R}$ は次の構造を持ちます。

$$\mathcal{R} = \begin{pmatrix} r_{11} & & & & & & \\ & r_{12} & & & & & \\ & & \ddots & & & & \\ & & & r_{21} & & & \\ & & & & r_{22} & & \\ & & & & & \ddots & \end{pmatrix}$$

r_{ij} は次のような形です。

$$r_{ij} = \begin{pmatrix} \mathcal{C}(p) & \mathcal{I} & & & \\ & \mathcal{C}(p) & \mathcal{I} & & \\ & & \ddots & \ddots & \\ & & & \mathcal{C}(p) & \mathcal{I} \\ & & & & \mathcal{C}(p) \end{pmatrix}$$

ここで、eij 個の $\mathcal{C}(p)$ ブロックが対角上に存在します。$\mathcal{C}(p)$ は既約多項式 p のコンパニオン行列です。

34.5.1 関数

ratjordan($\mathcal{A}$) は行列 $\mathcal{A}$ の有理 Jordan 標準形 $\mathcal{R}$ を計算します。

これは、$\{\mathcal{R}, \mathcal{P}, \mathcal{P}^{-1}\}$ を返します。ここで、$\mathcal{R}, \mathcal{P}$ および $\mathcal{P}^{-1}$ は $\mathcal{P}\mathcal{R}\mathcal{P}^{-1} = \mathcal{A}$ が成り立つ行列です。

$\mathcal{A}$ は正方行列です。

34.5.2 体の拡大

計算は $\mathcal{Q}$ で行なわれます。この体を拡張するには、ARNUM パッケージを使います。

34.5.3 モジュラー演算

ratjordan は素数を法とした合同演算の元ででも、計算可能です。

34.5.4 例

$$\mathcal{A} = \begin{pmatrix} x+y & 5 \\ y & x^2 \end{pmatrix}$$

ratjordan($\mathcal{A}$) =

$$\left\{ \begin{pmatrix} 0 & -x^3 - x^2 * y + 5 * y \\ 1 & x^2 + x + y \end{pmatrix}, \begin{pmatrix} 1 & x+y \\ 0 & y \end{pmatrix}, \begin{pmatrix} 1 & \frac{-(x+y)}{y} \\ 0 & \frac{1}{y} \end{pmatrix} \right\}$$

34.6 jordansymbolic

34.6.1 関数

jordansymbolic($\mathcal{A}$) は行列 $\mathcal{A}$ のジョルダン標準形 $\mathcal{J}$ を計算します。

これは、$\{\mathcal{J}, \mathcal{L}, \mathcal{P}, \mathcal{P}^{-1}\}$ を返します。ここで、$\mathcal{J}, \mathcal{P}$ および $\mathcal{P}^{-1}$ は、$\mathcal{P}\mathcal{J}\mathcal{P}^{-1} = \mathcal{A}$ を満たす行列です。また、$\mathcal{L} = \{\ ll\ , \xi\ \}$、ただし、$\xi$ は名前で、 ll は $p(\xi)$ の既約因子のリストです。

$\mathcal{A}$ は正方行列です。

34.6.2 体の拡大

計算は $\mathcal{Q}$ で行なわれます。この体を拡張するには、ARNUM パッケージを使います。

34.6.3 モジュラー演算

jordansymbolic は素数を法とした合同演算の元ででも、計算可能です。

34.6.4 例

```
on looking_good;
```

$$\mathcal{A} = \begin{pmatrix} 1 & y \\ y^2 & 3 \end{pmatrix}$$

jordansymbolic($\mathcal{A}$) =

$$\left\{ \begin{pmatrix} \xi_{11} & 0 \\ 0 & \xi_{12} \end{pmatrix}, \left\{ \left\{ -y^3 + \xi^2 - 4 * \xi + 3 \right\}, \xi \right\}, \right.$$
$$\left. \begin{pmatrix} \xi_{11} - 3 & \xi_{12} - 3 \\ y^2 & y^2 \end{pmatrix}, \begin{pmatrix} \frac{\xi_{11}-2}{2*(y^3-1)} & \frac{\xi_{11}+y^3-1}{2*y^2*(y^3+1)} \\ \frac{\xi_{12}-2}{2*(y^3-1)} & \frac{\xi_{12}+y^3-1}{2*y^2*(y^3+1)} \end{pmatrix} \right\}$$

solve($-\mathtt{y}^3 + \mathtt{xi}^2 - 4 * \mathtt{xi} + 3, \mathtt{xi}$);

$$\{\xi = \sqrt{y^3+1} + 2, \xi = -\sqrt{y^3+1} + 2\}$$

$\mathcal{J}$ = sub({xi(1, 1) = sqrt($\mathtt{y}^3 + 1$) + 2, xi(1, 2) = $-$sqrt($\mathtt{y}^3 + 1$) + 2},
first jordansymbolic ($\mathcal{A}$));

$$\mathcal{J} = \begin{pmatrix} \sqrt{y^3+1} + 2 & 0 \\ 0 & -\sqrt{y^3+1} + 2 \end{pmatrix}$$

34.7 jordan

34.7.1 関数

jordan($\mathcal{A}$) は行列 $\mathcal{A}$ のジョルダン標準形 $\mathcal{J}$ を計算します。

これは、$\{\mathcal{J}, \mathcal{P}, \mathcal{P}^{-1}\}$ を返します。ここで、$\mathcal{J}, \mathcal{P}$ および $\mathcal{P}^{-1}$ は、$\mathcal{P}\mathcal{J}\mathcal{P}^{-1} = \mathcal{A}$ を満たす行列です。

$\mathcal{A}$ は正方行列です。

34.7.2 体の拡大

計算は $\mathcal{Q}$ で行なわれます。この体を拡張するには、ARNUM パッケージを使います。

34.7.3 注意

ある多項式の場合、全ての根を計算するために fullroots がオンで計算が行なわれます。このため、計算に長時間かかることがあります。また、結果が非常に長大になることがあります。このような計算では、jordansymbolic を使った方がよいでしょう。

34.7.4 例

$$\mathcal{A} = \begin{pmatrix} -9 & -21 & -15 & 4 & 2 & 0 \\ -10 & 21 & -14 & 4 & 2 & 0 \\ -8 & 16 & -11 & 4 & 2 & 0 \\ -6 & 12 & -9 & 3 & 3 & 0 \\ -4 & 8 & -6 & 0 & 5 & 0 \\ -2 & 4 & -3 & 0 & 1 & 3 \end{pmatrix}$$

$\mathcal{J}$ = first jordan($\mathcal{A}$);

$$\mathcal{J} = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & i+2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -i+2 \end{pmatrix}$$

34.8 arnum

このパッケージは load_package arnum; でロードされます。$\mathcal{Q}$ を代数拡大した体上で演算を行なうことができます。例えば、defpoly sqrt2**2-2; は $\sqrt{2}$ (ここでは、sqrt2 と定義した). を付け加えて拡大した体上で計算が行なえます。ARNUM パッケージは Eberhard Schrüfer によって作成されました。

34.8.1 例

```
load_package normform;
load_package arnum;
defpoly sqrt2**2-2;
```
(以下では見やすさを考えて、sqrt2 を $\sqrt{2}$ として表しています。)

$$\mathcal{A} = \begin{pmatrix} 4*\sqrt{2}-6 & -4*\sqrt{2}+7 & -3*\sqrt{2}+6 \\ 3*\sqrt{2}-6 & -3*\sqrt{2}+7 & -3*\sqrt{2}+6 \\ 3*\sqrt{2} & 1-3*\sqrt{2} & -2*\sqrt{2} \end{pmatrix}$$

$$\texttt{ratjordan}(\mathcal{A}) \quad = \quad \left\{ \begin{pmatrix} \sqrt{2} & 0 & 0 \\ 0 & \sqrt{2} & 0 \\ 0 & 0 & -3*\sqrt{2}+1 \end{pmatrix}, \right.$$

$$\left(\begin{array}{ccc} 7*\sqrt{2}-6 & \frac{2*\sqrt{2}-49}{31} & \frac{-21*\sqrt{2}+18}{31} \\ 3*\sqrt{2}-6 & \frac{21*\sqrt{2}-18}{31} & \frac{-21*\sqrt{2}+18}{31} \\ 3*\sqrt{2}+1 & \frac{-3*\sqrt{2}+24}{31} & \frac{3*\sqrt{2}-24}{31} \end{array} \right),$$
$$\left(\begin{array}{ccc} 0 & \sqrt{2}+1 & 1 \\ -1 & 4*\sqrt{2}+9 & 4*\sqrt{2} \\ -1 & -\frac{1}{6}*\sqrt{2}+1 & 1 \end{array} \right) \Bigg\}$$

34.9 モジュラー

合同計算はスイッチ modular をオンにすることで計算できます。法は `setmod p;` で設定できます。ここで p 素数です。標準形は $\mathcal{Z}/\mathrm{p}\mathcal{Z}$ の要素を持ちます。

`on balanced_mod;` でスイッチを設定することにより、、対称なモジュラー表現を使った出力を得ることができます。

このモジュラーの操作に関する説明は、REDUCE User's Manual [27] の 9 章 (多項式と有理式) にあります。

34.9.1 例

```
load_package normform;
on modular;
setmod 23;
```

$$\mathcal{A} = \left(\begin{array}{cc} 10 & 18 \\ 17 & 20 \end{array} \right)$$

`jordansymbolic`$(\mathcal{A}) =$

$$\left\{ \left(\begin{array}{cc} 18 & 0 \\ 0 & 12 \end{array} \right), \{\{\lambda+5, \lambda+11\}, \lambda\}, \left(\begin{array}{cc} 15 & 9 \\ 22 & 1 \end{array} \right), \left(\begin{array}{cc} 1 & 14 \\ 1 & 15 \end{array} \right) \right\}$$

```
on balanced_mod;
```

`jordansymbolic`$(\mathcal{A}) =$

$$\left\{ \left(\begin{array}{cc} -5 & 0 \\ 0 & -11 \end{array} \right), \{\{\lambda+5, \lambda+11\}, \lambda\}, \left(\begin{array}{cc} -8 & 9 \\ -1 & 1 \end{array} \right), \left(\begin{array}{cc} 1 & -9 \\ 1 & -8 \end{array} \right) \right\}$$

第35章　NUMERIC:数値計算

Herbert Melenk

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Heilbronner Strasse 10
D–10711 Berlin – Wilmersdorf
Federal Republic of Germany

E–mail: *melenk@sc.zib–berlin.de*

NUMERIC パッケージでは、REDUCE の小数点数 (近似数) を基にした、数値 (近似) 算法をインプリメントしています。これらの算法は通常の場合に使えるようなアルゴリズムで、特別な悪条件下での問題には使えません。このような場合には、通常の数学ライブラリを利用してください。

35.1　構文

35.1.1　区間数、出発点

区間数 (Interval) は下限と上限を演算子‘..’で結合した形で表されます。通常、変数と等号で結ばれた形でよく使われます。例えば、

```
x= (2.5 .. 3.5)
```

は変数 x がその値として 2.5 から 3.5 までの数値を取ることを表しています。ここで、限界値は数式であっても構いませんが、評価した結果は数値になるような式でなければなりません。区間数が結果として返された場合、上限値もしくは下限値は、PART 演算子を使うことにより、それぞれ最初もしくは二番目の要素として取り出すことが出来ます。出発点 (starting point) は右辺値が数値であるような等式で表されます。例えば、

```
x=3.0
```

もし、多変数の式を扱う場合で、いくつかの座標についてそれを区間もしくは出発点として定義したい場合、それらを一つのパラメータにまとめて (リストとして) 与えることも出来ますし、また各々について別のパラメータとして与えることもできます。リスト形式でパラメータを与える方法は、そのリストを別の REDUCE のプログラムによって作成されるというように自動的に作成する場合には便利です。しかし、対話的に計算する場合には、別々のパラメータとして与える方法の方が便利でしょう。

35.1.2 精度の制御

キーワードパラメータ $accuracy = a$ および $iterations = i$、ここで a と i は正の整数、は逐次近似の計算を制御します。繰り返しの回数は誤差が 10^{-a} 以下になるまで行われます。また i 回の繰り返しを行ってもこれ以下にならない場合は、繰り返しを中止します。この時にはエラーメッセージが出力されます。いずれにせよ、その時点で得られた値が結果として返されます。

35.1.3 トレース

通常、計算時には必要最小限の出力しか行われません。もし計算結果がおかしい場合や、予期したよりも計算時間がかかるような場合には、次のように入力することで、繰り返し計算の途中の様子 (トレース) を出力されることが出来ます。

```
on trnumeric;
```

35.2 極小値

Fletcher Reeves による最沈降下法による算法を使って、一変数もしくは多変数関数の極小値を求めます。関数は全ての変数について連続な偏微分を持っている必要があります。極小値をとる点をどこから最初に捜していくか、その出発点を指定する必要があります。もし指定しない場合には、ランダムな値を出発点として計算します。最沈降下法では一般に極小値しか見つけることが出来ません。

構文:

NUM_MIN $(< 式 >, < 変数_1 > [= val_1][, < 変数_2 > [= val_2] \ldots]$

$[, accuracy = a][, iterations = i])$

または

NUM_MIN $(< 式 >, \{< 変数_1 > [= val_1][, < 変数_2 > [= val_2] \ldots]\}$

$[, accuracy = a][, iterations = i])$

ここで $< 式 >$ は関数式です。

$< 変数_1 >, < 変数_2 >, \ldots$ は $< 式 >$ に現れる変数名です。$val_1, val_2, \ldots$ は各々の変数に対する出発点で、これは指定しなくても構いません。

NUM_MIN は与えられた点から始めて、傾きが最も大きい接線に沿って下っていき、より小さい関数値を与える点を捜していきます。結果は、関数の最小値とそのときの各変数の値がリストの形で返されます。ここで変数の値は、変数名=値 という等式のリストとして返されます。

例:

```
num_min(sin(x)+x/5, x);
```

```
{4.9489585606,{X=29.643767785}}

num_min(sin(x)+x/5, x=0);

{ - 1.3342267466,{X= - 1.7721582671}}

% Rosenbrock の関数 (最小値を計算するのが難しい例としてよく知られている)
fktn := 100*(x1**2-x2)**2 + (1-x1)**2;
num_min(fktn, x1=-1.2, x2=1, iterations=200);

{0.00000021870228295,{X1=0.99953284494,X2=0.99906807238}}
```

35.2.1 関数の零点/方程式の解

適応型のニュートン法を使って関数の零点および方程式の近似値を求めます。内部では方程式に対して、左辺から右辺を引いた式 (関数) にニュートン法が適用されます。関数 (式) は全ての変数に関して連続な導関数を持っていなければなりません。出発点を指定することも出来ます。指定されない場合はランダムに選んだ初期値から計算を行います。方程式の数と未知変数の数が一致しない場合にはニュートン法は使えません。このような場合には式の絶対値の自乗の和が最小となる点が求められます。COMPLEX スイッチがオンで、方程式もしくは初期値が零でない虚数部を持っている場合には、複素数解が求められます。

構文:

NUM_SOLVE $(< \text{式}_1 >, < \text{変数}_1 > [= val_1][, accuracy = a][, iterations = i])$

または

NUM_SOLVE $(\{< \text{式}_1 >, \ldots, < \text{式}_n >\}, < \text{変数}_1 > [= val_1], \ldots, < \text{変数}_n > [= val_n]$

$[, accuracy = a][, iterations = i])$

または

NUM_SOLVE $(\{< \text{式}_1 >, \ldots, < \text{式}_n >\}, \{< \text{変数}_1 > [= val_1], \ldots, < \text{変数}_n > [= val_n]\}$

$[, accuracy = a][, iterations = i])$

ここで $< \text{式}_1 >, \ldots, < \text{式}_n >$ は関数で、

$< \text{変数}_1 >, \ldots, < \text{変数}_n >$ は未知変数を表す。

$val_1, \ldots, val_n$ は初期値で、これは指定しなくても構いません。

SOLVE は式の零点を求めます。結果は変数=値の形の等式のリストを返します。

計算の副作用として JACOBIAN という名前の変数にヤコビ行列が代入されます。

例:

```
num_solve({sin x=cos y, x + y = 1},{x=1,y=2});

{X= - 1.8561957251,Y=2.856195584}

jacobian;

[COS(X)  SIN(Y)]
[               ]
[  1       1    ]
```

35.3 数値積分

有限区間での数値積分は次のように計算されます:

1. もし、関数が積分区間内で不定積分を持てば、定積分は区間の両端での不定積分の値の差で求められます。
2. そうでなければ、Chebyshev 近似を計算します。これは 20 次から始めて、最大 80 次までの近似式が求められます。もし、これが十分収束していると判断されれば、この近似式の積分から値を求めます。
3. もし上の二つの方法がうまくいかない場合は、適応型の quadrature 法を使います。

多変数の積分に対しては、適応型の quadrature 法のみが使えます。この算法は分離された特異点を持っていても使うことが出来ます。値 *iterations* は局所的な区間の交差レベルを制限します。また *accuracy* は差分近似の相対誤差を指定します。

構文:

NUM_INT $(< \text{式} >, < \text{変数}_1 >= (l_1..u_1)[, < \text{変数}_2 >= (l_2..u_2)\ldots]$

$[, accuracy = a][, iterations = i])$

ここで $< \text{式} >$ は被積分関数です。

$< \text{変数}_1 >, < \text{変数}_2 >, \ldots$ は積分を行う変数です。

$l_1, l_2, \ldots$ は積分の下限で、

$u_1, u_2, \ldots$ は上限です。

結果は定積分の値を返します。

例:

```
num_int(sin x,x=(0 .. pi));

2.0000010334
```

35.4 常微分方程式

常微分方程式の実初期値に対する数値解を 3 次の Runge-Kutta 法を使って求めます。

構文:

NUM_ODESOLVE $(< 式 >, depvar_1 = start_1, indep = (from..to)$

$[, accuracy = a][, iterations = i])$

または、常微分方程式系に対しては

NUM_ODESOLVE $(\{< 式_1 >, < 式_2 > \ldots\}, \{depvar_1 = start_1, depvar_2 = start_2, \ldots\},$

$indep = (from..to)[, accuracy = a][, iterations = i])$

ここで

$depvar_1, \ldots$ および $start_1, \ldots$ は独立変数およびその初期値を指定しています。

$indep$, $from$ および to はそれぞれ独立変数、および積分区間 (出発点および終点) です。

$< 式_1 >, < 式_2 >, \ldots$ は等式 (または数式を与えてもよい。この場合は数式=0 なる方程式の解が求められる。) で、これは従属変数の独立変数に関する一階導関数を含んでいなくてはなりません。

常微分方程式は explicit な形式に変換されます。その式に対して Runge Kutta の方法が適用されます。繰り返しの回数は引数 i で指定されます。(通常 20)。ステップが粗すぎて求める精度に達しない場合には、この数字は自動的に増加されます。

結果は、常微分方程式の近似解を表す (変数の値と関数の値の) 対がリストの形で返されます。

例:

```
num_odesolve(df(y,x)=y,y=1,x=(0 .. 1), iterations=5);

{{0.0,1.0},{0.2,1.2214},{0.4,1.49181796},{0.6,1.8221064563},

 {0.8,2.2255208258},{1.0,2.7182511366}}
```

注意:

- もし < 式 > で微分の部分が左辺に分離されていない場合は、従属変数が DEPEND 文で明確に宣言されていることを保証して下さい。例えば、

  ```
  depend y,x;
  ```

  さもなければ形式的な微分の部分は REDUCE によって評価されてしまい、零になってしまいます。

- REDUCE の SOLVE パッケージを使って式は常微分方程式の形に変形されます。もしこの変形に失敗した場合はエラーを出して計算は終了します。

35.5 関数の値域

実数値関数の区間、もしくは多変数の場合は矩形領域内での上限および下限は、BOUNDS 演算子によって計算できます。計算の方法は、不等式の演算を使って行っています。変数の区間から出発して、不等号の計算規則に基づいて計算を行っていきます。ROUNDED モードがオンであれば、ABS, SIN, COS, EXP, LOG や分数べき乗等の特殊関数についてはその値が計算されます。(そうでなければ代数的な演算規則のみを使って評価されます。)

もし BOUNDS が区間内で特異点を見つけた場合、エラーを出力して計算は終了してしまいます。

構文:

BOUNDS $(< 式 >, < 変数_1 >= (l_1..u_1)[, < 変数_2 >= (l_2..u_2)\ldots])$

BOUNDS $(< 式 >, \{< 変数_1 >= (l_1..u_1)[, < 変数_2 >= (l_2..u_2)\ldots]\})$

ここで $< 式 >$ は考えている関数で、

$< 変数_1 >, < 変数_2 >, \ldots$ は $< 式 >$ 中の変数です。

$l_1, l_2, \ldots$ および $u_1, u_2, \ldots$ は変数の変域 (区間) を表します。

BOUNDS は指定された変数の変域での上限と下限を計算します。結果として区間を返します。

例:

```
bounds(sin x,x=(1 .. 2));

{-1,1}

on rounded;
bounds(sin x,x=(1 .. 2));

0.84147098481 .. 1

bounds(x**2+x,x=(-0.5 .. 0.5));

 - 0.25 .. 0.75
```

35.6 Chebyshev 近似

CHEBYSHEV_で始まる演算子のグループはチェビシェフ法による関数の近似と評価を行います。$T_n^{(a,b)}(x)$ を区間 (a, b) へ変換された n 次のチェビシェフ多項式とすると、関数 $f(x)$ は区間 (a, b) 内で次のような級数で近似されます。

$f(x) \approx \sum_{i=0}^{N} c_i T_i^{(a,b)}(x)$

関数 Chebyshev_fit は与えられた関数に対してこの近似式を求め、結果として最初の要素は級数の和である近似多項式、二番目の要素はチェビシェフ係数 c_i の列であるようなリストを返します。Chebyshev_df と Chebyshev_int はチェビシェフ係数のリストをそれぞれ微分または積分した係数リストに変換します。指定された点でのチェビシェフ近似を求めるには Chebyshev_eval を使うことが出来ます。Chebyshev_eval は再帰関係式に基づいて計算を行います。これは通常、多項式を直接計算するよりも、より安定した方法です。

CHEBYSHEV_FIT $(fcn, <\text{変数}>=(lo..hi), n)$

CHEBYSHEV_EVAL $(coeffs, <\text{変数}>=(lo..hi), <\text{変数}>=pt)$

CHEBYSHEV_DF $(coeffs, <\text{変数}>=(lo..hi))$

CHEBYSHEV_INT $(coeffs, <\text{変数}>=(lo..hi))$

ここで fcn は代数式で、$<$ 変数 $>$ は関数 fcn の変数、lo と hi は区間を表す実数値 $(lo < hi)$ です。整数 n は近似の次数で、もし指定されていない場合は 20 に設定されます。pt は区間内の数値で、$coeffs$ はチェビシェフ係数の列で、これは CHEBYSHEV_COEFF,CHEBYSHEV_DF または CHEBYSHEV_INT で求めた係数列を与えます。

例:

```
on rounded;

w:=chebyshev_fit(sin x/x,x=(1 .. 3),5);

                3           2
w := {0.03824*x  - 0.2398*x  + 0.06514*x + 0.9778,

      {0.8991,-0.4066,-0.005198,0.009464,-0.00009511}}

chebyshev_eval(second w, x=(1 .. 3), x=2.1);

0.4111
```

35.7 一般的な曲線による近似

演算子 NUM_FIT は指定された点を通る曲線を、指定された関数 (基底) の線形結合で近似します。この近似は最小自乗法 (誤差の自乗の和が最小となるように係数を決定する) で行われます。これは誤差の自乗の和を係数で微分した式が零となる方程式の解として求められます。

構文は

NUM_FIT $(vals, basis, <\text{変数}>=pts)$

ここで $vals$ は数値のリスト (値) で、

$<$ 変数 $>$ は近似で使う変数名、

pts はリストで与えた値を取る点での変数 $<$ 変数 $>$ の値です。

$basis$ は関数の列 (基底) でこの関数の線形結合として近似されます。

結果はリストの形で得られます。最初の要素は近似した多項式で、二番目の要素はこの近似多項式を基底で構成したときのそれぞれの係数をリストの形で表しています。

例:

```
 % 階乗関数を多項式で近似
pts:=for i:=1 step 1 until 5 collect i$
vals:=for i:=1 step 1 until 5 collect
        for j:=1:i product j$

num_fit(vals,{1,x,x**2},x=pts);

               2
{14.571428571*X  - 61.428571429*X + 54.6,{54.6,

     - 61.428571429,14.571428571}}

num_fit(vals,{1,x,x**2,x**3,x**4},x=pts);

               4                 3
{2.2083333234*X  - 20.249999879*X

                  2
  + 67.791666154*X  - 93.749999133*X

  + 44.999999525,

 {44.999999525, - 93.749999133,67.791666154,

   - 20.249999879,2.2083333234}}
```

第36章　ODESOLVE:常微分方程式の解

Malcolm A.H. MacCallum
Queen Mary and Westfield College, London
Email: *mm@maths.qmw.ac.uk*
Other contributors: **Francis Wright, Alan Barnes**

ODESOLVE パッケージは常微分方程式の解を求めます。現在のところ、非常に限られた問題に対してのみ使えます。

1. 一つのスカラー方程式で表された式のみが扱えます。

   また

2. 簡単な形の一階の常微分方程式、定数係数の線形方程式およびオイラー方程式のみを解くことが出来ます。

これらの解くことが出来る方程式の型は Lie 対称性の理論がなんら有効な情報を与えない方程式のクラスになっています。

36.1　使用法

ユーザが通常起動する関数は:

ODESOLVE(EXPRN:**式、方程式**,
　　VAR1:**変数**,
　　VAR2:**変数**):**リスト**

で、ODESOLVE は SOLVE と同様に方程式のリストを返します。

EXPRN は一つのスカラー式で、EXPRN = 0 が解こうとしている常微分方程式となるような式か、もしくはそれと同等な方程式。

VAR1 は従属変数名。

VAR2 は独立変数名。

(簡単のため以下ではこれらはそれぞれyとxとします。) 結果は解を表す式で、これには ARBCONST 演算子による任意定数が含まれています。

それ以外の関数としては、

SORTOUTODE(EXPRN:**代数式**, Y:**変数**, X:**変数**): **式**

が用意されています。これは EXPRN をYに関する微分方程式として、その ORDER や次数、この方程式のYへの線形性、最大の微分回数を変数 ODEORDER,ODEDEGREE ODELINEARITY,HIGHESTDERIV に設定します。 等価な常微分方程式もしくは、EXPRN が指定された変数に関する微分方程式でなければ 0 が、返されます。

36.2 トレース

TRODE スイッチによって計算のトレースを取ることが出来ます。(TRFAC や TRINT と同じ)

第37章　ORTHOVEC: 直交曲座標での3次元ベクトル解析

James W. Eastwood

AEA Technology
Culham Laboratory
Abingdon
Oxon OX14 3DB

Email: *eastwood#jim%nersc.mfenet@ccc.nersc.gov*

June 1990

ORTHOVEC の改訂版は、REDUCE 3.4 でスカラーとベクトルを操作するための関数と演算子の集まりです。これに含まれる演算は加算、減算、内積、外積、除算、束、発散 (div)、勾配 (grad)、curl、ラプラシアン (laplacian)、微分、積分、$\mathbf{a}\cdot\nabla$ それにテイラー展開です。第二版は [19] に要約されています。これは前の版 ([18]) と比べて表記が変更されたこととおよび機能が追加されている点が異なっています。

37.1　はじめに

ORTHOVEC の改訂版 [19] はもとの版 [18] と同じように、応用数学の多くの分野で現れるベクトルの操作や展開を行うための REDUCE の関数と演算子から構成されています。この改訂版では最初の版では欠けていた機能を追加し、使いやすくするために入出力を整理した。第一版からの変更点は

1. スカラーとベクトルに対する演算子 $+, -, *, /$ を統一した。
2. 割り算と微分の定義を拡張してベクトルに対しても演算できるようにした。
3. 新しいベクトルの依存関係の演算子を追加した。
4. 極限とテイラー展開の演算にロピタルの定理を使うようにした。
5. 要素を取り出す演算子を定義した。
6. LISP のベクトル要素を代数モードで出力できるようにした。

ORTHOVEC パッケージの関数は、初期設定、入出力、代数演算、微分演算と積分演算の五つに分けられる。これらの定義は以下の節で説明されている。

37.2 初期設定

関数 VSTART は ORTHVEC を初期設定します。これは最初に ORTHOVEC パッケージが読み込まれたときに実行され座標系を設定します。VSTART は用意された座標系からメニュー形式で選択できるようになっています。

1. cartesian (x, y, z) = `(x, y, z)`
2. cylindrical (r, θ, z) = `(r, th, z)`
3. spherical (r, θ, ϕ) = `(r, th, ph)`
4. general (u_1, u_2, u_3) = `(u1, u2, u3)`
5. その他

この内から数字で選択します。(1)-(4) を選択すると、座標系とスカラー因子は自動的に設定されます。(5) を選択するとどの様にして座標系を設定するかを表示します。もし、VSTART が一度も呼ばれないときには、標準として直交座標系が使われます。REDUCE の使用中にいつでも

```
VSTART $
```

と入力することで新しい座標系に再初期化することが出来ます。

37.3 入出力

ORTHOVEC は、全ての入力がスカラーかもしくは 3 次元ベクトルであると仮定しています。a を (c_1, c_2, c_3) を要素とするベクトルとして定義するには、SVEC を使って、次のように入力します。

```
a := svec(c1, c2, c3);
```

終端記号として ";" を使うと、ベクトルは要素を括弧で囲って、$[\cdots]$ と出力します。VOUT (これは引数を結果の値として返します。) は代数モードでラベル付けられた要素の出力を行うのに使えます。例えば、

```
b := svec (sin(x)**2, y**2, z)$
vout(b)$
```

演算子 _ は、ベクトルのいずれか特定の要素 (1、2 または 3) を出力するのに使います。例えば、

```
b_1 ;
```

37.4 代数演算

六つの内挿演算子、加算、引算、除算、乗算、べき乗、外積、および四つの前置演算子、正、負、逆、束がORTHOVECで定義されています。これらはスカラーとベクトルのそれぞれ適当な組合せで使うことが出来ます。引数がスカラーの場合は通常の演算が行われます。

これらの演算はそれぞれ記号

`+, -, /, *, ^, ><`

で表されます。

いま、**v** はベクトルを、*s* はスカラーを表すとします。この時、可能な引数の組合せを以下に示しています。ここで使っている記号の意味は、
結果 *:=***関数** *(***左側の引数***,* **右側の引数***)* または
結果 *:=(***左側の被演算式***)* **演算子** *(***右側の被演算式***)* .

ベクトルの加算

v	:=	VECTORPLUS(**v**)	または	**v**	:=	+ **v**
s	:=	VECTORPLUS(s)	または	s	:=	+ s
v	:=	VECTORADD(**v**,**v**)	または	**v**	:=	**v** + **v**
s	:=	VECTORADD(s,s)	または	s	:=	s + s

ベクトルの引算

v	:=	VECTORMINUS(**v**)	または	**v**	:=	+ **v**
s	:=	VECTORMINUS(s)	または	s	:=	+ s
v	:=	VECTORDIFFERENCE(**v**,**v**)	または	**v**	:=	**v** + **v**
s	:=	VECTORDIFFERENCE(s,s)	または	s	:=	s + s

ベクトルの除算

v	:=	VECTORRECIP(**v**)	または	**v**	:=	/ **v**
s	:=	VECTORRECIP(s)	または	s	:=	/ s
v	:=	VECTORQUOTIENT(**v**,**v**)	または	**v**	:=	**v** / **v**
v	:=	VECTORQUOTIENT(**v**, s)	または	**v**	:=	**v** / s
v	:=	VECTORQUOTIENT(s ,**v**)	または	**v**	:=	s / **v**
s	:=	VECTORQUOTIENT(s,s)	または	s	:=	s / s

ベクトルの乗算

v	:=	VECTORTIMES(s ,**v**)	または	**v**	:=	s * **v**
v	:=	VECTORTIMES(**v**, s)	または	**v**	:=	**v** * s
s	:=	VECTORTIMES(**v**,**v**)	または	s	:=	**v** * **v**
s	:=	VECTORTIMES(s , s)	または	s	:=	s * s

ベクトルの外積

v	:=	VECTORCROSS(**v**,**v**)	または	**v**	:=	**v** × **v**

ベクトルのべき乗

s := VECTOREXPT (**v**, s) または s := **v** ^ s
s := VECTOREXPT (s , s) または s := s ^ s

ベクトルの束

s := VMOD (s)
s := VMOD (**v**)

上に挙げた以外の引数の組合せはエラーになります。最初の二つのベクトルの乗算はベクトルのスカラー倍で、三番目の演算はスカラー同士の積で、四番目のはベクトルの内積 (ドット積) です。 前置演算子+, -, /はスカラーまたはベクトルの引数をとり、引数と同じ型の結果を返します。VMOD はスカラー値を返します。

これらが組合わされた式では、計算の順序を指定するために括弧で囲うことが出来ます。もし括弧を省略した場合は、計算の順序は

```
+ | - | dotgrad | * | >< | ^ | _
```

で決まります。ここでは後の方が優先度が高い演算子になっています。これらの演算子は REDUCE の優先順位では $<$ より高い順位になります。微分演算子 DOTGRAD は次の節で定義されています。そして要素を取り出す演算子_は 3 節で説明されています。

ベクトルの除算は次のように定義されています。いま、**a** と **b** がベクトルで c がスカラーとすると、

$$\mathbf{a}/\mathbf{b} = \frac{\mathbf{a}\cdot\mathbf{b}}{|\,\mathbf{b}\,|^2}$$
$$c/\mathbf{a} = \frac{c\mathbf{a}}{|\,\mathbf{a}\,|^2}$$

スカラー積とドット積は同じ記号で表されています。計算での優先順位をはっきりさせるために、例えば $(\mathbf{a}\cdot\mathbf{b})(\mathbf{c}\cdot\mathbf{d})$ のような式では括弧を使うことを進めます。

ベクトルのべき乗は束のべき乗として定義されています:
$\mathbf{a}^n \equiv \mathrm{VMOD}(a)^n = |\,\mathbf{a}\,|^n$

37.5 微分演算

微分演算子は div(発散), grad(勾配), curl(回転), delsq(ラプラシアン) と dotgrad が用意されています。 最後の演算子を除き、他はすべて一つのベクトルもしくはスカラーを取る前置演算子として定義されています。これらの演算の結果の型は表 37.1 に示されています。

これ以外の引数の型は許されません。もし使ったときにはエラーを起こします。微分演算子は通常の定義が与えられています [41]。これらの演算子で使われる座標系は VSTART (節 37.2 を参照のこと) で設定されます。ここでは名前 h1, h2 と h3 はスカラー因子の変数名として、また u1,u2 と u3 は座標系の名前として使われているので、他の目的には使えません。

$$
\begin{array}{rcl}
\mathrm{s} & := & \mathrm{div}\ (\mathbf{v}) \\
\mathbf{v} & := & \mathrm{grad}(\mathrm{s}) \\
\mathbf{v} & := & \mathrm{curl}(\mathbf{v}) \\
\mathbf{v} & := & \mathrm{delsq}(\mathbf{v}) \\
\mathrm{s} & := & \mathrm{delsq}(\mathrm{s}) \\
\mathbf{v} & := & \mathbf{v}\ \mathrm{dotgrad}\ \mathbf{v} \\
\mathrm{s} & := & \mathbf{v}\ \mathrm{dotgrad}\ \mathrm{s}
\end{array}
$$

表 37.1: 微分演算子と引数の型

REDUCE の DF 演算子のベクトルへの拡張として VDF が定義されています。これはベクトル (もしくはスカラー) をスカラーで微分した結果を返します。可能な引数と結果の型は VDF($\mathbf{v}$, s) → $\mathbf{v}$ と VDF(s, s) → s , で、例えば

$$\mathtt{vdf(B,x)} \equiv \frac{\partial \mathbf{B}}{\partial x}$$

となります。

REDUCE の DEPEND と NODEPEND 演算子は ベクトルの依存関係を定義しやすいように変更しています。例えば、

```
a := svec(a1,a2,a3)$;
depend a,x,y;
```

と入力すると、ベクトル a の三つの要素 a1,a2,a3 すべてが x と y の関数であると宣言されます。もちろん個々の要素に対して、

```
depend a3,z;
```

のように宣言することも可能です。

関数 VTAYLOR はベクトル値もしくはスカラー値の関数のテイラー級数展開を求めます。

```
vtaylor(vex,vx,vpt,vorder);
```

は式 VEX を変数 VX について点 VPT で次数 VORDER まで展開した級数を返します。 許される引数の組合せは表 37.2 に挙げています。

これ以外の組合せの場合はエラーを起こします。VORDER の要素は負でない整数で無ければなりません。さもないとエラーを起こします。もし、ベクトルの展開に対して、VORDER がスカラー値で与えられた場合は、各々の要素に対する展開がすべて同じ次数 VORDER で打ち切られます。

このパッケージで使っているテイラー展開は有理式に対してはロピタルの定理を使って係数の計算が行われており、従って $\sin(x)/(x)$ のような式の展開も正しく求めることが出来ます。この定理を使った LIMIT 演算子も用意されており、スカラー関数の極限値を求めることが出来ます。

```
ans := limit(ex,x,pt);
```

VEX	VX	VPT	VORDER
v	**v**	**v**	**v**
v	**v**	**v**	s
v	s	s	s
s	**v**	**v**	**v**
s	**v**	**v**	s
s	s	s	s

表 37.2: VTAYLOR の許される引数の組合せ

37.6 積分演算

ベクトルの不定積分および定積分、体積分、線積分を計算する関数が ORTHOVEC で定義されています。これらの関数の定義は以下の通りです。

$$\begin{aligned}
\mathrm{VINT}(\mathbf{v},x) &= \int \mathbf{v}(x)dx \\
\mathrm{DVINT}(\mathbf{v},x,a,b) &= \int_a^b \mathbf{v}(x)dx \\
\mathrm{VOLINT}(\mathbf{v}) &= \int \mathbf{v}h_1h_2h_3du_1du_2du_3 \\
\mathrm{DVOLINT}(\mathbf{v},\mathbf{l},\mathbf{u},n) &= \int_{\mathbf{l}}^{\mathbf{u}} \mathbf{v}h_1h_2h_3du_1du_2du_3 \\
\mathrm{LINEINT}(\mathbf{v},\omega,t) &= \int \mathbf{v}\cdot\mathbf{dr} \equiv \int v_ih_i\frac{\partial\omega_i}{\partial t}dt \\
\mathrm{DLINEINT}(\mathbf{v},\omega t,a,b) &= \int_a^b v_ih_i\frac{\partial\omega_i}{\partial t}dt
\end{aligned}$$

ベクトル積分と体積分で、 **v** はベクトルまたはスカラー w を、a, b, x と n はスカラーを表します。ベクトル **l** と **u** は積分の下限と上限を表します。整数の指数 n は積分の順序をどの様に取るかを指定します。この意味は次の通りです。

n	順序
1	$u_1\ u_2\ u_3$
2	$u_3\ u_1\ u_2$
3	$u_2\ u_3\ u_1$
4	$u_1\ u_3\ u_2$
5	$u_2\ u_1\ u_3$
それ以外	$u_3\ u_2\ u_1$

線積分での引数のベクトル ω は積分の経路に沿った線分 $\mathbf{u}(t)$ の指定を、パラメータ表示で表した関数で行います。

関数名		説明
VSTART		座標系の設定
SVEC		ベクトルの設定
VOUT		ベクトルの出力
VECTORCOMPONENT	_	ベクトルの要素の取り出し (1-3)
VECTORADD	+	ベクトルまたはスカラーの和
VECTORPLUS	+	ベクトルまたはスカラーの単項の和
VECTORMINUS	-	負のベクトルまたはスカラー
VECTORDIFFERENCE	-	ベクトルまたはスカラーの差
VECTORQUOTIENT	/	ベクトルのスカラーによる商
VECTORRECIP	/	ベクトルまたはスカラーの逆数 (逆数)
VECTORTIMES	*	ベクトルまたはスカラーの積 ベクトル/スカラー
VECTORCROSS	><	ベクトルの外積
VECTOREXPT	^	ベクトル束またはスカラーのべき
VMOD		ベクトルまたはスカラーの長さ
DIV		ベクトルの発散
GRAD		スカラーの勾配
CURL		ベクトルの curl
DELSQ		スカラーもしくはベクトルのラプラシアン
DOTGRAD		(ベクトル).grad(スカラーまたはベクトル)
VTAYLOR		スカラーまたはベクトルのテイラー展開
VPTAYLOR		スカラーのテイラー展開
TAYLOR		スカラーのテイラー展開
LIMIT		L'Hôpital による商の極限
VINT		ベクトルの積分
DVINT		ベクトルの定積分
VOLINT		体積分
DVOLINT		体定積分
LINEINT		線積分
DLINEINT		線定積分
MAPRIN		MAPRIN のベクトル版
DEPEND		DEPEND のベクトル版
NODEPEND		NODEPEND のベクトル版

表 37.3: ORTHOVEC での関数

第38章　PM: REDUCEのパターンマッチャー

Kevin McIsaac
The University of Western Australia
Australia
e–mail: *kevin@wri.com*

PM は SMP や Mathematica のようなシステムに存在するものと同様な一般的なパターンマッチャーです。

テンプレートは、5、a または a +1 のようなリテラル要素や?や??b といった特別なパターン変数で構成された任意の式です。'?' で始まる識別子は総称変数と呼ばれ、任意の式とマッチします。'??' で始まる識別子はマルチ総称変数で、任意の式やヌルや空列を含む任意の列とマッチします。

列とは、'[a1, a2,...]' のような形式の式です。f([a,1]) → f(a,1) や f(a,[1,2],b) → f(a,1,2,b) のように、関数の引数として現れた場合、角括弧は取り除かれます。

テンプレートはテンプレートが式と文字通りに等しい場合、あるいはテンプレートに含まれる総称変数あるいはマルチ総称変数に置換えを行なって式と同一にできる場合、マッチするといいます。これらの置換は総称変数の束縛と呼ばれます。置換は exp1 -> exp2 の形式 (これは exp1 が exp2 で置換されることを表す) もしくは exp1 --> exp2 の形式 (これは exp1 への置換が行なわれるまで exp2 が簡約されないという点を除けば同じ) をとります。もし式が、結合性、交換性もしくは同一性といった性質を持っている場合、マッチするか否かを判定するためにそれらの性質が利用されます。式にテンプレートを一致させる試みが失敗した場合、その時点での総称変数の束縛を解消し、別の選択が行なえるところまでバックトラックを行ないます。

マッチャーはまた意味的な一致を支援します。簡単にいうと、異なる構造を持っているため、ある部分テンプレートが対応する部分式にマッチしない場合、二つを等しいものとして、マッチャーは全ての総称変数が束縛されるまで残りの式のマッチングを行ないます。その後、同じになるかがチェックされます。これはスイッチ semantic で制御されます。通常はこれはオンです。

38.1　マッチ関数

M(exp,template)

テンプレートは式とマッチさせられます。テンプレートが式と文字通りに等しい場合、T が返されます。テンプレートは、それが含んでいる総称変数をすべて束縛されている値と置換することにより等しくなる場合、束縛の集合が返されます。そうでなければ、NIL が返されます。

```
OPERATOR F;
```

```
M(F(A),F(A));

T

M(F(A,B),F(A,?A));

{?A->B}

M(F(A,B),F(??A));

{??A->[A,B]}

m(a+b+c,c+?a+?b);

{?a->a,?b->b}

m(a+b+c,b+?a);

{?a->a + c}
```

この例は、+の結合則および交換則を使用して、意味的にマッチした結果を示します。

38.2 制約つきのマッチ

テンプレートはマッチする条件を指定する条件付きのオペレーター'_='、を使用使用することによって制約を加えることが可能です。テンプレート中に制約条件が含まれている場合、論理式に現れる全ての総称変数が束縛されるまで、条件式はとっておかれます。全ての総称関数が束縛されると、条件式は簡約され、結果が T で無ければ条件は失敗しパターンマッチは失敗してバックトラックが起こります。テンプレートが完全に解析された場合、全ての条件式が評価され T と比較されます。

```
load_package pm;

operator f;

if (m(f(a,b),f(?a,?b_=(?a=?b)))) then write "yes" else write"no";

no

m(f(a,a),f(?a,?b_=(?a=?b)));
```

```
{?B->A,?A->A}
```

38.3 代入による置換

オペレーター S は、式の内容を置換式で置き換えます。

```
S(exp,temp1->sub1,temp2->sub2,..., rept, depth);
```

は、広がり優先探索で最大 rept 回そして depth の深さまで、置換えを実行する。rept と depth は、それぞれ 1 と無限の場合、省略することができます。

```
SI(exp,temp1->sub1,temp2->sub2,..., depth)
```

は、式が変化しなくなるまで、無限に多くの代入を実行します。

```
SD(exp,temp1->sub1,temp2->sub2,..., rept, depth)
```

は、S と同じですが、深さ優先で探索します。

```
s(f(a,b),f(a,?b)->?b^2);

 2
b

s(a+b,a+b->a*b);

a*b


operator nfac;

s(nfac(3),{nfac(0)->1,nfac(?x)->? x*nfac(?x-1)});

3*nfac(2)

s(nfac(3),{nfac(0)->1,nfac(?x)->? x*nfac(?x-1)},2);

6*nfac(1)

si(nfac(4),{nfac(0)->1,nfac(?x)->? x*nfac(?x-1)});

24

s(a+b+f(a+b),a+b->a*b,inf,0);

f(a + b) + a*b
```

38.4 パターンによるプログラミング

パターンマッチャーをプログラミング言語として使用する機能があります。演算子:-は簡約において、マッチする全てのテンプレートを右辺式で置き換えられることを宣言します。演算子::-は左側の式が簡約されないという点を除いて:-と同じです。

```
operator fac, gamma;
fac(?x_=Natp(?x)) ::- ?x*fac(?x-1);
HOLD(FAC(?X-1)*?X)

fac(0)  :- 1;
1

fac(?x) :- Gamma(?x+1);
GAMMA(?X + 1)

fac(3);
6

fac(3/2);
GAMMA(5/2)
```

第39章　POLYDIV: 拡張された多項式の割算

Francis J. Wright
School of Mathematical Sciences
Queen Mary and Westfield College
University of London
Mile End Road, London E1 4NS, UK.
Email: *F.J.Wright@QMW.ac.uk*

このパッケージは、REDUCE の標準の多項式除算機能へのよりよいアクセスと多項式擬除算を提供します。また、除算のために使用される主要な変数に関する局所的な制御を提供します。

39.1　はじめに

polydiv パッケージは、多項式のユークリッド除算のための代数モードでの機能を拡張します。数の係数領域は常にグローバルに指定されたものが使用されます。より多くの例はテストおよびデモンストレーション中で提供されます。

39.2　多項式除算

polydiv パッケージは、ユークリッド商と除余を計算する、内挿演算 div および mod を提供します。

```
(x^2 + y^2) div (x - y);

        x + y

(x^2 + y^2) mod (x - y);

           2
        2*y
```

(それらは前置演算子としても使用することができます。)

これは、商と除余を同時に計算する、ユークリッドの除算 divide も提供します。

```
divide(x^2 + y^2, x - y);
```

```
                2
        {x + y,2*y }
```

(これは内挿演算子として使用することができます。)

ユークリッドの除算演算子 (接頭辞の形の中で使用され、標準の remainder 演算子を含んでいた時) はすべて、オプションとして 3 番めの引き数 (それは割算中に使用される主要な変数を指定する) を受け付けます。規定値は現在のグローバルな順序での主変数です。主変数の指定は、含まる他の変数の順序を変更しません。またグローバル環境を変更しません。例えば

```
div(x^2 + y^2, x - y, y);

         - (x + y)

remainder(x^2 + y^2, x - y, y);

           2
        2*x

divide(x^2 + y^2, x - y, y);

                       2
        { - (x + y),2*x }
```

x を主変数として指定することは、以前に示したようにデフォルトと同じ振舞をします。つまり

```
divide(x^2 + y^2, x - y, x);

                2
        {x + y,2*y }

remainder(x^2 + y^2, x - y, x);

           2
        2*y
```

39.3 多項式擬除算

上に議論された多項式除算は、体上の一変数多項式の場合に最も有用です。そうでなければ、除算は通常失敗して、商が 0 で除余が被除数と一致するつまらない結果を与えます。(係数環が体である場合だけが、一変数多項式環がユークリッド領域です。) 例えば、整数上で:

```
divide(x^2 + y^2, 2(x - y));
```

```
            2    2
        {0,x  + y }
```

単位元を持つ可換環上 (例えば整数や多項式環) で次数 m の多項式 $u(x)$ を次数 $n \leq m$ の多項式 $v(x)$ で除算することは、もし多項式 $u(x)$ を $\mathrm{lc}(v,x)^{m-n+1}$(ここで lc は主係数を表す) 倍しておけば常に実行可能です。これは**擬**除算と呼ばれます。polydiv パッケージは多項式の擬除算演算子 pseudo_divide, pseudo_quotient (もしくは pseudo_div) それに pseudo_remainder を提供しています。多変数の多項式の擬除算を計算する場合、除数の主係数が主変数に関して計算されるので、どの変数を主変数とするのかは重要です。したがって、もしデフォールトで計算した場合でなんらかの曖昧さがありうる場合、つまり多項式が多変数であるかまたは一つ以上のカーネルを持つ場合、擬除算の演算子はどの変数を主変数に選んだかを示す警告を発します。(通常、警告は off msg; とスイッチをオフにすることにより出さないようにすることができます。) 例えば

```
pseudo_divide(x^2 + y^2, x - y);

        *** Main division variable selected is x

                  2
        {x + y,2*y }

pseudo_divide(x^2 + y^2, x - y, x);

                  2
        {x + y,2*y }

pseudo_divide(x^2 + y^2, x - y, y);

                       2
        { - (x + y),2*x }
```

除数の主係数が 1 なら、除算と擬除算は同じになります。それ以外の場合はそうではありません。

```
divide(x^2 + y^2, 2(x - y));

            2    2
        {0,x  + y }

pseudo_divide(x^2 + y^2, 2(x - y));

        *** Main division variable selected is x

                     2
        {2*(x + y),8*y }
```

擬除算は、本質的には係数環の分数の体上での除算と同じ結果を与えます。(商と除余に共通の因子を除けば)

```
on rational;

divide(x^2 + y^2, 2(x - y));

          1             2
        {---*(x + y),2*y }
          2

pseudo_divide(x^2 + y^2, 2(x - y));

        *** Main division variable selected is x

                     2
        {2*(x + y),8*y }
```

多項式除算および擬除算は REDUCE の多項式についてのみ適用可能です。つまり分母が 1 である有理式。

```
off rational;

pseudo_divide((x^2 + y^2)/2, x - y);

                2    2
               x  + y
        ***** --------- invalid as polynomial
                  2
```

擬除算は (D. E. Knuth 1981, *Seminumerical Algorithms*, Algorithm R, page 407) のアルゴリズムを用いてインプリメントされています。これは実際にはいかなる割算もまったく行ないません。これは素朴なアルゴリズムより効率的です。またそれは係数領域上でのみ計算されるという利点があります。このような例としては、ファイル `polydiv.tst` 中にあるように、例えば整数に $\sqrt{2}$ を付け加えた代数的数を含んだ係数領域で計算を行なうような場合があります。このインプリメントは、除余のみが必要な場合にも商を計算してしまう点を除けば、十分効率的です。

第40章　RANDPOLY: ランダムな多項式の生成

Francis J. Wright
School of Mathematical Sciences, Queen Mary and Westfield College
University of London
Mile End Road
London E1 4NS, England
e–mail: *F.J.Wright@QMW.ac.uk*

RANDPOLY は少なくとも一つの引数を必要とします。これは、多項式の変数に対応して降り、一つの式かもしくは複数の式をリストにしたものです。RANDPOLY 内部での変数を使って生成した多項式中の変数名を入力された式で置き換えます。生成される多項式の次数は内部変数での次数とは異なっているかも知れません。

規定値では、生成される多項式の次数は 5 次で六つの項からなります。従って、もし一変数であれば密な多項式になります。また多変数多項式であれば、疎な多項式になります。

40.1　オプションの引数

オプションの引数を指定することができます。これは最初の引数の後に任意の順序で指定することができます。全ての引数は代数式として評価され、現在のスイッチの設定等の影響を受けます。引数は与えられた順に処理されていきます。従って、もし互いに影響し合うようなオプションを指定した場合、最後に指定したオプションのみが意味を持つことになります。オプションの引数は、キーワードであるかそれともキーワードが左辺にある式のいずれかです。

一般的に、多項式は dense キーワードをオプションの引数として指定しない限り、規定値では疎です。(キーワード sparse を指定することもできます。しかしこれは規定値です) 規定の次数を変更することは、

```
degree = <自然数>.
```

を指定することで可能です。多変数多項式の場合、これは全次数を意味します。より複雑な単項次数の制限は、以下に説明する係数関数を使って記述することができます。さらに randpoly は内部的には REDUCE の “(漸近)asymptotic” コマンド let, weight 等を使って生成される多項式の次数の制限を行なっています。このことから、生成される多項式に対して別の制御を行なうことも可能です。

疎な多項式の場合、生成される項の最大値は、オプション引数

```
terms = <自然数>.
```

で変更できます。実際に生成される項の数は、terms の値と指定された次数の密多項式の項数、変数の数等の内の最小の値になります。

40.2 RANDPOLY の進んだ使用法

規定のオーダー (最小の次数) はオプション引数

```
ord = <自然数>.
```

で変更できます。規定値はゼロです。

randpoly の入力は式であっても構いません。この場合オーダーの規定値は 0 でなく 1 になります。入力式は両辺の差が計算され、これが生成された多項式に代入されます。このことから、ある特定のゼロ点を持つ多項式が簡単に生成できます。例えば、

```
randpoly(x = a);
```

では、$x = a$ でゼロになる多項式が生成されます。

randpoly にはこれまで説明してきた以外に、式の形式で指定できる二つのオプションがあります。coeffs と expons です。右辺式は、引数を取らない関数出なければなりません。これは通常の代数モードでの関数です。coeffs に対しては、任意の代数式を返す関数で、expons に対しては、整数値を返す関数でなければなりません。これらの関数が返す値は、それぞれ生成される多項式の係数や巾になるため、通常ランダムな値であるべきです。

有用な関数として rand が用意されています。これは一つの引数を取る関数で、返す値は引数を取らないランダムな値を返す匿名の関数名です。rand の引数は通常 a .. b の形の整数の範囲です。ここで、a, b は $a < b$ を満たすような整数値です。例えば、expons の引数は次のような形式になるでしょう。

```
expons = rand(0 .. n)
```

ここで、n はそれぞれの変数に対する最大次数になります。coeffs に対しては、係数の範囲は正負でバランスしていることが多いので、

```
coeffs = rand(-n .. n)
```

のようになるでしょう。これは、$[-n, n]$ の範囲のランダムな整数値が係数となります。

より詳しい説明と例題が別にあります。

40.3 例

```
randpoly(x);
```

```
       5       4       3       2
 - 54*x  - 92*x  - 30*x  + 73*x  - 69*x - 67
```

```
randpoly({x, y}, terms = 20);
```

```
    5       4         4       3  2       3         3
31*x  - 17*x *y - 48*x  - 15*x *y  + 80*x *y + 92*x

       2  3      2         2         4         3         2
 + 86*x *y  + 2*x *y - 44*x  + 83*x*y  + 85*x*y  + 55*x*y

                       5       4      3       2
 - 27*x*y + 33*x - 98*y  + 51*y  - 2*y  + 70*y  - 60*y - 10
```

```
randpoly({x, sin(x), cos(x)});
```

```
                   4            3              3
sin(x)*( - 4*cos(x)  - 85*cos(x) *x + 50*sin(x)

                    2
         - 20*sin(x) *x + 76*sin(x)*x + 96*sin(x))
```

第41章　RATAPRX:有理近似

Lisa Temme
Wolfram Koepf
e-mail: *koepf@zib.de*

41.1　循環小数

41.1.1　関数の説明

整数を整数で割算を行なうと、しばしば循環小数が現れます。このパッケージにで定義さてた関数 rational2periodic は、このような有理数を認識し、循環小数で表します。また、この逆に循環表現から有理数に変換する関数 periodic2rational が定義されています。

有理数の循環小数表現

構文: `rational2periodic(n);`

入力: `n`　は有理数

結果: `periodic({a,b} , {c1,...,cn})`

ここで、a/b は循環しない部分
そして c1,...,cn は循環部分の数字.

例: $59/70$ は $0.8\overline{428571}$ と表される

```
1:  rational2periodic(59/70);
```

```
periodic({8,10},{4,2,8,5,7,1})
```

循環表現から有理数への変換

構文:
```
periodic2rational(periodic({a,b},{c1,...,cn}))
periodic2rational({a,b},{c1,...,cn})
```

入力:
- `a`　は整数
- `b`　は $1, -1$ もしくは 10 の整数倍

c1,...,cn は正の数字のリスト

結果: 有理数.

例: $0.8\overline{428571}$ は 59/70 となる

```
2:  periodic2rational(periodic({8,10},{4,2,8,5,7,1}));

 59
----
 70

3:  periodic2rational({8,10},{4,2,8,5,7,1});

 59
----
 70
```

もし a がゼロであれば、b は小数以下何桁目から循環が始まるかを示すことに注意して下さい。また、結果が負の場合、a の符号で表されます。a がゼロの場合には、b の符号で表されます。

41.1.2 エラーメッセージ

```
***** operator to be used in off rounded mode
```

***** この関数は rouded をオフにして使用して下さい。

この関数は、off rounded モードでのみ周期性の判定ができます。これは、逆関数に対しても同じです。

41.1.3 例

```
4: rational2periodic(1/3);

periodic({0,1},{3})

5: periodic2rational(ws);

 1
---
 3

6: periodic2rational({0,1},{3});
```

```
 1
---
 3

7: rational2periodic(-1/6);

periodic({-1,10},{6})

8: periodic2rational(ws);

  - 1
------
  6

9: rational2periodic(6/17);

periodic({0,1},{3,5,2,9,4,1,1,7,6,4,7,0,5,8,8,2})

10: periodic2rational(ws);

 6
----
 17

11: rational2periodic(352673/3124);

periodic({11289,100},{1,4,8,5,2,7,5,2,8,8,0,9,2,1,8,9,5,0,0,6,
                      4,0,2,0,4,8,6,5,5,5,6,9,7,8,2,3,3,0,3,4,
                      5,7,1,0,6,2,7,4,0,0,7,6,8,2,4,5,8,3,8,6,
                      6,8,3,7,3,8,7,9,6,4})

12: periodic2rational(ws);

 352673
--------
  3124
```

41.2 連分数

41.2.1 関数の説明

連分数 ([7] §4.2 を参照) は次の一般形を持ちます。

$$b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \ldots}}} .$$

これをもっとコンパクトな形であらわすと、

$$b_0 + \frac{a_1|}{|b_1} + \frac{a_2|}{|b_2} + \frac{a_3|}{|b_3} + \ldots .$$

これは、REDUCE では次のように表現されます。

$$\texttt{contfrac}(\text{有理近似}, \{\texttt{b0}, \{\texttt{a1}, \texttt{b1}\}, \{\texttt{a2}, \texttt{b2}\}, ...\})$$

構文:
```
cfrac(number);
cfrac(number,length);
cfrac(f, var);
cfrac(f, var, length);
```

入力:

number	任意の実数
f	関数
var	関数の変数

41.2.2 オプションの引数: length

length 引数はオプションです。有理的でない関数の入力に対して、length 引数は返される数値のペア $\{a_i, b_i\}$ の数を指定します。通常はこの値は 5 です。有理的な関数の入力に対しては、length 引数は答を打ち切ります。精度を上げても追加の数値のペアは計算されません。通常の値は、有理数に対しては完全な連分数を返します。数値入力に対しては、通常の値は計算の精度に依存します。そして、length 引数は通常の値の場合に返される数値のペアの数よりも小さな値を設定した場合のみ意味を持ちます。

41.2.3 例

```
13: cfrac(23.696);
```

```
          2962
contfrac(------,{23,{1,1},{1,2},{1,3},{1,2},{1,5}})
          125


14: cfrac(23.696,3);

          237
contfrac(-----,{23,{1,1},{1,2},{1,3}})
          10

15: cfrac pi;


          1146408
contfrac(---------,
          364913

         {3,{1,7},{1,15},{1,1},{1,292},{1,1},{1,1},{1,1},{1,2},{1,1}})

16: cfrac(pi,3);

          355
contfrac(-----,{3,{1,7},{1,15},{1,1}})
          113

17: cfrac(pi*e*sqrt(2),4);

          10978
contfrac(-------,{12,{1,12},{1,1},{1,68},{1,1}})
           909

18: cfrac((x+2/3)^2/(6*x-5),x,1);

             2
          9*x  + 12*x + 4    6*x + 13      24*x - 20
contfrac(-----------------,{----------,{1,-----------}})
             54*x - 45          36             9

19: cfrac((x+2/3)^2/(6*x-5),x,10);
```

```
             2
          9*x  + 12*x + 4    6*x + 13      24*x - 20
contfrac(-----------------,{----------,{1,----------}})
             54*x - 45          36             9


20: cfrac(e^x,x);



           3      2
          x  + 9*x  + 36*x + 60
contfrac(----------------------,{1,{x,1},{ - x,2},{x,3},{ - x,2},{x,5}})
               2
            3*x  - 24*x + 60

21: cfrac(x^2/(x-1)*e^x,x);

           6      4    2
          x  + 3*x  + x
contfrac(----------------,{0,
             4    2
          3*x  - x  - 1

                 2            2       2       2       2
            { - x ,1}, { - 2*x ,1}, {x ,1}, {x ,1}, {x ,1}})

22: cfrac(x^2/(x-1)*e^x,x,2);

             2
             x              2           2
contfrac(----------,{0,{ - x ,1},{ - 2*x ,1}})
             2
          2*x  - 1
```

41.3 パデ近似

41.3.1 関数の説明

パデ近似 (Padé 近似) は関数を二つの多項式の比で表現します。多項式の係数は、関数のテイラー級数展開から決定されます。([7] を参照)。与えられた巾級数

$$f(x) = c_0 + c_1(x-h) + c_2(x-h)^2 \ldots$$

および分子の次数,n, と分母の次数,d, から pade 関数はパデ近似

$$\frac{a_0 + a_1 x + \cdots + a_n x^n}{b_0 + b_1 x + \cdots + b_d x^d} .$$

の係数 a_i, b_i を決定します。

構文: `pade(f, x, h, n, d);`

入力:
- `f` 近似しようとする関数
- `x` 関数の引数
- `h` 近似する点
- `n` 分母の次数
- `d` 分子の次数

結果: パデ近似、つまり有理関数.

41.3.2 エラーメッセージ

```
***** not yet implemented
```

(***** まだインプリメントされていません。)

関数 f のテイラー級数展開は REDUCE の TAYLOR パッケージにインプリメントされていません。

```
***** no Pade Approximation exists
```

(***** パデ近似は存在しません。)

この関数に対するパデ近似は存在しません。

```
***** Pade Approximation of this order does not exist
```

(***** この次数のパデ近似は存在しません。)

この次数 (指定された分子と分母の次数) のパデ近似は存在しません。しかしながら、違った値の次数に対しては存在するかもしれません。

41.3.3 例

```
23: pade(sin(x),x,0,3,3);

          2
 x*( - 7*x  + 60)
------------------
       2
   3*(x  + 20)

24: pade(tanh(x),x,0,5,5);

     4        2
 x*(x  + 105*x  + 945)
-----------------------
      4       2
 15*(x  + 28*x  + 63)

25: pade(atan(x),x,0,5,5);

        4        2
 x*(64*x  + 735*x  + 945)
--------------------------
         4       2
 15*(15*x  + 70*x  + 63)

26: pade(exp(1/x),x,0,5,5);

***** no Pade Approximation exists

27: pade(factorial(x),x,1,3,3);

***** not yet implemented

28: pade(asech(x),x,0,3,3);

            2                          2                 2
- 3*log(x)*x  + 8*log(x) + 3*log(2)*x  - 8*log(2) + 2*x
-------------------------------------------------------
                          2
                     3*x  - 8

29: taylor(ws-asech(x),x,0,10);
```

```
              11
log(x)*(0 + O(x  ))

     13    6     43    8    1611    10      11
 + (-----*x  + ------*x  + -------*x   + O(x  ))
     768        2048        81920

30: pade(sin(x)/x^2,x,0,10,0);

***** Pade Approximation of this order does not exist

31:  pade(sin(x)/x^2,x,0,10,2);

     10        8         6           4            2
( - x   + 110*x  - 7920*x  + 332640*x  - 6652800*x

  + 39916800)/(39916800*x)

32: pade(exp(x),x,0,10,10);



  10        9         8           7            6
(x   + 110*x  + 5940*x  + 205920*x  + 5045040*x

              5               4               3
  + 90810720*x  + 1210809600*x  + 11762150400*x

                2
  + 79394515200*x  + 335221286400*x + 670442572800)/

      10        9         8           7            6
    (x   - 110*x  + 5940*x  - 205920*x  + 5045040*x

                      5               4
         - 90810720*x  + 1210809600*x

                         3               2
         - 11762150400*x + 79394515200*x

         - 335221286400*x + 670442572800)

33: pade(sin(sqrt(x)),x,0,3,3);
```

```
(sqrt(x)*
            3            2
  (56447*x  - 4851504*x  + 132113520*x - 885487680))/

              3         2
    (7*(179*x  - 7200*x  - 2209680*x - 126498240))
```

第42章　REACTEQN: 化学反応方程式

Herbert Melenk

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin–Dahlem, Germany

e–mail: *melenk@zib.de*

REDUCE のパッケージ REACTEQN は化学反応を質量反応の法則に対応した、通常の常微分方程式系に変換します。

関数 `reac2ode` が提供されています。

```
reac2ode {   <reaction> [,<rate> [,<rate>]]
           [,<reaction> [,<rate> [,<rate>]]]
                  ....
             };
```

`rate` は REDUCE の任意の式で、二つの `rate` は順反応および逆反応の対してのみ適応可能です。`reaction` は、順反応に対しては演算子 `->` を、可逆反応に対しては `<>` を使って、列変数の線形和としてコードされます。

結果は、右辺が多項式である常微分方程式で与えられます。副作用として次の変数に値が設定されます。

`rates` 系の反応率のリスト

`species` 系の種のリスト

`inputmat` 入力係数の行列

`outputmat` 出力係数の行列

行列において、行の数は入力反応の数に対応しており、列の数は種の数に対応している。

反応率が数値であれば、多くの場合 REDUCE の浮動小数点数計算のモードを選択するのが適当である。

`Inputmat` および `outputmat` は反応系の線形代数型を調べるのに利用できます。古典的な反応行列はこれらの行列の差です。しかしながら、両辺に現れる種は反応行列に影響を与えないため二つの行列はそれらの差行列よりもより多くの情報を持っている。

第43章　RESET:初期状態へのリセット

J. P. Fitch

School of Mathematical Sciences, University of Bath
BATH BA2 7AY, England
e–mail: *jpff@maths.bath.ac.uk*

このパッケージは、コマンド RESETREDUCE を定義しています。このコマンドは、入力の履歴を参照して、任意の規則、配列およびその他ユーザによって割り当てられたすべての値を取り除きます。さらに、様々なスイッチを初期の値に戻します。これは完全ではないが、徐々に記憶容量の損失を引き起こすようなほとんどの場合に動作します。

第44章　RESIDUE: 留数

Wolfram Koepf
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin–Dahlem, Germany
e–mail: *Koepf@zib.de*

このパッケージは留数の演算を行ないます。関数 $f(z)$ の点 $a \in \mathbb{C}$ における留数 $\operatorname{Res}_{z=a} f(z)$ 次の式で定義されます。

$$\operatorname{Res}_{z=a} f(z) = \frac{1}{2\pi i} \oint f(z)\, dz \,,$$

ここで積分は $z = a$ の閉曲線上に回転数 1 で行います。

このパッケージには次の二つの演算子が用意されています。

- `residue(f,z,a)` は、もし f が $z = a$ でメロモルフィックである時、点 $z = a$ での関数 f の留数を計算します。f の真性特異点での留数は計算できません。
- `poleorder(f,z,a)` は、もし f が $z = a$ でメロモルフィックである時、f の点 $z = a$ での極次数を計算します。

これらの関数は TAYLOR パッケージ (55 章) を使っています。

```
load_package residue;

residue(x/(x^2-2),x,sqrt(2));

 1
---
 2

poleorder(x/(x^2-2),x,sqrt(2));

1

residue(sin(x)/(x^2-2),x,sqrt(2));
```

```
 sqrt(2)*sin(sqrt(2))
----------------------
          4

poleorder(sin(x)/(x^2-2),x,sqrt(2));

1

residue((x^n-y^n)/(x-y)^2,x,y);

  n
 y *n
------
  y

poleorder((x^n-y^n)/(x-y)^2,x,y);

1
```

第45章　RLFI: LaTeX形式での出力

Richard Liska and Ladislav Drska

Computational Physics Group
Faculty of Nuclear Sciences and Physical Engineering
Czech Technical University in Prague
Brehova 7, 115 19 Prague 1, Czech Republic

E-mail: *liska@siduri.fjfi.cvut.cz*

May 23, 1995

REDUCE によって計算された結果の数式を LaTeX 形式で出力します。パッケージをロードして、スイッチ `latex` をオンにすると、以後、計算された結果は LaTeX 形式で出力されます。スイッチ `VERBATIM` をオンにすれば、入力した式を同時に LaTeX の `verbatim` 環境を使って出力に出します[1]

スイッチ `lasimp` は数式の評価を制御します。`lasimp` がオフの場合には、入力した式は評価されず、そのままの式が LaTeX 形式に変換されます。オンであれば、入力した式は通常の REDUCE の評価を行った結果が出力されます。このスイッチは通常オンです。

REDUCE の数式に現れる識別子に対して、`DEFID` 文でその出力名やフォント、アクセント記号を指定することができます。例えば、変数 a を出力ではギリシャ文字の α とし、フォントをボールドで出力したい場合には、

```
DEFID a,NAME=alpha,FONT=bold;
```

と指定します。`DEFID` の構文は次の通りです。

```
DEFID <identifier>,<d-equations>;
<d-equations> ::= <d-equation> | <d-equation>,<d-equations>
<d-equation> ::= <d-print symbol> | <d-font>|<d-accent>
<d-print symbol> ::= NAME = <print symbol>
<d-font> ::= FONT = <font>
<d-accent> ::= ACCENT = <accent>
<print symbol> ::= <character> | <special symbol>
<special symbol> ::= alpha|beta|gamma|delta|epsilon|
   varepsilon|zeta|eta|theta|vartheta|iota|kappa|lambda|
   mu|nu|xi|pi|varpi|rho|varrho|sigma|varsigma|tau|
```

[1] REDUCE3.4.1 版では少しバグがあり、`LATEX` をオンにした直後にこのスイッチをオンまたはオフに設定しないと、正しい出力が得られません。

```
    upsilon|phi|varphi|chi|psi|omega|Gamma|Delta|Theta|
    Lambda|Xi|Pi|Sigma|Upsilon|Phi|Psi|Omega|infty|hbar
<font> ::= bold|roman
<accent> ::=hat|check|breve|acute|grave|tilde|bar|vec|
    dot|ddot
```

添え字付きの変数は REDUCE のオペレータで表します。オペレータの引数が添え字に対応します。添え字をどうするか (上付き/下付き、右/左、またはオペレータの引数として出力する) を指定するには DEFINDEX 文を使います。

構文は次の通りです。

```
DEFINDEX <d-operators>;
<d-operators> ::= <d-operator> | <d-operator>,<d-operators>
<d-operator> ::= <prefix operator>(<descriptions>)
<prefix operator> ::= <identifier>
<descriptions> ::= <description> | <description>,
    <descriptions>
<description> ::= ARG | UP | DOWN | LEFTUP | LEFTDOWN
```

例えば、f^k_j と出力するには、

```
DEFINDEX f(UP,DOWN);

f(k,j);
```

とすればよい。

MATHSTYLE 文は式の出力形式を制御します。これで、math,displaymath,equation のいずれかの出力形式を選択します。

```
MATHSTYLE <m-style>;
<m-style> ::= math | displaymath | equation
```

変数 LALINE!*は出力の一行の長さを指定します。

分数は新しい REDUCE の中置演算子 \ を使って表します。例えば、$\frac{1}{3}$ は次のように入力します。

```
1 \ 3;
```

微分や積分等の記号は REDUCE の前置演算子の形で入力します。

`int(f(x),x)`	$\int f(x)dx$
`dint(0,1,f(x),x);`	$\int_0^1 f(x)dx$
`df(f(x),x);`	$\frac{\mathrm{d}f(x)}{dx}$
`pdf(f(x,y),x,2,y);`	$\frac{d^3f(x)}{dx^2dy}$
`sum(i=0,n,i**2);`	$\sum_{i=0}^{n} i^2$
`product(i=0,n,f(i));`	$\prod_{i=0}^{n} f(i)$
`sqrt(x)`	$\sqrt{x}$

現在の版ではまだ、次のような問題が未解決のままです。

- 一行に入りきらない数式を分割すること。
- 分数を表すのに/を使うかどうかを自動判定させること。
- 二文字以上の識別子を一文字のシンボルの積と区別すること。
- 行列の出力ができない。

注意

RLFI パッケージは `OFF RAISE` 文で小文字が使えるシステムでないと使えません。このパッケージでは大文字と小文字を区別しています。

第46章　ROOTS:高次方程式の解

Stanley L. Kameny

E-mail: *valley!stan@rand.org*

46.1　はじめに

このパッケージはこれ自身で独立して使用することもでき、また SOLVE パッケージと統合した形で、SOLVE 演算子から呼ばれる形でも使うことが出来ます。このパッケージでは一変数多項式の実数解もしくは複素数解を指定した精度で求めることが出来ます。

46.2　根を求める方法

このパッケージでは実際の計算をできるだけ簡単なものとするために、多項式に対してまずその因子を求めます。これには無平方分解と複素数係数の多項式を実数係数の多項式と複素数係数の多項式の積の形へ分解することが含まれています。このような分解が可能であれば、結果として得られた各々の多項式を独立に解いていきます。ただし、各々の多項式の解のうち非常に接近した解が存在する場合には、それらを分離するのに十分なだけの精度を上げた計算が行われます。もう一つの方法は、GCD 法によって多項式の次数を共通因子だけ下げ、解は簡約された多項式の解として求められます。この場合も、隣接根が存在する場合には、それを分離するために必要な処理が行われます。

46.3　トップレベルでの関数

トップレベルでの関数は演算子として代数モードで呼び出すことが出来るし、また直接記号モードで呼び出すこともできます。出力は代数モードで正しく印刷される形で行われます。

46.3.1　実数根のみを求める関数

実数根のみを求める関数が三つ用意されています。これらは 1 個、2 個もしくは 3 個の引数を取ることが出来ます。

最初の引数は多項式 p で、これは複素数多項式であっても構いませんし、また重複根や零を根とする多項式であっても構いません。第二引数および第三引数が指定されていない場合は、全て

の実数根が出力されます。もしなんらかの引数を指定している場合には、制限された範囲での根が求められます。

- もし引数が (p, arg_2) であれば、arg_2 は POSITIVE か NEGATIVE でなければなりません。もし arg_2 が NEGATIVE であれば p の負の根のみが出力されます。また arg_2 が POSITIVE であれば正の根のみが出力されます。零解は出力されません。

- もし引数が (p, arg_2, arg_3) の場合、 arg_2 および arg_3 は r(実数) または EXCLUDE r、または POSITIVE, NEGATIVE, INFINITY, -INFINITY のいずれかでなければなりません。EXCLUDE r は値 r は範囲から除くことを意味します。arg_2 と arg_3 の順序はいずれでも構いません。もし二つとも数値の場合は $arg_2 \leq arg_3$ が仮定されます。

  {-INFINITY,INFINITY} は {} と同じで全部の解を表す;
  $\{arg_2$,NEGATIVE$\}$ は $-\infty < r < arg_2$;
  $\{arg_2$,POSITIVE$\}$ は $arg_2 < r < \infty$;

  以下で **arg** を EXCLUDE **arg** に変更すると、$\leq$ が $<$ に変わります。

  $\{arg_2$,-INFINITY$\}$ は $-\infty < r \leq arg_2$;
  $\{arg_2$,INFINITY$\}$ は $arg_2 \leq r < \infty$;
  $\{arg_2, arg_3\}$ は $arg_2 \leq r \leq arg_3$;

- もし 0 が区間の内部に含まれている場合は、零解も出力されます。

REALROOTS この関数は多項式 p の実零点を求めます。REALROOT パッケージを使い、スツルム列を使って実根を分離します。それから必要な精度まで根の近似を上げていきます。計算の精度は指定された領域内の全ての実根が分離できるのに必要なだけ取られます。
(注意:多重根を扱うには MULTIROOT を使ってください。)

ISOLATER この関数は各領域に一つの実根のみを含む有理数の区間を返します。実際の根の値は計算しません。

RLROOTNO この関数は p の指定された領域内での実根の数を求めます。

46.3.2 実根と複素数根を同時に求める関数

ROOTS p; これは ROOTS パッケージの主な関数で、これは実数及び複素数の全ての解をそれぞれ分離するに必要な精度で計算します。 (少なくとも 6 桁の精度は計算されます。) 結果は全ての解に対して等式の形のリストで得られます。また ROOTS は実数解および複素数解について別々に大域変数 ROOTSREAL および ROOTSCOMPLEX に解のリストを保存します。

ROOTS によって求められる解の順序は計算されるシステムによって異なります。異なる計算機によって計算された結果を容易に比較できるように、ROOTS の結果は一定の規則で並べ変えられたものが出力されます。実数部分の小さいものから、また実数部分が同じ解は虚数部分が大きいものから順に並べられます。(複素共役な解は虚数部が正のものが先にくるようにしています。)

しかし、多項式の因数分解 (無平方分解や実数解と複素数解を分離ために行われる分解) が行われた場合、解の並べ替えはそれぞれの因子について独立に行われます。このため結果はあまり自明な順序にはなっていません。しかしこの結果はシステムによって変わることはありません。

ROOTS_AT_PREC p; ROOTS と同じですが、計算される解の最小の精度は現在のシステムの設定と同じ桁数まで行われます。

ROOT_VAL p; ROOTS_AT_PREC と同じですが、結果として解の等式のリストを返すのではなく、解そのもののリストを返します。この関数は SOLVE によって使われています。

NEARESTROOT(p,s); この関数は、繰り返し法を使って、与えられた出発点 s (これは複素数値でもかまわない) からこれに最も近い解を求めます。もし、s の近傍に複数の解が存在し s はその一つにそれ以外の解と比べて非常に近い値でなければ、実際に求められる解は s に最も近い解とは限りません。従って、この関数は出発点 s の近くには一つの解しか存在しないことがあらかじめ分かっているときのみ使うべきです。

FIRSTROOT p; 関数 ROOTS を使って解を求めますが、ROOTS が最初に見つけた解のみを計算します。注意しておくことは、最初に見つけた解が ROOTS の出力リストで一番最初に現れるとは限らないということです。これは ROOTS が解を整理して出力しているからです。また、解を求めるのが困難な方程式の場合では、求められた最初の解は正しくないことがあります。

46.3.3 その他の関数

GETROOT(n,rr); もし rr が ROOTS,REALROOTS または NEARESTROOTS の出力であれば、GETROOT は n 番目の解を取り出します。もし n が 1 から解の数までの何れかでなければエラーを起こします。

MKPOLY rr; この関数は解のリスト rr からそれを解に持ち、分母が 1 であるような方程式を再構成します。この関数を使って、もし $rr := ROOTS\ p$ でかつ $rr1 := ROOTS\ MKPOLY\ rr$ としたとき $rr1$ と rr が等しくなることで、正しく解になっていることを確かめることが出来ます。(これは MULTIROOT と RATROOT がオンで ROUNDED がオフのときのみ正しい。) しかしながら、 $MKPOLY\ rr - NUM\ p = 0$ は全ての解が正確に計算できたときにのみ正しい。

46.3.4 診断用の関数

GFNEWT(p,r,cpx); この関数は多項式 p と解 r に対する GFNEWTON の一ステップを実行する。もし cpx が T であれば、解の虚数部はそれがどんなに小さくとも保存されます。

GFROOT(p,r,cpx); この関数は GFROOTFIND の 1 ステップを実行する。もし cpx が T であれば、解の虚数部はそれがどんなに小さくとも保存されます。

46.4 入力時に使われるスイッチ

代数モードでの多項式の入力は COMPLEX, ROUNDED および ADJPREC スイッチによって影響されます。 正しくないスイッチの設定を行った場合は、予期しない入力した係数の打ち切りや丸め等が行われます。

ROUNDED がオンであり、次のいずれかの条件が満たされた場合は、丸めや打ち切り等が行われます。

1. 係数は小数点数または有理数の形で入力されている。
2. COMPLEX がオンで、係数は虚数かまたは複素数である。

従って、予期しない丸めや打ち切り誤差を防ぐには

1. ROUNDED はオフにし、入力を整数または有理数で行う。
2. もし現在設定されている有効桁数までの打ち切りあるいは丸めを行ってもよければ ROUNDED はオンにしてもよい。また ROUNDED と ADJPREC を共にオンにすると、入力された最大の係数に合わせて有効桁数を調整してくれます。
3. 入力に複素数係数が含まれており、その実数部と虚数部の大きさが非常に異なっている場合 ROUNDED、ADJPREC と COMPLEX の三つのスイッチをオンにしておく必要があります。

整数と複素数モード (ROUNDED がオフ) 任意の実数係数多項式は整数係数を使って入力することが出来ます。整数もしくは有理数係数は COMPLEX スイッチの設定に関係なく、任意の実数もしくは複素数多項式の入力に使うことが出来ます。これは最も融通性ののある入力方法で、任意の実数もしくは複素数係数多項式を正確に入力することが出来ます。

ROUNDED および COMPLEX-ROUNDED モード (ROUNDED はオン) 整数係数の多項式は任意の大きさのものが入力可能です。浮動小数点数係数は入力時にシステムで設定された有効桁数で打ち切りまたは丸められます。もし COMPLEX がオンであれば、実数係数は整数の形で入力することで任意の桁数で入力することが出来ますが、複素数係数の虚数部は打ち切りまたは丸めが行われます。

46.5 入力と出力で使われるスイッチ

REDUCE の代数モードでのスイッチ ROUNDED と COMPLEX はこのパッケージの動作に影響を与えます。これらのスイッチはこのパッケージ内の関数中で変更されることがありますが、その場合でも最終的には元の状態に戻されます。(ただし重大なエラーを起こした場合は別です。)

COMPLEX このパッケージでは COMPLEX スイッチを内部で操作しています。複素数係数の多項式を扱うときにはオンにされます。実数係数多項式に対しては、NEARESTROOT の出発点として $rl + im * I$ の形で複素数を指定することが出来ますが、この場合でも COMPLEX スイッチの設定に関係なく正しく扱われます。つまり、COMPLEX スイッチに関係なく、複素数値解

が求められます。しかし、COMPLEX がオフであれば、出力時に虚数部が先に出力されます。COMPLEX がオンであればこれとは逆の順序で出力されます。

ROUNDED このパッケージでの計算は整数、ガウス整数、浮動小数点数や複素数の浮動小数点数の内からその時点で要求される計算モードで代数計算が行われます。このため、内部では BFTAG スイッチによって計算モードおよび計算の有効桁数を制御しています。ROUNDEC や COMPLEX スイッチには影響されません。しかし、出力が行われるときにはこれらのスイッチの設定に応じた処置が取られます。

46.6 Root パッケージのスイッチ

注意:以前の版にあったスイッチ AUTOMODE, ISOROOT および ACCROOT はなくなりました。

RATROOT (通常オフ) もし RATROOT がオンであれば、全ての解は有理式の形で出力されます。モードが COMPLEX(つまり ROUNDED がオフである) とすると、求めた解は打ち切りまたは丸めの影響を受けずに REDUCE に再入力することが可能です。(上で説明した MKPOLY の項を参照して下さい。)

MULTIROOT (通常オン) もし多項式が重複解を持っているときは、関数 SQFRF によって無平方因子に分解されます。もし MULTIROOT スイッチがオンであれば、ROOTS または REALROOTS の出力中で同じ解を複数解出力することで重複解であることを示します。もし MULTIROOT がオフであれば、解は一度しか出力されず、全ての解は互いに異なったもののみになります。(同じ解が現れることはありえません。もし、解が非常に接近しており、初期の計算精度では解が分離できないときには、プログラムは二つの解が分離できるまで自動的に計算の精度を上げて計算を行います。もし、最初の計算精度が非常に悪く、解を分離することが出来なければ、プログラムはエラーを起こして中断してしまいます。)

TRROOT (通常オフ) もしスイッチ TRROOT がオンであれば、解の計算の途中経過をトレースします。

ROOTMSG (通常オフ) もしスイッチ ROOTMSG が TRROOT スイッチと共にオンであれば、ラゲール法およびニュートン法の繰り返しを追跡するのに助けとなるような追加の情報が出力されます。これらの情報はもともとプログラムのテストとして用意したものです。

46.7 パラメータとその設定

ROOTACC# (通常 6) このパラメータは関数 ROOTACC n; で設定することが出来ます。このとき、ROOTACC#は n と 6 の大きい方の値に設定されます。もし、ACCROOT スイッチがオンであれば、解は少なくとも ROOTACC#桁まで求められます。(もし解が非常に接近しているような場合は、より多くの桁まで計算が行われます。)

システムの精度 このパッケージではシステムの計算精度は必要に応じて変更されます。しかし通常はもとの精度に戻して終了します。しかし、もとの精度が少なくて、全ての解を分離して出力するのに十分な精度を持っていなければ、システムの精度が上げられます。

PRECISION n; もし利用者がシステムの精度を PRECISION n; コマンドで設定を行った場合、この効果はシステムの計算精度 (有効桁数) を n にすると共に、ROOTS コマンドに対して ROOTACC n; を実行したのと同じ影響を与えます。つまり、解は少なくとも n 桁まで求められます。最初の設定に戻すには、PRECISION RESET; または PRECISION NIL; とすることでできます。

ROOTPREC n; このパッケージは通常計算モードと有効桁数を自動的に設定します。もし ROOTPREC n; を実行すると、n がシステムの有効桁数よりも大きければ、計算は最小の有効桁数 n ですべて行われます。ROOTPREC 0; を実行すると最初の状態に戻ります。

46.8 入力での多項式の打ち切りを避ける方法

このパッケージでは内部での計算では多項式の打ち切りは行いません。しかし、多項式を入力するときに打ち切りが行われることがあります。特に、入力が浮動小数点数で行われている場合には、このような係数の打ち切りが行われる可能性があります。

困難を避けるために、入力は整数もしくはガウスの整数形式で行うことが出来ます。

Lisp システムで扱える浮動小数点数の有効桁数は大域変数!!NFPD に入っています。また浮動小数点数を出力したときの有効桁数が大域変数!!FLIM に入れてあります。 本パッケージではこれらの値を計算して求めており、またこれらの値は計算の進めるのに使われます。

TRROOT スイッチをオンにしたとき出力される途中での解の値は実際の計算が浮動小数点数で行われているときでも、多倍長浮動小数点数 BIGFLOAT 形式で出力されます。これは解を出力するときに余計な丸め等が入らないようにするためです。

第47章　RSOLVE: 多項式の有理数/整数解の計算

Francis J. Wright
School of Mathematical Sciences
Queen Mary and Westfield College
University of London
Mile End Road, London E1 4NS, UK.
E-mail: *F.J.Wright@QMW.ac.uk*

27 January 1995

このパッケージでは`r/i_solve` という一変数多項式の有理数または整数根を、合同計算を使って高速に計算する関数を定義している。

47.1　序論

このパッケージでは高速な合同計算を使って、一変数多項式の有理数もしくは整数のゼロ点を求める関数を定義している。使っている方法は R. Loos (1983): Computing rational zeros of integral polynomials by p-adic expansion, *SIAM J. Computing*, **12**, 286–293 による方法を使っている。関数`r_solve` はすべての有理数のゼロ点を計算し、`i_solve` は整数のゼロ点を計算します。`r_solve` と `i_solve` の使い方はほとんど同じで、`solve` 演算子と同じように使えるように設計しています。operator, although しかし、`r_solve` と `i_solve` はもし有理数や整数のゼロ点しか必要でないならば、より使いやすくなっています。プログラムは`solve` を使うよりも二倍以上高速です。このパッケージを拡張してガウス整数やガウス有理数での解を求められるようにすることを計画しています。

47.2　使い方

第一引数は一変数の多項式または多項式系で、係数は整数、有理数または浮動小数点数でなければなりません。係数にパラメータを含んでいるものは扱えません。また現在のところ複素数係数の多項式も扱えません。引数は整数係数の多項式の商として、分母は単に無視されます。

二番目以降の引数はオプションです。もし、多項式の変数 (未知変数) を指定する場合には、第二引数として与えなければなりません。また、第二引数が正当なオプションとして指定可能なものでなければ変数を指定したものと (誤って) 解釈されます。しかし、式が $0 = 0$ のような自明な

式に簡約される場合を除けば、未知変数は方程式から導くことが出来ます。自明な方程式の場合には、i_solve の場合には、arbint を使って、また r_solve の場合にはarbrat を使って結果が表されます。i_solve の方が r_solve よりも少しだけ計算が速い。

結果はリストの形で返されます。通常、solve による結果と同じ形式になっています。解は変数=根の等式の形で、重複度は変数root_multiplicities に代入されます。もし、multiplicities スイッチがオンの時には、結果のリスト中に重複度を含めた個数の解が与えられます。(このときにはroot_multiplicities には重複度は代入されません。)

キーワード引数は、出力に対して、一時的にスイッチを切り替えたのと同様な効果をもたらします。これらのオプションには以下に示すものが指定できます。

separate: 重複度を変数root_multiplicities に代入します。(これは通常オンです。)

expand **または** multiplicities: 解を重複度を含めて表します。(multiplicities スイッチがオンの場合の規定の出力です。)

together: 解リストは、解と重複度のリストの形で出力されます。

nomul: 重複度を計算しません。

noeqs: 一変数方程式の場合、解を等式の形ではなく根の値のみ出力します。

47.3 例

```
r_solve((9x^2 - 16)*(x^2 - 9), x);
```

$$\left\{x = \frac{-4}{3}, x = 3, x = -3, x = \frac{4}{3}\right\}$$

```
i_solve((9x^2 - 16)*(x^2 - 9), x);
```

$$\{x = 3, x = -3\}$$

詳しくはテストファイルrsolve.tst を参照して下さい。

47.4 トレース

スイッチ trsolve はアルゴリズムのトレース出力をオンにする。通常はオフです。

第48章　SETS: 基本的な集合計算

Francis J. Wright
School of Mathematical Sciences
Queen Mary and Westfield College
University of London
Mile End Road, London E1 4NS, UK.
Email: *F.J.Wright@QMW.ac.uk*

REDUCE3.5 以降バージョンのための SETS パッケージは、集合 (あるいは、明示的な集合を表わして) と見なされたリストに対する、および識別子によって表わされる暗黙の集合に対する代数演算の支援を提供します。これには集合を値に持つ演算 `union`、`intersection`(`intersect`) および `setdiff`(\, `minus`)、およびブール値の内挿演算子 (述語関数)`member`、`subset_eq`、`subset`, `set_eq` があります。union と intersection は n 項演算です。また、それ以外は 2 項演算です。リストは、オペレーター `mkset` の適用により明示的に正規の集合表現に変換することができます。(パッケージは、さらに、代数のモードでブール値を表示されるための演算子 `evalb` と呼ばれる集合演算とは直接関係しない演算子を提供します。)

48.1　はじめに

REDUCE は、代数のモードの中にも、内部表現としても、いずれにも集合のための特定の表現を持っていません。また、数学的に集合であるすべてのオブジェクトもリストとして REDUCE の中で表わされます。集合とリストの間の差は、集合では要素間の順序が意味を持たないことと重複した要素が許されない (あるいは無視される) ことです。従って、リストは集合 (しかし逆にではなく) として完全に自然で満足な表現を与えます。Maple のようないくつかの言語は、集合およびリスト (それらは集合がより効率的に処理されることを可能にするかもしれない) のために異なる内部表現を提供しています。しかしこれは必ずしも必要ではありません。

このパッケージはリストを使って集合理論の演算をサポートしています。また正常な代数のモードでのリストとして結果を表わしています。従って、リストに当てはまる他のすべての REDUCE の機能をは明示的な集合演算によって構築されたリストに対して適用することが可能です。このパッケージによって提供される代数モードの集合演算はすべて、記号モードでは以前から利用可能で、REDUCE では内部的に利用されてきました。従って、REDUCE の中に集合演算を機能いれるのは新しいことではありません。このパッケージが行うのは、それらを代数のモードで利用可能にし、union と intersection 演算子に対してその引数の数を拡張し、それらの計算が束縛されていない識別子による暗黙の集合に対しても機能するようにしたことです。それは記号集合式を簡約します。しかし現在のところこれはかなり ad hoc でおそらく不完全なものです。

SETS パッケージの演算の例については、テスト・ファイル sets.tst を参照して下さい。このパッケージは実験段階で、さらなる開発を考慮中です; もし、あなたが改良 (あるいは修正) のための提案を持っているのであれば、電子メールによって私 (FJW) のもとへそれらを送ってください。パッケージは、REDUCE3.5 以降バージョンの下で実行されるように意図されています; もっと初期のバージョンででも正しく動くでしょうが、私はそのような使用のための支援を提供することができません。

48.2 内挿演算子

集合演算子は、標準の REDUCE の演算子優先リスト (REDUCE 3.7 のユーザーズマニュアル 2.7 章、15 ページを参照) に以下のように挿入されます:

```
or and not member memq = set_eq neq eq >= > <= < subset_eq
subset freeof + - setdiff union intersection * / ^ .
```

48.3 明示的な集合の表現と mkset

明示的な集合はリストによって表わされます。また、このパッケージは、集合と見なされるリストについての形式上の制限は全く要求しません。しかしながら、集合中の重複した要素は同じ要素であるとみなされます。重複した要素を削除して、単一の要素によって表すことは広く行なわれており、また便利です。これを正準表現と呼びます。集合中の要素の順序が意味を持たないことから、ある順序で並べ変えるのが便利です。正準表現で、要素をある適当な順序で並べ変えたものは正規表現を与えます。これは、同一の 2 つの集合が同一の表現を持つことを意味します。この場合、標準の REDUCE の等しいことを調べる述語 (=) は正しく集合の同値性を決定します。正規表現でなければ、このことは成り立ちません。

重複がある集合値の演算子の引数を前処理して重複を取り除くことは常に行なわれます。なぜならば、重複を取り除くことによって計算の効率が良くなるからです。従って、これらの演算子の引数には重複した要素は現れません。

主として結果がよりきれいになるので、集合はさらにソートされます。変数の順序は REDUCE が内部で行なっているのと同じように、ordp で調べて真になる順序にされます。しかし、整数については、ordp が減少する方向に順序付けられているのに対して、これとは逆の増加する方向に順序つけしています。

任意の集合演算の結果に現われる、明示的な集合は、正規表現で現在返されます。さらにいかなる明示的な集合は、それを表わすリストに演算子 mkset を適用することによりこの形式にすることができます。例えば

```
mkset {1,2,y,x*y,x+y};

{x + y,x*y,y,1,2}
```

空集合は空リスト{}によって表わされます。

48.4 Union および intersection

演算子 intersection (この名前は内部で使われています) には省略形 intersect があります。これらの演算子は明示的な集合に対して、内挿の二項演算子としてもっと一般的に使用されるでしょう。

```
{1,2,3} union {2,3,4};
```

```
{1,2,3,4}
```

```
{1,2,3} intersect {2,3,4};
```

```
{2,3}
```

これらは n 項演算子として定義されており、任意の引数を取ることができます。この場合、前置演算子として入力することで入力を簡単にすることができます。

```
{1,2,3} union {2,3,4} union {3,4,5};
```

```
{1,2,3,4,5}
```

```
intersect({1,2,3}, {2,3,4}, {3,4,5});
```

```
{3}
```

完全性のため、これらは現在のところ単項演算子としても使用できます。この場合、引数それ自身を (正規型で) 返します。したがって、mkset と同じように動作します。

```
union {1,5,3,5,1};
```

```
{1,3,5}
```

48.5 記号集合式

引数の 1 つ以上が未定義の識別子である場合、これは暗黙の集合を表しているものと解釈されます。そして union や intersection を含んだ式は評価した後も union や intersection を含んだままで返されます。これらの二つの演算子は対称であり、それらの演算子の引数は通常の対称関数と同様に並べ変えが行なわれます。このような記号集合式は簡約することができるでしょうが、多くの場合不完全な簡約しかできません。例えば:

```
a union b union {} union b union {7,3};
```

```
{3,7} union a union b
```

```
a intersect {};

{}
```

記号集合表現は他の集合演算子に対する有効な引数です。

```
a union (b intersect c);

b intersection c union a
```

intersection は union に対して分配可能です。この分配則は通常の状態では適応しないようになっています。しかしながら変数 set_distribution_rule には、置換規則として設定されています。これを有効にすることにって、分配則を適応した結果を得ることができます。

```
a intersect (b union c);

(b union c) intersection a

a intersect (b union c) where set_distribution_rule;

a intersection b union a intersection c
```

48.6 集合の差

集合の差はシンボル\によって表わされます。演算子としては、setdiff もしくは minus が使えます。それは 2 項演算子です。そのオペランドは明示的あるいは暗黙の集合の任意の組合せです。また、これは別の演算子の引数の中に現れることもできます。ここに、いくつかの例はあります:

```
{1,2,3} \ {2,4};

{1,3}

{1,2,3} \ {};

{1,2,3}

a \ {1,2};

a\{1,2}

a \ a;

{}
```

```
a \ {};

a

{} \ a;

{}
```

48.7 集合の述語関数

これらはすべて内挿の 2 項演算子です。現在、REDUCE の全ての述語のように、それらは (if、(while) および repeat) の条件文もしくは evalb の引数の中でのみ使えます。また、これらがブール値でなければエラーを起こします。

evalb はブール値を表示するための関数として作られています。REDUCE は true や false が特定の意味を持たないため、返す値が REDUCE に意味が内という点を除けば、evalb は Maple と同じような関数です。従って、REDUCE では、evalb を使う必要はありません。

```
if a = a then true else false;

true

evalb(a = a);

true

if a = b then true else false;

false

evalb(a = b);

false

evalb 1;

true

evalb 0;

false
```

例の説明のため、if 文と明示的な evalb オペレーターを使用しています。

48.7.1 集合のメンバーシップ

集合のメンバーシップは、述語 member によってテストされます。左辺は潜在的な集合要素で、右辺は集合です。右辺が暗黙の集合であるかどうかについては、現在のところ判断できません。これには、現在インプリメントされていない暗黙の集合のメンバーシップ (暗黙の変数依存と同じ) の宣言が必要です。集合のメンバーシップは次のように動作します。

```
evalb(1 member {1,2,3});

true

evalb(2 member {1,2} intersect {2,3});

true

evalb(a member b);

***** b invalid as list
(*****b はリストとして無効です。)
```

48.7.2 集合の包含

集合包含は述語 subset_eq によってテストされます、どこで a subset_eq b は、集合 a が b の部分集合である、あるいは集合 b と等しいときに真になります。厳密な包含は述語 subset によってテストされます。これは a が b の真の部分集合である場合に真になります。a が b に等しい場合には、偽になります。これらの述語は、記号モードでの集合に対しても動作します。しかし、以下の例に示されるように、正確ではありません。

```
evalb({1,2} subset_eq {1,2,3});

true

evalb({1,2} subset_eq {1,2});

true

evalb({1,2} subset {1,2});

false


evalb(a subset a union b);

true
```

```
evalb(a\b subset a);

true

evalb(a intersect b subset a union b);
%%% BUG

false
```

決定不能な述語は通常の REDUCE のエラーを引き起こします。例えば、

```
evalb(a subset_eq {b});

***** Cannot evaluate a subset_eq {b} as Boolean-valued set expression
(*****はブール値の集合に対して subset_eq{b}を評価することができません。)

evalb(a subset_eq b);
%%% BUG

false
```

48.7.3 集合の同一性

上に説明されるように、標準の REDUCE の等しいことを判定する述語 (=) は正規形式の 2 つの集合の同一性を正しくテストすることができます。このパッケージはさらに正規形式ではない 2 つの集合の同一性をテストするために述語 set_eq を供給します。

2 つの述語は記号モードでの集合式に対しては同じ動作を行ないます。なぜならこれらは評価されれば常に正規形式になるからです。(現在、これは単純な場合においてのみ恐らく厳密に真実ですが) ここに、いくつかの例はあります:

```
evalb({1,2,3} = {1,2,3});

true

evalb({2,1,3} = {1,3,2});

false

evalb(mkset{2,1,3} = mkset{1,3,2});

true
```

```
evalb({2,1,3} set_eq {1,3,2});

true

evalb(a union a = a\{});

true
```

48.8 インストール

IN を使用することで、必要ならソース・ファイル sets.red を REDUCE に読み込むことができます。”プロ”バージョンを使用する場合、読み込む前に ON COMP をオンにして行なうべきです。FASLOUT を使用して、FASL ファイルとしてコードをコンパイルし、次に、LOAD_PACKAGE(あるいは LOAD) を使ってロードする方がはるかに良いでしょう。一層の詳細に関しては、REDUCE マニュアルおよび特定のインプリメンテーションのガイドを参照してください。

このパッケージは REDUCE 内部手続き mk!*mq を再定義しなければなりません。このため警告が出ますが、これは無視します。私は、この再定義した関数が安全で REDUCE の他の演算に対していかなる予期しない結果も持たないだろうと信じ (また期待し) ます。

48.9 可能な将来の開発

- 集合上の繰り返された結合/交差をインプリメントするため、単項の結合/交差。
- より多くの集合代数演算、集合表現の正規形式、より完全な簡単化。
- true/false の代わりに 1/0 を返す evalb のバージョン (evalb10?) によるブール変数の支援、あるいはその直接 1/0 を返す述語関数。

第 49 章　SPARSE:疎行列の計算

Stephen Scowcroft
Konrad-Zuse-Zentrum für Informationstechnik Berlin

49.1　はじめに

REDUCE の強力な機能の一つは行列計算が簡単に実行できることです。このパッケージは標準の利用可能な行列計算の機能を、疎行列に対しても適用できるように拡張します。また、疎行列での線形代数に有用な関数を提供します。

パッケージのロード

このパッケージは `load_package sparse;` でロードされます。

49.2　疎行列の計算

この型の計算が可能なように式の型として `sparse` を追加しました。

49.2.1　疎な変数

識別子は疎な変数と `SPARSE` で宣言できます。疎行列のサイズは行列宣言で明示的に宣言する必要があります。例えば、

```
sparse aa(10,1),bb(200,200);
```

は `AA` を 10×1 の (列) 疎行列として、また `Y` を 200×200 の疎行列として宣言します。宣言 `SPARSE` は `MATRIX` とよくにています。一度、シンボルを疎行列であると宣言すると、通常の配列や演算子、変数名としては使えません。もっと詳しいことは REDUCE User's Manual[27] の行列変数の項を見て下さい。

49.2.2 疎行列要素の代入

一度、行列が疎行列であると宣言されると、全ての要素は 0 に初期化されます。このため、疎行列が最初に使われた時に、メッセージ

```
"The matrix is dense, contains only zeros"
("ゼロ要素だけの密行列です。")
```

が出力されます。行列が出力される場合はゼロでない要素のみが出力されます。これは、ゼロでない要素のみを保存しているからです。宣言された行列の要素に値を代入するには次の構文で行ないます。AA と BB が疎行列と宣言されていると仮定すると、次のように入力すればよい。

```
aa(1,1):=10;
bb(100,150):=a;
```

これは、行列 aa の (1,1) 成分を 10 に設定します。また、行列 bb の 100 行 150 列めを a に設定します。

49.2.3 疎行列の要素の評価

一度疎行列の要素が代入されれば、通常の配列要素と同様な方法でその値を参照することができます。aa(2,1) は、疎行列 AA の 2 行目 1 列目の要素を表します。

49.3 疎行列の式

これは、通常の行列演算の規則に従います。和と積は整合性のあるサイズの行列でなければなりません。さもなければエラーが発せられます。同様に冪乗の計算は正方行列に対してのみ行なえます。負の冪乗は逆行列の冪乗で計算されます。より詳しいことは、REDUCE User's Manual[27] の行列演算の項を参照して下さい。

49.4 疎行列を引数とする演算子

疎行列パッケージの関数は、行列パッケージの関数と、NULLSPACE 関数が定義されていないことを除けば、同じです。詳しくは REDUCE User's Manual[27] の行列を引数に持つ関数の節を参照して下さい。

49.4.1 例

以後、例題で使用する行列 $\mathcal{AA}$ は、

$$\mathcal{AA} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 9 \end{pmatrix}$$

の行列であるとします。

```
det ppp;

135

trace ppp;

18

rank ppp;

4

spmateigen(ppp,eta);

{{eta - 1,1,

  spm(1,1) := arbcomplex(4)\$
  },

 {eta - 3,1,

  spm(2,1) := arbcomplex(5)\$
  },

 {eta - 5,1,

  spm(3,1) := arbcomplex(6)\$
  },

 {eta - 9,1,

  spm(4,1) := arbcomplex(7)\$
  }}
```

49.5 疎行列の線形代数パッケージ

このパッケージは REDUCE の線形代数パッケージの拡張です。これらの関数はアルファベット順に 6 節で記述されています。これらは、6.1 から 6.47 までラベル付けされています。大きく分けて四つのクラスに分類できます。

49.5.1 基本的演算

spadd_columns	...	6.1	spadd_rows	...	6.2
spadd_to_columns	...	6.3	spadd_to_rows	...	6.4
spaugment_columns	...	6.5	spchar_poly	...	6.9
spcol_dim	...	6.12	spcopy_into	...	6.14
spdiagonal	...	6.15	spextend	...	6.16
spfind_companion	...	6.17	spget_columns	...	6.18
spget_rows	...	6.19	sphermitian_tp	...	6.21
spmatrix_augment	...	6.27	spmatrix_stack	...	6.29
spminor	...	6.30	spmult_columns	...	6.31
spmult_rows	...	6.32	sppivot	...	6.33
spremove_columns	...	6.35	spremove_rows	...	6.36
sprow_dim	...	6.37	sprows_pivot	...	6.38
spstack_rows	...	6.41	spsub_matrix	...	6.42
spswap_columns	...	6.44	spswap_entries	...	6.45
spswap_rows	...	6.46			

49.5.2 構成子

疎行列を作成する関数。

spband_matrix	...	6. 6	spblock_matrix	...	6. 7
spchar_matrix	...	6. 8	spcoeff_matrix	...	6. 11
spcompanion	...	6. 13	sphessian	...	6. 22
spjacobian	...	6. 23	spjordan_block	...	6. 24
spmake_identity	...	6. 26			

49.5.3 高度な演算

spchar_poly	...	6.9	spcholesky	...	6.10
spgram_schmidt	...	6.20	splu_decom	...	6.25
sppseudo_inverse	...	6.34	svd	...	6.43

49.5.4 述語

例題:

以下の例では、行列 $\mathcal{A}$ は、

$$\mathcal{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

であるとします。残念ながら、サイズの制限のために、例で “大きな” 疎行列を使うのは実際的ではありません。そのため、例の結果はほとんど自明なものに見えますが、これは、関数がどのように計算を行なうかについての感覚を与えるものです。

記法

以下では、$\mathcal{I}$ は単位行列を表すのに使います。$\mathcal{A}^T$ は行列 $\mathcal{A}$ の転置行列を表します。

49.6 利用可能な関数

49.6.1 spadd_columns, spadd_rows

`spadd_columns(`$\mathcal{A}$`,c1,c2,expr);`

$\mathcal{A}$:- 疎行列.
c1,c2 :- 正の整数.
expr :- スカラー式.

要約:

`spadd_columns` は行列 $\mathcal{A}$ の列 c2 を式 $*$ column($\mathcal{A}$,c1) + column($\mathcal{A}$,c2) で置き換えます。.

`spadd_rows` は同様の計算を $\mathcal{A}$ の行について行ないます。

例:

$$\texttt{spadd\_columns}(\mathcal{A}, 1, 2, x) \quad = \quad \begin{pmatrix} 1 & x & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

$$\texttt{spadd\_rows}(\mathcal{A},2,3,5) \quad = \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 25 & 9 \end{pmatrix}$$

関連する関数:

spadd_to_columns, spadd_to_rows, spmult_columns, spmult_rows.

49.6.2 spadd_rows

spadd_columns を見よ.

49.6.3 spadd_to_columns, spadd_to_rows

spadd_to_columns($\mathcal{A}$,column_list,expr);

$\mathcal{A}$	:-	疎行列.
column_list	:-	正の整数あるいは正の整数からなるリスト.
expr	:-	スカラー式.

要約:

spadd_to_columns は column_list で指定された行列 $\mathcal{A}$ の各列に expr を加えます。.

spadd_to_rows は同様の計算を $\mathcal{A}$ の行に対して行ないます。

例:

$$\texttt{spadd\_to\_columns}(\mathcal{A},\{1,2\},10) \quad = \quad \begin{pmatrix} 11 & 10 & 0 \\ 10 & 15 & 0 \\ 10 & 10 & 9 \end{pmatrix}$$

$$\texttt{spadd\_to\_rows}(\mathcal{A},2,-x) \quad = \quad \begin{pmatrix} 1 & 0 & 0 \\ -x & -x+5 & -x \\ 0 & 0 & 9 \end{pmatrix}$$

関連する関数:

spadd_columns, spadd_rows, spmult_rows, spmult_columns.

49.6.4 spadd_to_rows

spadd_to_columns を見よ.

49.6.5 spaugment_columns, spstack_rows

spaugment_columns($\mathcal{A}$,column_list);

$\mathcal{A}$:- 疎行列.
column_list :- 正の整数あるいは正の整数からなるリスト.

要約:

spaugment_columns は column_list で指定された $\mathcal{A}$ の列を保持し、一つにします。

spstack_rows は同様の計算を $\mathcal{A}$ の行に対して行ないます。

例:

$$\texttt{spaugment\_columns}(\mathcal{A},\{1,2\}) = \begin{pmatrix} 1 & 0 \\ 0 & 5 \\ 0 & 0 \end{pmatrix}$$

$$\texttt{spstack\_rows}(\mathcal{A},\{1,3\}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

関連する関数:

spget_columns, spget_rows, spsub_matrix.

49.6.6 spband_matrix

spband_matrix(expr_list,square_size);

expr_list :- 一つのスカラー式あるいは奇数個のスカラー式のリスト.

square_size :- 正の整数.

要約:

spband_matrix は square_size 次元の疎な正方行列を作成する.

例:

$$\texttt{spband\_matrix}(\{x,y,z\},6) = \begin{pmatrix} y & z & 0 & 0 & 0 & 0 \\ x & y & z & 0 & 0 & 0 \\ 0 & x & y & z & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \\ 0 & 0 & 0 & 0 & x & y \end{pmatrix}$$

関連する関数:

spdiagonal.

49.6.7 spblock_matrix

`spblock_matrix(r,c,matrix_list);`

r,c :- 正の整数.
matrix_list :- 疎行列あるいは行列からなるリスト.

要約:

`spblock_matrix` は matrix_list で与えられた行列で満たされた $r \times c$ の疎行列を作成する。

例:

$$\mathcal{B} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathcal{C} = \begin{pmatrix} 5 \\ 0 \end{pmatrix}, \quad \mathcal{D} = \begin{pmatrix} 22 & 0 \\ 0 & 0 \end{pmatrix}$$

$$\texttt{spblock\_matrix}(2, 3, \{\mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{C}, \mathcal{B}\}) \quad = \quad \begin{pmatrix} 1 & 0 & 5 & 22 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 22 & 0 & 5 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

49.6.8 spchar_matrix

`spchar_matrix(`$\mathcal{A}, \lambda$`);`

$\mathcal{A}$:- 疎な正方行列.
λ :- シンボルあるいは代数式.

要約:

`spchar_matrix` は $\mathcal{A}$ の特性行列 $\mathcal{C}$ を求める。

これは、$\mathcal{C} = \lambda * \mathcal{I} - \mathcal{A}$.

例:

$$\texttt{spchar\_matrix}(\mathcal{A}, x) \quad = \quad \begin{pmatrix} x-1 & 0 & 0 \\ 0 & x-5 & 0 \\ 0 & 0 & x-9 \end{pmatrix}$$

関連する関数:

`spchar_poly`.

49.6.9 spchar_poly

`spchar_poly(`$\mathcal{A}, \lambda$`);`

$\mathcal{A}$:- 疎な正方行列.
λ :- シンボルもしくは代数式.

要約:

spchar_poly は $\mathcal{A}$ の特性多項式を計算する.

これは、$\lambda * \mathcal{I} - \mathcal{A}$ の行列式です。

例:

spchar_poly(A,x) $= x^3 - 15 * x^2 - 59 * x - 45$

関連する関数:

spchar_matrix.

49.6.10 spcholesky

spcholesky($\mathcal{A}$);

$\mathcal{A}$:- 数値要素のみの正定値の疎行列.

要約:

spcholesky は $\mathcal{A}$ の cholesky 分解を計算します。

これは、{$\mathcal{L}$,$\mathcal{U}$} を返します。ここで、$\mathcal{L}$ および $\mathcal{U}$ は $\mathcal{A} = \mathcal{L}\mathcal{U}$ であり、また $\mathcal{U} = \mathcal{L}^T$ をみたす、下三角行列および上三角行列です。

例:

$$\mathcal{F} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

$$\text{cholesky}(\mathcal{F}) = \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 0 & \sqrt{5} & 0 \\ 0 & 0 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & \sqrt{5} & 0 \\ 0 & 0 & 3 \end{pmatrix} \right\}$$

関連する関数:

splu_decom.

49.6.11 spcoeff_matrix

spcoeff_matrix({lin_eqn$_1$,lin_eqn$_2$, ...,lin_eqn$_n$});

lin_eqn$_1$,lin_eqn$_2$, ...,lin_eqn$_n$:- 線形方程式. *equation* = *number* の形式もしくは単に *equation* だけ.

要約:

spcoeff_matrix は線形方程式の係数行列 $\mathcal{C}$ を求める。

これは、$\mathcal{C}\mathcal{X} = \mathcal{B}$ を満たす、$\{\mathcal{C}, \mathcal{X}, \mathcal{B}\}$ を返します。

例:

spcoeff_matrix$(\{y - 20 * w = 10, y - z = 20, y + 4 + 3 * z, w + x + 50\}) =$

$$\left\{\begin{pmatrix} 1 & -20 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & 3 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}, \begin{pmatrix} y \\ w \\ z \\ x \end{pmatrix}, \begin{pmatrix} 10 \\ 20 \\ -4 \\ 50 \end{pmatrix}\right\}$$

49.6.12 spcol_dim, sprow_dim

column_dim($\mathcal{A}$);

$\mathcal{A}$:- 疎行列.

要約:

spcol_dim は $\mathcal{A}$ の列の次元を求める。

sprow_dim は $\mathcal{A}$ の行の次元を求める。

例:

spcol_dim$(\mathcal{A}) = 3$

49.6.13 spcompanion

spcompanion(poly,x);

poly :- モニックな x に関する一変数多項式.
x :- 変数名.

要約:

spcompanion は多項式 poly のコンパニオン行列 $\mathcal{C}$ を作る。

これは、n 次元の正方行列です。n は多項式 poly の x に関する次数です。

The entries of $\mathcal{C}$ の要素は: $\mathcal{C}$(i,n) = -coeffn(poly,x,i-1) for i = 1 ... n, $\mathcal{C}$(i,i-1) = 1 for i = 2 ... n それ以外は 0.

例:

$$\text{spcompanion}(x^4 + 17 * x^3 - 9 * x^2 + 11, x) \quad = \quad \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$$

関連する関数:

spfind_companion.

49.6.14 spcopy_into

spcopy_into($\mathcal{A}$,$\mathcal{B}$,r,c);

$\mathcal{A},\mathcal{B}$:- 疎行列もしくは行列.
r,c :- 正の整数.

要約:

spcopy_into は行列 $\mathcal{A}$ を、$\mathcal{A}$ の (1,1) 要素が $\mathcal{B}$ の (r,c) 要素になるように、$\mathcal{B}$ にコピーします。

例:

$$\mathcal{G} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\texttt{spcopy\_into}(\mathcal{A},\mathcal{G},1,2) \quad = \quad \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 9 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

関連する関数:

spaugment_columns, spextend, spmatrix_augment, spmatrix_stack, spstack_rows, spsub_matrix.

49.6.15 spdiagonal

spdiagonal({mat$_1$,mat$_2$, ...,mat$_n$});[1]

mat$_1$,mat$_2$, ...,mat$_n$:- 要素はスカラー式もしくは疎な正方行列あるいは通常の正方行列.

要約:

spdiagonal は入力されたものが対角要素となる疎行列を作る。

例:

$$\mathcal{H} = \begin{pmatrix} 66 & 77 \\ 88 & 99 \end{pmatrix}$$

[1] 括弧 {} は省略可能です。

$$\texttt{spdiagonal}(\{\mathcal{A}, x, \mathcal{H}\}) \quad = \quad \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 66 & 77 \\ 0 & 0 & 0 & 0 & 88 & 99 \end{pmatrix}$$

関連する関数:

spjordan_block.

49.6.16 spextend

spextend($\mathcal{A}$,r,c,expr);

$\mathcal{A}$:- 疎行列.
r,c :- 正の整数.
expr :- 代数式あるいはシンボル.

要約:

spextend は $\mathcal{A}$ を r 行 c 列の行列に拡張した行列を返す。新しい要素には、expr が代入される。してそれをコピーを返す。

例:

$$\texttt{spextend}(\mathcal{A}, 1, 2, 0) \quad = \quad \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

関連する関数:

spcopy_into, spmatrix_augment, spmatrix_stack, spremove_columns, spremove_rows.

49.6.17 spfind_companion

spfind_companion($\mathcal{A}$,x);

$\mathcal{A}$:- 疎行列.
x :- 変数名.

要約:

与えられた疎なコンパニオン行列に対して、spfind_companion は対応する多項式を計算する。

例:

$$\mathcal{C} = \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$$

$\texttt{spfind\_companion}(\mathcal{C}, x) = x^4 + 17 * x^3 - 9 * x^2 + 11$

関連する関数:

spcompanion.

49.6.18 spget_columns, spget_rows

spget_columns($\mathcal{A}$,column_list);

$\mathcal{A}$:- 疎行列.
c :- 正の整数あるいは正の整数のリスト.

要約:

spget_columns は行列 $\mathcal{A}$ から column_list で指定された列を取り除き、列行列のリストとして返す。

spget_rows は同様な計算を $\mathcal{A}$ の行に対して行なう。

例:

$$\texttt{spget\_columns}(\mathcal{A}, \{1,3\}) \quad = \quad \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 9 \end{pmatrix} \right\}$$

$$\texttt{spget\_rows}(\mathcal{A}, 2) \quad = \quad \left\{ \begin{pmatrix} 0 & 5 & 0 \end{pmatrix} \right\}$$

関連する関数:

spaugment_columns, spstack_rows, spsub_matrix.

49.6.19 spget_rows

spget_columns を見よ.

49.6.20 spgram_schmidt

spgram_schmidt({vec$_1$,vec$_2$, ...,vec$_n$});

vec$_1$,vec$_2$, ...,vec$_n$:- 線形独立なベクトル. 各ベクトルはあらかじめ疎行列と宣言された行列からなるリストとして表されている必要がある。例えば、eg: sparse a(4,1);, a(1,1):=1;

要約:

spgram_schmidt は入力されたベクトルの gram_schmidt の直交化を計算する。

正規化された直交ベクトルのリストを返す。

例:

a,b,c,d が次のような要素を持つ疎行列とする。lists:- {{1,0,0,0}, {1,1,0,0}, {1,1,1,0}, {1,1,1,1}}.

spgram_schmidt({{a},{b},{c},{d}}) = {{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}}

49.6.21 sphermitian_tp

sphermitian_tp($\mathcal{A}$);

$\mathcal{A}$:- 疎行列.

要約:

sphermitian_tp は $\mathcal{A}$ のエルミート転置行列を計算する.

例:

$$\mathcal{J} = \begin{pmatrix} i+1 & i+2 & i+3 \\ 0 & 0 & 0 \\ 0 & i & 0 \end{pmatrix}$$

$$\texttt{sphermitian\_tp}(\mathcal{J}) = \begin{pmatrix} -i+1 & 0 & 0 \\ -i+2 & 0 & -i \\ -i+3 & 0 & 0 \end{pmatrix}$$

関連する関数:

tp[2] .

49.6.22 sphessian

sphessian(expr,variable_list);

[2] 標準の転置行列の計算は REDUCE User's Manual[27] を参照して下さい.

expr :- スカラー式.
variable_list :- 一つの変数あるいは変数のリスト.

要約:

sphessian は variable_list で与えられた変数に関する expr のヘッセ行列を計算する.

例:

$$\texttt{sphessian}(x*y*z+x^2,\{w,x,y,z\}) \quad = \quad \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & z & y \\ 0 & z & 0 & x \\ 0 & y & x & 0 \end{pmatrix}$$

49.6.23 spjacobian

```
spjacobian(expr_list,variable_list);
```

expr_list :- 一つの代数式あるいは代数式のリスト.

variable_list :- 変数あるいは変数のリスト.

要約:

spjacobian は expr_list の変数 variable_list に関するヤコビ行列を計算する.

例:

$\texttt{spjacobian}(\{x^4, x*y^2, x*y*z^3\},\{w,x,y,z\}) =$

$$\begin{pmatrix} 0 & 4*x^3 & 0 & 0 \\ 0 & y^2 & 2*x*y & 0 \\ 0 & y*z^3 & x*z^3 & 3*x*y*z^2 \end{pmatrix}$$

関連する関数:

sphessian, df[3] .

49.6.24 spjordan_block

```
spjordan_block(expr,square_size);
```

expr :- 代数式あるいはシンボル.
square_size :- 正の整数

要約:

spjordan_block は square_size の大きさの正方ジョルダンブロック行列 $\mathcal{J}$ を計算する.

[3] 標準の微分演算については REDUCE User's Manual[27] を参照して下さい.

例:

$$\texttt{spjordan\_block(x,5)} \quad = \quad \begin{pmatrix} x & 1 & 0 & 0 & 0 \\ 0 & x & 1 & 0 & 0 \\ 0 & 0 & x & 1 & 0 \\ 0 & 0 & 0 & x & 1 \\ 0 & 0 & 0 & 0 & x \end{pmatrix}$$

関連する関数:

spdiagonal, spcompanion.

49.6.25 splu_decom

splu_decom($\mathcal{A}$);

$\mathcal{A}$:- 要素が数値あるいは数値係数の複素数のみの疎行列.

要約:

splu_decom は $\mathcal{A}$ の LU 分解を計算する, $\mathcal{A} = \mathcal{LU}$ を満たす下三角行列 $\mathcal{L}$ と上三角行列 $\mathcal{U}$ を計算し、{$\mathcal{L}$,$\mathcal{U}$} を返す.

注意:

使用しているアルゴリズムは、計算途中で $\mathcal{A}$ の行の入れ換えを行ないます。従って、$\mathcal{LU}$ does は $\mathcal{A}$ そのものには等しくならず、行同値な行列になります。このため、splu_decom は {$\mathcal{L}$,$\mathcal{U}$,vec} を返します。spconvert($\mathcal{A}$,vec) を呼び出すことで、分解されて行列表現が得られます。つまり、$\mathcal{LU}$ = spconvert($\mathcal{A}$,vec) となります。

例:

$$\mathcal{K} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

$$\texttt{lu} := \texttt{splu\_decom}(\mathcal{K}) \quad = \quad \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, [\,1\,2\,3\,] \right\}$$

$$\texttt{first lu * second lu} \quad = \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

$$\texttt{convert}(\mathcal{K},\texttt{third lu}) \quad = \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

関連する関数:

spcholesky.

49.6.26 spmake_identity

spmake_identity(square_size);

square_size :- 正の整数.

要約:

spmake_identity はサイズが square_size である単位行列を作ります.

例:

$$\texttt{spmake\_identity}(4) \quad = \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

関連する関数:

spdiagonal.

49.6.27 spmatrix_augment, spmatrix_stack

spmatrix_augment({mat$_1$,mat$_2$, ...,mat$_n$});[4]

mat$_1$,mat$_2$, ...,mat$_n$:- 行列.

要約:

spmatrix_augment は matrix_list 中の行列を水平方向に結合します。

spmatrix_stack は行列を垂直方向に結合します。

例:

$$\texttt{spmatrix\_augment}(\{\mathcal{A}, \mathcal{A}\}) \quad = \quad \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 5 & 0 & 0 & 5 & 0 \\ 0 & 0 & 9 & 0 & 0 & 9 \end{pmatrix}$$

[4] 括弧 {} は省略可能です。

$$\texttt{spmatrix\_stack}(\{\mathcal{A},\mathcal{A}\}) \quad = \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \\ 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

関連する関数:

spaugment_columns, spstack_rows, spsub_matrix.

49.6.28 matrixp

matrixp(test_input);

test_input :- 任意.

要約:

matrixp は入力されたものが疎行列あるいは行列である場合には t, それ以外の場合には nil を返すブール値の関数です。

例:

matrixp($\mathcal{A}$) = t

matrixp(doodlesackbanana) = nil

関連する関数:

squarep, symmetricp, sparsematp.

49.6.29 spmatrix_stack

spmatrix_augment を見よ.

49.6.30 spminor

spminor($\mathcal{A}$,r,c);

$\mathcal{A}$:- 疎行列.
r,c :- 正の整数.

要約:

spminor は $\mathcal{A}$ の (r,c) に関する余因子行列を計算します.

例:

$$\texttt{spminor}(\mathcal{A},1,3) \quad = \quad \begin{pmatrix} 0 & 5 \\ 0 & 0 \end{pmatrix}$$

関連する関数:

spremove_columns, spremove_rows.

49.6.31 spmult_columns, spmult_rows

spmult_columns($\mathcal{A}$,column_list,expr);

$\mathcal{A}$	:-	疎行列.
column_list	:-	正の整数あるいは正の整数のリスト.
expr	:-	代数式.

要約:

spmult_columns は $\mathcal{A}$ で column_list で指定した列を expr 倍した行列を返します.

spmult_rows は同様な計算を行に関して行ないます。

例:

$$\texttt{spmult\_columns}(\mathcal{A},\{1,3\},x) \quad = \quad \begin{pmatrix} x & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9*x \end{pmatrix}$$

$$\texttt{spmult\_rows}(\mathcal{A},2,10) \quad = \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 50 & 0 \\ 0 & 0 & 9 \end{pmatrix}$$

関連する関数:

spadd_to_columns, spadd_to_rows.

49.6.32 spmult_rows

spmult_columns を見よ.

49.6.33 sppivot

sppivot($\mathcal{A}$,r,c);

$\mathcal{A}$:- 疎行列.
r,c :- $\mathcal{A}$(r,c) が 0 でないような正の整数.

要約:

sppivot は (r,c) 要素をピボットとして $\mathcal{A}$ の消去を行なう。

これを実行するために、行列の各行には r 番めの行の倍数が加えられる。

結果として、(r,c) 要素を除いて c 列は 0 になります。

関連する関数:

sprows_pivot.

49.6.34 sppseudo_inverse

sppseudo_inverse($\mathcal{A}$);

$\mathcal{A}$:- 疎行列.

要約:

sppseudo_inverse は Moore-Penrose の逆行列とも呼ばれる $\mathcal{A}$ の擬逆行列を計算する。

例:

$$\mathcal{R} = \begin{pmatrix} 0 & 0 & 3 & 0 \\ 9 & 0 & 7 & 0 \end{pmatrix}$$

$$\texttt{sppseudo\_inverse}(\mathcal{R}) = \begin{pmatrix} -0.26 & 0.11 \\ 0 & 0 \\ 0.33 & 0 \\ 0.25 & -0.05 \end{pmatrix}$$

関連する関数:

spsvd.

49.6.35 spremove_columns, spremove_rows

spremove_columns($\mathcal{A}$,column_list);

$\mathcal{A}$:- 疎行列.
column_list :- 正の整数あるいは正の整数からなるリスト.

要約:

spremove_columns は $\mathcal{A}$ から column_list で指定された列を取り除く。

spremove_rows は同様な計算を $\mathcal{A}$ の行に対して行なう.

例:

$$\texttt{spremove\_columns}(\mathcal{A},2) \;=\; \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 9 \end{pmatrix}$$

$$\texttt{spremove\_rows}(\mathcal{A},\{1,3\}) \;=\; \begin{pmatrix} 0 & 5 & 0 \end{pmatrix}$$

関連する関数:

spminor.

49.6.36 spremove_rows

spremove_columns を見よ.

49.6.37 sprow_dim

spcolumn_dim を見よ.

49.6.38 sprows_pivot

sprows_pivot($\mathcal{A}$,r,c,{row_list});

$\mathcal{A}$:- 疎素行列.
r,c :- $\mathcal{A}$(r,c) が 0 でないような正の整数.
row_list :- 正の整数あるいは正の整数からなるリスト.

要約:

sprows_pivot は sppivot と同様な計算を、row_list で指定される行について行なう。

関連する関数:

sppivot.

49.6.39 sparsematp

sparsematp($\mathcal{A}$);

$\mathcal{A}$:- 行列.

要約:

sparsematp は行列が sparse であると宣言されていれば t, それ以外の場合は nil を返すブール値関数.

例:

L:= `mat((1,2,3),(4,5,6),(7,8,9));`

`sparsematp`($\mathcal{A}$) = t

`sparsematp`($\mathcal{L}$) = nil

関連する関数:

`matrixp`, `symmetricp`, `squarep`.

49.6.40 squarep

`squarep(`$\mathcal{A}$`);`

$\mathcal{A}$:- 行列.

要約:

`squarep` は行列が正方行列であれば t、それ以外の場合は nil を返すブール値関数。

例:

$$\mathcal{L} = \begin{pmatrix} 1 & 3 & 5 \end{pmatrix}$$

`squarep`($\mathcal{A}$) = t

`squarep`($\mathcal{L}$) = nil

関連する関数:

`matrixp`, `symmetricp`, `sparsematp`.

49.6.41 spstack_rows

`spaugment_columns` を見よ.

49.6.42 spsub_matrix

`spsub_matrix(`$\mathcal{A}$`,row_list,column_list);`

$\mathcal{A}$	:-	疎行列.
row_list, column_list	:-	正の整数あるいは正の整数からなるリスト.

要約:

`spsub_matrix` は row_list で指定される行と、column_list で指定される列からなる行列を生成する.

例:

$$\texttt{spsub\_matrix}(\mathcal{A},\{1,3\},\{2,3\}) \;=\; \begin{pmatrix} 5 & 0 \\ 0 & 9 \end{pmatrix}$$

関連する関数:

spaugment_columns, spstack_rows.

49.6.43 spsvd (特異値分解)

spsvd($\mathcal{A}$);

$\mathcal{A}$:- 数値を要素とする疎行列.

要約:

spsvd は $\mathcal{A}$ の特異値分解を求める。

この関数は、$\{\mathcal{U},\sum,\mathcal{V}\}$ を返します。ここで、 $\mathcal{A}=\mathcal{U}\sum\mathcal{V}^T$ そして $\sum = diag(\sigma_1,\ldots,\sigma_n)$。$\sigma_i$, $i=(1\ldots n)$ は $\mathcal{A}$ の特異値.

n は $\mathcal{A}$ の列の次元.

例:

$$\mathcal{Q} = \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix}$$

$$\texttt{svd}(\mathcal{Q}) \;=\; \left\{ \begin{pmatrix} -1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 1.0 & 0 \\ 0 & 5.0 \end{pmatrix}, \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \right\}$$

49.6.44 spswap_columns, spswap_rows

spswap_columns($\mathcal{A}$,c1,c2);

$\mathcal{A}$:- 疎行列.
c1,c1 :- 正の整数.

要約:

spswap_columns は $\mathcal{A}$ の列 c1 と c2 を交換する

spswap_rows は $\mathcal{A}$ の行を交換する

例:

$$\texttt{spswap\_columns}(\mathcal{A},2,3) \quad = \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 5 \\ 0 & 9 & 0 \end{pmatrix}$$

関連する関数:

spswap_entries.

49.6.45 swap_entries

spswap_entries($\mathcal{A}$,{r1,c1},{r2,c2});

$\mathcal{A}$:- 疎行列.
r1,c1,r2,c2 :- 正の整数.

要約:

spswap_entries は $\mathcal{A}$(r1,c1) と $\mathcal{A}$(r2,c2) を交換する.

例:

$$\texttt{spswap\_entries}(\mathcal{A},\{1,1\},\{3,3\}) \quad = \quad \begin{pmatrix} 9 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

関連する関数:

spswap_columns, spswap_rows.

49.6.46 spswap_rows

spswap_columns を見よ。

49.6.47 symmetricp

symmetricp($\mathcal{A}$);

$\mathcal{A}$:- 行列.

要約:

symmetricp はブール値の関数で、行列が対称である時に t を返し、それ以外は nil を返します。

例:

$$\mathcal{M} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

symmetricp($\mathcal{A}$) = t

symmetricp($\mathcal{M}$) = nil

関連する関数:

matrixp, squarep, sparsematp.

49.7 計算の高速化

fast_la スイッチをオンにすることによって、次の関数の計算が早くなります。

spadd_columns	spadd_rows	spaugment_columns	spcol_dim
spcopy_into	spmake_identity	spmatrix_augment	spmatrix_stack
spminor	spmult_column	spmult_row	sppivot
spremove_columns	spremove_rows	sprows_pivot	squarep
spstack_rows	spsub_matrix	spswap_columns	spswap_entries
spswap_rows	symmetricp		

速度の増加は、十分多く (数千回) 関数を使用しないと、あまり明確ではありません。このスイッチを使う時は、エラーチェックが最小限しか行なわれません。つまり、不正な入力に対しては奇妙なエラーメッセージが出力されるかもしれません。

49.8 謝辞

このパッケージは Matt Rebbeck[37] による REDUCE のための線形代数パッケージの拡張です。

spcholesky, splu_decom それに spsvd のアルゴリズムは、J.H. Wilkinson and C. Reinsch [38] による線形代数の本からとりました。

spgram_schmidt のコードは、Karin Gatermann による REDUCE のための Symmetry パッケージ [25] から来ています。

第50章　SPDE:偏微分方程式の対称性

Fritz Schwarz

GMD, Institut F1
Postfach 1240
5205 St. Augustin
GERMANY

Telephone: +49-2241-142782
Email: *fritz.schwartz@gmd.de*

SPDE パッケージでは与えられた偏微分方程式もしくは偏微分方程式系の Lie 対称性または点対称性の対称群を決定するため使用できる関数を定義しています。これは計算機の端末から対話的に計算を進めるよう考えられています。多くの場合、決定方程式は完全に自動的に解かれます。しかしながら、方程式によっては、利用者が必要な情報を与えないと解を求めることが出来ない場合もあります。

このパッケージはコンパイルした形でしか使用することが出来ません。

理論的な点に関しては、算法と数多くの例題が論文 “Automatically Determining Symmetries of Partial Differential Equations”, Computing vol. 34, pp. 91–106 (1985) と vol. 36, pp. 279–280 (1986), “Symmetries of Differential Equations: From Sophus Lie to Computer Algebra”, SIAM Review それに Lecture Notes “Computer Algebra and Differential Equations of Mathematical Physics” の第二章に載っています。

50.1　関数と変数の説明

偏微分方程式系の対称性の解析は三つの部分から構成されています。このパッケージで最も重要な部分は表 50.1 に挙げる関数です。

種々な出力を得るのに有用な関数は表 50.2 にあげています。

関数名	説明
CRESYS(< **引数** >)	決定方程式の構成
SIMPSYS()	決定方程式の解法
RESULT()	無限小生成子と交換関係の表の出力

表 50.1: SPDE の関数

関数名	説明
PRSYS()	決定方程式の出力
PRGEN()	無限小生成子の出力
COMM(U,V)	生成子 U,V の交換関係の出力

表 50.2: SPDE の有用な出力関数

変数名	意味
NN	独立変数の数
MM	従属変数の数
PCLASS=0, 1 または 2	出力の制御

表 50.3: SPDE で使われる整数値の変数名

システムで定義されている大域変数には、表 50.3 と 50.4 で説明されている以外の目的には使えないものがあります。整数値を取る変数は表 50.3 にあげています。

また上記に加えて、オペレータを値にもつ変数が表 50.4 にまとめてあります。

微分方程式はオペレータ `deq i` の値として、表 50.4 の記号に従って、代入しておかなければなりません。表の三番目と最後の項目は、高次微分の表し方を示しています。

i 番目の方程式 `deq i` に対して、解こうとする微分係数を変数 `sder i` に代入しておかなければなりません。ただし、単一の方程式の場合にはこの必要はありません。最も高い微分が選択されます。

変数 `deq` に対して代入が行われれば、NN と MM の値は自動的に設定されるので、利用者が設定する必要はありません。

関数 CRESYS は任意の引数で呼び出すことが出来ます。つまり、

```
CRESYS();  または CRESYS(deq 1,deq 2,... );
```

はいずれも正しい呼び出し方になっています。もし、何の引数の与えずに呼び出すと、現在 `deq` に設定されている全ての方程式が計算の対称となります。例えば、`deq 1`,`deq 2`, `deq 3` に微分方程式が設定されている場合。この三つの方程式で構成される微分方程式系の対称性を計算するには、

```
CRESYS();   または   CRESYS(deq 1,deq 2,deq 3);
```

の何れでも同じことです。

もし、後のセッションで `deq 1` のみの対称群を求めたいときには、次のように入力すればいいのです。

```
CRESYS deq 1;
```

変数名	意味
X(I)	独立変数 x_i
U(ALFA)	従属変数 u^{alfa}
U(ALFA,I)	u^{alfa} の x_i に関する微分係数
DEQ(I)	i 番目の微分方程式
SDER(I)	DEQ(I) を解くべき変数 (微分係数)
GL(I)	i 番目の決定方程式
GEN(I)	i 番目の無限小の生成子
XI(I), ETA(ALFA) ZETA(ALFA,I)	最初に述べた参考文献の説明を 参照して下さい。
C(I)	代入に用いられる i 番目の関数

表 50.4: SPDE で使われるオペレータ値の変数名

決定方程式が構成されれば、SIMPSYS によってこの方程式を解きます。この関数は引数なしで呼び出します。SIMPSYS が出力する中間での計算の途中結果は、変数 PCLASS で制御できます。既定値として 0 になっていますが、この場合には途中結果は出力されません。 PCLASS が 1 の時には、全ての中間結果が出力されます。これによって、計算の詳しい追跡が出来ます。計算が主なループを通る度に、

```
Entering main loop
```

が出力されます。PCLASS が 2 の場合には、多数の LISP 形式での出力が表示されますが、これは通常の利用者にはあまり関係がありません。

もし PCLASS=0 の時には、SIMPSYS は何も出力せずに終了します。このときには決定方程式は完全に解かれています。場合によっては、決定方程式を一回で完全に解くことが出来ないこともあります。一般的には、現在インプリメントされている算法では解くことが出来ない、真の微分方程式のみが残る場合に、このようなことが起こります。この場合には、SIMPSYS は解けずに残っている方程式を出力します。計算を続けるには、利用者が適当な代入、例えば返された方程式の解を与える等、を行わなくてはなりません。この時に導入される新しい関数は、c(k) という名前の演算子とし、また正しい依存関係を `DEPEND` で宣言しておかなくてはなりません。この番号は現在の関数の番号と合わせる必要があります。この値は SIMPSYS によっても返されます。

決定方程式が解ければ、RESULT を呼びます。これは、無限小の生成元とその交換関係を表示します。

50.2 パッケージの使い方

いくつかの例をとってこのパッケージの使い方を説明します。まず、拡散方程式を考えます。上で説明した記号を使って次のように表します。

```
deq 1:=u(1,1)+u(1,2,2);
```

deq 1 に値として代入します。これはただ一つの方程式なので、 sder 1 に値を代入する必要はありません。

決定方程式は次の入力で求められます。

```
CRESYS(); もしくは CRESYS deq 1;
```

もし 1 以外の i に対して deq i に値を代入しているような時には、後の方の呼び出し方を使わなくてはなりません。

もし deq に微分方程式が設定されていなければ、

```
***** Differential equations not defined
(***** 微分方程式が定義されていません。)
```

のエラーメッセージが出力されます。

もし、利用者が解を求めることを始める前に、決定方程式の内容を見たければ次のように入力します。

```
PRSYS();
```

そうすれば、次のような出力が得られます。

```
GL(1):=2*DF(ETA(1),U(1),X(2)) - DF(XI(2),X(2),2) -
       DF(XI(2),X(1))

GL(2):=DF(ETA(1),U(1),2) - 2*DF(XI(2),U(1),X(2))

GL(3):=DF(ETA(1),X(2),2) + DF(ETA(1),X(1))

GL(4):=DF(XI(2),U(1),2)

GL(5):=DF(XI(2),U(1)) - DF(XI(1),U(1),X(2))

GL(6):=2*DF(XI(2),X(2)) - DF(XI(1),X(2),2) - DF(XI(1),X(1))

GL(7):=DF(XI(1),U(1),2)

GL(8):=DF(XI(1),U(1))

GL(9):=DF(XI(1),X(2))

The remaining dependencies
```

```
XI(2) depends on U(1),X(2),X(1)

XI(1) depends on U(1),X(2),X(1)

ETA(1) depends on U(1),X(2),X(1)
```

後のメッセージは、XI(1),XI(2),ETA(1) の三つの関数は X(1),X(2),U(1) に従属していることを意味します。この情報無しでは、決定方程式を構成する GL(1) から GL(9) までの九つの方程式は意味をなしません。この決定方程式を解くには、次のように入力します。

```
SIMPSYS();
```

もし出力フラグ PCLASS が既定値の 0 であれば、中間の計算の記録は出力されず、次のような結果を返します。

```
Determining system is not completely solved

The remaining equations are

GL(1):=DF(C(1),X(2),2) + DF(C(1),X(1))

Number of functions is 16

The remaining dependencies

C(1) depends on X(2),X(1)
```

もし PCLASS が 1 であれば、約 6 ページにわたる中間計算の結果が出力されます。これによって、利用者は計算の各ステップを追跡することが出来ます。

いま考えている例では決定方程式は完全には解かれません。拡散方程式は線形であり、従って対称性の生成元には元の方程式の解となる関数に依存するものが含まれるため、これは当然予想される結果です。この時には、利用者は追加の情報を与えてやる必要があります。いまの例では、次のように入力すればよい。

```
DF(C(1),X(1)) := - DF(C(1),X(2),2);
```

この後、決定方程式を解く関数を再度呼び出すと、

```
SIMPSYS();
```

今度は何のメッセージもなく計算は終了します。つまり、システムが解けなかった方程式はもうこれ以上なく、決定方程式は完全に解かれたことを意味します。対称性の生成元を得るには次のように入力します。

```
RESULT();
```

これで、次のような答えを得ることが出来ます。

```
The differential equation

DEQ(1):=U(1,2,2) + U(1,1)


The symmetry generators are

GEN(1):= DX(1)

GEN(2):= DX(2)

GEN(3):= 2*DX(2)*X(1) + DU(1)*U(1)*X(2)

GEN(4):= DU(1)*U(1)

GEN(5):= 2*DX(1)*X(1) + DX(2)*X(2)

                     2
GEN(6):= 4*DX(1)*X(1)

       + 4*DX(2)*X(2)*X(1)

                          2
         + DU(1)*U(1)*(X(2)  - 2*X(1))

GEN(7):= DU(1)*C(1)

The remaining dependencies

C(1) depends on X(2),X(1)


Constraints

DF(C(1),X(1)):= - DF(C(1),X(2),2)


The non-vanishing commutators of the finite subgroup
```

```
COMM(1,3):= 2*DX(2)

COMM(1,5):= 2*DX(1)

COMM(1,6):= 8*DX(1)*X(1) + 4*DX(2)*X(2) - 2*DU(1)*U(1)

COMM(2,3):= DU(1)*U(1)

COMM(2,5):= DX(2)

COMM(2,6):= 4*DX(2)*X(1) + 2*DU(1)*U(1)*X(2)

COMM(3,5):=  - (2*DX(2)*X(1) + DU(1)*U(1)*X(2))

                        2
COMM(5,6):= 8*DX(1)*X(1)

          + 8*DX(2)*X(2)*X(1)

                               2
          + 2*DU(1)*U(1)*(X(2)  - 2*X(1))
```

対称性の生成元の後に出力される “制約条件” のメッセージに続く出力は、関数 C(1) は x(1) と x(2) の関数で、かつ拡散方程式を満たすような関数で無ければならないことを意味します。

もっと他の例は後の節に挙げています。

もし利用者が与えられた微分方程式の対称性の生成元に対して、ある仮定を確かめたい場合、次のようにします。まず、これまでに説明したように決定方程式を構成します。ついで生成元に対して適当な代入を行い、PRSYS() を呼び出します。この仮定のもとでの決定方程式が出力されます。例えば、偏微分方程式系としては上と同じく拡散方程式を考え、いま決定方程式が求められているとします。上の例で求めた生成元 GEN 3 が正しいことを調べるには、次の代入を行います。

```
XI(1):=0;  XI(2):=2*X(1);  ETA(1):=X(2)*U(1);
```

この後、PRSYS() を呼び出してみると、全ての生成元 GL(k) が 0 になっているので、仮定が正しいことが示せます。

指定した NN と MM の値に対して、関数 ZETA の値が知りたい場合があります。これには、まず NN,MM に値を代入しておきそれから ZETA を呼び出すことで出来ます。例えば、

```
MM:=1;  NN:=2;

FACTOR U(1,2),U(1,1),U(1,1,2),U(1,1,1);

ON LIST;
```

```
ZETA(1,1);

-U(1,2)*U(1,1)*DF(XI(2),U(1))

-U(1,2)*DF(XI(2),X(1))

       2
-U(1,1) *DF(XI(1),U(1))

+U(1,1)*(DF(ETA(1),U(1)) -DF(XI(1),X(1)))

+DF(ETA(1),X(1))


ZETA(1,1,1);

-2*U(1,1,2)*U(1,1)*DF(XI(2),U(1))

-2*U(1,1,2)*DF(XI(2),X(1))

-U(1,1,1)*U(1,2)*DF(XI(2),U(1))

-3*U(1,1,1)*U(1,1)*DF(XI(1),U(1))

+U(1,1,1)*(DF(ETA(1),U(1)) -2*DF(XI(1),X(1)))

              2
-U(1,2)*U(1,1) *DF(XI(2),U(1),2)

-2*U(1,2)*U(1,1)*DF(XI(2),U(1),X(1))

-U(1,2)*DF(XI(2),X(1),2)

       3
-U(1,1) *DF(XI(1),U(1),2)

       2
+U(1,1) *(DF(ETA(1),U(1),2) -2*DF(XI(1),U(1),X(1)))

+U(1,1)*(2*DF(ETA(1),U(1),X(1)) -DF(XI(1),X(1),2))

+DF(ETA(1),X(1),2)
```

もし変数 NN または MM に値を設定していないときには、エラーメッセージ、

```
***** Number of variables not defined
(***** 変数の数が定義されていません。)
```

が出力されます。

引数 ETA(ALFA) と XI(K) の特別な値での関数 ZETA の値が知りたい場合があります。これには、一度別の変数に値を代入しておき、その後その変数に対して値を代入することで関数 ZETA の値が得られます。例えば、

```
Z11:=ZETA(1,1)$   Z111:=ZETA(1,1,1)$
```

ついで XI(1),XI(2),ETA(1) に値を代入します。

```
XI 1:=4*X(1)**2; XI 2:=4*X(2)*X(1);

ETA 1:=U(1)*(X(2)**2  - 2*X(1));
```

これらは前の例で求めた生成元 GEN(6) に対応しています。これから、求めるべき方程式は次のようにして得ることが出来ます。

```
Z11;

                                  2
 - (4*U(1,2)*X(2) - U(1,1)*X(2)  + 10*U(1,1)*X(1) + 2*U(1))

Z111;

                                      2
 - (8*U(1,1,2)*X(2) - U(1,1,1)*X(2)  + 18*U(1,1,1)*X(1) +
   12*U(1,1))
```

50.3 例題

いくつかの例題をあげて置きます。ここではいくつかの方程式や方程式系の対称群を決定しています。変数 PCLASS の値は既定値では 0 ですが、これは利用者が計算を始める前に変更することが出来ます。実行した結果を参考文献の結果と比べてみて下さい。

%Burgers 方程式

```
deq 1:=u(1,1)+u 1*u(1,2)+u(1,2,2)$
```

```
cresys deq 1$ simpsys()$ result()$

%Kadomtsev-Petviashvili 方程式

deq 1:=3*u(1,3,3)+u(1,2,2,2,2)+6*u(1,2,2)*u 1

       +6*u(1,2)**2+4*u(1,1,2)$

cresys deq 1$ simpsys()$ result()$

%修正された Kadomtsev-Petviashvili 方程式

deq 1:=u(1,1,2)-u(1,2,2,2,2)-3*u(1,3,3)

       +6*u(1,2)**2*u(1,2,2)+6*u(1,3)*u(1,2,2)$

cresys deq 1$ simpsys()$ result()$

%非線形 Schroedinger 方程式の実部と虚部

deq 1:= u(1,1)+u(2,2,2)+2*u 1**2*u 2+2*u 2**3$

deq 2:=-u(2,1)+u(1,2,2)+2*u 1*u 2**2+2*u 1**3$

%これは単一の方程式ではないので、代入文

sder 1:=u(2,2,2)$  sder 2:=u(1,2,2)$

%が必要です。

cresys()$ simpsys()$ result()$

%四つの方程式からなるシステムの対称性

deq 1:=u(1,1)+u 1*u(1,2)+u(1,2,2)$

deq 2:=u(2,1)+u(2,2,2)$

deq 3:=u 1*u 2-2*u(2,2)$

deq 4:=4*u(2,1)+u 2*(u 1**2+2*u(1,2))$

sder 1:=u(1,2,2)$ sder 2:=u(2,2,2)$ sder 3:=u(2,2)$
sder 4:=u(2,1)$
```

%は次のように解くことが出来ます。

```
cresys()$ simpsys()$

df(c 5,x 1):=-df(c 5,x 2,2)$

df(c 5,x 2,x 1):=-df(c 5,x 2,3)$

simpsys()$  result()$
```

% 方程式 1 と 3 からなる%comprising
%部分システムの対称性は次のようにして求められます。

```
cresys(deq 1,deq 3)$ simpsys()$ result()$
```

```
% 全ての可能な部分システム全部の結果は、
% 論文 ‘‘Symmetries and Involution Systems: Some
% Experiments in Computer Algebra’’, contribution to the
% Proceedings of the Oberwolfach Meeting on Nonlinear
% Evolution Equations, Summer 1986
% に詳しく説明してあります。
```

第51章　SPECFN: 特殊関数のパッケージ

Chris Cannam, et. al.

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin – Dahlem
Federal Republic of Germany
E–mail: *neun@zib.de*
Version 2.5, October 1998

51.1　はじめに

このパッケージは、よく使われる特殊関数に対する演算を定義したものです。関数名やインプリメントの詳細については、このドキュメントに記述しています。

膨大な数の特殊関数があるので、特殊関数に対するパッケージは完全なものが出来上がるということは決してありえません。何人かのユーザから、このパッケージの最初の版には、いくつかの重要な関数が含まれていないという指摘を受けました。このようなコメントやヒントは非常に助けになりました。

このパッケージの初版は 1992-1993 年に交換留学生として ZIB ベルリンにいっている間に作成したものです。それ以後、このパッケージは現在 ZIB ベルリンで保守されています。従って、このパッケージに関する質問やバグの報告等は `neun@ZIB.de` に行ってください。REDUCE の 3.5 版と共に配布されてから以後、多くの寄与を統合しています。

このパッケージは、以下にあげたよく知られている特殊関数に対する代数操作や数値計算を行えるように設計されています。

- ベルヌーイ数;
- オイラー数;
- フィボナッチ数と多項式;
- スターリング数;
- 二項係数;
- Pochhammer 記号;
- Gamma 関数;

- Psi 関数 とその導関数;
- リーマンの Zeta 関数;
- ベッセル関数 J と 一種および二種の Y;
- 変形されたベッセル関数 I と K;
- Hankel 関数 H1 と H2;
- Kummer の超幾何関数 M と U;
- Beta 関数と Struve, Lommel および Whittaker 関数;
- Airy 関数;
- 指数積分、正弦積分と余弦積分;
- 双曲正弦積分と余弦積分;
- フレネル積分と誤差関数;
- Dilog 関数;
- ポリ対数関数と Lerch の Φ 関数;
- エルミート多項式;
- ヤコビ多項式;
- ルジャンドル多項式;
- 随伴ルジャンドル多項式 (球関数、Solid 調和関数);
- ラゲール多項式;
- チェビシェフ多項式;
- ゲーゲンバウアー多項式;
- Lambert の ω 関数;
- Jacobi の楕円関数;
- 楕円積分;
- 3j と 6j シンボル、クレブッシュ・ゴルドン係数;
- いくつかの有名な定数.

ソースプログラム中のプログラムで、特に明記されていないものは、Dover の数学関数ハンドブック [1] に載っている式や数列を参考に作成しています。

ファイル $reduce/plot/specplot.tst には、特殊関数の作図 (plot) 呼び出しの素晴らしいコレクションが入っています。これは [1] の本にある有名なグラフを描きます。

51.2 以前の版との互換性

CSL 版では、このパッケージは REDUCE3.5 以降の版で含まれている新しい bigfloat パッケージと一緒に使うように最適化されています。以前の版の bigfloat でも動きますが、新しい版へ最適化されている為、古い版では効率的ではありません。

51.3 簡約化と近似

このパッケージで出来るのは代数的な簡約規則の適用によって式を簡単化していくことで、これは特殊な場合の取り扱いや極、微分操作等の処理を含んでいます。もし、ROUNDED スイッチがオンであれば、数値計算が行われます。近似計算モード (ROUNDED モード) では、特殊関数の引数が実数や復素数値の場合で、値の計算が可能であれば、現在設定されている有効桁数まで数値的な近似値が計算されます。

特殊関数のパッケージでは、ほとんどの代数式の簡約は、REDUCE のルールセットの形式で定義されています。従って、簡約規則の詳細について簡単に調べるために、ShowRules 演算子を使うことが出来ます。

```
ShowRules BesselI;

                          1          ~z     - ~z
{besseli(~n,~z) => ---------------*(e   - e     )
                   sqrt(pi*2*~z)

                           1
  when numberp(~n) and ~n=---,
                           2

                          1          ~z     - ~z
 besseli(~n,~z) => ---------------*(e   + e     )
                   sqrt(pi*2*~z)

                             1
  when numberp(~n) and ~n= - ---,
                             2

 besseli(~n,~z) => 0

  when numberp(~z) and ~z=0 and numberp(~n) and ~n neq 0,

 besseli(~n,~z) => besseli( - ~n,~z) when numberp(~n)
```

```
 and impart(~n)=0 and ~n=floor(~n) and ~n<0,

besseli(~n,~z) => do*i(~n,~z)

 when numberp(~n) and numberp(~z) and *rounded,

df(besseli(~n,~z),~z)

     besseli(~n - 1,~z) + besseli(~n + 1,~z)
 => ----------------------------------------,
                        2

df(besseli(~n,~z),~z)

 => besseli(1,~z) when numberp(~n) and ~n=0}
```

いくつかの、REDUCE パッケージ (Sum や Limits 等) は SPECFN をロードした時には、異なった (多分、よりよい) 簡約結果を返します。これは、SPECFN パッケージには、他のパッケージから直接利用可能な、また有用な規則が定義されているからです。例えば、

```
sum(1/k^s,k,1,infinity); % を評価すると

zeta(s)
```

以前に計算された関数の近似値についてはすべて記録されており、再度同じ引数に対する関数値を求めるときには、以前求めた値の精度が今回求めようとする精度と同じかそれより高い場合には、以前に求めた値が使用されます。これは記憶領域を消費します。特に多くの点での関数値を計算し、それらが再び計算されることがないような場合には、無駄に消費されることになります。このような場合には `savefs` スイッチをオフにすることで近似値の記録を禁止することが出来ます。このスイッチは通常オンになっています。

51.4 定数

いくつかのよく知られた定数が定義されています。これらの定数についての、定数を定義するのにも使われている、重要な性質もまた定義してあります。ROUNDED スイッチがオンであれば、任意の精度での数値が求められます。

- Euler_Gamma : オイラー数, -$\psi(1)$;
- Catalan : カタラン数;
- Khinchin : Khinchin 数, [30] で定義されています。(計算には時間がかかります。);

- Golden_Ratio : $\frac{1+\sqrt{5}}{2}$

単項演算子 Bernoulli はベルヌーイ数を求めます。Bernoulli(n) は n 番目のベルヌーイ数 (Bernoulli(1) 以外は奇数のベルヌーイ数) を計算します。

計算の方法は Herbert Wilf によるもので、Sandra Fillebrown [24] によって与えられたものです。もし ROUNDED スイッチがオフの場合にはこの算法とまったく同じです。もしこれがオンの時はさらに近似を行っており、無駄な計算を行わないようにしています。そのため、ベルヌーイ数の近似値を求める場合には特に計算が早くなっています。

オイラー数は Euler 演算子で求められます。この計算は二項係数に対するパスカルの三角形より直接おこなっています。

51.5 フィボナッチ数とフィボナッチ多項式

Fibonacci(n) は n 番めのフィボナッチ数を与えます。これは、もし n が正または負の整数であれば、次の式で計算される値に等しくなります。

$$F_0 = 0; F_1 = 1; F_n = F_{n-1} + F_{n-2}$$

フィボナッチ多項式は FibonacciP で表されます。FibonacciP(n, x) は変数 x に関する n 番目のフィボナッチ多項式を返します。これは、n が正または負の整数に対して以下の式で定義された多項式です。

$$F_0(x) = 0; F_1(x) = 1; F_n(x) = xF_{n-1}(x) + F_{n-2}(x)$$

51.6 スターリング数

一種と二種のスターリング数の計算は Stirline1 と Stirling2 演算子で求められます。

これは公式から直接求めています。

51.7 Γ 関数および関連する関数

51.7.1 Γ 関数

この関数は単項演算子 Gamma で表されます。

ROUNDED スイッチがオフの状態では整数 n に対して、$\Gamma(n)$、$\Gamma(n+1/2)$、自然数 n と 64 以下の 2 のべき乗となる m に対して $\Gamma(n+1/m)$ は $\Gamma(1/m)(m=2$ の時には $\sqrt{\pi}$ で表される) で表されます。極となる引数に対しては INFINITY を返します。また、負の引数に対しては引数が正である Γ 関数で表されます。

数値的な近似値の計算には $\ln(\Gamma)$ に対する漸近展開式と Pochhammer 関数から得られるスケール因子を使っています。

$\Gamma'(z)$ を Γ と ψ によって表す変換式もこのパッケージに含まれています。

51.7.2 Pochhammer 記号

Pochhammer 記号 $(a)_k$ は `Pochhammer(a,k)` 演算子で表されます。`ROUNDED` モードがオフで、かつ引数 a と k が共に整数の場合には、その値が計算されます。それ以外の場合には可能な限り簡約が行われます。簡約の規則は Wolfram Koepf [31] による算法を元にしています。

51.7.3 Digamma 関数, ψ

この関数は単項演算子 `Psi` で表されます。

ψ に対する変換規則は Γ 関数とほとんど同じです。つまり、$\psi(x)$ は可能であれば $\psi(1)$ および $\psi(\frac{1}{2})$ で表されます。また負の引数に対しては正のもので書き換えられます。

ψ の数値計算は引数が実数の場合のみ行われます。計算の方法は、適当なスケール因子と漸近展開を使って行っています。

ψ の微分及び積分に関する関係式も定義しています。

51.7.4 Polygamma 関数, $\psi^{(n)}$

ψ 関数 の n 階導関数は二項演算子 `Polygamma` で表されます。この演算子の第一引数は n です。

`Polygamma(n,1)` および `Polygamma(n,3/2)` は Riemann の ζ 関数で書き直されます。また第一引数が零の場合は ψ で表されます。極も処理されます。

数値近似の計算は引数が実数の場合のみ行われます。計算方法は漸近展開を使った方法です。負の引数に対しては、$\cot(\pi z)$ の n 階導関数を使った反射公式が使われます。

微分および積分に対する単純な関係式は処理されます。

51.7.5 Riemann の ζ 関数

この関数は単項演算子 `Zeta` で表されます。

`ROUNDED` がオフでの時、$\zeta(z)$ は $-31 < z < 31$ の範囲の偶整数もしくは、$-30 < z < 16$ の範囲の奇整数に対しては数値的に計算します。それ以外の値に対しては、結果は多少扱いにくい形になります。

ζ 関数の数値計算は引数が実数の場合のみ行われます。計算方法は奇整数に対しては $\zeta(n)$ を $\psi^{n-1}(3)$ で表す公式を使って、また偶整数に対してはベルヌーイ数との関係式を使って行われます。それ以外の引数に対しては、大きな値に対しては、まず通常の素数展開から最初の数個の素

数の項を取り、それからこの素数では割り切れない自然数による級数を使います。また小さな値に対しては文献 [9] による級数を使っています。

ζ 関数の微分および積分は扱えません。

51.8 ベッセル関数

ベッセル 関数 J と Y、変形されたベッセル 関数 I と K、それに一種と二種の Hankel 関数が扱えます。演算子はそれぞれ、`BesselJ`、`BesselY`、`BesselI`、`BesselK`、`Hankel1` と `Hankel2` で、これらはすべて二項演算子として定義されています。

これらの関数に対しては次の演算が可能です。

- J, Y, I と K の自明な極;
- 負の第一引数に対する J, Y, I と K 関数の処理;
- 負の第二引数に対する J 関数の処理;
- 非整数または復素数の第二引数の Y または K 関数を J または I 関数で表すこと;
- J, Y と I に対する微分;
- 第一引数が 0 である K 関数の微分;
- Hankel 関数 の微分.

さらに、もし `COMPLEX` スイッチがオンで `ROUNDED` がオフの場合、Hankel 関数を含む式はベッセル関数で書き直します。

ベッセル K 関数に対する数値近似、および特別な場合の除いた Hankel 関数の数値計算は行えません。それ以外のベッセル関数の数値近似は通常の昇べき級数展開を、また J と Y については漸近展開を使っています。通常漸近展開式をまず最初に試し、もし引数が非常に小さくて必要な計算精度が得られない場合には、級数展開を使って計算します。また明らかに漸近展開が使えないような場合の判定も行っています。

ベッセルおよび Hankel 関数に対する積分は定義されていません。

51.9 超幾何関数およびその他の関数

このパッケージでは、その他の関数に対する以下のような演算も可能です。

- Beta 関数, これは Γ 関数 [1] の変形です。これは二項演算子 `Beta` で表されます;
- Struve の **H** および **L** 関数、これは二項演算子 `StruveH` と `SturveL` で表されます。特別な場合の処理と可能な場合には、より扱い易い関数への変換が行われます。また第二引数に対する微分計算が行えるようになります。

- 一種および二種の Lommel 関数、これは演算子 `Lomel1` と `Lommel2` で表されます。特別な場合の処理と可能な簡約が行われます。
- Kummer の超幾何関数 M と U (それぞれ超幾何関数 ${}_1F_1$ または Φ、そして $z^{-a}{}_2F_0$ または Ψ) 、これは演算子 `KummerM` と `KummerU` で表されます。特別な場合の処理と微分それに M 関数に対しては実数の引数に対する数値近似の計算が可能です。
- Whittaker の M および W 関数、これは Kummer 関数の変形です。二項演算子 `WhittakerM` と `WhittakerW` で表されます。これらの関数については、Kummer 関数で書き換えます。

51.10 積分関数

SPECFN パッケージはある種の積分関数、つまり、

erf, erfc, Si, Shi, si, Ci, Chi, Ei, li, Fresnel_C および Fresnel_S.

の代数操作および限られた条件での数値計算が出来ます。

積分の定義から、微分操作やある種の極限値、対称性のような単純な性質は定義してあります。

積分関数の近似値の計算では引数が約 10.0 以上では、計算精度が正しく制御されていないことと、大きな引数に対しても和公式を使っていることにより、指定した精度の結果が得られません。

li は $Ei(ln(z))$ と簡約されます。

51.11 Airy 関数

Airy 関数 Ai と Bi、Airyprime 関数 $Aiprime$ と $Biprime$ 関数が定義されています。演算子名はそれぞれ、`Airy_Ai`, `Airy_Bi`, `Airy_Aiprime`, および `Airy_Biprime` です。これらはすべて一引数の演算子です。

次の場合の処理が可能です:

- Airy_Ai と Airy_Bi の自明な場合、およびそれらの素数の場合が計算できます。
- 複素数および実数の引数の場合の計算が可能です。
- 活性化されていないルールセットを有効にすることによって、Airy 関数を Bessel 関数で表すことができます。

Airy 関数を Bessel 関数で表すルールを有効にするには、`let Airy2Bessel_rules;` と入力します。この結果、Airy_Ai 関数は、例えば、次の式で計算できます。

`Ai(z) =` $\frac{1}{3}\sqrt{z}[I_{-1/3}(\zeta)$ - $I_{1/3}(\zeta)]$, ただし $\zeta = \frac{2}{3}z^{\frac{2}{3}}$

注意:- 近似値を求めるには、`COMPLEX` と `ROUNDED` の両方のスイッチをオンにします。

Airy 関数の計算のアルゴリズムは、通常の級数展開と、漸近展開を使っています。引数の値によっては、級数展開よりも漸近展開を使った方が良い結果が得られます。この境界点の値は、求める精度によって変わり、プログラムで計算して求めています。

Airy 関数に関する積分規則は定義されていません。

51.12 多項式関数

二つのグループが定義されています。よく知られた直交多項式系 (Hermite, Jacobi, Legendre, Laguerre, Chebyshev, Gegenbauer) とオイラー多項式、ベルヌーイ多項式です。REDUCE の関数名は、多項式の名前の最後に P を付けた名前をつけています。例えば、オイラー多項式は EulerP という名前になっています。大部分の定義は文献 [1] と同じです。ただし (随伴)Legendre 多項式については違っています。

```
P(n,m,x) = (-1)^m *(1-x^2)^(m/2)*df(legendreP (n,x),x,m)
```

51.13 球調和関数と Solid 調和関数

演算子名は、SolidHarmonicY と SphericalHarmonicY です。

SolidHarmonicY 演算子は、以下で説明している Solid 調和関数をインプリメントしています。これは、$n, m, x, y, z, r2$ の六変数の関数で、$x, y, z, r2$ の多項式を返します。

SphericalHarmonicY は SolidHarmonicY の特別な場合の関数で、次のように定義されます。

```
algebraic procedure SphericalHarmonicY(n,m,theta,phi);
        SolidHarmonicY(n,m,sin(theta)*cos(phi),
                sin(theta)*sin(phi),cos(theta),1)$
```

次数 n の Solid 調和関数 (ラプラス多項式) は、x, y, z の次数 n の同次多項式です。これは、ラプラス方程式

```
df(P,x,2) + df(P,y,2) + df(P,z,2) = 0.
```

の解として与えられます。

与えられた $n >= 0$ に対して、$2n + 1$ 個の独立な多項式が存在します。

```
w!0 = z, w!+ = i*(x-i*y)/2, w!- = i*(x+i*y)/2,
```

これらは、フーリエ積分で与えられます。

```
S(n,m,w!-,w!0,w!+) =

      (1/(2*pi)) *
      for u:=-pi:pi integrate (w!0 + w!+ * exp(i*u) + w!- *
          exp(-i*u))^n * exp(i*m*u) * du;
```

これは、明らかに $|m| > n$ でゼロです。なぜなら、被積分関数を展開した式は、e^{iku}、$k \neq 0$ という因子を持っているからです。

$S(n, m, x, y, z)$ は

```
   r^n * Legendre(n,m,cos theta) * exp(i*phi)
```

に比例します。

今、$r2 = x^2 + y^2 + z^2$ および $r = \sqrt{r2}$ とします。

球調和関数は、Solid 調和関数を単に単位球面上に制限したものです。すべての球調和関数集合 $n >= 0; -n <= m =< n$ は完全直交基底を構成します。つまり、 $< n, m|n', m' > =$ Kronecker_delta$(n, n') \times$ Kronecker_delta(m, m') が成立します。$< ...|... >$ は、球面上での関数のスカラー積を表します。

Solid 調和関数の係数は、球調和関数が正規直交系となるように規格化されています。

多項式の性質から、Solid 調和関数に対して、多くの再帰公式が得られています。また、Legendre 関数に関する再帰公式を Solid 調和関数に “変換” した式も作ることが出来ます。しかしながら、ラプラスの定義から直接導き出すのが最も簡単です。また、Solid 調和関数の微分は、多項式の微分として、ほとんど自明な計算です。

$r2$ (r は x, y, z の有理式ではないので、現れません) に関する再帰公式を使うことで、大幅な簡約が可能です。形式的には、Solid 調和関数は、$(x, y, z, r2)$ に関する多項式としてよりコンパクトな形に表現できます。

再帰は、次の二つが必要です:-

(i) 対角方向に沿って:(n, n);

(ii) 定数 n の線に沿って: $(m, m), (m + 1, m), \ldots, (n, m)$.

これらの再帰は、数値的に安定です。

$m < 0$ に対して、次の式が成り立ちます。

```
   S(n,m,x,y,z) = (-1)^m * S(n,-m,x,-y,z);
```

51.14 Jacobi の楕円関数

次のような関数が定義されています。

- 12 個の Jacobi 関数

- 代数的幾何平均
- 下降 Landen 変換

51.14.1 Jacobi 関数

次のような Jacobi 関数が定義されています。

- Jacobisn(u,m)
- Jacobidn(u,m)
- Jacobicn(u,m)
- Jacobicd(u,m)
- Jacobisd(u,m)
- Jacobind(u,m)
- Jacobidc(u,m)
- Jacobinc(u,m)
- Jacobisc(u,m)
- Jacobins(u,m)
- Jacobids(u,m)
- Jacobics(u,m)

これらは、ROUNDED スイッチがオンの場合、数値的に計算されます。

51.14.2 振幅

u の振幅は、`JacobiAmplitude(u,m)` コマンドで求められます。

51.14.3 代数的幾何平均

初期値 a_0, b_0, c_0 での、AGM(Algebraic Geometric Mean) を求めるには、

`AGM_function(`a_0, b_0, c_0`)` と入力します。結果は、
$\{N, AGM, \{a_N, \ldots, a_0\}, \{b_N, \ldots, b_0\}, \{c_N, \ldots, c_0\}\}$, となります。ここで、N は AGM を要求する精度で計算するのに必要なステップ数です。

楕円積分 K(m), E(m) を計算するとき、初期値として $a_0 = 1$; $b_0 = \sqrt{1-m}$; $c_0 = \sqrt{m}$ を使っています。

51.14.4 下降 Landen 変換

phi と alph の下降 Landen 変換を求める関数は、次の式を使っています。

$(1+\sin\alpha_{n+1})(1+\cos\alpha_n)=2$ ただし $\alpha_{n+1}<\alpha_n$

$\tan(\phi_{n+1}-\phi_n)=\cos\alpha_n\tan\phi_n$ ただし $\phi_{n+1}>\phi_n$

`landentrans(`ϕ_0`,`α_0`)` のように呼び出すと、次の結果が得られます。
$\{\{\phi_0,\ldots,\phi_n\},\{\alpha_0,\ldots,\alpha_n\}\}$。

51.15 楕円積分

次の関数が定義されています。

- 第一種の楕円積分
- 第二種の楕円積分
- Jacobi の θ 関数
- Jacobi の ζ 関数

51.15.1 楕円関数 F

楕円関数 F は、 `EllipticF(`ϕ`,m)` で、計算出来ます。これは、 第一種の楕円積分 の値を返します。

51.15.2 楕円関数 K

楕円関数 K は `EllipticK(m)` で計算できます。第一種の完全楕円積分 の値を返します。これは、しばしば他の楕円関数の値を計算するのに用いられます。

51.15.3 楕円関数 K′

楕円関数 K′ は、`EllipticK!'(m)` で、計算できます。これは、 $\mathrm{K}(1-m)$ の値を返します。

51.15.4 楕円関数 E

楕円関数 E は二つの関数が定義されています。

二引数の関数として、`EllipticE(`ϕ`,m)` は、第二種の楕円積分 を返します。

`EllipticE(m)` は、第二種の完全楕円積分E を返します。

51.15.5 楕円 Θ 関数

これは、EllipticTheta(a,u,m) として計算されます。ここで、a は、θ 関数の添字 ($a = 1, 2, 3$ または 4) です。この関数は、H; H_1; Θ_1; Θ を返します。(テキストによっては、ϑ_1; ϑ_2; ϑ_3; ϑ_4 と表されています。)

51.15.6 Jacobi の Zeta 関数 Z

これは、JacobiZeta(u,m) のように使い、Jacobi の Zeta 関数を求めます。注意: Zeta 演算子は、Riemann の ζ 関数を計算します。

51.16 Lambert の W 関数

Lambert の W 関数は、関数 we^w の逆関数です。このため、Solve パッケージと一緒に使ったとき、特に有用な機能を与えます。この関数は、[39] で詳しく調べられています。現在のインプリメントでは、ROUNDED モードのもとで、主値に対する計算のみが可能です。

51.17 3j シンボルと Clebsch-Gordan 係数

演算子 ThreeJSymbol, Clebsch_Gordan は、[34] または [21] に基づいて定義されています。してあります。ThreeJSymbol は、引数として、三つの値リスト $\{j_i, m_i\}$ を取ります。

```
ThreeJSymbol({J+1,M},{J,-M},{1,0});
Clebsch_Gordan({2,0},{2,0},{2,0});
```

51.18 6j シンボル

演算子 SixJSymbol を、[34] または [21] に従って定義している。SixJSymbol は二つの値のリスト $\{j_1, j_2, j_3\}$ と $\{l_1, l_2, l_3\}$ を引数に取る。

```
SixJSymbol({7,6,3},{2,4,6});
```

現在の SixJSymbol のインプリメントでは、INEQ パッケージを使った和に関する最小値、最大値が、非常に制限された場合しか求められないため、特別な 6j-シンボル (例えば [34] を見よ) はたいていの場合に、見つけることが出来ません。

51.19 謝辞

Kerry Gaskell, Matthew Rebbeck, Lisa Temme, Stephen Scowcroft および David Hobbs(すべて Bath 大学から ZIB への 1 年間の留学生) による寄与は非常にこのパッケージの augment にたった。

Clebsch-Gordan と 3j, 6j シンボルのモジュールおよび球調和関数と Solid 調和関数についての、René Grognard (CSIRO , Australia) の助言は、非常にありがたかった。

51.20 演算子の表

関数			演算子
$\binom{n}{m}$	$\equiv$	$\frac{n!}{m!(n-m)!}$	`Binomial(n,m)`
B_n		$\frac{te^t}{e^t-1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!}$	`Bernoulli(n)`
E_n		$\frac{2e^t}{e^t+1} = \sum_{n=0}^{\infty} E_n \frac{t^n}{n!}$	`Euler(n)`
$S_n^{(m)}$		$x(x-1)\cdots(x-n+1) = \sum_{m=0}^{n} S_n^{(m)} x^m$	`Stirling1(n,m)`
$\mathbf{S}_n^{(m)}$	$\equiv$	$\frac{1}{m!}\sum_{k=0}^{m}(-1)^{m-k}\binom{m}{k}k^n$	`Stirling2(n,m)`
$\Gamma(z)$	$\equiv$	$\int_0^{\infty} t^{z-1}e^{-t}dt$	`Gamma(z)`
$(a)_k$	$\equiv$	$a(a+1)\cdots(a+k-1) = \frac{\Gamma(a+k)}{\Gamma(a)}$	`Pochhammer(a,k)`
$\psi(z)$	$\equiv$	$d\log\Gamma(z)/dx$	`Psi(z)`
$\psi^{(n)}(z)$	$\equiv$	$d^n\log\Gamma(z)/dx^n$	`Polygamma(n,z)`
$\zeta(z)$	$\equiv$	$\sum_{k=1}^{\infty} k^{-n}$	`Zeta(z)`
$J_\nu(z)$	$\equiv$	$\frac{z^\nu}{2^\nu}\sum_{k=0}^{\infty}(-1)^k\frac{z^{2k}}{2^{2k}k!\Gamma(\nu+k+1)}$	`BesselJ(n,z)`
$Y_\nu(z) or N_\nu(z)$	$\equiv$	$\frac{1}{\sin\nu\pi}[\cos\nu\pi J_\nu(z) - J_{-\nu}(z)]$	`BesselY(n,z)`
$I_\nu(z)$	$\equiv$	$e^{-\frac{\pi}{2}\nu i}J_\nu(e^{\frac{\pi}{2}i}z) \quad or \quad e^{\frac{3}{2}\pi\nu i}J_\nu(e^{-\frac{3}{2}\pi i}z)$	`BesselI(n,z)`
$K_\nu(z)$	$\equiv$	$\frac{\pi i}{2}e^{\frac{\pi}{2}\nu i}H_\nu^{(1)}(iz)$	`BesselK(n,z)`
$H_\nu^{(1)}(z)$	$\equiv$	$J_\nu(z) + iY_\nu(z)$	`Hankel1(n,z)`
$H_\nu^{(2)}(z)$	$\equiv$	$J_\nu(z) - iY_\nu(z)$	`Hankel2(n,z)`
$B(z,w)$	$\equiv$	$\int_0^1 t^{z-1}(1-t)^{w-1}dt = \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$	`BETA(z,w)`
$\mathbf{H}_\nu(z)$	$\equiv$	$(\frac{1}{2}z)^{\nu+1}\sum_{k=0}^{\infty}\frac{(-1)^k(\frac{1}{2}z)^{2k}}{\Gamma(k+\frac{3}{2})\Gamma(k+\nu+\frac{3}{2}}$	`StruveH(n,z)`
$\mathbf{L}_\nu(z)$	$\equiv -ie^{-\frac{i\nu\pi}{2}}\mathbf{H}_\nu(iz) =$	$(\frac{1}{2}z)^{\nu+1}\sum_{k=0}^{\infty}\frac{(\frac{1}{2}z)^{2k}}{\Gamma(k+\frac{3}{2})\Gamma(k+\nu+\frac{3}{2})}$	`StruveL(n,z)`
$s_{a,b}(z)$	$\equiv$	$\sum_{m=0}^{\infty}\frac{(-1)^m z^{a+1-2m}}{[(a+1)^2-b^2][(a+3)^2-b^2]\cdots[(a_2m+1)^2-b^2]}$	`Lommel1(a,b,z)`
$S_{a,b}(z)$	$\equiv$	$s_{a,b}(z) + 2^{a-1}\Gamma(\frac{a-b+1}{2})\Gamma(\frac{a+b+1}{2})\times$	`Lommel2(a,b,z)`
		$\{\sin\frac{(a-b)\pi}{2}J_b(z) - \cos\frac{(a-b)\pi}{2}Y_b(z)\}$	

関数			演算子
$Ai(z)$	$\equiv$	$\frac{1}{3}\sqrt{z}[I_{-1/3}\ (\frac{2}{3}z^{\frac{2}{3}})$ - $I_{1/3}(\frac{2}{3}z^{\frac{2}{3}})]$	`Airy_Ai(z)`
$Bi(z)$			`Airy_Bi(z)`
$Ai'(z)$			`Airy_Aiprime(z)`
$Bi'(z)$			`Airy_Biprime(z)`
$M(a,b,z)$ or ${}_1F_1(a,b;z)$ or $\Phi(a,b;z)$	$\equiv$	$1+\frac{az}{b}+\frac{(a)_2}{(b)_2}\frac{z^2}{2!}+\cdots+\frac{(a)_n}{(b)_n}\frac{z^n}{n!}+\cdots$	`KummerM(a,b,z)`
$U(a,b,z)$ or $z^{-a}{}_2F_0(a,b;z)$ or $\Psi(a,b;z)$	$\equiv$	$\frac{\pi}{\sin\pi b}\{\frac{M(a,b,z)}{\Gamma(1+a-b)\Gamma(b)} -z^{1-b}\frac{M(1+a-b,2-b,z)}{\Gamma(a)\Gamma(2-b)}\}$	`KummerU(a,b,z)`
$M_{\kappa,\mu}(z)$	$\equiv$	$e^{-\frac{1}{2}z}z^{\frac{1}{2}+\mu}M(\frac{1}{2}+\mu-\kappa,1+2\mu,z)$	`WhittakerM(k,m,z)`
$W_{\kappa,\mu}(z)$	$\equiv$	$e^{-\frac{1}{2}z}z^{\frac{1}{2}+\mu}U(\frac{1}{2}+\mu-\kappa,1+2\mu,z)$	`WhittakerW(k,m,z)`
$B_n(x)$	$\equiv$	$\frac{te^{xt}}{e^t-1}=\sum_{n=0}^{\infty}B_n(x)\frac{t^n}{n!}$	`BernoulliP(n,x)`
$E_n(x)$	$\equiv$	$\frac{2e^{xt}}{e^t+1}=\sum_{n=0}^{\infty}E_n(x)\frac{t^n}{n!}$	`EulerP(n,x)`
$C_n^{(\alpha)}(x)$	$\equiv$	$\frac{1}{\Gamma(\alpha)}\sum_{m=0}^{[\frac{n}{2}]}(-1)^m\frac{\Gamma(\alpha+n-m)}{m!(n-2m)!}(2x)^{n-2m}$	`GegenbauerP(n,a,x)`
$H_n(x)$	$\equiv$	$n!\sum_{m=0}^{[\frac{n}{2}]}(-1)^m\frac{}{m!(n-2m)!}(2x)^{n-2m}$	`HermiteP(n,x)`
$L_n(x)$	$\equiv$	$\sum_{k=0}^{n}(-1)^k\binom{n}{k}\frac{1}{k!}x^k$	`LaguerreP(n,x)`
$L_n^{(m)}(x)$	$\equiv$	$\sum_{k=0}^{n}(-1)^k\binom{n+m}{n-k}\frac{1}{k!}x^k$	`LaguerreP(n,m,x)`
$P_n(x)$	$\equiv$	$\frac{1}{2^n}\sum_{m=0}^{[\frac{n}{2}]}(-1)^m\binom{n}{m}\binom{2n-2m}{n}x^{n-2m}$	`LegendreP(n,x)`
$P_n^{(m)}(x)$	$\equiv$	$(-1)^m(1-x^2)^{\frac{m}{2}}\frac{d^m}{dx^m}P_n(x)$	`LegendreP(n,m,x)`
$P_n^{(\alpha,\beta)}(x)$	$\equiv$	$\frac{1}{2^n}\sum_{m=0}^{n}\binom{n+\alpha}{m}\binom{n+\beta}{n-m}(x-1)^{n-m} \times(x+1)^m$	`JacobiP(n,a,b,x)`
$U_n(x)$	$\equiv$	$\sum_{m=0}^{[\frac{n}{2}]}(-1)^m\frac{(n-m)!}{m!(n-2m)!}(2x)^{n-2m}$	`ChebyshevU(n,x)`
$T_n(x)$	$\equiv$	$\frac{n}{2}\sum_{m=0}^{[\frac{n}{2}]}(-1)^m\frac{(n-m-1)!}{m!(n-2m)!}(2x)^{n-2m}$	`ChebyshevT(n,x)`

関数		演算子
$P_n^{(\alpha,\beta)}(x)$		`JacobiP(n,alpha,beta,x)`
$U_n(x)$		`ChebyshevU(n,x)`
$T_n(x)$		`ChebyshevT(n,x)`
$Y_n^m(x,y,z,r2)$		`SolidHarmonicY (n,m,x,y,z,r2)`
$Y_n^m(\theta,\phi)$		`SphericalHarmonicY (n,m,theta,phi)`
$\begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{pmatrix}$		`ThreeJSymbol ({j1,m1},{j2,m2},{j3,m3})`
$(j_1 m_1 j_2 m_2 \vert j_1 j_2 j_3 - m_3)$		`Clebsch_Gordan ({j1,m1},{j2,m2},{j3,m3})`
$\begin{Bmatrix} j_1 & j_2 & j_3 \\ l_1 & l_2 & l_3 \end{Bmatrix}$		`SixJSymbol ({j1,j2,j3},{l1,l2,l3})`
$Si(z)$	$\equiv \int_0^z \frac{\sin t}{t} dt$ $= \sum_{n=0}^{\infty} (-1)^n \frac{z^{2n+1}}{(2n+1)(2n+1)!}$	`Si(z)`
$si(z)$	$\equiv -\int_z^{\infty} \frac{\sin t}{t} dt$ $= -\frac{\pi}{2} + \sum_{n=1}^{\infty} (-1)^{n+1} \frac{z^{2n-1}}{(2n-1)(2n-1)!}$	`s_i(z)`
$Ci(z)$	$\equiv -\int_z^{\infty} \frac{\cos t}{t} dt$	`Ci(z)`
$Shi(z)$	$\equiv \int_0^z \frac{\sinh t}{t} dt$	`Shi(z)`
	$= \sum_{n=0}^{\infty} \frac{z^{2n+1}}{(2n+1)(2n+1)!}$	
$Chi(z)$	$\equiv \int_0^z \frac{\cosh t}{t} dt - \psi(1) + \log z$	`Chi(z)`
	$= \sum_{n=1}^{\infty} \frac{z^{2n}}{2n(2n)!} - \psi(1) + \log z$	
$\mathrm{erf}(z)$	$\equiv \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$ $= \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{z^{2n+1}}{n!(2n+1)}$	`erf(z)`
$\mathrm{erfc}(z)$	$\equiv 1 - \mathrm{erf} z$	`erfc(z)`
$Ei(z)$	$\equiv \int_{-\infty}^z \frac{e^t}{t} dt$ $= -\psi(1) + \log z + \sum_{n=1}^{\infty} (-1)^n \frac{z^n}{nn!}$	`Ei(z)`
$dilog(z)$	$\equiv \int_z^{\infty} \frac{\log t}{t-1} dt$	`dilog(z)`

関数		演算子
$C(x)$	$\equiv \int_0^z \cos(\frac{\pi}{2}t^2)dt$ $= \sum_{n=0}^{\infty}(-1)^n \frac{(\pi/2)^{2n}}{(2n)!(4n+1)} z^{4n+1}$	`Fresnel_C(x)`
$S(x)$	$\equiv \int_0^z \sin(\frac{\pi}{2}t^2)dt$ $= \sum_{n=0}^{\infty}(-1)^n \frac{(\pi/2)^{2n+1}}{(2n+1)!(4n+3)} z^{4n+3}$	`Fresnel_S(x)`
$sn(u\|m)$		`Jacobisn(u,m)`
$dn(u\|m)$		`Jacobidn(u,m)`
$cn(u\|m)$		`Jacobicn(u,m)`
$cd(u\|m)$		`Jacobicd(u,m)`
$sd(u\|m)$		`Jacobisd(u,m)`
$nd(u\|m)$		`Jacobind(u,m)`
$dc(u\|m)$		`Jacobidc(u,m)`
$nc(u\|m)$		`Jacobinc(u,m)`
$sc(u\|m)$		`Jacobisc(u,m)`
$ns(u\|m)$		`Jacobins(u,m)`
$ds(u\|m)$		`Jacobids(u,m)`
$cs(u\|m)$		`Jacobics(u,m)`
$F(\phi\|m)$		`EllipticF(phi,m)`
$K(m)$		`EllipticK(m)`
$E(\phi\|m)$		`EllipticE(phi,m)`
$E(m)$		`EllipticE(m)`
$H(u\|m)$, $H_1(u\|m)$, $\Theta_1(u\|m)$, $\Theta(u\|m)$		`EllipticTheta(a,u,m)`
$\theta_1(u\|m)$, $\theta_2(u\|m)$, $\theta_3(u\|m)$, $\theta_4(u\|m)$		`EllipticTheta(a,u,m)`
$Z(u\|m)$		`Zeta_function(u,m)`

51.21 演算子と定数の表

関数	演算子
$J_\nu(z)$	`BesselJ(nu,z)`
$Y_\nu(z)$	`BesselY(nu,z)`
$I_\nu(z)$	`BesselI(nu,z)`
$K_\nu(z)$	`BesselK(nu,z)`
$H_\nu^{(1)}(z)$	`Hankel1(n,z)`
$H_\nu^{(2)}(z)$	`Hankel2(n,z)`
$\mathbf{H}_\nu(z)$	`StruveH(nu,z)`
$\mathbf{L}_\nu(z)$	`StruveL(n,z)`
$s_{a,b}(z)$	`Lommel1(a,b,z)`
$S_{a,b}(z)$	`Lommel2(a,b,z)`
$Ai(z)$	`Airy_Ai(z)`
$Bi(z)$	`Airy_Bi(z)`
$Ai'(z)$	`Airy_Aiprime(z)`
$Bi'(z)$	`Airy_Biprime(z)`
$M(a,b,z)$ or ${}_1F_1(a,b;z)$ or $\Phi(a,b;z)$	`KummerM(a,b,z)`
$U(a,b,z)$ or $z^{-a}{}_2F_0(a,b;z)$ or $\Psi(a,b;z)$	`KummerU(a,b,z)`
$M_{\kappa,\mu}(z)$	`WhittakerM(kappa,mu,z)`
$W_{\kappa,\mu}(z)$	`WhittakerW(kappa,mu,z)`
Fibonacci 数 F_n	`Fibonacci(n)`
Fibonacci 多項式 $F_n(x)$	`FibonacciP(n)`
$B_n(x)$	`BernoulliP(n,x)`
$E_n(x)$	`EulerP(n,x)`
$C_n^{(\alpha)}(x)$	`GegenbauerP(n,alpha,x)`
$H_n(x)$	`HermiteP(n,x)`
$L_n(x)$	`LaguerreP(n,x)`
$L_n^{(m)}(x)$	`LaguerreP(n,m,x)`
$P_n(x)$	`LegendreP(n,x)`
$P_n^{(m)}(x)$	`LegendreP(n,m,x)`

関数	演算子
$P_n^{(\alpha,\beta)}(x)$	`JacobiP(n,alpha,beta,x)`
$U_n(x)$	`ChebyshevU(n,x)`
$T_n(x)$	`ChebyshevT(n,x)`
$Y_n^m(x,y,z,r2)$	`SolidHarmonicY(n,m,x,y,z,r2)`
$Y_n^m(\theta,\phi)$	`SphericalHarmonicY(n,m,theta,phi)`
$\begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{pmatrix}$	`ThreeJSymbol({j1,m1},{j2,m2},{j3,m3})`
$(j_1 m_1 j_2 m_2 \vert j_1 j_2 j_3 - m_3)$	`Clebsch_Gordan({j1,m1},{j2,m2},{j3,m3})`
$\begin{Bmatrix} j_1 & j_2 & j_3 \\ l_1 & l_2 & l_3 \end{Bmatrix}$	`SixJSymbol({j1,j2,j3},{l1,l2,l3})`
$sn(u\vert m)$	`Jacobisn(u,m)`
$dn(u\vert m)$	`Jacobidn(u,m)`
$cn(u\vert m)$	`Jacobicn(u,m)`
$cd(u\vert m)$	`Jacobicd(u,m)`
$sd(u\vert m)$	`Jacobisd(u,m)`
$nd(u\vert m)$	`Jacobind(u,m)`
$dc(u\vert m)$	`Jacobidc(u,m)`
$nc(u\vert m)$	`Jacobinc(u,m)`
$sc(u\vert m)$	`Jacobisc(u,m)`
$ns(u\vert m)$	`Jacobins(u,m)`
$ds(u\vert m)$	`Jacobids(u,m)`
$cs(u\vert m)$	`Jacobics(u,m)`
$F(\phi\vert m)$	`EllipticF(phi,m)`
$K(m)$	`EllipticK(m)`
$E(\phi\vert m)$ または $E(m)$	`EllipticE(phi,m)` または `EllipticE(m)`
$H(u\vert m), H_1(u\vert m), \Theta_1(u\vert m), \Theta(u\vert m)$	`EllipticTheta(a,u,m)`
$\theta_1(u\vert m), \theta_2(u\vert m), \theta_3(u\vert m), \theta_4(u\vert m)$	`EllipticTheta(a,u,m)`
$Z(u\vert m)$	`Zeta_function(u,m)`
Lambert $\omega(z)$	`Lambert_W(z)`

定数	REDUCE の名前
Euler の γ 定数	`Euler_gamma`
Catalan の定数	`Catalan`
Khinchin の数	`Khinchin`
黄金比	`Golden_ratio`

関数	演算子
$\binom{n}{m}$	`Binomial(n,m)`
Motzkin(n)	`Motzkin(n)`
Bernoulli(n) または B_n	`Bernoulli(n)`
Euler(n) または E_n	`Euler(n)`
$S_n^{(m)}$	`Stirling1(n,m)`
$\mathbf{S}_n^{(m)}$	`Stirling2(n,m)`
$B(z,w)$	`Beta(z,w)`
$\Gamma(z)$	`Gamma(z)`
不完全 Beta $B_x(a,b)$	`iBeta(a,b,x)`
不完全 Gamma $\Gamma(a,z)$	`iGamma(a,z)`
$(a)_k$	`Pochhammer(a,k)`
$\psi(z)$	`Psi(z)`
$\psi^{(n)}(z)$	`Polygamma(n,z)`
Riemann の $\zeta(z)$	`Zeta(z)`
$Si(z)$	`Si(z)`
$si(z)$	`s_i(z)`
$Ci(z)$	`Ci(z)`
$Shi(z)$	`Shi(z)`
$Chi(z)$	`Chi(z)`
$erf(z)$	`erf(z)`
$erfc(z)$	`erfc(z)`
$Ei(z)$	`Ei(z)`
$li(z)$	`li(z)`
$C(x)$	`Fresnel_C(x)`
$S(x)$	`Fresnel_S(x)`
$dilog(z)$	`dilog(z)`
$Li_n(z)$	`Polylog(n,z)`
Lerch $\Phi(z,s,a)$	`Lerch_Phi(z,s,a)`

第52章　SPECFN2: 特殊関数のパッケージ

Victor S. Adamchik
Byelorussian University
Minsk, Belorus
and
Winfried Neun
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin–Dahlem, Germany
e–mail: *neun@zib.de*

52.1　GHYPER パッケージ

52.1.1　一般化された超幾何関数の簡約

ここではREDUCEの ghyper パッケージについて説明しています。このパッケージは一般化された超幾何関数を多項式、初等関数もしくは特殊関数またはより簡単な超幾何関数への簡約を行います。従って、このパッケージはREDUCE の特殊関数のパッケージと共に使われるべきです。

(一般化された) 超幾何関数

$$ {}_pF_q\left(\left.\begin{matrix}a_1,\ldots,a_p\\ b_1,\ldots,b_q\end{matrix}\right|z\right) \equiv \sum_{k=0}^{\infty}\frac{(a_1)_k(a_2)_k\cdots(a_p)_k}{(b_1)_k(b_2)_k\cdots(b_q)_k}\frac{z^k}{k!} $$

はテキストブックの特殊関数の項、例えば [6] で定義されています。多くのよく知られた関数がこのクラスに属しています。例えば、指数関数、対数関数、三角関数やベッセル関数等です。[36] にはこのような関数の和や基本的な性質それに応用についての記述があります。

数百の特殊な点での値が [6] に載っています。

52.1.2　HYPERGEOMETRIC 演算子

HYPERGEOMETRIC 演算子は三つの引数を取ります。最初の二つはパラメータリストで、最後は関数の引数です。パラメータリストは空リストであっても構いません。例えば、

```
hypergeometric ({},{},z);
```

```
 Z
E

hypergeometric ({1/2,1},{3/2},-x^2);

 ATAN(X)
---------
    X
```

52.1.3 HYPERGEOMETRIC 演算子の拡張

数百の特別な点での一般化された超幾何関数の値が、文献で得られます。この全てのものが HYPERGEOMETIRC 演算子で処理できるとは限りません。しかし、REDUCE の LET 文を使うことで、特別な点での簡約規則を定義することが出来ます。例えば、

```
let {hypergeometric({1/2,1/2},{3/2},-(~x)^2) => asinh(x)/x};
```

52.2 MEIJERG パッケージ

52.2.1 Meijer の G 関数

ここでは REDUCE の meijerG パッケージについて説明しています。これは、Meijer の G 関数の簡約を行います。この簡約は G 関数を多項式、初等または特殊関数、(一般化された) 超幾何関数で表します。このため、このパッケージは REDUCE の特殊関数や超幾何関数のパッケージ (GHYPER) と共に使います。

関数

$$G_{pq}^{mn}\left(z \;\middle|\; \begin{matrix}(a_p)\\(b_q)\end{matrix}\right) \equiv \frac{1}{2\pi i}\int_L \frac{\prod_{j=1}^{m}\Gamma(b_j-s)\prod_{j=1}^{n}\Gamma(1-a_j+s)}{\prod_{j=m+1}^{q}\Gamma(1-b_j+s)\prod_{j=n+1}^{p}\Gamma(a_j-s)}x^s ds$$

は 1936 年頃 C. S. Meijer によって研究され、これから Meijer の G 関数と呼ばれています。Meijer の G 関数の完全な定義は論文 [6]. に与えられています。多くのよく知られた関数は G 関数を使って表されます。例えば、指数関数、対数関数、三角関数、ベッセル関数や超幾何関数等です。

数百の特別な点での値が [6] に載っています。

52.2.2 MEIJERG 演算子

MEIJERG 演算子は三つの引数をとります。最初の二つはパラメータのリストで、これは空リストであっても構いません。最初が上のパラメータで次が下のパラメータです。3 番目は関数の引数を表します。

パラメータリストの最初の要素は、パラメータの最初の m または n リストでなければなりません。例えば、

$$G_{11}^{10}\left(x \,\middle|\, \begin{matrix}1\\0\end{matrix}\right)$$

は次のように入力します。

```
MeijerG({{},1},{{0}},x); % そして結果は:
```

```
 HEAVISIDE( - X + 1)
----------------------
      GAMMA(1)
```

となります。また、

$$G_{02}^{10}\left(\frac{x^2}{4} \,\middle|\, \begin{matrix}\\1+\frac{1}{4}\ 1-\frac{1}{4}\end{matrix}\right)$$

は

```
MeijerG({{}},{{1+1/4},1-1/4},(x^2)/4) * sqrt pi;
```

を表し、

```
                    1      2
 SQRT(PI)*BESSELJ(---,X)*X
                    2
----------------------------
             4
```

となります。

注意:特殊関数のパッケージを使うことでこれらの結果はもっと簡単な形に変形されます。

第53章　SUM:REDUCEの和分パッケージ

Fujio Kako
Department of Information and Computer Sciences
Faculty of Science , Nara Women's University ,
Nara 630, JAPAN
E-mail: *kako@ics.nara-wu.ac.jp*

このパッケージではGosperの算法により、級数の和を求める演算子SUMとPRODを定義しています。SUMは与えられた式の不定和分もしくは定和分を計算します。PRODは演算子は与えられた式の積を返します。これらの演算子の構文は次の通りです。

```
SUM(EXPR:数式、K:カーネル、[LOLIM:式 [,UPLIM:式]])
```

```
PROD(EXPR:数式、K:カーネル、[LOLIM:式 [,UPLIM:式]])
```

もし閉形式での解が求められなかった場合は、これらの演算子は入力された式をそのまま返します。UPLIMとLOLIMはオプションで、和(もしくは積)をとる上限値と下限値を指定します。もし、LOLIMが指定されていない場合には、上限値はK(和をとる変数と同じ)が指定されたものとして扱います。

例えば、

```
sum(n**3,n);
```

```
sum(a+k*r,k,0,n-1);
```

```
sum(1/((p+(k-1)*q)*(p+k*q)),k,1,n+1);
```

```
prod(k/(k-2),k);
```

Gosperの算法は比

$$\frac{\sum_{k=n_0}^{n} f(k)}{\sum_{k=n_0}^{n-1} f(k)}$$

がnの有理式で表される場合に、適応することができます。多項式、有理式や指数関数等の基本的な関数に対しては、関数SUM!-SQで処理しています。

SIN、COS等の三角関数に対しては、一度これらの関数を指数関数で表した後、Gosperの算法を適応しています。得られた結果はまたsin,cos,sinh,coshの関数で書き直して返すようにしています。

対数関数の和や指数関数の積に対しては次の公式によって計算しています。

$$\sum_{k=n_0}^{n} \log f(k) = \log \prod_{k=n_0}^{n} f(k)$$

$$\prod_{k=n_0}^{n} \exp f(k) = \exp \sum_{k=n_0}^{n} f(k)$$

それ以外の関数、例えば二項係数や形式的な積を含んだ式の和を求めるには、その関数に対して、引数の値が k と $k-1$ の場合に対する簡約規則を LET 文等で定義することによって求めることができます。

SUM パッケージのトレースを行う TRSUM というスイッチを用意しています。このスイッチは通常オフになっています。これをオンにすると、Gosper の算法による計算の途中結果が表示されます。

第54章　SYMMETRY: 対称行列上の演算

Karin Gatermann
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin–Dahlem
Federal Republic of Germany
E–mail: *gatermann@zib.de*

群の対称性を持つ行列に対して、対称性に適応された基底やブロック対角形式を計算する REDUCE のパッケージについて記述します。SYMMETRY パッケージは Dihedral 群のような小さな有限群のための線形表現に関する理論をインプリメントしたものです。

54.1　はじめに

対称性は数学、物理学および工学において重要な原理です。SYMMETRY パッケージの目標は小さい群の線形表現の理論を容易に扱えるようにすることです。例えば、Dihedral 群 D_3、D_4、D_5、D_6 が含まれています。理論への入門のためには Serre[40] あるいは Stiefel および Fässler[42] を参照して下さい。与えられた直交 (もしくはユニタリ) 線形表現

$$\vartheta : G \longrightarrow GL(K^n), \qquad K = R, C.$$

に対して指標 $\psi \to K$, 正準分解もしくは等方成分の基底をを計算します。行列 A が線形表現の対称性を持っている場合、

$$\vartheta_t A = A\vartheta_t \quad \forall\, t \in G,$$

は座標変換によってブロック対角形式に変換されます。実数体あるいは複素数体のいずれで行なうかはスイッチ c̄omplex で制御されます。このための例はテストファイル *symmetry.tst* の中で与えられます。

アルゴリズムが既約表現に関する情報を必要とするとともに、この情報はいくつかの群に備えて蓄えられます (3 節での演算子を参照)。直交表現 (ユニタリ表現) が与えられることを仮定されています。

パッケージは `load symmetry;` と入力することでロードされます。

54.2 線形表現のための演算子

最初に線形表現のデータ構造について説明します。*representation* は群の識別子と群の生成元に行列を対応させる方程式からなるリストです。

例:

```
rr:=mat((0,1,0,0),
        (0,0,1,0),
        (0,0,0,1),
        (1,0,0,0));

sp:=mat((0,1,0,0),
        (1,0,0,0),
        (0,0,0,1),
        (0,0,1,0));

representation:={D4,rD4=rr,sD4=sp};
```

直交表現 (ユニタリ表現) に体しては次の演算子が利用可能です。

canonicaldecomposition(representation) ;

は線形表現の正規分解を与える方程式を返します。

character(representation);

は線形表現の指標を計算します。 結果は群の識別子および群の要素の同値類中の群要素のリストおよび実もしくは複素数からなるリストのリストです。

symmetrybasis(representation,nr);

型 nr の既約表現に対応する等方なコンポーネントの基底を計算します。nr 番めの既約表現が多次元の場合、基底は対称が適応されます。基礎は適応された対称です。出力は行列です。

symmetrybasispart(representation,nr);

は `symmetrybasis` とにています。しかし、多次元の既約表現に対しては対称性を適応した基底の最初の部分のみしか計算されません。

allsymmetrybases(representation);

は `symmetrybasis` や `symmetrybasispart` と似ています。しかし、全ての等方要素の基底を計算します。従って、完全な座標変換が返されます。

diagonalize(matrix,representation);

与えられた線形表現 representation の対称性を持つ行列 matrix のブロック対角形式を返します。そうでなければ、エラー・メッセージを出力します。

もちろん、規約性は実数もしくは複素数の体 K に依存します。このため、アルゴリズムは体 K に依存します。計算のタイプは、スイッチ *complex* によって制御されます。

54.3 表示演算子

この節では、格納された群の情報へアクセスする演算子について、記述しています。最初に、抽象群の演算子について説明します。その後で、群の既約表現を得る方法を記述します。

availablegroups();

既約表現のような情報が保存されている全ての群のリストを返します。以下では、group は、常にこれらの群の名前のうちの 1 つです。

printgroup(group);

すべての群の要素のリストを返します;

generators(group);

群を生成する群要素のリストを返します。線形表現を定義するためには、これらの生成元のための行列を定義しなければなりません。

charactertable(group);

この群の既約表現に対応する特徴のリストを返します。

charactern(group,nr);

リストとして、この群の nr 番めの既約表現に対応する指標を返します (character を参照)。

irreduciblereptable(group);

群の既約表現のリストを返します。

irreduciblerepnr(group,nr);

群の既約表現を返します。出力は、群の識別子および群の要素に表現行列を割り当てる方程式のリストです。

54.4 新しい群の保存

ユーザが情報があらかじめ定められない群のために計算を行い時、パッケージ SYMMETRY はこの群の情報を供給する方法を用意しています。

これについては、次のデータ構造が使用されています。

elemlist = 識別子のリスト。

relationlist = 識別子と演算子 @ および ** による方程式のリスト。

grouptable = 行列で、その (1,1) 要素は grouptable であるもの

filename = ”myfilename.new”.

次のオペレーターはこの順に使用されなければなりません。

setgenerators(group,elemlist, relationlist);

例:

```
setgenerators(K4,{s1K4,s2K4},
  {s1K4^2=id,s2K4^2=id,s1K4@s2K4=s2K4@s1K4});
```

setelements(group,relationlist);

中立の要素以外の群の要素は、定義された生成元の積として与えられます。中立の要素は常に id と呼ばれます。

例:

```
setelements(K4,
     {s1K4=s1K4,s2K4=s2K4,rK4=s1K4@s2K4});
```

setgrouptable(group,grouptable);

群テーブルをインストールします。

例:

```
tab:=
 mat((grouptable,     id,    s1K4, s2K4, rK4),
     (id        ,     id,    s1K4, s2K4, rK4),
     (s1K4      ,    s1K4,     id,  rK4,s2K4),
     (s2K4      ,    s2K4,    rK4,   id,s1K4),
     (rK4       ,     rK4,   s2K4, s1K4,  id));

setgrouptable(K4,tab);
```

Rsetrepresentation(representation,type);

群の実の既約表現を定義するために使用されます。変数 type は既約表現の型を示す *realtype* または *complextype* のいずれかです。

例:

```
eins:=mat((1));
mineins:=mat((-1));
rep3:={K4,s1K4=eins,s2K4=mineins};
Rsetrepresentation(rep3,realtype);
```

Csetrepresentation(representation);

これは複素既約表現を定義します。

setavailable(group);

group203 の設定を終了します。それは、既約表現のいくつかの特性をチェックし、節 2 および 3 での演算子が利用可能にします。

storegroup(group,filename);

名前 *filename* でファイルに、group に関する情報を書きます。

loadgroups(filename);

ファイル *filename* から、ユーザが定義した群の情報をロードします。

第55章　TAYLOR:テイラー級数の計算

Rainer Schöpf
Zentrum für Datenverarbeitung der Universität Mainz
Anselm-Franz-von-Bentzel-Weg 12
D-55099 Mainz
Germany
E–mail: *Schoepf@Uni-Mainz.DE*

本節ではREDUCEでテイラー展開および得られたテイラー級数を効率よく扱うためのパッケージについて説明します。これには基本的な四則演算 (加算、減算、かけ算および除算) と代数関数および超越関数への応用が含まれています。また拡張として、ローラン展開やピュイズー展開を計算することもできます。

55.1　はじめに

テイラーパッケージはREDUCEで MACSYMA のTAYLOR 関数が提供する機能を使えるように作成したものですが、それにもまして私が望んだことは、より早くそして主記憶をあまり必要としないことです。特に、テイラー級数のLogや逆正接 (arctangent) を取ったときに、それがまたテイラー級数で表されるような機能を必要としました。そのため、プログラムの作成には予期していたよりも時間がかかりました。

このパッケージは現在開発されつつあるもので、これが利用者の役に立てればうれしいです。もしこれを使ってみて、何か欠けているような機能があれば知らせて下さい。

55.2　使い方

最も重要な演算子はTAYLOR です。これは次のように使います。

TAYLOR(EXP:**式** [,VAR:**カーネル**, VAR_0:**式**,ORDER:**整数**]...):**式**

ここでEXP は展開しようとしている式で、これはどんな式でもよく、テイラー級数を含んでいても構いません。VAR は展開しようとする変数名で、VAR_0 はこの変数に関してどの点の回りで展開するかを指定します。ORDER は展開の次数で、何次まで展開するかを指定します。もし一つ以上の (VAR, VAR0, ORDER) の三組を指定した場合、TAYLOR は式を各々の変数に関して独立に、順次展開を求めます。例えば、

```
taylor(e^(x^2+y^2),x,0,2,y,0,2);
```

では $X^2 * Y^2$ 次までの展開を求めます。一度テイラー展開を求めてしまうと、計算時に指定した次数以上の項は計算できません。(つまり、再度次数を大きくしてテイラー展開を求める以外には、高次の項を得る方法はありません。) VAR は変数名の代わりに変数名のリストを与えることもできます。この場合、展開は各々の変数の次数の和が ORDER を越えない範囲で求められます。もし VAR_0 が INFINITY であれば、EXP は 1/VAR の級数として展開されます。

展開は各変数に関して順次行われます。つまり、上の例では $\exp(x^2+y^2)$ はまず x に関して展開が行われ、次いでその展開の係数がそれぞれ y に関して展開されます。

暗黙のテイラー級数展開と、逆関数のテイラー級数展開を求める関数二つの演算子が定義されています。

IMPLICIT_TAYLOR(F:**式**,VAR1,VAR2:**カーネル**, $\mathrm{VAR1}_0$,$\mathrm{VAR2}_0$:**式**, ORDER:**整数**):**式**

は、変数 VAR1 と VAR2 に関する関数 F に対して、F(VAR1,VAR2) = 0 で定義される、関数 VAR2(VAR1) のテイラー展開を求めます。例えば、

```
implicit_taylor(x^2 + y^2 - 1,x,y,0,1,5);
```

INVERSE_TAYLOR(F:**式**,VAR1,VAR2:**カーネル**, $\mathrm{VAR1}_0$:**式**, ORDER:**整数**):**式**

は、変数 VAR1 の関数 F に対して、変数 VAR2 に関する F の逆関数のテイラー級数を求めます。例えば、

```
inverse_taylor(exp(x)-1,x,y,0,8);
```

ある決められた数の (零でない) 項のみが出力され、項数がそれ以上になると、(n terms) が出力されどれだけの項数が省略されたかが表示されます。出力される項の数は変数 TAYLORPRINTTERMS で決められます。この値は整数かまたは ALL です。ALL の場合は全ての項が表示されます。標準ではこの変数の値は 5 になっています。

もし、TAYLORKEEPORIGINAL スイッチがオンであると、もともとの式 EXP の値は後で参照できるように保存されます。この値は演算子

TAYLORORIGINAL(EXP:**式**):**式**

で取り出すことが出来ます。 もし EXP がテイラー級数でないかまたはオリジナルな式の値が保存されていなければ (つまり TAYLORKEEPORIGINAL スイッチがオフの状態で展開されたテイラー級数の場合) エラーになります。テイラー級数のテンプレート、つまり変数のリストと展開を行った点および展開の次数の組、は

TAYLORTEMPLATE(EXP:**式**):**リスト**

で取り出すことが出来ます。これは (VAR,VAR0,ORDER) をリストとして返します。もし EXP がテイラー級数でなければエラーを起こします。

TAYLORTOSTANDARD(EXP:**式**):**式**

は EXP 中のテイラー級数を通常の REDUCE の式に変換します。

TAYLORSERIESP(EXP:**式**):**論理値**

はEXPがテイラー級数かどうか判定します。 注意しておくことは、この演算子にはORDPやNUMBERPと同じ制限があることです。つまり、この演算子はIFもしくはLET文の論理式のところでのみ使うことが出来ます。最後に、

TAYLORCOMBINE(EXP:**式**):**式**

はEXP中に現れるテイラー級数をすべて一つの級数にまとめます。 現在可能な演算は:

- 加算、減算、乗算と除算
- 平方根、指数それに対数
- 三角関数および双曲関数およびそれらの逆関数

です。単項演算子に対してはほとんどすべての場合で計算可能です。二項演算子に関しては、それぞれの引数が同じテンプレートでのテイラー級数でなければいけません。つまり、展開の変数および展開する場所が同じでなければなりません。展開の次数については異なっていても構いません。この場合、計算が行われる前にどちらか片方の項が打ち切られます。

もしTAYLORKEEPORIGINALがオンでまたEXP中の全てのテイラー級数がその元の式を保持している場合、TAYLORCOMBINE exp; はこれらの式をまとめ、得られた式を結果のテイラー級数の元の式として記憶します。

数学的な不定を避けるため、いくつかの制約があります。項のない (つまり零の) テイラー級数の対数を取ったり、またこれで割り算を行うことは出来ません。展開の途中で現れる特異点を検出するための用意があり、展開点で分母が零になるため現れる極は検出され、正しく処理されます。つまりこの場合テイラー級数は負べきの項から始まります。(これはテイラー級数を計算するときに分子と分母をそれぞれ展開し、それから結果を求めているためです。) 上に述べた関数の真の特異点は正しく処理されます。

テイラー級数の微分を計算することが出来ます。もしテイラー変数に関して微分を行うと、次数が一つ下がったテイラー級数が得られます。

置換はちょっと制限があります。テイラー変数を別の変数に置き換えることのみ可能です。これには一つ例外があり、テイラー変数を定数で置き直すことは出来ます。注意することはREDUCEでは、与えられた式が定数であることを判定することが常に可能とは限りません。

単純なテイラー級数に対してのみ、その積分を計算することが出来ます。式の一部にテイラー級数が含まれているような数式は、TAYLORSTANDARD演算子で通常の数式に変換され、積分が計算されます。この場合警告のメッセージが出力されます。

テイラー級数 f の逆関数はTAYLORREVERTで求められます。

TAYLORREVERT(EXP:**式**,OLDVAR:**変数**, NEWVAR:**変数**):**式**

EXPは変数OLDVARに関するテイラー級数でなければなりません。例えば、

```
taylor (u - u**2, u, 0, 5);
taylorrevert (ws, u, x);
```

このパッケージではいくつかのスイッチが用意されています。

- TAYLORAUTOCOMBINE をオンにすると REDUCE はテイラー級数を自動的に結合します。これはテイラー級数を含む式の計算にすべて TAYLORCOMBINE を作用させるのと同じ効果を持ちます。通常はオンです。
- TAYLORAUTOEXPAND はテイラー級数を “contagious” にします。これは計算に置いて、すべてのテイラー級数でない式に対しては展開を行い、結果を結合します。既定値はオフです。
- TAYLORKEEPORIGINAL はオンあれば元の式を保存して置きます。テイラー級数に対する演算はすべてこの元の式にも適用され、TAYLORORIGINAL で取り出すことが出来ます。通常はオフになっています。
- TAYLORPRINTORDER がオンであればテイラー級数の剰余項は $+O(n)$ の形で表されます。オフの時には ... で表されます。通常はオンです。
- VERBOSELOAD スイッチがオンであると、パッケージがロードされるときその旨を出力します。通常はオフです。

55.3 注意

TAYLOR は最初の引数に関して解析的でない関数を検出します。例えば、$xy/(x+y)$ という関数は $(x,y)=(0,0)$ のまわりで解析的ではありません。従って、

```
taylor(x*y/(x+y),x,0,2,y,0,2);
```

を計算させると、エラーを起こします。

通常の REDUCE の演算子をテイラー級数に演算させることは通常出来ません。例えば、PART, COEFF, COEFFN は使えません。このようなことを行いたい場合は一度 TAYLORTOSTANDARD 演算子で通常の数式に変換してから行います。

第56章　TPS:べき級数

Alan Barnes
Dept. of Computer Science and Applied Mathematics
Aston University, Aston Triangle,
Birmingham B4 7ET
GREAT BRITAIN
Email: *barnesa@aston.ac.uk*
and
Julian Padget
School of Mathematics, University of Bath
Claverton Down, Bath, BA2 7AY
GREAT BRITAIN
Email: *jap@maths.bath.ac.uk*

56.1　はじめに

TPS パッケージは、一つの変数に関する形式巾級数展開を求めます。遅延評価を使っており、必要な項はその時点で計算されます。つまりコマンドが入力されたときには行われず、出力するときやべき級数を計算するために必要になったときに初めて行われます。また級数は全ての項の情報が失われずに保持されています。

56.2　PS 演算子

構文:

PS(EXPRN:**代数式**,DEPVAR:**変数**,ABOUT:**式**):**べき級数**

PS 演算子はべき級数を返します。べき級数は、タグを付けたドメインの要素として定義されています。これは変数 DEFVAR の点 ABOUT の周りでの EXPRN の一変数の形式的べき級数展開です。EXPRN はそれ自身べき級数を含んでいても構いません。

ABOUT は DEPVAR に依存しない式でなければなりません。さもなければエラーになります。もし ABOUT が INFINITY の場合は DEPVAR=∞ での展開が 1/DEPVAR の級数の形で求められます。

もしコマンドをセミコロンで終了すると、EXPRN のべき級数展開が計算され表示されます。

展開の次数は PSEXPLIM で指定された次数まで行われます。もし後になって PSEXPLIM の値が大きくされたときでも、すでに計算した項に対しては再計算する必要はなく、必要な余分の展開項のみを計算するのに十分なだけの情報がべき級数に保存されています。

もしコマンドがドル記号で終了した場合、べき級数オブジェクトは設定されますが、この時点では実際の展開はせいぜい最初の項が計算されるだけです。

もし関数が展開点で極を持っている時には正しいローラン展開が求められます。

以下は PS 演算子の使い方の例です。

```
psexplim 6;
ps(log x,x,1);
ps(e**(sin x),x,0);
ps(x/(1+x),x,infinity);
ps(sin x/(1-cos x),x,0);
```

新しくユーザが定義した関数は LET 文で以下の規則を定義することにより、べき級数展開を計算することが可能になります。

1. 展開点での関数の値
2. 関数の微分規則

例えば

```
operator sech;
forall x let df(sech x,x)= - sech x * tanh x;
let sech 0 = 1;
ps(sech(x**2),x,0);
```

不定積分のべき級数展開 (たとえ REDUCE がその積分を閉形式で求められなくとも) を求めることが出来ます。例えば、次の例を見て下さい。

```
ps(int(e**x/x,x),x,1);
```

注意しておくことは、もし積分変数が展開の変数と同じ場合、REDUCE の積分パッケージは呼ばれません。もしこの二つの変数名が異なっているときは、展開されたべき級数の各係数は積分されます。積分定数は通常零です。

56.3 PSEXPLIM 演算子

構文:

PSEXPLIM(UPTO:**整数**):**整数**

または

PSEXPLIM():**整数**

この関数は TPS パッケージの内部変数の値を UPTO に設定します。以前の設定値が返されます。最初はこの値は 6 になっています。

もし引数無しでこの関数を呼び出すと、現在の設定値が返されます。

56.4 PSORDLIM 演算子

構文:

PSORDLIM(UPTO:**整数**):**整数**

または

PSORDLIM():**整数**

この関数は TPS パッケージの内部変数の値を UPTO に設定します。以前の設定値が返されます。最初はこの値は 15 になっています。

もし引数無しでこの関数を呼び出すと、現在の設定値が返されます。

この関数はべき級数を計算するとき、最初に零でない項の次数を設定します。展開を計算するとき、この次数の項まで計算してすべて零であればエラーとして計算を終了します。これは以下の例のように、無限ループに陥るのを防ぐために有効です。

```
ps(1 - (sin x)**2 - (cos x)**2,x,0);
```

これは恒等的に零ですが、REDUCE は零とは認識できません。(PS は最初の零でない項を求めようとしますが、これは無限ループになります。)

56.5 PSTERM 演算子

構文:

PSTERM(TPS:**べき級数**,NTH:**整数**):**代数式**

PSTERM 演算子はべき級数 TPS から NTH 番目の項を取り出します。NTH が整数でなかったり、また TPS がべき級数でなければエラーになります。整数はべき級数として扱われます。

56.6 PSORDER 演算子

構文:

PSORDER(TPS:**べき級数**):**整数**

演算子 PSORDER はべき級数の次数 (order) を返します。これは係数が零でない最小次数の項の次数です。もし TPS がべき級数でなければエラーになります。もし TPS が零であれば、UNDEFINED

が返されます。

56.7 PSSETORDER 演算子

構文:

PSSETORDER(TPS:**べき級数**, ORD:**整数**):**整数**

演算子 PSSETORDER はべき級数 TPS の次数 (order) を ORD に設定します。これは以前の order を返します。この関数は、微分方程式で定義される関数のべき級数で TPS パッケージが order を自動的に決定することが出来ない場合に、その関数の order を設定するのに使用することができます。

56.8 PSDEPVAR 演算子

構文:

PSDEPVAR(TPS:**べき級数**):**識別子**

演算子 PSDEPVAR はべき級数 TPS の展開変数を返します。もし TPS が整数であれば、UNDEFINED が返されます。

56.9 PSEXPANSIONPT 演算子

構文:

PSEXPANSIONPT(TPS:**べき級数**):**代数式**

演算子 PSEXPANSIONPT はべき級数の展開点を返します。もし TPS が整数であれば、UNDEFINED が返されます。もし展開が無限遠で行われているときには INFINITY が返されます。

56.10 PSFUNCTION 演算子

構文:

PSFUNCTION(TPS:**べき級数**):**代数式**

演算子 PSFUNCTION はその展開がべき級数 TPS になる関数を代数式を返します。

56.11 PSCHANGEVAR 演算子

構文:

PSCHANGEVAR(TPS:**べき級数**, X:**カーネル**):**べき級数**

演算子 PSCHANGEVAR はべき級数 TPS の従属変数名を X に変更します。X はカーネル (変数名) でこれは TPS のパラメータには含まれてはいけません。新しい変数に書き直されたべき級数が返されます。

56.12 PSREVERSE 演算子

構文:

PSREVERSE(TPS:**べき級数**):**べき級数**

べき級数の逆関数を計算します。次の 4 つの場合が考えられます。

1. もし級数の次数が 1 の時、逆関数は零の周りでの展開が得られます。
2. もし次数が零で、かつ TPS の最初の項の次数が零でない場合、この場合逆関数の展開点は TPS の零次の項の係数になります。
3. もし次数が-1 のときは逆関数は無限大の周りでの展開が得られます。それ以外の場合は逆関数はべき級数としては得られないので REDUCE のエラーになります。これらの場合を扱うには Puiseux 級数展開が必要です。
4. もし TPS の展開点が有限であれば、それは逆関数の級数展開の零次の項になります。零の周りでの展開または無限大での展開では、その逆関数の次数は 1 になります。

もし TPS がべき級数でなければエラーになります。

ここにいくつかの例を挙げて置きます。

```
ps(sin x,x,0);
psreverse(ws); % asin x の x=0 の周りでの展開
ps(exp x,x,0);
psreverse ws; % log x の x=1 の周りでの展開
ps(sin(1/x),x,infinity);
psreverse(ws); % 1/asin(x) の x=0 の周りでの展開
```

56.13 PSCOMPOSE 演算子

構文:

PSCOMPOSE(TPS1:**べき級数**, TPS2:**べき級数**):**べき級数**

PSCOMPOSE べき級数の合成を計算します。べき級数 TPS1 と TPS2 が合成されます。つまり、TPS2 が TPS1 の変数に代入され、結果がべき級数として表されます。結果のべき級数の展開変数と展開点は TPS2 のものと同じになります。べき級数の合成については、以下の条件が満たされなければなりません。

1. もし TPS1 の展開点が零であれば、TPS2 の次数は少なくとも 1 でないといけません。

2. もし TPS1 の展開点が有限値であれば、それは TPS2 の零次の項の係数と一致していなければなりません。この場合 TPS2 の次数は零または正でなければいけません。

3. もし TPS1 の展開点が無限大であれば、TPS2 の次数は-1 以下でなければいけません。

もしこれらの条件が満たされなければ級数は合成できず、エラーを起こします。

例:

```
a:=ps(exp y,y,0);  b:=ps(sin x,x,0);
pscompose(a,b);
% x=0 での exp(sin x) べき級数を求める

a:=ps(exp z,z,1); b:=ps(cos x,x,0);
pscompose(a,b);
% x=0 での exp(cos x) べき級数を求める

a:=ps(cos(1/x),x,infinity);  b:=ps(1/sin x,x,0);
pscompose(a,b);
% x=0 での cos(sin x) べき級数を求める
```

56.14 PSSUM 演算子

構文:

PSSUM(J:**カーネル** = LOWLIM:**整数**, COEFF:**代数式**, X:**カーネル**,
ABOUT:**代数式**, POWER:**代数式**):**べき級数**

J が LOWLIM から INFINITY までの形式的べき級数の和

```
COEFF*(X-ABOUT)**POWER
```

または、もし ABOUT が INFINITY のときは

```
COEFF*(1/X)**POWER
```

が返されます。この返された式はべき級数の形で、このパッケージに含まれるべき級数を扱う関数で操作することができます。

J と X は異なった変数で、ABOUT, COEFF と POWER は変数 X に依存していてはいけません。また ABOUT は変数 J に依存してはいけません。POWER は J の関数で、J が LOWLIM から無限大まで動くとき、狭義の単調増大関数になっていなければいけません。

```
pssum(n=0,1,x,0,n*n);
```

```
% n=0 から無限大までの x**(n*n) の和

pssum(m=1,(-1)**(m-1)/(2m-1),y,1,2m-1);
% y=1 での atan(y-1) のべき級数展開

pssum(j=1,-1/j,x,infinity,j);
% 無限大での log(1-1/x) のべき級数展開

pssum(n=0,1,x,0,2n**2+3n) + pssum(n=1,1,x,0,2n**2-3n);
% n が負の無限大から無限大までの x**(2n**2+3n) の和
```

56.15 PSCOPY 演算子

構文:

PSCOPY(TPS: **べき級数**):**べき級数**

この関数は、べき級数 TPS のコピーを返します。コピーは、オリジナルのべき級数とは部分式を共有しません。これを使うことにより、もともとのべき級数に影響を与えることなく、級数に対して代入等の操作を行なうことができます。例えば、

```
clear a;
b := ps(sin(a*x)), x, 0);
b where a => 1;
```

は、巾級数の各項の中の a を 1 にし、式を簡約します。このとき、副作用の影響で、もともとの b 自身も変更されてしまいます。代入を行なう前に PSCOPY で、級数をコピーして置くことで、このような副作用による影響を避けることができます。つまり、次のようにする。

```
b := ps(sin(a*x)), x, 0);
pscopy b where a => 1;
```

56.16 PSTRUNCATE 演算子

構文: PSTRUNCATE(TPS:**巾級数** POWER: **整数**) :**代数式**

この演算子は、べき級数 TPS を POWER 次よりも高次の項で打ち切った多項式を返します。もし、TPS の計算された級数が POWER 次よりも小さい場合には、自動的に必要な次数まで計算されます。

```
b := ps(sin x, x, 0);
a := pstruncate(b, 11);
```

は、$\sin x$ を x^{11} より高次の項を打ち切った多項式が a に代入されます。

もし、POWER が級数の次数よりも小さい場合、0 が返されます。また、POWER が整数値でない場合や TPS がべき級数でない場合にはエラーを起こします。

56.17 代数演算

べき級数はドメイン要素として実現されており、これらと REDUCE の通常の数式との間の代数演算は REDUCE の通常の演算で行えます。

例えば、A と B はべき級数とするとき、

```
a*b;
a**2+b**2;
```

は A と B の積および A と B の自乗の和のべき級数をそれぞれ出力します。

56.18 微分

もし A はべき級数で X に依存しているとする。このとき df(a,x); は A を X で微分した式のべき級数を返します。

注意 積分を計算するときに、int(a,x); ではできません。べき級数 a の積分を計算する為には ps(int(a,x),x,0); と入力しなくてはなりません。

56.19 制約とバグ

もし A と B がべき級数で、X が変数とすると、a*x といった式は (たとえ各々の結果が形式的には正しいとしても) 単一のべき級数になりません。代わりに ps(a*x,x,0); として下さい。

同様に A べき級数としたとき、sin(A) はべき級数に展開されません。例えば

```
a:=ps(1/(1+x),x,0);
b:=sin a;
```

は sin(1/(1+x)) をべき級数に展開しません。代わりに、

```
SIN(1 - X + X**2 - X**3 + .....)
```

が出力されます。しかし、

```
b:=ps(sin(a),x,0);
```

もしくは

```
b:=ps(sin(1/(1+x)),x,0);
```

で希望する結果が得られます。

真の特異点をもつ関数は現在の所扱えません。通常次のようなエラーを起こします。

```
***** Essential Singularity      (***** 真の特異点)
```

または

```
***** Logarithmic Singularity      (***** 対数型特異点)
```

また場合によっては零での割り算のエラーやスタックが溢れたというエラーを起こすこともあります。

第57章　TRI: TeXとREDUCEのインターフェース

Werner Antweiler, Andreas Strotmann and Volker Winkelmann

University of Cologne Computer Center,
Abt. Anwendungssoftware, Robert-Koch-Straße 10
5000 K"oln 41, Germany

e–mail: *antweil@epas.utoronto.ca strotmann@rrz.uni-koeln.de winkelmann@rrz.uni-koeln.de*

REDUCE-TEX-インターフェースは、三つのレベルのTEX出力をを組込みます。それぞれ、行分割なし、行分割ありおよび行分割と段付けでの出力です。

パッケージのロード中に、いくつかの規定の初期化は実行されます。規定値ではページ幅は15センチメートルにセットされます。ページ分割に対する許容値は規定値では20にセットされます。さらに、TRIはギリシア文字を翻訳することができます。例えばTAUやPSIは、等価なTEXのシンボルのτやψにそれぞれ変換されます。文字はLOWERCASE宣言によって小文字で出力されます。

57.1　TRIのスイッチ

3つのTRIの三つの出力レベルはスイッチで選択することができます。スイッチ`TEX`をオンにすることで、標準のTEX-出力が得られます。スイッチ`TEXBREAK`は行分割を行なう出力を選択します。そして、`TEXINDENT`スイッチは行分割されたTEX-出力に加えて段付けを行なうことを指定します。従って、これらの三つのレベルは、次のようなスイッチの組合せで行なえます。

```
On TeX;          % スイッチ TeX をオン
On TeXBreak;     % スイッチ TeX と TeXBreak をオン
On TeXIndent;    % スイッチ TeX, TeXBreak と TeXIndent をオン
Off TeXIndent;   % スイッチ TeXIndent をオフ
Off TeXBreak;    % スイッチ TeXBreak と TeXIndent をオフ
Off TeX;         % 全てのスイッチをオフ
```

TRIがどのように行を分割するかはページ幅と許容値を設定することで制御できます。

```
TeXsetbreak(page_width, tolerance);
```

ページ幅はミリメートルで測定されます。また、許容値は区間 [0...10000] 内の正の整数値で指定します。許容値を大きくすることによって、より多くの点で分割可能になります。許容値を 0 にすると、分割しないことを意味します。また、10000 の許容値は任意の点で分割できることを意味します。段付けなしで行を分割する場合、許容値の適切な値は 10 から 100 の間です。経験的に、大きな値を使用すると項は深く入れ子にされます。段付けを使う場合にはもっと大きな許容値を指定して下さい。大体 700 から 1500 までの値が適切です。

57.1.1 変換の追加

しばしば、ある特定の REDUCE のシンボルに対して特別の TEX の記号を割り当てたいことがあります。そのような目的のため、TRI は関数 TeXlet を提供しています。この関数の呼び出しは次のシンタックスを持っています:

TeXlet(*REDUCE-symbol*, *TEX-item*);

例えば、

```
TeXlet('velocity,'!v);
TeXlet('gamma,\verb|'!\!G!a!m!m!a! |);
TeXlet('acceleration,\verb|'!\!v!a!r!t!h!e!t!a! |);
```

この方法に加えて、小文字およびギリシャ文字の (現在) 二つの文字セットを指定することができます。これらのセットは標準でロードされます。これらのセットは以下に示す関数を使って指定したり、あるいは現在使用しているセットから別のセットに切り替えることができます。

TeXassertset *setname*;
TeXretractset *setname*;

セット名 setname として定義されているのは'GREEK と 'LOWERCASE です。

関数 TeXitem を使うことによって、新しいセットを定義することができます。

57.2 使用例

以下にあげるいくつかの代表的な例で、TRI の機能を示します。

```
load_package tri;
    % TeX-REDUCE-Interface 0.50
    % set greek asserted
    % set lowercase asserted
    % \tolerance 10
    % \hsize=150mm

TeXsetbreak(150,250);
    % \tolerance 250
    % \hsize=150mm
```

```
on TeXindent;

(x+y)^16/(v-w)^16;
    $$\displaylines{\qdd
    \(x^{16}
      +16\cdot x^{15}\cdot y
      +120\cdot x^{14}\cdot y^{2}
      +560\cdot x^{13}\cdot y^{3}
      +1820\cdot x^{12}\cdot y^{4}
      +4368\cdot x^{11}\cdot y^{5}\nl
      \off{327680}
      +8008\cdot x^{10}\cdot y^{6}
      +11440\cdot x^{9}\cdot y^{7}
      +12870\cdot x^{8}\cdot y^{8}
      +11440\cdot x^{7}\cdot y^{9}
      +8008\cdot x^{6}\cdot y^{10}\nl
      \off{327680}
      +4368\cdot x^{5}\cdot y^{11}
      +1820\cdot x^{4}\cdot y^{12}
      +560\cdot x^{3}\cdot y^{13}
      +120\cdot x^{2}\cdot y^{14}
      +16\cdot x\cdot y^{15}
      +y^{16}
    \)
    /\nl
    \(v^{16}
      -16\cdot v^{15}\cdot w
      +120\cdot v^{14}\cdot w^{2}
      -560\cdot v^{13}\cdot w^{3}
      +1820\cdot v^{12}\cdot w^{4}
      -4368\cdot v^{11}\cdot w^{5}\nl
      \off{327680}
      +8008\cdot v^{10}\cdot w^{6}
      -11440\cdot v^{9}\cdot w^{7}
      +12870\cdot v^{8}\cdot w^{8}
      -11440\cdot v^{7}\cdot w^{9}
      +8008\cdot v^{6}\cdot w^{10}\nl
      \off{327680}
      -4368\cdot v^{5}\cdot w^{11}
      +1820\cdot v^{4}\cdot w^{12}
      -560\cdot v^{3}\cdot w^{13}
      +120\cdot v^{2}\cdot w^{14}
      -16\cdot v\cdot w^{15}
```

```
      +w^{16}
    \)
    \Nl}$$
```

これを TEX で出力したものは次のようになります。

$$(x^{16}+16\cdot x^{15}\cdot y+120\cdot x^{14}\cdot y^2+560\cdot x^{13}\cdot y^3+1820\cdot x^{12}\cdot y^4+4368\cdot x^{11}\cdot y^5$$
$$+8008\cdot x^{10}\cdot y^6+11440\cdot x^9\cdot y^7+12870\cdot x^8\cdot y^8+11440\cdot x^7\cdot y^9+8008\cdot x^6\cdot y^{10}$$
$$+4368\cdot x^5\cdot y^{11}+1820\cdot x^4\cdot y^{12}+560\cdot x^3\cdot y^{13}+120\cdot x^2\cdot y^{14}+16\cdot x\cdot y^{15}+y^{16})/$$
$$(v^{16}-16\cdot v^{15}\cdot w+120\cdot v^{14}\cdot w^2-560\cdot v^{13}\cdot w^3+1820\cdot v^{12}\cdot w^4-4368\cdot v^{11}\cdot w^5$$
$$+8008\cdot v^{10}\cdot w^6-11440\cdot v^9\cdot w^7+12870\cdot v^8\cdot w^8-11440\cdot v^7\cdot w^9+8008\cdot v^6\cdot w^{10}$$
$$-4368\cdot v^5\cdot w^{11}+1820\cdot v^4\cdot w^{12}-560\cdot v^3\cdot w^{13}+120\cdot v^2\cdot w^{14}-16\cdot v\cdot w^{15}+w^{16})$$

行列を使った簡単な例:

```
load_package ri;
    % TeX-REDUCE-Interface 0.50
    % set greek asserted
    % set lowercase asserted
    % \tolerance 10
    % \hsize=150mm

on Tex;

mat((1,a-b,1/(c-d)),(a^2-b^2,1,sqrt(c)),((a+b)/(c-d),sqrt(d),1));
    $$
    \pmatrix{1&a
             -b&
             \frac{1}{
                   c
                   -d}\cr
             a^{2}
             -b^{2}&1&
             \sqrt{c}\cr
             \frac{a
                   +b}{
                   c
                   -d}&
             \sqrt{d}&1\cr
             }
    $$
```

この例では次のような TEX 出力が得られます。

$$\begin{pmatrix} 1 & a-b & \frac{1}{c-d} \\ a^2-b^2 & 1 & \sqrt{c} \\ \frac{a+b}{c-d} & \sqrt{d} & 1 \end{pmatrix}$$

生成されるファイルは、例題と共に配布されているファイル `trigdefs.tex` で定義されている多くの TEX マクロを使用していることに注意して下さい。

第58章　TRIGINT: Weierstrass 置換

Neil Langmead
Konrad-Zuse-Zentrum für Informationstechnik (ZIB)
Takustrasse 7
D- 14195 Berlin Dahlem
Berlin Germany

58.1　はじめに

このパッケージは積分から「偽りの」不連続を取り除くために、D.J. ジェフリーおよび A.D. リッチ [29] によって提案された、新しいアルゴリズムのインプリメンテーションです。それらの論文は、三角法の積分を評価するためにほとんどのコンピューター代数システムの中で Risch アルゴリズムと共に使用されている、Weirstrass 置換 ($u = \tan(x/2)$) に注目します。この置換を使用した計算で得られる結果には、時々不連続があらわれ、このため結果の式が成立する領域が限定されるということが起こります。ここで示されたアルゴリズムは、被積分関数の不定積分でより広い領域で連続であるという意味で、よりよい表現を見つけます。

58.1.1　例

次の問題について考える:

$$\int \frac{3}{5-4\cos(x)}\,dx \tag{58.1}$$

REDUCE はこの不定積分を Weirstrass 置換 $u = \tan(\frac{x}{2})$ を使って変換し、その後 Risch のアルゴリズムを使って解を

$$\frac{2\arctan(3\tan(\frac{x}{2}))}{3}, \tag{58.2}$$

と計算します。しかしこの解は π の奇数倍のところで不連続な解です。もともとの関数は連続で、従って不定積分した関数もまた連続なはずです。

Jeffery および Richs のアルゴリズムは与えられた問題に下記を返します:

$$\int \frac{3}{5-4\cos(x)}\,dx = 2\arctan(3\tan(\frac{x}{2})) + 2\pi\lfloor\frac{x-\pi}{2\pi}\rfloor$$

これは先ほどの解とは定数 2π だけ違っています。

58.2 Weirstrass 置換

Weierstrass は次の表にある関数 $u = \Phi(x)$ のどれかを使った変換であると定義します。

Weirstrass 変換で使用する関数 $u = \Phi$ および対応する変換						
選択	$\Phi(x)$	$\sin(x)$	$\cos(x)$	dx	b	p
(a)	$\tan(x/2)$	$\frac{2u}{1+u^2}$	$\frac{1-u^2}{1+u^2}$	$\frac{2du}{1+u^2}$	π	2π
(b)	$\tan(\frac{x}{2}+\frac{\pi}{4})$	$\frac{u^2-1}{u^2+1}$	$\frac{2u}{u^2+1}$	$\frac{2du}{1+u^2}$	$\frac{\pi}{2}$	2π
(c)	$\cot(x/2)$	$\frac{2u}{1+u^2}$	$\frac{u^2-1}{1+u^2}$	$\frac{-2du}{1+u^2}$	0	2π
(d)	$\tan(x)$	$\frac{u}{\sqrt{1+u^2}}$	$\frac{1}{\sqrt{1+u^2}}$	$\frac{du}{1+u^2}$	$\frac{\pi}{2}$	π

変換にはこれら以外にも、例えば $\sin(x)$ や $\cos(x)$ を使った変換が考えられます。しかし、これらの変換は特異ではなく、従って不連続性は現れません。

与えられた被積分関数 $f(\sin(x), \cos(x))$ に対して、どの変換を使うのかは、発見法的な方法で次のように決められます。

(a) 被積分関数が $\sin(x)$ を含まない場合。

(b) 被積分関数が $\cos(x)$ を含まない場合。

(c) 変換 (a) で積分できない場合。

(d) Gradshteyn and Ryzhik による条件を満たす場合。

例えば、変換 (c) を選んだ場合、

$$\int f(\sin x, \cos x)\, dx = \int f(\frac{2u}{1+u^2}, \frac{u^2-1}{1+u^2})\, \frac{-2du}{1+u^2}.$$

となる。

変数 u に関する積分は通常のルーチンを使って行なわれ、得られた結果に u が代入されます。結果を $\hat{g}(x)$ とします。これに対して、

$$K = \lim_{x \to b^-} \hat{g}(x) - \lim_{x \to b^+} \hat{g}(x),$$

を計算する。ここで点 b は表で与えられます。正しい不定積分は、

$$g(x) = \int f(\sin x, \cos x)\, dx = \hat{g}(x) + K \lfloor \frac{x-b}{p} \rfloor,$$

で与えられます。ここで、周期 p は表で与えられます。

58.3 REDUCE でのインプリメンテーション

このアルゴリズムをインプリメントしたのが関数 `trigint` です。これは次の構文をとります。

trigint(exp,var),

ここで、exp は被積分関数で、var は積分変数です。

もし、trigint が置換を必要としない積分を計算するために使用された場合、標準ではない結果を返す場合があります。

例えば、

$$\textbf{trigint(cos(x),x),}$$

を計算すると、

$$\frac{2\tan\frac{x}{2}}{\tan\frac{x}{2}^2+1}$$

という結果が得られます。これは、簡単な三角関数の関係式をつかうと

$$\frac{2\tan\frac{x}{2}}{\tan\frac{x}{2}^2+1}\to\frac{2\tan\frac{x}{2}}{\sec^2\frac{x}{2}}\to 2\sin\frac{x}{2}\cos\frac{x}{2}\to\sin 2\frac{x}{2}\to\sin x,$$

と $\sin(x)$ という結果になります。

58.3.1 例

```
1: trigint(3/(5-4*cos(x)),x);


               x              - pi + x
2*(atan(3*tan(---)) + floor(-----------)*pi)
               2               2*pi

2: trigint(3/(5+4*sin(x)),x);

                                      2
               pi + 2*x             - pi  + 2*pi*x
2*(atan(3*tan(----------)) + floor(-----------------)*pi
                  4                       4

              - pi + 2*x
    + floor(-------------)*pi)
               4*pi



3: trigint(15/(cos(x)*(5-4*cos(x))),x);

               x                - pi + x                     x
8*atan(3*tan(---)) + 8*floor(-----------)*pi - 3*log(tan(---) - 1)
               2                 2*pi                        2
```

```
              x
 + 3*log(tan(---) + 1)
              2
```

58.4 トレース

パッケージは、内部の計算の詳細のトレースを出力する機能を持っています。スイッチ tracetrig を

```
on tracetrig;
```

として、オンにすれば詳細なメッセージが出力されます。このスイッチは通常はオフになっています。

第59章　TRIGSIMP: 三角関数の簡約

Wolfram Koepf

Andreas Bernig

Herbert Melenk

ZIB Berlin

Revised by **Francis Wright**

QMW London

E-mail: *F.J.Wright@Maths.QMW.ac.uk*

TRIGSIMP パッケージには trigsimp, trigfactorzie, triggcd の三つの関数があります。最初の関数は、三角関数や双曲線関数を含んだ式を簡約します。trigfactorize は因数分解です。そして、triggcd は二つの多項式の最大公約式を計算します。

59.1　三角関数を含んだ式の簡約

三角関数や双曲線関数に対しては正規形式が存在しません。このため、同じ関数が違う形で表されます。例えば、$\sin(2x) \leftrightarrow 2\sin(x)\cos(x)$。関数 `trigsimp` に幾つかのパラメータを指定することで、変換 の 方向を指定することが出来ます。三角関数や双曲線関数を含んだ有理式に対して、ゼロになるか否かの判定が可能です。

関数 `f` を簡約するには、`trigsimp(f[,options])` を使います。例えば、

```
2: trigsimp(sin(x)^2+cos(x)^2);
```

```
1
```

可能なオプションは次の通りです (* は通常の値を示します):

1. `sin` (*) または `cos`
2. `sinh` (*) または `cosh`
3. `expand` (*) または `combine` または `compact`
4. `hyp` または `trig` または `expon`
5. `keepalltrig`

それぞれのグループのオプションについて、最大限いずれかの一つのみが指定できます。最初のグループのオプションは、三角関数に対して、正弦関数か余弦関数いずれの方に結果を合わせるかを指定します。二番目のグループは、双曲線関数に対する指定です。三番目のグループは変換の型を指定します。通常は expand で、結果の関数の引数の和は展開されます。combine では、三角関数の積は和に変換します。

```
trigsimp(sin(x)^2,cos);

         2
 - cos(x)  + 1

trigsimp(sin(x)*cos(y),combine);


 sin(x - y) + sin(x + y)
-------------------------
            2
```

オプション compact では、REDUCE の関数 compact (9 章を参照のこと) が f に適用されます。これによって、通常より単純な形に簡約されます。しかしながら、expand とは違って正規型は得られません。

```
trigsimp((1-sin(x)**2)**20*(1-cos(x)**2)**20,compact);

      40       40
cos(x)  *sin(x)
```

四番目のグループのオプションは、三角関数 (trignommetric)、双曲線関数 (hyperbolic) もしくは指数関数 (exponential) で表すものです。

```
trigsimp(sin(x),hyp);

 - sinh(i*x)*i

trigsimp(e^x,trig);

     x          x
cos(---) + sin(---)*i
     i          i
```

通常、関数 tan, cot, sec, csc は sin や cos を使って表されます。時には、これを避けたい場合がありますが、これはオプション keepalltrig によって処理できます。

```
trigsimp(tan(x+y),keepalltrig);

  - (tan(x) + tan(y))
----------------------
  tan(x)*tan(y) - 1
```

別のグループのオプションを同時に使用することも出来ます。

59.2 三角関数の因数分解

関数 trigfactorize(p,x) は、変数 x に関する三角関数や双曲線関数を含んだ式 p を因数分解します。例えば、

```
trigfactorize(sin(x),x/2);

        x        x
{2,cos(---),sin(---)}
        2        2
```

もし多項式が対等でなかったりバランスしていない場合には、入力と同じものが返されます。この場合、変数 x を変えることによって分解を得ることができることがあります。

```
trigfactorize(1+cos(x),x);

{cos(x) + 1}

trigfactorize(1+cos(x),x/2);

        x        x
{2,cos(---),cos(---)}
        2        2
```

59.3 三角関数の GCD

関数 triggcd は trigfactorize の応用です。この関数によって、二つの三角関数や双曲線関数を含んだ式の最大公約式を計算できます。構文は triggcd(p,q,x) で、ここで p と q は多項式で x は使用する最小の単位です。例えば、

```
triggcd(sin(x),1+cos(x),x/2);

     x
cos(---)
     2

triggcd(sin(x),1+cos(x),x);

1
```

ASSIST パッケージも参照して下さい。(3 章).

第60章　WU:Wuの方法による方程式の解法

Russell Bradford

School of Mathematical Sciences,
University of Bath,
Claverton Down,
Bath, BA2 7AY

E-mail: *rjb@maths.bath.ac.uk*

これは Wu の “A Zero Structure Theorem for Polynomial-Equations-Solving,” Wu Wen-tsun, Institute of Systems Science, Academia Sinica, Beijing によるアルゴリズムをインプリメントしたものです。

これは彼のアルゴリズムを理解するために作成したもので、そのためプログラムは非常に単純で、また多くのプログラムの動作を追跡するためのコードを含んでいます。また、改善や最適化の余地をたくさん残しています。例えば、多項式のリストを適当な順序に入れ替えるとか、同じ計算をさけること等です。また、入力された多項式を因数分解することは、計算の効率化に役立つでしょう。

バグの修正や改善は大いに歓迎します。

使い方は、例えば:

```
wu( {x^2+y^2+z^2-r^2, x*y+z^2-1, x*y*z-x^2-y^2-z+1}, {x,y,z});
```

は、指定された多項式と、変数の順序 x > y > z で、計算を行います。この例では、r はパラメータです。

結果は

```
    2    3    2
{{{r  + z  - z  - 1,

    2  2    2      2    4    2  2    2
   r *y  + r *z + r  - y  - y *z  + z  - z - 2,

          2
   x*y + z  - 1},

  y},
```

```
   6  4      6  2    6      4  7      4  6      4  5      4  4
{{r *z  - 2*r *z  + r  + 3*r *z  - 3*r *z  - 6*r *z  + 3*r *z  + 3*

   4  3      4  2      4      2  10      2  9      2  8      2  7
  r *z  + 3*r *z  - 3*r  + 3*r *z   - 6*r *z  - 3*r *z  + 6*r *z  +

      2  6      2  5      2  4      2  3      2    13      12    11
   3*r *z  + 6*r *z  - 6*r *z  - 6*r *z  + 3*r  + z   - 3*z   + z

        10    9      8      7    6      4      3    2
   + 2*z   + z  + 2*z  - 6*z  - z  + 2*z  + 3*z  - z  - 1,

   2   2    3    2
  y *(r  + z  - z  - 1),

         2
  x*y + z  - 1},

     2    3    2
  y*(r  + z  - z  - 1)}}
```

となります。これは、特性多項式と制約条件のリストになっています。

つまり、最初の組み合わせは特性多項式

$$r^2 + z^3 - z^2 - 1,$$

$$r^2y^2 + r^2z + r^2 - y^4 - y^2z^2 + z^2 - z - 2,$$

$$xy + z^2 - 1$$

と制約条件 y になっています。

Wu の定理によれば、元の方程式系の解は、制約条件がゼロにならないという条件の元での特性多項式のゼロ点の集合になります。つまり、最初の組から、$y \neq 0$ の条件の元で $\{r^2+z^3-z^2-1, \dots\}$ の解を求めればよいことになります。これらの解は、($y(r^2 + z^3 - z^2 - 1) \neq 0$ という条件の元での) 二番目の特性多項式の解を合わせたものが元々の方程式の解全部を与えます。

もし、

```
on trwu;
```

とすれば、アルゴリズムがどのように動いているのかについての情報を出力します。この出力から、基本集合の選び方や、特性多項式の計算の詳細について知ることが出来ます。

第二引数 (変数のリスト) は省略することが出来ます。省略された場合には、入力された方程式に含まれるすべての変数が選択されます。また、それらの変数の順序は不定です。

第61章　XCOLOR: 非アーベルゲージ理論における色因子の計算

A.Kryukov

Institute for Nuclear Physics
Moscow State University
119899, Moscow, RUSSIA

E-mail: *kryukov@npi.msu.su*

Phone/Fax: (095) 939-0397

61.1　要約

xCOLOR プログラムは、非アーベルゲージ理論における色因子の計算を行うことを意図しています。これは、Cvitanovich のアルゴリズム (文献 [1]) をインプリメントしたものです。文献 [2] での “COLOR” プログラムと比べて多くの改善を行っています。プログラムはシンボリックモードで書かれ、[2] の版に比べて 10 倍以上高速になっています。

61.2　コマンドの説明

最初に

```
load xcolor;
```

として、パッケージをロードします。

61.2.1　SUdim コマンド

構文: SUdim < 任意の式 >;

SU 群の次数を決めます。通常は 3、つまり SU(3) です。

61.2.2 SpTT コマンド

構文: SpTT < 任意の式 >;

正規化係数 A ($Sp(T_iT_j) = A\Delta_{i,j}$) を決めます。通常は 1/2 です。

61.2.3 QG 演算子

構文: QG(inQuark,outQuark,Gluon)

クオークとグルーオンの交点を記述します。パラメータは任意の変数名で構いません。第一引数と第二引数はそれぞれ、入力と出力のクオークを表します。第三引数はグルーオンです。

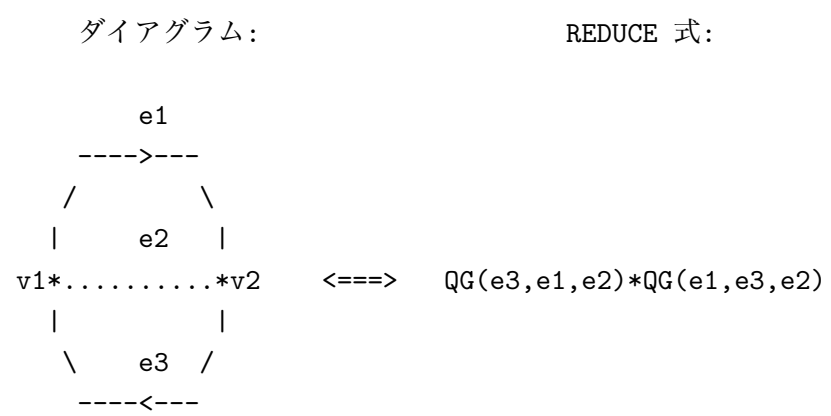
61.2.4 G3 演算子

構文: G3(Gluon1,Gluon2,Gluon3)

三つのグルーオンの交点を記述します。パラメータは任意の変数名で構いません。グルーオンの順序は時計回りの方向で付けます。

61.2.5 例

QG と G3 演算子を使って、色空間でのダイアグラムをこれらの演算子の積として与えます。例えば、



ここで: ---> クオーク

..... グルーオン

詳細については [2] を参照して下さい。

61.3 参考文献

- [1] P.Cvitanovic, Phys. Rev. **D14**(1976), p.1536.
- [2] A.Kryukov and A.Rodionov, Comp. Phys. Comm., **48**(1988), pp.327–334.

第62章　XIDEAL: 外積代数のGröbner基底

David Hartley
Institute for Algorithms and Scientific Computing
GMD — German National Research Center for Information Technology
D–53754 St
Augustin, Germany
Email: *Hartley@gmd.de*
St Augustin
Germany
and
Philip A. Tuckey
Laboratoire de Physique Moleculaire
UFR des Sciences et Techniques
Universite de Franche-Comte
25030 Besancon
France
Email: *pat@rs3.univ-fcomte.fr*

XIDEAL は Gröbner 基底の方法を外積代数に拡張したものです。

XIDEAL は左イデアルのメンバーシップ問題を解く為の Gröbner 基底、Gröbner 左イデアル基底、を計算します。次数つきのイデアル、全ての形式が次数に関して同次である場合、に対しては左イデアルと右イデアルは同じになります。さらに、生成元が全て同次であれば、次数なしと次数つきの Gröbner 基底は同じになります。この場合、もし望むならば、XIDEAL は Gröbner 基底の計算においてある最大次数で打ち切ることによって計算時間を節約できます。XIDEAL は EXCALC パッケージを使用しています。(16 章)

62.1　演算子

XIDEAL

`XIDEAL` は外積代数の Gröbner 左イデアル基底を計算します。構文は次の通りです。

```
XIDEAL(S:list of forms[,R:integer]):list of forms.
```

`XIDEAL` は `S` によって生成される左イデアルの Gröbner 左イデアル基底を、次数付けられた辞書式順序に基づいて計算します。もし、カーネルの順序を変更しなければ、このようにして計算し

たリストは XMODULOP による簡約に利用することができます。もし S の生成元の集合が次数つきの場合、オプションの引数 R を与えることが出来ます。この時、XIDEAL は R と同じ次数かそれ以下の外微分形式に打ち切ります。これは、大きな式に対しては計算時間と容量を節約します。しかし、このようにして求めた結果を R より大きな次数の外微分形式に使うことは出来ません。スイッチ XSTATS と XFULLREDUCTION を参照して下さい。

XMODULO

XMODULO は外微分形式を左イデアルに関して（唯一な）正規型に簡約します。構文は次の通りです。

```
XMODULO(F:form, S:list of forms):form
```

もしくは

```
XMODULO(F:list of forms, S:list of forms):list of forms.
```

これは、内挿演算子としても利用可能です。

```
F XMODULO S.
```

XMODULO(F,S) はまず最初に S によって生成される左イデアルの Gröbner 基底を計算し、その後 F を簡約します。F は一つの外微分形式であっても、また外微分形式のリストであっても構いません。S は外微分形式のリストです。もし、F がリストの場合には、各々の要素が簡約されます。そしてゼロに簡約されるものは取り除かれます。もし、この関数が一度以上使われた場合、かつ S がこの呼び出しの間に変化していなければ、Gröbner 基底が再度計算されることはありません。もし、S の生成元の集合が次数つきである場合、F の次数 (または F の最大次数) によって打ち切られた Gröbner 基底が計算されます。

XMODULOP

XMODULOP は外微分形式を（必ずしも唯一ではないが）外微分形式の集合に関して、正規型に簡約します。構文は次の通りです。

```
XMODULOP(F:form, S:list of forms):form
```

もしくは

```
XMODULOP(F:list of forms, S:list of forms):list of forms.
```

これは、内挿演算子としても利用可能です。

```
F XMODULOP S.
```

XMODULOP(F,S) は F を外微分形式の集合 S、これは必ずしも Gröbner 基底ではない、に関して簡約を行ないます。F は一つの外微分形式であっても、また外微分形式のリストであっても構いません。また、S は外微分形式のリストです。この関数は XIDEAL と共に使って、XMODULO と同じ計算を行なうことが出来ます。次数つきの集合 S から生成されたイデアル中の一つの形式 F に対して、F XMODULO S は F XMODULOP XIDEAL(S,EXDEGREE F) と同じです。

62.2 スイッチ

XFULLREDUCE

ON XFULLREDUCE は XIDEAL と XMODULO が簡約された (しかしながら必ずしも正規化されていない) Gröbner 基底を計算します。これは、引き継づく簡約を早く計算できるようにします。また、Gröbner 基底が唯一（倍数を除いて）であることを保証します。OFF XFULLREDUCE はこの機能を停止ます。こうすることで、Gröbner 基底自身の計算を早く行なうことができます。XFULLREDUCE は通常 ON です。

XSTATS

ON XSTATS は計算時間に関する情報を出力します。XIDEAL が実行されると、入力されたリストから取り出されたそれぞれの式に対してハッシュ記号 (#) が出力されます。引き継づいて、キャレット記号 (^) とドル記号 ($) の列が出力されます。それぞれのキャレットはウェッジ積によって得られた新しい基底要素を表しています。ドル記号は S-多項式から得られた新しい基底要素を表しています。最後に計算の要約が出力されます。XSTATS は通常 OFF になっています。

62.3 例

EXCALC と XIDEAL ロードされているとします。またスイッチは通常の設定であるとし、次の変数を外微分形式として宣言します。

```
pform x=0,y=0,z=0,t=0,f(i)=1,h=0,hx=0,ht=0;
```

可換な多項式環では、一つの多項式はそれ自身 Gröbner 基底になっています。しかし外積代数では、ゼロ因子を持つ為に、これは正しくありません。このために次のような簡約が可能です。

```
xideal {d x^d y - d z^d t};

     {d T^d Z + d X^d Y,

      d X^d Y^d Z,

      d T^d X^d Y}
```

```
f(3)^f(4)^f(5)^f(6)
  xmodulo {f(1)^f(2) + f(3)^f(4) + f(5)^f(6)};

     0
```

熱方程式 $h_{xx} = h_t$ は次の外微分形式で表される。

```
S := {d h - ht*d t - hx*d x,
      d ht^d t + d hx^d x,
      d hx^d t - ht*d x^d t};
```

XMODULO は外微分形式が外微分のもとで閉じていることをチェックする為に使われます。

```
d S xmodulo S;

     {}
```

次数無しの左および右イデアルはもはや同じではない。

```
d t^(d z+d x^d y) xmodulo {d z+d x^d y};

     0

(d z+d x^d y)^d t xmodulo {d z+d x^d y};

      - 2*d t^d z
```

高次の形式は低次の形式を簡約することが出来ます。

```
d x xmodulo {d y^d z + d x,d x^d y + d z};

     0
```

0-形式を含む任意の形式は全イデアルを生成します。

```
xideal {1 + f(1) + f(1)^f(2) + f(2)^f(3)^f(4)};

     {1}
```

第63章　ZEILBERG:不定和分および定和分

Wolfram Koepf and Gregor Stölting
Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin–Dahlem, Germany
e–mail: *Koepf@zib.de*

63.1　はじめに

このパッケージはそれぞれ超幾何級数の不定和分と定和分を計算する Gosper アルゴリズム[1] と Zeilberger のアルゴリズムの注意深いインプリメントです。さらに、最初の著者によって与えられたアルゴリズムの拡張を含んでいます。もし、a_k/a_{k-1} が k に関して有理関数になる場合、超幾何関数と呼ばれます。典型的な超幾何関数には、冪乗や階乗、Γ 関数、二項係数およびシフトされた階乗 (Pochhammer 記号) 等があります。

上で述べた Gosper および Zeilberger のアルゴリズムの拡張は、冪乗や階乗、Γ 関数、二項係数およびシフトされた階乗 (Pochhammer 記号) の積の比で表される項を含むような級数の和に対して有効です。

63.2　GOSPER の和分演算子

`gosper` 演算子は Gosper アルゴリズムのインプリメンテーションです。

- `gosper(a,k)` は和分の閉じた式を決定します。それが閉じた解を返さない場合、閉じた解は存在しません。
- `gosper(a,k,m,n)` は Gosper のアルゴリズムを使って

$$\sum_{k=m}^{n} a_k$$

を決定します。これは Gosper のアルゴリズムが適用可能な場合のみ成功します。

[1] `sum` パッケージは Gosper アルゴリズムの部分的なインプリメントを含んでいます。

例:

```
gosper((-1)^(k+1)*(4*k+1)*factorial(2*k)/
   (factorial(k)*4^k*(2*k-1)*factorial(k+1)),k);

              k
      - ( - 1) *factorial(2*k)
-------------------------------------
  2*k
 2   *factorial(k + 1)*factorial(k)

gosper(binomial(k,n),k);

 (k + 1)*binomial(k,n)
-----------------------
         n + 1
```

63.3 EXTENDED_GOSPER 演算子

extended_gosper 演算子は Gosper のアルゴリズムの拡張バージョンのインプリメンテーションです。

- extended_gosper(a,k) は a_k の和分 g_k 、$h_k - h_{k-m} = a_k$ を満たすような m が存在するかを決定します。ここで、h_k は m-重超幾何級数で、h_k/h_{k-m} は k に関して有理関数です。もしこのような解が存在しなければ、この和分に対する閉じた解は存在しません。
- extended_gosper(a,k,m) m-重の和分、a_k に対して、$h_k - h_{k-m} = a_k$ を満たすような h_k を決定します。

例:

```
extended_gosper(binomial(k/2,n),k);

                   k                           k - 1
 (k + 2)*binomial(---,n) + (k + 1)*binomial(-------,n)
                   2                             2
-------------------------------------------------------
                       2*(n + 1)

extended_gosper(k*factorial(k/7),k,7);

                   k
(k + 7)*factorial(---)
                   7
```

63.4 SUMRECURSION 演算子

sumrecursion 演算子は (高速な)Zeilberger アルゴリズムのインプリメンテーションです。

- sumrecursion(f,k,n) は
$$\mathtt{sum(n)} = \sum_{k=-\infty}^{\infty} f(n,k)$$
に対して、n に関する holonomic な再帰方程式を決定します。結果の式は 0 と等しい式が得られます。

- sumrecursion(f,k,n,j) は次数 j の holonomic な再帰方程式を探します。j が大きすぎる場合、再帰方程式が唯一に決まらない可能性があります。しかしある一つの解のみが返されるという点に注意して下さい。

```
sumrecursion(binomial(n,k),k,n);
```

```
2*sum(n - 1) - sum(n)
```

63.5 HYPERRECURSION 演算子

一般化された超幾何関数に対する再帰方程式が存在する場合、以下のいずれかを使うことができます。

- hyperrecursion(upper,lower,x,n) は
$${}_pF_q\left(\left.\begin{matrix} a_1, & a_2, & \cdots, & a_p \\ b_1, & b_2, & \cdots, & b_q \end{matrix}\right| x\right),$$
に対して、n に関する holonomic な再帰方程式を決定します。ここで、upper= $\{a_1, a_2, \ldots, a_p\}$ は分子パラメータのリストで、lower= $\{b_1, b_2, \ldots, b_q\}$ は n に依存する分母パラメータのリストです。

- hyperrecursion(upper,lower,x,n,j) ($j \in N$) は次数 j の holonomic な再帰方程式を探します。この演算子は、extended_sumrecursion を自動的には使用しません。

```
hyperrecursion({-n,b},{c},1,n);
```

```
(b - c - n + 1)*sum(n - 1) + (c + n - 1)*sum(n)
```

超幾何関数を超幾何関数の記法で与えられた場合には、hyperrecursion を使う方 sumrecursion を使うよりより自然です。

さらに REDUCE の演算子 hyperterm(upper,lower,x,k) は分子パラメータ upper= $\{a_1, a_2, \ldots, a_p\}$, と分母パラメータ lower= $\{b_1, b_2, \ldots, b_q\}$ をもつ超幾何級数項
$$\frac{(a_1)_k \cdot (a_2)_k \cdots (a_p)_k}{(b_1)_k \cdot (b_2)_k \cdots (b_q)_k \, k!} x^k$$
をかえします。

63.6 HYPERSUM 演算子

hypersum 演算子によって、超幾何級数の和は、拡張された Zeilberger のアルゴリズムによって得られた再帰方程式が二つの項しか含んでいない場合には、直接閉形式で出力されます。

- hypersum(upper,lower,x,n) は、${}_pF_q\left(\begin{array}{c} a_1,\ a_2,\ \cdots,\ a_p \\ b_1,\ b_2,\ \cdots,\ b_q \end{array}\middle| x\right)$ に対して、閉形式での表現を決定します。ここで、upper= $\{a_1, a_2, \ldots, a_p\}$, は分子パラメータのリストで、lower= $\{b_1, b_2, \ldots, b_q\}$ は分母パラメータのリストです。結果は、n に関する超幾何級数で表現されます。

  もし、結果が長さ m のリストであるならば、m-重対称と呼びます。これは、次のように解釈されます。この j 番目の部分は n が $n = mk + j - 1$ $(k \in N_0)$ の場合には有効な解となる。とくに、結果のリストが二つの項を含んでいるならば、最初の部分は偶数の n に対して解となっており、二番目の部分は奇数の n に対して解となっている。

```
hypersum({a,1+a/2,c,d,-n},{a/2,1+a-c,1+a-d,1+a+n},1,n);

 pochhammer(a - c - d + 1,n)*pochhammer(a + 1,n)
-------------------------------------------------
 pochhammer(a - c + 1,n)*pochhammer(a - d + 1,n)

hypersum({a,1+a/2,d,-n},{a/2,1+a-d,1+a+n},-1,n);

  pochhammer(a + 1,n)
-------------------------
 pochhammer(a - d + 1,n)
```

演算子 togamma は、階乗-Γ-二項式の-Pochhammer 記法の中で与えられた表現を純粋な Γ 関数表現に変換します:

```
togamma(hypersum({a,1+a/2,d,-n},{a/2,1+a-d,1+a+n},-1,n));

 gamma(a - d + 1)*gamma(a + n + 1)
-----------------------------------
 gamma(a - d + n + 1)*gamma(a + 1)
```

63.7 SUMTOHYPER 演算子

sumtohyper 演算子を使うことによって、階乗-Γ-関数-Pochhammer 記法を使った和分は超幾何級数の表現に変換されます。

- sumtohyper(f,k) は $\sum_{k=-\infty}^{\infty} f_k$ を超幾何級数で表した表現を決定します。つまり、その出力は

$$\sum_{k=-\infty}^{\infty} f_k = c \cdot {}_pF_q \left(\left. \begin{matrix} a_1, & a_2, & \cdots, & a_p \\ b_1, & b_2, & \cdots, & b_q \end{matrix} \right| x \right),$$

に対応して、c*hypergeometric(upper,lower,x) が得られます。ここで、upper= $\{a_1, a_2, \ldots, a_p\}$ および lower= $\{b_1, b_2, \ldots, b_q\}$ は分子と分母のパラメータです。

例:

```
sumtohyper(binomial(n,k)^3,k);
```

```
hypergeometric({ - n, - n, - n},{1,1},-1)
```

63.8 簡約演算子

式 a_k が超幾何項であるかを判定するため、a_k/a_{k-1} が k に関して有理関数であるかを調べなければなりません。冪乗や階乗、Γ 関数、二項係数、Pochhammer 記号を含んだ式が超幾何級数項であるかを決定するために、次の簡約演算子が使用できます。

- simplify_gamma(f) は有理式や巾項、Γ 関数のみを含んだ式 f を簡約します。
- simplify_combinatorial(f) は冪乗や階乗、Γ 関数、二項係数、Pochhammer 記号を含んだ式 f を、簡約します。これは、階乗や二項係数、Pochhammer 記号を Γ 関数に変換し、その結果に simplify_gamma を適用させて簡約します。結果が有理的でない場合、それは Γ 関数で表されます。

階乗を使って表した方が好ましい場合

- gammatofactorial (rule) は $\Gamma(x) \to (x-1)!$ を使って、Γ 関数を階乗で表します。
- simplify_gamma2(f) は Γ 関数の二倍公式を使って f を簡約します。
- simplify_gamman(f,n) は Γ 関数の積公式を使って f を簡約します。

simplify_combinatorial(f) を使用することは、任意の巾の積や、階乗、Γ 関数、二項係数それに Pochhammer 記号の商が有理的であるかを判定する安全な方法です。

例:

```
simplify_gamma2(gamma(2*n)/gamma(n));
```

```
  2*n        2*n + 1
 2   *gamma(---------)
                2
-----------------------
      2*sqrt(pi)
```

第64章　ZTRANS: Z-変換

Wolfram Koepf and Lisa Temme

Konrad–Zuse–Zentrum für Informationstechnik Berlin
Takustrasse 7
D–14195 Berlin–Dahlem, Germany

e–mail: *Koepf@zib.de*

列 $\{f_n\}$ に対する Z-変換はラプラス変換の離散化したもので、$\mathcal{Z}\{f_n\} = F(z) = \sum_{n=0}^{\infty} f_n z^{-n}$ で定義されます。この数列は円 $|z| = |z_0| = \limsup_{n\to\infty} \sqrt[n]{|f_n|}$ の外部で収束します。ラプラス変換が微分方程式を解く時に使われるのと同様に、Z-変換は差分方程式を解く為に使われます。

64.1　Z-変換

構文: `ztrans(`f_n`, n, z)`　　ここで f_n は式, そして n,z は識別子.

このパッケージは次に挙げる関数 f_nおよびそれらの組み合わせたものに対する Z-変換が計算できます。

1	$e^{\alpha n}$	$\frac{1}{(n+k)}$
$\frac{1}{n!}$	$\frac{1}{(2n)!}$	$\frac{1}{(2n+1)!}$
$\frac{\sin(\beta n)}{n!}$	$\sin(\alpha n+\phi)$	$e^{\alpha n}\sin(\beta n)$
$\frac{\cos(\beta n)}{n!}$	$\cos(\alpha n+\phi)$	$e^{\alpha n}\cos(\beta n)$
$\frac{\sin(\beta(n+1))}{n+1}$	$\sinh(\alpha n+\phi)$	$\frac{\cos(\beta(n+1))}{n+1}$
$\cosh(\alpha n+\phi)$	$\binom{n+k}{m}$	

その他の組合せ

線形性 $\mathcal{Z}\{af_n + bg_n\} = a\mathcal{Z}\{f_n\} + b\mathcal{Z}\{g_n\}$

n との積 $\mathcal{Z}\{n^k \cdot f_n\} = -z\frac{d}{dz}\left(\mathcal{Z}\{n^{k-1} \cdot f_n, n, z\}\right)$

λ^n との積 $\mathcal{Z}\{\lambda^n \cdot f_n\} = F\left(\frac{z}{\lambda}\right)$

シフト方程式 $\mathcal{Z}\{f_{n+k}\} = z^k\left(F(z) - \sum_{j=0}^{k-1} f_j z^{-j}\right)$

和 $\mathcal{Z}\left\{\sum_{k=0}^{n} f_k\right\} = \frac{z}{z-1} \cdot \mathcal{Z}\{f_n\}$

$\mathcal{Z}\left\{\sum_{k=p}^{n+q} f_k\right\}$ これらの組合せ

ただし k,$\lambda \in \mathbf{N}-\{0\}$; および a,b は変数、そして p,$q \in \mathbf{Z}$ もしくは
n の関数; そして α, β と ϕ は角度。

64.2 逆 Z-変換

正則関数のローラン係数の計算は、Z-変換の逆公式から得られる。

もし $F(z)$ が領域 $|z| > \rho$ で正則なら、$\mathcal{Z}\{f_n\} = F(z)$ なる数列 $\{f_n\}$ が存在して、

$$f_n = \frac{1}{2\pi i}\oint F(z)z^{n-1}dz$$

となる。

構文: `invztrans(`$F(z)$`, z, n)` ここで $F(z)$ は式,
そして z,n は識別子.

このパッケージは、以下のリストにあげる関数 $F(z)$に加えて、分母は $\mathbf{Q}$ で因数分解可能であるような任意の有理関数の逆 Z-変換が計算できる。

$\sin\left(\frac{\sin(\beta)}{z}\right)e^{\left(\frac{\cos(\beta)}{z}\right)}$ $\cos\left(\frac{\sin(\beta)}{z}\right)e^{\left(\frac{\cos(\beta)}{z}\right)}$

$\sqrt{\frac{z}{A}}\sin\left(\sqrt{\frac{z}{A}}\right)$ $\cos\left(\sqrt{\frac{z}{A}}\right)$

$\sqrt{\frac{z}{A}}\sinh\left(\sqrt{\frac{z}{A}}\right)$ $\cosh\left(\sqrt{\frac{z}{A}}\right)$

$z\ \log\left(\frac{z}{\sqrt{z^2-Az+B}}\right)$ $z\ \log\left(\frac{\sqrt{z^2+Az+B}}{z}\right)$

$$\arctan\left(\frac{\sin(\beta)}{z+\cos(\beta)}\right)$$

ここで、k,$\lambda \in \mathbf{N}-\{0\}$ そして A,B は変数 $(B>0)$ である。 また、 α,β, および ϕ は角度です。

例:

```
ztrans(sum(1/factorial(k),k,0,n),n,z);
```

```
  1/z
 e   *z
--------
 z - 1
```

```
invztrans(z/((z-a)*(z-b)),z,n);
```

```
  n    n
 a  - b
---------
  a - b
```

64.3 Z-変換の応用

64.3.1 差分方程式の解

ラプラス変換を用いて微分方程式を解くことが出来るのと同様に、Z-変換を用いて差分方程式を解くことが出来ます。k-次のオーダーの差分方程式

$$f_{n+k} + a_1 f_{n+k-1} + \ldots + a_k f_n = g_n \tag{64.1}$$

とその初期値 $f_0 = h_0$, $f_1 = h_1$, ..., $f_{k-1} = h_{k-1}$ (ただし h_j は与えられた数値) が与えられると、次のようにして解くことが出来ます。

もし、係数 $a_1, \ldots, a_k$ が定数であれば、(64.1) の Z-変換はシフト方程式を使って求めることが出来ます。得られた方程式は $\mathcal{Z}\{f_n\}$ の線形方程式です。これを逆 Z-変換することによって、(64.1) の解が求まります。係数 $a_1, \ldots, a_k$ が n の多項式の場合、(64.1) の Z-変換したものは、$\mathcal{Z}\{f_n\}$ の差分方程式になります。この差分方程式を解くことが出来れば、逆 Z-変換を行なうことによって、(64.1) の解が求まります。この方法の例が §64.4 にあります。

64.4 例

64.4.1 Z-変換の例

```
1: ztrans((-1)^n*n^2,n,z);

    z*( - z + 1)
---------------------
  3      2
 z  + 3*z  + 3*z + 1

2: ztrans(cos(n*omega*t),n,z);

   z*(cos(omega*t) - z)
---------------------------
                     2
 2*cos(omega*t)*z - z  - 1

3: ztrans(cos(b*(n+2))/(n+2),n,z);

                                  z
z*( - cos(b) + log(------------------------------)*z)
                                       2
                    sqrt( - 2*cos(b)*z + z  + 1)

4: ztrans(n*cos(b*n)/factorial(n),n,z);

  cos(b)/z       sin(b)                  sin(b)
 e        *(cos(--------)*cos(b) - sin(--------)*sin(b))
                   z                       z
-------------------------------------------------------
                            z
5: ztrans(sum(1/factorial(k),k,0,n),n,z);

  1/z
 e   *z
--------
 z - 1

6: operator f$

7: ztrans((1+n)^2*f(n),n,z);
```

```
                          2
df(ztrans(f(n),n,z),z,2)*z  - df(ztrans(f(n),n,z),z)*z
+ ztrans(f(n),n,z)
```

64.4.2 逆 Z-変換の例

```
8: invztrans((z^2-2*z)/(z^2-4*z+1),z,n);

              n       n                n
 (sqrt(3) - 2) *( - 1)  + (sqrt(3) + 2)
-----------------------------------------
                   2

9: invztrans(z/((z-a)*(z-b)),z,n);

  n    n
 a  - b
---------
  a - b

10: invztrans(z/((z-a)*(z-b)*(z-c)),z,n);

  n      n      n      n      n      n
 a *b - a *c - b *a + b *c + c *a - c *b
----------------------------------------
  2      2        2      2    2        2
 a *b - a *c - a*b  + a*c  + b *c - b*c

11: invztrans(z*log(z/(z-a)),z,n);

  n
 a *a
-------
 n + 1

12: invztrans(e^(1/(a*z)),z,n);

        1
-----------------
  n
 a *factorial(n)
```

```
13: invztrans(z*(z-cosh(a))/(z^2-2*z*cosh(a)+1),z,n);

cosh(a*n)
```

64.4.3 差分方程式

同次方程式

同次線形差分方程式、

$$f_{n+5} - 2f_{n+3} + 2f_{n+2} - 3f_{n+1} + 2f_n = 0$$

と初期値 $f_0 = 0,\ f_1 = 0,\ f_2 = 9,\ f_3 = -2,\ f_4 = 23$ を考える。左辺の Z-変換は $F(z) = P(z)/Q(z)$ となる。ここで、$P(z) = 9z^3 - 2z^2 + 5z$ そして、$Q(z) = z^5 - 2z^3 + 2z^2 - 3z + 2 = (z-1)^2(z+2)(z^2+1)$。この逆変換は、

$$f_n = 2n + (-2)^n - \cos\frac{\pi}{2}n_{\circ}$$

以下の REDUCE のセッションはこの問題を如何に解くかを示している。([12], p. 651, 例 1 を参照).

```
14: operator f$ f(0):=0$ f(1):=0$ f(2):=9$ f(3):=-2$ f(4):=23$


20: equation:=ztrans(f(n+5)-2*f(n+3)+2*f(n+2)-3*f(n+1)+2*f(n),n,z);

                                5                          3
equation := ztrans(f(n),n,z)*z  - 2*ztrans(f(n),n,z)*z

                                    2
            + 2*ztrans(f(n),n,z)*z  - 3*ztrans(f(n),n,z)*z

                                  3       2
            + 2*ztrans(f(n),n,z) - 9*z  + 2*z  - 5*z


21: ztransresult:=solve(equation,ztrans(f(n),n,z));
```

```
                                          2
                                   z*(9*z  - 2*z + 5)
ztransresult := {ztrans(f(n),n,z)=---------------------------}
                                    5      3      2
                                   z  - 2*z  + 2*z  - 3*z + 2
```

```
22: result:=invztrans(part(first(ztransresult),2),z,n);
```

```
                    n    n       n    n
          2*( - 2)  - i *( - 1)  - i  + 4*n
result := ----------------------------------
                          2
```

非同次差分方程式

次の非同次線形差分方程式、

$$f_{n+2} - 4f_{n+1} + 3f_n = 1$$

と初期値 $f_0 = 0$, $f_1 = 1$ を考える。

$$\begin{aligned} F(z) &= \mathcal{Z}\{1\}\left(\frac{1}{z^2-4z+3} + \frac{z}{z^2-4z+3}\right) \\ &= \frac{z}{z-1}\left(\frac{1}{z^2-4z+3} + \frac{z}{z^2-4z+3}\right) \end{aligned}$$

この逆変換は、

$$f_n = \frac{1}{2}\left(\frac{3^{n+1}-1}{2} - (n+1)\right)。$$

以下の REDUCE のセッションはこの問題を如何に解くかを示している。([12], p. 651, 例 2 を参照).

```
23: clear(f)$ operator f$ f(0):=0$ f(1):=1$
```

```
27: equation:=ztrans(f(n+2)-4*f(n+1)+3*f(n)-1,n,z);
```

```
                                3                        2
equation := (ztrans(f(n),n,z)*z  - 5*ztrans(f(n),n,z)*z

                                                        2
    + 7*ztrans(f(n),n,z)*z - 3*ztrans(f(n),n,z) - z )/(z - 1)
```

```
28: ztransresult:=solve(equation,ztrans(f(n),n,z));
```

```
                                      2
                                     z
result := {ztrans(f(n),n,z)=----------------------}
                               3      2
                              z  - 5*z  + 7*z - 3
```

```
29: result:=invztrans(part(first(ztransresult),2),z,n);
```

```
              n
           3*3  - 2*n - 3
result := ----------------
                 4
```

Z-変換で差分方程式になる例

次の差分方程式

$$(n+1)\cdot f_{n+1} - f_n = 0$$

と初期値 $f_0 = 1,\ f_1 = 1$ は、Z-変換したものが $\mathcal{Z}\{f_n\}$ に対して差分方程式になる。

```
30: clear(f)$ operator f$ f(0):=1$ f(1):=1$
```

```
34: equation:=ztrans((n+1)*f(n+1)-f(n),n,z);
```

```
                                          2
equation :=  - (df(ztrans(f(n),n,z),z)*z  + ztrans(f(n),n,z))
```

```
35: operator tmp;
```

```
36: equation:=sub(ztrans(f(n),n,z)=tmp(z),equation);
```

```
                                 2
equation :=  - (df(tmp(z),z)*z  + tmp(z))
```

```
37: load(odesolve);
```

```
38: ztransresult:=odesolve(equation,tmp(z),z);
```

```
                           1/z
ztransresult := {tmp(z)=e   *arbconst(1)}
```

```
39: preresult:=invztrans(part(first(ztransresult),2),z,n);
```

```
              arbconst(1)
preresult := --------------
              factorial(n)

40: solve({sub(n=0,preresult)=f(0),sub(n=1,preresult)=f(1)},
arbconst(1));

{arbconst(1)=1}

41: result:=preresult where ws;

                1
result := --------------
           factorial(n)
```

関連図書

[1] Milton Abramowitz and Irene A. Stegun, editors. *Handbook of Mathematical Functions*. Dover Publications, New York, 1972.

[2] V.S. Adamchik and O.I. Marichev. The algorithm for calculating integrals of hypergeometric type functions and its realization in reduce system. In *ISSAC 90:Symbolic and Algebraic Computation*. Addison-Wesley Publishing Company, 1990.

[3] B. Amrhein and O. Gloor. The fractal walk. In B. Buchberger an F. Winkler, editor, *Gröbner Bases and Applications*, volume 251 of *LMS*, pages 305 –322. Cambridge University Press, February 1998.

[4] B. Amrhein, O. Gloor, and W. Kuechlin. How fast does the walk run? In A. Carriere and L. R. Oudin, editors, *5th Rhine Workshop on Computer Algebra*, volume PR 801/96, pages 8.1 – 8.9. Institut Franco–Allemand de Recherches de Saint–Louis, January 1996.

[5] B. Amrhein, O. Gloor, and W. Kuechlin. Walking faster. In J. Calmet and C. Limongelli, editors, *Design and Implementation of Symbolic Computation Systems*, volume 1128 of *Lecture Notes in Computer Science*, pages 150 –161. Springer, 1996.

[6] Yu.A. Brychkov A.P. Prudnikov and O.I. Marichev. *Integrals and Series, Volume 3: More Special Functions*. Gordon and Breach Science Publishers, 1990.

[7] George A. Baker(Jr.) and Peter Graves-Morris. *Padé Approximants, Part I: Basic Theory, (Encyclopedia of mathematics and its applications, Vol 13, Section: Mathematics of physics)*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[8] T. Becker and V. Weispfenning. *Groebner Bases*. Springer, 1993.

[9] Carl M. Bender and Steven A. Orszag. *Advanced Mathematical Methods for Scientists and Engineers*. McGraw-Hill, 1978.

[10] W. Boege, R. Gebauer, and H. Kredel. Some examples for solving systems of algebraic equations by calculating Groebner bases. *J. Symbolic Computation*, 2(1):83–98, March 1986.

[11] R. J. Bradford, A. C. Hearn, J. A. Padget, and E. Schrüfer. Enlarging the REDUCE domain of computation. In *Proceedings of SYMSAC '86*, pages 100–106, 1986.

[12] I.N. Bronstein and K.A. Semedjajew. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, Thun und Frankfurt(Main), 1981.

[13] B. Buchberger. Groebner bases: An algorithmic method in polynomial ideal theory. In N. K. Bose, editor, *Progress, directions and open problems in multidimensional systems theory*, pages 184–232. Dordrecht: Reidel, 1985.

[14] B. Buchberger. Applications of Groebner bases in non-linear computational geometry. In R. Janssen, editor, *Trends in Computer Algebra*, pages 52–80. Berlin, Heidelberg, 1988.

[15] S. Collart, M. Kalkbrener, and D. Mall. Converting bases with the gröbner walk. *J. Symbolic Computation*, 24:465 – 469, 1997.

[16] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra, Systems and Algorithms for Algebraic Computation*. Academic Press, 1989.

[17] James Harold Davenport. On the integration of algebraic functions. In *Lecture Notes in Computer Science*, volume 102. Springer Verlag, 1981.

[18] James W. Eastwood. Orthovec: A REDUCE program for 3-D vector analysis in orthogonal curvilinear coordinates. *Comp. Phys. Commun.*, 47(1):139–147, October 1987.

[19] James W. Eastwood. ORTHOVEC: version 2 of the REDUCE program for 3-D vector analysis in orthogonal curvilinear coordinates. *Comp. Phys. Commun.*, 64(1):121–122, April 1991.

[20] K. H. Ebert and P. Deuflhard. Modelling of chemical reaction systems. In W. Jaeger, editor, *Springer Ser. Chem. Phys*, volume 18. Springer Verlag, 1981.

[21] A. R. Edmonds. *Angular Momentum in Quantum Mechanics*. Princeton University Press, 1957.

[22] Bruce W. Chat... [et al.]. *Maple (Computer Program)*. Springer-Verlag, 1991.

[23] J. C. Faugère, P. Gianni, D. Lazard, and T. Mora. Efficient computation of zero-dimensional Groebner bases by change of ordering. Technical report, 1989.

[24] Sandra Fillebrown. Faster computation of bernoulli numbers. *Journal of Algorithms*, 13:431–445, 1992.

[25] Karin Gatermann. Symmetry: A REDUCE package for the computation of linear representations of groups. Technical report, ZIB, Berlin, 1992.

[26] Rüdiger Gebauer and H. Michael Möller. On an installation of Buchberger's algorithm. *J. Symbolic Computation*, 6(2 and 3):275–286, 1988.

[27] Anthony C. Hearn. *REDUCE User's Manual 3.7*. 1999.

[28] Roger A. Horn and Charles A. Johnson. *Matrix Analysis*. Cambridge University Press, 1990.

[29] D.J. Jeffery and A.D. Rich. The evaluation of trigonometric integrals avoiding spurious discontinuities. *appearing in ACM Trans. Math Software*.

[30] Aleksandr J. Khinchin. *Continued Fractions*. University of Chicago Press, 1964.

[31] Wolfram Koepf. Power series in computer algebra. *Journal of Symbolic Computation*, 13:581–603, 1992.

[32] H. Kredel and V. Weispfenning. Computing dimension and independent sets for polynomial ideals. *J. Symbolic Computation*, 6(1):231–247, November 1988.

[33] Heinz Kredel. Admissible termorderings used in computer algebra systems. *SIGSAM Bulletin*, 22(1):28–31, January 1988.

[34] Landolt-Boernstein. *Zahlenwerte und Funktionen aus Naturwissenschaften und Technik*. Springer, 1968.

[35] H. Melenk, H. M. Möller, and W. Neun. On Gröbner bases computation on a supercomputer using REDUCE. Preprint SC 88-2, Konrad-Zuse-Zentrum für Informationstechnik Berlin, January 1988.

[36] O. Patashnik R. L. Graham, D. E. Knuth. *Concrete Mathematics*. Addison-Wesley Publishing Company, 1989.

[37] Matt Rebbeck. A linear algebra package for REDUCE. Technical report, ZIB , Berlin, 1994.

[38] J. H. Wilkinson & C. Reinsch. *Linear Algebra (volume II)*. Springer-Verlag, 1971.

[39] D.E.G. Hare R.M. Corless, G.H. Gonnet and D.J. Jeffrey. *On Lambert's W Function*. Preprint, University of Waterloo, 1992.

[40] J. P. Serre. *Linear Representations of Finite Groups*. Springer, New York, 1977.

[41] M . Speigel. *Vector Analysis*. Scheum Publishing Co., 1959.

[42] E. Stiefel and A. Fässler. *Gruppentheoretische Methoden und ihre Anwendung*. Teubner, Stuttgart, 1979.

[43] T.M.L.Mulders. Algoritmen in de algebra, a seminar on algebraic algorithms, nigmegen. 1993.

[44] T.M.L.Mulders and A.H.M. Levelt. The maple normform and normform packages. 1993.

[45] B. M. Trager. Algebraic factoring and rational function integration. In *Proceedings of SYMSAC '76*, pages 196–208, 1976.

a

索 引