MuPAD: Multi Processing Algebra Data Tool
MuPAD 1.4 デモンストレーションツアー

Frank Postel[1]

1997 年 9 月 24 日

[1] 幸谷智紀/角谷 悟 訳, Ver.0.3:1999 年 10 月 16 日 (土)

概 要

この文書の目的は，数式処理の事例を通して MuPAD 1.4 の基本機能を概観することにあります。内容の部分部分は，Dr.C.Heckler と G.Rüscher(共に Paderborn 大学，MuPAD グループに所属)との共同作業から生み出されたものです。

目 次

第1章　最初の一歩

多くの数式処理システムに共通する基本事項のデモンストレーションを行います。

1.1　数の演算

以下のようにセッションを初期化して下さい[1]。

```
>> reset();
```

MuPAD は電卓と同じ計算が出来ます。次の入力が行われると

```
>> 1 + 5/2;
```

分数 $\frac{7}{2}$ になります。

$$7/2$$

以下のように，MuPAD は整数と分数数との計算を正確に実行できます。

```
>> (1 + (5/2 * 3)) / (1/7 + 7/9)^2;
```

$$67473/6728$$

更に，MuPAD は非常に大きな数の演算も可能です。数の長さは，あなたが使っているコンピュータの使用可能メモリの量で制限されます。では，1234 の 123 乗を計算してみましょう。

```
>> 1234^123;
```

すると，381 桁の数になります。

```
170515806212727042875059727620626282654302313111068290470529619322183913 83\
486800747136630671706059857264159231455434590057058967067149970908610253 99\
048465147931356173055636699939501046220356820273557577550700832384441477 78\
396026387067042685700404003287042480639680696865587865016699383883388831 98\
045915994284537241460180942971772610762859524340680101441852976627983806 72\
03562799104
```

MuPAD は，239823234324324998432131232 1 という数が素数か否かも答えてくれます。素数とは，1 と自分自身でしか割り切れない数のことです。

[1] (訳注) この文書では，ユーザの入力した部分は行の先頭に>>が付いて示されている。自分で実行する際には，この>>は省いて入力すること。

```
>> isprime(23982323432432499984321312321);
```

```
                                FALSE
```

答えの FALSE というのは，与えられた数が素数ではないという意味です。23982323432432499984321312321 の全ての素因数が見たければ，Factor 関数を使いましょう。

```
>> Factor(23982323432432499984321312321);
```

```
                  3 733 15117701 72140687161773179
```

あなたはこの結果を信用できますか? 出来ないようであれば，素因数を掛け合わせてチェックすることもできます。

```
>> _mult(%);
```

```
                  23982323432432499984321312321
```

見てわかるように，正確に計算できています。しかし「正確な」計算とは一体何を意味しているのでしょうか?たとえば，$\sqrt{56}$ という無理数を入力すると，どういう結果になるのでしょうか?

1.1.1 正確な計算

この問題は次の例で解決することにしましょう。予想通り，MuPAD は $\sqrt{4}$ を正確に計算できます。

```
>> sqrt(4);
```

```
                                  2
```

では，$\sqrt{56}$ がどうなるかを聞いてみましょう。

```
>> sqrt(56);
```

```
                                 1/2
                             2 14
```

ご覧のように，MuPAD は $\sqrt{56}$ を $2\sqrt{14}$ としてくれます。MuPAD では sqrt(14) とは二次方程式 $x^2 = 14$ の正の解 $\sqrt{14}$ を意味します (sqrt 関数参照)。

他にも，$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = e$ という極限計算も可能です。

```
>> limit((1 + 1/n)^n, n=infinity);
```

```
                                exp(1)
```

exp(1) とは，自然対数の底 $e = 2.71828...$ を表わしています (exp 関数参照)。

1.1.2 近似計算 vs 正確な計算

正確な計算とは別に，近似計算をしたいこともあります。例えば，$\sqrt{56}$ の近似値を求めたいときは，float 関数を使います。

```
>> float(sqrt(56));
```

```
                                7.483314773
```

近似値の精度は，グローバル変数[2]DIGITS の値で決まり，デフォルト値は 10 です。

```
>> DIGITS; float(67473/6728);
```

```
                                    10

                                10.02868608
```

DIGITS は浮動小数点計算の有効桁数を与え，その値には 1 から $2^{32}-1$ までの整数を指定できます。

```
>> DIGITS := 100: float(67473/6728); unassign(DIGITS):

10.028686087990487514863258026159334126040428061831153388822829964328180 73\
72175980975029726516052 3186
```

unassign(DIGITS) 命令は DIGITS の値をデフォルト値にリセットします (unassign 命令参照)。

MuPAD では，定数 π と定数 e は正確な演算で使用することが出来ます。

```
>> cos(PI);
```

```
                                    -1
```

```
>> ln(E);
```

```
                                     1
```

これらの定数を浮動小数点数で近似することも可能です。以下の例は，一秒以下で π の値を正確に 1000 桁計算したものです。

```
>> DIGITS:= 1000: float(PI); unassign(DIGITS):

3.14159265358979323846264338327950288419716939937510582097494459230781640 6\
28620899862803482534211706798214808651328230664709384460955058223172535940\
81284811174502841027019385211055596446229489549303819644288109756659334461\
28475648233786783165271201909145648566923460348610454326648213393607260249\
14127372458700660631558817488152092096282925409171536436789259036001133053\
05488204665213841469519415116094330572703657595919530921861173819326117931\
```

[2](訳注)MuPAD での演算・操作全体で有効な変数のこと。

```
05118548074462379962749567351885752724891227938183011949129833673362440656\
64308602139494639522473719070217986094370277053921717629317675238467481846\
76694051320005681271452635608277857713427577896091736371787214684409012249\
53430146549585371050792279689258923542019956112129021960864034418159813629\
77477130996051870721134999999837297804995105973173281609631859502445945534\
69083026425223082533446850352619311881710100031378387528865875332083814206\
17177669147303598253490428755468731159562863882353787593751957781857780532\
17122680661300192787661119590921642019 9
```

長いけど，印象に残るでしょ?

1.2 表現と文

セッションを初期化して下さい。

```
>> reset();
```

前の節では数の正確な演算と浮動小数点演算だけを考えました。ここでは記号計算について考えたいと思います。

「表現 (expression)」を扱ってみましょう。

```
>> sin(3) + f(x)/5;
```

$$\frac{f(x)}{5} + \sin(3)$$

```
>> solve({x + y = 5, x - y = 2}, {x,y});
```

$$\{\{y = 3/2,\ x = 7/2\}\}$$

表現に加えて,「文 (statement)」の列を入力することもできます。文とは，ループ (`for-do`, `while-do`, `repeat-until`)，条件分岐 (`if-then`, `case-of`)，制御文 (`quit`, `break`, `next`)，表現の列，代入文など，通常のプログラミング言語で使われているもののことです。

```
>>n := 1:
  for i from 1 to 10 do
      n := n * ithprime(i);
  end_for:
  n;
```

$$6469693230$$

ここで，`ithprime` 関数は正の整数 i を引数として持ち，その名の通り i 番目の素因数を計算しています。

MuPAD はあらかじめ定義された「データ型 (data type)」を多数持っています。例えば，

```
>> [1, 1/3, 2 + 3*I];
```

```
                              [1, 1/3, 2 + 3 I]
```

という「リスト」は，DOM_LIST という基本データ型のオブジェクトであり，このリストの要素は順に，DOM_INT 型, DOM_RAT 型, DOM_COMPLEX 型のオブジェクトになっています。

オブジェクトのデータ型は domtype 関数で明らかにすることが出来ます。

```
>> domtype(1); domtype(1/3); domtype(2 + 3*I);
```

```
                                   DOM_INT

                                   DOM_RAT

                                 DOM_COMPLEX
```

sin(3) + f(x)/5 という表現は，DOM_EXPR というデータ型になります。

```
>> domtype(sin(3) + f(x)/5);
```

```
                                  DOM_EXPR
```

これは最も一般的なデータ型で，異なるデータ型オブジェクトを組み合わせると，大抵はこのデータ型のオブジェクトになります。

次の章では MuPAD の数学的な事項のデモンストレーションを行いたいと思います。

もし MuPAD のプログラミングに興味があれば，次の章はすっ飛ばして第 3 章に進んで下さい。

第2章　数学

様々な数学への応用に MuPAD が使えるということを全て記述するのは，この文書の範囲を超えてしまいます。

以下の節では，幾つかの数学の領域をちょっと通って，興味津々の読者を「Advanced Domonstration Tour」へ誘うことを目的にしています。この「Advanced ...」という文書はホンのちょっと特定の部分を詳しく記述しています。

2.1　解析学

```
>> reset();
```

有理関数 $f = \frac{(x-1)^2}{x-2} + a$($x$, a は実数) について考えてみましょう。特定区間でグラフを描かずに f を解析するため，その特徴を見極めます。

```
>> f := (x - 1)^2/(x - 2) + a: def := discont(f, x);
```

```
                                  {2}
```

詳細は `discont` 関数を参照して下さい。f の零点は `solve` 関数で求めることが出来ます。

```
>> na:= solve(f, x);
```

```
            {                 2 1/2          2 1/2         }
            {     a   (4 a + a )     (4 a + a )      a     }
            { 1 - - - -------------, ------------- - - + 1 }
            {     2         2              2         2     }
```

では，極値を求めてみましょう。

```
>> f1 := diff(f, x);
```

```
                                    2
                          2 x - 2   (x - 1)
                          ------- - --------
                           x - 2           2
                                  (x - 2)
```

```
>> ep := solve(f1 = 0, x);
```

```
{1, 3}
```

```
>> diff(f, x, x): eval(subs(%, x = 1)), eval(subs(%, x = 3));
```

```
-2, 2
```

この結果より，f は $(-\infty, 2)$ 区間内の $x = 1$ で最大値を，$(2, \infty)$ 区間内の $x = 3$ で最小値を持つことがわかります。$x = 1, 3$ での最大値・最小値は以下のようになります。

```
>> maximum := subs(f, x = 1); minimum := subs(f, x = 3);
```

```
a
```

```
a + 4
```

最後に，f の漸近的な振る舞いを調べてみましょう。f は $x = 2$ で特異点を持ち，左から近づくと $-\infty$ へ，右から近づくと ∞ へ飛んでいきます。

```
>> limit(f, x = 2, Left), limit(f, x = 2, Right);
```

```
-infinity, infinity
```

x が大きくなると，関数 f は漸近的に線型項 $x + a$ に等しくなります。

```
>> series(f, x = infinity);
```

```
                  1   2    4     / 1  \
x + a + - + -- + -- + -- + O| -- |
                  x    2    3    |  4 |
                      x    x     \ x  /
```

ということで，ここでは `limit` 関数と `series` 関数を紹介しました。

a に特定の値を入れることで，`plotfunc` 関数を使ってこの関数のグラフを見ることが出来ます (図 2.1)。

```
>> a := 0: plotfunc(f, x = -5..5);
```

2.2 線型代数

```
>> reset();
```

行列を生成するのに一番簡単な方法は以下のようにすることです。

```
>> M := Dom::Matrix();
```

```
Dom::Matrix(Dom::ExpressionField(id, iszero))
```

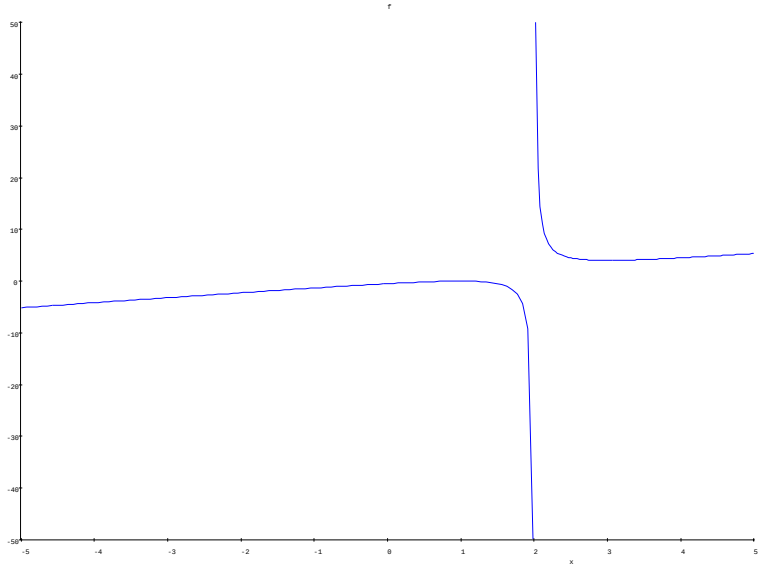


図 2.1: `plotfunc(f, x = -5..5)`

この出力結果については「Advanced Demonstration Tour」の「Linear Algebra」の章で詳しく述べられていますので，ここでは気にしないで下さい。

`Dom::Matrix` 関数を使って，行列の集合を作った，ということだけ知っておいて下さい。行列の要素には任意の表現が使えます。

```
>> A := M(
    [[1, x + y, 1/x^2], [sin(x), 0, cos(x)],
     [x*PI, 1 + I, -x*PI], [-1, 0, 1]]
);
```

$$\begin{pmatrix} 1 & x+y & \frac{1}{x^2} \\ \sin(x) & 0 & \cos(x) \\ x\,PI & 1+I & -x\,PI \\ -1 & 0 & 1 \end{pmatrix}$$

各行は行の要素を並べたリストになっており，更にその行を縦に並べたリストを作り，上の行列を入力しています。この行列の特定の要素を取り出すには，次のようにします。

```
>> A[2,1];
```

```
sin(x)
```

同様にして，特定の部分行列を取り出すこともできます。

```
>> A[1..3, 1..2];
```

```
+-               -+
|     1,   x + y  |
|                 |
|  sin(x),   0    |
|                 |
|   x PI,  1 + I  |
+-               -+
```

行列の付け足しも簡単にできます。4×2 行列を A に追加してみます。

```
>> A . M(4,2);
```

```
+-                              -+
|                    1           |
|     1,   x + y,   --,   0, 0   |
|                    2           |
|                   x            |
|                                 |
|  sin(x),   0,   cos(x), 0, 0   |
|                                 |
|   x PI,  1 + I,  -x PI, 0, 0   |
|                                 |
|    -1,     0,      1,   0, 0   |
+-                              -+
```

行列の演算はどうでしょう。MuPAD の標準的な演算子を使って行列演算が実行出来ます。

```
>> A := M(2, 2, [1, -1], Diagonal);
```

```
+-       -+
|  1,  0  |
|         |
|  0, -1  |
+-       -+
```

```
>> B := M([[1,2], [3,4]]);
```

```
+-      -+
|  1, 2  |
```

```
                    |        |
                    |  3, 4  |
                    +-      -+
```

```
>> (A + B)^3, -2*A + B;
```

```
                +-         -+  +-        -+
                |  50, 50   |  |  -1, 2   |
                |           |, |          |
                |  75, 75   |  |   3, 6   |
                +-         -+  +-        -+
```

以下は，逆行列が存在すれば，その逆行列を計算しています (存在していなければ，FAIL が返されます)。

```
>> 1/B;
```

```
              +-            -+
              |   -2,   1    |
              |              |
              |  3/2, -1/2   |
              +-            -+
```

検算します。

```
>> % * B, B * %;
```

```
               +-       -+  +-       -+
               |  1, 0   |  |  1, 0   |
               |         |, |         |
               |  0, 1   |  |  0, 1   |
               +-       -+  +-       -+
```

しかし，これだけでは線型代数を考えるには不十分です。そのため，MuPAD は linalg と呼ばれる「ライブラリ」を提供しています。

MuPAD の「ライブラリ」とは，文字列操作や Fortran, C のソースコードの生成といったものと同様，数論のような特別な目的向けの関数の集まりを意味します。

linalg ライブラリを構成する関数群を見てみましょう[1]。

```
>> info(linalg);
Library 'linalg': the linear algebra package
Interface:
linalg::addCol,         linalg::addRow,        linalg::adjoint,
linalg::angle,          linalg::basis,         linalg::charMatrix,
linalg::charPolynomial, linalg::cholesky,      linalg::col,
```

[1]原文は一部だけだが，面倒なので全部載っけてある。大した量じゃないし

```
linalg::concatMatrix,   linalg::crossProduct,  linalg::curl,
linalg::delCol,         linalg::delRow,        linalg::det,
linalg::dimen,          linalg::divergence,    linalg::eigenValues,
linalg::eigenVectors,   linalg::expr2Matrix,   linalg::extractMatrix,
linalg::factorLU,       linalg::factorQR,      linalg::freeSet,
linalg::gaussElim,      linalg::gaussJordan,   linalg::grad,
linalg::hermiteForm,    linalg::hessian,       linalg::intBasis,
linalg::inverseLU,      linalg::isHermitian,   linalg::isOrthogonal,
linalg::isPosDef,       linalg::jacobian,      linalg::jordanForm,
linalg::linearSolve,    linalg::linearSolveLU, linalg::multCol,
linalg::multRow,        linalg::ncols,         linalg::nonZeros,
linalg::normalize,      linalg::nrows,         linalg::nullSpace,
linalg::ogCoordTab,     linalg::ogSystem,      linalg::onSystem,
linalg::randomMatrix,   linalg::rank,          linalg::row,
linalg::scalarProduct,  linalg::setCol,        linalg::setRow,
linalg::stackMatrix,    linalg::sumBasis,      linalg::swapCol,
linalg::swapRow,        linalg::sylvester,     linalg::tr,
linalg::transpose,      linalg::vectorDimen,   linalg::vectorPotential
```

ライブラリの関数を使うには，ライブラリ名の後に二つのコロン (::) と関数名を繋げて入力しなければなりません。

従って，次の行列の固有値・固有ベクトルを求めるには，

```
>> A := Dom::Matrix()( [[-13, -10], [21,16]]);
```

```
                              +-          -+
                              |  -13, -10  |
                              |            |
                              |   21,  16  |
                              +-          -+
```

以下のように入力しなければなりません。

```
>> linalg::eigenVectors(A);
```

```
    -- --       -- +-      -+ -- --  --       -- +-      -+ -- -- --
    |  |        |  |  -5/7  |  |  |  |        |  |  -2/3  |  |  |  |
    |  |  1, 1, |  |        |  |  |, |  2, 1, |  |        |  |  |  |
    |  |        |  |    1   |  |  |  |        |  |    1   |  |  |  |
    -- --       -- +-      -+ -- --  --       -- +-      -+ -- -- --
```

これが面倒であるならば，linalg パッケージを取り込む事もできます。

```
>> export(linalg);
```

これを実行した後は，linalg::eigenVectors 関数も直接呼び出すことが出来ます。

上の行列の固有多項式を求めるデモンストレーションを行います。直接 charPolynomial 関数を呼び出すか，以下のように入力することで計算できます。

```
>> det(charMatrix(A,x));
```

$$-3x + x^2 + 2$$

2.3 グラフィックスと可視化

```
>> reset();
```

2.3.1 plotfunc 関数について

MuPAD は，グラフィックスを簡単に生成できる機能を複数提供しています。最も簡単かつ直接的な方法は，単純に関数をプロットしていく plotfunc 関数[2]を使うことです。

既に 2.1 節でその例を見てきました。ここでは別の例として，$[0, 4\pi]$ 区間で正弦関数のグラフを描いてみます (図 2.2)。

```
>> plotfunc(sin(x), x = 0..4*PI);
```

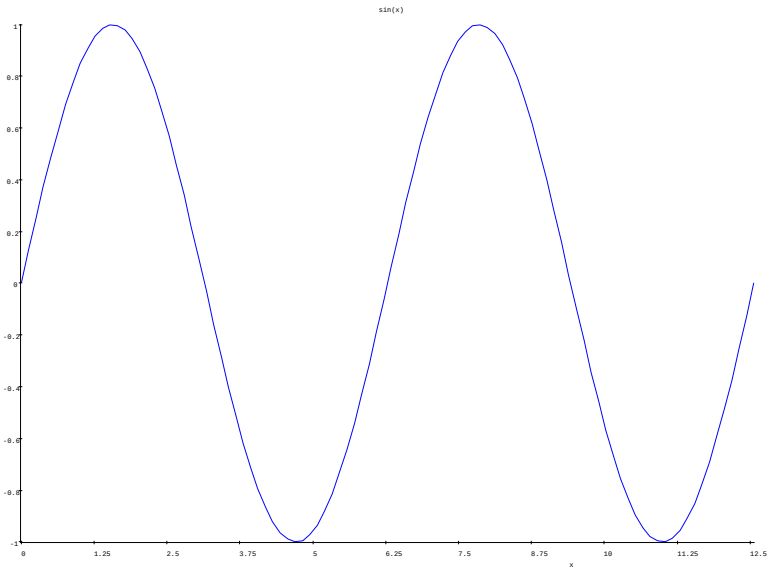


図 2.2: plotfunc(sin(x), x = 0..4*PI)

一つの画面に複数の関数をプロットする事もできます (図 2.3)。

```
>> plotfunc(sin(x), cos(x), x = 0..4*PI);
```

次の例のように，この plotfunc 関数はプロットする関数の特異性を制御することもできます (図 2.4)。

```
>> plotfunc(tan(x), x = -PI..PI);
```

[2](訳注) 関数っつったりコマンドっつったり，なんかニュアンスが違うんですかね?

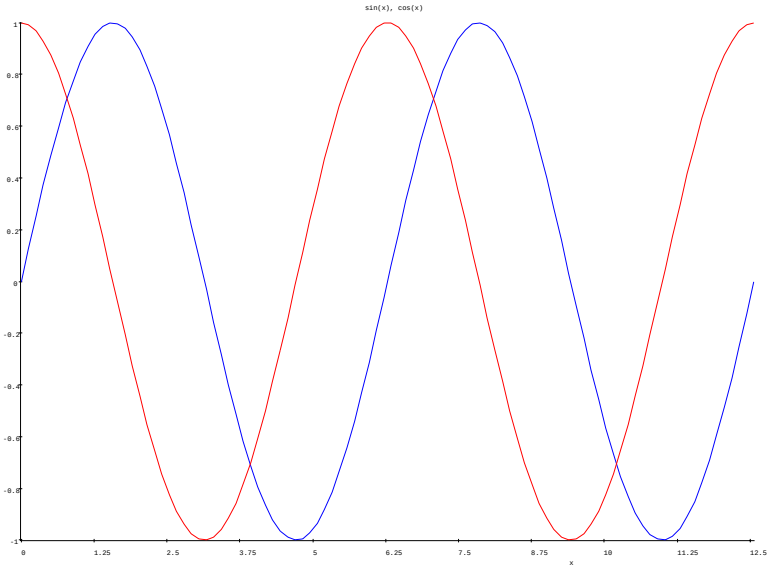


図 2.3: `plotfunc(sin(x), cos(x), x = 0..4*PI)`

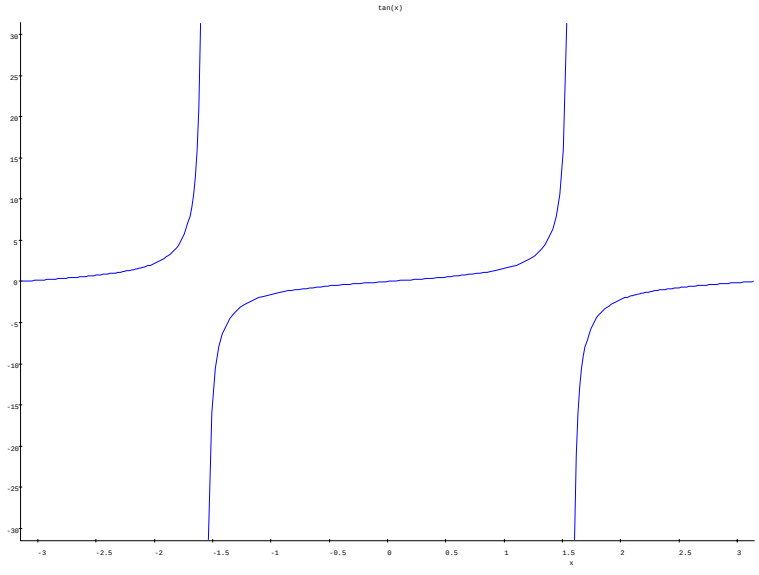


図 2.4: `plotfunc(tan(x), x = -PI..PI)`

2.3.2 plot2d 関数と plot3d 関数

plot2d 関数を使うと，関数のグラフだけでなく，様々な二次元画像を生成することが出来ます (図 2.5)。

```
>> plot2d([Mode = Curve, [sin(u), cos(u)], u = [-PI, PI], Grid = [50]]);
```

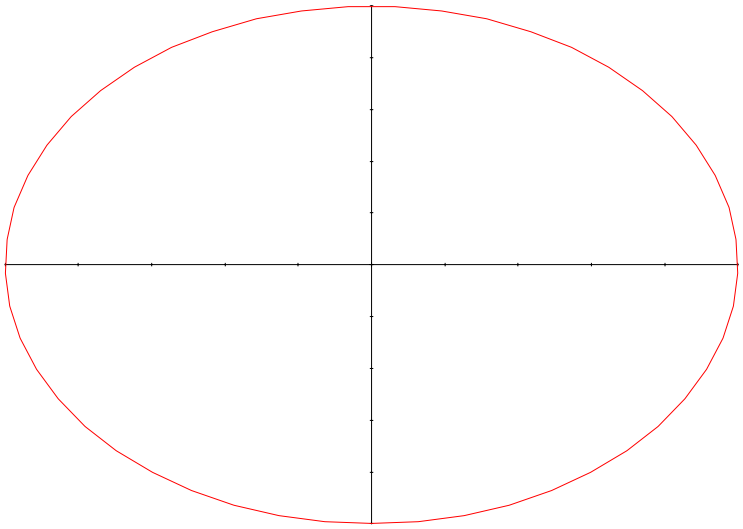


図 2.5: plot2d([Mode = Curve, [sin(u), cos(u)], u = [-PI, PI], Grid = [50]])

plot2d 関数による二次元画像の生成に加えて，MuPAD は三次元画像，特に三次元曲線・表面を描く plot3d 関数も提供しています (図 2.6)。

```
>> plot3d(Axes = None, Ticks = 0,
    Title = "Plot of sin(u^2 + v^2)",
    TitlePosition = Below,
    [    Mode = Surface,
         [u, v, 1/2*sin(u*u + v*v)],
         u = [0, PI], v = [0, PI],
         Grid = [30, 30],
         Color = [Height],
         Style = [ColorPatches, AndMesh]
    ]);
```

2.3.3 plotlib ライブラリ

グラフィックを描く高度なルーチンは plotlib ライブラリに揃っています。例えば，plotlib::xrotate 関数は曲線を x 軸を中心に回転させます (図 2.7)。

```
>> plotlib::xrotate(Axes = Box, [[x, sin(x)], x = [0, 4*PI], Angle = [0, PI],
```

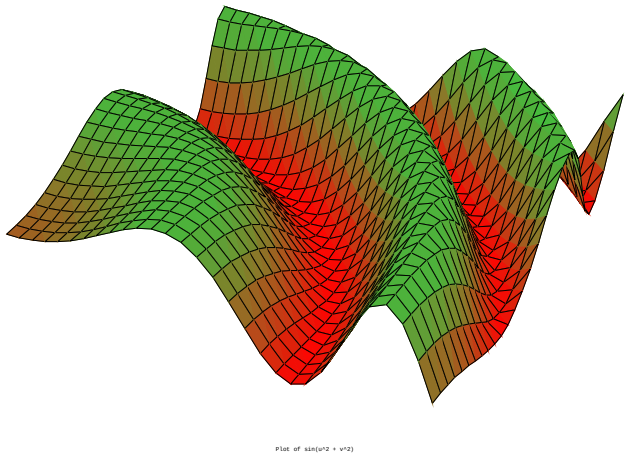


図 2.6: `plot3d(Axes = None, Ticks = 0,...`

```
                    Color = [Height], Grid = [50, 10]
                  ]
);
```

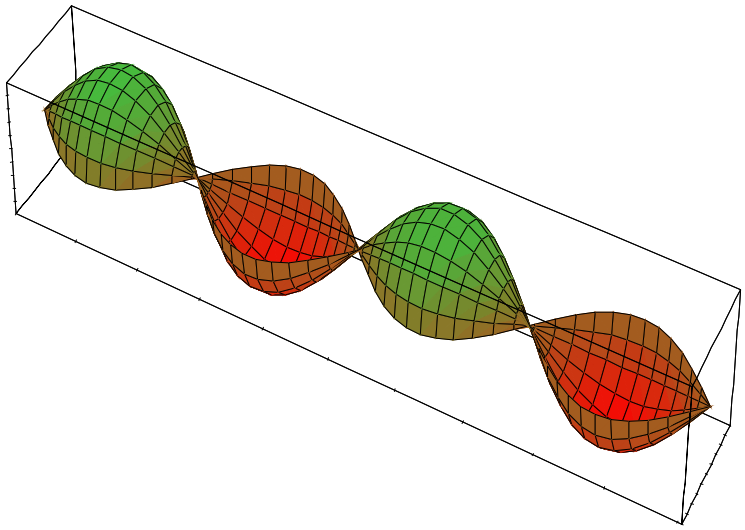


図 2.7: `plotlib::xrotate(Axes = Box,...`

同様に，`plotlib::yrotate` 関数は関数のグラフを y 軸を中心に回転させることが出来ます。
最後の例は `plotlib::fieldplot` 関数を使ってベクトル場を可視化させたものです (図 2.8)。

```
>> plotlib::fieldplot(Axes = Origin, Ticks = 0,
          [[-y^2, x^2], x = [-4, 4], y = [-4, 4],
           Grid = [15, 15], Color = [Height]
```

```
        ]
);
```

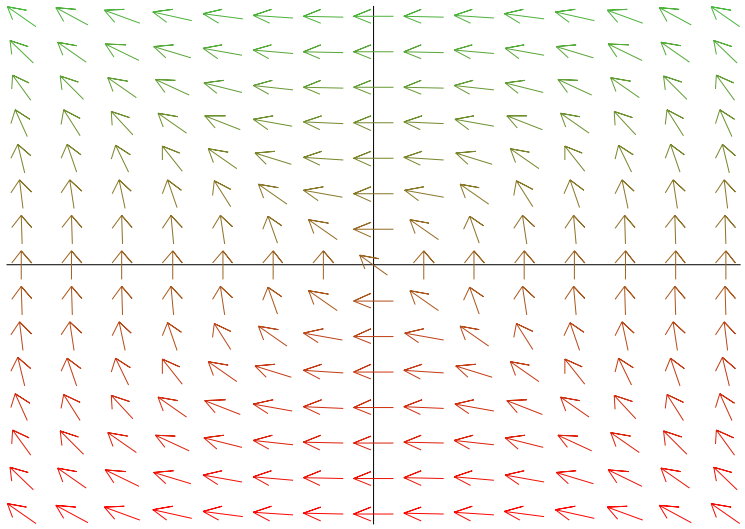


図 2.8: `plotlib::fieldplot(Axes = Origin,...`

2.4 数式表現の簡略化と書き換え

```
>> reset();
```

数式処理システムあるいは手計算の結果が複雑な形になることはよくあります。そんな時は，数式表現を簡略化したり書き換えたりすることで，読みやすく扱いやすいものにすることができます。

この簡略化を行う最も一般的な関数が，`simplify` 関数です。

```
>> simplify(sin(x)^2 + cos(x)^2 - 1);
```

```
0
```

この関数は，上のように 0 と等しいことを確認したり，下のように $\sqrt{4+2\sqrt{3}} = \sqrt{3}+1$ というような簡略化を行ったりするのに使われます。

```
>> simplify(sqrt(4 + 2*sqrt(3)), sqrt);
```

```
      1/2
     3    + 1
```

数式表現を書き換えたいときの例を示します。これは同時に簡略化も行っています。

```
>. f := (x - 1)^2/(x - 2) + a;
```

```
                                  2
                             (x - 1)
                        a + --------
                             x - 2
```

```
>> f1 := diff(f, x); f2 := diff(f, x, x);
```

```
                                        2
                           2 x - 2   (x - 1)
                           ------- - --------
                            x - 2           2
                                     (x - 2)
```

```
                                                2
                  2     2 (2 x - 2)   2 (x - 1)
                ----- - ----------- + ----------
                x - 2           2             3
                         (x - 2)       (x - 2)
```

これらの表現は展開されているので，Factor 関数で因数分解した形にすることができます。

```
>> Factor(f1); Factor(f2);
```

```
                              (x - 1) (x - 3)
                              ---------------
                                        2
                                 (x - 2)

                                    2
                                 --------
                                        3
                                 (x - 2)
```

有理関数に対して，Factor 関数を適用するのとは逆の働きを行うのが expand 関数です。この関数を使うと，加法定理を適用したのと同じ数式表現になることがよくあります。

この関数は Factor 関数以上の働きをします。というのは，例えば三角関数，指数関数，対数関数といったものを含む幅広い数式表現をも展開してしまうからです。

```
>> expand(cos(x + 2*y));
```

```
                    2                                          2
      cos(x) cos(y)  - 2 cos(y) sin(x) sin(y) - cos(x) sin(y)
```

項を特定の等価な表現に書き換えてしまうのが rewrite 関数です。以下の例は sin と cos の項を複素指数関数で置き換えています。

```
>> rewrite(sin(x)/cos(x), exp);

                    1/2 I exp(-I x) - 1/2 I exp(I x)
                    --------------------------------
                          exp(-I x)   exp(I x)
                          --------- + --------
                              2          2

>> simply(%);

                     / 1/2 I exp(-I x) - 1/2 I exp(I x) \
                simply| -------------------------------- |
                     |       exp(-I x)   exp(I x)       |
                     |       --------- + --------       |
                     \           2          2           /
```

ここでは，MuPAD が提供している simplify 関数，expand 関数，rewrite 関数が数式表現を簡略化したり書き換えたりする働きを見てきました。同様のことは，combine 関数，normal 関数，rectform 関数でも可能です。対応するヘルプのページを参照して下さい。

第3章　基本データ型

MuPAD を使い始めると，リスト，セット，テーブル，多項式というような様々なデータ型があるのがわかります。これらのデータ型は「基本データ型 (basic data type)」と呼ばれ，後づけで MuPAD システムで生成されたデータ型とは別のものです。ユーザは，データ型に対して必要な操作も含めて新たなデータ型を生成することができます。これを「ドメイン」と呼びます。

しかしドメインを自分で作るには，MuPAD とそのプログラミング言語に慣れていることが不可欠です。このあたりの MuPAD の特徴については「Advanced Demonstration Tour」でデモンストレーションをしています。第 5 章ではドメインがどんなことに役立つのか，2 つの例で示しています。

では，この章では基本データ型だけを見ていくことにしましょう。

3.1　数，表現，多項式など

セッションを初期化して下さい。

```
>> reset();
```

MuPAD では様々な種類の数や定数が使えます。

```
>> 2, -2/3,  2+3*I, PI, float(PI), E, float(E);

          2, -2/3, 2 + 3 I, PI, 3.141592653, exp(1), 2.718281828

>> map(%, domtype);

 DOM_INT, DOM_RAT, DOM_COMPLEX, DOM_IDENT, DOM_FLOAT, DOM_EXPR, DOM_FLOAT
```

ここでは map 関数を使っています。この関数は domtype 関数を上記のような異なる種類のオブジェクトの列に適用したのと同じ働きをします。

いわゆる「識別子 (数学で言うところの変数)[1]」と文字列も MuPAD で使えます。

```
>> domtype(x), domtype(MuPAD), domtype(x[2]);

                      DOM_IDENT, DOM_IDENT, DOM_EXPR

>> str := "The Computer Algebra System MuPAD";
```

[1] (訳注) 他に適当な訳語があったら教えて下さい。

文字列の取り出し・セットの挿入など，オブジェクトに対して様々な操作を行う関数は数多く取り揃えられています。この文書ではそのうちの幾つかを見ていくことになるでしょう。

「表現」は最も一般的なデータ型で，任意のデータ型のオブジェクトから組み立てられるものです。

```
>> ex := x^(-2) + sin(x)/cos(x) + f(2) + PI/2;
```

$$\frac{PI}{2} + f(2) + \frac{1}{2} + \frac{\sin(x)}{\cos(x)}$$

```
>> domtype(ex);
```

DOM_EXPR

多項式は次のように，表現として表わされます。

```
>> ex := x^4 + x^3 - 4*x^2 + 3*x - 1; domtype(ex);
```

$$3\,x - 4\,x^2 + x^3 + x^4 - 1$$

DOM_EXPR

しかし，より簡便に多項式を扱うため，MuPAD は一変数や多変数多項式のための特別なデータ型を提供しています。

```
>> p := poly(ex); domtype(p);
```

$$\mathrm{poly}(x^4 + x^3 + (-4)\,x^2 + 3\,x - 1, [x])$$

DOM_POLY

```
>> q := poly(x - 1);
```

$$\mathrm{poly}(x - 1, [x])$$

```
>> p/q, p*q + multcoeffs(p, 3);
```

$$\mathrm{poly}(x^3 + 2\,x^2 + (-2)\,x + 1, [x]), \mathrm{poly}($$

```
   5      4         3         2
  x  + 3 x  + (-2) x  + (-5) x  + 5 x - 2, [x])
```

多項式についてのより詳しい情報は poly 関数の項を参照して下さい。

3.2 セットとリスト

```
>> reset();
```

セットは次のように入力します。

```
>> M := {1, 2, 3, 4, a, b, 1, 2};

                          {a, b, 1, 2, 3, 4}

>> domtype(M);

                              DOM_SET
```

セットはいわゆる「表現の列」をその要素に持つことができます。

```
>> op(M);

                          a, b, 1, 2, 3, 4
```

「列演算子$」を使って列を直接生成することもできます。

```
>> unassign(i):
   x $ 10; A := i^2 $ i = 1..50;

                      x, x, x, x, x, x, x, x, x, x

1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289,

   324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961,

   1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849,

   1936, 2025, 2116, 2209, 2304, 2401, 2500
```

$演算子の左側のインデックス変数 (上の例では i) は値を持ってはいけません。でないとエラーメッセージを吐き出してしまいます。

ここでの unassign 関数は識別子が値を持たないようにリセットしています。

+演算子や*演算子と同じ働きをする関数[2]を扱う際には，列が重要な働きを担うことになるでしょう。

[2](訳注) 例を見れば分かるけど，_plus() や _mult() 関数のこと。

```
>> s := i^3 $ i = 1..100: _plus(s);
```

25502500

```
>> _mult(s);
```

```
81285103704665697929058034741394527800954175275203119077085794747670888482\
33730596856720188375050478138776220712647125923141159206411609199354037545\
83649069843601261900051908970248135107234498895796609463150334493880799668\
74258629176303020525059098874622860758365277162334136591629009247695685942\
95546721356189512751110077173732914733010540348420430895115846995709927414\
69705476383547415329993647980544000000000000000000000000000000000000000000\
00000000000000000000000000000
```

ではセットの基本演算を見ていきましょう。

```
>> M union {3, 4, a, c};
```

{a, b, c, 1, 2, 3, 4}

```
>> M intersect {3, 4, a, c};
```

{a, 3, 4}

```
>> contains(M, a), contains(M, hello);
```

TRUE, FALSE

セットとは逆に，列要素の順序を維持することができるようにするには,「リスト」を使います。

```
>> L := [a, b, a, b, c];
```

[a, b, a, b, c]

リストの場合，op 関数を使うかインデックスで指定すると要素を取り出せます。

```
>> L[2], L[5];
```

b, c

以下の例は，与えられたインデックスが 1 から 5 までのの範囲を超えているために，システムがエラーメッセージを出してしまいます。

```
>> L[100];
Error: Invalid index [list]
```

ドット演算子を使うとリストをつなぎ合わせることが出来ます。この演算子は_concat 関数の簡略形です。

```
>> unassign(L[5]): L;
```

```
                                [a, b, a, b]
```

```
>> [a, b] . [c, d] . [b, -a, a] . [c];
```

```
                            [a, b, c, d, b, -a, a, c]
```

ドット演算子は文字列，配列 (次の節を参照)，行列も連結させることができます。

3.3 配列とテーブル

```
>> reset();
```

MuPAD では多次元配列を扱うことが出来ます。

```
>> A := array(1..2, 1..3,
       (1,1) = array, (1,2) = with,
       (2,1) = not_defined, (2,3) = entries
   );
```

```
                    +-                                  -+
                    |     array,      with,  ?[1, 3]    |
                    |                                   |
                    |  not_defined, ?[2, 2], entries    |
                    +-                                  -+
```

```
>> A[1,2] := holding: A;
```

```
                    +-                                  -+
                    |     array,    holding, ?[1, 3]    |
                    |                                   |
                    |  not_defined, ?[2, 2], entries    |
                    +-                                  -+
```

```
>> A[2,1] * A[2,3];
```

```
                             entries not_defined
```

配列はインデックスに自然数しか使えません。しかし「テーブル (table)」では次のようにインデックスに任意の表現を使うことが出来ます。

```
>> S := table(1 = 2, 3 = 4);
```

```
                                  table(
```

```
    1 = 2,
    3 = 4
  )
```

```
>> S[{1}] := 11: S[a + b] := 22: S;
```

```
table(
  {1} = 11,
  a + b = 22,
  1 = 2,
  3 = 4
)
```

3.4 データ型を調べる

```
>> reset();
```

MuPAD で表現の型を調べるにに，幾つかの方法があります。

domtype 関数は既に前の例題で見てきました。これは表現のデータ型を明らかにするのに使われます。

しかし，表現についてのより多くの情報を得たいことはよくあります。例えば,「次の表現

```
>> ex := x^2 + y + (a - 1)^6/((a^2 + 1) * (a^2 - 1)) + sin(z) * cos(z);
```

$$y + \cos(z)\,\sin(z) + x^2 + \frac{(a - 1)^6}{(a^2 - 1)\,(a^2 + 1)}$$

は和になっている」,「[a, b, c] は 3 つの識別子を持つリストである」，というようにです。

3.4.1 type 関数

type 関数は任意の表現について，より多くの情報を得るために使われます。

```
>> type(ex);
```

```
"_plus"
```

ex のオペランドを type 関数で調べると，以下のような情報が得られるでしょう。

```
>> map(op(ex), type);
```

```
DOM_IDENT, "_mult", "_power", "_mult"
```

type 関数の結果は文字列か識別子です。後者の場合，type 関数の結果は引数のデータ型になります。

```
>> type([x, y, z]);
```

```
DOM_LIST
```

testtype 関数は以下のような特別な型を調べるのに使えます。

```
>> testtype(2 + x, "_plus");
```

```
TRUE
```

しかし，type 関数が複雑な型を調べるのに便利かというと，一概には言えません。型の選択肢が複数ある場合，あるいは，多項式の係数ドメインのようにパラメータに依存して型が決まる場合などは面倒です。

ということで，どうやったら以下の例が 4 つの正の整数から成るリストであるかということが判別できるでしょうね?

```
>> l := [2, 5, 1, 5];
```

```
[2, 5, 1, 5]
```

3.4.2 Type ライブラリ

Type ライブラリはオブジェクトの特定の型構造を明らかにするのに便宜を図る目的で作られました。以下のように，あらかじめ定義された「型表現 (type expression)」を提供します。

```
>> testtype(l, Type::ListOf(Type::PosInt,4));
```

```
TRUE
```

このライブラリを使うにあたってもう少し例を示しましょう。勿論，これがこのライブラリの出来ること全てを表わしているわけではありません (`info(Type)` と打つと，あらかじめて定義されていた型表現のリスト (表 3.1[3]) を取り出せます。詳細は対応するヘルプのページを見て下さい)。

- オブジェクトが正の整数か識別子かを調べるには?

  ```
  >> testtype(x, Type::Union(Type::PosInt, DOM_IDENT));
  ```

  ```
  TRUE
  ```

- 有理数上の 2 変数多項式かをテストするには？[4]

  ```
  >> testtype(poly(x*y^2 + x/3 + y), Type::PolyOf(Type::Rational, 2));
  ```

  ```
  TRUE
  ```

[3] (訳注) この表は原文にはありません。
[4] (訳注) 原文のように打ち込むと当然 FALSE になる。Bug だと思うけど。

表 3.1: info(Type)

```
>> info(Type);
Domain 'Type': Type expressions for testing types
Interface:
Type::AnyType,      Type::Complex,      Type::Divs,
Type::Even,         Type::Fraction,     Type::IV,
Type::Imaginary,    Type::IntImaginary, Type::Integer,
Type::Irrational,   Type::ListOf,       Type::ListOfIdents,
Type::ListProduct,  Type::NegInt,       Type::NegRat,
Type::Negative,     Type::NonNegInt,    Type::NonNegRat,
Type::NonNegative,  Type::Odd,          Type::PolyOf,
Type::PosInt,       Type::PosRat,       Type::Positive,
Type::Prime,        Type::Product,      Type::Rational,
Type::RealNum,      Type::Relation,     Type::SequenceOf,
Type::Series,       Type::SetOf,        Type::Singleton,
Type::TableOfEntry, Type::TableOfIndex, Type::Union,
Type::Zero
```

- 級数展開式が Taylor 展開，Laurent 展開，Puiseux 展開[5]のうちどれかを調べるには?

```
>> s := series(1/sin(x), x);
```

$$\frac{1}{x} + \frac{x}{6} + \frac{7\,x^3}{360} + O(x^4)$$

```
>> testtype(s, Type::Series(Taylor));
```

FALSE

```
>> testtype(s, Type::Series(Laurent));
```

TRUE

3.5 オブジェクトの操作

```
>> reset();
```

MuPAD はデータ型のオブジェクトをいろいろ操作できるツールです。ある特定のデータ型オブジェクトは抽出，削除，サブ表現の代入が可能な「オペランド (operand)」が複数集まって出来上

[5](訳注) という展開があるということを，一松先生より教えていただきました。

がっています。
　簡単な例でこのことを示してみましょう。a+b+c という表現は 3 つのオペランドの和です。

```
>> s := a + b + c: nops(s);
```

```
                                    3
```

```
>> op(s, 1), op(s, 2), op(s, 3);
```

```
                                 a, b, c
```

　subs 関数，subsop 関数のような関数は，新たな表現をサブ表現やオペランドに代入することが出来ます。

```
>> subs(s, a = 1);
```

```
                                 b + c + 1
```

　subs 関数は，文字の置き換え程度しか行わないことを覚えておいて下さい。
　以下の入力を行うと何の動作もしません。というのは，subs 関数だけでは，op 関数を使って直接抽出されたサブ表現だけを置き換えるにすぎないからです。

```
>> subs(a + b + c, a + b = 1);
```

```
                                 a + b + c
```

　以下の例ではきちんと動作します。

```
>> subs(a*b + d*e, a*b = 1);
```

```
                                  d e + 1
```

　subsex 関数は不完全なサブ表現を代入しても使うことが可能です。

```
>> subsex(a + b + c, a + b = 1);
```

```
                                    c + 1
```

　上で示した関数は，セット，リスト，テーブル，配列，その他のデータ型と同じように表現にも適用できます。

```
>> M := {1, 2, 3, 4, a, b, 1, 2};
```

```
                              {a, b, 1, 2, 3, 4}
```

```
>> S := subsop(M, 2 = Operand2nd, 5 = Operand5th);
```

```
                      {a, Operand2nd, 1, 2, 4, Operand5th}
```

従って，セットの要素は op 関数で取り出すことが出来ます。しかし，最後の出力結果で分かるように，セットの要素の順序はユーザには触れない MuPAD 内部の順序に従っています。よって，「セットの 5 番目のオペランド」というように直接指定してしまうと，それより以前にセットが操作されてしまう場合，意味がなくなってしまうこともありえます。

以下ではテーブルのオペランドを抽出しています。

```
>> S[{1}] := 11: S[a + b] := 22: S;
```

```
                              table(
                                {1} = 11,
                                a + b = 22
                              )
```

```
>> op(S);
```

```
                                {1} = 11, a + b = 22
```

見ての通り，オペランドは左辺にインデックス，右辺にそれに対応する要素が来る方程式の形になります。

今，a + b というような特定のインデックスを新しく代入するためには，テーブルの 2 番目のオペランドの左辺に与えなければなりません。

```
>> subsop(S, [2,1] = "hello");
```

```
                              table(
                                {1} = 11,
                                "hello" = 22
                              )
```

既に述べたように，プロシージャは基本データ型 DOM_PROC になっており，複数ののオペランドから成り立っています。

```
>> Max := proc(a, b) begin
       if a < b then b else a end_if
   end_proc:

   op(Max);
```

```
                    (a, b), NIL, NIL, (if a < b then
                      b
                    else
                      a
                    end_if), NIL, Max, NIL, NIL
```

プロシージャの本体と名前，引数を見て下さい。完全なオペランドの記述方法はマニュアルに書いてあります。

ここで，2 つの引数の最小値を求めるよう，このプロシージャを書き変えてみましょう。

```
>> Min := subsop(Max, [4,1] = _not(_less(a, b)), 6 = hold(Min));
```

```
proc(a, b)
  name Min;
begin
  if b <= a then
    b
  else
    a
  end_if
end_proc
```

```
>> Min(2,10);
```

```
2
```

MuPAD のステートメントは全て，複数のオペランドから成る正しい表現になっていることを覚えておいて下さい。最後の例では，`if-then` 文を代入させてみました。

```
>> subs(hold(_if)(a < b, a, b), a = x, b = y);
```

```
if x < y then
  x
else
  y
end_if
```

MuPAD はどんなオブジェクトに対しても，かなり自由な操作ができるということが分かると思います。

第4章　MuPADプログラミング言語

MuPADは解析ツール・デバッグツール付きの高機能なプログラミング言語を提供しています。

4.1　文の基本

```
>> reset();
```

ループとスイッチ文の基本的な文法を紹介していきましょう。

```
>> L := [1, 2, 3, 4]: result := []:
   for i in L do
       result := result . [i + c];
   end_for:
   result;
```

```
                        [c + 1, c + 2, c + 3, c + 4]
```

MuPAD言語はまた，上の例を以下のようにも書き換えることができます。

```
>> map(L, _plus, c);
```

```
                        [c + 1, c + 2, c + 3, c + 4]
```

プロシージャは任意個の引数を持つことが出来ます。以下の例ではランダムに選ばれた数の中から最大のものを取り出しています。

```
>> Max := proc() local m, i;
>> begin
       m := 0;
       for i in args() do
           if i > m then m := i; end_if;
       end_for;

       return(m);
   end_proc:

   Max(random() $ hold(i) = 1..100);
```

```
                              992987500442
```

勿論，この結果はあなたのそれと異なっているはずです[1]。

4.2 計算時間とその解析

```
>> reset();
```

time 関数は，引数を評価するのに費やされたトータルの CPU 時間 (ms) を返します。どのプロシージャで，どの部分が，どのくらい時間がかかったかを調べるには profile コマンドを使いましょう。

```
>> time( _plus(1/k $ k = 1..100));
```

```
                                    2
```

出力結果はコマンドを実行したマシン環境[2]によって変化することをお忘れなく。
どの式でどの

```
>> fib := proc(n)
   begin
       if n < 2 then 1; else fib(n - 1) + fib(n - 2); end_if;
   end_proc:

   profile(fib(7));

Total time: 5 ms
----------------
fib:100.0 % 5 ms total 41 call(s) 0 lookup(s) 0.1 ms/call

<fib> calls
   fib : 40 time(s)
```

```
                                   21
```

プロシージャに remember オプションを使うと，再帰的に呼び出されるプロシージャであれば，実行時間をかなり改善させることが期待できます。

```
>> fib := proc(n)
       option remember;
   begin
       if n < 2 then 1; else fib(n - 1) + fib(n - 2); end_if;
```

[1](訳注) 乱数の生成方法が変わらなければ同じ結果になる。
[2](訳注) この訳にある結果は全て，MuPAD 1.4.1 Light for Windows を Windows 98，CPU:Pentium II 450MHz, 384MB RAM の PC/AT 互換機で実行したもの。

```
   end_proc:

   profile(fib(7));

Total time: 5 ms
----------------
fib:100.0 % 5 ms total 13 call(s) 5 lookup(s) 0.3 ms/call

<fib> calls
   fib : 12 time(s)
```


関数は再帰的に呼び出される場合，関数の値はその都度計算するのではなく，プロシージャ内に記憶されているテーブルから引っぱり出されます。13 回の内 5 回というのは，その回数を意味しています。

remember オプションを使って記憶されているテーブルは，5 番目のオペランドに格納されています。

```
>> op(fib, 5);
```

```
table(
  0 = 1,
  1 = 1,
  2 = 2,
  3 = 3,
  4 = 5,
  5 = 8,
  6 = 13,
  7 = 21
)
```

最後に次の式を実行時間を解析してみましょう。

```
>> factor(poly(x^2 - 1, [x]));
```

```
[1, poly(x - 1, [x]), 1, poly(x + 1, [x]), 1]
```

前のものよりずっと複雑な結果が出てきます。

```
>> profile(factor(poly(x^2 - 1, [x])));

"Total time < 10 ms. No timing informations possible."
```

```
"--------------------------------------------------"
faclib::monomial :1 call(s) 0 lookup(s)
sign             :1 call(s) 0 lookup(s)
factor           :1 call(s) 0 lookup(s)
anonymous        :3 call(s) 0 lookup(s)
Type::PolyOf     :1 call(s) 0 lookup(s)
fun              :2 call(s) 0 lookup(s)
Type::testtype   :1 call(s) 0 lookup(s)
faclib::sbinomial:1 call(s) 1 lookup(s)
faclib::binomial :1 call(s) 0 lookup(s)
faclib::pfactor  :1 call(s) 0 lookup(s)

<faclib::pfactor> calls
   faclib::binomial : 1 time(s)

<faclib::binomial> calls
   faclib::sbinomial : 1 time(s)

<Type::testtype> calls
   anonymous : 1 time(s)

<anonymous> calls
   anonymous : 2 time(s)

<factor> calls
   faclib::pfactor  : 1 time(s)
   Type::testtype   : 1 time(s)
   fun              : 2 time(s)
   Type::PolyOf     : 1 time(s)
   sign             : 1 time(s)
   faclib::monomial : 1 time(s)
```

```
[1, poly(x - 1, [x]), 1, poly(x + 1, [x]), 1]
```

その他，MuPAD でのプログラミングに使えるユーティリティとしては，ソースコードデバッガ，トレース関数 (trace 関数，sharelib::trace 関数参照)，プロシージャのローカル変数・グローバル変数のチェック (misc::checkFunction 関数参照)，テスト適用範囲を生成するための関数 (misc::tcov 関数参照) などがあります。

第5章　ドメインを使った例

第 3 章ではユーザが定義したデータ型，いわゆる「ドメイン」を作ることができることを示しました。

この章ではデータ型を実装するだけでなく，それが何の役に立つのか？ということも学んで貰います。

ここでは MuPAD パッケージに実装されているデータ型を使います。ここで大事なのは，このデータ型は MuPAD システムがあらかじめ提供しているデータ型ではない，ということです。あらかじめ定義されているものは基本データ型と呼びます。

どちらの例でも，きちんと書かれているという前提のユーザ定義データ型の上であれば，線型代数ライブラリの関数が動作する，ということがわかると思います。そのようなアルゴリズムは「汎用性がある (generic)」と呼ばれます。汎用性のあるアルゴリズムは特定のデータ型だけでしか使用できないというものではなく，与えられたオブジェクトの（代数的なものであることが多い）性質にだけ依存しています。

5.1　区間演算

セッションを初期化して下さい。

```
>> reset(): export(linalg):
```

次の問題を考えます。

1. 連立一次方程式 $x + 5y = -2, 2x + y = 4$ を解く。
2. この方程式の係数全てに 0.01 の摂動を起こしたものを解く。

この問題解くには `linalg` ライブラリを使います。

```
>> Ab := expr2Matrix([x + 5*y = -2, 2*x + y = 4], [x, y], Dom::Rational);
```

```
                    --  +-       -+  +-     -+  --
                    |   |  1, 5   |  |  -2   |  |
                    |   |         |, |       |  |
                    |   |  2, 1   |  |   4   |  |
                    --  +-       -+  +-     -+  --
```

`linalg::linearSolve` 関数を使うと，前者の問題は簡単に解けます。

```
>> linearSolve(op(Ab));
```

```
                                +-       -+
```

```
|  22/9  |
|        |
|  -8/9  |
+-      -+
```

後者の問題を解くには，実数区間を表現する Dom::Interval ドメインを使います。これは標準的な区間演算を行う場を与えてくれます。

さて，区間演算場の上で行列を定義し，A と b を前と同じものに設定します。ただし 0.01 の摂動を要素ごとに入れておきます。

```
>> alias(IV = Dom::Interval):
   IAb := expr2Matrix(
    [IV(0.99..1.01)*x + IV(4.99..5.01)*y = IV(-2.01..-1.99),
     IV(1.99..2.01)*x + IV(0.99..1.01)*y = IV(3.99..4.01)]
   ,IV);
```

```
        -- +-                          -+  +-               -+ --
        |  |  0.99..1.01, 4.99..5.01   |  |  -2.01..-1.99   |  |
        |  |                           |, |                 |  |
        |  |  1.99..2.01, 0.99..1.01   |  |   3.99..4.01    |  |
        -- +-                          -+  +-               -+ --
```

こうして，新たな連立一次方程式を解きます。

```
>> linearSolve(op(IAb));
```

```
                  +-                                -+
                  |    2.309250509..2.584883855     |
                  |                                 |
                  |  -0.9079910213..-0.8702090209   |
                  +-                                -+
```

上の式の意味は，係数が 0.01 の精度であれば，解のうち x については 2.447 を中心に約 0.14 の誤差を持ち，y については -0.89 を中心に約 0.02 の誤差を持つ，ということです。

5.2 ブロック行列

セッションを初期化して下さい。

```
>> reset(): export(linalg):
```

ブロック行列を定義して演算をさせる例をお見せしたいと思います。最初に有理数体 $\mathbf{Q}$ 上の 2×2 行列環を定義します。

```
>> SqMat := Dom::SquareMatrix(2, D13 回の内 5 回というのは，om::Rational);
```

```
                  Dom::SquareMatrix(2, Dom::Rational)
```

では 4 つの正方行列を定義しましょう。

```
>> a := SqMat([[2,3], [1,0]]); b := SqMat([[1,0], [-2,3]]);
   c := SqMat([[0,-4],[2,1]]); d := SqMat([[0, 1], [2, -1]]);
```

```
+-      -+
|  2, 3  |
|        |
|  1, 0  |
+-      -+

+-       -+
|   1, 0  |
|         |
|  -2, 3  |
+-       -+

+-       -+
|  0, -4  |
|         |
|  2,  1  |
+-       -+

+-       -+
|  0,  1  |
|         |
|  2, -1  |
+-       -+
```

以下では 2×2 行列環上の行列の集合を定義しています。

```
>> MS := Dom::Matrix(SqMat);
```

```
            Dom::Matrix(Dom::SquareMatrix(2, Dom::Rational))
```

上の行列 a, b, c, d を要素として含む行列を定義します。

```
>> A := MS([[a, b], [c, d]]);
```

```
+-                           -+
|   +-      -+  +-       -+  |
|   |  2, 3  |  |   1, 0  |  |
|   |        |, |         |  |
|   |  1, 0  |  |  -2, 3  |  |
|   +-      -+  +-       -+  |
|                             |
```

```
  |  +-       -+  +-       -+  |
  |  |  0, -4  |  |  0,  1  |  |
  |  |         |, |         |  |
  |  |  2,  1  |  |  2, -1  |  |
  |  +-       -+  +-       -+  |
  +-                           -+
```

乱数行列も幾つか定義します。

```
>> f := fun(SqMat(
       randomMatrix(2, 2, Dom::Integer)
   )):
   B := MS(2, 2, f);
   C := MS(2, 2, f, Diagonal);
```

```
+-                                      -+
|  +-              -+  +-              -+  |
|  |   824,  -65   |  |  -979, -764   |  |
|  |               |, |               |  |
|  |  -814, -741   |  |   216,  663   |  |
|  +-              -+  +-              -+  |
|                                          |
|   +-             -+   +-             -+  |
|   |  880,  916   |   |  597, -245    |  |
|   |              |,  |               |  |
|   |  617, -535   |   |   79,  747    |  |
|   +-             -+   +-             -+  |
+-                                        -+
```

```
 +-                                      -+
 |  +-              -+     +-       -+    |
 |  |   477, -535   |     |  0, 0   |    |
 |  |               |,    |         |    |
 |  |  -906, -905   |     |  0, 0   |    |
 |  +-              -+     +-       -+    |
 |                                        |
 |     +-       -+     +-              -+  |
 |     |  0, 0   |     |  -266,  -8    |  |
 |     |         |,    |               |  |
 |     |  0, 0   |     |   765, 448    |  |
 |     +-       -+     +-              -+  |
 +-                                      -+
```

ランダムに行列を生成したのですから，上の結果はあなたの実行結果とは異なる可能性があることを覚えておいて下さい。以下のように，普通の演算も実行することが出来ます。

```
>> A*B + B - 2*C;

               +-                                      -+
               |  +-              -+   +-              -+  |
               |  |   -44,  -432   |   |  -1692, -548   |  |
               |  |                |,  |                |  |
               |  |  1913, -2433   |   |  -1720, 2630   |  |
               |  +-              -+   +-              -+  |
               |                                          |
               |   +-             -+  +-              -+  |
               |   |  4753, 3345   |  |   344,  -2134  |  |
               |   |               |, |                |  |
               |   |  2594,  961   |  |  -2078, -2251  |  |
               |   +-             -+  +-              -+  |
               +-                                        -+

>> A^5;

                +-                                      -+
                |   +-             -+  +-              -+  |
                |   |  558,  777    |  |   431, -222    |  |
                |   |               |, |                |  |
                |   |  290, -278    |  |  -551,  959    |  |
                |   +-             -+  +-              -+  |
                |                                          |
                |  +-              -+  +-              -+  |
                |  |  -537, -992   |  |  -649,  551    |  |
                |  |               |, |                |  |
                |  |   196,  424   |  |   345, -340    |  |
                |  +-              -+  +-              -+  |
                +-                                        -+

>> B*C - C*B;

           +-                                              -+
           |   +-                   -+   +-                 -+  |
           |   |  -376600, -747445   |   |   258497, 384693   |  |
           |   |                     |,  |                    |  |
           |   |   292942,  376600   |   |  -241755, 203127   |  |
           |   +-                   -+   +-                 -+  |
           |                                                    |
           |  +-                    -+  +-                   -+  |
           |  |  -171120, -1060404   |  |  -186793, -173730   |  |
```

```
|  |                      |, |                    |  |
|  | -170597,  -306980  |  |   58344,   186793  |  |
|  +-                   -+  +-                  -+  |
+-                                                  -+
```

最後に，A の逆行列を計算したいと思います。周知の通り，2×2 行列 $[[a, b], [c, d]]$ が逆行列を持つのは，$\det(a) \neq 0$ かつ $\det(d - \frac{bc}{a}) \neq 0$ の時です。

これを確認してみましょう。

```
>> det(a);
```

```
                                  -3
```

```
>> det(d - c*(1/a)*b);
```

```
                                  -3
```

従って，A の逆行列は次の式で計算することが出来ます。

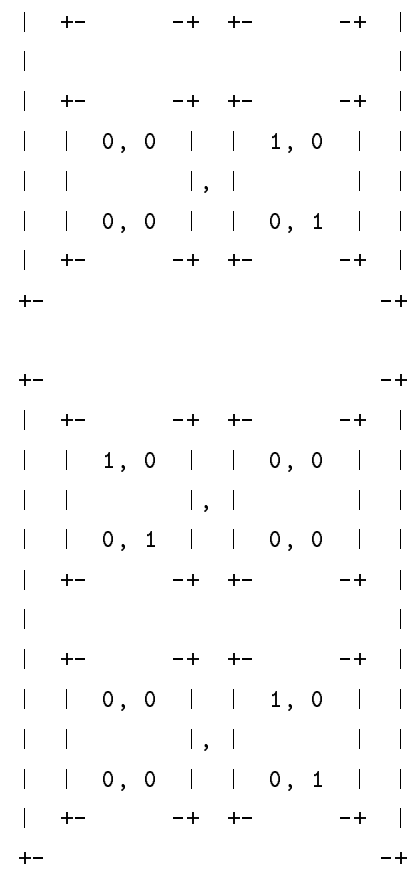
```
>> f := d - c*(1/a)*b:
>> iA := MS([[1/a + 1/a*b*1/f*c*1/a, -1/a*b*1/f],
            [-1/f*c*1/a               , 1/f           ]]);
```

```
+-                                          -+
|  +-           -+   +-           -+   |
|  |   -2,   1   |   |   -1,   2   |   |
|  |             |,  |             |   |
|  |  2/3, -2/9  |   |  1/9, -5/9  |   |
|  +-           -+   +-           -+   |
|                                        |
|  +-           -+  +-             -+  |
|  |   3,  -4/3  |  |   5/3,  -7/3  |  |
|  |             |, |               |  |
|  |  8/3, -8/9  |  |  13/9, -20/9  |  |
|  +-           -+  +-             -+  |
+-                                          -+
```

最後に，検算してみます。

```
>> A * iA; iA * A;
```

```
+-                        -+
|  +-      -+  +-      -+  |
|  |  1, 0  |  |  0, 0  |  |
|  |        |, |        |  |
|  |  0, 1  |  |  0, 0  |  |
```


0, 0
1, 0
0, 1
0, 0
1, 0
0, 0
0, 1
0, 0

第6章　訳者より

この未だ不完全な翻訳を丹念に読みご意見を寄せていただいた，静岡理工科大学大学院生・満田賢一郎氏に感謝いたします。なお，直訳調が直っていないのは訳者の日本語能力・英語能力の欠如によるものです。すんません。

〒 437-8555　静岡県袋井市豊沢 2200-2
静岡理工科大学
理工学部情報システム学科
幸谷智紀
E-Mail: tkouya@cs.sist.ac.jp

〒 270-1445　千葉県東葛飾郡沼南町岩井 678-3
千葉県立沼南高等学校
角谷　悟