Universität Gesamthochschule Paderborn
Automath
Institut für Automatisierung und Instrumentelle Mathematik

MuPAD

Multi Processing Algebra Data Tool

A Demonstration Tour through MuPAD 1.4

Frank Postel

24 September 1997

Automath Technical Report

Abstract

This document gives a survey of basic features of MuPAD 1.4 going through some different aspects of computer algebra. Parts of this article come from a joint work together with Dr. Christian Heckler and Gerald Rüscher (both member of the MuPAD group, University of Paderborn).

`demo` – A Demonstration Tour through MuPAD 1.4

Contents

1 First Steps

Let us first demonstrate some basic items which the most of all computer algebra systems have in common.

1.1 Arithmetic with Numbers

Please initialize the session:

```
>> reset():
```

MuPAD can be used as a pocket calculator to perform arithmetical computations. The input

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

is simplified to the rational number $\frac{7}{2}$:

```
7/2
```

As you can see MuPAD performs computations with integers and rationals exactly:

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

    67473/6728
```

Moreover, the system is able to efficiently handle computations with very large numbers. The length of a number is only restricted by the memory available on your computer. Let us compute the 123rd power of 1234:

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

that is a number with 381 digits[1]:

```
1705158062127270428750597276206262826543023131110682904705 2\
9619322183913834868007471366306717060598572641592314554345 9\
0057058967067149970908610253990484651479313561730556366999 3\
9501046220356820273557577550700832384441477783960263870670 4\
2685700404003287042480639680696865587865016699383883388831 9\
8045915994284537241460180942971772610762859524340680101441 8\
5297662798380672035627991 04
```

We can ask MuPAD whether the number 2398232343243249984321312321 is prime or not. If number is prime if it is only divisible by 1 and itselves:

```
>> isprime( 2398232343243249984321312321 );

    FALSE
```

The answer `FALSE` means that the given number is not prime. If you want to see all factors of 2398232343243249984321312321 use the function `Factor`:

```
>> Factor( 2398232343243249984321312321 );

    72140687161773179 3 733 15117701
```

Do you believe the system? If not, you may check the result by multiplying the factors:

```
>> _mult( % );

    2398232343243249984321312321
```

[1] In the following output, the backslash \ at the end of the lines indicates that the output is continued in the next line.

As we have seen, *all* digits are calculated exactly. But what does "exact" computations really mean? What does the system return when, for instance, the input is an irrational number like $\sqrt{56}$?

1.1.1 Exact Computations

We will try to answer this question by the following examples. As you might have expected, MuPAD is able to compute $\sqrt{4}$ exactly:

```
>> sqrt(4);

    2
```

Now, let us ask the system for the real number $\sqrt{56}$:

```
>> sqrt(56);

        1/2
    2 14
```

As you can see, MuPAD simplifies $\sqrt{56}$ to the exact value of $2\sqrt{14}$, where `sqrt(14)` is an expression that represents in MuPAD the positive solution $\sqrt{14}$ of the equation $x^2 = 14$ (see function `sqrt`).

Another example may be given by calculating the limit $\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = e$:

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

    exp(1)
```

The value `exp(1)` represents in MuPAD the base of the natural logarithm, i.e. $e = 2.71828\ldots$ (see function `exp`).

1.1.2 Approximative versus Exact Computations

In contrast to exact arithmetic you may want to compute approximatively. For example, if you want to have an approximative value for $\sqrt{56}$ you must use the function `float`:

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

    7.483314773
```

The precision of the approximation depends on the value of the global variable `DIGITS` which has the default value of 10:

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

    10

    1.002868608e1
```

`DIGITS` gives the number of significant digits when performing floating point arithmetic. You can change its value to any integer between 1 and $2^3 2 - 1$:

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

    10.02868608799048751486325802615933412604042806183115338882\

    2829964328180737217598097502972651605231866
```

The statement `unassign( DIGITS )` reset the value of `DIGITS` to its default value (see `unassign]`).

The constants π and e can be used for *exact* calculations in MuPAD:

```
>> cos( PI );

    -1
```

or

```
>> ln( E );

    1
```

It is also possible to compute a floating point approximation of these constants, if wanted. The following example computes π on one thousand digits exactly in less than 1 second:

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

```
3.1415926535897932384626433832795028841971693993751058209749\
94459230781640628620899862803482534211706798214808651328230\
66470938446095505822317253594081284811174502841027019385211\
05559644622948954930381964428810975665933446128475648233786\
78316527120190914564856692346034861045432664821339360726024\
91412737245870066063155881748815209209628292540917153643678\
92590360011330530548820466521384146951941511609433057270365\
75959195309218611738193261179310511854807446237996274956735\
18857527248912279381830119491298336733624406566430860213949\
46395224737190702179860943702770539217176293176752384674818\
46766940513200056812714526356082778577134275778960917363717\
87214684409012249534301465495853710507922796892589235420199\
56112129021960864034418159813629774771309960518707211349999\
99837297804995105973173281609631859502445945534690830264252\
23082533446850352619311881710100031378387528865875332083814\
20617177669147303598253490428755468731159562863882353787593\
    751957781857780532171226806613001927876611195909216420199
```

The output is rather long but impressive, isn't it?

1.2 Expressions and Statements

Please initialize the session:

```
>> reset():
```

In the section before we did only consider exact and floating point arithmetic with numbers. We now want to consider symbolic computations.

Here we handle with so-called *expressions*:

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

   f(x)
   ---- + sin(3)
    5
```

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

    {{y = 3/2, x = 7/2}}
```

Beside of expressions you can also enter a sequence of *statements* to the system. Statements are loops (`for–do`, `while–do` and `repeat–until`), case switches (`if–then` and `case–of`), flow controls (`quit`, `break` and `next`), expression sequences and assignments, as they may be known to you from ordinary programming languages.

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

    6469693230
```

Here, the function `ithprime` expects a positive integer i as its argument and calculates the ith prime, as its name indicates.

MuPAD offers many predefined *data types*. For example the *list*

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

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

is an object of the basic types `DOM_LIST` and its entries are objects of, in that order, `DOM_INT`, `DOM_RAT` and `DOM_COMPLEX`.

The data type of an object can be determined by the function `domtype`:

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

    DOM_INT

    DOM_RAT

    DOM_COMPLEX
```

The expression `sin(3)+f(x)/5` is of the data type `DOM_EXPR`:

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

    DOM_EXPR
```

This is the most general data type and its objects can be (and they usually are) a combination of objects of different data types.

The next chapter will demonstrate some mathematical aspects of MuPAD.

If you are more interested in programming aspects of MuPAD you can leave out the next chapter and continue with chapter 3.

2 Mathematics

To describe all the possibilities where you can use MuPAD for different applications of mathematics would be beyond the scope of this introduction.

In the following sections we will briefly go through some domains of mathematics and refer the interested reader also to the report "Advanced Demonstration Tour" which describes certain parts a little bit more detailed.

2.1 Calculus

```
>> reset():
```

Consider the rational function $f := \dfrac{(x-1)^2}{x-2} + a$ for real x and a. To analyze f first without simply plot its graph at a certain interval we look at some characteristics:

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

    {2}
```

See `discont` if you want to have more details. The zeros of `f` can be computed with `solve`:

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

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

Now, let us ask for extremal points:

```
>> 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
```

We see from the result that f in $(-\infty, 2)$ has a maximum at $x = 1$ and in $(2, \infty)$ a minimum at $x = 3$. The corresponding values are:

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

      a

    a + 4
```

Let us finally ask for the asymptotic behaviour of f. f has a singularity at $x = 2$ where it tends to $-\infty$ from the left and ∞ from the right:

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

    -infinity, infinity
```

For large x, the function f is asymptotically equal to the linear term $x + a$:

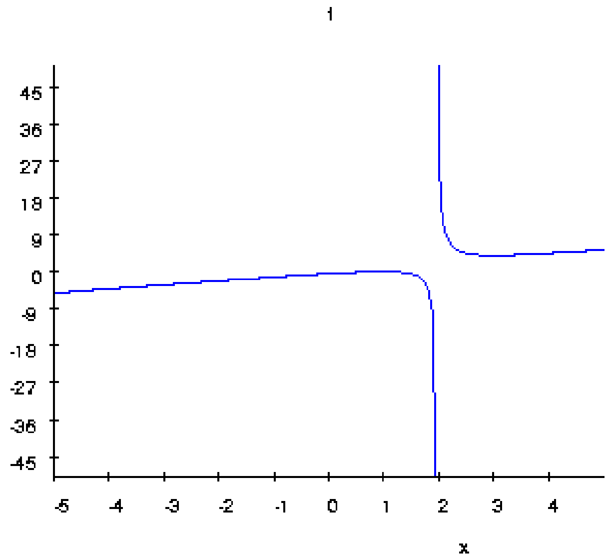
```
>> series( f,x=infinity );

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

Here we introduced the functions `limit` and `series`.

You can have a look at a graph of this function for some `a` with the function `plotfunc`:

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



2.2 Linear Algebra

```
>> reset():
```

The easiest way to create matrices is given by the following input:

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

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

Don't worry about the output of this statement which is described more detailed in the report "Advanced Demonstration Tour" in chapter "Linear Algebra".

What you need to know is that you have created a set of matrices (using `Dom::Matrix`) where the entries of a matrix can be arbitrary expressions like:

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

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

We entered this matrix by giving a list of rows where each row is a list of row entries. A certain entry of this matrix can be extracted as follows:

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

    sin(x)
```

In a similar way it is possible to extract a certain submatrix:

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

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

Likewise easily is to append a matrix, e.g. the 4×2 zero matrix, to `A`:

```
>> 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  |
    +-                             -+
```

But what is about matrix arithmetic? You can perform matrix arithmetic by using the standard arithmetic operators of MuPAD, e.g.

```
>> 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    |
    +-         -+  +-         -+
```

The following computes the inverse of a matrix, if it exists (otherwise the value `FAIL` will be returned):

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

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

We check the result:

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

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

But that may not be sufficient for doing linear algebra, and therefore MuPAD offers a certain *library* called `linalg`.

A *library* in MuPAD always means a collection of functions for a special purpose like for number theory as well as for string operations or for generating Fortran or C code.

Let us see of which functions the library `linalg` consits:

```
>> 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,             linalg::concatMatrix,
.....
```

To access a function of a library you have to enter the library name followed by two semicolons and the name of the library function.

Thus, to compute the eigensystem of the matrix

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

you have to enter:

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

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

If this is found to be too awkward the methods of the linalg package may be exported:

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

After this call the function `linalg::eigenVectors` is also directly available.

We demonstrate this by computing the characteristic polynomial of the above matrix that can be done directly with the function `charPolynomial` or with the following input:

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

             2
    - 3 x + x  + 2
```

Note that `det` and `charMatrix` are exported functions of the `linalg` package.

2.3 Graphics and Visualization

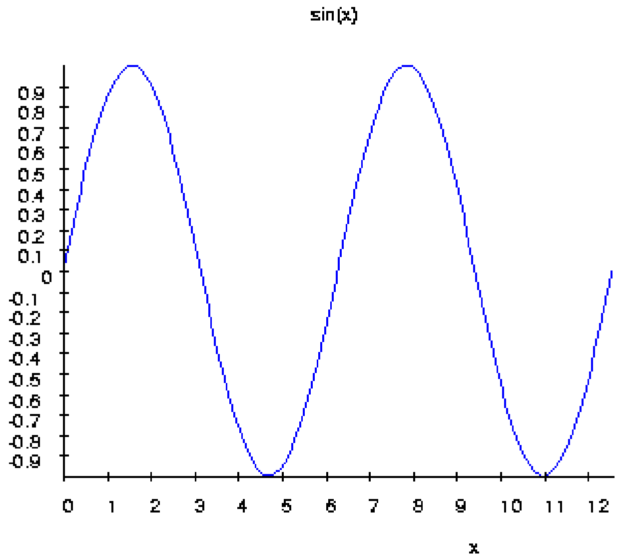
```
>> reset():
```

2.3.1 About `plotfunc`

MuPAD offers the possibility to easily create graphical scenes in different ways. The easiest and most straightforward way is to use the command `plotfunc` which can plot simple function graphs.

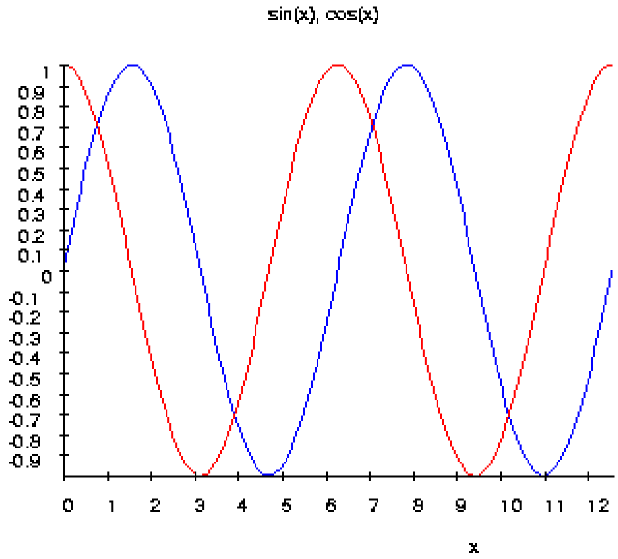
We already have seen an example in section 2.1. Other examples may be given by plotting the graph of the sine-function at $[0, 4\pi]$:

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



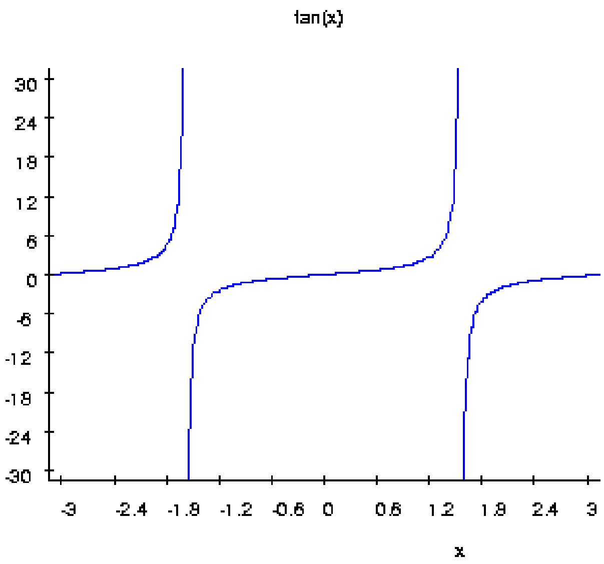
or by plotting more than one function graph in one scene:

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



This function is also able to handle singularities of a function, as it can be seen by the following example:

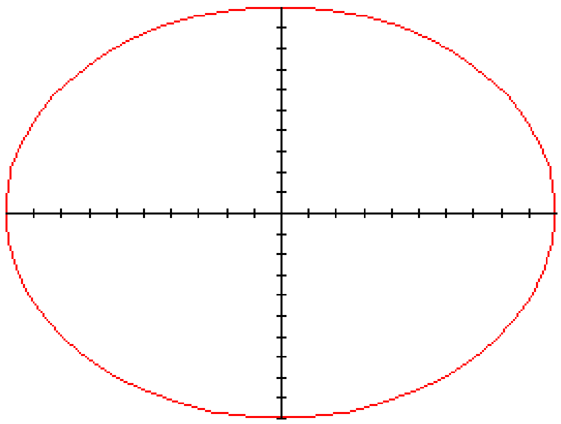
```
>> plotfunc( tan(x),x=-PI..PI );
```



2.3.2 The `plot2d` and `plot3d` Functions

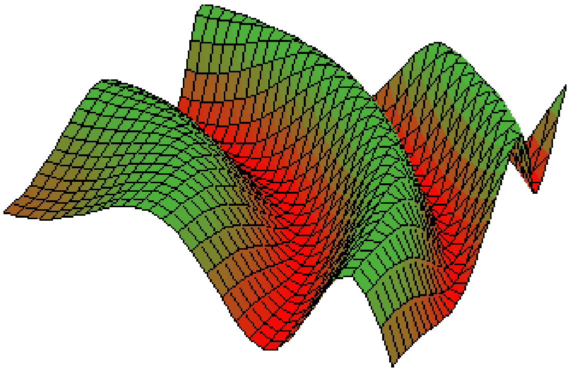
With `plot2d` you are not restricted to function graphs but can create a large variety of different graphical, two-dimensional scenes.

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



In addition to the two-dimensional scenes created by `plot2d`, MuPAD offers the function `plot3d` to plot three-dimensional graphics, in particular 3D-curves and -surfaces:

```
>> 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]
          ]);
```



Plot of sin(u^2+v^2)

2.3.3 The `plotlib` Library

Advanced routines for plotting graphical scenes can be found in the library `plotlib`[2].

You can, for example, rotate a curve around the x-axis using `plotlib::xrotate`:

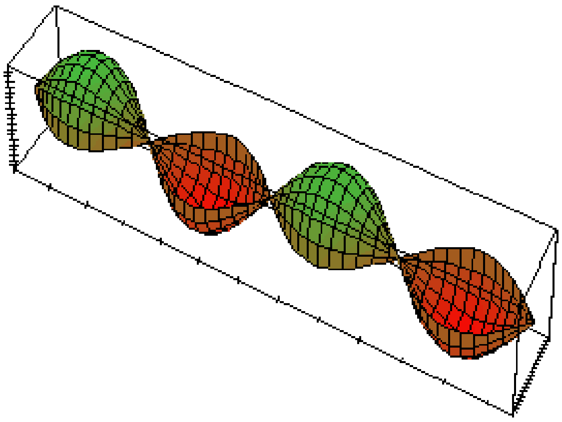
```
>> plotlib::xrotate( Axes=Box, [ [x, sin(x)], x=[0,4*PI], Angle = [0,PI],
                       Color=[Height], Grid=[50,10]
                     ]
   );
```

Likewise, function graphs can also be rotated around the y-axis using `plotlib::yrotate`.

As a final example let us demonstrate how visualize a field of vectors using `plotlib::fieldplot`

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

[2]As usual, an overview about the functions of this library can be obtained with `info(plotlib)`.



2.4 Simplify and Rewrite Arithmetical Expressions

```
>> reset():
```

Often results of computation (made by hand or by computer algebra systems) are in a complicated form and simplification or rewriting may be helpful to make the result more readable and usable.

The most general function for simplification of expressions is `simplify`:

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

    0
```

It can be used to ask for zero equivalence or to simplify expressions like $\sqrt{4+2\cdot\sqrt{3}} = \sqrt{3}+1$:

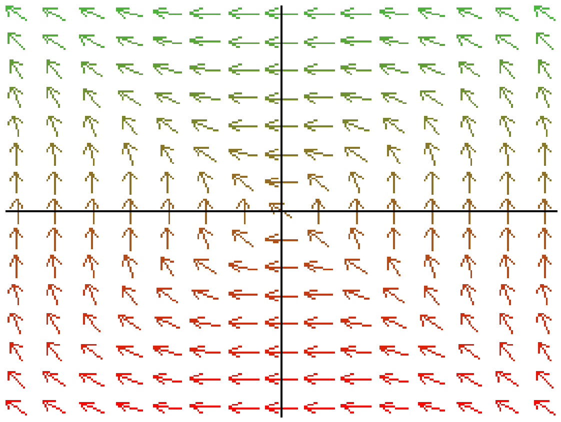
```
>> simplify( sqrt(4+2*sqrt(3)), sqrt );

     1/2
    3    + 1
```

The next examples demonstrate how MuPAD can be used to rewrite mathematical expressions (which also may mean a simplification):

```
>> 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)
```

These expressions are in an expanded form and can factor them using the function `Factor`:

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

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

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

A contrary of `Factor` concerning *rational functions* is the function `expand`. It works out an expression which often means that sum rules are applied.

But it does more than `Factor` because it also expands more general expressions involving trigonometric, exponential or logarithmic functions, for instance.

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

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

A function for rewriting a term into a certain equivalent representation is `rewrite`. The following example converts sinus and cosinus terms into their complex exponential form:

```
>> 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


>> simplify( % );

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

We have seen that MuPAD offers the functions `simplify`, `expand` and `rewrite` to simplify or rewrite mathematical expressions, as well as the functions `combine`, `normal` or `rectform`, for which the reader is referred to the corresponding help pages.

3 Basic Data Types

Starting MuPAD, a lot of data types like lists, sets, tables or a type for polynomials are known. We call these the *basic data* types to point out the difference to data types which are created and known to the system afterwards. The user is able to create data types with all the necessary operations to be used for them (so-called *domains*).

But this may need experiences in the use of MuPAD and its programming language. We demonstrate this characteristic feature of MuPAD in the “Advanced Demonstration Tour”. We will give two examples in chapter 5 which demonstrate what domains are good for.

Now let us concentrate on the basic data types.

3.1 Numbers, Expressions, Polynomials and more

Please initialize the session:

```
>> reset():
```

MuPAD offers the user different kinds of numbers and constants:

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

    2, -2/3, 2 + 3 I, PI, 3.141592653, E, 2.718281828
```

```
>> map( %, domtype );

    DOM_INT, DOM_RAT, DOM_COMPLEX, DOM_IDENT, DOM_FLOAT,

    DOM_IDENT, DOM_FLOAT
```

Here we made use of the function `map` which applies the function `domtype` to the sequence of different objects given above.

So-called *identifiers* (which may represent variables in a mathematical context) and strings are also known to MuPAD:

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

    DOM_IDENT, DOM_IDENT, DOM_EXPR


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

    "The Computer Algebra System MuPAD"
```

There are many functions to operate on such objects (like extracting of substrings, intersecting of sets and so on). We will see some of them during our demonstration.

Expressions are the most general form of a data type and they can be build up from objects of any other data type:

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

    PI   sin(x)          1
    -- + ------ + f(2) + --
    2    cos(x)           2
                         x


>> domtype( ex );

    DOM_EXPR
```

Polynomials can be represented as expressions, like in

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

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

    DOM_EXPR
```

But to achieve more efficiency in handling polynomials MuPAD offers a special data type for univariate and multivariate polynomials:

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

                4    3         2
    poly(x  + x  + (-4) x  + 3 x - 1, [x])

    DOM_POLY
```

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

    poly(x - 1, [x])

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

          3      2
    poly(x  + 2 x  + (-2) x + 1, [x]), poly(

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

See the function `poly` for more information about polynomials.

3.2 Sets and Lists

```
>> reset():
```

A set is entered as follows:

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

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

>> domtype( M );

    DOM_SET
```

A set is builded up by a so-called *expression sequence* of their elements:

```
>> op( M );

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

You can directly generate such sequences using the so-called *sequence operator* `$`:

```
>> 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
```

An index variable (here: `i`) inside the `$` operator must have no value, otherwise the system will report an error message.

The function `unassign` reset an identifier to have no value.

Sequences may also be important when dealing with functions like the functional equivalents of the operators + or *:

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

    25502500

>> _mult( s );

    81285103704665697929058034741394527800954175275203119077085\

    79474767088848233730596856720188375050478138776220712647125\

    92314115920641160919935403754583649069843601261900051908970\

    24813510723449889579660946315033449388079966874258629176303\

    02052505909887462286075836527716233413659162900924769568594\

    29554672135618951275111007717373291473301054034842043089511\

    58469957099274146970547638354741532999364798054400000000000\

    00000000000000000000000000000000000000000000000000000000000\

       00
```

Let us now show some basic operations for sets:

```
>> 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
```

In contrast to sets, to be able to keep the order of elements of a sequence, use *lists* instead of sets:

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

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

For lists, op can be used to extract their entries or use the index operator:

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

    b, c
```

The following will be reported with an error message by your system because the given index is not in the range $1 \ldots 5$:

```
>> L[100];

    Error: Invalid index [list]
```

Elements of a list can be removed by the use of `unassign`:

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

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

You can concatenate lists by using the *dot operator*, an abbreviation for the `_concat` function:

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

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

The dot operator can also be used to concatenate strings, arrays (see next section) or matrices, for example.

3.3 Arrays and Tables

```
>> reset():
```

In MuPAD it is possible to handle multi-dimensional arrays:

```
>> 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
```

Arrays are restricted to have natural numbers for their indices. However, so-called *tables* can be used if an arbitrary expression for indexing is wanted:

```
>> 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 Testing on Types

```
>> reset():
```

MuPAD offers the user different methods to test on types of expressions.

The function `domtype` was already seen in previous examples. It is used to determine the **data** type of an expression.

But it is often needed to have more information about the expression, for example that

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

                                                6
                                2       (a - 1)
    y + cos(z) sin(z) + x   + -----------------
                                  2        2
                              (a  + 1) (a  - 1)
```

is a sum, or that `[a, b, c]` is a list of three identifiers.

3.4.1 The Function `type`

The function `type` can be used to determine more information about an arbitrary expression:

```
>> type( ex );

    "_plus"
```

If we map the function `type` on the operands of `ex` we get the following information about the terms of `ex`:

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

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

The result of `type` is either a string or an identifier. If the last case occurs, then the result of `type` is the data type of its argument:

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

    DOM_LIST
```

The function `testtype` can be used to ask for a special type, e.g.

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

    TRUE
```

However, the function `type` does not provide a *comfortable* method to test on more complex types, like alternative of types, or for types which may depend on parameters (e.g. the coefficient domain for polynomials).

So, how it is possible to determine (in a comfortable way) that

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

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

is a *list* of *four positive integers*?

3.4.2 The `Type` Library

The library `Type` was designed for a comfortable way of determining a certain type structure of objects. It offers predefined so-called *type expressions*, like

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

    TRUE
```

Let us give further examples for the use of this library. Of course, they do not reflect the whole possibilities offered by this library (enter `info( Type )` for a list of the pre-defined type expressions and see the corresponding help pages for details).

- *How to test whether an object is a positive integer* or *an identifier?*

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

      TRUE
  ```

- *How to test on a polynomial in 2 indeterminates over the rationals?*

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

      TRUE
  ```

- *How to test whether a series expansion is of type Taylor, Laurent or Puiseux?*

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

                  3
  1   x   7 x        4
  - + - + ---- + O(x )
  x   6   360
  ```

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

    FALSE

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

    TRUE
```

3.5 Manipulation of Objects

```
>> reset():
```

MuPAD is a powerful tool for manipulating objects of its data types. An object of a certain data type consists of a corresponding number of *operands* which can be extracted, removed or substituted by a subexpression.

Let us demonstrate this capability first on an easy example. The expression `a+b+c` is a sum of three operands:

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

    3

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

    a, b, c
```

Functions like `subs` and `subsop` allow to substitute subsexpressions or operands by new expressions:

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

    b + c + 1
```

Note that `subs` does only a *syntactical* substitution.

The following input will not work because with `subs` only those subsexpressions can be substituted which can be *directly* extracted with `op`:

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

    a + b + c
```

whereas the following will work:

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

    d e + 1
```

However, to achieve the substitution also of incomplete subsexpressions the function `subsex` can be used:

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

    c + 1
```

The functions shown above can be applied to expressions as well as sets, lists, tables, arrays and the other data types known to the system. Here are some further examples:

```
>> 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}
```

Thus, an element of a set can be extracted using the function `op`. But as you can see with the last output ordering of entries of a set depends on the internal ordering used by MuPAD which can not be influenced by the user. Hence, to speak of *the* 5nd operand of a set may not make sense if the set was manipulated before.

Let us extract operands of a table:

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

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


>> op( S );

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

As you can see its operands are *equations* with the indices on their left sides and the corresponding entries on the right sides.

Now, to substitute a certain index, say `a+b`, by a new one you have to give the first of the second operand of that table:

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

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

As already mentioned, a procedure is of the basic data type `DOM_PROC` and consist of several operands:

```
>> 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
```

You see the body, the name and the arguments of the procedure. The complete description of the operands can be found in the manual.

Now let us change the meaning of this procedure in order to compute the minimum of two arguments:

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

    proc(a, b)
      name Min;
    begin
      if not(a < b) then
        b
      else
        a
      end_if
    end_proc

>> Min( 2, 10 );

    2
```

Note that also each MuPAD statement is a valid expression with several operands. In the last example, we made a substitution on the operands of a `if−then` statement.

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

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

You see that MuPAD has a quite flexible possibility to manipulate arbitrary objects.

4 The MuPAD Programming Language

MuPAD provides the user with a high-level programming language together with profile and debug tools.

4.1 Basic Language Statements

```
>> reset():
```

Let us begin our demonstration by introducing some basic statements for loops and case switches[3]:

```
>> 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]
```

[3]some of them we already have seen in previous examples

The MuPAD language also consists of functional aspects which allows you to write the above sequences also in the following function call:

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

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

A procedure may have an arbitrary number of arguments. The following example extracts the maximum in a squence of randomly chosen numbers:

```
>> 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
```

Of course, the output may differ to yours.

4.2 Performance and Profiling

```
>> reset():
```

The function `time` returns the total CPU time (in ms) used to evaluate its argument:

```
>> time( _plus(1/k $ k=1..1000) );

    390
```

Note that this output depends on the machine you entered the command.

To find out how much time an evaluation spends in which procedure, from where every procedure is invoked issue the command `profile`:

```
>> 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: 30 ms
    -----------------
    fib: 99.9 % 30 ms total 41 call(s) 0 lookup(s) 0.7 ms/call

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

The use of the option `remember` for a procedure would lead to considerable improvement of its running time if the procedure is often called with identical arguments:

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

   profile( fib(7) );

    Total time: 10 ms
    -----------------
    fib: 99.9 % 10 ms total 13 call(s) 5 lookup(s) 0.7 ms/call

    <fib> calls
       fib : 12 time(s)

                                                      21
```

We see that there were 5 of 13 calls which mean that the function was repeadedly called with identical arguments whereby the corresponding function values had not been calculated again but were taken from the remember table of the procudure.

The remember table for a procudure with option `remember` is stored in its the 5th operand:

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

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

Let us finally profile the call

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

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

to give a more complex example:

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

```
Total time: 30 ms
-----------------
factor           : 33.3 % 10 ms total 1 call(s) 0 lookup(s) 10.0 ms/call
anonymous        : 33.3 % 10 ms total 3 call(s) 0 lookup(s) 3.3 ms/call
faclib::sbinomial: 33.3 % 10 ms total 1 call(s) 1 lookup(s) 10.0 ms/call
faclib::monomial :  0.0 % 0 ms total 1 call(s) 0 lookup(s) 0.0 ms/call
sign             :  0.0 % 0 ms total 1 call(s) 0 lookup(s) 0.0 ms/call
Type::PolyOf     :  0.0 % 0 ms total 1 call(s) 0 lookup(s) 0.0 ms/call
Type::testtype   :  0.0 % 0 ms total 1 call(s) 0 lookup(s) 0.0 ms/call
faclib::binomial :  0.0 % 0 ms total 1 call(s) 0 lookup(s) 0.0 ms/call
faclib::pfactor  :  0.0 % 0 ms total 1 call(s) 0 lookup(s) 0.0 ms/call

<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)
   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]
```

There a further utilites for programming in MuPAD, like a source code debugger, a trace function (see `trace` but also `sharelib::trace`), a possibility for checking on global and local variables of procedures (see `misc::checkFunction`) or a function to create a test coverage (see `misc::tcov`).

5 Examples for Working with Domains

In chapter 3 we indicate a possibility to create user defined data types, so-called *domains*.

In this chapter we want to give you an impression not on how to implement a data type but on what they may be good for.

We use here already implemented data types comming up with your MuPAD package. Please note that they do *not* belong to data types offered by the MuPAD system, which we called the *basic data types*.

Both examples will show that functions of the linear algebra package can operate on user defined data types (if they were written in a sufficient way). Such algorithm are called *generic*, hence they do not depend on a fixed set of data types but on several (often algebraic) properties of the given objects.

5.1 Interval Arithmetic

Please initialize the session:

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

Consider the following problem:

1. solve the system $x + 5y = -2, 2x + y = 4$;
2. solve the same system, after perturbing each coefficient by 0.01.

To solve this problem, we make use of the `linalg` library:

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

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

With the function `linalg::linearSolve` it is easy to solve the first task:

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

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

To solve the second task we will use the domain `Dom::Interval` representing real intervals, which builds a field by using the standard interval arithmetic.

Now we define the matrices over this field of real intervals and define the same matrix as A and b but perturbing each coefficent by 0.01:

```
>> 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  |  |
    -- +-                        -+  +-            -+ --
```

and solve the new linear system:

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

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

This means that if you know your coefficients with a precision of 0.01, then the error on the result is about 0.14 on x by taking $x = 2.447$, and about 0.02 on y by taking $y = -0.89$.

5.2 Block Matrices

Please initialize the session:

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

We want to show a small but nice example how to define and to do arithmetic with block matrices.

Let us first define the ring of 2×2 matrices over $\mathbb{Q}$ (see `Dom::SquareMatrix`):

```
>> SqMat:= Dom::SquareMatrix( 2,Dom::Rational );

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

Now we define four square matrices:

```
>> 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 |
    +-     -+
```

The following defines the set of matrices over the ring of 2×2 matrices:

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

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

We define a matrix containing `a, b, c` and `d`

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

    +-                    -+
    |  +-     -+  +-     -+ |
    |  | 2, 3 |  |  1, 0 | |
    |  |      |, |       | |
    |  | 1, 0 |  | -2, 3 | |
    |  +-     -+  +-     -+ |
    |                       |
    | +-      -+  +-     -+ |
    | | 0, -4 |  | 0,  1 | |
    | |       |, |       | |
    | | 2,  1 |  | 2, -1 | |
    | +-      -+  +-     -+ |
    +-                    -+
```

and some random matrices:

```
>> 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 | |
     |     +-    -+     +-          -+ |
     +-                              -+
```

Please note that your output may be different (we have created the matrices randomly). We can now perform standard arithmetic:

```
>> 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 | |
    | +-                  -+  +-                  -+ |
    +-                                               -+
```

Finally we want to compute the inverse of matrix `A`. It is well known that for 2×2 matrices a, b, c, d the matrix $[[a, b], [c, d]]$ is invertible if $\det(a) <> 0$ and $\det(d - \frac{b\ c}{a}) <> 0$.

Let us prove the suppositions:

```
>> det( a );

    -3


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

    -3
```

Hence we can apply the following formula for the inverse of `a`:

```
>> 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 | |
    | +-          -+  +-            -+ |
    +-                              -+
```

Finally, we check our result:

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

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

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