EMT untuk Perhitungan Aljabar

Rasyid Shalahuddin
22305144016
matematika E 2022

Pada notebook ini Anda belajar menggunakan EMT untuk melakukan berbagai perhitungan terkait dengan materi atau topik dalam Aljabar. Kegiatan yang harus Anda lakukan adalah sebagai berikut:

- Membaca secara cermat dan teliti notebook ini;
- Menerjemahkan teks bahasa Inggris ke bahasa Indonesia;
- Mencoba contoh-contoh perhitungan (perintah EMT) dengan cara meng-ENTER setiap perintah EMT yang ada (pindahkan kursor ke baris perintah)
- Jika perlu Anda dapat memodifikasi perintah yang ada dan memberikan keterangan/penjelasan tambahan terkait hasilnya.
- Menyisipkan baris-baris perintah baru untuk mengerjakan soal-soal Aljabar dari file PDF yang saya berikan;
- Memberi catatan hasilnya.
- Jika perlu tuliskan soalnya pada teks notebook (menggunakan format LaTeX).
- Gunakan tampilan hasil semua perhitungan yang eksak atau simbolik dengan format LaTeX. (Seperti contoh-contoh pada notebook ini.)

Contoh pertama

Menyederhanakan bentuk aljabar:

$$6x^{-3}y^5 \times -7x^2y^{-9}$$

```
>$&6*x^(-3)*y^5*-7*x^2*y^(-9)
```

Menjabarkan:

```
>$&showev('expand((6*x^(-3)+y^5)*(-7*x^2-y^(-9))))
```

The Command Line

A command line of Euler consists of one or several Euler commands followed by a semicolon ";" or a comma ",". The semicolon prevents the printing of the result. The comma after the last command can be omitted.

The following command line will only print the result of the expression, not the assignments or the format commands.

```
>r:=2; h:=4; pi*r^2*h/3
```

```
 16.7551608191
```

Commands must be separated with a blank. The following command line prints its two results.

```
>pi*2*r*h, %+2*pi*r*h // Ingat tanda % menyatakan hasil perhitungan terakhir sebelumnya
```

```
 50.2654824574
```

```
    100.530964915
```

Command lines are executed in the order the user presses return. So you get a new value each time you execute the second line.

```
>x := 1;
>x := cos(x) // nilai cosinus (x dalam radian)
```

```
    0.540302305868
```

```
>x := cos(x)
```

```
    0.857553215846
```

If two lines are connected with "..." both lines will always execute simultaneously.

```
>x := 1.5; ...
 x := (x+2/x)/2, x := (x+2/x)/2, x := (x+2/x)/2,
```

```
    1.41666666667
    1.41421568627
    1.41421356237
```

This is also a good way to spread a long command over two or more lines. You can press Ctrl+Return to split a line in two at the current cursor position, or Ctlr+Back to join the lines.

To fold all multi-lines press Ctrl+L. Then the subsequent lines will only be visible, if one of them has the focus. To fold a single multi-line start the first line with "%+ ".

```
>%+ x=4+5; ...
  // This line will not be visible once the cursor is off the line
```

A line starting with %% will be completely invisible.

```
    81
```

Euler supports loops in command lines, as long as they fit into one single line or a multi-line. In programs, this restrictions does not hold, of course. For more information consult the following introduction.

```
>x=1; for i=1 to 5; x := (x+2/x)/2, end; // menghitung akar 2
```

```
    1.5
    1.41666666667
    1.41421568627
    1.41421356237
    1.41421356237
```

It is okay to use a multi-line. Make sure the line ends with " ...".

```
>x := 1.5; // comments go here before the ...
 repeat xnew:=(x+2/x)/2; until xnew~=x; ...
    x := xnew; ...
 end; ...
 x,
```

```
1.41421356237
```

Conditional structures do also work.

```
>if E^pi>pi^E; then "Thought so!", endif;
```

```
Thought so!
```

When you execute a command, the cursor can be at any position in the command line. You can go back to a previous command or skip to the next command with the arrow keys. Or you can click into the comment section above the command to go to the command.

When you move the cursor along the line the opening and closing pairs of brackets or parentheses will highlight. Also, watch the status line. After the opening bracket of the sqrt() function, the status line will display a help text for the function. Execute the command with the return key.

```
>sqrt(sin(10°)/cos(20°))
```

```
0.429875017772
```

To see help for the most recent command, open the help window with F1. There, you can enter text to search for. On an empty line, the help for the help window will be displayed. You can press escape to clear the line, or to close the help window.

You can double click on any command to open the help for this command. Try double clicking the exp command below in the command line.

```
>exp(log(2.5))
```

```
2.5
```

You can copy and paste in Euler too. Use Ctrl-C and Ctrl-V for this. To mark a text, drag the mouse or use shift together with any cursor key. Moreover, you can copy the highlighted brackets.

Basic Syntax

Euler knows the usual mathematical functions. As you have seen above, trigonometric functions work in radian or degree. To convert to degrees, append the degree symbol (with the F7 key) to the value, or use the function rad(x). The square root function is called sqrt in Euler. Of course, x^(1/2) is also possible.

To set variables, use either "=" or ":=". For the sake of clarity, this introduction uses the latter form. Spaces do not matter. But a space between commands is expected.

Multiple commands in one line are separated with "," or ";". The semicolon suppresses the output of the command. At the end of the command line a "," is assumed, if ";" is missing.

```
>g:=9.81; t:=2.5; 1/2*g*t^2
```

```
30.65625
```

EMT uses a programming syntax for expressions. To enter

you have to set the correct brackets and use / for fractions. Watch the highlighted brackets for assistance. Note that the Euler constant e is named E in EMT.

```
>E^2*(1/(3+4*log(0.6))+1/7)
```

```
 8.77908249441
```

To compute a complicate expression like

you need to enter it in line form.

```
>((1/7 + 1/8 + 2) / (1/3 + 1/2))^2 * pi
```

```
 23.2671801626
```

Carefully put brackets around sub-expressions that need to be computed first. EMT assists you by highlighting the expression that the closing bracket finishes. You will also have to enter the name "pi" for the Greek letter pi.

The result of this computation is a floating point number. It is by default printed with about 12 digits accuracy. In the following command line, we also learn how we can refer to the previous result within the same line.

```
>1/3+1/7, fraction %
```

```
 0.47619047619
 10/21
```

An Euler command can be an expression or a primitive command. An expression is made of operators and functions. If necessary, it must contain brackets to force the correct order of execution. In doubt, setting a bracket is a good idea. Note that EMT shows opening and closing brackets while editing the command line.

```
>(cos(pi/4)+1)^3*(sin(pi/4)+1)^2
```

```
 14.4978445072
```

The numerical operators of Euler include

```
+ unary or operator plus
- unary or operator minus
*, /
. the matrix product
a^b power for positive a or integer b (a**b works too)
n! the factorial operator
```

and many more.

Here are some of the functions you might need. There are many more.

```
sin,cos,tan,atan,asin,acos,rad,deg
log,exp,log10,sqrt,logbase
```

```
bin,logbin,logfac,mod,floor,ceil,round,abs,sign
conj,re,im,arg,conj,real,complex
beta,betai,gamma,complexgamma,ellrf,ellf,ellrd,elle
bitand,bitor,bitxor,bitnot
```

Some commands have aliases, e.g. ln for log.

```
>ln(E^2), arctan(tan(0.5))
```

```
2
0.5
```

```
>sin(30°)
```

```
0.5
```

Make sure to use parentheses (round brackets), whenever there is doubt about the order of execution! The following is not the same as (2^3)^4, which is the default for 2^3^4 in EMT (some numerical systems do it the other way).

```
>2^3^4, (2^3)^4, 2^(3^4)
```

```
2.41785163923e+24
4096
2.41785163923e+24
```

Real Numbers

The primary data type in Euler is the real number. Reals are represented in IEEE format with about 16 decimal digits of accuracy.

```
>longest 1/3
```

```
     0.3333333333333333
```

The internal dual representation takes 8 bytes.

```
>printdual(1/3)
```

```
1.0101010101010101010101010101010101010101010101010101*2^-2
```

```
>printhex(1/3)
```

```
5.5555555555554*16^-1
```

Strings

A string in Euler is defined with "...".

```
>"A string can contain anything."
```

```
 A string can contain anything.
```

Strings can be concatenated with | or with +. This also works with numbers, which are converted to strings in that case.

```
>"The area of the circle with radius " + 2 + " cm is " + pi*4 + " cm^2."
```

```
 The area of the circle with radius 2 cm is 12.5663706144 cm^2.
```

The print function does also convert a number to a string. It can take a number of digits and a number of places (0 for dense output), and optimally a unit.

```
>"Golden Ratio : " + print((1+sqrt(5))/2,5,0)
```

```
 Golden Ratio : 1.61803
```

There is a special string none, which does not print. It is returned by some functions, when the result does not matter. (It is returned automatically, if the function does not have a return statement.)

```
>none
```

To convert a string to a number simply evaluate it. This works for expressions too (see below).

```
>"1234.5"()
```

```
 1234.5
```

To define a string vector, use the vector [...] notation.

```
>v:=["affe","charlie","bravo"]
```

```
 affe
 charlie
 bravo
```

The empty string vector is denoted by [none]. String vectors can be concatenated.

```
>w:=[none]; w|v|v
```

```
 affe
 charlie
 bravo
 affe
 charlie
 bravo
```

Strings can contain Unicode characters. Internally, these strings contain UTF-8 code. To generate such a string, use u"..." and one of the HTML entities.

Unicode strings can be concatenated like other strings.

```
>u"&alpha; = " + 45 + u"&deg;" // pdfLaTeX mungkin gagal menampilkan secara benar
```

```
 α = 45°
```

I

In comments, the same entities like α, β etc. can be used. This may be a quick alternative to Latex. (More details on comments below).

There are some functions to create or analyze unicode strings. The function strtochar() will recognize Unicode strings, and translate them correctly.

```
>v=strtochar(u"&Auml; is a German letter")
```

```
 [196,  32,  105,  115,  32,  97,  32,  71,  101,  114,  109,  97,  110,
 32,  108,  101,  116,  116,  101,  114]
```

The result is a vector of Unicode numbers. The converse function is chartoutf().

```
>v[1]=strtochar(u"&Uuml;")[1]; chartoutf(v)
```

```
 Ü is a German letter
```

The function utf() can translate a string with entities in a variable into a Unicode string.

```
>s="We have &alpha;=&beta;."; utf(s) // pdfLaTeX mungkin gagal menampilkan secara benar
```

```
 We have α=β.
```

It is also possible to use numerical entities.

```
>u"&#196;hnliches"
```

```
 Ähnliches
```

Boolean Values

Boolean values are represented with 1=true or 0=false in Euler. Strings can be compared, just like numbers.

```
>2<1, "apel"<"banana"
```

```
 0
 1
```

"and" is the operator "&&" and "or" is the operator "||", as in the C language. (The words "and" and "or" can only be used in conditions for "if".)

```
>2<E && E<3
```

```
  1
```

Boolean operators obey the rules of the matrix language.

```
>(1:10)>5, nonzeros(%)
```

```
  [0,  0,  0,  0,  0,  1,  1,  1,  1,  1]
  [6,  7,  8,  9,  10]
```

You can use the function nonzeros() to extract specific elements form a vector. In the example, we use the conditional isprime(n).

```
>N=2|3:2:99 // N berisi elemen 2 dan bilangan2 ganjil dari 3 s.d. 99
```

```
  [2,  3,  5,  7,  9,  11,  13,  15,  17,  19,  21,  23,  25,  27,  29,
  31,  33,  35,  37,  39,  41,  43,  45,  47,  49,  51,  53,  55,  57,
  59,  61,  63,  65,  67,  69,  71,  73,  75,  77,  79,  81,  83,  85,
  87,  89,  91,  93,  95,  97,  99]
```

```
>N[nonzeros(isprime(N))] //pilih anggota2 N yang prima
```

```
  [2,  3,  5,  7,  11,  13,  17,  19,  23,  29,  31,  37,  41,  43,  47,
  53,  59,  61,  67,  71,  73,  79,  83,  89,  97]
```

Output Formats

The default output format of EMT prints 12 digits. To make sure that we see the default, we reset the format.

```
>defformat; pi
```

```
  3.14159265359
```

Internally, EMT uses the IEEE standard for double numbers with about 16 decimal digits. To see the full number of digits, use the command "longestformat", or we use the operator "longest" to display the result in the longest format.

```
>longest pi
```

```
        3.141592653589793
```

Here is the internal hexadecimal representation of a double number.

```
>printhex(pi)
```

```
 3.243F6A8885A30*16^0
```

The output format can be changed permanently with a format command.

```
>format(12,5); 1/3, pi, sin(1)
```

```
     0.33333
     3.14159
     0.84147
```

The default is format(12).

```
>format(12); 1/3
```

```
 0.333333333333
```

Functions like "shortestformat", "shortformat", "longformat" work for vectors in the following way.

```
>shortestformat; random(3,8)
```

```
   0.66    0.2   0.89   0.28   0.53   0.31   0.44    0.3
   0.28   0.88   0.27    0.7   0.22   0.45   0.31   0.91
   0.19   0.46  0.095    0.6   0.43   0.73   0.47   0.32
```

The default format for scalars is format(12). But this can be changed.

```
>setscalarformat(5); pi
```

```
 3.1416
```

The function "longestformat" set the scalar format too.

```
>longestformat; pi
```

```
 3.141592653589793
```

For reference, here is a list of the most important output formats.

```
shortestformat shortformat longformat, longestformat
format(length,digits) goodformat(length)
fracformat(length)
defformat
```

The internal accuracy of EMT is about 16 decimal places, which is the IEEE standard. Numbers are stored in this internal format.

But the output format of EMT can be set in a flexible way.

```
>longestformat; pi,
3.141592653589793
```

```
>format(10,5); pi
```

```
   3.14159
```

The default is defformat().

```
>defformat; // default
```

There are short operators which print only one value. The operator "longest" will print all valid digits of a number.

```
>longest pi^2/2
```

```
      4.934802200544679
```

There is also a short operator for printing a result in fractional format. We have already used it above.

```
>fraction 1+1/2+1/3+1/4
```

```
25/12
```

Since the internal format uses a binary way to store numbers, the value 0.1 will not be represented exactly. The error adds up a bit, as you see in the following computation.

```
>longest 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1-1
```

```
 -1.110223024625157e-16
```

But with the default "longformat" you will not notice this. For convenience, the output of very small numbers is 0.

```
>0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1-1
```

```
0
```

Expressions

Strings or names can be used to store mathematical expressions, which can be evaluated by EMT. For this, use parentheses after the expression. If you intend to use a string as an expression, use the convention to name it "fx" or "fxy" etc. Expressions take precedence over functions.

Global variables can be used in the evaluation.

```
>r:=2; fx:="pi*r^2"; longest fx()
```

```
      12.56637061435917
```

Parameters are assigned to x, y, and z in that order. Additional parameters can be added using assigned parameters.

```
>fx:="a*sin(x)^2"; fx(5,a=-1)
```

```
-0.919535764538
```

Note that expression will always use global variables, even if there is a variable in a function with the same name. (Otherwise the evaluation of expressions in functions could have very confusing results for the user that called the function.)

```
>at:=4; function f(expr,x,at) := expr(x); ...
 f("at*x^2",3,5) // computes 4*3^2 not 5*3^2
```

```
36
```

If you want to use another value for "at" than the global value you need to add "at=value".

```
>at:=4; function f(expr,x,a) := expr(x,at=a); ...
 f("at*x^2",3,5)
```

```
45
```

For reference, we remark that call collections (discussed elsewhere) can contain expressions. So we can make the above example as follows.

```
>at:=4; function f(expr,x) := expr(x); ...
 f({{"at*x^2",at=5}},3)
```

```
45
```

Expressions in x are often used just like functions.
Note that defining a function with the same name like a global symbolic expression deletes this variable to avoid confusion between symbolic expressions and functions.

```
>f &= 5*x;
>function f(x) := 6*x;
>f(2)
```

```
12
```

By way of convention, symbolic or numerical expressions should be named fx, fxy etc. This naming scheme should not be used for functions.

```
>fx &= diff(x^x,x); $&fx
```

A special form of an expression allows any variable as an unnamed parameter to the evaluation of the expression, not just "x", "y" etc. For this, start the expression with "@(variables) ...".

```
>"@(a,b) a^2+b^2", %(4,5)
```

```
@(a,b) a^2+b^2
41
```

This allows to manipulate expressions in other variables for functions of EMT which need an expression in "x".

The most elementary way to define a simple function is to store its formula in a symbolic or numerical expression. If the main variable is x, the expression can be evaluated just like a function.

As you see in the following example, global variables are visible during the evaluation.

```
>fx &= x^3-a*x;  ...
 a=1.2; fx(0.5)
```

```
-0.475
```

All other variables in the expression can be specified in the evaluation using an assigned parameter.

```
>fx(0.5,a=1.1)
```

```
-0.425
```

An expression needs not be symbolic. This is necessary, if the expression contains functions, which are only known in the numerical kernel, not in Maxima.

Symbolic Mathematics

EMT does symbolic math with the help of Maxima. For details, start with the following tutorial, or browse the reference for Maxima. Experts in Maxima should note that there are differences in the syntax between the original syntax of Maxima and the default syntax of symbolic expressions in EMT.

Symbolic math is integrated seamlessly into Euler with &. Any expression starting with & is a symbolic expression. It is evaluated and printed by Maxima.

First of all, Maxima has an "infinite" arithmetic which can handle very large numbers.

```
>$&44!
```

This way, you can compute large results exactly. Let us compute

```
>$& 44!/(34!*10!) // nilai C(44,10)
```

Of course, Maxima has a more efficient function for this (as does the numerical part of EMT).

```
>$binomial(44,10) //menghitung C(44,10) menggunakan fungsi binomial()
```

To learn more about a specific function double click on it. E.g., try double clicking on "&binomial" in the previous command line. This opens the documentation of Maxima as provided by the authors of that program.

You will learn that the following works too.

```
>$binomial(x,3) // C(x,3)
```

If you want to replace x with any specific value use "with".

```
>$&binomial(x,3) with x=10 // substitusi x=10 ke C(x,3)
```

That way you can use a solution of an equation in another equation.

Symbolic expressions are printed by Maxima in 2D form. The reason for this is a special symbolic flag in the string.

As you will have seen in previous and following examples, if you have LaTeX installed, you can print a symbolic expression with Latex. If not, the following command will issue an error message.

To print a symbolic expression with LaTeX, use $ infront of & (or you may ommit &) before the command. Do not run the Maxima commands with $, if you don't have LaTeX installed.

```
>$(3+x)/(x^2+1)
```

Symbolic expressions are parsed by Euler. If you need a complex syntax in one expression, you can enclose the expression in "...". To use more than a simple expression is possible, but strongly discouraged.

```
>&"v := 5; v^2"
```

```
                                    25
```

For completeness, we remark that symbolic expressions can be used in programs, but need to be enclosed in quotes. Moreover, it is much more effective to call Maxima at compile time if possible.

```
>$&expand((1+x)^4), $&factor(diff(%,x)) // diff: turunan, factor: faktor
```

Again, % refers to the previous result.

To make things easier we save the solution to a symbolic variable. Symbolic variables are defined with "&=".

```
>fx &= (x+1)/(x^4+1); $&fx
```

Symbolic expressions can be used in other symbolic expressions.

```
>$&factor(diff(fx,x))
```

A direct input of Maxima commands is available too. Start the command line with "::". The syntax of Maxima is adapted to the syntax of EMT (called the "compatibility mode").

```
>&factor(20!)
```

```
                             2432902008176640000
```

```
>::: factor(10!)
```

```
                                  8   4   2
                                 2   3   5   7
```

```
>:: factor(20!)
```

```
                          18   8   4   2
                         2    3   5   7   11  13  17  19
```

If you are an expert in Maxima, you may wish to use the original syntax of Maxima. You can do this with ":::".

```
>::: av:g$ av^2;
```

```
                                    2
                                   g
```

```
>fx &= x^3*exp(x), $fx
```

```
                                  3   x
                                 x   E
```

Such variables can be used in other symbolic expressions. Note, that in the following command the right hand side of &= is evaluated before the assignment to Fx.

```
>&(fx with x=5), $%, &float(%)
```

```
                                      5
                                 125 E

                          18551.64488782208
```

```
>fx(5)
```

```
    18551.6448878
```

For the evaluation of an expression with specific values of the variables, you can use the "with" operator.

The following command line also demonstrates that Maxima can evaluate an expression numerically with float().

```
>&(fx with x=10)-(fx with x=5), &float(%)
```

```
                                 10         5
                          1000 E    - 125 E

                         2.20079141499189e+7
```

```
>$factor(diff(fx,x,2))
```

To get the Latex code for an expression, you can use the tex command.

```
>tex(fx)
```

```
 x^3\,e^{x}
```

Symbolic expressions can be evaluated just like numerical expressions.

```
>fx(0.5)
```

```
 0.206090158838
```

In symbolic expressions, this does not work, since Maxima does not support it. Instead, use the "with" syntax (a nicer form of the at(...) command of Maxima).

```
>$&fx with x=1/2
```

The assignment can also be symbolic.

```
>$&fx with x=1+t
```

The command solve solves symbolic expressions for a variable in Maxima. The result is a vector of solutions.

```
>$&solve(x^2+x=4,x)
```

Compare with the numerical "solve" command in Euler, which needs a start value, and optionally a target value.

```
>solve("x^2+x",1,y=4)
```

```
 1.56155281281
```

The numerical values of the symbolic solution can be computed by evaluation of the symbolic result. Euler will read over the assignments x= etc. If you do not need the numerical results for further computations you can also let Maxima find the numerical values.

```
>sol &= solve(x^2+2*x=4,x); $&sol, sol(), $&float(sol)
```

```
 [-3.23607,  1.23607]
```

To get a specific symbolic solution, one can use "with" and an index.

```
>$&solve(x^2+x=1,x), x2 &= x with %[2]; $&x2
```

To solve a system of equations, use a vector of equations. The result is a vector of solutions.

```
>sol &= solve([x+y=3,x^2+y^2=5],[x,y]); $&sol, $&x*y with sol[1]
```

Symbolic expressions can have flags, which indicate a special treatment in Maxima. Some flags can be used as commands too, others can't. Flags are appended with "|" (a nicer form of "ev(...,flags)")

```
>$& diff((x^3-1)/(x+1),x) //turunan bentuk pecahan
>$& diff((x^3-1)/(x+1),x) | ratsimp //menyederhanakan pecahan
>$&factor(%)
```

Functions

In EMT, functions are programs defined with the command "function". It can be a one-line function or multiline function.
A one-line function can be numerical or symbolic. A numerical one-line function is defined by ":=".

```
>function f(x) := x*sqrt(x^2+1)
```

For an overview, we show all possible definitions for one-line functions. A function can be evaluated just like any built-in Euler function.

```
>f(2)

    4.472135955
```

This function will work for vectors too, obeying the matrix language of Euler, since the expressions used in the function are vectorized.

```
>f(0:0.1:1)

    [0,  0.100499,  0.203961,  0.313209,  0.430813,  0.559017,  0.699714,
    0.854459,  1.0245,  1.21083,  1.41421]
```

Functions can be plotted. Instead of expressions, we need only provide the function name.

In contrast to symbolic or numerical expressions, the function name must be provided in a string.

```
>solve("f",1,y=1)

    0.786151377757
```

By default, if you need to overwrite a built-in function, you must add the keyword "overwrite". Overwriting built-in functions is dangerous and can cause problems for other functions depending on them.

You can still call the built-in function as "_...", if it is function in the Euler core.

```
>function overwrite sin (x) := _sin(x°) // redine sine in degrees
>sin(45)

    0.707106781187
```

We better remove this redefinition of sin.

```
>forget sin; sin(pi/4)
```

```
    0.707106781187
```

Default Parameters

Numerical function can have default parameters.

```
>function f(x,a=1) := a*x^2
```

Omitting this parameter uses the default value.

```
>f(4)
```

```
    16
```

Setting it overwrites the default value.

```
>f(4,5)
```

```
    80
```

An assigned parameter overwrite it too. This is used by many Euler functions like plot2d, plot3d.

```
>f(4,a=1)
```

```
    16
```

If a variable is not a parameter, it must be global. One-line functions can see global variables.

```
>function f(x) := a*x^2
>a=6; f(2)
```

```
    24
```

But an assigned parameter overrides the global value.

If the argument is not in the list of pre-defined parameters, it must be declared with ":="!

```
>f(2,a:=5)
```

```
    20
```

Symbolic functions are defined with "&=". They are defined in Euler and Maxima, and work in both worlds. The defining expression is run through Maxima before the definition.

```
>function g(x) &= x^3-x*exp(-x); $&g(x)
```

Symbolic functions can be used in symbolic expressions.

```
>$&diff(g(x),x), $&% with x=4/3
```

They can also be used in numerical expressions. Of course, this will only work if EMT can interpret everything inside the function.

```
>g(5+g(1))
```

```
 178.635099908
```

They can be used to define other symbolic functions or expressions.

```
>function G(x) &= factor(integrate(g(x),x)); $&G(c) // integrate: mengintegralkan
```

```
>solve(&g(x),0.5)
```

```
 0.703467422498
```

The following works too, since Euler uses the symbolic expression in the function g, if it does not find a symbolic variable g, and if there is a symbolic function g.

```
>solve(&g,0.5)
```

```
 0.703467422498
```

```
>function P(x,n) &= (2*x-1)^n; $&P(x,n)
>function Q(x,n) &= (x+2)^n; $&Q(x,n)
>$&P(x,4), $&expand(%)
>P(3,4)
```

```
 625
```

```
>$&P(x,4)+ Q(x,3), $&expand(%)
>$&P(x,4)-Q(x,3), $&expand(%), $&factor(%)
>$&P(x,4)*Q(x,3), $&expand(%), $&factor(%)
>$&P(x,4)/Q(x,1), $&expand(%), $&factor(%)
>function f(x) &= x^3-x; $&f(x)
```

With &= the function is symbolic, and can be used in other symbolic expressions.

```
>$&integrate(f(x),x)
```

With := the function is numerical. A good example is a definite integral like

which can not be evaluated symbolically.

If we redefine the function with the keyword "map" it can be used for vectors x. Internally, the function is called for all values of x once, and the results are stored in a vector.

```
>function map f(x) := integrate("x^x",1,x)
>f(0:0.5:2)
```

```
[-0.783431,  -0.410816,  0,  0.676863,  2.05045]
```

Functions can have default values for parameters.

```
>function mylog (x,base=10) := ln(x)/ln(base);
```

Now the function can be called with or without a parameter "base".

```
>mylog(100), mylog(2^6.7,2)
```

```
2
6.7
```

Moreover, it is possible to use assigned parameters.

```
>mylog(E^2,base=E)
```

```
2
```

Often, we want to use functions for vectors at one place, and for individual elements at other places. This is possible with vector parameters.

```
>function f([a,b]) &= a^2+b^2-a*b+b; $&f(a,b), $&f(x,y)
```

Such a symbolic function can be used for symbolic variables.

But the function can also be used for a numerical vector.

```
>v=[3,4]; f(v)
```

```
17
```

There are also purely symbolic functions, which cannot be used numerically.

```
>function lapl(expr,x,y) &&= diff(expr,x,2)+diff(expr,y,2)//turunan parsial kedua
```

```
                    diff(expr, y, 2) + diff(expr, x, 2)
```

```
>$&realpart((x+I*y)^4), $&lapl(%,x,y)
```

But of course, they can be used in symbolic expressions or in the definition of symbolic functions.

```
>function f(x,y) &= factor(lapl((x+y^2)^5,x,y)); $&f(x,y)
```

To summarize

- &= defines symbolic functions,
- := defines numerical functions,
- &&= defines purely symbolic functions.

Solving Expressions

Expressions can be solved numerically and symbolically.

To solve a simple expression of one variable, we can use the solve() function. It needs a start value to start the search. Internally, solve() uses the secant method.

```
>solve("x^2-2",1)
```

```
 1.41421356237
```

This works for symbolic expression too. Take the following function.

```
>$&solve(x^2=2,x)
>$&solve(x^2-2,x)
>$&solve(a*x^2+b*x+c=0,x)
>$&solve([a*x+b*y=c,d*x+e*y=f],[x,y])
```

```
>px &= 4*x^8+x^7-x^4-x; $&px
```

Now we search the point, where the polynomial is 2. In solve(), the default target value y=0 can be changed with an assigned variable.
We use y=2 and check by evaluating the polynomial at the previous result.

```
>solve(px,1,y=2), px(%)
```

```
 0.966715594851
 2
```

Solving a symbolic expression in symbolic form returns a list of solutions. We use the symbolic solver solve() provided by Maxima.

```
>sol &= solve(x^2-x-1,x); $&sol
```

The easiest way to get the numerical values is to evaluate the solution numerically just like an expression.

```
>longest sol()
```

```
     -0.6180339887498949       1.618033988749895
```

To use the solutions symbolically in other expressions, the easiest way is "with".

```
>$&x^2 with sol[1], $&expand(x^2-x-1 with sol[2])
```

Solving systems of equations symbolically can be done with vectors of equations and the symbolic solver solve(). The answer is a list of lists of equations.

```
>$&solve([x+y=2,x^3+2*y+x=4],[x,y])
```

The function f() can see global variables. But often we want to use local parameters.

with a=3.

```
>function f(x,a) := x^a-a^x;
```

One way to pass the additional parameter to f() is to use a list with the function name and the parameters (the other way are semicolon parameters).

```
>solve({{"f",3}},2,y=0.1)
```

```
 2.54116291558
```

This does also work with expressions. But then, a named list element has to be used. (More on lists in the tutorial about the syntax of EMT).

```
>solve({{"x^a-a^x",a=3}},2,y=0.1)
```

```
 2.54116291558
```

Menyelesaikan Pertidaksamaan

Untuk menyelesaikan pertidaksamaan, EMT tidak akan dapat melakukannya, melainkan dengan bantuan Maxima, artinya secara eksak (simbolik). Perintah Maxima yang digunakan adalah fourier_elim(), yang harus dipanggil dengan perintah "load(fourier_elim)" terlebih dahulu.

```
>&load(fourier_elim)
```

```
        C:/Program Files/Euler x64/maxima/share/maxima/5.35.1/share/f\
 ourier_elim/fourier_elim.lisp
```

```
>$&fourier_elim([x^2 - 1>0],[x]) // x^2-1 > 0
>$&fourier_elim([x^2 - 1<0],[x]) // x^2-1 < 0
>$&fourier_elim([x^2 - 1 # 0],[x]) // x^-1 <> 0
>$&fourier_elim([x # 6],[x])
>$&fourier_elim([x < 1, x > 1],[x]) // tidak memiliki penyelesaian
>$&fourier_elim([minf < x, x < inf],[x]) // solusinya R
>$&fourier_elim([x^3 - 1 > 0],[x])
>$&fourier_elim([cos(x) < 1/2],[x]) // ??? gagal
```

```
>$&fourier_elim([y-x < 5, x - y < 7, 10 < y],[x,y]) // sistem pertidaksamaan
>$&fourier_elim([y-x < 5, x - y < 7, 10 < y],[y,x])
>$&fourier_elim((x + y < 5) and (x - y >8),[x,y])
>$&fourier_elim(((x + y < 5) and x < 1) or  (x - y >8),[x,y])
>&fourier_elim([max(x,y) > 6, x # 8, abs(y-1) > 12],[x,y])
```

```
        [6 < x, x < 8, y < - 11] or [8 < x, y < - 11]
  or [x < 8, 13 < y] or [x = y, 13 < y] or [8 < x, x < y, 13 < y]
  or [y < x, 13 < y]
```

```
>$&fourier_elim([(x+6)/(x-9) <= 6],[x])
```

The Matrix Language

The documentation of the EMT core contains a detailed discussion on the matrix language of Euler.

Vectors and matrices are entered with square brackets, elements separated by commas, rows separated by semicolons.

```
>A=[1,2;3,4]
```

```
                1              2
                3              4
```

The matrix product is denoted by a dot.

```
>b=[3;4]
```

```
                3
                4
```

```
>b' // transpose b
```

```
    [3,  4]
```

```
>inv(A) //inverse A
```

```
               -2              1
              1.5           -0.5
```

```
>A.b //perkalian matriks
```

```
               11
               25
```

```
>A.inv(A)
```

```
                1              0
                0              1
```

The main point of a matrix language is that all functions and operators work element for element.

```
>A.A
```

```
                7             10
               15             22
```

```
>A^2 //perpangkatan elemen2 A
```

```
                 1             4
                 9            16
```

```
>A.A.A
```

```
                37            54
                81           118
```

```
>power(A,3) //perpangkatan matriks
```

```
                37            54
                81           118
```

```
>A/A //pembagian elemen-elemen matriks yang seletak
```

```
                 1             1
                 1             1
```

```
>A/b //pembagian elemen2 A oleh elemen2 b kolom demi kolom (karena b vektor kolom)
```

```
          0.333333      0.666667
              0.75             1
```

```
>A\b // hasilkali invers A dan b, A^(-1)b
```

```
                -2
               2.5
```

```
>inv(A).b
```

```
                -2
               2.5
```

```
>A\A   //A^(-1)A
```

```
                 1             0
                 0             1
```

```
>inv(A).A
```

```
                1              0
                0              1
```

```
>A*A //perkalin elemen-elemen matriks seletak
```

```
                1              4
                9             16
```

This is not the matrix product, but a multiplication element by element. The same works for vectors.

```
>b^2 // perpangkatan elemen-elemen matriks/vektor
```

```
                9
               16
```

If one of the operands is a vector or a scalar it is expanded in the natural way.

```
>2*A
```

```
                2              4
                6              8
```

E.g., if the operand is a column vector its elements are applied to all rows of A.

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

```
                1              4
                3              8
```

If it is a row vector it is applied to all columns of A.

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

```
                2              6
                6             12
```

One can imagine this multiplication as if the row vector v had been duplicated to form a matrix of the same size as A.

```
>dup([1,2],2) // dup: menduplikasi/menggandakan vektor [1,2] sebanyak 2 kali (baris)
```

```
                1              2
                1              2
```

```
>A*dup([1,2],2)
```

```
                1              4
```

```
                    3            8
```

This does also apply for two vectors where one is a row vector and the other is a column vector. We compute i*j for i,j from 1 to 5. The trick is to multiply 1:5 with its transpose. The matrix language of Euler automatically generates a table of values.

```
>(1:5)*(1:5)' // hasilkali elemen-elemen vektor baris dan vektor kolom
```

```
                    1            2            3            4            5
                    2            4            6            8           10
                    3            6            9           12           15
                    4            8           12           16           20
                    5           10           15           20           25
```

Again, remember that this is not the matrix product!

```
>(1:5).(1:5)' // hasilkali vektor baris dan vektor kolom
```

```
    55
```

```
>sum((1:5)*(1:5)) // sama hasilnya
```

```
    55
```

Even operators like < or == work in the same way.

```
>(1:10)<6 // menguji elemen-elemen yang kurang dari 6
```

```
    [1,  1,  1,  1,  1,  0,  0,  0,  0,  0]
```

E.g., we can count the number of elements satisfying a certain condition with the function sum().

```
>sum((1:10)<6) // banyak elemen yang kurang dari 6
```

```
    5
```

Euler has comparison operators, like "==", which checks for equality.

We get a vector of 0 and 1, where 1 stands for true.

```
>t=(1:10)^2; t==25 //menguji elemen2 t yang sama dengan 25 (hanya ada 1)
```

```
    [0,  0,  0,  0,  1,  0,  0,  0,  0,  0]
```

From such a vector, "nonzeros" selects the non-zero elements.

In this case, we get the indices of all elements greater than 50.

```
>nonzeros(t>50) //indeks elemen2 t yang lebih besar daripada 50
```

```
[8,  9,  10]
```

Of course, we can use this vector of indices to get the corresponding values in t.

```
>t[nonzeros(t>50)] //elemen2 t yang lebih besar daripada 50
```

```
[64,  81,  100]
```

As an example, let us find all squares of the numbers 1 to 1000, which are 5 modulo 11 and 3 modulo 13.

```
>t=1:1000; nonzeros(mod(t^2,11)==5 && mod(t^2,13)==3)
```

```
[4,  48,  95,  139,  147,  191,  238,  282,  290,  334,  381,  425,
433,  477,  524,  568,  576,  620,  667,  711,  719,  763,  810,  854,
862,  906,  953,  997]
```

EMT is not completely effective for integer computations. It uses double precision floating point internally. However, it is often very useful.

We can check for primality. Let us find out, how many squares plus 1 are primes.

```
>t=1:1000; length(nonzeros(isprime(t^2+1)))
```

```
112
```

The function nonzeros() works only for vectors. For matrices, there is mnonzeros().

```
>seed(2); A=random(3,4)
```

```
    0.765761      0.401188      0.406347      0.267829
     0.13673      0.390567      0.495975      0.952814
    0.548138      0.006085      0.444255      0.539246
```

It returns the indices of the elements, which are not zeros.

```
>k=mnonzeros(A<0.4) //indeks elemen2 A yang kurang dari 0,4
```

```
           1             4
           2             1
           2             2
           3             2
```

These indices can be used to set the elements to some value.

```
>mset(A,k,0) //mengganti elemen2 suatu matriks pada indeks tertentu
```

```
    0.765761      0.401188      0.406347             0
           0             0      0.495975      0.952814
    0.548138             0      0.444255      0.539246
```

The function mset() can also set the elements at the indices to the entries of some other matrix.

```
>mset(A,k,-random(size(A)))
```

```
         0.765761      0.401188      0.406347     -0.126917
        -0.122404     -0.691673      0.495975      0.952814
         0.548138     -0.483902      0.444255      0.539246
```

And it is possible to get the elements in a vector.

```
>mget(A,k)
```

```
    [0.267829,  0.13673,  0.390567,  0.006085]
```

Another useful function is extrema, which returns the minimal and maximal values in each row of the matrix and their positions.

```
>ex=extrema(A)
```

```
         0.267829             4      0.765761             1
          0.13673             1      0.952814             4
         0.006085             2      0.548138             1
```

We can use this to extract the maximal values in each row.

```
>ex[,3]'
```

```
    [0.765761,  0.952814,  0.548138]
```

This, of course, is the same as the function max().

```
>max(A)'
```

```
    [0.765761,  0.952814,  0.548138]
```

But with mget(), we can extract the indices and use this information to extract the elements at the same positions from another matrix.

```
>j=(1:rows(A))'|ex[,4], mget(-A,j)
```

```
                1             1
                2             4
                3             1
    [-0.765761,  -0.952814,  -0.548138]
```

Other Matrix Functions (Building Matrix)

To build a matrix, we can stack one matrix on top of another. If both do not have the same number of columns, the shorter one will be filled with 0.

```
>v=1:3; v_v
```

```
                1              2              3
                1              2              3
```

Likewise, we can attach a matrix to another side by side, if both have the same number of rows.

```
>A=random(3,4); A|v'
```

```
         0.032444      0.0534171       0.595713       0.564454              1
          0.83916       0.175552       0.396988        0.83514              2
        0.0257573       0.658585       0.629832       0.770895              3
```

If they do not have the same number of rows the shorter matrix is filled with 0.

There is an exception to this rule. A real number attached to a matrix will be used as a column filled with that real number.

```
>A|1
```

```
         0.032444      0.0534171       0.595713       0.564454              1
          0.83916       0.175552       0.396988        0.83514              1
        0.0257573       0.658585       0.629832       0.770895              1
```

It is possible to make a matrix of row and column vectors.

```
>[v;v]
```

```
                1              2              3
                1              2              3
```

```
>[v',v']
```

```
                1              1
                2              2
                3              3
```

The main purpose of this is to interpret a vector of expressions for column vectors.

```
>"[x,x^2]"(v')
```

```
                1              1
                2              4
                3              9
```

To get the size of A, we can use the following functions.

```
>C=zeros(2,4); rows(C), cols(C), size(C), length(C)
```

```
2
4
[2,  4]
4
```

For vectors, there is length().

```
>length(2:10)
```

```
9
```

There are many other functions, which generate matrices.

```
>ones(2,2)
```

```
              1             1
              1             1
```

This can also be used with one parameter. To get a vector with another number than 1, use the following.

```
>ones(5)*6
```

```
[6,  6,  6,  6,  6]
```

Also a matrix of random numbers can be generated with random (uniform distribution) or normal (Gauß distribution).

```
>random(2,2)
```

```
        0.66566      0.831835
          0.977      0.544258
```

Here is another useful function, which restructures the elements of a matrix into another matrix.

```
>redim(1:9,3,3) // menyusun elemen2 1, 2, 3, ..., 9 ke bentuk matriks 3x3
```

```
              1             2             3
              4             5             6
              7             8             9
```

With the following function, we can use this and the dup function to write a rep() function, which repeats a vector n times.

```
>function rep(v,n) := redim(dup(v,n),1,n*cols(v))
```

Let us test.

```
>rep(1:3,5)
```

```
[1,  2,  3,  1,  2,  3,  1,  2,  3,  1,  2,  3,  1,  2,  3]
```

The function multdup() duplicates elements of a vector.

```
>multdup(1:3,5), multdup(1:3,[2,3,2])
```

```
[1,  1,  1,  1,  1,  2,  2,  2,  2,  2,  3,  3,  3,  3,  3]
[1,  1,  2,  2,  2,  3,  3]
```

The functions flipx() and flipy() revert the order of the rows or columns of a matrix. I.e., the function flipx() flips horizontally.

```
>flipx(1:5) //membalik elemen2 vektor baris
```

```
[5,  4,  3,  2,  1]
```

For rotations, Euler has rotleft() and rotright().

```
>rotleft(1:5) // memutar elemen2 vektor baris
```

```
[2,  3,  4,  5,  1]
```

A special function is drop(v,i), which removes the elements with the indices in i from the vector v.

```
>drop(10:20,3)
```

```
[10,  11,  13,  14,  15,  16,  17,  18,  19,  20]
```

Note that the vector i in drop(v,i) refers to indices of elements in v, not the values of the elements. If you want to remove elements, you need to find the elements first. The function indexof(v,x) can be used to find elements x in a sorted vector v.

```
>v=primes(50), i=indexof(v,10:20), drop(v,i)
```

```
[2,  3,  5,  7,  11,  13,  17,  19,  23,  29,  31,  37,  41,  43,  47]
[0,  5,  0,  6,  0,  0,  0,  7,  0,  8,  0]
[2,  3,  5,  7,  23,  29,  31,  37,  41,  43,  47]
```

As you see, it does not harm to include indices out of range (like 0), double indices, or unsorted indices.

```
>drop(1:10,shuffle([0,0,5,5,7,12,12]))
```

```
[1,  2,  3,  4,  6,  8,  9,  10]
```

There are some special functions to set diagonals or to generate a diagonal matrix.

We start with the identity matrix.

```
>A=id(5) // matriks identitas 5x5
```

```
                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
```

Then we set the lower diagonal (-1) to 1:4.

```
>setdiag(A,-1,1:4) //mengganti diagonal di bawah diagonal utama
```

```
                1             0             0             0             0
                1             1             0             0             0
                0             2             1             0             0
                0             0             3             1             0
                0             0             0             4             1
```

Note that we did not change the matrix A. We get a new matrix as result of setdiag().

Here is a function, which returns a tri-diagonal matrix.

```
>function tridiag (n,a,b,c) := setdiag(setdiag(b*id(n),1,c),-1,a); ...
 tridiag(5,1,2,3)
```

```
                2             3             0             0             0
                1             2             3             0             0
                0             1             2             3             0
                0             0             1             2             3
                0             0             0             1             2
```

The diagonal of a matrix can also be extracted from the matrix. To demonstrate this, we restructure the vector 1:9 to a 3x3 matrix.

```
>A=redim(1:9,3,3)
```

```
                1             2             3
                4             5             6
                7             8             9
```

Now we can extract the diagonal.

```
>d=getdiag(A,0)
```

```
    [1,  5,  9]
```

E.g. We can divide the matrix by its diagonal. The matrix language takes care that the column vector d is applied to the matrix row by row.

```
>fraction A/d'
```

```
           1         2         3
         4/5         1       6/5
         7/9       8/9         1
```

Vectorization

Almost all functions in Euler work for matrix and vector input too, whenever this makes sense.

E.g., the sqrt() function computes the square root of all elements of the vector or matrix.

```
>sqrt(1:3)
```

```
[1,  1.41421,  1.73205]
```

So you can easily create a table of values. This is one way to plot a function (the alternative uses an expression).

```
>x=1:0.01:5; y=log(x)/x^2; // terlalu panjang untuk ditampikan
```

With this and the colon operator a:delta:b, vectors of values of functions can be generated easily.

In the following example, we generate a vector of values t[i] with spacing 0.1 from -1 to 1. Then we generate a vector of values of the function

```
>t=-1:0.1:1; s=t^3-t
```

```
[0,  0.171,  0.288,  0.357,  0.384,  0.375,  0.336,  0.273,  0.192,
0.099,  0,  -0.099,  -0.192,  -0.273,  -0.336,  -0.375,  -0.384,
-0.357,  -0.288,  -0.171,  0]
```

EMT expands operators for scalars, vectors, and matrices in the obvious way.

E.g., a column vector times a row vector expands to matrix, if an an operator is applied. In the following, v' is the transposed vector (a column vector).

```
>shortest (1:5)*(1:5)'
```

```
      1       2       3       4       5
      2       4       6       8      10
      3       6       9      12      15
      4       8      12      16      20
      5      10      15      20      25
```

Note, that this is quite different from the matrix product. The matrix product is denoted with a dot "." in EMT.

```
>(1:5).(1:5)'
```

```
55
```

By default, row vectors are printed in a compact format.

```
>[1,2,3,4]
```

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

For matrices the special operator . denotes matrix multiplication, and A' denotes transposing. A 1x1 matrix can be used just like a real number.

```
>v:=[1,2]; v.v', %^2
```

```
 5
 25
```

To transpose a matrix we use the apostrophe.

```
>v=1:4; v'
```

```
                  1
                  2
                  3
                  4
```

So we can compute matrix A times vector b.

```
>A=[1,2,3,4;5,6,7,8]; A.v'
```

```
                 30
                 70
```

Note that v is still a row vector. So v'.v is different from v.v'.

```
>v'.v
```

```
                  1             2             3             4
                  2             4             6             8
                  3             6             9            12
                  4             8            12            16
```

v.v' computes the norm of v squared for row vectors v. The result is a 1x1 vector, which works just like a real number.

```
>v.v'
```

```
 30
```

There is also the function norm (along with many other function of Linear Algebra).

```
>norm(v)^2
```

```
 30
```

Operators and functions obey the matrix language of Euler.

Here is a summary of the rules.

- A function applied to a vector or matrix is applied to each element.

- An operator operating on two matrices of same size is applied pairwise to the elements of the matrices.
- If the two matrices have different dimensions, both are expanded in a sensible way, so that they have the same size.

E.g., a scalar value times a vector multiplies the value with each element of the vector. Or a matrix times a vector (with *, not .) expands the vector to the size of the matrix by duplicating it.

The following is a simple case with the operator ^.

```
>[1,2,3]^2
```

```
[1,  4,  9]
```

Here is a more complicated case. A row vector times a column vector expands both by duplicating.

```
>v:=[1,2,3]; v*v'
```

```
              1             2             3
              2             4             6
              3             6             9
```

Note that the scalar product uses the matrix product, not the *!

```
>v.v'
```

```
14
```

There are numerous functions for matrices. We give a short list. You should to consult the documentation for more information on these commands.

```
sum,prod computes the sum and products of the rows
cumsum,cumprod does the same cumulatively
computes the extremal values of each row
extrema returns a vector with the extremal information
diag(A,i) returns the i-th diagonal
setdiag(A,i,v) sets the i-th diagonal
id(n) the identity matrix
det(A) the determinant
charpoly(A) the characteristic polynomial
eigenvalues(A) the eigenvalues
```

```
>v*v, sum(v*v), cumsum(v*v)
```

```
[1,  4,  9]
14
[1,  5,  14]
```

The : operator generates an equally spaces row vector, optionally with a step size.

```
>1:4, 1:2:10
```

```
[1,  2,  3,  4]
[1,  3,  5,  7,  9]
```

To concatenate matrices and vectors there are the operators "|" and "_".

```
>[1,2,3]|[4,5], [1,2,3]_1
```

```
[1,  2,  3,  4,  5]
           1              2              3
           1              1              1
```

The elements of a matrix are referred with "A[i,j]".

```
>A:=[1,2,3;4,5,6;7,8,9]; A[2,3]
```

```
6
```

For row or column vectors, v[i] is the i-th element of the vector. For matrices, this returns the complete i-th row of the matrix.

```
>v:=[2,4,6,8]; v[3], A[3]
```

```
6
[7,  8,  9]
```

The indices can also be row vectors of indices. : denotes all indices.

```
>v[1:2], A[:,2]
```

```
[2,  4]
           2
           5
           8
```

A short form for : is omitting the index completely.

```
>A[,2:3]
```

```
           2              3
           5              6
           8              9
```

For purposes of vectorization, the elements of a matrix can be accessed as if they were vectors.

```
>A{4}
```

```
4
```

A matrix can also be flattened, using the redim() function. This is implemented in the function flatten().

```
>redim(A,1,prod(size(A))), flatten(A)
```

```
 [1,  2,  3,  4,  5,  6,  7,  8,  9]
 [1,  2,  3,  4,  5,  6,  7,  8,  9]
```

To use matrices for tables, let us reset to the default format, and compute a table of sine and cosine values. Note that angles are in radians by default.

```
>defformat; w=0°:45°:360°; w=w'; deg(w)
```

```
              0 
             45 
             90 
            135 
            180 
            225 
            270 
            315 
            360 
```

Now we append columns to a matrix.

```
>M = deg(w)|w|cos(w)|sin(w)
```

```
              0             0             1             0 
             45      0.785398      0.707107      0.707107 
             90        1.5708             0             1 
            135       2.35619     -0.707107      0.707107 
            180       3.14159            -1             0 
            225       3.92699     -0.707107     -0.707107 
            270       4.71239             0            -1 
            315       5.49779      0.707107     -0.707107 
            360       6.28319             1             0 
```

Using the matrix language, we can generate several tables of several functions at once.

In the following example, we compute t[j]^i for i from 1 to n. We get a matrix, where each row is a table of t^i for one i. I.e., the matrix has the elements

A function which does not work for vector input should be "vectorized". This can be achieved by the "map" keyword in the function definition. Then the function will be evaluated for each element of a vector parameter.

The numerical integration integrate() works only for scalar interval bounds. So we need to vectorize it.

```
>function map f(x) := integrate("x^x",1,x)
```

The "map" keyword vectorizes the function. The function will now work
for vectors of numbers.

```
>f([1:5])
```

```
 [0,  2.05045,  13.7251,  113.336,  1241.03]
```

Sub-Matrices and Matrix-Elements

To access a matrix element, use the bracket notation.

```
>A=[1,2,3;4,5,6;7,8,9], A[2,2]
```

```
                  1               2               3
                  4               5               6
                  7               8               9
    5
```

We can access a complete line of a matrix.

```
>A[2]
```

```
    [4,   5,   6]
```

In case of row or column vectors, this returns an element of the vector.

```
>v=1:3; v[2]
```

```
    2
```

To make sure, you get the first row for a 1xn and a mxn matrix, specify all columns using an empty second index.

```
>A[2,]
```

```
    [4,   5,   6]
```

If the index is a vector of indices, Euler will return the corresponding rows of the matrix.

Here we want the first and second row of A.

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

```
                  1               2               3
                  4               5               6
```

We can even reorder A using vectors of indices. To be precise, we do not change A here, but compute a reordered version of A.

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

```
                  7               8               9
                  4               5               6
                  1               2               3
```

The index trick works with columns too.

This example selects all rows of A and the second and third column.

```
>A[1:3,2:3]
```

```
                 2              3
                 5              6
                 8              9
```

For abbreviation ":" denotes all row or column indices.

```
>A[:,3]
```

```
                 3
                 6
                 9
```

Alternatively, leave the first index empty.

```
>A[,2:3]
```

```
                 2              3
                 5              6
                 8              9
```

We can also get the last line of A.

```
>A[-1]
```

```
 [7,  8,  9]
```

Now let us change elements of A by assigning a submatrix of A to some value. This does in fact change the stored matrix A.

```
>A[1,1]=4
```

```
                 4              2              3
                 4              5              6
                 7              8              9
```

We can also assign a value to a row of A.

```
>A[1]=[-1,-1,-1]
```

```
                -1             -1             -1
                 4              5              6
                 7              8              9
```

We can even assign to a sub-matrix if it has the proper size.

```
>A[1:2,1:2]=[5,6;7,8]
```

```
                  5            6           -1
                  7            8            6
                  7            8            9
```

Moreover, some shortcuts are allowed.

```
>A[1:2,1:2]=0
```

```
                  0            0           -1
                  0            0            6
                  7            8            9
```

A warning: Indices out of bounds return empty matrices, or an error message, depending on a system setting. The default is an error message. Remember, however, that negative indices may be used to access the elements of a matrix counting from the end.

```
>A[4]
```

```
 Row index 4 out of bounds!
 Error in:
 A[4] ...
     ^
```

Sorting and Shuffling

The function sort() sorts a row vector.

```
>sort([5,6,4,8,1,9])
```

```
 [1,  4,  5,  6,  8,  9]
```

It is often necessary to know the indices of the sorted vector in the original vector. This can be used to reorder another vector in the same way.

Let us shuffle a vector.

```
>v=shuffle(1:10)
```

```
 [4,  5,  10,  6,  8,  9,  1,  7,  2,  3]
```

The indices contain the proper order of v.

```
>{vs,ind}=sort(v); v[ind]
```

```
 [1,  2,  3,  4,  5,  6,  7,  8,  9,  10]
```

This works for string vectors too.

```
>s=["a","d","e","a","aa","e"]
```

```
a
d
e
a
aa
e
```

```
>{ss,ind}=sort(s); ss
```

```
a
a
aa
d
e
e
```

As you see, the position of double entries is somewhat random.

```
>ind
```

```
[4,  1,  5,  2,  6,  3]
```

The function unique returns a sorted list of unique elements of a vector.

```
>intrandom(1,10,10), unique(%)
```

```
[4,  4,  9,  2,  6,  5,  10,  6,  5,  1]
[1,  2,  4,  5,  6,  9,  10]
```

This works for string vectors too.

```
>unique(s)
```

```
a
aa
d
e
```

Linear Algebra

EMT has lots of functions to solve linear systems, sparse systems, or regression problems.

For linear systems Ax=b, you can use the Gauss algorithm, the inverse matrix or a linear fit. The operator A\b uses a version of the Gauss algorithm.

```
>A=[1,2;3,4]; b=[5;6]; A\b
```

```
             -4
            4.5
```

For another example, we generate a 200x200 matrix and the sum of its rows. Then we solve Ax=b using the inverse matrix. We measure the error as the maximal deviation of all elements from 1, which of course is the correct solution.

```
>A=normal(200,200); b=sum(A); longest totalmax(abs(inv(A).b-1))
```

```
    8.790745908981989e-13
```

If the system does not have a solution, a linear fit minimizes the norm of the error Ax-b.

```
>A=[1,2,3;4,5,6;7,8,9]
```

```
                 1             2             3
                 4             5             6
                 7             8             9
```

The determinant of this matrix is 0.

```
>det(A)
```

```
  0
```

Symbolic Matrices

Maxima has symbolic matrices. Of course, Maxima can be used for such simple linear algebra problems. We can define the matrix for Euler and Maxima with &:=, and then use it in symbolic expressions. The usual [...] form to define matrices can be used in Euler to define symbolic matrices.

```
>A &= [a,1,1;1,a,1;1,1,a]; $A
>$&det(A), $&factor(%)
>$&invert(A) with a=0
>A &= [1,a;b,2]; $A
```

Like all symbolic variables, these matrices can be used in other symbolic expressions.

```
>$&det(A-x*ident(2)), $&solve(%,x)
```

The eigenvalues can also be computed automatically. The result is a vector with two vectors of eigenvalues and multiplicities.

```
>$&eigenvalues([a,1;1,a])
```

To extract a specific eigenvector needs careful indexing.

```
>$&eigenvectors([a,1;1,a]), &%[2][1][1]
```

```
                                    [1, - 1]
```

Symbolic matrices can be evaluated in Euler numerically just like other symbolic expressions.

```
>A(a=4,b=5)
```

```
                1              4
                5              2
```

In symbolic expressions, use with.

```
>$&A with [a=4,b=5]
```

Access to rows of symbolic matrices work just like with numerical matrices.

```
>$&A[1]
```

A symbolic expression can contain an assignment. And that changes the matrix A.

```
>&A[1,1]:=t+1; $&A
```

There are symbolic functions in Maxima to create vectors and matrices. For this, refer to the documentation of Maxima or to the tutorial about Maxima in EMT.

```
>v &= makelist(1/(i+j),i,1,3); $v
```

```
>B &:= [1,2;3,4]; $B, $&invert(B)
```

The result can be evaluated numerically in Euler. For more information about Maxima, see the introduction to Maxima.

```
>$&invert(B)()
```

```
               -2              1
              1.5           -0.5
```

Euler has also a powerful function xinv(), which makes a bigger effort and gets more exact results.

Note, that with &:= the matrix B has been defined as symbolic in symbolic expressions and as numerical in numerical expressions. So we can use it here.

```
>longest B.xinv(B)
```

```
                          1                       0
                          0                       1
```

E.g. the eigenvalues of A can be computed numerically.

```
>A=[1,2,3;4,5,6;7,8,9]; real(eigenvalues(A))
```

```
 [16.1168,  -1.11684,  0]
```

Or symbolically. See the tutorial about Maxima for details on this.

```
>$&eigenvalues(@A)
```

Numerical Values in symbolic Expressions

A symbolic expression is just a string containing an expression. If we want to define a value both for symbolic expressions and for numerical expressions, we must use "&:=".

```
>A &:= [1,pi;4,5]
```

```
                1       3.14159
                4             5
```

There is still a difference between the numerical and the symbolic form. When transferring the matrix to the symbolic form, fractional approximations for reals will be used.

```
>$&A
```

To avoid this, there is the function "mxmset(variable)".

```
>mxmset(A); $&A
```

Maxima can also compute with floating point numbers, and even with big floating numbers with 32 digits. The evaluation is much slower, however.

```
>$&bfloat(sqrt(2)), $&float(sqrt(2))
```

The precision of the big floating point numbers can be changed.

```
>&fpprec:=100; &bfloat(pi)
```

```
            3.14159265358979323846264338327950288419716939937510582097494\
    4592307816406286208998628034825342117068b0
```

A numerical variable can be used in any symbolic expressions using "@var".

Note that this is only necessary, if the variable has been defined with ":=" or "=" as a numerical variable.

```
>B:=[1,pi;3,4]; $&det(@B)
```

Demo - Interest Rates

Below, we use Euler Math Toolbox (EMT) for the calculation of interest rates. We do that numerically and symbolically to show you how Euler can be used to solve real life problems.

Assume you have a seed capital of 5000 (say in dollars).

```
>K=5000
```

```
    5000
```

Now we assume an interest rate of 3% per year. Let us add one simple rate and compute the result.

```
>K*1.03
```

```
 5150
```

Euler would understand the following syntax too.

```
>K+K*3%
```

```
 5150
```

But it is easier to use the factor

```
>q=1+3%, K*q
```

```
 1.03
 5150
```

For 10 years, we can simply multiply the factors and get the final value with compound interest rates.

```
>K*q^10
```

```
 6719.58189672
```

For our purposes, we can set the format to 2 digits after the decimal dot.

```
>format(12,2); K*q^10
```

```
     6719.58
```

Let us print that rounded to 2 digits in a complete sentence.

```
>"Starting from " + K + "$ you get " + round(K*q^10,2) + "$."
```

```
 Starting from 5000$ you get 6719.58$.
```

What if we want to know the intermediate results from year 1 to year 9? For this, Euler's matrix language is a big help. You do not have to write a loop, but simply enter

```
>K*q^(0:10)
```

```
 Real 1 x 11 matrix

     5000.00     5150.00     5304.50     5463.64     ...
```

How does this miracle work? First the expression 0:10 returns a vector of integers.

```
>short 0:10
```

```
[0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10]
```

Then all operators and functions in Euler can be applied to vectors element for element. So

```
>short q^(0:10)
```

```
[1,  1.03,  1.0609,  1.0927,  1.1255,  1.1593,  1.1941,  1.2299,
1.2668,  1.3048,  1.3439]
```

is a vector of factors q^0 to q^10. This is multiplied by K, and we get the vector of values.

```
>VK=K*q^(0:10);
```

Of course, the realistic way to compute these interest rates would be to round to the nearest cent after each year. Let us add a function for this.

```
>function oneyear (K) := round(K*q,2)
```

Let us compare the two results, with and without rounding.

```
>longest oneyear(1234.57), longest 1234.57*q
```

```
                  1271.61
                1271.6071
```

Now there is no simple formula for the n-th year, and we must loop over the years. Euler provides many solutions for this.

The easiest way is the function iterate, which iterates a given function a number of times.

```
>VKr=iterate("oneyear",5000,10)
```

```
Real 1 x 11 matrix

    5000.00     5150.00     5304.50     5463.64     ...
```

We can print that in a friendly way, using our format with fixed decimal places.

```
>VKr'
```

```
    5000.00
    5150.00
    5304.50
    5463.64
    5627.55
    5796.38
    5970.27
    6149.38
    6333.86
```

```
        6523.88
        6719.60
```

To get a specific element of the vector, we use indices in square brackets.

```
>VKr[2], VKr[1:3]
```

```
        5150.00
        5000.00     5150.00     5304.50
```

Surprisingly, we can also use a vector of indices. Remember that 1:3 produced the vector [1,2,3].

Let us compare the last element of the rounded values with the full values.

```
>VKr[-1], VK[-1]
```

```
        6719.60
        6719.58
```

The difference is very small.

Solving Equations

Now we take a more advanced function, which adds a certain rate of money each year.

```
>function onepay (K) := K*q+R
```

We do not have to specify q or R for the definition of the function. Only if we run the command, we have to define these values. We select R=200.

```
>R=200; iterate("onepay",5000,10)
```

```
    Real 1 x 11 matrix

        5000.00     5350.00     5710.50     6081.82     ...
```

What if we remove the same amount each year?

```
>R=-200; iterate("onepay",5000,10)
```

```
    Real 1 x 11 matrix

        5000.00     4950.00     4898.50     4845.45     ...
```

We see that the money decreases. Obviously, if we get only 150 of interest in the first year, but remove 200, we lose money each year.

How can we determine the number of years the money will last? We would have to write a loop for this. The easiest way is to iterate long enough.

```
>VKR=iterate("onepay",5000,50)
```

```
   Real 1 x 51 matrix

       5000.00     4950.00     4898.50     4845.45     ...
```

Using the matrix language, we can determine the first negative value in the following way.

```
>min(nonzeros(VKR<0))
```

```
        48.00
```

The reason for this is that nonzeros(VKR<0) returns a vector of indices i, where VKR[i]<0, and min computes the minimal index.

Since vectors always start with index 1, the answer is 47 years.

The function iterate() has one more trick. It can take an end condition as an argument. Then it will return the value and the number of iterations.

```
>{x,n}=iterate("onepay",5000,till="x<0"); x, n,
```

```
       -19.83
        47.00
```

Let us try to answer a more ambiguous question. Assume we know that the value is 0 after 50 years. What would be the interest rate?

This is a question, which can only be answered numerically. Below, we will derive the necessary formulas. Then you will see that there is no easy formula for the interest rate. But for now, we aim for a numerical solution.

The first step is to define a function which does the iteration n times. We add all parameters to this function.

```
>function f(K,R,P,n) := iterate("x*(1+P/100)+R",K,n;P,R)[-1]
```

The iteration is just as above

But we do longer use the global value of R in our expression. Functions like iterate() have a special trick in Euler. You can pass the values of variables in the expression as semicolon parameters. In this case P and R.

Moreover, we are only interested in the last value. So we take the index [-1].

Let us try a test.

```
>f(5000,-200,3,47)
```

```
       -19.83
```

Now we can solve our problem.

```
>solve("f(5000,-200,x,50)",3)
```

```
         3.15
```

The solve routine solves expression=0 for the variable x. The answer is 3.15% per year. We take the start value of 3% for the algorithm. The solve() function always needs a start value.

We can use the same function to solve the following question: How much can we remove per year so that the seed capital is exhausted after 20 years assuming an interest rate of 3% per year.

```
>solve("f(5000,x,3,20)",-200)
```

```
       -336.08
```

Note that you cannot solve for the number of years, since our function assumes n to be an integer value.

Symbolic Solutions to the Interest Rate Problem

We can use the symbolic part of Euler to study the problem. First we define our function onepay() symbolically.

```
>function op(K) &= K*q+R; $&op(K)
```

We can now iterate this.

```
>$&op(op(op(op(K)))), $&expand(%)
```

We see a pattern. After n periods we have

The formula is the formula for the geometric sum, which is known to Maxima.

```
>&sum(q^k,k,0,n-1); $& % = ev(%,simpsum)
```

This is a bit tricky. The sum is evaluated with the flag "simpsum" to reduce it to the quotient.

Let us make a function for this.

```
>function fs(K,R,P,n) &= (1+P/100)^n*K + ((1+P/100)^n-1)/(P/100)*R; $&fs(K,R,P,n)
```

The function does the same as our function f before. But it is more effective.

```
>longest f(5000,-200,3,47), longest fs(5000,-200,3,47)
```

```
        -19.82504734650985
        -19.82504734652684
```

We can now use it to ask for the time n. When is our capital exhausted? Our initial guess is 30 years.

```
>solve("fs(5000,-330,3,x)",30)
```

```
         20.51
```

This answer says that it will be negative after 21 years.

We can also use the symbolic side of Euler to compute formulas for the payments.

Assume we get a loan of K, and pay n payments of R (starting after the first year) leaving a residual debt of Kn (at the time of the last payment). The formula for this is clearly

```
>equ &= fs(K,R,P,n)=Kn; $&equ
```

Usually this formula is given in terms of

```
>equ &= (equ with P=100*i); $&equ
```

We can solve for the rate R symbolically.

```
>$&solve(equ,R)
```

As you can see from the formula, this function returns a floating point error for i=0. Euler plots it nevertheless.

Of course, we have the following limit.

```
>$&limit(R(5000,0,x,10),x,0)
```

Clearly, without interest we have to pay back 10 rates of 500.

The equation can also be solved for n. It looks nicer, if we apply some simplification to it.

```
>fn &= solve(equ,n) | ratsimp; $&fn
```