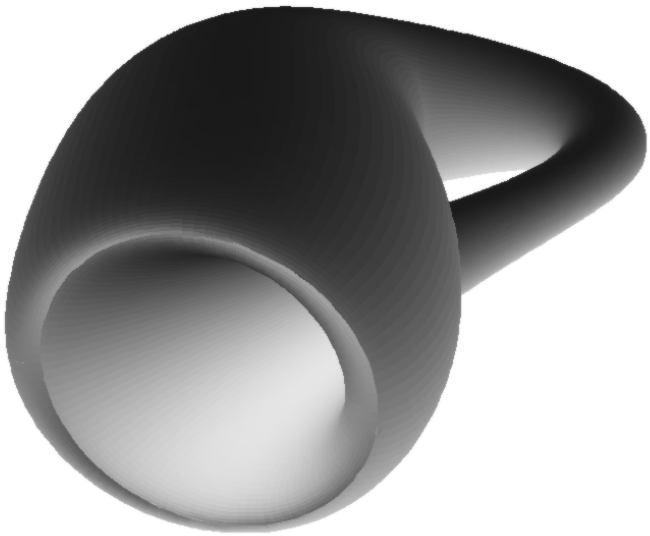
# Euler Math Toolbox
# An Introduction

René Grothmann

Juli 2021



# Contents

# Preface

This is an introduction to Euler Math Toolbox (EMT). The aim of this text is to provide a good start into EMT. After reading this text, you should be able to find your way through EMT with the help of the documentation and the reference.

If you want to see some examples first start with the first chapter. The second chapter provides a thorough introduction to the user interface and the main features of EMT. Subsequent chapter deal with details about various aspects of this mighty program, such as planar or spacial plots, numerical analysis, symbolic mathematics, complex numbers, regression analysis, sparse matrices, statistics, optimization and the programming language of EMT.

EMT has been written for math at the University level, designed by a mathematician with the frequent need for numerical and symbolic computations and graphical representations of the results. An additional aim of EMT was to be useful on the school level. The EMT language, combined with the Algebra system Maxima are ideal tools for this purpose. Tasks of all levels can be performed with EMT and its programming language. However, EMT is not a simple click and run system presenting all tools in iconic form. So this introduction should be a big help.

EMT does not compete with Matlab. Both system started at about the same time in the 80ies. But EMT was always an open system incorporating other open systems (Maxima, Python, C, Povray) seamlessly under a common hood, while Matlab strove to become the industry standard for numerical computations. The author believes that advanced research should be done with open numerical libraries available freely and for many programming languages. A gigantic closed system like Matlab is not a good solution and hinders development.

For education in numerical analysis, EMT or Matlab are useful tools, but the knowledge of a basic programming language is indispensable for a professional career. A practical course in applied mathematics should also contain an introduction into numerical programming with a real world programming language. EMT can be useful for demonstrations and tests, however.

Currently, the advanced version of EMT is only available for Windows. On Linux or OSX, Wine can be used to run EMT with some restrictions. Plans for a Java interface to EMT will depend on user demand and community help.

I wish to thank a lot of friends, users and developers, first of all my university for giving me the opportunity to work on this project. Then I thank Eric Boucharé, who ported EMT for Linux. Currently, I do not have the time to update his version to the level of the Windows version. I also thank H.D. Gellißen for his German handbook. Many users have contributed to Euler with programs, notebooks and bug hints, especially Alain Busser, Radovan Omorjan and Horst Vogel. I also thank the developers of Maxima for making their system available for EMT.

Have a good and successful time!

René Grothmann
Eichstätt 2019

The EMT web site is at http://www.euler-math-toolbox.de

# Chapter 1

# First Examples

## 1.1 Get Started

You should start with the tutorials that are installed with EMT. They are available in HTML form to be read in the browser, and they are contained in the installation as notebooks you can load into the program and work with. If you want to save your changes you need to save a copy into a directory with write access, of course. EMT creates a directory for this purpose in your documents folder.

Currently, the following introductory notebooks are available.

- Overview and Introduction
  This is a first welcome and an overview of Euler which shows the potential of the program. Have a look into this tutorial to start with EMT.
- A Crash Course in Euler
  This tutorial contains a quick introduction for impatient users or for anyone who is already familiar with similar programs.
- Interest Rates
  A Demo.
- Monte-Carlo Simulation
  A Demo.
- Plots in Euler
  A Demo.

- The Syntax of Euler
  Contains everything you need to know about the syntax of commands and expressions, including the symbolic expressions.

- The Matrix Language
  The matrix language is the basic tool for efficient computations in EMT. For short and quick ways to solve problems you should be familiar with this way to handle numerical computations.

- Complex Numbers
  Besides real numbers, EMT can handle complex numbers, find zeros of complex polynomials, and solve complex equations.

- Intervals
  Interval arithmetic, together with the exact scalar product, is the main tool to get guaranteed inclusions for solutions of linear and non-linear equations or differential equations.

- Maxima
  This tutorial introduces symbolic expressions and the interface to Maxima. EMT seamlessly incorporates Maxima using a special syntax for symbolic expressions and functions.

- More about Maxima
  Some more details about Maxima.

- 2D Plots
  All about plots of one variable, curves, or implicit plots in the plane.

- 3D Plots
  All about plots of two variables, surfaces, or implicit plots in the 3D space.

- Linear Algebra
  Shows how to solve linear systems, compute least square fits, eigenvalues or singular values, using numerical and symbolic arithmetic.

- Numerical Analysis
  EMT can solve non-linear equations and systems, or integrals.

- Differential Equations
  Shows numerical and symbolic solutions of differential equations

- Input and Output
  Read and write from files or from the Internet.

- Statistics
  Statistics is a perfect application for EMT. This tutorial introduces statistical distributions, functions, and special plots for statistics. Moreover, it shows how to do Monte-Carlo simulations in EMT.
- Optimization
  A tutorial about linear, non-linear, and integer optimization. EMT contains the efficient LPSOLVE package for linear optimization.
- Large Systems
  EMT can handle large and sparse systems. The tutorial contains examples connected to graph theory.
- Fast Fourier Transform
  This explains FFT and how to produce or analyze sound in EMT.
- Programming Language
  Programming Euler
- Compiled Code
  A tutorial about C code in Euler. Euler comes with the TinyC compiler.
- Python in Euler
  Python can be used as a scripting language in EMT. Functions in Python can be called from EMT and vice versa. This includes might packages like MatPlotlib.
- Povray and Euler
  Tutorial on using Povray from Euler with many examples.
- Media Files
  A collection of examples generating, loading and saving sounds and images.
- Geometry Explains numerical and symbolic functions for geometry in EMT.

In this first chapter, you will see some simple examples to introduce the main features of EMT and Maxima. I assume for now that you are able to handle the interface of the program. The main documentation contains a page introducing the interface in all details.

For now, it suffices to say that you can enter commands in the text window at the prompt and run these commands with the enter key. You can edit your commands at any time and run them again. The cursor does not have to be at the end of the command line for this. If you want to execute all commands in a section once more, press shift-return. Copy and paste will also work as usual.

If you need help or more information on any of the commands used in this introduction, you can either look into the full HTML reference of EMT and Maxima, or try the help system (available with F1 or a double click on any command in EMT). There you can enter any command in the search line, or you can double click on a command in the text window or the help text to search for this command.

## 1.2 Symbolic versus Numerical Computations

With EMT and Maxima, you hold two tools in your hand, one for fast numerical, and the other for exact symbolic arithmetic. Sometimes it is better to use one tool, and sometimes the other. Both tools can work together to give you even better answers. By default, the numerical part of EMT is the main tool and Maxima commands need a special trigger in the command line. But it is also possible, to use EMT as an interface for Maxima. Moreover, there is a very easy syntax to use symbolic expressions and functions in EMT seamlessly.

Assume you want to do some basic computations with fractions. You get different results in EMT and Maxima. With EMT you get a floating point result with about 16 digits accuracy (the IEEE standard), printed with 12 digits by default.

```
>1+1/2+1/3+1/4  // compute the fraction with Euler
   2.08333333333
```

Maxima computes the result as a fraction in its "infinite" integer arithmetic.

```
>& 1+1/2+1/3+1/4  // compute the fraction with Maxima

                                        25
                                        --
                                        12
```

The Maxima command starts with & optionally followed by a blank. The character & starts a symbolic expression. It is possible to call Maxima directly. But for this introduction, we use only symbolic expressions which are the right way to integrate symbolic computations into EMT seamlessly.

You can also print decimal floats as fractions in EMT, and convert fractions to float in Maxima.

```
>fracprint(1+1/2+1/3+1/4)  // print as fraction in Euler
   25/12
>& float(1+1/2+1/3+1/4)  // convert to float in Maxima

                                  2.083333333333334
```

Why then two programs? Because Maxima is a lot slower than EMT, when it comes to elaborate numerical stuff. Moreover, EMT has some numerical algorithms Maxima does not have. On the other side, Maxima does a lot of things EMT cannot do. The two programs complement each other very well.

> *EMT consists of a numerical system and a symbolic system. The latter is handled by Maxima in the background. Symbolic computations, if feasible, are very slow, but accurate.*

Let us try some more computations in EMT or Maxima. We compute

$$\sum_{k=1}^{1000} \frac{1}{k}$$

in three ways. First in EMT, then as a fraction in Maxima, and finally as a `float` in Maxima. The fractional result is not very useful. You might even have problems to spot the "/".

```
>sum(1/(1:1000)) // fast numerical result in Euler
         7.48547086055
>& sum(1/k,k,1,1000) // fractional result in Maxima

         53362913282294785045591045624042980409652472280384260097101349248456268\
   89497101757506097901985035691409088731550468098378442172117885009464302344326\
   66022502100278425632852081405544941210442510142672770294774712708917963967779\
   10453224692426866468888281582071984897105110796873249319155529397017508931564\
   19976085734473014183284011724412280649074307703736683170055800293659235088589\
   60235285852808160759574737836655413175508131522517/71288652746650930531663841\
   57142729206683588618858930404520019911543240875811114994764441519138715869117\
   78170195752565129802640676210092514658710043051310726862681432001966099748627\
   59371883437050154344525237397452989631456749821282369562328237940110688092623\
   77088619795407912477545580493264757378299233527517967352480424636380511370343\
   12147817468508784534856780218880753732499219956720569320290993908916874876726\
   7950931603520000

>&sum(1.0/k,k,1,1000) // float sum in Maxima

                                  7.485470860550343
```

In the last command, Maxima treated `1.0` as a float, and will do the whole computation in floating point.

Maxima can often compute exact results, where EMT only gives a floating point answer.

```
>sin(45°)
        0.7071067811865
>&sin(pi/4)
```

$$\frac{1}{\text{sqrt}(2)}$$

The answers are equivalent, but there is no easy way to get the Maxima result using the numerical kernel of EMT. Anyways, the symbolic result is useful only for pure mathematics.

Another example is the following infinite harmonic series, where EMT can only compute an approximation, but Maxima knows the exact value.

```
>&sum(1/k^2,k,1,inf)|simpsum, &float(%)
```

$$\frac{\text{pi}^2}{6}$$

```
                                1.644934066848226
```

```
>longest sum(1/(1:1000000)^2)
          1.64493306684877
```

The syntax of the first command line involves the flag `simpsum` to make Maxima evaluate the sum. The second symbolic command in this line uses the `%` sign to refer to the previous result. The second command contains the operator `longest` which prints the result to 16 digits.

There are many ways, EMT and Maxima can interact. The most easy way is by symbolic expressions. Moreover, Maxima can be used to write efficient functions in EMT whenever a symbolic formula is involved. More on this in the following sections.

## 1.3 Numerical Examples

We want to present a few examples for numerical computations with EMT. So we won't use Maxima for these examples. They are all done by the numerical kernel of EMT.

For a first example, we observe a ball at an apparent diameter of $\alpha = 12.5^\circ$ and we know that its true diameter is $d = 100$ meter. How far away is the center of the ball? The formula for that is of course

$$r \tan\left(\frac{\alpha}{2}\right) = \frac{d}{2}.$$

Thus we get the following answer.

```
>a=12.5°; d=100; (d/2)/tan(a/2)
  456.54674095
```

We have used two variables here, and several command in one line. More on that later. The answer is printed with 12 digits which is not a sensible way to print this result since the input data are not that exact. We can change the output format at any time or we can use the function `print` which has parameters to limit the number of digits after the comma.

```
>a=12.55°; d=99.5m; print((d/2)/tan(a/2),1)
      452.4
>a=12.45°; d=100.5m; print((d/2)/tan(a/2),1)
      460.7
```

To get a feeling for the accuracy, we have computed two extreme values under the assumption that the input is correct only to the given digits.

By the way, EMT has an interval arithmetic to handle this situation. Interval arithmetic is not widely known. But it is sometimes very nice to have.

```
>a=12.5°±0.05°; d=100±0.5; (d/2)/tan(a/2)
 ~452.4,460.7~
```

If things get more involved we might want to define a function. The easiest functions in EMT are one-line functions. We implement the function

$$l_c(v) = \sqrt{1 - \frac{v^2}{c^2}}$$

were $c$ is the speed of light. This constant is defined in EMT as a unit. Units always end with a dollar character.

```
>cl = cLight$
  299792458
>function lc(v) := sqrt(1-v^2/cl^2)
>lc(cl/4)
 0.968245836552
```

That is the relativistic contraction at one quarter of light speed. If we have units we want to convert from and to these units in an easy way. EMT uses the operator -> for this. Moreover, we demonstrate that units can simply be appended to numbers and the result will be converted into IS (international standard).

Another feature of EMT output is the special space character. It can split numbers into groups of digits. You can enter this space with Ctrl-Space. The normal space will not work.

```
>lc(100 000 km/h) // use Ctrl-Space for the spaces!
 0.999999995707
>cl -> " km/h"
 1079252848.8 km/h
>print(cl->km/h,0,sep=" ",unit=" km/h")
 1 079 252 849 km/h
```

The laster number is the speed of light in kilometer per second. EMT has also non-metric units like miles, feet and many more.

The matrix language of EMT allows to apply functions and operators to vectors element by element. More details will be given in later chapters. For now, let us evaluate

$$f(x) = \sum_{n=1}^{10} \frac{x^n}{n^2}.$$

We need a vector of numbers $1, \ldots, n$ which can easily be generated in EMT, and we need the function sum which sums a vector.

```
>n=1:10
 [1,  2,  3,  4,  5,  6,  7,  8,  9,  10]
>n^2
 [1,  4,  9,  16,  25,  36,  49,  64,  81,  100]
>0.5^n
 [0.5,  0.25,  0.125,  0.0625,  0.03125,  0.015625,  0.0078125,
 0.00390625,  0.00195313,  0.000976563]
>sum(0.5^n/n^2)
 0.582233518878
```

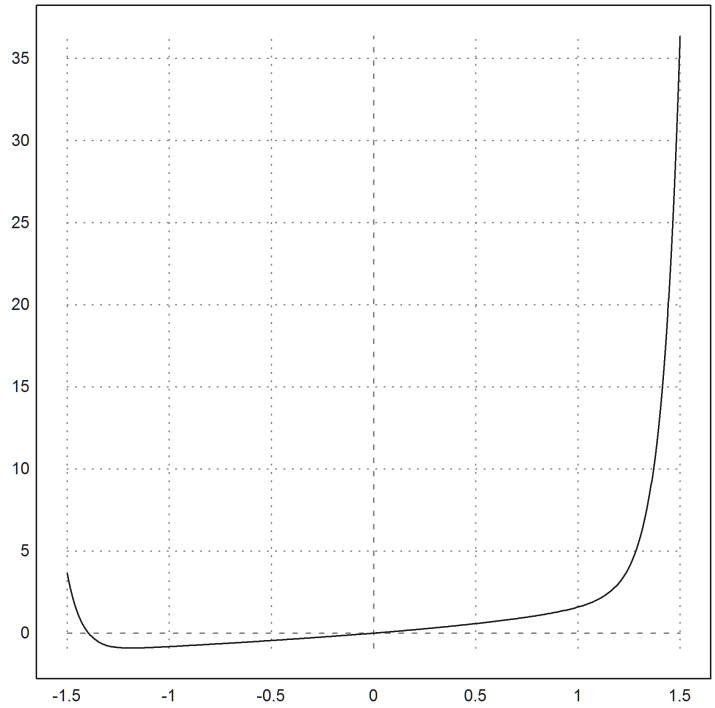



Figure 1.1: Plot of $f(x)$ in $[-1.5, 1.5]$

To make that a function that does work for vector input, but is defined for scalar input, we define a vectorized function with the keyboard `map`. We can plot this function with `plot2d` and the name of the function as an argument.

```
>function map f(x) := sum(x^(1:20)/(1:20)^2)
>f(0:0.1:1)
 [0,  0.102618,  0.211004,  0.32613,  0.449283,  0.582241,  0.727586,
 0.889374,  1.07472,  1.29823,  1.59616]
>plot2d("f",-1.5,1.5):
```

We can even use the function in numerical algorithms. EMT has numerical algorithms of many kinds. In the following, we demonstrate the integral and the solver.

```
>integrate("f",0,1)
 0.643782291532
>sum(1/((1:20)^2*(2:21))) // the exact value
 0.643782291532
```

```
>solve("f",1,y=1), f(%) // solve and check f(x)=1
 0.761556096695
 1
```

The correct value for the integral is

$$\int_0^1 f(t)\,dt = \sum_{k=1}^{20} \frac{1}{k^2\,(k+1)}.$$

EMT get this value very accurately using an adaptive Gauss method in the function `integrate`. The solution of $f(x) = 1$ can be determined with `solve` and the parameter `y=1`. The solver needs a start value and uses a Secant algorithm to find the solution. Note that the `%` in the last command refers to the previous result.

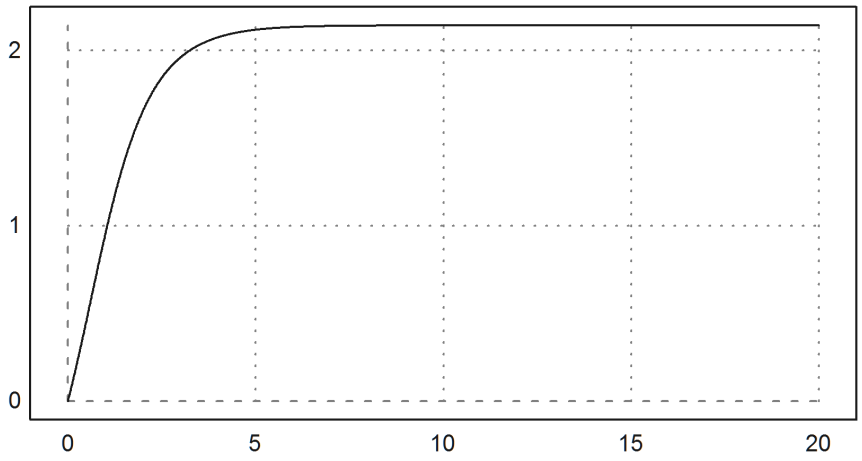



Figure 1.2: Solution of $y' = \sin(1+y)$

Differential equations can also be solved by EMT using the LSODA algorithm or Runge-Kutta formulas. For a try, let us solve

$$y'(x) = \sin(1 + y(x)), \quad y(0) = 0.$$

We solve it in the interval $[0, 20]$.

```
>x=0:0.01:20;
>function f(x,y) := sin(1+y);
>y=ode("f",x,0);
>aspect(2); plot2d(x,y):
```

For now, these are enough examples of the numerical part of EMT to show that numerical results are useful and indispensable, and they are easy to get in EMT. We did not discuss the big topic of numerical linear algebra or optimization. The program includes a lot of tools for these problems too.

## 1.4 Variables

If values or results are needed later, you need to assign them to variables. You can use := or =. Symbolic variables are set with &=.

```
>a := 2 // numerical variable (alternatively a=2)
                    2
>a &= x^2 // symbolic variable

                                          2
                                         x
```

One should be aware and I like to stress it already at this point that a numerical variable is not known in symbolic expressions. So, even if x had a numerical value in EMT its value could not be used in any symbolic expression.

If we want to set a numerical value to a variable which can be used in symbolic expressions there is &:=. In this case, the variable can no longer be used as a symbolic variable, of course.

```
>a &:= 1.5
   1.5
>& a^2

                                       9
                                       -
                                       4

>a^2
   2.25
```

The symbolic value will be a close rational approximation as you see. There is another way to use a numerical number from EMT in Maxima: Refer to the value of a variable as @variable in symbolic expressions.

```
>& @a^2
```

$$2.25$$

To suppress the printing of results use ; after the command. To print the result of one command in an command line using several command, use a comma.

```
>a:=2; a^3-a
   6
>a:=2, a^3-a
   2
   6
```

A variable with a value is of course replaced by the value. That is true for symbolic expressions too. To remove a value from a variable, use remvalue.

```
>a &= 4
```

$$4$$

```
>& a^2
```

$$16$$

```
>remvalue(a)
>& a^2
```

$$2$$
$$a$$

To assign a value to a variable in a symbolic expression for just one computation, there is the | operator or with. The previous value of the variable a is not used here.

```
>& a^2*b | a=4 | b=2
```

$$32$$

```
>& a^2*b with a=4
```

$$16\ b$$

```
>& a^2*b with [a=4,b=2]
```

32

It is good to know some internals of EMT. The Maxima system runs in the background and has its own variables. But &= defines the variable in Maxima. In EMT, such variables are strings, which print in symbolic form using the formatting of Maxima.

Often, we wish to define a variable with a numerical value in Maxima and in EMT. As already mentioned above, this is done with &:=.

```
>a &:= 4;
>&sin(a), &float(sin(a))
```

sin(4)

- 0.75680249530793

```
>sin(a)
   -0.756802495308
```

## 1.5 Expressions and Symbolic Expressions

All non-symbolic expressions in EMT are stored in strings. EMT can evaluate expressions. It uses this feature in many places. This avoids having to write a function for simple calculations.

The following examples apply some numerical methods to simple expressions. For these methods to work properly, the expressions must be expressions in "x" (resp. "x" and "y"). That is a useful convention which greatly simplifies handling expressions.

> *By convention, the default variables in expressions for numerical methods in EMT are x, y, z.*

Of course, there are ways to use other variables as arguments.

```
>solve("cos(x)-x",1) // solve cos(x)=x near 1
   0.739085133215
>integrate("exp(-x^2)/sqrt(pi)",0,10) // numerical integration
   0.5
>plot2d("x^3-x",-1,1); // 2D plot
>plot3d("x*y"); // 3D plot
```

If symbolic manipulations like differentiation and integration are necessary, we use the symbolic expressions. Of course, this means that Maxima has to process these expressions.

```
>&diff(log(x)/x^2,x) // symbolic derivative
```

$$\frac{1}{x^3} - \frac{2\log(x)}{x^3}$$

```
>&integrate(log(x)/x^2,x,1,E) // symbolic integration
```

$$1 - 2\,E^{-1}$$

```
>expr &= x^x;
>solve(&diff(expr,x),0.5) // numerical solution of expr'(x)=1/2
   0.367879441171
>plot2d(&x^3-x,r=2); // plots a symbolic expression
>plot2d(&diff(x^3-x,x),color=5,add=1): // plot will appear below
```

The plot of the function appears below the plot command in the EMT notebook due to the : after the command. You can see the plot in figure 1.3. We will later discuss plots.

The functions in EMT which accept expressions evaluate the expression in the following ways.

```
>expr:="sin(x)*x"; expr(1.2)
   1.11844690316
>"x*y"(2,3)
   6
```

This works for numerical or symbolic expressions, as long as the variables are x, ym z. Other variables must either be global or passed by assigned arguments.

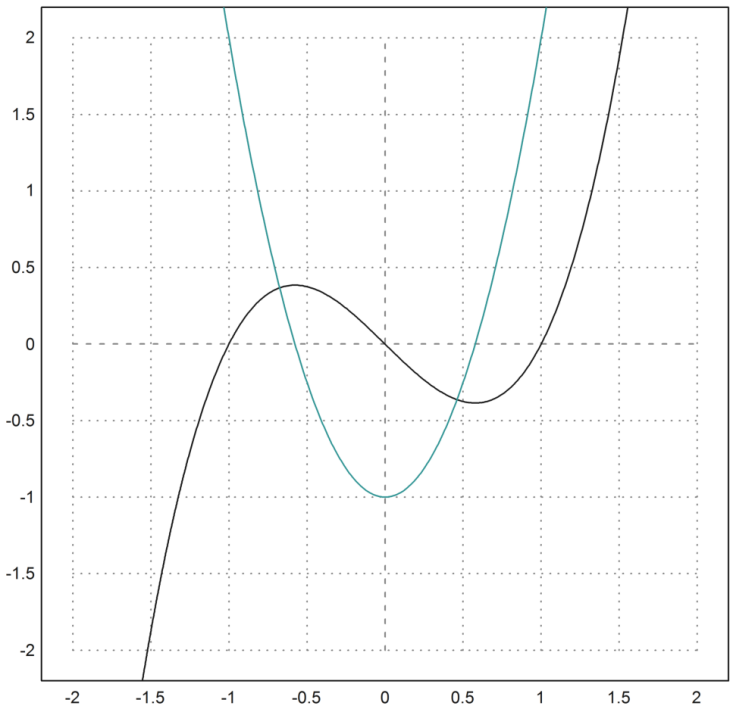



Figure 1.3: Plot of $x^3 - x$ and its derivative

```
>"a*x^2"(6,a=2)
   72
```

To evaluate a symbolic expression in a symbolic way, use `with`.

```
>df &= diff(x/(x^2+1),x)
```

$$\frac{1}{x^2+1} - \frac{2\,x^2}{(x^2+1)^2}$$

```
>&factor(df with x=a+1)
```

$$-\frac{a\,(a+2)}{(a^2+2\,a+2)^2}$$

There is a quick way to evaluate a symbolic expression numerically. If the symbolic expression starts with `&:` it is first evaluated by Maxima. Then the result is evaluated numerically.

```
>&:integrate(x*log(x),x,0,2)
 0.38629436112
>integrate("x*log(x)",0,2) // numerical integration
 0.38629436112
```

Note that the numerical integration is as accurate as the symbolic result. It is faster, however, unless the symbolic integration is done only once and evaluated in many points.

## 1.6 Discussing Functions

For an example, let us discuss a one-dimensional function. EMT can very nicely cooperate with Maxima for this purpose. For simple functions, which can be solved analytically, Maxima will be the tool of first choice.

First, we plot the function in EMT. Maxima has a very nice plotting tool too, but EMT is my choice for plots.

```
>expr &= log(x)/x;
>plot2d(expr,a=0.1,b=10,c=-3,d=3):
```

The most elementary form of the `plot2d` command in EMT can handle a single expression in `x`, or the name of a function `f(x)` defined in EMT (more on functions later). It can also handle data plots, bar plots, point plots, curves in the plane, and much more. We set options for this command (like the plot limits) using assigned arguments. Remember that the `:` after the plot command inserts the plot into the notebook window.

Storing the function as a symbolic expression has the advantage that we can use it easily in Maxima and in EMT. We can find the maximum of the function, or the inflection point, in symbolic form. The constant `E` is the Euler constant $e$, of course.

```
>&solve(diff(expr,x)=0,x)
                                    [x = E]
```

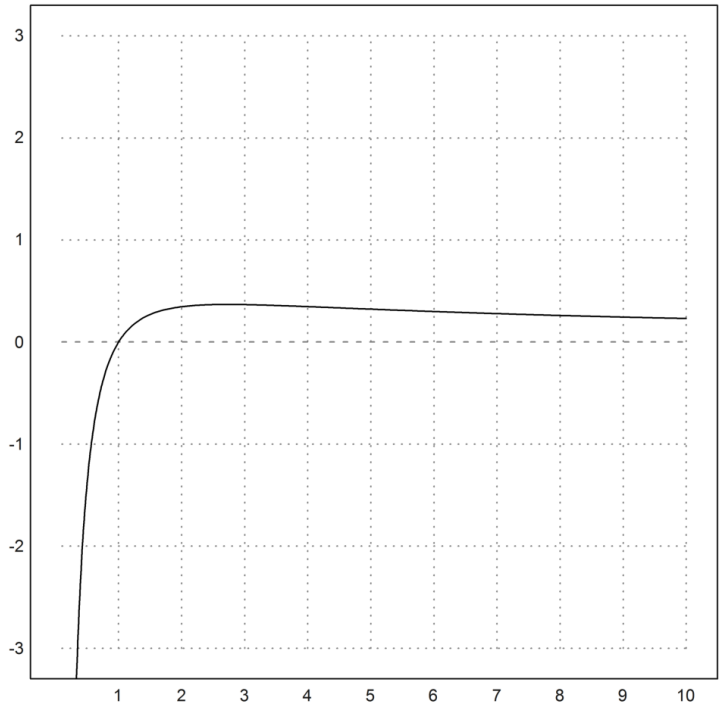



Figure 1.4: Simple plot of $\log(x)/x$

```
>&solve(diff(expr,x,2)=0,x)
```

$$[x = E^{3/2}]$$

Many exact integrals can be computed with Maxima. Try

$$\int_1^\infty \frac{\log(x)}{x^2}\,dx.$$

```
>&integrate(log(x)/x^2,x,1,inf)
```

$$1$$

We can also use the expression as a numerical expression. E.g., the solution of

$$\frac{\log x}{x} = -3$$

has no closed form. Consequently, Maxima cannot solve it. But with the right starting point we can find a numerical solution with the Secant method and the function `solve`, or with the Bisection method.

```
>&solve(expr=-3,x)
```

$$[x = -\frac{\log(x)}{3}]$$

```
>solve(expr,0.0001,y=-3)
   0.349969631655
>bisect(expr,epsilon,1,y=-3)
   0.349969631654
```

Besides storing expressions in strings, it is possible to define functions in EMT and Maxima. The syntax is similar. The following is a numerical function in EMT.

```
>function f(x) := x^3-x // numerical function
```

Functions with symbolic expressions can be defined in both worlds simultaneously. We call these functions symbolic functions.

```
>function f(x) &= diff(x^x,x) // define a symbolic function
```

$$x^x \, (\log(x) + 1)$$

```
>f(5) // use in Euler numerically
   8154.49347636
>&f(a) // and in a symbolic expression
```

$$a^a \, (\log(a) + 1)$$

If an expression string is defined only in EMT it can still be used in Maxima using the `@...` syntax. The same syntax can be used to define a function from an expression.

```
>expr := "x^3-x"; // defined only in EMT
>&diff(@expr,x) // use in symbolic expression
   3*x^2-1
>function f(x) := @expr // make it a function
```

Of course, the previous examples in this section could have been computed by hand easily. So we try a problem, which involves a lot of computations. While the student still has to know how to solve it, tedious hand computations can be avoided using a tool like Euler Math Toolbox.

### Example

The following is a more complicated example to show the benefits that you get from symbolic systems. While the computations can be done by hand they are tedious and error-prone.

We compute the paraboloid with maximal volume, which fits into a cone. Since this paraboloid must obviously touch the cone, and we assume it to be symmetric with respect to the center axis of the cone (it is not obvious why we can do this!), we compute the symmetric parabola, which touches the function $1-|x|$ first.

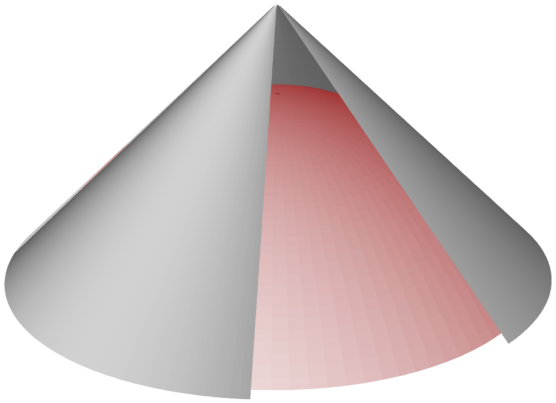


Figure 1.5: Maximal paraboloid below a cone

We search the $x$ value, where a symmetric parabola $ax^2+b$ has derivative $-1$.

```
>function p(x,a,b) &= a*x^2+b
```

$$a\,x^2+b$$

```
>sx &= solve(diff(p(x,a,b),x)=-1,x)
```

$$[x = -\frac{1}{2\,a}]$$

Now, we need to solve $p(x) = 1 - x$ for $a$ and $b$. Instead of inserting the value $-1/(2a)$ by hand, we use `with`. We get an equation, which we can solve for $a$.

```
>&p(x,a,b)=1-x, sa &= solve(% with sx,a)
```

$$a\,x^2 + b = 1 - x$$

$$[a = \frac{1}{4\,b - 4}]$$

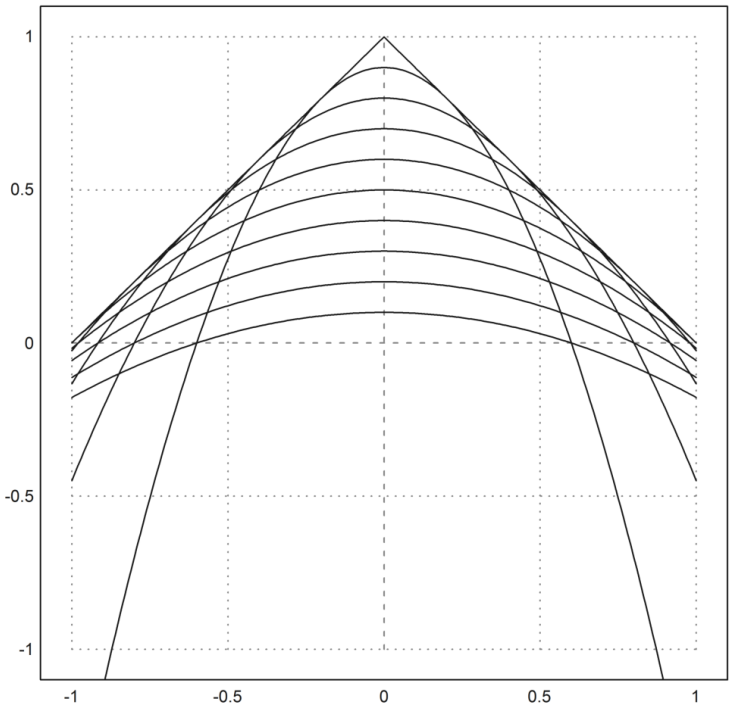



Figure 1.6: Touching parabolas

With this knowledge, we define the touching parabolas $pt_b(x)$. Using the EMT matrix language, we can plot the parabolas for several values of $b$ at once.

```
>function pt(x,b) &= at(p(x,a,b),sa)
```

$$\frac{x^2}{4\,b-4}+b$$

```
>b:=(0.1:0.1:0.9)';
>plot2d("1-abs(x)",r=1); plot2d(&pt(x,b),>add):
```

Now we rotate these parabolas around the $y$ axis and compute the volumes of the paraboloids. To be able to do this, we need the inverse function, which will be the radius of the cut through the paraboloid at height $y$.

```
>sol &= solve(y=pt(x,b),x)
```

$$[x = -2\sqrt{b\,y-y-b^2+b},\ x = 2\sqrt{b\,y-y-b^2+b}]$$

```
>function ptinv(y,b) &= rhs(sol[2])
```

$$2\sqrt{b\,y-y-b^2+b}$$

With this, we can compute the volume, and finally maximize it.

```
>function F(b) &= integrate(pi*ptinv(y,b)^2,y,0,b)
```

$$-2\,\pi\,(b^3-b^2)$$

```
>&solve(diff(F(b),b)=0,b)
```

$$[b = \frac{2}{3},\ b = 0]$$

```
>F(2/3)
   0.93084226773
```

The details of this computation, especially the Maxima syntax, are not self evident. For a closer explanation, have a look into the chapter about Maxima in this introduction.

## 1.7 Solving Equations

Maxima has the `solve` command to solve an equation or a system of equations. It is able to handle most school book examples, and goes very much beyond it. In the following example, we get complex results.

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

$$[x = \frac{1 - \sqrt{3}\,I}{2}, x = \frac{\sqrt{3}\,I + 1}{2}]$$

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

$$0$$

To insert the solutions in subsequent expressions, use `with`. Here, we inserted the second solution of the form `x=...` into the equations. We had to use `expand` to simplify this to 0.

However, `solve` cannot solve all equations, not even all polynomial equations. Then, we have to be content with numerical solutions. In the following example, we use the numerical solver of EMT to compute the zero, which we can clearly see in the plot.

```
>plot2d("x^6-10*x+1"):
>longest solve("x^6-10*x+1",0,1)
           0.1000001000006
```

To show more digits of the result, we are using the operator `longest`. There are more operators of this kind. Try `shortest` or `fraction`.

EMT can also use interval arithmetic to get a close and guaranteed inclusion of the result. Since the interval Newton method needs the derivative for this, it calls Maxima.

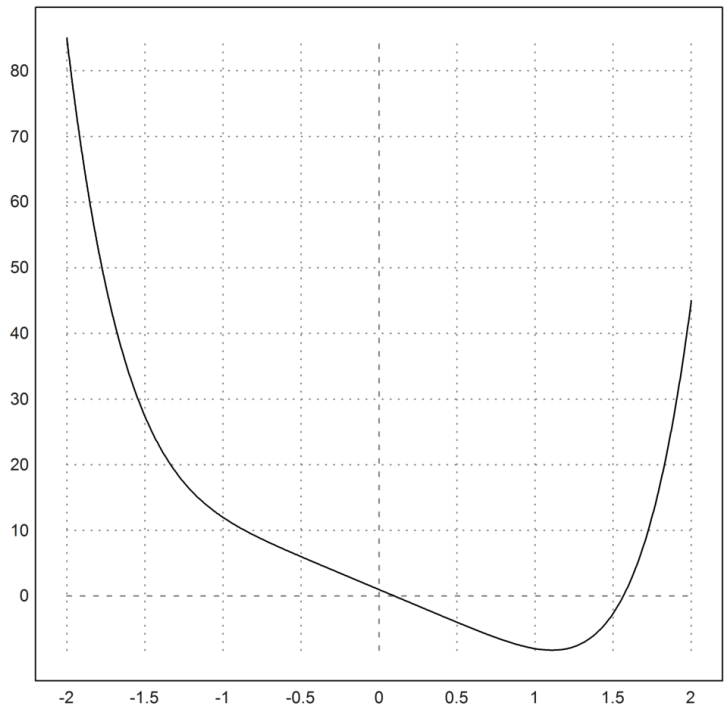



Figure 1.7: $y = x^6 - 10x + 1$

```
>mxminewton("x^6-10*x+1",~0,1~)
   ~0.10000010000059993,0.10000010000060007~
```

Systems of equations can be linear or non-linear. Both can be handled by Maxima in the same way.

```
>&solve([a+b=4,a-2*b=3],[a,b])
```

$$[[a = \frac{11}{3}, b = \frac{1}{3}]]$$

```
>&solve([a*b*c=1,a^2+b^2+c^2=3,a+b+c=3],[a,b,c])
```

$$[[a = 1, b = 1, c = 1]]$$

If there are infinitely many solutions Maxima delivers a parametric representation. The following example has three equations with three unknowns, but one equation depends on the other two.

```
>& solve([a+2*b+3*c=6,4*a+5*b+6*c=15,7*a+8*b+9*c=24],[a,b,c])

                    [[a = %r1, b = 3 - 2 %r1, c = %r1]]
```

EMT has also numerical methods to solve equations. Linear systems can be written with a matrix as $Ax = b$, and are solved with \. This works for rather large matrices. Let us try the example above.

```
>A=[1,1;1,-2]
                1              1
                1             -2
>b=[4;3]
                4
                3
>fraction A\b
        11/3
         1/3
```

If there is no solution, we can use `fit` or svdsolve, which will yield the best possible value (minimizing $\|Ax - b\|$). In case of more than one solution of the minimization problem `svdsolve` will return the solution with the minimal norm.

```
>A=[1,2;3,4;1,2]
                1              2
                3              4
                1              2
>fit(A,[1;1;1])
               -1
                1
```

A special application of data fitting are polynomial fits. In the following example, we fit a polynomial of degree 1 (a line) to data in the plane.

```
>x=0:5; y=5-2*x+normal(size(x));
>p=polyfit(x,y,1)
   [3.89899,  -1.70281]
>plot2d(x,y,>points,a=0,b=5,c=-5,d=5); ...
>plot2d("polyval(p,x)",>add,color=red):
```

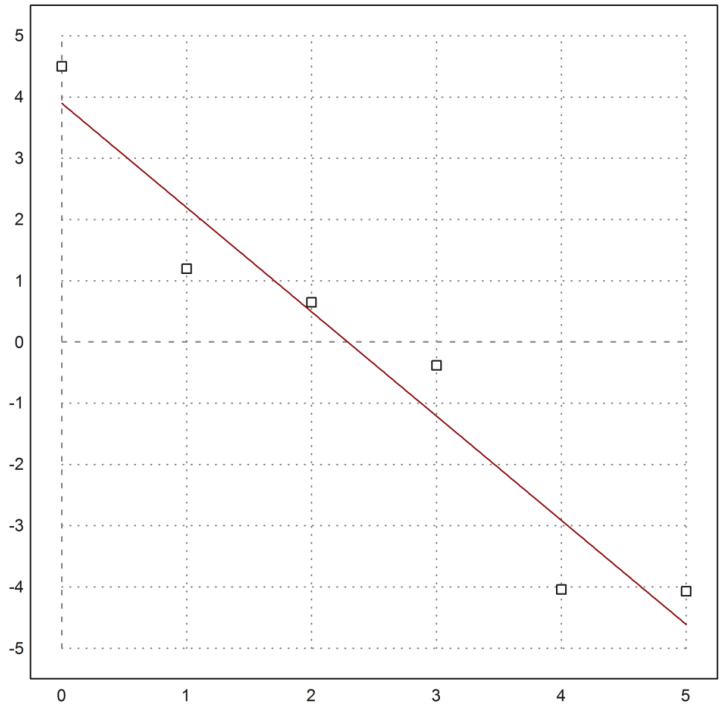



Figure 1.8: Linear Regression Line

Non-linear systems can often be solved with the Broyden algorithm. We try

$$xy = 1, \quad x^2 + y^2 = 4.$$

For the Broyden algorithm, we need to define a vector valued function first. The algorithm seeks the zero of the function. We use a symbolic vector expression to define the symbolic function. The function parameters `f([x,y])` take care, that the function can be used for two scalar values (`f(x,y)`), or a vector value (`f(v)` with a vector v).

```
>expr &= [x*y-1,x^2+y^2-4]
```

$$\left[x\,y - 1,\ y^2 + x^2 - 4\right]$$

```
>function f([x,y]) &= expr
```

$$y^2 + x^2$$

```
                          [x y - 1, y  + x  - 4]
```

```
>broyden("f",[1,2])
   [0.517638,  1.93185]
```

We can also use the very stable Nelder-Mead minimization method in this case. The Nelder-Mead algorithm minimizes functions.

```
>function g(v) := norm(f(v))
>neldermin("g",[1,2])
   [0.517638,  1.93185]
```

Of course, this example can be computed by hand too. Maxima can find all four solutions. Our solution is the fourth solution. To get its numerical value, we evaluate the vector `[x,y]` using the solution, and evaluate the result in EMT.

```
>sol &= solve(expr,[x,y]); &sol[4], %()
```

$$[x = \sqrt{2 - \sqrt{3}},\ y = \frac{1}{\sqrt{2 - \sqrt{3}}}]$$

```
 [0.517638,  1.93185]
```

Note the evaluation of the expression in the last line with `%()`. `%` refers to the previous result which in this case is a symbolic expression. The `x =` part is ignored.

To find the minimal values of a function $f$, we can use the Nelder-Mead method to find the zero of the gradient by minimizing its norm. This will succeed, even if the Jacobian of the gradient (the Hessian matrix fo $f$) is singular. However, the numerical stability of this procedure is not very good. This is due to the mathematical fact the minimum of a smooth function is not well determined by the values of the function.

```
>function f([x,y]) &= x^4+(y-1)^4
```

$$(y - 1)^4 + x^4$$

```
>function gradf([x,y]) &= gradient(f(x,y),[x,y])
```

```
                                  3           3
                               [4 x , 4 (y - 1) ]
```

```
>function h(v) := norm(gradf(v))
>nelder("h",[1,1])
   [2.92159e-005,  1.00002]
>&solve(gradf(x,y))[1]
```

```
                                [y = 1, x = 0]
```

## 1.8 Vectors and Matrices

One good reasons to use EMT is its matrix language. Of course, EMT can compute linear algebra expressions. But the main advantage is that EMT can use vectors to compute and hold tables of values.

The basic rule is the following:

> *If any EMT function or operator with scalar arguments and a scalar result is applied to vectors or matrices, it is applied to each element and the result is a vector or matrix of the same size.*

This is called vectorization. The operators and functions of EMT vectorize to their arguments.

```
>v=1:5
 [1,  2,  3,  4,  5]
>v^2
 [1,  4,  9,  16,  25]
>v*v // element-wise multiplication
 [1,  4,  9,  16,  25]
>sqrt(v)
 [1,  1.41421,  1.73205,  2,  2.23607]
```

Let us compute a vector of values of the binomial function, and compute the sum, the mean and the maximal value of the vector.

```
>n=0:20; b=bin(20,n); // Compute bin(20,0) ... bin(20,20)
>plot2d(n,b,points=1,yl="bin(20,n)",xl="n",>smaller):
>sum(b), 2^20 // the sum is 2^20
      1048576
      1048576
>sum(n*b/2^20) // expected value
      10
>max(b)
      184756
```

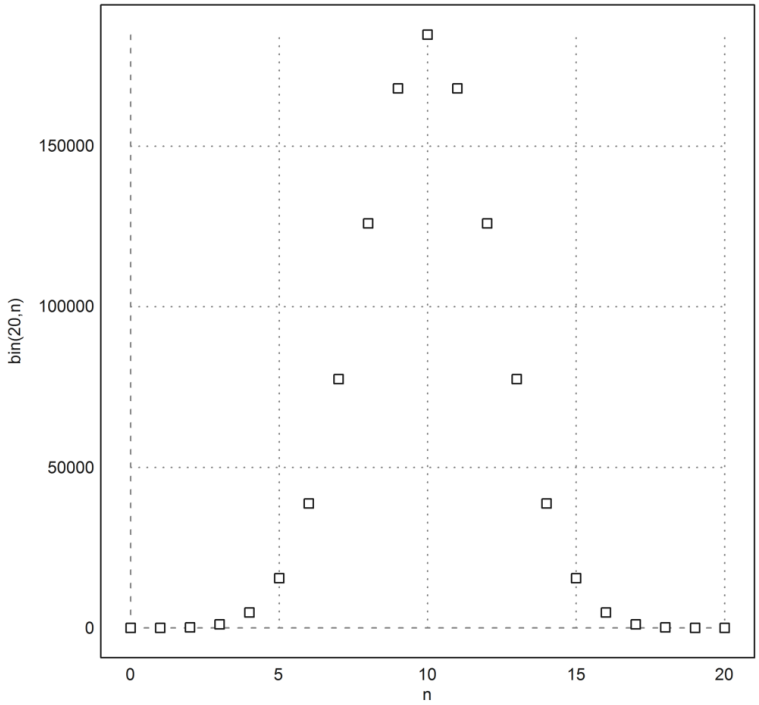



Figure 1.9: bin(20,n)

All the built-in procedures in EMT are applied to matrix input elementwise. If the parameters are a vector and matrix, the elements are taken in a natural way. E.g., if $v$ is a column vector, and $w$ is a row vector the $A = v * w$ is the matrix

$$A[i,j] = v[i] \cdot w[j].$$

Of course, this is not the matrix product. It is called a tensor product of the two vectors. The matrix product is computed with the dot . in EMT.

```
>format(5,0); A=[1,2;3,4]
```

```
       1    2
       3    4
>A*A
       1    4
       9   16
>A.A
       7   10
      15   22
```

### Example

We compute a table of multiplication up to 10. The expression `n'` returns the transposed vector of the row vector $n$, a column vector.

```
>n=1:10; A=n'*n;
>format(5,0); A,
       1    2    3    4    5    6    7    8    9   10
       2    4    6    8   10   12   14   16   18   20
       3    6    9   12   15   18   21   24   27   30
       4    8   12   16   20   24   28   32   36   40
       5   10   15   20   25   30   35   40   45   50
       6   12   18   24   30   36   42   48   54   60
       7   14   21   28   35   42   49   56   63   70
       8   16   24   32   40   48   56   64   72   80
       9   18   27   36   45   54   63   72   81   90
      10   20   30   40   50   60   70   80   90  100
>defformat; // reset to default format
>reset; // does the same and more
```

To plot a function we can generate a table of values of the function first. It is the only way if the vector is generated elementwise, e.g. by some random process. As an example, we generate a Brownian motion by adding random numbers. That is, we compute

$$s[i] := t[1] + \ldots + t[i]$$

for $i = 1, \ldots, 1000$, where the $t[i]$ are normal distributed values. The function `cumsum` does just what we need.

```
>t:=normal(1,1000); s:=cumsum(t); plot2d(s):
```


Figure 1.10: Brownian motion

## 1.9 Sequences

A simple sequence can be generated with the `:` operator and the matrix language of EMT. Functions like `sum`, `cumsum` or `differences` can be applied to sequences. Note that the factorial function `!` vectorizes too.

```
>n=1:10; n^2
   [1,  4,  9,  16,  25,  36,  49,  64,  81,  100]
>sum(n^2)
   385
>differences(n^2)
   [3,  5,  7,  9,  11,  13,  15,  17,  19]
>cumsum(n)
   [1,  3,  6,  10,  15,  21,  28,  36,  45,  55]
>n!
   [1,  2,  6,  24,  120,  720,  5040,  40320,  362880,  3.6288e+006]
```

In Maxima there is a function that can produce symbolic sequences. For an example we create the polynomials

$$e^{-x^2}\,\frac{d^n}{dx^n}e^{x^2}$$

for $n = 1, 2, 3, 4$. The third parameter in `diff` is the order of the derivative.

```
>&create_list(expand(diff(exp(x^2),x,n)*exp(-x^2)),n,1,4)
```

```
                    2         3             4       2
[2 x, 4 x  + 2, 8 x  + 12 x, 16 x  + 48 x  + 12]
```

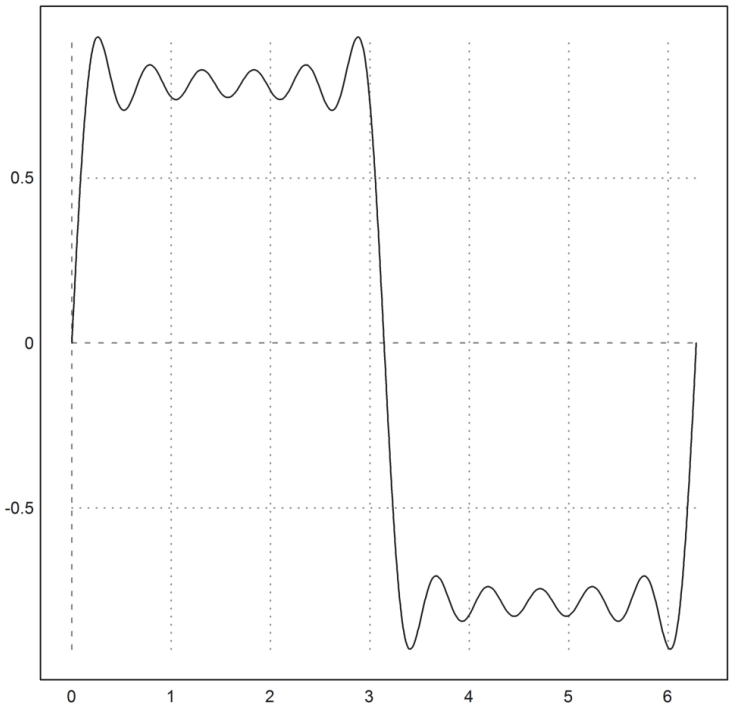



Figure 1.11: A trigonometric sum

Moreover, the function `sum` can be used in Maxima to generate a sum over a sequence.

```
>f &= sum(sin((2*n-1)*x)/(2*n-1),n,1,6)
```

```
          sin(11 x)   sin(9 x)   sin(7 x)   sin(5 x)   sin(3 x)
          --------- + -------- + -------- + -------- + -------- + sin(x)
```

```
              11          9          7          5          3
```

```
>plot2d(f,0,2pi):
```

The same plot can be generated in the matrix language of EMT. We take a row vector $x$ and a column vector $n$ and combine those to compute $\sin(nx)/n$. Then we take the sum of the columns of the result. Since `sum` takes the sum of the rows, we have to transpose twice.

```
>n=(1:2:11)'; x=linspace(0,2pi,500); y=sum((sin(n*x)/n)')';
>plot2d(x,y):
```

There are also simple functions for recursively defined sequences. One example is `sequence`. The expression for this command can involve a vector $x$ containing the previous elements of the sequence, and the index $n$ of the new element.

```
>shortformat;
>sequence("x[n-2]+x[n-1]",[1,1],10) // Compute the Fibonacci sequence
   1  1  2  3  5  8  13  21  34  55
```

Of course we can also use a loop. Loops on the command line (outside of functions) can be used, as long as they fit into one command line or a multi-line (see below). The details of the syntax for loops are explained in the section about programming.

```
>v=ones(10); for i=3 to 10; v[i]=v[i-1]+v[i-2]; end; v,
   [1,  1,  2,  3,  5,  8,  13,  21,  34,  55]
```

Note that we have defined the vector beforehand. We could also append each new element to the vector. This would be slightly less efficient.

```
>v=[1,1]; for i=3 to 10; v=v|(v[-2]+v[-1]); end; v,
   [1,  1,  2,  3,  5,  8,  13,  21,  34,  55]
```

Note the negative indices `v[-1]` and `v[-2]`. They count from the end of the vector.

There is a simple function which iterates an expression or another function to convergence. Of course, it works only if the iteration converges. It stops in a fixed point.

```
>iterate("cos(x)",1), cos(%)
   0.739085133216
   0.739085133215
```

If the process converges the result is a fixpoint of the function (i.e. $f(x) = x$). This works for vector valued functions too. Let us try

$$f(x,y) = \left(\frac{x}{2} - \frac{y}{4}, 1 + x^3 + \frac{y}{2}\right).$$

```
>function f([x,y]) := [x/2-y/4,1+x^3+y/2]
>v=iterate("f",[1,1]), f(v)
   [-0.682328,  1.36466]
   [-0.682328,  1.36466]
```

### Example

For an example, we compute the length of the $3n+1$-sequence starting from any value $n_0$. The sequence is defined by

$$n_{k+1} = \begin{cases} n_k/2, & n \text{ even}, \\ 3n_k + 1, & n \text{ odd}. \end{cases}$$

It has been conjectured that the sequence reaches $n_k = 1$ for some $k$ for all starting points $n_0$.

For the iteration we can use `iterate` with an end condition and a maximal number of iterations. The argument `till` is a stopping condition for the iteration.

We could use an expression (with `case`) here.

```
>iterate("case(mod(x,2),3*x+1,x/2)",17,till="x==1",n=1000)
 [17,  52,  26,  13,  40,  20,  10,  5,  16,  8,  4,  2,  1]
```

But we also want to demonstrate a multi-line function in the programming language of EMT.

```
>function step (n) ...
$  if mod(n,2)==0 then return n/2;
$  else return 3*n+1;
$  endfunction
>iterate("step",17,till="x==1",n=1000)
   [17,  52,  26,  13,  40,  20,  10,  5,  16,  8,  4,  2,  1]
```

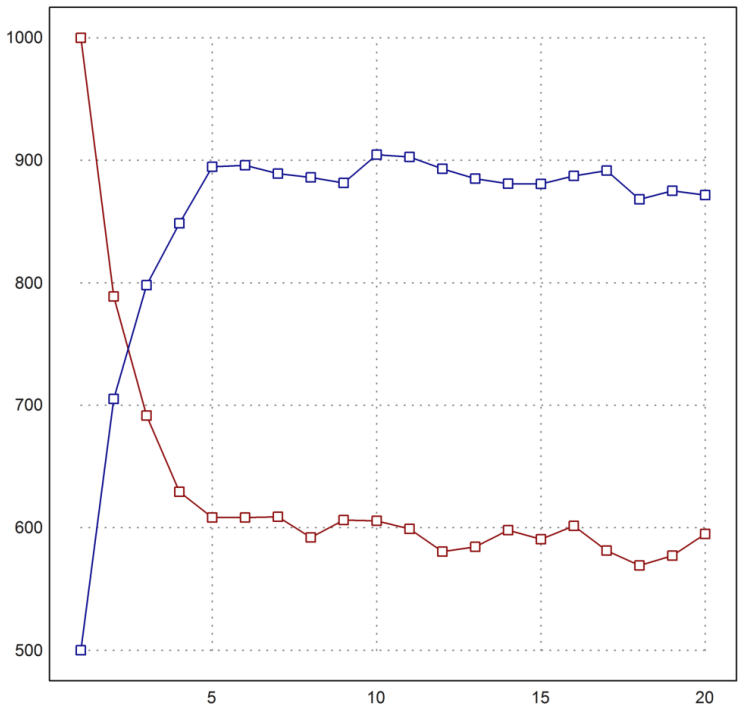



Figure 1.12: Stochastic Changes

### Example

For another example, we use vector iteration to apply a stochastic matrix to some start values over and over again, adding a stochastic variable in each step.

$$x_{n+1} = Ax_n + X_n.$$

We use the function `sequence` for this. The trick is to take the last column of `x` for the next step. The matrix `x` contains the values computed so far in its columns.

```
>A=[0.7,0.2;0.3,0.8]
              0.7           0.2
              0.3           0.8
>x=sequence("A.x[,-1]+10*normal(2,1)",[1000;500],20);
>plot2d(x,>addpoints,color=[red,blue]):
```

Of course, we could also use a loop. We show how a multi-line function simulating the process might look like.

```
>function simulate (x,A,s,n) ...
$  loop 1 to n;
$     x=A.x+s*normal(size(x));
$  end;
$  return x;
$endfunction
```

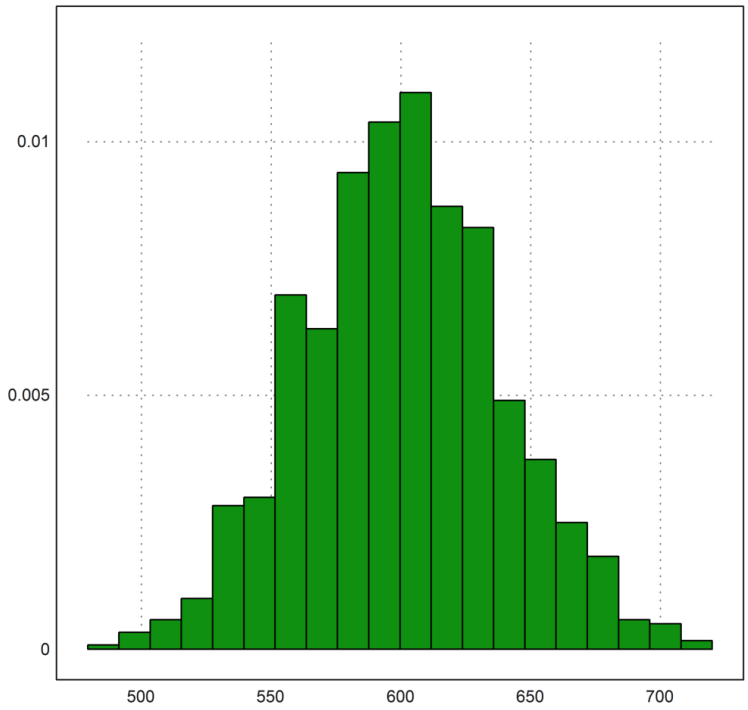



Figure 1.13: Distribution of `x[1]`

Now we do this 1000 times and plot the distribution of the first value of the vectors.

```
>m=1000; v=zeros(m);
>for k=1 to m; v[k]=simulate([1000;500],A,10,20)[1]; end;
>plot2d(v,>distribution):
```

This is a nice Monte-Carlo simulation. The expected value depends on the eigenvector of the matrix to the eigenvalue 1, i.e., the solution of $Ax = x$, and also on the total number of individuals in the starting $x$. In our case, we start with 1500 individuals.

```
>v=kernel(A-id(2)); fraction v
         2/3
           1
>1500/sum(v')*v
             600
             900
```

Monte-Carlo simulations are a nice and useful application of numerical software. We show more of this type of applications in the chapter about statistics.

# Chapter 2

# Introduction

## 2.1 Overview

Euler Math Toolbox (EMT for short) is a numerical and algebraic software, a mixture between a computer algebra system (CAS) and a numerical matrix language in the style of Matlab. The numerical part has been programmed by R. Grothmann, a mathematician at the University of Eichstätt. The algebraic part uses Maxima, a mature software maintained by a group of enthusiasts.

The numerical kernel of EMT is based on a matrix language. This language can not only handle simple numbers, but also vectors and matrices of numbers. Moreover, it can compute expressions with complex numbers, intervals and strings. All these computations can be programmed in functions, which might be loaded from external files into EMT. Indeed, a large part of the EMT syntax is based on functions written in the EMT programming language.

EMT can also produce graphics, store these graphics in an internal meta format, and output the graphics for various media, e.g., the graphics window. Graphics can also be imported into the notebook window, exported to files or to the clipboard. The notebook window can be exported as an HTML page for the web.

The Maxima subsystem communicates with EMT through pipes. It remains a separated system. However, there are various interactions between EMT and Maxima, and the computer algebra is seamlessly integrated by symbolic expressions. This is a mighty environment to do mathematics.

Advanced features of EMT are an exact scalar product providing guaranteed solutions together with the interval arithmetic, interfaces to Povray, Python, and TinyC,

SVG export of graphics, HTML or PDF export, Latex formulas in notebooks and graphics, and much more.

EMT started about 1988. The aim was to get an interactive mathematical system on an Atari ST. The program was never a clone of Matlab, and went its own path since the beginning. The current version works on Windows. Older versions for OS/2 or Linux are no longer up to date. However, EMT will run in Linux or OSX under Wine, including computer algebra support by Maxima. The native version for Linux, based on the port by Eric Boucharé is very restricted and outdated, and does not provide symbolic features.

## 2.2 First Steps

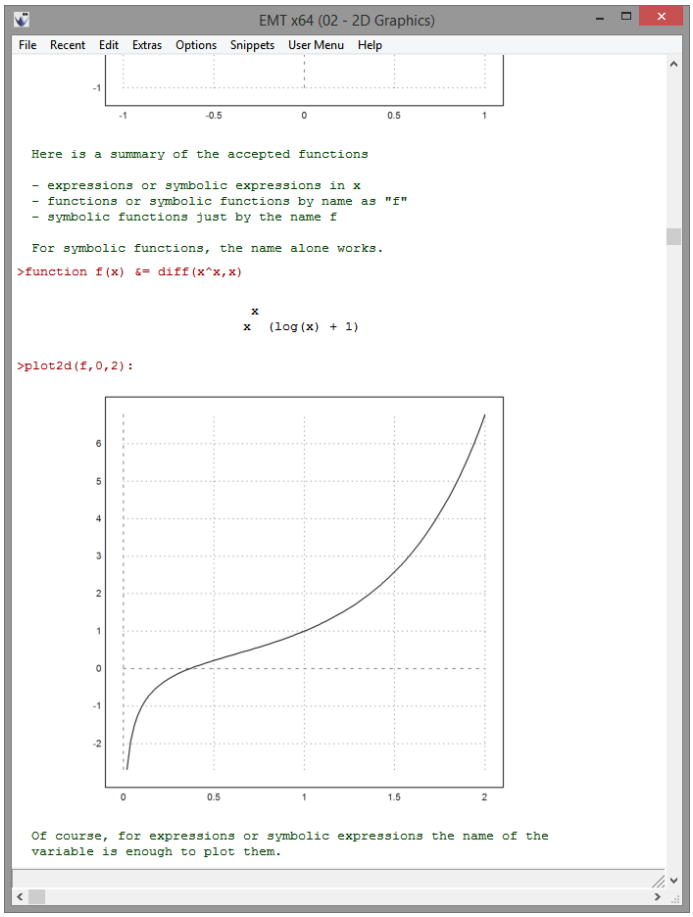



Figure 2.1: The text window of EMT

From the user viewpoint, EMT looks like a command shell in a text window. Here, commands can be entered and executed, and the output will appear in the same window below the command. There is a second window, the graphics window, for the graphical output of EMT. The current graphics in the graphics window can be inserted into the text window. All commands in the text window can be edited and executed at any time. It is also possible to add comments. We refer to the content of the text window as an EMT notebook. Notebooks can be exported to HTML pages.

The following windows of EMT can be opened at any time.

- The notebook window containing commands, output, comments and graphics.
- The graphics window containing the current graphics. The current graphics can be inserted into the notebook text using : at the end of the command. Graphics can alternatively be displayed in the notebook window (toggle with Ctrl-G) and brought to view by the tabulator key TAB.
- The help window containing a search line, help text and a search history.

After installing EMT with Maxima, you will find a shortcut in the start menu and on the desktop to start the program. Start Euler, and enter the following commands for a first test.

```
>1+1
                        2
>exp(-0.5)*sin(3/2*pi)
         -0.6065306597126
```

If you make a mistake, go to the error with the mouse or the cursor keys, and correct the input. Press return anywhere in the line to start the computation. Try double clicking on exp or sin to open the help window, or use F1.

For a next step, I suggest opening the tutorial notebooks. Use the help menu to find the tutorials. They should be contained in the EMT installation. After reading a few of these introduction notebooks, you should have a very good idea of EMT.

Since EMT uses commands, you should take the time to learn the proper syntax of the program. This text tries to give you a good start on all aspects of EMT.

| Further Information | |
|---|---|
| Open Help Window (F1) | Help on commands. |
| Open Tutorials | Opens one of the tutorials. |
| Browse ... | Open help in the browser. |
| About Euler | Version information. |

## 2.3 The Interface

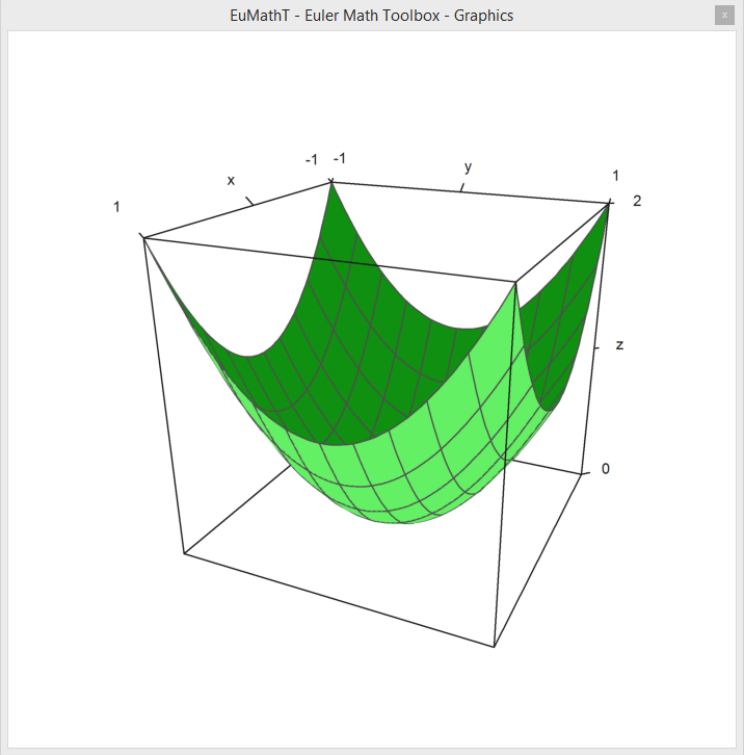



Figure 2.2: Graphics window of EMT

There is a HTML file in the documentation with a full description of the Euler GUI. Here, we give only a quick overview.

The interface of EMT is relatively simple with its notebook window and its graphics window. The notebook window has the usual menu, the text area and a status line. Both windows can be enlarged. EMT will remember the sizes and the positions of the windows depending on the current screen resolution. You can also try F11 to toggle maximizing the window layout on your screen.

The advantage of such a simple interface is that it is not complicated to learn. The disadvantage is that it does not offer graphical icons listing all the available commands. Since it is impossible to squeeze all options of EMT and Maxima into menus or icons, EMT currently has no further icons. However, there is a reference and extensive help available in the help window and the browser.

The graphics window contains the current graphics. It can be resized. The default aspect ratio is square. This can be changed in the settings globally, or with the

aspect command for the next plot.

```
>aspect(16/9); plot2d("sin(x)/x",0,2pi,xl="x",yl="sin(x)/x"):
>aspect; // reset aspect ratio
>reset; // reset default values
```

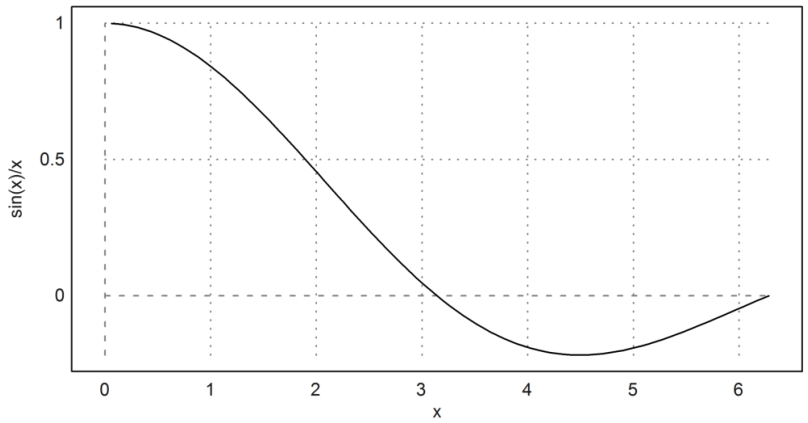



Figure 2.3: 16:9 aspect ratio in EMT

Note that the layout of the graphic depends on the aspect ratio. The function shrinkwindow will be called to fine-tune the graphics so that the labels have enough space.

The colon : after the plot command inserts the current graphics into the notebook. These graphics will be exported and saved along with the notebooks in a subdirectory images. The graphics window can also be dismissed (Ctrl-G). Then the graphics can be viewed in the notebook window. Press the TAB key to see the graphics in this mode or when the graphic is hidden. It is possible to show the graphics in EMT programs with the wait command after a plot.

The content of the graphics window can be exported in various formats. Either use the file menu, or the functions savepng, savesvg, saveps.

| | Further Information |
|---|---|
| TAB key | Switches between text and graphics. |
| Ctrl-G | Toggles the graphics window. |
| : after a plot command | Inserts the current graphics into the notebook. |

## 2.4 The Command Line

All commands are entered into the notebook window in the current command line. The commands are in dark red and start with the prompt > (followed by optional toggles for Maxima or Python). The user cannot delete the prompt >. Furthermore, the text window may contain comments in green color, and function definitions in blue color starting with $. Note that colors are not shown in this introduction. All EMT commands and the output is printed in blue in this text.

Enter commands just like in any other program which uses a command shell. Use the arrow keys or the mouse to move the cursor. The Ctrl key together with the left and right key move the cursor word by word. Have a look into the edit menu for more options. Press the Return key anywhere in a line to execute the command. This will put the cursor into the next command line. To walk between command lines use Cursor-Up or Cursor-Down. For other navigational keys see the Edit menu.

Note that command lines do not automatically execute if they are changed. You need to press Return to execute the command. If you want to recompute a section of commands, use Shift-Return. This will execute all commands in the current section. A section is marked by a heading in the comment or by an empty command line.

For an example have a look at the following two commands.

```
>x=2
   2
>x=(x+2/x)/2
   1.5
```

The second line puts a new value to the variable x. If you execute this line over and over (with Cursor-Up and Return) you get new values in x. The values quickly converge to $\sqrt{2}$. If you press Shift-Return, the state as printed above is restored. Of course, you can run a command several times with a loop.

```
>x=2
   2
>loop 1 to 5; x=(x+2/x)/2, end;
   1.5
   1.41666666667
   1.41421568627
   1.41421356237
   1.41421356237
```

Command lines can be deleted with Alt-Back. The deleted lines are accumulated and can be inserted anywhere with Alt-U. Alternatively, you can use cut and paste as described below. A new line can be inserted with Alt-Insert. The shortcuts for this are listed in the edit menu.

To stop a long computation press the Esc key. This will also interrupt the print of long vectors. (By default long vectors do not print completely, but this can be forced with the operator showlarge.)

To add a comment to a command line, start the comment editor with F5. The syntax of comments will be explained later. For now, just type paragraphs without using the enter key, and separate paragraphs by empty lines.

Simple one line comments can be appended to the command line using //.

```
>sum(1:1000) // computes the sum from 1 to 1000
                  500500
```

If lines end with three dots ... EMT will execute the complete command at once (muli-line commands). This will work, even if the cursor is in the second line of the multi-line command.

```
>plot2d("x^y",a=0,b=3,c=0,d=3, ...
>  levels=-100:0.2:100, ...
>  title="x^y"):
```

To split a line into a multi-line, press Ctrl-Return. To join two multi-lines, go to the start of the second line and press Ctrl-Backspace. Alternatively, open the internal editor with F9 to edit all lines of the multi-line at once.

You can mark text in the text window by dragging the mouse over it. Marked text can be copied to the clipboard. The copied text can then be inserted into the current command line, if it is less than one line, or in front of the current command line, if it contains several lines. This is also a quick way to send EMT commands by mail.

There are special menu entries to copy commands only, or to copy a formatted version of the notebook. Copying commands only is designed to generate an EMT file from the marked commands. It will not contain any output. Formatted copying is good for generating documentation files for EMT. The output can still be pasted back to EMT.

| Further Information | |
|---|---|
| , | Separates the commands in one line. |
| ; | Separates the commands and suppresses output. |

## 2.5 Syntax

EMT distinguishes commands, expressions and assignments.

```
>list sin // command

     *** Functions:
   alsingular antialiasing arcsin asin asinh isinterval sin sinc sinh

     *** Maxima:

   sin sinh sinsert sinnpiflag sinvertcase
>a := sin(pi/2) // assignment
   1
>sin(pi/4)^2+cos(pi/4)^2 // expression
   1
```

`list` is a command to find all functions or commands containing the string.

The most elementary expressions are mathematical expressions. EMT will evaluate these expressions with the usual order of evaluation.

```
>(1+2*4)*(3+4)/(4+2)
                     10.5
```

In case of doubts, use brackets. Exponents evaluate from right to left in EMT just like in the majority of programs (with the exception of Matlab, but not Scilab).

```
>3^3^2, (3^3)^2, 3^(3^2)
   19683
   729
   19683
```

Note that the division is done with `/`. Currently, there is no 2D editor in EMT for expressions with fractions. You can print expressions in 2D with Maxima or Latex.

```
>& a^2/(a+1)

                                    2
                                   a
                                  -----
                                  a + 1

>$ a^2/(a+1)
```

The last command will print the expression formatted by Latex if Latex is properly installed.

$$\frac{a^2}{a+1}$$

There are functions which work like commands. The result of these functions is of no interest (or it is the internal value none which will never print anything). In this case, you can prevent the output of the result with ;.

An assignment contains a variable name on the left side, and an expression on the right side of :=. Here, the semicolon ; will often be used to suppress the output.

```
>a := 2;
>a := a^2;
>a
                          4
>a := a^2;
>a
                          16
```

EMT does also understand the syntax a=2 for assignments. However, in cases like a=a^2 this looks confusing, so I selected the verbose form $:=$ for most examples in this introduction.

There are also multiple assignments to assign multiple return values of functions to several variables. Read more about this in the section about programming.

| Further Information | |
|---|---|
| >x^2 | Sends the command to EMT. |
| >& x^2 | Sends the command to Maxima. |
| >>> print 4^2 | Sends the command to Python. |
| 2^2^2 | Evaluates to 2^(2^2). |
| Ctrl-Cursor up | Calls an old command from the command history. |

## 2.6 Notebooks

The content of the notebook window is called a notebook. Notebooks can be saved for later reload, or exported to HTML for web pages or printing, and to PDF using Latex. Notebooks can contain

- commands, your input,
- output, the result of EMT or Maxima computations,
- comments, formatted with the comment editor,
- graphics, inserted via `:` or loaded from file.

If a notebook containing images is saved, the images are saved in separate files in the PNG format. By default, images are saved into a sub-directory `images`.

To navigate through the commands in a notebook use the mouse or the cursor keys. The comments and the output of a command as well as the graphics belong to the command. If the command is deleted these items are deleted too. To edit the comment of a command, use the comment editor with `F5` (see below for the syntax). The output cannot be edited. Command lines can be inserted and deleted only together with their comments and their output.

Note that the order of execution may not be the order of the commands in the notebook. Entering the commands in the following notebook yields the output printed here. If you go back from the third to the second line and execute it once more, $a = 4$ will be used, and the output changes to 16.

```
>a:=2;
>a*a
                          4
>a:=4;
```

To run all commands in a notebook use the corresponding menu entry or `Ctrl-R`. All variables will be set in the correct order then. To run all commands in a section press `Shift-Return`. A section starts end ends with a heading in the comment or with an empty line. For a clean start, re-load the current notebook via the list of recent notebooks. By default, EMT will restart automatically, when a new notebook is loaded, i.e., all values of all variables are reset and lost and the graphics is reset via the `reset` command.

Saved notebooks should have the extension `*.en`. These files are ordinary text files. However, it is not recommended to edit notebooks with an external text editor. The character encoding of the notebooks is the current encoding of the system the notebook was saved with. Notebooks with comments in foreign languages may not look well in wrong encodings. Currently, Euler does not save notebooks in Unicode encoding.

Images will be saved in separate files in PNG (portable network graphics) format, and are referred from the notebook file by file name.

By default, EMT performs a fresh restart when a notebook is loaded. This can be disabled with a switch in the options menu. A restart affects Maxima too. The Maxima process ends, and starts again.

It is possible to open a second process of EMT with `Ctrl-Shift-o`. The second process will have the graphics window hidden and will not save any settings. This helps to look up commands in other notebooks or to copy and paste parts of another notebook. Of course, a new process of EMT can also be started with the program icon, or by shift-clicking on the icon in the task bar if EMT has been added to the task bar by the user.

| Further Information | |
|---|---|
| Open Notebook | Open notebooks in the current directory. |
| Open User Notebook | Open notebooks in the user directory. |
| Open Tutorials or Examples | Open an included notebook. |

## 2.7 Comments

Comments belong to a command line. To edit the comment for this command line, press F5. A simple internal editor will open.

You do not need to break lines in the editor. This is done automatically when the paragraph is inserted into the notebook. It looks good to separate paragraphs by empty lines. The line length of the inserted comment is set to the default of 80. But you can change the width of the output in EMT in the settings. If you like you can change it to the width of the notebook window.

Some formatting and some special items are possible in comments. Here is an example.

```
* Heading

This is a paragraph. There is no need to break lines within a
paragraph. EMT does this automatically when the comment is inserted
into the notebook.

Break paragraphs with empty lines. The following are formulas in
```

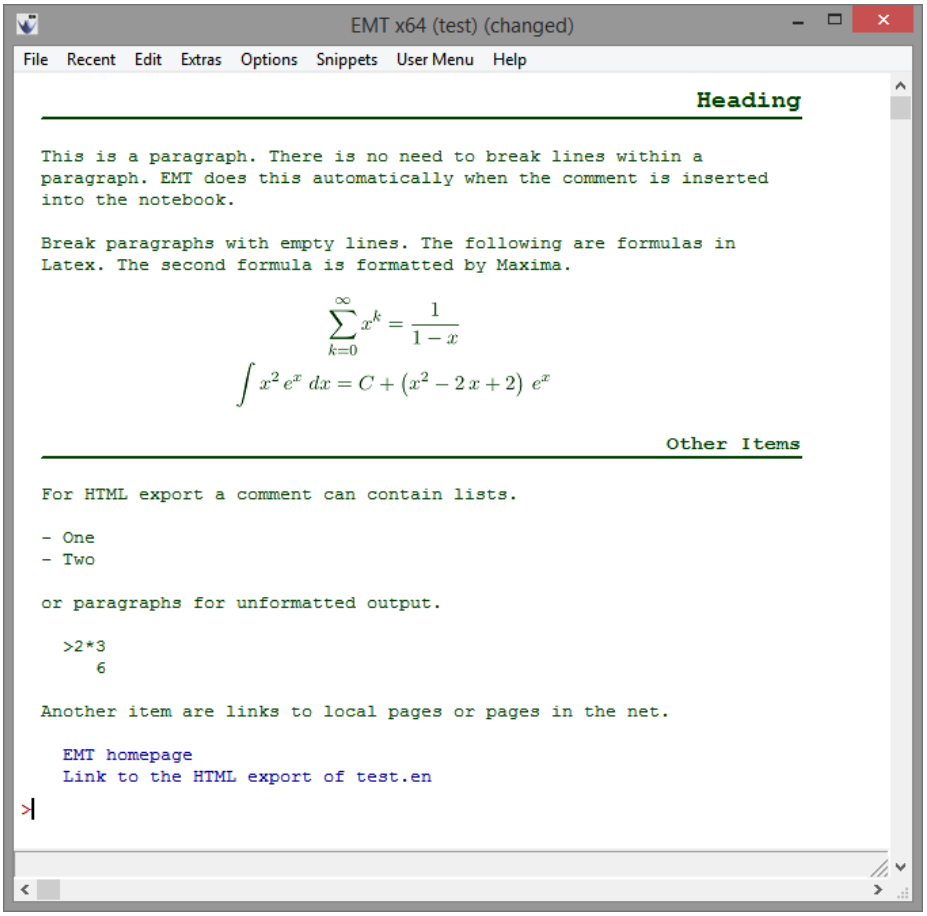



Figure 2.4: Example for Comments

```
Latex. The second formula is formatted by Maxima.

latex: \sum_{k=0}^\infty x^k = \frac{1}{1-x}
maxima: 'integrate(x^2*exp(x),x) = integrate(x^2*exp(x),x) + C
```

The formulas are parsed by Latex. The second formula is parsed by Maxima before the Latex code goes from Maxima to Latex. Instead of `latex:` you can also use `mathjax:`. This will still be parsed by Latex, but the HTML export will produce a link to MathJax.

For the HTML output, paragraphs can contain lists

```
** Other Items
```

```
For HTML export a comment can contain lists.

- One
- Two

or paragraphs for unformatted output.

    >2*3
       6

Another items are links to local pages or pages in the net.

See: http://www.euler-math-toolbox.de | EMT homepage
See: test.html | Link to the HTML export of test.en
```

See figure 2.4 for the notebook with these comments. For the example, I have put both comments to subsequent empty command lines. If a command line consists of nothing but an empty comment `//` it is empty and will display only if it has the focus.

Comments can also load images from files. For details, check the help text for comments as explained in the following section.

## 2.8 The Help Window

EMT comes with a complete reference in English, a reference to the Euler interface, quick tips, tutorials, demos, examples, and this documentation. Use the help menu to access these items, or press F1 to open the help window. The help menu can also be opened by double clicking a command in the notebook.

In the help window, you can search for help on a command by typing the command into the input line. While you type, the commands starting with the string appear. You can double click any item to open this command. In the help section of the command you can double click on any other command. Use the breadcrumbs in the first lines to return to a previous command.

For Maxima commands add an ampersand as in `&integrate`, or a blank before the command name.

To search for any string in any help section, use `?string`. This will find lots of places where the string appears. Double click on the command you want to see.

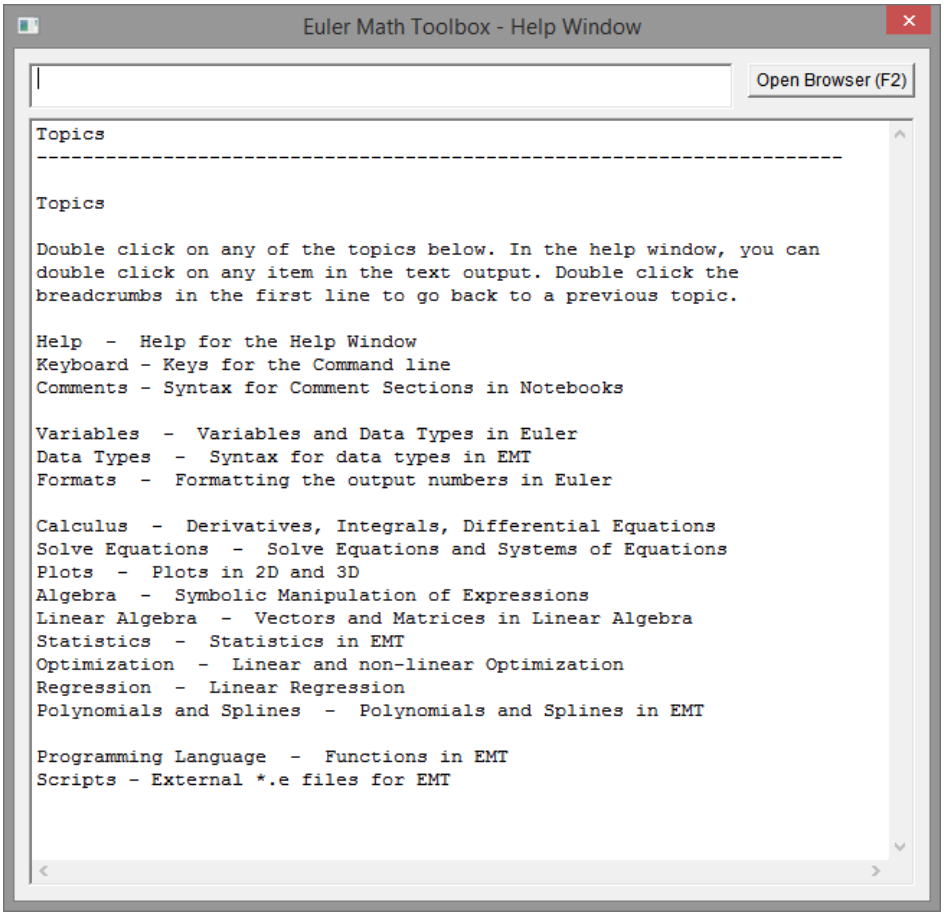



Figure 2.5: The help window

An empty search line displays a number of basic topics. Click on a topic to open a text and examples about the topic. This is not the ideal tool to learn EMT, but rather a reference. For starters, there are the tutorials.

Also observe the status line in the text window while typing a function. After the open bracket of the function, you can find a list of parameters and an explanation there. This works for EMT and Maxima commands. Pressing F1 at this point opens the help window with the help command for the command.

From the help window, you can open the command in the browser. Usually, the reference opens positioned at the links to the command. Select the link you wish to see.

| Further Information | |
|---|---|
| Escape key | Clears the input line or closes the help window. |
| F1 key | Switches between EMT and the help window. |

## 2.9 Euler Files

If you want to develop longer and more complicated programs, it becomes useful to put all function definitions and all commands into external Euler files, also known as Scripts. These files should have the extension `*.e`, and can be loaded into EMT with the `load` command.

Files in the current directory will be found by their name. The current directory is the directory, where the current notebook is loaded from or saved to. Otherwise, use the full path of the file, or include the directory of the file into the EMT search path. See the help on `path` for more details.

The following loads an included script for a package. The script is found, because it is in the internal EMT path.

```
>load interest
 Computes interest rates for investments.
```

You can double click on the load command to open the help window with the help for this script.

If you write your own scripts, save the notebook into one of your directories, e.g., `Euler Files` in the documents folder. Then enter a line containing the load command.

```
>load filename
```

Pressing F9 will then open the internal editor, and pressing F10 the external editor. Either editor lets you edit the content of the file named `filename.e`. Close the internal editor to save the file. For an external editor, it is not necessary to close the editor, but you will have to save your file, of course. Then press return to run the load command and interpret the file.

If the command line does not contain a load command the editors edit the temporary script `EulerTemp.e` in the user directory (F9 or F10). However, other files can be loaded into the editor, or the editor file can be saved to some other file.

The file is saved by the internal editor in the current directory, usually the directory where the notebook is saved. Because of this, you want to save the notebook before you generate a script.

Scripts can contain all commands of EMT, especially definitions of functions. The lines of functions in scripts can optionally start with `$`. Then you can paste the function into a notebook if you want to have it there and edit it directly in the notebook, optionally using the internal editor.

Here is an example of an Euler file.

```
comment
This text is shown when the file loads.
endcomment

/*
Multi-line comment
Some variables:
*/

g:=9.81; // earth gravitational acceleration
t:=2.5; // time
s:=1/2*g*t^2; // height

// Output of s:
s,
```

Save this file into a file named `test.e` as shown above, and load it with the `load` command as follows

```
>load test
This text is shown when the file loads.
                 30.65625
```

The section between `comment` and `endcomment` is printed when the script loads. For other comments use `//`.

## 2.10 Internal and External Editors

The internal editor is just a simple text dialog. It starts with F9. Any external editor starts with F10. It can only be used to edit scripts. The external editor can

be any editor on your system. By default, it is the included Java Editor JE. JE has syntax highlighting for EMT scripts. To be able to use this editor, Java has to be installed, of course. But a good text editor like Notepad++ will do the job too. EMT does not wait for the external editor to finish. You can leave the editor open, switch to Euler, and run the load command there. Do not forget to save any changes in the external editor.

If the line starts with a `function` command, the internal editor will edit the function. After editing the function, a missing `endfunction` will be added. Moreover, the `function` line will be closed with `...` so that the function definition can be interpreted with one stroke of the return key.

```
>function f(x) ...
$ // function body
$endfunction
>
```

On any other line the internal editor will edit the current line. Even multi-line commands ending with `...` can be edited this way.

As shown above, the internal or external editor can be used to edit scripts.

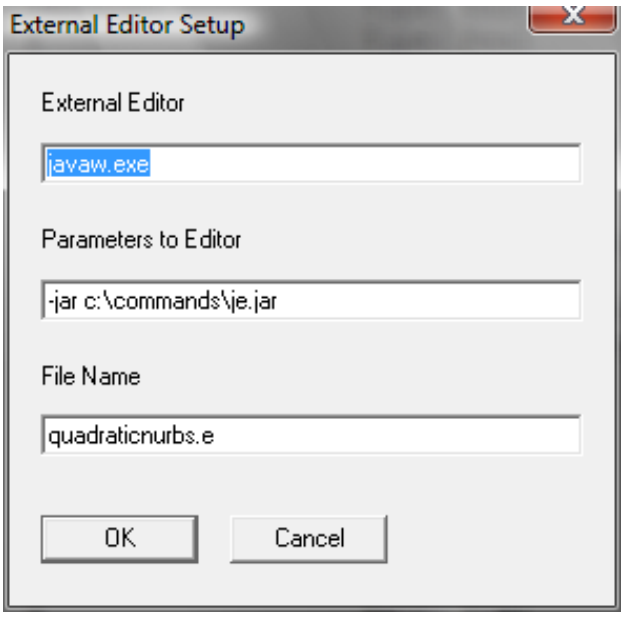



Figure 2.6: Configuration of the external editor

The external editor will find the file in the current directory, i.e., the directory of the current notebook. For other files, use the full path in the load command, and press F10.

# Chapter 3

# Expressions and Plots

## 3.1 Elementary Expressions

Of course, EMT knows all elementary mathematical operations, and evaluates expressions following the usual conventions. Nevertheless, sometimes you should use brackets to avoid errors.

```
>-2^2
   -4
>2^-2
   0.25
>2^3^4 // evaluated as 2^(3^4)
   2.41785163923e+024
>(2^3)^4
   4096
```

Note that in contrast to Matlab (but not Scilab) the power operator evaluates from left to right in EMT. This is the same order that all other math programs follow.

The list of available functions and operators is quite long. Note, that the natural logarithm can be computed with `log` or `ln`, There is also the decimal logarithm `log10`. The square root function is called `sqrt`. To get a list of these functions, open the help in the browser and check the Reference page.

Expressions can produce an error message. E.g., the square root of negative numbers does not exist, unless we convert the number to a complex number.

```
>1/0
   Floating point error!
   Error in:
   1/0 ...
      ^
>sqrt(-1)
   Floating point error!
   Error in sqrt
   Error in:
   sqrt(-1) ...
           ^
>sqrt(complex(-1))
   0+1i
```

It is possible to switch the error messages off. In that case, the result is `NAN` (not a number). The `plot2d` command uses this feature, so that the user does not have to care about the definition set of the plotted function.

```
>errors off; 1/0, errors on;
   NAN
>plot2d("x^x",r=1); // x^x is not defined for negative x
```

Powers with integer exponents are well defined, even for negative numbers. Other powers of negative numbers are not defined. Powers of complex numbers are always defined.

```
>(-2)^3
   -8
>0^0
   1
>(-1)^(1/2)
   ^ defined for positive numbers or integer exponent!
   Use complex numbers?
   Error in ^
   Error in:
   (-1)^(1/2) ...
             ^
```

For very large or small numbers, use the exponential format.

$$2.4\text{e}20 = 2.4 \cdot 10^{20}$$
$$2.4\text{e}{-20} = 2.4 \cdot 10^{-20}$$

```
>1.2e10
   12000000000
>1.2e-10
   1.2e-010
```

EMT knows some other data types, but it handles integers and booleans as reals. The result of a boolean expression is `0` for false and `1` for true. The constants `true` and `false` are defined in EMT. Besides the obvious comparison operators, EMT uses `!=` or `<>` for not equal. Note that equality is checked with `==`. There is a special operator `~=`, which checks for equality with a relative error of `epsilon`. The boolean "and" is `&&`, and the boolean "or" is `||`. Moreover, in condition for `if` statements, `or` and `and` can be used.

```
>1<=2, 2<=1
   1
   0
>1==2 || 2==2
   1
>1+epsilon~=1
   1
```

| Further Information | |
|---|---|
| `2**3` | Alternative for `2^3`. |
| `round` | Rounding etc. |

## 3.2 Accuracy

EMT uses IEEE floating point numbers with about 16 decimal digits. To see this, we evaluate the sine function in $\pi$, and increase the number of digits in the output. In the default format, small numbers are rounded towards 0 (for the display only).

```
>longest sin(pi)
    1.224646799147353e-016
```

The output can be tuned in many ways. The most important formats are the floating point formats of various lengths, and the fractional format.

```
>longformat; 1/3
   0.333333333333
>longestformat; 1/3
   0.3333333333333333
>shortformat; 1/3
   0.333333
>fracformat; 1/3
   1/3
>fracformat(10); 1/[1,2,3;4,5,6;7,8,9]
           1       1/2       1/3
         1/4       1/5       1/6
         1/7       1/8       1/9
>defformat; // set the default
```

We demonstrated the `fracformat` for a matrix. That is very useful for matrix which are known to contain fractional values. The parameter 10 is the number of total places for the output.

For just one output there are operators.

```
>longest 1/3
        0.3333333333333333
>shortest 1/3
       0.333
>fraction 1/3
   1/3
```

There are some special formats and even user defined formats (see `userformat` in the help window). In the following example we use a very narrow, but equally spaced format.

```
>format(5,0); 1:10, longformat;
      1    2    3    4    5    6    7    8    9   10
```

Some formats print row vectors and scalar numbers in a dense way. This is controlled with the command `denseformat(n)`, where `n` is the number of spaces between the elements. The `format` command with two parameters (total space and digits after the comma) sets `denseformat(0)`, which prevents the dense output.

```
>longformat; 1:10 // dense output (the default)
   [1,  2,  3,  4,  5,  6,  7,  8,  9,  10]
>format(10,4); 1:6
      1.0000    2.0000    3.0000    4.0000    5.0000    6.0000
```

There is also a special format for scalars. As you see in the following example, a vector prints shorter numbers than a scalar.

```
>(1:4)*pi
   [3.14159,  6.28319,  9.42478,  12.5664]
>pi
   3.14159265359
```

To disable the special scalar format use `scalarformat(false)`. To set the digits for the scalar format use `setscalarformat(n)`. Have a look at functions like `goodformat` or operators like `longest` to learn to write your own functions for formats and operators. To the function definition, enter `type longest` in the help window.

The internal representation of a number can be printed with `printdual` or `printhex`. The following example shows that 0.1 is not exactly representable in a dual computer.

```
>printdual(0.1)
   1.1001100110011001100110011001100110011001100110011010*2^-4
>printhex(0.1)
   1.999999999999A*16^-1
>longest (0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1)-1
   -1.110223024625157e-016
```

If more accuracy than 16 digits is needed, EMT offers a long accumulator and exact arithmetic. More on this later. Moreover, Maxima has an infinite integer arithmetic, and a long floating point arithmetic.

To round a number use `round`, and to get the integer or fractional part use `floor`.

```
>round(pi,2)
                  3.14
>floor(pi)
                     3
```

Comparisons of real numbers deliver a boolean result, represented in EMT by 1 or 0. The comparison `~=` tests for equality up to an internal accuracy `epsilon`. `==` tests for exact equality.

```
>1/3+2/3 == 1
                    1
>0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 ~= 1
                    1
```

| Further Information | |
|---|---|
| `degformat(flag)` | Print in degrees. |
| `degprint(x)` | Print x in degrees. |
| `&2^1000` | "Infinite" arithmetic in Maxima. |
| `epsilon` | Returns the internal epsilon. |
| `a~=b` | $a$ about equal to $b$. |

## 3.3 Variables

Results can be stored in variables for later use. To assign a value to a variable, EMT has `:=` and `=`. We prefer `:=`, since an assignment like `a=a^2` looks confusing.

```
>g:=9.81; t:=2.5; s:=1/2*g*t^2;
>s
                  30.65625
```

EMT, as well as Maxima, can refer to the previous result with `%`. This should only be used within a single line.

```
>solve("cos(x)-x",1), cos(%)-%
   0.739085133215
   0
```

Variables are cleared, when EMT is restarted, or with the `clear varname` function (or with `remvalue varname`). This will also clear symbolic variables (see below).

EMT variables do not have a type. They can take data of any type. However, the type can be checked with `typeof`, or the functions `isreal`, `iscomplex` etc. To check for a matrix, use `size`.

```
>isreal(1:10)
   1
>iscomplex(I)
   1
>length(2)
   1
>length(1:10)
   10
>length(random(3,2))
   3
```

To list all variables containing a string, use `listvars string`. This can be done as a command or in the help window. Note that many variables are predefined and should not be changed. By the way, `list string` will list all functions with names containing the string.

The constants I, E and pi look like variables, but are functions without parameters. There also variants of this constants like `i%` and `e%` for Maxima.

When multiplying constants with variables, the multiplication sign `*` can often be omitted. In this case, multiplication binds stronger than division. This feature is useful for units (see below). Note that this is contradicting the convention, that there should be a space between the number and the unit.

```
>a:=4; 3a+4a^2
   76
>3/4a, 3/4*a
   0.1875
   3
>36km/2h // 36 km in 2 h are 5 m/s
   5
```

Maxima does not support this kind of abbreviation, and needs multiplication signs in all cases.

## 3.4 Units

Variables in EMT do not have a unit. We assume that physical numbers are in meter, kg and seconds etc. (the so called IS system). To convert to other units, there is the `->` syntax. If used like this units can be called by the normal name, else a `$` has to be appended.

```
>90km/h // result is in m/s
   25
>25->km/h // result is a number
   90
>25->"km/h" // result is a string
   90km/h
```

For another example, we compute the potential energy of a body with 100 kg in 10 m height in Joule. From that, we get the speed it hits the ground in km/h. The gravity constant at sea level is contained in the constant `g$`.

```
>Epot:=g$*100kg*10m
   9806.65
>sqrt(2*Epot/100kg)->" km/h"
   50.41709710009 km/h
```

The explicit units `kg` and `m` are superfluous. These constants are 1.

How many minutes takes the light from sun to earth on average?

```
>AU$/c$->min
   8.316746396769
```

To print a number in a specified format with units use `print`. Alternatively, append the string to the number. For special formats, there is also `printf`, which uses the format conventions of the computer language C (secured against abuse). `printf` in EMT can handle only one number, and there should be only one format such as `%10.5f` for floating points in the format string.

```
>print(1/3,10,2,"km/h")
   0.3333333333km/h
>0.25|" J"
   0.25 J
>printf("%10.2fkm/h",1/3)
         0.33km/h
```

To get detailed help about units or list all units, visit the help window and the topic `units`. The units are defined in an Euler file. You can see this file in the help window using the search term `units.e`.

## 3.5 2D Plots

Discussing a function is a common task in mathematics. For a start, we explain simple plots here. More details can be found in the tutorial about 2D Plots. For a first overview, the function plot2d can plot the following:

- Functions of one variable with adaptive or fixed spacing.
- A list of functions of one variable.
- Two vectors of x- and y-values with line or point style.
- A path with two functions of one variable for x- and y-values.
- A filled area outlined by a closed path with two functions of one variable for x- and y-values or two vectors of data.
- Matrices of x- and y-data with one plot for each corresponding row of the matrix.
- A complex matrix with one path in the complex plane for each row of the matrix.
- The level lines of a function of two variables.
- Bar plots in some styles.

The easiest way to plot a function in EMT is to use an expression, numerical or symbolic, in the variable x. Other variables must be defined globally.

```
>a:=2; b:=3;
>plot2d("a+b*x+x^2",-3,3,title="Cubic p(x)",xl="x",yl="p(x)"):
```

This example produced the graphics in 3.1. The graphics window will only be displayed for a short time. As soon as EMT produces text output, the notebook window will become the top window. To switch to the graphics window or back, press the TAB key. This is also possible if the plot window is hidden and inserted into the notebook window.

Note the : after the plot command. This inserts the current graphics in the notebook window. An alternative is the command insimg(lines), which accepts the number of text lines the plot should cover.

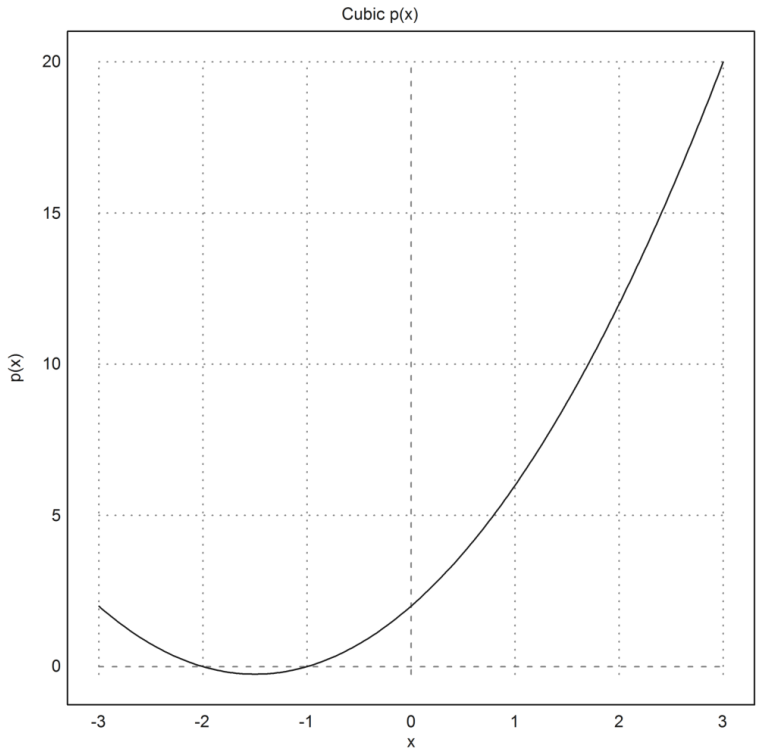



Figure 3.1: Simple plot of a function

We already added a title and x- and y-labels to the plot. By default, the y-label is vertical. These items are assigned arguments for the function `plot2d` in the form `name=value`. Many variables of this kind are possible to configure the plot. As usual, a good starting point is the tutorial about 2D plots.

Plots with expressions or functions can also be interactive, using the parameter `>user` (an abbreviation of `user=true`). Then the user can press the cursor keys to move the plot window, or the `+` or `-` key to zoom in or out. The mouse can be used to mark a new region for the plot. The space bar resets to the default view. Pressing return ends the plot.

```
>plot2d("a+b*x+x^2",a=-3,b=3,>user);
```

Another method to show the plot for a longer time is to use `wait`. This commands waits for a specific time, or until the user presses any key.

```
>plot2d("a+b*x+x^2",-3,3); wait(60);
```

Of course, it is also possible to specify the plot range exactly. For this, set the parameter `a,b,c,d`, or the parameter `r` (for the radius around the a point, by default the origin). `plot2d` has many more parameters. In the following examples, we change the color and thickness of lines. Moreover, we add one plot to another using the boolean argument `>add`. Note that we have entered a multi-line command here, so that the two plot commands evaluate together.

In the example, we plot a function $f(x)$ and the tangent to $f$ in $x = 1$ in one plot.

```
>function f(x) &= x^2*exp(-x); ...
>function df(x) &= diff(expr,x); ...
>function T(x) &= f(1)+df(1)*(x-1); ...
>plot2d("f(x)",a=-1,b=2,c=0,d=1,thickness=2,grid=3); ...
>plot2d("T(x)",>add,thickness=2,color=blue,style="--"); ...
>labelbox(["function","Tangent"],colors=[black,blue],styles=["-","--"]):
```

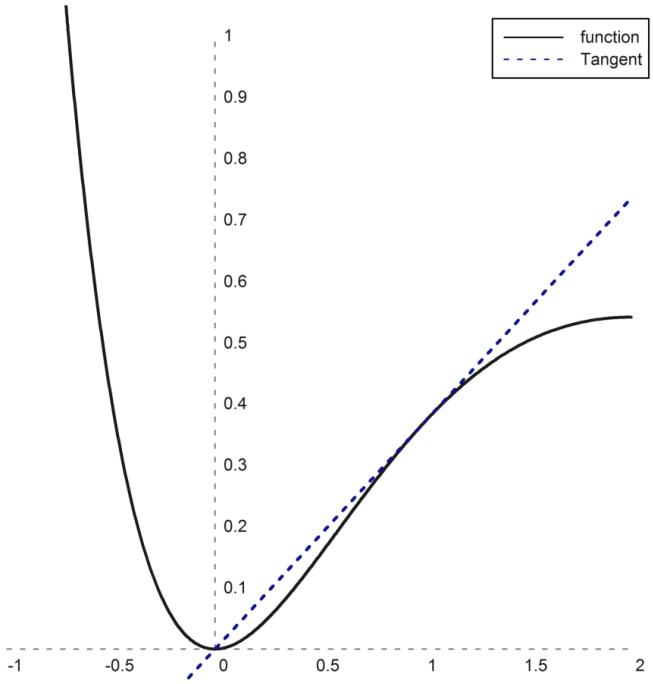



Figure 3.2: Parabola and Tangent

We use a different grid style for this plot using `grid=3`. The `labelbox` adds a description, by default in the upper right corner of the plot.

Besides functions and curves, the plot2d function can plot point clouds, distribution of data in bar plot form, and a lot more. Some of these plots are discussed in later chapters of this introduction. For a complete overview have a look at the tutorials. Here, we have to restrict ourselves to a few examples.

To plot the binomial distribution for $n = 20$ and $p = 0.4$, we first use the matrix language of EMT to compute the values. Then we call plot2d with two vectors, one vector of x-coordinates, and another vector of y-coordinates. We add the points to the plot with the parameters >points and >add. Alternatively, one plot with >addpoints would do the same.

```
>n=20; k=0:20; p=0.4; y=bin(n,k)*p^k*(1-p)^(n-k); ...
>plot2d(k,y,title="Binomial Distribution with p=0.4");  ...
>plot2d(k,y,>points,>add):
```

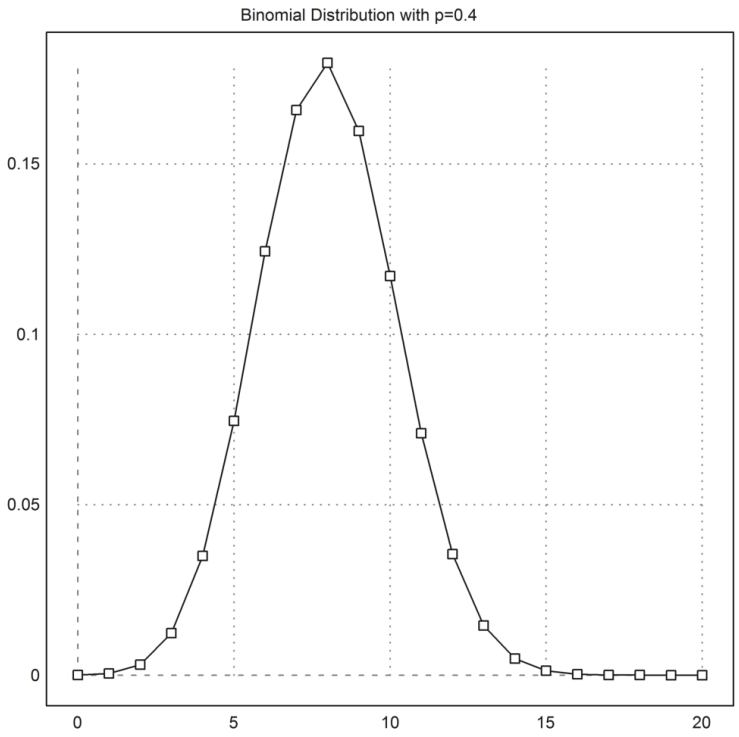
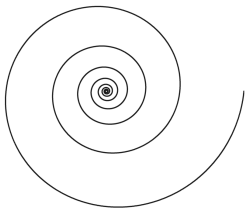



Figure 3.3: Line and point plot of coordinate vectors

Since we can now plot vectors of x- and y-coordinates we can also plot spirals. The following is the logarithmic spiral.

```
>phi=linspace(-12pi,4pi,10000); r=exp(phi/10);
>plot2d(r*cos(phi),r*sin(phi),r=3.2,grid=0,thickness=2):
```



Note that `r*cos(phi)` returns a vector of values, combining the values in `r` and `phi`.

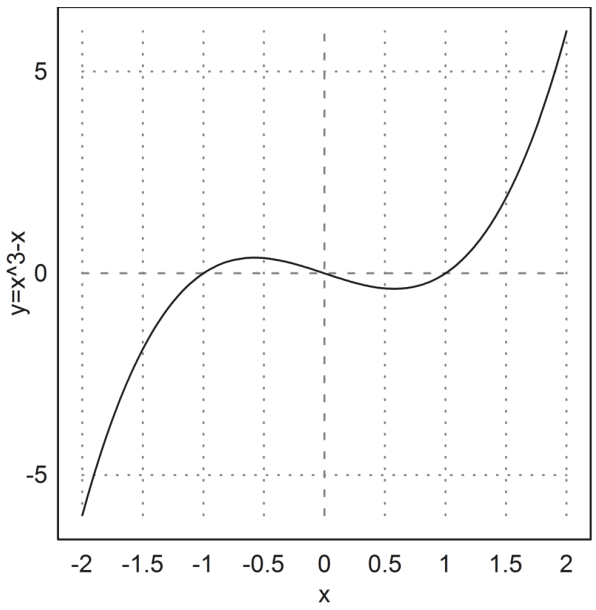



Figure 3.4: Bigger font for small graphics

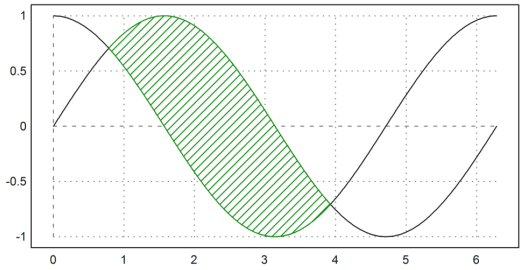
For the small graphics on this page, we need to increase the thickness of the lines. This can be done globally too. Moreover, for many plots the fonts are too small. We can set the font in EMT for such small plots.

```
>setfont(10pt,8cm); window(100,0,1000,900);
>plot2d("x^3-x",xl="x",yl="y=x^3-x"):
>reset;
```

The example does also demonstrate the `window` command. It can be used to set the portion of the screen that a graphics covers. You need to make sure that there is enough room for the labels and the title. The screen coordinates range from 0 to 1024 in each direction. In the example, we took away the room for the title. The command `reset` resets to the default.

In the following example, we take a different aspect ratio. We plot the sine and the cosine and fill an area between the curves. This is a bit tricky. We need to define a curve which goes around that area. The x-coordinates of the curve will be formed by a vector `t` and the inverted vector `t`, since we need to go left and the right. Likewise, the y-coordinates will be `sin(t)` and the inverted vector `cos(t)`.

```
>aspect(16/9);
>plot2d("sin(x)",0,2pi); plot2d("cos(x)",>add);
>t=linspace(pi/4,5pi/4,1000);
>plot2d(t|fliplr(t),sin(t)|fliplr(cos(t)),>filled,>add,style="/"):
```




## 3.6 Numerical Analysis

To solve expressions like

$$x^x = 2$$

needs numerical methods. There is no symbolic solution, and Maxima just returns the equation.

```
>&solve(x^x=2)
```

$$\left[x = 2\right]$$

There are many functions in EMT to solve such equations. All of them accept expressions (besides names of functions). The following example solves $x^x = 2$ using the stable bisection method and the faster secant method which is used by `solve`. The target value is 0 by default, but it can also be specified as `y`.

```
>bisect("x^x",1,2,y=2)
   1.55961046946
>solve("cos(x)-x",1)
   0.739085133215
```

The secant method `secant` can also be started with two start values. But it does not guarantee to stay in the interval between these values. To help out there is `secantin` which never goes outside the interval and is still reasonably fast.

```
>secantin("sqrt(x)*exp(-x)",0,0.4,y=0.3)
   0.112769901579
```

The even faster Newton method needs a derivative. We can use Maxima to compute this derivative for us. The function `mxmnewton` calls Maxima automatically to compute the derivative.

```
>newton("x^2-2","2x",1)
   1.41421356237
>expr&=x^2-2; newton(expr,&diff(expr,x),1)
   1.41421356237
>mxmnewton("x^2-2",1)
   1.41421356237
```

Not all integrals can be solved by symbolic means. So EMT has functions for numerical integration and differentiation. To integrate, either use the robust and fast Gauß integration, or the exact Romberg method. Of course, the very stable Simpson method is also available. The Gauß method is exact up to polynomials of degree 19, and uses only 10 evaluations of the function. Moreover, the interval can be subdivided into a number of subintervals to increase the accuracy.

```
>fraction gauss("x^19",0,1)
   1/20
>gauss("exp(-x^2/2)/sqrt(2pi)",-20,20,20) // 20 subintervals
   1
>gauss("x^x",1,2)
```

```
   2.05044623453
>romberg("x^x",1,2)
   2.05044623453
>simpson("x^x",1,2)
   2.05044623596
```

The basic function `integrate` uses an adaptive integration, and finds many integrals with maximal accuracy.

```
>longest integrate("1/sqrt(pi)*exp(-x^2)",-10,10)
                          1
```

Often Maxima can compute an exact form of the integral.

```
>romberg("x*exp(x)",-1,1)
   0.735758882343
>&:integrate(x*exp(x),x,-1,1)
   0.735758882343
```

In principle, numerical differentiation has a limited accuracy, especially with higher derivatives. Maxima can be used for exact differentiation.

```
>diff("x^20/20",1)
   0.999999999999
>diff("x^20/20",1,2)
   18.9999926252
>&diffat(x^20/20,x=1,2)
                                    19
```

To find the points of extrema of a function use `fmin` or `fmax`. There is also a function `fextrema`, delivering two values, a list of minima, and a list of maxima.

```
>fmin("x^3-x",0,1)
   0.57735026525
>fmax("x^3-x",-1,0)
   -0.577350272719
>{xmin,xmax}:=fextrema("x^3-x",-1,1); xmin, xmax,
   0.577350265249
   -0.577350272196
```

## 3.7 Definition of Functions

Since there will be a later chapter on programming in EMT, you will find here the basics only. To program very simple functions, there are one-line functions in EMT.

```
>function f(x) := 1/(x^2+1)
>f(2)
   0.2
>plot2d("f",-3,3,r=5);
```

Most EMT functions, which accept expressions like `plot2d` also accept functions. You simply pass the function by name contained in a string.

```
>x0:=solve("f",1,y=0.4), f(x0)
   1.224744871392
   0.4
```

Of course, a function can have more than one parameter. The parameters can have default values. These default values can be changed with `assigned arguments`. Moreover, global variables are visible from within one-line functions.

```
>function f(x) := a*x^2+x+1
>a:=4; f(1) // use the global a
   6
>function g(x,a=4) := a*x^2+x+1
>g(2)
   19
>g(1,5) // overwrite a
   7
>g(1,a=6)
   8
```

A problem arises, if `plot2d` is used to plot a function with more than one parameter. `plot2d` can handle this by passing additional arguments to the function to be plotted. Those arguments are appended to the call of `plot2d` after a semicolon `;`. We call them semicolon arguments.

```
>function f(x,a) := exp(-x^2)*sin(a*x)
>plot2d("f",0,10;9); // with a=9
```

There is the rule that assigned arguments for `plot2d` (like the plot range) must be given after the semicolon parameters. So the order is: normal arguments, semicolon arguments, assigned arguments.

Another option is to use a collection to pass the function name and the additional parameters to the algorithm which calls the function. Collections are a data type in EMT collecting other elementary data of any type and have the syntax `{{a,b,c}}`. In this introduction, we will not explain collections in detail. But for plots, the following example will show how they can be used to pass the extra parameters.

```
>function f(x,a) := exp(-x^2)*sin(a*x)
>plot2d({{"f",9}},0,10): // also with a=9
```

This use of collections works for expressions too. For an example, let us compute the zero of $x^2 - a$ for the variable $a$ using the bisection method. But in this case, the additional parameters for the expression (besides `x`, `y` etc.) must be named.

```
>bisect({{"x^2-a",a=3}},1,2)
   1.73205080757
```

More complex EMT functions can be defined using the full syntax for EMT programs and the `function` and `endfunction` commands. This is necessary for functions using control statements. Those function start with `function`, and end with `endfunction`. In the simplest form, enter the function line by line, and press the Esc key to end the definition.

```
>function f(x)
$ if x>0 then return x
$ else return x^2
$ endif
$endfunction
>f(-2), f(2),
                        4
                        2
```

The notebook will be in a special function input state when a function is entered. This is natural, since lines in a function must be entered one by one. You can no longer go up or down with the cursor keys. To exit this state, enter `endfunction` or press the escape key in an empty line.

However, it is easier to use the internal editor to edit such a function in the notebook. Press F9 in the `function` line to start this editor. `endfunction` will be added, if it is missing. Moreover, the `function` line will end with `...` so that the function can be interpreted with one stroke of the return key.

Note, that the `if` command in the example above must end with `endif`. `return` ends the function and returns the value. If it is missing, a special string `none` is returned, which does not print. The function is then called a procedure. For other control structures, see the chapter about programming.

There is another mode to enter a function. For this press `Ctrl-Return` after `function` line. You can then enter the function by inserting new lines, inserting lines between lines (`Ctrl-Return`) or deleting lines (`Alt-Back`). You can also walk up or down in the function with the cursor. To end this mode, press `Return` in the last line.

Such a function might look as follows. The `...` in the first line evaluates the function if `Return` is pressed in this line. Press `Ctrl-Return` to edit the function, or simply click into the function body. Exit editing the function with Return.

```
>function f(x) ...
$  if x>0 then return x^2;
$  else return x^3
$  endif;
$endfunction
```

Tabulator characters will be converted to spaces in function definitions. Pressing the tabulator key will just insert two spaces in the edit mode. Note that exports of functions contain `$` characters in front of each line. Thus they can be pasted back into EMT notebooks.

The mode to edit functions can be entered at any time for an existing function by clicking into the function. This mode may be more convenient than the internal editor, which you can start with F9 in the `function` line.

For an example, let us compute the distance of the horizon depending on the height of the viewer, and plot a graph for this. The formula can be derived from the theorem of Pythagoras applied to the triangle formed by the viewer, the horizon point, and the center of the earth.

```
>function horizon(h) := sqrt(2*rEarth$*h+h^2)
>horizon(30)/km$
   19.5296190439
>plot2d("horizon(x)/km$",0,1000,title="Distance of Horizon");
>xlabel("Height (m)"); ylabel("Distance (km)");
```

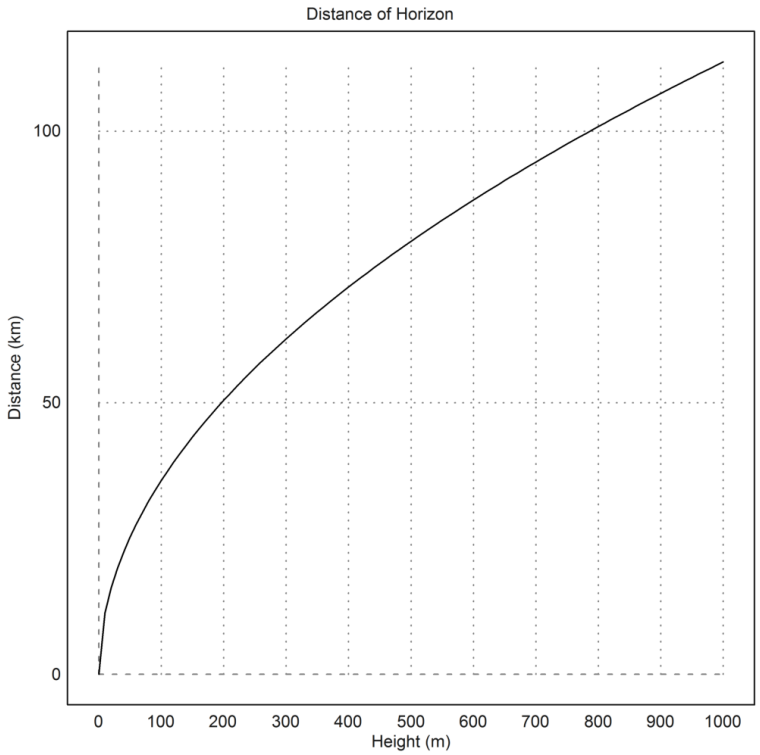



Figure 3.5: Visibility in km vs. height in m

We can now solve the inverse problem. At what height is the distance of the horizon 80km?

```
>secant("horizon(x)",0,100,y=80km$)
   503.381806208
```

To make a function of any expression in `x`, there is a special trick. If the expression is stored in the string variable `str`, the syntax `@str` will be replaced with the content of the expression.

```
>expr := "x^3-x";
>function f(x) := @expr
>type f
   function f (x)
   useglobal;  return  x^3-x
   endfunction
>f(2)
                    6
```

We have used type to make the function definition visible. It does indeed contain the correct expression. You can also see that it contains the command useglobal. This command allows a function access to any global variable.

If the function uses a symbolic expression it is possible to define it as a symbolic function. The following lines of code are similar to the previous example, but Maxima is involved to define the expression and the function. The function exists in Maxima and in the numerical part of EMT.

```
>expr &= x^3-x;
>function f(x) &= expr
```

$$x^3 - x$$

```
>&diff(f(x),x)
```

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

```
>&f(a+1)
```

$$(a + 1)^3 - a - 1$$

```
>f(1:5)
   [0,  6,  24,  60,  120]
```

The last command shows that a symbolic function can also be evaluated numerically. A symbolic function defined with &= does also work like a numerical function.

Note that it may happen that a function can only be evaluated numerically. E.g., the function may contain a numerical integration. In this case, there is no point in using a symbolic function simply define the one-line function with :=.

On the other hand, some functions can only be evaluated symbolically. E.g., the function computes a symbolic derivative or a special function of Maxima which is not implemented in EMT. These functions must be defined with &&=. In the section about programs in EMT we will give more details on this.

It is also possible to define elaborate functions in Maxima. More on this later.

# Chapter 4

# Maxima

## 4.1 Introduction

Maxima is an algebraic, symbolic system, which can be called from EMT. You can find a lot of examples and tutorials for Maxima in the Net. E.g., there is "The Maxima Book" on SourceForge. Just for another example, the PDF files by Gilberto E. Urroz are very nice too. For German speakers, there is the comprehensive book by Wilhelm Haager.

The books and tutorials use the original syntax of Maxima, of course. EMT uses a somewhat different syntax which is much closer to the syntax of EMT. You can use the original syntax in the direct mode in EMT. However, it is not difficult to translate the code to the compatibility mode or to symbolic expressions.

After the first call, EMT will start an instance of Maxima in the background, and communicate with it through pipes. By default, Maxima starts, whenever a new notebook is loaded, or EMT is restarted. There is a global switch to start Maxima later, when it is first used. Alternatively, start Maxima with the command `mxmstart` and stop it with `mxmstop`.

There are several ways to use Maxima from EMT.

- Directly in the compatibility mode. The output of Maxima is printed directly to the notebook in formula output. The syntax is adapted to the EMT syntax. To use this mode, start the command line with `>::` followed by a blank.
- Directly in the direct mode. The output is also printed to the notebook window, but the syntax is the original syntax of Maxima. Start the command line with `>:::` (or with `>:` if the switch to allow direct mode is on).

- In Maxima mode. This starts and ends with the command `maximamode`. In this mode, all commands are sent directly to Maxima, and EMT commands need to start with `euler`. It is available using direct or compatibility mode.
- In symbolic expressions of the form `&expression`. The expressions are evaluated in Maxima and the return string can then be used in EMT as an expression.
- In symbolic functions. These functions are one-line functions defined with `&=` and evaluate the function body at compile time. The functions are visible in EMT and in Maxima.
- Maxima can also be used at compile time in functions. The syntax for this is `&:expression`. The expression is evaluated in Maxima and the result is inserted into the function as if the user had entered the expression directly.
- Via mxm-Functions. There are a lot of EMT functions, which use Maxima at run time. An example is the Newton method `mxmnewton`.

We now explain these modes in detail.

## 4.2 Direct Input of Maxima Commands

The preferred method for direct input of Maxima commands is the compatibility mode with the prompt `>::`. In this mode, EMT will do a lot of formatting to the command before it is sent to Maxima. Instead of the Maxima command separations `;` and `$`, you can use the EMT separators `,` and `;`. The Maxima assignment to variables is replaced by `:=`. Maxima functions can be written with `function ...`, and flags are appended with `|`. We demonstrate these changes below.

The direct mode with the `>:::` prompt (or `:` if the flag to allow this is on) still does a little bit of editing. You can add comments with `//` just like in EMT commands. Moreover, you need not finish the command with `$` or `;`, since EMT will add `;` if it is missing to suppress the output. We will use the compatibility mode in this document. For the direct mode, see the Maxima documentation.

The Maxima output is printed to the EMT notebook just like EMT output. However, by default Maxima uses a 2D non-linear output to format formulas. The numbering of output by Maxima is removed by default, since this makes no sense in a notebook environment. Use variables to hold values instead. Only within the same line, use `%` to refer to the previous result.

```
>:: (a+b)^2/(a-b)
```

$$\frac{(b + a)^2}{a - b}$$

```
>:: res := expand((a+b)^5) // expand a product
```

$$b^5 + 5\,a\,b^4 + 10\,a^2\,b^3 + 10\,a^3\,b^2 + 5\,a^4\,b + a^5$$

```
>:: factor(diff(res,a)) // factor a product
```

$$5\,(b + a)^4$$

As you see, Maxima uses symbolic variables.

> Symbolic variables must not have a value. This behavior is in contrast to EMT variables, which cannot be used, *unless* they have a value.

By the way, the value of a variable can be removed with the Maxima command `remvalue variable`.

If you want to use EMT as a Maxima interface primarily, you can switch to Maxima mode. In this mode, Maxima commands are entered just like EMT commands after the prompt `>`. The command `maximamode` will toggle the compatibility mode by default. For more control use `maximamode on` or `maximamode direct`. There is also a menu option to start EMT in one of the Maxima modes whenever it starts. In Maxima mode, `euler ...` will send a command to the EMT sub-system.

```
>maximamode on
   Maxima mode is on (compatibility mode)
>1/3
```

$$\frac{1}{3}$$

```
>euler 1/3
```

```
      0.3333333333333
>maximamode off
   Maxima mode is off
```

Maxima uses an "infinite" arithmetic for integers and fractions. It will keep all digits of a computation. Floating point numbers are only used on specific request.

```
>:: 1+1/3+1/7

                                        31
                                        --
                                        21

>:: 100!/(50!*50!)

                    100891344545564193334812497256
```

The last example works in EMT too, but larger factorials cause an overflow in EMT.

```
>100!/(50!*50!)
      1.008913445456e+029
```

For the binomial function, EMT can use the function `bin`, which uses a different algorithm, and works for large numbers. Moreover it is very much faster than any method in Maxima. The function `float` converts a Maxima expression to the same type of floating numbers as in EMT.

```
>:: float(1000!/(500!*500!)) // convert to IEEE floating point

                          2.7028824094543655E+299

>bin(1000,500) // Euler function for binomials
      2.702882409454e+299
```

There is also a floating point arithmetic with an adjustable number of digits in Maxima. To use it, we have to assign the number of digits to the variable `fpprec` (floating point precision).

```
>:: fpprec:=40; bfloat(sqrt(2))

              1.4142135623730950488016887242096980785 7b0
```

In compatibility mode (prompt `>::`), variables in Maxima are assigned with a `:=`. To separate commands in the compatibility mode, use a comma as in EMT, and to separate without printing, use a semicolon `;`.

```
>:: expr := expand((1+x)^5);
>:: factor(expr)
```

$$(x + 1)^5$$

```
>:: expr := expr*expr, factor(expr)
```

$$(x^5 + 5\,x^4 + 10\,x^3 + 10\,x^2 + 5\,x + 1)^2$$

$$(x + 1)^{10}$$

If a Maxima expression is too long for one line, it can be spread over many lines. Use `...` just for these multi-lines.

```
>:: expand((1+x+x^2+x^3) * ...
>::   (1-x))
```

$$(1 - x)^4$$

The names of some Maxima functions can be appended to an expression, and work like flags for the evaluation function. E.g., `expand` can be used this way. Also, various simplification hints can be given, like `ratsimp`, which simplifies rational expressions. For most of these flags, there is also a function version, which might be preferred.

```
>:: (x+4)^2/((x+4)*(x+3)) | expand, % | ratsimp
```

$$\frac{x^2}{x^2 + 7\,x + 12} + \frac{8\,x}{x^2 + 7\,x + 12} + \frac{16}{x^2 + 7\,x + 12}$$

```
                                    x + 4
                                    -----
                                    x + 3
>:: log(16)/log(2) | radcan

                                      4
```

Trigonometric simplification with `trigsimp` does not work as a flag, and needs to be used as a function.

```
>:: sin(x)^2+cos(x)^2, trigsimp(%)

                                  2        2
                                sin (x) + cos (x)


                                      1
>:: trigreduce(sin(x)^3)

                                3 sin(x) - sin(3 x)
                                -------------------
                                        4
```

## 4.3 Symbolic Expressions

We already showed many examples of symbolic expressions. Those are the preferred way to use Maxima in EMT, allowing a seamless integration of symbolic mathematics in the numerical part of EMT.

> Symbolic expressions are strings in EMT, which are evaluated in Maxima.

The result is a string, which can in turn be used in EMT, where Euler accepts a string. e.g., for an expression. The syntax is `&expr`. EMT scans the brackets in the expression, and stops at commas or blanks. Alternatively, use `&"expr"` to make sure your expression is scanned correctly or to help EMT.

```
>&diff(x^4-2*x^3+x^2-x+5,x)
```

$$4\,x^3 - 6\,x^2 + 2\,x - 1$$

```
>plot2d(&diff(x^4-2*x^3+x^2-x+5,x),r=2);
>solve(&diff(x^4-2*x^3+x^2-x+5,x),1)
   1.2606898534
```

Symbolic expressions are strings with a special flag to mark them as symbolic. As you see, the symbolic expression defined above is printed by Maxima in 2D as a formula. The reason for this is the symbolic flag in the string which is returned by Maxima after evaluating the expression.

To make sure this is understood, let us summarize how symbolic expressions work.

- When the EMT interpreter finds a symbolic expression `&...` it passes a formatted version to Maxima for evaluation.
- The Maxima output is then scanned by EMT to find the result of the evaluation. This result is stored in a string in EMT with the symbolic flag.
- The string can be used like any other string. E.g., it can work as an expression, stored into variables, or print as output. The print would be done by Maxima due to the symbolic flag.

The `&=` syntax defines a variable in EMT which contains a symbolic string. It does also define a variable with the same name in Maxima.

```
>expr &= diff(x^x,x)
```

$$x^x\,(\log(x) + 1)$$

```
>&solve(expr=0)[1] // use expr in Maxima
```

$$x = E^{-1}$$

```
>solve(expr,1) // use expr in EMT
 0.367879441171
>1/E
 0.367879441171
```

If you just want to store a symbolic expression in EMT, you can use := and a symbolic expression on the right hand side. In the example, we demonstrate how to generate a more complex expression wth Maxima features.

```
>f := &sum(k*cos(k*x),k,1,5)
```

$$5\cos(5x) + 4\cos(4x) + 3\cos(3x) + 2\cos(2x) + \cos(x)$$

```
>plot2d(f,0,2pi):
```

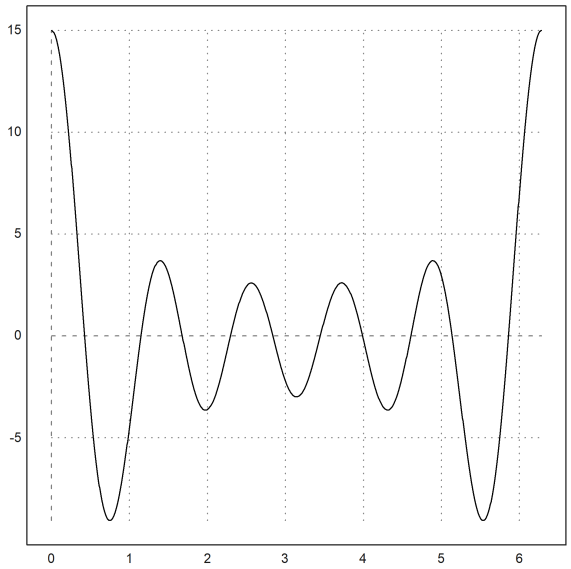



Figure 4.1: Fourier Series generated by Maxima

Maxima flags can be used in symbolic expressions too.

```
>&sum(k^2,k,1,n)|simpsum|factor
```

$$\frac{n\,(n+1)\,(2n+1)}{6}$$

If you want to use + for string concatenation define the symbolic strings with quotes. Note that the symbolic flag is lost after the concatenation.

```
>&"diff(x^x,x)"+"+a"
   x^x*(log(x)+1)+a
```

You can use use an EMT expression in Maxima or a symbolic expression without defining it as a variable in Maxima. The necessary string manipulations are done automatically with the syntax `@str`. This syntax pastes the content of the string variable `str` into the symbolic expression.

```
>expr:="x^3";
>&diff(@expr,x)
                                      2
                                   3 x
```

```
>function f(x) := @expr
>f(4)
   64
```

As you see above, the syntax does also work in numerical EMT functions.

To evaluate expressions in EMT, we already mentioned the syntax `expr(...)`. It evaluates the expression with the values assigned to the variable `x`, and `y`, `z` in case of further values. There are also assigned variables, e.g. `expr(3,a=5)`. This can all be used for symbolic expressions, of course.

```
>expr &= sum(x^k/(n*k),k,1,10)
          10     9     8     7     6     5     4     3     2
         x      x     x     x     x     x     x     x     x     x
         ---- + --- + --- + --- + --- + --- + --- + --- + --- + -
         10 n   9 n   8 n   7 n   6 n   5 n   4 n   3 n   2 n   n
```

```
>n=2; expr(0.4)
 0.255409795007
>expr(0.4,n=2)
 0.255409795007
```

To evaluate an expression in Maxima, there are several methods. The keyword `with` that EMT adds to the symbolic syntax works like `at` in Maxima. It can be used for one or more variables. The simple `|` does also work. And there is a more complicated substitution.

```
>expr &= a*x^2;
>&expr with x=4
```

$$16\,a$$

```
>&expr with [x=4,a=2]
```

$$32$$

```
>&expr|x=4|a=2
```

$$32$$

```
>&subst(x+1,x,expr)
```

$$a\,(x+1)^2$$

Often we want to define a value in EMT and Maxima at the same time. Using `&=` would enter the value as a string (i.e. a symbolic expression) in EMT. We can use `&:=` for this. This is useful for non-symbolic matrices.

```
>M &:= [1,2;3,4]
                1             2
                3             4
>&eigenvalues(M), &"%[1]"()
```

$$\left[\left[\frac{5-\sqrt{33}}{2},\ \frac{\sqrt{33}+5}{2}\right],\ [1,\ 1]\right]$$

```
   [-0.372281,  5.37228]
>eigenvalues(M)
   [ -0.372281+0i ,  5.37228+0i  ]
```

Note that the result of the function `&eigenvalues` is a vector of vectors. In this example, we evaluated the first element of this vector (`&"%[1]"` in EMT (which contains the eigenvalues) with the usual method to evaluate an expression (`expr()`). This only looks complicated at first sight.

## 4.4 Differentiation and Integration

Calculus is a typical application for symbolic computations. Maxima can compute the derivative of all its functions, and knows also many integrals.

```
>&diff(x^5,x)
```

$$5\,x^4$$

```
>&diff(x^5,x,2) // second derivative
```

$$20\,x^3$$

```
>&integrate(x^5,x)
```

$$\frac{x^6}{6}$$

```
>&integrate(2/(x+1),x,1,2)
```

$$2\,(\log(3)-\log(2))$$

Of course, we can also compute the anti-derivative first, and then insert the limits. Use `with` to evaluate an expression at some values.

```
>s &= integrate(t/(1+t^2),t)
```

$$\frac{\log(t^2+1)}{2}$$

```
>&(s with t=2)-(s with t=1), %()
```

$$\frac{\log(5)}{2}-\frac{\log(2)}{2}$$

```
   0.458145365937
```

In connection with some integrals, Maxima may ask questions to the user. By default, these questions are automatically answered.

```
>&integrate(x^n,x)
 Answering "Is n equal to -1?" with "no"

                                  n + 1
                                 x
                                 ------
                                 n + 1
```

To avoid this without the default automatic answering, it is possible to set assumptions for the variables.

```
>&assume(n>0); &integrate(x^n,x)

                                   n + 1
                                  x
                                  ------
                                  n + 1

>&forget(n>0);
```

Maxima uses functions either as verbs or nouns. The difference is, that verbs evaluate, and nouns do not. To enter differential equations, we must use the differentiation `diff` as a noun. To do this, we have to enter `'diff`.

```
>eq &= 'diff(y,x) = y + x^2

                                 dy        2
                                 -- = y + x
                                 dx

>&ode2(eq,y,x)

                          2             - x        x
                 y = ((- x  - 2 x - 2) E    + %c) E
```

| Further Information | |
|---|---|
| `diffat(sin(x)*exp(x),x=3)` | Differentiate and evaluate at $x = 3$. |

## 4.5 Symbolic Functions

In Maxima, we can define one line functions just like in EMT. To do this seamlessly there are symbolic functions. These functions can be defined only for Maxima (purely symbolic with &&=) or for the numerical part of EMT too (with &=).

```
>function f(x) &= integrate(x/(x^3+1),x)
```

$$\frac{\log(x^2 - x + 1)}{6} + \frac{\operatorname{atan}\left(\frac{2x-1}{\sqrt{3}}\right)}{\sqrt{3}} - \frac{\log(x+1)}{3}$$

```
>f(4) // used numerically
   0.657867619545
>&factor(f(4)) // used symbolically
```

$$\frac{6\operatorname{atan}\left(\frac{7}{\sqrt{3}}\right) + \sqrt{3}\log(13) - 2\sqrt{3}\log(5)}{2^{3/2}\,3}$$

As you see, the body of the function is evaluated before the function is compiled. These functions can be used numerically or symbolically.

Purely symbolic functions do not evaluate the body prior to the definition. Of course, they cannot be evaluated numerically. Here is an example, which computes

$$D(u) = \frac{d^2}{dx^2}u + \frac{d^2}{dy^2}u$$

for an expression u which depends on the variables x and y. We apply it to the real part of an analytic function. Since this is a harmonic function the result must be 0.

```
>function D(u) &&= diff(u,x,2)+diff(u,y,2)
```

$$\text{diff}(u, y, 2) + \text{diff}(u, x, 2)$$

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

$$y^4 - 6x^2y^2 + x^4$$

$$0$$

Of course, it is also possible to use the direct input to define functions in Maxima. These functions will be known to Maxima only, of course. You can use the compatibility syntax `:=` for such definitions. In this case, the body of the function is not evaluated before the definition. To evaluate the body, use the Maxima command `define`.

```
>:: function D(u) := diff(u,x)
```

$$D(u) := \text{diff}(u, x)$$

```
>:: D(x^4+5*x)
```

$$4x^3 + 5$$

```
>:: define(f(x),diff(x^3,x))
```

$$f(x) := 3x^2$$

For the Maxima experts, the same is possible in the non-compatible direct mode (`>:::  ...`) using the book syntax of Maxima. We do not explain that here.

### Example

The `solve` command of Maxima finds exact solutions for expressions in one or more variables. We use it to compute the area between two functions, as shown in the plot.

```
>function f(x) &= (x^3-9*x^2+2)/3
```

$$\frac{x^3 - 9x^2 + 2}{3}$$

```
>function g(x) &= 5/3*x+2/3
```

$$\frac{5\,x}{3}+\frac{2}{3}$$

```
>plot2d([&f(x),&g(x)],0,10):
```

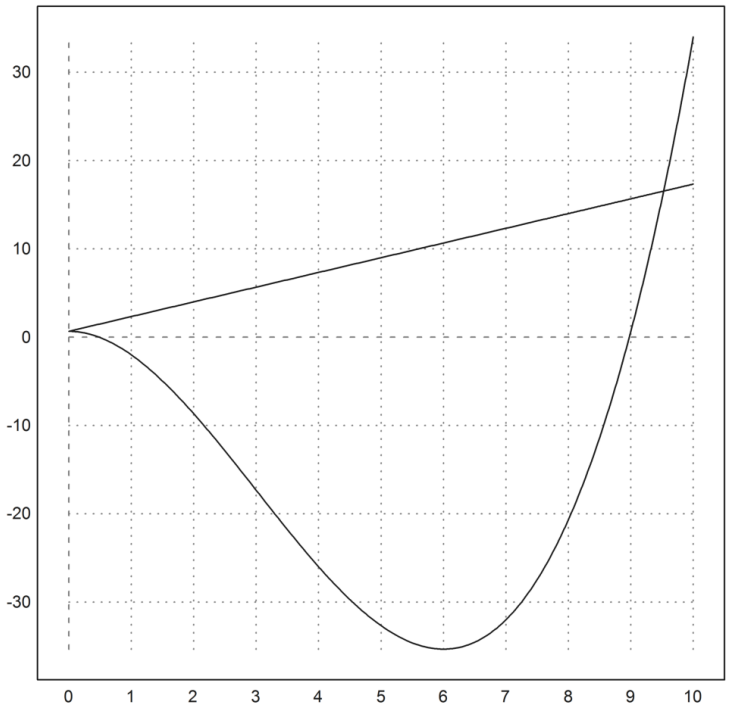



Figure 4.2: Two functions

For `plot2d` we used a vector of expression to plot two functions into one plot. We clearly see that the functions intersect at about $x = 9.5$. We now compute this intersection symbolically and perform the integration.

```
>sol &= solve(f(x)=g(x),x)
```

$$\left[x=\frac{9-\sqrt{101}}{2},\ x=\frac{\sqrt{101}+9}{2},\ x=0\right]$$

```
>&integrate(g(x)-f(x),x,0,x with sol[2]), &float(%)
```

$$\frac{3 \, 101^{3/2} + 3047}{24}$$

$$253.8380130499846$$

The expression `x with sol[2]` is a simple way to extract the second solution.

For a comparison, we do the same with numerical integration. Of course, in general this will be the way to go since most real world functions do not have a definite integral.

```
>b=solve("g(x)-f(x)",10)
   9.52493781056
>longest integrate("g(x)-f(x)",0,b)
         253.8380130499845
```

Calling Maxima in numerical functions at runtime is not very fast. It is far better to include the Maxima result in EMT functions. This is very easily done by calling Maxima at compile time. In the definition of an EMT function, use `&:"..."` to evaluate the string in Maxima, and paste the result into the EMT function.

```
>function f(x) ...
$  if x<0 then return &:"diff(x^2*exp(-x^2),x)"
$  else return &:"diff(x^2*exp(x^2),x)"
$  endif;
$endfunction
>type f
   function f (x)
     if x<0 then return 2*x*E^-x^2-2*x^3*E^-x^2
     else return 2*x^3*E^x^2+2*x*E^x^2
     endif;
   endfunction
```

The internal definition of the function `f` can be printed with `type` (more on this in the section about programming). As you see, the function contains the correct expression for the derivatives.

If an expression in a string is needed there is the syntax `"&:..."`.

```
>function f(x) := integrate("&:diff(x^x,x)",1,2)
>type f
   function f (x)
   useglobal; return integrate("x^x*(log(x)+1)",1,2)
   endfunction
```

## 4.6 Exchanging Values between Maxima and Euler

Sending direct commands to Maxima keeps the results of EMT and Maxima separated from each other. Moreover, the output of Maxima is in a 2D format, which cannot be cut and paste to EMT. Additionally, variables in Maxima and in EMT are separated from each other.

But we can exchange results and values between EMT and Maxima easily even if we do not use symbolic expressions.

A simple way to set variables for both worlds with one command is the `&:=` assignment. This should be used for shorter matrices only since the value of the matrix is printed and then evaluated by Maxima.

```
>v &:= 1:10;
>&v
```

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

To set variables in Maxima which are defined in EMT, we can use `mxmset`. This should be used for big matrices. It will set a variable with the same name and value in Maxima. The inverse is `mxmget`, which is almost identical to `mxmeval`.

```
>A:=[1,2;3,4];
>mxmset(A); // set the variable A in Maxima
>B:=mxmget("invert(A)") // inverse with Maxima transferred to EMT
             -2             1
            1.5          -0.5
>A.B
              1             0
              0             1
```

Note that a variable in Maxima generated by `mxmset` is not known in EMT. Moreover, it is always a variable of type `float`. In the example `5/6` is parsed to a double variable in EMT already. Note that `mxmget` evaluates the sum in Maxima and transfers the result to EMT which will be a value in EMT, not a symbolic string.

```
>mxmset("x",5/6); mxmget("sum(x^k,k,1,10)")
 4.19247208555
>&sum(x^k,k,1,10)

                              4.192472085550772

>x
 Variable x not found!
 Error in:
 x ...
  ^
>sum((5/6)^(1:10))
 4.19247208555
```

The functions `mxm` and `mxmeval` provide alternatives to symbolic expressions. Together with the `@expr` syntax, this can be used to exchange and evaluate expressions between Maxima and EMT. In most cases, it is easier to use the syntax of the symbolic expressions.

```
>remvalue(x)
>mxm("diff(x^x,x)") // same as &diff(x^x,x)
   x^x*(log(x)+1)
>expr:="x^x"; mxm("diff(@expr,x)")
   x^x*(log(x)+1)
```

The @-syntax inserts the expression string at this position. It works in Maxima commands, symbolic expressions, and in EMT functions. In functions, the expression is inserted at compile time. The function does not change automatically, if the expression is changed.

Note the command `remvalue(x)`. It is a very frequent error to assume that a variable does not have a value and can be used as a symbolic variable. If the variable has a value it will be used in the expression immediately.

## 4.7 Matrices in Maxima

As one example of the previous section shows, Maxima can also handle matrices. The simple EMT syntax with semicolons and commas can be used to define matrices, even symbolic matrices.

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

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & x \end{bmatrix}$$

```
>&solve(det(A)=0,x)
```

$$[x = 9]$$

Note that the matrix must not be defined with `&:=`, unless the variable `x` has a numerical value. The variable `A` in EMT contains a symbolic string describing the matrix. This symbolic expression can be evaluated numerically as usual.

```
>""+A
 matrix([1,2,3],[4,5,6],[7,8,x])
>A(-4)
             1             2             3
             4             5             6
             7             8            -4
```

But it could have been defined for Maxima only with `&&:`. Then the variable `A` would be undefined in EMT.

We defined it with `&:`. So it is currently an expression in EMT. We can use this expression to numerically solve the same problem. If we define a function first, the expression for the matrix can be inserted into the function with `...`

```
>function f(x) := det(@A)
>longest solve("f",8)
                      9
```

Often Maxima uses vectors to collect results, such as solutions of equations. As an example, we compute the rather complicated zeros of a cubic polynomial and evaluate the result in EMT. Two of the zeros are complex values.

```
>sol &= solve(x^3+x^2+5*x+4=0,x);
>&sol[1]
```

$$x = -\frac{14\left(\frac{\sqrt{3}\,I}{2}-\frac{1}{2}\right)}{9\left(\frac{\sqrt{563}}{2\,3^{3/2}}-\frac{65^{1/3}}{54}\right)}-\frac{\left(\frac{\sqrt{563}}{2\,3^{3/2}}-\frac{65^{1/3}}{54}\right)+\left(-\frac{\sqrt{3}\,I}{2}-\frac{1}{2}\right)}{3}-\frac{1}{3}$$

```
>&"x with sol[1]"()
   -0.0880464-2.20163i
```

To see how the solutions in `sol` look, here is a more simple example. EMT can evaluate the expression.

```
>sol &= solve(x^2-x=3)
```

$$[x = \frac{1-\sqrt{13}}{2},\ x = \frac{\sqrt{13}+1}{2}]$$

```
>sol()
   [-1.30278,  2.30278]
```

### Example

For a more complicate example, we find all Pythagorean triples $a^2+b^2=c^2$. For this, we choose two numbers $2<q<p$ without a common divisor and solve the equation

$$x^2+y^2=1, \quad y=1-\frac{p}{q}x.$$

The simple idea is to set

$$x = \frac{a}{c}, \quad y = \frac{b}{c}$$

which yields $a^2 + b^2 = c^2$. In the end, we have computed the intersections of a line through $(0, 1)$ with the unit circle. Since one intersection is rational the other must be rational too.

```
>sol &= solve([x^2+y^2=1,y=1-x*p/q],[x,y])
```

$$\left[\left[x = \frac{2pq}{q^2+p^2}, y = \frac{q^2-p^2}{q^2+p^2}\right], [x = 0, y = 1]\right]$$

Maxima returns a list with two pairs of solutions for $x$ and $y$. We use `with` to set $x$ and $y$ to these solutions.

```
>x0 &= x with sol[1], y0 &= y with sol[1],
```

$$\frac{2pq}{q^2+p^2}$$

$$\frac{q^2-p^2}{q^2+p^2}$$

The solution is

$$a = 2pq, \quad b = q^2 - p^2, \quad c = q^2 + p^2.$$

This is a famous formula for the Pythagorean triples.

If we want to check

$$x^2 + y^2 = 1$$

we need to call `ratsimp` to simplify the solution (`factor` will work too in this example).

```
>&x0^2+y0^2, &%|ratsimp
```

$$\frac{(q^2 - p^2)^2}{(q^2 + p^2)^2} + \frac{4\,p^2\,q^2}{(q^2 + p^2)^2}$$

$$1$$

In this introduction, we cannot mention all simplifications and transformations that are possible with Maxima. You will need to refer to a tutorial about Maxima for this.

## 4.8 Compatibility and Direct Mode

If you wish to study one of the various introductions to Maxima, you will have to use the direct mode, and the original Maxima syntax. To enter a command in this mode, start the command with `>:::` or use the direct Maxima mode with `maximamode direct`. Note that the `mxm` command in EMT uses the direct syntax too. This makes no difference, since it is used for simple expressions anyway.

The translation between direct and compatibility mode is rather straightforward. We collect the differences of the compatibility mode and the direct mode in the following list.

| Further Information | |
|---|---|
| `>:::...` | Command for Maxima in compatibility mode |
| `>:...` | Command for Maxima in direct mode |
| `,` and `;` | Separation of commands in compatibility mode |
| `;` and `$` | Separation of commands in direct mode |
| `\|` | Append options in compatibility mode |
| `,` | Append options in direct mode |
| `:=` | Set variables compatibility mode, define evaluating functions in compatibility mode |
| `:=` | Define non-evaluating functions in direct mode |
| `:` | Set variables in direct mode |
| `function` | Keyword for non-evaluating functions in compatibility mode |
| `define` | Define evaluating functions in both modes |

The main differences are the command and option separators. The following are two lines of code with the same meaning.

```
>:: diff(x^2/(1+x^2),x); %|factor,
```

$$\frac{2\,x}{(x^2+1)^2}$$

```
>::: diff(x^2/(1+x^2),x)$ %,factor;
```

$$\frac{2\,x}{(x^2+1)^2}$$

Variables are set differently too. The direct mode uses `:` and the compatibility mode uses `:=` just as Euler does.

```
>::: a:3$ a^2
```

$$9$$

```
>:: a:=3; a^2
```

$$9$$

For the definition of a function, the direct mode uses either `:=`, which does not evaluate the function body, or `define`, which does. Alternatively, the evaluation can be forced with `''(...)`

```
>::: f(x) := integrate(x^2,x)
```

$$f(x) := \text{integrate}(x^2, x)$$

```
>::: f(x) := ''(integrate(x^2,x))
```

$$f(x) := \frac{x^3}{3}$$

As you see, the simple `:=` can lead to very wrong answers. In the compatibility mode, we use the Euler form `function ...` to set a function, which does not evaluate the body, and `f(x):=...` to evaluate the function body. This is logical, since `:=` works in the same way for variables.

```
>:: integrate(x^2,x); f(x) := %
```

$$f(x) := \frac{x^3}{3}$$

```
>:: function f(x) := integrate(x^2,x)
```

$$f(x) := \text{integrate}(x^2, x)$$

# Chapter 5

# Numbers and Data

## 5.1 Complex Numbers

So far we have used floating point real numbers most of the time. EMT has two other scalar numerical data types, which can be used in matrices: complex numbers and intervals. In this section, we introduce complex numbers.

Complex numbers are entered by appending `i` to the imaginary part, or using the constant `I`, which has the aliases `%i` and `I$`.

All computations in EMT are real, as long as there are no other data types involved. Thus `sqrt(-1)` will not work. To turn a real number `x` into a complex number, use `complex(x)`. For the converse, use `real(z)`. This will yield an error message if the imaginary part of `z` is not close to zero. Of course, `re(z)` and `im(z)` simply compute the real and imaginary part of a complex number.

```
>1+2i
   1+2i
>(1+2*I)^2
   -3+4i
>real(I*I)
   -1
>sqrt(complex(-1))
   0+1i
>re(2+3i), im(2+3i)
   2
   3
```

Note that `sqrt(complex(-1))` is not exactly `I`. But the usual format rounds the result to the expected value. To stop this, use `longestformat`, or `zerorounding(false)`.

EMT can compute most numerical functions in the complex range. The results of the logarithm and the power function use the main part, with the complex plane cut along the negative axis.

```
>real(exp(I*Pi))
   -1
>sqrt(-1+0.1i), sqrt(-1-0.1i)
   0.0499378+1.00125i
   0.0499378-1.00125i
>real(sin(I)^2+cos(I)^2)
   1
>log(complex(-1))
   0+3.14159i
```

It is also possible to plot complex numbers. See the section about the EMT matrix language for an example.

| Further Information | |
|---|---|
| `abs(z)` | Modulus |
| `arg(z)` | Argument in $]-\pi,\pi]$ |
| `conj(z)` | Complex conjugate $\overline{z}$ |
| `plot2d` | Knows how to plot complex grids and curves. |

## 5.2 Intervals

The third data type in EMT are intervals. Interval arithmetic is used to control rounding errors. EMT can produce guaranteed inclusions of solutions for many problems using its interval arithmetic.

Intervals are entered using the `~a,b~` notation as in the following examples. `~x~` creates a small interval around `x`.

```
>~1,2~
   ~1,2~
>interval(1,2)
   ~1,2~
>left(~-1,2~), right(~-1,2~), middle(~-1,2~)
```

```
   -1
   2
   0.5
>~pi~
   ~3.141592653589792,3.141592653589794~
```

Alternatively, the engineering notation ± can be used. This special character is not on the keyboard, and can be entered using the function key F8.

```
>1±0.1
   ~0.9,1.1~
```

The basic rule of interval arithmetic is that the result includes all values from any of the parameter intervals. Thus `x*x` and `x^2` can yield different results. `x*x` multiplies any value in `x` with any other value in `x`, while `x^2` takes the square of any value in `x`.

```
>~0.9,1.1~ + ~0.9,1.1~
   ~1.8,2.2~
>2 * ~0.9,1.1~
   ~1.8,2.2~
>~-1,1~ * ~-1,1~
   ~-1,1~
>~-1,1~ ^ 2
   ~0,1~
```

The same is true for functions. The function is evaluated in all elements of the parameter interval, and the result is an interval containing all results. But due to efficiency, often this interval is not the smallest interval containing all results.

```
>sin(~1,2~)
   ~0.84,1~
>left(sin(~1,2~)), sin(1), sin(2)
   0.841470984808
   0.841470984808
   0.909297426826
```

Note that the sine function has a maximum in the interval. The closest inclusion would be [sin(1), 1].

A better approximation can often be obtained using a simple split of the interval into subintervals using `ieval`. Sometimes, the function `mxmieval` delivers better results. However, it uses Maxima to compute the derivative of the function. In `mxmieval` it is possible to subdivide the interval for even better results.

```
>z:=~0.1,0.5~; z^3+z^2-2*z
   ~-0.99,0.18~
>ieval("x^3+x^2-2*x",~0.1,0.5~)
   ~-0.7,-0.17~
>mxmieval("x^3+x^2-2*x",~0.1,0.5~,10)
   ~-0.63,-0.18~
```

### Example

We measure the angle of a tower in 100 m distance as 11.25° above ground. How high is the tower? All measurement have only the given accuracy. So we use proper intervals to estimate the result. For comparison, we use a pseudo-exact computation.

```
>sin(11.25°±0.005°)*(100±1)
   ~19.3,19.8~
>sin(11.25°)*100
   19.5090322016
```

There is also a simple way to estimate the error. Because of

$$\frac{\Delta \sin(x)}{x} = \frac{\Delta \sin(x)}{\delta x} \cdot \frac{\Delta x}{x} \approx \cos(x) \cdot \frac{\Delta x}{x}$$

the sine function decreases the relative error for $x \neq 0$. And the multiplication of two values $a$ and $b$ satisfies

$$\frac{\Delta ab}{ab} = \frac{(a+\Delta a)(b+\Delta b) - ab}{ab} \approx \frac{\Delta a}{a} + \frac{\Delta b}{b}$$

neglecting $\Delta a \Delta b$. Thus the multiplication adds the relative errors. In our example, we get a relative error of approximately 1% from the bad measurement of the distance, which is approximately $0.2m$ of height.

### Example

The error of the Simpson formula with step size $h = (b-a)/n$ is equal to

$$\frac{h^4}{180} \sup_{x \in [a,b]} f^{(4)}(\xi)$$

for some $\xi$ in the interval. We use that to get a guaranteed inclusion for the integral in EMT. The function `mxmisimpson` uses Maxima to compute the fourth derivative. The exact value can be computed using the `erf` function of Maxima.

```
>mxmisimpson("exp(-x^2)",0,1,100)
   ~0.7468241328113,0.7468241328136~
>longest romberg("exp(-x^2)",0,1)
        0.7468241328139768
>longest &:float(integrate(exp(-x^2),x,0,1))
          0.74682413281243
```

This can be done by hand too. We need an estimate for the fourth derivative of the function. This derivative increases for $x > 0$. A computation in Maxima yields the following.

```
>&diff(exp(x^2),x,4), f4max := %(1)
```

$$16\,x^4\,E^{x^2} + 48\,x^2\,E^{x^2} + 12\,E^{x^2}$$

```
   206.589418963
```

We take 210 as a good estimate. Now we compute the Simpson integral

$$S(h) = \frac{h}{3}\left(f(0) + 4f(h) + 2f(2h) + \ldots + 2f(1-2h) + 4f(1-h) + f(1)\right)$$

and add the estimate for the error.

```
>n=1000; f=ones(n+1); f[2:2:n]=4; f[3:2:n-1]=2; f[1:5], f[-5:-1]
   [1,  4,  2,  4,  2]
   [2,  4,  2,  4,  1]
>h=1/~n~; x=(0:n)*h; sum(exp(-x^2)*f)*h/3+h^4/180*~0,210~
   ~0.7468241328123,0.7468241328137~
```

Finally, we can also use the function `mxmiint` to get a very narrow inclusion for the integral using Maxima and even higher derivatives.

```
>mxmiint("exp(-x^2)",0,1)
   ~0.746824132812408,0.746824132812446~
```

| Further Information | |
|---|---|
| `a && b` | Intersection of intervals |
| `a \|\| b` | Interval containing the union of intervals |
| `a << b` | Tests, if $a$ is contained in $b$ |
| `a <<= b` | The same, where equality is allowed |
| `expand(a,f)` | Expands $a$ by the factor $f$ |
| `mxminewton` etc. | See the chapter about exact arithmetic |

## 5.3 Strings

Strings are the non-numerical data type in EMT. They are used for expressions, and also for things like file input and output, or for labels in plots. Many functions accept expressions, as we have already seen. Strings are also used to communicate with Maxima. Strings can form vectors, but not matrices.

Here are some examples of string functions.

```
>s:="This is a test",
   This is a test
>strlen(s)
   14
>ascii(s) // first character in a
   84
>char(84)
   T
>substring(s,6,8)
   is
>substring(s,-4,-1)
   test
>substring(s,strfind(s,"a",0),-1)
   a test
>"affe"<="bravo"
   1
>s+" "+s
   This is a test This is a test
```

There are also vectors of strings. `plot2d` uses these vectors to plot more than one expression.

```
>plot2d(["sin(x)","cos(x)"],0,2pi):
```

For interactive programs, there is the `input` function. The input is interpreted as a numerical expression. If there is an error, EMT will prompt the user again, until the expression is valid or the user presses the Esc key.

**Example**

The following commands should be entered into an EMT file. To do this, press F9 in an empty line to open the internal editor, and copy and paste the following text into the editor.

```
"Prime factors.",
n=input("Enter a number: ");
"Factors:", factor(n),
```

If this file is loaded, we get the following dialog between the user and EMT.

```
>load "C:\Users\ReneEMTEulerTemp.e"
   Prime factors.
   Enter a number: ? >10234
   Factors:
   [2,  7,  17,  43]
```

| Further Information | |
|---|---|
| strfind | Finds substrings |
| tolower | Converts to lower case |
| toupper | Converts to upper case |
| sort | Sort a vector of strings |

## 5.4 Collections and Lists

Collections can contain any data type. In EMT, collections are immutable objects. Collections can collect data at one place. E.g., it is an alternative way to return multiple values from a function (see the section about programming).

To create a collection, use {{...}}. To access an element of the collection use indexing.

```
>C={{"first",1:3,3}}
   first
   [1,  2,  3]
   3
>C[2]
   [1,  2,  3]
```

Collections are not meant to be used as lists. EMT has an own list type. Such lists are stored globally in a separate variable space. Lists are mutable by the `glist` functions.

```
>glist("test");
>for k=1 to 10; glistadd("test","element"+k); end;
>glistvar("test",5)
   element5
```

One of the main purpose of collections is to collect function names and additional parameters (besides `x` etc.) for algorithms and plot functions.

```
>function f(x,y,a,b) := a*x^2+b*x*y+y^2;
>plot3d({{"f",2,-1}},>spectral,cp=2): // with a=2, b=-1
```

This works, because collections can be evaluated if the first element is the name of a function or an expression.

```
>function f(x,a) := a*exp(x/a);
>L={{"f",4}}; L(5) // with a=4
   13.9613718298
```

Note that the argument `5` of the call `L(5)` is used for the parameter `x`. But `f` needs another parameter. If `a` had a default value the same technique would work just as well.

```
>function f(x,a=4) := a*exp(x/a);
>L(5)
   13.9613718298
>L={{"f"}}; L(5)
   13.9613718298
>L={{"f",6}}; L(5)
   13.8058553454
```

The same trick works for expressions. But then, additional parameters must be named. Note that expressions can see global variables anyway. But if algorithms are called in functions the evaluation cannot see the local variable of the function. Nevertheless, we demonstrate the use evaluation of a collection with an expression on the global command line.

```
>solve({{"a*x^2-x/a",a=7}},1), "a*x^2-x/a"(%,a=7)
   0.0204081632653
   0
```

## 5.5 Polynomials

Polynomials are not a primitive data type. They are stored in EMT as vectors of coefficients starting with the constant coefficient. EMT can evaluate, add, multiply, divide, and differentiate polynomials. Here are some examples. We compare the results to Maxima.

```
>p:=[1,2,3]; polyval(p,2)
   17
>pd:=polydif(p); polyval(pd,2)
   14
>&diff(1+2*x+3*x^2,x)(2)
   14
>p2:=polymult(p,p)
   [1,  4,  10,  12,  9]
>&expand((1+2*x+3*x^2)^2)
```

$$9\,x^4 + 12\,x^3 + 10\,x^2 + 4\,x + 1$$

```
>&expand((1+2*x+3*x^2)^2)(2)
   289
>polyval(p2,2)
   289
```

The zeros of polynomials can be computed in EMT using a numerical method. EMT finds all complex zeros. Here is a comparison with Maxima.

```
>polysolve(p)
   [ -0.333333-0.471405i,  -0.333333+0.471405i ]
>sol &= solve(1+2*x+3*x^2=0,x)
```

$$\left[x = \frac{-\sqrt{2}\,I - 1}{3},\ x = \frac{\sqrt{2}\,I - 1}{3}\right]$$

```
>sol()
   [ -0.333333-0.471405i,  -0.333333+0.471405i ]
```

Of course, EMT has methods to interpolate with polynomials, fit polynomials to values or for FFT. Those numerical methods are described later.

| Further Information | |
|---|---|
| `polydiv(p,q)` | Divides with remainder (multiple returns!) |
| `polytrunc` | Minimizes the degree of the polynomial |
| `polyadd` | Adds polynomials |
| `polycons` | Generates a polynomial from its zeros |

# Chapter 6

# The Matrix Language

## 6.1 Matrices and Vectors

A matrix in EMT is entered using brackets [...]. The values of each row are separated by commas, the rows are separated by semicolons. Incomplete rows are filled with zeros.

```
>[1,2;3,4]
               1              2
               3              4
>a:=3;
>short [1,a;0,1,a;0,0,1,a]
             1           3           0           0
             0           1           3           0
             0           0           1           3
```

An alternative, which does also work in Maxima is the matrix command.

```
>matrix([1,2],[3,4])
                    1                   2
                    3                   4
```

Long vectors and matrices may not print in full size. To change this use the operator showlarge, or set the long display permanently with largematrices on.

```
>showlarge random(10,10)
   Column 1 to 5:
        0.569967        0.4412      0.654399      0.451402      0.785921
        0.731288      0.521804     0.0518588     0.0406859     0.0696735
        0.144432      0.240932      0.896927      0.449284      0.180431
        0.819952      0.525137      0.128366      0.545064      0.279446
        0.232776      0.611871       0.56501      0.874601     0.0899351
        0.266388      0.502768      0.490759      0.108418        0.4919
        0.962636      0.417781      0.186796      0.997298      0.143285
        0.829819      0.579399      0.206967      0.821727      0.446005
        0.312018      0.234889      0.713802      0.342169      0.837105
        0.840361      0.624917      0.404306      0.447133      0.892916
   Column 6 to 10:
        0.673755      0.189846     0.0871873      0.178585      0.660332
        etc.
```

A vector is simply a matrix with one row or column. The transposition of a matrix is indicated by A'.

```
>A:=[1,2;3,4;5,6]
                1             2
                3             4
                5             6
>A'
                1             3             5
                2             4             6
```

An often used way to generate matrices is the : operator. It can have a step value (may be negative), or the default step value 1. By numerical reasons, adding the steps will not always exactly meet the final value. EMT uses the internal epsilon to check for this.

```
>matrix([1,2],[3,4])
                1             2
                3             4
>0:0.1:1
   [0,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1]
>1:3
   [1,  2,  3]
>sum(1:1000)
   500500
```

There are many functions that generate a matrix. Here are some examples.

```
>zeros(2,2)
                0              0
                0              0
>ones(5)
   [1,  1,  1,  1,  1]
>id(2)
                1              0
                0              1
>A=2*id(3); setdiag(A,1,[-1,-1])
                2             -1              0
                0              2             -1
                0              0              2
```

Matrices can also be appended to each other, horizontally or vertically.

```
>v:=1:3
   [1,  2,  3]
>v|v
   [1,  2,  3,  1,  2,  3]
>v_v
                1              2              3
                1              2              3
>v_1
                1              2              3
                1              1              1
```

To access a single matrix element, or a submatrix, EMT uses square brackets as in `A[i,j]`. Round brackets (parentheses) are supported, but switched off by default. Omitting the column index returns a line of $A$, unless $A$ is a row vector. For a column, leave the row index open or set it to `:`.

Always remember, that EMT matrices start with the index 1. `A[0]` yields an empty matrix, or an error message depending on a global flag!

If the indices are vectors, we get a submatrix. The indices may even be unsorted. In this case, we get a permutation of the rows or columns. Negative indices count from the end of the matrix. Here are some examples.

```
>v:=[1,2,3,4,5]
   [1,  2,  3,  4,  5]
>v[4]
   4
```

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

It is possible to assign values to a submatrix. Take care that the values must fit into the submatrix. It is also possible to assign a single number to a submatrix. In this case, all elements of the submatrix will be set to this number.

```
>v:=1:5
   [1,  2,  3,  4,  5]
>v[2:3]:=0
   [1,  0,  0,  4,  5]
>A:=[1,2,3,4,5;1,2,3,4,5]
               1             2             3             4             5
               1             2             3             4             5
>A[[1,2],[2,3]]:=[8,9;10,11]
               1             8             9             4             5
               1            10            11             4             5
```

Note, that submatrix indexing can also be applied to the result of functions, if the result is of the matrix type. E.g., the `shuffle` function shuffles vectors randomly, and we can extract the first 6 values with index `1:6`.

```
>sort(shuffle(1:49)[1:6])'
```

27
28
32
44
45
48

| Further Information | |
|---|---|
| `linspace(a,b,n)` | $n$ equidistant points in $[a,b]$ |
| `equispace(a,b,n)` | points distributed in the arcsine distribution |
| `size(A)` | The size `[c,r]` of $A$ |
| `cols(A)` | Number of columns of $A$ |
| `rows(A)` | Number of rows of $A$ |
| `diag(A,k)` | $k$-th diagonal of $A$ |
| `redim(A,n,m)` | Reformat $A$ into a new matrix |

## 6.2 The Matrix Language

The basic rule of the matrix language is that all functions and operators are evaluated element for element (vectorized). So the multiplication `*` is not the matrix multiplication, but the elementwise multiplication. The matrix multiplication uses a dot as in `A.B`.

```
>v:=[1,2,3]; v*v
   [1,  4,  9]
>sqrt([1,2;3,4])
               1       1.41421
         1.73205             2
```

When combining a matrix with a vector in one operation, EMT tries to bring the vector to the same size as the matrix by duplicating the vector in a natural way, so that the result can be computed element for element. A row vector and a column vector will combine to a matrix (like a tensor product). Moreover, using a scalar number and a matrix in one operation will combine the scalar number with all elements of the matrix.

```
>v:=[1,2,3]; 2*v+4*v
   [6,  12,  18]
```

```
>w:=[1;-1;0]
                1
               -1
                0
>v*w
                1             2             3
               -1            -2            -3
                0             0             0
```

In the last example above, the resulting matrix $A$ has the property that

$$a_{i,j} = v_j \cdot w_i.$$

I.e., EMT uses the elements from the rows of $w$ and the columns of $v$ to form the matrix. Imagine $v$ duplicated three times vertically, and $w$ appended three times horizontally. Of course, this is not the matrix product!

The same rule applies to all operations, also for comparisons like `>`, which deliver 1 or 0 elementwise.

### Example

The Pascal triangle is usually written as

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

Since we do not have triangle matrices in EMT, we use a square matrix, filled with 0. This is very easy to achieve, since `bin(n,m)` follows the matrix rules, and moreover delivers 0, if $m < 0$ or $m > n$.

```
>goodformat(5,2); n:=0:5; bin(n',n)
      1     0     0     0     0     0
      1     1     0     0     0     0
      1     2     1     0     0     0
      1     3     3     1     0     0
      1     4     6     4     1     0
      1     5    10    10     5     1
```

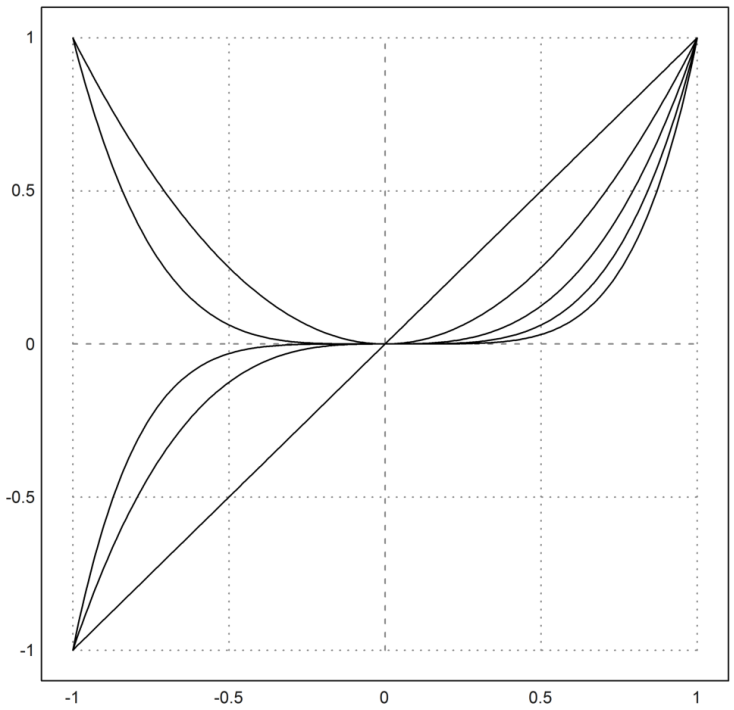



Figure 6.1: The functions $x^n$

### Example

We already plotted functions using expressions. However, it is also possible to plot a table of values.

```
>x:=-1:0.01:1; n:=(1:5)'; plot2d(x,x^n):
```

The second example works, since `n` is a column vector. Thus $x^n$ is the matrix

$$\begin{pmatrix} x_1 & \dots & x_m \\ \vdots & & \vdots \\ x_1^5 & \dots & x_m^5 \end{pmatrix}.$$

The function `plot2d` plots every row of the matrix as a separate function, using the same $x$-values for all rows.

We remark that expressions in `plot2d` can also produce several plots in one command very easily. Simply use a column vector in the expression.

```
>plot2d("x^(1:10)'",0,1);
```

### Example

We compute the Gauß algorithm for

$$\begin{aligned} x+y+z &= 1 \\ 2x+z &= 2 \\ x-y &= 1 \end{aligned}$$

step by step.

```
>fracformat(10);
>A:=[1,1,1;2,0,1;1,-1,0]; b:=[1;2;1]; M:=A|b
            1          1          1          1
            2          0          1          2
            1         -1          0          1
>M[2]:=M[2]-2*M[1]; M[3]:=M[3]-M[1]
            1          1          1          1
            0         -2         -1          0
            0         -2         -1          0
>M[2]:=M[2]/(-2); M[3]:=0
            1          1          1          1
            0          1        1/2          0
            0          0          0          0
>M[1]:=M[1]-M[2]
            1          0        1/2          1
            0          1        1/2          0
            0          0          0          0
```

The result reads as

$$x_1 = 1 - \frac{x_3}{2}, \quad x_2 = -\frac{x_3}{2},$$

where $x_3$ is arbitrary.

There is the function `pivotize` which makes everything easier. It changes the Gauß tableau in each step.

```
>fracformat(10);
>A:=[1,1,1;2,0,1;1,-1,0]; b:=[1;2;1]; M:=A|b
            1          1          1          1
            2          0          1          2
            1         -1          0          1
>pivotize(M,1,1)
            1          1          1          1
```

```
           0        -2        -1         0
           0        -2        -1         0
>pivotize(M,2,2)
           1         0       1/2         1
           0         1       1/2         0
           0         0         0         0
```

Of course, there is also the function `echelon` which does everything in one step.

```
>fracformat(10);
>A:=[1,1,1;2,0,1;1,-1,0]; b:=[1;2;1]; M:=A|b
           1         1         1         1
           2         0         1         2
           1        -1         0         1
>echelon(M)
           1         0       1/2         1
           0         1       1/2         0
```

### Example

Here is an example in the complex plane. The following code generates 1000 complex numbers $z$ evenly spaced around the unit circle (roots of unity). Then we map these numbers with $z \mapsto z^2 + z$, and plot those values. The function `plot2d` knows how to plot a path of complex values.

```
>x:=linspace(0,2pi,1000); z:=exp(I*x);
>plot2d(z+z^2,r=2);
```

We could also plot a grid of complex numbers.

```
>x:=-1:0.1:1; y:=x'; z:=x+1i*y;
>plot2d(exp(z),r=3):
```

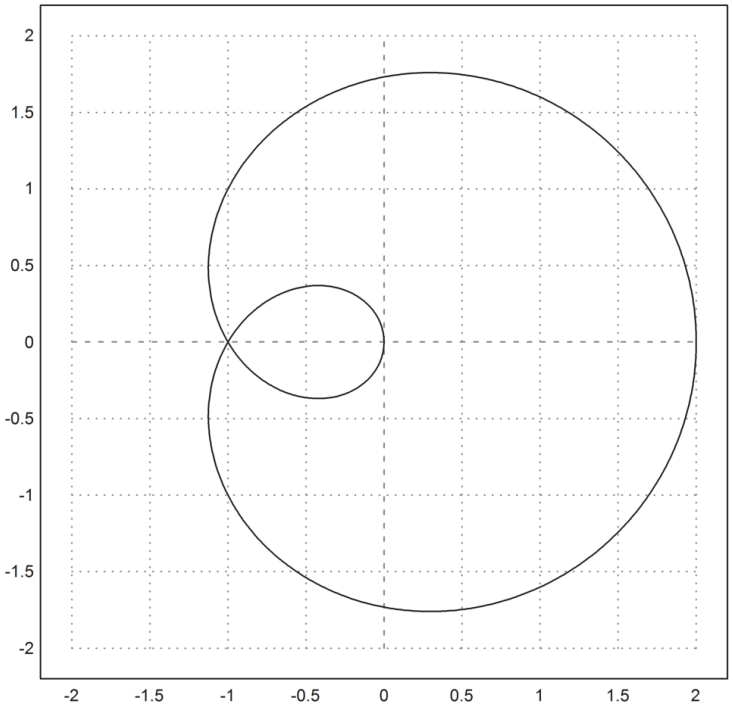



Figure 6.2: Image of the unit circle with $w = z^2 + z$

### Example

How many prime numbers are less than $n$? And which of them have the form $m^2+1$?

```
>n:=100000; k:=3:2:n; sum(isprime(k))+1
 9592
>k[nonzeros(isprime(k) && k==floor(sqrt(k))^2+1)]
 [5,  17,  37,  101,  197,  257,  401,  577,  677,  1297,  1601,  2917,
 3137,  4357,  5477,  7057,  8101,  8837,  12101,  13457,  14401,
 15377,  15877,  16901,  17957,  21317,  22501,  24337,  25601,  28901,
 30977,  32401,  33857,  41617,  42437,  44101,  50177,  52901,  55697,
 57601,  62501,  65537,  67601,  69697,  72901,  78401,  80657,  90001,
 93637,  98597]
```

This examples show that boolean operators like && are also vectorized.

| Further Information | |
|---|---|
| max(A),min(A) | Maximum and minimum of the rows of $A$ |
| totalmax(A),totalmin(A) | Maximum of all values in $A$ |
| extrema(A) | Maximum and minimum plus the indices |
| sort(v) | Sorts $v$ |

## 6.3 Linear Algebra

For the multiplication of matrices we use the dot A.x. Thus v'.w is the scalar product between column vectors (also scalp(v,w)). The cross product is computed with crossproduct (works for row an column vectors).

```
>v:=[1;2;3]
                1
                2
                3
>v'.v
   14
>crossproduct(v,[1;1;2])
                1
                1
               -1
>crossproduct(v',[1;1;2]')
   [1,  1,  -1]
```

The inverse matrix can be computed with inv(A). Note that A^(-1) would deliver the inverses of the elements of $A$. By the same reason, the matrix powers have to be computed with matrixpower(A,n).

```
>shortformat; A:=[1,2;2,1]
           1         2
           2         1
>det(A)
   -3
>fracprint(inv(A));
        -1/3       2/3
         2/3      -1/3
>matrixpower(A,-5).matrixpower(A,5)
           1         0
           0         1
```

Of course, the last example does not deliver the identity matrix exactly. The default output format rounds this, however.

To solve a linear system, use the backslash operator.

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

Note, that this is more efficient then `inv(A).b`. If $A$ is not regular (numerically $\det A = 0$), you will get an error message.

| Further Information | |
|---|---|
| `kernel(A)` | Kernel of $A$ |
| `svdkernel(A)` | Orthogonal basis of the kernel |
| `image(A)` | Image of $A$ |
| `svdimage(A)` | Orthogonal basis of the image |
| `rank(A)` | Rank of $A$ |
| `lu(A)` | LU decomposition of $A$ (See `help lu`) |

## 6.4 Regression Analysis

If a system $Ax = b$ has no solution, we can try to minimize the norm $\|Ax - b\|$ using the function `fit`. This is always necessary if there are more equations than unknowns.

```
>A:=[1,1;2,1;3,1]
                1             1
                2             1
                3             1
>b:=[2;3;2]
```

```
                2
                3
                2
>x:=fit(A,b)
                0
          2.33333
>norm(A.x-b)
   0.816496580928
```

`fit` uses Givens rotations. The alternative is to use `fitnormal`, which uses the normal equation. It fails, if $A$ does not have maximal rank. The function `svdsolve` uses a more complicated method involving singular values. As an advantage, `svdsolve` delivers the result with minimal norm, if there is more than one result.

```
>A:=[1,2,3;4,5,6;7,8,9]
                1              2              3
                4              5              6
                7              8              9
>det(A)
   0
>b:=A.ones(3,1)
                6
               15
               24
>svdsolve(A,b)
                1
                1
                1
```

There is the function `polyfit` to fit polynomials of a given degree to given data.

### Example

Let us fit a polynomial of degree 2 to the exponential function on $[-1,1]$. The error is approximately 0.07.

```
>x:=-1:0.1:1; y:=exp(x); p:=polyfit(x,y,2)
   [0.995583,  1.11404,  0.540186]
>plot2d(x,y,>points); plot2d("polyval(p,x)",>add,color=red):
```

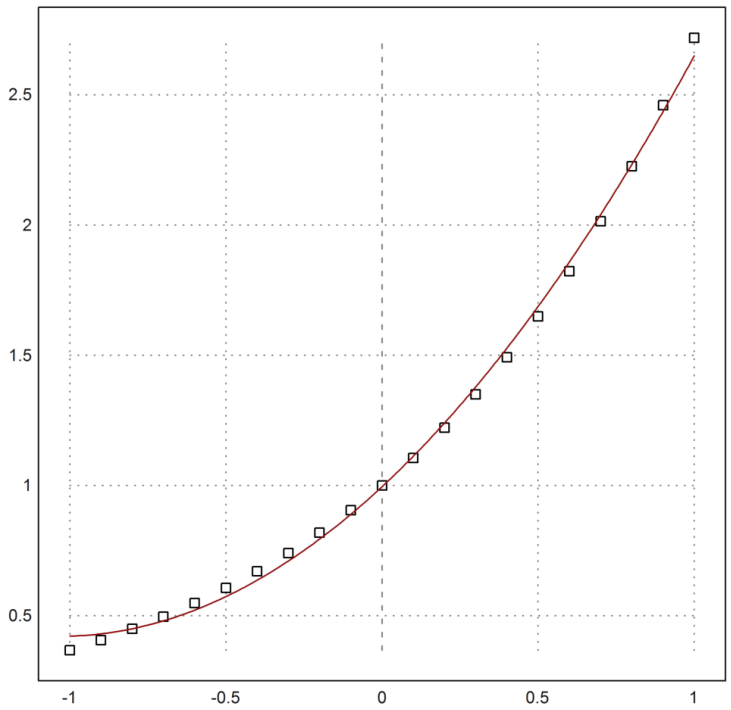



Figure 6.3: A best polynomial fit

### Example

In this example we fit functions of the form

$$f(x) = ae^{-x} + be^{-2x} + c$$

to measurements

$$(x_1, y_1), \ldots, (x_n, y_n).$$

We generate the measurements artificially adding random errors to a given function. The matrix for the fit contains the basis functions evaluated at the points of measurement.

$$\begin{pmatrix} e^{-x_1} & e^{-2x_1} & 1 \\ \vdots & \vdots & \vdots \\ e^{-x_n} & e^{-2x_n} & 1 \end{pmatrix}.$$

Such a matrix can be generated very easily using the matrix language of EMT.

```
>function f(x,a,b,c) := a*exp(-x)+b*exp(-2*x)+c
```

```
>x:=0:0.1:1; y:=f(x,1,0.8,0.5)+normal(size(x))*0.01;
>A:=f(x',1,0,0)|f(x',0,1,0)|f(x',0,0,1);
>s:=fit(A,y')
         0.93761
        0.855681
        0.510332
>plot2d(x,y,>points); plot2d(x,(A.s)',>add,color=red):
```

**Example**

For non-linear fits, EMT must use numerical methods to minimize functions of several variables. We will talk about these methods later. For an example, we fit with the following type of functions.

$$f_p(x) = p_1 \cos(p_2 x) + p_2 \sin(p_1 x).$$

The function name is provided to `modelfit` as an argument, plus a start point, the data and a switch to use Powell's method for minimization.

```
>function model(x,p) := p[1]*cos(p[2]*x)+p[2]*sin(p[1]*x);
>xdata = [-2,-1.64,-1.33,-0.7,0,0.45,1.2,1.64,2.32,2.9]; ...
>ydata = [0.699369,0.700462,0.695354,1.03905,1.97389,2.41143, ...
>  1.91091,0.919576,-0.730975,-1.42001];
>pbest=modelfit("model",[1,0.2],xdata,ydata,>powell)
   [1.88185,  0.70023]
>plot2d(xdata,ydata,>points); plot2d("model(x,pbest)",color=red,>add):
```

## 6.5 Eigenvalues and Singular Values

Eigenvalues are computed with `eigenvalues`, eigenvectors with `eigenspace`. Of course, we can also compute the characteristic polynomial, and its zeros using `polysolve`.

In the following example, we compute the eigenvector with the function `kernel`. The eigenvalues will always be complex. However, we can convert to real, since our matrix is known to have real eigenvalues.

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

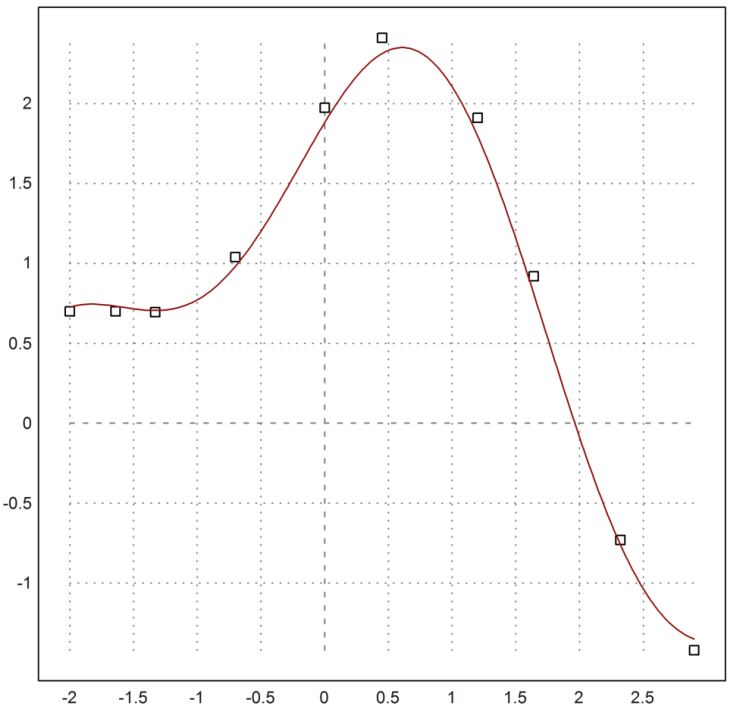



Figure 6.4: Fit with a function $f_p(x)$

```
>charpoly(A)
   [-3,  -2,  1]
>real(polysolve(charpoly(A)))
   [-1,  3]
>l:=re(eigenvalues(A))
   [3,  -1]
>eigenspace(A,-1)
       -0.707107
        0.707107
>v:=kernel(A-l[2]*id(2))
              -1
               1
>A.v
               1
              -1
>{l,M}:=eigen(A); re(l)
   [3,  -1]
>re(M.A.inv(M))
              -1             0
```

```
                0               3
```

The function `eigen` returns two values, a vector with the eigenvalues, and a matrix $M$ of eigenvectors. It is known that

$$MAM^{-1} = D.$$

If $A$ is symmetric, EMT delivers an orthogonal $M$, and we have $M^{-1} = M'$.

Singular Values compute a decomposition of an $n \times m$ matrix $A$ in the form

$$A = UDW'.$$

The $n$ columns of $U$ will be orthonormal. $D$ is a $m \times m$ diagonal matrix, and $W$ is orthogonal.

```
>A:=[1,2,3;4,5,6]
                1               2               3
                4               5               6
>{U,d,W}:=svd(A);
>d
   [9.50803,  0.77287,  0]
>U
       -0.386318      -0.922366               0
       -0.922366       0.386318               0
>U.U'
                1               0
                0               1
>W.W'
                1               0               0
                0               1               0
                0               0               1
>U.diag(3,3,0,d).W'
                1               2               3
                4               5               6
```

| Further Information | |
|---|---|
| `xeigenvalue(A,l)` | Tries to improve the eigenvalue $l$ numerically. |
| `jacobi(A)` | Jacobi method for symmetric $A$ |

## 6.6 Sparse Matrices

EMT has support for sparse matrices. This is a special data type, which only saves the non-zero indices of a matrix. There are functions to handle or modify sparse matrices, and to convert to and from ordinary matrices.

The function cpx compresses a matrix into the internal compressed mode. The function determines zero entries using the internal epsilon. To convert back to ordinary matrices, use decpx.

```
>A=id(6)*2+diag(6,6,-1,1)+diag(6,6,1,1); short A
 Real 6 x 6 matrix

          2          1          0          0     ...
          1          2          1          0     ...
          0          1          2          1     ...
          0          0          1          2     ...
          0          0          0          1     ...
          0          0          0          0     ...
>R=cpx(A)
 Compressed 6x6 matrix
     1     1                    2
     1     2                    1
     2     1                    1
     2     2                    2
     2     3                    1
     3     2                    1
     3     3                    2
     3     4                    1
     4     3                    1
     4     4                    2
     4     5                    1
     5     4                    1
     5     5                    2
     5     6                    1
     6     5                    1
     6     6                    2
```

Compressed matrices can be multiplied rapidly. The following would take much longer using the usual multiplication.

```
>n=1000; A=id(n)*2+diag(n,n,-1,1)+diag(n,n,1,1);
>R=cpx(A);
>R2=cpxmult(R,R);
```

### Example

In the following example we build a random sparse matrix, and set $b$ in $Ax = b$ equal to the sums of the rows. To get such a matrix, we use `cpxset`, which takes rows of the form $(i, j, a_{i,j})$. To make sure the matrix is regular, we set the diagonal to a large value 20.

Then we solve the linear system, using the conjugate gradient method for large, sparse systems, as implemented in the function `cpxfit`.

```
>H=cpxzeros(1000,1000);
>ind=intrandom(10000,2,1000);
>H=cpxset(H,ind|normal(rows(ind),1));
>H=cpxsetdiag(H,0,20);
>b=cpxsum(R);
>x=cpxfit(R,b);
>totalmax(cpxmult(R,x)-b)
   2.27965202271e-010
```

Sparse matrices are used as incidence matrices of graphs holding the values of the edges running between the points of the graph. There are special functions to create and modify those matrices.

# Chapter 7

# Functions of Several Variables

## 7.1 3D Plots

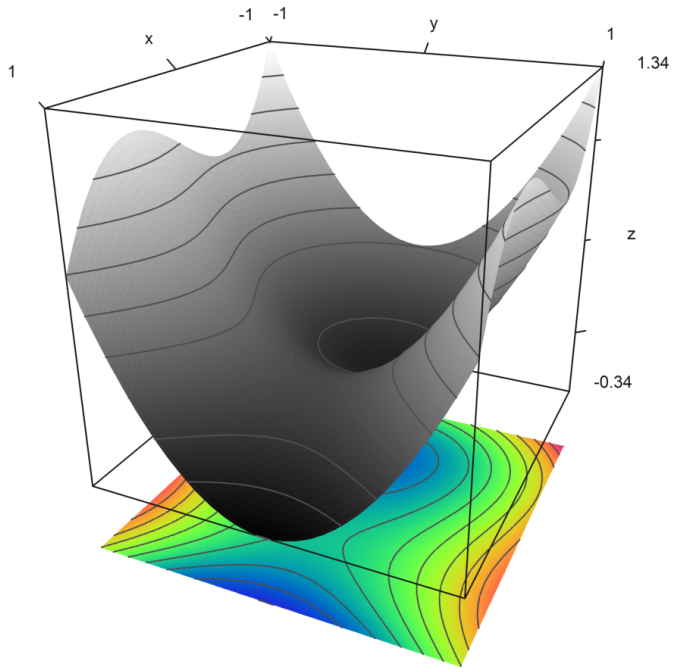



Figure 7.1: $f(x, y) = y^2 - x^2 \sin(x) + x/2$

Just as `plot2d` plots planar graphs, the function `plot3d` plots functions of two

variables and other objects in 3D graphs using central projection (vanishing line projection).

There are the following basic types of 3D plots in EMT.

- Solid plots. Plots the graph of a function in two variables, or a surface determined by three matrices of $x$- $y$- and $z$-coordinates. The surface can have two different color sides or shading. Some types compute the shading by the light source, others by the $z$-coordinate. The shading can be shades of a single color, or different types of spectral shadings.
- Line plots. These plots show only lines in 3D. The lines can also form a grid.
- Point plots. These plots show a cloud of points in space.

EMT has no true 3D scenes. The plots are done by sorting the objects from back to front. It is possible to plot several items into one plot if the order is carefully observed. There are some examples for this in the tutorial. If you need true photo realistic 3D scenes consider using the interface to Povray that EMT provides.

Let us show some basic plots. We first want to plot the graph

$$G = \{(x, y, f(x, y)) : a \le x \le b,\ c \le y \le d\}$$

of a function of two variables.

There are two basic types. The first type is a grid plot with two sides of different configurable colors (see figure 7.2).

```
>plot3d("x^2+y^2",a=-2,b=1,c=-2,d=1,>user);
```

The second type shows a surface with a hue and level lines. The hue can depend on the $z$-coordinate and can be a spectral color or a simple color with shades. Or the shading can depend on the light fall on the plot. This type of plots can also contain level lines. The level line of level $c$ is

$$N_c = \{(x, y, f(x, y)) : f(x, y) = c\}.$$

It is drawn directly on the 3D-plot in the height $z = c$. It is possible to add a color plane underneath the plot (see figure 7.1).

```
>plot3d("y^2-x^2*sin(x)+x/2",>hue,>levels,n=100, ...
>  >hue,cp=1,cpcolor=spectral,cpdelta=0.2,zoom=2.8):
```

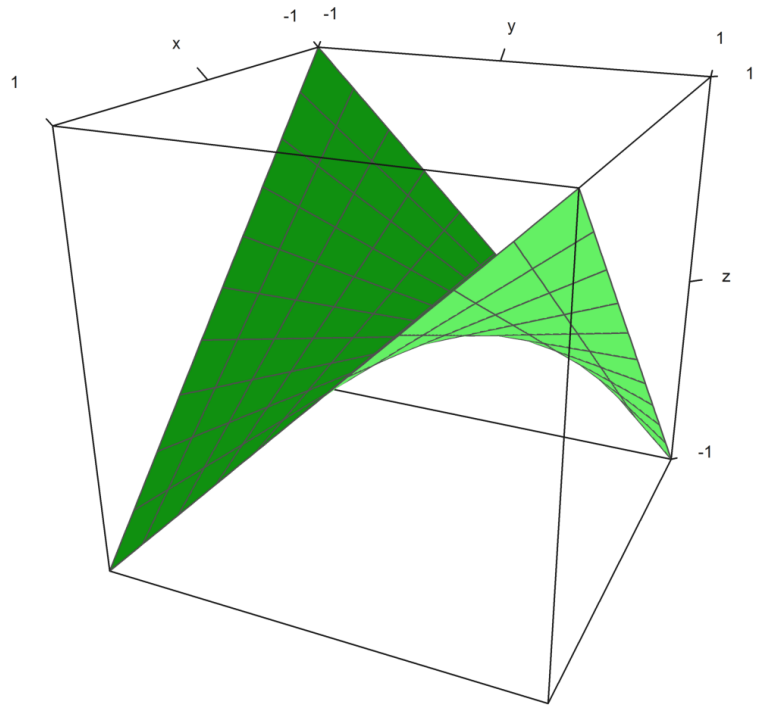



Figure 7.2: The function $f(x,y) = xy$

Plots of functions or expressions are scaled by default to fit into the unit cube (`scale=1`). Moreover, the function values $z = f(x,y)$ are scaled to agree with the $x$-$y$-range. To change this set `<fscale`, or set the value to some other maximal function value.

The range for the plot can be set with `a`, `b`, `c`, `d`, or with a radius `r`, by default around the origin. For the view, the most important options are the `angle` and the `height` of the plot, or the `zoom` for magnification.

User interaction is done with `>user`. The user can then turn the plot with the cursor keys, zoom in or out with `+` or `-`, move the center of the view, or generate an anaglyph plot with `a`. To reset the view, press `Return`.

The center of the plot is the point the view looks at. It is usually the origin of the coordinate space, but can be set to any other point. If the plot is interactive (`>user`), the user can set the center with the cursor keys, once he toggled into the center mode with the key `c`.

All plots can be drawn as anaglyph plots. To view this properly, you need red/cyan glasses. The 3D effect is amazing.

```
>aspect(2); plot3d("sin(x^2+y^2)*exp((-x^2-y^2)/5)",r=4,>polar, ...
>  <frame,n=200,fscale=0.8,>hue,scale=3,>anaglyph,center=[0,0,0.5]):
```

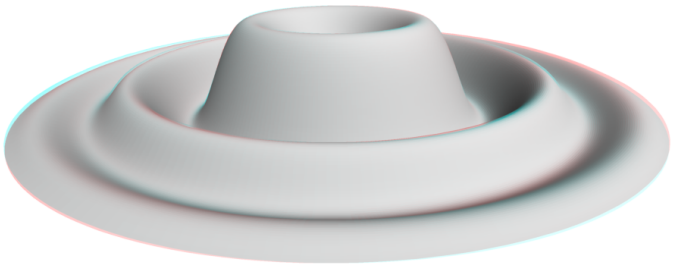


Figure 7.3: Anaglyph plot for red/cyan glasses

There are many more settings to change the look of the plot. Have a look into the documentation of `plot3d`.

| Further Information | |
|---|---|
| `zoom(5)` | Global setting for zoom factor. |
| `viewdistance(2.5)` | Viewing distance. |
| `viewangle(45°)` | Angle of view. |
| `viewheight(45°)` | Angle above the $x$-$y$-plane. |
| `fillcolor(n,m)` | Set the colors simple 3D plots. |
| `reset()` | Resets to default parameters for graphics. |

## 7.2 Surfaces, Curves and Points

This section contains some examples for other types of 3D plots. A parameterized 3D surface can be done with three functions or expressions in $x$ and $y$, or with three matrices. The surface is modeled by a mapping $f : Q \to \mathbb{R}^3$, where $Q$ is a rectangle in the plane $\mathbb{R}^2$.

```
>allwindow; ...
>plot3d("cos(x)*cos(y)","sin(x)*cos(y)","sin(y)", ...
> a=0,b=2*pi,c=-pi/2,d=pi/2, ...
> >hue,color=blue,light=[1,0,1],<frame, ...
> n=90,grid=[18,36],wirecolor=darkgray,zoom=4.5):
```

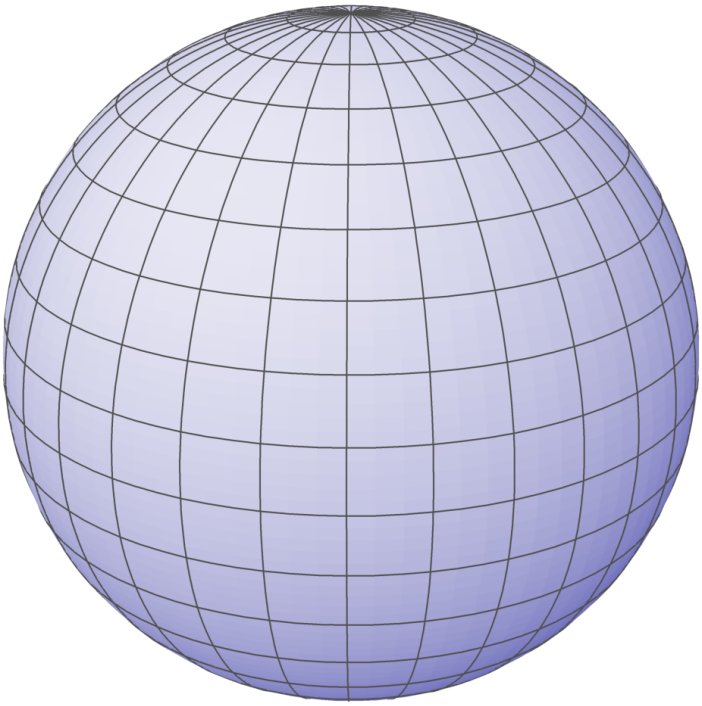


Figure 7.4: Shaded 3D Object

The expressions in this code define a ball. The parameter functions are simply expressions in $x$ and $y$. Of course, functions of the form $f(x, y)$ can be used too.

The command `allwindow` takes the full window for the plot. Otherwise the plot leaves room for the plot title. For a plot without a grid it is often better to set the zoom manually. The zoom is simply set so that the ball fills the image. The rectangle for the parameterization is defined by `a`, `b`, `c`, `c`.

For the plot, we used a shaded model with a specific light source. The frame was disabled. We want 18 grid lines for the lines of latitude and 36 for the lines of longitude. The number of grid lines should be a divisor of the number of surface elements. Both items can be different in each direction of the rectangle.

The following plot is the Möbius strip. This time. we compute three matrices for the three parameters. The parameter `hue=2` turns shading on, but the hue will not depend on the side of the surface. The value `max` determines the maximal darkness.

```
>aspect(16/9); allwindow; ...
```

```
>x:=linspace(0,2*pi,100); y:=(-1:0.1:1)'; ...
>plot3d(cos(x)*(1+y/2*cos(x/2)),sin(x)*(1+y/2*cos(x/2)),y/2*sin(x/2), ...
>  <frame,hue=2,max=0.9,scale=2.7):
```

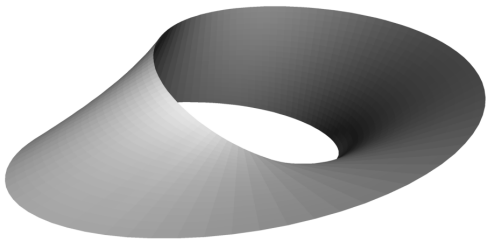


Figure 7.5: Moebius strip

3D Curves are drawn with three parameter functions and the optional parameter `>lines`. We have enable interaction with `>user`. The plot shows a helix.

```
>reset; ...
>plot3d("sin(x)","cos(x)","x/2Pi",>lines,xmin=0,xmax=10pi,n=100,>user):
```

Alternatively, three vectors or matrices can be used. If `>lines` is set the plot will only show the lines in one direction. The coordinates are then in the rows of the matrices. Here is the Möbius strip again with lines only.

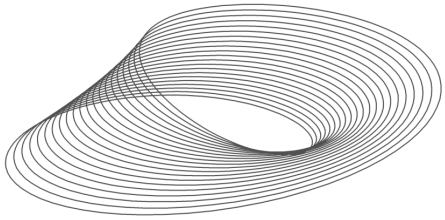


Figure 7.6: Möbius strip

Points in 3D are drawn with the parameter `>points`. We need three vectors for the coordinates of the points. The following commands generate a cloud of randomly normal distributed values in space.

```
>A:=normal(3,10000); plot3d(A[1],A[2],A[3],>points,style=".");
```

The following plot shows a Brownian motion, which we get with a cumulative sum of the coordinates. We show the plot as an anaglyph plot. You need red/cyan glasses to see the fascinating 3D effect.

```
>A:=normal(3,10000); B:=cumsum(normal(3,1000));  ...
>plot3d(B[1],B[2],B[3],>wire, ...
>   linewidth=1,>anaglyph,zoom=3.5):
```

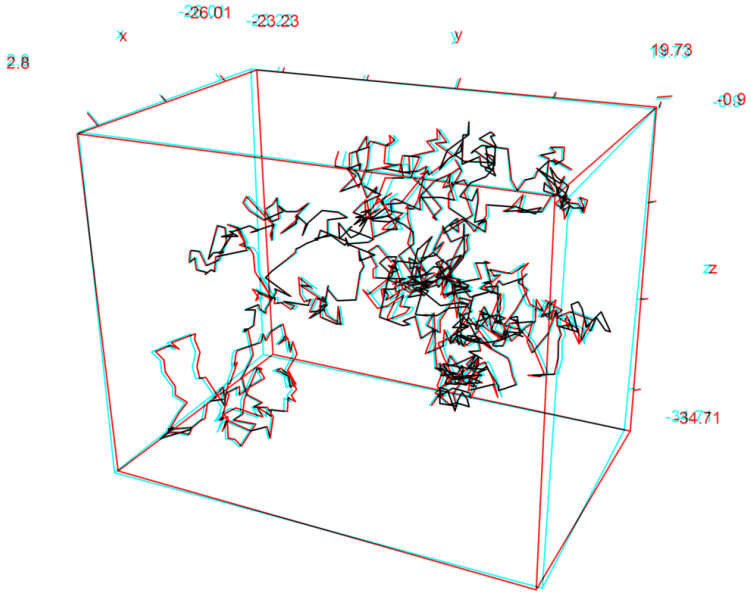



Figure 7.7: Brownian path as anaglyph

There are many more tricks for 3D plots in EMT. For details, see the tutorial about 3D plots and the many examples of such plots in the notebooks that are installed with EMT.

## 7.3 Solving Equations of Several Variables

To solve non-linear systems of equations of several variables, EMT has the fast Newton algorithm and the Broyden method. The stable Nelder-Mead or the Powell algorithms for minimization can also be used. Moreover, there is a variant of the Newton algorithm delivering guaranteed inclusions.

### Example

Let us solve the system of equations

$$x^2+y^2=1, \quad y=e^{-xy}.$$

For this, we write a function $f$ as

$$[x,y]\mapsto[x^2+y^2-1, y-\exp(-xy)].$$

We seek the zeros of this function. First we plot the level line for the value 0 of both functions. In the plot we can clearly see two solutions of the system of equations. We ignore the obvious solution $x=0$, $y=1$.

```
>f1 &= x^2+y^2-1; f2 &= y-exp(-x*y);
>plot2d(f1,r=1.2,level=0); plot2d(f2,level=0,>add):
```

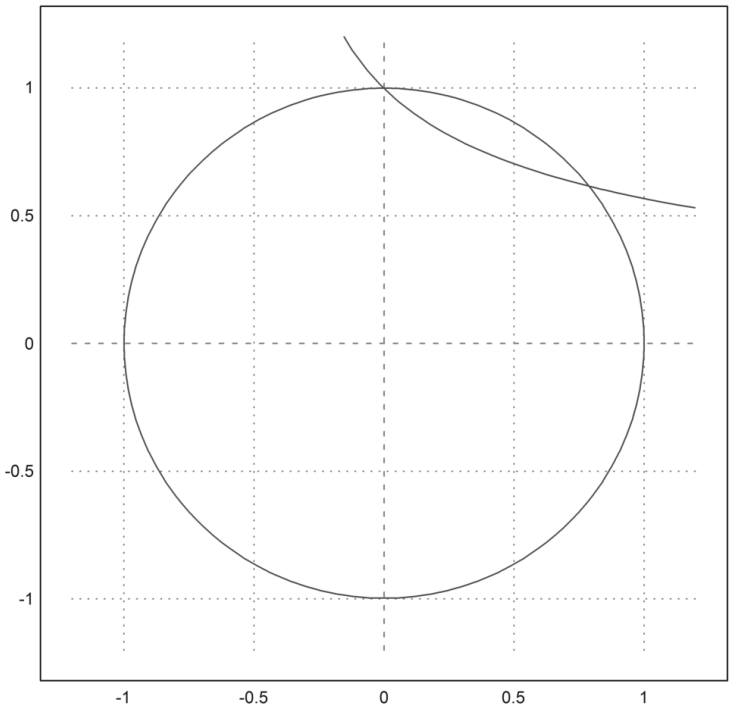



Figure 7.8: Solutions of two equations

The Broyden method works very well here. We need to define a vector valued function for `broyden`. We can define a symbolic function using our symbolic expressions.

```
>function f([x,y]) &= [f1,f2]
```

$$\begin{bmatrix} y^2 + x^2 - 1, & y - E^{-x\,y} \end{bmatrix}$$

```
>type f
 function f ([x, y])
 useglobal; return [y^2+x^2-1,y-E^-(x*y)]
 endfunction
```

Note that `f([x,y])` is a very specific function definition which allows `f` to be used for a vector as in `f(v)` or for two values as in `f(x,y)`. We could define `f` numerically only in the following way.

```
>function f(v) := [v[1]^2+v[2]^2-1,v[2]-exp(-v[1]*v[2])];
```

Now we can start the Broyden algorithm and find the solution in the first quadrant. For a starting point we take $x = 1$, $y = 0$.

```
>longest broyden("f",[1,0])
      0.7880470947327539      0.6156149579755725
```

For the Newton method, we need the Jacobian matrix of derivatives. We can compute it with Maxima at compile time of `Df`.

```
>function Df([x,y]) &= jacobian([f1,f2],[x,y])
```

$$\begin{bmatrix} 2\,x & 2\,y \\ -x\,y\,E^{-x\,y} & -x\,y\,E^{-x\,y}+1 \end{bmatrix}$$

```
>type Df
 function Df ([x, y])
 useglobal;
 return matrix([2*x,2*y],[y*E^-(x*y),x*E^-(x*y)+1]);
 endfunction
```

The function `newton2` implements the Newton method, and the function `inewton2` implements the interval Newton method, which provides a guaranteed and good inclusion of the result.

```
>longest newton2("f","Df",[1,0])
      0.7880470947327539      0.6156149579755725
>inewton2("f","Df",[1,0])
 [ ~0.788047094732753,0.7880470947327548~,
 ~0.61561495797557209,0.61561495797557297~ ]
```

Another stable method is the Nelder-Mead algorithm, which minimizes functions $f : \mathbb{R}^n \to \mathbb{R}$. We use it for the function $f(v) = \|v\|$.

```
>function g(v) := norm(f(v))
>longest neldermin("g",[1,0])
      0.7880470947328121      0.6156149579760124
```

The algorithm is not very fast. For very high dimensions, it cannot be recommended.

There is also a generalization of the Newton method for functions from a lower into a higher dimension, the Levenberg-Algorithm. This algorithm minimizes $\|v\|$ for $v = f(w)$. It needs a Jacobian too, which we can compute with Maxima. For a test, we compare with the Nelder-Mead algorithm.

```
>expr &= [x^3+y,x-y,x+y-1] // function from two to three variables

                              3
                       [y + x , x - y, y + x - 1]

>function f([x,y]) &= expr

                              3
                       [y + x , x - y, y + x - 1]

>function Df([x,y]) &= jacobian(@expr,[x,y])

                              [    2      ]
                              [ 3 x    1  ]
                              [           ]
                              [  1    - 1 ]
                              [           ]
                              [  1     1  ]

>longest nlfit("f","Df",[2,1])
      0.4063570133455459      0.3109666275848427
>function g([x,y]) := norm(f([x,y]))
>longest nelder("g",[2,1])
      0.4063568110959406      0.3109663066957633
```

Solutions of non-linear systems of equations are also used for best fits in regression analysis. We provide a typical example, where a function with a linear part and an exponential part is fitted to data in least square sense.

```
>x=-4:4; y=exp(x)-x+normal(size(x))/3;
>function model(x,[a,b,c,d]) := a+b*x+c*exp(d*x);
>p=modelfit("model",[2,2,2,2],x,y)
 [-0.496495,  -1.10157,  1.24337,  0.950208]
>plot2d(x,y,>points); plot2d("model(x,p)",>add):
```

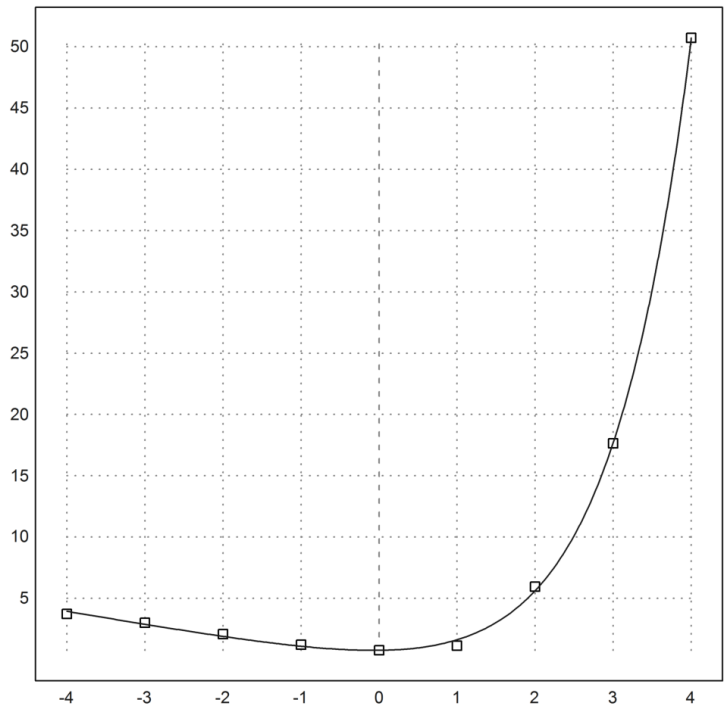



Figure 7.9: Nonlinear Fit

Depending on the model, the process is very sensitive to starting values. The Nelder-Mead method is usually more stable than the algorithm of Powell.

## 7.4 Implicit Functions

EMT can draw the solutions of $f(x,y)=c$ for one or more values $c$ (contour plots), and also implicit surfaces with an equation $f(x,y,z)=0$. Use `plot2d` with the parameter `levels=...` for this, or `plot3d` with the parameter `implicit=...`.

For `plot2d`, the values of $c$ are either a number, or a row vector of values. The parameters `levels="auto"` or `>contour` will use equal spaced values. The additional parameter `>hue` indicates the values of the functions with a shading. Black means low values, and white means high values. The color can be set with `color`. There is also a spectral scheme with `>spectral`.

Alternatively, the function `plot3d` can be used with `>contour`. This yields a three dimensional plot with shading and contour lines.

Implicit plots of $f(x,y)=c$ can show ranges of levels $c_1 \le f(x,y) \le c_2$ in 2D and 3D. These ranges are given as matrices with the lower values in the first row and the upper values in the second row. Thus we can mark a region which satisfies a side condition depending on one function.

```
>plot2d("2*x^2+y^2+x*y+x+2*y",r=3,levels=[1;2], ...
>  style="/",color=green,grid=6):
```

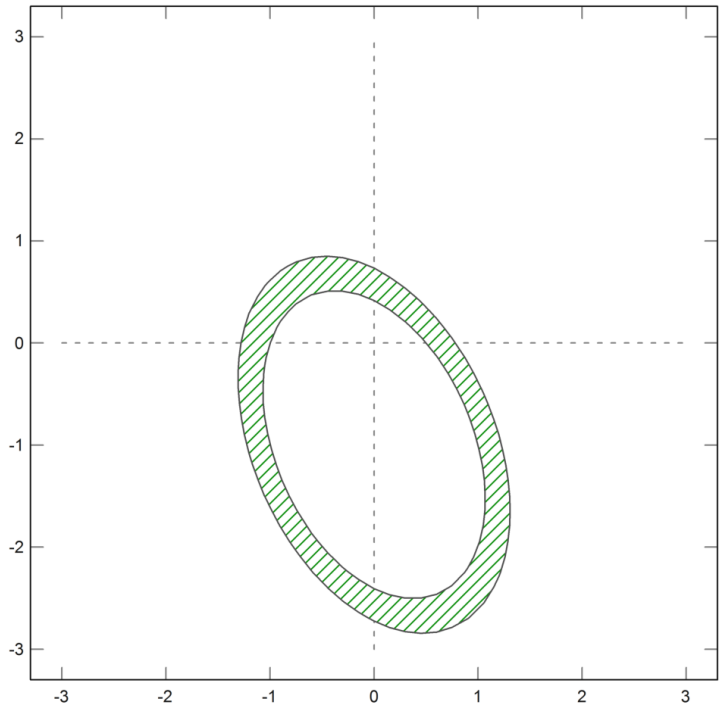



Figure 7.10: Solutions of $1 \le 2x^2+y^2+xy+x+2y \le 2$

### Example

For an example, we investigate the function $f(x,y) = x^y - y^x$ for $x, y > 0$. First we plot the solution of the equation $x^y = y^x$ in the range $0 \le x, y \le 5$.

```
>function f(x,y) := x^y-y^x;
>plot2d("f",a=0,b=5,c=0,d=5,n=100, ...
>  level=0,>hue,>spectral,contourcolor=red,contourwidth=3):
```

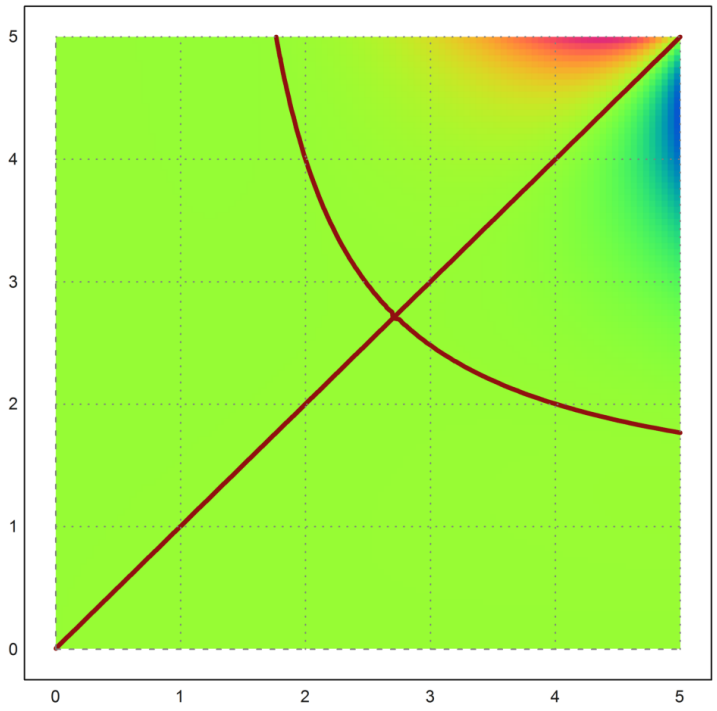



Figure 7.11: Solutions of $x^y = y^x$

It is interesting to see the 3D plot of $f(x,y)$ in the region $2 \le x, y \le 3$. Add user interaction, so that the user can inspect the saddle point form all sides. The default `>contour` without specification of level values will plot ranges of levels.

```
>plot3d("f",xmin=2,xmax=3,ymin=2,ymax=3,>user,>spectral,>contour):
```

You can find more examples, also for surfaces, in the introduction notebooks.

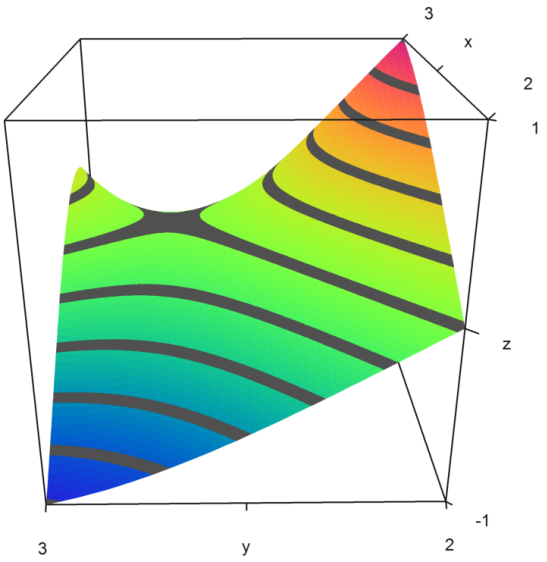



Figure 7.12: Plot of $x^y - y^x$

# Chapter 8

# Programming Euler

## 8.1 Functions

In the introduction, we have already met one-line functions. This kind of function definition is the most elementary and often good enough. One-line functions can use global variables automatically. Alternatively, variables can be passed as arguments.

```
>function f1(x) := m*x+t
>t:=1; m:=-0.25; f1(3)
   0.25
>function f2(x,m,t) := m*x+t
>f2(0:4,-0.25,1)
   [1,  0.75,  0.5,  0.25,  0]
```

EMT can assign default values to parameters (default parameters). Those parameters need no value, when the function is called.

```
>function logb(x,base=10) := log(x)/log(base)
>logb(100)
   2
>logb(1024,2)
   10
```

Multi-line functions begin with `function` and end with `endfunction`. They must contain a `return` statement somewhere if a value must be returned.

```
>function f3 (x)
$  if x<0 then return x^3
$  else return x^2
$  endif
$endfunction
>plot2d("f3",-1,1):
```

If a multi-line function does not return a value, it returns the value `none`, a string consisting of the ascii character 1, which will never print.

It is possible to enter multi-line functions in the text window, either directly, or with the help of the internal editor. To use the internal editor, press F9 in the `function` line. Have a look into Section 3.7 for details.

When replaying the notebook, the user has to go through the function definition line by line. To avoid this, end the `function` line with three dots `...` This will read the function with one return. To make things easier, the `...` string is automatically appended by the internal editor.

Besides with the internal editor, a function can be edited by clicking into it. Use `Ctrl-Return` to split a line or insert a new line, and `Ctrl-Return` to join a line to the previous line. Moreover, you can use `Alt-Back` to delete a line or `Alt-Insert` to insert a new line.

To start editing in the notebook window at the first definition of the function, press `Ctrl-Return` at the end of the line starting with `function`. This will add `...` at the end of the line. To exit the editing press `Return` at the last line. The editor will add `endfunction` for you.

It might be more comfortable to edit all the functions needed in a notebook in an external file. Edit the file with the external or internal editor, save it with the `filename.e` extension, and load it into EMT with `load filename`. The file should be in the same directory as the notebook. Otherwise, the load command must provide the complete path to the file.

You can use any simple text editor to edit EMT files. Euler comes with `je`, a Java editor. There is also an internal editor. Both editors will use a temporary file in the user directory if they are started on empty lines. They will use the file of the `load` command if the current line starts with such a command. To start the internal editor for editing files, press F9 in a line starting with `load`, to start the external editor use F10.

### Example

Enter the following text into the internal editor (or paste it from the documentation), and press OK. Simply press F9 or F10 in an empty line to start the editor with a temporary file. The text between comment and endcomment will be printed as a comment while loading.

```
comment
Definition of f3(x).
endcomment

function map f3 (x)
    if x<0 then return x^3
    else return x^2
    endif
endfunction
```

This is how the notebook looks.

```
>load "C:\Users\ReneEMTEulerTemp.e"
  Definition of f3(x).
>f3(-2:2)
  [-8,  -1,  0,  1,  4]
```

Within an EMT function, comments can be entered at any place with //, or at the beginning of the line with ##. The ## comment lines directly at the start of the function definition are used as help lines. These lines are printed with help functionname.

### Example

We enter the following code with the internal editor. To do this, type function into an empty line, and press F9. In the internal editor, paste or type the following lines. End the editor with the OK button, and press return to define the function.

```
function map sinctest (x) ...
## Computes sin(x)/x taking care of x=0
    if x~=0 then return 1
    else return sin(x)/x
    endif
endfunction
```

sinctest then works just like any other function, and help sinctest prints help for this function.

```
>plot2d("sinctest",0,2pi);
>help sinctest
   sinctest is an Euler function.

   function map sinctest (x)

    Entered from command line.
    Computes sin(x)/x taking care of x=0
```

All pre-defined EMT functions contain help lines. The first help line should contain a function summary, since this line is shown in the status line as immediate help, after the user has entered the opening bracket for the function parameters. Moreover,

```
>help util.e
```

will print all function definitions in the EMT file util.e, including the first line of the help. You can also enter util.e in the help window to see this information.

An EMT file with functions can also contain items for the user menu. These items are commands the user can paste at the current cursor positions. The commands can contain placeholders like ?expression. If the user starts typing in front of a placeholder the placeholder is removed. The cursor right key positions the cursor to the next placeholder. The syntax for the menu is as follows.

```
submenu My Functions
addmenu myfunction("?expression",?xstart)
addmenu ?variable=solve("?expression",?xstart)
```

The submenu can be omitted. In this case a submenu with the file name will be used. Weather the command is a full command as in the second example or only a function is a matter of taste.

Euler uses typeless variables and parameters by default. To avoid strange error messages and bugs, it is possible to require a type for a parameter of a function. Enter the type after the variable separated by :.

In the following example, we do not want to use the function for complex numbers or intervals, since it will no longer work properly.

```
>function map signum (x:real) ...
$  if x<0 then return -1;
$  elseif x==0 then return 0;
$  else return 1;
$  endif;
$endfunction
>signum(I)
 Function signum needs a real for x
 Error in map.
 Error in:
 signum(I) ...
          ^
```

There are the following types for parameters.

| Further Information | |
|---|---|
| `real` | real numbers |
| `complex` | complex or real numbers |
| `interval` | intervals or real numbers |
| `numerical` | any number |
| `integer` | integer numbers |
| `positive` | positive numbers |
| `nonnegative` | non-negative numbers |
| `scalar` | numbers, but not matrices or vectors |
| `vector` | row vectors of numbers |
| `column` | column vectors of numbers |
| `natural` | short for `nonnegative integer scalar` |
| `index` | short for `positive integer scalar` |
| `indices` | short for `positive integer vector` |
| `number` | short for `real scalar` |
| `string` | strings |
| `cpx` | compressed sparse matrices |

Some of these types can be combined. E.g., `real scalar` means a real number, not a complex number or an interval, which is also not a vector or a matrix. `string vector` allows only vectors of strings.

Overwriting built-in or pre-defined functions is not allowed by default. With the keyword `overwrite`, this is possible nevertheless. To access a built-in function with the same name, prepend an underscore.

```
>function overwrite sin(x) := _sin(rad(x))
```

```
>sin(45)
   0.707106781187
>_sin(45°)
   0.707106781187
```

Redefining built-in functions must be done with care. The redefintion of `sin`, which works for degrees, is not recommended, of course. It is wiser to use a separate name line `sindeg` for this purpose.

To write a comment on a function im Maxima there are special comment functions. For Maxima functions, use `maximafunction`.

```
>function D(u,x,y) &&= diff(u,x,2)+diff(u,y,2)

                    diff(u, x, 2) + diff(u, y, 2)

>maximafunction f(u,x,y) ...
$  ## Laplace of u
$endfunction
>help &f
 function &f (u, x, y)
 Entered from command line.
 Laplace of u
>:: D(realpart((x+I*y)^4),x,y)

                                    0
```

## 8.2 Functions and the Matrix Language

Many functions work for matrices automatically. This is so because expressions in EMT work for matrix and vector input. However, if the function contains control structures it will usually not work for vector input.

EMT can vectorize a function to matrix arguments. The easiest way is to append `map` to the function name as in `fmap(v)`. Alternatively, there is a function `map("f",...)`, which maps the function `f` to the other arguments.

But a function can also be defined as a mapping function. In this case, the function must have scalar parameters, and must return a scalar, not a vector. EMT will automatically map the function to the elements of the matrix.

```
>function map f(x:number) ...
$  if x<0 then return x^2
$  else return x^3
$  endif
$endfunction
>f(-1:0.5:1)
   1  0.25  0  0.125  1
```

The function would not work for vector input without mapping. The reason is, that the `if` does not branch down to the elements of the vector `x`. By default, `if` conditions for vectors are deprecated, and need to be replaced by e.g. `all(x>0)`. This avoids errors with vector arguments for non-vectorized functions. But it does not help in mapping to the elements.

By the way, conditions on vectors make sense in many cases, e.g. for a fixpoint iteration, as in the following example.

```
>A=[0.2,0.5;0.4,0.6]; b=[1;1];
>function fixpointiter (x) ...
$  global A,b;
$  repeat
$  xnew=A.x-b;
$  if all(xnew~=x) then return xnew endif;
$  x=xnew;
$  end;
$endfunction
>fixpointiter([0;0])
          -7.5
           -10
```

Arguments can be protected from mapping with the semicolon. This works either at runtime or at compile time of the function.

### Example

We want to evaluate with different polynomials depending on $x > 0$ or $x \leq 0$. The evaluation points and the polynomials shall be parameters of `f`. Since the function does not work for vector values, we define it using `map`. But we do not want to map `f` to the polynomial coefficients, of course. The semicolon in the list of parameters prevents the mapping.

```
>function map f(x:scalar; p:vector, q:vector) ...
$  if x>0 then return polyval(p,x)
$  else return polyval(q,x)
$  endif
$endfunction
>plot2d("f(x,[0,1,2],[0,1])",-1,1); ...
>textbox("p(x)",x=0.3,y=0.1,w=0.2,>center);  ...
>textbox("q(x)",x=0.8,y=0.1,w=0.2,>center):
```

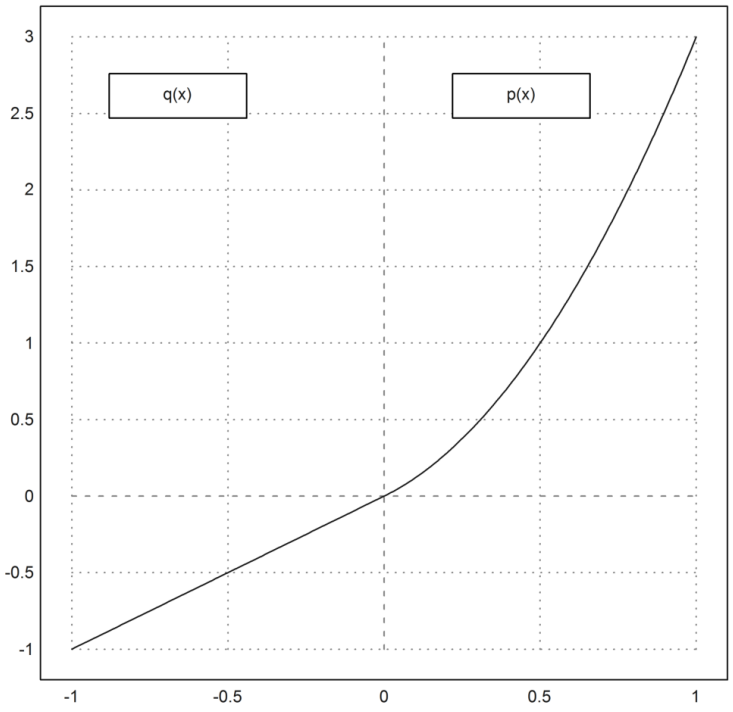



Figure 8.1: Function with two branches

This example could also be done in the following way. We compute the functions everywhere and use logical vectors to combine the functions. Note that `true=1` and `false=0` in EMT.

```
>function f(x,p,q) := (x>0)*polyval(p,x)+(x<=0)*polyval(q,x);
```

### Example

The quick computation of Chebyshev polynomials uses different algorithms for $x < 1$, $x \in [-1,1]$ and $x > 1$. Note, that the following function is already contained in EMT as `cheb`.

```
>function overwrite map cheb (x: number, n: nonnegative integer)
$     signum=-mod(n,2)*2+1;
$     if x>1 then
$         w=(x+sqrt(x^2-1))^n;
$         return (w+1/w)/2;
$     elseif x<-1 then
$         w=(-x+sqrt(x^2-1))^n;
$         return signum*(w+1/w)/2;
$     else
$         return cos(n*acos(x));
$     endif;
$endfunction
```

Now we can compute many Chebyshev polynomials of different degrees at once combining a row and a column vector. The vectorization with the keyword `map` handles this for us.

```
>x:=-1.1:0.01:1.1; n:=(1:4)'; plot2d(x,cheb(x,n)):
```

## 8.3 Multiple Return Values

Often we need to return multiple return values from a function. We do not want to use global variables for this. Instead, EMT can return more than value with the `return` statement. The syntax for this uses curly brackets around comma separated values.

Of course, these return values can be assigned to multiple variables. The list of variables is enclosed by curly brackets too.

**Example**

We sort two values, and return both in the correct order.

```
>function sort2 (a,b) ...
$    if a<b then return {a,b}
$    else return {b,a}
$    endif
$  endfunction
>{x,y}:=sort2(2,3); x, y,
   2
   3
>{x,y}:=sort2(3,2); x, y,
   2
   3
```

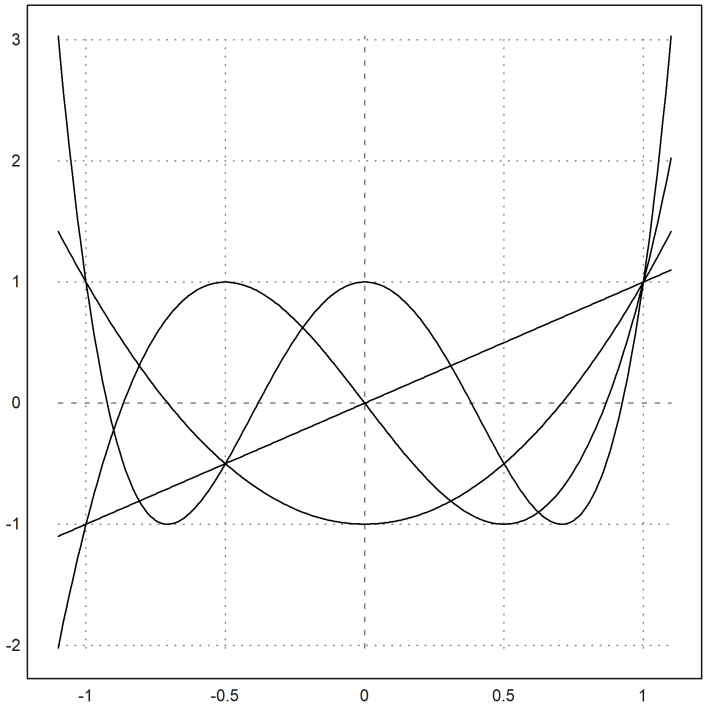



Figure 8.2: Chebyshev polynomials $T_1$ to $T_4$

### Example

To sort a vector EMT has the `sort` function. The function returns the sorted vector, and the new indices of the elements, such that in the following example `i[ind]` is the sorted vector `j`. This is useful to permute any other vector in the same way.

```
>i:=shuffle(6:10)
   [7,  10,  8,  6,  9]
>{j,ind}=sort(i); j
   [6,  7,  8,  9,  10]
>ind
   [4,  1,  3,  5,  2]
>i[ind]
   [6,  7,  8,  9,  10]
```

It is a bit tricky to get the inverse permutation. However this can be done by sorting the indices.

```
>{k,invind}=sort(ind);
>j[invind]
   [7,  10,  8,  6,  9]
```

| Further Information | |
|---|---|
| find | Finds a number in a sorted vector. |
| lexsort | Sort the rows of a matrix in lexicographical order. |

## 8.4 Global Variables

Inside a function, only the local variables and the parameters are visible. Local variables are the variables declared in the function, or the parameters with default values.

```
>C = 2pi
 6.28318530718
>function test (x) ...
$return C*x
$endfunction
>test(5)
 Variable C not found!
 Use global or local variables defined in function test.
 Try "trace errors" to inspect local variables after errors.
 test:
     return C*x
 Error in:
 test(5) ...
        ^
```

Global variables defined on the command line are not visible in multi-line functions. An exception are global variables ending with $, which are visible from every function. Those variables should be used for units. This is used to define units.

```
>C$ = 2pi
   6.28318530718
>5C
   31.4159265359
>function test (x) ...
$  return C$*x
$  endfunction
>test(5)
   31.4159265359
```

Moreover, any global variable can be changed with `setglobal var` to be visible in functions.

To access invisible global variables, use the `useglobal` command in the function, or the command `global varname`. Note that one-line functions always contain the `useglobal` statement. So they can see global variables.

```
>function test(x) := C*x
>type test
   function test (x)
   useglobal; return C*x
   endfunction
```

## 8.5 Reference Parameters

EMT passes variables by reference. However, it is not possible to assign a new value to a variable which is passed as an argument to the function. This will always create a new local variable with the same name.

### Example

In the following example we set the diagonal of a global matrix to some given value.

```
>function setA (A,s) ...
$    A=setdiag(A,0,s);
$  endfunction
>A:=[1,2;3,4]
               1             2
               3             4
>setA(A,4);
>A // is unchanged!
               1             2
               3             4
```

You could of course access the variable `A` with `global A` instead of using a parameter.

To circumvent this restriction for parameters, start the name of the parameter with a `%`. In this case, changing the parameter will actually change the value of any variable passed to it.

```
>function setA (%A,s) ...
$      %A=setdiag(%A,0,s);
$    endfunction
>A:=[1,2;3,4]
                1              2
                3              4
>setA(A,4);
>A
                4              2
                3              4
```

There is an exception to this rule. A matrix can be changed even if the name of the parameter does not start with %. I.e., new values can be assigned to the matrix elements or to sub-matrices. The behavior is the same as in other programming languages.

```
>function test (v) ...
$  i=nonzeros(v==4);
$  v[i]=0;
$  endfunction
>w=1:6
   [1,  2,  3,  4,  5,  6]
>test(w); w
   [1,  2,  3,  0,  5,  6]
```

One of the functions in EMT that uses this feature is `pivotize`. It performs one typical step of the Gauß algorithm.

```
>fracformat(10);
>A=[1,2,3,4;5,6,7,8;9,10,11,12]
           1         2         3         4
           5         6         7         8
           9        10        11        12
>pivotize(A,2,2)
        -2/3         0       2/3       4/3
         5/6         1       7/6       4/3
         2/3         0      -2/3      -4/3
```

Here the value of the matrix `A` is indeed changed. The function could also have been implemented in a way such that we had to write an assignment of the result after we called it.

```
>A = pivotize(A,2,2);
```

In fact many built-in functions of EMT work this way. E.g., setdiag sets a diagonal of a matrix to a new value. It does not change the matrix, but it returns a new copy of the matrix. To implement such a function have a look at the following example. It makes a copy of the matrix before it does any changes.

```
>function zerodiag (A) ...
$  B=A;
$  loop 1 to min(rows(A),cols(A));
$     B[#,#]=0;
$  end;
$  return B;
$  endfunction
>A=[1,2;3,4]; zerodiag(A)
               0             2
               3             0
>A
               1             2
               3             4
```

## 8.6 Default Values

EMT functions can have default values for parameters. If there is no argument for a parameter with a default value, the default value is used. Parameters with default values can also be set with assigned arguments.

```
>function f(x,a=4) := a*x
>f(5)
   20
>f(5,6)
   30
>f(5,a=7)
   35
>f(5,7,b=6)
   Argument b not in parameter list of function f.
   ...
```

Assigned arguments can be passed to a function only to overwrite default values of parameters. The reason for this restriction is to prevent typos in parameter names which happens easily. If the variable name is not in the parameter list an assigned argument is not allowed.

But it is possible to force a variable in the function with `:=`, even if the variable was not in the list of variable with default values.

```
>function f(x) := x*a
>f(6,a:=5)
   30
```

If the function is defined with the modifier `allowassigned` this is possible with a simple `=`.

An exception to this rule is the evaluation of an expression, which allows assigned arguments for variables in the expression. We already know that the variable names `x`, `y` etc. are taken from the list of arguments automatically.

```
>"a*x"(5,a=7)
   35
```

Another important difference between expressions and functions is that expressions can see global variables.

The utility functions `plot2d` and `plot3d`, as well as many other functions in EMT, use a mixture of parameters with default values and assigned parameters.

```
>plot2d("x^3-x",color=green);
```

## 8.7 Control Structures

Control structures change the flow of execution in a function. There are two types: conditional branches and loops.

Conditional branches are possible with the `if` statement. `if` always has to end with `endif`. After the condition, there is an optional, but recommended `then`. An alternative branch is possible with `else`. `elseif` works like an `if` statement in the `else` branch. But only one `endif` is needed.

### Example

We can implement the signum function in EMT as follows. Note, that EMT has this function already as sign. When comparing with 0, it is wise to use ~=. This comparison uses the internal epsilon.

```
>function signum (x)
$  if x~=0 then return 0
$  elseif x>0 then return 1
$  else return -1
$  endif
$endfunction
```

If conditions are connected with and or or, they are evaluated only as much as necessary. This is called a condition shortcut.

```
>function test (a,b) ...
$  if a^3+1>=0 and b<sqrt(a^3+1) then return true
$  else return false
$  endif;
$endfunction
```

In this example $\sqrt{a^3+1}$ is not computed, if $a^3+1<0$.

These logical operators can only be used in conditions, such as if. To compute a logical expression, use the ordinary logical operators &&, || of EMT.

```
>function isbetween(x,a,b) := (x>=a) && (x<=b)
```

There are also several loops in EMT. The basic repeat loop is an eternal loop. To break it, use the break statement, preferably inside an if structure.

```
>function s1
$  n:=1
$  s:=0
$  repeat
$    s:=s+n
$    n:=n+1
$    if n>10 then break; endif;
$  end;
$  return s;
$endfunction
```

The `until` statement breaks the loop if the condition is true. Note that the loop is continued after `until` if the condition is not true. There has to be an `end` after each `repeat`.

```
>function f(x) ...
$  repeat
$     xnew=(x+2/x)/2;
$     until x~=xnew;
$     x=xnew;
$  end;
$  return xnew;
$  endfunction
>longest f(1)
          1.41421356237469
```

There is also a `while` statement which works like `until`. But the condition is reversed.

```
>function f(x) ...
$  repeat
$     xnew=(x+2/x)/2;
$     while abs(x-xnew)>epsilon;
$     x=xnew;
$  end;
$  return x;
$  endfunction
>longest f(1)
          1.41421356237469
```

A loop can contain several `until`, `while`, `break` or `continue` (jumps to the start of the loop).

The `loop` loop is an integer loop running between two integer values. The `for` loop counts a variable (loop index) with an optional step value from the starting value to an end value. It can also assign the loop index to all values of a vector.

For a demonstration, we use the loops in a command line now. This is possible within a single command line or a multi-line.

```
>v=1:10;
>s=0; loop 1 to 10; s=s+v[#]; end; s,
   55
```

```
>s=0; for i=1 to 10; s=s+v[i]; end; s,
   55
>s=0; for i=10 to 1 step -1; s=s+v[i]; end; s,
 55
>s=0; for x=v; s=s+x; end; s,
   55
```

Of course loops can be put one into the other. This is called a double loop. If the inner loop is broken with a `break` statement, the outer loop will continue to run.

### Example

For a more complicated example, we write a function, which prints the numbers till 3000 as Roman numbers.

```
>function roman (x : positive integer) ...
$    if x>3000 then error("x>3000"); endif
$    s:="";
$    repeat
$      if x>=1000 then s:=s|"M"; x:=x-1000;
$      elseif x>=900 then s:=s|"CM"; x:=x-900;
$      elseif x>=500 then s:=s|"D"; x:=x-500;
$      elseif x>=400 then s:=s|"CD"; x:=x-400;
$      elseif x>=100 then s:=s|"C"; x:=x-100;
$      elseif x>=90 then s:=s|"XC"; x:=x-90;
$      elseif x>=50 then s:=s|"L"; x:=x-50;
$      elseif x>=40 then s:=s|"XL"; x:=x-40;
$      elseif x>=10 then s:=s|"X"; x:=x-10;
$      elseif x>=9 then s:=s|"IX"; x:=x-9;
$      elseif x>=5 then s:=s|"V"; x:=x-5;
$      elseif x>=4 then s:=s|"IV"; x:=x-4;
$      elseif x>=1 then s:=s|"I"; x:=x-1;
$      else break;
$      endif;
$    end;
$    return s;
$  endfunction
>roman(1968)
   MCMLXVIII
```

Functions can call themselves recursively (recursive functions). This yields very elegant programs. The stack for recursive calls is limited, however.

**Example**

We breakup a string in substrings separated by a blank. The function `strfind` finds a string in another string. If the string contains a blank, we print the first part, and proceed to scan the second with a recursive call.

```
>function breakup (str) ...
$  n=strfind(str," ");
$  if n>0 then
$    substring(str,1,n-1),
$    breakup(substring(str,n+1,-1));
$  else
$    str,
$  endif;
$endfunction
>breakup("This is a test");
 This
 is
 a
 test
```

We remark that this kind of splitting of a string can be done with `strtokens`. Moreover, there are `strxfind` and `strxrepl` for a search with regular expressions.

| Further Information | |
|---|---|
| `and, or` | Connects conditions in `if` (with shortcut). |
| `&&, \|\|` | Boolean "and" and "or", also for vectors. |
| `error(String)` | Error message and function abort. |
| `index` | Alternative for the loop index `#`. |

## 8.8 Functions as Parameters

Many functions in EMT expect an expression or a function name. Examples are `plot2d` or `plot3d`. In this section, we explain how to write those functions.

To evaluate a function given by name, simply use a variable `fvar` containing the name, and call the contained function as in `fvar(...)`. To check if a string variable contains the name of a function, you can use `isfunction`.

To evaluate an expression contained in a string, you can use `evaluate` or `expr()`.

```
>function test(x,f$)
$    return f$(x)
$  endfunction
>test(2,"sin")
   0.909297426826
>test(2,"x^2")
   4
```

Suppose `f` is called inside `g(f)`, where `f` is given as a string parameter to `g`. Then we need a way to pass additional parameters to `f` via `g`. This is done using the `args()` function and the semicolon `;` (semicolon parameters). `args()` simply denotes all arguments after the semicolon. As a rule semicolon parameters must be given before any assigned parameters.

In the following example, we pass the parameter `5` of `test` to `f`. The actual call to `f` is `f(2,5)`.

```
>function test (x,f$)
$    return f$(x,args())
$  endfunction
>function f(x,a) := a*x
>test(2,"f";5)
   10
```

The following commands work well, since `a` is a global variable.

```
>a:=2; plot2d("x^2-a",1,2);
```

However, the same commands will not work inside a function, unless `a` is defined globally. We need to pass the local variable `a` as a semicolon parameter to the plot function. Of course, values cannot be passed to expressions in the same way, since expressions look for variables by name. But variables can be passed by reference to the function evaluating the expression.

```
>function test (a) ...
$    plot2d("x^2-a",0,2;a);
$  endfunction
>test(4):
```

### Example

We compute the Fourier coefficients of functions numerically. Those coefficients are defined by

$$a_k = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(kx)\,dx$$

For the integral, we use the adaptive Gauß method as implemented in `integrate`.

We define a function, evaluating $f(x)\cos(kx)$. The function name $f$ and the degree $k$ will be given to `integrate` as additional parameters. The function `integrate` will pass these parameters to the function it integrates. Using `map`, we compute the coefficients from $k=1$ to $k=5$.

```
>function fcos (x,f,k) := f(x)*cos(k*x)
>function map fcoeff(f,k) := integrate("fcos",-pi,pi;f,k)/pi
>function f(x) := x^2
>fracprint(fcoeff("f",1:5));
   [-4,  1,  -4/9,  1/4,  -4/25]
```

The result can be checked with Maxima.

```
>&create_list(integrate(x^2*cos(n*x),x,-%pi,%pi)/%pi,n,1,5)
```

$$\left[-4,\ 1,\ -\frac{4}{9},\ \frac{1}{4},\ -\frac{4}{25}\right]$$

Instead of semicolon parameters we can also use collections to pass variables to functions. The evaluation of a collection as a function works if the first element of the collection is the name of a function or an expression.

```
>c={{"sin",4}}; c()
 -0.756802495308
>c={{"a*x^2",4,a=2}}; c()
 32
```

This can be used to pass a function which requires additional parameters to function such as `plot2d` or to an numerical algorithm instead of semicolon parameters.

```
>function f(x,a) := (x-a)/(x+a)
>integrate({{"f",4}},0,2) // with a collection
   -1.24372086487
>integrate("f(x,4)",0,2) // test
   -1.24372086487
>integrate("f",0,2;4) // with semicolon parameter
   -1.24372086487
```

## 8.9 Maxima at Compile Time

Often, we want to use Maxima to compute derivatives or other expressions. This can be done using symbolic expressions, as explained in the chapter about Maxima.

But it might be much better, to do this during the definition of a function. To call Maxima at compile time, use the `&:...` syntax. The expression in the string will be evaluated by Maxima and the result will be inserted into the function body.

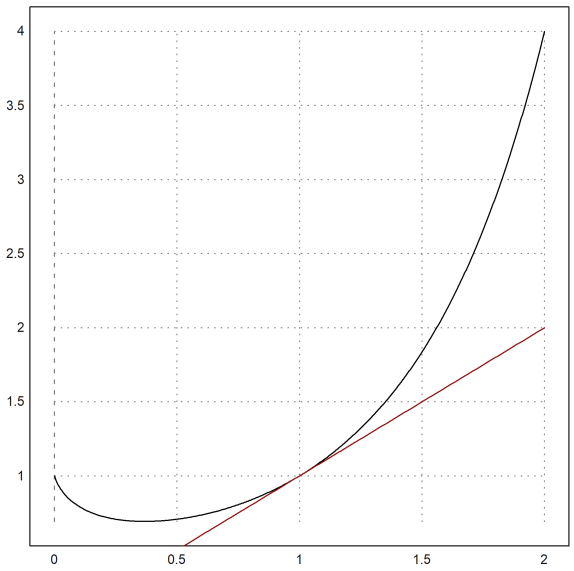



Figure 8.3: $x^x$ and Tangent

### Example

We write a function for the tangent $g(x)$ to the function $x^x$ in a point $s$. The derivative of the function is computed while the function is defined. To see this, have a look at the internal print of g.

```
>function g(x,s) ...
$    return &:s^s+diff(s^s,s)*(x-s)
$  endfunction
>type g
   function g (x, s)
   return s^s*(log(s)+1)*(x-s)+s^s
   endfunction
>plot2d("x^x",0,2); plot2d("g(x,1)",color=red,>add):
```

A symbolic one-line function is a shortcut for this. These functions evaluate the function expression in Maxima at compile time.

### Example

The Newton method requires a function, which returns the Jacobian of a function. We can define such a symbolic function using the following syntax.

```
>fx &= x*y-1; fy &= x-y;
>function f([x,y]) &= [fx,fy]
```

$$[x\ y - 1,\ x - y]$$

```
>function Df([x,y]) &= jacobian([fx,fy],[x,y])
```

$$\begin{bmatrix} y & x \\ 1 & -1 \end{bmatrix}$$

```
>newton2("f","Df",[2,1])
   [1,  1]
```

To create such a function for varying expressions, we cannot evaluate at compile time. However, the Maxima can be called with `&"..."` at run time. To use the expression `f` inside the call to Maxima use the syntax `@f`.

```
>function g(t,x,f) ...
$    return f(x)+&"diff(@f,x)"(x)*(t-x)
$  endfunction
>g(0.5,1,"x^x")
   0.5
>expr:="x^x";  ...
>plot2d("g",0,1;1,expr,color=red);  ...
>plot2d(expr,>add):
```

We need to call Maxima with a given expression at each call of the tangent function. Note, that this is not very effective. If possible, it is better to call Maxima beforehand once.

## 8.10 Error messages in Euler

EMT is a very flexible system, using an interpreter and typeless variables. The downside of this is that the user is sometimes faced with strange error messages. In this section, we like to explain some of theses problems.

The first versions of EMT used the Matlab style `v(i)` to access an element of a vector, as an alternative to `v[i]`. This opens the door to all sorts of errors. In fact, the results of `v(i)` depends on the type of `v`. For functions, strings and vectors there is a different meaning. In fact, an expression like `(3+4)(4+5)` worked, resulting in an empty vector.

EMT now has flags to prevent these problems. By default, these flags are set. The user must explicitly define the function as a relaxed function, if he wants to use the relaxed style. To do this, use the command `relax` at the start of the EMT file containing the definition of the function. This, of course, is not recommended.

Here some examples of error messages you might get.

```
>v=1:3; v(2)
   Unexpected "(". Index () not allowed in strict mode!
   In Euler files, use relax to avoid this.
   Error in:
   v=1:3; v(2) ...
              ^
>sin[0.4]
   sin is not a variable!
   Error in:
   sin[0.4] ...
           ^
>(1+4)(5+6)
   Unexpected "(". Index () not allowed in strict mode!
   In Euler files, use relax to avoid this.
   Error in:
   (1+4)(5+6) ...
             ^
>(1+4)[5+6]
   Index 11 out of bounds!
   Error in:
   (1+4)[5+6] ...
             ^
```

As mentioned in the section about parameters, untyped parameters may lead to very cryptic error messages, if a parameter of unexpected type is used.

```
>function myplot(f,a,b) := plot2d(f,a,b,color=10,thickness=3,title="test")
>myplot("x^2",-1,1);
>myplot("x^2",-1,1:4);
 Plot needs a real vector or matrix!
 plot2d:
     if auto then plotarea(xx,yy); endif;
 Try "trace errors" to inspect local variables after errors.
 myplot:
     useglobal; return plot2d(f,a,b,color=10,thickness=3,title="te ...
```

It is not really clear what happened. Thus it is much better to avoid this confusion with typed parameters.

```
>function myplot(f:string,a:number,b:number) ...
$    plot2d(f,a,b,color=10,thickness=3,title="test")
$  endfunction
>myplot("x^2",-1,1);
>myplot("x^2",-1,1:4);
   Function myplot needs a scalar for b
   Error in:
   myplot("x^2",-1,1:4); ...
                        ^
```

## 8.11 Debugging

EMT has some features which help to find errors in functions. First of all, you can trace a specific function or all functions.

```
>function test (x) ...
$  if x>0 then return 1;
$  elseif x==0 then return 0;
$  else return -1;
$  endif;
$  endfunction
>trace test;
   Tracing test
>test(-5)
   test: if x>0 then return 1;
   test: else return -1;
   -1
>trace test;
   No longer tracing test
```

The possible keyboard commands will be printed if you press a key which is not valid.

```
>trace on; test(-5)
   test: if x>0 then return 1;

   Keys :
   cursor_down   Single step
   cursor_right  Step over subroutines
   cursor_up     Go until return
   insert        Evaluate expression
   escape        Abort execution
   cursor-left   Stop tracing

   -1
>test(-5)
   -1
```

In this example, I pressed cursor-left. It is possible to evaluate an expression. The rules for visibility apply with respect to the current function.

```
>trace on;
>test(-5)
   test: if x>0 then return 1;
   Expression? >x
   -5
   -1
```

It is also possible to trace errors only. This will start tracing on errors and lets you evaluate expressions at that point. Press return on an empty expression to stop the evaluation.

```
>trace errors;
>test(1:10)
   Cannot use vectors for conditions, use all(...)!
   Maybe you need to vectorize the function with "map".
   Expression? >x
   [1,  2,  3,  4,  5,  6,  7,  8,  9,  10]
   Expression? >
   test:
       if x>0 then return 1;
```

In functions, `traceif` starts tracing conditionally.

```
>function test (x) ...
$  traceif x<0;
$  if x>0 then return 1;
$  elseif x==0 then return 0;
$  else return -1;
$  endif;
$  endfunction
>test(5)
   1
>test(-5)
   test: if x>0 then return 1;
   test: else return -1;
   -1
```

There is also the good old printing method to find errors. Maybe combined with throwing an error.

```
>function test (x) ...
$  if iscomplex(x) then x, error("x is complex"); endif;
$  if x>0 then return 1;
$  elseif x==0 then return 0;
$  else return -1;
$  endif;
$  endfunction
>test(5)
   1
>test(I)
   0+1i
   Error : x is complex

   Error generated by error() command

   Try "trace errors" to inspect local variables after errors.
   test:
       if iscomplex(x) then x, error("x is complex"); endif;
```

EMT can also call a function and catch errors using the function `errorlevel`.

```
>{res,err}=errorlevel("1/0");
>err
```

```
   1
>{err,res}=errorlevel("1/2");
>err, res
   0
   0.5
```

## 8.12 C Code

EMT can use external DLLs (dynamic link libraries). These libraries can be written in any language. But for C, the necessary header files can be found in the installation directory of EMT. A Tiny C compiler is provided with EMT which can translate a C file into a DLL linking the necessary modules. In this introduction, we have to refer to the documentation for details.

It is also possible to write single C functions and bind them to DLLs which are automatically loaded after the compilation.

### Example

For an example, we program the arithmetic geometric means starting with two values $a_0 < b_0$. I.e.,

$$a_{n+1} = \sqrt{a_n b_n}, \quad b_{n+1} = \frac{a_n + b_n}{2}.$$

We do that in EMT first to be able to check the result.

```
>function map agm1 (a:positive number,b: positive number) ...
$  repeat
$     {a,b}={sqrt(a*b),(a+b)/2};
$     until a~=b;
$  end;
$  return a
$  endfunction
>agm1(1,2)
   1.45679103105
```

Since we have vectorized the function, we can use it for vector input too.

```
>agm1(1:0.1:2,2)
   [1.45679,  1.51644,  1.57449,  1.63117,  1.68663,  1.74101,  1.79442,
   1.84695,  1.89868,  1.94968,  2]
```

Another option is to use iterate.

```
>iterate("[sqrt(x[1]*x[2]),(x[1]+x[2])/2]",[1,2])
   [1.45679,  1.45679]
```

Now, let us compile this iteration in C. The easiest method is to do just one iteration in C, and to do the vectorization in EMT. Let us first show the C code.

```
>function tinyc agm2 (a,b,eps) ...
$  ARG_DOUBLE(a); ARG_DOUBLE(b);
$  CHECK(a>0 && b>0,"Need positive reals!");
$  ARG_DOUBLE(eps);
$  CHECK(eps>0,"Need a positive epsilon!");
$  while (1) {
$     if (fabs(a-b)/fabs(a+b)<eps) break;
$     double h=sqrt(a*b);
$     b=(a+b)/2; a=h;
$  }
$  new_real(a);
$  endfunction
>agm2(1,2,epsilon)
   1.45679103105
```

The C macros `ARG_...` take the values from the stack of EMT and make the values usable for C. E.g., `ARG_DOUBLE(a)` takes the first argument and defines a `double`-Variable `a` for C. To place the result on the stack of EMT we have called `new_real`.

The C macro `CHECK` throws an error message, if the functions is not user properly.

```
>agm2(-1,2,0)
   agm2 returned an error:
   Need positive reals!
   Error in:
   agm2(-1,2,0) ...
                ^
```

To vectorize such a function in EMT we build a function which calls the C code.

```
>function map agm3 (a,b,eps=epsilon()) := agm2(a,b,eps);
>agm3(1:0.1:2,2)
   [1.45679,  1.51644,  1.57449,  1.63117,  1.68663,  1.74101,  1.79442,
   1.84695,  1.89868,  1.94968,  2]
```

Another option is to do the vectorization in C. This is a bit tricky if we want to allow one of the vectors to be a scalar. We use other macros to read the matrix from the stack of EMT and to generate a matrix for the result. Note, that the vectorization still does not work for matrix input.

```
>function tinyc agm4 (a,b,eps) ...
$  ARG_DOUBLE_MATRIX(av,ra,ca); ARG_DOUBLE_MATRIX(bv,rb,cb);
$  CHECK(ra==1 && rb==1 && ca>0 && cb>0,
$     "Need row vectors of real numbers");
$  int c=(ca>cb)?ca:cb;
$  ARG_DOUBLE(eps); CHECK(eps>0,"Need positive epsilon");
$  RES_DOUBLE_MATRIX(cv,1,c);
$  for (int i=0; i<c; i++) {
$     double a=av[i<ca?i:0],b=bv[i<cb?i:0];
$     while (1) {
$         if (fabs(a-b)/fabs(a+b)<eps) break;
$         double h=sqrt(a*b);
$         b=(a+b)/2; a=h;
$     }
$     cv[i]=a;
$  }
$  endfunction
>agm4(1:0.1:2,2,epsilon)
   [1.45679,  1.51644,  1.57449,  1.63117,  1.68663,  1.74101,  1.79442,
   1.84695,  1.89868,  1.94968,  2]
```

To compare the speed of the solutions, we use `tic` and `toc`. Indeed, the C solution is much faster.

```
>tic; agm1(1:0.0001:2,2); toc;
   Used 0.134 seconds
>tic; agm3(1:0.0001:2,2,epsilon); toc;
   Used 0.022 seconds
>tic; agm4(1:0.0001:2,2,epsilon); toc;
   Used 0.003 seconds
```

## 8.13 Python Code

If Python is installed properly it can be called from EMT. There is an intermediate DLL for this which is installed with EMT. It takes care of the translations between the data types of EMT and Python.

On my system, I have installed Anaconda. This version of Python contains MatPlotlib and other libraries.

```
>>> print sys.version
   2.7.7 |Anaconda 2.0.1 (64-bit)| (default, Jun 11 2014, ...
```

It is possible to enter the results of MatPlotlib into the EMT notebook. The following is a very simple plot. There are more elaborate and more beautiful examples on the home page of MatPlotlib.

```
>>> from pylab import *
>>> t = arange(0.0,2.0,0.01)
>>> s = sin(2*pi*t)
>>> plot(t,s)
>pyins()
```

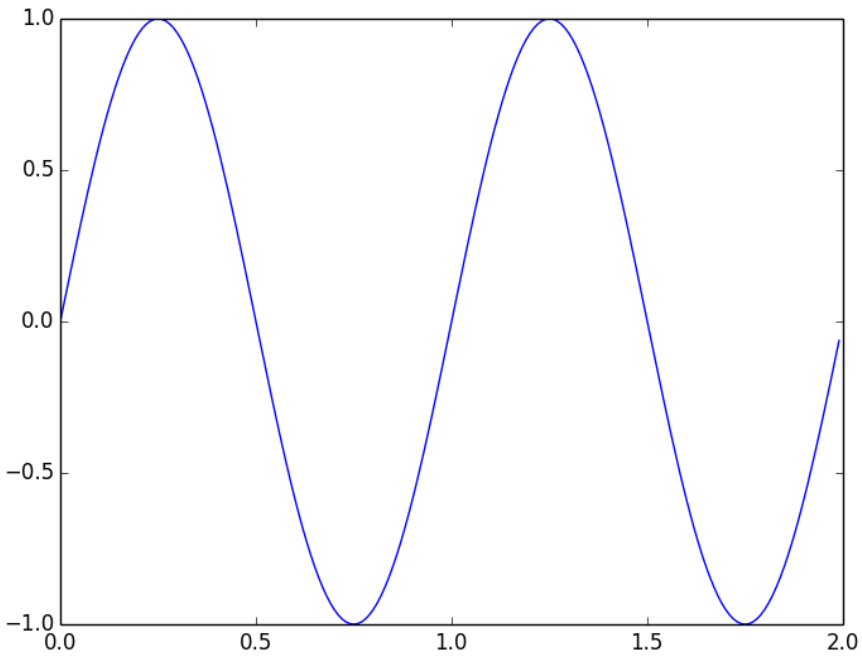



Figure 8.4: Very simple Python Plot

The function `pyins` puts a PNG file in the user home directory of EMT. For more information, consult the documentation of Python in EMT.

Besides calling Python directly, Python functions can be defined and called from EMT.

**Example**

Let us redo the arithmetic geometric mean from the previous section in Python.

```
>function python agm5 (a,b,eps) ...
$  import math
$  a=float(a)
$  b=float(b)
$  while abs(a-b)/abs(a+b)>eps:
$     h=math.sqrt(a*b)
$     b=(a+b)/2
$     a=h
$  return a
$  endfunction
>agm5(1,2,epsilon)
   1.45679103105
```

The speed is only slightly better than EMT if we vectorize this later.

```
>function map agm6 (a,b,eps=epsilon()) := agm5(a,b,eps);
>tic; agm6(1:0.0001:2,2); toc;
   Used 0.054 seconds
```

If we vectorized in Python the speed would improve.

Instead of calling one Python command on each line or writing a Python function, we can write a dummy function which will be executed immediately. It is merely a collection of Python commands.

```
>function python ...
$  s=0
$  for i in range(1,101):
$    s+=i
$  print s
$  endfunction
   5050
```

Calling functions of EMT from Python is also possible. It is even possible to call an EMT function by name. For this, see the tutorial about Python in EMT.

# Chapter 9

# Statistics

## 9.1 Random Numbers

EMT can generate vectors and matrices filled with random numbers. The function random generates a matrix of numbers equi-distributed in $[0, 1]$, and normal generates 0-1 normal distributed numbers.

```
>random(10)
 [0.87411,  0.381891,  0.433033,  0.764686,  0.340088,  0.551344,
 0.814243,  0.43166,  0.0914262,  0.14314]
>a=random(1000000); mean(a), dev(a)
 0.49998211665
 0.288726587903
>normal(10)
 [-1.53286,  1.1423,  -1.49752,  1.33404,  0.889909,  0.0095027,
 0.058976,  0.803505,  -1.01251,  -0.018801]
>a=normal(1000000); mean(a), dev(a)
 -0.000383388642921
 0.999590841865
>normal(2,2)
     -0.179568      0.133036
       0.84618       1.96802
```

EMT starts with the same random sequence at each run. seed can be used for setting a fixed start value. One can use daynow for a seed specific at the current point in time.

```
>seed(1/2); random, random
 0.292592617623
 0.477948650078
>seed(1/2); random
 0.292592617623
>seed(daynow); random
 0.612769859417
```

For more convenience, there is the function randnormal wich allows to change the parameters of the normal distribution. We generate 10000 normal distributed values with mean value 1000 and standard deviation 5, and plot a histogram of the values. The function plot2d can compute and display a histogram of the data automatically. We add the density of the normal distribution using qnormal.

```
>a:=randnormal(1,10000,1000,5); ...
>plot2d(a,distribution=20,style="/"); ...
>plot2d("qnormal(x,1000,5)",color=red,thickness=2,>add):
```

The distribution plot could also be set inside a plot rectangle. All bounds must be specified.

```
>plot2d(a,distribution=20,a=970,b=1030,c=0,d=0.1,style="/"):
```

For more control, the function histo prepares the data for the histogram plot, computing the frequencies in a given number of intervals or a vector of interval bounds.

To adjust this to the normal distribution we must take care of the length of the intervals and the number of random numbers. The expected number of data in a small interval of length $d$ at the point $x$ is $n\,d\,g(x)$. So we divide the count of data in each interval by $d\,n$. This should be close to the density function $g$.

```
>n=10000; a:=randnormal(1,n,1000,5); ...
>d=2; {x,y} = histo(a,v=980:d:1020); ...
>plot2d(x,y/(n*d),>bar,style="/"); ...
>plot2d("qnormal(x,1000,5)",color=red,thickness=2,>add):
```

We can also count the number of elements in subintervals using getfrequencies instead of using the plot function histo.

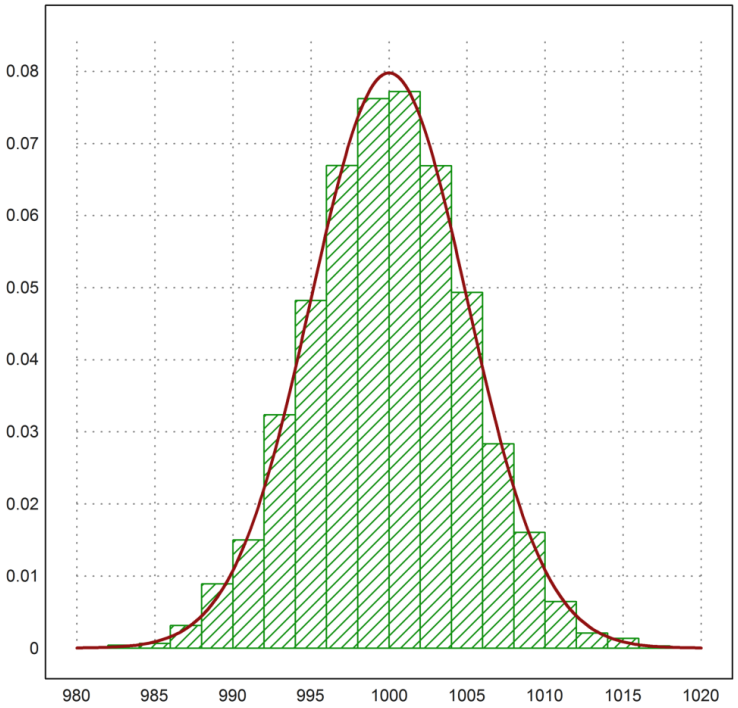



Figure 9.1: Normal distributed values

```
>a=random(1000000);
>getfrequencies(a,0:0.2:1), sum(%)
   [199750,  200668,  199954,  200034,  199594]
   1000000
```

For discrete values, we can use `getmultiplicites`. In the following example, `intrandom` generates 600 integer random variables from 1 to 6 to simulate 600 throws of a die.

```
>a=intrandom(1,600,6);
>getmultiplicities(1:6,a)
   [107,  97,  90,  110,  95,  101]
```

There are many more random variables which can be used for Monte Carlo methods. For an overview, see the documentation of the statistical functions in EMT.

The generation of random variables is good for Monte Carlo simulations. For an example, we determine the mean maximal value of 10 uniform random values in

$[0,1]$. The expected value for this is

$$\frac{10}{11} = 0.909090\ldots$$

```
>n=1000000; m=10; x=random(n,m);
>y=max(x); mean(y')
 0.909089448749
```

One must be careful to program big loops in EMT. Matrix languages are not very efficient for this. If it is unavoidable a bit of C or Python code can be used. Often a clever vectorization is possible. This will be faster even if it looks worse on first sight.

**Example**

We ask how long it takes to get two successive 6 in die throws. The answer is the famous number 42.

```
>function simulate (n) ...
$  v=zeros(n);
$  m=500;
$  loop 1 to n;
$     x=intrandom(1,m,6);
$     v[#]=firstnonzero(x[1:m-1]==6 && x[2:m]==6)+1;
$  end;
$  return mean(v);
$  endfunction
>tic; simulate(1000000), toc;
   41.978379
   Used 31.081 seconds
```

This code will take some seconds. It could be made slightly faster by taking a smaller $m$ and taking an appropriate action if two successive 6 are not among the random numbers.

Of course, a version in Tiny C is faster. But we have to use the random number generate of Tiny C. In Python, we could call the random number generator of EMT, but Python is a bit slower.

```
>function tinyc simulate (n) ...
$  ARG_DOUBLE(x);
$  int n=(int)x;
$  CHECK(n>=1,"Need a positive integer");
$  double sum=0.0;
$  for (int k=0; k<n; k++) {
$     int a=(rand()%6)+1,b=(rand()%6)+1;
$     int count=2;
$     while (1) {
$         if (a==6 && b==6) break;
$         a=b; b=(rand()%6)+1;
$         count++;
$     }
$     sum=sum+count;
$  }
$  new_real(sum/n);
$  endfunction
>tic; simulate(1000000), toc;
   42.020853
   Used 0.731 seconds
```

### Example

Sometimes, own methods must be implemented for special distributions. Assume we want to use the density function

$$g(x) = \cos(x)$$

for $0 \le x \le \pi/2$. This is a probability distribution $Y$ since

$$\int_0^{\pi/2} \cos(x)\,dx = 1.$$

One way to get a random number generator is to apply the inverse function of the distribution function $\sin(x)$ to a uniform random variable $X$ in $[0,1]$. This works because of

$$P(\arcsin(X) \le c) = P(X \le \sin(c)) = \sin(c) = P(Y \le c).$$

Thus $\arcsin(X)$ has the correct distribution for a uniform random variable in $[0,1]$.

```
>function randcos(n,m) := asin(random(n,m));
>plot2d(randcos(1,10000),>distribution,style="/"); ...
>plot2d("cos",>add,color=red):
```

Another method is the rejection method. We produce uniformly distributed random variables in the rectangle $[0, \pi/2] \times [0, 1]$ and reject those that are not under the graph of the function $\cos(x)$.

```
>function randcos (n) ...
$  x=random(n)*pi/2; y=random(n);
$  return x[nonzeros(y<cos(x))];
$endfunction
>plot2d(randcos(1000000),>distribution,style="/"); ...
>plot2d("cos",>add,color=red):
```

To see this method in graphical form, we show the points in a plot and mark the rejected points in red.

```
>n=10000; x=random(n)*pi/2; y=random(n);
>i=nonzeros(y<cos(x)); c=ones(size(x)); c[i]=red;
>aspect(pi/2);
>plot2d(x,y,>points,color=c,style=".");
>plot2d("cos",>add,thickness=2):
```

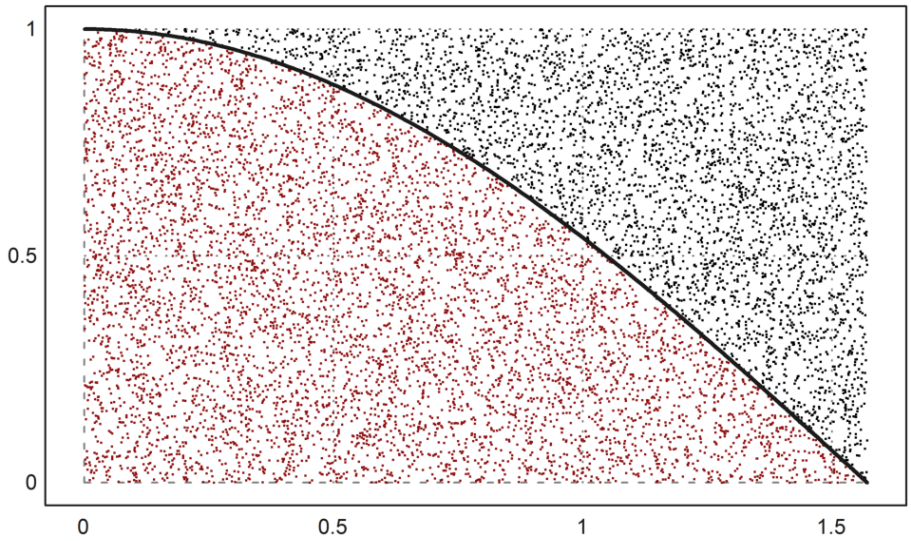



Figure 9.2: Rejection Method

EMT can also shuffle vectors with `shuffle`. As an example we generate lottery numbers 6 out of 49.

```
>z:=shuffle(1:49); sort(z[1:6])
   10  13  27  31  43  48
```

There is also a package for permutations. It allows to compute with permutations and to let a function walk over all permutations of n numbers. For more, see the demo on Monte Carlo methods among the tutorials for EMT.

## 9.2 Distributions

EMT can compute many distributions and their inverses. The most important one is the normal distribution.

**Example**

At 1000 throws of a coin the expected numbers of heads is distributed with mean value 500 and standard deviation

$$\sigma = \sqrt{1000 \cdot 0.5 \cdot 0.5}.$$

We compute the probability to get more than 520 times head, and when the probability gets less than 0.1% approximating the binomial distribution with the normal distribution.

```
>n=1000; p=0.5;  ...
>m=p*n; s=sqrt(n*p*(1-p));  ...
>1-normaldis(520,m,s)
   0.102951605366
>ceil(invnormaldis(99.9%,m,s))
   549
```

Note that the function `normaldis` in EMT scales in another way than the function `erf`, which is also available. All distributions in EMT are implemented as distribution functions, which grow from 0 to 1.

The approximation to the binomial distribution in this example can also be computed exactly.

```
>1-bindis(520,1000,0.5)
   0.0973831642309
>invbindis(99.9%,1000,0.5)
   548.347804004
```

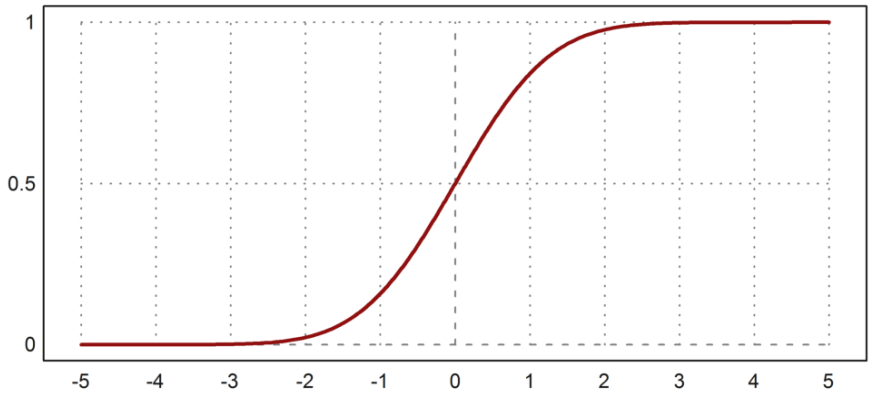



Figure 9.3: Normal Distribution `normaldis(x)`

EMT contains many more distributions. For details, see the reference. Many examples are contained in the tutorial about statistics.

| Further Information | |
|---|---|
| `chidis` | *chi* distribution. |
| `tdis` | Student T-distribution. |
| `invtdis` | Inverse T-distribution. |
| `fdis` | F-distribution. |

## 9.3 Data Input and Output

It is often necessary to read and write data files. EMT supports this with a lot of functions on different levels. EMT can open only one file for read and one for write at each time. To open a file use the `open` command, and to close it, use the `close` command. Some functions open and close files automatically.

Without a path, files are opened in the current directory. Opening or saving a notebook sets the current directory to the notebook directory. Another good place for notebooks is the user folder `Euler Files` in the home directory of the user. To get the path to this folder use `userhome`.

```
>filename=userhome()|"test.dat";
```

In the following, we generate 10000 random numbers, and write these numbers into a file `test.dat`. Then we read these numbers back from the file. The high level functions `writematrix` and `readmatrix` open and close the file automatically. The matrix is written line by line. To avoid very long lines, it is better to use a column vector for the random matrix.

```
>n=10000; a=random(n);
>writematrix(a',filename);
>b=readmatrix(filename)';
>longest max(abs(a-b))
    5.551115123125783e-017
```

The vectors are not completely identical since a decimal output is used in the file.

To append another matrix to the same file, we open the file in the appending mode `"a"`. The write mode `"w"` would delete the content of the file. Then we write the matrix with `writematrix`, and close the file. To read both matrices, we open the file with the read mode `"r"`, and use `readmatrix` twice without the file name parameter.

```
>open(filename,"a");
>writematrix(random(2,2));
>close();
>open(filename,"r");
>readmatrix(); // skip random vector
>readmatrix()
      0.770952      0.402944
      0.648543     0.0532717
>close();
```

There are numerous elementary functions to read and write data. One useful function is `getvector`, or `getvectorline`. This function reads an unknown, but limited number of numbers (but with a decimal point, not a comma). Intermediate text is skipped. `getvectorline` stops at the end of each line. The functions return the data and the count. Additionally, `getvectorline` returns the line in string from. The functions are more effective and faster than `readmatrix`.

In the following example, we use the `write` command to output a string, and `putchar(10)` for a line feed.

```
>open("test.dat","w");
>write("Two Numbers: 1000 2000"); putchar(10);
```

```
>close();
>open("test.dat","r");
>{v,n,s}:=getvectorline(1000);
>close();
>s
Two Numbers: 1000 2000
>n
               2
>v
            1000          2000
```

Another method to read and write data to files are tables. Tables are frequently used in statistics to hold data. A table can have a header line, row labels. Each item in the file can be a number or a string. Strings are translated to numbers via string vectors.

In the following example, we generate a table with a number, a yes/no string, and an age. The function `writetable` prints the table with the tokens in column replaces by the strings in the string vector `yn`.

```
>data=[1000,1,55;1010,1,58;1020,2,45;1030,2,60]
          1000             1             55
          1010             1             58
          1020             2             45
          1030             2             60
>yn=["yes","no"];
>hd=["No","Y/N","Age"];
>writetable(data,labc=hd,tok2=yn,labr=1:4)
                    No       Y/N       Age
          1       1000       yes        55
          2       1010       yes        58
          3       1020        no        45
          4       1030        no        60
```

We can also write the table to a file. From there, we can read the table with known translations for the token strings.

```
>filename=eulerhome()+"test.dat";
>writetable(data,labc=hd,tok2=yn,labr=1:4,file=filename)
>{d,hd,toks,rows}=readtable(filename,tok2=yn,>rlabs);
>d
            1000             1             55
```

```
          1010            1           58
          1020            2           45
          1030            2           60
>hd
  No
  Y/N
  Age
>rows
  1
  2
  3
  4
```

Note that `>rlabs` has to be set, since `readtable` needs to know about the labels in each line.

We can also read the table without knowing the translations. Then the token strings will be selected in the specified columns into a string vector.

```
>{d,hd,toks,rows}=readtable(filename,ctok=[2],>rlabs);
>toks
   yes
   no
```

For special purposes, EMT can scan lines in the file with regular expressions. The function `strxfind` can return the position where a string is found, the matched string and a vector of sub-matches. The function `strxrepl` can be used to replace substrings.

The syntax of regular expressions is quite mighty. You can find examples on the help page of the function.

```
>{pos,found,v}=strxfind("This is Test34!","([A-Za-z]+)([0-9]+)");
>pos
   9
>found
   Test34
>v
   Test
   34
>strxrepl("This is Test34!","([A-Za-z]+)([0-9]+)","$1-$2")
   This is Test-34!
```

A more elementary way to read data is strtokens. The function can break a string into tokens with specified separator characters.

```
>v=strtokens("4 5, 5.6; pi^2;exp(0.1)",";, ")
   4
   5
   5.6
   pi^2
   exp(0.1)
>for k=1 to length(v); v[k](), end;
   4
   5
   5.6
   9.86960440109
   1.10517091808
```

EMT can also read a file from the Internet via IP. E.g., the function getstock reads stock data from Yahoo's historical stock data. For details on the implementations, we refer to the tutorial. You can also study the sources. Enter type getstock in the help window for this.

```
>load getstock;
>v=getstock("IBM",day(2012,1,1));
>showstock(v,title="IBM Stock Data"):
```

| Further Information | |
|---|---|
| getline | Reads a line as a string. |
| eof | Test if the file has been read completely. |
| dir | Lists the current directory. |
| dir("*.e") | Lists all EMT files. |
| fileremove | Deletes a file. |
| cd(...) | Changes the working directory. |

## 9.4 Statistical Tests

EMT can compute some statistical tests.

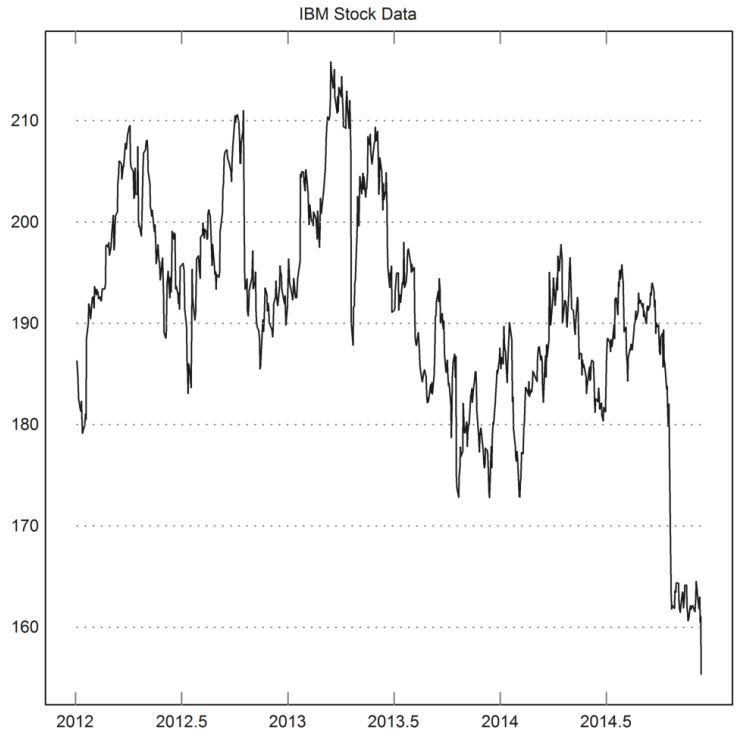



Figure 9.4: Stock Data from the Yahoo

### Example

We test a series of 0-1 normal distributed random numbers for the mean value 0.5 with a student-T test. Of course, we will most likely be able to reject the hypothesis that the sample is from distribution with mean value 0.5 or higher.

```
>a:=normal(1,20);
>ttest(mean(a),dev(a),20,0.5)
        0.00101056390067
```

For other tests refer to the following table, or to the EMT reference.

| Further Information | |
|---|---|
| `tcomparedata` | Tests two sample on equal distribution. |
| `chitest` | Test on equal distribution using the $\chi^2$ test. |
| `tabletest` | Test on independence of table rows. |
| `varanalysis` | Test on the same mean value. |
| `mediantest` | Test on same mean value. |
| `ranktest` | Test on same mean value. |
| `wilcoxon` | Compares mean values. |

# Chapter 10

# Numerical Algorithms in Euler

## 10.1 Differential Equations

Solving differential equations numerically is one of the important applications of a software like EMT. In this section, we discuss various methods and methods to obtain guaranteed inclusions.

For a start, we solve the intial value problem

$$y' = -\frac{y}{x}, \quad y(1) = 1.$$

using the Runge method. The solution is $y = 1/x$. `runge` accepts an expression in $x$ and $y$, or a function. We can either get all intermediate values, or only the values of the solution at selected points. In the latter case, we need to specify the number of points to be computed between our selected points.

```
>x=1:0.01:10; y=runge("-y/x",x,1); // get all 1000 values
>plot2d(x,y):
>y[-1] // should be 1/10
   0.1
>runge("-y/x",[1,10],1,100) // get only one value, and use 100 points
   [1,  0.1]
```

A more advanced function is the LSODA algorithm which uses an adaptive step size and works for stiff equations too. It is the default for the function `ode`. The algorithms is very good and does not need intermediate points. But it fails sometimes.

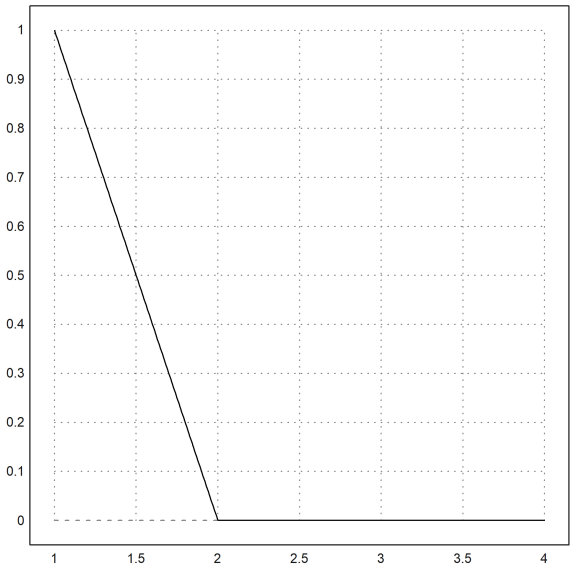



Figure 10.1: Solution of a Stiff Equation

The following equation would be solved as $y' = -1$ with solution $y = 1 - x$ with the Runge method. This happens for small $c$ where the Runge method solves the approximation $y' = 1$. But the LSODA algorithm is able to see the barrier at $y = 0$. Admittedly, it would be easier to set $y' = 0$ for $y < 0$ to produce such a barrier.

```
>function f(x,y,c) := -y/(c+y);
>x=1:0.01:4; plot2d(x,ode("f",x,1;0.0001)):
```

Note that we specified the parameter `c` for the function `f(x,y,c)` as a semicolon parameter in the call to `ode`. This parameter is passed to `f` from `ode`.

The function `ode` can also work with vector valued functions $y$. The system of differential equations

$$u' = \begin{pmatrix} -xu_2(x) \\ -x^2u_1(x) \end{pmatrix}$$

turns out to be almost periodic. We plot the curve $(u_1, u_2)$ for $x$ from 0 to $-20$. As you see, it is possible to specify $x$ values in decreasing order.

```
>function f(x,y) := [-x*y[2],-x^2*y[1]]
>x=0:-0.001:-20; ysol=ode("f",x,[1,0]);
>plot2d(ysol[1],ysol[2]):
```

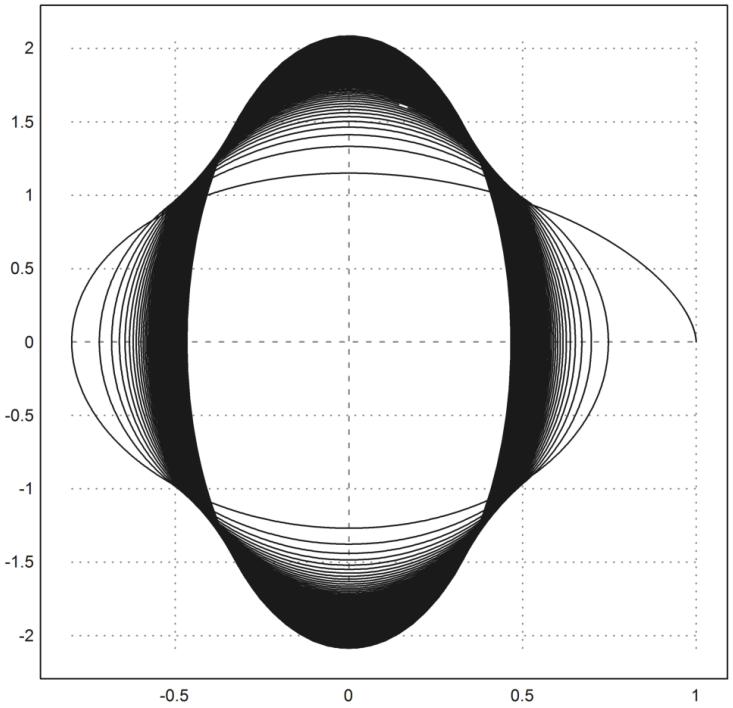



Figure 10.2: Periodic Solution

### Example

For an example of a second order equation, we solve the initial value problem

$$y'' = -a\sin(y), \quad y(0) = 0, \quad y'(0) = b$$

using the adaptive Runge method. This second order differential equation must be rewritten into a first order equation in the plane.

$$\frac{d}{dx}\begin{pmatrix} y(x) \\ y'(x) \end{pmatrix} = \begin{pmatrix} y'(x) \\ -a\sin(y(x)) \end{pmatrix}$$

Then `runge` and `adpativerunge` expect a function `f(x,y)`, computing $y'$. Both $y$ and $y'$ have to be row vectors.

```
>function f(x,y) := [y[2],-a*sin(y[1])]
>a:=1; b:=1;
>x:=0:0.1:10; y:=adaptiverunge("f",x,[0,b]);
>plot2d(x,y[1]):
```

We then want to determine the first zero of the solution to get the frequency of this true pendulum. To do this, we set up a function to solve for $y(x)$, and solve $y(x) = 0$ with the secant method.

```
>function h(x) ...
$     global b;
$     v=ode("f",[0,x],[0,b]);
$     return v[1,2];
$  endfunction
>solve("h",3,4)
   3.37150070962
```

Of course, the solution of the approximating differential equation $y'' = -ay$ for the pendulum has its first zero at $\pi$. Taking a smaller amplitude $b$ makes the error much smaller.

```
>b=0.01;
>secant("h",3,4)
   3.14161228868
```

### Example

To solve a boundary value problem

$$y'' = -a\sin(y), \quad y(0) = 0, \quad y(\pi) = 1$$

in EMT, we use the shooting method. We set $y'(0) = x$, and solve $y(\pi) = 1$ for $x$.

```
>a=1;
>function g(x)
$ v=adaptiverunge("f",[0,pi],[0,x]);
$ return v[1,2]-1;
$endfunction
>secant("g",1)
        1.521612414054
```

EMT can plot the direction field of a differential equations with the functions `vectorfield` and `vectorfield2`.

The function `vectorfield` displays a vector field for $y' = f(x, y)$, where $y$ is a scalar function.

```
>vectorfield("-2*x+y",0,2,0,2);
>x=0:0.01:2; y=ode("-2*x+y",x,1);
>plot2d(x,y,>add,color=red,thickness=2):
```

The function vectorfield2 displays the vector field for vector valued functions in the plane.

```
>vectorfield2("y","-sin(x)",-2,2,-2,2);
>function f(x,y) := [y[2],-sin(y[1])]
>x:=0:0.01:10; y:=runge("f",x,[0,1]);
>plot2d(y[1],y[2],>add,color=red,thickness=2):
```

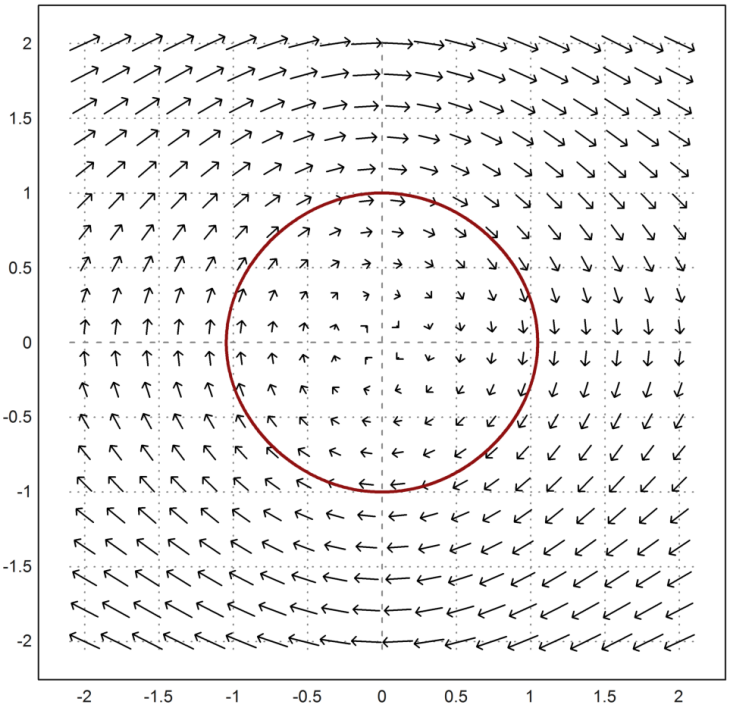



Figure 10.3: Pendulum $y'' = \sin(y)$

With Maxima, we can solve many differential equations exactly. Use ode2 for the general solution. With ic1 and ic2, you can get the constants for initial value problems or boundary value problems.

### Example

We solve the initial value problem

$$y'' + y' + y = \sin(x), \quad y(0) = 0, \quad y'(0) = 1$$

with Maxima, and plot the result in EMT.

```
>:: eq := 'diff(y,x,2)+'diff(y,x)+y=sin(x)
```

$$\frac{d^2 y}{dx^2} + \frac{dy}{dx} + y = \sin(x)$$

Note the apostrophe in the definition of the equation. It prevents the execution of the `diff` function. To insert an initial value, use `ic2`. For more, refer to the tutorial about differential equations or the documentation of Maxima.

```
>gsol &= ode2(eq,y,x)
```

$$y = e^{-x/2}\left(\%k1 \sin\left(\frac{\sqrt{3}\,x}{2}\right) + \%k2 \cos\left(\frac{\sqrt{3}\,x}{2}\right)\right) - \cos(x)$$

```
>sol &= y with ic2(gsol,x=0,y=0,'diff(y,x)=1)
```

$$e^{-x/2}\left(\sqrt{3} \sin\left(\frac{\sqrt{3}\,x}{2}\right) + \cos\left(\frac{\sqrt{3}\,x}{2}\right)\right) - \cos(x)$$

```
>plot2d(&sol,0,2*pi):
```

To get a guaranteed inclusion for an initial value problem, we can use the function `mxmidgl`. This function computes very high derivatives of the differential expression with Maxima. The degree of the approximation can be adjusted with the `deg` parameter. We compare the result with the exact solution in Maxima.

```
>x=linspace(0,pi,100); y=mxmidgl("sin(x)*y",x,1); y[-1]
   ~7.3890560989303,7.389056098931~
>&ode2('diff(y,x)=sin(x)*y,y,x); sol &= y with ic1(%,x=0,y=1)
```

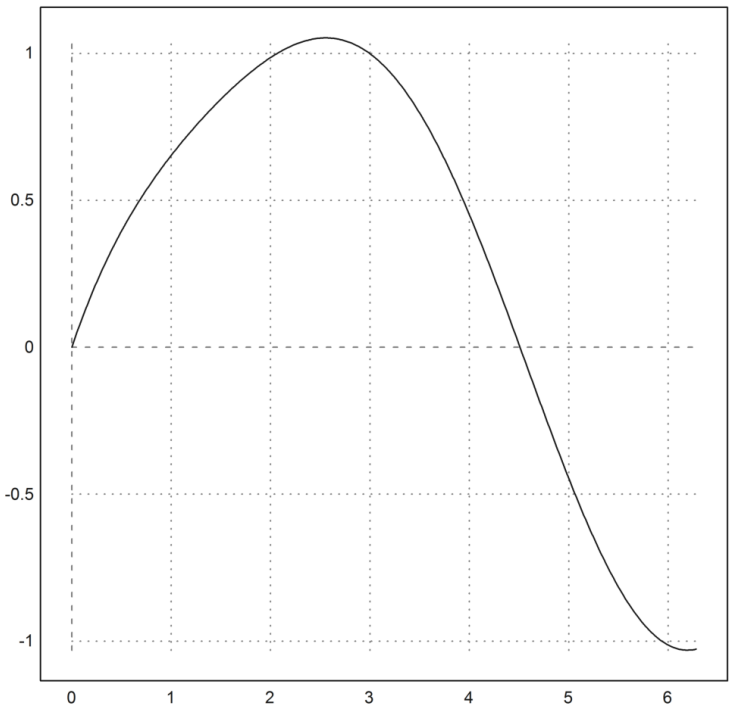



Figure 10.4: Exact Solution with Maxima

```
                    1 - cos(x)
                   E
```

```
>longest sol(pi)
          7.38905609893065
```

This can be used for exact inclusions of integrals. We compute the Gauß normal distribution.

```
>0.5+mxmiint("1/sqrt(2*pi)*exp(-x^2/2)",0,2)
   ~0.977249868051802,0.97724986805184~
>longest normaldis(2)
        0.9772498680518208
```

## 10.2 Iteration and Recursion

For recursive sequences, EMT has the functions iterate and sequence. The most elementary form is iterate, which iterates a recursive sequence

$$x_{n+1} = f(x_n)$$

starting from some point $x_0$ until convergence occurs. Convergence is checked with the internal epsilon. If the iteration cannot find a fixed point, you will have to stop it with the Esc key. Alternatively, you can enter a maximal number of iterations. In this case, the function returns all $x_i$ so far. The function will work for complex, and even for interval iteration, as long as $f$ is a contracting function and the iterations starts close enough.

```
>iterate("cos(x)",1)
   0.739085133216
>iterate("cos(x)",1+I)
   0.739085+0i
>iterate("cos(x)",~0.7,0.8~)
   ~0.7390851332142,0.7390851332166~
>iterate("(x+2/x)/2",1,5)
   [1,  1.5,  1.41667,  1.41422,  1.41421,  1.41421]
```

iterate can also handle sequences of vectors. In this case, a function is better than an expression.

### Example

We iterate the arithmetic geometric mean

$$a_{n+1} = \sqrt{a_n b_n}, \quad b_{n+1} = \frac{a_n + b_n}{2}.$$

```
>function agm([x,y]) := return [sqrt(x*y),(x+y)/2]
>longest iterate("agm",[1,2])
         1.456791031046907       1.456791031046907
```

A more flexible recursive function is sequence. It can handle recursions of the form

$$x_n = f(x_1, \dots, x_{n-1}, n).$$

iterate returns a row vector with the sequence elements. Make sure, enough start values are provided.

```
>sequence("x[n-1]+x[n-2]",[1,1],10) // Fibonacci numbers
   [1,  1,  2,  3,  5,  8,  13,  21,  34,  55]
>sequence("n*x[n-1]",1,6) // factorials
   [1,  2,  6,  24,  120,  720]
>sequence("sum(x[n-4:n-1])/4",[1,2,3,4],10) // sliding mean
   [1,  2,  3,  4,  2.5,  2.875,  3.09375,  3.11719,  2.89648,  2.99561]
>longformat; v:=sequence("sum(x[n-4:n-1])/4",[1,2,3,4],1000); v[-1]
   3
```

## 10.3 Fast Fourier Transformation

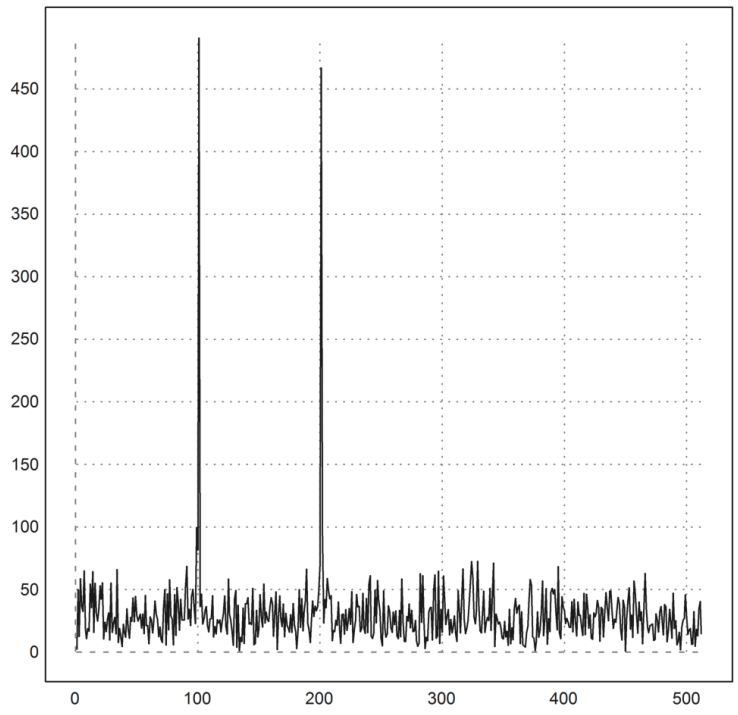



Figure 10.5: Frequency Analysis with FFT

The Fast Fourier Transformation FFT is a technique, which evaluates trigonometric sums very quickly in equidistant points. The inverse operation interpolates with Fourier sums. It turns out that the same fast FFT can be used for the inverse. This is useful for frequency analysis. For the optimal speed of the function the number of points should be a power of 2, or at least have only small prime factors.

There is a notebook explaining the FFT in more detail.

### Example

We generate values for the function $\sin(100x) + \cos(200x)$, add noise, and analyze the frequencies of the result. The analysis uses FFT to interpolate the values in the roots of unity. The absolute values of the coefficients then shows the contribution of the frequencies. However, the relevant entries are only between frequency 1 and $n/2$, were $n$ is the number of data.

```
>t:=linspace(0,2pi,1023); s:=sin(100t)+cos(200t)+normal(size(t));
>f:=abs(fft(s)); plot2d(f[1:512]);
```

There are some functions to generate and analyze sound. The following lines generate a sound, and analyzes its frequencies using FFT, presenting a graphical representation.

```
>t=soundsec(5);
>s=sin(t*440)+sin(t*880)/2;
>analyzesound(s);
```

The sound can be played using `playwave`, or stored with `savewave`. Of course, sound can be loaded into memory with `loadwave` too. All these functions work with a default sampling rate of 22050 Hz. To analyze a sound with changing frequencies, use `mapsound`. This function will use a windowed FFT.

FFT can be used to fold two vectors. This is due to the fact, that folding two vectors is equivalent to multiplying the Fourier transforms. Note that the signal vector is assumed to be periodic in this case.

### Example

Let us fold the vector $(-1)^k$ with another vector such that the new vector contains the averages of three neighboring points.

```
>k=(-1)^(1:1024);
>f=zeros(size(k)); f[1:3]=[1/3,1/3,1/3];
>k1=real(ifft(fft(k)*fft(f)));
>k1[1:3]
   [-0.333333333333,  0.333333333333,  -0.333333333333]
```

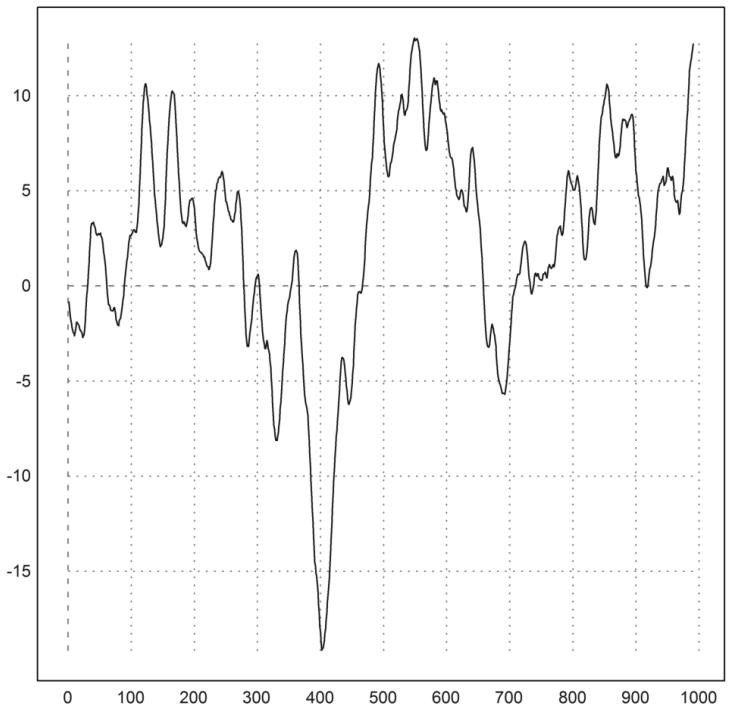



Figure 10.6: Smoothed Random Walk

The folding assumes that the samples are periodic. For simple folding, use `fold`. Here is a smoothing example with a plot of the smoothed data. Note that the data gets shorter if is folded with with vector. In this example,

$$\tilde{a}_k = \frac{1}{10}\sum_{i=0}^{9} a_{i+k}$$

for $k = 1, \dots, n-9$.

```
>n=1000; k=cumsum(normal(n));
>plot2d(1:n-9,fold(k,ones(10)/10)):
```

Graphics can also be handled by Euler, and there is a two dimensional FFT. Graphics can be loaded into memory with `loadpixels`, and saved back with `savepixels`. The format is an RGB matrix, containing one pixel per entry. To get the red, green and blue channels, use `getred` etc. To create an RGB matrix from the channels, use `putred` etc., or `rgb`. To plot such an image into the plot window, use `plotrgb`. Alternatively, insert the image into the notebook with `insrgb`.

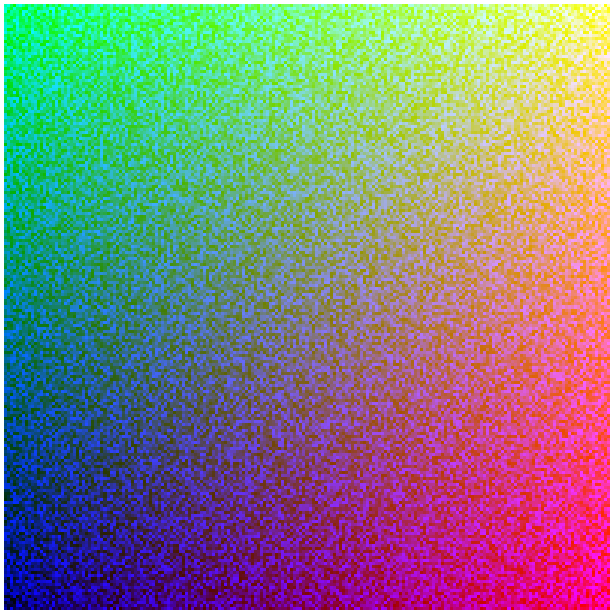


Figure 10.7: Image generated by EMT

```
>red:=0:0.005:1; green:=red'; blue:=random(cols(red),cols(red));
>M:=rgb(red,green,blue); plotrgb(M);
>insrgb(M);
>savergb(M,userhome()|"test.png");
```

## 10.4 Linear Programming

EMT has the Simplex algorithm for linear optimization. The function `simplex` maximizes (with `>max`) or minimizes (by default) a linear target $c^T \cdot x$ under the conditions

$$A \cdot x \leq b, \quad x \geq 0.$$

Here, $A$ is a matrix, each line containing one inequality.

**Example**

Maximize

$$5x + 8y$$

under the conditions

$$\frac{x}{10}+\frac{y}{8}\le 1,$$
$$\frac{x}{9}+\frac{y}{11}\le 1,$$
$$\frac{x}{12}+\frac{y}{7}\le 1,$$

and $x,y\ge 0$.

```
>A=[1/10,1/8;1/9,1/11;1/12,1/7]; b=[1;1;1];
>x=simplex(A,b,[5,8],>max); fraction x
       60/13
       56/13
```

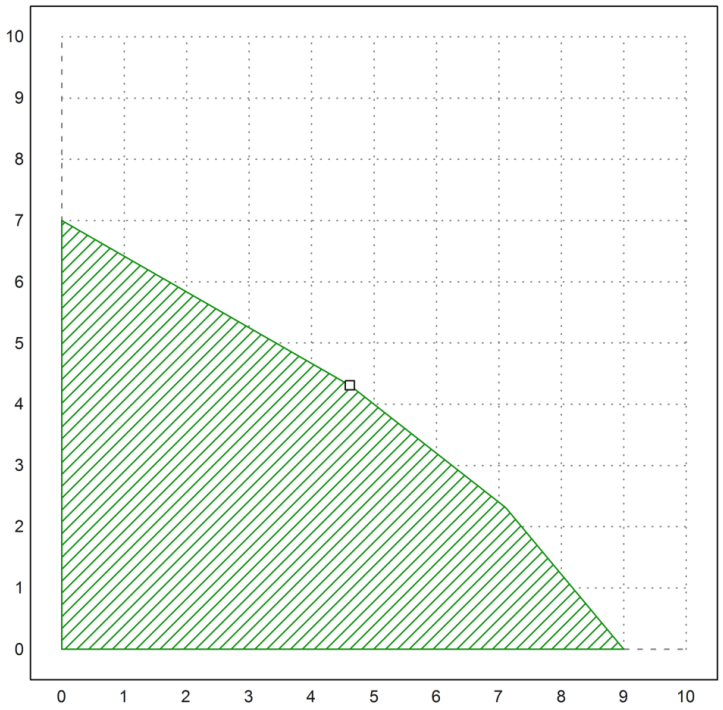



Figure 10.8: Feasible Points and Solution

In the case of two variables, EMT can compute and plot the feasible set.

```
>xa=feasibleArea(A,b);
>plot2d(xa[1],xa[2],>filled,style="/",a=0,b=10,c=0,d=10);
>plot2d(x[1],x[2],>add,>points):
```

An integer solution can be found with `intsimplex`, or with the function `ilpsolve` from the LPSOLVE package. The implementation in EMT is due to Peter Notebaert.

```
>intsimplex(A,b,[5,8],>max)
                5
                4
>ilpsolve(A,b,[5,8],>max)
                5
                4
```

The function `simplex` assumes inequalities of the form "$\le$". Alternatively, it is possible to use inequalities "$\ge$" or equalities "$=$" in each or in all conditions. For this, a vector `eq` must be provided, containing $-1$, 0, or 1 for "$\le$", "$=$", or "$\ge$" respectively. If the vector is only a scalar number, it is valid for all conditions. The default is $-1$.

Moreover, it is possible to relax the condition $x_i \ge 0$ for each or for all variables. The functions `simplex` and `intsimplex` use a row vector `restrict`, containing flags 0 or 1, for unrestricted or restricted variables. As above, a number can be used for all variables. The default is 1.

### Example

We minimize $2x+y$ under the condition $|x|+|y| \le 1$. This condition can be written as

$$-2 \le x+y \le 2, \quad -2 \le x-y \le 2.$$

We solve the problem with and without restrictions.

```
>A=[1,1;1,1;1,-1;1,-1]
                1              1
                1              1
                1             -1
                1             -1
>b=[2;-2;2;-2]
                2
               -2
                2
               -2
>eq=[-1;1;-1;1]
               -1
                1
               -1
```

```
                1
>c=[2;1]
                2
                1
>simplex(A,b,c',eq,0)
               -2
                0
>simplex(A,b,c',eq,1)
                0
                0
```

The default form of the Simplex algorithm throws an error message, if the problem has no solution or is bounded. This can be prevented with `<check`. Then the algorithm returns a flag. For more information on this, see the reference for `simplex`.

The function `pivotize` allows to solve a problem step by step. For an example, we solve the problem at the start of the chapter. The layout of the Simplex scheme in the example below contains the target function in the last line.

To see the fractions, we use a fractional output with a width of 10 places.

```
>A=[1/10,1/8;1/9,1/11;1/12,1/7]; b=[1;1;1]; c=[5;8];
>fracformat(10);
>M=A|id(3)|b_c'
        1/10       1/8         1         0         0         1
         1/9      1/11         0         1         0         1
        1/12       1/7         0         0         1         1
           5         8         0         0         0         0
>pivotize(M,2,1)
           0    19/440         1     -9/10         0      1/10
           1      9/11         0         9         0         9
           0    23/308         0      -3/4         1       1/4
           0     43/11         0       -45         0       -45
>pivotize(M,1,2)
           0         1    440/19   -396/19         0     44/19
           1         0   -360/19    495/19         0    135/19
           0         0  -230/133   429/532         1    41/532
           0         0  -1720/19    693/19         0  -1027/19
>pivotize(M,3,4)
           0         1   -280/13         0    336/13     56/13
           1         0    480/13         0   -420/13     60/13
           0         0  -920/429         1   532/429    41/429
           0         0   -160/13         0   -588/13   -748/13
```

EMT contains many more algorithms for global optimization or optimization with restrictions. There is also a basic form of the Newton-Barrier algorithm. Other algorithms can be implemented easily.

## 10.5 Exact Scalar Product

Solving a linear system $Ax = b$ sometimes yields large errors, even if the system is given exactly. This is the case, if the matrix $A$ has a high condition number. I.e., it has very large and very small eigenvalues. We have

$$\frac{\|\Delta x\|}{\|x\|} \le \|A\| \cdot \|A^{-1}\| \frac{\|\Delta b\|}{\|b\|}$$

with any compatible matrix norm.

### Example

Since $12345^2 - 2 \cdot 13 \cdot 5861501 = 1$, the following matrix has large elements, but a very small determinant. The matrix is almost singular, and not well conditioned. Consequently, we get large rounding errors in the Gauß algorithm.

```
>A=[12345,26;5861501,12345]
           12345            26
     5.8615e+006         12345
>b=A.[1;1]; longest A\b
         1.000000021578259
          0.99998975447644
```

If we compute the first step of the Gauß algorithm in EMT and Maxima, we clearly see the problem. The accuracy of the result is only 7 digits.

```
>B=A; B[2]=B[2]-(B[2,1]/B[1,1])*B[1]; longest B
                     12345                        26
                         0 -8.100445484160446e-005
>&"12345-5861501/12345*26"()
   -8.1004455245e-005
```

To improve this result, we use a residual iteration. For this, we compute the error $r = Ax - b$, and correct the wrong solution $x$ by the solution $d$ of $Ad = r$. For this technique to work, we must compute the residuum exactly. To do this, EMT has an exact scalar product using a long accumulator.

```
>x=A\b; longest x
         1.000000021578259
          0.99998975447644
>r=residuum(A,x,b)
   -2.98916e-012
   -5.89345e-010
>x=x-A\r; longest x
        0.9999999999999999
         1.000000000000048
```

The procedure is implemented in the function `xlgs`.

```
>longest xlgs(A,b)
                           1
                           1
```

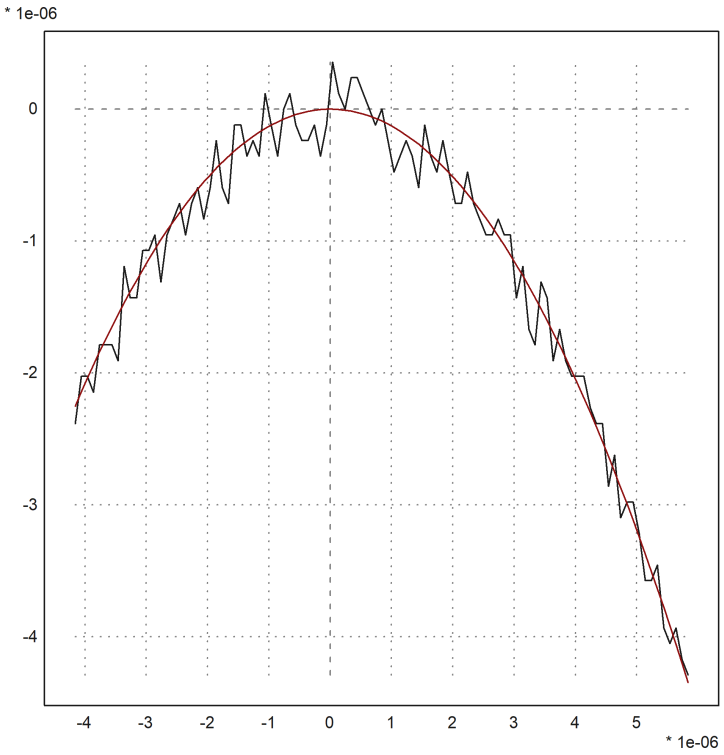



Figure 10.9: Simple Evaluation versus Exact Evaluation

We remark that it does not help to compute the residuum the simple way. The residuum will be too inaccurate to help.

We want to stress here, that the method is only useful for systems, which are given in an exact way. If the systems contain errors in the parameters, the seemingly exact results of xlgs have to be checked carefully using interval methods (see the next section).

**Example**

The following example is due to Rump et al. The residual iteration can be used to evaluate badly conditioned polynomials. EMT implements this in the xpolyval command.

```
>p=[-945804881,1753426039,-1083557822,223200658];
>t=linspace(1.618015,1.618025,100);
>plot2d(t-1.61801916,polyval(p,t));
>plot2d(t-1.61801916,xpolyval(p,t,eps=1e-17),color=red,>add):
```

## 10.6 Guaranteed Inclusions

The residual iteration is useful to find exact solutions of exact equations. If the parameters of the equations are inexact, we can only hope to produce as close inclusions of the solution as possible.

EMT uses the function ilgs to get a guaranteed and narrow inclusion of the solution of a linear system. The idea has been described by Rump et al. This is an iteration using interval arithmetic. The guaranteed inclusion follows from a fix point theorem.

**Example**

We solve the equation of the last section with ilgs. The experiments below show that we get a narrow inclusion only if we assume that the equation is exact. Extending the parameter intervals leads to bad inclusions or even failure. In this case the parameter intervals include cases, where the system does not have a solution.

```
>ilgs(A,b)
    ~0.99999999999999978,1.0000000000000002~
    ~0.99999999999999978,1.0000000000000002~
>ilgs(~A~,~b~)
                  ~0.9999998615,1.000000139~
                       ~0.9999342,1.000066~
>ilgs(A±0.5,b±0.5)
   Error : Pseudo inverse not good enough.
```

```
   Error generated by error() command

   Try "trace errors" to inspect local variables after errors.
   ilgs:
       if (rho>=1); error("Pseudo inverse not good enough."); endif; ...
```

The inclusion method can also be used for non-linear systems, if we can compute the derivative of the function. EMT implements the interval Newton method. The derivative can be computed by Maxima.

### Example

Compute the solution of $e^{-x} = x$. The result is a very narrow interval. `mxminewton` calls Maxima to compute the derivative automatically.

```
>inewton("exp(-x)-x","-exp(-x)-1",0,1)
   ~0.56714329040978362,0.56714329040978406~
>mxminewton("exp(-x)-x",0,1)
   ~0.56714329040978362,0.56714329040978406~
```

### Example

The interval Newton method is available in several dimensions too. In the following example, we solve

$$xy = 1, \quad x^2 + y^2 = 4.$$

It is necessary to compute the Jacobian matrix. We use Maxima for this task in `Df` at compile time. See the section about programming EMT for an explanation of this technique.

```
>f1&=x*y-1; f2&=x^2+y^2-4;
>function f([x,y]) &= [f1,f2]
```

$$\left[x\,y - 1,\ y^2 + x^2 - 4\right]$$

```
>function Df([x,y]) &= jacobian(f(x,y),[x,y])
```

$$\begin{bmatrix} y & x \\ 2\,x & 2\,y \end{bmatrix}$$

```
>x=inewton2("f","Df",[1,2])
   [ ~0.51763809020504115,0.51763809020504203~,
   ~1.931851652578135,1.931851652578138~ ]
```

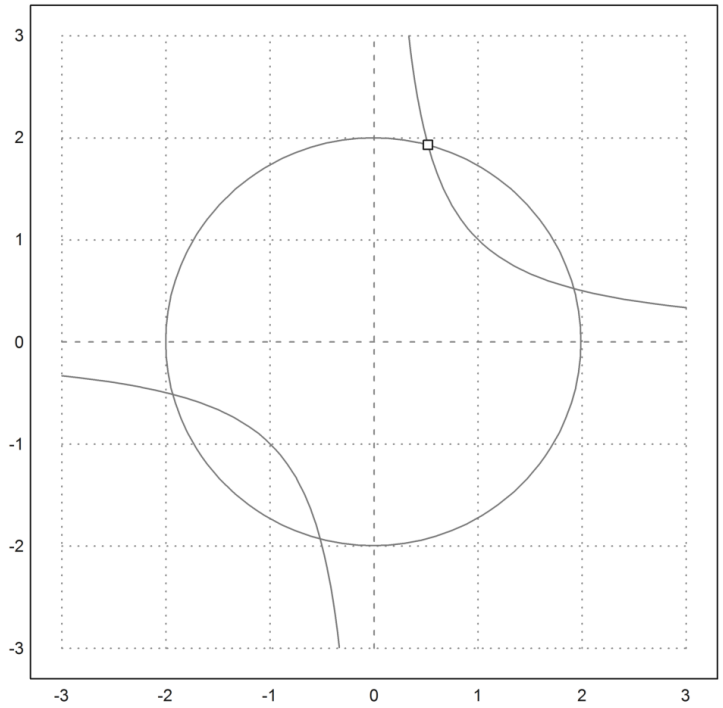



Figure 10.10: Solution of $xy = 1, x^2 + y^2 = 4$

To see the solution, we can use an implicit plot in EMT.

```
>plot2d(f1,level=0,r=3);
>plot2d(f2,level=0,>add);
>plot2d(middle(x[1]),middle(x[2]),>points,>add):
```

# Index