Computer Algebra using Maple
Part I: Basic concepts

Winfried Auzinger, Kevin Sturm (SS 2019)

1 General principles

Goals:

-- Understanding basic principles underlying a computer algebra system

-- What can you expect from such a system, and what not?

-- Interactive usage, possible pitfalls

-- Writing your own programs (i.e., implementing your own solutions)

-- In particular: Introduction to the system Maple (syntax, interactive usage, programming language, libraries)

```
> restart; # restart engine, clear workspace
```

(i) **Maple** (current version: Maple 2016) is an interactive system for doing symbolic and high-precision numerical computations, and for visualization purposes. It comprises a large amount of mathematical knowledge, built-in into the system, and a large number of packages (libraries) for special mathematical and application topics.

Maple is a commercial product by Maplesoft Inc.

(ii) The computational engine of Maple is called the **kernel.** The most important Maple functions are built-in to the kernel as optimized binary code or Maple code.

Many extensions ('**packages**') are not contained in the kernel, but have to be activated by the user if required.

Example: package **Linear Algebra** for symbolic and numerical Linear Algebra.

(iii) The normal use of Maple is **interactive.** You enter a command (Maple input) and get back an answer (Maple output).

In this way you create a **worksheet,** wich you can save, print, read again, modify, and so on. In introduction on interactive opereration on worksheets is given in the lecture.

Besides the worksheet mode, there is also a more general format called document mode. For normal use we prefer worksheet mode. Normal text can also be included similarly as in a typical text processor like Word.

The file type of a Maple worksheet is **mw.** This is stored as a text file in XML format, it contains all input and output.

The contents of the Maple memory is not saved to a worksheet.
If you load a worksheet in order to re-use it, you have to process it again.

Upper case and lower case letters are not identified.

(iv) In its principal mode of operation, a computer algebra system tries to give **exact** answers.

When evaluating, for instance, the square root of 2, you will get

```
> sqrt(2);
```

$$\sqrt{2}$$

Since sqrt(2) is not a rational number, any numerical answer would be **inexact.**

But Maple knows how to compute with such an object:

```
> sqrt(2) * sqrt(2);
```

$$2$$

(v) Maple includes a powerful, **interpreted programming language** with a syntax similar to PASCAL. Many types exist, but there are no strict typing rules and, in general, no declarations are required.

The system is not suitable for running numerically intensive programs. However, it is very suitable for programming symbolic algorithms, for high-precision numerical calculations and as a tool for generating numerical codes.

Support for code parallelization on multi-core machines is provided by the **Grid** package.

(vi) For activating **help** on a command, type ? *command*, e.g,

```
> ? mod
```

(vii) On the CompMath homepage you find a simple worksheet preformatted like this lecture sheet (with larger magenta input font replacing the standard red font).

(viii) NOTE: A (more basic) computer algebra kernel (MuPAD) is also integrated in recent versions

of **Matlab (Symbolic Math Toolbox)**. You may define symbolic variables and perform symbolic operations, e.g., automatic differentiation of expressions.

(ix) Similar systems: e.g.,

Mathematica: commercial product by Wolfram Research

SageMath: Python-based, basic version is free and can be used online, see http://www.sagemath.org/

2 Entering commands

```
> restart;
```

On normal use: the prompt > waits for input:

Enter an expression, followed by ;
Maple interprets the expression and writes output back to the screen:

```
> 1+2+3;
```

$$6$$

Remark: In current versions the ; may be omitted.
However, it is required to separate several command written in s single line
or preceding a comment string (# ......)

In these notes we do not omit ; but it works:

```
> 1+2+3
```

$$6$$

<ctrl><t> removes the prompt > . Now you can enter ordinary text.

<ctrl><m> moves you back to prompt > . Now you can enter math input again.

<ctrl><k> or **<ctrl><j>** inserts a new line (before or after current line) for entering math input.

For brief annotations in the input (comments), just use # in math input mode.

```
> 1; # I am Number One
```

$$1$$

: instead of **;** suppresses the output on screen.

```
> 1: # I am Number One, hiding before yourself
```

This is important for suppressing the output of lengthy results!

```
> 1000!: # This has about 3000 digits
```

You can enter several command in one pass, separated by **;** or **:**
To begin a new line (without immediate evaluation), use **<shift><enter>:**

```
> a;
```

```
A;
```

$$a$$
$$A$$

<shift><enter> is also used for entering more complex multiline code e.g., procedures (to be discussed later on).

2.2 Greek letters

Greek letters can be used in naming Maple objects.
On output, they are displayed in Greek style:

```
> alpha,lambda,pi,Omega;
```

$$\alpha, \lambda, \pi, \Omega$$

3 Basic operations; types of objects

```
> restart;
```

There are many types of objects, hierachically structured
(symbols, integers, fractions, reals, floats, sets, vectors, matrices, ...),
but explicit declarations are generally not required.

3.1 Computing with numbers

Integer arithmetic:

```
> -1+3;
```

$$2$$

```
> whattype(2); # the constant 2 is of type integer
  is(2,real);  # but it is also a real number
```

$$integer$$

$$true$$

Note: **is(***object,property***)** decides if object has a certain property.

Several xpressions separated by comma (a so-called **expression sequence**):

```
> 0+1,3,5,8-1
```

$$1, 3, 5, 7$$

Rational arithmetic, with automatic simplification:

```
> 1+2/6;
```

$$\frac{4}{3}$$

```
> whattype(4/3);
  is(4/3,integer);
```

$$fraction$$

$$false$$

An **irrational** number (square root) :

```
> sqrt(3);
```

$$\sqrt{3}$$

```
> is(sqrt(3),rational);
```

$$false$$

Pi is a predefined transcendental number:

```
> Pi;
```

$$\pi$$

Warning: Euler's constant **exp(1)** is not predefined as a variable.

```
> e, exp(1);
```

$$e, \mathrm{e}$$

Complex numbers (**I** = sqrt(-1) is predefined) :

```
> sqrt(-1);
```

$$\mathrm{I}$$

```
> whattype(I);
  is(I,real);
```

$$complex(extended\_numeric)$$

$$false$$

```
> 1/(1-I);
```

$$\frac{1}{2} + \frac{\mathrm{I}}{2}$$

```
> sin(1+I);
```

$$\sin(1 + \mathrm{I})$$

evalc evaluates to Cartesian form:

```
> evalc(sin(1+I));
```

$$\sin(1)\cosh(1) + \mathrm{I}\cos(1)\sinh(1)$$

For decimal representation (approximation), use **evalf:**

```
> evalf(1/3); # default = 10 digit expansion
```

$$0.3333333333$$

```
> evalf[5](1/3); # 5-digit approximation
```

$$0.33333$$

```
> evalf(sqrt(3));
```

$$1.732050808$$

```
> evalf(Pi);
```

$$3.141592654$$

```
> evalf(exp(I*Pi)+I);
```

$$-1. + \mathrm{I}$$

The environement variable **Digits** (accuracy of evalf) defaults to 10.

It can be set to an arbitrary value:

```
> Digits;
```

$$10$$

```
> Digits := 2;
```

$$Digits := 2$$

```
> evalf(1/3);
```

$$0.33$$

```
> Digits := 10;
```

$$Digits := 10$$

```
> evalf(1/3);
```

$$0.3333333333$$

Note: The *displayed* # of digits is another parameter (interface variable **displayprecision).** See **Tools / Options**

```
> interface(displayprecision=20):
> evalf[5](1/3);
```

$$0.33333000000000000000$$

```
> interface(displayprecision=10):
> evalf[5](1/3);
```

$$0.3333300000$$

Further basic operations:

```
> 2^5; # power
```

$$32$$

```
> 2^(-5), 2^(-5.0); # negative power
```

$$\frac{1}{32}, 0.0312500000$$

```
> 5!;  # factorial
```

$$120$$

Invalid expressions trigger an error message:

```
> 0/0;
Error, numeric exception: division by zero
> 1/0:
Error, numeric exception: division by zero
```

Reuse of results using ditto operator **%** (use with care):

```
> evalf(4/12);
```

$$0.3333333333$$

```
> %^2;
```

$$0.1111111111$$

% represents the recently computed result. Furthermore: **%%** **%%%**

Quick but dirty.

3.2 Computing with variables (symbols)

```
> x; # Hi, I am a variable. My name is x. I have no value
  preassigned.
```

$$x$$

```
> whattype(x);
```

$$symbol$$

For general variables, normal rules of field arithmetic (real or complex) are assumed:

```
> x+x;
```

$$2\,x$$

```
> (x+y)-x;
```

$$y$$

```
> x*y - y*x;
```

$$0$$

You can perform symbolic operations,
or mixed numeric and symbolic operations.
Basic simplifications are automatically applied.

```
> (x^2)^3;
```

$$x^6$$

```
> 1/(5*% + y^2);
```

$$\frac{1}{5\,x^6+y^2}$$

```
> sqrt(%);
```

$$\sqrt{\frac{1}{5\,x^6+y^2}}$$

3.3 [Un]assiging variables; substituting values for variables

Any valid Maple object can be assigned a **name** by assigning it to a **variable.**

:= is the **assignment** operator.

```
> x := 3.14;
```

$$x := 3.1400000000$$

Now, x takes the value 3.14.

```
> x+x;
```

$$6.2800000000$$

```
> evalf(Pi-x);
```

$$0.0015926540$$

This resets x to be **unassigned:**

```
> x := 'x':  # or: unassign('x');
> y := x+x:
> y;
```

$$2\,x$$

NOTE:
In any case, '*variable*' (with quotes) will return the **name** of the variable (assigned or not).

```
> x := 3.14;
```

$$x := 3.1400000000$$

```
> whattype(x);
```

$$\mathit{float}$$

```
> x,'x';
```

$$3.1400000000, x$$

```
> %;
```

$$3.1400000000, 3.1400000000$$

Here, the foregoing result (a sequence consisting of 2 values) was evaluated.

NOTE:
Evaluation of '*expression*' removes quotes, and evaluates what is inside.

A similar example:

```
> y:=z;
```

$$y := z$$

```
> ''y'';
```

$$'y'$$

```
> %;
```

$$y$$

```
> %;
```

$$z$$

HINT: Often it is useful to enter an expression quoted, to see (essentially) the 'input as output', and then evaluate:

```
> a:=1:
> '(a+b)^3'; %;
```

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

Syntax for **multiple** simulteneous assignmenmt:

```
> a,b,c := 1,2,3;
```

$$a, b, c := 1, 2, 3$$

Use **subs** to substitute a particular value for a variable in a symbolic expression:

```
> expr := (x+1)^n;
```

$$expr := 4.1400000000^n$$

```
> subs(x=1,expr); # substitute value 1 for x
```

$$4.1400000000^n$$

```
> subs(x=1,n=5,expr); # multiple substitution
```

$$1216.1907770000$$

3.4 Using built-in functions

Maple comes with a large collection of built-in functions,
which can, in general, be applied to numeric or symbolic values.

Examples:

```
> evalf(Pi); # floating-point evaluation (approximation)
```

$$3.1415926540$$

```
> sqrt(Pi); # square root
```

$$\sqrt{\pi}$$

```
> min(Pi,3.14), max(Pi,3.14,3.141); # minimum, maximum
```

$$3.1400000000, \pi$$

```
> iquo(11,3);   # integer division
  irem(11,3);   # remainder
```

$$3$$
$$2$$

```
> binomial(n,n-1); # binomial coefficient
```

$$n$$

```
> sin(Pi), cos(Pi); # trigonometric functions
```

$$0, -1$$

```
> arcsin(1), arccos(1); # inverse trigonometric functions
```

$$\frac{\pi}{2}, 0$$

```
> exp(0), ln(1); # exponential and logarithmic functions
```

$$1, 0$$

```
> quo(z^3+z+1,z^2,z); # polynomial division
  rem(z^3+z+1,z^2,z); # remainder
```

$$z$$
$$z + 1$$

... and many, many others ...

3.5 Strings

Strings are objects for storing text:

```
> s:="I am a string"; whattype(s);
```

$$s := \text{"I am a string"}$$

$$string$$

```
> s[3]; # third character of string
```

$$\text{"a"}$$

```
> s[1..3]; # substring
```

$$\text{"I a"}$$

3.6 Manipulating and converting expressions

Often a result of a symbolic computation is the outcome of a mathematical conversion rule. In Maple, there are several ways how you can influence such simplifications and conversions.

simplify:

This command tries to represent an expresssion in a 'simple' form; however, this may not always represent what you are aiming for, because ' simple' is not well-defined and context-dependent.

```
> w := u^2+2*u*v+v^2;
```

$$w := u^2 + 2\,u\,v + v^2$$

```
> simplify(w); # ? Is this the most 'simple' form ?
```

$$(u + v)^2$$

expand:

Well-defined operation multiplying out product expressions:

```
> w := (u+v)^3;
```

$$w := (u + v)^3$$

```
> w_expanded := expand(w);
```

$$w\_expanded := u^3 + 3\,u^2\,v + 3\,u\,v^2 + v^3$$

factor:

Tries to factorize into a product (a converse to **expand**).

```
> factor(w_expanded);
```

$$(u + v)^3$$

```
> factor(z^2+4*z+4);
```

$$(z + 2)^2$$

NOTE:

Some Maple functions do not try to simplify their output automatically.
Thus, manually invoking **simplify** is often a good idea, especially in order not to overlook trivial simplifications.

Manipulating rational expressions:

```
> r := 1/u + v/(u+v);
```

$$r := \frac{1}{u} + \frac{v}{u+v}$$

```
> r:=normal(r); # normalize as 'numerator/denominator'
```

$$r := \frac{u\,v + u + v}{u\,(u+v)}$$

```
> numer(r), denom(r); # numerator, denominator
```

$$u\,v + u + v,\, u\,(u+v)$$

convert is a general conversion tool with many different options.

Example:

```
> convert(r,parfrac,v); # partial fraction decomposition w.r.
  t. v
```

$$\frac{u+1}{u} - \frac{u}{u+v}$$

3.7 Sums and such: add, sum, mul, product. Implicit loop syntax

The commands for generating sequences and summation, multiplication of sequences of of several values, use an **implicit loop construct** with an arbitrary index variable.

In these constructs, the

'from-to operator' ..

is used.

```
> seq(i,i=1..10);
```

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

```
> seq(i,i=1..10,2); # stride 2
```

$$1, 3, 5, 7, 9$$

```
> seq(i,i=10..1,-1); # negative stride -1
```

$$10, 9, 8, 7, 6, 5, 4, 3, 2, 1$$

```
> add(i^2,i=1..5); # add up
```

$$55$$

```
> add(i^2,i=1..10,3); # add up; stride 3
```

$$166$$

? Does Maple know the general formula for such a sum ?

```
> add(k^2,k=1..n); # n has no value assigned!
Error, unable to execute add
```

NOTE:

add can only add a finite set of values, like a calculator.

In contrast,

sum is able to perform **symbolic summation** (but not stride <>1 allowed):

```
> sum(k^2,k); # indefinite summation (without summation constant)
```

$$\frac{1}{3}k^3 - \frac{1}{2}k^2 + \frac{1}{6}k$$

```
> simplify(subs(k=k+1,%)-%); # check: O.K.
```

$$k^2$$

```
> sum(k^2,k=1..n); # definite summation
```

$$\frac{(n+1)^3}{3} - \frac{(n+1)^2}{2} + \frac{n}{6} + \frac{1}{6}$$

```
> factor(%);
```

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

An **infinite** series (summming up to infinity):

```
> sum(1/n^2,n=1..infinity);
```

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

Analogous commands for products instead of sums:

mul and **product**

```
> mul(i^2,i=1..5); # calculator
```

$$14400$$

```
> sum(i^2,i=1..n); # finds symbolic product expression
```

$$\frac{(n+1)^3}{3} - \frac{(n+1)^2}{2} + \frac{n}{6} + \frac{1}{6}$$

(NOTE: On output, factorials are usually expressed via the Gamma function.)

```
> n!; GAMMA(n+1);
```

$$n!$$

$$\Gamma(n+1)$$

```
> simplify(%-%%);
```

$$0$$

WARNING: **sum** and some other commands treat index variables as *global* (**? bug or feature ?**) :

```
> i := 1;
```

$$i := 1$$

```
> sum(i,i=1..5); # this does not work!
Error, (in sum) summation variable previously assigned, second argument evaluates to 1 = 1 .. 5
> sum('i','i'=1..5); # OK; or first unassign('i');
```

$$15$$

But:

```
> add(i,i=1..5); # O.K.
```

$$15$$

3.8 Logical operations

The logical (Boolean) constants are **true, false,** and **FAIL** (undecidable).

Usually, boolean values result from checking equalities or inequalites, or the like.
These are evaluated using **evalb:**

```
> 1=1, 1=2; #  these are mathematical objects, namely
  equations
```

$$1=1,\, 1=2$$

```
> evalb(1=1), evalb(1=2);
```

$$true, false$$

```
> q := evalb(1<>2);
```

$$q := true$$

The following result is **generic,** i.e., it is valid in general
(irrespective of possible special cases, i.e. when X is the same as Y)

```
> X,Y; evalb(X=Y);
```

$$X, Y$$
$$false$$

Logical relations are combined using **and, or**, **xor:**

```
> evalb(((a=b) or (b=c)) and (c<>e xor c<>d));
```

$$false$$

Negation via **not:**

```
> evalb(not(1=2));
```

$$true$$

We can solve equations using **solve:**

```
> eq := (x+1=a^2-2*x);
```

$$eq := 4.1400000000 = -5.2800000000$$

```
> solve(eq,x); # solve for x
Warning, solving for expressions other than names or functions is not recommended.
Error, (in solve) a constant is invalid as a variable, 3.14
```

4 Graphics

There are many tools for 2D and 3D plots and animations.
We will mainly explore some of them in course of the practical exercises.

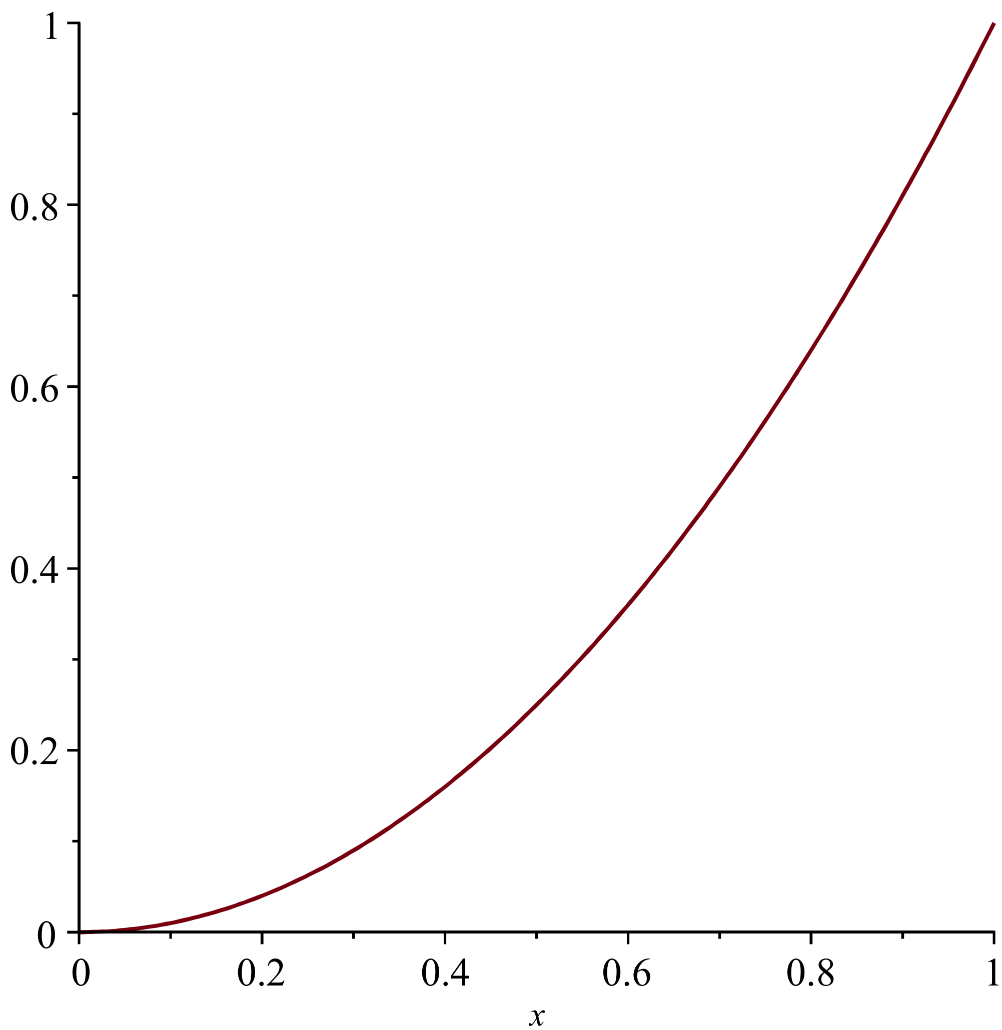
Plot commands generate **plot data structures** and immediately display the corresponding figure.
These plot structures can also be stored (assigned to variables) for later use.

Plots can be **manipulated interactively** (context menu) and **exported** to different formats.

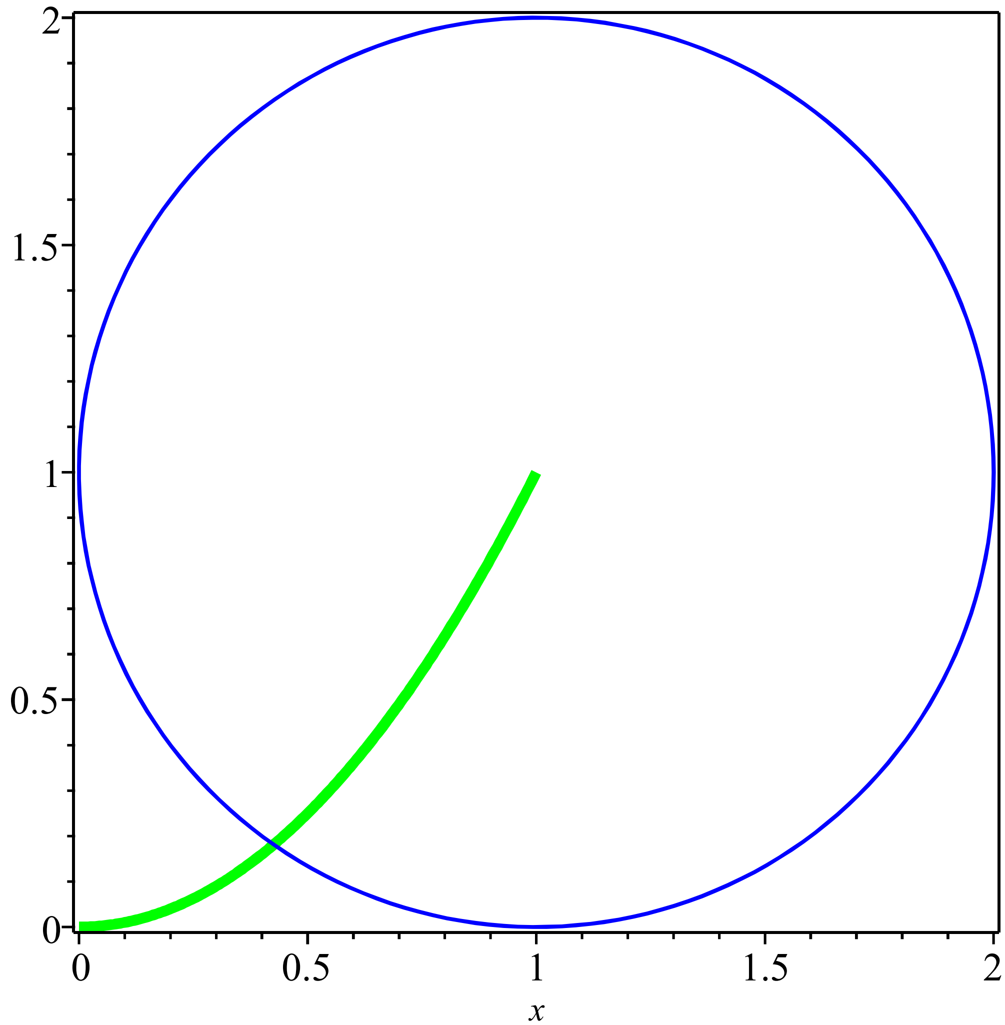
The package **plots** contains many different plotting commands, in particular, **display** for displaying plot structures.
The package **plottools** provide many pre-defined graphical objects in form of plot structures.

Some examples:

```
> x:='x':
> plot(x^2,x=0..1);
```



```
> plot1:=plot(x^2,x=0..1,axes=boxed,color=green,thickness=4):
> plot2:=plottools[circle]([1,1],1,color=blue):
> plots[display](plot1,plot2);
```



5 Fundamental data structures

```
> restart;
```

The fundamental data structures are

expression sequences (type exprseq)

lists (type list)

sets (type set)

of **arbitrary Maple objects**. These serve as 'data containers' for collecting related objects, e.g., names of variables or equations in a system of equations. Objects of **different types** may also be collected in this way.

Note that expression sequences, lists and sets are essentially **static, read-only** data structures. You can create and work with such objects but you cannot change them (rather create new ones from given ones).

There is one *exception:* You can change an individual element of a list (see **4.3**).

5.1 A basic construct: expression sequences

Any sequence of Maple objects separated by commas is an **expression sequence.**

```
> es := alpha,beta,gamma;
```

$$es := \alpha, \beta, \gamma$$

```
> whattype(es);
```

$$exprseq$$

```
> es[2];  # second component of es
```

$$\beta$$

The **seq** command is a constructor for an expressions sequence based on an implicit loop:

```
> s := seq(i^2,i=1..5);
```

$$s := 1, 4, 9, 16, 25$$

Different types of objects in a sequence:

```
> s := 1,a,Vector(2),Matrix(2); # Vector, Matrix: explained
  later on
```

$$s := 1, a, \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

The expression sequence is a basic syntactic construct which is used in the definition of 'higher' data structures, with particular properties.

Extract subsequences:

```
> s[1..3]; # index from 1 to 3
  s[..2];  #       from start to 2
  s[3..];  #       from 3 to end
  s[..];   # all
```

$$1, a, \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$1, a$$

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$1, a, \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

seq constructs with general increments:

```
> seq(i,i=1..7,2);  # an increment > 1
```

$$1, 3, 5, 7$$

```
> seq(i,i=8..1,-3); # a negative increment
```

$$8, 5, 2$$

Shortcut for arithmetic progression:

```
> $1..10; # same as seq(i,i=1..10);
```

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

5.2 Sets

A (finite) **set** is an expression sequence enclosed in **{ }**.

It represents a finite set (in the sense of set theory) consisting of Maple objects, with no particular order. Thus, the `i-th element' in a set is not well-defined.

```
> my_set := {0,1,x,Pi,NULL}; # NULL means 'nothing' (empty
  object)
```

$$my\_set := \{0, 1, \pi, x\}$$

```
> numelems(my_set);      # number of elements
```

$$4$$

```
> whattype(my_set);
```

$$set$$

```
> A,B := {seq(i^2,i=1..5)},{seq(i^2,i=3..7)};
```
$$A, B := \{1, 4, 9, 16, 25\}, \{9, 16, 25, 36, 49\}$$

Union, intersection, difference:

```
> 'A union B'; %;
```
$$A \cup B$$
$$\{1, 4, 9, 16, 25, 36, 49\}$$

```
> 'A intersect B'; %;
```
$$A \cap B$$
$$\{9, 16, 25\}$$

```
> 'A minus B'; %;
```
$$A \setminus B$$
$$\{1, 4\}$$

```
> U := A union B; # Now, also U is a set.
```
$$U := \{1, 4, 9, 16, 25, 36, 49\}$$

```
> U[]; # extract all elements as expression sequence
```
$$1, 4, 9, 16, 25, 36, 49$$

Empty set:

```
> emptyset := {};
```
$$emptyset := \varnothing$$

```
> emptyset intersect A;
```
$$\varnothing$$

in decides whether an object is contained in a set:

```
> '5 in A'; evalb(%);
```
$$5 \in A$$
$$false$$

subset decides whether a set is a subset of another set:

```
> 'A subset U'; evalb(%);
```
$$A \subseteq U$$
$$true$$

5.3 Lists

A **list** is an expression sequence enclosed in **[ ]**.
It represents a finite sequence of Maple objects, with a particular **order** defined by the **position** in the list.

In contrast to sets, multiple occurrence is well-defined.

* Like sets, lists are **static** data structures. On initialization, the number of elements gets fixed.

* Internally, a list is an array of pointers to its entries.

* Like sets, lists are essentially **read-only** data structures. The only possible `write-operation' is to change the value of a single element (see below).

```
> L := [eins,2,3,a,x,y,oops];
```

$$L := [eins, 2, 3, a, x, y, oops]$$

```
> whattype(L);
  is(L,list);
```

$$list$$

$$true$$

```
> L[1],L[2]; # first element, second element, ...
```

$$eins, 2$$

```
> L[-1],L[-2]; # last element, second to last element, ...
```

$$oops, y$$

```
> subL := L[2..4]; # extract sublist
```

$$subL := [2, 3, a]$$

```
> L[]; # extract all elements as expression sequence
```

$$eins, 2, 3, a, x, y, oops$$

```
> numelems(L);      # number of elements
```

$$7$$

Change the value of an element in a list (this is a valid operation):

```
> L := [1,1,2,3];
```

$$L := [1, 1, 2, 3]$$

```
> L[4]:=999: L;
```

$$[1, 1, 2, 999]$$

Like for sets, **in** decides whether an object is contained in a list:

```
> 999 in L; evalb(%);
```

$$999 \in [1, 1, 2, 999]$$

$$true$$

A conversion list -> set, and back:

```
> convert(L,set); # this removes multiple elements
```

$$\{1, 2, 999\}$$

```
> convert(%,list);
```

$$[1, 2, 999]$$

An empty list:

```
> [];
```

$$[\ ]$$

Adding an element to a list:
The list has to be rebuilt from scratch.

```
> L;
  L:=[op(L),new_element];
```

$$[1, 1, 2, 999]$$
$$L := [1, 1, 2, 999, new\_element]$$

NOTE: set-theoretical operations do not apply to lists.

5.4 Special form of implicit loop, using in

With the following syntax for an implicit loop, all elements of an expression sequence, a set, or a list can be addressed. It is not neccessary to count the number of elements.

```
> es := alpha,beta,gamma,delta;
```

$$es := \alpha, \beta, \gamma, \delta$$

```
> seq(x^2, x in es);
```

$$\alpha^2, \beta^2, \gamma^2, \delta^2$$

```
> add(x^2, x in es);
```

$$\alpha^2 + \beta^2 + \delta^2 + \gamma^2$$

```
> ses := {es};
```

$$ses := \{\alpha, \beta, \delta, \gamma\}$$

```
> add(x^2, x in ses);
```

$$\alpha^2 + \beta^2 + \delta^2 + \gamma^2$$

```
> les := [es];
```

$$les := [\alpha, \beta, \gamma, \delta]$$

```
> mul(x, x in les);
```

$$\alpha \beta \gamma \delta$$

SHORT syntax for this (new): e.g.,

```
> seq(les)
```

$$\alpha, \beta, \gamma, \delta$$

```
> add(les);   #  analogous to sum in Matlab
```

$$\alpha + \beta + \gamma + \delta$$

```
> mul(les);   #  analogous to prod in Matlab
```

$$\alpha \beta \gamma \delta$$

5.5 Shortcuts for selecting or removing objects from a data structure

Two examples demonstrate the functionality:

```
> numbers:=[$1..10];
```

$$numbers := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

```
> select(isprime,numbers); # isprime checks prime number
  property (true / false)
```

$$[2, 3, 5, 7]$$

```
> remove(isprime,numbers);
```

$$[1, 4, 6, 8, 9, 10]$$

```
> selectremove(isprime,numbers);
```

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

Analogously with a criterion represented by your own Boolean function: (see Sec. 6 for user-defined functions):

```
> divisible_by_3 := x->evalb(x mod 3=0): # this is a user-
  defined function - see next section!
```

```
> select(divisible_by_3,numbers);
```

$$[3, 6, 9]$$

6 User-defined functions

```
> restart;
```

You can define your own functions using **arrow notation**, assign it to a name, and use it like a built-in function.

6.1 This is NOT a function:

Consider

```
> f(x) := x^3;
```

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

```
> f(x);
```

$$x^3$$

This looks like the definition of a function, but it is **not**. Here, **f(x)** has been defined as a 'synonym' for the expression **x^3**, but you cannot pass an argument to this 'function'.

```
> f(x), f(2), f(z);
```

$$x^3, f(2), f(z)$$

6.2 This IS a function:

Use **'arrow' notation.**

Note that the name of the argument (dummy variable; here: x) is irrelevant, just like the name of an index in a sum.

```
> x -> x^2;
```

$$x \mapsto x^2$$

```
> (y -> y^2)(z);
```

$$z^2$$

```
> (mickey_mouse -> mickey_mouse^2)(3);
```

$$9$$

More practically: Give it a name, i.e., assign it to a variable.
Then the value of this variable is the function!

```
> f := y -> y^3; eval(f);
```

$$f := y \mapsto y^3$$
$$y \mapsto y^3$$

```
> f(2), f(z);
```
$$8, z^3$$

For functions in several variables, use an argument sequence enclosed by (...) :

```
> g := (x,y) -> log[y](x); # logarithm of x w.r.t base y
```
$$g := (x, y) \to \log_y(x)$$

```
> g(u,2);
```
$$\frac{\ln(u)}{\ln(2)}$$

```
> h := (u,v,w) -> u*v + w;
```
$$h := (u, v, w) \mapsto v\, u + w$$

```
> h(alpha,beta,gamma);
```
$$\beta\, \alpha + \gamma$$

Function arguments can be of general type.
The following function computes the sum of elements of two lists:

```
> lsum := (l1,l2) -> add(l1)+add(l2);
```
$$lsum := (l1, l2) \mapsto add(l1) + add(l2)$$

```
> lsum([1,2],[3,4,5]);
```
$$15$$

6.3 Functions as argument and/or result of a function

Function arguments and results may be `arbitrary' objects, e.g., again functions.

Look at the following example:

```
> finv := f -> (x->f(1/x));
```
$$finv := f \mapsto x \mapsto f\left(\frac{1}{x}\right)$$

```
> g:=finv(exp);
```
$$g := x \mapsto \exp\left(\frac{1}{x}\right)$$

```
> g(x);
```
$$e^{\frac{1}{x}}$$

7 Equations and systems of equations

```
> restart;
```

7.1 A very simple equation. evalb and solve

An expression of the form

Maple object = Maple object

is of type equation (`=`).

```
> eq := x=1; whattype(%);
```

$$eq := x = 1$$
$$`=`$$

Is eq true for false? Use **evalb:**

```
> evalb(eq);
```

$$false$$

This answer is 'generically correct', because x is not defined.
Only in the very special case, where x takes the value 1, eq would be true.

```
> evalb(subs(x=1,eq));
```

$$true$$

Let's **solve** eq for x:

```
> solve(eq,x);
```

$$1$$

Note that the solution is NOT automatically assigned to the variable name:

```
> x;
```

$$x$$

assign converts an equation into a an assignment:

```
> x=1; x;
```

$$x = 1$$
$$x$$

```
> assign(%%); x;
```

$$1$$

7.2 Some more interesting equations

Solving a linear equation with respect to a variable (y):

Note that the solution is 'generically correct', but it would make no sense in the special case a=0.

```
> solve(a*y+b=c,y);
```

$$-\frac{b-c}{a}$$

Assign result to variable:

```
> y:=%;
```

$$y := -\frac{b-c}{a}$$

A quadratic equation:

```
> x:='x';
  eq := a*x^2 + b*x + c;
```

$$x := x$$

$$eq := a\,x^2 + b\,x + c$$

```
> sol :=solve(eq,x);
```

$$sol := \frac{-b+\sqrt{-4\,a\,c+b^2}}{2\,a},\ -\frac{b+\sqrt{-4\,a\,c+b^2}}{2\,a}$$

This is the general pair of solutions, returned in form of an expression sequence.

We convert it to a list and **substitute** values for the parameters using **subs**:

```
> sol := [sol];
  subs(a=1,b=2,c=3,sol);
```

$$sol := \left[\frac{-b+\sqrt{-4\,a\,c+b^2}}{2\,a},\ -\frac{b+\sqrt{-4\,a\,c+b^2}}{2\,a}\right]$$

$$\left[-1+\frac{\sqrt{-8}}{2},\ -1-\frac{\sqrt{-8}}{2}\right]$$

```
> evalf(%);
```

$$[-1.0000000000 + 1.4142135620\,I,\ -1.0000000000 - 1.4142135620\,I]$$

A cubic equation:

Here, Maple uses Cardano's formula.

```
> eq := x^3+x^2+2*x+1;
```

$$eq := x^3 + x^2 + 2\,x + 1$$

```
> solve(eq,x); # eq is shortcut for eq=0
```

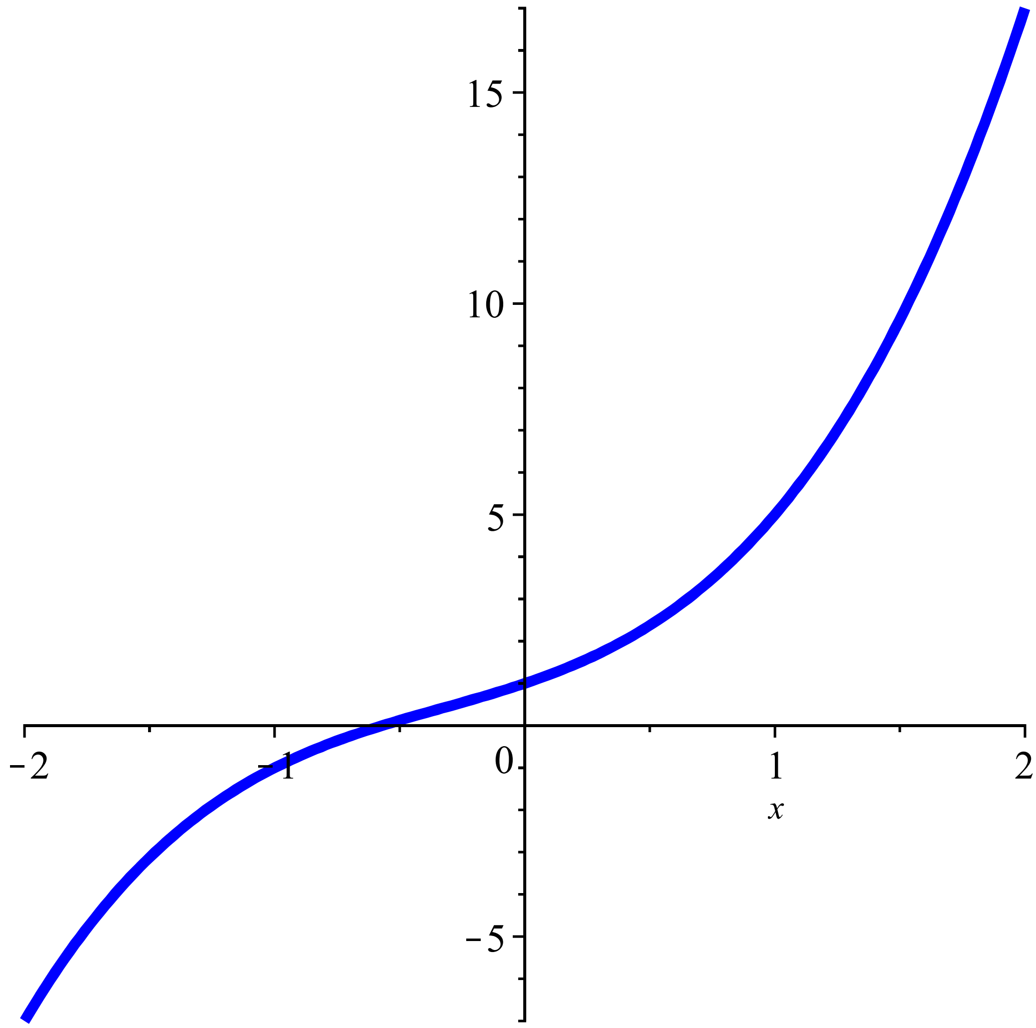
$$-\frac{\left(44+12\sqrt{69}\right)^{1/3}}{6}+\frac{10}{3\left(44+12\sqrt{69}\right)^{1/3}}-\frac{1}{3}, \frac{\left(44+12\sqrt{69}\right)^{1/3}}{12}-\frac{5}{3\left(44+12\sqrt{69}\right)^{1/3}}-\frac{1}{3}+\frac{I\sqrt{3}\left(-\frac{\left(44+12\sqrt{69}\right)^{1/3}}{6}-\frac{10}{3\left(44+12\sqrt{69}\right)^{1/3}}\right)}{2}, \frac{\left(44+12\sqrt{69}\right)^{1/3}}{12}-\frac{5}{3\left(44+12\sqrt{69}\right)^{1/3}}-\frac{1}{3}-\frac{I\sqrt{3}\left(-\frac{\left(44+12\sqrt{69}\right)^{1/3}}{6}-\frac{10}{3\left(44+12\sqrt{69}\right)^{1/3}}\right)}{2}$$

```
> evalf([%]);
```

$$[-0.5698402912, -0.2150798545-1.3071412790\, I, -0.2150798545+1.3071412790\, I]$$

A plot of the function defining the equation:

```
> plot(eq,x=-2..2,axes=normal,thickness=4,color=blue);
```



A higher-order polynomial equation:

```
> eq := x^5 + x^2 - 1;
```

$$eq := x^5 + x^2 - 1$$

```
> sol := solve(eq=0,x);
```

$sol := RootOf(\_Z^5 + \_Z^2 - 1, index = 1), RootOf(\_Z^5 + \_Z^2 - 1, index = 2), RootOf(\_Z^5 + \_Z^2 - 1, index = 3), RootOf(\_Z^5 + \_Z^2 - 1, index = 4), RootOf(\_Z^5 + \_Z^2 - 1, index = 5)$

An exact solution representation cannot be found. A *RootOf* expression means: This number is a root of the given equation.

We can evaluate it this numerically or use the numerical solver **fsolve**:

```
> evalf(sol);
```

$0.8087306005, 0.4649122016 + 1.0714738400\,I, -0.8692775018 + 0.3882694066\,I, -0.8692775018 - 0.3882694066\,I, 0.4649122016 - 1.0714738400\,I$

```
> fsolve(eq); # this finds only the real root.
```

$$0.8087306005$$

```
> fsolve(eq,complex); # this finds all roots.
```

$$-0.8692775018 - 0.3882694066\, I,\ -0.8692775018 + 0.3882694066\, I,\ 0.4649122016 - 1.0714738403\, I,\ 0.4649122016 + 1.0714738403\, I,\ 0.8087306005$$

A transcendental equation:

```
> eq := exp(x)-3*x;
```

$$eq := e^x - 3\,x$$

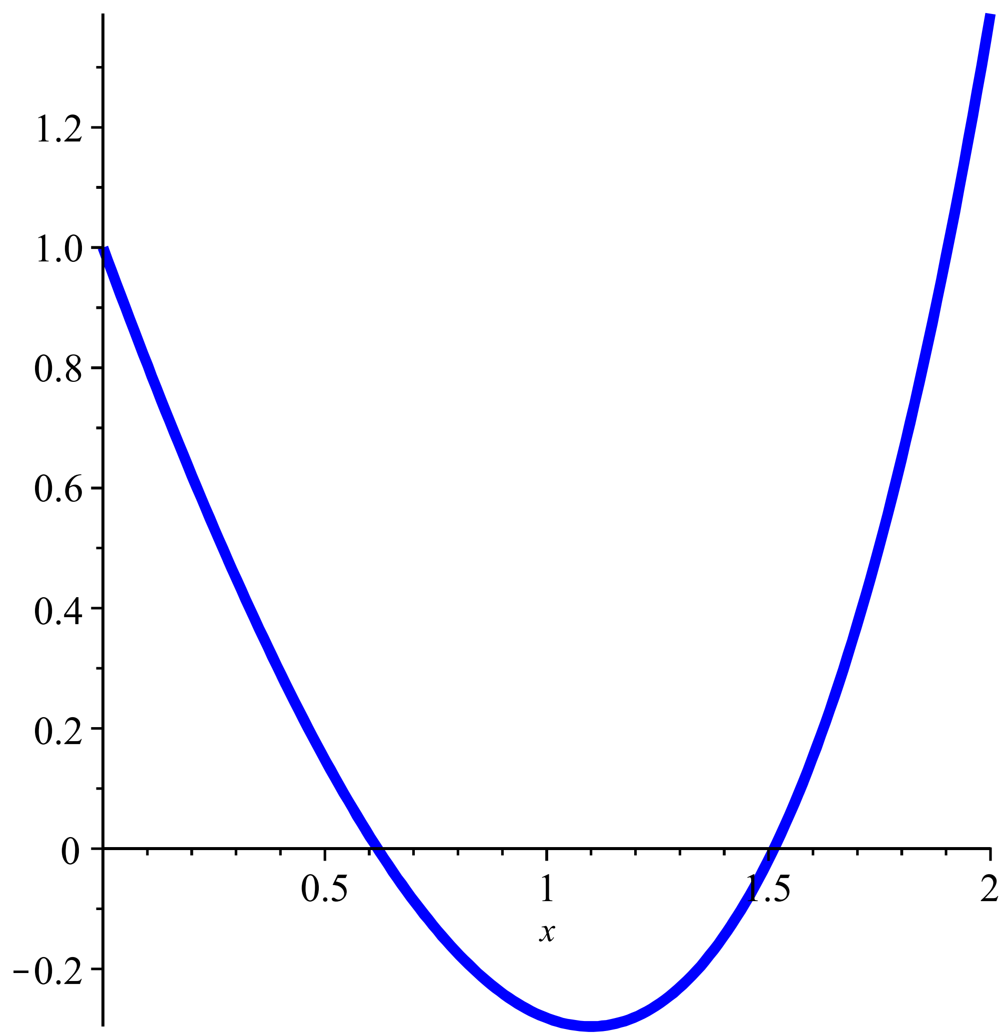
```
> solve(eq); # shortcut for solve(eq=0,x);
```

$$-\mathrm{LambertW}\left(-\frac{1}{3}\right),\ -\mathrm{LambertW}\left(-1, -\frac{1}{3}\right)$$

```
> evalf(%);
```

$$0.6190612866,\ 1.5121345520$$

```
> plot(eq,x=0..2,axes=normal,thickness=4,color=blue);
```



7.3 A system of two linear equations

We use **variable names with indices,**
and lists as containers for equations and variable names.

In this case, **solve** delivers solutions in form of lists or lists or lists.

This system has no solution:

```
> variables := [x[1],x[2]];
  equations := [x[1] + 2*x[2] = 2,
                2*x[1] + 4*x[2] = 5];
```

$$variables := [x_1, x_2]$$

$$equations := [x_1 + 2\,x_2 = 2, 2\,x_1 + 4\,x_2 = 5]$$

```
> solve(equations,variables);
```

$$[\ ]$$

solve delivers an empty list: A solution does not exist.

This system has a unique solution:

```
> variables := [x[1],x[2]];
  equations := [x[1] + 2*x[2] = 2,
                2*x[1] + 5*x[2] = 6];
```

$$variables := [x_1, x_2]$$

$$equations := [x_1 + 2\,x_2 = 2, 2\,x_1 + 5\,x_2 = 6]$$

```
> solve(equations,variables);
```

$$[[x_1 = -2, x_2 = 2]]$$

This system has infinitely many solution:

```
> variables := [x[1],x[2]];
  equations := [x[1] + 2*x[2] = 2,
                2*x[1] + 4*x[2] = 4];
```

$$variables := [x_1, x_2]$$

$$equations := [x_1 + 2\,x_2 = 2, 2\,x_1 + 4\,x_2 = 4]$$

```
> solve(equations,variables);
```

$$[[x_1 = 2 - 2\,x_2, x_2 = x_2]]$$

Here, x[2] can take an arbitrary value (linear solution manifold).

For larger systems of linear equations, vector and matrix notation is employed
(see also discussion of package **LinearAlgebra** - later).

== end of Part I ==