Manual for Maple

This is a very condensed manual for Maple. Its main purpose is to serve as a handy reference, therefore the material is organized by topics, not by difficulty/usefullness. This makes it less suitable as a guide to learn Maple. However, if at first reading the reader skips parts that seem unimportant, it is also possible to learn Maple from this manual. As a guide we use star to mark sections that can be safely skipped at first.

This manual introduces the main core of Maple, things that probably everybody who want to use Maple needs to know. Thus it can serve as a springboard for further exploration if you want to become an advanced user. In particular, we do not really look into programming here. To become an expert in Maple, one definitely needs more than a condensed manual. However, a casual user may well do with what is written here, in particular thanks to a pretty good on-line help in Maple.

Maple has lots of versions, and changes in behaviour are sometimes baffling. In order to avoid messy discussions, in this manual we restricted ourselves to things that were tested to work with Maple 9, 12 and 13. Obviously this means that newer versions offer more capabilities than we show here, but most just extend topics described below and Maple's built-in help system should be enough to guide you. We refer to the very same system for packages that were only quickly introduced here (or skipped entirely), the purpose of this manual is to show a common trunk, you can branch out depending on your needs.

Newer versions of Maple offer GUI, clickable graphical interface for entering formulas. We do not address this interface here, we feel that the traditional command-line style is faster, safer and more convenient for a user that knows what (s)he is doing. Fortunately, newer versions can be switched to the command-line interface.

0. Intro

Work is done in a "worksheet" which is divided into *execution groups*, a group is marked by a square bracket along the left edge. On its first line there is the prompt `>` where you type. Kernel executes when you hit Enter, and it executes everything that is written in the current execution group regardless of actual position of cursor within this group (the only exception is text typed after `#` which is considered a comment). This execution group then also contains the output from the kernel. We can go back and fro within worksheet and re-execute groups by hitting Enter in any order; then one has to be careful since contents of some registers may have changed from what it used to be when this group was executed previously.

Ctrl-K inserts an execution group above the current position, Ctrl-J below the current position. Execution groups can be split and joined.

Execution groups can be of two kinds. Typical execution groups contain commands as explained above. This is the default kind of execution group. Another type is the text group: It allows for some explanations to be parts of a worksheet, Ctrl-T converts an execution group into a text group. Within a text group Enter works as usual - starts a new line.

We will naturally focus on execution groups with commands. Commands are entered on cursor line and ended with semicolon `;` or colon `:` (in which case the output of this particular command is not shown). A command for a restart (clearing variables, unlodading packages) thus looks like this: `>restart;`

One can put more commands on one line, each ended with semicolon or colon. Conversely, a command can be entered on more lines, Shift+Enter goes to the next line without executing.

Maple is case sensitive!

One can refer to the output of previously processed commands: The percent sign `%` represents the output of the last command, `%%` is the second last output etc). Careful! It means the last (second last etc.) command processed, not the previous (second previous etc.) command in the worksheet. Precisely, it refers to output hidden in memory, This makes a difference in case you move within a worksheet, executing groups in a different order than they were entered.

One can get information about inner workings of Maple by telling it to be more verbose.
Example: `infolevel[solve]:=0` (default) or `infolevel[solve]:=1` or `infolevel[solve]:=2` ...

`help` calls Maple's help

`?limit` provides Maple's help for `limit`

`example(plot)` shows examples for `plot`

Input and output can be simplified by introducing aliases. There are two ways to do so. We can introduce a symbolic name a for any expression b using `macro(`a`=`b`)` or `alias(`a`=`b`)`. What is the difference? Alias affects both input and output. For instance, after `>alias(C=cos);` we can type `>evalf(C(Pi));` and we get -1 as expected, on the other hand, after `>diff(sin(x),x);` we get $C(x)$ instead of $\cos(x)$ (change also in the output). Moreover, expressions are evaluated immediately when using `alias`. On the other hand, `macro` delays evaluation of expressions and affects only input. After `>macro(C=cos);` we do `>evalf(C(Pi));` to get -1, but `>diff(sin(x),x);` gives the usual $\cos(x)$. Actually, `macro` is more powerful than `alias`, since it allows also for tricks like this: If we do `>macro(sin(x)=0);`, then `>sin(x)+1;` yields 1.

We will now stop writing those `>`, the reader knows that commands are typed at prompt and ended with a (semi)colon. We will use it only exceptionally when refering to a concrete conversation with Maple.

Macro also protects the new symbol from another assignment, which makes it very useful for defining constants that we want to keep safe, say, `macro(s=1.75)`. What if we want to use `s` for something else? We release it with `macro(s=s)`.

1. Data types

1.1. Numbers

Numbers and algebraic operations work as usual (including precedence rules). Example: `7*(3!-7)/4^2`, note that multiplication signs are obligatory. For the power `a^b` there is an equivalent notation `a**b`. Maple prefers to work symbolically (with integers, fractions and constants), it evaluates in floating point only if at least one of the numbers in the input is a floating point number or if we force it to. For instance, `>6/12;` yields $\frac{1}{2}$, while `>6./12;` yields 0.5.

The command `evalf(`*ex*`)` evaluates expression *ex* as a floating point number. For instance, after `>6/12;` we can do `>evalf(%);` and get 0.5. Note that this is not the same as `evalf(6/12)` since this command forces evaluation of its argument in float, therefore this way it would skip the symbolical simplification to 1/2 and evaluate 6 divided by 12 numerically. This can be less precise. It is important to know how precisely Maple evaluates. Maple default is 10 digits of accuracy. The command `Digits:=`*n* tells it to do all subsequent calculations with *n* digits accuracy. One can also specify precision just for one evaluation as a parameter, e.g. `evalf(`*ex*`,15)`.

Very large numbers can be inputed using exponents, but there is a diference. The notation *m*`*10^`*n* creates a symbolic expression while *m*`e`*n* creates a floating point number, see the difference described above.

Operations: The usual algebraic operations, also factorial. For roots there are three possibilities. The power as in `x^(1/n)` is kept symbolical as much as possible, to change it into a number one uses `evalf`. The command `root(`*ex*`,`*n*`)` or also `root[`*n*`](`*ex*`)` yields the principal root defined as $e^{\frac{1}{n}\ln(ex)}$, it supplies a number, but note that sometimes it can yield a complex numbers although there are real roots available. For real roots use `surd(`*ex*`,`*n*`)`, for the square root also `sqrt(`*ex*`)`, for instance `sqrt(2)` gives $\sqrt{2}$, `evalf(sqrt(5),2)` gives 2.24. One can convert, for instance `>surd(x,3);` yields $\sqrt[3]{x}$, then `>convert(%,power);` yields $x^{1/3}$. Conversely, `convert((9*x)^(1/3),surd)` gives $3^{2/3}\sqrt[3]{x}$.

For integers we have the usual operation `mod`.

Other interesting functions: `max` and `min` expect a set of real values, for instance `max(1.2,-3,sqrt(5))`.

One can use lots of other functions as well, see 1.2 Expressions.

Maple knows the following constants: `Pi` for π, `I` for the imaginary unit, `infinity` is ∞. For e use `exp(1)`. It also has Boolean constants `true` and `false`. For constants that we want to create ourselves it is best to use `macro`, say, `macro(E:=exp(1))`.

It can normally work with complex numbers, for instance `3+5*I`. For complex evaluation use `evalc(`*ex*`)`.

Angles: Maple knows both universal units, e.g. `30*degrees`. Conversion: `convert(30*degrees, radians)`.

It is possible to create vectors and matrices in Maple, but it has no standard tools to work with them. Thus it is better to wait for the package `linalg` (see 7.3), where vectors, matrices and operations with them are implemented.

1.2. Algebraic expressions

Maple can work with algebraic expressions (that is, expressions that feature letters and operations), for example `x^2+2*x+2`. Expressions can also feature functions (see below) and other operators (`int` etc.). One can also use Greek letters, they are inputed as words (`aplha` etc.) and shown as α etc.

Assignment to a variable: `>e1:=x^2+2*x+2;`.

To **evaluate** at a point: `eval(x^2+2*x+2,x=0)` or `eval(e1,x=0)` using the above defintion. This command substitutes the given value and also tries to simplify the outcome. It is also fairly intelligent, for instance if the given expression features the commands for differentiation, limit, integral etc., then this command knows that they should be first evaluated before the specified value is substituted in. Example: When constructing a tangent line to `f:=x^2` at 2, we need $f'(2)$ and `eval(diff(f,x),x=2)` does exactly the right thing.

One can also evaluate equations or more complicated objects this way, like `eval({`*eqn*`,`*ex*`,`*ex*`},x=...)` (here we evaluate a set). For instance, after `eval([x*y=5*z,{3*y,a*y}],y=2)` Maple returns $[2x = 5z, \{6, 2a\}]$.

One can also substitute expressions, commands `eval((x-1)^2+2*x,x=y^3+1)` and `eval(x*y+2,x=2)` do what one would expect, the former yields $y^6 + 2y^3 + 2$. We can also substitute more variables, `eval(x*y+2,{x=1,y=3})` or `eval(x*y+2,[x=1,y=3])`.

An example of handling previous calculations: The command `>sol:=solve({x^2+y^2=10,x+y=4});` gives the sequence $\{x = 3, y = 1\}, \{x = 1, y = 3\}$, but the order of individual parts is essentially random. For extracting a concrete variable value we therefore need something different, for instance `eval(x,sol[1])` returns 3. When we do `eval([x,y],sol[1])`, we get [3,1] (useful e.g. for plotting). One can also do do `eval(x^4+y^2,sol[2])` and get 10.

People often use another command for evaluation, namely `subs(x=0,x^2+2*x+2)` or `subs(x=0,e1)`. This also handles symbolic substitutions, if we do `>e2:=subs(x=2*t,e1);` we get a new expresion `e2` equal to $4t^2 + 4t + 2$.

However, note that `subs` is for symbolic substitution and not for evaluation, in particular it does not work on simplifying the answer. For instance, `eval(cos(x),x=0)` yields 1 while `subs(x=0,cos(x))` yields cos(0) (after `evalf` we get that 1, but as a floating point). Thus it is better to use `eval` for true evaluation. There is one more reason: Since `subs` does not really think about what we ask from it, so the command `subs(x=2,diff(cos(x),x))` may not work, on some versions of Maple it is intrpreted as `diff(cos(2),2)`.

On the other hand, it is more powerful in some ways. When substituting more values we can do it as with `eval` using set notation `subs({x=y^2,y=x^2},sin(x)+cos(y))` which does substitutions simultaneously and yields $\sin(y^4)+\cos(x^2)$. On the other hand, we can also just put the values one after another `subs(x=y^2,y=x^2,sin(x)+cos(y))`, the substitutions are then done successively and we obtain $\sin(x^4) + \cos(x^2)$.

One can also substitute for more complex expressions, but they have to fit well, for instance `subs(x+y=4,sin(x+y))` gives $\sin(4)$. For more creative substitutions we can try `algsubs`, for instance `algsubs(x+y=4,x+y+z)` gives $4+z$. Other examples: `algsubs(x^2=a,x^5)` gives a^2x, `algsubs(s/t=v,s/t^2)` gives $\frac{v}{t}$. Note that in the last example the given equality was used to eliminate one variable. We may specify which variable should be eliminated, `algsubs(s/t=v,s/t^2,[t])` gives $\frac{v^2}{s}$. We may suggest that only perfect fit be used using `algsubs(`*eqn*`,`*ex*`,exact)`.

An interesting type of expression is **piecewise functions**, where the value of expression depends on some condition. For instance, `expr:=piecewise(x<=-1,x,x<3,2*x,x-1)` defines $expr = \begin{cases} x, & x \le -1; \\ 2x, & -1 < x < 3; \\ x-1, & \text{otherwise.} \end{cases}$

That is, the specification is read as "if cond1 then exp1 or else if cond2 then exp2 or else exp3". One can put arbitrarily many pairs "condition-expression" and then one expression for remaining cases. The condition can be more complicated including logical operators.

One can plot expressions using `plot(x^2+2*sin(x)+2^x)` (see section 6).

Manipulation:

`simplify(`*ex*`)` simplifies the expression *ex* as much as possible, for instance using common denominator, cancelling etc. Sometimes further simplification is possible if we make some assumptions about symbols appearing there. To suggest such a step, use the option `symbolic`. For instance, `simplify(sqrt(1/x))` yields $\sqrt{\frac{1}{x}}$, whereas `simplify(sqrt(1/x),symbolic)` yields $\frac{1}{\sqrt{x}}$.

One can restrict the scope to only certain identities using `simplify(`*ex*`,`*option*`)`, for instance options `ln` for logarithmic identities, `exp` for exponential, also `power`, `trig`, `sqrt` or `radical` for identities with roots. One can supply more expressions by putting them in as a set, one can also specify as an optional argument equalities that can be used when simplifying. Note that one has to use the set notation even if just one condition is used. For instance, `simplify(x^2+xy,{x-y=1})` may give $x^2 + x(x-1)$. One never knows what variable Maple decides to take out, we can specify which variable(s) should be eliminated, so `simplify(x^2+xy,{x-y=1},{x})` yields $(y+1)^2 + (y+1)y$.

Of course, no-one really knows what the simples form is, opinions differ, so what Maple produces is not always what we need. Then we can try other commands.

`combine(`*ex*`)` attempts to simplify expression *ex* by combining terms. We may get better results if we restrict its scope, for instance `combine(`*ex*`,trig)` uses only trigonometric identities. We can also suggest other libraries, for instance `exp`, `ln`, or `power`.

There is a special command for expanding products and powers, `expand((x+2)*(x-1))` yields `x^2+x-2`. We can focus Maple's attention to just some identities for expanding, for instance `expand(`*ex*`,trig)` expands using trigonometric identities.

If we are not happy with the order of expanded expression, we can use `sort` or `collect` which collect coefficients of like powers. The difference is that `collect` also factors out common powers of the main variable. If there are more variables present, we can specify the one we feel is important as in `collect(`*ex*`,x)`, we can also supply more variables `collect(`*ex*`,[a,x])`. Also other expressions may be tried, for instance `collect(`*ex*`,sin(x))` but it works less well. Special trick: `collect(`*ex*`,diff)` collects derivatives of various orders.

We can also ask for the opposite direction, `factor(x^2+x-2)` yields `(x+2)*(x-1)`. This does not work perfectly and fails on some relatively simple factorizations.

Last thing about polynomials: we can find degree using `degree(`*ex*`)` or `degree(`*ex*`,x)`.

There are interesting commands for work with fractions. Generally, `numer(`*ex*`)` extracts numerator from the expression *ex*, `denom(`*ex*`)` returns denominator. For rational functions (ratios of polynomials) we have more.

`quo(x^2+2,x-1,x)` divides the first polynomial by the second (long division, equivalently, divides the second into the first) and shows just the result, not the remainder (here the output is `x+1`).

`rem(x^2+2,x-1,x)` divides the first polynomial by the second and shows the remainder (here the output is `3`).

`normal(`*ex*`)` writes the given expression as one rational function, for instance `normal(1/x+1/(x+1))` gives $\frac{x^2+x+1}{x(x+1)}$. If we want to get $\frac{x^2+x+1}{x^2+x}$, we use `normal(1/x+1/(x+1),expanded)`.

`convert(x/(x^3-1),parcfrac,x)` converts into partial fractions. The command `convert` is actually much more powerful. One can create polynomials with `convert(`*ex*`,polynom)`, command `convert(`*ex*`,exp)` tries to write the given expression using exponentials, `convert(`*ex*`,confrac)` tries to express the given expression as a continued fraction.

`convert(`*pol*`,horner,x)` rewrites the given polynomial into the Horner scheme form. This command also allows for changing between different forms of writing expressions defined by cases. In one direction: `convert(`*ex*`,Heaviside)`, in the other direction: `convert(`*ex*`,piecewise)` or `convert(`*ex*`,piecewise,x)`, for instance `convert(abs(x)*sign(x-1),piecewise)`.

A very powerful command is `select` that can choose certain terms in linear combinations of terms. There are several possibilities of specifying how to choose, for instance `select(has,`*ex*`,sin(x))` chooses all terms with $\sin(x)$, one can also use Boolean functions as in `select(p->degree(p,x)>4,`*ex*`)`. For other uses of `select` see 1.4. Sequences.

Interesting command `match(`*ex1*`=`*ex2*`,x,'var')` compares the two expressions and checks whether there are values of other variables than `x` that would make the two expressions equal. The outcome is *true* or *false*, in case of *true* also the values are given and this all is stored in the given variable, it must be put in apostrophes.

Example: `match(x^2-1=(x-a)*(x-b),x,'sol')` gives *true* and $sol = \{a = 1, b = -1\}$.

Why do we put the name of variable in quotes? If that variable has never been used before, then this would make no difference, But if it had some value assigned to it, then the value would get replaced in the expression, for instance if we did `>sol:=3;` before, then the above command would be interpreted as `>match(x^2-1=(x-a)*(x-b),x,3);`. By quoting we prevent this, we also make it possible to re-run this command.

Note: There is no meaning, hierarchy or relation assigned by Maple to various letters in expressions, they are all of equal rank to it. After defining `ex:=a*x+b` one can do `int(ex,x)` or `int(ex,a)` or `int(ex,c)`.

Expressions versus functions:

Maple distinguishes between expressions and functions (see 1.3.) In most cases it actually expects expressions (integrals, derivatives, equation solving etc.) To get an expression out of a function `f` (of `x`), write `f(x)`.

One can create more complicated expressions from simple ones using operators like sum, limit, integral etc. Note that some expressions that are created in this way are **inert**, that is, they are just a representation of some notion (expressions featuring `Sum`, `Int`, etc.). For instance, while `int(2x+a,x)` creates the expression $x^2 + ax$, `Int(2x+a,x)` creates the abstract expression $\int 2x + a\,dx$. Inert expressions can be changed into the evaluated form using `value(`*ex*`)`.

This is related to another concept. Usually when we give some expression to Maple, it is evaluated right away. Sometimes we want it to be taken as is, to be evaluated later. Then we can use single forward quotes ' to delay evaluation of an expression, so if we do `'int(`*ex*`,x)'`, it is as if we did `Int(`*ex*`,x)`. We will see examples below.

1.3. Functions

Functions are mappings in one or more variables, to variables they assign expressions.

Example: `x->x^2+2*x+2`, but also `u->u^2+2*u+2` or perhaps `u->u^2+a*u+2`.

More variables: `(x,y)->(x+y,a*x-y,b*x*y^2)` defines a function from $\mathbb{R}^2$ into $\mathbb{R}^3$ (and with two parameters).

In a typical case the function is defined by some expression *ex*, that is, by a formula that features operations, standard functions etc. Functions can be combined, for instance if we define `f1` and `f2`, then we can write things like `(f1+f2)(x)` or `(f1+2*f2)(z)`. There is also a special operator for composition, `(f1@f2)(x)` stands for $f1(f2)$. Trick: composition of f with itself n-times is denoted `(f@@`*n*`)(x)`.

Typically we assign functions to variables (see below), for instance `f:=x->x^2+2*x+2`. Note that then by typing `>f;` we just get f. To see the formula, use `eval(f)`.

To evaluate such a function at a point: `f(-1/3)`, as usual Maple prefers symbolic evaluation. If we define `f:=(x,y)->1-sin(x^2+y^2)`, then we should use `evalf(f(1,2))` to get a number, since `f(1,2)` gives $1 - sin(5)$.

One can also plug in symbolic values like `f(0,2*a)`, then the outcome is a symbolic expression.

This is the standard way for changing a function into an expression, simply use `f(x)`, `f(x,y)` etc. For instance, if we define `>f:=u->2*u-1;`, then `f(u)` yields an expression `2*u-1` and `f(x)` yields an expression `2*x-1` that can be used just like any other expression. With the definition `f:=(x,y)->1-sin(x^2+y^2)` we can do `simplify(f(a^2-b^2,b^2-a^2))`.

Going the other way, transformation of an expression *ex* (featuring variable `x`) into a function can be done in three ways.

– Usually we do it by hand `f:=x->`*ex*. Note that in this way the expression is only stored, not immediately evaluated. This is important, sometimes it is a great advantage, sometimes it troubles us. For instance, in order to simplify work with derivative one may be tempted to use `fd:=x->diff(f(x),x)`, but this would not work. Since the differentiation is not evaluated at the moment of definition, but at the moment we use it, we run in trouble when we substitute. When we type `>f(2);`, the outcome is `diff(f(2),2)`, which is a nonsense. We will see how to do it properly right away.

– One can change an expression into a proper function using `f:=unapply(`*ex*`,x)`, now the expression is evaluated at the moment of creating the function. It has some advantages in more complicated situations, the derivative from previous paragraph can be done using `fd:=unapply(diff(f(x),x),x)` and now it works.

– One can also use the general form of procedure `f:=proc(x)` *ex* `end`. This is the most general way, since it can actually involve complicated algorithms.

For instance, for **piecewise functions** we can use the approach with expression given by cases, as in the definition `f:=x->piecewise(x<=-1,x,x<3,2*x,x-1)` that creates the function $x \mapsto \begin{cases} x, & x \le -1; \\ 2x, & -1 < x < 3; \\ x-1, & \text{otherwise.} \end{cases}$ Using procedural approach it can be defined `f:=proc(x) if x<=-1 then x elif x<3 then 2*x else x-1 fi: end`

Another example defines the periodic extension of some `f` defined on $[0, 3]$:

```
g:=proc(x) local a,b; a:=trunc(x) mod 3; b:=frac(x); f(a+b); end
```

Finally, an example of a recursive function, namely the Fibonacci sequence:

```
f:=proc(n::nonnegint) if n=0 then 0 elif n=1 then 1 else f(n-1)+f(n-2) fi; end:
```

Procedures can do more than just define functions and we look at them closer in the section 8. on programming.

Maple has lots of predefined functions, for instance these:
`abs`, `sign`, `sqrt`, `exp`, `ln`, `log10` (this one can only work numerically, not really good for symbolic calculations); the function `log` takes an optional argument specifying base of logarithm `log[`*b*`](`*ex*`)`, when not specified it is taken for natural logarithm.
`sin`, `cos`, `tan`, `cot`, `sec`, `csc`
`arcsin`, `arccos`, `arctan`, `arccot`, `arcsec`, `arccsc`
`sinh`, `cosh`, `tanh`, `coth`, `sech`, `csch`
`arcsinh`, `arccosh`, `arctanh`, `arccoth`, `arcsech`, `arccsch`
An interesting function is `arctan(`x`,`y`)` that calculates $\arctan\left(\frac{y}{x}\right)$, but it also takes into account signs of x, y and provides angle from the range $-\pi..\pi$.
Commands for rounding: `round`, `floor`, `ceil`, `trunc` (the integer part of a number), `frac` (the part after the decimal period).

There are many more functions, like `Heaviside`, `Dirac`, `GAMMA`, `Beta`, `Zeta`, `BesselJ`, symbolic names for outcomes of important integrals etc.

We can use `binom(`m`,`n`)` also when m, n are not integers.

An interesting function is `rand` providing a random integer number from the range 0..1024. Funny thing: It must be used with an empty argument, for instance `a:=rand()`. We can also supply another argument of our own, `rand(`n`)` produces a random number from the range $0..x-1$, while `rand(`a`..`b`)` from the range $a..b$ inclusive. We still have to use the empty argument, `a:=rand(0..100)()/100`.

One can plot functions using `plot(f)`, see chapter 6. on plots.

1.4. Sets, lists, sequences

We start with a **sequence**, which is expressions separated by commas. In a sequence, order matters and terms may be repeated. Many commands show their output as a sequence, for instance `solve`. The main disadvantage is that sequences cannot be used as building blocks for other structures.

For that we have **lists**, which are essentially sequences enclosed by square braces, for instance `[3,-1,sqrt(12)]`. Lists are used in cases when we need to supply ordered information.

The last type is **set**, for instance {`3,-1,sqrt(2)`}. In sets duplicates are removed and the order does not matter, in fact Maple would rearrange sets on its own so one cannot rely on order of elements. For sets we have operations `union`, as in {`1,4,2`}`union`{`2,-1`}, also `intersect` and `minus`.

Lists and sets can be nested.

All three structures can be easily defined, `a:=4,2,exp(2)` makes a sequence, `a:=[4,2,exp(2)]` makes a list and `a:=`{`4,2,exp(2)`} makes a set. However, many such structures are created according to some pattern and there is a special way for that. The simplest is the operator `@` for repeating symbols, *ex*`@`n will make a sequence by repeating *ex* n-times. For instance, `a:=x@3` yields $a = x, x, x$.

The command `seq(`*ex*`,k=`a`..`b`)` creates a sequence with entries given by an expression *ex*.
Example: `seq([i,i^2],i=-1..1)` creates the sequence $[-1, 1], [0, 0], [1, 1], [2, 4]$. If we want this to be a list, we enclose it in brackets, `[seq([x],i=-1..1)]` makes $[x, x, x]$. Note that a and b need not be integers, then i is put equal to a, $a + 1$, ... up to the largest $a + k$ such that $k \in \mathbb{N}$ and $a + k \leq b$. If we need the step to be different than 1, we have to use some trick, see next.

The range can also be specified using a sequence or a list. For instance, `seq(i^2,i=1,-2,4)` produces $1, 4, 16$. Trick: `seq(i^2,i=seq(j*0.5,j=2..5))` will produce $1^2, (1.5)^2, 2^2, (2.5)^2$, we have a step by 0.5.

Manipulation: We can access terms of sequences, lists and sets by their position, the k-th term in variable *ex* is obtained using *ex*`[`k`]`. For instance, if we denote `l:=[a,b,c,d,e]`, then `l[4]` yields d. If we specify a range, we get a subsequence, we get the list $[b, c, d]$ using `l[2..4]`.

Nested (embedded) situatons are handled by chains of indices. After defining
`nest:=[4,[{"text",4*7,{6,[8,exp(x)],x^2},Pi}],sin(x)]` the expression `next[2][1][3][2][2]` yields `exp(x)`. Or not, there are sets involved and the order of elements may change. To choose elements from sets one should use different criteria than position, for instance values (`max` etc.).

We can also draw terms out of a list using `select`. There are three possibilities. One is to choose certain terms using `select(has,`*lst*`,`*ex*`)`, where *ex* is a set of values. It is also possible to choose by type using `select(type,`*ex*`,`*type*`)`, where the type specification can be `odd`, `even`, `integer`, etc., see section 8. Finally, one can select using a Boolean function, for instance `select(x->x>10,`*lst*`)`.

We can check whether some expression is a member of a list/set/sequence using `member(`*ex*`,`*lst*`)`, it answers *true* or *false*. If we supply a name of a variable as another argument (it is best to put it in quotes in case it was used previously, see Expressions above), then the position of the expression in list is given. Example: `>member(`*ex*`,`*lst*`,'pos');`.

The numer of elements in a list/sequence can be obtained using `nops(`*lst*`)`.

We can transform sequences into lists easily by `[`*seq*`]`. To transform a list into a sequence we need to use `op(`*lst*`)`.

Appending to sequences is very simple, `seq:=3,a,17` and then `seq2:=sqrt(2),seq,-1,2*x` yields $\sqrt{2}, 3, a, 17, -1, 2x$. If we want to append lists, we have to transform them into sequences for that and then transform back into a list: With `list:=[3,a]` we can do `list2:=[-1,13,op(list)]`.

Conversely, changing terms in a list is simple using `subsop(k=ex,lst)`, which in list *lst* replaces the term on position k with expression *ex*. This can then be used to also change sequences by a double transformation. For instance, after `seq:=2,4,6,8,10` we can do `seq2:=op(subsop(3=13,[seq]))` and obtain $seq2 = 2, 4, 13, 8, 10$.

Operations:

The usual algebraic operations are performed termwise. With functions we have to be more sophisticated. For instance, `sqrt[1,2,3]` yields $\sqrt{[1,2,3]}$, which is usually not what we want. For termwise action we use the function `map` that makes it possible to apply a certain function to all terms of a sequence/list/set. Using `map(sqrt,[1,2,3])` we get $[\sqrt{1}, \sqrt{2}, \sqrt{3}]$. This command does not work with embedded lists.

One can also try more functions. For instance, `map([sin,cos],[1,2])` yields $[[\sin(1), \cos(1)], [\sin(2), \cos(2)]]$.

Another useful function is `zip` that allows merging of two lists into one, we can specify the manner in which it is supposed to happen. It can be used for instance to create a table of values of a certain function given by expression *f*.

```
> datax:=[seq(n*Pi/8,n=0..24)];
> datay:=[evalf(seq(eval(f,x=n),n=datax))];
> values:=zip((x,y)->[x,y],datax,datay);
```

Then `values` is the list $[[0, f(0)], [\pi/8, f(\pi/8)], \dots]$.

One can use also other functions, for instance if $d1 = [a, b, c]$ and $d2 = [1, 2, 3]$, then `zip((x,y)->x+2*y,d1,d2)` yields the list $[a + 2, b + 4, c + 6]$.

*1.5. Tables and arrays

While we can include lists in lists, the resulting object does not have any higher structure. We can make it so when we order data into an array. Then the data gets ordered into matrices. For instance, `A:=array([[a,b,c],[d,e,f]])` creates the matrix $\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$. Its terms are now accessed using two-dimensional indices, for instance `> A[2,3];` gives f. We can also have more dimensions, an example of a three-dimensional object is
`> array([[[a,b,c],[A,B,C]],[[alpha,beta,gamma],[1,2,3]]]);`.

By default, terms are indexed 1,2, ... We can change it, for instance `> A:=array(3..4,7..9,[[a,b,c],[d,e,f]]);`, then d has index [4,7]. It is actually a good idea to help Maple this way.

A vector: `> array(1..3,[14,sqrt(2),a]);`. A column vector: `> array(1..3,1..1,[[14],[sqrt(2)],[a]]);`.

To see the contents of an array one has to use `> eval(A);`. One- and two-dimensional arrays with indices started with 1 are displayed as vectors or matrices, more dimensional arrays are shown by listing terms in the form that we show below for tables.

It is easy to change individual terms, for instance if we do `> A[2,2]:=sin(x);` with the first matrix in this section, then the matrix becomes $\begin{bmatrix} a & b & c \\ d & \sin(x) & f \end{bmatrix}$.

Array is also a good tool for creating matrices using a pattern. For instance, `> array(identity,1..4,1..4);`.

The index range can be preceded by an "index function" that helps in defining the array. For instance,
`> array(identity,2..4,2..4);` automatically creates an identity matrix as an array with rows and columns indexed 2 to 4. More-dimensional arrays possible, `> A:=array(identity,1..3,1..3,1..3);` has $A[1, 1, 1] = A[2, 2, 2] = A[3, 3, 3] = 1$ and all other entries are zero. Other possibiblities:

`> array(sparse,-1..1,-1..1);` creates an array whose all terms are 0. One can change some entries to other values, but Maple expects that there will be only few such entries and saves such an array in a memory-saving way. The option `diagonal` is obvious.

The option `symmetric` creates a symmetric matrix, the condition is that $A[i, j, k, \dots] = A[I, J, K, \dots]$ if $[I, J, K, \dots]$ is a permutation of $[i, j, k, \dots .]$

The option `antisymmetric` is more complicated, the rule is that $A[i, j, k, \dots] = A[I, J, K, \dots]$ if $[I, J, K, \dots]$ is an even permutation of $[i, j, k, \dots .]$ $A[i, j, k, \dots] = -A[I, J, K, \dots]$ for odd permutations and $A[i, j, k, \dots] = 0$ if at least two indices match.

Just like with lists etc, we can apply a function to all individual terms of an array using `map`. For instance, after `> A:=array([[a,b],[c,d]]);` we can do `> map(x->x^2,A);` and obtain $\begin{bmatrix} a^2 & b^2 \\ c^2 & d^2 \end{bmatrix}$.

Tables are a generalization of arrays allowing for arbitrary structures and arbitrary indices. Regular tables indexed $1..n$ can be defined as `> T:=table([x,y]);`. Evaluation: After `> eval(T);` we see

T=*table*([
(1) = x
(2) = y
])

More general tables are defined term by term. For instance, we can define `> tab[A]:=x+y: tab[B]:=sqrt(2);` and obtain a table whose contents is revealed after `> eval(tab);` as

tab=*table*([
(A) = $x + y$
(B) = $\sqrt{2}$
])

We put in few more terms: `>tab[1,2]:=anotherOne: tab[x^2+y^2]:=1;` and now we have table
tab=*table*([
$(A) = x + y$
$(B) = \sqrt{2}$
$(1, 2)$ =*anotherOne*
$(x^2 + y^2) = 1$
])
Note that the index `1,2` does not imply any structure like in arrays, it is simply a string marking one entry in this table. If we want a structure, we need to embedd those indices.
`>newtab[1][a]:=first: newtab[1][b]:=second: newtab[2]:=third;`, now after `>eval(newtab);` we get
tab=*table*([
(1) =*table*([
(*a*) =*first*
(*b*) =*second*
])
(2) =*third*
])
Alternative specification: `>newtab:=table([(1,a)=first,(1,b)=second,(2)=third]);`. One can use tricks, for instance if we do `>table(symmetric,[(1,1)=1,(1,2)=2,(2,2)=3]);` then the item $(2, 1) = 2$ is filled in automatically.
Again, accessing and changing entries is easy, `>tab(x^2+y^2);` yields 1, `>newtab[2]:=fourth;` changes the item. Note that packages are in fact tables, for instance if we let `>trigsquares=table([(sin2)=sin^2,(cos2)=cos^2]);` then calling `>trigsquares[sin2](x);` yields $\sin^2(x)$.

Tables and arrays are an exception when it comes to assigment. If `A` is an array or a table and we do `>B:=A;`, then it just creates a pointer, it does not copy the evaluated contents of A into B like for other structures. If we want a hard copy, we have to use `>B:=copy(A);` but only if the array/table is not of embedded kind.

Universal conversions between sequences, lists, sets, arrays and tables are possible, but embedded arrays/tables only work with type `listlist`.

Conversion **to a sequence**:	Conversion **to a list**:	Conversion **to a set**:
`op(`*list*`)`	`[`*seq*`]`	{*seq*}
`op(`*set*`)`	`[op(`*set*`)]`	{`op(`*list*`)`}
`op(convert(`*array/table*`,list))`	`convert(`*set/array/table*`,list)`	`convert(`*list/array/table*`,set)`
	`convert(`*array*`,listlist)`	
Conversion to an array:	**Conversion to a table**:	
`convert(`*list/set/table*`,array)`	`table([`*seq*`])`	
`convert([`*seq*`],array)`	{`table(`*list/set*`)`}	
	`table(convert(`*array*`,list))`	

Other ways are also possible. For instance, we have a matrix `>A:=array([[1,2,3],[4,5,6],[7,8,9]]);` and we want to find its maximal entry. For that we need to change it into a sequence, we can do `>data:=convert(A,listlist);` yielding $[[1, 2, 3], [4, 5, 6], [7, 8, 9]]$, then `>seq(op(data[i]),i=1..3);` makes $1, 2, 3, 4, \ldots, 9$ and now we can call `>max(%);`.

1.6. Relations, logic

Relations: `x=3`, `x<>3`, `x<3`, `x<=3`, `x>3`, `x>=3`. Logical operators `not`, `and`, `or`, `xor`, `implies`.
`evalb(`*ex*`)` evaluates a Boolean expression *ex*, returns *true* or *false*

1.7. Strings

Strings are sequences of symbols delimited by double quotes `"`. Example: `>print("Hello, world.");` Two successive double quotes are interpreted as one that is a part of the string, so `"mo""re"` is interpreted as *mo"re*. If there is a space (or line break) between successive quotes, they are ingored, so `"mo" "re"` becomes *more*.
Strings can be assigned: `>word:="swearing";`. Operations: Individual letters can be obtained using index as with lists etc., we can also use ranges, for instance `word[3..5]` yields *ear*. Negative numbers count from the end, so `word[3..-4]` also yields *ear*.
Strings can be concatenated, either using `||` ot `cat`, both `"h"||word[3..5]` and `cat("h",word[3..5])` return *hear*.
Finally, `length(`*sting*`)` returns its lenght.

2. Variables in Maple

Variable name can be any combination of letters, digits and underscores _ that begins with a letter. If we really want to use another string as a variable, we have to enclose it in single back quotes ‘ as in `‘Var one‘:=13`.
Variables are free or assigned. Any letter that is used in an expression is an unassigned variable and is preserved in manipulations and evaluations as a parameter unless either it is specified earlier, or it is given a special role by the operation we are performing (the working variable in integral etc.)

Assignment: `y:=`*x* defines `y` to be the same thing as *x*. This *x* can be anything—a number, an expression, a function, an equation, etc.
Example: `>y:=2*x+2;` then `solve(y=0)` is the same as `solve(2*x+2=2)`.
Note that when doing this assignment, `x` is immediately evaluated and the outcome is assigned to `y`. This is sometimes important. For instance, if we set `>x:=13;` and then `>y:=x;`, then we in fact say `>y:=13;`. Sometimes we do not want it, we want *y* to be "the same thing as *x*" so that the contents of *x* would be taken into account only when we actually use *y*. This can be achieved by delaying evaluation in the usual way, that is, by setting `>y:='x';`.
This may strongly influence how things work. For instance, if we do `>x:=3:y:=x:x:=5;`, then the command `>y;` shows $y = 3$. On the other hand, if we do `>x:=3:y:='x':x:=5;` then calling `>y;` shows $y = 5$. There is an exception to this rule, if a variable denoting an array/table is assigned to another, then just a pointer is created as if it were a delayed assignment. For a real copying of an array/table use `copy`, see section 1.5.

Alternative assignment: `>assign(y=x);`. This command also accepts a set of assignments, which makes it useful in some situations. For instance, if we solve `>solve({x+y=4,x-y=2},{x,y});` then the answer is the set $\{x = 3, y = 1\}$. If we want to save answers as variables, we simply do `>assign(%);`.
Some variables are global and should not be messed up with, they start with the underscore _. For instance, the differential equation solver uses `_C1`, `_C2` etc. as integration constants.

Note that a definition like `>y=2*x+2;` does not define a function—Maple does not see this `y` as a variable dependent on `x`. This `y` is simply a label assigned to that linear expression.
The same variable can be re-assigned, the old meaning is lost.
One can "unassign" a variable by assigning it to itself, for example `>k:='k';`, or using `unassign('k')`. This is very important in case we want to use this variable as a working variable. For instance `>int(sin(x),x);` makes no sense if we previously assign some meaning to `x`. Similarly, if after `>x:=4;` we issue command `>solve(x+1=2,x);` it will be interpreted as `solve(4+1=2,4)`, which is a nonsense. If we are not sure, we can always ask `assigned(`*var*`)`, the outcome is *true* or *false*. We can also unassign all variables by reseting using `restart`.

To see the meaning of a variable we use `eval(`*var*`)`. By default this recursively traces all assignments. For instance, if we define `>a:=b;b:=c+1;c:=3;` then `>eval(a);` yields 3. However, we can specify how deep the tracing should go, `>eval(a,1);` gives *b* and `>eval(a,2);` gives $c + 1$.

`about(var)` tells us what Maple knows about variable *var* (especially its type and value).

Free variables can be used in expressions as parameters. This sometimes complicates things for Maple, for instance if we ask it to do `>solve(x^2=a,x);`. We get more interesting answer if we first do `>assume(a>0);`. Using `assume` we can tell Maple what conditions can be taken for true, this command accepts a set of conditions.
Example: `>assume(n odd,a integer,r real,f continuous,b<=7);`. Note that every such command erases previous assumptions. If you want to add to existing assumptions, use `additionally`.
Variables that some assumptions are made about can be recognized, in output they have tilde ~ appended. We can ask whether some condition is or is not assumed. For instance, if we do `>assume(x>=0);` and then use `>is(x>1);is(x<1);`, the outcome will be *true, false*.

With variables we can sometimes use the operation called concatenation. For instance, `a.b.c` yields *abc*. However, we can define the meaning of the individual variables and concatenation creates a new name of variable: after `>a:=x:b:=1:c:=7;` the expression ta.b.c reads `a17`, since the basic concatenation rule says that the first variable is always taken literally. A little trick to get around it: `.a.b.c` yields `x17`.

3. Equation solving

We can access sides of a given equation *eqn*: `rhs(`*eqn*`)` returns its right-hand side as expression, analogously `lhs(`*eqn*`)`.
The command `solve(`*eqn*`)` tries to solve the equation *eqn* symbolically (that is, precisely). For instance, the command `>solve(x^2+2*x=2);` solves the given equation, in the answer there is `I` that stands for $\sqrt{-1}$. The command `solve` tries to solve any equation (not just with polynomials), but naturally it often fails.
One can also write `solve(`*ex*`)` for solving the equation $ex = 0$. In both situations we can tell Maple which variable to solve for using `solve(`*eqn*`,`*var*`)` or `solve(`*ex*`,`*var*`)`, this is especially important if there are more letters in the equation. For instance, `>solve(a*x+b,x);` or `>solve(a*x+b=0,x);` yield $x = -\frac{b}{a}$, while `>solve(a*x+b,b);` yields $b = -ax$.
Solve also handles situations with more equations, they should be supplied as a set.
Example: `>solve({x+y=2,x-y=0},{x,y});` yields the solution $\{x = 1, y = 1\}$. When there are more solutions, they are returned as a sequence. For instance, `>solve({x^2+y=2,x^2-y=0},{x,y});` yields $\{x = 1, y = 1\}, \{x = -1, y = 1\}$. Similarly `>solve(x^2=4);` yields $x = -2, x = 2$. Note that if there are more solutions, then the order is not predetermined, it depends on many factors and runing the same command at different times may yield different orders of solutions.
Solutions may feature constants, for instance $\_Z$ stands for an arbitrary integer. Using `>about(%);` one can learn more about them. It may also help to process the original solution using `>allvalues(%);`. Another possibility for encouraging more solutions is to set a flag `>_EnvAllSolutions:=true;` before solving.
Trick: If `>s:=solve(`*eqn*`);` yields more solutions and we want to substitute all of them into some expression, then we can use `seq(subs(i,`*ex*`),i=s)`.

One can also work with inequalities, e.g. `>solve({x+y>=5,x-y>1,y-x<=1/2},{x,y});` or `>solve(abs(x)<1,x);`.
One can even solve for functions if the equation is not too complicated, for instance `>solve(f(x)+f(x)^2=1+x^3,f);`.

`fsolve(`*eqn*`)` tries to solve *eqn* numerically, always returns a floating point number and (with a few exceptions) only one solution even if more are possible. Alternative: `fsolve(`*ex*`)` tries to solve the equation $ex = 0$. Because it works numerically, no parameters are possible, only one variable may appear in the expression. As usual, it is a good idea to help Maple by emphasising the variable, say, `>solve(cos(x)=x,x);`. It may also help to suggest a range where the solution should be found, say, `>solve(cos(x)=x,x,x=0..Pi/2);`.
If we want a complex solution, we do it `fsolve(`*eqn*`,x,complex)`.
Also `fsolve` can handle systems of equations. Example: `>fsolve({2*x+3*y=1,1-x*y=2},{x,y});`.
One has to be a bit careful when using `fsolve`. If we try `>s1:=fsolve(sinh(x)=cosh(x));`, then we get a solution $s1$. We can check, `>evalf(subs(x=s1,cosh(x)-sinh(x)));` indeed yields 0. However, if we increase precision with `>Digits:=20;` and then repeat the whole procedure, we again get a solution, but this time a different one, and again the check works. Another increase, another solution. In fact there is no solution. Reliability is a problem common to all numerical calculations.

Special command `rsolve` for solving recursive functions: `rsolve({f(n)=f(n-1)+f(n-2),f(0)=0,f(1)=1},f(n))`.
There is also a special command `isolve` for finding integer solutions (diophantine equations). We have to specify parameters to be used, for instance `>isolve(x+2*y+3*z=10,{k,l});` gives $\{y = k, z = l, x = 10 - 2k - 3l\}$.

4. Calculus

4.1. Working with functions

Inverse function: To find an inverse to a function $f(x)$ given by some expression *ex*, do `solve(y=`*ex*`,x)`. For instance, after `>f:=2*x-3;`, the command `>solve(y=f,x);` returns the answer $\frac{y+3}{2}$.
How to check? Define `>g:=(y+3)/2;`, do `>eval(g,y=f);` and `>simplify(%);`, also `>simplify(eval(f,x=g));`.
The same using functions: After `>f:=x->2*x-3;` the command `>solve(y=f(x),x);` returns the answer $\frac{y+3}{2}$. Create a function out of it: `>g:=unapply(%,y);`, then do `>simplify(g(f(x)));` and `>simplify(f(g(y)));` (or `>simplify(f(g(x)));`).

Limit: `limit(`*ex*`,x=`*a*`)` yields limit of expression *ex* at *a*.
Inert (unevaluated) form: `Limit`. Using `value(Limit)` we get the same as using `limit`.
`limit` always tries to evaluate symbolically. We can change the answer to a real number using `evalf`. On the other hand, if we use `evalf(Limit)`, Maple skips the symbolic evaluation step and uses numerical algorithms to (try to) evaluate the limit. It may be faster but also less reliable.
Symbolic evaluation is fairly good, for instance `>limit((sin(x+h)-sin(x))/h,h=0);` yields $\cos(x)$.
One-sided: `>limit(sin(x)/x,x=0,left);` or `>limit(e1,x=-infinity,right);`. Note that if we use a complex function, say `>limit(f(z),z=1+3*I);`, then Maple approaches that `z` in the direction of the real line from the right and from the left and uses real point of view, for instance it evaluates a sum of ∞ and $-\infty$ as *undefined*. If we want it to apply the complex point of view (lmit from all directions, complex infinity), we have to use `>limit(f(z),z=1+3*I,complex);`.
An interesting trick to identify vertical asymptotes of a function:

```
>s:=solve(f(x)=val,x);
>limit(s[1],val=infinity);
```

Maple also handles more variables, as in `>limit(x/(x+y),{x=0,y=0});`.

4.2. Derivative and extrema

`diff(`*ex*`,`*var*`)` differentiates the expression *ex* with respect to variable *var* (the output is an expression again). The expression may feature more variables, then we get derivative with parameter or partial derivative. Higher order: `diff(`*ex*`,x$3)` finds the 3rd derivative of the expression *ex* with respect to x. It is actually a shortcut for `diff(`*ex*`,x,x,x)`. Note: `diff(`*ex*`,x$0)` is not correct, this is important to know when we want to use `diff(`*ex*`,x$k)`.
Inert (unevaluated) form: `Diff`. We evaluate it using `value` as usual, so `value(Diff( ... ))` is the same as `diff(`*dots*`)`.
We can tell Maple that some constant is a function like this: `y(x)`, it then reacts accordingly, for instance, compare: `>diff(x*a,x);` yields a, while `>diff(x*a(x),x);` yields $a(x)+x\frac{\mathrm{d}}{\mathrm{d}x}a(x)$. One can tell globally using `alias(a=a(x))`.
If Maple cannot evaluate, it stores symbolic meaning. For instance, if we set `>dr:=diff(f(x),x$2);`, Maple remembers $dr = \frac{\mathrm{d}^2}{\mathrm{d}x^2}f(x)$. Then we can do `>subs(f=sin(x),dr);` and get $\frac{\mathrm{d}^2}{\mathrm{d}x^2}\sin(x)$, applying `>value(%);` to this innert form we get $-\sin(x)$.

Note: `diff(`*eqn*`,`*var*`)` differentiates both sides of equation *eqn*. Example: `>diff(x^2+x=7,x);` yields $2x + 1 = 0$.

One can also differentiate functions with `D(`*fun*`)`, where *fun* is a function of one variable. For instance, the outcome of `>D(sin);` is cos. One can use variable to get an expression as the outcome, even a different one, `>D(sin)(alpha);` yields $\cos(\alpha)$. Higher derivative: `D(D(`*fun*`))` is the second derivative, in this way it is hard to do higher orders, one can use `(D@@k)(`*fun*`)`.
This operation can differentiate function and also procedures, even those including conditions and loops.

More variables: `D[`i`](f)` is derivative with respect to the i-th variable, they can be repeated, but instead of chains like `D[1][1][1][2][3][3](f)` it is better to do `D[1$3,2,3$2](f)`, in the innert case Maple writes it as $(D_{1,1,1,2,3,3})f$.

Conversion: After `>diff(f(x,y),x,y$2);` we get $\frac{\partial^3}{\partial x \partial y^2} f(x,y)$. Calling `>convert(%,D);` we get $(D_{1,2,2})(f)(x,y)$, then after `>convert(%,diff);` we are back to $\frac{\partial^3}{\partial x \partial y^2} f(x,y)$.

Implicit derivatives: see section 4.5 Implicit functions.

Differential calculus for more variables: see the linear algebra package (section 7.3) and `plots` package (7.1).

For finding global extrema of an expression over the given set we have `minimize(`*ex*`,`*range*`)`, it may also give $-\infty$ as the answer. For instance `>minimize(sin(x),x=-1..1);` gives $\sin(-1)$ and `>minimize(1/x,x=-1..1);` gives $-\infty$. One can leave out the range, then it looks for global extrema with respect to domain. Similarly works `maximize`, which is actually implemented as $-\min(-f)$ in Maple, so it behaves in exactly the same way.

These two commands also accept expressions of more variables, `>minimize(x*y-x,x=-1..1,y=-2..3);`. Trick: `>minimize(x*y-x,x=-1..1);` will find the minimum dependent on the parameter y.

Note that these commands work symbolically, so if Maple cannot solve the equation $ex' = 0$, then the procedure fails.

If we use the option `location=true`, then Maple returns first the desired minimum/maximum and then a set of points where the extreme happens. For instance, `>maximize(sin(x),x=-1..3.5,location=true);` would yield $1, \{[\{x = 0\}, 1], [\{x = Pi\}, 1]\}$. Similarly `>minimize(x^4-2*x^2+y^2,location=true);` yields $-1, \{[\{x = -1, y = 0\}, -1], [\{y = 0, x = 1\}, -1]\}$. Note the order of variables in the outcome, it is unpredictable and one cannot rely on it. Another example: `>maximize(x^2,location=true);` returns $\infty, \{[\{x = \infty\}, \infty]\}$.

Note that since the procedure depends on symbolical solutions of $ex' = 0$, it may miss some points where minima/maxima are attained.

Note that both `minimize` and `maximize` are redefined upon loading the `simplex` package, then one can do for instance `>maximize(2*x-3*y+7*z,x-y-z<=1,3*x+y+2*z>=-1);`.

Symbolical work also limits the usefulness of the command `extrema` that finds (candidates for) global extrema. In its simplest form `extrema(`*ex*`,{},`*var*`)`, where *ex* is an expression featuring the variable *var*, it finds stationary points (if any) of *ex* and returns their values. Therefore it does not return $\pm\infty$ for unbounded cases, it also does not sort the values for us and/or make any conclusions. If we use `extrema(`*ex*`,{},`*var*`,'`*var2*`')`, then apart from showing the value(s) of potential extrema, the command also stores locations of those candidates into the specified variable *var2*.

For instance, `>minimize(x^4-2*x^2+y^2,{},{x,y},'pts');` returns $\{-1, 0\}$ and after `>pts;` we see $\{\{y = 0, x = 0\}, \{y = 0, x = 1\}, \{x = -1, y = 0\}, \}$. Note that the order of points does not correspond to the values returned by the command. It is up to us to deduce that -1 is minimum and 0 is value at the saddle point $[0, 0]$.

The empty set stands for a condition. Using `>extrema(x*y-x,x+y=1,{x,y});` we minimize the function $xy - x$ with the constraint $x + y = 1$. More constraints: `>extrema(x*y-3*z,{x+y+z=1,x^2+y^2=4},{x,y,z});`. Also `extrema` can work with parameters, calling `extrema(x*y-3*z,{x+y+z=1,x^2+y^2=4},{x,y})` means that x, y are variables and z becomes a parameter.

4.3. Integral

`int(`*ex*`,`*var*`)` evaluates the indefinite integral of the expression *ex* with respect to variable *var* and makes it into an expression—assuming that it can do it. Otherwise it returns the integral as a symbolic representation, just like if we used `Int`. `Int(`*ex*`,`*var*`)` creates an expression that represents the indefinite integral of the expression *ex* with respect to *var*. As usual, we can evaluate `Int` with `value` to get `int`.

`int(`*ex*`,`*var*`=`*a*`..`*b*`)` evaluates the definite integral of the expression *ex* from *a* to *b* (limits can also be `infinity`). It evaluates symbolically, that is, it tries to give the answer in a closed form. If it can actually evaluate the integral, it will do so and express it as some formula (which often has to be further simplified with `simplify`), otherwise it returns symbolic expression just like `Int` does, the inert form `>Int(`*ex*`,`*var*`=`*a*`..`*b*`);` creates an expression that represents the definite integral.

Maple can integrate with a parameter, `>int(a*x,x=0..2);` returns $2a$. Repeated integrals work as expected, for instance `int(int(x*y^2,y=0..x^2),x=0..1)`.

To evaluate the result as a floating point (if there are no parameters), use `evalf`. Note the following difference: If we use `evalf(int)`, then the symbolic answer is found and then evaluated. If we use `evalf(Int)`, then the whole integration is done from scratch using a suitable numerical method. One can specify the method as an optional argument of `evalf`, possibilities are `CCquad` for the Clenshaw-Curtis method (default), `NCrule` for the Newton-Cote method and `Dexp` for the double exponential method.

For more about numerical approximation see package `student` (section 7.4).

Recall that if we want to integrate a function `f`, we can turn it into an expression by putting `f(x)` into `int`.

Integration works quite well, `int` can handle absolute value `abs(x)` and also discontinuities, in particular expressions featuring `Heaviside(x-a)` or `Dirac`.

Some integration can be only done for specific values of parameters, then we can help Maple using `assume`. For instance, after `>assume(a<b);` we can do `>func:=x->1/(a+b*cos(x));` and then `>int(func(x),x);`. We check correctness of this answer by `>simplify(func(x)-diff(%,x));`.

Improper integrals work as expected. `> int(1/x^4,x=1..infinity);` yields $\frac{1}{3}$, but `> int(1/x^4,x=-1..1);` yields ∞ and `> int(1/x^3,x=-1..1);` gives *undefined*. We can use `> int(1/x^3,x=-1..1,CauchyPrincipalValue);`.

Some improper integrals are better approached using limit, say `> evalf(Limit(Int(1/x^2,x=a..infinity),a=0));`.

Measuring time: `> start:=time();` then `> int((sin(x))^2000,x);` and finally `> (time()-start)*sec;`

*4.4. Sums and Series

`Sum(`*ex*`,`*var*`=`*a*`..`*b*`)` creates an inert expression that sums up expressions *ex* for variable *var* ranging from *a* to *b*. `> sum(`*ex*`,`*var*`=`*a*`..`*b*`);` tries to evaluate this sum symbolically and express the answer in closed form. If it can't, it gives the same expression as with `> Sum;`. Trick: `> Sum(k,k=1..n)=sum(k,k=1..n);` gives $\sum_{k=1}^{n} k = \frac{1}{2}n^2 + \frac{1}{2}n$. To get a nicer answer we can use for instance `factor`.

As usual, we can use `value(Sum)` as an equivalent of `sum`. If we instead apply `evalf` to `Sum`, Maple tries to evaluate the sum numerically. This is faster than `evalf(sum)`, where Maple first tries to evaluate symbolically and then express the answer as a number, only in case of failure it switches to numerical work. However, results using intermediate symbolical step may be more reliable.

Nice trick: `> sp:=a->sort(factor(sum(k^a,k=1..n)));` and then `> sp(2);` gives $\frac{1}{6}n(n+1)(2n+1)$.

We can use `infinity` as a limit. Maple tries to do sums like `sum(`*ex*`,k=1..infinity)` the best it can, but sometimes it stumbles. In some cases it is better to go through partial sums, first, `sN:=n->Sum(1/k^2,k=1..n)`, then `limit(sN(n),n=infinity)`. For most series Maple cannot work algebraically, so `evalf(limit(sN(n),n=infinity))` is a good idea.

One can even try `f:=x->sum(x^k,k=0..infinity)` with partial sums `fN:=(x,n)->sum(x^k,k=0..n)`.

If we have variables `x1`, `x2`, `x3`, `x4`, `x5` and we want to add them, we can do `sum('x.i',i=1..5)`. Note how the evaluation was delayed for `i` to take its values.

If we just want to add a finite number of concrete values, it is faster to use `add`, for instance `add(1/k^2,k=1..1000)`.

Products are handled similarly, for instance `> evalf(product(1+1/k^2,k=1..infinity));`. For multiplication of numbers we also have `> mul(1+sqrt(k),k=[1,2,4,5,7,8]);`.

Taylor polynomial:

`taylor(`*ex*`,x=`*a*`,`*n*`)` creates Taylor polynomial of degree $n-1$ with center a for expression *ex* and outputs it as an expression that includes the term $O(x^n)$. You get the polynomial itself using `> convert(%,polynom);`.

Trick: `> T:=n->convert(taylor(sin(x)/x,x=Pi,n+1),polynom);` creates an expression-valued function, now, say, `> T(5);` yields an expression that is the 5th degree Taylor plynomial of $\sin(x)/x$ with center π and we can use it, for instance evaluate it: `> eval(T(5),x=0.7);`.

We can also create Taylor polynomials ourselves, a general procedure for an expression `fe` and center stored in variable `a` can be done for example like this: `T:=n->eval(fe,x=a)+sum(eval(diff(fe,x$k),x=a),k=1..n)`.

Another possibility is to use the command `series`. It finds a power series expansion, if needed it finds a Laurent series, it also accepts complex expressions and complex center. Its outcome is a special type called `series`, but in fact it is a finite truncation of a series, the order is given by the global variable `Order`. To extract something useful one can convert it into a polynomial, but it will not always work, since the resulting series can have also powers with exponents that are not natural numbers. Trick: remove the O part using `> f:=%-select(has,%,O);`

Multi-variable Taylor polynomial: for instance `> mtaylor(sin(x*y),[x=Pi,y=0],3);`.

Fourier series

```
> fe:=proc(f) fnormal(eval(f)); end:;
> a:=proc(f,T,n) if n=0 then fe((1/T)*Int(f,x=0..T));
    else fe((2/T)*Int(f*cos(n*2*Pi*x/T),x=0..T));
    fi; end:
> b:=proc(f,T,n) if n=0 then 0;
    else fe((2/T)*Int(f*sin(n*2*Pi*x/T),x=0..T));
    fi; end:
> FS:=(f,T,n)->sum('a(f,T,k)*cos(k*2*Pi*x/T)+b(f,T,k)*sin(k*2*Pi*x/T)',k=0..n);
```

Then `> fs(exp(x),1,10);` yields T_{10} for $f(x) = e^x$ on $[0,1]$.

Perhaps less elegant, but in some sense simpler alternative:

```
> a0:=(f,T)->(1/T)*Int(f(x),x=0..T);
> a:=(f,T,k)->(2/T)*int(f(x)*cos(k*2*Pi*x/T),x=0..T);
> b:=(f,T,k)->(2/T)*int(f(x)*sin(k*2*Pi*x/T),x=0..T);
> FS:=(f,T,n)->a0(f,T)+sum(a(f,T,k)*cos(k*2*Pi*x/T)+b(f,T,k)*sin(k*2*Pi*x/T),k=1..n);
```

*4.5. Implicit functions

First we set up an implicit equation, `> if:=x*y^3+y=0;`. This can be sometimes solved for y using `solve(if,y)`, but often one does not get anything useful this way.

Implicit differentiation by hand: `diff(if,x)` would differentiate both sides, but on the left it would simply do a partial derivative, not what we want. The correct procedure: First Maple needs to be told that `y` depends on `x`, by typing directly `y(x)` instead of `y` in the equation or, if we already have typed it, using `subs(y=y(x),if)`. Now we can differentiate, `ifder:=diff(%,x)` differentiates both sides the way we want it and the result (an equation) features derivative of `y(x)`. To find it, give it a name `yder:=diff(y(x),x)` and solve for it `solve(ifder,yder)` (or do directly `solve(ifder,diff(y(x),x))`).

There is also a standard procedure for this in Maple. Given an equation *eqn* with x and y, we do `implicitdiff(`*eqn*`,y,x)`. This is actually a shortcut, the full notation is `implicitdiff({`*eqn*`},{y(x)},{y},x)`. Higher derivative: use for instance `x$3`. The answers are sometimes nicer if we use the special option `Diff`.
If *eqn* features x, y, z and we want to consider $z = z(x, y)$, we can find its derivatives using `implicitdiff(`*eqn*`,z,x)` or `implicitdiff({`*eqn*`},{z(x,y)},{z},x)`, again we can specify other derivatives using for instance `x,y$2`.
For plotting see section 7.1. Advanced plotting.

*5. Differential equations

For convenience, give an equation some name. Example: `>ODE:=diff(y(x),x)/(x+y(x))=exp(x);` Note: We used `y(x)` instead of just `y` to tell Maple to treat `y` as a dependent variable, depending on `x`.
`dsolve(ODE,y(x))` solves the above equation for $y(x)$. Often the output is in implicit form, this can be sometimes helped by `solve(%,y(x))`.
To force explicit form (if possible): `dsolve(ODE,y(x),explicit)` but then this sometimes does not return any solution at all, if the only form it can get is the implicit one.
Explicit solution appears as an equality *y(x)*= ... . To extract it as an expression one can use `rhs(%)`.
If we specify the option `output=basis`, then the answer is given in the form `[[`*fund. system*`],`y_p`]`.
Initial conditions: Example: `>IC:=y(0)=1;` then use `dsolve({ODE,IC},y(x))`
Higher order possible, using parameters also possible.
Example: `>ysol:=dsolve({diff(y(x),x$2)+y(x)=x*cos(x),y(0)=y0,D(y)(0)=v0},y(x));`
To get solution for particular values one can do e.g. `subs({y0=5,v0=-1},ysol)`.

Trick for solving for initial condition later. Assume that `ysol` is a solution obtained using `dsolve` and that it has one integration constant `_C1` in it. For initial condition $y(1) = y0$ we get the right constant like this:
`>subs(x=1,y(1)=y0,ysol): solve(%,_C1): subs(_C1=%,ysol);`
Specific methods of solution can be specified in `dsolve` using options:
`type=series`: Maple expresses the solution using series. By default it uses order 6, `Order:=`n changes the order globally.
`type=Laplace`: Maple uses Laplace transform. It is necessary when `Heaviside` or `Dirac` are used.
`type=numeric`: Maple solves numerically, it works only if an appropriate number of initial conditions is specified.

Systems of equations possible, for equations featuring $x(t), y(t)$ either using `dsolve({`*eq1*`,`*eq2*`},{x(t),y(t)})` or also with initial conditions `sol:=dsolve({`*eq1*`,`*eq2*`,x(0)= ... ,y(1)= ... },{x(t),y(t)})`. How to extract formulas from the solution $sol = \{x(t) = \dots, y(t) = \dots\}$? For instance using `xsol=subs(sol,x(t))` or `eval(x(t),sol)`.

Many tools for solving and visualizing can be found in package DEtools that we call `with(detools)`. For instance, it has commands `dfieldplot` and `phaseportrait`.
See also section 7.1. Advanced plotting.

Solving differential equations using transformations: using `with(inttrans)`.
`fourier(fe,t,w)` takes an expression `fe` with variable `t` and returns its Fourier transform with variable `w`. To get back: `invfourier(ge,w,t)`.
Similarly we have `laplace(fe,t,p)` and `laplace(fe,p,t)` for the Laplace transform.
The Z-transform of a sequence $f(n)$: `ztrans(fe,n,z)`, also `invztrans(fe,z,n)`.

6. Plotting

Maple has two main commands for plotting expressions, one is for 2D and one for 3D pictures. The plot is shown in the worksheet and one can use mouse to change it size or (for 3D) turn it around. In order to send the resulting picture out, the following way is recommended. First, prepare the output pipe for instance like this:

```
>plotsetup(plotdevice=postscript,plotoutput='/temp/picure.ps',plotoptions='portrait,noborder,
    width=350,height=260');
```

where dimensions are in postscript points, so 100 pt is about 3.5 cm. Then one can call `plot` as usual and the picture is saved in `picture.ps` as postscript.

6.1. Plotting in 2D

Plotting expressions: `plot(`*ex*`,`*var*`=`*a*`..`*b*`)` will plot the graph over the given range. We can also plot functions using `plot(`*fun*`,`*a*`..`*b*`)`. If there are no other options, then with functons we can leave out also the range specification and Maple uses default $-10 \le x \le 10$.

Example: `plot(sin)` is the same as `plot(sin(x),x=-10..10)`. One can use expressions for determining range, say, `plot(x*sin(x),x=-Pi..2*Pi)`. One interresting choice: `plot(x/(x+1),x=-infinity..infinity)`.

The range for the axis y is determined by the actual values, but one can also specify it using `y=`a`..`b to determine what part of the y-axis should be visible, possible excess values are cut off. Similarly for functions, for example `plot({x->x^3},1..6,-1..3.4)`. Note that implicitly axes are scaled so that the resulting picture is a square (see below).

Example: `plot({x,x^2,e1},x=-4..6,y=-5..7)` will draw three graphs in one picture, here `e1` is some previously defined expression. When list is used instead of a set, graphs are ordered and one can use separate options for every expression (see below) but range is just one for all graphs.

Maple may have problems when plotting functions defined as procedures (and functions defined by cases). In order to avoid problems, such functions (expressions) should be delayed, if `f` is such a function, we best draw it `plot('f(x)', ... )`. Why? In `plot` first the function is uniquely determined (expression in the first argument is evaluated) and only then values are substituted for the graph. In case of more complicated procedures Maple can't figure them out without knowing substituted value x (see also below). Note also that `plot` cannot handle complex functions, for that there is a different command.

Some options:

Ranges for `x` and `y`: see above.

`style=line` is the default, then one can use `linestyle=`n to determine what kind of line, n can range between 1 (full line) and 5 (various dashes). The other possibility is `style=point`, unfortunately one cannot influence size of these points; however, one can change the symbol using `symbolstyle=` with values `BOX`, `CROSS`, `CIRCLE`, `DIAMOND`, `POINT` (note the upper case).

Newer versions of Maple allow n in `linestyle` to range between 1 and 7 and one can also use equivalent names, in the same order they are `solid`, `dot`, `dash`, `dashdot`, `longdash`, `spacedash`, and `spacedot`.

`numpoints=`n specifies at how many points the functions should be sampled. This is actually a lower limit, Maple automatically uses more points at places were it has troubles.

`discont=true` warns Maple that discontinuities are to be expected.

`axes=normal` (default, go through the origin) or `axes=boxed` or `axes=none` or `axes=frame` (along the lower and left edge).

`scaling=constrained` prevents rescaling of axes (thus showing the true shape of pictures), the default is `scaling=unconstrained` where axes are rescaled to have equal lengths in picture.

`color=red` etc. Example with ordered list of functions: `plot([x,x^2],x=-2..2,color=[red,blue])` Values in Maple: `white`, `yellow`, `gold`, `wheat`, `pink`, `orange`, `red`, `magenta`, `maroon`, `coral`, `violet`, `green`, `aquamarine`, `blue`, `cyan`, `navy`, `turquoise`, `plum`, `sienna`, `brown`, `tan`, `gray`, `black`. It is also possible to determine color using floating point numbers between 0 and 1 and the form `color=COLOR(RGB,`r`,`g`,`b`)` or `color=COLOR(HUE,`h`)`.

`thickness=1` (default) `thickness=2` (good for printing) and `thickness=3` (good for projecting).

`legend="`*text*`"` draws a little segment of the same color and graph and *text* next to it. If a list of expression is supplied, one can also supply a list of texts.

`xtickmarks=[]` for no tick marks on x-axis, `xtickmarks=[-1,0,2,Pi]` etc. It is also possible to specify `xtickmarks=`n to have approximately n equally spaced marks, `xtickmarks=0` makes none.

`ytickmarks=` for tick marks on y-axis, see above.

`filled=true` fills the area between two graphs, they must be specified as an ordered pair (see above).

`title=`*title* is printed over the picture.

`labels=["`*xlabel*`","`*ylabel*`"]` puts labels next to axes, unfortunately not at their ends and one cannot move them.

Options `font`, `titlefont`, `axesfont`, `labelfont` determine font to be used for text in textput (see below), title in a picture, figures at axes and labels at axes, respectively. All work the same, we will illustrate it on the first one. Possible forms: `font=[`*font*`]`, possibilities are `courier`, `helvetica`, `times`, and `symbol`. One can also specify `font=[`*font*`,`*size*`]`, but note that `times` cannot be used this way. The last possibility is `font=[`*font*`,`*style*`,`*size*`]`. Here it gets tricky. Both `courier` and `helvetica` can be `bold`, `oblique`, and `boldoblique`. On the other hand, `times` can be `roman` (default, normal style), `bold`, `italic`, and `bolditalic`.

Trick: define options into a variable `plotops:=x=-1..1,y=-2..2,color=red` and then plot using `plot(`*ex*`,plotops)`.

Trick for graphs with parameters: if we have a function `f:=(a,b,x)->a*x+b` then `plot(f(2,-1,x),x=-5..5)` (note that in the argument of `plot`, including `x` in `f` made it into an expression). With an expression f one can do `plot(subs({a=2,b=-1},f),x=-5..5)`. Note that it works since Maple first evaluates the expression given and only then substitutes for x.

One can also plot a broken line using `plot([[`*x1*`,`*y1*`],[`*x2*`,`*y2*`], ... ],`*options*`)`.

Other plotting commands: `textplot([`a`,`b`,`*text*`],font=[helvetica,bold,14])` prints *text* starting at coordinate $[a,b]$. Another good font: `font=[times,italic,10]`. By default, the text is centred over the given point. One can change it using `align=LEFT` or `RIGHT` or `ABOVE` or `BELOW`, one can also combine two, `align={LEFT,ABOVE}`. Note the upper case.

`circle([`a`,`b`],`r`)` plots circle with center $[a,b]$ and radius r.

`arrow([`a`,`b`],[`A`,`B`],`c`,`d`,`e`)`

All these plots can be included in `display`, see 7.1. Advanced plotting.

Other coordinate systems:

Polar: `plot(phi*sin(phi),phi=-Pi..2*Pi,coords=polar)` will draw the curve given by the specified function $r = r(\varphi)$, it also shows polar coordinate system (the main axis and a circle around the circumference for angles).

The coordinate system itself can be plotted using for instance `coordplot(polar,[0..1,0..3*Pi/2])`.

6.2. Plotting in 3D

`plot3d(`*ex*`,x=`*a*`..`*b*`,y=`*A*`..`*B*`)` here `ex` is an expression with variables `x` and `y`.

Options: see `plot`, also

`view=`*za*`..`*zb* determines the range of values of f that should be shown in the graph.

One can also use `view=[`*xa*`..`*xb*`,`*ya*`..`*yb*`,`*za*`..`*zb*`]`.

`style=patch` (default, surface colored and marked by a square grid) or `style=patchnogrid` (surface colored, but no grid). Conversely, two styles draw just grid but do not fill the surface using colors, `style=wireframe` (shows all) and `style=hidden` (takes visibility into account). Finally, `style=contour` means surface outlined using elevation curves and `style=patchcontour` shows surface colored with elevation curves on it. One can also use `style=point`. Then it is a good idea to use asymetric grid setting.

`numpoints=`n now specifies the total number of points used, so in one direction it is actually about $\sqrt{n}$ points.

`grid=[`*nx*`,`*ny*`]` sets up grid for x-axis and y-axis. As with `numpoints`, Maple can add points as it sees fit.

`orientation=[`α`,`β`]` determines viewpoint, α gives horizontal direction, $\alpha = 0$ views the graph from the direction of ∞ on the x-axis, $\alpha = 90$ from the direction of ∞ on the y-axis. β gives vertical angle, $\beta = 0$ means view from straight above, $\beta = 90$ from the level of the xy-plane.

`projection=`r where r is a floating point number between 0 and 1 defines what degree of distortion due to perspective should be used. $r = 0$ means strong distortion, $r = 1$ no distortion (default).

One can control the way color is distributed in a graph. The option `shading=none` means that color does not change, Maple uses uniform black for grid lines and white for surfaces. Other options: `shading=xyz` means that color depends on values of x, y, z, similarly `shading=xy` and `shading=z`. The latter has special forms `shading=zhue` (more vivid than `z`) and `shading=zgrayscale`.

`color` controls coloring in a different way. Note that when `color` is used, then settings of `shade` are ignored! This option accepts values as described above for `plot` that define constant colors, but one can also define variable colors that depend on positions of points. `color=`*ex* defines "tint", that is, the way color passes from red through blue, green, yellow back to red, based on the given expression *ex* with coordinates x, y. `color=COLOR(GRB,`*r-ex*`,`*g-ex*`,`*b-ex*`)`, where *r-ex*, *g-ex* and *b-ex* are expressions of variables x, y, defines colors directy. Note that expressions in the above options can have arbitrary range. Maple goes through the graph, find maximal and minimal values for these expressions and then rescales them to the range 0..1.

If we use `color`, we can also specify illumination of the surface. Two possibilities:

`ambientlight=[`r, g, b`]` illuminates equally from all directions using the light of the specified color.

`light=[`*phi*`,`*theta*`,`r, g, b`]` sets up a spotlight of specified color that illumintes the surface from the given angle, where *phi* is vertical angle (0=top, 90=front) and *theta* is the horizontal angle (0=front, 90=right), in fact the order of angles is the opposite of the order in `orientation`. Note that these angles are with respect to the observer, not the coordinate axes, so when `orientation` is changed, then the observer sees the surface to rotate, but sources of light stay the same. One can set up more lights.

Other coordinate systems:

Spherical: `plot3d(sin(phi)*cos(theta),phi=-Pi..2*Pi/3,theta=0..Pi,coords=spherical)` will draw the surface given by the specified function $r = r(\varphi, theta)$.

Cylidrical expects `coords=cylindrical` and $r = r(\theta, z)$

Coordinate systems can be plotted using for instance `>coordplot3d(cylidrical,[0..1,0..7*Pi/4,-1..1]);`.

*6.3. Parametric curves

Plotting parametric curve given by $x =$*ex1*, $y =$*ex2*: `plot([`*ex1*`,`*ex2*`,t=`*a*`..`*b*`])`

Parametric curve in 3D: `plot3d([`*ex1*`,`*ex2*`,`*ex3*`],s=`*a*`..`*b*`,t=`*A*`..`*B*`)`

Options: See options for `plot`.

To draw a curve $[x(t), y(t), z(t)]$ one can use `plot3d` and simply not use s is formulas, but the range must be specified and `s=0..0` is not permissible. One can try, however, `t=`*ta*`..`*tb*`,s=0..1` and then `grid=[`*nt*`,1]`. There is also a special command for this situation, see section 7.1. on Advanced plotting.

*7. Packages

Packages contain further commands. They are loaded using `>with(`*package*`);`, then one can use thise new commands. It is also possible to do it in one step, for instance instead of `>with(linalg);` and `>det(m);` one can do `>linalg[det](m);`.

7.1. Advanced plotting

The package is called `>with(plots);`.

With this package, one can set options for all successive plots using `setoptions( ... )` and `setoptions3d(`*dots*`)`.

An important command from this package is `display`. It allows one to first create several plots and then show them in one picture. First assign them to variables like this: `>p1:=plot();`, `>p2:=plot();`, `>p3:=implicitplot();` etc. and then print using `>display([p1,p2,p3]);`, they are overlayed. If `p1` through `p6` are plots, one can arrange them into a matrix of two rows by three using
`>B:=array(1..2,1..3,[[p1,p2,p3],[p4,p5,p6]]);` and then `>display(B);`.

Fonts (`font`, `axesfont` etc.) can be specified in `display` and then they are common for all pictures. Note that when using `display`, orientation specified in individual graphs is ignored and the resulting picture uses the default orientation or the one specified as an option in `display`. Other options that are usually set globally by `display` are `labels`, `linestyle`, `scaling`, `symbol`, `title`, and `thickness`.

If `display` is given a list of plots and the special option `insequence=true` is used, then these plots are not merged, but shown in sequence, one after another. In this way an animation can be made.

There is also the command `animate(`*ex*`,x=`*a*`..`*b*`,k=`*A*`..`*B*`,frames=`*n*`)` that creates an animation. The expression *ex* should feature variables `x` (treated as independent variable) and `k` (treated as index). The animation shows n frames, during which the index `k` increases from A to B in equal steps (note: $B-A$ need not be divisible by n, they may not even be integers). Similarly `animate3d` makes 3D-animations.

New plotting commands:

This package offers special versions of `plot` that use logarithmic scales for axes: `logplot` scales the y-axis, `semilogplot` scales the x-axis and `loglogplot` scales both. One can tell at which points to sample the given function using the option `sample=[ ... ]`.

Special commands for visualizing functions of two variables: The command `>densityplot(`*ex*`,`*rangex*`,`*rangey*`);` uses shades of grey to show values of the function *ex*.

`contourplot(`*ex*`,`*rangex*`,`*rangey*`)` will draw contours in the xy-plane, one can set `contours=25`, and `filled=true` fills the areas between contour lines with colors, this can be influenced using `coloring`, e.g. `coloring=[white,black]` produces greys. `contourplot3d` combines the `contourplot` picture with the real shape of the graph, it is good to use `shading=none` to make contour lines black. This command can also handle parametric surfaces.

`spacecurve([`*ex1*`,`*ex2*`,`*ex3*`,t=`*a*`..`*b*`])` plots the parametric curve given by $x =$*ex1*, $y =$*ex2*, $z =$*ex3*. It by default attempts to apply coloring to this curve, whic does not look good, use `shading=none`.

`tubeplot([`*ex1*`,`*ex2*`,`*ex3*`,t=`*a*`..`*b*`])` works similarly, but instead of a curve it plots a tube of radius 1. Other radii can be specified using the option `radius=`*r*.

Example: `tubeplot([t,cos(t),2*sin(t)],t=0..4*Pi,radius=t/8,scaling=constrained)`.

A special version of `spacecurve` is `pointplot3d(`*points*`)`. It accepts points in the form $[[x1, y1, z1], [x2, y2, z2], \dots]$ and plots them. If these points are given by formulas, one can prepare such a list easily, e.g.
`>points:=[seq(evalf([t*Pi/10,cos(t*Pi/10),sin(t*Pi/10)]),t=0..40)];`

There is also a 3D version for placing texts: `textplot([`*a*`,`*b*`,`*c*`,`*text*`])`.

Surfaces are constructed by evaluating the given function at grid points and using the z-values to draw four-sided polygons. We can supply the z-values ourselves and draw the corresponding survace with the command `listplot(`*values*`)`, where *values* is a list of lists specifying values in the form $[[z11, z12, \dots, z1n], [z21, z22, \dots], \dots, [zn1, \dots, znn]]$. Maple then constructs a surface over the square $[1, \dots, n] \times [1, \dots, n]$.

If we also want to specify the domain of the surface, we have to use the command `surfdata(`*data*`)`, where now *data* is of the form $[[p11, p12, \dots, p1n], [p21, \dots], \dots, [pn1, \dots, pnn]]$ with $pij = [xij, yij, zij]$. Such a list can be constructed for instance as follows. If the values are given by a function f, we can do
`data:=[seq([seq([i/2,j/2,eval(`$f(x,y)$`,x=i/2,y=j/2)],i=-10..10)],j=-10..10)]`.

As usual, these commands use the same options as `plot3d`.

The command `inequal` expects a set of inequalities, draws the border for each and shows the intersection of permissible areas (that is, the set of solutions to all inequalitites). Various aspects can be controlled using a specific type of option `option=(`*specifications*`)`. Specifications are: `optionsclosed` for drawing borders that are included, `optionsopen` for borders that are not included (sharp inequalities), `optionsfeasible` for the region of solutions, and `optionsexcluded` for its complement. Example:
`inequal({x+y<=1,x-y*5>-3},x=-4..4,y=-4..4,optionsfeasible=(color=grey),`
`optionsexcluded=(color=white),optionsclosed=(thickness=3),optionsopen=(linestyle=3,thickness=1))`

Special command for drawing regular polyhedra: `polyhedraplot([0,1,1],polytype=hexahedron, polyscale=2)`, this command expects center, the size of the solid is determined by `polyscale` and the type can be `tetrahedron`, `hexahedron`, `octahedron`, `dodecahedron`, and `icosahedron`.

Vector calculus:

`gradplot(`*ex*`,x=`*a*`..`*b*`,y=`*A*`..`*B*`)` prints gradient field of a function given by expression *ex* using little arrows.

`fieldplot([`*ex1*`,`*ex2*`],x=`*a*`..`*b*`,y=`*A*`..`*B*`)` attempts to give an idea of values of a vector function.

Both commands have 3D versions, `gradplot3d` and `fieldplot3d`. All four commands accept the option `arrows=SLIM`, other values are `THIN`, `THICK`, `LINE`.

Spherical coordinates:
`sphereplot(`*rad*`,theta=`*a..b*`,phi=`*A..B*`)` plots curve given by the radius function *rad* in polar coordinates.
If the curve cannot be expressed in this way using polar coordinates, one can do also
`sphereplot([`ϱ`,`θ`,`φ`],s=`*a..b*`,t=`*A..B*`)` for $\varrho = \varrho(s,t)$, $\theta = \theta(s,t)$, $\varphi = \varphi(s,t)$.

Cylindrical coordinates:
`cylinderplot(`r`,theta=`*a..b*`,z=`*A..B*`)` or `cylinderplot([`r`,`θ`,`z`],s=`*a..b*`,t=`*A..B*`)` with $r = r(s,t)$, $\theta = \theta(s,t)$, $z = z(s,t)$.

Implicit functions:
`implicitplot(`*eqn*`,x=`*a..b*`,y=`*A..B*`)` plots the implicit curve given by equation *eqn* that features `x` and `y`. Both ranges must be specified. For instance, `implicitplot(y^2=x^3-x,x=-3..3,y=-3..3)`.
Typical options: `numpoints=`n or `grid=[`nx`,`ny`]`. As usual, one can draw more curves in one picture.
Alternative: For a function f one can use `implicitplot(f,`$xa..xb$`,`$ya..yb$`)` to plot the curve given by $f = 0$.
`implicitplot3d(`*eqn*`,x=`*a..b*`,y=`*A..B*`,z=`$\alpha..\beta$`)` plots the implicit curve given by equation *eqn* that features `x`, `y` and `z`.

Differential equations:
When we solve a differential equation using
`sol:=dsolve({diff(x(t),t)=x(t)-sin(t),diff(x(t),t)=-y(t),x(0)=0,y(0)=1},{x(t),y(t)},numeric)`,
we can plot the solution using `>odeplot(sol,[t,x(t)],0..10);` (this shows x as a function) or
`odeplot(sol,[t,y(t)],0..10)` to see $y(t)$ or `odeplot(sol,[x(t),y(t)],0..10)` to see the parametric curve given by these
or `odeplot(sol,[t,x(t),y(t)],0..10)` to se a spatial curve in 3D.
As usual, one can plot more curves in one picture by enclosing them in a list, we can even add the usual plotting options, as in `[[t,x(t),color=blue,style=point],[t,y(t),color=red,thickness=2]]`. One can use the option `view=[-10..10,-10..10]` to widen the field of view.
`fieldplot([`*ex1*`,`*ex2*`],x=`*a..b*`,y=`*A..B*`)` plots the vector field of the differential equation $ex1 \cdot y' = ex2$, where *ex1* and *ex2* are expressions that feature `x` and `y`.

All plots are internally stored using polyshapes and commands `PLOT` and `PLOT3D`, but this is a more advanced subject and a casual Maple user need not know this. One can create graphics directly using these atomic structures, or indirectly using tools described above, other commands suitable for generating `PLOT` structures are in the package `plottools`.

7.2. Geometry

The package is called `>with(geometry);`. It allows one to define geometric objects and then do various operations with them. There is one peculiar feature, variable names of the created objects are supplied as the first argument, as usuall it is recommended to delay with quotes if letters are used in more roles.
`point(A,1,1);point(B,2,3)` defines two points, it shows A, B. If we want to see details, we have to ask:
`>coordinates(A);` shows $[1,1]$.
`line(l1,[A,B])` creates a line passing through these points. Which line is it? After `>detail(l1);` we see some info including the equation $1-2x+y=0$. One can also define directly `line(l2,x+3*y+1=0)`. Trick for points not defined earlier: `line(l3,[A,point(``,3,6)])`.
`circle(c1,[point(``,3,6),4])` uses specification `[`*pt*`,`*rad*`]`, it is also possible to use `[`*pt2*`,`*pt2*`,`*pt3*`]` for a circle through three points.
`triangle(t,[`*pt2*`,`*pt2*`,`*pt3*`])` sets up a triangle, there are more ways to put in data (angles and lengths in various combinations).
Mirrorring can be done about a point using `reflect(`*var*`,`*object*`,`*pt*`)` or about a line using `reflect(`*var*`,`*object*`,`*line*`)`.
Rotation about a point: `rotation(`*var*`,`*object*`,clockwise,`*pt*`)`, one can also use `counterclockwise`.
One can intersect lines and circles using `intersection(IP,l2,c1)`, in this particular case `IP` is a sequence of points, so one has to use `map(coordinates,IP)` and get $[[-1,6],[3,2]]$.
One can also get a picture using `draw([A,B,l1,c1], scaling=constrained)`.

7.3. Linear algebra

The package is called `>with(linalg);`. It introduces new commands, but two of them actually rewrite two standard ones. `trace` was originally used for tracking procedures and `norm` applied to polynomials. To recover these old meanings use `readlib(trace)` and `readlib(norm)`.
This package works exclusively with row vectors. It can be forced to show a vector in the column form, but it would not work with it this way.
`v1:=vector(3,[a,b,c])` sets up vector $v1 = [a,b,c]$. This command also allows for functional specifications, for instance `v2:=vector(3,n->2*n)` creates $[2,4,6]$ and `v3:=vector(3,n->x^n)` creates $[x, x^2, x^3]$.

To see values of vectors (and matrices) we use `evalm`. They are also used to perform algebraic operations. Where appropriate, constants are interpreted as constant vectors, for instance `evalm(v2+1)` yields $[3,5,7]$. Also `evalm(v1*2)` does $[2a,2b,2c]$ and `evalm(v1+v3)` makes $[a+x,b+x^2,c+x^3]$.

Inner product: `dotprod(`*vec1*`,`*vec2*`)`. Note that if the second vector includes complex parts, then it is automatically conjugated in this operation (Herminitan operations), so `dotprod([1,I],[1,I])` yields $(1+i)\bullet\overline{1+i}=2$. If we do not want it, we have to specify it, `dotprod([1,I],[1,I], orthogonal)` yields 0.

Cross product: `crossprod([a,b,c],[A,B,C])` yields $[bC-Bc,cA-aC,aB-bA]$. It looks better as a column vector: `convert(%,matrix)`.

Column vectors can be entered directly using `matrix([[a],[b],[c]])` or `matrix(3,1,[a,b,c])`, but to work with them one needs to change them into row vectors, `convert(%,vector)`.

Angle between two vectors `angle(`*vec1*`,`*vec2*`)` is usually shown in the form $\arccos(\alpha)$, then one can use `evalf`.

To find the norm of a vector use `norm(`*vec*`,`*p*`)`, where p ranges between 1 and infinity. The default value `norm([a,b])` is `norm([a,b],infinity)`, that is, $\max(|a|,|b|)$, for $1\le p<\infty$ it shows the ℓ_p-norm $\left(\sum|x_i|^p\right)^{1/p}$. In particular, the usual Euclidean magnitude of a vector is `norm(`*vec*`,2)`.

One can also use `normalize(`*vec*`)` that normalizes with respect to the Euclidean norm, but the outcome often needs some simplification.

Matrices can be set up using arrays exactly as described in 1.5., including things like `sparse` or `identity`. This package also offers an alternative, `matrix` works similarly like `array` but is more powerful. or a simple matrix we can use `array([[a,b],[c,d]])` or `matrix([[a,b],[c,d]])`, both commands also accept size, `matrix(2,2,[[a,b],[c,d]])` or `matrix(2,2,[a,b,c,d])`.

Matrix also offers a functional specification, for instance `matrix(2,2,(m,n)->n*x^m)` creates the matrix $\begin{bmatrix}x & x^2\\ 2x & 2x^2\end{bmatrix}$. Using `matrix(3,3,0)` we get a 3×3 matrix with all terms equal to 0, but the alternative `array(sparse,1..3,1..3)` would take up less memory.

Special matrices: `diag(a,b,3)` creates $\begin{bmatrix}a&0&0\\0&b&0\\0&0&3\end{bmatrix}$, while `band([-1,x,1],4)` creates $\begin{bmatrix}x&1&0&0\\-1&x&1&0\\0&-1&x&1\\0&0&-1&x\end{bmatrix}$.

Jacobi matrix: `jacobian([`$f(x,y)$`,`$g(x,y)$`],[x,y])`.

Random matrix: say, `randmatrix(3,3)` has random integer entries with range $-99..99$ included, one can specify the range using `randmatrix(3,3,entries=rand0..10))`.

Modifications:

Changing an element of a vector/matrix is simple, `m[1,2]:=sin(x)` changes the term in row 1, column 2.

We can pass to a submatrix using `>submatrix(A,`*range*`,`*range*`);`.

Example: Let's set up `m2:=matrix(4,4,[[a,b,c,d],[A,B,C,D],[1,2,3,4],[x,x^2,x^3,x^4]])`, obtaining $m2=\begin{bmatrix}a&b&c&d\\A&B&C&D\\1&2&3&4\\x&x^2&x^3&x^4\end{bmatrix}$.

Then `submatrix(m2,[1,3],3..4)` yields $\begin{bmatrix}c&d\\3&4\end{bmatrix}$, the order in the list matters, `submatrix(m2,[3,1],3..4)` would flip the rows. We can also create subvectors, `subvector(m2,2,1..3)` creates $[A,B,C]$ and `subvector(m2,[4,3,1],3)` yields $[x^3,3,c]$, note that each time we have a row vector.

When we want whole rows or vectors, we may prefer `row(m2,3)` giving $[1,2,3,4]$ or `col(m2,2)` giving $[b,B,2,x^2]$.

For transferring the contents of one matrix into another one can use `copyinto(A,B,`*r0*`,`*c0*`)` which copies A into B with offset given by $(r0,c0)$, that is, $A[1,1]$ is copied into $B[r0,c0]$, $A[1,2]$ is copied into $B[r0,c0+1]$ etc., extra parts of A (if any) are ignored.

Or one can write a procedure, which allows for more complicated transfers, for instance
`for i=1 to 4 do B[i,5-i]:=v1[i] od`.

Finally, `transpose(`A`)` yields A^T and `htranspose(`A`)` transposes and also applies complex conjugation.

Operations:

In the following examples we will use `m:=matrix([[a,b],[c,d]])`.

We evaluate matrices using `evalm`. We have the usual addition and multiplication by a scalar. When adding, scalar is understood as a diagonal matrix, so `evalm(m+x)` yields $\begin{bmatrix}a+x&b\\c&d+x\end{bmatrix}$.

We can apply functions to terms of matrices using `map`, for instance `map(sqrt,m)` gives $\begin{bmatrix}\sqrt{a}&\sqrt{b}\\\sqrt{c}&\sqrt{d}\end{bmatrix}$.

Matrix multiplication is denoted `&*`, for instance `evalm(m &* [2,3])` returns $[2a+3b,2c+3d]$, which is obviously the outcome of $\begin{bmatrix}a&b\\c&d\end{bmatrix}\cdot\begin{bmatrix}2\\3\end{bmatrix}$. Maple knows powers, `m^3` means `m&*m&*m`.

`inverse(`A`)` yields inverse matrix of A. One can also try `evalm(1/m)`.

`det(A)` calculates the determinant of matrix A. As usual it works symbolically, `det(m)` yields $ad - bc$. Maple also knows `trace` and `rank`.

Matrix norms: Both `norm(A)` and `norm(A,infinity)` do $\max_i\left(\sum_j |a_{ij}|\right)$. `norm(A,1)` does $\max_j\left(\sum_i |a_{ij}|\right)$, and `norm(A,frobnius)` returns $\sqrt{\sum_{i,j} |a_{ij}^2|}$. Finally, `norm(A,2)` returns $\max(\sigma(AA^T))$, it is very calculation intensive.

Equations et al.:

`linsolve(A,b` returns the solution of $Ax = b$ for matrix A of size $n \times m$ and a row vector b of dimension n. Solution is given as a list, for instance, for `b:=vector(2,[2,3])` the command `linsolve(m,b)` gives $[(-2d+3b)/(bc-ad), (-2c+3a)/(bc-ad)]$.

If there are more solutions, Maple uses parameters `_t1`, `_t2` etc. Then one migh appreciate a more general version of this command. `linsolve(A,b,'var','v')` will store the rank of A in variable *var* and uses `_v1`, `_v2`, ... as parameters.

If there is no solution, we may be interested in `kernel(A)` which shows the solution of $Ax = 0$.

What if we are given a system, for instance `eqnsys:={2*x+y=1,x-y=3}`? Maple has useful commands for turning this information into matrices. `A:=genmatrix(eqnsys,[x,y])` will create $A = \begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix}$. What about the right-hand side of the system? After `B:=genmatrix(eqnsys,[x,y],flag)` we get $B = \begin{bmatrix} 2 & 1 & -1 \\ 1 & -1 & -3 \end{bmatrix}$. Note the signs in the third column of B, Maple automatically changes equations into the form with zero on the right, here $2x + y - 1 = 0$ and $x - y - 3 = 0$. This has to be taken into account. Another thing to keep in mind is that b must be a vector. Therefore `b:=submatrix(B,1..2,3)` will not do, one has to use `b:=convert(submatrix(B,1..2,3),vector)` or `b:=col(B,3)`. Obviously `A:=submatrix(B,1..2,1..2)`.

The command `leastsqrs(A,b)` finds x_0 so that $\|Ax_0 - b\|_2 = \min_x \|Ax - b\|_2$.

`gausselim(A)` returns upper-triangular matrix obtained using Gauss elimination. It works symbolically, so the answer will most likely contain fractions. One can try `ffgausselim(A)` to prevent it.

`eigenvals(A)` returns a sequence of eigenvalues. Note that symbolic calculations get often beastly, then one can try `eigenvals(map(evalf,A))`

`eigenvects(m)` returns list, where each entry has the form (in this 2×2 case) `[λ,k,{[a,b]}]`: an eigenvalue, its multiplicity, and a list of eigenvectors (here there is only one). If the matrix has floating point entries, Maple uses the superior QR method.

`charpoly(A,'x')` returns the characteristic polynomial of A with x as variable. Alternative: `det(evalm(x-A))`. Having the characteristic polynomial we can deduce eigenvalues by definition using `solve(%)`.

`GramSchmidt([v1,v2,...])`returns an orthogonal basis as a sequence. If we want an orthonormal basis, we may follow it with `seq(normalize(i),i=%)`. The outcome may be quite ugly, so the next natural step is `seq(map(combine,i,power),i=%)`.

This package also handles diagonalization, see `Svd`.

Vector calculus:

It is also done with the `linalg` package. Commands work with functions of more variables $f(x, y, \dots)$ and where appropriate also with vector functions, for example $f = [fx(x, y, z), fy(x, y, z)]$. First we set up functions on which we will show the commands: `> sf:=x*y-exp(z):` `vf:=[x*y*z,x+y+z,ln(x)+sin(y)-cos(z)];`

`grad(sf,[x,y,z])` yields $[y, x, -e^z]$.

`diverge(f,[x,y,z])` will return $\left[\frac{\partial}{\partial x} fx(x, y, z), \frac{\partial}{\partial y} fy(x, y, z), \frac{\partial}{\partial z} fz(x, y, z),\right]$, so `diverge(vf,[x,y,z])` yields $[yz, 1, \sin(z)]$

Rotation is obtained with `curl(f,[x,y,z])` that will return $\left[\frac{\partial}{\partial y} fz(x, y, z) - \frac{\partial}{\partial z} fy(x, y, z), \frac{\partial}{\partial z} fx(x, y, z) - \frac{\partial}{\partial x} fz(x, y, z), \frac{\partial}{\partial x} fy(x, y, z) - \frac{\partial}{\partial y} fx(x, y, z),\right]$, so `curl(vf,[x,y,z])` yields $[\cos(y) - 1, xy - \frac{1}{x}, 1 - xz]$. It may be useful to handle it in a column form, `convert(%,matrix)` turns it into $\begin{bmatrix} \cos(y) - 1 \\ xy - \frac{1}{x} \\ 1 - xz \end{bmatrix}$.

Maple can also work with other coordinate systems. For instance, `diverge(vf,[x,y,z],coords=cylindrical)` will inerpret `x` as r, `y` as φ and `z` as z.

The command `potential(vf,[x,y,z],'p')` returns *true* if there is a potential to the given function, that is, a function whose gradient is the given vector function. In case of success the potential is stored in the specified variable. The command `vecpotent` tries to identify a function whose curl is the given vector function.

7.4. Exploring calculus

The package is called `> with(student);`.

`showtangent(ex,x=a)` shows the graph of the expression *ex* and also plots the tangent line to it at the specified point *a*. Possible option: `showtangent(x^2,x=1,x=-0.5..2)`.

`slope([a,b],[A,B])` finds the slope of the line given by points $[a, b]$ and $[A, B]$.

Integration methods:
Set up an integral `>F:=Int(x^3*sin(x^2+1),x);`
Substitution: `changevar(u=x^2+1,F,u)` here `u` tells Maple what the new integrating variable should be.
By parts: `intparts(F,x^2)` tells Maple to calculate the integral `F` using integration by parts with choice $f = x^2$.

Integral approximation:
Numerical integration of expression *ex* from a to b with partition into n equal parts, using rectangles: `rightsum(`*ex*`,x=`$a..b,n$`)`, `leftsum(`*ex*`,x=`$a..b,n$`)`, `middlesum(`*ex*`,x=`$a..b,n$`)`; using trapezoids: `trapezoid(`*ex*`,x=`$a..b,n$`)`; using parabolas: `simpson(`*ex*`,x=`$a..b,n$`)`;
`rightbox(`*ex*`,x=`$a..b,n$`)` plots the expression *ex* from a to b, divides $[a, b]$ into n equal parts and draws rectangles corresponding to the right sum; similarly `leftbox(`*ex*`,x=`$a..b,n$`)

7.5. Combinatorics

Maple has the basic command `binomial(`n,k`)` for $\binom{n}{k}$, for more we have to call `>with(combinat);`.
Then we have `multinomial(`$n,k1,\ \ldots\ ,kN$`)`.
Combinations: `choose(`*list*`)` generates all possible combinations (selections without regard for order) and gives them as a list. Alternative: `choose(`*list*`,`k`)` generates only combinatins with k elements. Example: `choose([a,b,b,c],2)` generates $[[a, b], [a, c], [b, b], [b, c]]$. If we want to make combinations from $[1, 2, \ldots , n]$, we just write `choose(n)` or `choose(n,`k`)`.
It has a modification `numbcomb` that returns number of such combinations, for instance `numbcomb([a,b,b,c],2)` yields 4 and `numbcomb(n,k)` is the same as `binomial(n,k)`.
There is also a command `randcomb(`*list*`,`k`)` or `randcomb(`n,k`)` that generates one random combination.
If we care about the order, we use analogous commands `permute(`*list*`)`, `permute(`*list*`,`k`)`, `permute(`n`)`, `permute(`n,k`)`, `numbperm`, `randperm`.

The command `composition(`n,k`)` generates all possible decompositions $[n1, \ldots , nk]$, where $ni \in \mathbb{N}$ and $\sum ni = n$, order matters. Number of such decompositions can be obtained by `numbcomp`.
Similarly, `partition(`n`)` generates a list of all decompositions $[n1, \ldots , nk]$, where $k \in \mathbb{N}$, $ni \in \mathbb{N}$ and $\sum ni = n$, here the order does not matter. Number of such partitions: `numbpart`.

7.6. Statistics

The package is called `>with(stats);`.
It works with lists of data. Such a list typically contains numbers, but it can have also other "data points". Specification *x1..x2* means an unknown number that lies within the specified range. It is often interpreted as the average of $x1$ and $x2$. Both concrete data points and range data points can be given a weight, such a data point is specified using `Weight(`*DataPoint*`,`*weight*`)`.
Example: `data:=[1,Weight(5,3),Weight(2..4,2)]` creates a list that behaves like [1,5,5,5,?,?], where ? stands for an unknown number from the interval $[2, 4]$. We will use this list in examples below.
One can use any positive number for weight. Note that `Weight` is just a description of data, it is not a function and cannot be evaluated.
Sometimes it is useful to suggest that an unknown data point should be at some position in list, this is done using `missing`.
Data can be loaded from files using the command `importdata('`*file*`')` but one has to be careful about specifying the proper path. Data points should be separated with spaces or line feeds and missing data points are specified with `*`.

Commands in the `stats` package are further divided into subpackages that have to be loaded. Optional arguments are often specified using square brackets.

Subpackage for basic statistics is loaded `>with(describe);`
`count(`*list*`)` gives number of data points with weights factored in, `count(data)` shows 6. Using `countmissing` we learn how many data points are missing.
`mean(`*list*`)` yields the mean μ_X. Ranges are interpreted as averages, for instance data point `2..7` is considered to be $\frac{9}{2}$ for the purpose of calculating mean, `mean(data)` yields $\frac{1+5\cdot 3+3\cdot 2}{6} = \frac{11}{3}$. Variations: `quadraticmean`, `harmonicmean`, `geometricmean`.
`variance(`*list*`)` yields the variance $\mathrm{Var}(X)$, `sqrt(variance(`*list*`))` then yields the standard deviation σ_X, but we also have `standarddeviation`. Variation: `meanvariance`.
Other descriptive statistics: `median`, `quartile`, `decile`, `percentile`, `quantile`, these by default do lower quantiles etc but one can modify this, for instance lower three quantiles can be obtained using one of these: `quartile[3](data)`, `quartile[3/4](data)`, or `percentile[75](data)`.
Comparing two pieces of data: `covariance(`*list1*`,`*list2*`)` and `linearcorrelation(`*list1*`,`*list2*`)`.

Subpackage for manipulating data is loaded `>with(transform);`
`statvalue` removes weight, `statvalue(data)` yields $[1, 5, 2..4]$.
`classmark` replaces ranges with averages, `classmark(data)` yields $[1, Weight(5, 3), Weight(3, 2)]$.
Trick: after `statvalue(classmark(data))` we have $[1, 5, 3]$.

`frequency` lists weights, `frequency(data)` gives $[1,3,2]$. Cumulative frequencies (partial sums of frequencies): `cumulativefrequency(data)` gives $[1,4,6]$.

`statsort` orders data points in increasing way, `statsort(data)` yields $[1, Weight(2..4,2), Weight(5,3)]$. It works only if ranges and numbers do not overlap. If they do, one can try `statsort(classmark(`*list*`))`.

`scaleweight` multiplies all scales by a specified factor, `scaleweight[1/count(data)](data)` yields $[Weight(1,\frac{1}{6}), Weight(5,\frac{1}{2}), Weight(2..4,\frac{1}{3})]$.

`apply[`*ex*`](`*list*`)` applies a specified expression to data points (weights are preserved), `apply[sqrt(x)](data)` yields $[1, Weight(\sqrt{5},3), Weight(\sqrt{2}..2,2)]$.

For combining more lists we have `multiapply[`*fun*`](`*lst*`)`, where *fun* is a function of k variables and *lst* is a list of k sublists. Example: `multiapply[(x,y)->x*y]([[1,2,3],[a,b,c]])` yields $[a, 2b, 3c]$.

`split[`*k*`](`*list*`)` splits the given list into k sublists of equal number of data points. If the number of data points in the given list is not divisible by k, then some data points are split between sublists with appropriate weights. Example: `split[2]([1,2,3,4])` yields $[[1,2],[3,4]]$, `split[2]([1,2,3,4,5])` yields $[[1,2,Weight(3,\frac{1}{2})],[Weight(3,\frac{1}{2}),4,5]]$.

`tallyinto(`*list*`,`*ListOfRanges*`)` determines for each range $x1..x2$ from *ListOfRanges* how many data points x from *list* satisfy $x1 \le x < x2$ and presents the answer using the language of weights.

Example: `tallyinto([1,3,6,4,2,3,9,5],[1..3,3..7,7..10])` yields $[Weight(1..3,2), Weight(3..7,5), Weight(7..10,1)]$. This can be used for instance for making histograms, regular lists of ranges are easily prepared like this: `L:=[seq(10*k..10*k+10,k=0..3)]` yields $[0..10, 10..20, 20..30, 30..40]$. Then `frequency(tallyinto(`*inputdata*`,L))` shows histogram for the partition L.

The `stats` package knows lots of distributions, but to access then one has to load `>with(statevalf);` For instance, `normald[`*mu*`,`*sigma*`]` is the usual normal distribution, but in order to use it one has to first turn it into a function. `f1:=statevalf[pdf,normald[0,1]]` creates the probability density function, `f2:=statevalf[cdf,normald[0,1]]` creates the cumulative distribution function, and `f3:=statevalf[icdf,normald[0,1]]` creates the inverse cumulative distribution function (for quantiles).

Some continuous distributions: `lognormal[`*mu*`,`*sigma*`]`, `uniform[`*a*`,`*b*`]`, `exponential[`*lambda*`]`, `beta[`*nu1*`,`*nu2*`]`, `studentst[`*nu*`]`, `chisquare[`*nu*`]`, `fratio[`*nu1*`,`*nu2*`]`, `gamma[`*a*`,`*b*`]`.

For discrete distributions one has to use different keywords, `pf` for probability functions, `dcdf` for discrete cumulative distributive function, and `idcdf` for inverse discrete cumulative distributive function. Some distributions: `binomial[`*n*`,`*p*`]`, `discreteuniform[`*a*`,`*b*`]`, `poison[`*lambda*`]`, `empirical[`*ListOfProb*`]`, `negativebinomial[`*n*`,`*p*`]`, `hypergeometric[`*n1*`,`*n2*`,`*n*`]`.

The subpackage loaded `statplots` contains several commands for plotting various scatter plots and histograms.

`scatterplot(`*list*`)` draws all points with $y = 1$ and x-coordinate given by the data given. `scatterplot(`*list1*`,`*list2*`)` draws points in 2D with x and y coordinates taken from lists in the specified order.

`boxplot(`*list*`)` shows the range of the given data as a vertical segment, with quartils and median indicated.

`histogram(`*list*`)` is tricky, it works best when *list* consists only of ranges, then column heights correspond to weights. A good idea is to add the option `area=count`.

The subpackage loaded `>with(fit);` contain tools for interpolation and regression.

The command `interp(`*list1*`,`*list2*`)` creates a polynomial of degree $n-1$ going through points $[x1,y1],\ldots,[xn,yn]$, where the data are supplied as two lists, $list1 = [x1,\ldots,xn]$ and $list2 = [y1,\ldots,yn]$.

The command `spline(`*list1*`,`*list2*`,x,`*k*`))` creates an interpolating spline through the specified points using polynomials of degree k. The result is given as an expression with variable `x` specified in the command.

For regression we use the command `leastsquare` whose task is to take a given function with some parameters and determine which values of these parameters make the function fit the given data best. It is best explained by an example.

`leastsquares[[x,y],y=a0+a1*x+a2*x^2,{a0,a1,a2}]([`*list1*`,`*list2*`])` finds which parabola best fits the given data. The answer is given as, say, $y = 3 + 4x + 7x^2$. More variables can be also used.
`leastsquares[[x1,x2,y],y=a0+a1*x1+a2*x2,{a0,a1,a2}]([`*list1*`,`*list2*`,`*list3*`])`.

8. Programming

8.1. Procedures

The general form of procedure is

`proc(`*args*`)`
`local` *vars1*`;`
`global` *vars2*`;`
`options` *ops*`;`
stats`;`
`end;`

and the only obligatory part is the statements *stats*.

For some introductory examples see section 1.3. Functions.

The **output** of a procedure is given by the output of the last statement in the procedure. Sometimes this is not desirable or situation is confusing (when a procedure is very complicated), we can emphasise what is the output of a procedure using `RETURN(`*ex*`)`. Example: `> square:=proc(x) 2*x; x^2; end;` and then `> square(4);` gives 16.

Note: the outcome of individual statements in a procedure is not shown, so it makes no difference whether we use `:` or `;`. If we do want something to show when running a procedure, we use `print(`*something*`)` or `lprint(`*something*`)`. The semi-colon after `end` means that after defining, the code for the procedure is shown. If we put a colon `:` there, nothing is shown.

We can see what a procedure does using `print(`*procedure*`)` or `eval(`*procedure*`)`, for instance `eval(square)`.

Arguments specify what a procedure expects. When we call it, we have to supply it with at least as many arguments as are in the head of the procedure and their values are then used, original variables used in such a call (if any) are not influenced.

If we submit more arguments than specified in the procedure definition, then the extra ones are ignored, but we can access them using `args`, which is a set of all values that were passed to the procedure when it was called. In the procedure above, `args[1]` is the same as `x`. However, if a procedure uses extra arguments, then it also requires them when called.

If we set `> exper:=proc(x) [x-args[1],3*args[2]]; end;` and then call `> exper(2,3);` we get $[0, 9]$. However, if we call `> exper(2);`, we get an error. One can test for presence of parameters, `nargs` is a register than contains the number of parameters specified when the procedure was called.

Often one has to guarantee that parameters are of certain types. This is done in the head of procedure using `::`. For instance, a procedure that takes a root would be done like this:

```
> rt:=proc(x::nonnegint) sqrt(x); end;
```

Now if somebody calls it with a negative number, Maple will complain.

Popular types:

`integer`, `posint`, `negint`, `nonposint`, `nonnegint`, `even`, `odd`, `fraction`

`float`, `positive`, `negative`, `nonnegative`, `nonpositive`

`polynom`, `algebraic` (algebraic expression), `function`, `procedure`, `equation`, `series`

`set`, `list`, `array`, `table`, `vector`, `matrix`, `point`

`range` (type `xa..xb`), `name` (variable name), `operator`, `anything`

Specification of the form `x::{integer,float}` means union, that is, the argument can be of either type. Note that `float` does not include `integer`, Maple works numerically with the former and symbolically with the latter, see section 1.1. There is a special type `numeric` that includes types `integer`, `fraction` and `float`.

One can check on the type of an epxression using `type(`*ex*`,`*type*`)` and it returns *true* or *false*. This can be also used in a procedure. An interesting trick to do more checking at once: `type([x,y],[positive,nonnegative])`.

Side remark: If the user supplies arguments in square brackets as in `f[3,7](15)`, then 3 and 7 become parts of the name of the procedure. These can be also accessed, 3 is `op(1,procname)` and 7 is `op(2,procname)`.

Variables:

Local variables are specific for each call of the procedure, so even if a procedure calls itself again, it wil spawn new copies of local variables. Global variables serve to communicate with outside environment. A nice trick: We expect that somebody supplies information by setting the variable `how_why`. When a procedure is called, it needs to know whether this variable is indeed assigned a value. The condition `if how_why='how_why'` is *true* if the variable is unassigned.

A simple example of local variable in action:

```
> diffn:=proc(fn) local i;
    seq(diff(f,x$i),i=1..n); end:
```

Now if we type `> diffn(arctan(x),4);` we get a sequence containing the first four derivatives of arctangent (as expressions).

Maple's environmental variables (`Digits`, `Order` etc.) have one important feature: Any changes made to them in a procedure are local.

Maple's handling of variables in procedures is weird, although it might make sense for Maple wizards. Parameters are not evaluated at all, while local variables are evaluated only one level down. Thus if we have `local x,y` and define `x:=y:y:=7` in the procedure, then calling `x` in the procedure returns y! Of course, calling it in the interactive mode would yield 7. If we want a full evaluation in a procedure, we have to use `eval`.

Options:

The option `remember` tells Maple to keep a table of results from all calculations, which can speed up calls to iterative procedures. When `system` is also specified, it tells Maple that if it is running short of memory, it can erase this table.

The option `trace` is useful for debugging.

The option `Copyright "`*text*`"` codes the *text* into the procedure.

The option `operator` means that this procedure is considered a functional operator. This has no effect on its functioning. If we also add `arrow`, the output uses the arrow notation.

Example: We define `>test:=proc(x) options operator,arrow; x^2; end;` and then `>print(test);` shows $x \mapsto x^2$ instead of
proc(x) x^2 **end proc**.

Operators:
We can define procedures to act as operators, such procedures have names starting with `&`. We show examples with 1, 2 and 3 operands. Note the quotes (left quotes, for quoting variable names, not delaying) used in the definition. For each definition we will show a typical call.
`>‘&op1‘:=proc(x) 2*x+1; end;` then `>&op1 4;` returns 9.
`>‘&op2‘:=proc(x,y) sqrtx^2+y^2; end;` then `>3 &op2 4;` returns 5.
`>‘&op3‘:=proc(x,y,z) x+y-z; end;` then `>&op(3,4,5);` returns 2.
Note that the call for operators with three and more operands is the same as for functions, so it actually makes no difference whether we define it as a function or an operator.

8.2. Controlling the flow

For branching we use the general form
`if` *cond1* `then` *commands*
`elif` *cond2* `then` *commands*
`elif` *cond3* `then` *commands*
⋮
`elif` *condn* `then` *commands*
`else` *commands*
`fi;` where `elif` and `else` parts are optional.

For loops we can use `seq` or the general pattern `do` *commands* `od` that by itself defines an endless loop. We can end it in several ways, namely by preceding it with `for` or `while` constructions or by interrupting it with `break`.
a) `for` *var values* `do` *commands* `od`
Previous value of *var* (if any) is erased, after the loop it has the last value it had there. Specification of values can be using the traditional form `from 1 to 5`, with modification `from 5 to 1 by -2`, or using a list `in [3,sqrt(5),u]`.
Note that the loop `for i from to 3 by -1` is not executed at all.
b) Another possibility is `while(`*cond*`) do` *commands* `od`

These are all equivalent, they print 1, 2, 3:
```
>x:=0: do x:=eval(x)+1; if x>3 then break; fi; print(x); od;
>for i from 1 to 3 do print(i); od;
>for i from 1 to 3 by 1 do print(i); od;
>for i in [1,sqrt(2),-sqrt(3)] do print(i^2); od;
>x:=0:  while(x<3) do x:=eval(x)+1; od;
```
Note the `eval` command, otherwise the loop would never end. Also, the `while` loop has the special property that it shows all outcomes of commands, so there si no need to print.
One can interrupt any loop with `break`. The command `next` means that all successive commands up to `do` are skipped.
Loops have problem with evaluating. In particular, the `from to` construction cannot accept expressions requiring symbolic evaluation. For instance, `for i from 0 to Pi by (Pi/8)` will not work.
Similarly, while logically we could have replaced the `while` condition above with `while(x<Pi)`, Maple will not be able to handle it. Here one can help it using `while(evalf(x)<evalf(Pi))`.