Technische Universität Ilmenau
Fakultät für Mathematik
und Naturwissenschaften
Institut für Mathematik

Postfach 10 0565
98684 Ilmenau
Germany
Tel.: 03677/692652
Fax: 03677/691241
Telex: 33 84 23 tuil d.
email: W.Neundorf@mathematik.tu-ilmenau.de

Preprint No. M 07/99

Programming in Maple V Release 5

Extended Basics

Werner Neundorf

Februar 1999

‡MSC (1991): 68-01, 68-04, 68Q40, 68Q25

Abstract

This is a tutorial on programming in Maple version V Release 5.
It is based on the script of M.Monagan, "Programming in Maple: The Basics.", written for Maple V Release 3. Here, there are involved and adapted new or other Maple commands and further examples.
The aim is to show how you can write simple programs in Maple for doing numerical calculations, linear algebra, and programs for simplifying or transforming symbolic expressions, polynomials or mathematical formulas. It is assumed that the reader is familiar with using Maple interactively.

Preface

It is an introduction to programming in Maple with examples and exercises.
This gives more insight and examples than the Language Reference Manual does and covers. I have adapted the worksheets in the given script of M.Monagan [1] to Maple V Release 5, making some additional explanations and several corrections. There are added new tutorial aspects.

A few words to those who are familiar with other programming languages [1].

- Maple is a procedural programming language. It also includes a number of functional programming constructs. If you have written programs in Basic, Pascal, Algol, C, Lisp, or Fortran, you should be able to write numerical programs in Maple very quickly.
- Maple is not strongly typed like C and Pascal. No declarations are required. Maple is more like Basic and Lisp in this respect. However types exist. Type checking is done at run time and must be programmed explicitly.
- Maple is interactive and the programming language is interpreted. Maple is not suitable for running numerically intensive programs because of the interpreter overhead. Though it is suitable for high-precision numerical calculations and as a tool for generating numerical codes.

This document is based on Maple V Release 5. Maple development continues. New versions come out every one or two years which contain not only changes to the mathematical capabilities of Maple, but also changes to the programming language and user interface. We are pleased and somewhat surprised to see how quickly this movement is already happening. For this reason, I have applied constructs in the language that will be probable in the language in future versions of Maple. Some of the important things being stated in [1] are not mentioned elsewhere, or they are buried in this documentation.
By itself or when coupled with other computing environments and Computer Algebra Systems, Maple can be incorporated effectively into the curriculum to enhance the understanding of both fundamental and advanced topics, while enabling the student actively to put theory into practice.
Use the Maple online help facility by typing `?command` at the Maple input, where `command` is the name of the function about which you would like information. Even better, access help from the menu. In the bibliography there are given some useful sources of information and supplemental workbooks for Maple.

Contents

1 Introduction **4**

1.1 Evaluation 4
1.2 Expressions: Sums, Products, Powers, Functions 5
1.3 Indexed names and Functions 7
1.4 Statements: Assignment, Conditional, Loops 9

2 Data Structures **12**

2.1 Sequences 12
2.2 Lists or Vectors 14
2.3 Sets 16
2.4 Tables 18
2.5 Arrays 20
2.6 Records 27
2.7 Linked Lists 29

3 Procedures **32**

3.1 Parameters, Local Variables, RETURN, ERROR 32
3.2 Tracing Procedure Execution: printlevel 35
3.3 Arrow Operators 38
3.4 Scope Rules: Parameters, Locals, Globals 40
3.5 Evaluation Rules: Actual, Formal Parameters, Locals, Globals 41
3.6 Recurrence Equations and Option Remember 43
3.7 Types and Map 45
3.8 Variable Number of Arguments: args and nargs 49
3.9 Returning Unevaluated 50
3.10 Simplifications and Transformation Rules 51
3.11 Optional Arguments and Default Values 55
3.12 Returning Results Through Parameters 58

4 Programming **60**

4.1 Matrix and Vector Computation 60
4.2 Numerical Computation 67
4.3 Computing with Polynomials 73
4.4 Reading and Saving Procedures: read and save 79
4.5 Debugging Maple Programs 82
4.6 Interfacing with other Maple Facilities 83
4.7 Calling External Programs 86

4.8 File Input/Output of Graphics and Numerical Data 87
4.9 Fortran and C output 89

5 Examples 93
5.1 Einfache Differential- und Integralrechnung 93
5.2 Schneeflockenkurven 94
5.3 Newtoniteration mit Filearbeit und Graphik 98
5.4 Hornerschema 100

6 Appendix 103

A Maple Export As

B Basic Commands

C WWW and Files Addresses

D Maple Text Styles

1 Introduction

1.1 Evaluation

If an identifier has not been assigned a value, then it stands for itself. It is a symbol. Symbols are used to represent unknowns in equations, variables in polynomials, summation indices, etc.

Assignment statement

```
> restart;
> p := x^2+4*x+4;
```

$$p := x^2 + 4\,x + 4$$

The identifier p has been assigned the formula $x^2 + 4x + 4$.
The identifier x has not been assigned a value, it is just a symbol, an unknown.
The identifier p has been assigned a value. It is now like a programming variable, and its value can be used in subsequent calculations just like a normal programming variable.

Assigned identifier (value)

```
> p;
```

$$x^2 + 4\,x + 4$$

Unassigned identifier (symbol)

```
> x;
```

$$x$$

Because a variable can be assigned a value which contains symbols, the issue of evaluation arises. p should it evaluate the polynomial.

```
> x := 3; x;
  p;
```

$$x := 3$$
$$3$$
$$25$$

This differences between Maple and traditional programming languages, where identifiers can be used for both programming variables and mathematical unknowns, is nice. But be careful not to mix the two. Many Problems that users encounter have to do with using identifiers for both symbols and programming variables.
What happens, if we now try to compute an integral?

```
> int(p,x);
```

```
Error, (in int) wrong number (or type) of arguments
```

An error has occured in the integration function *int*. Here we are thinking of x as a symbol, the integration variable. But x was previously assigned the integer 3. Maple evaluates the arguments to the *int* function and tries to integrate 25 with respect to 3. It doesn't make sense!
How does one convert x back into a symbol?

Unassigns the variable x

```
> x := 'x';
```

$$x := x$$

```
> int(p,x); # x as integration variable
```

$$\frac{1}{3}x^3 + 2x^2 + 4x$$

Assignments for the control variable in traditional loops

```
> i := 1; s := 0;
  for i from 1 to 10 do s:=s+i^2; od:
  s;
```

$$i := 1$$
$$s := 0$$
$$385$$

In programming languages it's no problem, but in Maple ...

```
> i := 1; sum(i^2,i=1..10);
```

$$i := 1$$

```
Error, (in sum) summation variable previously assigned,
second argument evaluates to, 1 = 1 .. 10
```

Unassign the variable i

```
> i := 'i'; sum(i^2,i=1..n);
```

$$i := i$$

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

1.2 Expressions: Sums, Products, Powers, Functions

In Maple, mathematical formulas, e.g. things like $\sin(x+\pi/2)$ and $x^3y^2 - 2/3$ are called *expressions*. They are made up of symbols, numbers, arithmetic operators and functions. The formula $p = x^2y + 3x^3z + 2$, which is a polynomial, is input in Maple as

```
> restart;
  p := x^2*y + 3*x^3*z + 2;
```

$$p := x^2\,y + 3\,x^3\,z + 2$$

and the formula $\sin(x+\pi/2)\ e^{-x}$ is input as

```
> sin(x+Pi/2)*exp(-x);
```

$$\cos(x)\,e^{(-x)}$$

Notice that Maple simplified $\sin(x+\pi/2)$ to $\cos(x)$, because the cos function is prefered. Formulas in Maple are represented as expression trees or **DAGs** (Directed Acyclic Graphs) in computer jargon. When we program Maple functions to manipulate formulas, we are basically manipulating expression trees. The are fore basic routines for examining these expression trees and other useful functions.

function	syntax	content, remark
type	$type(f,t)$	expression f of type t $\rightarrow$ true
whattype	$whattype(f)$	type of expression f
nops	$nops(f)$	number of operands or terms of expression f
op	$op(i,f)$	i'th operand of expression f, $0 \leq i \leq nops$
	$op(i..j,f)$	operands of f from i to j
	$op(1..nops(f),f)$	abbreviation is $op(f)$

Basic types: *name, symbol, string, integer, fraction, rational, float, +, * , ^*
function, procedure, indexed, numeric, anything
sequence, list, set, vector, matrix,...

Two other numerical types are *rational* and *numeric*. The type *rational* refers to the rational numbers, i.e. integers and fractions. The type *float* refers to floating point numbers, i.e. numbers with a decimal point in them. The type *numeric* refers to any of these kinds of numbers, i.e. numbers of type *rational* or *float*.
Maple users will have noticed that Maple distinguishes between exact rational numbers and approximate numbers. The presence of a decimal point is significant!

```
> 2; 2/3; # rational numbers
  2/3.0; # 10 digits standard
```

$$2$$
$$\frac{2}{3}$$
$$.6666666667$$

```
> whattype(2); whattype(2.0);
  whattype(2/3); whattype(2/3.0);
  type(2/3,rational);
```

integer
float
fraction
float
true

Polynomial p is a sum of 3 terms.
The second term of the sum p is a product of 3 factors.

```
> type(p,integer); type(p,numeric);
  whattype(p);
  type(p,`+`); # `+` back quote character
```

false
false
$+$
true

```
> nops(p);
  op(1,p); op(2,p); op(3,p);
  op(p); op(1..3,p);
  op(0,p); # the type of p
```

$$3$$
$$x^2\,y$$
$$3\,x^3\,z$$
$$2$$
$$x^2\,y,\;3\,x^3\,z,\;2$$
$$x^2\,y,\;3\,x^3\,z,\;2$$
$$+$$

```
> q := 'q'; op(0,q); op(1,q);
```

$$q := q$$
symbol
$$q$$

```
> type(op(2,p),`*`); # a product
  nops(op(2,p)); # it has 3 factors
  op(1,op(2,p)); # here is the first factor
  op(op(2,p)); # here is a sequence of all 3 factors
```

$$true$$
$$3$$
$$3$$
$$3,\ x^3,\ z$$

1.3 Indexed names and Functions

Maple has two kinds of variables or names as Maple calls them.
There are strings like x, sin, Pi, which are of type *name* or *symbol* and *indexed names* or subscripted variables like $A[1]$, $A[i, j]$, $A[i][j]$, which are of type *indexed*: They occur i.e. in connections with vectors or matrices.
Most functions in Maple accept both kinds of variables.
The Maple type *name* means either a symbol or a subscript.

```
> restart;
  type(sin,function); whattype(sin);
  type(sin(x),function); whattype(sin(x));
  type(a,name);
  type(a,symbol);
  type(a,string);
```

$$false$$
$$symbol$$
$$true$$
$$function$$
$$true$$
$$true$$
$$false$$

```
> whattype(Pi);
  whattype(`hallo`); # back quotes
  whattype("Hallo"); # string in double quote characters
  print(`hallo`,"Hallo");
```

$$symbol$$
$$symbol$$
$$string$$
$$hallo, \text{"Hallo"}$$

```
> type(A[1],name); type(A[1],string);
  type(A[1],indexed); whattype(A[1]);
  A[1] := 1.0; whattype(A[1]);
```

$$true$$
$$false$$
$$true$$
$$indexed$$
$$A_1 := 1.0$$
$$float$$

Vector, matrix and indexed name

```
> B := vector([2,4,6]);
  B[1];
  whattype(B[1]);
```

$$B := [2, 4, 6]$$
$$2$$
$$integer$$

```
> C := matrix(2,2,[i,j,k,l]);
  whattype(C[1,1]);
```

$$C := \begin{bmatrix} i & j \\ k & l \end{bmatrix}$$
$$symbol$$

If f is an indexed name, then the *nops* function returns the number of indices, and the *op* function returns the i'th index, $op(0, f)$ returns the indexed name.

```
> nops(A[i,j]);
  op(1,A[i,j]); op(2,A[i,j]);
  op(0,A[i,j]);
```

$$2$$
$$i$$
$$j$$
$$A$$

```
> A[i][j];
  nops(A[i][j]);
  op(1,A[i][j]);
  op(0,A[i][j]);
```

$$A_{i\,j}$$
$$1$$
$$j$$
$$A_i$$

Functions work very similarly to indexed names.
The syntax for a function call is $f(x1, x2, ...)$, where f is the name of the function, and $x1, x2, ...$ are the arguments.
The *nops* function returns the number of arguments, and the *op* function returns the i'th argument, $op(0, f)$ returns the name of the function.

```
> nops(f(x,y,z));
  op(1,f(x,y,z)); op(1..3,f(x,y,z)); op(f(x,y,z));
  op(0,f(x,y,z));
```

$$3$$
$$x$$
$$x,\ y,\ z$$
$$x,\ y,\ z$$
$$f$$

Attention

```
> op(f); op(1,f); op(0,f);
```

$$f$$
$$f$$
$$symbol$$

Now we create and pick apart any Maple formula (not a function) interactively.

```
> f := sin(x[1])^2*(1-cos(Pi*x[2]));
```

$$f := \sin(x_1)^2\,(1 - \cos(\pi\, x_2))$$

Type, operands and parts of formula

```
> type(f,name);
  whattype(f);
  nops(f); op(1,f); op(2,f);
```

$$false$$
$$*$$
$$2$$
$$\sin(x_1)^2$$
$$1 - \cos(\pi\, x_2)$$

```
> whattype(op(1,f));
  op(1,op(1,f)); op(2,op(1,f));
  op(0,op(1,op(1,f))); op(1,op(1,op(1,f)));
```

$$\wedge$$
$$\sin(x_1)$$
$$2$$
$$\sin$$
$$x_1$$

```
> whattype(op(1,op(1,op(1,f)))); # type of x1
```

$$indexed$$

1.4 Statements: Assignment, Conditional, Loops

Maple syntax for the assignment, *if*, *for* and *while* statements is taken from Algol 60.

Assignment statement

$$\boldsymbol{name := expr}$$

expr is any expression and *name* is a variable name.
We want to mention here an evaluation problem. Its arises because variables can be assigned expressions which contain symbols as well as numbers.

```
> restart;
  p := x^2+4*x+4;
  p := p+x;
```

$$p := x^2 + 4\,x + 4$$
$$p := x^2 + 5\,x + 4$$

What happens if a variable arises at the right hand sight and was not already assigned a value?

```
> q := q+x;

Warning, recursive definition of name
```

$$q := q + x$$

```
> q := q+x;

Error, too many levels of recursion
```

Well, you may say, obviously the user forgot to assign q a value first.
Indeed that is probably the case, but let us continue and see what happens to Maple. Maple certainly allows q to not have a value just like x in the previous example.
The above statement resulted in a warning from Maple.
The name q is now assigned a formula which involves q. What happens now if we try to evaluate q? To evaluate $q + x$ we have to evaluate q again. Therein lies an infinite evaluation loop. Maple gives a warning.

To try the same once more Maple runs out of Stack space and is able to stop just before it dies and report that there is a problem. On some systems Maple will die.

Note that Maple does not detect all recursive assignments because this would be too expensive to do in general.

To recover from such a error message, simply unassign the variable q.

```
> q := 'q';
  q := q+x;
```

$$q := q$$

```
Warning, recursive definition of name
```

$$q := q + x$$

In a conventional programming language, this problem of a recursive definition cannot arise because all variables must have values.
If they haven't been assigned a value, this is really an error. Depending on the language, variables either get a default value, or they get whatever junk happens to be present in memory at the time, or in a few cases, the language attemps to detect this and issue an error.

Conditional statement or if statement

if ***expr*** **then** ***statseq***
[**elif** ***expr*** **then** ***statseq***]*
[**else** ***statseq***]
fi

statseq is a sequence of statements separated by semi-colons,
[ ] denotes an optional part,
* denotes a part which can be repeated zero or more times.

```
> x:=3;
  if x<0 then -1 elif x=0 then 0 else 1 fi;
  if x<0 then signx:=-1 elif x=0 then signx:=0 else signx:=1 fi;
```

$$x := 3$$

$$1$$

$$signx := 1$$

Loop statement or for - while statement

The loop statements has two variants.

(1) For-while loop

[**for** ***name***] [**from** ***expr***] [**by** ***expr***] [**to** ***expr***] [**while** ***expr***]
do ***statseq*** **od**

statseq is a sequence of statements separated by semi-colons,
[ ] denotes an optional part.
Thus each of the *for*, *from*, *by*, *to*, *while* clauses can be omitted.
If omitted, the default values are a *dummy* variable, 1,1 *infinity* and *true* respectively.

Special cases

For loop:

```
> for i from 1 to 3 do print(i^2) od:
```

While loop:

One omits the *for*, *from*, *by* and *to* clauses.

```
> i := 10^10+1;
  while not isprime(i) do i := i+2:  od:  # 10 000 000 019 is prime
  print(i);
```

$$i := 10000000001$$
$$10000000019$$

Combining *for* and *while* loops:

```
> for i from 10^10+1 by 2 while not isprime(i) do od;
  print(i);
```

$$10000000019$$

(2) For-in loop

[**for** ***name***] [**in** ***expr***] [**while** ***expr***]
do ***statseq*** **od**

statseq is a sequence of statements separated by semi-colons,
[ ] denotes an optional part.

```
> s := [1,2,3]; # a list
  f := x->x^2; # a function
```

$$s := [1, 2, 3]$$
$$f := x \to x^2$$

```
> nops(s);
  for i to nops(s) do f(op(i,s)) od;
```

$$3$$
$$1$$
$$4$$
$$9$$

equivalent to the statement

```
> for i in s do f(i) od;
```

$$1$$
$$4$$
$$9$$

2 Data Structures

More complicated programs involve manipulating and storing data.
Maple has a good set of data structures: **sequences, lists (or vectors), sets, tables** (or hash tables), and **arrays**.
Maple does not have records or linked lists, but these can be implemented otherwise.

2.1 Sequences

A sequence is a sequence of expressions separated by commas.

```
> restart;
  s := 1,4,9,16,25;
```

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

```
> t := sin,cos,tan;
```

$$t := \sin, \cos, \tan$$

A sequence of sequences simplifies into one sequence, that is, sequences are *associative*.

```
> s := 1,(4,9,16),25;
  s,s;
```

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

The special symbol **NULL** is used for the **empty sequence**.

```
> es := NULL;
```

$$es :=$$

Sequences are used for many purposes. The next sections show how lists and sets are constructed from sequences.
Here we note that function calls are really constructed from sequences.

Some functions constructed from sequences

```
> max(s);
  min(s,0,s);
```

$$25$$
$$0$$

The *min* and *max* functions take an arbitrary number of values as arguments, i.e. a sequence of arguments.
The *op* and *nops* functions cannot be applied to sequences. This is because the sequence itself becomes the arguments to a function, which result in an error.

```
> op(s);
  nops(s);

Error, wrong number (or type) of parameters in function op
Error, wrong number (or type) of parameters in function nops
```

Put a sequence into a **list** first if you want to use *op* or *nops*.

Sequence operator $

This generates sequences.

```
> x$5;
  y^2 $ y=1..3;
```

$$x, x, x, x, x$$
$$1, 4, 9$$

Index oprator [ ]

One can also use the subscript notation to access the i'th element or more elements.

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

$$1$$
$$1, 4, 9$$

Calculation results in Maple are often given as sequences.
So it's convenient to choose one or more parts of the result by indexed variable.

```
> res := solve(x^2-1=0,x);
  res[1];
  res[1..2];
```

$$res := 1, -1$$
$$1$$
$$1, -1$$

Creating sequences by the function seq

Two kinds of **for** loops.

(1) seq(f(i), i=m..n), where f is a formula or function.

The *seq* works is just as if you had programmed the following loop.

s := NULL;
for i from m by 1 to n do s := s, f(i) od;

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

$$1, 4, 9, 16, 25$$

```
> f := x->x^3;
  seq(f(j), j=1..3 );
```

$$f := x \to x^3$$
$$1, 8, 27$$

```
> s := NULL;
  for i from 1 to 5 do s := s, i^2 od;
```

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

Note that *seq* is more efficient than the *for* loop because it does not create the intermediate sequences.

(2) seq(f(i), i=a), where f is i.e. a formula, function or operator and a is i.e. a list, set or polynomial.

This is equivalent to

seq(f(op(i,a)), i=1..nops(a))

```
> s1 := [1,2,3]; # a list
  seq(2*k, k=s1);
```

$$s1 := [1, 2, 3]$$
$$2, 4, 6$$

```
> D(sin); # derivative operator
  g := x->x^2;
  D(g);
  D(g)(x);
  seq(D(f), f=[x,g,sin,cos,tan,exp,ln]);
```

$$\cos$$
$$g := x \to x^2$$
$$x \to 2\,x$$
$$2\,x$$
$$\mathrm{D}(x),\ x \to 2\,x,\ \cos,\ -\sin,\ 1 + \tan^2,\ \exp,\ a \to \frac{1}{a}$$

```
> a := 3*x^3+y*x-11; # polynomial in x,y
  degree(a,x); # degree of polynomial a(x)
  degree(a,y);
  coeff(a,x,0); # absolute term of polynomial a(x)
  seq(coeff(a,x,i), i=0..degree(a,x));
```

$$a := 3\,x^3 + y\,x - 11$$
$$3$$
$$1$$
$$-11$$
$$-11,\ y,\ 0,\ 3$$

2.2 Lists or Vectors

Lists are constructed from sequences.
A list is a data structure for collecting objects together.
Square brackets are used to create lists.

```
> restart;
  l := [x,1,1-z,x];
  whattype(l);
```

$$l := [x, 1, 1 - z, x]$$
$$\mathit{list}$$

The **empty list** is denoted by [ ].

```
> el := [];
```

$$el := []$$

Functions for list

(1) nops, op function

$nops(l)$ returns the number of elements and $op(i, l)$ extracts the i'th element. One can also use the subscript notation to access the i'th element. Choosing two or more element by $l[m..n]$ one obtains a list.

```
> nops(l);
  op(1,l);
  l[1]; # a element
  op(1..3,l);
  l[1..3]; # a list of 3 elements
```

$$4$$
$$x$$
$$x$$
$$x,\ 1,\ 1-z$$
$$[x,\ 1,\ 1-z]$$

$op(l)$ transforms a list to a sequence. You can assign to a component of a list.

```
> seq1 := op(l);
  l[1] := d;
  op(l);
```

$$seq1 := x,\ 1,\ 1-z,\ x$$
$$l_1 := d$$
$$d,\ 1,\ 1-z,\ x$$

$convert(l, set)$ transforms list to a set.

```
> convert(l,set);
```

$$\{x,\ d,\ 1,\ 1-z\}$$

(2) Connection of two or more lists by op and [ ]

```
> ll := [u,10,11];
  bigl := [op(l),op(ll)];
```

$$ll := [u,\ 10,\ 11]$$
$$bigl := [d,\ 1,\ 1-z,\ x,\ u,\ 10,\ 11]$$

(3) member function

$member(x, s)$ returns true if the element x is in the list l.

```
> member(1,l);
```

$$true$$

A loop that prints true if a list l contains the element x, and false otherwise.

```
> v:=1;
  for i to nops(l) while l[i] <> v do od;
  if i>nops(l) then print(false) else print(true) fi;
```

$$v := 1$$
$$true$$

(4) Appending a new element

One can append a new element x to a list l by doing

```
> y := z;
  l;
  l1 := [op(l),y];
  l2 := [y,op(l)];
```

$$y := z$$
$$[d,\ 1,\ 1-z,\ x]$$
$$l1 := [d,\ 1,\ 1-z,\ x,\ z]$$
$$l2 := [z,\ d,\ 1,\ 1-z,\ x]$$

(5) Removing a element

One can remove the i'th element of a list l by doing

```
> i := 2;
  l31 := [op(l[1..i-1]), op(l[i+1..nops(l)]) ];
  l32 := [ l[1..i-1], l[i+1..nops(l)] ]; # a list of list
```

$$i := 2$$
$$l31 := [d,\ 1-z,\ x]$$
$$l32 := [[d],\ [1-z,\ x]]$$

Or, better, use the **subsop** function.

```
> i := 2;
  l4 := subsop(i=NULL,l); # remove the 2'nd element
```

$$i := 2$$
$$l4 := [d,\ 1-z,\ x]$$

The *subsop* function note can be used for any type of expression.

2.3 Sets

Sets are constructed from sequences.
A set is a data structure for collecting unordered objects together.
The difference between lists and sets is that duplicates are removed from sets.
Squiggly brackets are used to create sets.

```
> restart;
  s := {x,1,1-z,x};
  whattype(s);
```

$$s := \{1,\ x,\ 1-z\}$$
$$set$$

The **empty set** is denoted by **{ }**.

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

$$es := \{\}$$

Functions for set

(1) nops, op function

$nops(s)$ returns the number of elements and $op(i,s)$ extracts the i'th element. One can also use the subscript notation to access the i'th element.

```
> nops(s);
  op(1,s);
  s[1]; # a element
  op(1..3,s);
  s[1..2]; # a subset
```

$$3$$
$$1$$
$$1$$
$$1,\ x,\ 1-z$$
$$\{1,\ x\}$$

$op(s)$ transforms a set to a sequence. You can't assign to a component of a set.

```
> seq1 := op(s);
  s[1] := d;
```

$$seq1 := 1,\ x,\ 1-z$$

```
Error, cannot assign to a set
```

(2) Connection or Union of two or more sets by op and { }

```
> ss := {u,10,11};
  bigs := {op(s),op(ss)}; # no order
```

$$ss := \{10,\ 11,\ u\}$$
$$bigs := \{1,\ 10,\ 11,\ x,\ u,\ 1-z\}$$

(3) member function

$member(x,s)$ returns true if the element x is in the set s.

```
> member(1,s);
```

$$true$$

A loop that prints true if a set s contains the element x, and false otherwise.

```
> x1:=1;
  for i to nops(s) while s[i] <> x1 do od;
  if i>nops(s) then print(false) else print(true) fi;
```

$$x1 := 1$$
$$true$$

(4) Appending a new element (Union)

One can append a new element x to a set s by doing

```
> y := z;
  s;
  s1 := {op(s),y};
  s2 := {y,op(s)};
  if s1 = s2 then true fi;
  evalb(s1=s2); # boolean calculation of set relation
  s3 := s union {y}; # set operator
```

$$y := z$$
$$\{1,\ x,\ 1-z\}$$
$$s1 := \{1,\ x,\ z,\ 1-z\}$$
$$s2 := \{1,\ x,\ z,\ 1-z\}$$

$$true$$
$$true$$
$$s3 := \{1, x, z, 1-z\}$$

(5) Removing a element

One can remove the i'th element of a set s by doing

```
>  i := 2;
   s31 := { op(s[1..i-1]), op(s[i+1..nops(s)]) };
   s32 := { s[1..i-1], s[i+1..nops(s)] }; # set of set
```

$$i := 2$$
$$s31 := \{1, 1-z\}$$
$$s32 := \{\{1\}, \{1-z\}\}$$

Or use the **subsop** function.

```
>  i := 2;
   s4 := subsop(i=NULL,s);
```

$$i := 2$$
$$s4 := \{1, 1-z\}$$

(6) Set operators union,intersect, minus

```
>  s := {x,1,1-z};
   t := {u,x,z};
   r := {u,10,11};
   s union t;
   s intersect t;
   s minus t;
   s intersect r;
   s minus r;
```

$$s := \{1, x, 1-z\}$$
$$t := \{x, z, u\}$$
$$r := \{10, 11, u\}$$
$$\{1, x, z, u, 1-z\}$$
$$\{x\}$$
$$\{1, 1-z\}$$
$$\{\}$$
$$\{1, x, 1-z\}$$

Maple orders the elements of sets in a strange order which seems to be unpredictable. The algorithms that Maple uses to remove duplicates, and to do set union, intersection and difference all work by first sorting the elements of the input sets. Maple sorts by *machine address*, i.e. in the order that the elements occur in computer memory.

2.4 Tables

Tables or hash tables are extremely useful for writing efficient programs.
A table is a one-to-many relation between two discrete sets of data.

```
>  Table[1] := 1,2;
   Table[2] := alpha, beta, gamma;
   Table[3] := NULL; # no entry, assigned an empty entry
```

$$Table_1 := 1, 2$$
$$Table_2 := \alpha, \beta, \gamma$$
$$Table_3 :=$$

The domain and range values of table definition can be sequences of zero or more values. The domain values in the table are called the *keys* or *indices*. The range values in the table are called the *values* or *entries*.

Function for tables

The **indices** function returns a sequence of them (a sequence of lists). The **entries** function returns a sequence of them.

```
> indices(Table);
  entries(Table);
```

$$[1], [2], [3]$$
$$[1, 2], [\alpha, \beta, \gamma], []$$

A table **T_COLOUR** of colour translations for English to other languages.
The domain of the table is the name of the colour in English. The range of the table is a sequence of the names of the colours in French, German and Polish.

```
> T_COLOUR[red] := rouge,rot,cerwony;
  T_COLOUR[blue] := bleu,blau,niebieski;
  T_COLOUR[yellow] := jaune,gelb,zolty;
```

$$T_COLOUR_{red} := rouge,\ rot,\ cerwony$$
$$T_COLOUR_{blue} := bleu,\ blau,\ niebieski$$
$$T_COLOUR_{yellow} := jaune,\ gelb,\ zolty$$

```
> indices(T_COLOUR);
  entries(T_COLOUR);
```

$$[red], [blue], [yellow]$$
$$[rouge, rot, cerwony], [bleu, blau, niebieski], [jaune, gelb, zolty]$$

Note, the order that the indices and entries functions return the table indices and entries is not necessarily the same order as the order in which they were entered into the table. This is because Maple makes use of **hashing** to make table searching very fast and as a consequence, the order in which the entries were made is lost. However, there is a one to one correspondence between the output of the indices and the entries functions.

Applications of tables

(1) Quick look up the corresponding table entry by given key.

```
> T_COLOUR[red];
```

$$rouge,\ rot,\ cerwony$$

How quickly? Approximately in constant time, no matter how large the table is.
Even if our table contains 1000 different colours, the time to search the table will be very fast.

(2) Test if an entry is in the table using the assigned function.

```
> assigned(T_COLOUR[blue]);
  assigned(Table[3]);
```

$$true$$
$$true$$

(3) Remove entries by unassignment.

```
> Table[3] := 'Table[3]'; # no Table[3]
  assigned(Table[3]);
```

$$Table_3 := Table_3$$

$$false$$

(4) Output of tables

```
> print(Table);
  print(T_COLOUR);
```

$$\mathrm{table}([\; 1 = (1,\, 2),\; 2 = (\alpha,\, \beta,\, \gamma) \;])$$

$$\mathrm{table}([\; red = (rouge,\, rot,\, cerwony),\; blue = (bleu,\, blau,\, niebieski),\; yellow = (jaune,\, gelb,\, zolty) \;])$$

2.5 Arrays

One- or more-dimensional arrays are created by the **array** command.
Matrices and vectors are special arrays.
The index is restricted to be in a fixed integer range. Arrays are more efficient to access than tables and range checking on the indices is done. An array can be initialized with a list of entries or without entries.

(1) One-dimensional array

array(m..n), array(m..n, list)

This creates an array with indices $m,\ m+1, \ldots,\ n$. Often m is 1.
Entries can be inserted into the array by initializing or assignment as for tables.

```
> restart;
  with(linalg): # load of the linalg packages
  A1 := array(0..3);
  A2 := array(0..3,[2,4,6,8]);

Warning, new definition for norm
Warning, new definition for trace
```

$$A1 := \mathrm{array}(0..3,\, [])$$

$$A2 := \mathrm{array}(0..3,\, [\; (0) = 2,\; (1) = 4,\; (2) = 6,\; (3) = 8 \;])$$

```
> A3 := array(1..3,[2,4,6]);
  A4 := vector(3);
  A4[1] := 1;
  A4;
  A5 := vector(3,[4,5,6]);
  A6 := vector([6,7,8]);
```

$$A3 := [2,\ 4,\ 6]$$
$$A4 := \mathrm{array}(1..3,\ [\,])$$
$$A4_1 := 1$$
$$A4$$
$$A5 := [4,\ 5,\ 6]$$
$$A6 := [6,\ 7,\ 8]$$

```
> print(A2);
  print(A4);
  print(A5);
```

$$\begin{array}{l} \mathrm{array}(0..3,\ [\\ \quad (0) = 2 \\ \quad (1) = 4 \\ \quad (2) = 6 \\ \quad (3) = 8 \\ \quad]) \end{array}$$
$$[1,\ A4_2,\ A4_3]$$
$$[4,\ 5,\ 6]$$

```
> n := 8;
  v := array(1..n);
  v[1] := a;
  for i from 2 to n do v[i] := a*v[i-1] mod n od;
```

$$n := 8$$
$$v := \mathrm{array}(1..8,\ [\,])$$
$$v_1 := a$$
$$v_2 := a^2$$
$$v_3 := a^3$$
$$v_4 := a^4$$
$$v_5 := a^5$$
$$v_6 := a^6$$
$$v_7 := a^7$$
$$v_8 := a^8$$

```
> a := 3;
  v[1]; v[2];
  print(v);
```

$$a := 3$$
$$3$$
$$9$$
$$[a,\ a^2,\ a^3,\ a^4,\ a^5,\ a^6,\ a^7,\ a^8]$$

Applications

(1.1) Using a one-dimensional array v to sort a sequence of numbers.

The code presented here sorts v into ascending order using the *bubblesort* algorithm.

```
> n := 10;
  v := array(1..n);
  for i to n do v[i] := n-i od:
  `old:` ; seq(v[i],i=1..n);
  for i to n-1 do
    for j from i+1 to n do
       if v[i] > v[j] then temp := v[i]; v[i] := v[j]; v[j] := temp fi
    od
  od;
  `new:`; seq(v[i],i=1..n);
```

$$n := 10$$
$$v := \mathrm{array}(1..10, [])$$
$$old :$$
$$9, 8, 7, 6, 5, 4, 3, 2, 1, 0$$
$$new :$$
$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$$

(1.2) Intermediate data structure.

Suppose we represent a polynomial $A(x) = a_0 + a_1 x + ... a_m x^m$ as a list $[a_0, ..., a_m]$ of coefficients.

Suppose we are given also a polynomial $B(x)$ of degree n.

We can use an array c to compute the product $C = A * B$ as follows.

```
> A := 1+2*x+3*x^2;
  B := 1-2*x-3*x^2-4*x^3;
  m := degree(A,x);
  n := degree(B,x);
  la := [seq(coeff(A,x,i),i=0..m)];
  lb := [seq(coeff(B,x,i),i=0..n)];
  a := array(0..m,la);
  b := array(0..n,lb);
```

$$A := 1 + 2\,x + 3\,x^2$$
$$B := 1 - 2\,x - 3\,x^2 - 4\,x^3$$
$$m := 2$$
$$n := 3$$
$$la := [1, 2, 3]$$
$$lb := [1, -2, -3, -4]$$

$$a := \mathrm{array}(0..2, [$$
$$(0) = 1$$
$$(1) = 2$$
$$(2) = 3$$
$$])$$

$$b := \text{array}(0..3, [\\ (0) = 1\\ (1) = -2\\ (2) = -3\\ (3) = -4\\ ])$$

```
> m := nops(la)-1; # degree of A
  n := nops(lb)-1; # degree of B
```

$$m := 2$$
$$n := 3$$

(a) Variant 1

```
> c := array(0..m+n); # allocate storage for C
  for k from 0 to m+n do c[k] := 0; od:
   for i from 0 to m do
     for j from 0 to n do
       c[i+j] := c[i+j] + a[i]*b[j];
         # c[i+j] := c[i+j] + la[i+1]*lb[j+1];
     od:
   od:
  `Coefficients c[0..m+n] of product`;
  [seq(c[k],k=0..n+m)]; # put the product in a list
```

$$c := \text{array}(0..5, [])$$

$$\textit{Coefficients } c[0..m+n] \textit{ of product}$$

$$[1, 0, -4, -16, -17, -12]$$

(b) Variant 2

Complexity: The same flops(*) number like in variant 1, but less index manipulation.

```
> c := array(0..m+n); # allocate storage for C
  for k from 0 to m+n do
   hc := 0;
   for i from max(0,k-n) to min(m,k) do
     hc := hc + a[i]*b[k-i];
   od:
   c[k] := hc;
  od:
  `Coefficients c[0..m+n] of product`;
  [seq(c[k],k=0..n+m)]; # put the product in a list
```

$$c := \text{array}(0..5, [])$$

$$\textit{Coefficients } c[0..m+n] \textit{ of product}$$

$$[1, 0, -4, -16, -17, -12]$$

(2) Two-dimensional array

array(c..d, m..n), array(c..d, m..n, list)

This creates an array with row indices $c,\ c+1, ...,\ d$ and column indices $m,\ m+1, ...,\ n$ (matrix). Often $c,\ m$ is 1.
Higher dimensional arrays work similarly.
Entries can be inserted into the array by initializing or assignment as for tables.

```
> restart;
  with(linalg):  # load of the linalg packages
  A1 := array(0..3,0..1);
  A2 := array(0..3,0..1,[[2,4],[6,8],[9,10],[22,33]]);

Warning, new definition for norm
Warning, new definition for trace
```

$$A1 := \mathrm{array}(0..3,\ 0..1,\ [])$$

$$A2 := \mathrm{array}(0..3,\ 0..1,\ [$$
$$(0,\ 0) = 2$$
$$(0,\ 1) = 4$$
$$(1,\ 0) = 6$$
$$(1,\ 1) = 8$$
$$(2,\ 0) = 9$$
$$(2,\ 1) = 10$$
$$(3,\ 0) = 22$$
$$(3,\ 1) = 33$$
$$])$$

```
> A3 := array([[2,4,6],[1,2,3]]);
  A4 := matrix(2,3);
  A4[1,1] := 1;
  A4;
  A5 := matrix(2,2,[[4],[5,6]]);
  A6 := matrix([[6,7],[8,9]]);
```

$$A3 := \begin{bmatrix} 2 & 4 & 6 \\ 1 & 2 & 3 \end{bmatrix}$$

$$A4 := \mathrm{array}(1..2,\ 1..3,\ [])$$

$$A4_{1,1} := 1$$

$$A4$$

$$A5 := \begin{bmatrix} 4 & A5_{1,2} \\ 5 & 6 \end{bmatrix}$$

$$A6 := \begin{bmatrix} 6 & 7 \\ 8 & 9 \end{bmatrix}$$

```
> print(A2); # like eval(A2) or evalm(A2)
  print(A4);
  print(A5);
  print(A6);
```

$$\begin{aligned}&\mathrm{array}(0..3,\ 0..1,\ [\\&\quad(0,\,0)=2\\&\quad(0,\,1)=4\\&\quad(1,\,0)=6\\&\quad(1,\,1)=8\\&\quad(2,\,0)=9\\&\quad(2,\,1)=10\\&\quad(3,\,0)=22\\&\quad(3,\,1)=33\\&\quad])\end{aligned}$$

$$\begin{bmatrix} 1 & A4_{1,2} & A4_{1,3} \\ A4_{2,1} & A4_{2,2} & A4_{2,3} \end{bmatrix}$$

$$\begin{bmatrix} 4 & A5_{1,2} \\ 5 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 6 & 7 \\ 8 & 9 \end{bmatrix}$$

Applications

(2.1) Exchange the i'th and j'th rows of a matrix.

```
> restart;
  with(linalg):  # load of the linalg packages
  m := 3; n := 4;
  A := matrix(m,n);
  for i to m do
   for j to n do A[i,j] := 1/(i+j-1); # Hilbert matrix
  od; od;
  eval(A);
```

```
Warning, new definition for norm
Warning, new definition for trace
```

$$m := 3$$

$$n := 4$$

$$A := \mathrm{array}(1..3,\ 1..4,\ [])$$

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \end{bmatrix}$$

```
> i := 2; j := 3;
  if i<>j then
    for k to n do
      temp := A[i,k]; A[i,k] := A[j,k]; A[j,k] := temp;
    od;
  fi;
  print(A);
```

$$i := 2$$
$$j := 3$$
$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

(2.2) Derivative of a multivariate vector function.

There are given the vector v of symmetric polynomials in 3 variables $x_1,\ x_2,\ x_3$.

```
> v := array(1..4);
  v[1] := 1;
  v[2] := x[1] + x[2] + x[3];
  v[3] := x[1]*x[2] + x[1]*x[3] + x[2]*x[3];
  v[4] := x[1]*x[2]*x[3];
```

$$v := \mathrm{array}(1..4, [])$$
$$v_1 := 1$$
$$v_2 := x_1 + x_2 + x_3$$
$$v_3 := x_1\,x_2 + x_1\,x_3 + x_2\,x_3$$
$$v_4 := x_1\,x_2\,x_3$$

Let us construct a two-dimensional array or matrix J, where $J_{i,j}$ is the derivative of v_i wrt x_j. J is the so called **Jacobian**.

```
> J := array(1]..4,1..3):  # matrix(4,3)
  for i to 4 do
    for j to 3 do J[i,j] := diff(v[i],x[j])
  od od:
  J;
  J := eval(J);
  evalm(J); # evaluate a matrix expression
  print(J): # the same output like evalm(J)
```

$$J$$
$$J := \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ x_2 + x_3 & x_1 + x_3 & x_1 + x_2 \\ x_2\,x_3 & x_1\,x_3 & x_1\,x_2 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ x_2+x_3 & x_1+x_3 & x_1+x_2 \\ x_2\,x_3 & x_1\,x_3 & x_1\,x_2 \end{bmatrix}$$

Note that the value of J is just the name of the array J, because of often unassigned entries.
Evaluation rules for arrays and tables are special.
Whenever you want to print an array or table or return an array or table from a procedure, use a variant of evaluation functions like *eval*, *evalm*, *evalf*,... or the *print* command.

Traditional mathematical notation

```
> J: %=evalm(%);
```

$$J = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ x_2+x_3 & x_1+x_3 & x_1+x_2 \\ x_2\,x_3 & x_1\,x_3 & x_1\,x_2 \end{bmatrix}$$

2.6 Records

Maple doesn't explicitly have a record data structure like Pascal's *record* or C's *struct*.
By a record data structure we mean a data structure for keeping together a heterogeneous collection of objects, i.e. a set of objects not necessarily of the same type.
There are several possibilities for representing records in Maple.

<u>**Illustration**</u>

(1) A record data structure would be in choosing a data structure to represent a quarternion.

A quarternion is a number of the form $a+b\boldsymbol{i}+c\boldsymbol{j}+d\boldsymbol{k}$, where a, b, c, d are real numbers.
To represent a quarternion, we need to store only the four quantities a, b, c, d.

(1.1) Representing record as list.

The simplest, and most obvious, is to use a list. I.e. we would represent the quarternion $a+b\boldsymbol{i}+c\boldsymbol{j}+d\boldsymbol{k}$ as the list $[a, b, c, d]$, we have many possibilities to operate with them.

```
> restart;
  lst:= [a,b,c,d];
```

$$lst := [a,\, b,\, c,\, d]$$

(1.2) Use a function call.

We could represent $a+b\boldsymbol{i}+c\boldsymbol{j}+d\boldsymbol{k}$ as **QUARTERNION(a,b,c,d)**.
An advantage of this representation is to do various operations on functions.

```
> QUARTERNION(2,3,0,1);
```

$$\mathrm{QUARTERNION}(2,\, 3,\, 0,\, 1)$$

Further, we shall mention that you can define how to *pretty print* a function.

```
>   `print/QUARTERNION` := proc(a,b,c,d)
        a + b*'i' + c*'j' + d*'k'
    end:
    QUARTERNION(2,3,0,1);
```

$$2 + 3\,i + k$$

Here we have defined a printing procedure or subroutine.
This routine is called once for each different **QUARTERNION** function in a result from Maple prior to displaying the result.
Note the use of quotes in the procedure, because we want to see the identifiers ***i,j,k*** in the output, and not the value of the variables i, j, k, which we might be using for something else.

(1.3) Representing a record as a multivariate polynomial.

Here we can store the fields names as values in the coefficients.
This is quite useful when the fields are numerical and you wish to be able to do arithmetic on the fields. So we could represent a quarternion as a polynomial in the variables i, j, k.

```
>   z1 := 2 + 3*i + 0*j + 1*k;
```

$$z1 := 2 + 3\,i + k$$

```
>   z2 := 2 - 3*i + 2*j + 2*k;
```

$$z2 := 2 - 3\,i + 2\,j + 2\,k$$

```
>   coeff(z1,i); # the coefficient of i
    coeff(z1,i,0); # the abs.  term in z1(i)
    degree(z2,j);
    z1+z2;
```

$$3$$
$$2 + k$$
$$1$$
$$4 + 3\,k + 2\,j$$

(2) Use a data structure to represent the factorization of a polynomial.

Let us consider the real polynomial $a(x) = a_0 + a_1 x + ... + a_n x^n$ of degree n with real roots in $Q[x]$.
The factorization of $a(x)$ looks like $a(x) = c * f1^{e1} * ... * fn^{en}$,
where each of the factors fi in $Q[x]$ is monic and irreducible.
For example

```
>   a := x->2*x^6+18*x^5+48*x^4+4*x^3-126*x^2-54*x+108;
    factor(a(x));
```

$$a := x \to 2\,x^6 + 18\,x^5 + 48\,x^4 + 4\,x^3 - 126\,x^2 - 54\,x + 108$$
$$2\,(x+2)\,(x-1)^2\,(x+3)^3$$

We need to store the factors fi, the exponents ei and the unit c.
We represent the factorization of $a(x)$ as the list $[c, f]$, where f is a list of lists of the form $[fi, ei]$.
We use subscripts to refer to a component of the data structure, e.g. the unit part of a factorization would be given by $la[1]$.
You can use the **macro facility** to define an identifier to be equal to a constant, if you prefer to use a symbol to reference a component as shown in the following example.

```
> la := [-1/2,[[x+1,2],[x-1,1]]];
```

$$la := [\frac{-1}{2}, [[x+1, 2], [x-1, 1]]]$$

```
> macro(unit=1,factors=2,base=1,exponent=2);
> la[unit];
```

$$\frac{-1}{2}$$

```
> la[factors][1][base];
  la[factors][1][exponent];
```

$$x+1$$
$$2$$

```
> a := la[unit]*la[factors][1][base]^la[factors][1][exponent]
      *la[2][2][1]^la[2][2][2];
```

$$a := -\frac{1}{2}(x+1)^2(x-1)$$

```
> p := x->a;
  p(x);
  p(0);
  eval(a,x=1);
  eval(p(x),x=0);
```

$$p := x \to a$$
$$-\frac{1}{2}(x+1)^2(x-1)$$
$$-\frac{1}{2}(x+1)^2(x-1)$$
$$0$$
$$\frac{1}{2}$$

2.7 Linked Lists

Maple lists are not linked lists. Maple lists are really arrays of pointers to their entries.
In lists, tables or arrays you can assign to their component.
Linked lists are recursive data structures.
The difference can be seen by studying an example.

Consider representing a real polynomial $a(x) = a_0 + a_1x + ... + a_nx^n$ of degree n. One just known possibility would be to store the coefficients $a_0, ..., a_n$ in a list.
For instance, we could represent the polynomial $p = x^4 + 3x^2 + 2x + 11$ as follows.

```
> lp := [11,2,3,0,1];
  d := nops(lp)-1; # degree
```

$$lp := [11, 2, 3, 0, 1]$$
$$d := 4$$

Alternatively we could use a linked list.

```
> li := [ 1, [ 0, [ 3, [ 2, [ 11, NIL ]]]]];
```

$$li := [1, [0, [3, [2, [11, NIL]]]]]$$

We see that the linked list is a recursive data structure.
It is either list of two values, traditionally called the CAR and CDR-terminology from the LISP programming language, or it is the special value **NIL** signifying the **empty linked list**.

- The first field contains a data value, in our case, a coefficient.
- The second field is a pointer to another linked list which contains the rest of the data values.

Note that in order to compute the degree d of a polynomial represented by a linked list, we have to compute the **depth** of the linked list.

```
> p := li;
  for d from -1 while p <> NIL do p := p[2] od;
  d;
```

$$p := [1, [0, [3, [2, [11, NIL]]]]]$$
$$p := [0, [3, [2, [11, NIL]]]]$$
$$p := [3, [2, [11, NIL]]]$$
$$p := [2, [11, NIL]]$$
$$p := [11, NIL]$$
$$p := NIL$$
$$4$$

Applications

(1) Putting a new entry onto the front of a linked list.

Suppose we wanted to add the term $5x^5$ to our polynomial p.
In list representation we must create a new list with 6 entries.

```
> lp := [op(lp),5];
```

$$lp := [11, 2, 3, 0, 1, 5]$$

This requires at least 6 words of storage for the new list. This takes constant time and uses no storage but now, we have sequence (11, 2, 3, 0, 1), 5 which results in the new larger sequence 11, 2, 3, 0, 1, 5. This is where 6 words of storage get allocated.
In general, adding $a_{n+1}x^{n+1}$ to a polynomial of degree n takes $\mathcal{O}(n)$ time and storage.

But what about the linked list?

```
> p := li;
  p := [5,p];
```

$$p := [1, [0, [3, [2, [11, NIL]]]]]$$
$$p := [5, [1, [0, [3, [2, [11, NIL]]]]]]$$

Here only needs to create a new list of length two, hence constant storage.
This takes $\mathcal{O}(1)$ time for linked lists.

(2) Putting a new entry at the end.

Suppose we wanted to add the term $12x^{-1}$ to our new polynomial p.
In list representation we must create a new list with 7 entries.

```
> lp := [12,op(lp)];
```

$$lp := [12, 11, 2, 3, 0, 1, 5]$$

But what about the linked list?

```
> p := [5, [ 1, [ 0, [ 3, [ 2, [ 11, [12, NIL]]]]]]];
```

$$p := [5, [1, [0, [3, [2, [11, [12, \mathit{NIL}]]]]]]]$$

3 Procedures

3.1 Parameters, Local Variables, RETURN, ERROR

(1) Procedure Syntax and Parameters

A procedure has the following syntax

> **proc** (*nameseq*)
> [**local** *nameseq* ;]
> [**global** *nameseq* ;]
> [**options** *nameseq* ;]
> *statseq*
> **end**

where *nameseq* is a sequence of symbols separated by commas,
and *statseq* is a sequence of statements separated by semicolons.

```
> restart;
  proc(x,y) x^2 + y^2 end;
```

$$\mathbf{proc}(x,\, y)\, x^2 + y^2\, \mathbf{end}$$

This procedure has two formal parameters x and y.
It has no local variables, no global variables, no options, and only one statement.
The value returned by the procedure is $x^2 + y^2$.
The procedure has no name, it is **anonyme**.

Procedure named $f(x, y, z)$ with 3 formal parameters and a local parameter.

```
> f := proc(x,y,z)
    local h;
    h := sin(x*y);
    h^2 + h*z-1
  end;
  f(1,2,3); # unevaluated formula
  value(f(1,2,3));
  evalf(f(1,2,3)); # 10 Digits standard
  f(a,b,c);
  u := f(a,b,c);
```

$$f := \mathbf{proc}(x,\, y,\, z)\, \mathbf{local}\, h;\; h := \sin(x \times y)\,;\; h^2 + h \times z - 1\, \mathbf{end}$$

$$\sin(2)^2 + 3\sin(2) - 1$$

$$\sin(2)^2 + 3\sin(2) - 1$$

$$2.554714090$$

$$\sin(a\, b)^2 + \sin(a\, b)\, c - 1$$

$$u := \sin(a\, b)^2 + \sin(a\, b)\, c - 1$$

Variables that appear in a procedure which are neither parameters nor locals and are not explicitly declared global are given a **default implicite declaration** local if either the following is true.

- A variable or suscripted variable on the left hand side of an assignment.
- A variable or suscripted variable in a *for* or *seq* loop index.

Part of the reason for this is to catch likely warnings or errors.

```
> proc(x,n) s := 1; for i to n do s := s+x^i od; s end;

Warning, `s` is implicitly declared local
Warning, `i` is implicitly declared local
```

$$\mathbf{proc}(x,\,n)\,\mathbf{local}\,s,\,i;\;s:=1\,;\;\mathbf{for}\,i\,\mathbf{to}\,n\,\mathbf{do}\,s:=s+x^i\,\mathbf{od}\,;\;s\,\mathbf{end}$$

```
> proc(x,n) local s,i;
    s := 1; for i to n do s := s+x^i od; s
  end;
```

$$\mathbf{proc}(x,\,n)\,\mathbf{local}\,s,\,i;\;s:=1\,;\;\mathbf{for}\,i\,\mathbf{to}\,n\,\mathbf{do}\,s:=s+x^i\,\mathbf{od}\,;\;s\,\mathbf{end}$$

```
> proc(x) local s,i;
    s := t; for i to n do s := s+x^i od; s
  end;
    # t on the rigth hand side is not implicitly declared
    # n as bound in loop is not implicitly declared
```

$$\mathbf{proc}(x)\,\mathbf{local}\,s,\,i;\;s:=t\,;\;\mathbf{for}\,i\,\mathbf{to}\,n\,\mathbf{do}\,s:=s+x^i\,\mathbf{od}\,;\;s\,\mathbf{end}$$

Notice in all cases that Maple has declared s and i to be local.

If you want variables to really be global, you should declare them explicitly using the global statement.

Local variable s

```
> restart;
  s;
  p1 := proc(x,n) local s,i;
    s := 1; for i to n do s := s+x^i od; s
  end;
  p1(x,2);
  s;
```

$$s$$

$$p1 := \mathbf{proc}(x,\,n)\,\mathbf{local}\,s,\,i;\;s:=1\,;\;\mathbf{for}\,i\,\mathbf{to}\,n\,\mathbf{do}\,s:=s+x^i\,\mathbf{od}\,;\;s\,\mathbf{end}$$

$$1+x+x^2$$

$$s$$

Global variable s

```
> restart;
  s;
  p2 := proc(x,n) local i; global s;
    s := 1; for i to n do s := s+x^i od; s
  end;
  p2(x,2);
  s;
  x := 2;
  s;
```

$$s$$

$$p2 := \mathbf{proc}(x, n)\, \mathbf{local}\, i;\ \mathbf{global}\, s;\ s := 1\,;\ \mathbf{for}\, i\, \mathbf{to}\, n\, \mathbf{do}\, s := s + x^i\, \mathbf{od}\,;\ s\, \mathbf{end}$$

$$1 + x + x^2$$

$$1 + x + x^2$$

$$x := 2$$

$$7$$

(2) RETURN

In general the value returned by a procedure is the last value computed unless there is an explicit return statement using **RETURN**.

MEMBER1(x, a) returns true, if x is in the list a, false otherwise (dont confuse with the built in *member* function).

```
> MEMBER1 := proc(x,a)
    local v;
    for v in a do
      if v = x then RETURN(true) fi od;
    false
  end;
```

MEMBER1 :=

$$\mathbf{proc}(x, a)\, \mathbf{local}\, v;\ \mathbf{for}\, v\, \mathbf{in}\, a\, \mathbf{do}\, \mathbf{if}\, v = x\, \mathbf{then}\, \mathrm{RETURN}(\mathit{true})\, \mathbf{fi}\ \mathbf{od}\,;\ \mathit{false}\, \mathbf{end}$$

```
> L := [i,j,k];
  MEMBER1(i,L);
  MEMBER1(l,L);
```

$$L := [i, j, k]$$

$$\mathit{true}$$

$$\mathit{false}$$

The MEMBER1 procedure has a local variable v so that it does not interfere with the global user variable named v.

(3) ERROR

The **ERROR** function can be used to generate an error message from within a procedure. For example, the MEMBER2 routine should check that the argument really is a list and issue an appropriate error message otherwise.

```
> MEMBER2 := proc(x,a)
    local v;
    if not type(a,list) then
      ERROR(`2nd argument must be a list`)
    fi;
    for v in a do if v = x then RETURN(true) fi od;
    false
  end;
```

$MEMBER2 :=$ **proc**$(x,\ a)$
  local v;
    if not type$(a,\ list)$ **then** ERROR(‘*2nd argument must be a list*‘) **fi** ;
    for v **in** a **do if** $v = x$ **then** RETURN(*true*) **fi od** ;
    false
  end

```
> L := [i,j,k];
  MEMBER2(i,L);
  MEMBER2(i,L1);
```

$$L := [i,\ j,\ k]$$
$$true$$

```
Error, (in MEMBER2) 2nd argument must be a list
```

3.2 Tracing Procedure Execution: printlevel

The simplest tool for looking at the execution of a procedure is the ***printlevel*** **facility**. **The *printlevel* variable is a global variable that is initially assigned 1.**
If you set it to a higher value, a **trace** of all assignments, procedure entries and exits is printed.

Here is a procedure which computes the **greatest common divisor** of two non-negative integers using the Euclidean algorithm.
For example, the greatest common divisor of the integers 21 and 15 is 3 because 3 is the largest integer that divides both 21 and 15.
By the way, the Euclidean algorithm is one of the oldest known algorithms in Mathematics. It dates back to around 3000 years.

```
> restart;
  GCD := proc(a,b)
   local c,d,r;
   c := a;
   d := b;
   while d <> 0 do r := irem(c,d); c := d; d := r od;
   c # returned function value
  end;
```

$GCD :=$ **proc**$(a,\ b)$
  local $c,\ d,\ r$;
    $c := a$; $d := b$; **while** $d \neq 0$ **do** $r :=$ irem$(c,\ d)$; $c := d$; $d := r$ **od** ; c
  end

The *irem* function in GCD computes the **i**nteger **rem**ainder of two integers.

How does the GCD routine really work, we look by printlevel.
Let's see what happens when we compute GCD(21,15).

```
> GCD(21,15);
```

$$3$$

```
> printlevel := 4; # >=5, 5 -> without some inner levels
      # 6,7,...-> full level presentation
  GCD(21,15);
  printlevel := 5;
  GCD(21,15);
  printlevel := 6;
  GCD(21,15);
```

$$printlevel := 4$$
$$3$$
$$printlevel := 5$$

```
{--> enter GCD, args = 21, 15
```

$$c := 21$$
$$d := 15$$
$$3$$

```
<-- exit GCD (now at top level) = 3}
```

$$3$$
$$printlevel := 6$$

```
{--> enter GCD, args = 21, 15
```

$$c := 21$$
$$d := 15$$
$$r := 6$$
$$c := 15$$
$$d := 6$$
$$r := 3$$
$$c := 6$$
$$d := 3$$
$$r := 0$$
$$c := 3$$
$$d := 0$$
$$3$$

```
<-- exit GCD (now at top level) = 3}
```

$$3$$

```
> printlevel := 6;
  GCD(1206,333);
```

$$printlevel := 6$$

```
{--> enter GCD, args = 1206, 333
```

$$c := 1206, d := 333, r := 207$$
$$c := 333, d := 207, r := 126$$
$$c := 207, d := 126, r := 81$$
$$c := 126, d := 81, r := 45$$
$$c := 81, d := 45, r := 36$$
$$c := 45, d := 36, r := 9$$
$$c := 36, d := 9, r := 0$$
$$c := 9$$
$$d := 0$$
$$9$$

```
<-- exit GCD (now at top level) = 9}
```

$$9$$

We see that the input arguments to the GCD procedure are displayed together with the value returned. The execution of each assignment statement is also displayed for printlevel being enough big.

An interesting point about our GCD procedure is that this routine works for integers of any size because Maple uses arbitrary precision integer arithmetic.
For example, here is the greatest common divisor between 100! and 2^100.

```
> printlevel := 1;
  g1 := GCD(100!,2^100);
  ifactor(g1); # factorization of integers
```

$$printlevel := 1$$

$$g1 := 158456325028528675187087900672$$

$$(2)^{97}$$

Recursive routine

The GCD procedure could also have been been written recursively and should also include type checking on the input parameters.

```
> GCD1 := proc(a,b)
    if b = 0 then a else GCD1(b,irem(a,b)) fi
  end;
  GCD1(15,21);
  GCD1(15.0,21);
```

$$GCD1 := \textbf{proc}(a,\, b)\,\textbf{if}\, b = 0\,\textbf{then}\, a\,\textbf{else}\,\mathrm{GCD1}(b,\, \mathrm{irem}(a,\, b))\,\textbf{fi}\,\textbf{end}$$

$$3$$

```
Error, (in GCD1) wrong number (or type) of parameters in function irem
```

```
> printlevel := 1;
  GCD2 := proc(a::integer,b::integer) # parameter check
    if b = 0 then a else GCD2(b,irem(a,b)) fi
  end;
  GCD2(15,21);
  GCD2(1.0,6);
```

$$printlevel := 1$$

$$GCD2 := \textbf{proc}(a{::}integer,\, b{::}integer)\,\textbf{if}\, b = 0\,\textbf{then}\, a\,\textbf{else}\,\mathrm{GCD2}(b,\, \mathrm{irem}(a,\, b))\,\textbf{fi}\,\textbf{end}$$

$$3$$

```
Error, GCD2 expects its 1st argument, a, to be of type integer, but
received 1.0
```

The recursive version is simpler and easier to understand.
Let us see a trace of the recursive version to see how the printlevel facility can show the flow of computation. Please check also the printlevel number giving the full information.

```
> printlevel := 30; # if <30 no full flow
  GCD2(15,21);
```

$$printlevel := 30$$

```
{--> enter GCD2, args = 15, 21
{--> enter GCD2, args = 21, 15
{--> enter GCD2, args = 15, 6
{--> enter GCD2, args = 6, 3
{--> enter GCD2, args = 3, 0
```

$$3$$

```
<-- exit GCD2 (now in GCD2) = 3}
```

$$3$$

```
<-- exit GCD2 (now in GCD2) = 3}
```

$$3$$

```
<-- exit GCD2 (now in GCD2) = 3}
```

$$3$$

```
<-- exit GCD2 (now in GCD2) = 3}
```

$$3$$

```
<-- exit GCD2 (now at top level) = 3}
```

$$3$$

3.3 Arrow Operators

There are a procedures (statements) which compute only a formula.

```
> restart;
  a := x^2+1;
```

$$a := x^2 + 1$$

To construct functions we have several possibilities.

(1) Procedures

```
> f0 := proc(x) x^2+1 end;
  f0(x);
```

$$f0 := \mathbf{proc}(x)\, x^2 + 1\, \mathbf{end}$$

$$x^2 + 1$$

(2) Arrow syntax

This mimics the arrow syntax for functions often used in algebra.

For functions of one parameter (symbol can be also in parentheses).

symbol —> [**local** ***nameseq*** ;] ***expr***

For 0 or more parameters, parameters are put in parentheses.

(***nameseq***) —> [**local** ***nameseq*** ;] ***expr***

```
> f1 := x -> sin(x)+1;
  f1(1);
  evalf(f1(1)); # now sinus computation
```

$$f1 := x \to \sin(x) + 1$$

$$\sin(1) + 1$$

$$1.841470985$$

```
> f2 := (x,y) -> x^2+y^2;
  f2(2,3);
```

$$f2 := (x,\, y) \to x^2 + y^2$$

$$13$$

Definition of piecewise functions

if statement

```
> g1 := x -> if x<0 then 0
      elif x<1 then x
      elif x<2 then 2-x else 0
      fi;
  g1(1.5);
  g1(x); # some disadvantages in calculation rules
```

$g1 :=$ **proc**(x)
  option *operator*, *arrow*;
    if $x < 0$ **then** 0 **elif** $x < 1$ **then** x **elif** $x < 2$ **then** $2 - x$ **else** 0 **fi**
  end

$$.5$$

```
Error, (in g1) cannot evaluate boolean
```

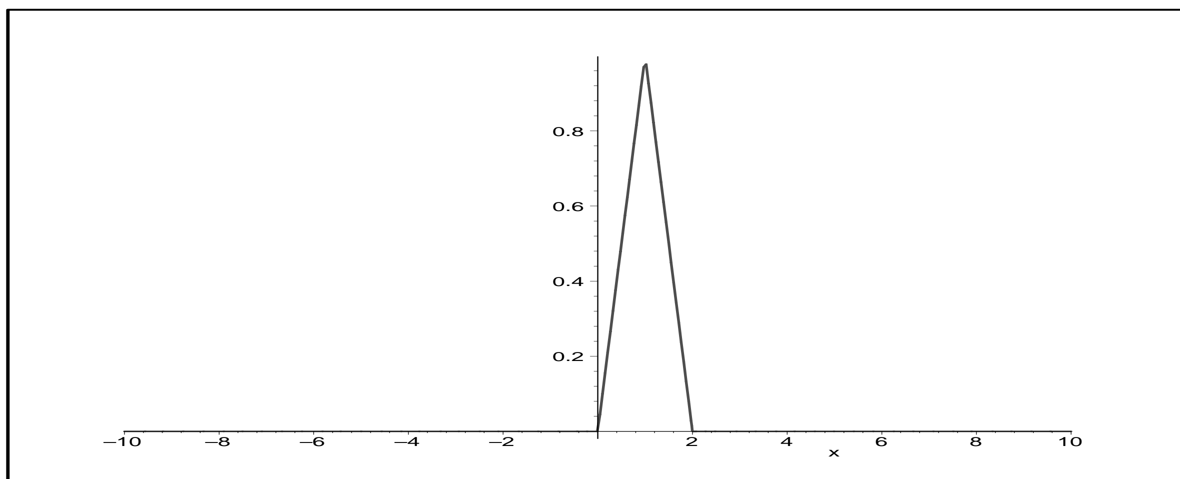
piecewise command

```
> g2 := x -> piecewise(x<0,0,
      x<1,x,
      x<2,2-x,
      0);
  g2(x);
```

$$g2 := x \to \mathrm{piecewise}(x < 0,\, 0,\, x < 1,\, x,\, x < 2,\, 2 - x,\, 0)$$

$$\begin{cases} 0 & x < 0 \\ x & x < 1 \\ 2 - x & x < 2 \\ 0 & otherwise \end{cases}$$

```
> smartplot(piecewise(x < 0,0,x < 1,x,x < 2,2-x,0));
```



```
> #smartplot
```

Notation and evaluation of function $g2(x)$ is more efficient.

(3) Unapply function

Syntax

unapply (***expr, nameseq***)

```
> a := x^2+1;
  unapply(a,x); # anonyme function
```

$$a := x^2 + 1$$

$$x \to x^2 + 1$$

```
> g := unapply(x-y,x,y);
  g(3,3);
```

$$g := (x, y) \to x - y$$

$$0$$

```
> h := unapply(x+y);
  h(); # our function
  x := 1;
  y := 2;
  h();
  h(z1); # arguments ignored
  h(z1,z2);
  h; # a symbol
```

$$h := () \to x + y$$

$$x + y$$

$$x := 1$$

$$y := 2$$

$$3$$

$$3$$

$$3$$

$$h$$

3.4 Scope Rules: Parameters, Locals, Globals

Maple supports nested procedures.
But pay attention on the scopes of defintion of the parameters.

```
> restart;
  f1 := proc(x)
    local g;
    g := x -> x+1;
    x*g(x) # returned value
  end;
  f1(2);
```

$$f1 := \mathbf{proc}(x)\,\mathbf{local}\,g;\ g := x \to x + 1\,;\ x \times g(x)\,\mathbf{end}$$

$$6$$

Procedure $f1$ has a local variable g which is a procedure.
$f1$ computes $x*(x+1)$. However, nested parameters and local variables do not use nested scoping rules.

E.g. the above procedure is not equivalent to this one

```
>  f2 := proc(x)
     local g;
     g := () -> x+1; # this x is global
     x*g() # `x` is a lexically scoped parameter
   end;
```

$$f2 := \mathbf{proc}(x)\,\mathbf{local}\,g;\ g := () \to x+1\,;\ x \times g()\,\mathbf{end}$$

because the reference to x in the g procedure does not refer to the parameter x in $f2$. It refers to the global variable x.

Consider these examples.

```
>  a := 'a'; x := 'x';
   f1(a);
   f2(a);
   x := 7;
   f1(x);
   f2(a);
```

$$a := a$$
$$x := x$$
$$a\,(a+1)$$
$$a\,(a+1)$$
$$x := 7$$
$$56$$
$$a\,(a+1)$$

One similarly cannot refer to local variables in outer scopes.
Although Maple supports nested procedures, which can be returned as function values, it does not support nested lexical scopes, so you cannot return closures directly.

3.5 Evaluation Rules: Actual, Formal Parameters, Locals, Globals

Consider the function call $f(x1, x2, ..., xn)$.
The execution of this function call proceeds as follows. The function name f is evaluated.
Next the arguments $x1, x2, ..., xn$ are evaluated from left to right.
Then if f evaluated to a procedure, the procedure is executed on the evaluated arguments.
There are only 6 exceptions to this, including *eval*, *assigned*, and *seq*.

Now, what about the evaluation of variables inside procedures?
Consider the following procedures.

```
>  restart;
   f := proc() # no parameters
     local p;
     global x;
     p := x^2+4*x+4;
     x := 5;
```

```
  p # eval(p) --> 49
end;
f();
```

$$f := \textbf{proc}()\,\textbf{local}\,p;\;\textbf{global}\,x;\;p := x^2 + 4\times x + 4\,;\;x := 5\,;\;p\,\textbf{end}$$

$$x^2 + 4\,x + 4$$

Implicitly declared local variable p.

```
> restart;
  f1 := proc() # no parameters
   global x;
   p := x^2+4*x+4;
   x := 5;
   p # `p` is implicitly declared local
  end;
   f1();

Warning, `p` is implicitly declared local
```

$$f1 := \textbf{proc}()\,\textbf{local}\,p;\;\textbf{global}\,x;\;p := x^2 + 4\times x + 4\,;\;x := 5\,;\;p\,\textbf{end}$$

$$x^2 + 4\,x + 4$$

For reasons of efficiency and desirability, the Maple designers have decided that local variables and parameters evaluate one level, i.e. the value of p in the two examples f and g is the polynomial $x^2 + 4x + 4$, not the value 49.

```
> x := 'x':
  g := proc(p) # p is a parameter
   global x;
   x := 5;
   p
  end;
  g(x^2+4*x+4);
```

$$g := \textbf{proc}(p)\,\textbf{global}\,x;\;x := 5\,;\;p\,\textbf{end}$$

$$x^2 + 4\,x + 4$$

The *eval* function can be used to get full evaluation for local variables and parameters, and one level evaluation of global variables should you ever need it.
Full evaluation only occurs for global variables.

```
> x := 'x':
  g1 := proc(p) # p is a parameter
   global x;
   x := 5;
   eval(p)
  end;
  g1(x^2+4*x+4);
```

$$g1 := \textbf{proc}(p)\,\textbf{global}\,x;\;x := 5\,;\;\mathrm{eval}(p)\,\textbf{end}$$

$$49$$

See also

```
> x := 'x':
  p := x^2+4*x+4;
  x := 5;
  p;
```

$$p := x^2 + 4\,x + 4$$
$$x := 5$$
$$49$$

Here x and p are global variables. Global variables are evaluated fully, i.e. recursively, hence the result is 49.

3.6 Recurrence Equations and Option Remember

The **Fibonacci numbers** F_n are defined by the linear recurrence $F_0 = 0,\ F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$.
This can be coded directly.

```
> restart;
  F := proc(n)
    if n = 0 then 0 elif n = 1 then 1 else
      F(n-1)+F(n-2) fi
  end;
```

$$F := \mathbf{proc}(n)\,\mathbf{if}\,n = 0\,\mathbf{then}\,0\,\mathbf{elif}\,n = 1\,\mathbf{then}\,1\,\mathbf{else}\,\mathrm{F}(n-1) + \mathrm{F}(n-2)\,\mathbf{fi}\,\mathbf{end}$$

Here are the first few Fibonacci numbers.

```
> F(25); # please patience
  seq( F(i), i=0..10 );
```

$$75025$$
$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55$$

However this is a not an efficient way to compute the Fibonacci numbers. In fact, you will never be able to compute $F(100)$ using this procedure even on the fastest computer. If you count the number of calls to the F procedure, you will see that it is called repeatedly on the same arguments.

```
> printlevel := 100;
  F(3);
```

$$printlevel := 100$$

```
{--> enter F, args = 3
{--> enter F, args = 2
{--> enter F, args = 1
                                    1
<-- exit F (now in F) = 1}

{--> enter F, args = 0
                                    0
<-- exit F (now in F) = 0}
                                    1
<-- exit F (now in F) = 1}
{--> enter F, args = 1
                                    1
<-- exit F (now in F) = 1}
                                    2
<-- exit F (now at top level) = 2}
                                    2
```

It is clear that one should remember the previous two values when computing the next value. This could be done in three ways.

(1) With a loop

```
> F1 := proc(n) local i,fnm1,fnm2,f;
    if (n = 0) or (n = 1) then RETURN(n) fi;
    fnm2 := 0; fnm1 := 1;
    for i from 2 to n
      do f:=fnm1+fnm2; fnm2:=fnm1; fnm1:=f od;
    fnm1
  end;
```

$F1 :=$ **proc**(n)
local i, $fnm1$, $fnm2$, f;
if $n = 0$ **or** $n = 1$ **then** RETURN(n) **fi** ;
$fnm2 := 0$;
$fnm1 := 1$;
for i **from** 2 **to** n **do** $f := fnm1 + fnm2$; $fnm2 := fnm1$; $fnm1 := f$ **od** ;
$fnm1$
end

```
> printlevel := 1;
  F1(100); # very fast computation
```

$$printlevel := 1$$

$$354224848179261915075$$

(2) Use the option remember

This option is used to store values as they are computed so that they can be used when they are needed.

```
> F2 := proc(n) option remember;
    if n=0 then 0 elif n=1 then 1 else
      F2(n-1)+F2(n-2) fi
  end;
```

$F2 :=$ **proc**(n)
option *remember*;
if $n = 0$ **then** 0 **elif** $n = 1$ **then** 1 **else** F2$(n-1)$ + F2$(n-2)$ **fi**
end

```
> F2(100);
```

$$354224848179261915075$$

This program computes $F2(100)$ quite quickly.
Each procedure has an **associated remember table**.
The table index is the arguments and the table entry is the function value. When $F2$ is called with n, Maple first looks up $F2$'s remember table to see if $F2(n)$ has already been computed. If it has, it returns the result from $F2$'s remember table.
Otherwise, it executes the code for the procedure $F2$, and automatically stores the pair $n, F2(n)$ in $F2$'s remember table.

(3) Functional assignment

We also illustrate the possibility of explicitly saving values in a remember table by using the so called **functional assignment**.
This is more flexible than the remember option because it allows one to save only selected values in the remember table.

```
> F3 := proc(n) F3(n) := F3(n-1)+F3(n-2) end;
  F3(0) := 0; F3(1) := 1;
```

$$F3 := \textbf{proc}(n)\, \mathrm{F3}(n) := \mathrm{F3}(n-1) + \mathrm{F3}(n-2)\, \textbf{end}$$

$$\mathrm{F3}(0) := 0$$

$$\mathrm{F3}(1) := 1$$

```
> F3(100); # very fast computation
```

$$354224848179261915075$$

3.7 Types and Map

The **type** function can be used to code a routine that does different things depending on the type of the input.

Applications

(1) Actions depending on the type of routine parameter

The **DIFF1** routine differentiates expressions which are a power in the given variable x. Further, we apply the **ERROR** function for error message.

```
> restart;
  DIFF1 := proc(a::algebraic,x::name)
    local u,v;
    if type(a,numeric) then 0
    elif type(a,name) then
      if a = x then 1 else 0 fi
    elif type(a,name^integer) then
      u := op(1,a); v := op(2,a);
      v*DIFF1(u,x)*u^(v-1)
    else ERROR(`don't know how to differentiate`,a)
    fi;
  end:
> DIFF1(1,x);
  DIFF1(y,x);
  DIFF1(x^5,x);
  DIFF1(x^(-5),x);
  DIFF1(x+1,x);
```

$$0$$

$$0$$

$$5\,x^4$$

$$-5\,\frac{1}{x^6}$$

```
Error, (in DIFF1) don't know how to differentiate, x+1
```

(2) Map function and structured types

The **DIFF2** routine differentiates expressions which are a polynomials in the given variable x. Further, we apply the **ERROR** function for error message and the **map** function.

```
> restart;
  DIFF2 := proc(a::algebraic,x::name)
    local u,v;
    if type(a,numeric) then 0
    elif type(a,name) then
      if a = x then 1 else 0 fi
    elif type(a,`+`) then map(DIFF2,a,x)
    elif type(a,`*`) then
      u := op(1,a); v := a/u;
      DIFF2(u,x)*v + DIFF2(v,x)*u
    elif type(a,anything^integer) then
      u := op(1,a); v := op(2,a);
      v*DIFF2(u,x)*u^(v-1)
    else ERROR(`don't know how to differentiate`,a)
    fi;
  end:
```

Types are used in the DIFF2 procedure for two different purposes.

The first usage is for type checking. The inputs must be an algebraic expression or a formula, and a name for the differentiation variable.
The second usage is to examine the type of the input - is it a sum, product or power - and decide what differentiation rule is to be used.

```
> a := x^2-x-1;
  DIFF2(a,x);
```

$$a := x^2 - x - 1$$
$$2\,x - 1$$

The DIFF2 example shows the use of the **map** function, a very useful function, which we will now explain.

Syntax of map function

map(*f* , *a*, *x1*,..., *xn*)

The meaning is to apply the function f to the operands of the expression a passing the additional arguments $x1, ..., xn$ to f.

In our DIFF2 procedure, there is one additional argument x. Often, there are no additional arguments.
Formally, this is similar to computing the sequence

seq(f(op(i,a), x1,...,xn), i=1..nops(a))

and connect the sequence components by the main operator of a, i.e. creating from this sequence a value of the same type as the type of a.

```
> restart;
  f := z -> sin(z);
  a := x^2+2*x+1;
  map(f,a); # only one argument of f
  s := seq(f(op(i,a)),i=1..nops(a)); # a sequence
```

```
o := whattype(a);
l := [s]; # a list
if o='+' then sum(l[i],i=1..nops(l)) fi;
```

$$f := \sin$$

$$a := x^2 + 2\,x + 1$$

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

$$s := \sin(x^2),\ \sin(2\,x),\ \sin(1)$$

$$o := +$$

$$l := [\sin(x^2),\ \sin(2\,x),\ \sin(1)]$$

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

```
> b := x^(3^i);
  map(f,b);
  map(tan,-x^2-x-1);
```

$$b := x^{(3^i)}$$

$$\sin(x)^{\sin(3^i)}$$

$$-\tan(x^2) - \tan(x) - \tan(1)$$

Using the *map* operation with a formal map function F (not defined).

```
> p := x^3+2*x+1;
  q := map(y -> y^2,p);
  map(F,p);
```

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

$$q := x^6 + 4\,x^2 + 1$$

$$\mathrm{F}(x^3) + \mathrm{F}(2\,x) + \mathrm{F}(1)$$

Many standard functions can be involved in the map operation.
But pay attention on the number and type of arguments of this standard routine.

```
> n := degree(p,x);
  map(degree,p,x);
  seq(degree(op(i,p),x), i=1..nops(p));
```

$$n := 3$$

$$4$$

$$3,\ 1,\ 0$$

Structured type

The DIFF2 function also shows the use of a **structured type**.
The types *anything*, *name*, '+' and '*' are simple types. The type *anything^integer* is a structured type. It means that the value must be a power, and the base can be anything, i.e. any type, but the exponent must be an integer. It is equivalent to writing.

```
> a := x^2;
  if type(a,'^') and type(op(2,a),integer) then true fi;
  if type(a,anything^integer) then true fi;
```

$$a := x^2$$

$$\mathit{true}$$

$$\mathit{true}$$

Structured types allow you to replace long type tests with concise tests. Let us illustrate another common case.

(1) Integration bounds

The type *name = algebraic..algebraic* is the type of the second argument for definite integration, and definite summation.

```
> int( exp(-2*t)*ln(t), t=0..infinity );
  sum( 1/(i^2-1), i=2..infinity );
```

$$-\frac{1}{2}\ln(2) - \frac{1}{2}\gamma$$

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

Of course, the integration and summation commands also allow indefinite integration and summation.

```
> restart;
  int( exp(-2*x)*ln(x), x );
  sum( 1/(i^2-1), i=2..n-1 );
  sum( 1/(n^2-1), n ); # main sum from first summands
  sum( 1/(i^2-1), i=0..n ); # i<>1
```

$$-\frac{1}{2}e^{(-2\,x)}\ln(x) - \frac{1}{2}\,\mathrm{Ei}(1,\,2\,x)$$

$$-\frac{1}{2}\,\frac{-1+2\,n}{n\,(n-1)} + \frac{3}{4}$$

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

$$\sum_{i=0}^{n}\frac{1}{i^2-1}$$

So the type of the second argument can be either just a variable *name* or the above type *name = algebraic..algebraic*.

(2) Solving equations

Many routines take a set or list of names or equations as arguments.
For example, the *solve* command allows one to solve a set of equations for a set of unknowns.

```
> restart;
  solve( x^2-1, x );
  solve( x^2-1 , {x});
  solve( {x+y=2, x-y=3});
  solve( {x+y=2, x-y=3}, {x,y} );
```

$$1,\,-1$$

$$\{x=1\},\,\{x=-1\}$$

$$\{x=\frac{5}{2},\,y=\frac{-1}{2}\}$$

$$\{x=\frac{5}{2},\,y=\frac{-1}{2}\}$$

A set of zero or more equations can be tested with the type *set* or *equation*, and as set of zero or more unknowns with the type *set* or *name*.
I.e. the call *type*(*expr*, *equation*) checks to see if *expr* is of type *equation*.

```
> q := x+y=2;
  p := x+y-2;
  q1 := {x+y=2,x-y=3};
  q2 := {x,y};
  type(q,equation);
  type(p,algebraic);
  type(q1,set);
  type(q2,name);
```

$$q := x + y = 2$$
$$p := x + y - 2$$
$$q1 := \{x + y = 2,\ x - y = 3\}$$
$$q2 := \{x,\ y\}$$
$$true$$
$$true$$
$$true$$
$$false$$

But the *solve* command also allows a set of algebraic formulas which are implicitly equated to zero, i.e. the example above could have been input this way.

```
> solve( {x+y-2, x-y-3}, {x,y} );
```

$$\{x = \frac{5}{2},\ y = \frac{-1}{2}\}$$

3.8 Variable Number of Arguments: args and nargs

It is possible for a function to take a variable number of parameters.
An example of such a function is the **max** function.
Here is an initial attempt to code up this function.

```
> restart;
  MAX1 := proc(x1)
    local maximum,i;
    maximum := x1;
    for i from 2 to nargs do
      if args[i] > maximum then maximum := args[i] fi
    od;
    maximum
  end:
```

Analogue to the built in *max* function.

```
> max(1);
  max(3,2,4);
  max(1,2,x);
```

$$1$$
$$4$$
$$\max(x,\ 2)$$

The special variable *nargs* is the number of arguments, and the variable *args* is a sequence of the arguments, hence $args[i]$ is the i'th argument.

```
> MAX1(2);
  MAX1(2,4,1,7);
  MAX1(x,x+1); # symbolic evaluation
  MAX1(2,4,1,7,x); # a problem
```

$$2$$
$$7$$
$$x + 1$$

```
Error, (in MAX1) cannot evaluate boolean
```

```
> sq := 1,2,3;
  MAX1(sq);
```

$$sq := 1,\ 2,\ 3$$
$$3$$

3.9 Returning Unevaluated

The MAX1 procedure that we have just written works for numerical arguments only (numbers of type *numeric*).
If you try the Maple function *max* you will see that it also works for symbolic arguments.

```
> MAX1(1,2,x);
  MAX1(Pi,sqrt(2));
```

```
Error, (in MAX1) cannot evaluate boolean
Error, (in MAX1) cannot evaluate boolean
```

Maple cannot execute the procedure MAX1 because it cannot compute whether $args[i] < maximum$ for a non-numeric value.
We want MAX1 of some numbers to compute the answer, but otherwise, we want MAX1(x, y) to stay as MAX1(x, y) so we can compute with MAX1(x, y) symbolically.

To help us write such a maximum function, we will make use of the **signum** function which provides a more powerful comparison of two real values.
The *signum* function returns -1 if it can show that the $x < 0$, +1 if $x \geq 0$, otherwise it returns *unevaluated*, i.e. it returns $signum(x)$.

```
> signum(2.0);
  signum(Pi-sqrt(2));
  signum(z);
  signum(a-b);
```

$$1$$
$$1$$
$$\mathrm{signum}(z)$$
$$-\mathrm{signum}(-a + b)$$

Let us employ the *signum* function to make our MAX2 function smarter and also let our MAX2 function handle symbolic arguments.

```
> MAX2 := proc()
    local a,i,j,n,s;
    n := nargs; # First, put the arguments in an array
    a := array(1..n);
    for i to n do a[i] := args[i] od;
```

```
      # Compare a[i] with a[j] for 1 <= i < j <= n
    i := 1;
    while i < n do
      j := i+1;
      while j <= n do
        s := signum(a[i]-a[j]);
        if s = 1 then # i.e.  a[i] >= a[j]
          a[j] := a[n];
          n := n-1;
        elif s = -1 then # i.e.  a[i] < a[j]
          a[i] := a[j]; a[j] := a[n];
          j := n; n := n-1; i := i-1;
        else # cannot determine the sign
        j := j+1
        fi
      od;
      i := i+1;
    od;
    if n = 1 then RETURN(a[1]) fi;
    'MAX2'( seq(a[i], i=1..n) );
  end:
> MAX2(1,x);
  MAX2(Pi,sqrt(2),3*sin(1));
  MAX2(x,1,sqrt(2),x+1);
```

$$\mathrm{MAX2}(1,\, x)$$

$$\pi$$

$$\mathrm{MAX2}(x+1,\, \sqrt{2})$$

What is most interesting about the above code is the last line.
The back quotes ' are used to prevent the MAX2 function call from executing as otherwise it would go into an infinite loop. Instead, the unevaluated function call MAX2(...) is returned, indicating that the maximum could not be computed. However, some simplifications may have taken place.

3.10 Simplifications and Transformation Rules

Often one wants to introduce simplifications which can be described algebraically or by transformation rules. For instance, given a function f, we may know that f is both commutative and associative.
max is in fact such a function, i.e. it is true that $max(a,b) = max(b,a)$ and $max(a, max(b,c)) = max(max(a,b),c) = max(a,b,c)$.
How can we implement these properties in Maple?
What we want is a **canonical way** for writing expressions involving the max function.

Implementation

- **Commutativity** by sorting the arguments.
- For **associativity** we can unnest any nested max calls.
  I.e. we would transform both $max(max(a,b),c)$ and $max(a, max(b,c))$ into $max(a,b,c)$.

Actually this also implements $max(max(a)) = max(a)$, i.e. max is **idempotent**.

```
> restart;
  max(1,2);
  max(max(b,c),a);
  max(max(a));
  max(i,max(k,l,j),j); # duplicates removed
```

$$2$$
$$\max(c,\ b,\ a)$$
$$a$$
$$\max(i,\ j,\ l,\ k)$$

Implementation of sort algorithm for symbolic elements

(1) Helpful commands

Sort a list.

A list of numbers is sorted into numerical order, and a list of strings (symbols) into lexicographical order. Sets or mixed list are sorting by machine address, and it is session dependent.

```
> restart;
  lst1 := [b,a,c,aa];
  sort(lst1); # sort(lst1,lexorder)
  op(sort(lst1));
  lst2 := [7,3,2];
  sort(lst2);
  lst3 := [23,2,k,ii,i,k1,j,k,1];
      # sorting by machine address is session dependent
  sort(lst3);
  sort({a,b,c});
```

$$\mathit{lst1} := [b,\ a,\ c,\ aa]$$
$$[a,\ aa,\ b,\ c]$$
$$a,\ aa,\ b,\ c$$
$$\mathit{lst2} := [7,\ 3,\ 2]$$
$$[2,\ 3,\ 7]$$
$$\mathit{lst3} := [23,\ 2,\ k,\ ii,\ i,\ \mathit{k1},\ j,\ k,\ 1]$$
$$[ii,\ i,\ \mathit{k1},\ j,\ 1,\ 2,\ k,\ k,\ 23]$$
$$\{b,\ a,\ c\}$$

Type of function and command op

```
> restart;
  g := proc() x*y end;
  type(g,procedure);
  h := x*y;
  type(h,algebraic);
  op(h);
  op(0,h);
  type(f(x,y),function);
  op(f(x,y));
  op(0,f(x,y));
```

$$g := \mathbf{proc}()\, x * y\, \mathbf{end}$$
$$\mathit{true}$$
$$h := x\, y$$
$$\mathit{true}$$

$$x,\ y$$
$$*$$
$$true$$
$$x,\ y$$
$$f$$

Map command

```
> la := [1,2,3];
  n := 2;
  fm := proc(m,l) m*l end;
  fm(n,la);
  fm(la,t);
  p := x^2+2*x+1;
  map(fm,p,la); # polynomial x^2*la+2x*la+la
```

$$la := [1,\ 2,\ 3]$$
$$n := 2$$
$$fm := \mathbf{proc}(m,\ l)\, m * l\, \mathbf{end}$$
$$[2,\ 4,\ 6]$$
$$[1,\ 2,\ 3]\, t$$
$$p := x^2 + 2\,x + 1$$
$$x^2\,[1,\ 2,\ 3] + 2\,x\,[1,\ 2,\ 3] + [1,\ 2,\ 3]$$

Mapping of a list means creating a list of its mapped elements.

```
> map(fm,la,p); # list [la[1]*p,la[2]*p,la[3]*p]
```

$$[x^2 + 2\,x + 1,\ 2\,x^2 + 4\,x + 2,\ 3\,x^2 + 6\,x + 3]$$

Variable number of procedure parameters: args and nargs

```
> mm := proc()
    if nargs=0 then nargs else args[1..nargs] fi
  end;
  mm();
  mm(i,j,k);
```

$$mm := \mathbf{proc}()\,\mathbf{if}\,\mathrm{nargs} = 0\,\mathbf{then}\,\mathrm{nargs}\,\mathbf{else}\,\mathrm{args}_{1..\mathrm{nargs}}\,\mathbf{fi}\,\mathbf{end}$$
$$0$$
$$i,\ j,\ k$$

(2) Basic variant MAX1

```
> restart;
  flat := proc(y,f)
    if type(y,function) and op(0,y) = f then op(y) else y fi
  end:
  MAX1 := proc() local a;
    a := [args];
    a := map( flat, a, MAX1 ); # unnest all nested MAX1 calls
    'MAX1'( op(sort(a)) );
  end:
> MAX1(b,c,a);
  MAX1(a,a);
  MAX1(MAX1(a));
  MAX1(b,MAX1(a,a));
  MAX1(MAX1(a,MAX1(b,c)),a);
```

$$\text{MAX1}(a,\, b,\, c)$$
$$\text{MAX1}(a,\, a)$$
$$\text{MAX1}(a)$$
$$\text{MAX1}(a,\, a,\, b)$$
$$\text{MAX1}(a,\, a,\, b,\, c)$$

(3) Variant MAX2 without duplicates

We see that we should also recognize the property that $max(a, a) = a$.
To do this, instead of putting the arguments in a list, we will put them in a set so that duplicates are removed.
Also, since sets are sorted automatically by machine address, we have to put at the end the set in a list and apply the call to sort.

```
> MAX2 := proc()
    local a;
    a := {args};
    a := map( flat, a, MAX2 ); # unnest all nested MAX2 calls
    'MAX2'( op(sort([op(a)])) );
  end:
```

```
> MAX2(b,c,a);
  MAX2(a,a);
  MAX2(MAX2(a));
  MAX2(b,MAX2(a,a));
  MAX2(MAX2(a,MAX2(b,c)),a);
```

$$\text{MAX2}(a,\, b,\, c)$$
$$\text{MAX2}(a)$$
$$\text{MAX2}(a)$$
$$\text{MAX2}(a,\, b)$$
$$\text{MAX2}(a,\, b,\, c)$$

Applying MAX2 with numbers.

```
> MAX2(2,MAX2(2,1));
  max(2,max(2,1));
```

$$\text{MAX2}(1,\, 2)$$
$$2$$

The reader may be a little puzzled as to just what our MAX2 procedure is doing.
We have seen earlier that if we assign a positive integer to the *printlevel* variable, we get a trace of all statements executed.
However, often the output from this simple tracing facility is too much. In this case, we would also get the output from the flat procedure too. The **trace** function can be used instead to selectively trace procedures.

```
> trace(MAX2);
  MAX2(a,MAX2(b,a),c);
```

$$MAX2$$

```
{--> enter MAX2, args = b, a
```

$$a := \{a,\, b\}$$
$$a := \{a,\, b\}$$
$$\text{MAX2}(a,\, b)$$

```
<-- exit MAX2 (now at top level) = MAX2(a,b)}
{--> enter MAX2, args = a, MAX2(a,b), c
```

$$a := \{a,\, c,\, \text{MAX2}(a,\, b)\}$$
$$a := \{a,\, b,\, c\}$$
$$\text{MAX2}(a,\, b,\, c)$$

```
<-- exit MAX2 (now at top level) = MAX2(a,b,c)}
```

$$\mathrm{MAX2}(a,\, b,\, c)$$

```
> MAX2(a,MAX2(b,MAX2(d,a)),c);

{--> enter MAX2, args = d, a
```

$$a := \{a,\, d\}$$
$$a := \{a,\, d\}$$
$$\mathrm{MAX2}(a,\, d)$$

```
<-- exit MAX2 (now at top level) = MAX2(a,d)}
{--> enter MAX2, args = b, MAX2(a,d)
```

$$a := \{b,\, \mathrm{MAX2}(a,\, d)\}$$
$$a := \{a,\, b,\, d\}$$
$$\mathrm{MAX2}(a,\, b,\, d)$$

```
<-- exit MAX2 (now at top level) = MAX2(a,b,d)}
{--> enter MAX2, args = a, MAX2(a,b,d), c
```

$$a := \{a,\, c,\, \mathrm{MAX2}(a,\, b,\, d)\}$$
$$a := \{a,\, b,\, c,\, d\}$$
$$\mathrm{MAX2}(a,\, b,\, c,\, d)$$

```
<-- exit MAX2 (now at top level) = MAX2(a,b,c,d)}
```

$$\mathrm{MAX2}(a,\, b,\, c,\, d)$$

3.11 Optional Arguments and Default Values

Many Maple routines accept optional arguments.
This is often used to allow the user to use default values instead of having to specify all parameters.
Examples are the functions **factor, collect, degree, plot,** and **series**.

(1) Factor a univariate and multivariate polynomial

```
> restart;
  p := x^2-1;
  factor(p);
  q := x->x^2-y^2;
  factor(q(x));
```

$$p := x^2 - 1$$
$$(x-1)\,(x+1)$$
$$q := x \to x^2 - y^2$$
$$(x-y)\,(x+y)$$

```
> factor(x^2+1,complex);
```

$$(x + 1.000000000\, I)\,(x - 1.\, I)$$

(2) Collect coefficients of like powers

```
> g := int(x^2*(exp(x)+exp(-x)),x);
  collect(g,exp(x));
```

$$g := x^2\, e^x - 2\, x\, e^x + 2\, e^x - \frac{x^2}{e^x} - 2\,\frac{x}{e^x} - 2\,\frac{1}{e^x}$$
$$(2 + x^2 - 2\, x)\, e^x + \frac{-2\, x - 2 - x^2}{e^x}$$

Expanding a polynomial in one variable

```
> f := x*(x+1)+y*(x+1);
  collect(f,x);
```

$$f := x\,(x+1) + y\,(x+1)$$

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

Writing a multivariate polynomial in different forms

```
> p := x*y+a*x*y+y*x^2-a*y*x^2+x+a*x;
  collect( p, [x,y], recursive ); # x of higher priority
  collect( p, [y,x], recursive );
```

$$p := x\,y + a\,x\,y + y\,x^2 - a\,y\,x^2 + x + a\,x$$

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

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

(3) Degree of a polynomial

The **degree** function computes the degree of a univariate polynomial in one variable, and for multivariate polynomials, the **total degree**.
Sometimes you will want to compute the degree in a specific variable. This can be done by specifying an optional second argument to the *degree* function, namely, the variable.

```
> restart;
  p1 := x^3+2*x+1;
  degree(p1); # also degree(p1,x)
```

$$p1 := x^3 + 2\,x + 1$$

$$3$$

```
> p2 := x^3+2*x+1+y;
  degree(p2);
  degree(p2,y);
```

$$p2 := x^3 + 2\,x + 1 + y$$

$$3$$

$$1$$

```
> q := 3*x^2*y+2*y^2-x*z-7;
  degree(q);
  degree(q,x);
  degree(q,{x,z});
```

$$q := 3\,x^2\,y + 2\,y^2 - x\,z - 7$$

$$3$$

$$2$$

$$2$$

(4) Code an own degree function

Let us assume that the input is a formula and if an optional second argument is given, it is a name or set of names for the variables.
Using several functions, like **type, ERROR, nargs, member**, and **indets**.
The **indets** function used here returns a set of all the *indeterminates* (or variables) that appear in the input.

```
> restart;
  a := x^2+y-y*z^3+alpha+x*y*z;
  s := indets(a); # set of variables in a
```

```
whattype(sin);
whattype(s);
type(s,set);
type(s,set(name));
```

$$a := x^2 + y - y\,z^3 + \alpha + x\,y\,z$$
$$s := \{x,\, y,\, z,\, \alpha\}$$
$$symbol$$
$$set$$
$$true$$
$$true$$

For a multivariate polynomial we compute the total degree and a sequence of total degrees of all its terms.

```
> degree(a,s);
  ss := seq(degree(t,s), t=a);
  ss[5];
  max(ss);
```

$$4$$
$$ss := 2,\, 1,\, 4,\, 1,\, 3$$
$$3$$
$$4$$

A different way for computing the degree of 5'th term (last term) in polynomial a.

```
> a1 := op(5,a);
  s1 := indets(a1);
  k := 0;
  for t in a1 do k := k + degree(t,s1) od:
  k;
```

$$a1 := x\,y\,z$$
$$s1 := \{x,\, y,\, z\}$$
$$k := 0$$
$$3$$

We leave it to the reader to study each rule that is being used here, and the order in which the cases are done. The second parameter is optional.

```
> DEGREE1 := proc(a::algebraic,x::{name,set(name)})
    local s,t;
    if nargs = 1 then # determine the variable(s) for the user
      s := indets(a); # the set of all the variables of a
      if not type(s,set(name)) then ERROR(`input not a polynomial`) fi;
      DEGREE1(a,s)
    elif type(a,constant) then 0
    elif type(a,name) then
      if type(x,name) then if a = x then 1 else 0 fi
        else if member(a,x) then 1 else 0 fi
      fi
    elif type(a,`+`) then max( seq( DEGREE1(t,x), t=a ) )
    elif type(a,`*`) then
      s := 0;
      for t in a do s := s + DEGREE1(t,x) od;
      s
    elif type(a,algebraic^integer) then DEGREE1(op(1,a),x) * op(2,a)
```

```
     else ERROR(`cannot compute degree`)
     fi
   end:
> a;
  DEGREE1(x/y-1,x);
  DEGREE1(sin(x)-1);
```

$$x^2 + y - y\,z^3 + \alpha + x\,y\,z$$
$$1$$

```
Error, (in DEGREE1) input not a polynomial
```

```
> DEGREE1(a,x);
  DEGREE1(a,y);
  DEGREE1(a,{z});
  DEGREE1(a,u);
  DEGREE1(a,{x,alpha});
  DEGREE1(a,{y,z});
  DEGREE1(a);
  n := DEGREE1(a);
  `DEGREE1(a)` = n;
  DEGREE1(a) := 'DEGREE1(a)'; # our procedure is lost
  %=n;
```

$$2$$
$$1$$
$$3$$
$$0$$
$$2$$
$$4$$
$$4$$
$$n := 4$$
$$\mathit{DEGREE1}(a) = 4$$
$$\mathrm{DEGREE1}(x^2 + y - y\,z^3 + \alpha + x\,y\,z) := \mathrm{DEGREE1}(a)$$
$$\mathrm{DEGREE1}(a) = 4$$

3.12 Returning Results Through Parameters

Many functions in Maple return more than one value.
Of course, it is always possible to return more than one value in a sequence or list. However, it is also possible to return values through parameters like in other programming languages, and often, this is more convenient.

Applications

(1) Return values as list

```
> restart;
  f := proc(n) [1$n] end;
  f(3);
```

$$f := \mathbf{proc}(n)\,[1\,\$\,n]\,\mathbf{end}$$
$$[1,\,1,\,1]$$

(2) Polynomial long division by divide function

The call *divide*(a, b) returns true if and only if the polynomial b divides the polynomial a with no remainder.

```
> restart;
  divide(x^3-1,x-1);
  divide(x^3-1,x+1);
```

$$true$$
$$false$$

But usually, if b divides a, one wants to do something with the quotient q. This can be done by giving the divide function a third parameter, which is a name which will be assigned the quotient if b divides a.

```
> if divide(x^3-1,x-1,'q') then print(q) fi;
  p := q;
  if not divide(x^3-1,x+1,'r') then print(r) fi;
      # r will not be affected
```

$$x^2 + x + 1$$
$$p := x^2 + x + 1$$
$$r$$

Notice the use of quotes here to pass to the *divide* function the name q and not the value of q.

```
> restart; # or q := 'q';
  if divide(x^3-1,x-1,q) then print(q) fi;
```

$$x^2 + x + 1$$

(3) Assign a value to an optional parameter

Consider the MEMBER1 function which tested to see if a value x appears in a list L. Let us modify our function such that MEMBER1$(x, L,'p')$ still returns whether x appears in the list L, and in addition, assigns the name p the position of the first appearance of x in L.

```
> MEMBER1 := proc(x,L::list,p::name)
    local i;
    for i to nops(L) do
      if x = L[i] then if nargs = 3 then p := i fi;
        RETURN(true)
      fi
    od;
    false
  end:
> MEMBER1(4,[1,3,5],'position');
  MEMBER1(3,[1,3,5],'position');
  position;
```

$$false$$
$$true$$
$$2$$

We see that the effect of the assignment to the formal parameter p inside the MEMBER1 procedure is that the actual parameter *position* is assigned.

4 Programming

4.1 Matrix and Vector Computation

A *vector* in Maple is represented by a one-dimensional *array* indexed from 1, and a *matrix* is represented by a two-dimensional array, with row and column indices starting from 1.
Here is one way to create a 4*4 Hilbert matrix. Recall that a Hilbert matrix is a symmetric matrix whose (i, j)'th entry is $1/(i + j - 1)$.

```
> restart;
  H := array(1..4,1..4):
  for i to 4 do for j to 4 do H[i,j] := 1/(i+j-1) od od;
  H;
  eval(H);
  H: % = evalm(%);
```

$$H$$

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

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix}$$

Notice that the value of H is just the name of the matrix H. Evaluation rules for arrays, hence matrices and vectors are special. The reason is technical. For the moment, whenever you want to print a matrix or vector use the *eval* or *evalm* function.

Package *linalg*

The *linalg* package contains many functions for computing with vectors and matrices in Maple. This matrix could also have been created in the following way using the *matrix* command in the linear algebra package.

```
> linalg[matrix](4,4,(i,j) -> 1/(i+j-1));
```

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

More useful is to involve the package by the general *with* command, so you can work with all its routines.

```
> with(linalg):

Warning, new definition for norm
Warning, new definition for trace
```

Some possibilities of generating matrices

```
> A1 := matrix(3,3,[[1,2,3],[3,4,5],[0,1,0]]);
```

$$A1 := \begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

```
> A2 := x->matrix(3,3,[[0,0,x],[0,x^2$2],[x^3$3]]);
  A2(2);
```

$$A2 := x \to \mathrm{matrix}(3,\, 3,\, [[0,\, 0,\, x],\, [0,\, x^2\,\$\, 2],\, [x^3\,\$\, 3]])$$

$$\begin{bmatrix} 0 & 0 & 2 \\ 0 & 4 & 4 \\ 8 & 8 & 8 \end{bmatrix}$$

```
> A3 := matrix(3,6, (i,j) -> 1/(i+j-1));
```

$$A3 := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \end{bmatrix}$$

```
> A4 := diag(1,1,1);
```

$$A4 := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
> A5 := matrix(2,2);
  entermatrix(A5);
```

$$A5 := \mathrm{array}(1..2,\, 1..2,\, [])$$

```
enter element 1,1 > 2;
```

$$2$$

```
enter element 1,2 > 22;
```

$$22$$

```
enter element 2,1 > 4;
```

$$4$$

```
enter element 2,2 > 66;
```

$$66$$

```
> B := matrix(3,3);
  for i to rowdim(B) do for j to coldim(B) do B[i,j] := (i+1)*j od od;
  B[2,2] := 1:  B[3,3] := 0:
  evalm(B);
```

$$B := \mathrm{array}(1..3,\ 1..3,\ [])$$

$$\begin{bmatrix} 2 & 4 & 6 \\ 3 & 1 & 9 \\ 4 & 8 & 0 \end{bmatrix}$$

Maple can compute the determinant and inverse of the matrix and many other matrix operations. See `?linalg` for a list of the operations (online help).

Here we show a program for doing a so called **row reduction** on a matrix using **Gaussian elimination**. Examples by using the *linalg* procedure *gausselim*.

```
> A := matrix(3,4, (i,j) -> 1/(i+j-1));
  gausselim(A,'r'); # rank r
  r;
```

$$A := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \end{bmatrix}$$

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ 0 & \frac{1}{12} & \frac{1}{12} & \frac{3}{40} \\ 0 & 0 & \frac{1}{180} & \frac{1}{120} \end{bmatrix}$$

$$3$$

```
> A := matrix(3,4, (i,j) -> i+j-1);
  gausselim(A,'r');
  r;
```

$$A := \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -1 & -2 & -3 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$2$$

```
> A := matrix(3,4, (i,j) -> i+j-2);
  gausselim(A,'r'); # with row exchange
  r;
```

$$A := \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$2$$

```
> A := matrix(4,3, (i,j) -> i+j-1):
  A[4,3]:=7:
  evalm(A);
  gausselim(A,'r');
  r;
```

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

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$3$$

```
> A := matrix(4,3, (i,j) -> i+j-2):
  A[4,3]:=7:
  evalm(A);
  gausselim(A,'r'); # with row exchange
  r;
```

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 7 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

$$3$$

GaussianElimination(A,'r') computes also the reduced matrix, an upper triangular matrix. It also has an optional 2nd argument which, if given, is assigned the rank of the matrix.
The type $matrix(rational)$ specifies a matrix, whose entries are all rational numbers. The original matrix is not changed by the procedure.

```
> GaussianElimination := proc(A::matrix(rational),rank::name)
    local m,n,i,j,B,r,c,t;
    n := linalg[rowdim](A); # the number of rows of the matrix
    m := linalg[coldim](A); # the number of columns of the matrix
    B := array(1..n,1..m); # local matrix for transformation
    for i to n do for j to m do B[i,j] := A[i,j] od od;
    r := 1; # r and c are row and column indices
    for c to m while r <= n do
      for i from r to n while B[i,c] = 0 do od;
          # search for a pivot in column c
      if i <= n then
        if i <> r then # interchange row i with row r
        for j from c to m do
          t := B[i,j]; B[i,j] := B[r,j]; B[r,j] := t
        od
        fi;
        for i from r+1 to n do
          if B[i,c] <> 0 then t := B[i,c]/B[r,c];
          for j from c+1 to m do B[i,j] := B[i,j]-t*B[r,j] od;
          B[i,c] := 0
          fi
        od;
        r := r+1 # go to next row
      fi
    od; # go to next column
    if nargs>1 then rank := r-1 fi;
    eval(B)
  end:
```

Our own procedure works similar to *gausselim*.

```
> GaussianElimination(A,'r');
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

To look at the source code in library routines, if it is admisible. Here we take the *gausselim* procedure. Compare with *GaussianElimination*.

```
> interface(verboseproc=2); # print the body of all procedures
  print(linalg[gausselim]);
  interface(verboseproc=1);
   # (default) print the body of user-defined procedures
```

$$\textbf{proc}(AA,\ rank,\ det)\ \ldots\ \textbf{end}$$

Solving linear equation systems $Ax=b$

(1) Row reduction variant

To use one of procedures *gausselim* or *GaussianElimination*.

```
> A := matrix(3,3, (i,j) -> 1/(i+j-1));
  i := 'i'; j := 'j';
  b := vector(3, [seq(sum(1/(i+j-1), j=1..3),i=1..3)]);
  x := vector(3);
  AE := matrix(3,4);
  for i to 3 do for j to 3 do AE[i,j] := A[i,j] od od:
  for i to 3 do AE[i,4] := b[i] od:
      # AE := concat(A,b);
  AE: %=evalm(%);
  AT := gausselim(AE,'r');
  r;
  if r=3 then
    x[3] := AT[3,4]/AT[3,3]; # backward solve
    for i from 2 by -1 to 1 do
      xh := AT[i,4];
      for j from i+1 to 3 do xh := xh-x[j]*AT[i,j] od;
      x[i] := xh/AT[i,i]
    od:
  fi:
  x:  %=eval(%);
```

$$A := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

$$i := i$$

$$j := j$$

$$b := \left[\frac{11}{6}, \frac{13}{12}, \frac{47}{60}\right]$$

$$x := \text{array}(1..3, [])$$

$$AE := \text{array}(1..3, 1..4, [])$$

$$AE = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{11}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{13}{12} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{47}{60} \end{bmatrix}$$

$$AT := \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{11}{6} \\ 0 & \frac{1}{12} & \frac{1}{12} & \frac{1}{6} \\ 0 & 0 & \frac{1}{180} & \frac{1}{180} \end{bmatrix}$$

$$3$$

$$x = [1, 1, 1]$$

(2) Using the Maple routine *linsolv*

```
> linsolve(A,b,'r');
```

$$[1, 1, 1]$$

General, to find the matrix X which solves the matrix equation $AX = B$. For the identity matrix B the result is the inverse of A.

```
> B := diag(1,1,1):
  AA := linsolve(A,B): Inverse(A)=%;
  evalm(AA &* A); # control
```

$$\text{Inverse}(A) = \begin{bmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{bmatrix}$$

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

4.2 Numerical Computation

Floating point numbers in Maple can be input either as decimal numbers or directly using the **Float** function.

$$\textbf{Float(m,e)} = m * 10^e$$

where the mantissa m is an integer of any size, but the exponent e is restricted to be a machine size integer, typically 31 bits.

```
> restart;
  x := 3.1;
  x := Float(31,-1);
```

$$x := 3.1$$
$$x := 3.1$$

It works for mantisse as fixed point number too.

```
> x := 1.2E-3;
  y := Float(12,-3);
  z := Float(1.12345678912,-3); # no conversion, decimal point is showing
  eval(z);
  evalf(z,10); # software float, 10 (significant) digits
```

$$x := .0012$$
$$y := .012$$
$$z := \mathrm{Float}(1.12345678912,\ -3)$$
$$\mathrm{Float}(1.12345678912,\ -3)$$
$$.001123456789$$

The *op* function can be used to extract the mantissa and exponent of a floating point number.

```
> z1 := Float(12,-3);
  op(1,z1);
  op(2,z1);
  z2 := Float(12.1,-3);
  op(1,z2);
  op(2,z2);
  z3 := 12.1E-3;
  op(1,z3);
  op(2,z3);
```

$$z1 := .012$$
$$12$$
$$-3$$
$$z2 := \mathrm{Float}(12.1,\ -3)$$
$$12.1$$
$$-3$$
$$z3 := .0121$$
$$121$$
$$-4$$

The floating point constant 0.0 is treated specially and is simplified to 0, i.e. the integer 0 automatically.

```
> z := Float(0,0);
  evalf(z);
```

$$z := 0$$
$$0$$

Floating point arithmetic

It is done in decimal with rounding. The precision used is controlled by the global variable *Digits* which has 10 as its default value. It can be set to any value however.
The *evalf* function is used to evaluate an exact symbolic constant to a floating point approximation.

```
> z := Float(1123456789123,-12); # conversion
  Digits := 10:  # software float
  evalf(z);
  evalf(z,10);
  evalhf(z); # hardware float, 14 digits
  z:  evalf(%,3); # 3 digits
```

$$z := 1.1234567891230$$
$$1.123456789$$
$$1.123456789$$
$$1.123456789123000$$
$$1.12$$

```
> Digits := 25:
  sin(1.0);
  sin(1);
  evalf(%);
```

$$.8414709848078965066525023$$
$$\sin(1)$$
$$.8414709848078965066525023$$

Applications

(1) Random number generator

The built-in Maple function *rand* does not return a random number in the given range. It returns instead a Maple procedure (a random number generator) which when called returns random integers in the given range.

```
> rand(); # returns a 12 digit random integer >=0
  rand(1..6); # random numbers in [1..6]
  r1 := rand(1..6):
  r1();
  `Dice:`; seq(r1(), i=1..6 );
  r2 := rand(6):  # = random(0..5)
  r2();
```

$$427419669081$$

proc()
  local t;
  global $_seed$;
    $_seed := \mathrm{irem}(427419669081 * _seed,\ 999999999989)\,;\ t := _seed\,;\ \mathrm{irem}(t,\ 6) + 1$
  end

$$3$$
$$Dice:$$
$$4,\ 6,\ 5,\ 3,\ 6,\ 3$$
$$1$$

Here is a uniform random number generator that generates uniform random numbers with exactly 6 decimal digits precision in the range [0,1).

```
> UniformInteger := rand(0..10^6-1):
  UniformFloat := proc()
      Float(UniformInteger(),-6)
  end:
  seq( UniformFloat(), i=1..6 );
```

$$.037409, .952655, .487163, .490457, .594160, .571674$$

(2) Computing elementary and special functions

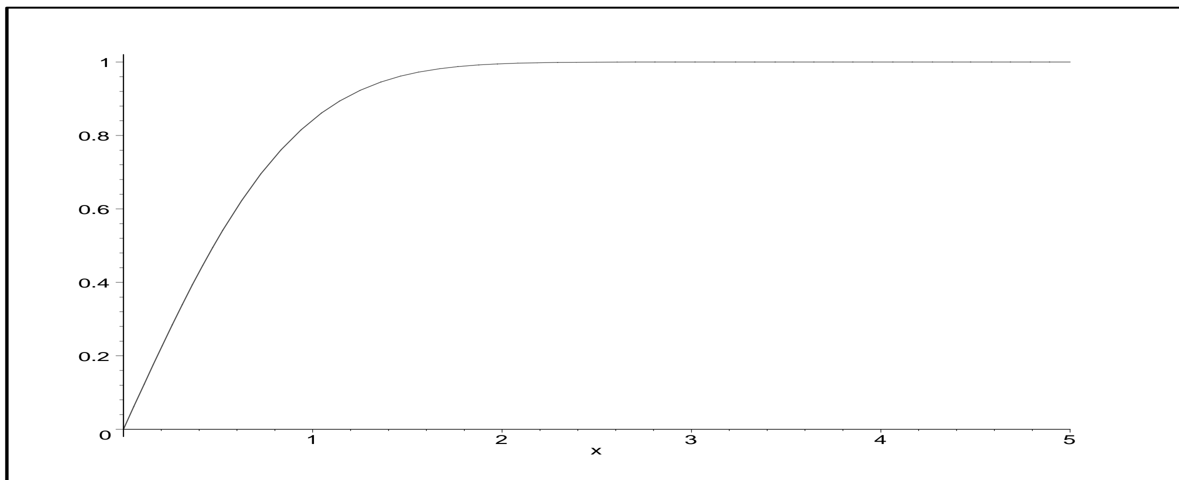
Maple knows about the elementary functions and many special functions such as $J_v(x)$, $\Gamma(x)$, and $\zeta(x)$. In Maple these are $BesselJ(v,x)$, $GAMMA(x)$, and $Zeta(x)$ respectively. They are all computed to high precision by summing various series. The model of floating point arithmetic used is that the relative error of the result is less than $10^{1-Digits}$.
This is a stronger model than that used by hardware implementations of floating point arithmetic (14 Digits) and it requires that intermediate calculations to be done at higher precision to obtain such accurate results.

Summing a **Taylor series** to compute the error function $erf(x)$.

```
> restart;
  2/sqrt(Pi)*Int(exp(-t^2),t=0..x);
```

$$2\,\frac{\int_0^x e^{(-t^2)}\,dt}{\sqrt{\pi}}$$

```
> # smartplot
> plot(erf(x),x=0..5); # plot(erf,0..5)
```



For small argument x we can make use of the Taylor series for $erf(x)$ about $x = 0$.

```
> sqrt(Pi)/2*erf(x) = Sum((-1)^n*x^(2*n+1)/(n!*(2*n+1)),n=0..infinity);
```

$$\frac{1}{2}\sqrt{\pi}\,\mathrm{erf}(x) = \sum_{n=0}^{\infty} \frac{(-1)^n\, x^{(2\,n+1)}}{n!\,(2\,n+1)}$$

Expand the Taylor series

```
> expand(sum((-1)^n*x^(2*n+1)/(n!*(2*n+1)),n=0..4),x);
```

$$x - \frac{1}{3}x^3 + \frac{1}{10}x^5 - \frac{1}{42}x^7 + \frac{1}{216}x^9$$

Computing $erf(x)$ for small x, $-1 < x < 1$, with no error checking.

```
> ErfSmall := proc(x)
    local n,xx,x2,result,term,sumold,sumnew;
    xx := evalf(x);
      # evaluate the input at Digits precision
    Digits := Digits+2; # add some guard digits
    sumold := 0;
    term := xx;
    sumnew := xx;
    x2 := xx^2;
    for n from 1 while sumold<>sumnew do
      sumold := sumnew;
      term := -term*x2/n;
      sumnew := sumold+term/(2*n+1);
    od;
    result := evalf(2/sqrt(Pi)*sumnew);
    Digits := Digits-2;
    evalf(result) # round the result to Digits precision
  end:
> ErfSmall(0.1);
  ErfSmall(0.9);
```

$$.1124629160$$
$$.7969082124$$

For large x, this routine will be slow, inaccurate (no convergence to 1) and inefficient. We have to use higher precision and an other algorithm.

```
> for x from 1 to 10 do ErfSmall(x) od;
```

$$.8427007930$$
$$.9953222650$$
$$.9999779093$$
$$1.000000312$$
$$1.001219948$$
$$205.0250992$$
$$553819.2908$$
$$-.1224648312\,10^{15}$$
$$.3170538374\,10^{22}$$
$$-.1369812404\,10^{29}$$

(3) Iterative algorithms for finding roots of equation

Let us consider writing a **Newton iteration** to compute the square root of an integer number to 48 Digits.
Recall what is the Newton iteration for solving an equation ***f(x)=0***.

```
> restart;
  x[k+1] = x[k] - f(x[k])/fs(x[k]); # fs=f'
```

$$x_{k+1} = x_k - \frac{\mathrm{f}(x_k)}{\mathrm{fs}(x_k)}$$

In our case, to compute $\sqrt{a}$ we want to solve $f(x) = x^2 - a = 0$.

```
> x[k+1] = 1/2*(x[k] + a/x[k]); # x[0] start point
```

$$x_{k+1} = \frac{1}{2}\,x_k + \frac{1}{2}\,\frac{a}{x_k}$$

The following routine will do the job. We take a relative error for iteration break.
We have put in the *print* statement to show the result of each iteration.

```
> SqrtNewton := proc(a::integer)
   local xk,xkm1,k;
   if a < 0 then ERROR(`square root of a negative integer`) fi;
   Digits := 50; # add some guard digits
   k := 0; # counter
   xkm1 := 0;
   xk := evalf(a/2); # initial floating point approximation
   print(xk);
   while abs(xk-xkm1) > abs(xk)*1E-48 do
     k := k+1;
     xkm1 := xk;
     xk := (xk+a/xk)/2;
     print(xk);
   od;
   Digits := 48;
   print(`k=`,k);
   evalf(xk); # round the result to 48 Digits
  end:
> SqrtNewton(2);
```

$$1.$$
$$1.5000000000000000000000000000000000000000000000000$$
$$1.4166666666666666666666666666666666666666666666667$$
$$1.4142156862745098039215686274509803921568627450981$$
$$1.4142135623746899106262955788901349101165596221158$$
$$1.4142135623730950488016896235025302436149819257762$$
$$1.4142135623730950488016887242096980785696718753773$$
$$1.4142135623730950488016887242096980785696718753770$$
$$k =,\ 7$$
$$1.41421356237309504880168872420969807856967187538$$

It is known that a Newton iteration will converge quadratically provided $f'(x_k)$ is not close to zero and x_k is sufficiently close to a root of f, as illustrated in the above example. For very high precision though, i.e. Digits>1000, one does not want to do every iteration at full precision as the early steps are not accurate to full precision. Why do all that work when you are only getting a few digits correct?

Modify the Newton iteration used in the *SqrtNewton* procedure appropriately so that it doubles the number of Digits at each step. How much faster does this make the iteration run?

Iteration at full precision

```
> SqrtNewton1 := proc(a::integer)
    local k,xk, xkm1;
    if a < 0 then ERROR(`square root of a negative integer`) fi;
    Digits := 1537; # add one guard digit
    xkm1 := 0;
    xk := evalf(a/2); # initial floating point approximation
    print(xk);
    k := 0;
    while k <=11 do
      xkm1 := xk;
      xk := (xk+a/xk)/2;
      k := k+1;
      # print(xk);
    od;
    Digits := 1536;
    evalf(xk); # round the result to 1536 Digits
  end:
> SqrtNewton1(2);
```

1.

1.41421356237309504880168872420969807856967187537694807317667973799\
907324784621070388503875343276415727350138462309122970249 24\
....
989175409881593486400834570851814722318142040704265090565 32\
3333984364578657967965192672923998753666172159825788603

Double the number of digits at each step

It takes less than the half of time of *SqrtNewton1*.

```
> SqrtNewton2 := proc(a::integer)
    local D,k,xk, xkm1;
    if a < 0 then ERROR(`square root of a negative integer`) fi;
    D := 1;
    Digits := D+1; # add one guard digit
    xkm1 := 0;
    xk := evalf(a/2); # initial floating point approximation
    # print(Digits);
    print(xk);
    k := 0;
    while k <=11 do
      xkm1 := xk;
```

```
      D := 2*D;
      Digits := D+1;
      if k=1 then D:=3; Digits := D+1 fi;
      xk := (xk+a/xk)/2;
      k := k+1;
      # print(Digits);
      # print(xk);
    od;
    Digits := 1536;
    evalf(xk); # round the result to 1536 Digits
  end:
> SqrtNewton2(2); # the same result like above
```

1.
1.41421356237309504880168872420969807856967187537694807317667973 79\
....
33339843645786579679651926729239987536661721598257886 03

Because Maple's floating point model is implemented in software, it is much slower than hardware floating point arithmetic.
Maple also has a function called *evalhf* for evaluating in hardware floating point arithmetic. This function uses the builtin C library routines for floating point arithmetic. Consequently it is much faster than Maples software floats, but it is still slower than hardware floats because it is not compiled.

4.3 Computing with Polynomials

Some basic functions for computing with polynomials are *indets, degree, collect, expand, factor, coeff, coeffs,* and *divide.*
There are many other functions for computing with polynomials in Maple, including facilities for polynomial division, greatest common divisors, resultants, etc. Maple can also factor polynomials over different number fields including finite fields. See the on-line help for *polynom* for a list of facilities.

Summarizing routines

```
> restart;
  p := x^3-(x-3)*(x^2+x)+1;
  q := x*y+a*x*y+y*x^2-a*y*x^2+x+a*x;
  q1:= x*y+a*x*(1+y)+y*x^2*(1-a)+x; # q1=q
```

$$p := x^3 - (x-3)(x^2+x) + 1$$

$$q := x\,y + a\,x\,y + y\,x^2 - a\,y\,x^2 + x + a\,x$$

$$q1 := x\,y + a\,x\,(1+y) + y\,x^2\,(1-a) + x$$

(1) Return a set of all polynomial variables

```
> indets(p);
  indets(q);
```

$$\{x\}$$

$$\{x, y, a\}$$

(2) Degree of a polynomial

The *degree* function computes the degree of a univariate polynomial in one variable, for multivariate polynomials the *total degree*, and the degree in a specific variable or of special components.

```
> degree(p);
  degree(q);
  degree(q,x);
```

$$3$$
$$4$$
$$2$$

It depends on the shape of polynomial.

```
> seq(degree(t,indets(q)),t=q);
  seq(degree(t,indets(q1)),t=q1);
```

$$2,\ 3,\ 3,\ 4,\ 1,\ 2$$
$$2,\ 3,\ 4,\ 1$$

(3) Collect coefficients of like powers

```
> collect(p,x);
  collect(q,[x,y],recursive);
```

$$2\,x^2 + 3\,x + 1$$
$$y\,x^2\,(1-a) + ((1+a)\,y + 1 + a)\,x$$

(4) Expanding a polynomial

```
> expand(p);
  expand(q1);
```

$$2\,x^2 + 3\,x + 1$$
$$x\,y + a\,x\,y + y\,x^2 - a\,y\,x^2 + x + a\,x$$

(5) Factor a univariate and multivariate polynomial

```
> factor(p);
  factor(q);
  factor(q1);
```

$$(x+1)\,(2\,x+1)$$
$$-x\,(-y - a\,y - x\,y + a\,x\,y - 1 - a)$$
$$-x\,(-y - a\,y - x\,y + a\,x\,y - 1 - a)$$

(6) Coefficients

The *coeffs* function insists on having the input polynomial expanded because you cannot compute the coefficient(s) otherwise. If you want to compute the coefficients of a polynomial in some variable, you should always expand the polynomial in that variable first. You can use the *expand* function but this expands the polynomial in all variables. Alternatively, you can use the *collect* function.

```
> coeffs(p); # first p in expanded form

Error, invalid arguments to coeffs
```

```
> coeffs(expand(p));
  coeffs(q,x);
```

$$1,\ 2,\ 3$$
$$y - a\,y,\ y + a\,y + 1 + a$$

Extract a coefficient or all coefficients of a polynomial.

```
> coeff(q,a,1);
  coeff(q1,a,1);
  seq(coeff(p,x,i),i=0..degree(p));
```

$$x\,y - y\,x^2 + x$$
$$x\,(1+y) - y\,x^2$$
$$1,\ 3,\ 2,\ 0$$

(7) Polynomial long division

The call $divide(a, b)$ returns true if and only if the polynomial b divides the polynomial a with no remainder.

```
> divide(p,x+1);
  divide(p,x+1/2,'r');
  r;
  divide(q,x^2);
```

$$true$$
$$true$$
$$2\,x+2$$
$$false$$

(8) Generating a polynomial by a list

The *convert* function is used to convert an expression from one form to another.

```
> l := [1,2,3];
  for i from 1 to 3 do l[i] := l[i]*x^(3-i) od:
  l;
  convert(l,`+`);
```

$$l := [1,\ 2,\ 3]$$
$$[x^2,\ 2\,x,\ 3]$$
$$x^2 + 2\,x + 3$$

Applications

Computing with polynomials and also rational functions is Maple's forte.

(1) Computing a norm

Here is a program that computes the Euclidean norm of a polynomial.
I.e. given the polynomial $p(x)$ in x.

```
> restart;
  p := sum(a[i] * x^i, i=0..n);
```

$$p := \sum_{i=0}^{n} a_i\,x^i$$

It computes

```
> sqrt(sum(a[i]^2, i=0..n));
```

$$\sqrt{\sum_{i=0}^{n} {a_i}^2}$$

```
> EuclideanNorm := proc(q)
    sqrt( convert( map( x -> x^2, [coeffs(expand(q))] ),`+`) )
  end;
```

$$EuclideanNorm := \mathbf{proc}(q)\,\mathrm{sqrt}(\mathrm{convert}(\mathrm{map}(x \to x^2,\ [\mathrm{coeffs}(\mathrm{expand}(q))]),\ `+`))\ \mathbf{end}$$

Reading this one liner inside out, the input an univariate polynomial is first expanded. The *coeffs* function returns a sequence of the coefficients which we have put in a list. Each element of the list is squared by *map* operator yielding a new list. Then the list of squares is converted to a sum. At least apply the square root function.

```
> p := x^3 - (x-3)*(x^2+x) + 1;
  expand(p);
  coeffs(expand(p));
  EuclideanNorm(p):
  % = evalf(%);
```

$$p := x^3 - (x-3)(x^2+x) + 1$$

$$2x^2 + 3x + 1$$

$$3, 2, 1$$

$$\sqrt{14} = 3.741657387$$

The *EuclideanNorm* procedure works for multivariate polynomials in the sense that it computes the square root of the sum of the squares of the numerical coefficients.

```
> p := u*x^2 + y^2 + v;
  EuclideanNorm(p);
```

$$p := u\,x^2 + y^2 + v$$

$$\sqrt{3}$$

However, you may want to view this polynomial as a polynomial in x, y, whose coefficients are symbolic coefficients in u, v.

We really want to be able to tell the *EuclideanNorm*1 routine what the polynomial variables are. We can do that by specifying an additional optional parameter.

```
> EuclideanNorm1 := proc(a,w::{name,set(name),list(name)})
    if nargs = 1 then
         sqrt( convert( map( x -> x^2, [coeffs(expand(a))]),`+`) )
      else
         sqrt( convert( map( x -> x^2, [coeffs(expand(a),w)]),`+`) )
    fi
  end:
> EuclideanNorm1(p,{x,y});
  EuclideanNorm1(p,{u,v});
```

$$\sqrt{u^2 + 1 + v^2}$$

$$\sqrt{x^4 + 1 + y^4}$$

The type *name*, *set*(*name*), *list*(*name*) means that the 2nd parameter w may be a single variable, or a set of variables, or a list of variables.

Notice that the *coeffs* function itself accepts this argument as a 2nd optional argument.

Finally, our routine doesn't ensure that the input is a polynomial. Let's add this.

```
> EuclideanNorm2 := proc(a,w::{name,set(name),list(name)})
    if nargs = 1 then
      if not type(a,polynom) then
        ERROR(`1st argument is not a polynomial`,a) fi;
      sqrt( convert( map( x -> x^2, [coeffs(expand(a))]),`+`) )
      else
      if not type(a,polynom(anything,w)) then
        ERROR(`1st argument is not a polynomial in`,w) fi;
```

```
    sqrt( convert( map( x -> x^2, [coeffs(expand(a),w)]),`+`) )
  fi
end:
```

(2) The type *polynom*

It has the following general syntax

polynom(R,X)

which means a polynomial whose all coefficients are of the same type R in the variables X.
An example is $polynom(rational, x)$ which specifies a univariate polynomial in x with rational coefficients, i.e. a polynomial in $Q[x]$. If R and X are not specified, the expression must be a polynomial in all its variables.

```
> q2 := 2*x^2-1;
  q3 := 1.5*a*x+1.0;
  q4 := 1/2*x^3-3*x^2;
```

$$q2 := 2\,x^2 - 1$$
$$q3 := 1.5\,a\,x + 1.0$$
$$q4 := \frac{1}{2}\,x^3 - 3\,x^2$$

```
> whattype(p); # polynomial=sum
  type(p,polynom);
  type(q2,polynom(float,x)); # integer coefficients
  type(q3,polynom);
  type(q3,polynom(float,{x,a}));
  type(q4,polynom(rational,x));
  type(q4,polynom(anything,{x,b}));
```

$$+$$
$$true$$
$$false$$
$$true$$
$$true$$
$$true$$
$$true$$

We include here a simple program for computing the greatest common divisor **GCDpoly** of two univariate polynomials a, b using the Euclidean algorithm.
Note, Maple does not use the Euclidean algorithm because this turns out to be very inefficient for polynomials of high degree.
However, this example summarizes many of the things we have mentioned so far.

```
> GCDpoly := proc(a,b,x)
   local c,d,r;
   if not type(x,name) then
     ERROR(`3rd argument (variable) must be a name`)
   fi;
   if not ( type(a,polynom(anything,x)) and
     type(b,polynom(anything,x))) then
       ERROR(`1st and 2nd arguments must be polynomials in `,x)
   fi;
```

```
    c := collect(a,x,recursive,normal);
    d := collect(b,x,recursive,normal);
    while d <> 0 do r := rem(c,d,x); c := d; d := r od;
    if c <> 0 then # Make c monic
      c := collect(c/lcoeff(c,x),x,recursive,normal);
    fi;
    c
  end:
```

The use of *collect* here is used to simplify the coefficients of a polynomial, in particular, to recognize the zero polynomial.

The call $collect(a, x, recursive, normal)$ means group together the coefficients of like powers of x, and apply the *normal* function to them to simplify them.

```
> normal(x^2-(x+1)*(x-1)-1); # normal function
```

$$0$$

```
> lcoeff(2*x^2-x-3,x); # leading coefficient of a polynomial
```

$$2$$

```
> a := x^4-6*x^2+8*x-3;
  b := x^4+3*x^3-3*x^2-7*x+6;
  qq := GCDpoly(a,b,x);
  factor(qq);
```

$$a := x^4 - 6\,x^2 + 8\,x - 3$$

$$b := x^4 + 3\,x^3 - 3\,x^2 - 7\,x + 6$$

$$qq := 3 + x^2 - 5\,x + x^3$$

$$(x + 3)\,(x - 1)^2$$

(3) Polynomials over finite fields

To illustrate facilities for computing with polynomials over finite fields, we conclude with a program to compute the **first primitive trinomial** of degree n over $GF(2)$ if one exists.

That is, we want to find an irreducible polynomial a of the form $x^n + x^m + 1,\ m \geq 1$, such that x is a primitive element in $GF(2)[x]/(a)$.

The *iquo* function computes the integer quotient of two integers.

```
> trinomial := proc(n::integer)
    local i,t;
    for i to iquo(n+1,2) do
      t := x^n+x^i+1;
      if Primitive(t) mod 2 then RETURN(t) fi;
    od;
    FAIL
  end:
> Primitive(x^4+x+1) mod 2;
    # univariate polynomial a over the integers mod p is "primitive"
```

$$true$$

```
> a := x^4+x^3+x^2+x+1;
  Primitive(a) mod 2;
  alpha := -1/2+sqrt(5)/2;
  a1 := (x^2+(1+alpha)*x+1)*(x^2-alpha*x+1); # factorization of a
  a2 := collect(a1,x);
  simplify(a2);
```

$$a := x^4 + x^3 + x^2 + x + 1$$

$$\mathit{false}$$

$$\alpha := -\frac{1}{2} + \frac{1}{2}\sqrt{5}$$

$$a1 := (x^2 + (\frac{1}{2} + \frac{1}{2}\sqrt{5})\,x + 1)\,(x^2 - (-\frac{1}{2} + \frac{1}{2}\sqrt{5})\,x + 1)$$

$$a2 := x^4 + x^3 + (2 + (\frac{1}{2} + \frac{1}{2}\sqrt{5})\,(\frac{1}{2} - \frac{1}{2}\sqrt{5}))\,x^2 + x + 1$$

$$x^4 + x^3 + x^2 + x + 1$$

```
> trinomial(4);
  trinomial(5);
  trinomial(6);
  trinomial(7);
  trinomial(8);
```

$$x^4 + x + 1$$

$$x^5 + x^2 + 1$$

$$x^6 + x + 1$$

$$x^7 + x + 1$$

$$\mathit{FAIL}$$

4.4 Reading and Saving Procedures: read and save

You can write one or two line Maple programs or procedures interactively. But for larger programs you will want to save them in a file of type *txt* or *m* (extension). The filename must be enclosed in quotation marks " " or ' ' and usually with path declaration. Typically, one would use an editor to write the source code and save it into a file.

Saving

You can save your Maple procedures or any formulas that you have computed in your Maple session in a file from inside Maple using the **save** statement.

Syntax **save *f1, f2, ..., filename;***

save *f1, f2, ..., filename.txt;*

This saves the values of the variables (names) $f1,\ f2, ...$ in text format in the file *filename*. Specified variables will be written into filename as a sequence of assignment statements. You can also save Maple data in internal or language format or the so called ***m*** format. This format is more compact and can be read faster by Maple. One simply appends *.m* to the filename. This saves the values of $f1,\ f2, ...$ in the file *filename.m*.

save *f1, f2, ..., filename.m;*

(1) Saving two procedures together

Useually it save it in a file of type m.

```
> # Maximum function MAX2 for names without duplicates
  restart;
> flat := proc(y,f)
    if type(y,function) and op(0,y) = f then op(y) else y fi
  end:
> MAX2 := proc()
    local a;
    a := {args};
    a := map( flat, a, MAX2 ); # unnest all nested MAX2 calls
    'MAX2'( op(sort([op(a)])) );
  end;
> save flat,MAX2,`D:/Neundorf/Maple/program/max2.m`;
```

(2) Saving one procedure

```
> MAX21 := proc()
    local a,flat;
    flat := proc(y,f)
      if type(y,function) and op(0,y) = f then op(y) else y fi
    end;
    a := {args};
    a := map( flat, a, MAX21 ); # unnest all nested MAX21 calls
    'MAX21'( op(sort([op(a)])) );
  end:
> save MAX21,`D:/Neundorf/Maple/program/max21.m`;
```

(3) Saving a program

```
> restart;
  testp1 := proc()
    restart;
    with(linalg):
    A := diag(1,5,10);
    max(1,4,2,A[2,2])
  end;
  testp1();
```

```
Warning, `A` is implicitly declared local
```

testp1 :=

$$\textbf{proc}()\,\textbf{local}\,A;\ restart\,;\ \text{with}(linalg)\,;\ A := \text{diag}(1,\,5,\,10)\,;\ \max(1,\,4,\,2,\,A_{2,2})\,\textbf{end}$$

```
Warning, new definition for norm
Warning, new definition for trace
```

$$5$$

```
> save testp1,`D:/Neundorf/Maple/program/testp1.m`;
```

(4) Saving the Maple worksheet

Use the *FileMenu* and *Save* in the editor.

Reading

A program can be read into Maple using the **read** command.

read *filename;*

read *filename.txt;*

read *filename.m;*

For example, if we have written a Maple procedures $flat$, $MAX2$ in file $max2.m$ and $MAX21$ in file $max21.m$.
In Maple we read this file into Maple. The file with extension $.m$ are read and executed, but the statements are not echoed automatically to the display. It does i.e. the following *print* function.

```
> restart;
  read `D:/Neundorf/Maple/program/max2.m`;
> print(flat,MAX2); # control the file content
```

proc(y, f) **if** type$(y, \mathit{function})$ **and** op$(0, y) = f$ **then** op(y) **else** y **fi end**,

proc()
  local a;
    $a := \{\text{args}\}$; $a := \text{map}(\mathit{flat}, a, \mathit{MAX2})$; '$\mathit{MAX2}$'(op(sort([op($a$)])))
  end

```
> MAX2(a,a,b,c,a);
```

$$\text{MAX2}(a, b, c)$$

```
> read `D:/Neundorf/Maple/program/max21.m`;
> print(MAX21); # control the file content
```

proc()
  local a, flat;
    flat := **proc**(y, f) **if** type$(y, \mathit{function})$ **and** op$(0, y) = f$ **then** op(y) **else** y **fi end**;
    $a := \{\text{args}\}$;
    $a := \text{map}(\mathit{flat}, a, \mathit{MAX21})$;
    '$\mathit{MAX21}$'(op(sort([op(a)])))
  end

```
> MAX21(a,a,b,c,a);
```

$$\text{MAX21}(a, b, c)$$

```
> restart;
  a := 3;
  b := a^2+1;
  c := sin(a);
  d := 'd';
```

$$a := 3$$
$$b := 10$$
$$c := \sin(3)$$
$$d := d$$

```
> save a,b,c,d,"D:/Neundorf/Maple/program/erg1";

Warning, unassigned variable `d` in save statement
```

```
> save a,b,c,"D:/Neundorf/Maple/program/erg1.txt";
  save a,b,c,"D:/Neundorf/Maple/program/erg1.m";
```

You can then read them back in to Maple using the **read** command.

```
> read "D:/Neundorf/Maple/program/erg1"; # with display
```

$$a := 3$$
$$b := 10$$
$$c := \sin(3)$$
$$d := d$$

```
> read "D:/Neundorf/Maple/program/erg1.txt"; # with display
```

$$a := 3$$
$$b := 10$$
$$c := \sin(3)$$

```
> read "D:/Neundorf/Maple/program/erg1.m";
  a; b; c;
  print(a,b,c);
```

$$3$$
$$10$$
$$\sin(3)$$
$$3,\ 10,\ \sin(3)$$

4.5 Debugging Maple Programs

Two debugging tools

(1) The simplest debugging tool is the printlevel facility.

printlevel is a global variable that is initially assigned 1.
If you set it to a higher value, a trace of all assignments, procedure entry and exits are printed. The higher the value of printlevel, the more levels of procedure execution that will be traced.

Examples of this tool have already been given in this document. We mention here one further tool. If you set the printlevel variable to 3 or higher, then if a run-time error occurs, Maple will print a stack trace of the calling sequence at the time the error occurs. Specifically, it will print the arguments to all procedures currently being executed, and the values of the local variables and the statement being executed in the procedure where the error occured.

```
> restart;
  f := proc(x) local y; y := 1; g(x,y); end:
  g := proc(u,v) local s,t; s := 0; t := v/s; s+t end:
  printlevel := 4:
  f(3);

f called with arguments: 3
# (f,2): g(x,y)
g called with arguments: 3, 1
# (g,2): t := v/s;
Error, (in g) division by zero
locals defined as: s = 0, t = t
```

(2) Trace function

Often, however, the output from the printlevel facility will be too much. The *trace* function (*debug* is a synonym) allows you to trace the execution of the specified functions only.

```
>  restart;
   f := proc(x) local y; y := x * 2; sin(y)/4 end:
>  trace(f);
   f(3);
```

$$f$$

```
{--> enter f, args = 3
```

$$y := 6$$

$$\frac{1}{4}\sin(6)$$

```
<-- exit f (now at top level) = 1/4*sin(6)}
```

$$\frac{1}{4}\sin(6)$$

```
>  f(3):  # without trace, only enter/exit lines

{--> enter f, args = 3
<-- exit f (now at top level) = 1/4*sin(6)}
>  untrace(f);
```

$$f$$

4.6 Interfacing with other Maple Facilities

We have shown how one can tell Maple properties about a function by coding them as a procedure.

```
>  restart;
   QUARTERNION1 := proc(a,b,c,d)
       a + b*'i' + c*'j' + d*'k' # return value displayed
   end:
   QUARTERNION1(2,3,-4,1);
   print(QUARTERNION1(alpha,beta,gamma,delta));
```

$$2 + 3\,i - 4\,j + k$$

$$\alpha + \beta\,i + \gamma\,j + \delta\,k$$

Here we have defined a printing procedure or subroutine. This routine is called once for each different QUARTERNION function in a result from Maple prior to displaying the result. We see, how to *pretty print* a function. Because the return value is displayed, it is generally not necessary to write the additional print command.

```
>  `print/QUARTERNION2` := proc(a,b,c,d)
       a + b*'i' + c*'j' + d*'k'
   end:
   QUARTERNION2(2,3,-4,1);
   print(QUARTERNION2(alpha,beta,gamma,delta));
```

$$2 + 3\,i - 4\,j + k$$

$$\alpha + \beta\,i + \gamma\,j + \delta\,k$$

Further, we shall mention that you can define how to *pretty do* a function in other cases. Suppose instead you wish to teach Maple to differentiate a formula involving f. You may want to teach Maple how to evaluate f numerically so that f can be plotted, or how to simplify expressions involving f etc.
What do you need to do? Many Maple routines have interfaces that allow you to teach these routines about your f function.

These include *evalf, expand, diff, combine, simplify, series,* etc.

Further examples

(1) Evaluate a function numerically

One can also tell Maple how to evaluate your own function f numerically by defining a Maple procedure **'evalf/f'**, such that '$evalf/f$'(x) computes $f(x)$.
For example, suppose we wanted to use the Newton iteration shown earlier for computing the square root of a numerical value. Our function might look like the following.

```
> restart;
  `evalf/SQRT` := proc(a)
   local x,xk,xkm1;
   x := evalf(a); # evaluate the argument in floating point
   if not type(a,numeric) then RETURN( SQRT(x) ) fi;
   if x<0 then ERROR(`square root of a negative number`) fi;
   Digits := Digits + 3; # add some guard digits
   xkm1 := 0;
   xk := evalf(x/2); # initial floating point approximation
   while abs(xk-xkm1) > abs(xk)*10^(-Digits) do
     xkm1 := xk;
     xk := (xk + x/xk)/2;
   od;
   Digits := Digits - 3;
   evalf(xk); # round the result to Digits precision
  end:
```

Built in function *sqrt*

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

$$\sqrt{3}$$
$$1.732050808$$

Our *pretty* $evalf$ function

```
> SQRT(3);
  evalf(SQRT(3));
```

$$\mathrm{SQRT}(3)$$
$$1.732050808$$

```
> Digits := 50;
  x := SQRT(3);
  evalf(x);
  Digits := 10:
```

$$Digits := 50$$
$$x := \mathrm{SQRT}(3)$$
$$1.7320508075688772935274463415058723669428052538104$$

(2) Expanding polynomials

Suppose we want to manipulate symbolically the Chebyshev polynomials of the first kind $T_n(x)$. We can represent these in Maple as $T(n,x)$.
Recall that $T_n(x)$ satisfies the linear recurrence $T_0(x) = 1,\ T_1(x) = x,$ and $T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x)$.

```
> restart;
  with(orthopoly); # orthopoly package
```

$$[G, H, L, P, T, U]$$

```
> T(2,x); # Chebyshev polynomials
```

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

Suppose also that for a particular value of n we want to expand $T_n(x)$ out as a polynomial in x. We can tell Maple's expand function how to do this by writing the own routine **'expand/TT'**.
When *expand* sees $T(n,x)$, it will invoke *'expand/TT'*(n,x).
Hence we can write the following.

```
> `expand/TT` := proc(n,x) option remember;
   if n = 0 then 1
     elif n = 1 then x
       elif not type(n,integer) then TT(n,x) # can't do anything
         else expand(2*x*TT(n-1,x) - TT(n-2,x))
   fi
  end:
```

This routine is recursive, but because we used the remember option, the routine will compute $T(100,x)$ quite quickly.

```
> whattype(TT(4,x));
  TT(4.0,x);
  TT(4,x);
```

function
TT(4.0, x)
TT(4, x)

```
> expand(TT(4,x));
```

$$8\,x^4 - 8\,x^2 + 1$$

```
> expand(TT(24,x)); # polynomial sorted up to degree 24
```

$$8388608\,x^{24} - 50331648\,x^{22} + 132120576\,x^{20} - 199229440\,x^{18} + 190513152\,x^{16} - 120324096\,x^{14} + 50692096\,x^{12} - 14057472\,x^{10} + 2471040\,x^8 - 256256\,x^6 + 13728\,x^4 - 288\,x^2 + 1$$

```
> sort(expand(TT(30,x))); # use sort
```

$$536870912\,x^{30} - 4026531840\,x^{28} + 13589544960\,x^{26} - 27262976000\,x^{24} + 36175872000\,x^{22} - 33426505728\,x^{20} + 22052208640\,x^{18} - 10478223360\,x^{16} + 3572121600\,x^{14} - 859955200\,x^{12} + 141892608\,x^{10} - 15275520\,x^8 + 990080\,x^6 - 33600\,x^4 + 450\,x^2 - 1$$

```
> p := sort(expand(TT(100,x))):# no display
  coeff(p,x,0);
  coeff(p,x,100);
```

$$1$$
$$633825300114114700748351602688$$

(3) Differentiate a function with cain rule

To teach Maple how to differentiate a function $W(g(x))$ one writes a routine called **'diff/W'**. If the *diff* routine is called with an expression $f(x)$ which contains $W(g)$ then the *diff* routine will invoke $'diff/W'(g,x)$ to compute the derivative of $W(g)$ with respect to x, i.e $W'(g)g'(x)$.

Suppose we know that $W'(z) = W(z)/(1+W(z))$. Then we can write the following procedure which explicitly codes the chain rule.

```
> 'diff/W' := proc(g,x)
   diff(g,x)*W(g)/(1+W(g))
  end;
```

$$diff/W := \mathbf{proc}(g, x)\,\mathrm{diff}(g, x) \times \mathrm{W}(g)/(1+\mathrm{W}(g))\,\mathbf{end}$$

Hence we have

```
> diff(W(x),x);
```

$$\frac{\mathrm{W}(x)}{1+\mathrm{W}(x)}$$

```
> diff(W(x^2),x);
```

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

```
> g := x->exp(x^2);
  diff(W(g(x)),x);
```

$$g := x \rightarrow e^{(x^2)}$$

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

4.7 Calling External Programs

Often one wants to get Maple to call an external program, which might have been written in C or Fortran.

There are really two reasons we might want to do this. Obviously, if Maple cannot so something, and another program can, it may make sense to use the other program, instead of reimplementing an algorithm in Maple. The other reason is efficiency. Although Maple is efficient at symbolic computations, it is not generally efficient at machine precision numerical computations. E.g. you may want to use a Fortran library routine to compute numerical eigenvectors.

Communication of data in Maple must be done via files.

The Maple program must write the data needed by the external program out to an input file. Maple can start running the external program with the *system* command.

The external program must read the data from the input file and must write its results into an output file. After the program has finished executing, Maple reads the results from the output file back into Maple. A sketch of the Maple code to do this would be

interface(quiet=true); # suppress all auxiliary printing
writeto(input);
... # write any data into the file input

```
writeto(terminal);
interface(quiet=false);
system(...); # execute the external program
read output;
... # continue processing in Maple
```

Let us look at this more closely.
The first command, *interface(quiet = true)* turns off all diagnostics from Maple, i.e. bytes used messages, warnings, etc. so that they will not be written into the input file. This is reset by *interface(quiet = false)* after the data has been written to the input file.
The next command *writeto(input)* opens the file named *input* for writing. All Maple output from this point will go into this file. Output will typically be created by the *lprint* or the *printf* commands. The command *writeto* overwrites the file. If you want to just append some data to what is already in the file, use the command *appendto* instead of *writeto*.

After the data has been written to the file, the file is closed implicitly by resetting output back to the terminal with the command *writeto(terminal)*.

The *system* command is used to execute the external program. The exact call that the system command makes will depend on the system. Under Unix, the call might look something like

```
system(`foo < in > out`);
```

The external program *foo* is executed on the input file *in* and its results are written to the file *out*.
Note that the system command returns a value indicating whether the external program terminated normally. Under Unix, it returns 0 for normal termination.

Finally, we read the results in the *out* file back into Maple using the *read* command if the data is in Maple's format, or the *readline*, *sscanf*, or *readdata* commands described below for reading arbitrary data.

In this mode, Maple calls the external program like a subroutine. After reading the results back into Maple, Maple continues execution. This is actually a very simple means for calling an external program, and easy to debug. You can check the files to make sure that they have the data in the format expected. However, the disadvantage is that communicating data via a file of text is not the most efficient approach. In many applications this inefficiency will not matter if the work done by the external program is significant.

4.8 File Input/Output of Graphics and Numerical Data

(1) Graphic Output

The function *interface* is provided as a unique mechanism of communication between Maple and the user interface. Specifically, this function is used to set and query all variables which affect the format of the output but do not affect the computation.
Its arguments are specifies, i.e. for plots

plotdevice : The name of the plotting device.

plotoutput : Name of a file where the plot output will be stored.

plotoptions : Contain device specific options to be passed to the device driver.

Unfortunately, in Maple does not exist an implemented menu function for the direct export of graphics. Using the *interface* command you are able to store pictures like JPEG, PCX, or PS files.

```
> restart:
  with(plots):
  plot3d(sin(x)*exp(y), x=0..10,y=1..4);
> interface(plotdevice=pcx,
      plotoutput=`D:/Neundorf/Maple/program/bild1.pcx`,
      plotoptions=`width=800,height=600`);
  plot3d(sin(x)*exp(y), x=0..10,y=1..4);
> interface(plotdevice=win); # remove the standard output
```

(2) File Input/Output of Numerical Data

We wish now to discuss in more detail how to output data from Maple into a file suitable for reading by another program, and how the data should be formatted by the external program so that Maple can read it in.

In Maple the file I/O facilities now include the the routines *printf*, *sscanf* and *readline*. The routine *printf* is used for general formatted printing. % symbol begins the format specification.
It is modelled after the *printf* routine from C and is the same except that it also accepts the %a option for algebraic output.
Here is a typical example of using *printf* to print floating point data.

```
> restart;
  x := exp(1.0);
  printf(`A float in fixed point:  %f and scientific notation:  %e\n`,
      x, x );
```

$$x := 2.718281828$$

```
A float in fixed point: 2.718282 and scientific notation: 2.718282e+00
```

The *printf* function takes a string as it's first argument which specifies text to be printed and how the values, which are given as additional arguments are to be printed using % control sequences, i.e. %e control sequence. The escape sequence \n is used to print a new line. Other useful control sequences are %d and %s for printing integers and strings, and also the control sequence %a for printing a Maple algebraic expression.
See `?printf` for other options.

The *readline* routine is used to read a single line of text from a text file as a string. The *sscanf* routine can be used to scan a string and extract any numbers. It is the inverse of *printf*.
So to read in some data, one can read each line of a file using *readline* and for each line use *sscanf* to extract data.

The utility routine *writedata* has been given to write files of text which contain columns of numerical data separated by blanks or tabs.

```
> readlib(writedata):
  fd := fopen(`D:/Neundorf/Maple/program/foo2.txt`,WRITE,TEXT):
      # Explicitly open the file of type TEXT
  H1 := linalg[hilbert](3):
  print(H1);
  writedata(terminal,H1); # Printed on the user's terminal
  writedata(fd,H1); # Write the matrix
  fclose(fd); # Close the file
```

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

```
1           .5          .3333333333
.5          .3333333333 .25
.3333333333 .25         .2
```

The utility routine *readdata* has been written to read files of text which contain columns of numerical data separated by blanks or tabs. The *readdata* routine returns a list of the data, or a list of lists of the data if more than one column is specified.
Suppose the file *foo1.txt* looks like this.

```
2.1   3.2   1.2
2.4   3.1   3.4
3.9   1.1   5.5
```

Here is the result of reading in the data.

```
> readlib(readdata):  # load the readdata function from the library
  readdata(`D:/Neundorf/Maple/program/foo1.txt`,2);
      # read the first two columns of the data
```

$$[[2.1, 3.2], [2.4, 3.1], [3.9, 1.1]]$$

```
> readdata(`D:/Neundorf/Maple/program/foo2.txt`,1);
      # read the first column of the Hilbert matrix
```

$$[1., .5, .3333333333]$$

4.9 Fortran and C output

The *fortran* and *C* commands generate output of Maple formulas in a format suitable for a Fortran or C compiler.
This is useful when you have developed a formula in Maple, or perhaps a vector or matrix of formulas, and you wish to evaluate these formulas in a Fortran or C subroutine.

How do you translate the Maple formulas into Fortran or C? The Maple functions *fortran* and *C* are written for this purpose.

Here is an example. Suppose we have created the following polynomial in Maple which, by the way, is an approximation to the complementary error function $erfc(x) = 1 - erf(x)$ on the range [2,4] accurate to 5 decimal digits.

```
> restart;
  f := - 3.902704411*x + 1.890740683 - 1.714727839*x^3
    + 3.465590348*x^2 - .0003861021174*x^7 + .5101467996*x^4
    - .09119265524*x^5 + .009063185478*x^6;
```

$$f := -3.902704411\,x + 1.890740683 - 1.714727839\,x^3 + 3.465590348\,x^2 - .0003861021174\,x^7 + .5101467996\,x^4 - .09119265524\,x^5 + .009063185478\,x^6$$

Generate Fortran code

```
> fortran(f);
  fortran(f,optimized); # single precision
  fortran(f,optimized,precision=double);

      t0 = -0.3902704E1*x+0.1890741E1-0.1714728E1*x**3+0.346559E1*x**2
     # -0.3861021E-3*x**7+0.5101468E0*x**4-0.9119266E-1*x**5+0.9063185E-2*x**6

      t2 = x**2
      t3 = t2*x
      t6 = t2**2
      t14 = -0.3902704E1*x+0.1890741E1-0.1714728E1*t3+0.346559E1*t2
     # -0.3861021E-3*t6*t3+0.5101468E0*t6-0.9119266E-1*t6*x+0.9063185E-2*t6*t2

      t2 = x**2
      t3 = t2*x
      t6 = t2**2
      t14 = -0.3902704411D1*x+0.1890740683D1-0.1714727839D1*t3+0.3465590348D1*t2
     # -0.3861021174D-3*t6*t3+0.5101467996D0*t6-0.9119265524D-1*t6*x
     # +0.9063185478D-2*t6*t2
```

Generate C code

```
> readlib(C): # loaded library
  C(f);
  C(f,optimized);

      t0 = -0.3902704411E1*x+0.1890740683E1-0.1714727839E1*x*x*x
     +0.3465590348E1*x*x-0.3861021174E-3*x*x*x*x*x*x*x+0.5101467996*x*x*x*x
     -0.9119265524E-1*x*x*x*x*x+0.9063185478E-2*x*x*x*x*x*x;

      t2 = x*x;
      t3 = t2*x;
      t6 = t2*t2;
      t14 = -0.3902704411E1*x+0.1890740683E1-0.1714727839E1*t3
     +0.3465590348E1*t2-0.3861021174E-3*t6*t3+0.5101467996*t6
     -0.9119265524E-1*t6*x+0.9063185478E-2*t6*t2;
```

It is not important to know what the complementary error function is for the purpose of this example though. It is in fact related to the Normal distribution in statistics. Neither is it important here to know how we created the approximation f. We needed a rough approximation to this function in the given range because our Fortran and C libraries did not have this function built in.

For the interested reader, we used the command $chebyshev(erfc(x), x = 2..4, 10^{-5})$, to create a Chebyshev series approximation and used our **'expand/T'** routine that we wrote earlier to convert it to a polynomial.

```
> `expand/TT1` := proc(n,x) option remember;
    if n = 0 then 1
      elif n = 1 then x
        else expand(2*x*TT1(n-1,x) - TT1(n-2,x)) # without data control
    fi
  end:
  expand(TT1(3,x-3)); # control
```

$$4\,x^3 - 36\,x^2 + 105\,x - 99$$

```
> C1 := chebyshev(erfc(x),x=2..4,10^(-5));
  seq1 := NULL:
  for n from 0 to 7 do seq1 := seq1,coeff(C1,T(n,x-3)) od:
  lst1 := [seq1];
  sum1 := 0:
  for n from 0 to 7 do sum1 := sum1 + lst1[n+1]*expand(TT1(n,x-3)) od:
  sum1 := %;
```

$$\begin{aligned} C1 := {} & .0008856791120\,\mathrm{T}(0,\ x-3) - .001582100607\,\mathrm{T}(1,\ x-3) \\ & + .001128590463\,\mathrm{T}(2,\ x-3) - .0006442721728\,\mathrm{T}(3,\ x-3) \\ & + .0002941338997\,\mathrm{T}(4,\ x-3) - .0001065184703\,\mathrm{T}(5,\ x-3) \\ & + .00002984503165\,\mathrm{T}(6,\ x-3) - .6032845583\,10^{-5}\,\mathrm{T}(7,\ x-3) \end{aligned}$$

$$\begin{aligned} lst1 := [& .0008856791120,\ -.001582100607,\ .001128590463,\ -.0006442721728, \\ & .0002941338997,\ -.0001065184703,\ .00002984503165,\ -.6032845583\,10^{-5}] \end{aligned}$$

$$\begin{aligned} sum1 := {} & 1.890740683 - 3.902704410\,x + 3.465590347\,x^2 - 1.714727838\,x^3 \\ & + .5101467996\,x^4 - .09119265523\,x^5 + .009063185477\,x^6 - .0003861021173\,x^7 \end{aligned}$$

To evaluate the approximation f above efficiently, we want to write the polynomial in Horner form.

Then we want to generate Fortran code.

```
> h := convert(f,horner);
```

$$\begin{aligned} h := {} & 1.890740683 + (-3.902704411 + (3.465590348 + (-1.714727839 + \\ & (.5101467996 + (-.09119265524 + (.009063185478 - .0003861021174\,x)\,x)\,x)\,x) \\ & x)x)x \end{aligned}$$

```
> fortran(h);

      t0 = 0.1890741E1+(-0.3902704E1+(0.346559E1+(-0.1714728E1+(0.5101468E0
     # +(-0.9119266E-1+(0.9063185E-2-0.3861021E-3*x)*x)*x)*x)*x)*x)*x
```

Maple has generated two lines of Fortran code complete with continuation character since the formula is longer than one line. The floating point numbers have automatically been translated to single precision Fortran E notation and have been truncated to 7 decimal digits.

And Maple has put the result in the variable $t0$ for us. The global names $t0,\ t1,\ t2, \ldots$ are reserved for use by *fortran* for the purpose storing the assignment statements.

Let us now output C code into a file *temp*1*.c* assigned to the variable r. Remember, the C function must first be loaded into Maple.

```
> readlib(C):
  C([r=h],filename=`D:/Neundorf/Maple/program/temp1.c`);
```

If we look at the file *temp*1*.c* we find that it contains

```
r = 0.1890740683E1+(-0.3902704411E1+(0.3465590348E1+(-0.1714727839E1+
    (0.5101467996+(-0.9119265524E-1+(0.9063185478E-2-0.3861021174E-3*x)
    *x)*x)*x)*x)*x)*x;
```

The reader can study the help pages for `?fortran` and `?C` for additional information capabilities and options to these commands.

5 Examples

5.1 Einfache Differential- und Integralrechnung

Differenzieren und Integrieren von mathematischen Formeln sowie ihre Vereinfachung. Im allgemeinen wird der generische Fall betrachtet, das heisst die Umformungen seien zulässig.

```
> restart;
  diff(x/(1-x^2),x);
  simplify(%);
```

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

```
> f := x->x*sin(a*x)+b*x^2;
  Diff(f(x),x);
  fd := value(%);
  Int(fd,x);
  value(%);
  simplify(%);
```

$$f := x \to x\sin(a\,x)+b\,x^2$$
$$\frac{\partial}{\partial x}\,(x\sin(a\,x)+b\,x^2)$$
$$fd := \sin(a\,x)+x\cos(a\,x)\,a+2\,b\,x$$
$$\int \sin(a\,x)+x\cos(a\,x)\,a+2\,b\,x\,dx$$
$$-\frac{\cos(a\,x)}{a}+\frac{\cos(a\,x)+a\,x\sin(a\,x)}{a}+b\,x^2$$
$$x\sin(a\,x)+b\,x^2$$

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

$$1$$

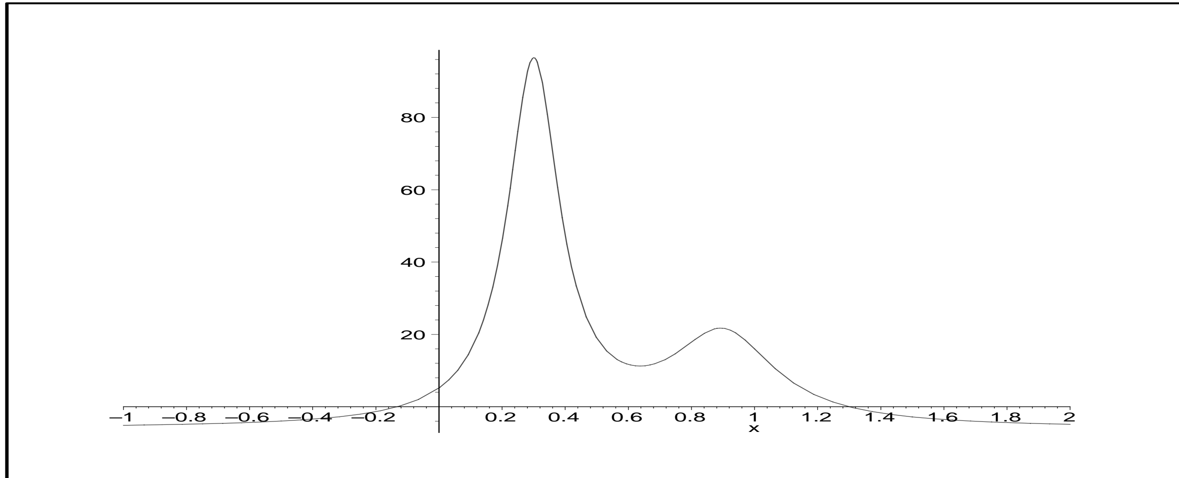
```
> simplify(exp(a*ln(b)));
```

$$b^a$$

```
> restart;
  f := x->1/((x-0.3)^2+0.01)+1/((x-0.9)^2+0.04)-6.0;
  s := solve(f(x),x);
  plot(f(x),x=-1..2);
```

$$f := x \to \frac{1}{(x-.3)^2+.01}+\frac{1}{(x-.9)^2+.04}-6.0$$

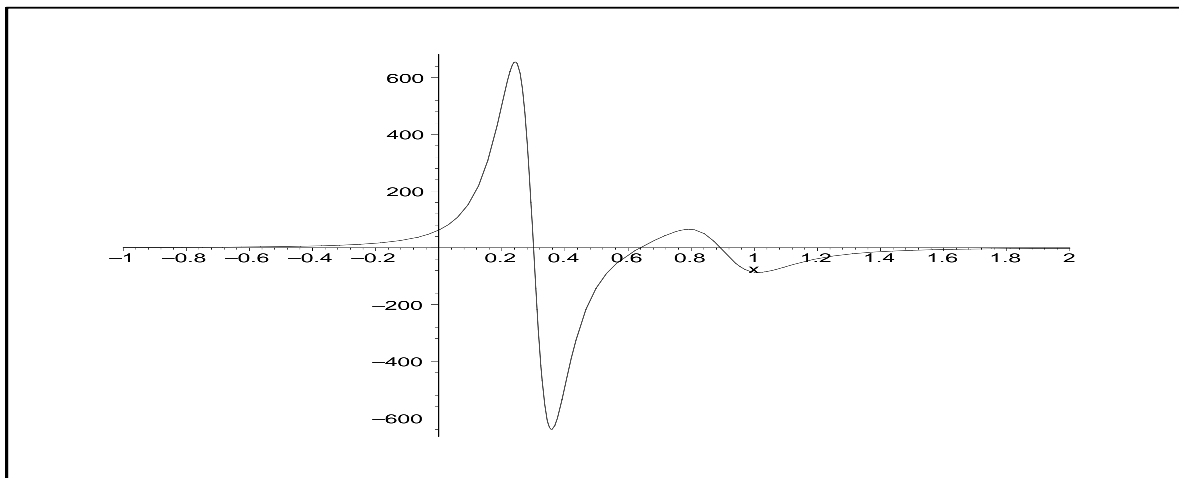
$s := -.1316180181,\ .6160341678-.2219033632\,I,\ .6160341678+.2219033632\,I,$
1.299549683




```
> f1 := unapply(diff(f(x),x),x);
  s1 := solve(f1(x),x);
  plot(f1(x),x=-1..2);
```

$$f1 := x \to -\frac{2\,x-.6}{((x-.3)^2+.01)^2} - \frac{2\,x-1.8}{((x-.9)^2+.04)^2}$$

$s1 := .3003756216,\ .5849495452 - .5644821962\,I,\ .5849495452 + .5644821962\,I,$
$.6370089847,\ .8927163033$



5.2 Schneeflockenkurven

Zur Entstehung der Schneeflockenkurven

Der Rand der Kurven ist ein Polygonzug.

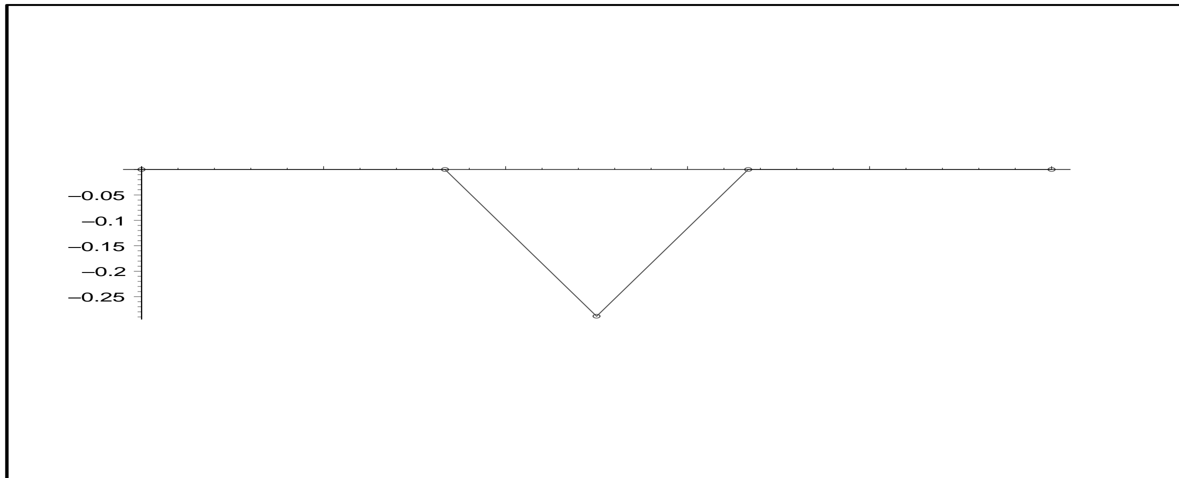
Er entsteht, indem ausgehend von einem gleichseitigen Dreieck jede Teilseite, definiert durch ihre beiden Eckpunkte, jeweils gedrittelt wird und ein Dreieck auf der "Mitte" dieser Seite errichtet wird.

Der benutzte Datentyp ist die Liste, denn aus einer Liste mit zwei Punkten (Punkt ist selbst Liste seiner Koordinaten) kann man einfach einen Polygonzug durch fünf Punkte erzeugen.

```
> restart:
  with(plots):  with(plottools):
> u := [0,0]; v := [1,0];
  x0 := op(1,u):  y0 := op(2,u):
  x1 := op(1,v):  y1 := op(2,v):
  x2 := x0+1/3*(x1-x0):  y2 := y0+1/3*(y1-y0):
  x3 := x0+2/3*(x1-x0):  y3 := y0+2/3*(y1-y0):
  x4 := 1/2*(x2+x3+(y3-y2)*sqrt(3)):  y4 := 1/2*(y2+y3+(x2-x3)*sqrt(3)):

  pz := [u,[x2,y2],[x4,y4],[x3,y3],v];
  p1 := plot(pz,style=point,symbol=CIRCLE):
  p2 := plot(pz,thickness=2):
  plots[display](p1,p2,scaling=constrained);
```

$$u := [0,\ 0]$$
$$v := [1,\ 0]$$
$$pz := [[0,\ 0],\ [\frac{1}{3},\ 0],\ [\frac{1}{2},\ -\frac{1}{6}\sqrt{3}],\ [\frac{2}{3},\ 0],\ [1,\ 0]]$$



Diesen Prozess realisiert die Prozedur $auft$: zu zwei gegebenen Punkten erzeugt sie die fünf neuen Punkte einer Seite (Verfeinerung).
Wenn man dem zusätzlichen reellen Parameter vz andere Werte als 1 gibt, entstehen mehr oder weniger bizarre Kurven.

```
> auft := proc(u,v::list)
    local x,y,d,n,x0,y0,x1,y1,x2,y2,x3,y3,x4,y4,p,vz;
    vz := 1; # Parameter
    x0 := op(1,u); y0 := op(2,u);
    x1 := op(1,v); y1 := op(2,v);
    x2 := x0+vz*1/3*(x1-x0); y2 := y0+vz*1/3*(y1-y0);
    x3 := x0+vz*2/3*(x1-x0); y3 := y0+vz*2/3*(y1-y0);
    x4 := 1/2*(x2+vz*x3+(y3-y2)*sqrt(3));
    y4 := 1/2*(y2+vz*y3+(x2-x3)*sqrt(3));
    u,[x2,y2],[x4,y4],[x3,y3],v;
  end:
```

Ecken des Ausgangsdreiecks und seine Darstellung

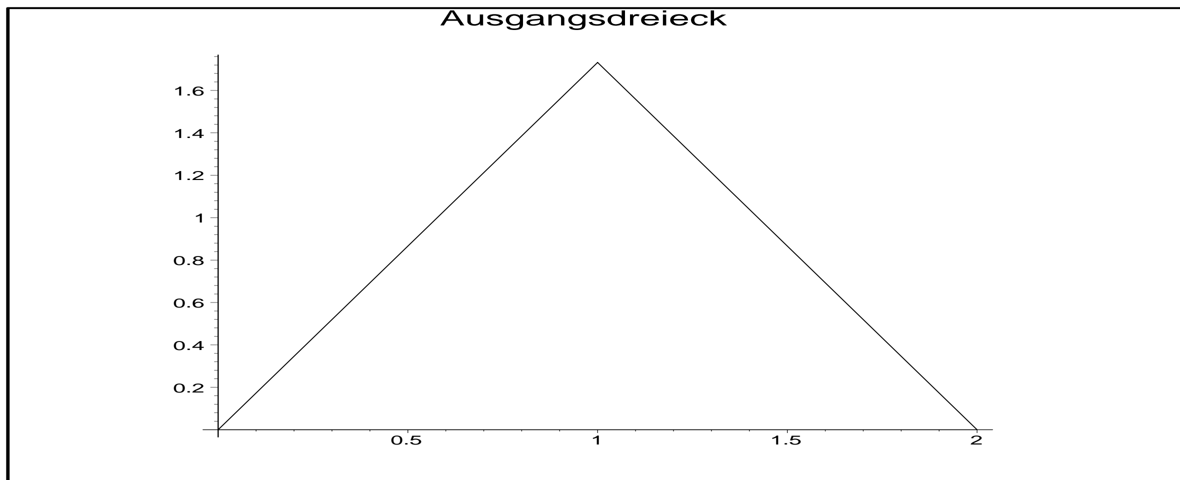
```
> Liste := [[0,0],[2,0],[1,3^(1/2)],[0,0]];
  n := nops(Liste)-1;
  pmax := 5; # Anzahl der Schneeflocken
  p := array(1..pmax,[]):
  p[1] := polygon(Liste); # Ausgangsdreieck = 1.Schneeflocke
  plots[display](p[1],scaling=constrained,title=`Ausgangsdreieck`);
```

$$Liste := [[0, 0], [2, 0], [1, \sqrt{3}], [0, 0]]$$

$$n := 3$$

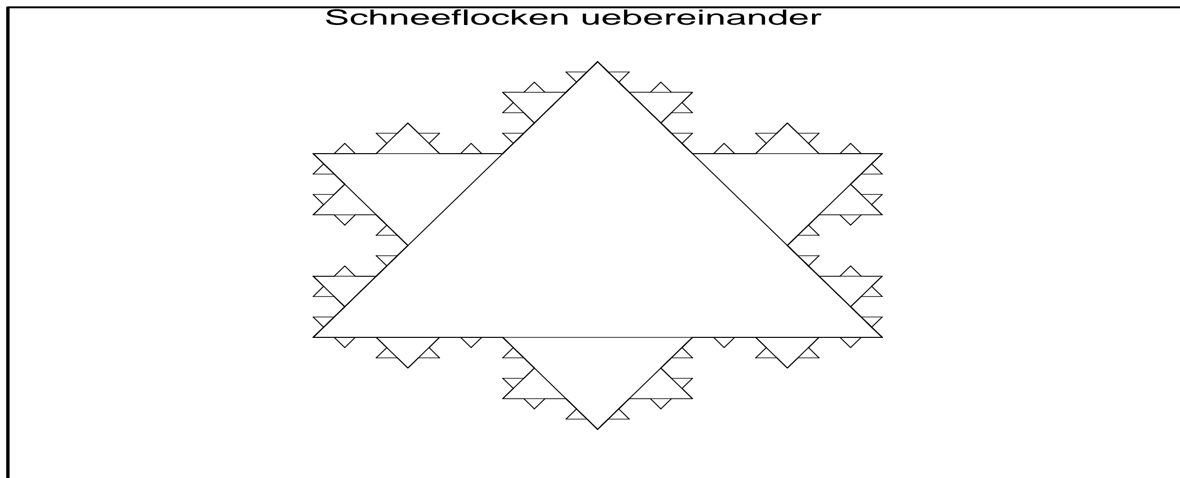
$$pmax := 5$$

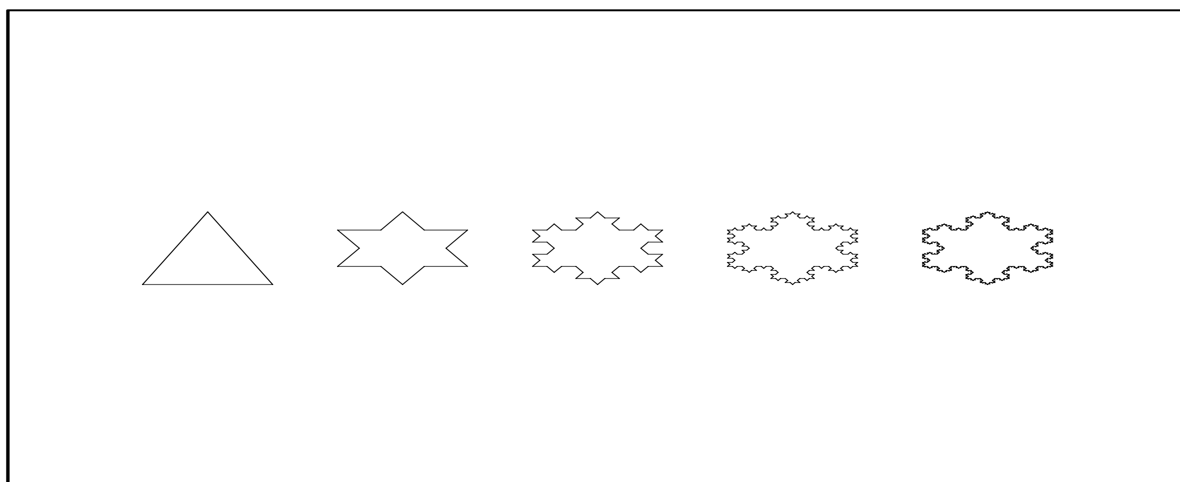
$$p_1 := \mathrm{POLYGONS}([[0, 0], [2., 0], [1., 1.732050808], [0, 0]])$$



Jede Seite der Schneeflocke wird aus der Liste in eine Subliste übertragen und mittels *auft* verfeinert.
Die Zusammenfassung aller Sublisten ergibt die nächste Schneeflocke.

```
> for k from 2 to pmax do
    n := nops(Liste)-1:
    for i from 1 to n do SubListe.i := [op(i,Liste),op(i+1,Liste)] od:
    for i from 1 to n do SubListe.i := [auft(op(SubListe.i))] od:
    Liste:=[]:
    for i from 1 to n do
      Liste := [op(Liste),op(1..nops(SubListe.i)-1,SubListe.i)] od:
    Liste := [op(Liste),op(nops(SubListe.n),SubListe.n)]:
    p[k] := polygon(Liste):
  od:
> plots[display](seq(p[j],j=1..pmax-1),axes=none,scaling=constrained,
    title=`Schneeflocken uebereinander`);
  plots[display](p,axes=none,scaling=constrained); # SF nebeneinander
```





Umfang der letzten Schneeflocke

Liste enthält alle Ecken des Polygons.

```
> n := nops(Liste)-1:
  for i from 1 to n do SL[i] := [op(i,Liste),op(i+1,Liste)]; od:
  i := 'i':
  sum(((SL[i][1][1]-SL[i][2][1])^2+
      (SL[i][1][2]-SL[i][2][2])^2)^(1/2),i=1..n);
```

$$\frac{256}{2187}\sqrt{4}\sqrt{6561}$$

```
> radnormal(%); # etwas besser verstaendlich
```

$$\frac{512}{27}$$

Der Umfang der Kurven nimmt in jedem Schritt um 4/3 zu.

```
> L:=6;
  for i from 2 to pmax do L:=4/3*L; od;
```

$$L := 6$$

$$L := 8$$

$$L := \frac{32}{3}$$
$$L := \frac{128}{9}$$
$$L := \frac{512}{27}$$

5.3 Newtoniteration mit Filearbeit und Graphik

Erinnern wir uns an das Newtonsche Näherungsverfahren zur Lösung der skalaren Gleichung ***f(x)*=0**.

```
> restart;
  x[k+1] = x[k] - f(x[k])/fs(x[k]); # fs=f'
```

$$x_{k+1} = x_k - \frac{f(x_k)}{fs(x_k)}$$

Ziel ist es, das Iterationsverfahren mit einer vorgegebenen Anzahl n von Iterationen zu rechnen, und dabei sowohl die Zwischenwerte in einer Datei *newton1.res* zu speichern als auch den Verlauf der Annäherung an die Nullstelle graphisch zu illustrieren.

Also nehmen wir die grafische Darstellung der Funktion und des Iterationsverlaufes als letzten Befehl in der Prozedur auf, da dort nur die letzte Anweisung zurückgegeben wird. Die numerischen Werte werden "vorher" über die *print* oder *fprintf*-Anweisung ausgegeben.

```
> restart;
  with(plots):
  Newton1 := proc(f::procedure,xstart::numeric,n::posint)
    local i,x,xold,xnew,xiter,df,fxold,fxnew,delta,file1,
       curve,s,t,ps,pt,plts,pltt,xleft,xright,Digits_old;
  Digits_old:= Digits;
  Digits := 22;
  file1 := fopen(`D:/Neundorf/Maple/program/newton1.res`,WRITE);
  df := D(f); # Ableitung von f
  xold := xstart;
  xiter := xold;
  fxold := evalf(f(xold));
  s := [[xold,0],[xold,fxold]];
  ps[0] := plot(s,color=black); # Anfangsordinate
  fprintf(default,` x = %.20e, f(x)= %.20e\n`,xold,fxold);
  fprintf(file1, `x = %.20e, f(x)= %.20e\n`,xold,fxold);
  for i from 1 to n do
    xnew := evalf(xold-fxold/df(xold));
    delta := abs(xold-xnew);
    fxnew := evalf(f(xnew));
    xiter := xiter,xnew;
    s := [[xnew,0],[xnew,fxnew]];
    t := [[xold,fxold],[xnew,0]];
    ps[i] := plot(s,color=black); # Ordinate
    pt[i] := plot(t,color=blue); # Tangente
    xold := xnew;
    fxold := fxnew;
    fprintf(default,` x = %.20e, f(x)= %.20e\n`,xold,fxold);
```

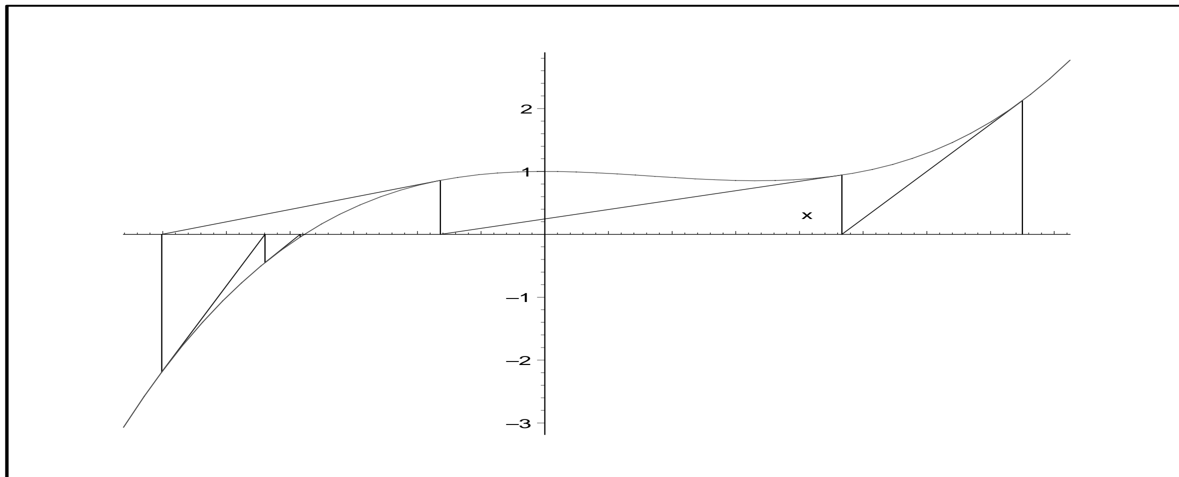
```
  fprintf(file1, `x = %.20e, f(x)= %.20e\n`,xold,fxold);
od;
fprintf(default,` delta = %.20e\n`,delta);
fprintf(file1, `delta = %.20e\n`,delta);
fclose(file1);
print(`xiter=`,xiter);
plts := seq(ps[i],i=0..n);
pltt := seq(pt[i],i=1..n);
xleft := min(xiter)-0.1*abs(min(xiter));
xright:= max(xiter)+0.1*abs(max(xiter));
curve := plot(f(x), x=xleft..xright);
Digits := Digits_old;
display(curve,plts,pltt);
end:
```

```
> f := x->x^3-x^2+1;
  Newton1(f,1.5,8);
```

$$f := x \to x^3 - x^2 + 1$$

```
x =   1.50000000000000000000e+00,  f(x)=   2.12500000000000000000e+00
x =   9.33333333333333333333e-01,  f(x)=   9.41925925925925925926e-01
x =  -3.28174603174603174605e-01,  f(x)=   8.56957494303570928664e-01
x =  -1.20311660303173787304e+00,  f(x)=  -2.18898828343201634623e+00
x =  -8.78759664481079919828e-01,  f(x)=  -4.50813059994179589027e-01
x =  -7.68108290150601923652e-01,  f(x)=  -4.31668208071645282770e-02
x =  -7.55051916522017205589e-01,  f(x)=  -5.61058895011453755000e-04
x =  -7.54877697030048531745e-01,  f(x)=  -9.91001278544930000000e-08
x =  -7.54877666246693721015e-01,  f(x)=  -3.09361500000000000000e-15
delta =  3.07833548107297000000e-08
```

$$xiter =, 1.5, .9333333333333333333333, -.3281746031746031746047, \\ -1.203116603031737873040, -.8787596644810799198278, \\ -.7681082901506019236520, -.7550519165220172055889, \\ -.7548776970300485317451, -.7548776662466937210154$$

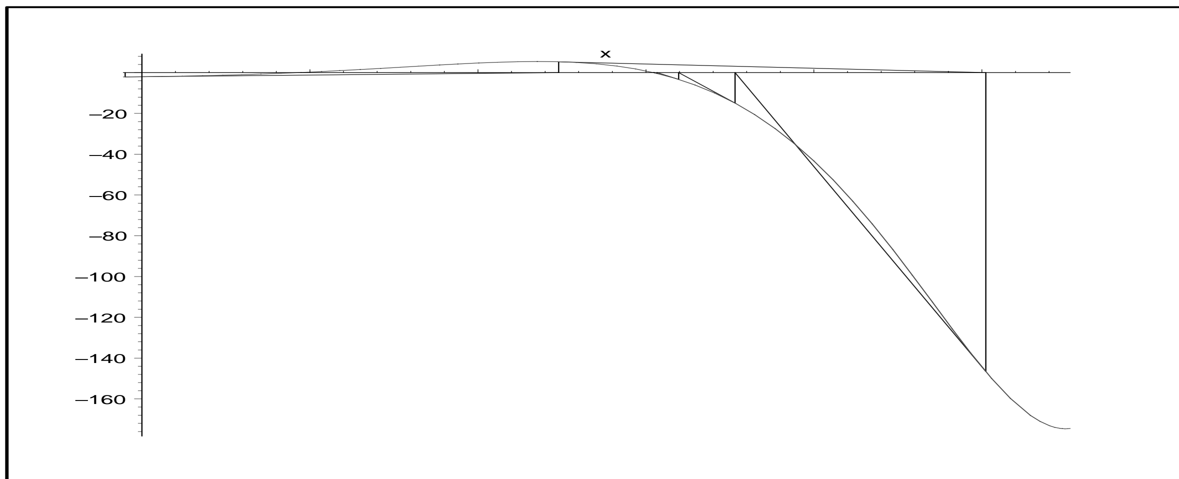


```
> g := x->sin(x)*exp(x)-2;
  Newton1(g,-0.1,8);
```

$$g := x \to \sin(x)\, e^x - 2$$

```
x = -1.000000000000000000000e-01,  f(x)=  -2.090333010952424170270e+00
x =  2.480709050570077394140e+00,  f(x)=   5.334922051035510897440e+00
x =  5.022603293500686515260e+00,  f(x)=  -1.465600078758056998530e+02
x =  3.530432104550907634380e+00,  f(x)=  -1.494249099166628890100e+01
x =  3.194892658992309369280e+00,  f(x)=  -3.300306864119281549720e+00
x =  3.066341992140242904380e+00,  f(x)=  -3.864004132386183691320e-01
x =  3.046815878924072302470e+00,  f(x)=  -8.103038800553464209000e-03
x =  3.046388545315807405550e+00,  f(x)=  -3.825852245980284000000e-06
x =  3.046388343359421006050e+00,  f(x)=  -8.542278340000000000000e-13
delta =  2.019563863995020000000e-07
```

$xiter =, -.1, 2.48070905057007739413 7, 5.022603293500686515262,$
$3.530432104550907634380, 3.194892658992309369279,$
$3.066341992140242904379, 3.046815878924072302469,$
$3.046388545315807405547, 3.046388343359421006045$



5.4 Hornerschema

Das $(n+1)$-zeilige oder vollständige Hornerschema für reelle Polynome n-ten Grades

$$p_n(x) = a_0 x^n + a_1 x^{n-1} + ... + a_{n-1} x + a_n,\ a_0 \neq 0,$$

dient der Berechnung von Funktionswert (0-te Ableitung) und aller Ableitungswerte des Polynoms für gegebenes Argument x. Das Ergebnis ist ein Vektor p mit $p_n = p_n(x),\ p_{n-1} = p_n'(x), ...,\ p_0 = p_n^{(n)}(x) = n! a_0$.

Wir geben eine Prozedur für das k-zeilige Hornerschema $(k \leq n+1)$ an.

```
> restart;
  VHS := proc(n::posint,k::posint,a::array,p::array,x::numeric)
    local i,j,fak,b;
```

```
if k>n+1 then print(`k decrease to n+1`); k := n+1 fi;
b := array(0..n,[]);
for i from 0 to n do b[i] := a[i] od; # not assignment b:=a;
for i from 0 to k-1 do
  for j from 1 to n-i do b[j] := b[j-1]*x+b[j] od od;
p[n] := b[n];
fak := 1;
for i from 1 to k-1 do
  fak := fak*i;
  p[n-i] := b[n-i]*fak
od;
k # stage of scheme is the returned value
end:
```

Beispiel

```
> n := 4;
  a := array(0..n,[1$n+1]):
  p := array(0..n,[]):
  seq(a[i],i=0..n);
  i := 'i':  x:='x':
  pn := sum(a[i]*x^(n-i),i=0..n);
```

$$n := 4$$

$$1, 1, 1, 1, 1$$

$$pn := x^4 + x^3 + x^2 + x + 1$$

```
> x := 2;
  k := n+1:
  VHS(n,k,a,p,x):
  for i from n by -1 to n+1-k do
    fprintf(default,` %g - te Ableitung bei x = %g ist %g\n`,n-i,x,p[i])
  od:
```

$$x := 2$$

```
0       - te Ableitung bei x = 2       ist 31
1       - te Ableitung bei x = 2       ist 49
2       - te Ableitung bei x = 2       ist 62
3       - te Ableitung bei x = 2       ist 54
4       - te Ableitung bei x = 2       ist 24
```

Achtung!

In der Prozedur VHS wird der Eingangsvektor a der Polynomkoeffizienten auf ein Hilfsfeld b umgespeichert. Dazu benutzen wir eine Laufanweisung. Die als Kommentar ergänzte kürzere Version $b := a$ führt zu einem inhaltlichen Fehler, denn bei einem wiederholten Aufruf von VHS würden nunmehr im Vektor a schon die inzwischen neuberechneten Koeffizienten von b stehen.
Dieser eigentlich unübliche Effekt ist am folgenden Beispiel ebenfalls zu erkennen. Er betrifft Felder allgemein, aber insbesondere auch Vektoren und Matrizen. Eine mögliche Deutung diese Sachverhalts ist, dass mit $b := a$ die auf a gesetzte Referenz (Zeiger) von b übernommen wird, und damit a und b den gleichen Speicherbereich adressieren, was bei Feldern Speicherplatz spart.

Macht man eine der beiden Variablen atomar, so bleibt der Zeiger auf die andere erhalten.

Komponentenweise Umspeicherung

```
> restart;
  aa := vector([1,2,3,4]);
  bb := vector(4);
  for i from 1 to 4 do bb[i] := aa[i] od:
  seq(bb[i],i=1..4);
  bb[1] := -1;
  seq(bb[i],i=1..4);
  seq(aa[i],i=1..4); # aa unveraendert wie Original
```

$$aa := [1, 2, 3, 4]$$
$$bb := \mathrm{array}(1..4, [])$$
$$1, 2, 3, 4$$
$$bb_1 := -1$$
$$-1, 2, 3, 4$$
$$1, 2, 3, 4$$

Umspeichern mit Zeigertechnik

```
> bb := aa;
  seq(bb[i],i=1..4);
  bb[1] := -1;
  seq(bb[i],i=1..4);
  seq(aa[i],i=1..4); # aa identisch mit bb
  bb := 'bb';
  seq(aa[i],i=1..4);
```

$$bb := aa$$
$$1, 2, 3, 4$$
$$bb_1 := -1$$
$$-1, 2, 3, 4$$
$$-1, 2, 3, 4$$
$$bb := bb$$
$$-1, 2, 3, 4$$

6 Appendix

Anhang A

Export von Maple Worksheets in LaTeX

Einige Bemerkungen zum Export von Maple Worksheets in LaTeX unter besonderer Berücksichtigung von Graphiken (Plots).
Was den Text betrifft ist es ratsam, im Worksheet einen üblichen Zeichensatz mit normalem Schriftstil und Schriftgrössen zu benutzen. Andere Schriftformate können im LaTeX-File eventuell Probleme bereiten.

Was passiert bei gewöhnlichen Standardplots?
Zunächst bringt der Druck des Worksheets den darin enthaltenen Bildschirmplot ungefähr in der Grösse $10 \times 10cm$ horizontal zentriert aufs Papier.
Der Export des Worksheets *name.mws* in LaTeX erzeugt neben dem LaTeX-File *name.tex* für die darin enthaltenen Plots zusätzlich die entsprechenden Postscript-Dateien (*eps*-Format) *name*01.*eps*, *name*02.*eps*, usw.
Ihr Dateikopf ist

```
%!PS-Adobe-3.0 EPSF
%%Title: Maple plot
%%Creator: MapleV
%%Pages:  1
%%BoundingBox: (atend)
%%DocumentNeededResources: font Helvetica
%%EndComments
```

Darüber hinaus ist in der Datei ersichtlich, dass die Graphik sich in einer Box mit den Grenzen (72,72,719,540), also der Dimension $647 \times 468pt = 227 \times 165mm$, befindet und diese um 90 Grad gedreht und etwas verschoben wird.

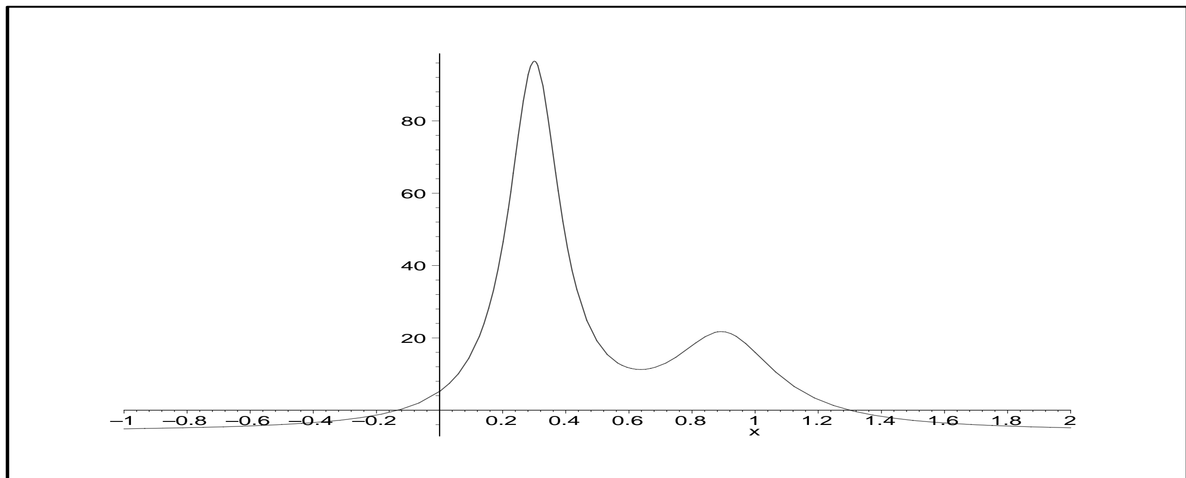
```
%%%%This function is needed for drawing text
%%IncludeResource: font Helvetica
540.000000 72.000000 translate
90 rotate
...
% The following draws a box around the plot,
% if the variable drawborder is true
...
showpage
grestore
end
%%Trailer
%%BoundingBox: 72 72 719 540
%%EOF
```

Wie kann nun die Ausgabe erfolgen?

1. Ausgabe im Rahmen von Maple-Gruppen

   Wir notieren eine Beispielvariante.

```
\begin{maplegroup}
 \begin{mapleinput}
  \mapleinline{active}{1d}{plot(f1(x),x=-1..2);}{}
 \end{mapleinput}
 \mapleresult
 \begin{maplelatex}
  \mapleplot{exam1a01.eps}
 \end{maplelatex}
 \end{maplegroup}
```

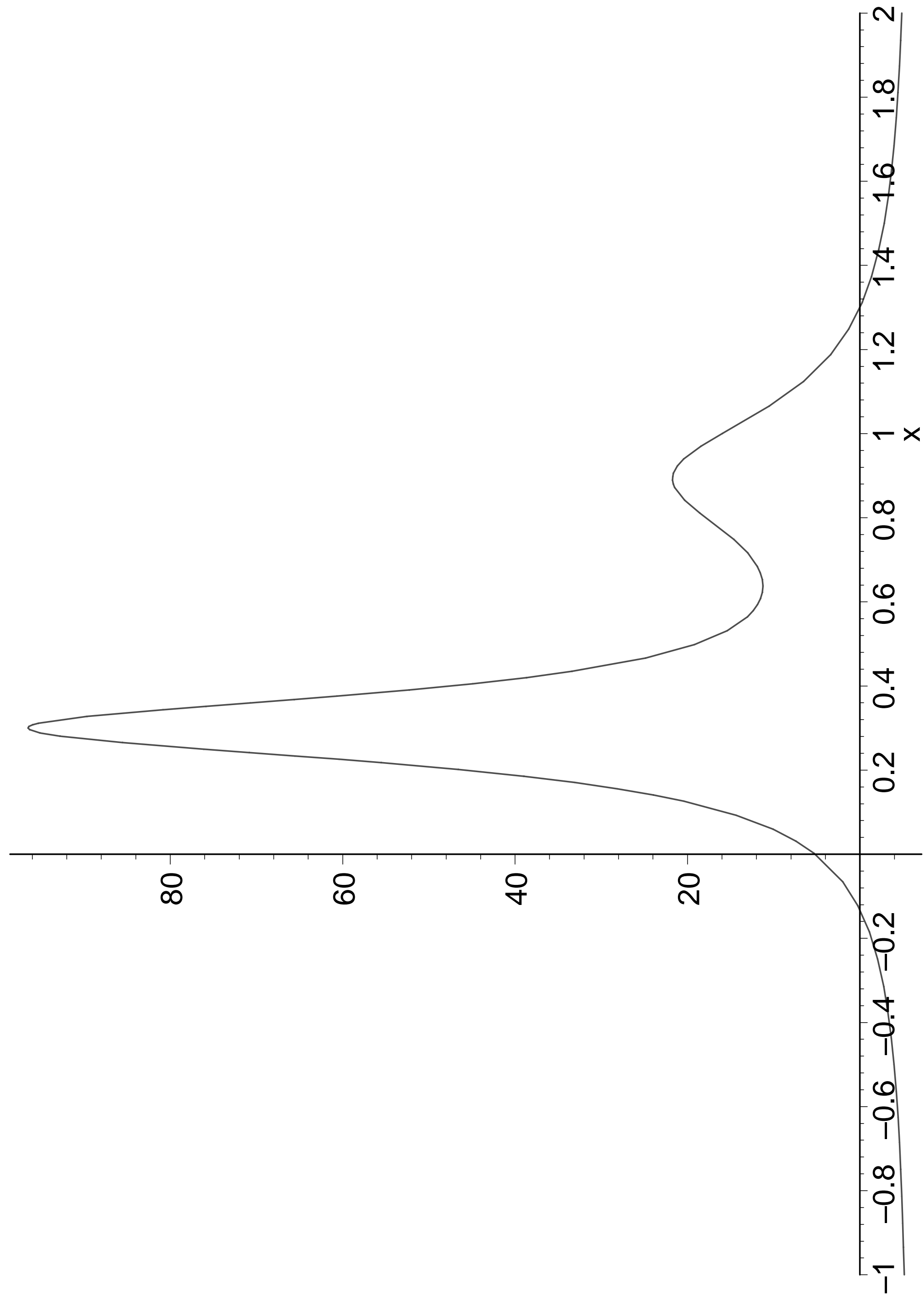


   Die Graphik im File *name*01.*eps* erscheint in der Ausgabe in einem berandeten Rechteck der Dimension von ca 157×63mm in einer Box von ca 160×90mm, also auf der ganzen Breite der Seite mit etwas Platz darunter. Damit wird die Druckgraphik auch im Vergleich zum Original in der Breite gestreckt, siehe Verältnis 157:63≈2 und 227:165≈1.4 .
   Skalierungsangaben $width = ...$ bzw. $height = ...$ sind im Befehl `\mapleplot{exam1a01.eps}` ohne Wirkung.

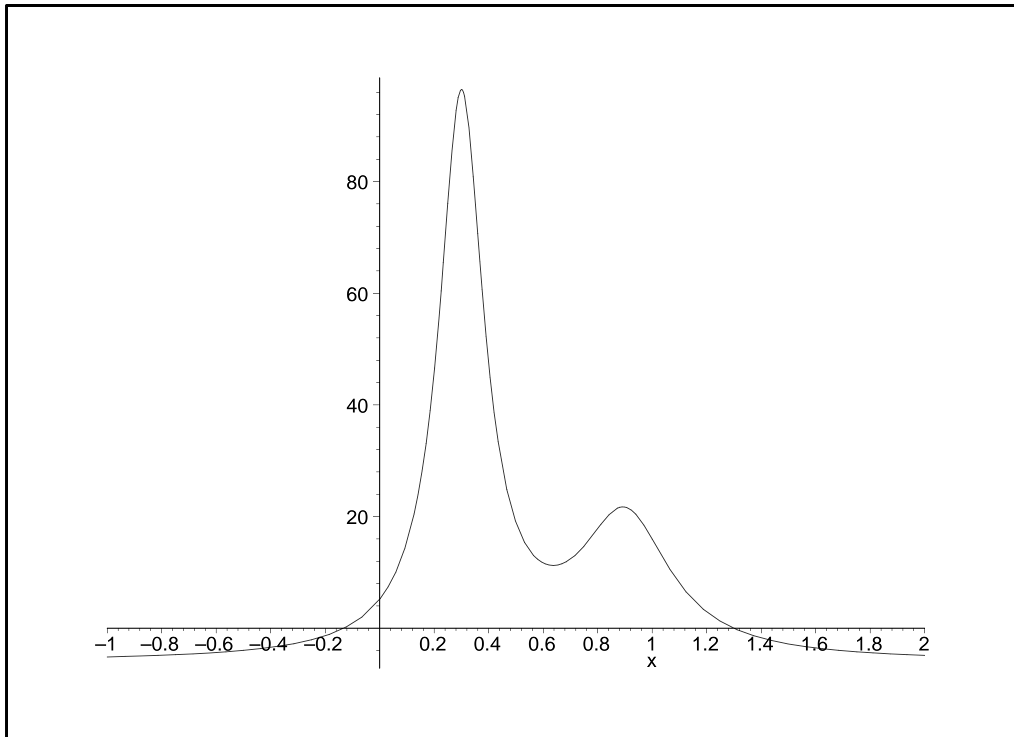
2. Ausgabe ausserhalb von Maple-Text

   Wir notieren Beispielvarianten mit dem Stil *psfig.sty*.

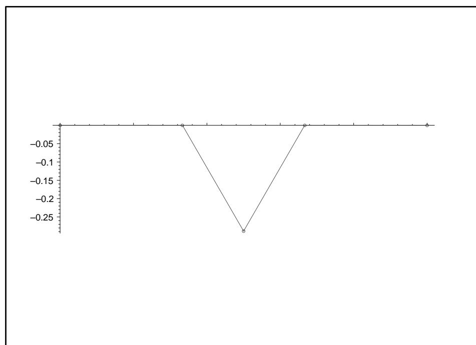
```
% im Original ist Graphik gedreht
\psfig{figure=exam1a01.eps}
% normale Ansicht etwas verkleinert
\psfig{figure=exam1a01.eps,width=15cm,angle=-90}
```

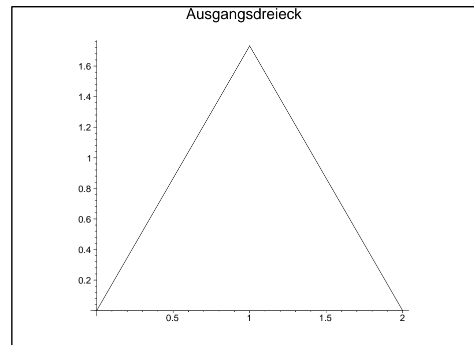


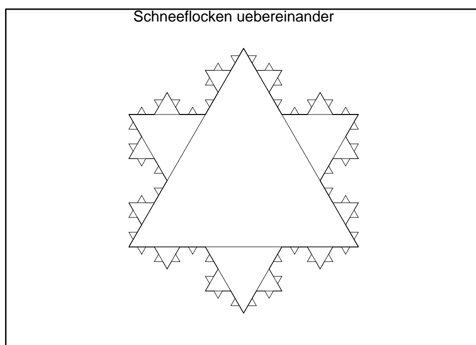
80
60
40
20
x
−1
−0.8
−0.6
−0.4
−0.2
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2

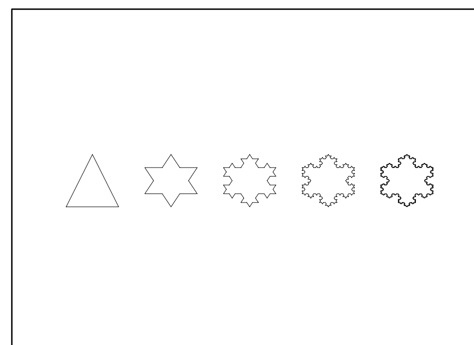


Die nicht verzerrten aber verkleinerten Bilder zu den Schneeflocken in Übersicht









Anhang B

Zusammenstellung von wichtigen Maple-Befehlen

1. Allgemeines und wichtige Konstanten

;	: Schliesst ein Kommando ab
:	: Wie oben, allerdings wird das Ergebnis nicht angezeigt
%	: Zugriff auf das letzte Ergebnis
% %	: Zugriff auf das vorletzte Ergebnis
with(pa)	: Aktivieren des Pakets (Package) *pa* von Funktionen, z.B. with(plots) Grafik - Programme
#	: Kommentar
?	: Aufruf der Online-Hilfe, z.B. `?evalf`
restart	: Neustart: alle bisherigen Variablendefinitionen werden gelöscht
Digits	: Systemvariable. Sie legt fest, mit welcher Genauigkeit numerische Berechnungen durchgeführt werden sollen. Standard ist 10, z.B. Digits:=50
'x'	: Löschen des Wertes der Variablen x: x:='x'
Pi	: $\pi = 3.141592...$
exp(1)	: Eulersche Zahl $e = 2.718281...$

2. Exakte Arithmetik und Gleitpunktarithmetik

:=	: Operator für Variablenzuweisung
<>	: Ungleichheitsoperator
>, >=, <, <=, =	: grösser-, grösser-gleich-, kleiner-, kleiner-gleich und gleich-Operator
+, -, *, /	: Grundrechenoperationen (Addition, Subtraktion, Multiplikation, Division)
^,**	: Potenz
!	: Fakultät
and, or, not	: logische Oprartoren
sqrt(x)	: Quadratwurzel $\sqrt{x}$
exp(x)	: Exponentialfunktion e^x
log(x), ln(x)	: natürlicher Logarithmus $\ln(x)$
sin(x), cos(x), tan(x)	: trigonometrische Funktionen
evalf(ausdr)	: numerische Auswertung von symbolischen Ausdrücken
evalf(ausdr,n)	: numerische Auswertung, n Stellenzahl
convert(expr,rational,n)	: Umrechnung des Ausdruckes *expr* in eine rationale Zahl, n Stellenzahl

3. Polynome und rationale Terme

subs(x = a, p)	: Ersetzt im Ausdruck p die Variable x durch a (a kann Zahl oder Ausdruck sein)
collect(p,x)	: Ordnet den Ausdruck (Polynom) p nach Potenzen von x
coeff(p,x,n)	: Bestimmt den Koeffizienten von x^n des Polynoms p, z.B. coeff($x^3 - 7 * x + 1, x, 1$) ergibt -7
factor(p)	: Versucht das Polynom p in die zugrundeliegenden Faktoren aufzuspalten, z.B. $x^2 - 3 * x + 2 = (x-1) * (x-2)$
expand(expr)	: Multipliziert die im Ausdruck *expr* enthaltenen Faktoren aus. Bei rationalen Funktionen betrifft expand nur den Nenner. Wenn Zähler und Nenner ausmultipliziert werden sollen, ist normal(expr, var, expanded) zielführend.
simplify(expr)	: Versucht den Ausdruck *expr* zu vereinfachen, z.B. simplify($(x^2-1)/(x-1)-1$) ergibt x
radsimp(expr)	: Vereinfachung des Ausdrucks *expr*, der Radikale (Wurzeln) enthält, z.B. radsimp($(x^2 - 2 * x + 1)^{(1/2)}$) ergibt $x - 1$
numer(ratfunc)	: Ermittelt den Zähler der rationalen Funktion $ratfunc$
denom(ratfunc)	: Ermittelt den Nenner der rationalen Funktion $ratfunc$
rationalize(ratfunc)	: Macht den Nenner der rationalen Funktion $ratfunc$ rational, z.B. rationalize($1/(\sqrt{2}-1)$) ergibt $\sqrt{2}+1$

4. Gleichungen

lhs(eq), rhs(eq)	: Liefert die linke Seite bzw. rechte Seite der Gleichung eq
solve(eq, x)	: Löst die Gleichung eq nach der Variablen x auf. Dabei werden alle Lösungen bestimmt (falls möglich), z.B. ergibt solve($x^2 - 1 = 0, x$) {-1,1}
allvalues(s)	: Ermittelt alle Lösungen numerisch vom RootOf-Ausdruck s
fsolve({gl},{var},options)	: Löst die Menge der Gleichung gl nach der Menge der Variablen var auf. Mögliche Optionen sind: complex - komplexe Lösungen fulldigits - maximale Genauigkeit bzw. Bereiche für die Variablen. In der Regel wird nur eine Lösung bestimmt, z.B. fsolve($\{x^2+y^2=1, y=2x\}, \{x,y\}, \{x=0..1, y=0..1\}$)

5. Funktionen

–>	: Definiert eine ein- bzw. mehrparametrige Funktion mit dem Pfeiloperator, z.B. $f := x\text{–>}\ x^2 + \sin(x)$ $g := (u,v,w)\text{–>}\ u + v * w + 1$
unapply(expr(x,y,...), x,y,...)	: Wandelt den Ausdruck expr der Variablen x, y... in einen funktionalen Operator in Pfeilschreibweise um. Aus unapply($\sin(x*y), x, y$) wird $(x,y)\text{–>} \sin(x*y)$

6. Grafik mit Maple

with(plots)	: Laden der Grafikprogramme
plot(f(x),x=a..b) bzw. plot(f,a..b)	: Zeichnet die Kurve $y = f(x)$ für x aus $[a, b]$
plot([f1(x),f2(x)],x=a..b, color=[red,blue])	: Zeichnet die Kurven $y = f1(x)$ und $y = f2(x)$ für x aus $[a, b]$ in den Farben rot und blau
plot([fx,fy,t=a..b])	: Zeichnet die Kurve $(x = fx(t), y = fy(t))$ mit t aus $[a, b]$
implicitplot(f(x,y)=0, x=a..b, y=c..d)	: Zeichnet die Lösungsmenge der Gleichung $f(x, y) = 0$ Zeichenbereich der x-Achse bzw. der y-Achse ist das Intervall $[a, b]$ bzw. $[c, d]$
plot([[x1,y1],[x2,y2], [x3,y3],...])	: Verbinden der Punkte $P1(x1, y1), P2(x2, y2), P3(x3, y3), \ldots$ durch einen Polygonzug. Bei einem geschlossenen Polygonzug muss der erste Punkt mit dem letzten übereinstimmen.
plot(...,style=point)	: Zeichnet eine Punktmenge
display(...)	: Vereint mehrere Grafiken in einer Abbildung z.B. p1:=plot(...): p2:=plot(...): display([p1,p2],scaling=constrained)

7. Matrizen und Vektoren

with(linalg)	: Paket für Lineare Algebra
M:=matrix(m,n) M[1,1]:=1;...	: Elementweise Eingabe einer Matrix M vom Typ (m, n) mit $M_{(1,1)} = 1, ..$
M:=matrix(m,n,[1,...])	: Zeilenweise Eingabe einer Matrix M vom Typ (m, n)
M:=matrix(m,n,f)	: Eingabe einer Matrix M vom Typ (m, n) durch Funktion f, d.h. mit $M_{(i,j)} = f(i, j)$
M[i,j]	: Element von M in Zeile i und Spalte j
v:=vector(n) v[1]:=1;...	: Elementweise Eingabe eines Vektors mit n Komponenten $v_1 = 1; \ldots$
v:=vector(n,[1,...])	: Wie oben
v[i]	: i-te Komponente von v
rowdim(A)	: Anzahl der Zeilen von A
coldim(A)	: Anzahl der Spalten von A
vectdim(v)	: Anzahl der Komponenten des Vektors v
evalm	: Auswertung von Ausdrücken mit Matrizen und Vektoren
A + B, matadd(A,B)	: Summe von A und B
α * A	: α-faches von A
A &* B, multiply(A,B)	: Produkt von A und B
transpose(A)	: Transponierte Matrix von A
innerprod(v,w)	: Inneres Produkt, Skalarprodukt von Vektoren v, w
crossprod(v,w)	: Kreuzprodukt

8. Endliche Folgen und Listen

sq:=a,b,c	: Folge
lst:=[a,b,c]	: Liste
c[n]	: Zugriff auf das n-te Element der Folge bzw. der Liste c
c[i..j]	: Erzeugt eine Teilfolge bzw. Teilliste der Folge bzw. der Liste c mit Elementen $c[i], c[i+1], ..., c[j]$
seq(f(i),i=2..4)	: Ergibt die Folge $f(2), f(3), f(4)$
seq(f(i),i=[2, 7, 15])	: Ergibt die Folge $f(2), f(7), f(15)$
nops(lst)	: Anzahl der Elemente der Liste lst
op(lst),op(1..nops(lst),lst)	: Wandelt die Liste lst in eine Folge um
lst:=[sq]	: Erzeugung der Liste lst aus der Folge sq

Anhang C

Zusammenstellung von wichtigen Adressen

1. World Wide Web (WWW) mit Maple Seiten

http://daisy.uwaterloo.ca/	Maple from Symbolic Computation Group (MAPLE resources, Univ. Waterloo, Ontario CA)
http://www.maplesoft.com	Maple WWW Home Page (Waterloo Maple Inc.)
http://daisy.uwaterloo.ca/	Maple Developers' Home Page
http://saaz.lanl.gov/maple/Maple_Home.html	A Maple Tutorial (The Cluster Team Presents)
http://saaz.lanl.gov/maple/Maple_Page13.html	Maple V on the Web
http://web.mit.edu/afs/athena/software/maple/www/home.html	Maple at MIT (Cambridge MA, USA)
ftp://ftp.unibe.ch/pub/Maple/algcurve/	AlgCurve Package maintained at University Bern
ftp://galois.maths.qmw.ac.uk/pub/mapledoc	Maple Introduction
http://krum.rz.uni-mannheim.de/cafgbench.html	Computer Algebra Benchmarks (RZ/Uni Mannheim)

2. Verzeichnisse mit Maple Worksheets und Maple Files

Zu den Kapiteln 1-5 des Skripts liegen die entsprechenden *mws*-Worksheets, *m*-Files und diverse Datenfiles vor. Die dortigen Verzeichnispfade sind i.a. anzupassen.

```
intro1.mws, data1.mws, data2.mws, proc1,mws. proc2.mws,
prog1.mws, prog2.mws, prog3.mws, exam1.mws,
*.m, *.txt, *.c, *.res, *.pcx
```

Die Dateien sind zu finden im Novell-Netz PIVOT des Instituts für Mathematik bzw. auf der persönlichen Homepage im Internet.

\\PIVOT\SHARE Q:\NEUNDORF\STUD_M93\MAPLE1

Homepage Navigator → Publications → Computeralgebra → Maple1

e-mail: ***neundorf@mathematik.tu-ilmenau.de***

Homepage: ***http://imath.mathematik.tu-ilmenau.de/ ~neundorf/index_de.html***

Anhang D

Schriften in Maple V Release 5

Für die Erstellung und Kommentierung von Worksheets $*.mws$ ist eine Schriftwahl einzelner Textabschnitte entsprechend ihrer Verwendung möglich, z.B. als Maple-Input, Output, Warnung, Fehlermeldung, Titel, Kommentare usw.

Achtung! Es sind nicht alle Varianten im eingestellten Schriftstil (Style) mit ausgewählter Zeichengrösse (Size) darstellbar. Das System legt dann eigenen Stil/Grösse fest.

Für den Export eines Worksheets als LaTeX-File ist es ratsam, den Text im Zeichensatz **Times New Roman** mit Stil **P Normal** zu schreiben, bei Schriftgrössen 12 bzw. 10, 14, 18. Andere Schriftformate verursachen im LaTeX-File eventuell die Generierung spezieller Fonts und somit Darstellungsprobleme.

Form der Notation der aufgeführten Stile im Zeichensatz **Times New Roman 12**, in Klammern die Farbangabe abweichend von *schwarz*: *(b)lau, (r)ot, (t)ürkis, (l)ila.*

P Author

- Item

- Dash Item

P Diagnostic (t)

P Error (l)

P Fixed Width

P Heading 1

P Heading 2

P Heading 3, 4

P List Item

P Maple Output

P Maple Plot

P Normal

P Text Output (b)

P Title

P Warning (b)

C 2D Comment

C 2D Input (r)

C 2D Math

C 2D Output (b)

C Help Heading 14 (nur bold)

C Help Normal

C Hyperlink (t)

C LaTeX

C Maple Input (r) (Courier New 12)

C Output Labels 8

C Plot Text 8

C Plot Title 10

C Popup (t)

References

[1] Monagan,M.: Programming in Maple: The Basics.
Institut für Wissenschaftliches Rechnen ETH-Zentrum, CH-8092 Zürich.

[2] Monagan,M. et al.: Maple V Programming Guide. Springer New York 1996.

[3] Klimek,G. und Klimek,M.: Discovering Curves and Surfaces with Maple. Springer 1997.

[4] Kofler,M.: Maple V Release 4. Addison Wesley Bonn 1996.

[5] Walz,A.: Maple V, Rechnen und Programmieren mit Release 4. Oldenburg 1998.

[6] Westermann,T.: Mathematik für Ingenieure mit Maple. Springer 1996.

[7] Werner,W.: Mathematik lernen mit Maple, Bd. 1,2. Ein Lehr-und Arbeitsbuch für das Grundstudium. dpunkt Heidelberg 1996, 1998 (CD).

[8] Char,B. et al.: Maple V Language Reference Manual. Springer 1991.

[9] Release Notes for Maple V Release 3. Waterloo Maple Software.

[10] First Leaves: Tutorial Introduction to Maple. Springer.

[11] Waterloo Maple Inc.: Maple V Student Version Release 4. Springer Berlin 1996 (CD Windows/Mac).

Anschrift:

Dr. Werner Neundorf
Technische Universität Ilmenau, Institut für Mathematik
PF 10 0565
D - 98684 Ilmenau

e-mail : neundorf@mathematik.tu-ilmenau.de