Algorithmen und Programmierung

(Studienjahr 2005/2006)

Doz.Dr.habil.Werner Vogt

Technische Universität Ilmenau
Fakultät für Mathematik und Naturwissenschaften
98684 Ilmenau, Germany

e-mail: werner.vogt@tu-ilmenau.de

Teil 2/3

Inhaltsverzeichnis

Kapitel 5

Symbolische Algorithmen und Grafik

Einer der Hauptzwecke der Computerentwicklung war stets das **wissenschaftliche Rechnen**. Obwohl Computer die Fabrikation, Verwaltung, Kommunikation und den Kommerz vollkommen durchdrungen haben, werden die leistungsfähigsten Computer für das wissenschaftliche Rechnen entwickelt.

Wissenschaftliches Rechnen (Scientific Computing)
ist die Computer–Simulation technischer, naturwissenschaftlicher oder sozialwissenschaftlicher Systeme mit Hilfe mathematischer Methoden. Es ist ein interdisziplinär orientiertes Gebiet, das insbesondere

- numerische Verfahren entwickelt und testet,
- effiziente Algorithmen auf leistungsfähigen Rechnern implementiert,
- numerisches und symbolisches Rechnen einsetzt und die
- Computersimulation komplizierter Prozesse unterstützt.

Bedeutung des Scientific Computing:

- Simulation als Alternative zum Bau teurer Prototypen:
  - Vermeidung teurer Versuchsanlagen
  - Luftfahrt- und Raumforschung
  - Vermeidung von Crashtests

- Simulation nicht durchführbarer Experimente:
  - Veränderungen des Weltklimas
  - Zusammenstoß zweier Galaxien
  - Epidemiemodelle
  - Fortschritte bei der Lösung der *grand challenge*-Probleme
- Lösung von Alltagsaufgaben:
  - Steuerung der Heizanlage von Wohnungen
  - Konstruktion strom- und wassersparender Waschautomaten
  - Energiesparwirkung von Dämmmaßnahmen
  - Vorhersage von Schadstoffausbreitungen etc.

5.1 Computeralgebra-Systeme

Moderne Computer sind universelle Maschinen, die prinzipiell jeden beliebig gearteten Algorithmus - z.B. im Sinne von Turing - auszuführen imstande sind. Algebraische (symbolische) Algorithmen bilden hierbei keine Ausnahme - sie können von einem Computer genauso leicht und effizient ausgeführt werden wie arithmetische (numerische) Algorithmen.

Wissenschaftliches Rechnen

Numerisches Rechnen	Symbolisches Rechnen
$\boldsymbol{E}$: $\boldsymbol{c := (a+b)^3}$	$\boldsymbol{E}$: $\boldsymbol{c := (a+b)^3}$
$\boldsymbol{a := 2.00; b := 3.00;}$	$\boldsymbol{expand(c);}$
$\boldsymbol{A}$: $\boldsymbol{(2.00+3.00)^3 =}$	$\boldsymbol{A}$: $\boldsymbol{(a+b)^3 =}$
$\boldsymbol{125.00}$	$\boldsymbol{a^3+3a^2b+3ab^2+b^3}$

Definition 1 : Computeralgebra (CA)

Computeralgebra (symbolisches Rechnen) beinhaltet die Algorithmierung, Implementation und Ausführung von endlichen mathematischen Transformationsvorschriften auf mathematischen Ausdrucksklassen mit Hilfe von Computern.
Computeralgebra ist symbolisches, algebraisches Rechnen, d.h. Rechnen mit Symbolen, die mathematische Objekte darstellen.

4 Vorteile des symbolischen Rechnens (CA)

1. Einsparung wertvoller Rechenzeit
   - Algebraische Vereinfachung von Formeln vor deren numerischer Auswertung
   - Frage: Wann ist der Einsatz eines CA-Systems sinnvoll ?
   - Beispiel: Nullstellenbestimmung $\boldsymbol{f(x) = 0}$
2. Exaktheit der Ergebnisse in der CA
   - Numerische Rechnungen sind mit Rundungsfehlern behaftet, weshalb Fehleranalysen erforderlich sind !
   - Beispiel: Berechnung von Integralen
3. Ingenieure und Naturwissenschaftler wollen Formeln und keine Zahlen
   - "Der Sinn der Berechnungen liegt darin, Einsichten zu gewinnen – und nicht Zahlen." (Richard W. Hamming)
   - Beispiel: Parameterabhängigkeit von Reihen
4. Computeralgebra erweitert die "Grenzen des Machbaren" in den Naturwissenschaften
   - Theorie $\Rightarrow$ Schlußfolgerungen $\Rightarrow$ Experimentelle Prüfung
   - Die komplizierten algebraischen Berechnungen sind fehleranfällig und kaum von anderen nachvollziebar !
   - Beispiel: Berechnung der Mondbahn durch

     Charles E. Delaunay : 20 Jahre !

Definition 2 : Computeralgebra-System (CAS)

Ein CAS ist ein Programmsystem, mit dessen Hilfe symbolische mathematische Transformationen auf mathematischen Ausdrucksklassen ausführbar sind.

Historie der CAS:

- 1953 – Diplomarbeit von H.G.Kahrimanian zur analytischen Differentiation (Assemblerprogramm)
- 1962 – LISP Programmers Manual (MIT) von J. McCarthy u.a.
- 1967 – Polynom-Manipulationssystem von G.E.Collins (Fortran)
- 1968 – REDUCE an der Uni von Utah (A.C.Hearn u.a.) entwickelt
- 1969 – MACSYMA am MIT (J. Moses u.a.) auf LISP-Basis entwickelt
- 1974 – SCRATCHPAD (später AXIOM) mit guten algebraischen Eigenschaften; von IBM vertrieben
- 1980 – Erste Version von MAPLE an der Uni Waterloo, Kanada
- 1987 – REDUCE 3.3 für Personalcomputer
- 1987 – MATHEMATICA (Kern in C) mit bedienerfreundlicher grafischer Oberfläche von S.Wolfram u.a. vorgestellt
- 1988 – DERIVE für 386er PC aus MuMath weiterentwickelt
- 1992 – 1. Release von MuPAD an Uni Paderborn (D) vorgestellt; 1994 mit European Academic Software Award prämiiert
- 1996 – MATHEMATICA 3.0 mit vielen Verbesserungen (neuer Kern zahlreiche Paletten etc.) und schneller Numerik
- 2000 – MAPLE 6 mit Numerik-Funktionen zur linearen Algebra aus der exzenllenten NAG-Bibliothek
- 2002 – MAPLE 8 mit GUI Maplets und mathematischen Erweiterungen

Teilgebiete von CA-Systemen

- Exakte Ganzzahl- und Rationalzahl-Arithmetik
- Gleitpunkt-Arithmetik beliebiger Genauigkeit
- Manipulation von rationalen Funktionen
- Vektor- und Matrixalgebra
- Formelmanipulation und Mustervergleich
- Symbolische Lösung von algebraischer Gleichungen
- Elementare Analysis (Differentiation, Integration)
- Symbolische Lösung von Differentialgleichungen
- Höhere Analysis (Vektoranalysis, Tensoranalysis)

Anwendungsgebiete von CA-Systemen

- Mathematik:
  Algebraische Geometrie, Zahlentheorie, Numerik, Verzweigungstheorie, Tensoranalysis etc.
- Physik:
  Himmelsmechanik, allgemeine Relativitätstheorie, Strukturmechanik, Thermodynamik etc.
- Technik:
  Robotik, Schaltkreisentwurf, Finite-Elemente-Methode, Tragflächenentwurf, Analyse dynamischer Netzwerke etc.

5.2 Maple - Aufbau und interaktive Nutzung

Maple ist ein kommerzielles Computeralgebra-System, das

- komplizierte **symbolische** mathematische Transformationen und
- hochgenaue **numerische** Berechnungen durchführen sowie
- Funktionen und Daten in 2D- und 3D **graphisch** darstellen kann.

****

Maple = Symbolik + Numerik + Graphik

****

Historie von Maple:

- 1980 - Erste eingeschränkte Version durch Keith Geddes und Gaston Gonnet (Univ. Waterloo, Provinz Ontario, Kanada) entwickelt
- 1984 - Version 3.3 (Kern in C programmiert) vermarktet
- 1988 - Waterloo Maple Software gegründet; Maple 4.3 läuft auf 20 verschiedenen Computerplattformen - u.a. auf 386er PC
- 1990 - Maple V - Version 1 erscheint: 3D-grafikfähig, neue Oberfläche, untersttzt auch UNIX XWindow- Oberfläche (*Anm.: Reaktion auf kommerziellen Erfolg von Mathematica!)*
- 1994 - Maple V Release 3 erscheint: verbesserte symbolische und numerische Verfahren, verbesserte Programmiersprache, Exportmöglichkeiten nach LaTeX, leichter handhabbare Benutzeroberfläche
- 1997 - Maple V Release 5: Einbau weiterer Pakete, Tabellenkalkulation, Smartplot in 2D und 3D , Multiple Worksheets mit Hyperlinks
- 2000 - Maple 6 erscheint: Einbau der Linearen Algebra aus der exzellenten NAG-Bibliothek
- 2002 - Maple 8 mit GUI Maplets und mathematischen Erweiterungen

****

Was kann Maple ?

- **Analytische Operationen:**
  Grenzwertbestimmung, Differentiation, Integration (unbestimmt, bestimmt, uneigentliche Integrale), Taylorreihen, Differentialgleichungen, Vektoranalysis, Tensoranalysis etc.

- **Algebraische Operationen:**
  Manipulation univariater oder multivariater Ausdrücke über
  – rationalen Zahlen
  – algebraischen Zahlkörpern
  – endlichen Körpern,
  Lineare Algebra, Algebraische und transzendente Gleichungssysteme

- **Numerische Operationen:**
  Exakte numerische Rechnungen (Rationalarithmetik), Gleitpunktarithmetik beliebiger Genauigkeit , Näherungsverfahren für algebraische Gleichungssysteme und Differentialgleichungen, Funktionsapproximation

- **Graphische Visualisierung:**
  Vielzahl 2- und 3-dimensionaler Darstellungen, z.B. Parameterdarstellungen von Kurven und Flächen, implizite Plots, Konturenplots, Vektorfelder, Polygone und Polyeder, Datendiagramme, Animation

- **Prozedurale Programmierung:**
  Datenstrukturen (Listen, Mengen, Felder), Kontrollstrukturen (if, for, while), Funktionen und Prozeduren , Dateioperationen, Pakete und Bibliotheken

****

****

Hauptunterschiede zu „klassischen“ Programmiersprachen (Fortran, Pascal, C) :

– Interpretative Abarbeitung -> Geschwindigkeitsverlust
+ Gemischte Verarbeitung symbolischer und numerischer Daten
+ Entwickelte symbolische Fähigkeiten
+ Rationalarithmetik und hochgenaue Gleitpunktarithmetik
+ Umfangreicher Vorrat an mathematischen Funktionen
+ Integriertes komfortables Graphiksystem
– Relativ bescheidene Programmierelemente (Kontrollstrukturen)
– Keine objektorientierte Programmierung

Man beachte stets: **Maple ist eine Applikation von C !!!**

****

Aufbau des Maple - Systems

(1) **Kernel:**
10 % des Systems, in optimiertem C-Code, eingebaute Funktionen

(2) **Libraries:**
90 % des Systems, in Maple, Main Library, Miscelleaneous Library, Packages

(3) **Share–Library:**
offenes System, Vielzahl elektronischer Ressourcen

****

5.2.1 Arbeit im Befehlsmodus

Eingabe von Befehlen

Maple-Befehle sind durch *Semikolon* oder *Doppelpunkt* voneinander zu trennen ! Der Doppelpunkt unterbindet die Bildschirmausgabe. Die Ausführung der Befehle erfolgt erst nach Eingabe von *Enter* !

```
> 9^10;
```

$$3486784401$$

```
> 99^100:
```

Bem.: Da der Beendigungsbefehl *Quit (Stop, Done)* stets am Ende einer Sitzung folgt, sind die vorhergehenden Befehle stets mit einem der beiden Zeichen abzuschließen.

Es können beliebig viele Befehle in einer einzigen Zeile notiert werden; andererseits kann ein (meist langer) Befehl auch über mehrere Zeilen hinweg eingegeben werden.

```
> sin(1.2)-2.0725; 13/347+29/111; 19!;
```

$$-1.140460914$$

$$\frac{11506}{38517}$$

$$121645100408832000$$

```
> 13/17
> +
> 29/111;
```

$$\frac{1936}{1887}$$

```
> cos(1)*2^1+
> cos(2)*2^2+
> cos(3)*2^3+
> cos(4)*2^4;
```

$$2\cos(1) + 4\cos(2) + 8\cos(3) + 16\cos(4)$$

Aufbau von Befehlen

Befehlsnamen sind meist mit den üblichen Bezeichnungen der Mathematik identisch bzw. orientieren sich an der Bedeutung des Befehls. Argumente sind in der Regel in ( ) zu setzen; mehrere Argumente sind als Folge - getrennt durch Kommata - zu notieren.

Mathematische Funktionen (Beispiele):

```
> exp(1); ln(x); log[2](356.5);
```

$$e$$

$$\ln(x)$$

$$8.477758266$$

```
> tan(pi/4); arcsin(1/2);
```

$$\tan(\frac{1}{4}\,\pi)$$

$$\frac{1}{6}\,\pi$$

```
> abs(x + ln(x+1)); erf(0.8043);
> signum(ln(0.3)); signum(y + sqrt(y^2+a));
```

$$|x + \ln(x+1)|$$

$$.7446506070$$

$$-1$$

$$\mathrm{signum}(y + \sqrt{y^2 + a})$$

```
> n!; factorial(n);
```

$$n!$$

$$n!$$

Mathematische Verfahren (Beispiele):

```
> round(exp(7.9));
```

$$2697$$

```
> ifactor(20!);
```

$$(2)^{18}\ (3)^8\ (5)^4\ (7)^2\ (11)\ (13)\ (17)\ (19)$$

```
> diff(tan(x),x);
```

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

```
> diff(tan(x),x,x,x);
```

$$2\,(1 + \tan(x)^2)^2 + 4\,\tan(x)^2\,(1 + \tan(x)^2)$$

```
> int(cos(x)*sin(x)^5,x);
```

$$\frac{1}{6}\,\sin(x)^6$$

```
> limit(sin(t)/t,t=0);
```

$$1$$

```
> solve(x^4-1=0);
```

$$1,\ -1,\ I,\ -I$$

Case-Sensitivität und starre Befehle

Maple unterscheidet zwischen Groß- und Kleinschreibung ! Die Namen der meisten Befehle beginnen mit Kleinbuchstaben (siehe oben).

```
> abs(x+1) - abs(1+x);
```

$$0$$

```
> abs(x+1) - abs(1+X);
```

$$|x+1| - |X+1|$$

```
> GAMMA(4.0); # Gammafunktion gamma; # Eulersche Constante
```

$$6.$$

$$\gamma$$

```
> limit((1+1/n)^n, n = infinity);
```

$$e$$

```
> limit((1+1/n)^n, n = Infinity);
```

$$\left(\frac{Infinity+1}{Infinity}\right)^{Infinity}$$

```
> Limit((1+1/n)^n, n = infinity);
```

$$\lim_{n\to\infty} \left(1+\frac{1}{n}\right)^n$$

Einige Maple-Funktionen existieren in 2 Versionen:

```
Limit      -    limit
Sum        -    sum
Diff       -    diff
Int        -    int
Signum     -    signum
Eval       -    eval
Expand     -    expand
Factor     -    factor
Roots      -    roots   usw.
```

Die starren Formen **Diff, Int, Limit usw.** unterdrücken die Ausführung der Berechnung und geben das Ergebnis nur formal an (Dieses kann z.B. mittels **eval** später ausgewertet werden).

```
> limit((1+1/n)^n, n = infinity);
```

$$e$$

```
> Limit((1+1/n)^n, n = infinity);
```

$$\lim_{n\to\infty} \left(1+\frac{1}{n}\right)^n$$

```
> diff(sin(x+y),x,y);
```

$$-\sin(x+y)$$

```
> Diff(sin(x+y),x,y);
```

$$\frac{\partial^2}{\partial y\,\partial x}\sin(x+y)$$

```
\textit{ }
> Sum(k^2, k=1..n);
```

$$\sum_{k=1}^{n} k^2$$

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

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

Benutzung vorhergehender Resultate

Ein Ergebnis eines Befehles kann als Eingabe für den nachfolgenden Befehl benutzt werden, indem der Ditto-Operator % benutzt wird. Bezugnahme auf vorletztes Ergebnis durch %% , auf das davorliegende Ergebnis durch %%% usw.

```
> int(cos(x)*sin(x)^5,x);
```

$$\frac{1}{6}\sin(x)^6$$

```
> diff(%,x);
```

$$\cos(x)\sin(x)^5$$

```
> diff(%,x);
```

$$-\sin(x)^6+5\cos(x)^2\sin(x)^4$$

```
> diff(%%%,x);
```

$$\cos(x)\sin(x)^5$$

Bemerkung: Bei Wiederholung der Rechnung in abweichender Reihenfolge entstehen meist andere Ergebnisse ! Empfehlung deshalb bei Bezugnahme auf vorhergehende Resultate: Benutzung von Variablen für die Zwischenergebnisse !

```
> res1 := int(cos(x)*sin(x)^5,x);
```

$$res1 := \frac{1}{6}\sin(x)^6$$

```
> res2 := diff(res1,x);
```

$$res2 := \cos(x)\sin(x)^5$$

```
> res3 := diff(res2,x);
```

$$res3 := -\sin(x)^6+5\cos(x)^2\sin(x)^4$$

```
> res4 := diff(res1,x);
```

$$res4 := \cos(x)\sin(x)^5$$

```
> res1, res2, res3;
```

$$\frac{1}{6}\sin(x)^6,\ \cos(x)\sin(x)^5,\ -\sin(x)^6+5\cos(x)^2\sin(x)^4$$

Abbruch von Berechnungen, Beenden von Maple sowie Hilfesystem

Abbruchversuch bei langandauernden Berechnungen mittels des STOP - Buttons bzw. mittels *CONTROL - C* oder *CONTROL - BREAK.*

```
> ifactor(2^101+1);
```

$$(3)\ (845100400152152934331135470251)$$

```
> # ifactor(2^201+1);
```

Beendigungsbefehle sind *quit , stop , done .* Sie entsprechen in Windows dem Menüpunkt *Close* im Menü *File .* Diesem letzten Befehl folgt nun kein Semokolon bzw. Doppelpunkt. Variablen behalten ihre Werte ! In Windows Beendigung von Maple über das Menü *File* , Menüpunkt *Exit* bzw. mittels *ALT + F4* .

```
> quit
> stop
> done
```

Hilfebefehl: Durch Angabe von **?Befehlsname** wird in einem separaten Fenster eine ausführliche Beschreibung des Befehls angegeben mit

- Darstellung des Befehlsinhaltes
- Mustern zu Anzahl und Typen der Parameter
- Beispielen zur Benutzung des Befehls
- Querverweisen.

```
> ?igcd;
```

Hilfefenster unter Windows: Mit dem *Help*- Menüpunkt kann man die üblichen Hilfsmechanismen ansprechen. Äquivalent zu *?Befehlsname* ist das Markieren des Befehlsnamens mit Maustaste und Auswahl von

Help-Menüpunkt und *Help on Topic "Befehlsname"*

bzw. *CONTROL-F1* .

```
> # igcd
```

Empfehlung für den Anfang: Geöffnetes Hilfefenster neben dem Arbeitsfenster auf dem Bildschirm anordnen. Das Hilfefenster enthält einen Browser mit der Hierarchie der Maple-Befehle !

```
> ?igcd;
```

Hilfe zu Befehlsnamen mit Anfangsbuchstaben: **?Buchstaben**

```
> ?co
```

Hilfe zum Hilfesystem: **?**

```
> ?
> quit
```

5.2.2 Numerisches und symbolisches Rechnen

Einfache Berechnungen

Maple verfügt über die allgemein üblichen Operatoren auf verschiedenen Datentypen, insbesondere:

– Arithmetische Operatoren: **+ , - , * , / , ^ , ** , !**

– Vergleichsoperatoren: = , <> , < , > , <= , >=

– Logische Operatoren: *not , and , or*

– Mengenoperatoren: *union , intersect , minus*

– Verkettungsoperator (Zeichenketten): **.**

– Zuweisungsoperator: **:=**

```
> 13*27 - 88*35/34 + 1/111;
```

$$\frac{491414}{1887}$$

```
> 13*27 - 88*(35/34 + 1/111);
```

$$\frac{489901}{1887}$$

```
> -(12!/7 + 8!*(3/5! - 7/4!));
```

$$-68418048$$

```
> 8/3/2/2;
```

$$\frac{2}{3}$$

```
> --36; 25 + -3;
```

```
‘-‘ unexpected
```

```
> -(-36); 25 + (-3);
```

$$36$$
$$22$$

```
> 2^-25;
```

```
‘-‘ unexpected
```

```
> 2^(-25);
```

$$\frac{1}{33554432}$$

```
> 2^3^4;
```

```
‘^‘ unexpected
```

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

$$4096$$
$$241785163922925834941235 2$$

Syntaxfehler werden bei Fehlermeldung sofort angezeigt: der Strichcursor zeigt auf die Stelle, an der der Fehler erkannt wurde!

```
> 13*(27 - 88*(35/34 + 1/111) + 3/77;
```

```
';' unexpected
```

Mathematische Standardfunktionen können mit numerischen und symbolischen Argumenten benutzt werden. Beispiele für mathematische Funktionen:

```
> exp(1); ln(x+1); sin(arcsin(1));
```

$$e$$

$$\ln(x+1)$$

$$1$$

```
> sin(1.22)^2 + cos(1.22)^2;
```

$$1.000000000$$

```
> abs(x + ln(x+1)); erf(0.8043); signum(ln(0.3));
> signum(y + sqrt(y^2+a));
```

$$|x+\ln(x+1)|$$

$$.7446506070$$

$$-1$$

$$\mathrm{signum}(y+\sqrt{y^2+a})$$

```
> n!; binomial(n,m); binomial(8,3);
```

$$n!$$

$$\mathrm{binomial}(n,\,m)$$

$$56$$

```
> BesselJ(0,x);
```

$$\mathrm{BesselJ}(0,\,x)$$

Smartplot des Ergebnisses durch Anklicken mit rechter Maustaste und Auswahl von Plot ! Änderung des Intervalles im angezeigten Menü ist dann möglich.

```
> BesselJ(1,x);
```

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

Einfügen weiterer Kurven in den Smartplot durch Markieren der gewünschen Formel und Ziehen der linken Maustaste zum Kopf des Smartplots!

```
> diff(%,x);
```

$$\mathrm{BesselJ}(0,\,x)-\frac{\mathrm{BesselJ}(1,\,x)}{x}$$

Auswahl von Strichart, Strichstärke und Farbe im angezeigten Menü ist möglich!

Zahlenarten in Maple

Maple unterscheidet *genaue und genäherte Zahlen*.

Genaue Zahlen (mit exakter Arithmetik):

– integer # Ganzzahlen beliebiger Länge ($< 535\,000$)

– fraction # Brüche

– complex (integer) # Gaußsche Zahlen

```
> 20!;
```

$$2432902008176640000$$

```
> -200!;
```

$$-78865786736479050355236321393218506229513597768717326329474253324435944\backslash$$
$$99634033429203042840119846239041772121389196388302576427902426371\backslash$$
$$05061926624952829931113462857270763317237396988943922445621451664\backslash$$
$$24025403329186413122742829485327752424240757390324032125740557956\backslash$$
$$86602260319041703240623517008587961789222227896237038973747200000\backslash$$
$$00000000000000000000000000000000000000000000$$

```
> 27^5 +84^5 +110^5 +133^5 = 144^5;
```

$$61917364224 = 61917364224$$

```
> 1/(2^101+1) - 1/(2^100+1);
```

$$\frac{-126765060022822940149670320 5376}{32138760885179805510839241846861281568450906757700757802188 81}$$

```
> (1+3*I)^27;
```

$$-21264712441856 + 23405384097792\, I$$

Bemerkung: Mathematische Ausdrücke mit genauen Zahlen werden stets - soweit „möglich“ - genau ausgewertet!

```
> sqrt(5^4) + sqrt((600+40)/2^2);
```

$$25 + 4\sqrt{10}$$

```
> arctan(sin(Pi/2)); arctan(sin(Pi/4));
```

$$\frac{1}{4}\pi$$

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

```
> sin(Pi/4); sin(Pi/8); sin(Pi/16);
```

$$\frac{1}{2}\sqrt{2}$$

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

$$\sin(\frac{1}{16}\pi)$$

Genäherte Zahlen (mit Gleitpunkt-Arithmetik):

- Software - float # GP-Zahlen beliebiger Länge ($< 535\,000$)
- Hardware - float # GP-Zahlen fester Länge (14)

```
> sqrt(5^4) + sqrt((600.0+40)/2^2);
```

$$\mathbf{37.64911064}$$

```
> sqrt(5.0^4) + sqrt((600+40)/2^2);
```

$$\mathbf{25.00000000} + 4\sqrt{\mathbf{10}}$$

```
> evalf(sqrt(5^4) + sqrt((600+40)/2^2));
```

$$\mathbf{37.64911064}$$

```
> evalf(sqrt(5^4) + sqrt((600+40)/2^2),25);
```

$$\mathbf{37.649110640673517327995574}$$

```
> evalf(sqrt(5^4) + sqrt((600+40)/2^2),250);
```

$$\mathbf{37.649110640673517327995574177730874134878220557300867307430019411170377\backslash}$$
$$\mathbf{75455695288537699243351720118074938913661136022059419542412181552\backslash}$$
$$\mathbf{00587620783868006156133796866287170397626366006138964533579364963\backslash}$$
$$\mathbf{41267718308361886305844177477031516248151234439 77}$$

```
> Digits;
```

$$\mathbf{10}$$

```
> Digits := 40;
```

$$\boldsymbol{Digits} := \mathbf{40}$$

```
> evalf(sqrt(5^4) + sqrt((600+40)/2^2));
```

$$\mathbf{37.6491106406735173279955741773087413488}$$

Beispiel:

```
> Digits := 25;
```

$$\boldsymbol{Digits} := \mathbf{25}$$

```
> 262537412640768744; exp(Pi*sqrt(163));
```

$$\mathbf{262537412640768744}$$
$$e^{(\pi\sqrt{\mathbf{163}})}$$

```
> 262537412640768744; evalf(exp(Pi*sqrt(163)));
```

$$\mathbf{262537412640768744}$$
$$\mathbf{.26253741264076874400000000}\,\mathbf{10}^{\mathbf{18}}$$

```
> Digits := 40;
```

$$\boldsymbol{Digits} := \mathbf{40}$$

```
> 262537412640768744; evalf(exp(Pi*sqrt(163)));
```

$$\mathbf{262537412640768744}$$
$$\mathbf{.2625374126}\,\mathbf{10}^{\mathbf{18}}$$

```
> Digits := 10; # Nicht vergessen !
```

$$\boldsymbol{Digits} := \mathbf{10}$$

Genauigkeit bei der Berechnung mathematischer Funktionen:

```
> 6*arcsin(1/2);
```

$$\pi$$

```
> Digits := 100;
```

$$Digits := 100$$

```
> evalf(Pi,40); evalf(6*arcsin(0.5),40);
```

$$3.1415926535897932384626433832795028841 97$$
$$3.1415926535897932384626433832795028841 97$$

```
> evalf(Pi,90) - evalf(6*arcsin(0.5));
```

$$.4657882932\,10^{-89}$$

```
> Digits := 2000;
```

$$Digits := 2000$$

```
> evalf(Pi,1990) - evalf(6*arcsin(0.5));
```

$$-.4780275901\,10^{-1989}$$

```
> Digits := 10; # Nicht vergessen !
```

$$Digits := 10$$

Hardware - Float: Beschleunigung rein numerischer Verfahren durch Double precision Format! Abfrage der Anzahl der durch die Rechnerhardware unterstützten dezimalen Mantissenstellen:

```
> evalhf(Digits);
```

$$14.$$

Bestimmung der Rechenzeit erfolgt durch time( ) .

Vergleich der Rechenzeit für evalf und evalhf:

```
> st := time():
> for i from 1 to 10000 do evalf(exp(1.3)) od:
> time()-st;
```

$$2.966$$

```
> st := time():
> for i from 1 to 10000 do evalhf(exp(1.3)) od:
> time()-st;
```

$$1.263$$

Zuweisungen an Variablen

Maple unterscheidet nicht-zugewiesene (atomare) Variablen und zugewiesene (nicht-atomare) Variablen. Die Zuweisung eines „Ausdrucks-Wertes“ *expression* an eine Variable *name* erfolgt mittels

name := expression;

Atomare Variablen besitzen sich selbst als „Ausdrucks-Wert“. Alle Zuweisungen werden in einer Tabelle abgespeichert.

Beispiele:

```
> a := 1+I; b := u+v*I; # a,b zugewiesen c; u; v; # c,u,v atomar
```

$$a := 1 + I$$
$$b := u + I\,v$$
$$c$$
$$u$$
$$v$$

```
> integral_1 := Int(sin(x)/x^3,x);
```

$$integral\_1 := \int \frac{\sin(x)}{x^3}\,dx$$

```
> gleichung := cot(t) = t;
```

$$gleichung := \cot(t) = t$$

```
> log_aus := not p and not q;
```

$$log\_aus := \textbf{not } (p \textbf{ or } q)$$

```
> m := ‘Das ist eine Zeichenkette¡;
```

$$m := Das\ ist\ eine\ Zeichenkette!$$

Atomisierung von Variablen:

Freigabe einer Variablen durch Zuweisung ihres eigenen Namens als Zeichenkette.

```
> a := 'a';
```

$$a := a$$

```
> a;
```

$$a$$

```
> b; b := 'b': b;
```

$$u + I\,v$$
$$b$$

```
> gleichung := 'gleichung': gleichung;
```

$$gleichung$$

Substitutionsregel:

Tritt eine zugewiesene Variable in einem nachfolgenden Ausdruck auf, so wird bei Auswertung des Ausdruckes ihr *aktueller* Wert eingesetzt (substituiert). Dieser Vorgang wird - rekursiv - solange wiederholt, bis der Audruck nur noch atomare Variablen bzw. Konstanten enthält.

Man beachte: Dieses Substitutionsproblem tritt in klassischen Sprachen (PASCAL, FORTRAN, C) nicht auf.

Einfache Substitution:

```
> a := 3+I;
```

$$a := 3 + I$$

```
> b := a^3 + 7;
```

$$b := 25 + 26\,I$$

```
> c := b^2 - a^2;
```

$$c := -59 + 1294\,I$$

```
> a; b; c;
```

$$3 + I$$
$$25 + 26\,I$$
$$-59 + 1294\,I$$

```
> a := 'a':
> a; b; c;
```

$$a$$
$$25 + 26\,I$$
$$-59 + 1294\,I$$

```
> b := 'b': a; b; c;
```

$$a$$
$$b$$
$$-59 + 1294\,I$$

Mehrfache Substitution:

```
> c := b^2 - a^2;
```

$$c := b^2 - a^2$$

```
> b := a^3 + 7;
```

$$b := a^3 + 7$$

```
> c; # Einfaches Einsetzen
```

$$(a^3 + 7)^2 - a^2$$

```
> b := 'b': c := 'c':
> c := b^2 - a^2;
```

$$c := b^2 - a^2$$

```
> b := a^3 + 7;
```

$$b := a^3 + 7$$

```
> a := 3+I;
```

$$a := 3 + I$$

```
> c; # Mehrfaches Einsetzen
```

$$-59 + 1294\,I$$

Verzögerte Auswertung eines Ausdruckes: Einschließen eines Ausdruckes in Apostrophe unterbindet die komplette Auswertung. Erst beim darauffolgenden Aufruf erfolgt diese (deshalb „verzögert“).

```
> y := diff(exp(x + sin(x)),x);
```

$$y := (1+\cos(x))\, e^{(x+\sin(x))}$$

```
> y := 'diff(exp(x + sin(x)),x)';
```

$$y := \frac{\partial}{\partial x}\, e^{(x+\sin(x))}$$

```
> y;
```

$$(1+\cos(x))\, e^{(x+\sin(x))}$$

```
> x := 15:
> u := '''x^2+1''';
```

$$u := ''x^2+1''$$

```
> v := u;
```

$$v := 'x^2+1'$$

```
> w := v;
```

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

```
> w;
```

$$226$$

Neustart und Informationen zu Variablen

Neustart des Kernes mit Löschung aller während der bisherigen Sitzung zugewiesenen Variablen (leider auch der evtl. geladenen Bibliotheksfunktionen und Pakete):

```
> restart;
```

Empfehlung: Beginn jeder Sitzung mit restart !

Abfrage aller zugewiesenen Variablen mittels anames();
Abfrage aller atomaren Variablen mittels unames() - liefert allerdings auch die Namen zahlreicher intern benutzter Variablen!

```
> anames();
> a := 3+b; c := a^5-2;
```

$$a := 3+b$$

$$c := (3+b)^5-2$$

```
> anames();
```

$$a,\ c$$

Frage: Wie kann man - außer der Hilfefunktion - weitere Informationen über den **internen Aufbau** von Maple-Funktionen und -befehlen erhalten ?

Anzeige des Quellcodes - falls zugelassen - mittels des **print** - Befehles:

```
> interface(verboseproc = 2);
> print(ithprime);

> print(abs);

> print(diff);
```

proc() **option** *builtin*, *remember*; **93 end**

```
> print(sin);
```

Frage: Wie ist der Integrierer (Funktion: int) aufgebaut?

```
> print(int);
> interface(verboseproc = 1);
```

Zusatz für ganz Neugierige: Infolevel

Mit infolevel kann man Informationen während der Abarbeitung von Funktionen erhalten. Infolevel = 1 (Default) liefert nur die notwendigsten Informationen, wogegen Infolevel = 5 (Maximum) alle Informationen im Detail ausgibt. Rücksetzen auf Infolevel = 1 sollte nicht vergessen werden!

```
> infolevel[all] := 5;
> int(1/(x^4+1),x);
> # Vorsicht! int(1/(x^6+1),x);
> infolevel[all] := 1; # Nicht vergessen!
```

5.2.3 Funktionen in Maple

Funktionen als Abbildungen in Maple

Maple bietet folgende 3 wesentlichen Möglichkeiten, eigene Funktionen (d.h. eindeutige Abbildungen) zu definieren:

(1) **Abbildungsschreibweise (Pfeilfunktionen)** - für einzeilige Funktionsausdrücke
(2) **Prozedurschreibweise (Proceduren)** - für beliebige ein- und mehrzeilige Verfahren
(3) **Umwandlung von Ausdrücken** in Funktionen - mit dem Befehl *unapply*

Pfeilfunktionen einer Veränderlichen:

```
> restart;
> f := x -> x^3 + x + 1;
```

$$f := x \to x^3 + x + 1$$

```
> f(0); f(1); f(4);
```

$$1$$
$$3$$
$$69$$

```
> f(z+1);
```

$$(z+1)^3 + z + 2$$

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

$$((x^3+x+1)^3+x^3+x+2)^3+(x^3+x+1)^3+x^3+x+3$$

```
> # smartplot
> fsolve(f(z)=0,z, complex);
```

$$-.6823278038,\ .3411639019 - 1.161541400\,I,\ .3411639019 + 1.161541400\,I$$

```
> integral := int(f(x), x = 0..1);
```

$$integral := \frac{7}{4}$$

```
> g := D(f); # D - Differentialoperator, # angewandt auf f
```

$$g := x \to 3\,x^2 + 1$$

```
> g(x+2);
```

$$3\,(x+2)^2 + 1$$

```
> # smartplot;
```

Definition von Pfeilfunktionen mehrerer Variabler:

```
> phi := (x,y) -> sin(x/3)*cos(y/4);
```

$$\phi := (x,\, y) \rightarrow \sin(\frac{1}{3}\, x)\cos(\frac{1}{4}\, y)$$

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

$$\sin(\frac{1}{3}\, x + \frac{1}{3}\, y)\cos(\frac{1}{4}\, x - \frac{1}{4}\, y)$$

```
> # smartplot3d ;
```

Umwandlung von Ausdrücken in Funktionen

Die Funktion **unapply**, angewandt auf einen mathematischen Ausdruck, erzeugt eine (anonyme) Pfeilfunktion, die diesen Ausdruck berechnet.

Syntax:

unapply(<Ausdruck>, <Var1>, <Var2>, . . .)

```
> restart;
> a := x^3 + x + 1;
```

$$a := x^3 + x + 1$$

```
> unapply(a,x);
```

$$x \rightarrow x^3 + x + 1$$

```
> %; # Anonyme Funktion
```

$$x \rightarrow x^3 + x + 1$$

```
> f := unapply(a,x); # Zuweisung an einen Namen
```

$$f := x \rightarrow x^3 + x + 1$$

```
> f;
```

$$f$$

```
> f(z+1);
```

$$(z+1)^3 + z + 2$$

Funktionen mehrerer Variabler:

```
> diff(sin(x/3)*cos(y/4),x,y);
```

$$-\frac{1}{12}\cos(\frac{1}{3}\, x)\sin(\frac{1}{4}\, y)$$

```
> psi := unapply(%,x,y);
```

$$\psi := (x,\, y) \rightarrow -\frac{1}{12}\cos(\frac{1}{3}\, x)\sin(\frac{1}{4}\, y)$$

```
> psi(3,y);
```

$$-\frac{1}{12}\cos(1)\sin(\frac{1}{4}\,y)$$

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

$$-\frac{1}{12}\cos(\frac{1}{3}\,x-\frac{1}{3}\,y)\sin(\frac{1}{4}\,x+\frac{1}{4}\,y)$$

Prozedurnotation von Funktionen

Frage: Warum ist die Verwendung von Funktionen der Benutzung von Ausdrücken (Termen) vorzuziehen ?

(a) Funktionen erhöhen die Übersichtlichkeit bei symbolischen Rechnungen gegenüber Ausdrücken!

Beispiel: Differentiation einer Funktion / eines Ausdruckes.

```
> y := alpha;
```

$$y := \alpha$$

```
> diff(a,x); diff(a,t); diff(a,y);
```

$$3\,x^2+1$$
$$0$$
$$0$$

```
> diff(f(x),x); diff(f(t),t); diff(f(y),y);
```

$$3\,x^2+1$$
$$3\,t^2+1$$
$$3\,\alpha^2+1$$

(b) Numerische Auswertung von Ausdrücken ist umständlicher als die Benutzung von Funktionen!

Beispiel: Summe von 3 numerischen Werten 1.5 , 1.9 , 3.4 .

```
> x := 1.5; s := a; x := 1.9; s := s+a; x := 3.4; s := s+a;
```

$$x := 1.5$$
$$s := 5.875$$
$$x := 1.9$$
$$s := 15.634$$
$$x := 3.4$$
$$s := 59.338$$

```
> sneu := f(1.5) + f(1.9) + f(3.4);
```

$$sneu := 59.338$$

Prozedurschreibweise (Procedure) :

Diese 3. Methode ist für beliebige (ein- und mehrzeilige) Verfahren anwendbar. Hier sollen nur sehr einfache Prozeduren notiert werden.

Syntax:

<Name> := proc (<Parameterliste>) < Deklaration> end;

Beispiele:

```
> restart;
> f := proc(x) x^3+x+1 end;
```

$$f := \mathbf{proc}(x)\, x^3 + x + 1\, \mathbf{end}$$

```
> sneu := f(1.5) + f(1.9) + f(3.4);
```

$$sneu := 59.338$$

```
> diff(f(z),z);
```

$$3\,z^2 + 1$$

```
> g := proc(x,y,z) # 3 Variablen
> u := f(x+y-z)-f(y); # 2 Anweisungen
> u^2-3*u+1 # Rückgabewert
> end;
```

```
Warning, `u` is implicitly declared local
```

$$g := \mathbf{proc}(x,\, y,\, z)\, \mathbf{local}\, u;\; u := \mathrm{f}(x + y - z) - \mathrm{f}(y)\,;\; u^2 - 3 \times u + 1\, \mathbf{end}$$

```
> g(2,3,4); u; g(x,0,0);
```

$$869$$

$$u$$

$$(x^3 + x)^2 - 3\,x^3 - 3\,x + 1$$

```
> g(s,t,u);
```

$$((s + t - u)^3 + s - u - t^3)^2 - 3\,(s + t - u)^3 - 3\,s + 3\,u + 3\,t^3 + 1$$

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

Verknüpfung von Funktionen

Erzeugung neuer Funktionen aus nutzerdefinierten Funktionen:

```
> restart;
> f := x -> 5*x^4-x+1;
```

$$f := x \to 5\,x^4 - x + 1$$

```
> g := x -> sqrt(x-4);
```

$$g := x \to \sqrt{x - 4}$$

Addition, Multiplikation etc. (2 wesentliche Möglichkeiten):

```
> h := x -> sqrt(f(x))+g(x^2)*f(x);
```

$$h := x \to \sqrt{\mathrm{f}(x)} + \mathrm{g}(x^2)\,\mathrm{f}(x)$$

```
> h(x); h(5.0);
```

$$\sqrt{5\,x^4 - x + 1} + \sqrt{x^2 - 4}\,(5\,x^4 - x + 1)$$

$$14358.08465$$

```
> hh := unapply(sqrt(f(x))+g(x^2)*f(x),x);
```

$$hh := x \to \sqrt{5\,x^4 - x + 1} + \sqrt{x^2 - 4}\,(5\,x^4 - x + 1)$$

```
> hh(x); hh(5.0);
```

$$\sqrt{5\,x^4 - x + 1} + \sqrt{x^2 - 4}\,(5\,x^4 - x + 1)$$

$$14358.08465$$

Verknüpfung von Funktionen durch Verkettung (Hintereinanderausführung):

```
> f := x-> exp(x); g := x -> sqrt(x^2-4);
```

$$f := \exp$$

$$g := x \to \sqrt{x^2 - 4}$$

```
> h := x -> f(g(x));
```

$$h := x \to \mathrm{f}(\mathrm{g}(x))$$

```
> h(x);
```

$$e^{(\sqrt{x^2-4})}$$

Stückweise definierte Funktionen:

Soll eine Funktion mit mehreren Bereichen definiert werden, so kann dies mittels geschachtelter **if** - Anweisungen oder mittels der **piecewise** - Anweisung geschehen.

Beispiel:

$$\phi := \begin{cases} -x^3 & \text{für } x < 0 \\ 1 & \text{für } 0 \le x < 0.5\pi \\ \sin x & \text{für } x \ge 0.5\pi \end{cases}$$

```
> phi := x -> if x<0 then -x^3 elif x<evalf(Pi/2) then 1 else sin(x)
fi;
```

$$\phi := \mathbf{proc}(x)$$
$$\quad \mathbf{option}\ operator,\ arrow;$$
$$\quad\quad \mathbf{if}\ x < 0\ \mathbf{then}\ -x^3\ \mathbf{elif}\ x < \mathrm{evalf}(1/2 \times \pi)\ \mathbf{then}\ 1\ \mathbf{else}\ \sin(x)\ \mathbf{fi}$$
$$\quad \mathbf{end}$$

```
> phi(4.5); phi(t);
```

$$-.9775301177$$

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

```
> plot(phi(x), x=-1.0 .. -4);
```

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

Die **if** - Anweisung besitzt Nachteile bei der Auswertung - trotz Anwendung von *evalf* ! Empfehlung: Notation mit der **piecewise** - Anweisung.

Syntax:

piecewise(<Bed1> , <Ausdruck1> ,
<Bed2> , <Ausdruck2> ,
. . . . . ,
<Sonst-Zweig>);

```
> psi := x -> piecewise(x<0, -x^3, x<Pi/2, 1, sin(x));
```

$$\psi := x \to \mathbf{piecewise}(x < 0,\ -x^3,\ x < \frac{1}{2}\,\pi,\ 1,\ \sin(x))$$

```
> psi(4.5); psi(t);
```

$$-.9775301177$$

$$\begin{cases} -t^3 & t < 0 \\ 1 & t < \frac{1}{2}\,\pi \\ \sin(t) & otherwise \end{cases}$$

```
> # smartplot!
```

Differentiation und Integration sind möglich.

```
> diff(psi(x),x);
```

$$\begin{cases} -3\,x^2 & x < 0 \\ 0 & x \le \frac{1}{2}\,\pi \\ \cos(x) & \frac{1}{2}\,\pi < x \\ undefined & x = 0 \end{cases}$$

Es wird erkannt, daß die Ableitung bei x = 0 nicht existiert !

```
> stamm := int(psi(x),x);
```

$$stamm := \begin{cases} -\frac{1}{4}\,x^4 & x \le 0 \\ x & x \le \frac{1}{2}\,\pi \\ -\cos(x) + \frac{1}{2}\,\pi & \frac{1}{2}\,\pi < x \end{cases}$$

Die Integrationskonstanten werden so gewählt, daß die Stammfunktion stetig wird.

```
> sigma := unapply(stamm,x);
```

$$\sigma := x \to \mathbf{piecewise}(x \le 0,\ -\frac{1}{4}\,x^4,\ x \le \frac{1}{2}\,\pi,\ x,\ \frac{1}{2}\,\pi < x,\ -\cos(x) + \frac{1}{2}\,\pi)$$

5.2.4 Datenstrukturen in Maple

Folgen, Listen und Mengen

Maple unterscheidet folgende **grundlegenden** Datenstrukturen (auch in ihrer unterschiedlichen internen Darstellung):

- Folge (expression sequence)
- Liste (list)
- Menge (set)
- Feld (array) , speziell Vektor und Matrix
- Tabelle (table) .

Felder sind - anders als z.B. in MATHEMATICA - nicht identisch mit Listen!

1. Folge von Ausdrücken

Eine Folge besteht aus einem oder mehreren Ausdrücken, getrennt durch Kommata. NULL steht für die leere Folge.

```
> restart;
> loesung := solve(x^6-1=0, x);
```

$$loesung := 1, -1, -\frac{1}{2} - \frac{1}{2} I\sqrt{3}, -\frac{1}{2} + \frac{1}{2} I\sqrt{3}, \frac{1}{2} + \frac{1}{2} I\sqrt{3}, \frac{1}{2} - \frac{1}{2} I\sqrt{3}$$

```
> variable := x,y,z: Diff(x^2 * (y+z)^3,variable);
> diff(x^2 * (y+z)^3,variable);
```

$$\frac{\partial^3}{\partial z\, \partial y\, \partial x} x^2 (y+z)^3$$

$$12\, x\, (y+z)$$

Zugriff auf einzelne Elemente der Folge erfolgt mit dem Auswahloperator **[ ]**:

```
> loesung[3], ReTeil := (loesung[5]+loesung[6])/2;
```

$$-\frac{1}{2} - \frac{1}{2} I\sqrt{3}\,, \quad ReTeil := \frac{1}{2}$$

```
> Diff(x^2 *(y+z)^3, variable[3]);
```

$$\frac{\partial}{\partial z} x^2 (y+z)^3$$

Der Folgenoperator $ generiert Folgen mit Bildungsvorschrift:

```
> x $6; diff(x^20, x$12);
```

$$x, x, x, x, x, x$$

$$60339831552000\, x^8$$

```
> y^2 $ y = 2 .. 7; x^ithprime(n) $ n = 30 .. 37;
```

$$4, 9, 16, 25, 36, 49$$

$$x^{113}, x^{127}, x^{131}, x^{137}, x^{139}, x^{149}, x^{151}, x^{157}$$

2. Liste von Ausdrücken

Eine Liste ist eine in [ ] eingeschlossene Folge von Ausdrücken, deren Elemente geordnet sind; die leere Liste ist [] .

```
> monome := [1,x,x^2,x^3,x^4,x^5];
```
$$monome := [1, x, x^2, x^3, x^4, x^5]$$
```
> varlist := [x,y,z];
```
$$varlist := [x, y, z]$$
```
> monome[4]-5*monome[6]; # Zugriff auf # Einzelelemente
```
$$x^3 - 5\,x^5$$

Der Zugriff auf einen Teilbereich der Liste geschieht mittels **[a .. b]** ; dagegen liefert **op(<Liste>);** alle Listenelemente in der Form einer Folge. Die Anzahl aller Listenelemente erhält man mittels des Kommandos **nops(<Liste>);**

```
> monome[3 .. 5];
```
$$[x^2, x^3, x^4]$$
```
> op(monome); # Ergebnis ist eine Folge!
```
$$1, x, x^2, x^3, x^4, x^5$$
```
> nops(monome);
```
$$6$$

Das Zusammenhängen von Listen führt man mittels op aus:

```
> biglist := [op(monome),op(varlist)];
```
$$biglist := [1, x, x^2, x^3, x^4, x^5, x, y, z]$$

Die Anwendung von Funktionen f auf alle Elemente einer Liste `list` kann mittels des Befehles `map` vorgenommen werden:

map(<Funktionsname>, <Liste>);

```
> map(exp,monome);
```
$$[e, e^x, e^{(x^2)}, e^{(x^3)}, e^{(x^4)}, e^{(x^5)}]$$
```
> map(phi,biglist);
```
$$[\phi(1), \phi(x), \phi(x^2), \phi(x^3), \phi(x^4), \phi(x^5), \phi(x), \phi(y), \phi(z)]$$

Besitzt die Funktion f mehr als ein Argument, so sind diese zusätzlichen Argumente als weitere Argumente in `map` anzugeben. Um z.B. alle Elemente der Liste `biglist` nach x zu differenzieren, notiert man

```
> map(diff,biglist,x);
```
$$[0, 1, 2\,x, 3\,x^2, 4\,x^3, 5\,x^4, 1, 0, 0]$$

3. Menge von Ausdrücken

Eine Menge ist eine in { } eingeschlossene Folge von Ausdrücken, deren Elemente **ungeordnet** sind. Die leere Menge ist { } .

```
> monome := {1,x,x^2,x^3,x^4,x^5};
```
$$monome := \{x^2, x^3, 1, x, x^4, x^5\}$$
```
> varset := {x,y,z};
```
$$varset := \{x, z, y\}$$
```
> monome[4]-5*monome[6]; # Zugriff auf # Einzelelemente
```
$$x - 5\,x^5$$

Der Zugriff auf alle Mengenelemente erfolgt mittels **op(<Menge>);** die Elementezahl der Menge kann mit dem Kommando **nops(<Menge>);** abgefragt werden.

```
> op(monome); # Ergebnis ist eine Folge!
```
$$x^2, x^3, 1, x, x^4, x^5$$
```
> nops(monome);
```
$$6$$

Die Mengenoperationen (Elemente, Durchschnitt, Vereinigung und Differenzmenge) können mittels der Kommandos **op, intersect, union, minus** ausgeführt werden:

```
> bigset := {op(monome),op(varset)};
```
$$bigset := \{x^2, x^3, 1, x, z, y, x^4, x^5\}$$
```
> vereinigung := monome union varset;
```
$$vereinigung := \{x^2, x^3, 1, x, z, y, x^4, x^5\}$$
```
> durchschnitt := monome intersect varset;
```
$$durchschnitt := \{x\}$$

Umwandlungen zwischen **Folgen, Listen** und **Mengen** kann man mittels **op** leicht vornehmen. Achtung: Bei Mengendarstellung wird die Reihenfolge oft intern geändert (siehe obiges Beispiel) !

```
> monome := 1,x,x^2,x^3,x^4,x^5;
```
$$monome := 1, x, x^2, x^3, x^4, x^5$$
```
> mon_list := [monome];
```
$$mon\_list := [1, x, x^2, x^3, x^4, x^5]$$
```
> mon_set := {monome};
```
$$mon\_set := \{x^2, x^3, 1, x, x^4, x^5\}$$
```
> list_to_set := {op(mon_list)};
```
$$list\_to\_set := \{x^2, x^3, 1, x, x^4, x^5\}$$

Ein Anwendungsbeispiel für Listen

Man entwickle eine Funktion cartes zur Umwandlung von Kugelkoordinaten (r,phi,theta) in cartesische Koordinaten (x,y,z).

Hinweis: Funktionen liefern stets genau 1 Resultat, dieses kann jedoch strukturiert sein - also z.B. eine Liste oder eine geschachtelte Liste !

```
> restart;
> cartes1 := (r,phi,theta) -> [r*cos(phi)*cos(theta),
  r*sin(phi)*cos(theta), r*sin(theta)];
```

$$cartes1 := (r, \phi, \theta) \rightarrow [r\cos(\phi)\cos(\theta), r\sin(\phi)\cos(\theta), r\sin(\theta)]$$

```
> p1 := cartes1(10,0,Pi/2); p2 := cartes1(10,Pi/2,0);
  p3 := cartes1(10,Pi/4,0);
```

$$p1 := [0, 0, 10]$$
$$p2 := [0, 10, 0]$$
$$p3 := [5\sqrt{2}, 5\sqrt{2}, 0]$$

```
> x_3 := p3[1];
```

$$x\_3 := 5\sqrt{2}$$

Anstelle der Pfeilfunktion kann die Transformation auch als Prozedur dargestellt werden:

```
> cartes2 := proc(r,phi,theta)
    [r*cos(phi)*cos(theta), r*sin(phi)*cos(theta),
    r*sin(theta)]
  end;
```

$$cartes2 := \textbf{proc}(r, \phi, \theta)\,[r \times \cos(\phi) \times \cos(\theta), r \times \sin(\phi) \times \cos(\theta), r \times \sin(\theta)]\ \textbf{end}$$

```
> p1 := cartes2(10,0,Pi/2); p2 := cartes2(10,Pi/2,0);
  p3 := cartes2(10,Pi/4,0);
```

$$p1 := [0, 0, 10]$$
$$p2 := [0, 10, 0]$$
$$p3 := [5\sqrt{2}, 5\sqrt{2}, 0]$$

```
> y_2 := p2[2];
```

$$y\_2 := 10$$

Unnötige Doppelberechnungen lassen sich durch Definition lokaler Variablen vermeiden:

```
> cartes3 := proc(r,phi,theta)
    local h;
    h := r*cos(theta);
    [cos(phi)*h, sin(phi)*h, r*sin(theta)]
  end;
```

$$cartes3 := \textbf{proc}(r, \phi, \theta)\ \textbf{local}\ h;\ h := r \times \cos(\theta)\ ;\ [\cos(\phi) \times h, \sin(\phi) \times h, r \times \sin(\theta)]\ \textbf{end}$$

```
> p1 := cartes3(10,0,Pi/2); p2 := cartes3(10,Pi/2,0);
  p3 := cartes3(10,Pi/4,0);
```

$$p1 := [0,\ 0,\ 10]$$
$$p2 := [0,\ 10,\ 0]$$
$$p3 := [5\sqrt{2},\ 5\sqrt{2},\ 0]$$

```
> z_1 := p1[3];
```

$$z\_1 := 10$$

Felder (arrays) und Pakete (packages)

Ein Feld ist eine **mehrdimensionale** Datenstruktur, die durch Bereiche ganzer Zahlen indiziert wird.

```
> restart;
> a := array(0 .. 2, -3 .. -2); a[1,-3] := 10;
  a[1,-2] := 11; print(a);
```

$$a := \mathrm{array}(0..2,\ -3..-2,\ [])$$
$$a_{1,-3} := 10$$
$$a_{1,-2} := 11$$

$$\mathrm{array}(0..2,\ -3..-2,\ [$$
$$(0,\ -3) = a_{0,-3}$$
$$(0,\ -2) = a_{0,-2}$$
$$(1,\ -3) = 10$$
$$(1,\ -2) = 11$$
$$(2,\ -3) = a_{2,-3}$$
$$(2,\ -2) = a_{2,-2}$$
$$])$$

```
> b := array(1..2,1..2,1..2): print(b);
```

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

Hinweis: Zur einfacheren Arbeit mit 1-dimensionalen Feldern (Vektoren) und 2-dimensionalen Feldern (Matrizen) benutze man das Paket **linalg** !

Nutzung von Befehlen aus Paketen

Befehle eines Paketes können einzeln geladen - d.h. aktiviert - werden, indem dem Befehlsnemen der Name des Paketes vorangestellt wird. Benötigt man jedoch häufiger die Befehle eines Paketes, z.B. für grafische Darstellungen oder für Operationen in der linearen Algebra, so empfiehlt sich das komplette Laden des betreffenden Paketes vor der Benutzung einzelner Befehle. Diese können dann so aktiviert werden, als seien es Funktionen des Maple-Kerns.

(1) Laden eines einzelnen Befehls command aus einem package:

Syntax:

package[command](<Parameter>)

Beispiel:

```
> v1 := linalg[vector]([a,a+3,a+5,a+7]);
```

$$v1 := [a,\ a+3,\ a+5,\ a+7]$$

```
> v2 := linalg[vector]([1,1,1,1]);
```

$$v2 := [1,\ 1,\ 1,\ 1]$$

```
> v2a := linalg[scalarmul](v2,a);
```

$$v2a := [a,\ a,\ a,\ a]$$

```
> res := linalg[matadd](v1,v2a);
```

$$res := [2\,a,\ 2\,a+3,\ 2\,a+5,\ 2\,a+7]$$

(2) Laden eines kompletten Paketes und anschließende Nutzung seiner Befehle wie Befehle des Kernes:

Syntax:

with (package):
command(<Parameter>)

Beispiel:

```
> with(linalg):

Warning, new definition for norm

Warning, new definition for trace
> w1 := vector([b,b+3,b+5,b+7]);
```

$$w1 := [b,\ b+3,\ b+5,\ b+7]$$

```
> w2 := vector([1,1,1,1]);
```

$$w2 := [1,\ 1,\ 1,\ 1]$$

```
> w2b := scalarmul(v2,b);
```

$$w2b := [b,\ b,\ b,\ b]$$

```
> res := matadd(w1,w2b);
```

$$res := [2\,b,\ 2\,b+3,\ 2\,b+5,\ 2\,b+7]$$

Auflisten aller Funktionen eines Paketes sowie Nutzung der Help-on-Topic- Funktion :

```
> with(linalg);
```

```
Warning, new definition for norm

Warning, new definition for trace
```

[*BlockDiagonal*, *GramSchmidt*, *JordanBlock*, *LUdecomp*, *QRdecomp*, *Wronskian*, *addrow*, *adj*, *adjoint*, *angle*, *augment*, *backsub*, *band*, *basis*, *bezout*, *blockmatrix*, *charpoly*, *cholesky*, *col*, *coldim*, *colspace*, *colspan*, *companion*, *concat*, *cond*, *crossprod*, *curl*, *definite*, *delcols*, *delrows*, *det*, *diag*, *diverge*, *dotprod*, *eigenvals*, *eigenvalues*, *eigenvectors*, *eigenvects*, *entermatrix*, *equal*, *exponential*, *extend*, *ffgausselim*, *fibonacci*, *forwardsub*, *frobenius*, *gausselim*, *gaussjord*, *geneqns*, *grad*, *hadamard*, *hermite*, *hessian*, *hilbert*, *htranspose*, *ihermite*, *indexfunc*, *intbasis*, *inverse*, *ismith*, *issimilar*, *iszero*, *jacobian*, *jordan*, *kernel*, *laplacian*, *linsolve*, *matadd*, *matrix*, *minor*, *minpoly*, *mulcol*, *mulrow*, *multiply*, *norm*, *nullspace*, *orthog*, *permanent*, *pivot*, *potential*, *randmatrix*, *randvector*, *rank*, *row*, *rowdim*, *rowspace*, *rowspan*, *rref*, *scalarmul*, *singularvals*, *smith*, *submatrix*, *subvector*, *sumbasis*, *swapcol*, *swaprow*, *sylvester*, *toeplitz*, *trace*, *vandermonde*, *vecpotent*, *vectdim*, *vector*, *wronskian*]

5.2.5 Elementare lineare Algebra

Fundamentale Matrixarithmetik (linalg)

Die Nutzung des Paketes linalg ist empfehlenswert!

```
> restart: with(linalg):
```

```
Warning, new definition for norm

Warning, new definition for trace
```

Die Definition von Vektoren erfolgt mittels der Funktionen **vector** (<Liste>) bzw. **vector** (n,<Liste>)

```
> u := vector([b,b+3,b+5]);
```

$$u := [b,\ b+3,\ b+5]$$

```
> v := vector(3,[b,b-1,b-2]);
```

$$v := [b,\ b-1,\ b-2]$$

```
> delta := matadd(u,-v);
```

$$\delta := [0,\ 4,\ 7]$$

```
> skal_prod := dotprod(u,v);
```

$$skal\_prod := b\,\overline{b} + (b+3)\,(-1+\overline{b}) + (b+5)\,(-2+\overline{b})$$

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

$$3\,|\Re(b)+I\,\Im(b)|^2+5\,\Re(b)-11\,I\,\Im(b)-13$$

```
> vekt_prod := crossprod(u,v);
```

$$vekt\_prod := [(b+3)\,(b-2)-(b+5)\,(b-1),\ (b+5)\,b-b\,(b-2),\ b\,(b-1)-(b+3)\,b]$$

```
> w := simplify(%);
```

$$w := [-3\,b-1,\ 7\,b,\ -4\,b]$$

```
> norm(w); norm(w,2); norm(w,15); normalize(w);
```

$$\max(|3\,b+1|,\ 7\,|b|)$$

$$\sqrt{|3\,b+1|^2+65\,|b|^2}$$

$$(|3\,b+1|^{15}+4748635251767\,|b|^{15})^{(1/15)}$$

$$\left[\frac{-3\,b-1}{\sqrt{|3\,b+1|^2+65\,|b|^2}},\ 7\,\frac{b}{\sqrt{|3\,b+1|^2+65\,|b|^2}},\ -4\,\frac{b}{\sqrt{|3\,b+1|^2+65\,|b|^2}}\right]$$

Die Definition von Matrizen erfolgt mittels der Funktionen

matrix (<Liste>) bzw.
matrix (m,n,<Liste>) bzw.
matrix (m,n,<Funktion>) bzw.
matrix (m,n, vector (<Liste>))

```
> A := matrix(2,2,[a,b,c,d]);
```

$$A := \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

```
> B := matrix([[p,q],[r,s]]);
```

$$B := \begin{bmatrix} p & q \\ r & s \end{bmatrix}$$

```
> C := matadd(A,B);
```

$$C := \begin{bmatrix} a+p & b+q \\ c+r & d+s \end{bmatrix}$$

```
> P := multiply(B,A);
```

$$P := \begin{bmatrix} p\,a+q\,c & p\,b+q\,d \\ r\,a+s\,c & r\,b+s\,d \end{bmatrix}$$

```
> transpose(A); trace(A); det(A); rank(A);
    # a,b,c,d sind beliebig!
  inverse(A);
```

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

$$a + d$$

$$a\,d - b\,c$$

$$2$$

$$\begin{bmatrix} \dfrac{d}{a\,d - b\,c} & -\dfrac{b}{a\,d - b\,c} \\ -\dfrac{c}{a\,d - b\,c} & \dfrac{a}{a\,d - b\,c} \end{bmatrix}$$

```
> norm(A); norm(A,1); norm(A,frobenius);
```

$$\max(|a| + |b|,\ |c| + |d|)$$

$$\max(|a| + |c|,\ |b| + |d|)$$

$$\sqrt{|a|^2 + |b|^2 + |c|^2 + |d|^2}$$

```
> A; # keine Darstellung evalm(A); # explizite Darstellung
```

$$A$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

```
> G := matrix(2,3,[s,t,t-s,3*s,3*t,3*(t-s)]);
```

$$G := \begin{bmatrix} s & t & t - s \\ 3\,s & 3\,t & 3\,t - 3\,s \end{bmatrix}$$

```
> rank(A), rank(C), rank(G); # s und t sind beliebig!
```

$$2,\ 2,\ 1$$

Lineare Gleichungssysteme

Für die Lösung kleiner Gleichungssysteme kann **solve** benutzt werden:

solve(<Gleichung> , <Variable>)
solve({<Gleichungen>} , {<Variablen>})

```
> solve({2*x+y=4, x-lambda*y=-2}, {x,y});
```

$$\{y = 8\,\frac{1}{1 + 2\,\lambda},\ x = 2\,\frac{2\,\lambda - 1}{1 + 2\,\lambda}\}$$

```
> lambda := -1/2: solve({2*x+y=4, x-lambda*y=2}, {x,y});
  # Lineare Abhängigkeit
```

$$\{y = -2\,x + 4,\ x = x\}$$

```
> solve({2*x+y=4, x-lambda*y=-2}, {x,y}); # Keine Lösung
```

Für große Gleichungssysteme ist die Matrixschreibweise und die Nutzung von **linsolve** aus dem Paket **linalg** vorzuziehen !

```
> restart: with(linalg):

Warning, new definition for norm

Warning, new definition for trace
> A := matrix([[1,1,-2],[1,-1,-2],[2,3,-4]]);
```

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

```
> rank(A);
```

$$2$$

```
> b := vector([0,0,0]); # Darstellung von Vektoren geschieht #
stets zeilenweise!
```

$$b := [0, 0, 0]$$

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

$$[2\_t_1, 0, \_t_1]$$

```
> nullspace(A); kernel(A);
```

$$\{[2, 0, 1]\}$$
$$\{[2, 0, 1]\}$$

Parameterabhängige Gleichungssysteme können in vielen Fällen mit Maple gelöst werden:

```
> A[3,3] := -4+epsilon: evalm(A);
```

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

Die Parameter werden als beliebige (reelle bzw. komplexe) Zahlen angenommen. Dieser **„generische Fall“** schließt solche speziellen Werte des Parameters aus, die zu „degenerierten Fällen“ führen.

```
> linsolve(A,b), rank(A);
```

$$[0, 0, 0], 3$$

```
> b[1] := 1: linsolve(A,b);
```

$$\left[\frac{1}{2}\ \frac{\varepsilon-10}{\varepsilon}, \frac{1}{2}, -\frac{5}{2}\ \frac{1}{\varepsilon}\right]$$

```
> nullspace(A);
```

$$\{\}$$

Beispiel: Normalbestimmtes lineares Gleichungssystem

(vgl. WESTERMANN, Teil1, S.117f):

```
> restart: with(linalg):
```

Gegeben sei das lineare Gleichungssystem mit Koeffizientenmatrix

```
> A := matrix([[-3,0,6,0],[1,1,-2,5], [1,0,-2,0],[-2,-2,4,-10]]);
```

$$A := \begin{bmatrix} -3 & 0 & 6 & 0 \\ 1 & 1 & -2 & 5 \\ 1 & 0 & -2 & 0 \\ -2 & -2 & 4 & -10 \end{bmatrix}$$

und dem Vektor der rechten Seiten

```
> b := vector([-3,2,1,-4]);
```

$$b := [-3, 2, 1, -4]$$

Vergleich des Rangs von A und (A—b) liefert:

```
> rank(A);
```

$$2$$

```
> augment(A,b): rank(%);
```

$$2$$

Die allgemeine Lösung mit 2 Parametern lautet:

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

$$[1 + 2\,\_t_1,\ 1 - 5\,\_t_2,\ \_t_1,\ \_t_2]$$

```
> kernel(A);
```

$$\{[2, 0, 1, 0], [0, -5, 0, 1]\}$$

Veränderung von b und Vergleich des Rangs von A und (A—b) ergibt:

```
> b[4] := -4 - 10^(-30): evalm(b);
```

$$\left[-3, 2, 1, \frac{-4000000000000000000000000000001}{1000000000000000000000000000000}\right]$$

```
> rank(A); augment(A,b): rank(%);
```

$$2$$
$$3$$

```
> linsolve(A,b); # Keine Lösung
```

Eigenwerte und -vektoren von Matrizen

Das Maple-Package linalg bietet Kommandos zur Ermittlung der Eigenwerte und Eigenvektoren symbolischer Matrizen. Nachfolgend werde eine parameterabhängige Matrix A definiert:

```
> restart: with(linalg):
> A := matrix([[1,1,-2],[1,-1,-2],[2,3,-4]]);
```

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

```
> A[3,3] := -4+epsilon; evalm(A);
```

$$A_{3,3} := -4 + \varepsilon$$

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

Die Bestimmung von Eigenwerten kann mittels **charpoly + solve** oder mittels **eigenvals** vorgenommen werden:

```
> p := charpoly(A,lambda);
```

$$p := \lambda^3 + 4\,\lambda^2 - \lambda^2\,\varepsilon + 8\,\lambda + 2\,\varepsilon$$

```
> solve(p,lambda);
> epsilon := 0: # Spezieller Parameterwert eigenvals(A);
```

$$0,\ -2 + 2\,I,\ -2 - 2\,I$$

Bestimmung von Eigenwerten, deren Vielfachheiten und der Basen der zugehörigen Eigenräume:

Syntax: **linalg [eigenvects] (<Matrix>)**

Das Ergebnis ist eine Folge von Listen; jede Liste enthält:

(1) den Eigenwert
(2) die algebraische Vielfachheit des Eigenwertes
(3) die Basis des zugehörigen Eigenraumes.

```
> eigenvects(A);
```

$$[0,\ 1,\ \{\}],\ [-2+2\,I,\ 1,\ \{\left[\frac{8}{17} + \frac{2}{17}\,I,\ \frac{6}{17} + \frac{10}{17}\,I,\ 1\right]\}],\ [-2-2\,I,\ 1,\ \{\left[\frac{8}{17} - \frac{2}{17}\,I,\ \frac{6}{17} - \frac{10}{17}\,I,\ 1\right]\}]$$

Parameterabhängige Matrizen können vorteilhaft **als Funktion** definiert werden:

```
> M := epsilon -> matrix(2,2,[4,epsilon,0,4]);
```

$$M := \varepsilon \rightarrow \mathbf{matrix}(\mathbf{2}, \mathbf{2}, [\mathbf{4}, \varepsilon, \mathbf{0}, \mathbf{4}])$$

```
> M(-2.7+alpha^2); M(0); det(M(Pi));
```

$$\begin{bmatrix} 4 & -2.7 + \alpha^2 \\ 0 & 4 \end{bmatrix}$$

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

$$16$$

1. Fall : Geometrische Vielfachheit = 2 (diagonalisierbar), falls eps = 0 :

```
> eigenvects(M(0));
```

$$[4, 2, \{[0, 1], [1, 0]\}]$$

2. Fall : Geometrische Vielfachheit = 1 (nicht diagonalisierbar), falls eps <> 0 :

```
> epsilon := 'epsilon': # Parameter wird atomar! evalm(M(epsilon));
```

$$\begin{bmatrix} 4 & \varepsilon \\ 0 & 4 \end{bmatrix}$$

```
> eigenvects(M(epsilon));
```

$$[4, 2, \{[1, 0]\}]$$

5.3 Kontrollstrukturen in Maple

Maple verfügt über 3 Typen nicht-prozeduraler Anweisungen, die den Kontrollfluß eines Programmes steuern:

- die if-Anweisung für Alternativen(Selektionen)
- die for-while-Anweisung für Zyklen (Schleifen)
- die break- und die next-Anweisung für Zyklusabbrüche.

Auch done, quit, stop und andere Anweisungen beeeinflussen den Kontrollfluß.

5.3.1 Die if-Anweisung für Alternativen

Die allgemeine Syntax der if-Anweisung lautet:

```
if condition then
    statement-sequence
elif condition then
    statement-sequence
elif condition then
    statement-sequence
. . . . . .
else
    statement-sequence
fi;
```

Die elif- und die else-Klauseln sind optional. Es sind mehrere elif-Klauseln zulässig.
Abarbeitung:

1. Berechnung der Bedingung (Boolescher Ausdruck)
2. Falls sein Wert true ist, so Berechnung der Anweisungsfolge der then-Klausel und Beendigung der if-Anweisung
3. Falls sein Wert false ist, so
   (a) Übergang zur elif-Klausel, falls vorhanden
   (b) Übergang zur else-Klausel, falls elif nicht vorhanden bzw.
   (c) Beendigung der if-Anweisung ohne Aktion, falls keine der beiden Klauseln vorhanden ist.

Bedingungen und Boolesche Operatoren

Es stehen die Operatoren and, or, not mit den üblichen Bedeutungen und Prioritäten zur Verfügung. Not bindet stärker als and, and bindet stärker als or. Die Auswertung erfolgt nach den McCarthy-Regeln: Abarbeitung von links nach rechts mit vorzeitigem Abbruch bei true (or) bzw. false (and).

```
> restart:
> is(sin(1), positive);
```

$$true$$

```
> x:=0: bed1:= x<>0 and x^2>2 and exp(x-1.0)>0;
```

$$bed1 := false$$

```
> x:=0: bed2:= x<>0 and 1/x=0;
```

$$bed2 := false$$

```
> x:=2.5: y:=x^2+pi: bed3:= x<>0 and y^2>2 or
  not(x-y=0 and exp(x-1.0)>0);
```

$$bed3 := true$$

```
> tautologie1 := not(p and q) = (not p or not q);
```

$$tautologie1 := true$$

```
> unbestimmt1 := not p and not q;
```

$$unbestimmt1 := \textbf{not } (p \textbf{ or } q)$$

```
> unbestimmt2 := a^2+b^2-c^2;
```

$$unbestimmt2 := a^2 + b^2 - c^2$$

```
> is(unbestimmt2,positive);
```

$$FAIL$$

```
> a:=3: b:=4: c:=5: is(unbestimmt2,positive);
```

$$false$$

Ist eine Bedingung nicht vollständig auswertbar, so ist ihr Wert „unbestimmt“, wofür Maple die Konstante FAIL benutzt. Die Berechnung Boolescher Ausdrücke geschieht dann mittels einer dreiwertigen Logik mit den Wahrheitswerten {false, true, FAIL} (vgl. Wertetafeln).

Frage: Wie werden if-Anweisungen ausgewertet, wenn FAIL auftritt ?

```
> if FAIL then print(1) else print(2) fi;
```

$$2$$

```
> if FAIL then print(1) fi;
> if not FAIL then print(1) fi;
> if not FAIL then print(1) else print(2) fi;
```

$$2$$

Einfache Alternativen (if-then-else)

Beispiel 1: Dreiecksuntersuchung

```
> restart:
> pythagoras:= a^2+b^2-c^2;
```

$$pythagoras := a^2 + b^2 - c^2$$

```
> if pythagoras=0 then 'rechtwinklig' fi; # Keine Zahlenwerte ->
FAIL !
> a:=6: b:=8: c:=10: if pythagoras=0 then 'rechtwinklig' fi;
```

$$rechtwinklig$$

```
> a:=5: b:=8: c:=10: if pythagoras=0 then 'rechtwinklig' fi; #
Keine Aussage im anderen Fall !
> a:=5: b:=8: c:=10: if pythagoras=0 then 'rechtwinklig' else
   'nichtrechtwinklig' fi;
```

$$nichtrechtwinklig$$

Beispiel 2: Maximum zweier und dreier Zahlen

```
> restart:
> maxi2 := (x,y)-> if x>y then x else y fi;
```

$maxi2 :=$ **proc**(x, y) **option** $operator, arrow$; **if** $y < x$ **then** x **else** y **fi end**

```
> maxi2(x,y);

Error, (in maxi2) cannot evaluate boolean
> maxi2(3,-2);
```

$$3$$

```
> maxi3 := (x,y,z)->
    if x>y then
       if x>z then x
              else z
       fi
    else
      if y<z then y
             else z
      fi
    fi;
```

$maxi3 :=$ **proc**(x, y, z)
option $operator, arrow$;
if $y < x$ **then if** $z < x$ **then** x **else** z **fi else if** $y < z$ **then** y **else** z **fi fi**
end

```
> maxi3(4,2,6); maxi3(4,2,4); maxi3(sin(1.0),0,-1);
```

$$\mathbf{6}$$
$$\mathbf{4}$$
$$\mathbf{.8414709848}$$

```
> maxi3(sin(1),sqrt(2),Pi); # Exakte Darstellung, keine GP-Zahlen

Error, (in maxi3) cannot evaluate boolean

> maxi3(sin(1.0),sqrt(2.0),Pi); # Pi ist Symbol!

Error, (in maxi3) cannot evaluate boolean

> maxi3(sin(1.0),sqrt(2.0),evalf(Pi));
```

$$\mathbf{1.414213562}$$

Mehrfache Alternativen (if-then-elif)

Beispiel 1: Dreiecksuntersuchung (verbessert)

Die 3 Fälle recht-, spitz- und stumpfwinkliger Dreiecke sollen unterschieden werden.

```
> restart:
> pythagoras:= a^2+b^2-c^2;
```

$$pythagoras := a^2 + b^2 - c^2$$

```
> if pythagoras=0 then 'rechtwinklig' elif pythagoras<0 then
  'stumpfwinklig' else 'spitzwinklig' fi;
  # Keine Zahlenwerte -> FAIL !

Error, cannot evaluate boolean

> a:=6: b:=8: c:=10: if pythagoras=0 then 'rechtwinklig' elif
  pythagoras<0 then 'stumpfwinklig' else 'spitzwinklig' fi;
```

$$rechtwinklig$$

```
> a:=2: b:=8: c:=10: if pythagoras=0 then 'rechtwinklig' elif
  pythagoras<0 then 'stumpfwinklig' else 'spitzwinklig' fi;
```

$$stumpfwinklig$$

```
> a:=6: b:=8: c:=1: if pythagoras=0 then 'rechtwinklig' elif
  pythagoras<0 then 'stumpfwinklig' else 'spitzwinklig' fi;
```

$$spitzwinklig$$

```
> a:=2: b:=3: c:=10: if pythagoras=0 then 'rechtwinklig' elif
  pythagoras<0 then 'stumpfwinklig' else 'spitzwinklig' fi;
```

$$stumpfwinklig$$

Das ist natürlich Unsinn, denn mit diesen drei Seiten existiert kein Dreieck ! Setzen wir voraus, daß 0 < a <= b <= c gilt, so ist zuerst zu prüfen, ob mit diesem Tripel ein Dreieck möglich ist !

```
> a:=2: b:=3: c:=10: if a+b <= c then 'unzulaessig'elif
  pythagoras=0 then 'rechtwinklig' elif pythagoras<0 then
  'stumpfwinklig'else 'spitzwinklig' fi;
```

$$unzulaessig$$

Beispiel 2: Fallunterscheidung bei 4 Funktionen

```
> restart:

> f := (x,n)->
       if   n=0 then 0
       elif n=1 then sin(x)
       elif n=2 then cos(x)
       elif n=3 then tan(x)
       else ERROR(‘unzulaessiges argument‘, n)
       fi;
```

$$f := \mathbf{proc}(x,\ n)\ \mathbf{option}\ operator,\ arrow;\ \mathbf{if}\ n = 0\ \mathbf{then}\ 0\ \mathbf{elif}\ n = 1\ \mathbf{then}\ \sin(x)\ \mathbf{elif}\ n = 2\ \mathbf{then}\ \cos(x)\ \mathbf{elif}\ n = 3\ \mathbf{then}\ \tan(x)\ \mathbf{else}\ \mathrm{ERROR}(‘unzulaessiges\ argument‘,\ n)\ \mathbf{fi}\ \mathbf{end}$$

```
> f(t,2); f(phi,0); f(Pi/4,3);
```

$$\cos(t)$$
$$0$$
$$1$$

```
> f(y,7);

Error, (in f) unzulässiges argument, 7
```

Man erkennt: elif-Klauseln werden nicht mit einem fi abgeschlossen.

5.3.2 Die for-while-Anweisung für Zyklen

Die for-while-Anweisung existiert in 2 syntaktischen Formen:

for variable from start by change to finish while condition do
statement-sequence
od;

sowie

for variable in expression while condition do
statement-sequence
od;

Bedeutung der Klauseln:

- for variable : definiert die Laufvariable, die in jedem Durchlauf um einen festen Wert verändert wird
- from start : definiert den Anfangswert der Laufvariablen (Default: 1)
- by change : definiert die Schrittweite der Laufvariablen (Default: 1)
- to finish : definiert den Endwert der Laufvariablen (Default: infinity)
- while condition : definiert eine Bedingung, die vor jedem Durchlauf überprüft wird; bei Nicht-Erfülltheit wird der Zyklus sofort beendet
- in expression : enthält einen Ausdruck (bzw. Folge, Liste, Menge, Polynom,...), der gliedweise durchlaufen wird (2. Form)
- do Anweisungsfolge od : definieren den Schleifenkörper, d.h. die gesamte Anweisungsfolge wird - falls keine weiteren Klauseln angegeben worden sind - unendlich oft abgearbeitet (Endlosschleife). Die Anweisungsfolge kann leer sein.

Bemerkungen:

Mit Ausnahme von do und od können einige oder alle der Klauseln weggelassen werden. Die Reihenfolge der from-, by-, to-Klauseln ist beliebig. Zyklen können geschachtelt werden.

Die for-from- Schleife

```
> restart:
> for i from 2 to 9 by 2 do i^2 od;
```

$$4$$
$$16$$
$$36$$
$$64$$

Die Reihenfolge der Klauseln kann - bis auf for - beliebig sein. Der Zyklus ist „abweisend“, d.h. erst wird getestet, danach die Anweisungsfolge ausgeführt.

```
> for i by 2 to 9 from 2 do i^2 od;
```

$$4$$
$$16$$
$$36$$
$$64$$

```
> for i to 5 do i^2 od;
```

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

```
> for i from 0 to -5 do i^2 od; # Leerer Laufbereich !
> for i from 0 to -5 by -2 do i^3 od;
```

$$0$$
$$-8$$
$$-64$$

Die Laufvariable muß vom Typ numeric sein (d.h. float, integer, fraction). Sie darf auch während der Anweisungsfolge zusätzlich verändert werden. Ihren Endwert erhält sie beim letztmaligen Beendigungstest; d.h. den ersten nicht zulässigen Wert.

```
> for i from 0 to -5 by -2 do i^3 od: i;
```

$$-6$$

```
> for i from 0 to 12 do i^3; i:=1.5*i; od;
```

$$0$$
$$i := 0$$
$$1$$
$$i := 1.5$$
$$15.625$$
$$i := 3.75$$
$$107.171875$$
$$i := 7.125$$
$$536.3769531$$
$$i := 12.1875$$

```
> i;
```

$$13.1875$$

```
> for i from 0 to 50 do lprint(i,i^2,i^3,i^4); i:=2*i+1; od:
0   0   0   0
2   4   8   16
6   36   216   1296
14   196   2744   38416
30   900   27000   810000
```

Endet die Zyklusanweisung mit Doppelpunkt, so findet keinerlei Ausgabe statt (auch nicht bei Semikolon hinter den Anweisungen!). `lprint` dient z.B. dann der Ausgabe innerhalb einer Zeile.

```
> for i from 0 to 50 do lprint(i,‘ ‘,i^2,‘ ‘,i^3,‘ ‘,i^4);
  i:=2*i+1; od:
0       0       0       0
2       4       8       16
6       36       216       1296
14       196       2744       38416
30       900       27000       810000
```

Beispiel 1: Binomialkoeffizient (n über k)

```
> restart:
> n:=6; k:=4;
```

$$n := 6$$
$$k := 4$$

```
> if k>n-k then k:=n-k fi;
  bk := 1;
  for i from 1 to k do
    bk := bk * (n - i + 1) / i
  od;
```

$$k := 2$$
$$bk := 1$$
$$bk := 6$$
$$bk := 15$$

```
> if k>n-k then k:=n-k fi:
  bk := 1:
  for i from 1 to k do
    bk := bk * (n - i + 1) / i
  od:
  lprint(‘Binomialkoeffizient‘,n,‘über‘,k,‘ =‘,bk);
```

```
Binomialkoeffizient   6   über   2    =   15
```

```
> n:=49; k:=6;
```

$$n := 49$$
$$k := 6$$

```
> if k>n-k then k:=n-k fi:
  bk := 1:
  for i from 1 to k do
    bk := bk * (n - i + 1) / i
  od:
  lprint(‘Binomialkoeffizient‘,n,‘über‘,k,‘ =‘,bk);
```

```
Binomialkoeffizient   49   über   6    =   13983816
```

```
> n:='n'; k:=6;
```

$$n := n$$
$$k := 6$$

```
> #  if k>n-k then k:=n-k fi:
  bk := 1:
  for i from 1 to k do
    bk := bk * (n - i + 1) / i
  od:
  lprint(‘Binomialkoeffizient‘,n,‘über‘,k,‘ =‘,bk);
```

```
Binomialkoeffizient   n   ber   6    =
1/720*n*(n-1)*(n-2)*(n-3)*(n-4)*(n-5)
```

Beispiel 2: Tabelle der Binomialkoeffizienten (n über k)

```
> restart:
> nmax := 4;
```

$$nmax := 4$$

```
> for n from 0 to nmax do
    for k from 0 to n do
      if k > n - k then kk := n - k else kk := k fi;
      bk := 1;
      for i from 1 to k do
        bk := bk * (n - i + 1) / i
      od:    # i-Zyklus
      lprint(‘Binomialkoeffizient‘,n,‘ber‘,k,‘ =‘,bk);
    od:    # k-Zyklus
    lprint(‘   ‘);
  od:    # n-Zyklus
```

```
Binomialkoeffizient   0   über   0    =   1

Binomialkoeffizient   1   über   0    =   1
Binomialkoeffizient   1   über   1    =   1

Binomialkoeffizient   2   über   0    =   1
Binomialkoeffizient   2   über   1    =   2
Binomialkoeffizient   2   über   2    =   1

Binomialkoeffizient   3   über   0    =   1
Binomialkoeffizient   3   über   1    =   3
Binomialkoeffizient   3   über   2    =   3
Binomialkoeffizient   3   über   3    =   1
```

```
Binomialkoeffizient   4   über   0    =   1
Binomialkoeffizient   4   über   1    =   4
Binomialkoeffizient   4   über   2    =   6
Binomialkoeffizient   4   über   3    =   4
Binomialkoeffizient   4   über   4    =   1
```

Abgekürzte for-from-Schleifen:

(1) Bei Fehlen des from bzw. step werden die Default-Werte 1 gesetzt.

(2) Bei Fehlen des for wird die Anweisung entsprechend oft iteriert.

```
> for i to 5 do i^2 od;
```

$$\mathbf{1}$$
$$\mathbf{4}$$
$$\mathbf{9}$$
$$\mathbf{16}$$
$$\mathbf{25}$$

```
> zahl := rand(1..6): to 8 do zahl(); od;
```

$$\mathbf{2}$$
$$\mathbf{6}$$
$$\mathbf{3}$$
$$\mathbf{5}$$
$$\mathbf{1}$$
$$\mathbf{5}$$
$$\mathbf{4}$$
$$\mathbf{2}$$

Die for-in- Schleife

Die Schleife durchläuft alle Komponenten eines Ojektes L. Dieses kann
- eine Liste oder Menge,
- ein Ausdruck (eine Summe, ein Produkt o.a.)

sein. Die Anweisungen des Schleifenkörpers werden auf alle Elemente dieses Objektes angewendet.

Beispiel 1: Erzeugen rationaler Ausdrücke

```
> restart:
> liste1 := [1,4,9,25,36,49,64,100,33]:

> for zahl in liste1 do
    a := (x^zahl-1)/(x-1); b:= simplify(a);
    lprint(‘Quotient = ‘,a,‘ = ‘,b); lprint();
  od:
```

```
Quotient =    1    =    1


Quotient =    (x^4-1)/(x-1)    =    x^3+x^2+x+1


Quotient =    (x^9-1)/(x-1)    =    x^8+x^7+x^6+x^5+x^4+x^3+x^2+x+1


Quotient =    (x^25-1)/(x-1)    =
x^3+x^2+x^9+x^8+x^6+x^7+x^5+x^16+x^15+x^13+x^14+x^12+x^11+x^10+x^24+x^
22+x^23+x^21+x^20+x^19+x^18+x^17+x+1+x^4


Quotient =    (x^36-1)/(x-1)    =
x^35+x^33+x^34+x^32+x^31+x^30+x^29+x^28+x^27+x^26+x^3+x^2+x^9+x^8+x^6+
x^7+x^5+x^16+x^15+x^13+x^14+x^12+x^11+x^10+x^25+x^24+x^22+x^23+x^21+x^
20+x^19+x^18+x^17+x+1+x^4


Quotient =    (x^49-1)/(x-1)    =
x^35+x^33+x^34+x^32+x^31+x^30+x^29+x^28+x^27+x^26+x^48+x^46+x^47+x^45+
x^44+x^43+x^42+x^41+x^40+x^39+x^38+x^37+x^3+x^2+x^9+x^8+x^6+x^7+x^5+x^
16+x^15+x^13+x^14+x^12+x^11+x^10+x^25+x^24+x^22+x^23+x^21+x^20+x^19+x^
18+x^17+x^36+x+1+x^4


Quotient =    (x^64-1)/(x-1)    =
x^35+x^33+x^34+x^32+x^31+x^30+x^29+x^28+x^27+x^26+x^49+x^48+x^46+x^47+
x^45+x^44+x^43+x^42+x^41+x^40+x^39+x^38+x^37+x^63+x^61+x^62+x^60+x^59+
x^58+x^57+x^56+x^55+x^54+x^53+x^52+x^51+x^50+x^3+x^2+x^9+x^8+x^6+x^7+x
^5+x^16+x^15+x^13+x^14+x^12+x^11+x^10+x^25+x^24+x^22+x^23+x^21+x^20+x^
19+x^18+x^17+x^36+x+1+x^4


Quotient =    (x^100-1)/(x-1)    =
x^35+x^33+x^34+x^32+x^31+x^30+x^29+x^28+x^27+x^26+x^49+x^48+x^46+x^47+
x^45+x^44+x^43+x^42+x^41+x^40+x^39+x^38+x^37+x^64+x^63+x^61+x^62+x^60+
x^59+x^58+x^57+x^56+x^55+x^54+x^53+x^52+x^51+x^50+x^81+x^80+x^78+x^79+
x^77+x^76+x^75+x^74+x^73+x^72+x^71+x^70+x^67+x^66+x^65+x^69+x^68+x^3+x
^2+x^9+x^8+x^6+x^7+x^5+x^16+x^15+x^13+x^14+x^12+x^11+x^10+x^25+x^24+x^
22+x^23+x^21+x^20+x^19+x^18+x^17+x^36+x+x^99+x^97+x^98+x^96+x^95+x^94+
x^93+x^92+x^91+x^90+x^89+x^88+x^87+x^86+x^85+x^84+x^83+x^82+1+x^4


Quotient =    (x^33-1)/(x-1)    =
x^32+x^31+x^30+x^29+x^28+x^27+x^26+x^3+x^2+x^9+x^8+x^6+x^7+x^5+x^16+x^
15+x^13+x^14+x^12+x^11+x^10+x^25+x^24+x^22+x^23+x^21+x^20+x^19+x^18+x^
17+x+1+x^4
```

Beispiel 2: Auflisten der Operanden eines Ausdruckes

```
> for term in (x^8-1)/(x+1) do term od;
```

$$x^8 - 1$$
$$1$$
$$\frac{}{x+1}$$

```
> for term in simplify((x^8-1)/(x+1)) do term od;
```

$$x^7$$
$$-x^6$$
$$x^5$$
$$-x^4$$
$$x^3$$
$$-x^2$$
$$x$$
$$-1$$

```
> f1 :=diff(x^x,x); for term in f1 do term od;
```

$$f1 := x^x \left(\ln(x) + 1\right)$$
$$x^x$$
$$\ln(x) + 1$$

```
> f3 :=diff(x^(x^x),x); for term in f3 do term od;
```

$$f3 := x^{(x^x)} \left(x^x \left(\ln(x) + 1\right) \ln(x) + \frac{x^x}{x}\right)$$
$$x^{(x^x)}$$
$$x^x \left(\ln(x) + 1\right) \ln(x) + \frac{x^x}{x}$$

Die while-Schleife

Hierbei fehlen sämtliche Klauseln der for-from-Schleife mit Ausnahme der while-Klausel:

while Bedingung do Anweisungen od;

Die Bedingung muß zu true, false oder FAIL auswertbar sein. Die Schleife wirkt abweisend, d.h. zuerst wird die Bedingung geprüft und danach werden die Anweisungen abgearbeitet.

Beispiel 1: Bestimmung der Gleitpunkt-Genauigkeit

```
> restart: Digits;
```

$$10$$

```
> x := 1.0:
  while x+1.0 <> 1.0 do
    x := 0.5*x;
  od:
  lprint('Rechengenauigkeit = ',x);
```

```
Rechengenauigkeit =    .4656612883e-9
```

```
> Digits := 100: x := 1.0:
  while x+1.0 <> 1.0 do
    x := 0.5*x;
  od:
  lprint(‘Rechengenauigkeit = ‘,x);
```

```
Rechengenauigkeit =
.45719495651290999288631341932213638378076399792911946448605033729105
3
05436269527827017920874406881 85e-99
```

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

Beispiel 2: Effiziente Berechnung der Potenz x^n

```
> restart: p:=1: w:=2048: i:=10:
  while  i>0 do
    if i mod 2 = 1 then p:=p*w; i:=i-1; fi;
    i:= i/2;
    w := w^2;
  od:
  p;
```

$$129807421463370690713262408230 5024$$

```
> p:=1: w:=a+2: i:=4:
  while  i>0 do
    if i mod 2 = 1 then p:=p*w; i:=i-1; fi;
    i:= i/2;
    w := w^2;
  od:
  p; expand(p);
```

$$(a+2)^4$$

$$a^4+8\,a^3+24\,a^2+32\,a+16$$

Die for-while-Schleife

Hierbei werden die Klauseln der for-from-Schleife mit der while-Klausel kombiniert:

for Name from Ausdruck1 to Ausdruck2 by Ausdruck3
while Bedingung do
** Anweisungen**
od;

Die Bedingung muß zu true, false oder FAIL auswertbar sein. Die Schleife wirkt abweisend, d.h. zuerst wird

– die Zulässigkeit der Laufvariablen und

– die Bedingung geprüft .

Danach werden die Anweisungen abgearbeitet. Wird der Laufbereich der for-from-Schleife überschritten oder ist die while-Bedingung nicht mehr erfüllt, so wird die Schleife verlassen.

Beispiel 1: Nullstellenbestimmung per Bisektion

```
> restart:

> lprint(‘Bisektionsverfahren  fuer ‘);
  lprint(‘f(x) = sin(x) - exp(-x)‘);

  a := 0: b := 2: epsi := 0.0001:
  lprint(‘Genauigkeit = ‘,epsi);
  y := sin(a) - exp(-a):

  if y >= 0.0 then
    x := a: a:=b: b:=x:
  fi:     # Nun ist f(a)<0 und f(b)>0

  for i to 20 while abs(a - b) > epsi * abs(a) do
    x := (a +b) / 2.0;
    y := sin(x) - exp(-x);
    if y < 0.0 then a := x else b := x fi;
  od:     #  Ende  des Zyklus

  lprint(‘Nullstelle  = ‘,x);
```

```
Bisektionsverfahren  fuer

f(x) = sin(x) - exp(-x)

Genauigkeit =    .1e-3

Nullstelle  =    .5885314944
```

Bemerkung: Wird die Schleife wegen nicht-erfüllter while-Bedingung verlassen, so wird die Laufvariable nicht noch einmal verändert - anders als bei Beendigung der einfachen from-for-Schleife !

Beispiel 1: Bestimmung desjenigen n mit n! >= bound

```
> bound := 10^50+1:
  for n while n! < bound do od;
  lprint(`n = `,n);
```

```
n =    42
```

```
> lprint(n-1,` => `, (n-1)!);
  lprint(`bound => `, bound);
  lprint(n,` => `, n!);
```

```
41    =>    33452526613163807108170062053440751665152000000000
bound =>    100000000000000000000000000000000000000000000000001
42    =>    1405006117752879898543142606244511569936384000000000
```

Beispiel 2: Effiziente Berechnung der Potenz x^n (s.o)
mit Beschränkung der Durchlaufzahl

```
> restart: p:=1: w:=2: i:=20:
  to 100 while  i>0 do
    if i mod 2 = 1 then p:=p*w; i:=i-1; fi;
    i:= i/2;
    w := w^2;
  od:
  p;
```

1048576

5.3.3 Die Sprungbefehle break und next

Die Kommandos break und next dienen der vorzeitigen Beendigung von Schleifen; sie treten innerhalb von Schleifen auf.

- Erreichen eines break in einer Schleife beendet diese (innerste Schleife) sofort und setzt die Abarbeitung mit der nachfolgenden Anweisung fort.
- Erreichen eines next in einer Schleife bedeutet sofortigen Übergang zum nächsten Durchlauf; d.h. es wird sofort an den Schleifenkopf zurckgesprungen und die Abbruchbedingung mit dem nächsten Laufwert bzw. dem nächsten Listenelement fortgesetzt.

```
> restart:
> L := [4,6,8,15,21,23,28,33,37,41,44,49,53]:

> for n in L do
    if isprime(n) then break fi;
    print(n);
  od;
```

$$4$$
$$6$$
$$8$$
$$15$$
$$21$$

```
> for n in L do
    if isprime(n) then next fi;
    print(n);
  od;
```

$$4$$
$$6$$
$$8$$
$$15$$
$$21$$
$$28$$
$$33$$
$$44$$
$$49$$

5.4 Prozeduren in Maple

Das Kommando proc definiert Prozeduren, also Verfahren, die beliebig viele Eingangsparameter (entries) in - beliebig viele - Ausgangsparameter (exits) transformieren können. Behandelt werden insbesondere
- Prozedurdefinition und -aufruf,
- die Begriffe der formalen und aktuellen Parameter,
- die lokalen und globalen Variablen sowie
- rekursive Prozeduren und Erinnerungstabellen.

5.4.1 Prozedurdefinition

Prozeduren können für beliebige (ein- und mehrzeilige) Verfahren definiert werden.

Syntax (vorläufig)

<Name> := proc(<Parameterliste>) < Anweisungsteil> end;

Erläuterung

Die Parameterliste ist eine Folge von Namen, getrennt durch Kommata. Der Anweisungsteil besteht aus einer Aufeinanderfolge von Anweisungen, getrennt durch Semikolon.

Beispiel 1: Funktion einer Veränderlichen

```
> f := proc(x) x^3-4*x+1-sin(x) end;
```

$$f := \mathbf{proc}(x)\, x^3 - 4 \times x + 1 - \sin(x)\, \mathbf{end}$$

Aufruf der Prozedur

Er erfolgt wie bei Pfeilfunktionen, d.h. Ersetzung der formalen Parameter (hier: x) durch die aktuellen Parameter. Es erfolgt stets ein Wertaufruf (call by value).

```
> summe := f(1.5) + f(1.9) + f(3.4);
```

$$summe := 23.64974603$$

```
> int(f(z),z);
```

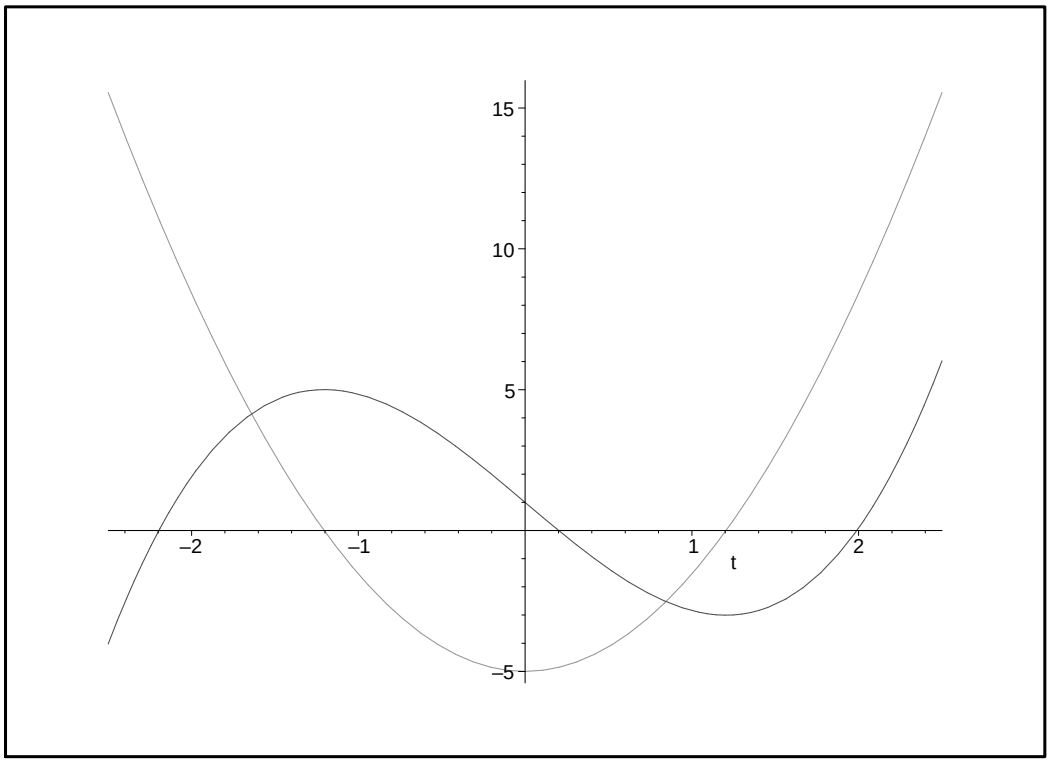
$$\frac{1}{4}\, z^4 - 2\, z^2 + z + \cos(z)$$

Definierte Funktionen (z.B. f) dürfen - wie auch eingebaute Funktionen - in den Deklarationen weiterer Prozedurdefinitionen benutzt werden.

```
> f_1 := proc(t) diff(f(t),t) end;
```

$$f\_1 := \mathbf{proc}(t)\,\mathrm{diff}(\mathrm{f}(t),\,t)\,\mathbf{end}$$

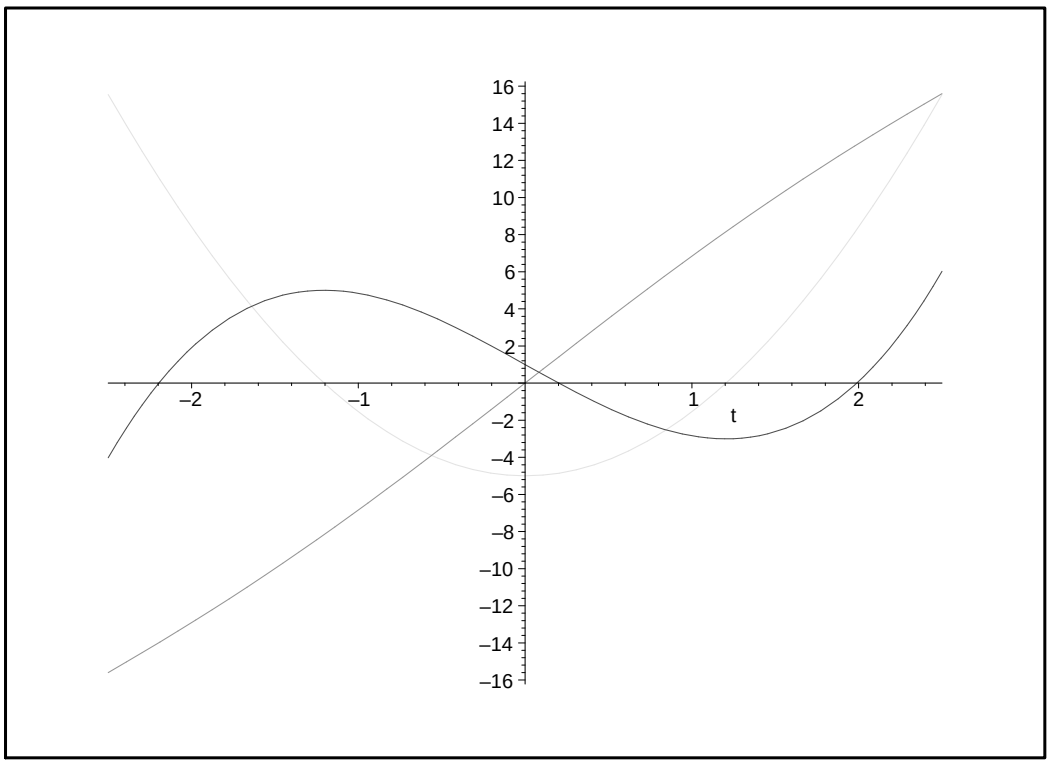
```
> plot([f(t), f_1(t)], t=-2.5 .. 2.5);
```




```
> f_2 := proc(t) diff(f_1(t),t) end;
```

$$f\_2 := \mathbf{proc}(t)\,\mathrm{diff}(\mathrm{f\_1}(t),\,t)\,\mathbf{end}$$

```
> plot([f(t), f_2(t), f_1(t)], t=-2.5 .. 2.5, thickness=2);
```



Symbolische Argumente

Prozeduren dürfen auch auf symbolische Argumente und Ausdrücke angewendet werden. Dabei gilt es jedoch aufzupassen: Die Prozeduren f_1(t) und f_2(t) beinhalten die Differentiation nach dem Argument t . Ist der aktuelle Parameter ein echter Ausdruck (z.B. t+1), so kann `diff(f(t+1),t+1)` nicht ausgeführt werden !

```
> f(alpha+2); simplify(%);
```

$$(\alpha+2)^3-4\,\alpha-7-\sin(\alpha+2)$$
$$\alpha^3+6\,\alpha^2+8\,\alpha+1-\sin(\alpha+2)$$

```
> f(f_1(t)+f_2(t));
```

$$(3\,t^2-4-\cos(t)+6\,t+\sin(t))^3-12\,t^2+17+4\cos(t)-24\,t-4\sin(t)$$
$$-\sin(3\,t^2-4-\cos(t)+6\,t+\sin(t))$$

```
> f_1(t+1);

Error, (in f_1) wrong number (or type) of parameters in function diff
```

Mehrere Anweisungen im Prozedurkörper

Zur Vermeidung von Mehrfachberechnungen können „Hilfsvariablen“ (hier: u) benutzt werden. Falls nicht anders deklariert, sind diese Variablen stets lokal, d.h. ihr Gültigkeitsbereich ist das Innere der Prozedurdefinition ! Der letzte in der Prozedurdefinition berechnete Ausdruck liefert den Rückgabewert der Prozedur.

Beispiel 2: Funktion mehrerer Veränderlicher

```
> g := proc(x,y,z)              #  3 Formalparameter
       u := f(x+y-z)-f(y);      #  2 Anweisungen
       u^2-3*u+1                #  Rueckgabewert
  end;
```

```
Warning, `u` is implicitly declared local
```

$$g := \mathbf{proc}(x,\, y,\, z)\ \mathbf{local}\ u;\ u := \mathrm{f}(x+y-z) - \mathrm{f}(y)\,;\ u^2 - 3\times u + 1\ \mathbf{end}$$

```
> g(2,3,4); u; g(x,0,0);
```

$$(-18 - \sin(1) + \sin(3))^2 + 55 + 3\sin(1) - 3\sin(3)$$

$$u$$

$$(x^3 - 4\,x - \sin(x))^2 - 3\,x^3 + 12\,x + 3\sin(x) + 1$$

```
> g(s,t,u);
```

$$((s+t-u)^3 - 4\,s + 4\,u - \sin(s+t-u) - t^3 + \sin(t))^2 - 3\,(s+t-u)^3 + 12\,s - 12\,u + 3\sin(s+t-u) + 3\,t^3 - 3\sin(t) + 1$$

Deklaration lokaler Variablen

Lokale (Hilfs-)Variablen sollten explizit als lokal gekennzeichnet werden. Dies bewirkt die Deklaration

local <Variablen>;

unmittelbar vor den Anweisungen.

```
> g := proc(x,y,z)          #  siehe oben
    local u;                #  u ist lokal
    u := f(x+y-z)-f(y);
    u^2-3*u+1               #  Rueckgabewert
  end;
```

$$g := \mathbf{proc}(x,\, y,\, z)\ \mathbf{local}\ u;\ u := \mathrm{f}(x+y-z) - \mathrm{f}(y)\,;\ u^2 - 3\times u + 1\ \mathbf{end}$$

Ausgabe von Prozedurinhalten

(1) Steht hinter dem abschliessenden end ein Semikolon, so wird der komplette Prozedurtext angezeigt (siehe obige Beispiele), bei einem Doppelpunkt wird diese Anzeige unterdrückt. Syntaxfehler werden jedoch immer gemeldet.

(2) Werte und Anweisungen während der Prozedurabarbeitung werden normalerweise nicht angezeigt - ein Semikolon bewirkt hier keine Ausgabe.

Beispiel 2: s.oben

```
> g := proc(x,y,z)
    local u;
    u := f(x+y-z)-f(y);
    u^2-3u+1        #  Syntaxfehler eingebaut!
  end:
```

```
missing operator or ';'
```

```
> g := proc(x,y,z)
    local u;
    u := f(x+y-z)-f(y);
    u^2-3*u+1
  end:        #  Ausgabe unterdrueckt!
```

Abruf des Prozedurinhaltes

```
> f(x); f; # Kein Prozedurinhalt wird angezeigt!
```

$$x^3 - 4\,x + 1 - \sin(x)$$

$$f$$

```
> interface( verboseproc=2 );
> print(int); # Eingebaute Prozedur!
```

proc() ... **end**

```
> print(f);
```

proc(x) $x^3 - 4 \times x + 1 - \sin(x)$ **end**

```
> print(g);
```

proc(x, y, z) **local** u; $u := \mathrm{f}(x + y - z) - \mathrm{f}(y)$; $u^2 - 3 \times u + 1$ **end**

```
> interface( verboseproc = 1 ); # Rueckstellen !
```

5.4.2 Verfolgung der Prozedurausführung (tracing)

Ausgabeanweisungen

Bildschirmausgaben von Werten während der Prozedurabarbeitung sind mglich mit den Anweisungen

- print : Pretty-Print,
- lprint : Linear-Print,
- printf : Print in File .

```
> h := proc(x,y,z)
    u := f(x+y-z)/f(y);
    print(‘u ‘ = u);    #  lprint(‘ u =‘, u);
    u^2-3*u+1
  end:
```

```
Warning, ‘u‘ is implicitly declared local
```

```
> h(sigma,3,6);
```

$$u = \frac{(\sigma-3)^3-4\,\sigma+13-\sin(\sigma-3)}{16-\sin(3)}$$

$$\frac{((\sigma-3)^3-4\,\sigma+13-\sin(\sigma-3))^2}{(16-\sin(3))^2}-3\,\frac{(\sigma-3)^3-4\,\sigma+13-\sin(\sigma-3)}{16-\sin(3)}+1$$

Beispiel 3: Größter gemeinsamer Teiler GCD

Euklidischer Algorithmus (ca.300 v.d.Z.) in iterativer Form:

```
> restart:
```

```
> GCD := proc(a,b)
    c := a; d := b;          # Anfangswerte
    while d<>0 do
      r := irem(c,d);        # Integer Remainder
      lprint(‘Rest =‘,r);    # Ausgabe aller Reste
      c := d; d := r;
    od;
    c                        # Rueckgabewert
  end:
```

```
Warning, ‘c‘ is implicitly declared local

Warning, ‘d‘ is implicitly declared local

Warning, ‘r‘ is implicitly declared local
```

```
> ggt := GCD(12003,9906);
```

```
Rest =   2097

Rest =   1518

Rest =   579

Rest =   360

Rest =   219

Rest =   141
```

```
Rest =   78

Rest =   63

Rest =   15

Rest =   3

Rest =   0
```

$$ggt := 3$$

```
>  ggt := GCD(100!,2^100);

Rest =   79228162514264337593543950336O

Rest =   47536897508558602556126370201 6

Rest =   31691265005705735037417580134 4

Rest =   15845632502852867518708790067 2

Rest =   0
```

$$ggt := 158456325028528675187087900672$$

Erhöhung des Printlevels

Die (globale) Variable printlevel besitzt den Defaultwert 1 . Erhöhung ihres Wertes ergibt die Anzeige

- aller Prozedur-Entries,
- aller Prozedur-Exits,
- aller internen Zuweisungen.

```
>  GCD := proc(a,b)
     c := a; d := b;        # Anfangswerte
     while d<>0 do
       r := irem(c,d);      # Integer Remainder
       c := d; d := r;
     od;
     c                      # Rueckgabewert
   end:

Warning, ‘c‘ is implicitly declared local

Warning, ‘d‘ is implicitly declared local

Warning, ‘r‘ is implicitly declared local

>  printlevel := 100;
```

$$printlevel := 100$$

```
> GCD(15,10);
```

```
{--> enter GCD, args = 15, 10
```

$$c := 15$$
$$d := 10$$
$$r := 5$$
$$c := 10$$
$$d := 5$$
$$r := 0$$
$$c := 5$$
$$d := 0$$
$$5$$

```
<-- exit GCD (now at top level) = 5}
```

$$5$$

```
> ggt := GCD(12003,9906): # Nur entries und exits!
```

```
{--> enter GCD, args = 12003, 9906

<-- exit GCD (now at top level) = 3}
> printlevel := 1; # Ruecksetzen nicht vergessen!
```

$$printlevel := 1$$

Euklidischer Algorithmus (ca.300 v.d.Z.) in rekursiver Form

```
> GCD := proc(a,b)
    if b = 0 then a else GCD(b,irem(a,b)) fi
  end:

> ggt_rekursiv := GCD(15,10);
```

$$ggt\_rekursiv := 5$$

```
> printlevel:=100: ggt_rekursiv:=GCD(15,10); printlevel:=1:

{--> enter GCD, args = 15, 10

{--> enter GCD, args = 10, 5

{--> enter GCD, args = 5, 0
```

$$5$$

```
<-- exit GCD (now in GCD) = 5}
```

$$5$$

```
<-- exit GCD (now in GCD) = 5}
```

$$5$$

```
<-- exit GCD (now at top level) = 5}
```

$$ggt\_rekursiv := 5$$

Anwendung des trace-Kommandos

Die trace-Funktion ist ein Debugging Tool zur Verfolgung der Ausführung der Procedure(n) f, g, h usw. Während der Abarbeitung werden

- die Entry-Punkte,
- die Resultate der ausgeführten Anweisungen und
- die Exit-Punkte

der betreffenden Procedure(n) gedruckt. An den Entry-Punkten werden die aktuellen Parameter ausgegeben, an den Exit-Punkten die zurückgegebene Funktion. Das Kommando untrace hebt die Trace-Funktion wieder auf. Debug und undebug sind Synonyme für trace und untrace.

Syntax :

trace(f), trace(f,g,h,...), debug(f), debug(f,g,h,...),
untrace(f), untrace(f,g,h,...), undebug(f), undebug(f,g,h,...)

Parameter :

f, g, h, ... - Name(n) von Procedure(n), die getraced werden sollen

```
> GCD := proc(a,b)
    c := a; d := b;          # Anfangswerte
    while d<>0 do
      r := irem(c,d);        # Integer Remainder
      c := d; d := r;
    od;
    c                        # Rueckgabewert
  end:
```

```
Warning, `c` is implicitly declared local

Warning, `d` is implicitly declared local

Warning, `r` is implicitly declared local
```

```
> trace(GCD):
> GCD(6,9);
```

```
{--> enter GCD, args = 6, 9
```

$$c := 6$$
$$d := 9$$
$$r := 6$$
$$c := 9$$
$$d := 6$$
$$r := 3$$

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

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

$$3$$

```
> untrace(GCD):
```

5.4.3 Prozedurparameter

Parameterübergabe

(1) Der Aufruf der Prozedur <Name> erfolgt durch

<Name> (<Parameterliste>)

mit der Folge <Parameterliste> der aktuellen Parameter.

(2) Ist ein Aktuellparameter selbst eine Folge, so wird die entstehende Folge von Folgen in einer einziger Folge zusammengefaßt (flattened).

(3) Dann erfolgt die Zuordnung der aktuellen zu den formalen Parametern - nur nach ihrer Reihenfolge - von links nach rechts.

(4) Stehen zu wenig aktuelle Parameter in der Liste, so erfolgt eine Fehlermeldung, falls ein fehlender Parameter während der Abarbeitung gebraucht wird. Überflssige Parameter hingegen werden ignoriert.

(5) Maple ersetzt im Prozedurkörper alle Auftreten der formalen Parameter durch die entsprechenden aktuellen Ausdrücke und führt danach die Anweisungen des Prozedurkörpers aus.

Beispiel 1: Zuordnungsreihenfolge und Parameterzahl

```
> f := proc(x,y,z) if x>y then x else z fi end:

> f(1,2,3);
    # Zuordnung  :  x => 1 , y => 2 , z => 3
    # Ausfuehrung:  if 1>2 then 1 else 3 fi => 3
```

$$3$$

```
> f(1,2); # Zu wenig Parameter, z = 3 fehlt !
Error, (in f) f uses a 3rd argument, z, which is missing
```

```
> f(2,1); # Then-Zweig ausfuehrbar!
```
$$\mathbf{2}$$
```
> f(1,2,3,4,5,6); # Zu viele Parameter - macht nix !
```
$$\mathbf{3}$$
```
> argumente := 3,5,7;  #  Argumentfolge
  f(argumente);        #  Aktuelle Parameter: 3,5,7
```
$$\mathit{argumente} := \mathbf{3},\ \mathbf{5},\ \mathbf{7}$$
$$\mathbf{7}$$
```
> paar1 := 3,5:
  paar2 := 7,9:    # Argumentfolge: paar1,paar2
  f(paar1,paar2);  # Aktuelle Parameter: 3,5,7,9
```
$$\mathbf{7}$$

Beispiel 2: Substitution formaler durch aktuelle Parameter
```
> phi := proc(t) t^3+cos(t) end;
```
$$\phi := \mathbf{proc}(t)\, t^3 + \cos(t)\, \mathbf{end}$$
```
> phi(x); phi(2*x+Pi);
```
$$x^3 + \cos(x)$$
$$(2\,x + \pi)^3 - \cos(2\,x)$$
```
> phi_2 := proc(t) diff(phi(t),t,t)-int(phi(t),t) end:
> phi_2(x); # Name als aktueller Parameter
```
$$6\,x - \cos(x) - \frac{1}{4}\,x^4 - \sin(x)$$
```
> phi_2(2*x+Pi); # Ausdruck als aktueller Parameter

Error, (in phi_2) wrong number (or type) of parameters in function
diff
```

Grund für die Fehlfunktion von phi_2 : Eine formale Substitution liefert `diff(phi(2*x+Pi),2*x+Pi,2*x+Pi)`. Lösung : Differentiationsvariable und Integrationsvariable als 2. Parameter definieren!

```
> phi_besser := proc(arg,t)
      diff(phi(arg),t,t)-int(phi(arg),t) end;
```
$$\mathit{phi\_besser} := \mathbf{proc}(\mathit{arg},\ t)\, \mathrm{diff}(\phi(\mathit{arg}),\ t,\ t) - \mathrm{int}(\phi(\mathit{arg}),\ t)\, \mathbf{end}$$
```
> phi_besser(x,x);
```
$$6\,x - \cos(x) - \frac{1}{4}\,x^4 - \sin(x)$$
```
> phi_besser(2*x+Pi,x);
```
$$48\,x + 24\,\pi + 4\cos(2\,x) - \frac{1}{8}\,(2\,x + \pi)^4 + \frac{1}{2}\sin(2\,x)$$

Deklaration der Formalparameter

Für jeden Formalparameter ist eine Typdeklaration möglich. Die Syntax lautet:

<Parameter> :: <Typ>

Beim Prozeduraufruf überprüft Maple, ob der aktuelle Parameter vom angegebenen Typ ist und bricht ggf. mit einer Fehlermeldung ab (Typüberprüfung). Wichtige Typarten sind z.B. numeric, boolean, complex, list, procedure, negint (siehe ?type).

Beispiel 1: Zahlenvergleich (s.oben)

```
> f := proc(x,y,z)
    if x>y then x else z  fi
  end:

> f(1,2,rho);
```

$$\rho$$

```
> f(rho,2,z); f(1,Pi,z);

Error, (in f) cannot evaluate boolean

Error, (in f) cannot evaluate boolean

> f_besser := proc(x::numeric,y::numeric,z)
    if x>y then x else z  fi
  end:

> f_besser(rho,2,3); f_besser(1,Pi,3);

Error, f_besser expects its 1st argument, x, to be of type numeric,
but received rho

Error, f_besser expects its 2nd argument, y, to be of type numeric,
but received Pi
> f_besser(1,evalf(Pi),3); # Vorsicht im Umgang mit Pi !!!
```

$$3$$

Nutzung der type-Funktion

Zur internen Typunterscheidung kann die Funktion

type(<Ausdruck>, <Typangabe>)

genutzt werden.

Beispiel : Differentiation einfacher Funktionen

Ausdrücke werden intern als gerichtete azyklische Graphen (DAG) dargestellt: Die Wurzel enthält die Information über den Typ des Ausdruckes, der mittels type erfragt werden kann. Die Operanden des Ausdruckes bilden die Elemente einer Folge und können mittels op(Ausdruck) aufgelistet werden. Die Operanden selbst stellen erneut Teilbäume dar etc. Schließlich enthalten die Endknoten (Blätter) die Konstanten bzw. die Atome. Um Speicherplatz zu reduzieren, werden diese Konstanten bzw. Atome nur einmal gespeichert.

Die Summenregel der Differentiation erhält man, indem man die zu erstellende Funktion `deriv` auf sämtliche Elemente der Operandenliste einer Summe - ggf. rekursiv - anwendet.

```
> deriv := proc(a::algebraic, x::name)
    if type(a,numeric) then 0                  # Konstante
      elif type(a,name) then                   # Variable
        if a=x then 1 else 0 fi
      elif type(a,‘+‘) then map(deriv,a,x)  # Summenregel
      else ERROR(‘: kann‘,a,‘noch nicht differenzieren!‘)
    fi
  end;
```

$$
\begin{aligned}
&\mathit{deriv} := \textbf{proc}(a{::}\mathit{algebraic},\ x{::}\mathit{name})\\
&\quad \textbf{if } \mathrm{type}(a,\ \mathit{numeric}) \textbf{ then } 0\\
&\quad \textbf{elif } \mathrm{type}(a,\ \mathit{name}) \textbf{ then if } a = x \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi}\\
&\quad \textbf{elif } \mathrm{type}(a,\ `+`) \textbf{ then } \mathrm{map}(\mathit{deriv},\ a,\ x)\\
&\quad \textbf{else } \mathrm{ERROR}(`:\ \mathit{kann}`,\ a,\ `\mathit{noch\ nicht\ differenzieren}!`)\\
&\quad \textbf{fi}\\
&\textbf{end}
\end{aligned}
$$

```
> deriv(67,x); deriv(Pi,x); deriv(rho,rho);
```

$$0$$
$$0$$
$$1$$

```
> deriv(x+y-2.3+Pi,y); deriv(x+deriv(x+2,x)+3,x);
```

$$1$$
$$1$$

```
> deriv(3*x^2+x+1,x);
Error, (in deriv) : kann, 3*x^2, noch nicht differenzieren!
```

Leere Parameterliste

Prozeduren müssen keine formalen Parameter besitzen. Die Anzahl der aktuellen Parameter kann dann beliebig - also auch Null - sein.

5.4.4 Rückgabewerte von Prozeduren

Die Rückgabe des von der Prozedur gelieferten Wertes kann durch folgende 4 Mechanismen erfolgen:

(1) Standard-Return : Rückgabe des Wertes der letzten bearbeiteten Anweisung
(2) Explizites Return : Rückgabe durch eine RETURN-Anweisung
(3) Fehler-Return : Beendigung mit einer Fehlermeldung durch ERROR-Anweisung
(4) Namensparameter : Rückgabe zusätzlicher Werte über Namensparameter

Standard-Return

Wird nichts anderes veranlaßt, so gibt jede MAPLE-Prozedur den letzten berechneten Wert zurück. Häufig ist dies der letzte Ausdruck in der Anweisungsfolge. Bei if-Anweisungen und Zyklusanweisungen kann der letzte berechnte Wert allerdings auch im Inneren der Anweisungsfolge stehen.

Beispiel 1: Funktion einer und mehrerer Veränderlichen (s.o)

```
> f := proc(x) x^3-4*x+1-sin(x) end; # 1 Anweisung -> Rückgabewert
```

$$f := \mathbf{proc}(x)\, x^3 - 4 \times x + 1 - \sin(x)\, \mathbf{end}$$

```
> g := proc(x,y,z)          #  2 Anweisungen!
    local u;                #  u ist lokal
    u := f(x+y-z)-f(y);
    u^2-3*u+1               #  Rueckgabewert!
  end;
```

$$g := \mathbf{proc}(x, y, z)\, \mathbf{local}\, u;\ u := \mathrm{f}(x + y - z) - \mathrm{f}(y)\ ;\ u^2 - 3 \times u + 1\, \mathbf{end}$$

Beispiel 2: Größter gemeinsamer Teiler GCD (s.o.)

Euklidischer Algorithmus in iterativer Form; der zurückzugebende Wert c steht nicht in der letzten Anweisung. Durch Einfügen der Variablen c als letzte Anweisung erreicht man den gewünschten Rückgabewert:

```
> GCD := proc(a,b)
    local c,d,r;
    c := a; d := b;          # Anfangswerte
    while d<>0 do
       r := irem(c,d);       # Integer Remainder
    #  lprint(‘Rest =‘,r); # Ausgabe aller Reste
       c := d; d := r;
    od;
```

```
    c                              # Rueckgabewert
  end:
> ggt := GCD(12003,9906);
```

$$ggt := 3$$

```
> ggt := GCD(100!,2^100);
```

$$ggt := 158456325028528675187087900672$$

Beispiel 3 : Zahlenvergleich (s.oben)

```
> f := proc(x::numeric,y::numeric,z)
   if x>y then x else z  fi
  end;
```

$$f := \textbf{proc}(x::numeric,\ y::numeric,\ z)\ \textbf{if}\ y < x\ \textbf{then}\ x\ \textbf{else}\ z\ \textbf{fi}\ \textbf{end}$$

```
> z := f(-1,2,3.0); f(1,sqrt(z),sin(x+1));
```

$$z := 3.0$$
$$\sin(x+1)$$

Merke: Die if-Anweisung war hier die letzte ausgefhrte Anweisung!

Beispiel 4 : Differentiation einfacher Funktionen (s.oben)

```
> deriv := proc(a::algebraic, x::name)
    if type(a,numeric) then 0              # Konstante
    elif type(a,name) then                 # Variable
       if a=x then 1 else 0 fi
    elif type(a,'+') then map(deriv,a,x)   # Summenregel
    fi     #  else-Zweig  fehlt!
  end;
```

$$deriv := \textbf{proc}(a::algebraic,\ x::name)$$
$$\quad \textbf{if}\ \mathrm{type}(a,\ numeric)\ \textbf{then}\ 0$$
$$\quad \textbf{elif}\ \mathrm{type}(a,\ name)\ \textbf{then if}\ a = x\ \textbf{then}\ 1\ \textbf{else}\ 0\ \textbf{fi}$$
$$\quad \textbf{elif}\ \mathrm{type}(a,\ `+`)\ \textbf{then}\ \mathrm{map}(deriv,\ a,\ x)$$
$$\quad \textbf{fi}$$
$$\textbf{end}$$

```
> deriv(67,x), deriv(Pi,x), deriv(rho,rho);
```

$$0,\ 0,\ 1$$

```
> deriv(x+y-2.3+Pi,y);
```

$$1$$

```
> deriv(3*x^2+x+1,x);

Error, (in deriv) invalid types in sum
```

Aufgabe: Man überlege, welche Teilanweisung jeweils zuletzt ausgeführt wurde.

Häufig sind mehrere Werte zurückzugeben. Da Folgen, Listen und Mengen „gewöhnliche“ Datentypen sind, empfiehlt sich hierzu die Zusammenfassung der Ergebnisdaten zu einer Folge oder Liste (Mengen sind weniger geeignet), die auf „übliche“ Weise zurückgegeben werden kann. Allerdings muß nach erfolgter Wertzuweisung die aktuelle Folge bzw. Liste zerlegt werden.

Beispiel 5 : Division zweier Polynome

Man bestimme von 2 Polynomen p und q den Quotienten und den Rest bei Division.

```
> divpol := proc(p::polynom,q::polynom,x::name)
    quo(p,q,x), rem(p,q,x)
    #  Funktionen fuer Quotient und Rest
  end:

> pp := simplify((x^10-1)/(x-1));  qq := x^4-2;
  result := divpol(pp,qq,x);
```

$$pp := x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$$

$$qq := x^4 - 2$$

$$result := x^5 + x^4 + x^3 + x^2 + 3\,x + 3,\ 7 + 3\,x^3 + 3\,x^2 + 7\,x$$

```
> probe := result[1]*qq+result[2];
```

$$probe := (x^5 + x^4 + x^3 + x^2 + 3\,x + 3)\,(x^4 - 2) + 7 + 3\,x^3 + 3\,x^2 + 7\,x$$

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

$$x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$$

Beispiel 6 : Koordinatentransformation in 3D

Man entwickle eine Funktion cartes zur Umwandlung von Kugelkoordinaten (r,phi,theta) in cartesische Koordinaten (x,y,z).

Variante 1: Pfeilfunktion (s.oben)

```
> cartes1 := (r,phi,theta) ->
             [r*cos(phi)*cos(theta),
             r*sin(phi)*cos(theta),
             r*sin(theta)];
```

$$cartes1 := (r,\ \phi,\ \theta) \rightarrow [r\cos(\phi)\cos(\theta),\ r\sin(\phi)\cos(\theta),\ r\sin(\theta)]$$

```
> p1 := cartes1(10,0,Pi/2);  p2 := cartes1(10,Pi/2,0);
  p3 := cartes1(10,Pi/4,0);
```

$$p1 := [0,\ 0,\ 10]$$

$$p2 := [0,\ 10,\ 0]$$

$$p3 := [5\sqrt{2},\ 5\sqrt{2},\ 0]$$

Variante 2: Prozedur zu Vermeidung unnötiger Doppelberechnungen durch Definition lokaler Variablen

```
> cartes := proc(r,phi,theta)
            local h;
            h := r*cos(theta);
            [cos(phi)*h,
             sin(phi)*h,
             r*sin(theta)]
          end:
```

```
> p1 := cartes(10,0,Pi/2); p2 := cartes(10,Pi/2,0);
  p3 := cartes(10,Pi/4,0);
```

$$p1 := [0,\ 0,\ 10]$$

$$p2 := [0,\ 10,\ 0]$$

$$p3 := [5\sqrt{2},\ 5\sqrt{2},\ 0]$$

```
> z_1 := p1[3];
```

$$z\_1 := 10$$

Beispiel 7 : Approximation des Matrixexponenten exp(A(x))

```
> restart: with(linalg):
```

A(x) liefert für jedes x eine (2,2)-Matrix :

```
> A := proc(x) matrix(2,2,[x,0,0,-x]) end;
```

$$A := \mathbf{proc}(x)\ \mathbf{matrix}(2,\ 2,\ [x,\ 0,\ 0,\ -x])\ \mathbf{end}$$

```
> A(x);
```

$$\begin{bmatrix} x & 0 \\ 0 & -x \end{bmatrix}$$

```
> A(1);
```

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

```
> evalm(A(x)^0+A(x)+1/2!*A(x)^2);
```

$$\begin{bmatrix} x+\frac{1}{2}x^2+1 & 0 \\ 0 & -x+\frac{1}{2}x^2+1 \end{bmatrix}$$

```
> evalm(sum(1/k!*A(x)^k, k=0 .. 5));
```

$$\begin{bmatrix} x+\frac{1}{2}x^2+\frac{1}{6}x^3+\frac{1}{24}x^4+\frac{1}{120}x^5+1 & 0 \\ 0 & -x+\frac{1}{2}x^2-\frac{1}{6}x^3+\frac{1}{24}x^4-\frac{1}{120}x^5+1 \end{bmatrix}$$

B(x) liefert folgende Matrixfunktion:

```
> B := proc(x) matrix(2,2,[0,x,-x,0]); end;
```

$$B := \mathbf{proc}(x)\,\mathbf{matrix}(2,\,2,\,[0,\,x,\,-x,\,0])\,\mathbf{end}$$

```
> B(x), B(1);
```

$$\begin{bmatrix} 0 & x \\ -x & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

```
> evalm(B(x)^0+B(x)+1/2!*B(x)^2+1/3!*B(x)^3);
```

$$\begin{bmatrix} -\frac{1}{2}x^2+1 & x-\frac{1}{6}x^3 \\ -x+\frac{1}{6}x^3 & -\frac{1}{2}x^2+1 \end{bmatrix}$$

Wir definieren die Matrixfunktion M(x) :

```
> M := proc(x,n)
    evalm(sum(1/k!*B(x)^k, k=0 .. n));
  end;
```

$$M := \mathbf{proc}(x,\,n)\,\mathbf{evalm}(\mathbf{sum}(\mathrm{B}(x)^k/k!,\,k=0..n))\,\mathbf{end}$$

Approximation von sin , cos , sinh , cosh :

```
> Digits := 25:

> evalf(M(1,20));  [-sin(1.0), cos(1.0)];
  exponential(B(1));    #  Matrixexponent (aus linalg)
  exponential(B(1.0));  #  genaeherter Matrixexponent
```

$$\begin{bmatrix} .5403023058681397174018247 & .8414709848078965066329680 \\ -.8414709848078965066329680 & .5403023058681397174018247 \end{bmatrix}$$

$$[-.8414709848078965066525023,\ .5403023058681397174009366]$$

$$\begin{bmatrix} \cos(1) & \sin(1) \\ -\sin(1) & \cos(1) \end{bmatrix}$$

$$\begin{bmatrix} .5403023058681397174009366 & .8414709848078965066525023 \\ -.8414709848078965066525023 & .5403023058681397174009366 \end{bmatrix}$$

```
> evalf(M(I,15));
  matrix(2,2,[[cosh(1.0), sinh(1.0)*I],
     [-sinh(1.0)*I, cosh(1.0)*I]]);
  exponential(B(I));
  exponential(B(1.0*I));
```

$$\begin{bmatrix} 1.54308063481519582710058 9 & 1.17520119364379863718488 1\, I \\ -1.17520119364379863718488 1\, I & 1.54308063481519582710058 9 \end{bmatrix}$$

$$\begin{bmatrix} 1.54308063481524377847790 6 & 1.17520119364380145688238 2\, I \\ -1.17520119364380145688238 2\, I & 1.54308063481524377847790 6\, I \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{2}\, e^{(-1)} + \frac{1}{2}\, e & I\,(\frac{1}{2}\, e - \frac{1}{2}\, e^{(-1)}) \\ I\,(-\frac{1}{2}\, e + \frac{1}{2}\, e^{(-1)}) & \frac{1}{2}\, e^{(-1)} + \frac{1}{2}\, e \end{bmatrix}$$

$$\begin{bmatrix} 1.54308063481524377847790 6 & 1.17520119364380145688238 2\, I \\ -1.17520119364380145688238 2\, I & 1.54308063481524377847790 6 \end{bmatrix}$$

```
> evalm(M(t,8));
```

$$\begin{bmatrix} -\frac{1}{2}\, t^2 + \frac{1}{24}\, t^4 - \frac{1}{720}\, t^6 + \frac{1}{40320}\, t^8 + 1 & t - \frac{1}{6}\, t^3 + \frac{1}{120}\, t^5 - \frac{1}{5040}\, t^7 \\ -t + \frac{1}{6}\, t^3 - \frac{1}{120}\, t^5 + \frac{1}{5040}\, t^7 & -\frac{1}{2}\, t^2 + \frac{1}{24}\, t^4 - \frac{1}{720}\, t^6 + \frac{1}{40320}\, t^8 + 1 \end{bmatrix}$$

Explizites Return (RETURN)

Syntax der RETURN-Anweisung: **RETURN(<Rueckgabewert>);**

Die Prozedur beendet ihre Abarbeitung, bestimmt den Rückgabewert und übergibt ihn an die aufrufende Umgebung. Fehlt ein RETURN, so liefert die letzte ausgeführte Anweisung den Rückgabewert.

Beispiel 1 : Zugehörigkeit zu einer Liste

Es ist zu prüfen, ob ein Element x zu einer Liste L gehört oder nicht.

```
> restart:

> inlist := proc(x,L)
    local v;
    for v in L do
      if v = x then RETURN(true) fi
    od;
    false
  end:
> L1 := [2,3,5,7,11,13,17,19,23,29];
```

$$L1 := [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]$$

```
> inlist(7,L1); inlist(22,L1);
```

$$true$$
$$false$$

```
> readlib(ifactors):
> L2 := ifactors(30!)[2]; # Primfaktorenzerlegung von 30!
```

$$L2 := [[2, 26], [3, 14], [5, 7], [7, 4], [11, 2], [13, 2], [17, 1], [19, 1], [23, 1], [29, 1]]$$

```
> inlist([7,4],L2); inlist([17,2],L2);
```

$$true$$
$$false$$

Aufgabe: Man entwickle eine Prozedur, die zu gegebener natürlicher Zahl n untersucht, wie oft ein Faktor m in n enthalten ist.

Fehler-Return (ERROR)

Syntax der ERROR-Anweisung: **ERROR(<Argumentfolge>);**

Die Prozedur beendet unmittelbar ihre Abarbeitung mit folgender Fehlermeldung:

Error, (in <Procname>), <Argumentfolge>

Die Argumentfolge sollte einen Fehlerhinweis enthalten.

Beispiel 1 : Zugehörigkeit zu einer Liste (s.oben)

Es ist zu überprüfen, ob L eine Liste darstellt oder nicht.

```
> inlist := proc(x,L)
    local v;
    if not type(L,list) then
      ERROR(‘2.Argument muss eine Liste sein!‘)
    fi;
    for v in L do
      if v = x then RETURN(true) fi
    od;
    false
  end:
> L1 := [2,3,5,7,11,13,17,19,23,29]: inlist(7,L1); inlist(22,L1);
```

$$true$$
$$false$$

```
> inlist(7,LL); inlist(L1,7);

Error, (in inlist) 2.Argument muss eine Liste sein!

Error, (in inlist) 2.Argument muss eine Liste sein!
```

Beispiel 2 : Differentiation algebraischer Funktionen (s.oben)

Nachfolgend wird die Produktregel der Differentiation (rekursiv) eingebaut:

```
> deriv := proc(a::algebraic, x::name)
    local u,v;
    if type(a,numeric) then 0              # Konstante
    elif type(a,name) then                 # Variable
      if a=x then 1 else 0 fi
    elif type(a,`+`) then map(deriv,a,x)   # Summenregel
    elif type(a,`*`) then
      u:=op(1,a); v:=a/u; deriv(u,x)*v+deriv(v,x)*u
    else ERROR(`: kann`,a,`noch nicht differenzieren!`)
    fi;
  end:

> deriv(x+y-2.3+Pi*y,y); deriv(x*deriv(x*2,x)+3,x);
```

$$\frac{1+\pi}{2}$$

```
> deriv(3*x-(x-4)*(6*x+1)*x*(x+4),x);
```

$$3-(6x+1)x(x+4)-(6x(x+4)+(2x+4)(6x+1))(x-4)$$

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

$$19-24x^3-3x^2+192x$$

```
> deriv(5*x^4+3,x);

Error, (in deriv) : kann, x^4, noch nicht differenzieren!

> p := 1+x*(2+x*(3+x*(4+x*(5+x))));
```

$$p := 1+x(2+x(3+x(4+x(5+x))))$$

```
> deriv(p,x); simplify(%); # eigener Differezierer!
```

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

$$2+6x+12x^2+20x^3+5x^4$$

```
> diff(p,x); simplify(%); # Standarddifferenzierer von MAPLE!
```

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

$$2+6x+12x^2+20x^3+5x^4$$

Offenbar ist es sinnvoll, am Ende jedes Aufrufes von deriv eine Simplifikation des Ergebnisses vor dessen Rückgabe an das aufrufende Niveau vorzunehmen. Dies vermeidet die - leider in diff auftretenden - sehr langen Ausdrücke!

```
> deriv := proc(a::algebraic, x::name)
    local u,v;
    if type(a,numeric) then 0              # Konstante
    elif type(a,name) then                 # Variable
      if a=x then 1 else 0 fi
    elif type(a,`+`) then map(deriv,a,x)   # Summenregel
    elif type(a,`*`) then
      u:=op(1,a); v:=a/u; deriv(u,x)*v+deriv(v,x)*u
    else ERROR(`: kann`,a,`noch nicht differenzieren!`)
    fi;  simplify(%)
  end:
> deriv(3*x-(x-4)*(6*x+1)*x*(x+4),x);
```

$$19 - 24\,x^3 - 3\,x^2 + 192\,x$$

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

$$19 - 24\,x^3 - 3\,x^2 + 192\,x$$

```
> p := 1+x*(2+x*(3+x*(4+x*(5+x))));
```

$$p := 1 + x\,(2 + x\,(3 + x\,(4 + x\,(5 + x))))$$

```
> deriv(p,x); # eigener Differenzierer!
```

$$2 + 6\,x + 12\,x^2 + 20\,x^3 + 5\,x^4$$

```
> diff(p,x); # Standardddifferenzierer von MAPLE!
```

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

Aufgabe: Man programmiere zusätzlich die Quotientenregel und Kettenregel!

Namensparameter (call by name)

Ein Formalparameter p wird als Namensparameter explizit mittels **p :: evaln** deklariert. Diese Deklaration garantiert, daß Maple diesen Parameter zu einem Namen auswertet und keine volle Evaluierung vornimmt. So wird der Name p des Parameters, nicht jedoch der Wert von p, an die Prozedur übergeben.

Beispiel 1 : Zugehörigkeit eines Ausdruckes x zu einer Liste L

Es ist zu prüfen, ob ein Ausdruck x zu einer Liste L gehört oder nicht. Wenn die Funktion inlist mit einem 3. Argument p aufgerufen wird, so soll sie auch die Position von x in L an p zurückgeben.

Beispiele:

inlist (x^2, [1, x, y, x^3, y^3, x*y]); liefert false

inlist (x^3, [1, x, y, x^3, y^3, x*y] , pos); liefert true und 4

```
> inlist := proc(x::anything,L::list,pos::evaln)
    local i;
    for i to nops(L) do
      if x = L[i] then
        if  nargs > 2 then  pos:=i  fi;
        RETURN(true)
      fi
    od;  false
  end:
```

Bemerkung: Die Standardvariable nargs enthält bei jedem Funktionsaufruf die aktuelle Anzahl der Funktionsargumente (der aktuellen Parameter). Damit kann z.B. erfragt werden, ob 2 oder 3 aktuelle Parameter benutzt werden.

```
> L1 := [1, x, y, x^3, y^3, x*y ]:
> inlist(x^2,L1);
```

$$\mathit{false}$$

```
> inlist(x^3,L1);
```

$$\mathit{true}$$

```
> i := NULL:  inlist(x^2,L1,i), i;
  i := NULL:  inlist(x^3,L1,i), i;
```

$$\mathit{false}$$

$$\mathit{true}, 4$$

```
> L2 := [2,3,5,7,11,13,17,19,23,29]:

> pos := NULL:  inlist(23,L2,pos), pos;
  pos := NULL:  inlist(22,L2, pos), pos;
```

$$\mathit{true}, 9$$

$$\mathit{false}$$

```
> readlib(ifactors):
> L3 := ifactors(50!)[2]; # Primfaktorenzerlegung von 50!
```

$$L3 := [[2, 47], [3, 22], [5, 12], [7, 8], [11, 4], [13, 3], [17, 2], [19, 2], [23, 2], [29, 1], [31, 1], [37, 1], [41, 1], [43, 1], [47, 1]]$$

```
> pos := NULL:  inlist([7,4], L3, pos), pos;
  pos := NULL:  inlist([17,2],L3, pos), pos;
```

$$\mathit{false}$$

$$\mathit{true}, 7$$

5.4.5 Lokale und globale Variablen

Die allgemeine Syntax der Procedure-Anweisung lautet:

```
procedurname := proc (<formale Parameter>) # Procedure-Kopf
   global <Folge der globalen Variablen>; # Deklarationen
   local <Folge der lokalen Variablen>; # Deklarationen
   options <Folge der Einstellungen>; # Deklarationen
   description <Zeichenkette>; # Informationsteil
   <Folge der Procedure-Anweisungen> # Anweisungsteil
end; # Ende der Procedure
```

Variablen innerhalb einer Prozedur, die keine Parameter sind, heißen lokale oder globale Variablen. Empfehlung: Man deklariere alle derartigen Variablen explizit in der Prozedurdefinition !

Lokale Variablen

Falls nicht deklariert, so wird eine in der Prozedur stehende Variable als lokal betrachtet, falls sie

- links in einer Zuweisung steht oder
- als Indexvariable (Laufvariable) in for, seq auftritt.

Eine Warnung wird ausgegeben. Besser ist eine explizite Deklaration!

Beispiel 1: Funktion einer und mehrerer Veränderlicher (s.o)

```
> restart:
> f := proc(x) x^3-4*x+1-sin(x) end: # 1 Anweisung

> g := proc(x,y,z)          #  2 Anweisungen!
                            #  local u;
    u := f(x+y-z)-f(y);
    u^2-3*u+1               #  Rueckgabewert!
  end:

Warning, `u` is implicitly declared local
> g(t+1,2,3);
```

$$(t^3 - 4\,t - \sin(t) + \sin(2))^2 - 3\,t^3 + 12\,t + 3\sin(t) - 3\sin(2) + 1$$

Beispiel 2 : Zugehörigkeit zu einer Liste (s.oben)

Es ist zu überprüfen, ob L eine Liste darstellt oder nicht.

```
> inlist := proc(x,L)
    local v;
    if not type(L,list) then
      ERROR(‘2.Argument muss eine Liste sein!‘)
    fi;
    for v in L do
      if v = x then RETURN(true) fi
    od;  false
  end;
```

$$\textit{inlist} := \textbf{proc}(x,\ L)\ \textbf{local}\ v;\ \textbf{if not}\ \mathrm{type}(L,\ \textit{list})\ \textbf{then}\ \mathrm{ERROR}(\text{‘}\textit{2.Argument muss eine Liste sein!}\text{‘})\ \textbf{fi};\ \textbf{for}\ v\ \textbf{in}\ L\ \textbf{do if}\ v = x\ \textbf{then}\ \mathrm{RETURN}(\textit{true})\ \textbf{fi od};\ \textit{false}\ \textbf{end}$$

```
> L1 := [2,3,5,7,11,13,17,19,23,29]:
  inlist(7,L1); inlist(22,L1);
```

$$\textit{true}$$
$$\textit{false}$$

Der Gültigkeitsbereich lokaler Variablen ist der Prozedurkörper. In der Prozedurumgebung stehende Variablen gleichen Namens sind dann nicht erreichbar, sie werden überdeckt.

Beispiel 1: Funktion einer und mehrerer Veränderlicher (s.o)

```
> restart:

> f := proc(x)  x^3-4*x+1-sin(x)  end:
  g := proc(x,y,z)
    local u;        #  explizite Deklaration
    u := f(x+y-z)-f(y);
    lprint(‘Lokales u  = ‘, u);
    u^2-3*u+1
  end:
> u := exp(1.0);
```

$$u := 2.718281828$$

```
> lprint(‘Globales u = ‘, u);  g(4,2,3):
  lprint(‘Globales u = ‘, u);
```

```
Globales u =    2.718281828

Lokales u  =    15-sin(3)+sin(2)

Globales u =    2.718281828
```

Export lokaler Variablen

Im Gegensatz zu klassischen Programmiersprachen müssen explizit deklarierte lokale Variablen <var> nicht unbedingt einen Wert erhalten, sie können atomar bleiben. Existiert eine Variable des Namens <var> auch außerhalb der Prozedur, so treten dann ggf. 2 Exemplare (Instanzen) von <var> nach dem Prozeduraufruf auf!

Beispiel 3 : Export einer lokalen Variablen

```
> h := proc(x) local var;
    x+var    #  var im Rueckgabewert
  end:
> h(par);
```

$$par + var$$

```
> var := 23; h(1); var;
```

$$var := 23$$
$$1 + var$$
$$23$$

```
> d := var-h(1);
```

$$d := 22 - var$$

```
> var; d;
```

$$23$$
$$22 - var$$

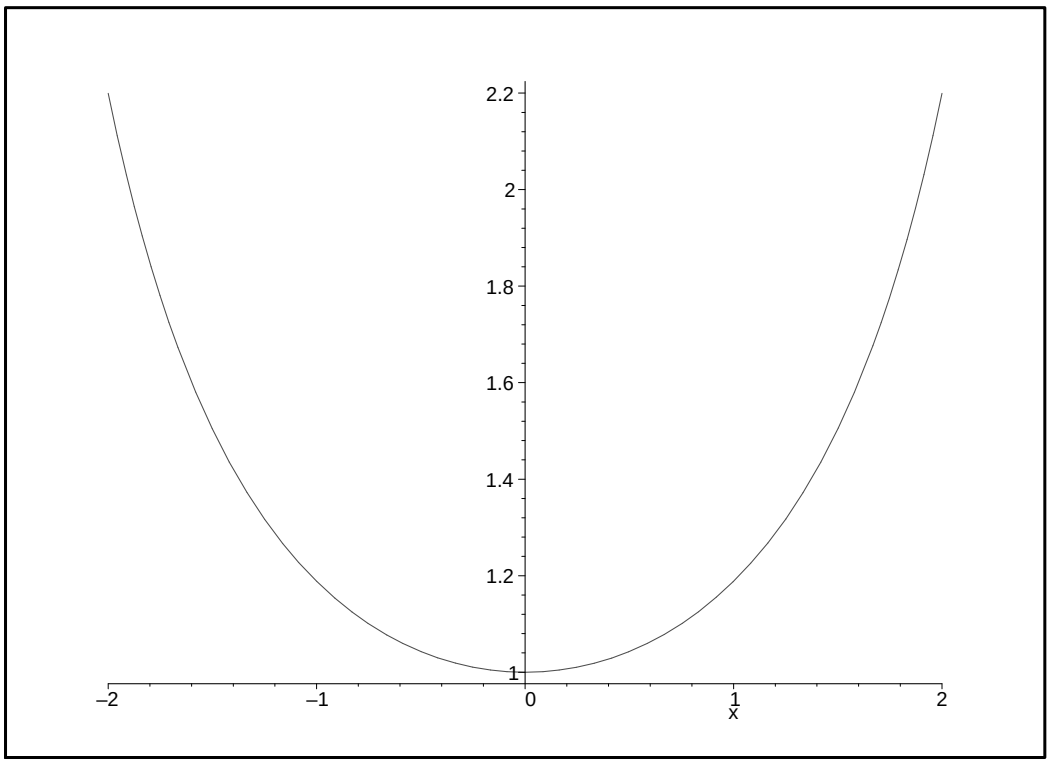
Dieses lokale <var> kann nun nicht mehr verändert werden. Auf diese Weise können beliebig viele Instanzen ein- und derselben Variablen generiert werden ...

Globale Variablen

Um Unklarheiten zu vermeiden, sollten alle Variablen aus der Prozedurumgebung, die im Prozedurkörper benutzt werden, explizit als global deklariert werden. Diese globalen Variablen sollten in der Prozedur möglichst nicht verändert werden („Prozeduren mit Nebeneffekt“).

Beispiel 1: Funktion mit Zusatz-Parameter

```
> restart:
> f := proc(x) x/sin(x) end: # Funktion 1 Variablen
> plot(f(x), x=-2 .. 2);
```




```
> int(f(x),x); # nicht geschlossen integrierbar
> w1 :=Int(f(x), x=-2 .. 2);
```

$$w1 := \int_{-2}^{2} \frac{x}{\sin(x)}\, dx$$

```
> evalf(w1);
```

$$5.248819044$$

Nun wird ein Zusatzparameter a mit $0.5 \le a \le 1.5$ in y = f(x) eingeführt. Die Abhängigkeit der Integralwerte von a soll als Tabelle dargestellt werden.

```
> f := proc(x)
    global a;
    x/sin(a*x)
  end:    #  Funktion mit Parameter a
> int(f(x),x); # nicht geschlossen integrierbar
> w := a -> evalf(Int(f(x), x=-2 .. 2));
```

$$w := a \rightarrow \mathrm{evalf}\left(\int_{-2}^{2} \mathrm{f}(x)\, dx\right)$$

```
> a := 1.25: w(a);
```

$$\mathbf{5.271167015}$$

```
> for a from 0.5 by 0.125 to 1.5 do
    lprint(‘a = ‘, a, ‘   w = ‘, w(a));
  od:
```

```
a =    .5      w =    8.478102347

a =    .625      w =    7.024493823

a =    .750      w =    6.126722736

a =    .875      w =    5.567032326

a =    1.000      w =    5.248819044

a =    1.125      w =    5.140592597

a =    1.250      w =    5.271167015

a =    1.375      w =    5.794263378

a =    1.500      w =    7.515795773
```

Evaluation globaler und lokaler Variablen

Maple evaluiert globale Variablen vollständig, also (rekursiv) solange, bis keine weiteren Substitutionen mehr möglich sind. Lokale Variablen werden jedoch - wie in klassischen Programmiersprachen - nur auf einem Niveau evaluiert!

Beispiel 2: Evaluation von Variablen

```
> restart:
> wert := x+y; x := z/y; z := y^2+2; wert;
```

$$wert := x + y$$
$$x := \frac{z}{y}$$
$$z := y^2 + 2$$
$$\frac{y^2 + 2}{y} + y$$

```
> # Kontrolle der Entwicklungsniveaus
  eval(wert,1);  eval(wert,2);  eval(wert,3);
```

$$x + y$$
$$\frac{z}{y} + y$$

$$\frac{y^2+2}{y}+y$$

```
>  f := proc()
     global x,y,z,wert;   # volle Evaluation
     wert := x+y;
     x    := z/y;
     z := y^2+2;
     wert
   end:
>  f();
```

$$\frac{y^2+2}{y}+y$$

```
>  f := proc()
     local x,y,z,wert;    # 1 Niveau
     wert := x+y;
     x    := z/y;
     z := y^2+2;
     wert
   end:
>  f();
```

$$x+y$$

```
>  # Ausweg: Lokale Variablen und eval
   f := proc()
     local x,y,z,wert;
     wert := x+y;    x := z/y;
     z    := y^2+2;  eval(wert)
   end:
>  f();
```

$$\frac{y^2+2}{y}+y$$

5.5 Grafik und Visualisierung mit Maple

Visualisierung beschäftigt sich mit der Repräsentation, Manipulation und Darstellung von

- wissenschaftlichen Daten (scientific visualization)
- Wirtschaftsdaten (business-data visualization) und
- Daten aus Datenbanken (information visualization).

Bedeutung der Visualisierung: Da durch unsere Wahrnehmung Bildinformationen sehr rasch registriert werden, ermöglicht die Visualisierung durch grafische Repräsentation eine rasche Analyse und besseres Verständnis der zugrunde liegenden Daten.

Aspekte der Visualisierung: Der wichtigste Aspekt ist die korrekte und leicht verständliche Anzeige der Daten; Aspekte wie realistische Wiedergabe sind zweitrangig... (vgl. Informatik-Handbuch, 2.Aufl., S.848)

Beispiel: Die Klein'sche Flasche

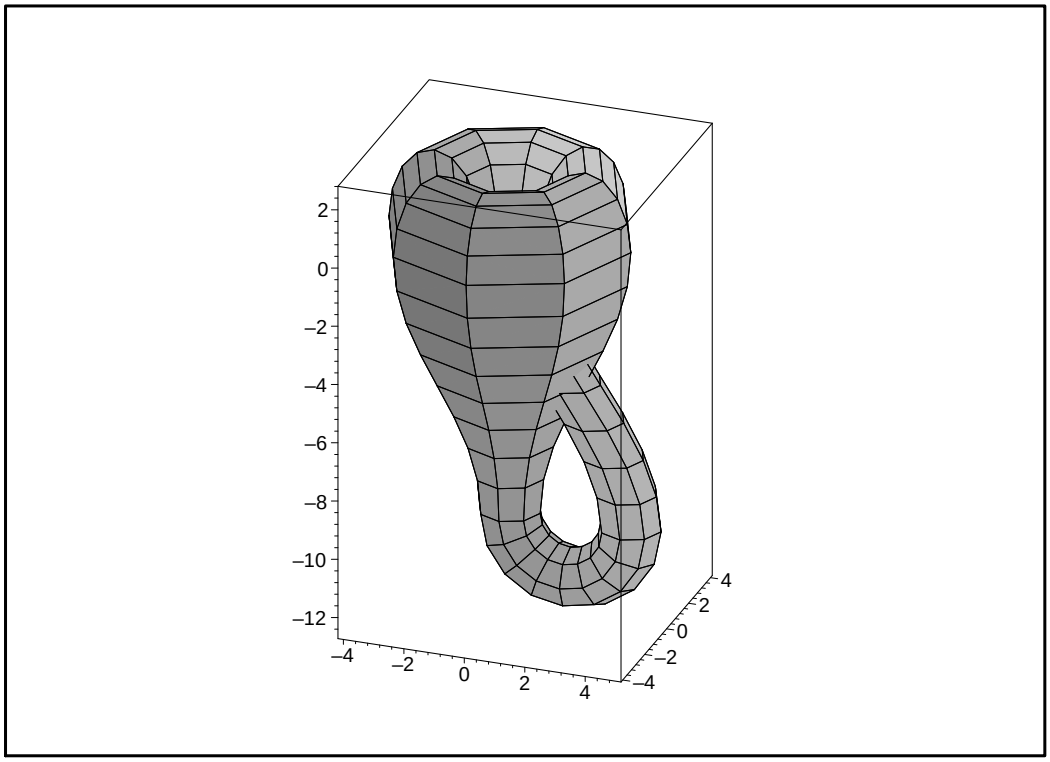
Im Gegensatz zu Kugelfläche und Torusfläche besitzt die sog. Klein'sche Flasche lediglich eine einzige Seite, also kann hier nicht zwischen „Innen“ und „Außen“ unterschieden werden!

(Eine Darstellung in der üblichen Parametrisierung findet man z.B. bei M.B.Monagan u.a.: Maple V - Programming Guide. Springer Verlag 1998, S.296.)

```
> restart: with(plots):

> kleinpoints := proc()
    local bottom,middle,handle,top,p,q;
    top := [(2.5 +1.5*cos(v))*cos(u),
       (2.5 +1.5*cos(v))*sin(u),-2.5*sin(v)]:
    middle := [(2.5 +1.5*cos(v))*cos(u),
       (2.5 +1.5*cos(v))*sin(u),3*v-6*Pi]:
    handle := [2.0 -2.0*cos(v)+sin(u),cos(u),3.0*v-6*Pi]:
    bottom := [2+(2+cos(u))*cos(v),sin(u),
       -3*Pi+ (2.0+cos(u))*sin(v)]:
    p := plot3d(\{bottom,middle,handle,top\},
         u=0..2*Pi, v=Pi..2*Pi, grid=[9,9]):
    p := select( x-> op(0,x) = MESH, [op(p)]);
    seq( convert(q, POLYGONS), q=p);
  end:
```

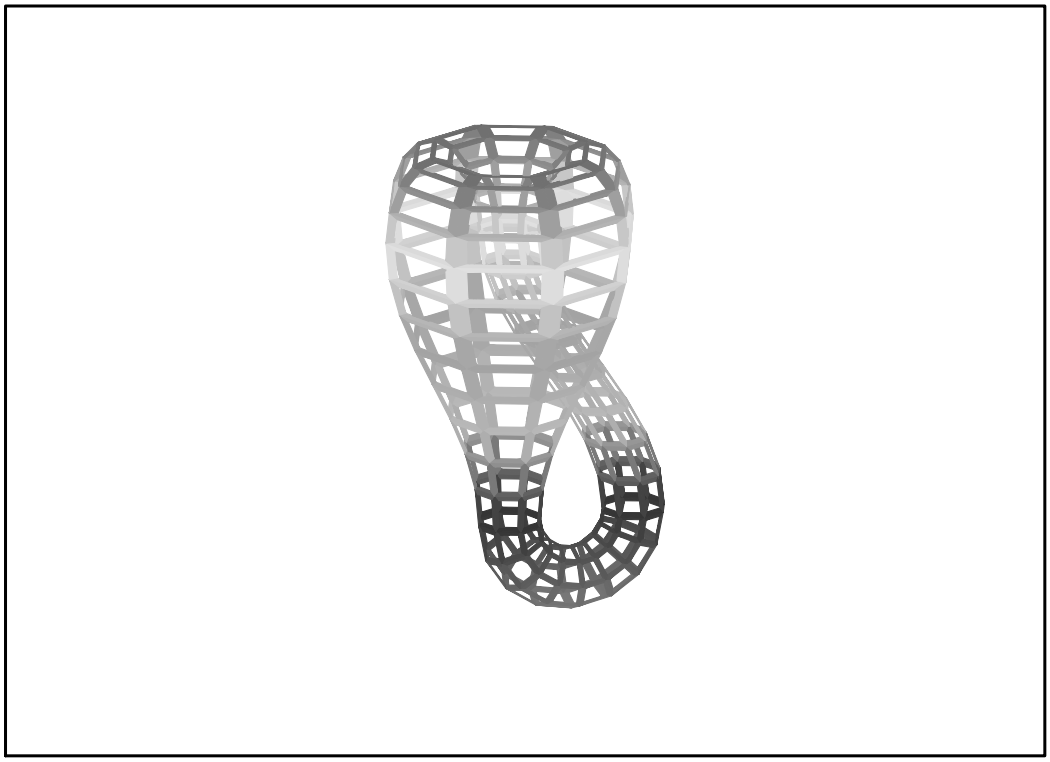
```
> display(kleinpoints(), style=patch, scaling=constrained,
          orientation=[-70,65], axes=boxed);
```



Darstellung mit ausgeschnittenen Patches:

```
> with(plottools):

> display(seq(cutout(k, 3/4), k=kleinpoints()),
          scaling=constrained,orientation=[-110,71],
          shading=zhue);
```



Hinweis: Die folgenden 3 Abschnitte basieren weitgehend auf der Literatur Heck, A.: Introduction to Maple. 2nd edition,Chapter 15. Springer New York, Berlin 1996.

5.5.1 Zweidimensionale Grafik

Einfache 2D-Plots

Das Kommando plot dient der Darstellung von Kurven, die durch y = f(x) über x parametrisiert sind. Der Bereich für x ist anzugeben. Im einfachsten Fall werden die Koordinatenachsen gezeichnet und markiert, aber nicht beschriftet.

Laden des Plot-Packages mit Anzeige der Plot-Kommandos:

```
> with(plots); # Laden des Grafik-Pakets
```

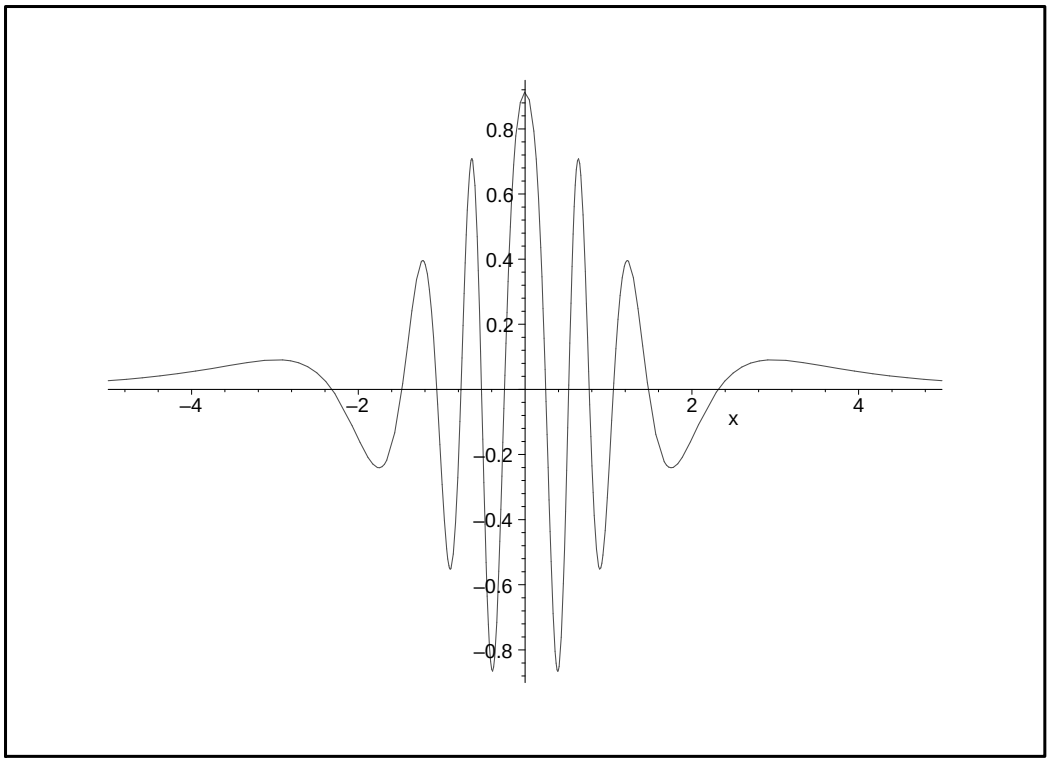
[*animate*, *animate3d*, *animatecurve*, *changecoords*, *complexplot*, *complexplot3d*, *conformal*, *contourplot*, *contourplot3d*, *coordplot*, *coordplot3d*, *cylinderplot*, *densityplot*, *display*, *display3d*, *fieldplot*, *fieldplot3d*, *gradplot*, *gradplot3d*, *implicitplot*, *implicitplot3d*, *inequal*, *listcontplot*, *listcontplot3d*, *listdensityplot*, *listplot*, *listplot3d*, *loglogplot*, *logplot*, *matrixplot*, *odeplot*, *pareto*, *pointplot*, *pointplot3d*, *polarplot*, *polygonplot*, *polygonplot3d*, *polyhedra_supported*, *polyhedraplot*, *replot*, *rootlocus*, *semilogplot*, *setoptions*, *setoptions3d*, *spacecurve*, *sparsematrixplot*, *sphereplot*, *surfdata*, *textplot*, *textplot3d*, *tubeplot*]

Darstellung einer Funktion einer Variablen:

```
> f := x -> 1/(x^2+1) * sin(20/(x^2+1));
```

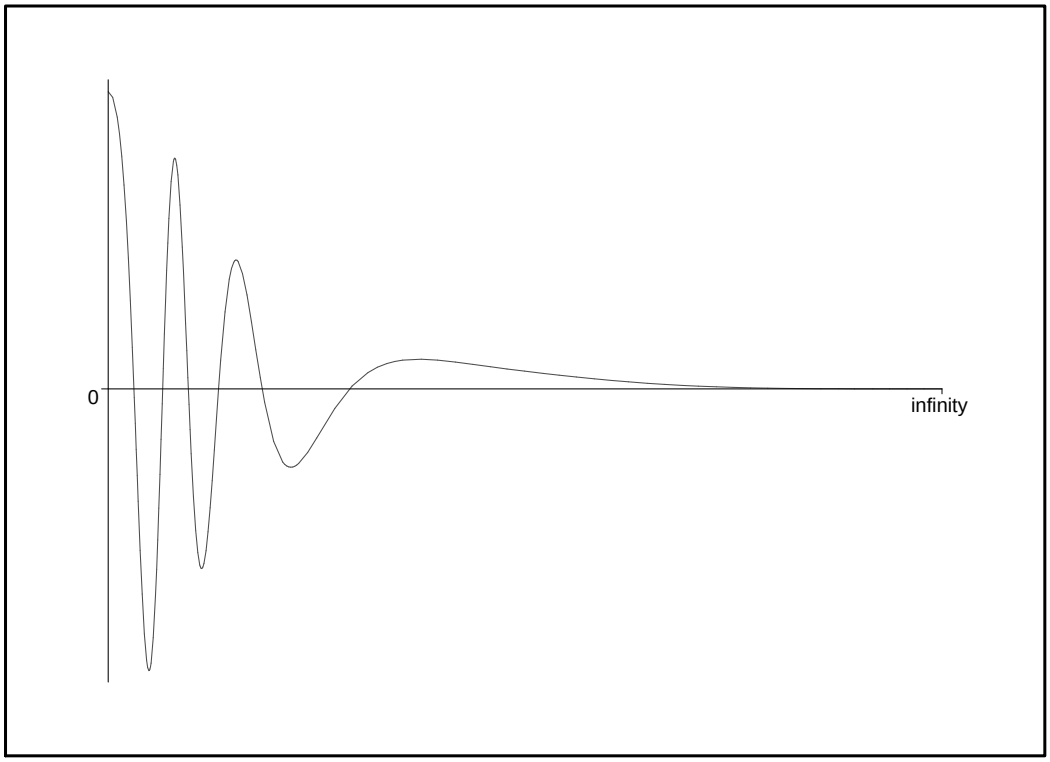
$$f := x \rightarrow \frac{\sin(20\,\frac{1}{x^2+1})}{x^2+1}$$

```
> plot( f(x), x = -5..5 );
```



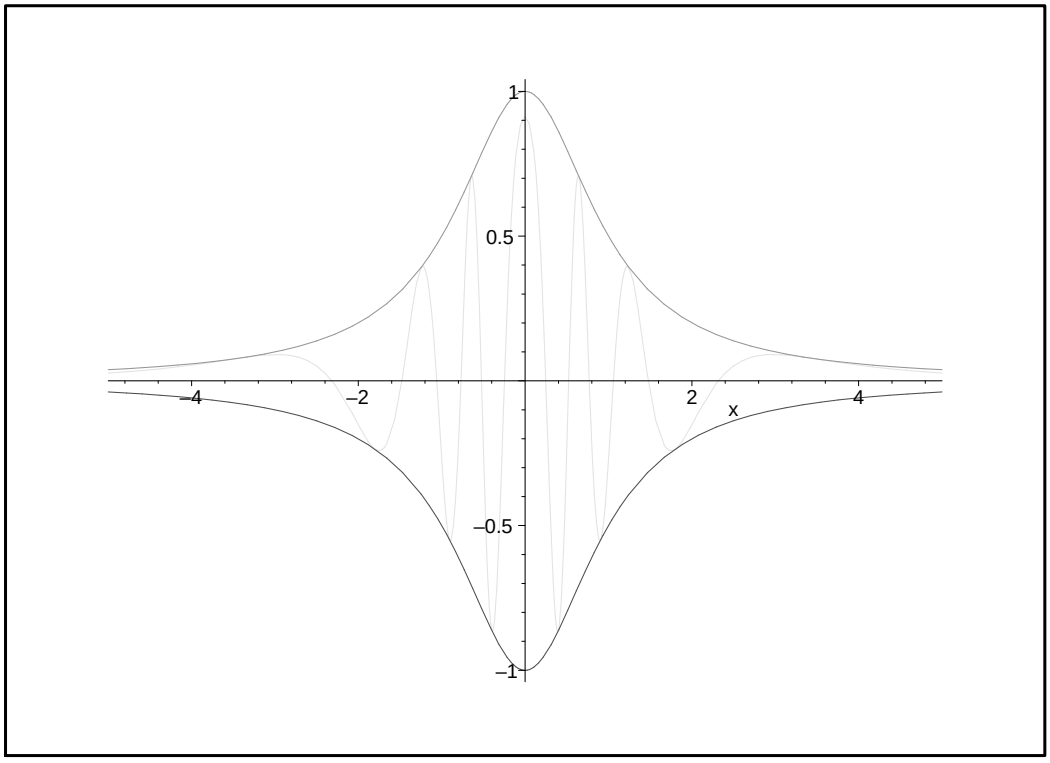
Darstellung einer Funktion in funktionaler Form (ohne Angabe von Variablen):

```
> plot( f, 0 .. infinity );
```



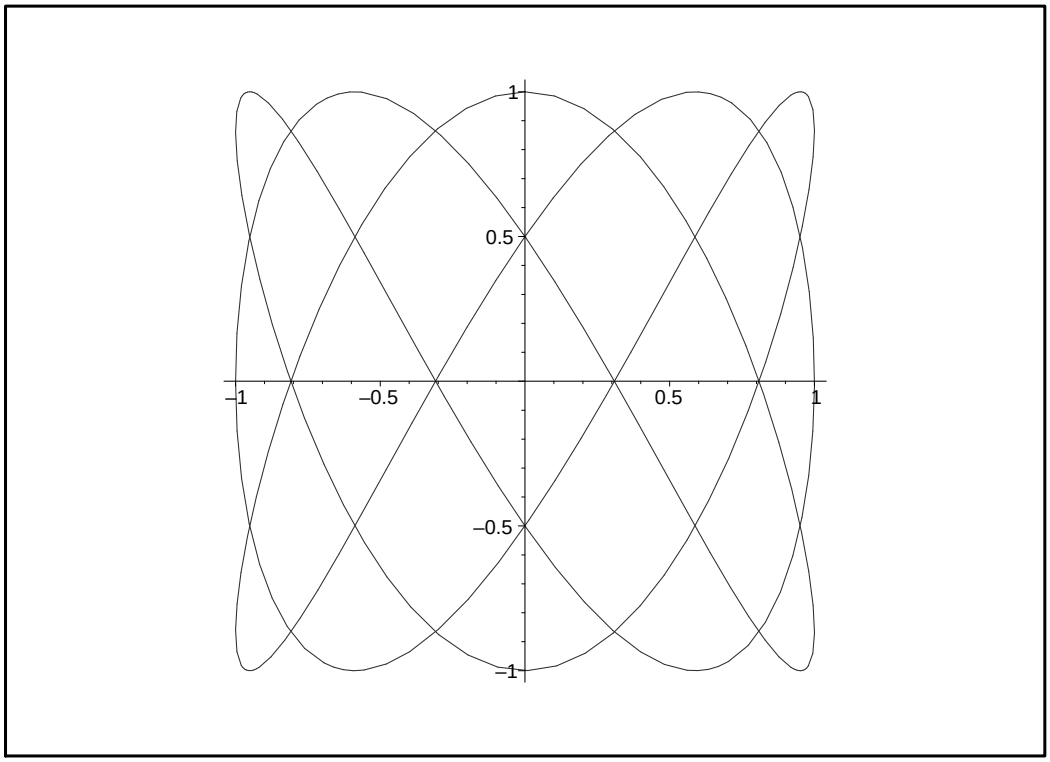
Darstellung mehrer Funktion in einem Bild ; die Farben werden automatisch gesetzt:

```
> plot( {f(x), 1/(x^2+1), -1/(x^2+1)}, x=-5..5);
```



Die Darstellung einer Kurve in Parameterform x = f(t), y=g(t) erfolgt durch das Kommando plot ([f(t), g(t), t = a .. b], Optionen);

```
> plot( [ sin(3*t), cos(5*t), t=0..2*Pi ],
        color=navy, scaling=constrained );
```

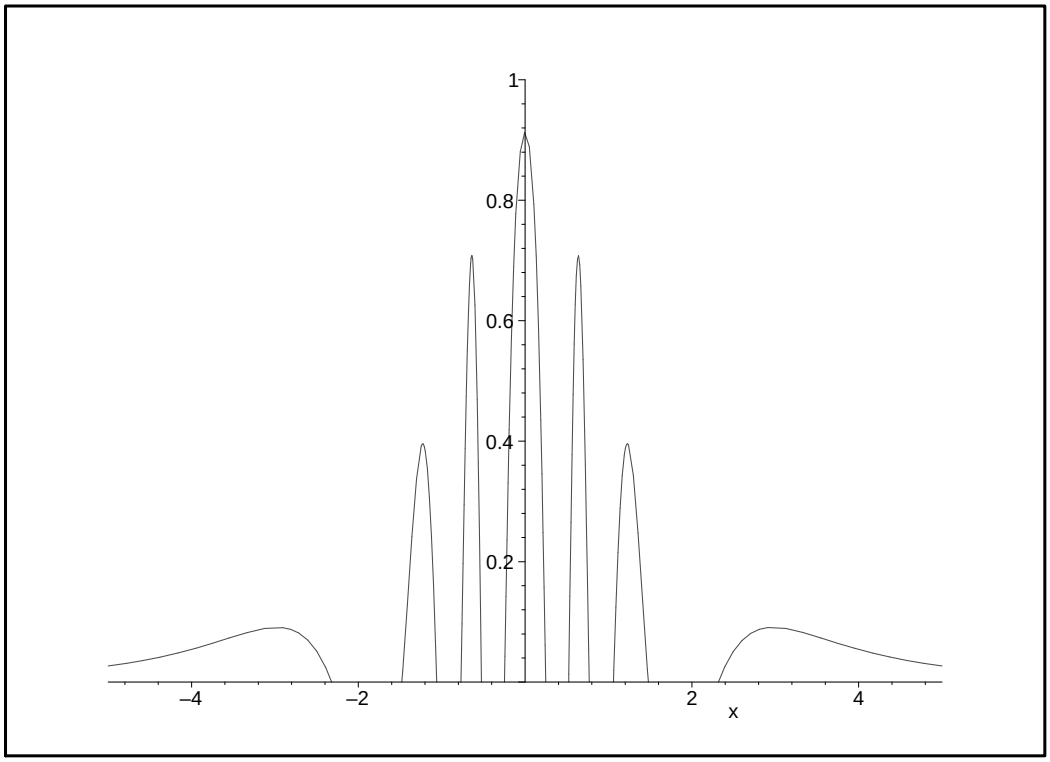


Optionen von plot

Maple verfügt über zahlreiche Optionen, mit denen die Standard-Darstellung modifiziert werden kann. Hier werden die wesentlichsten Optionen genannt.

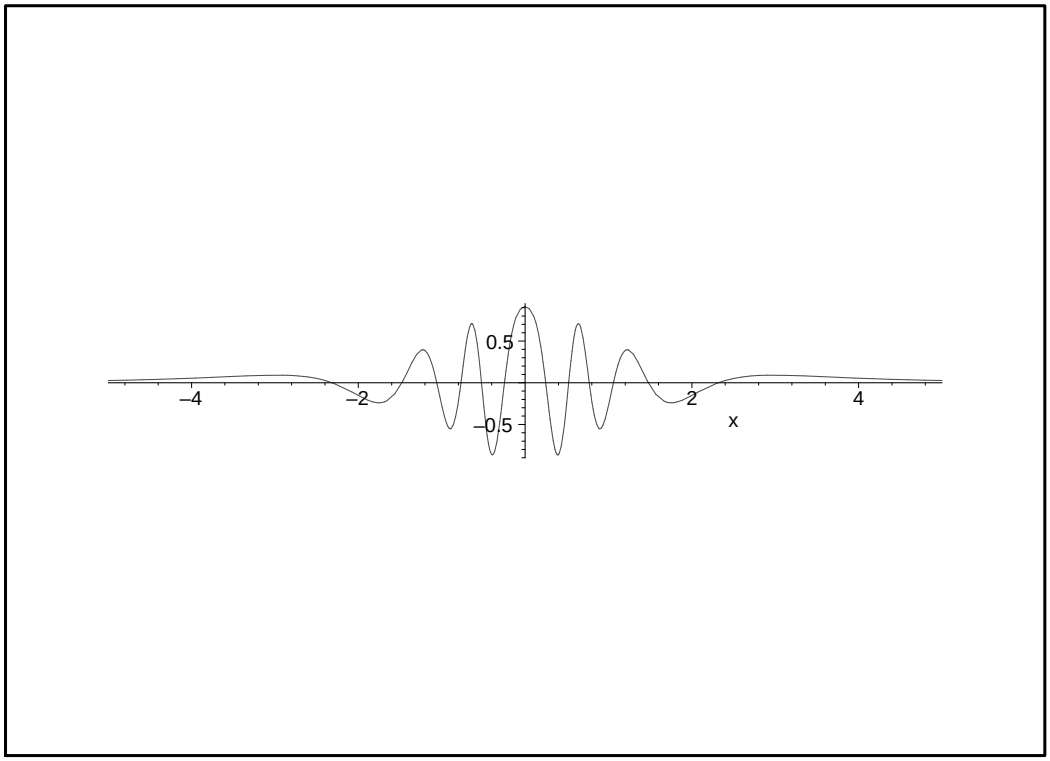
Der vertikale Zeichenbereich kann angegeben werden:

```
> plot( f(x), x=-5..5, 0..1 );
```



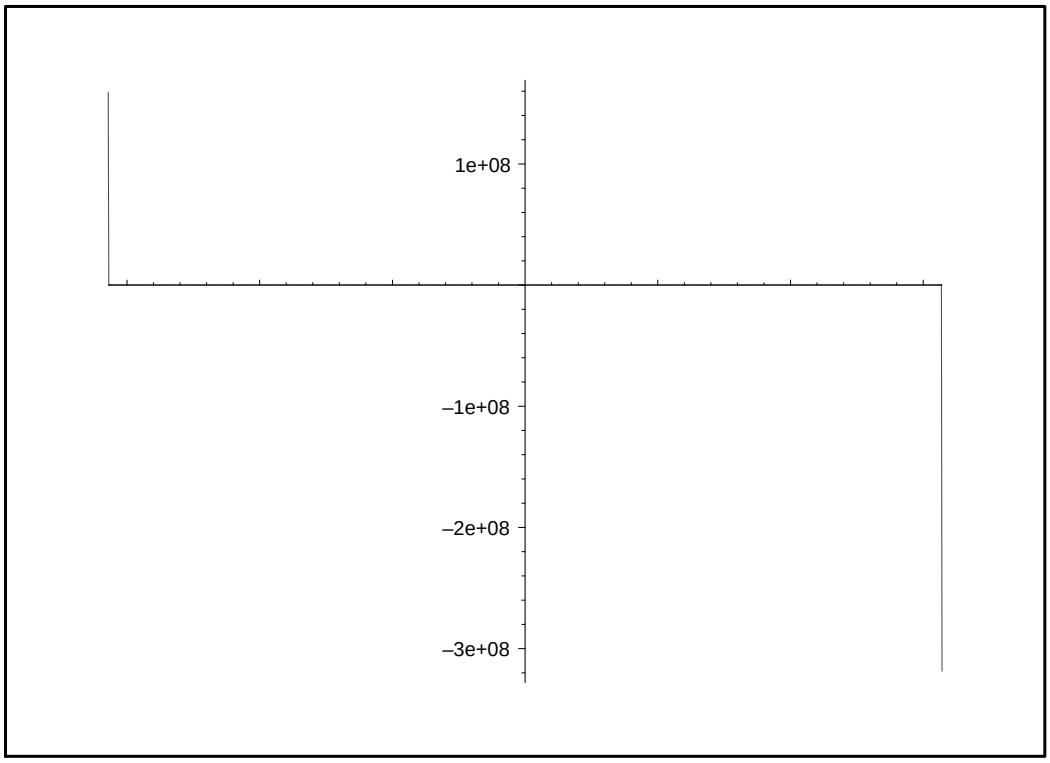
Gleiche Skalierung beider Achsen (andernfalls wird das Bildformat 2:3 eingestellt):

```
> plot( f(x), x=-5..5, scaling=constrained );
```

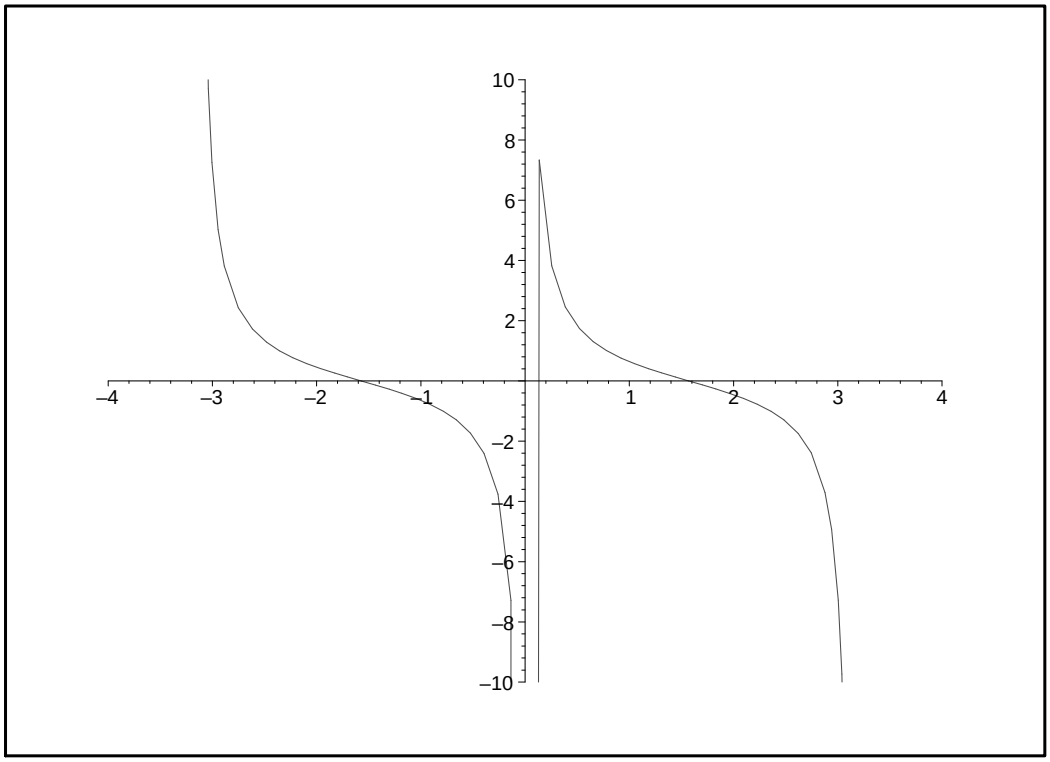


Ein Bildausschnitt kann mittels view, z.B. bei Funktionen mit Polstellen, ausgewählt werden. Das Kommando display berechnet den Graphen nicht nochmals neu.

```
> plot( cot, -Pi..Pi );
```

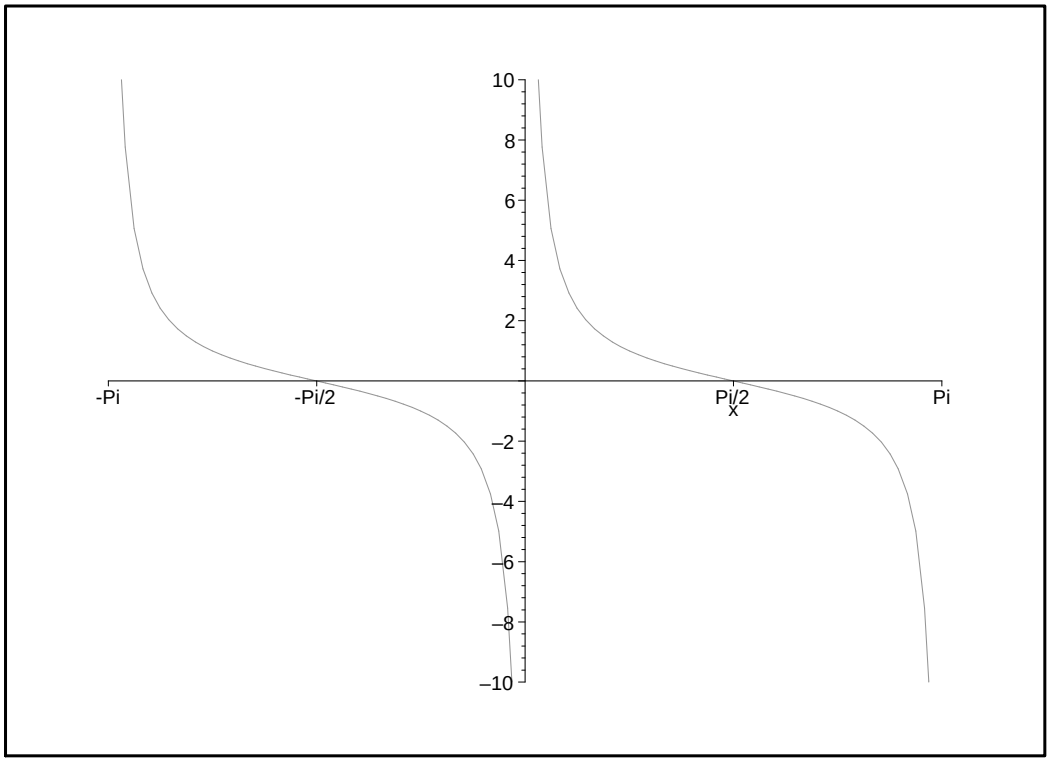



```
> display( %, view=[-4..4,-10..10] );
```



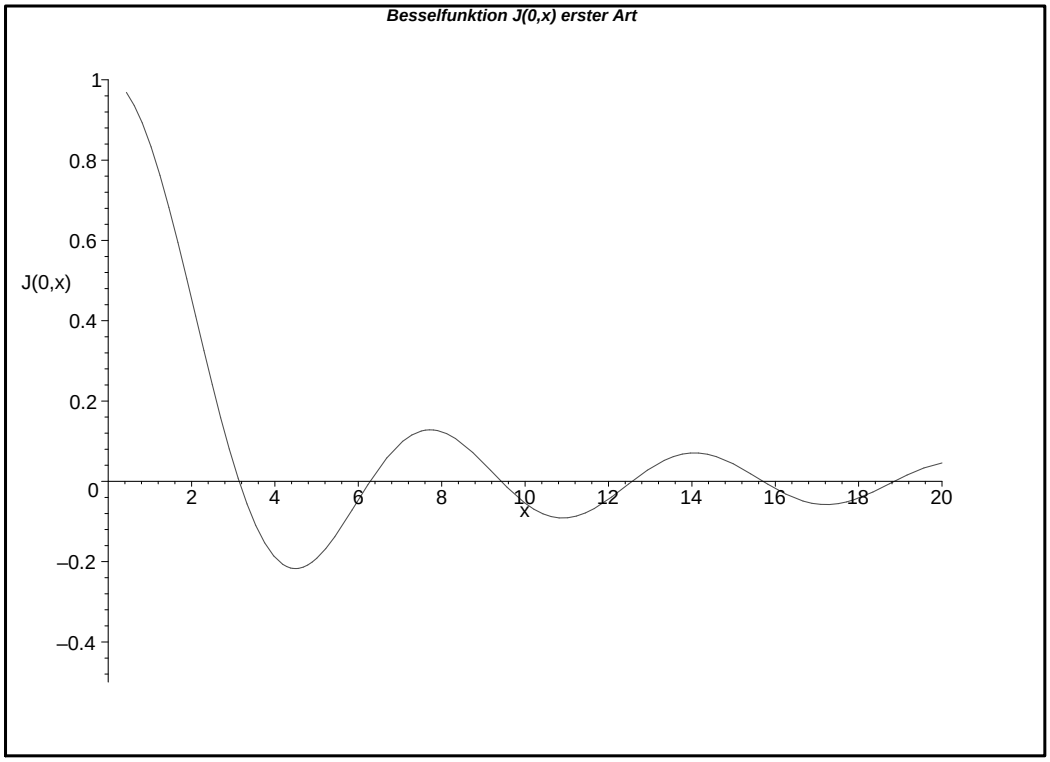
Bei Unstetigkeiten der Funktion kann dies mitgeteilt werden; dann erfolgt keine Verbindung von Funktionswerten über die Unstetigkeitsstellen hinweg:

```
> plot( cot(x), x=-Pi..Pi, -10..10, discont=true,
> xtickmarks=[ -3.14=‘-Pi‘, -1.57=‘-Pi/2‘,
> 1.57=‘Pi/2‘, 3.14=‘Pi‘ ] );
```



Explizite Angabe der Zahl von Achsenmarkierungen in x- und in y-Richtung sowie Ausgabe einer Überschrift:

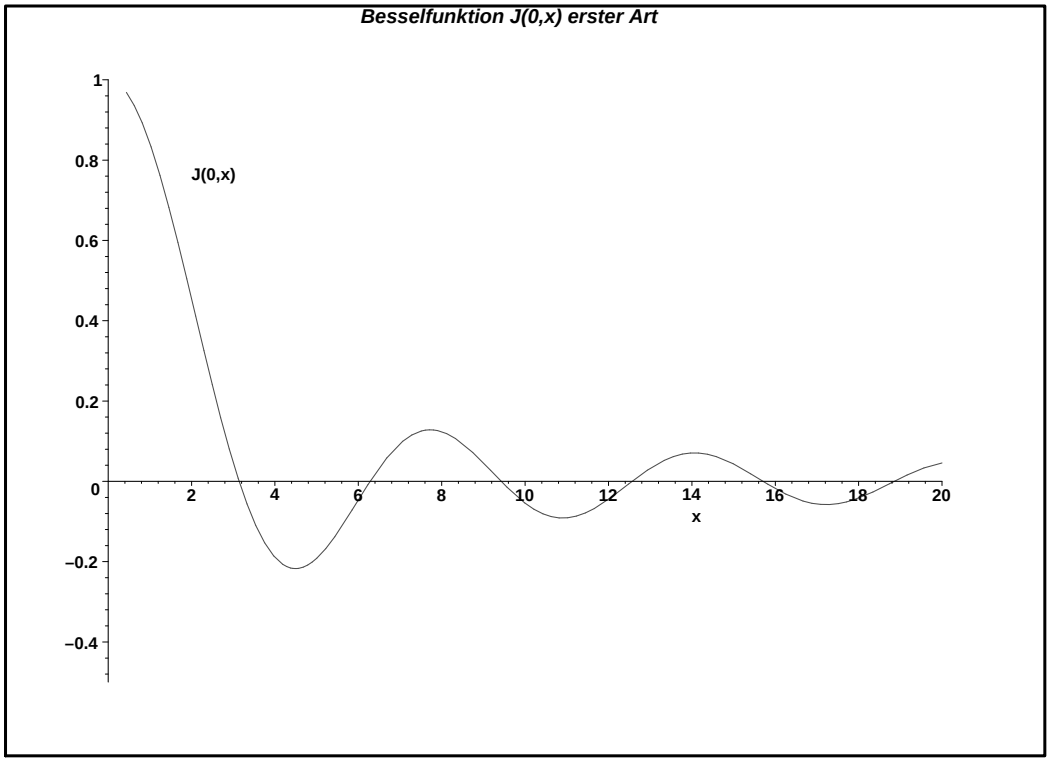
```
> J := (n,x) -> sqrt(Pi/(2*x)) * BesselJ(n+1/2,x):
> plot( J(0,x), x=0..20, `J(0,x)`=-0.5..1,
> xtickmarks=8, ytickmarks=4, title=
> `Besselfunktion J(0,x) erster Art`,
> titlefont=[HELVETICA,BOLDOBLIQUE,10] );
```



Zur Ausgabe von Kurven und Text in einem Bild verfährt man wie folgt:

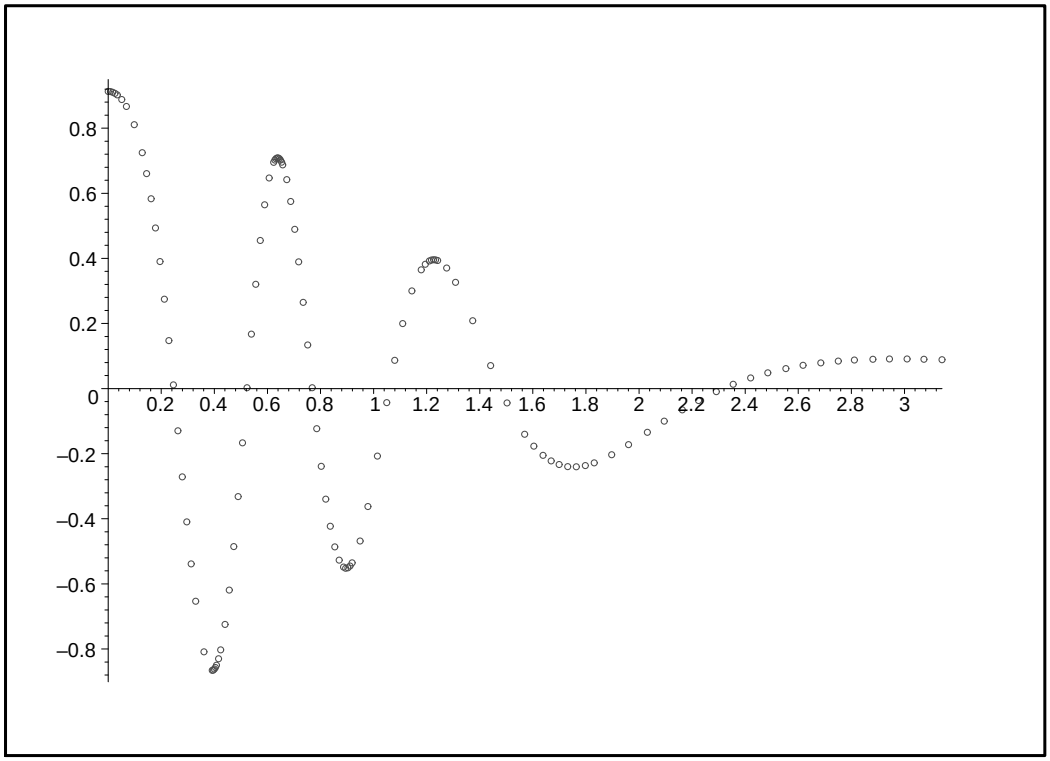
1. Erzeugung eines Graphik-Objektes curve mit dem Funktionsgraph, den Achsen und deren Markierung sowie einer Überschrift
2. Erzeugung des Objektes text mittels textplot zur Beschriftung der Kurve und der x-Achse in gewähltem Schriftfonds
3. Darstellung beider Objekte in einem Bild mittels display.

```
> curve := plot( x->J(0,x), 0..20, -0.5..1,
> xtickmarks=8, ytickmarks=4, title=
> ‘Besselfunktion J(0,x) erster Art‘,
> titlefont=[HELVETICA,BOLDOBLIQUE,12],
> axesfont=[HELVETICA,BOLD,10] ):
> text := plots[textplot]( { [2,0.75,‘J(0,x)‘],
> [14,-0.1,‘x‘] }, align={ABOVE,RIGHT},
> font=[HELVETICA,BOLD,10] ):
> plots[display]( { curve, text } );
```



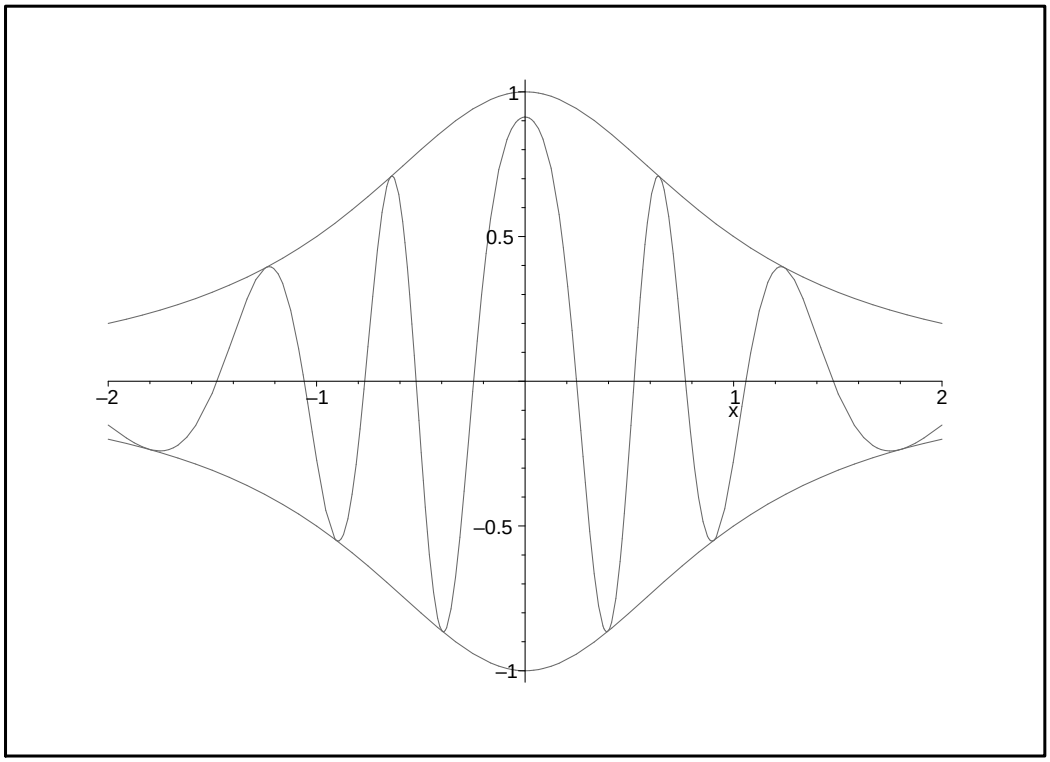
Maple verfügt über verschiedene Plot-Stile, z.B.:

```
> plot( f, 0..Pi, style=point,symbol=circle );
```



Maple verfügt über mehrere Linienstile und Linienstärken, z.B.:

```
> plot( {f(x), 1/(x^2+1), -1/(x^2+1)}, x=-2..2,
        thickness=2, color= magenta);
```



Die Struktur von 2D-Grafikobjekten

Die Ausführung des Plot-Kommandos erfolgt in 2 Schritten:

1. Berechnung der Plot-Punkte und Abspeicherung in einem PLOT-Objekt
2. Darstellung des Objektes auf dem Bildschirm.

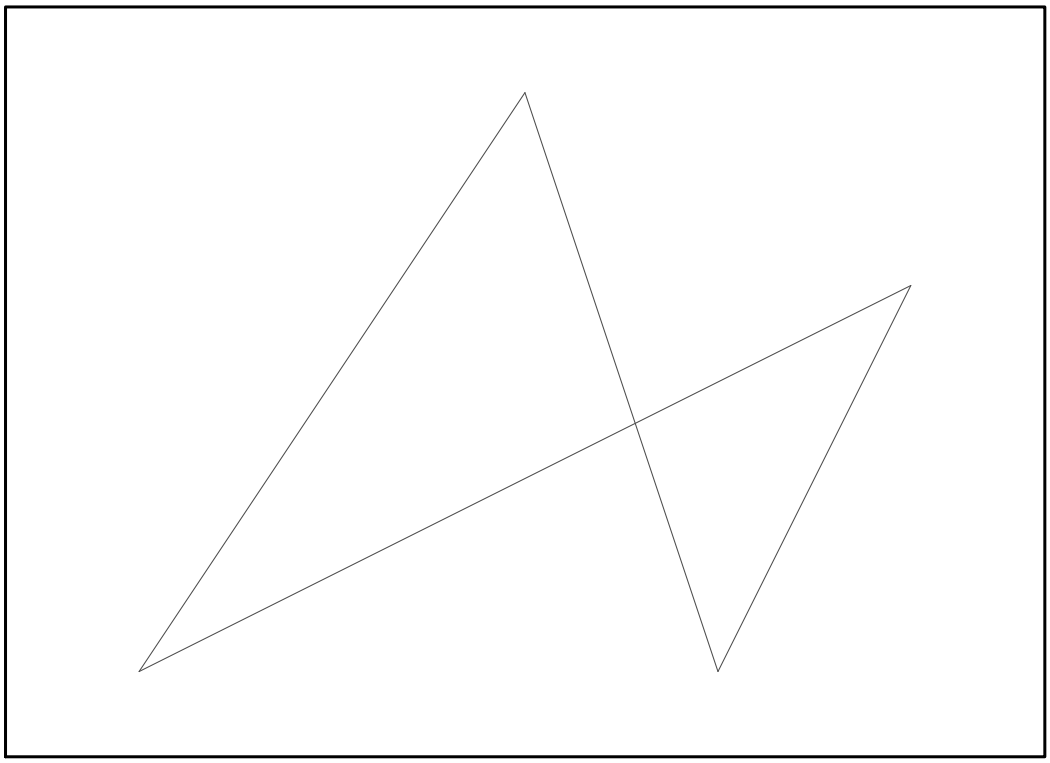
Bei Abschluß mit Doppelpunkt wird das PLOT-Objekt (als Text) ausgegeben bzw. einer Variablen zugewiesen.

Wir konstruieren zuerst ein einfaches PLOT-Objekt:

```
> restart; with(plots):
> P := plot( [ [1,1], [3,4], [4,1], [5,3], [1,1] ],
> axes=none, scaling=constrained ):
> lprint( P ); # Druck der Plotstruktur
```

```
PLOT(CURVES([[1., 1.], [3., 4.], [4., 1.], [5., 3.], [1.,
1.]],COLOUR(RGB,1.0,0,0)),AXESLABELS(``,``),AXESSTYLE(NONE),SCALING(CO
NSTRAINED),VIEW(DEFAULT,DEFAULT))
```

```
> P; # Druck des Grafik-Objekts
```

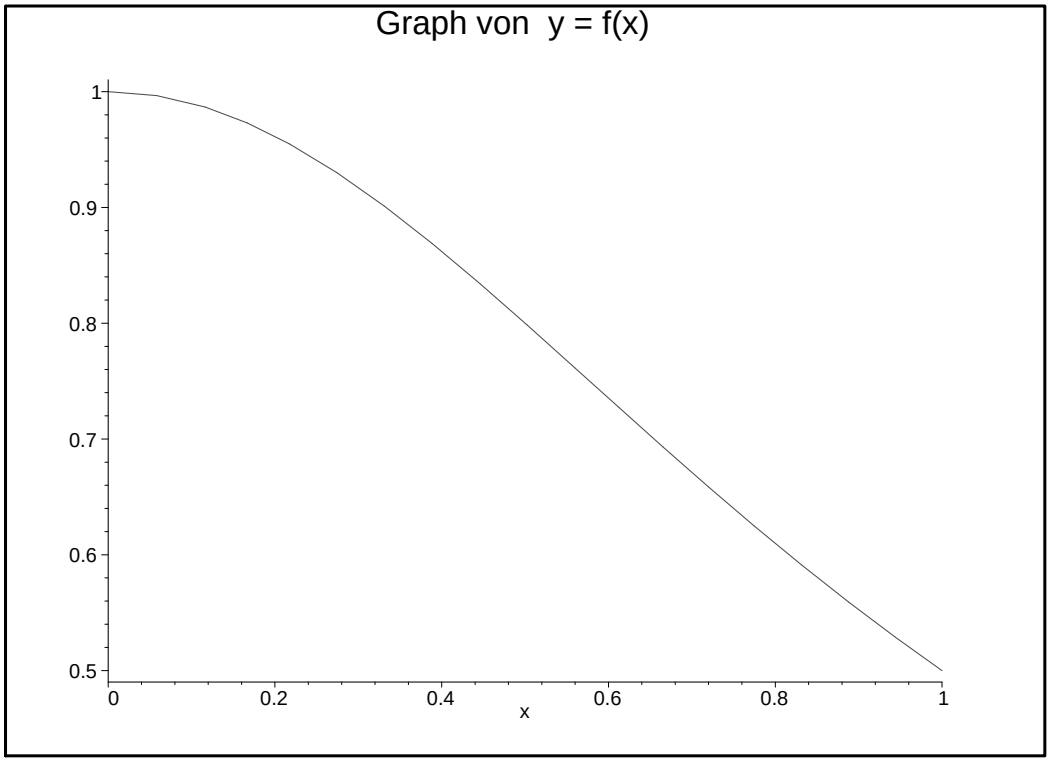


Nun schauen wir uns das PLOT-Objekt einer Funktionsgrafik an:

```
> f := x -> 1/(x^2+1):
> P := plot( f(x), x=0..1, numpoints=10,
> title=‘Graph von y = f(x)‘ ):
> lprint( P ); # Ausgabe des Graphik-Objektes
```

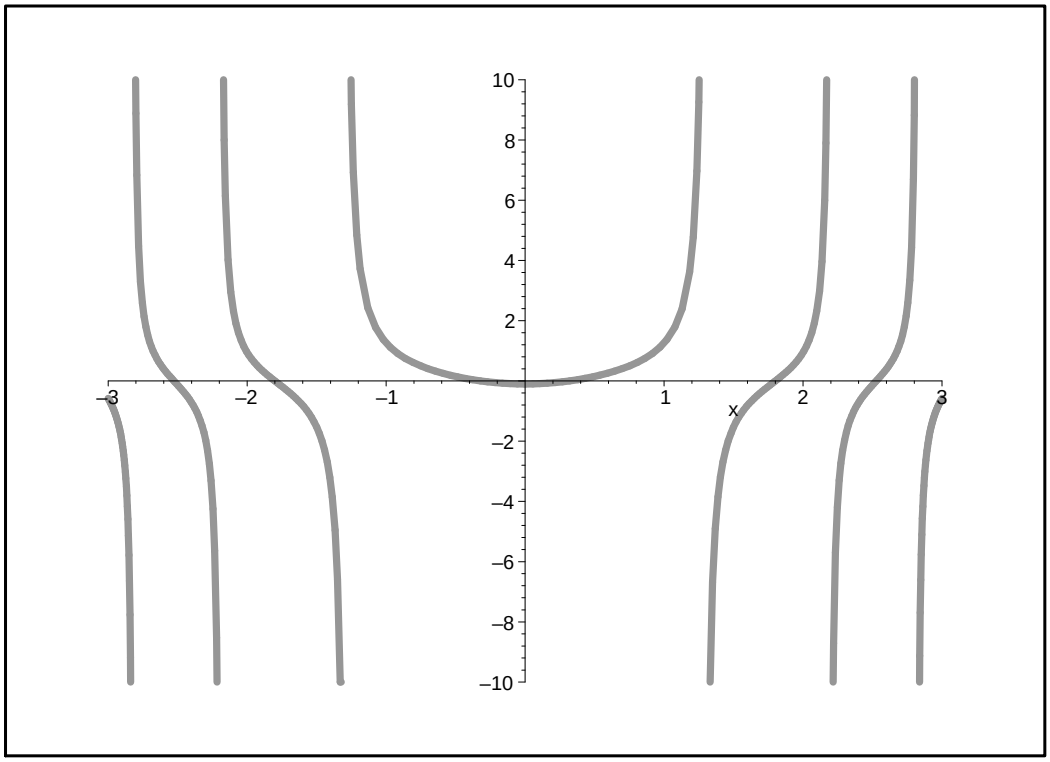
```
PLOT(CURVES([[0, 1.], [.5812574444444444e-1, .9966327743486623],
[.1162514888888889, .9866657952237375], [.1668263883333333,
.9729225498577577], [.2174012877777778, .9548696884268696],
[.2742780516666666, .9300349152292006], [.3311548155555555,
.9011740850329836], [.3884085955555555, .8689144857451291],
[.4456623755555555, .8342962829797348], [.5026440383333333,
.7983069319642314], [.5596257011111111, .7615096904776945],
[.6124549966666666, .7272192446912888], [.6652842922222222,
.6931913136571584], [.7199859677777778, .6585967516726312],
[.7746876433333333, .6249449483130620], [.8312601944444444,
.5913683249362686], [.8878327455555555, .5592068706963709],
[.9439163727777777, .5288268489514161], [1.,
.5000000000000000]],COLOUR(RGB,1.0,0,0)),AXESLABELS("x",‘‘),TITLE(‘Gra
ph von  y = f(x)‘),VIEW(0 .. 1.,DEFAULT))
```

```
> P; # Darstellung des Graphen
```

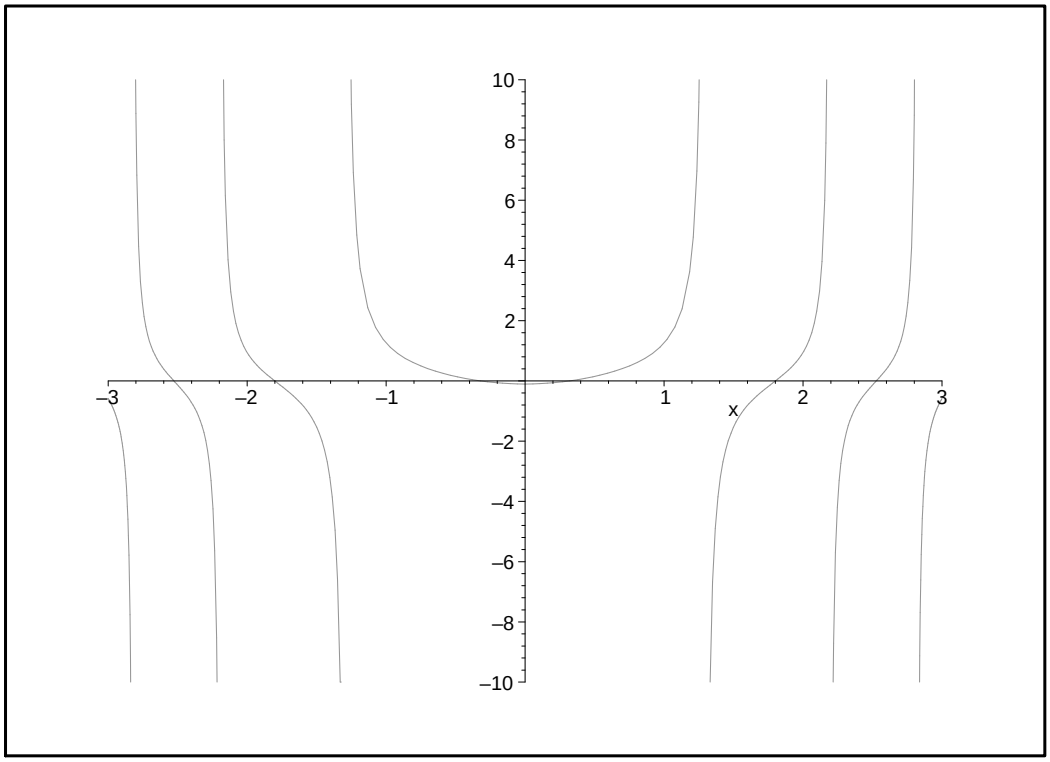


PLOT-Objekte als Datenstrukturen können manipuliert und gespeichert werden:

```
> step := plot( tan(x^2-1/10), x=-3..3, -10 .. 10,
> discont=true, thickness=8):
> step; # Darstellung
```




```
>  save step, ‘graph.m‘:
>  step := NULL: # Atomisieren der Variablen
>  read ‘graph.m‘: # Laden der Grafik
>  step; # Darstellung der Grafik
>  subs( COLOUR(RGB,0,1.00000000,0)=COLOUR(RGB,0.5,0,0.5), step
);
>  subs( THICKNESS(8)=THICKNESS(2), step );
```

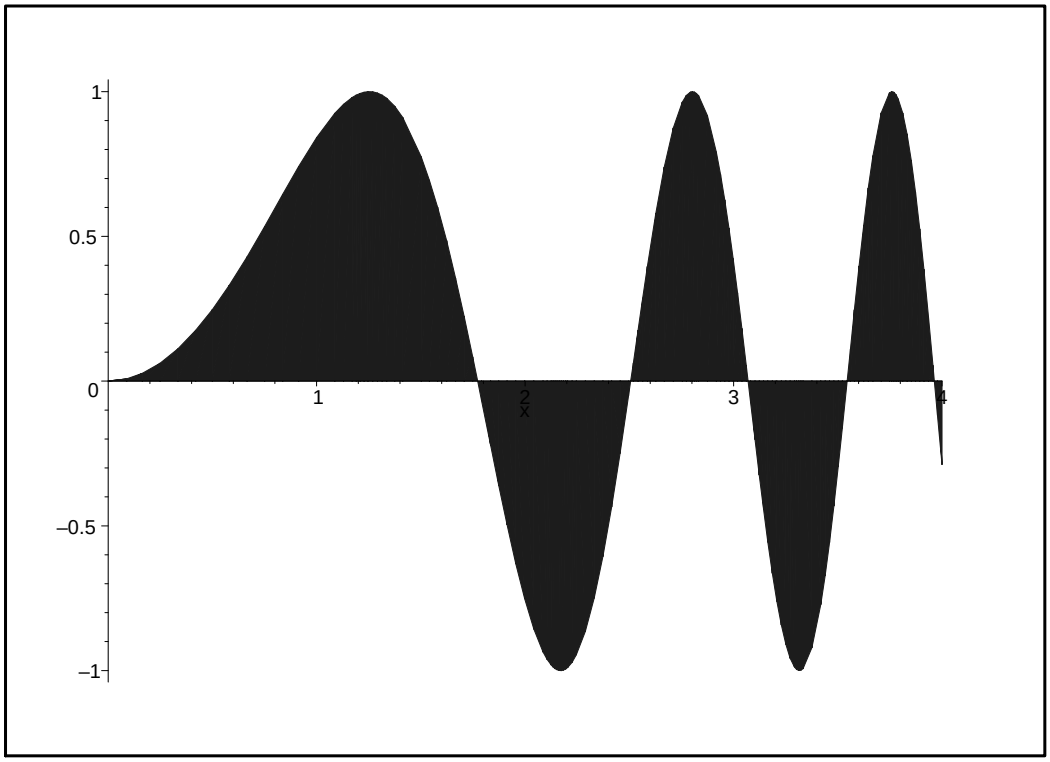


Spezielle 2D-Plots

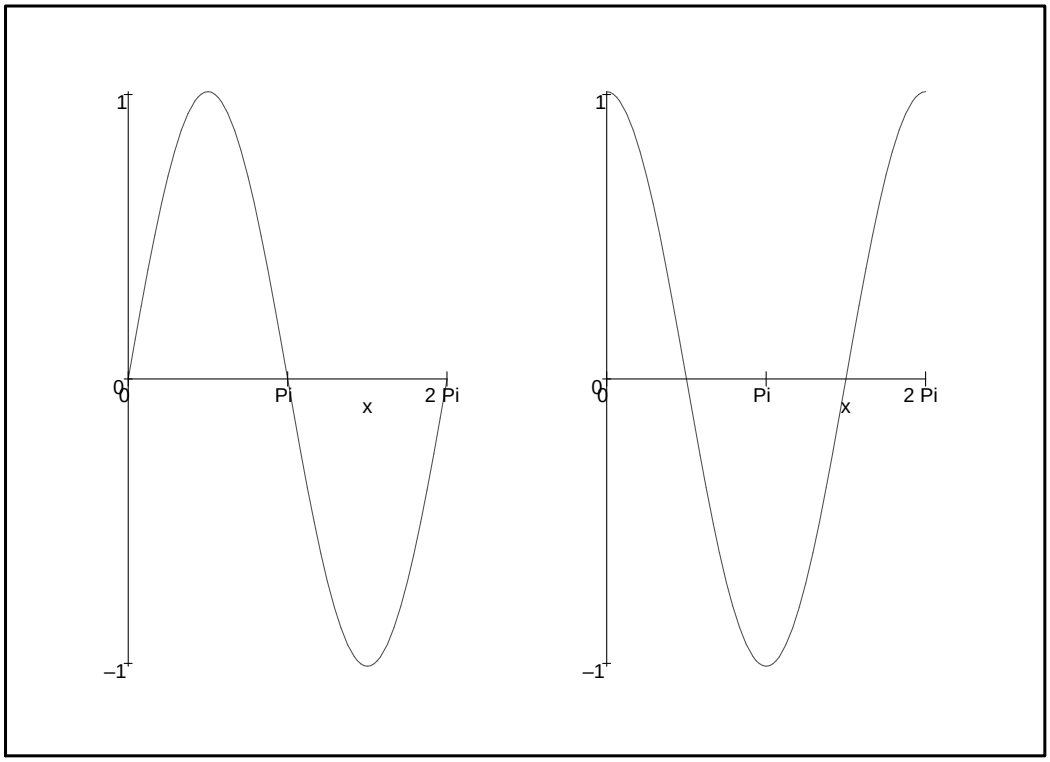
Plots lassen sich kombinieren, z.B.

- mehrere Kurven in 1 Bild sowie Füllung
- mehrere Bilder neben- bzw. bereinander (Grafik-Arrays).

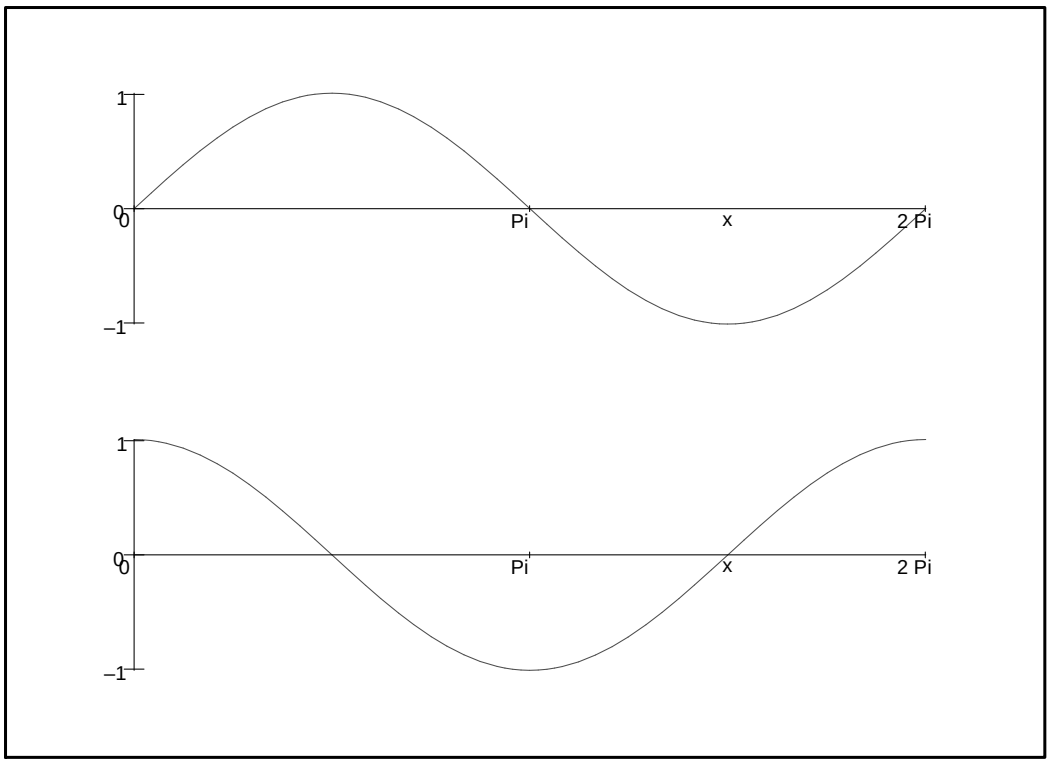
```
> plot( sin(x^2) , x=0..4, filled=true,color=blue );
```



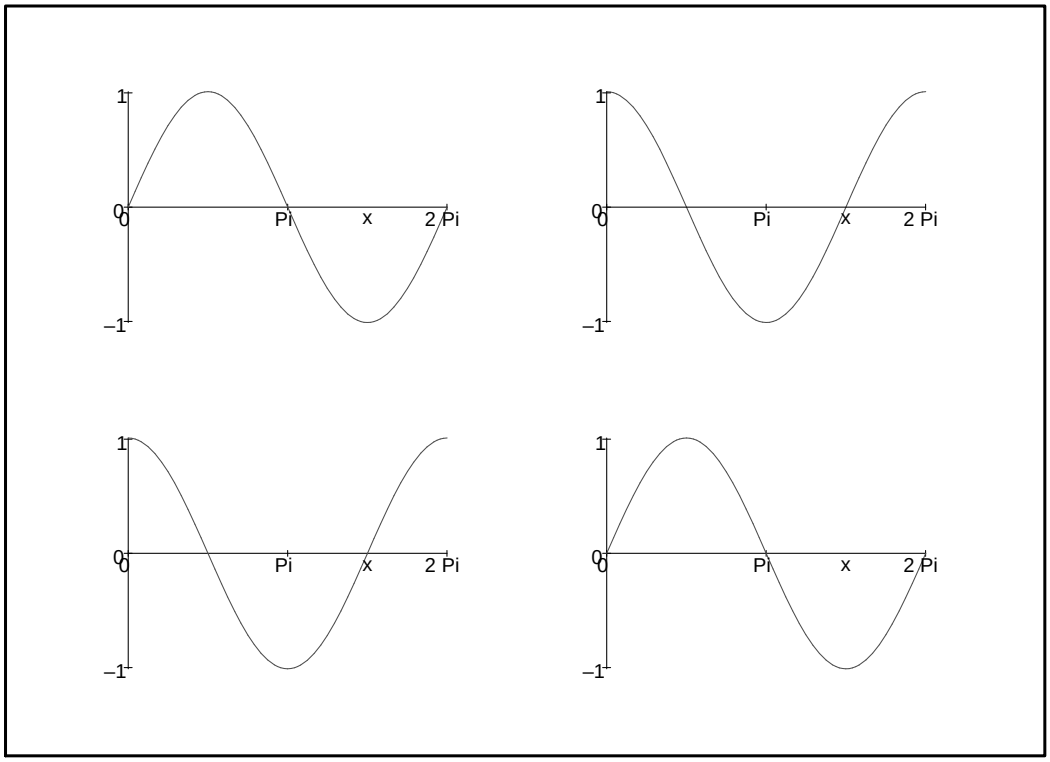

```
> ticks := [ [0=‘0‘, 3.14=‘Pi‘, 6.28=‘2 Pi‘],
> [-0.99=‘-1‘,0=‘0‘,0.99=‘1‘] ]:
> sine := plot( sin(x), x=0..2*Pi, tickmarks=ticks ):
> cosine := plot( cos(x), x=0..2*Pi, tickmarks=ticks ):
> display( array([sine,cosine]) ); # Zeile von Plots
```




```
>  display( array(1..2,1..1,[[sine],[cosine]]) );
```



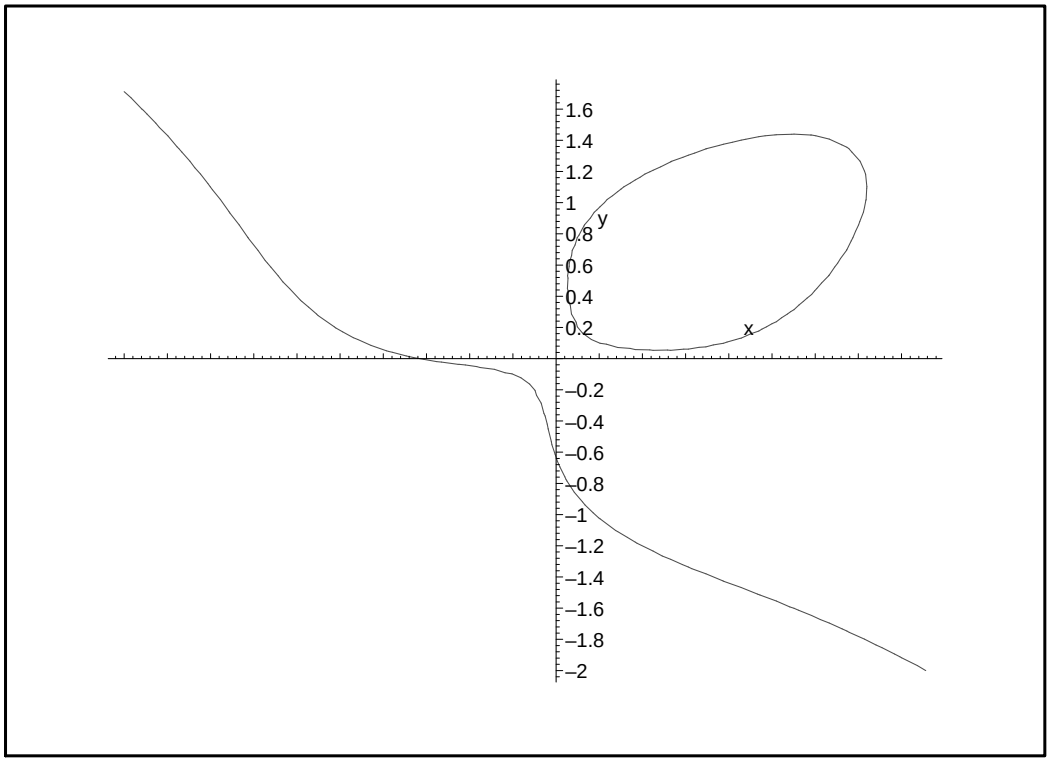

```
>  display( array(1..2,1..2,[[sine,cosine],[cosine,sine]]) );
```



Implizit definierte Kurven können mittels implicitplot dargestellt werden.

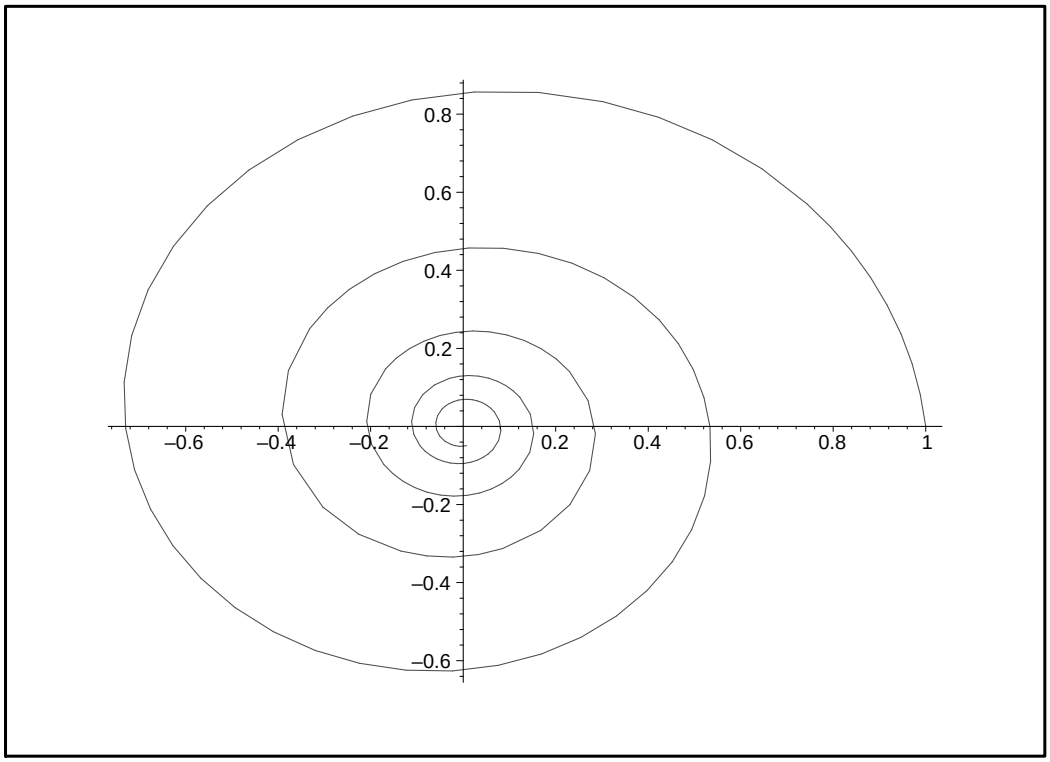
Beispiel: x^5 +y^5 -5xy +1/10 = 0

```
> restart;
> with(plots): with(plottools):
> implicitplot( x^5 + y^5 - 5*x*y + 1/10 = 0,
> x=-2..2, y=-2..2, grid=[50,50] );
```



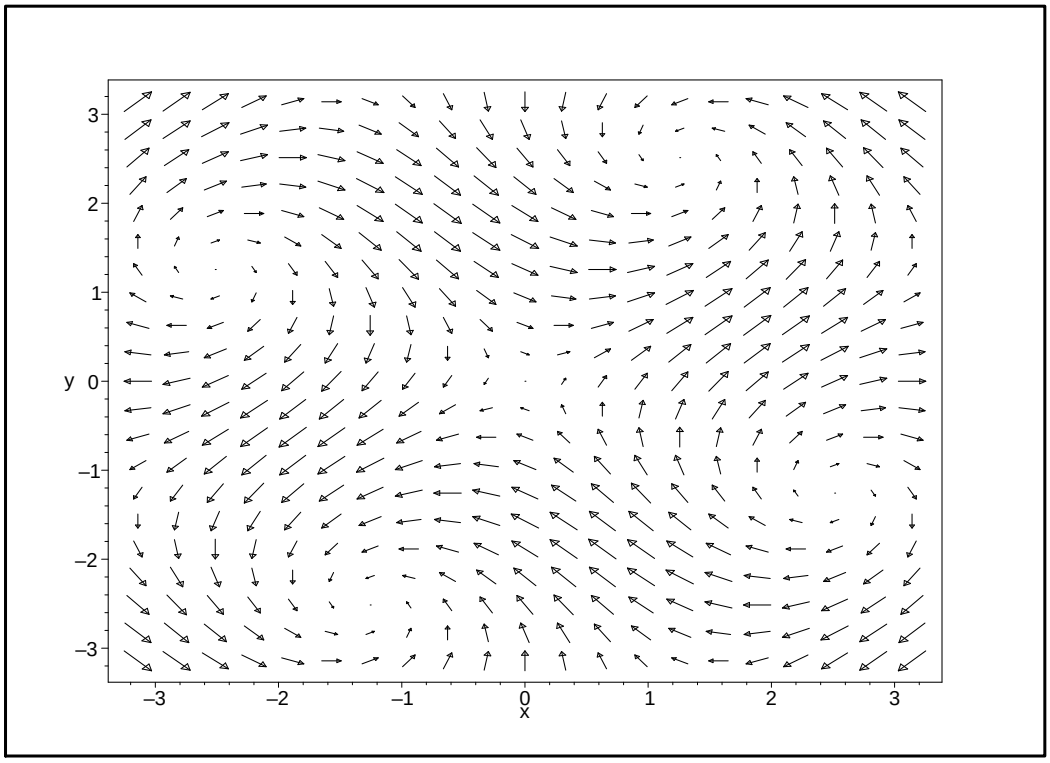
Komplexe Kurven , d.h. Abbildungen aus R in C, können mit complexplot dargestellt werden:

```
>  complexplot( exp( -t/10 + t*I ), t = 0..30 );
```



Feldplots und Gradientenplots 2-dimensionaler Vektorfelder sind möglich:

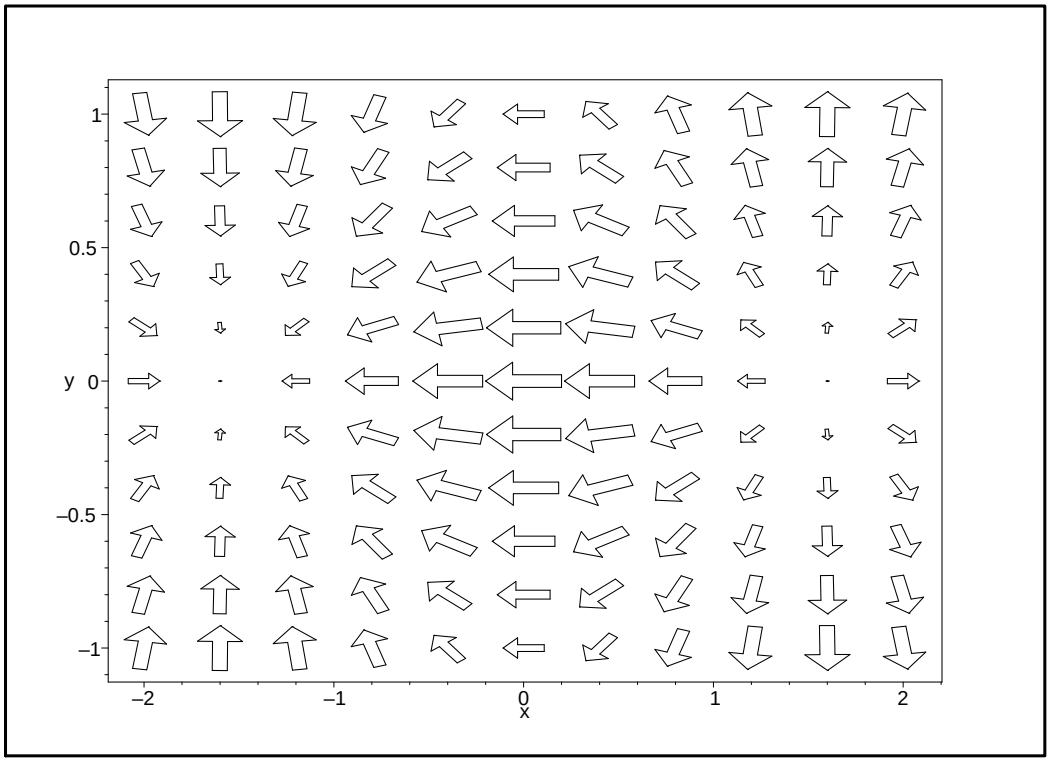
```
> restart;
> with(plots): with(plottools):
> fieldplot( [sin(x/2+y),sin(x-y/2)], x=-Pi..Pi, y=-Pi..Pi,
> arrows=SLIM, grid=[21,21], axes=box );
```




```
> phi := sin(x+y) + sin(x-y);
```

$$\phi := \sin(x+y) + \sin(x-y)$$

```
> gradplot( -phi, x=-2..2, y=-1..1,
> arrows=THICK, grid=[11,11], axes=box );
```



Konturplots und Dichteplots (z.B. von elektrostatischen Feldern) kann Maple leicht darstellen:

```
> contourplot( phi, x=-2..2, y=-1..1, color=black,
> numpoints=500, axes=box, contours=10 );
> contourplot( phi, x=-2..2, y=-1..1,
> numpoints=500, axes=box, filled=true,
> contours=[seq(i/4,i=-6..6)], coloring=[white,black] );
```

Frage: Was gilt es beim Gebrauch von plot zu beachten?

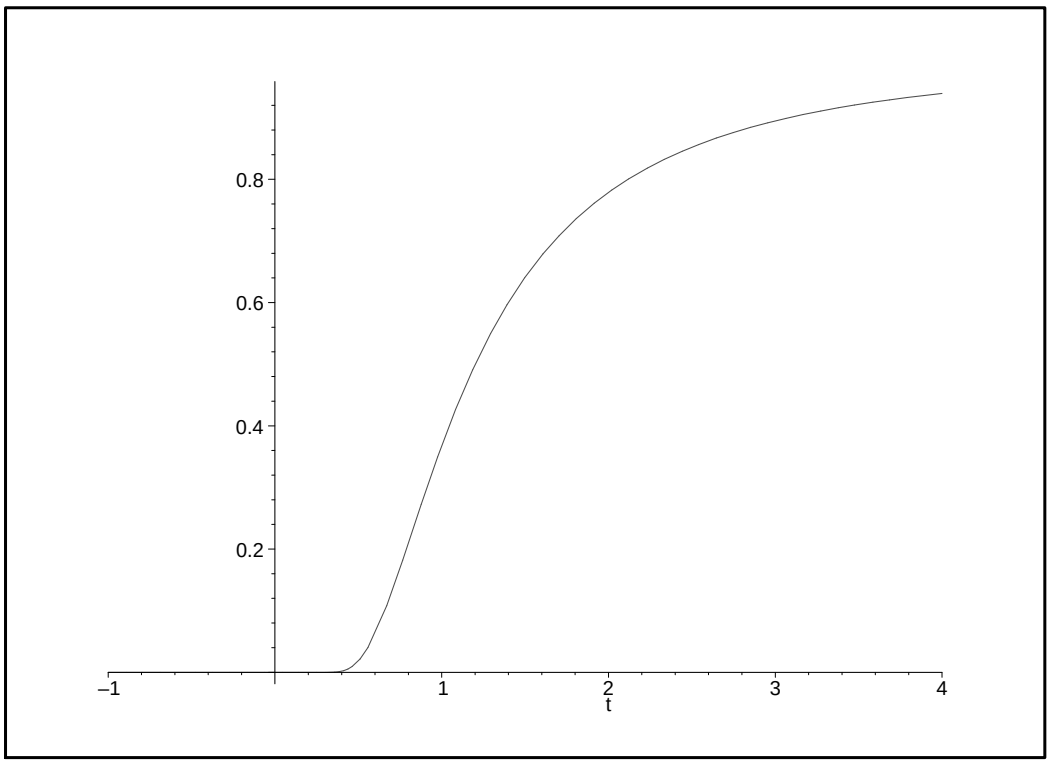
Wenn Maple ein Kommando der Form plot(f(x), x= a.. b); abarbeitet, wird zuerst f(x) ausgewertet - und damit ein symbolischer Ausdruck erzeugt. Danach werden die Zahlenwerte für x in f(x) eingesetzt und die Funktionswerte numerisch berechnet. Folge: Wenn f(x) einen Vergleich enthält, so kann dieser nicht zu true oder false ausgewertet werden!

```
> restart;
> f := t -> if t>0 then exp(-1/t^2) else 0 fi:
> plot( f(t), t=-1..4 );

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

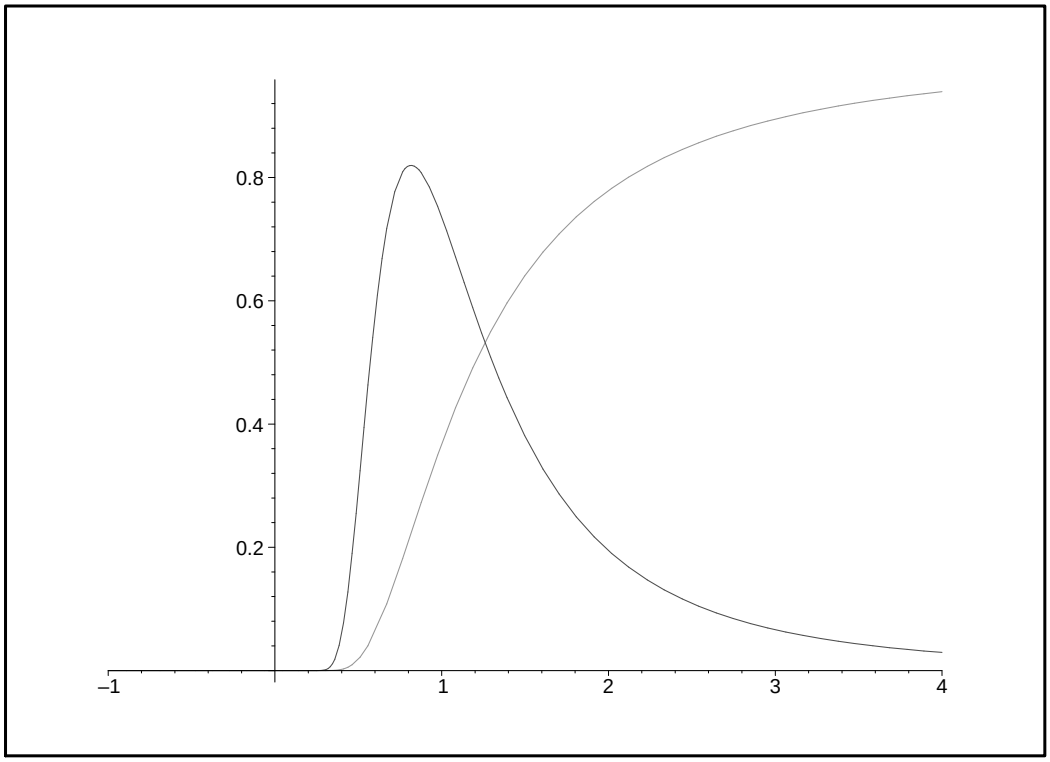
Ausweg 1: Verzögerte Auswertung des Ausdruckes f(x) mittels Apostroph:

```
> plot( 'f(t)', t=-1..4 );
```



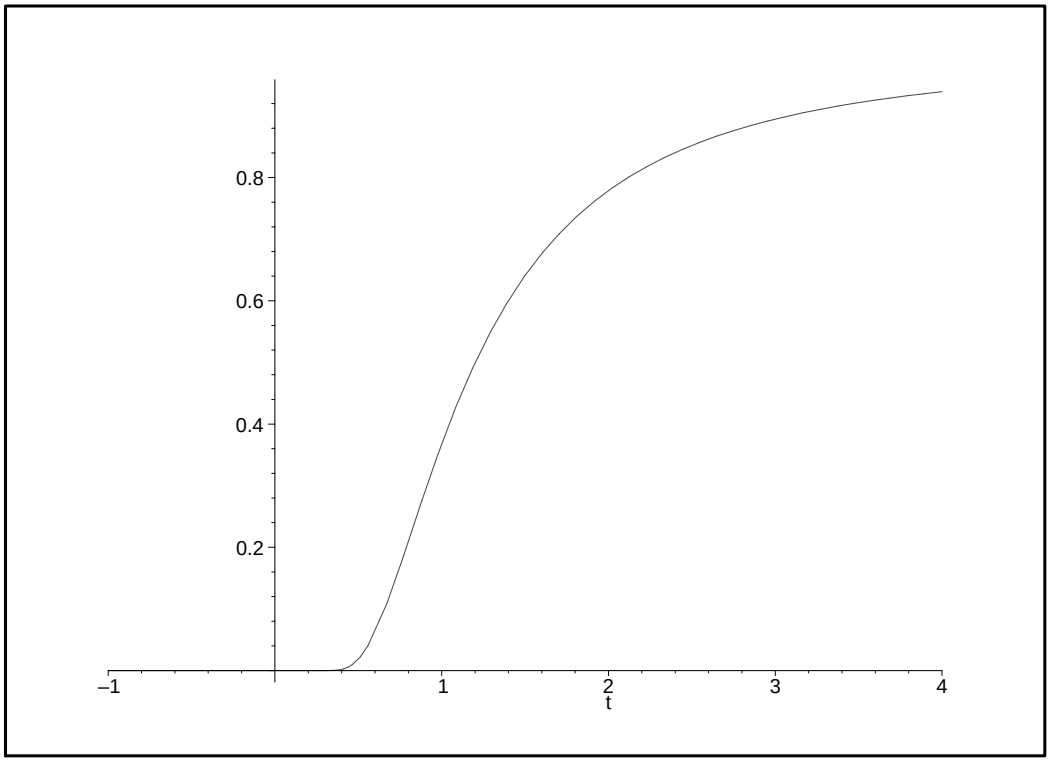
Ausweg 2: Funktionale Notation der Funktion f (und ggf . der Ableitung):

```
> plot( {f,D(f)}, -1..4 );
```



Ausweg 3: Hier bietet sich die piecewise-Funktion an, die diesen Nachteil nicht besitzt:

```
> f := t -> piecewise( t>0, exp(-1/t^2), 0 ):
> plot(f(t), t = -1 .. 4); # Kein Problem
```

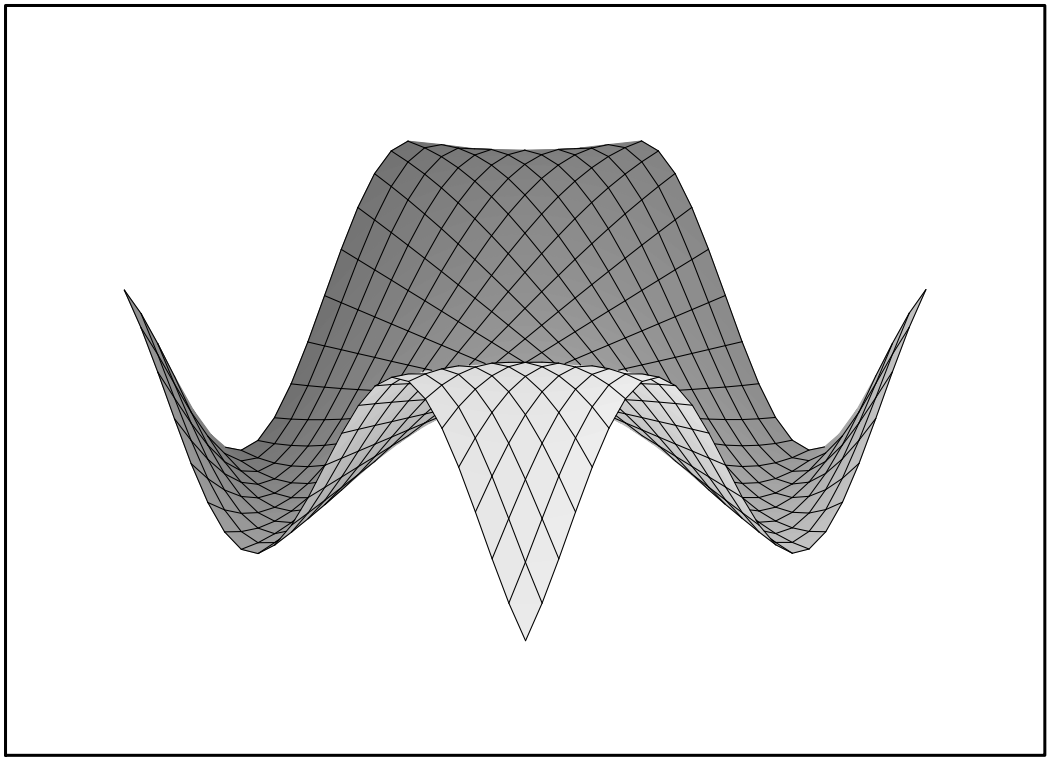

0.8
0.6
0.4
0.2
–1
1
2
3
4
t

5.5.2 Dreidimensionale Grafik

Einfache 3D-Plots, Optionen von plot3d

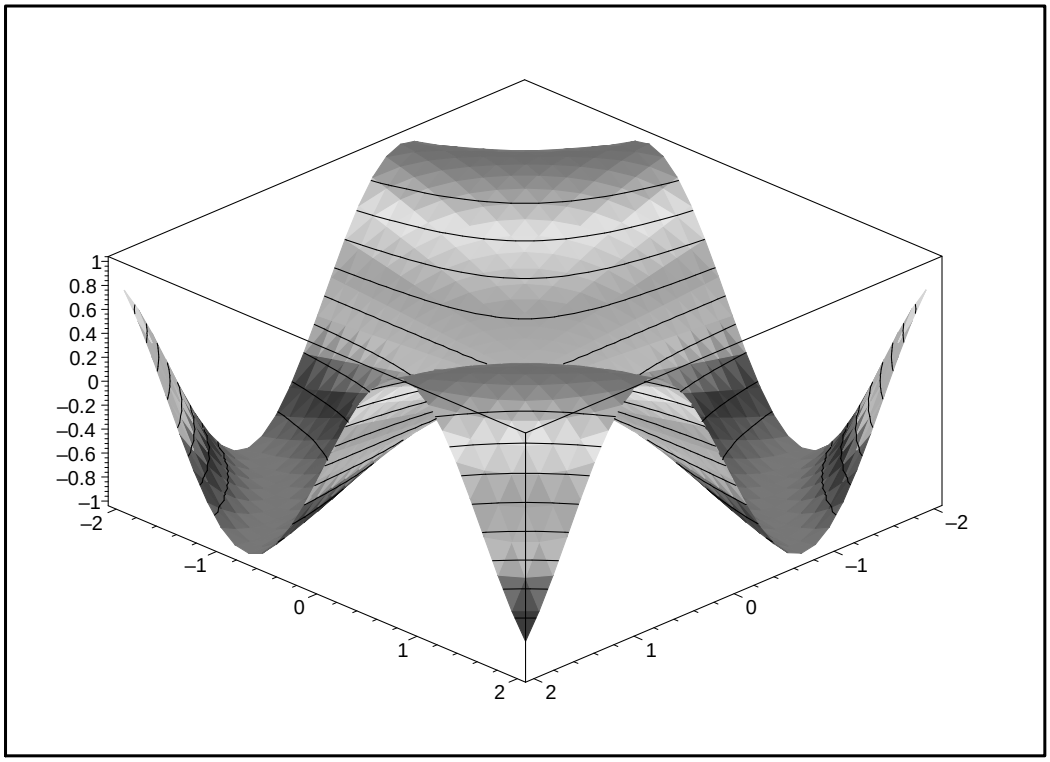
Das Kommando plot3d dient der Darstellung von Flächen, die durch z = f(x,y) über (x,y) parametrisiert sind. Die Bereiche für x und y sind anzugeben. Im einfachsten Fall werden keine Achsen gezeichnet, die Fläche wird in xyz-Farbe als Patches in einer standardisierten Orientierung dargestellt.

```
> restart;
> plot3d( sin(x*y), x=-2..2, y=-2..2 );
```



Alternativ dazu ist die funktionale Darstellung möglich:

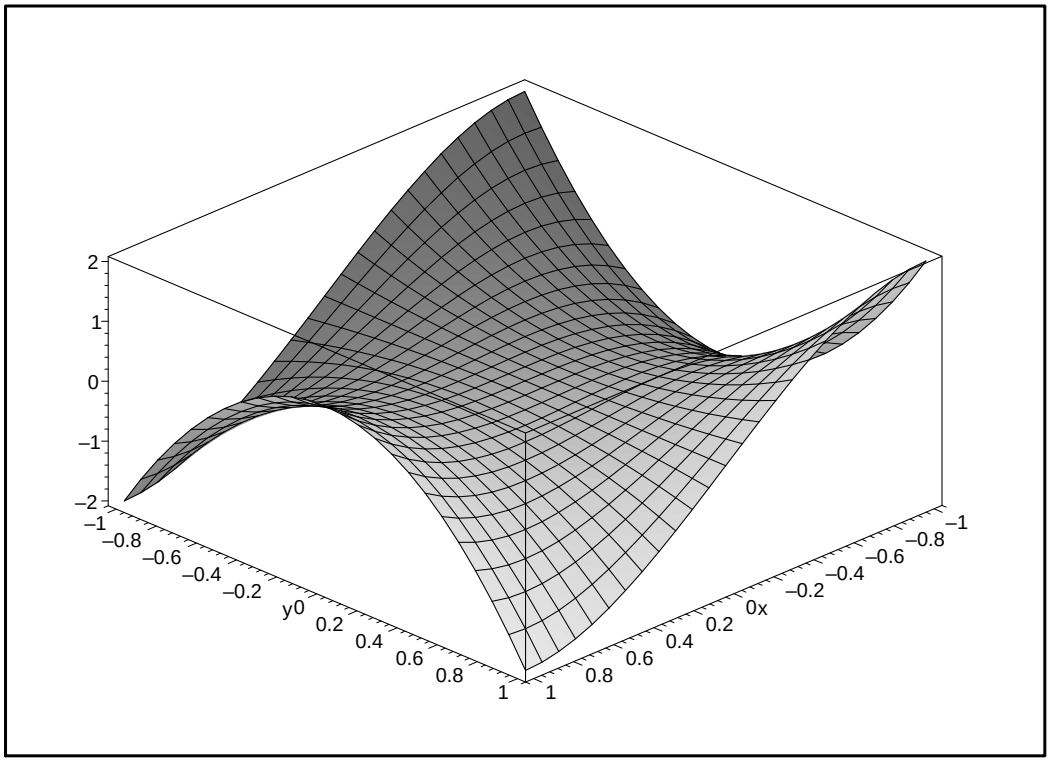
```
> f := (x,y) -> sin(x*y):
> plot3d( f, -2..2, -2..2, grid=[30,30], axes=box,
> scaling=unconstrained, style=patchcontour,
> shading=zhue );
```



Zahlreiche Optionen können per Menü bedient werden, z.B.

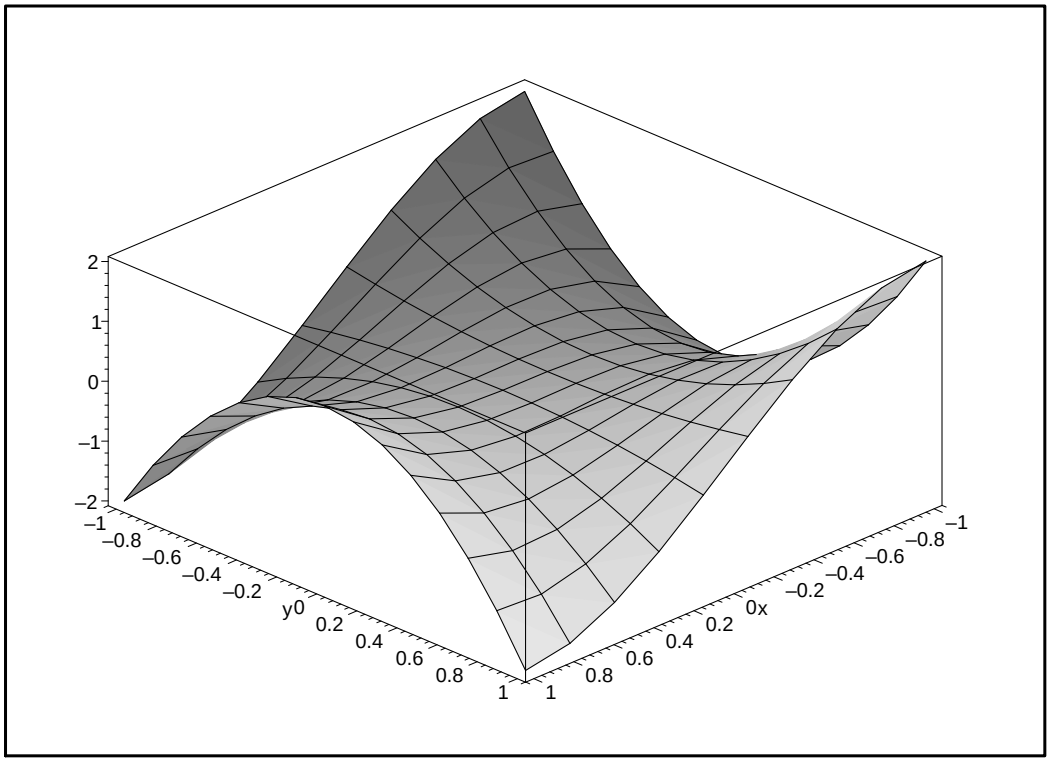
- Darstellungsstil (patches, wireframe,...)
- Schattierung
- Achsen (keine, Box, Rahmen, ...)
- Orientierung und Projektion (0..1)

```
> plot3d( x^3-3*x*y^2, x=-1..1, y=-1..1,
> style=patch, axes=box );
```



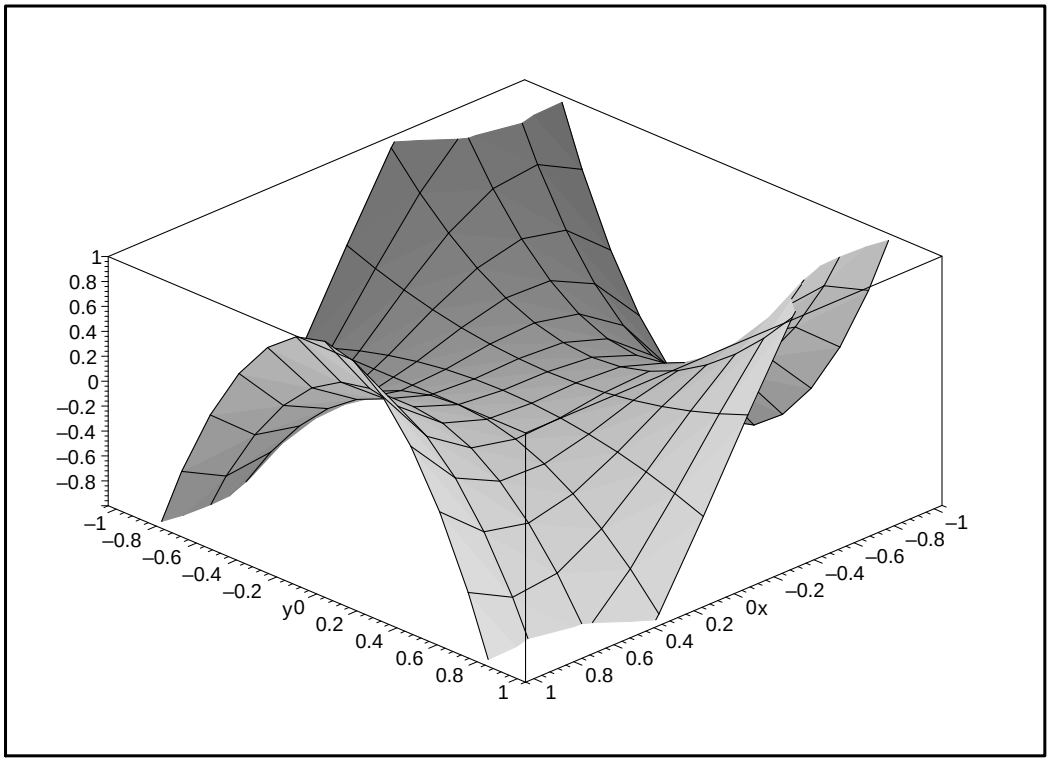
Standardmäßig werden 25 x 25 Gitterpunkte vorgegeben; mit grid kann das Gitter verfeinert oder vergröbert werden:

```
> plot3d( x^3-3*x*y^2, x=-1..1, y=-1..1,
> style=patch, axes=box , grid=[10,15] );
```



Schließlich kann der vertikale Bereich mittels view = a .. b (wie im Falle von 2D-Plots) beschränkt werden:

```
> plot3d( x^3-3*x*y^2, x=-1..1, y=-1..1,
> style=patch, axes=box , grid=[10,15], view= -1 .. 1 );
```



Weitere Optionen betreffen die Beleuchtung (Lichtquelle, Lichtfarben, ambientes Licht) und Konturendarstellung.

Hinweise:

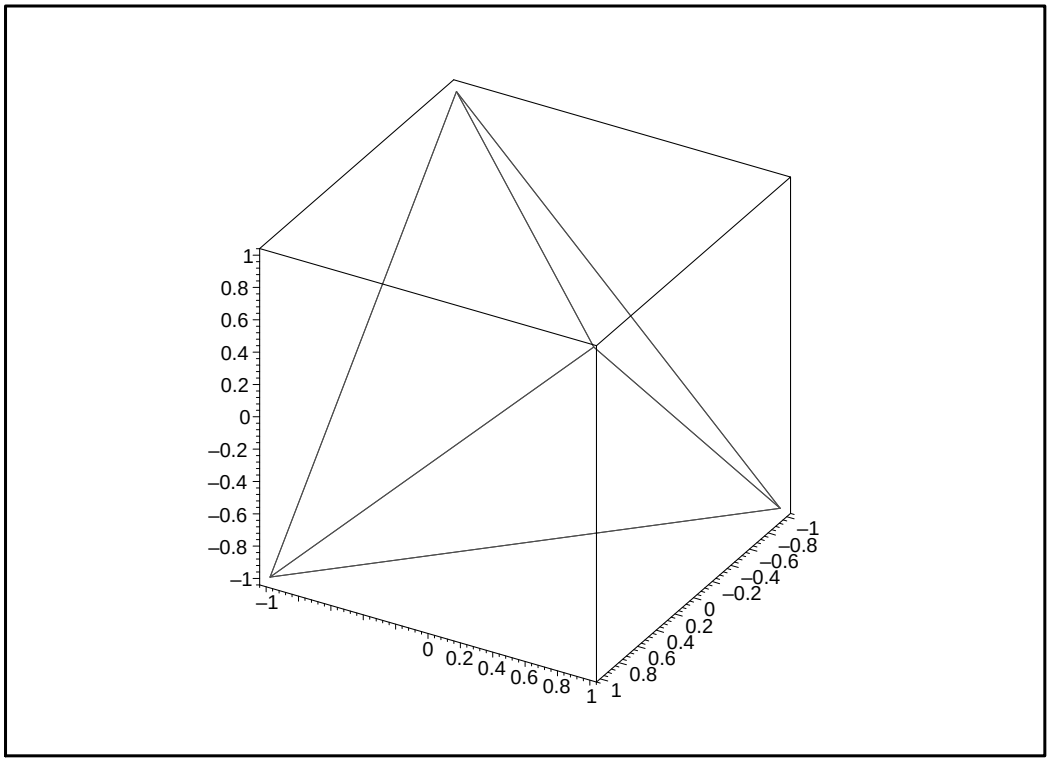
- Man studiere die Hilfe zu `plot3d[option]`
- Folgende Optionen können interaktiv per Menü verändert werden:
  style, linestyle, thickness, symbol, gridstyle, shading, light, axes, projektion.

Die Struktur von 3D-Grafikobjekten

```
> restart; with(plots):
```

Ein PLOT3D-Objekt ist ähnlich wie ein PLOT-Objekt aufgebaut. Nachfolgend wird ein Tetraeder gezeichnet:

```
> PLOT3D( POLYGONS( [ [1,1,1], [-1,-1,1], [-1,1,-1] ],
> [ [1,1,1], [-1,-1,1], [1,-1,-1] ],
> [ [-1,1,-1], [1,-1,-1], [-1,-1,1] ],
> [ [-1,1,-1], [1,-1,-1], [1,1,1] ] ),
> STYLE(LINE), COLOR(RGB,1,0,0), AXESSTYLE(BOX),
> ORIENTATION(30,60), SCALING(CONSTRAINED) );
```



Das externe Kommando hat folgende Gestalt. Zuerst wird das PLOT3D-Objekt P erzeugt. Bei Aufruf von P wird das Bild gezeichnet:

```
> P := polygonplot3d( [
> [ [1,1,1], [-1,-1,1], [-1,1,-1] ],
> [ [1,1,1], [-1,-1,1], [1,-1,-1] ],
> [ [-1,1,-1], [1,-1,-1], [-1,-1,1] ],
> [ [-1,1,-1], [1,-1,-1], [1,1,1] ] ],
> style=line, color=red, axes=box,
> orientation=[30,60], scaling=constrained ):
> lprint( P );
```

```
PLOT3D(POLYGONS([[1., 1., 1.], [-1., -1., 1.], [-1., 1., -1.]],[[1.,
1., 1.], [-1., -1., 1.], [1., -1., -1.]],[[-1., 1., -1.], [1., -1.,
-1.], [-1., -1., 1.]],[[-1., 1., -1.], [1., -1., -1.], [1., 1.,
1.]]),ORIENTATION(30.,60.),AXESSTYLE(BOX),STYLE(LINE),SCALING(CONSTRAI
NED),COLOUR(RGB,1.00000000,0,0))
```

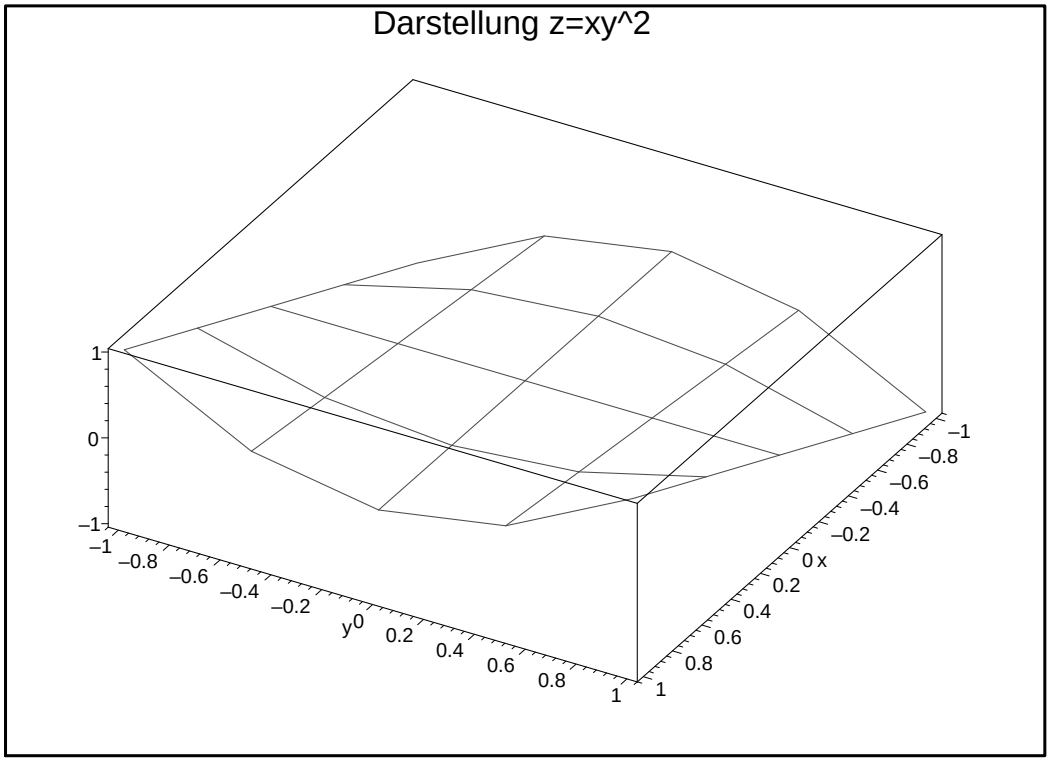
```
> P;
```

Funktionen in 2 Veränderlichen erzeugen ebenfalls ein PLOT3D-Objekt.

```
> P := plot3d( x*y^2 , x=-1..1, y=-1..1, grid=[5,5],
> axes=box, orientation=[30,30], style=line,
> color=red, title='Darstellung z=xy^2' ):
> lprint( P );
```

```
PLOT3D(GRID(-1. .. 1.,-1. ..
1.,hfarray(1..5,1..5,[[-1,-0.25,0,-0.25,-1],[-0.5,-0.125,0,-0.125,-0.5
],[0,0,0,0,0],[0.5,0.125,0,0.125,0.5],[1,0.25,0,0.25,1]]),COLOR(RGB,1.
00000000,0,0)),ORIENTATION(30.,30.),AXESLABELS(x,y,''),AXESSTYLE(BOX),
STYLE(LINE),TITLE('Darstellung z=xy^2'))
```

```
> P;
```

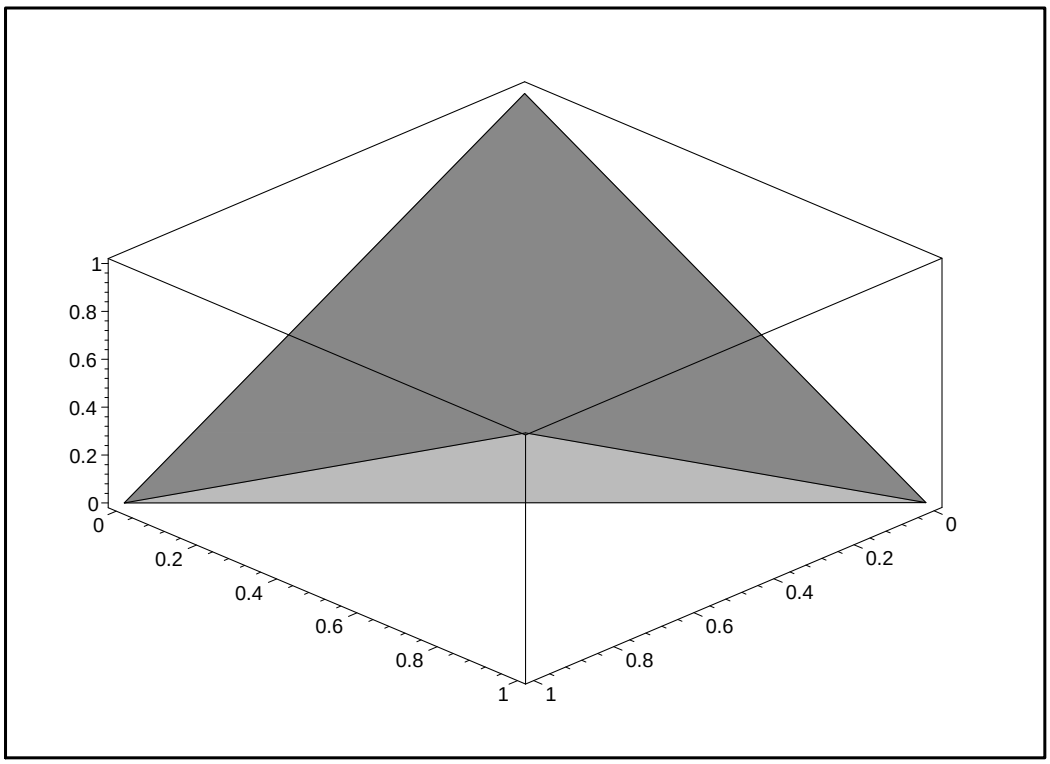


PLOT3D-Objekte können manipuliert werden (s.o.). Mit `convert` kann man sehen, wie Optionen in Grafik-Primitive umgewandelt werden.

```
> q :=PLOT3D( MESH([ [ [1,1,1],[0,1,0],[0,0,1] ],
                   [ [1,1,1],[1,0,0],[0,0,1] ] ] ),
                  STYLE(PATCH), AXES(BOX) );
```

$$q := \mathbf{PLOT3D}(\mathbf{MESH}([[[1,\,1,\,1],\,[0,\,1,\,0],\,[0,\,0,\,1]],\,[[1,\,1,\,1],\,[1,\,0,\,0],\,[0,\,0,\,1]]]),\ \mathbf{STYLE}(\boldsymbol{PATCH}),\ \mathbf{AXESSTYLE}(\boldsymbol{BOX}))$$

```
> q;
```




```
> nops(q); op(1,q); op(3,q);
```

$$3$$

$$\mathbf{MESH}([[[1, 1, 1], [0, 1, 0], [0, 0, 1]], [[1, 1, 1], [1, 0, 0], [0, 0, 1]]])$$

$$\mathbf{AXES}(\boldsymbol{BOX})$$

```
> subs(AXES(BOX)=AXES(NONE), q);
> convert( [ color=red, orientation=[20,60] ],
> PLOT3Doptions );
```

$$[\mathbf{ORIENTATION}(20., 60.), \mathbf{COLOUR}(\boldsymbol{RGB}, 1.00000000, 0, 0)]$$

Spezielle 3D-Plots

Die Maple-Pakete plots und plottools sollten stets geladen werden, da sie zahlreiche Funktionen für Darstellungen in 3 Dimensionen bereitstellen.

```
> restart; with(plots): with(plottools):
```

Parametrische Kurvendarstellungen x=f(t), y=g(t), z=h(t) , t=a .. b , erhält man mittels `spacecurve`.

Beispiel: Kurven auf einem Torus

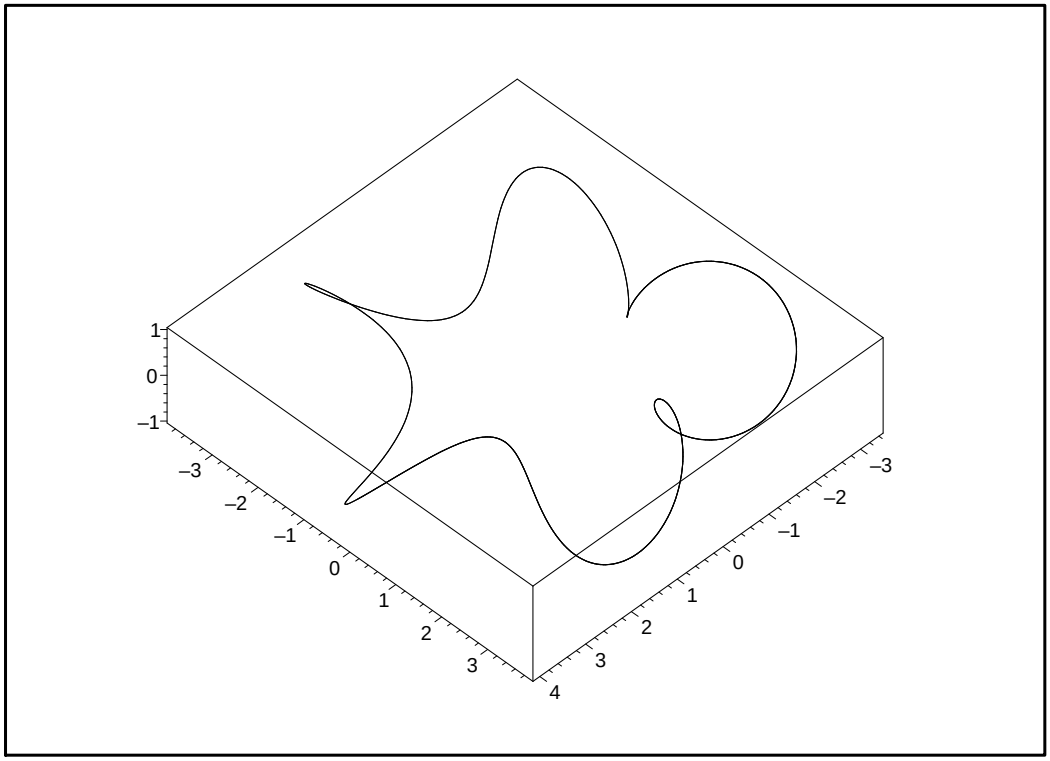
Wir definieren die Parameterdarstellung eines Ringes (Torus) durch

```
> torus := (R,r,theta,t) -> [(R+r*cos(theta))*cos(t),
          (R+r*cos(theta))*sin(t), r*sin(theta)]:
```

Für ein periodisches Verhalten 5:1 setzt man 2 Basisfrequenzen an:

```
> omega1 := 5.0: omega2 := 1.0: # Basisfrequenzen

> kurve := spacecurve( torus(3,1,omega1*t,omega2*t),
          t=0 .. 5*Pi, numpoints=500, color=black,
          axes=BOXED, scaling=constrained):
  display(kurve);
```



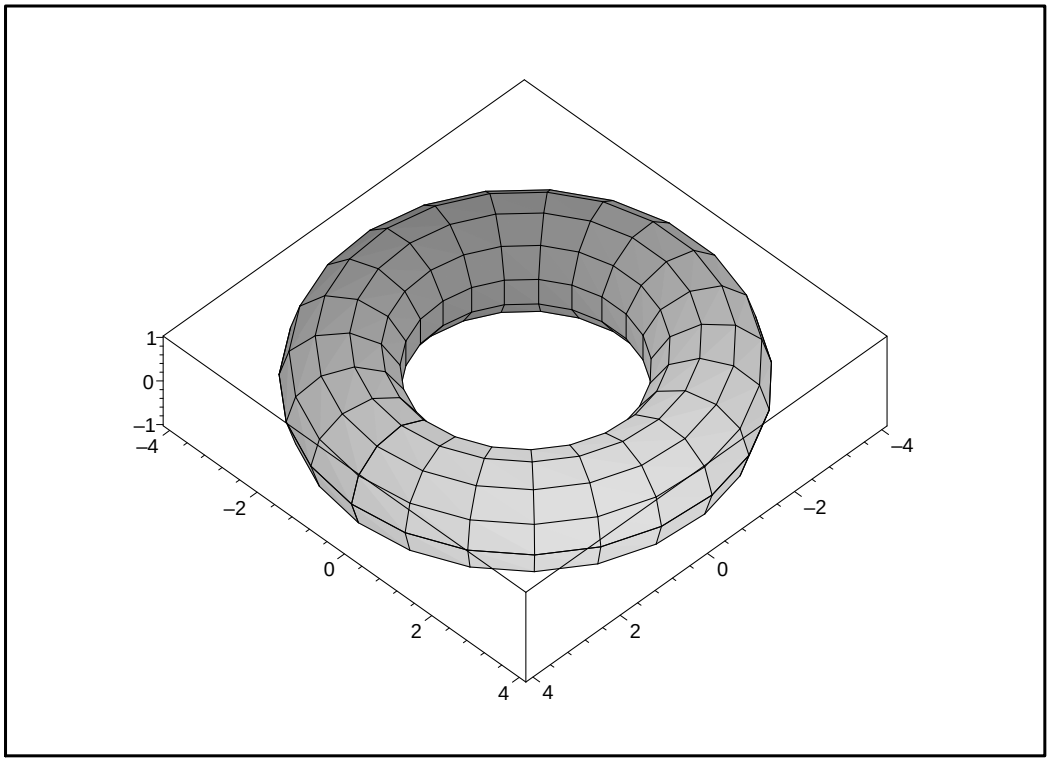
Flächen mit Parameterdarstellung x=f(s,t), y=g(s.t), z=h(s,t) werden mittels des Kommandos

plot3d ([f(s,t),g(s,t),h(s,t)], s=a .. b, t=c .. d, Optionen);

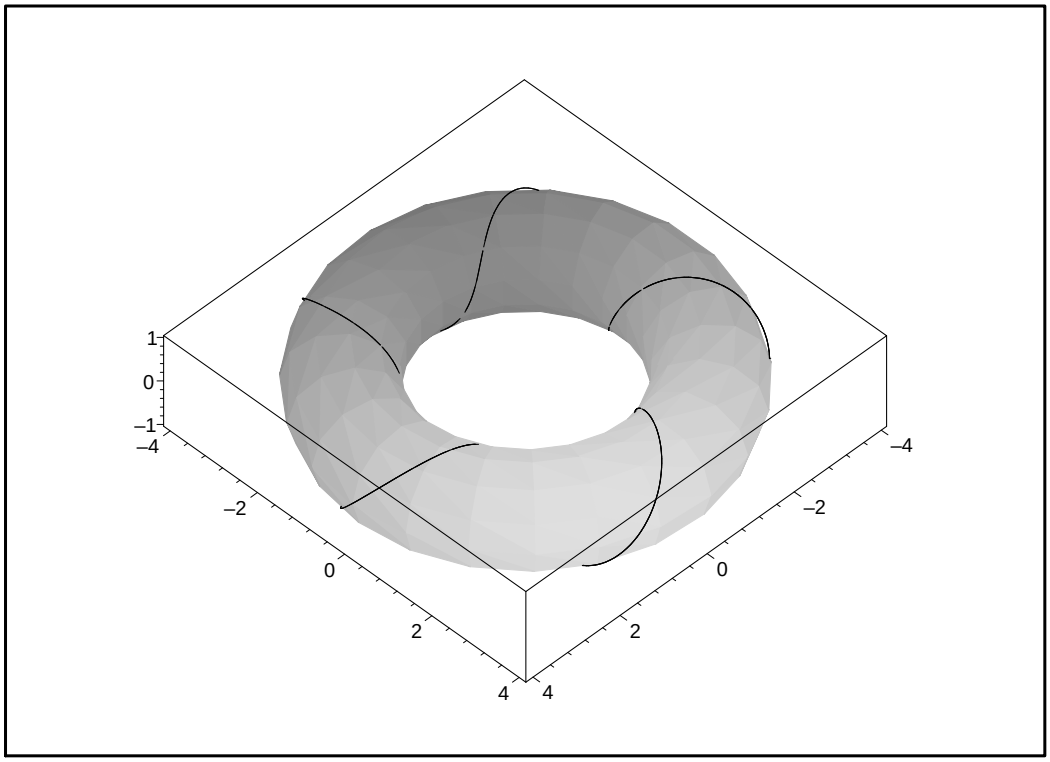
dargestellt.

Beispiel: Darstellung eines Torus

```
> tor := plot3d(torus(3.0,1,theta,t), t=0..2*Pi,
         theta=0..2*Pi, grid=[24,12],
         scaling = constrained,
         style=PATCH, axes=BOXED):
  display(tor);
```




```
> tor := plot3d(torus(3.0,1,theta,t), t=0..2*Pi,
         theta=0..2*Pi, grid=[24,12],
         scaling = constrained, style=PATCHNOGRID):
  display(tor,kurve);
```

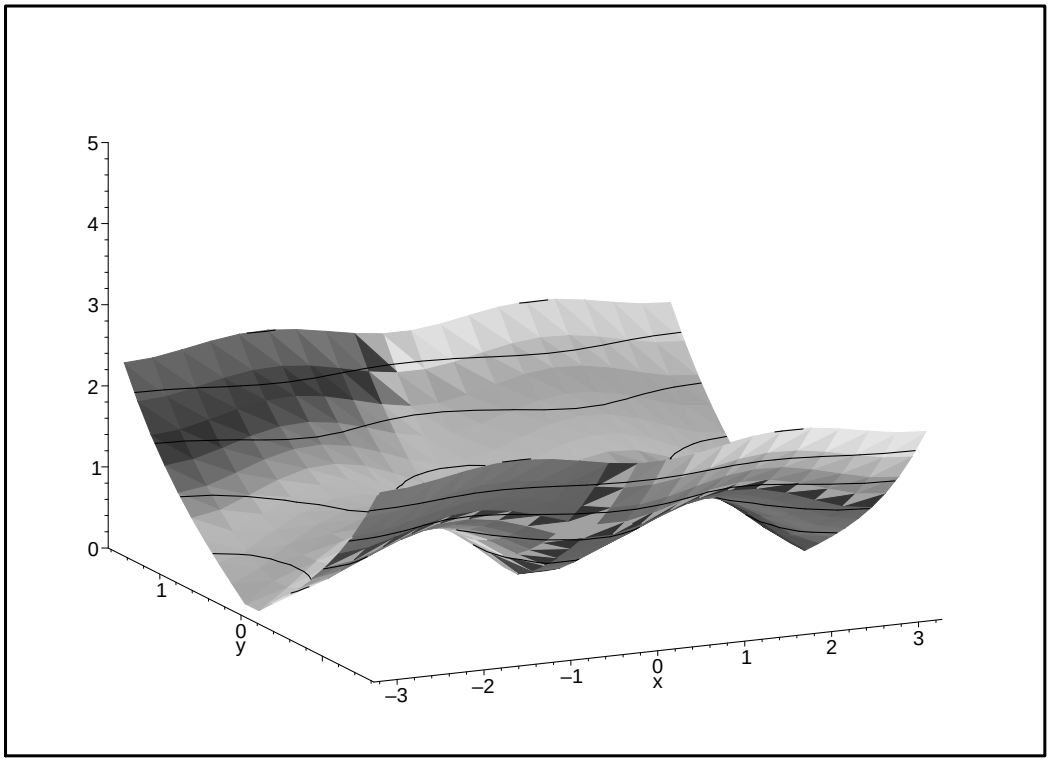


Unter den weiteren speziellen 3D-Plots sind besonders bedeutsam

- die Darstellung implizit definierter Flächen (implicitplot3d)
- die Darstellung komplexwertiger Funktionen (complexplot3d).

Beispiel: Darstellung der komplexen Sinusfunktion sin(z), z komplex

```
> restart; with(plots): with(plottools):
> complexplot3d( sin(z), z= -Pi-Pi/2*I .. Pi+Pi/2*I,
> view=0..5, grid=[20,20], orientation=[-115,70],
> axes=frame, style=patchcontour, thickness=2 );
```



Weitere spezielle Plots dienen der Darstellung von Polytopen und Diagrammen etc.

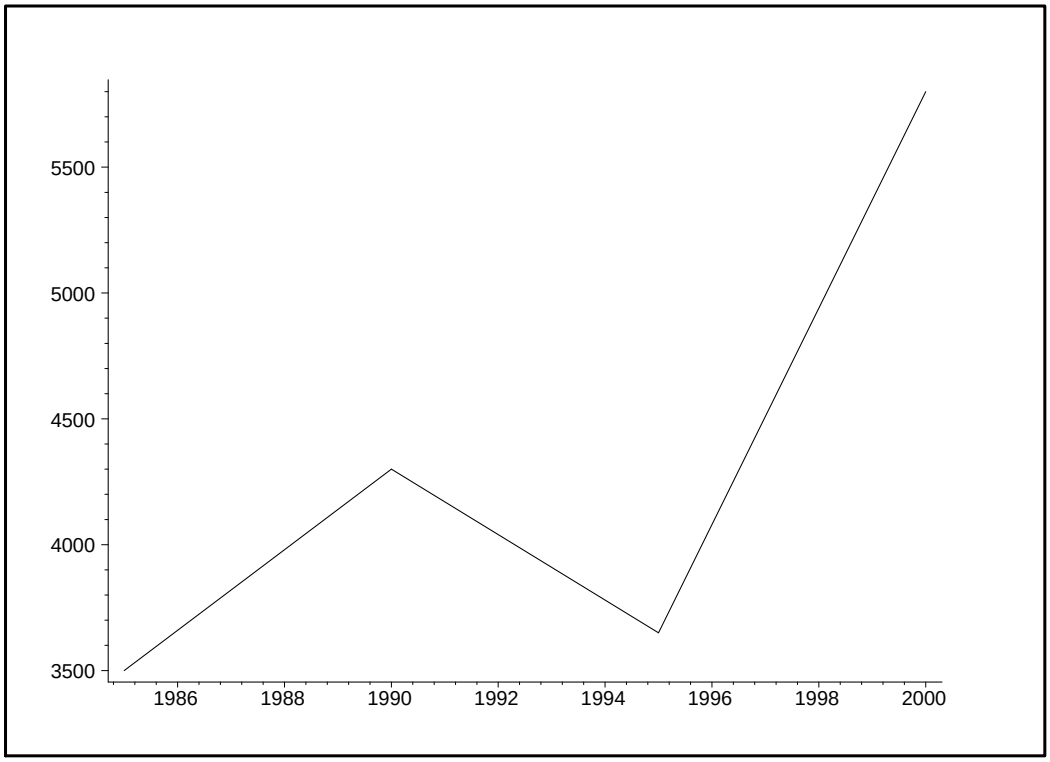
5.5.3 Datenplot und Animation

Sind die darzustellenden Daten nicht als Ausdrücke oder Funktionen, sondern in Form von Listen [x1,y1], [x2,y2],..., [xn,yn] gegeben, so nutzt man die listplot - Kommandos (oder im Falle spezieller Statistik die Kommandos des Pakets stats).

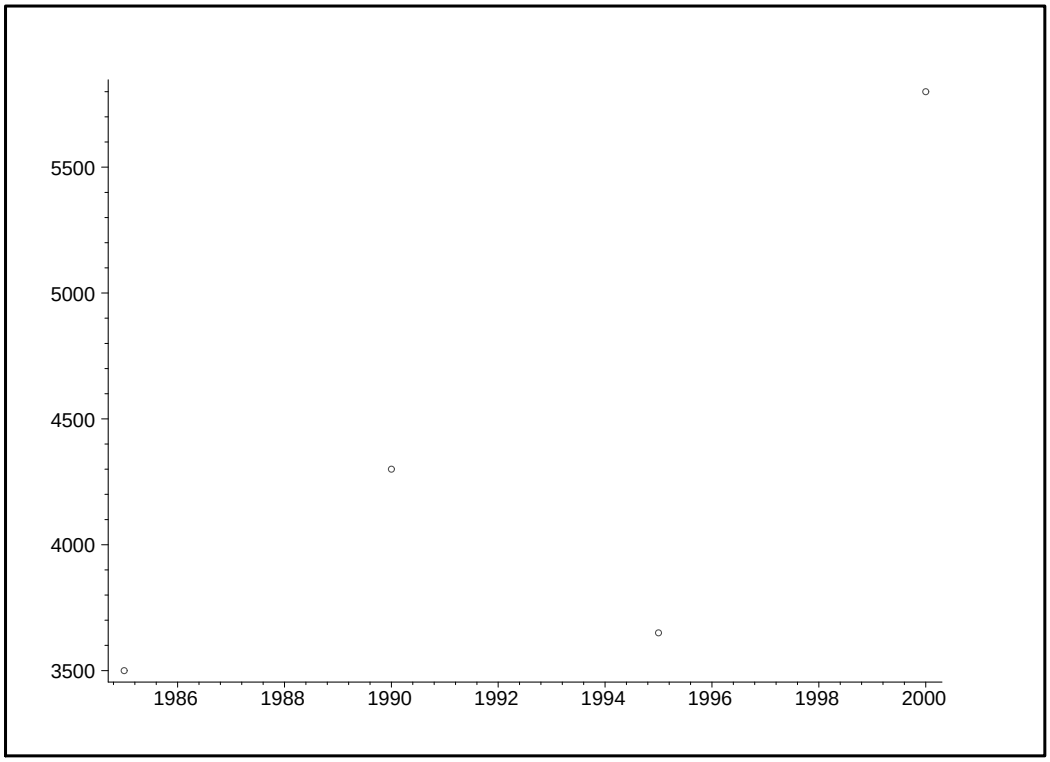
Kommando	Wirkung wie	Grafikobjekt
listplot	plot	2D-plot einer Datenliste
listplot3d	plot3d	3D-plot einer Datenliste
listcontplot	contourplot	2D-Konturplot eines Datenfeldes
listcontplot3d	contourplot3d	3D-Konturplot eines Datenfeldes
listdensityplot	densityplot	Dichteplot eines Datenfeldes

Beispiel 1: Anwendung von listplot zur Studentenstatistik einer Universität

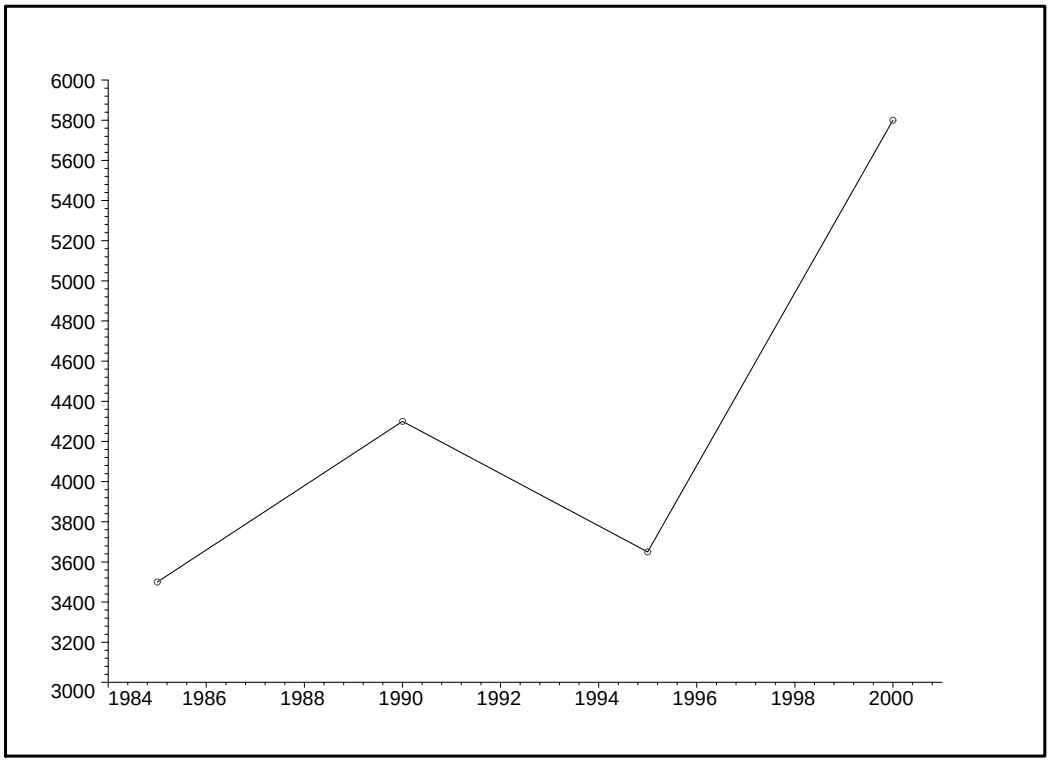
```
> restart; with( plots ):
> studenten[insgesamt] :=
> [ [1985,3500], [1990,4300], [1995,3650], [2000,5800] ]:
> studenten[ingenieure] :=
> [ [1985,2960], [1990,2300], [1995,1900], [2000,2440] ]:
> studenten[informatik] :=
> [ [1985,0], [1990,560], [1995,830], [2000,1100] ]:
> listplot( studenten[insgesamt] );
```




```
> listplot( studenten[insgesamt], style=point, symbol=circle,
> color=red );
```



```
> display( {%,%%}, view=[1984..2001,3000..6000] );
```

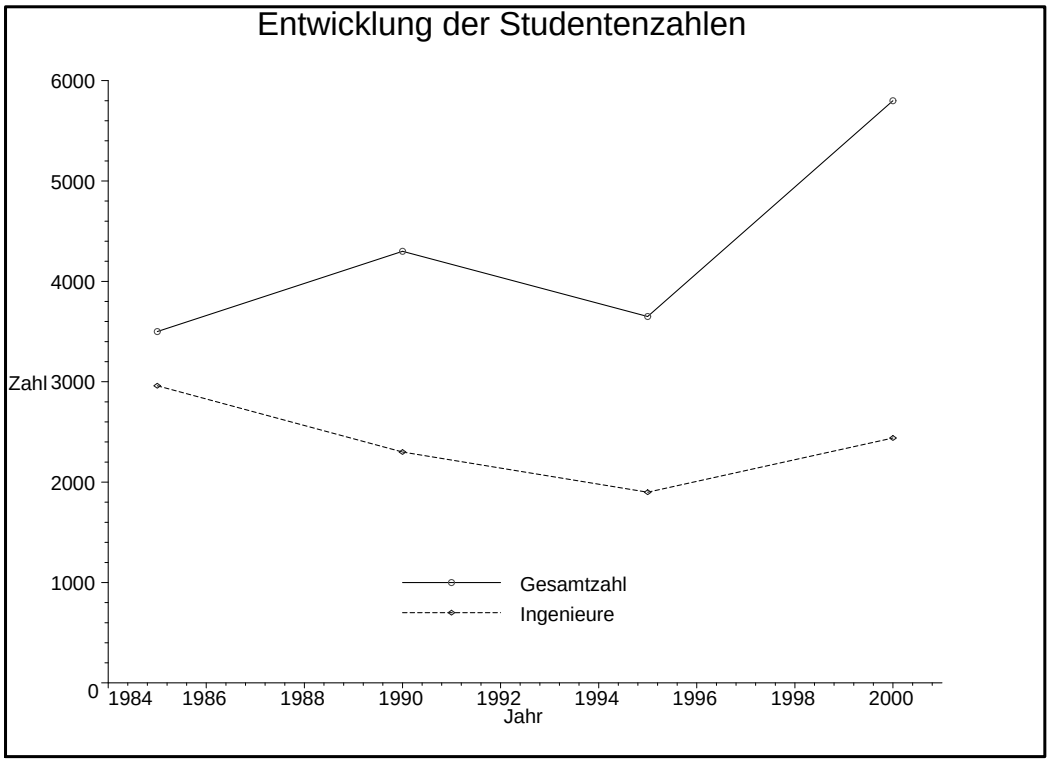


Nachfolgend werden erstellt:

- Kurve und Punkte für Gesamt-Studentenzahl
- Kurve und Punkte für Ingenieur-Studiengänge
- Überschrift und Legende

```
> bp := listplot( studenten[insgesamt], style=point,
> symbol=circle, color=red ):
> bl := listplot( studenten[insgesamt] ):
> sp := listplot( studenten[ingenieure], style=point,
> symbol=diamond, color=blue ):
> sl := listplot( studenten[ingenieure], linestyle=3 ):
> with( plottools, [line,point] ):
> bL := line( [1990,1000], [1992,1000] ):
> bP := point( [1991,1000], symbol=circle, color=red ):
> bT := textplot( [ 1992.4, 1000, Gesamtzahl ], align=RIGHT ):
> sL := line( [1990,700], [1992,700], linestyle=3 ):
> sP := point( [1991,700], symbol=diamond, color=blue ):
> sT := textplot( [ 1992.4, 700, ‘Ingenieure‘ ],
> align=RIGHT ):
```

```
> display( {bp,bl,sp,sl,bL,bP,bT,sL,sP,sT},
> labels=[Jahr,Zahl],
> title=‘Entwicklung der Studentenzahlen‘,
> view=[1984..2001,0000..6000] );
```



Nachfolgend wird ein Histogramm zu den gegebenen Daten erstellt. Zuerst werden die Daten der „Meßreihen“ zeilenweise in eine Matrix umgespeichert.

```
> print(studenten);
```

table([
$\boldsymbol{informatik} = [[1985, 0], [1990, 560], [1995, 830], [2000, 1100]]$
$\boldsymbol{ingenieure} = [[1985, 2960], [1990, 2300], [1995, 1900], [2000, 2440]]$
$\boldsymbol{insgesamt} = [[1985, 3500], [1990, 4300], [1995, 3650], [2000, 5800]]$
])

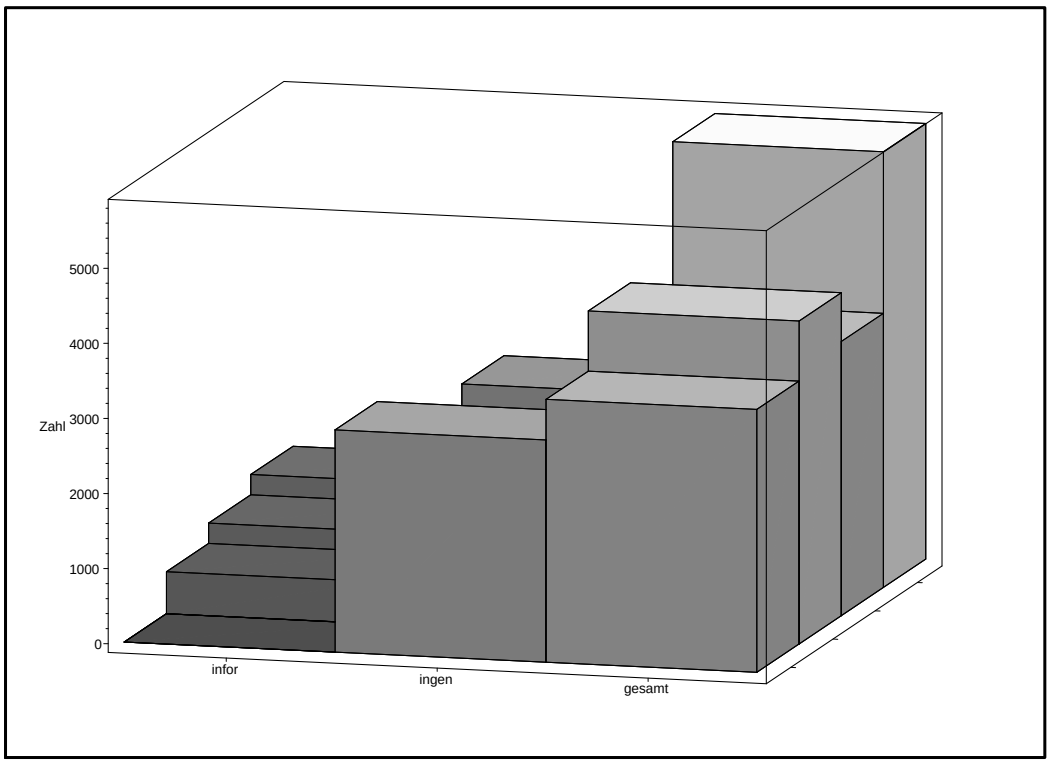
```
> M := map( d -> map2(op,2,op(d)), [entries(studenten)] ):
> setattribute( M, matrix );
```

$$\begin{bmatrix} 0 & 560 & 830 & 1100 \\ 2960 & 2300 & 1900 & 2440 \\ 3500 & 4300 & 3650 & 5800 \end{bmatrix}$$

```
> indices( studenten );
```

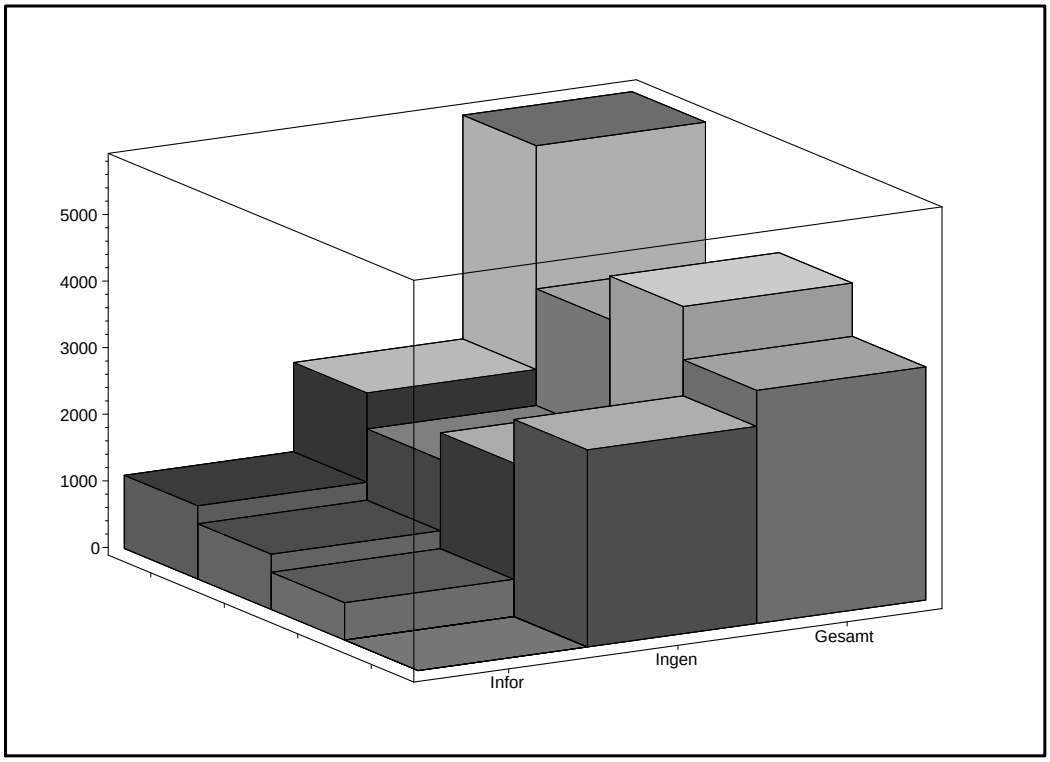
$[\boldsymbol{informatik}], [\boldsymbol{ingenieure}], [\boldsymbol{insgesamt}]$

```
> matrixplot( M, heights=histogram, style=patch,
> shading=zgrayscale, orientation=[-75,75], axes=box,
> tickmarks=[[1.5=infor,2.5=ingen,3.5=gesamt],
> [1.5=1985,2.5=1990,3.5=1995,4.5=2000],5],
> labels=[‘‘,‘‘,Zahl], font=[HELVETICA,DEFAULT,8]
> );
```



Drehung, Einfärben und Beleuchtung verbessern die Anschaulichkeit:

```
> matrixplot( M, heights=histogram, style=patch,
> shading=zhue, orientation=[-120,70], axes=box,
> tickmarks=[[1.5=Infor,2.5=Ingen,3.5=Gesamt],
> [1.5=1985,2.5=1990,3.5=1995,4.5=2000],5],
> labels=[‘‘,‘‘, ‘‘], font=[HELVETICA,DEFAULT,10]
> ); # Aenderung von Orientierung, Farbdarstellungen (xyz) und
# Beleuchtung leicht moeglich!
```



Beispiel 2:

Einlesen einer Meßreihe aus der Textdatei `kennlin.dat`, grafische Darstellung der Meßpunkte und danach Ausgleich nach der Methode der kleinsten Quadrate (curve fitting)

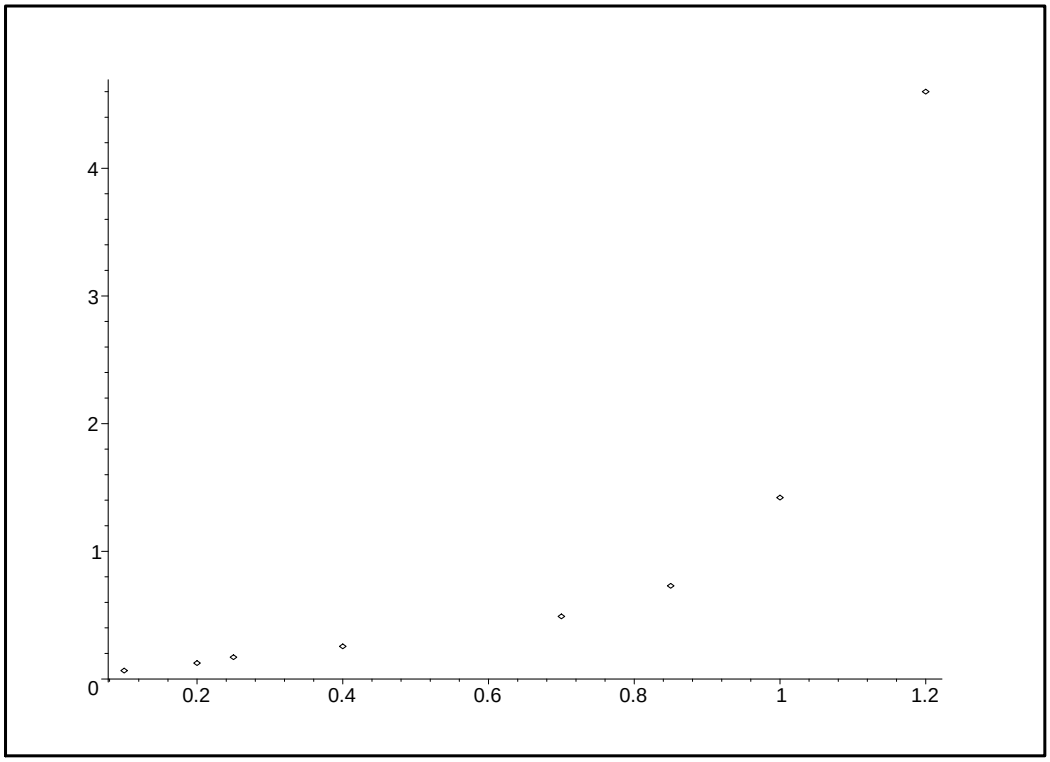
```
> restart; with(plots):
```

Schritt 1: Zeilenweises Einlesen der Daten aus der Datei kennlin.dat (2-spaltig) in die Liste Daten und anschließende Darstellung der Meßpunkte mit dem Kommando listplot

```
> Daten := readdata( ‘kennlin.dat‘, 2 );
```

$$\mathit{Daten} := [[.1, .066], [.2, .125], [.25, .170], [.4, .255], [.7, .49], [.85, .73], [1.0, 1.42], [1.2, 4.60]]$$

```
> dataplot := listplot( Daten, style=point,
                        symbol=diamond, color=black ):
> dataplot;
```



Schritt 2: Laden des Statistik-Paketes stats (sowie statplots) und Bereitstellen der Meßpunkte aus kennlin.dat für den Ausgleich mit der Funktion fit[leastsquare]

Problem: Die Funktion benötigt die Daten spaltenweise, d.h. die Meßpunkte und Meßwerte in 2 Vektoren! Lösung durch Lesen mit importdata.

```
> with( stats): with(statplots):
> datacol := importdata( `kennlin.dat`, 2 );
```

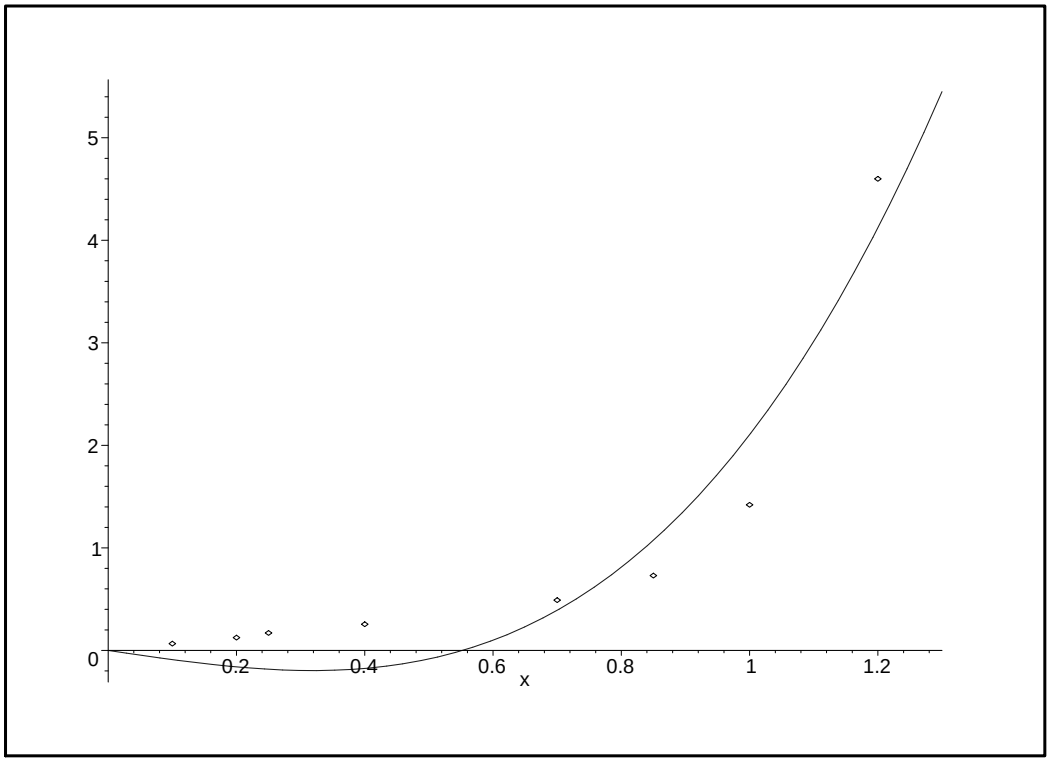
$$datacol := [.1, .2, .25, .4, .7, .85, 1.0, 1.2], [.066, .125, .170, .255, .49, .73, 1.42, 4.60]$$

Schritt 3: Auswahl der Ansatzfunktion z = phi(x) = a*x + b*x^3
und Ausgleich sowie Darstellung der Meßpunkte und der Ausgleichsfunktion

```
> fit[leastsquare[ [x,z], z = a*x + b*x^3 ,
> {a,b} ]]( [datacol] );
```

$$z = -.9243601597\,x + 3.027664481\,x^3$$

```
> curve := plot( rhs(%), x=0..1.3, color=blue ):
> display( { dataplot, curve } );
```

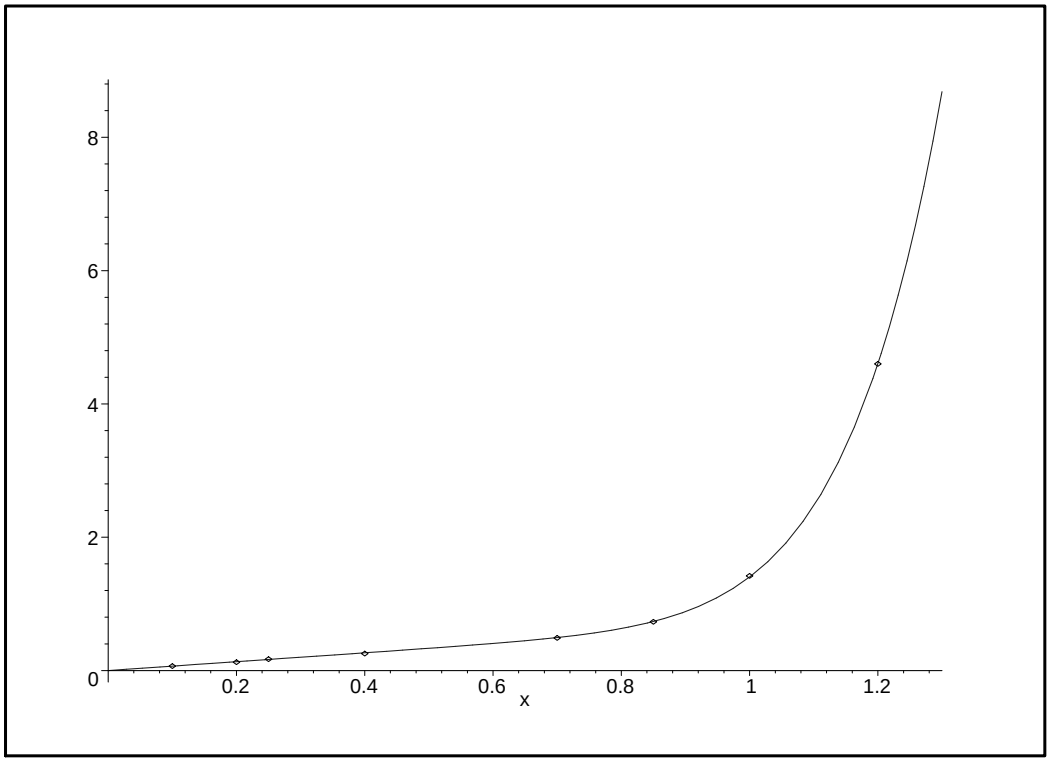


Schritt 4: Verbesserung des Ansatzes mittels des speziellen Polynoms 9.Grades z = phi(x) = a*x + b*x^9 und Ausgleich sowie Darstellung der Meßpunkte und der Ausgleichsfunktion

```
> fit[leastsquare[ [x,z], z = a*x + b*x^9 ,
> {a,b} ]]( [datacol] );
```

$$z = .6649091544\,x + .7374757884\,x^9$$

```
> curve := plot( rhs(%), x=0..1.3, color=blue ):
> display( { dataplot, curve } );
```



Animation

Animation ist in 2D und 3D möglich und sinnvoll, falls ein hinreichend schnelles Gerät zur Verfgung steht. Wir beschränken uns auf einfache Animationen. Die Kommandos `animate` und `animate3d` sind im Paket plots verfügbar.

Struktur: animate(F(x,t), x=a..b, t=c..d, Optionen)

Dabei ist x die unabhängige Variable von F, wogegen t der Animationsparameter ist.

Die Zahl der Bilder ist standardmäßig 16, kann mittels der Option frames aber verändert werden.

```
> restart; with(plots):

> animate( sin(2*Pi*(x+t)), x=0..4, t=0..1,
           numpoints=500, frames=8 );
```

Darstellung der Einzelbilder in einem Array:

```
> display(%):
```