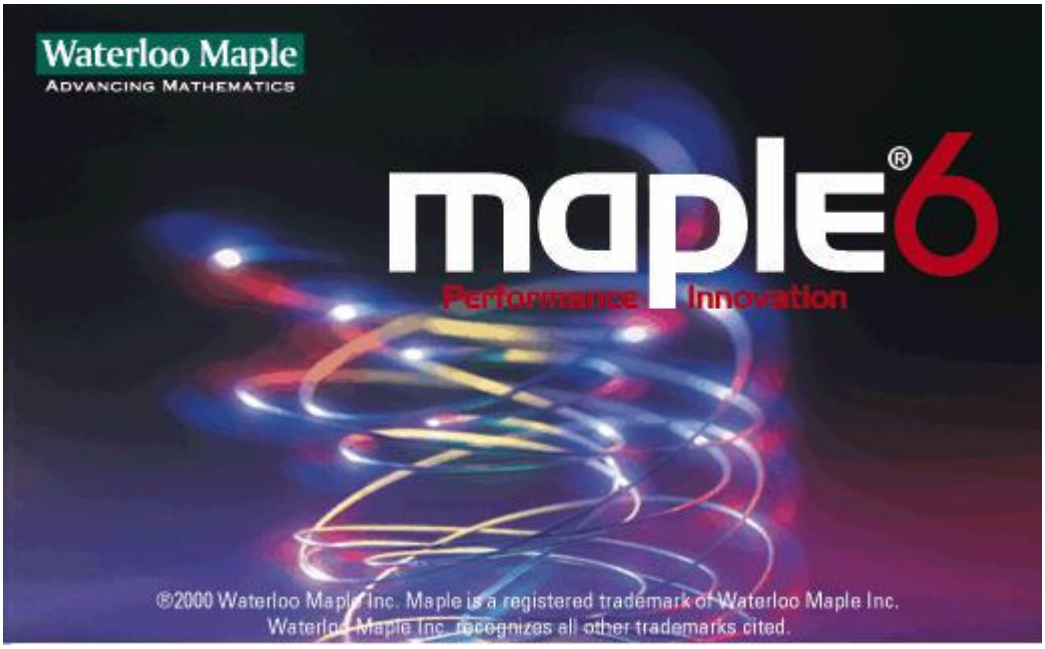
Einstieg ins Maple-Praktikum

Dieses Skript entstand im Rahmen eines QQ2 Projekts und soll Neulingen eine Orientierungsmöglichkeit im Rahmen des Mathe Praktikums bieten

Daniel Langer 2005



Einleitung

Das Maple System (im Folgenden mit Maple abgekürzt) gehört wie Mathematica, Derive oder Axiom zur Klasse der Computer-Algebra Systeme. Es ist ein interaktives Werkzeug zur Lösung numerischer und symbolischer mathematischer Probleme und zur grafischen Darstellung der Ergebnisse. Maple verwendet eine eigene, an Pascal angelehnte Programmiersprache und besitzt ca. 2500 vordefinierte Funktionen. Im Unterschied zu Programmiersprachen wie C, FORTRAN, Pascal oder APL und den zugehörigen numerischen Bibliotheken kann Maple auch Ausdrücke mit symbolischen Elementen auswerten oder vereinfachen.

Die Nullstellen von $x^2+p\,x+q=0$ können bei konventionellen Programmen nur bestimmt werden wenn p und q feste Werte haben. Maple dagegen liefert die bekannte analytische Lösung:

$$-\frac{1}{2}p+\frac{1}{2}\sqrt{p^2-4\,q}\,,-\frac{1}{2}p-\frac{1}{2}\sqrt{p^2-4\,q}\quad .$$

Maple beinhaltet Bibliotheksfunktionen für eine Vielzahl mathematischer Anwendungsgebiete, unter anderen:

· allgemeine Mathematik
· Kombinatorik
· Differential- und Integralrechnung
· Lineare Algebra
· Lineare Optimierung
· Statistik

Zur Darstellung von mathematischen Funktionen, Datenpunkten und Messwerten steht einem in Maple eine Vielzahl von graphischen Bibliotheksfunktionen zur Verfügung, unter anderen:

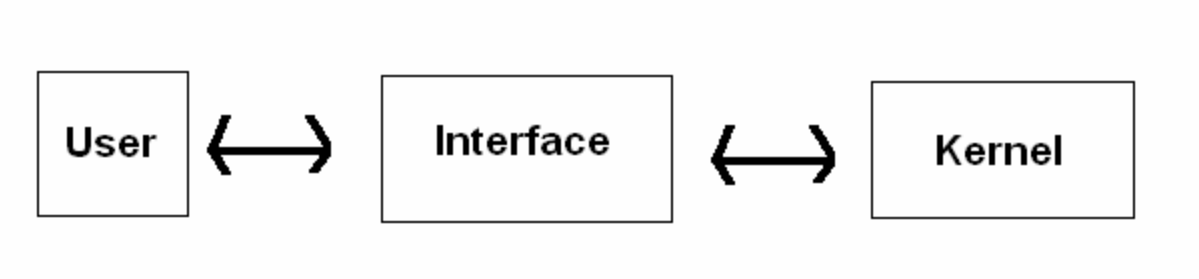
• x-y Grafiken
• x-y-z Grafiken
• parametrisierte Kurven und Flächen
• Konturlinien und -flächen
• 2D- und 3D-Animationen

Viele Funktionen in Maple funktionieren nur wenn die entsprechenden Pakete in denen Sie definiert sind auch zu Beginn geladen worden sind. Im Rahmen des Praktikums kommen nur eine Handvoll wichtiger Pakete in Betracht. Diese werden später noch ausführlich aufgelistet.

Architektur

Maple besteht aus einer Benutzerschnittstelle **(***User-Interface***)**, einem in C geschriebenen Kern, der mathematische Berechnungen ausführt, und einer Bibliothek von Funktionen, die bei Bedarf vom Kern geladen werden. Eine Maple Sitzung vollzieht sich in einem interaktiven Dialog zwischen dem Benutzer, der "Fragen" stellt, und dem Maple System, das "Antworten" auf diese Fragen gibt.

Der Benutzer gibt seine Fragen in der Maple-Syntax ein und drückt RETURN. Das User-Interface leitet die Kommandos zur Berechnung an den Kern weiter und gibt die Antwort in mathematischer Notation auf dem Bildschirm aus. Der Kern lädt ggf. zur Berechnung die notwendigen Funktionen aus der Maple-Bibliothek.



Die Vorteile dieser Architektur liegen unmittelbar auf der Hand:

- Durch die Trennung von Benutzerschnittstelle und mathematischen Kern kann eine Benutzerschnittstelle optimal auf eine Plattform angepaßt werden, z.B. durch Ausnutzung vorhandener Technologien wie OSF oder Microsoft Windows.

- Der mathematische Kern und die Bibliothek sind vom Funktionsumfang auf allen Plattformen identisch, die Bibliothek ist sogar als Datei Plattform übergreifend ohne Änderungen austauschbar.

Für das Mathe Praktikum sind neben der Maple Syntax und den mathematischen Kenntnissen der jeweiligen Themengebiete auch Programmierkenntnisse erforderlich. Die Funktionsweise eine for-Schleife, while-Schleife oder If-Anweisung wird vorausgesetzt, ebenso sollte einem auch der Modulo-Operator (mod) bekannt sein (in Java und C wird diese Operation mit dem %-Operator implementiert). Diese sind schon für das bestehen des 1 Praktikums erforderlich. Im Folgenden werden die Grundlagen der Programmierung die im Rahmen des Praktikums benötigt werden kurz vorgestellt. Dabei wird immer stets Bezug auf die Praktikumsaufgaben hinsichtlich der Maple Syntax genommen.

Programmierkenntnisse

Begriffe wie Deklaration, Definition, Initialisierung, Variablen, Array, if-Bedingungen, das Funktionsprinzip einer for-Schleife und einer while-Schleife, sollten einem bekannt und vertraut sein. Die Aufgaben im Praktikum lassen sich zum Teil nicht anders lösen. Im Verlauf der Veranstaltung Algorithmen und Programmierung 1 werden diese Gebiete alle vorgestellt, dies ist aber zum Zeitpunkt des 1 Praktikums noch nicht der Fall. Deswegen sollte man sich diese Grundlagen schon im Voraus aneignen.

Eine kurze und grobe Einführung kann man diesem Skript entnehmen. Für ausführlichere und weitergehende Erklärungen ist auf weiterführende Fachliteratur zu verweisen bzw. auf die Veranstaltung Algorithmen und Programmierung 1.

In der Programmierung stellt man sich am besten unter einer Variablen eine Schublade vor in die man einen Inhalt eines bestimmten Typs packen kann. Der Name und der Typ der Information der Variable ist das Namensschild der Schublade.

Variablen müssen bevor sie benutzt werden dem Interpreter (*Compiler*) bekannt gemacht werden. Dies geschieht in der „Deklaration“ einer Variablen. In den meisten Programmiersprachen gibt man bei einer Deklaration den Datentyp und den Namen der Variable an. In Maple braucht man keinen Datentyp bei der Deklaration von Variablen anzugeben. Dies geschieht implizit mit der Initialisierung. Initialisiert man eine Variable a:=5, wird sie als Integer interpretiert (Ganze Zahl). Initialisiert man eine Variable a:=5.5, wird Sie als float interpretiert (Fließkommazahl). Der Ausdruck a:=1/2 wird in Maple zu einem Bruch 1/2 ausgewertet. Um ein Ergebnis von 0.5 zu erhalten, muss man mit a:=1.0/2.0 rechnen. Dann erhält man als Ausgabe die 0.5. Addiert man a:=5 und b:=1.0 zusammen (also einen Integer und einen float) erhält man als Ergebnis 6.0, also ein float.

Damit zu Beginn eines Programms sich nicht irgendwelche zufälligen Werte in den Variablen befinden, weist man den Variablen am besten gleich bei der Deklaration einen Wert zu. Diese Wertzuweisung wird als Initialisierung bezeichnet.

Eine Funktion f(x) weißt einem Wert x immer eindeutig einen Wert y zu beim Aufruf f(Variable). Eine Funktion wird mit dem so genannten „Pfeiloperator“ deklariert. f:= x -> x^2

In Maple gibt es unter anderem die Möglichkeit neben Funktionen auch Prozeduren anzulegen. Die Definition der Prozedur beginnt mit dem Schlüsselwort proc und endet mit dem Schlüsselwort end. Dabei wird eine Parameterliste in Klammern hinter dem Schlüsselwort proc angegeben, die beim Aufruf an die Prozedur übergeben wird. Bei jedem Aufruf der Prozedur werden alle Anweisungen die dort implementiert sind ausgeführt.

If – Anweisung

If Anweisungen eignen sich besonders gut um Fallunterscheidungen zu implementieren.

```
if (Ausdruck) then

        Anweisung 1;
        Anweisung 2;
        :
        Anweisung m;

else

        Anweisung m+1;
        :
        Anweisung m+p;

fi;
```

Wenn die Evaluierung von **Ausdruck** in der ersten Klammer TRUE ergibt, wird **Anweisung 1 – Anweisung m** einmal streng sequentiell durchgeführt. Für den anderen Fall wenn die Evaluierung FALSE ergibt wird die **Anweisung m+1 – Anweisung m+p** ausgeführt. Der else Teil ist optional. If-Bedingungen können beliebig ineinander geschachtelt werden. Es muss nur auf die korrekte Klammerung geachtet werden. Mit dem Schlüsselwort **fi** wird in Maple eine If-Anweisung abgeschlossen. Es handelt sich hierbei um das umgedrehte „if“

Betrachten wir mal das Beispiel die Division zweier ganze Zahlen a durch b. Für den Fall, dass der Teiler den Wert Null annimmt, soll eine Fehlermeldung ausgegeben werden, ansonsten soll das Ergebnis ausgeben werden. Man kann diesen Ablauf testen, indem man vor der Ausführung die Variablen a und b verändert.

```
restart:
a:=3;
b:=4;
c:=0;

if (b=0) then
     print("Keine Divsion möglich da Nenner 0");
else
     c:=a/b;
     print("Die Divison von",a,"/",b,"war erfolgreich ",
evalf(c,2));

fi:
```

$$a := 3$$

$$b := 4$$

$$c := 0$$

"Die Divison von", 3, "/", 4, "war erfolgreich ", 0.75

Für den Fall, dass zu Beginn b= 0 gilt, kommt die Fehlermeldung:

$$a := 3$$

$$b := 0$$

$$c := 0$$

"Keine Divsion möglich da Nenner 0"

for-Schleife

Eine for-Schleife läuft über einen Index der bis zu einer Abbruchbedingung iterativ inkrementiert wird. Der Index ist im Beispiel mit der Variablen **i** gekennzeichnet.

```
for i from n to m do
     Anweisung 1;
     .
     .
```

```
    Anweisung n;

od;
```

Zu Beginn hat i den Wert n. i wird solange um 1 inkrementiert bis i=m als Abbruchbedingung die Iterationen abbricht. Anweisung 1 – Anweisung n werden in jeder Iteration einmal sequentiell ausgeführt. Mit dem Schlüsselwort od wird die for-Schleife in Maple beendet. Dabei handelt es sich um das umgedrehte „do“ was in der ersten Zeile die Anweisungen einleitet.

Geben wir einmal alle 10 Iterationen aus bei dem folgendem Beispiel aus:

```
for i from 1 to 10 do
    print(i,"-te Iteration");
od;
```

1, "-te Iteration"

2, "-te Iteration"

3, "-te Iteration"

4, "-te Iteration"

5, "-te Iteration"

6, "-te Iteration"

7, "-te Iteration"

8, "-te Iteration"

9, "-te Iteration"

10, "-te Iteration"

while-Schleife

Jede For Schleife kann auch mithilfe einer while-Schleife implementiert werden. Der Unterschied hierzu ist, dass die Abbruchbedingung in der while-Schleife implementiert ist und nicht im Header der Schleife.

```
while (Ausdruck) do
    Anweisung 1;
    .
    .
    Anweisung n;

od;
```

Solange die Auswertung **Ausdruck** TRUE ergibt, wird **Anweisung 1 – Anweisung n** durchgeführt. Ergibt die Auswertung FALSE, ist die while-Schleife beendet.

Betrachten wir das gleiche Szenario wie für die For-Schleife. Der Output ist kongruent.

```
m:=11:
i:=1:

while (i<m) do
     print(i,"-te Iteration");
     i:=i+1:
od;
```

1, "-te Iteration"

2, "-te Iteration"

3, "-te Iteration"

4, "-te Iteration"

5, "-te Iteration"

6, "-te Iteration"

7, "-te Iteration"

8, "-te Iteration"

9, "-te Iteration"

10, "-te Iteration"

Generell können alle Schleifen mit einem **break;** verlassen werden.

Der Modulo Operator

Eine wichtige Operation in Programmiersprachen ist die ganzzahlige Division mit Rest. Dies wird mit dem Modulo Operator implementiert.

a mod b;

Durch diese Anweisung wird der Rest, welcher bei der ganzzahligen Division von a durch b entsteht ermittelt.

```
12 mod 5;
```

2

Beispiel:

Das folgende Beispiel soll das Funktionsprinzip einer For-Schleife mit anschließender Fallunterscheidung demonstrieren. Es geht nur darum auszugeben, ob eine Zahl gerade oder ungerade ist. Das Zahlenintervall hläuft von 1-20. Mit Hilfe der äußeren for-Schleife durchläuft die Variable a nacheinander alle Werte von 1-20.

```
for a from 1 to 20 do

        if (a mod 2)=0 then
                print (a, „ist gerade“)
        else
                print (a, ”ist ungerade“)
        fi;

od;
```

1, "ist ungerade"
2, "ist gerade"
3, "ist ungerade"
4, "ist gerade"
5, "ist ungerade"
6, "ist gerade"
7, "ist ungerade"
8, "ist gerade"
9, "ist ungerade"
10, "ist gerade"
11, "ist ungerade"
12, "ist gerade"
13, "ist ungerade"
14, "ist gerade"
15, "ist ungerade"
16, "ist gerade"
17, "ist ungerade"
18, "ist gerade"

19, "ist ungerade"

20, "ist gerade"

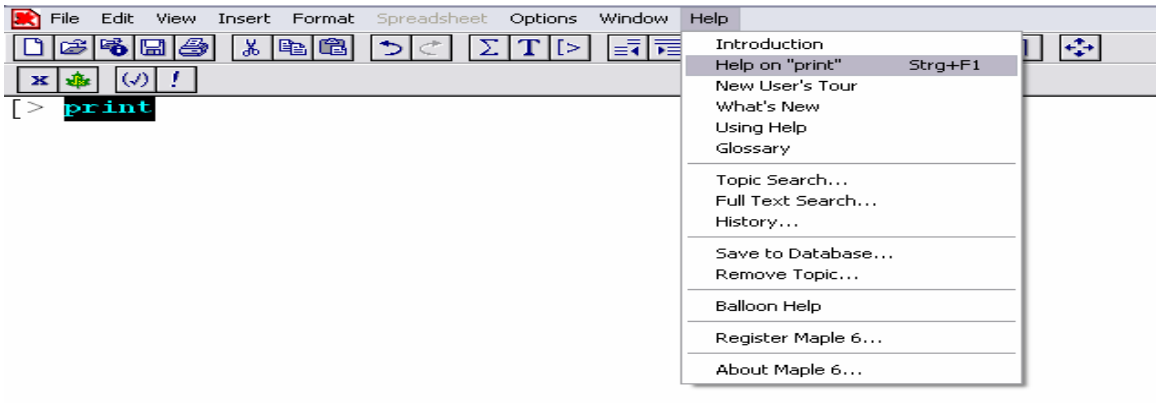
In der ersten Iteration hat a den Wert 1, also a=1. Nun werden alle Anweisungen innerhalb der For-Schleife einmal ausgeführt. Um zu testen ob eine Zahl gerade oder ungerade ist, können wir uns den Modulo Operator zur Hilfe nehmen. (a mod 2) ergibt 0 wenn kein Rest durch die ganzzahlige Division entsteht. Folglich kann a nur gerade sein. Ist der Rest ungleich 0 ((a mod 2)!=0) war die Zahl ungerade. Dies wird mit dem else-Teil abgedeckt, ohne das explizit ein weiterer Ausdruck evaluiert werden muss.

Für den Fall, dass also (a mod 2) gleich 0 ergibt, können wir ausgeben, dass a eine gerade Zahl war. Ist hingegen der Rest nicht Null, springen wir in den else-Teil und geben aus, dass die Zahl ungerade war. Die If-Anweisung wird also insgesamt 20 Mal durchgeführt. Hat a den Wert 20 erreicht, ist die For-Schleife beendet.

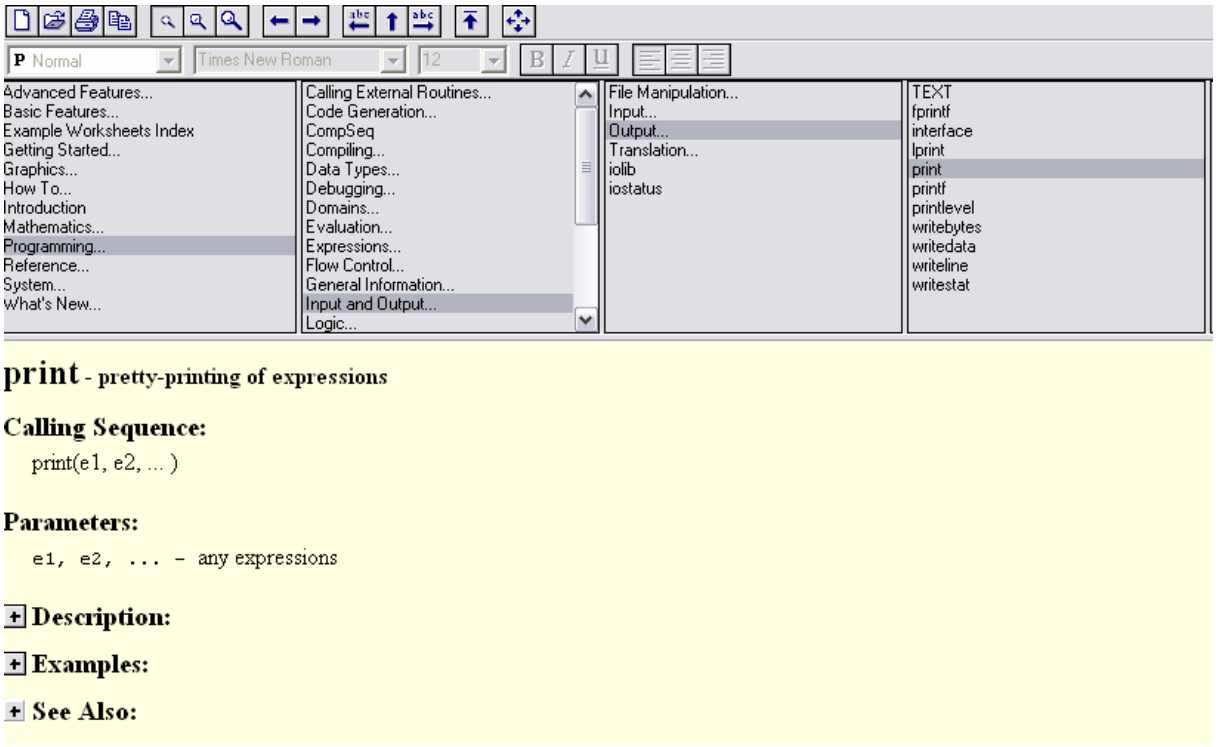
Hilfe-Funktion

Ein sehr hilfreiches Tool ist die integrierte „Help on“ Funktion. Wenn man irgendeine Information bezüglich einer Funktion erhalten möchte, schreibt man einfach den Namen der Funktion und markiert diesen. Dann erscheint der markierte String im Menüaufruf „Hilfe“ und leitet einen bei Auswahl direkt an die entsprechende Stelle der dokumentierten Funktions-Definition.

Schauen wir uns die Hilfe-Funktion für den „print“ Befehl an. Man schreibt einfach den Namen der Funktion von dem man mehr Informationen erhalten will, markiert diesen und wählt dann im Hilfe Menü „Help on „print“ „ aus oder man drückt einfach STRG+F1.



Betrachten wir nun die Ausgabe etwas genauer:



In der ersten Zeile wird der Befehl den man aufgerufen hat angezeigt. In diesem Beispiel ist das die Funktion „print“.

Unter „Calling Sequence“ wird die Aufruf-Syntax der Funktion allgemein beschrieben. Wir sehen, dass man die Funktion mit „print (e1, e2, ..)“ aufrufen kann. e1 und e2 sind zwei allgemeine Beispielparameter die man beim Aufruf mitgeben kann. Um ganz genau zu wissen, was man beim Aufruf als Parameter mitgeben kann, schaut man unter Parameter“ nach. Dort sind die Parameter genauer beschrieben. In diesem Fall sehen wir, dass e1 und e2 irgendein Ausdruck sein kann. Dies kann ein String sein, z.B. „Hello World“

```
>print("Hello World");
>
```

"Hello World"

Oder aber auch nur Zahlen:

```
>print(4,5,6);
>
```

4, 5, 6

Oder auch verwendete Variablen:

```
>b:=4;
```

```
print(b,5);
>
```

$$b := 4$$

$$4, 5$$

Natürlich kann man auch beides gemeinsam ausgeben, einen String und eine Variable:

```
>b:=4;
print("Die Variable b hat den Wert:", b);
>
```

$$b := 4$$

$$\text{"Die Variable b hat den Wert:"}, 4$$

Unter dem Punkt „Description“ ist dann eine genaue Erklärung was man wie mit der Funktion machen kann aufgeschlüsselt.

Description:

- The function print displays the values of the expressions appearing as arguments, and returns NULL as the function value. The expressions printed are se by a comma and a blank. Note that print(``) will print a blank line. The ditto commands, % and %%, will not recall the output of the print command, sin return value of the function is NULL.
- The following Iris variables control the format. See interface for setting Iris variables.
- The Iris variable prettyprint is checked to determine the format in which the expressions are to be printed. If set to true (the default value) the expression displayed in a two-dimensional format, centered if possible. If set to false the expressions will be line printed in one dimension.
- The Iris variable verboseproc is checked to determine how the body of Maple procedures (i.e. the code) will be displayed. It can be displayed in full, wi indentation, or abbreviated to simply ``...''. By default user procedures are displayed in full, and Maple library procedures are abbreviated.
- The Iris variable screenwidth specifies the number of characters that fit on the output device being used. Since most CRTs are 80 characters wide, the d value is 79.
- The Iris variable labelling (or labeling) enables the use of % variables to reduce the size of the output. These % labels identify common subexpressions (t appearing more than once) in the output.

Um sich ein paar Beispiele anzusehen, schaut man einfach unter „Examples“ nach.

Examples:

```
> print(red,rouge,rot);
```

$$red, rouge, rot$$

```
> v := array([1,2,3]):
  v;
```

$$v$$

```
> print(v);
```

$$[1, 2, 3]$$

```
> sin;
```

$$\sin$$

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

$$\mathbf{proc}(x::algebraic) \ \ldots \ \mathbf{end\ proc}$$

Überblick der wichtigsten Maple-Funktionen

Es folgen alle wichtigen Maple-Funktionen wie sie nach und nach im Praktikum auftauchen.
Dies ist nur ein gesamter Überblick aller Funktionen die man im Mathe-Praktikum benötigt. Eine genaue Beschreibung mit Beispielen ist denn Maple-Übungsblätter zu entnehmen.

Praktikum 1:

`restart`	Alle Variablen werden gelöscht
`sqrt(a)`	Wurzelausdruck
`evalf(e, Komastellen)`	Fließkomadarstellung mit Möglichkeit der Stellenangabe
`sum(Ausdruck, Intervall)`	Endliche und Unendliche Summen
`product (Ausdruck, Intervall)`	Endliche und Unendliche Produkte
`plot (Funktion, Darstellungsintervall, Parameter)`	Zeichenroutine für 2D Grafiken
`plot3d(Funktion, DarstellungX,DarstellungY,Paramter)`	Zeichenroutine für 3D-Grafiken
`with (plots)`	Lädt alle Zeichenroutinen
`animate(Funktion, Intervall, Animiationsnintervall)`	Animierte Zeichenroutine
`expand()`	Potenzschreibweise
`factor()`	Gegenteil von expand
`simplify()`	Vereinfachung von Ausdrücken
`f:= x-> Funktionsausdruck`	Definition einer Funktion
`f:= proc(x)`	Definition einer Prozedur
`seq(Ausdruck, Intervall)`	Geordnete Menge von Ausdrücken
`solve(Gleichung,Variable)`	Löst Gleichung nach Variable auf
`solve({eqn1,eqn2,eqn3}, {x,y,z})`	Löst Gleichungssystem

```
if (Ausdruck) then
        Anweisung 1;
        Anweisung m;
else
        Anweisung m+1;
        Anweisung m+p;
fi;
```
If-Anweisung

```
for i from n to m do
        Anweisung 1;
        Anweisung n;
od;
```
For-Schleife

```
while (Ausdruck) do
        Anweisung 1;
        Anweisung n;
od;
```
While-Schleife

Praktikum 2:

`ln(p)`	Logarithmus von p
`exp(p)`	e-Funktion (e^p)
`Limit(Funktion,Intervall)`	Limes Schreibweise (Großes L!)
`limit(Funktion,Intervall)`	Berechnet Grenzwert (Kleines l!)
`abs(a)`	Absolutbetrag

Praktikum 3:

`sin(x)`	Sinus
`cos(x)`	Cosinus
`tan(x)`	Tangens
`display([plot1],[plot2], …)`	Liste von Plots zur Darstellung
`diff(Funktion, Variable)`	Ableitung nach Variable
`unapply(Funktion,Variable)`	Ableitungsfunktion von Variable

int(Funktion,Variable,Integrationsintervall)		Integration von Funktion nach Variable über Intervall
Re(z)	1	Realteil einer Komplexen Funktion
Im(z)		Imaginäranteil einer Komplexen Funktion

Praktikum 4:

with(linalg)	Lädt Packet für Lineare Algebra
vector([x,y,z])	3D Vektor
evalm(Matrix)	Wertet Matrix Ausdruck aus
crossprod(u,v)	Kreuzprodukt von Vektor u und Vektor v
dotprod(u,v)	Skalarprodukt von Vektor u und Vektor v
norm(u)	Betrag Vektor u
multiply(u,v)	Vektorprodukt u*v

Praktikum 5:

matrix (n,m)	Definiert eine n*m Matrix
polygonplot(Polygone, Optionen)	Zeichnet 3D Polygon plot

Praktikum 6:

with(networks)	Packet zur Graphentheorie
new(Graph)	Definiert einen neuen Graphen
addvertex({A,B, …}, Graph)	Fügt Knoten A,B, … hinzu
addedge([{A,B}, …]Graph);	Fügt Kante zwischen A und B hinzu
vertices(Graph)	Gibt die Menge aller Knoten von Graph aus
nops(vertices(Graph))	Gibt Anzahl der Knoten aus
edges(Graph)	Gibt alle Kanten aus
ends(Graph)	Gibt Knoten von den Kantenenden aus
eweight(Kante,Graph)	Bewertung der Kante aus
neighbors(A,Graph)	Nachbarknoten von A aus
arrivals(A,Graph)	Knoten die über Kanten in A münden
departures(A,Graph)	Knoten die über Kanten von A erreichbar sind
delete({A,B},Graph)	Löscht Knoten A und B und damit verbundenen Kanten
adjacency(Graph)	Erzeugt Adjazenzmatrix

Praktikum7:

with(stats)	Stats Packet
with(stats[statplots])	Unterpacket statplots
rand(1..n);	Erzeugt Zufallszahl zwischen 1 und n
histogram([Sequenz], Optionen)	Zeichnet ein Histogramm
scatterplot([Sequenz], Optionen)	Zeichnet diskrete Verteilung
random[binomiald[1,2/3]](25);	25 Zufallszahlen wobei 2/3 "1" und Rest "0" ->Binomialverteilt
random[normald[10,2]](25);	25 Zufallszahlen mit Mittelwert 10 und Standartabweichung 2 → Normalverteilt
with(describe)	Packet für die üblichen Statistischen Messwerte
mean(Sequenz)	Mittelwert
standarddeviation[0](Sequenz)	Standardabweichung
variance[0](Sequenz)	Varianz
with(combinat,randcomb)	Packet für Stichproben
randcomb(Sequenz,n)	n Werte aus Sequenz

Literaturhinweise

Algorithmen und Programmierung 1 Skript von Prof. Dr. Frank Victor

Mathematik Skript von Dr. Ane Schmitter

Maple Übungsblätter von Dr. Elmar Lau

Mathematik Skript von Prof. Dr. Konen www.gm.fh-koeln.de/~konen