Einfuehrung in *Mathematica*

(@Mail: mibeer@uos.de)

Grundlegendes

Mathematica erscheint zunaechst wie ein ganz normaler Editor.Die Eingabe erfolgt auf die gleiche Weise, wenn auch Mathematica merkwuerdig formatiert.Will man in Mathematica tatsaechlich nur einen Text schreiben, der so bleiben soll, wie man ihn eingibt, so kann man ihn nachdem man ihn markiert hat, als "Text" mittels Format -> Style -> Text als solchen kennzeichnen.Danach erscheint er genau so, wie man es in einem Editor erwarten wuerde.In diesem Menue kann man *Mathematica* auch veranlassen, Text als Ueberschriften ua.zu formatieren.

Mathematica organisiert alle Eingaben in Zellen.Jede Zelle wird von *Mathematica* rechts mit einer Klammer eingeschlossen.-Klickt man diese Klammer an, so kann man Formatierungen etc.auf die gesamte Zelle anwenden.Zellen entsprechen in etwa den bakannten Absaetzen in "gewoehnlichen" Texten.Beim reinen Tippen von Text wirkt sich das jedoch zunaechst nicht aus. Wenn man den Text aber organisieren will, verpackt man am besten jede Ueberschrift und jeden Absatz in eigene Zellen.Diese lassen sich dann nach Bedarf mittels Doppelklick auf die Klammer ein - und ausklappen. *Mathematica* kuemmert sich um die Formatierung. Wichtig: Ein Zeilenumbruch (mit [Enter]) ist fuer *Mathematica* keine Trennung.
Will man die Zelleinteilung veraendern, so gelingt dies mittels des Menues Cell.
Will man Mathematicadazu bewegen, das Eingegebene auszuwerten, muss man[Shift] + [Eingabe] druecken.Ausgewertet wird immer die ganze Zelle, die dazugehoerenden Ausgaben erscheinen unterhalb der Zelle (und koennen weggeklappt werden)

Arithmetik

Das einfachste, was Mathematica anders als gewoehnliche Editoren kann, ist es, arithmetische Ausdruecke auszuwerten, z.B.

```
4.5*3
```

```
2*4 / 3
```

Will man kompliziertere Formeln eingeben, sind die bereitgestellten Paletten hilfreich, mit denen man sich per Mausklick z.B.obigen Bruch in der gewohnten Weise eingeben kann (Palette BasicMathInput->Bruch). Zwischen den einzelnen Eingabekaestchen kann man mittels Maus oder [Tabulator] wechseln:

$$\frac{2*4}{3}$$

```
(7 / 9)^5
```

Will man Funktionen aufrufen, so ist zu beachten, dass *Mathematica* die Argumente nicht in runden (), sondern eckigen [] Klammern erwartet. Weiter unterscheidet *Mathematica* zwischen gross - und kleingeschriebenen Buchstaben, alle von *Mathematica* zur Verfuegung gestellten Objekte beginnen mit einem Grossbuchstaben:

```
sin(1.2) (*Klappt nicht *)
```

```
Sin[1.2] (*Klappt *)
```

Andere wichtige Bezeichner:

```
E
```

```
I
```

```
Pi
```

Mathematica rechnet wenn moeglich symbolisch, d.h. Sin[2] wird als Log[2] belassen. Will man *Mathematica* zur numerischen Berechnung eines Termes zwingen, so funktioniert dies mit der Funktion N[] :

```
Log[2]

N[Log[2]]

Log[2] // N
```

Will man das Ergebnis mit einer bestimmten Genauigkeit, so kann man dies N mitteilen:

```
N[Pi, 5000]
```

Den Kernel kann man jederzeit mit Evaluation->Abort Evaluation abbrechen. Bei Problemen mit dem Kernel kann man diesen mit Evaluation->Quit Kernel->Local beenden und Evaluation->Start Kernel->Local neu starten.

Auf das letzte Ergebnis kann mit **%**, auf das vorletzte mit **%%** usw. zugegriffen werden:

```
%
```

Will man zB. eine Zufallszahl mit **Random** erzeugen, hat aber nur einen vagen Begriff von der Funktion, so bietet sich die Hilfe an. Man kann sie einmal ueber Help->Documentation Center, um zur Hauptseite zu gelangen. Zum Anderen kann man zu dem aktuell unter dem Cursor stehenden Begriff wie ueblich mittels [F1] eine entsprechende Hilfeseite aufrufen. Dort erfaehrt man uA., ob der Begriff eine Konstante oder Funktion darstellt, wie viele Parameter ie Funktion erwartet und was sie bedeuten... Oft sind die Bezeichnungen der Funktionen intutitiv. Will man zB. eine Gleitkommazahl als Bruch darstellen, bietet sich die Funktion Rationalize[] an.
Funktionen muessen nicht unbedingt numerische Werte zurueckliefern, auch "True/False" etc sind moeglich:

```
PrimeQ[7]
```

Variablen

Rechnen mit Variablen ist, aehnlich wie in Java oder C, moeglich, man sollte jedoch folgende Regeln zur Benennung von Variablen und allgemein Bezeichnern beachten:

```
x + x (* Buchstaben sind normalerweise kein Problem *)

xx (* Variablen koennen mehrere Zeichen enthalten *)

2 x + 4 x(* Zahlen zu Anfang werden als Mutliplikation mit der entspr. Zahl gelesen! *)

x y xy (* Bei der Multiplikation kann der Stern weggelassen werden,
 es muss dann jedoch ein Leerzeichen eingefuegt werden *)
```

Mathematica stellt wie oben schon gesehen viele Konstanten usw. zur Verfuegung. Diese beginnen stets mit einem Grossbuchstaben. Um Verwechslungen mit diesen auszuschliessen bietet es sich an, eigene Bezeichner immer mit einem Kleinbuchstaben beginnen zu lassen. Es sind auch gewisse Sonderzeichen als Bezeichner moeglich.
Weiterhin gibt es einige hilfreiche Funktionen:

```
Expand[x (3 x + 7 y) + 2 x^2 ] (* ausmultiplizieren *)

Factor[x^3 + 2 x^2 + 3 x] (* In Faktoren zerlegen *)

Simplify[(x + 4)^2 - (5 x - 4)^2] (* Vereinfachen (schnell aber nicht unbedingt vollstaendig *)

FullSimplify[(x + 4)^2 - (5 y - 4)^2] (* Vollstaendiges Vereinfachen *)

Cancel[(x^2 - 2 x) / (2 x)] (* Aufforderung zum Kuerzen *)

Together[3 / x + 4 / x^2] (* Zusammenfassen auf einen Bruchstrich*)
```

```
Apart[(4 + 3 x)/x^2] (* Partialbruchzerlegung durchfuehren *)
```

Wertzweisungen an Variablen erfolgt mehr oder weniger intuitiv:

```
a = x + 2
```

```
x = 2
```

```
a (* wie erwartet *)
```

Zy beachten ist, dass alle so def. Variablen global, also ueberall im NOtebook gueltig sind, mit allen Vor- und Nachteilen, wie sie auch aus anderen Programmiersprachen bekannt sind . Will man Probleme vermeiden, kann man die Symbole, die man verwenden will, zuvor loeschen:

```
Clear[x, a] (* alternativ a=. *)
```

Wenn man sich zu sehr in den Symbolen verheddert hat, hilft ein Beenden des Kernels mit Evaluation->Quit Kernel->Local. Dann startet man mit einem komplett neuen Kernel.

Generell ist es besser, Variablen nicht direkt auf Werte zu setzen (x=4), sondern, will man einen symbolische nAusdruck numerisch auswerten, nur fuer die eine Auswertung best. Symbole zu setzen:

```
a = x^2
```

```
a /. x → 2
```

```
a
```

Enthaelt der Ausdruck mehrere Variablen, sogibt man anstatt einer einzelnen Ersetzungsregel eine Liste von Regeln an. Listen werden generell mit [] umschlossen, die einzelnen Elemente durch Komma getrennt:

```
a = x^2 + y
```

```
a /. {x → 2, y → 4}
```

Will man die unmittelare Auswertung vermeiden, so benutzt man := an Stelle von =. Die Auswertung erfolgt dann erst beim Zugriff auf b und nicht schon bei der Zuweisung.

```
b := a
```

```
b
```

Funktionen

Selbstverstaendlich ist es moeglich, eigene Funktionen zu definieren. Auch sie werden mit [] Klammern an Stelle von runden Klammern aufgerufen und relativ intuitiv definiert:

```
f[g_] := g + 3
```

```
f[4]
```

Wichtig ist hierbei, die Parameter jeweils gefolgt von einem Unterstrich anzugeben, da *Mathematica* sonst den Parameter als Teil des Bezeichners interpretiert und nicht als veraenderlichen Parameter. Dies ist eine echte Funktion. Auch wenn g an einer anderen Stelle einen Wert zugewiesen bekaeme, wuerde f[] sich noch wie gewuenscht verhalten:

```
g = 4
```

```
f[1]
```

Ganz intuitiv werden mehrparametrige Funktionen defniert:

```
g[x_, potenz_] := x^potenz

g[2, 3]
```

Man kann selbstverstaendlich Funktionen auch symbolisch benutzen:

```
Clear[a, b]; g[a, b]
```

Eine variable Zahl von Parametern kann man verwenden, indem man an Stelle es einfachen Unterstriches "_" einen doppelten "__" verwendet:

```
h[x_, y__] := x + y

h[2, 1, 1, 1]
```

Drei Unterstriche lassen auch die Moeglichkeit, keinen Parameter zu uebergeben, zu:

```
funk[x_, z___] := 2 * x + z

funk[a]

funk[a, b, c, d]
```

Es sei angemerkt, dass darueber hinaus noch eine weitere Moeglichkeit besteht, Funktionen zu definieren :

```
pure = #1^2 &
pure[3]
```

Diese Art der Deklaration nennt sich "Pure Function", "#1" bedeutet den Wert des ersten Parameters, "&" ist zur Kennzeichnung einer Pure Function notwendig. Der Umgang mit Pure Functions unterscheidet sich nicht von dem "normaler" Funktionen.

Listen

Mathematica kann neben Skalaren auch mit einfachen und mehrfach verschachtelten Listen, also Vektoren und Matrizen, umgehen. Eine Liste wird generell einfach dadurch erzeugt, indem man die einzelnen Listenelemente durch Kommata getrennt aufzaehlt und mit geschweiften Klammern {} umgibt:

```
liste = {1, 2, 3, 5, 11}
```

Listen koennen generell alles moegliche was auch einzelne Variablen enthalten koennen, aufnehmen, also neben Zahlen auch Variablen, andere Listen und sogar Grafiken. Verschachtelte Listen haben dabei folgende Gestalt:

```
verschachtelteListe = {{1, 3}, {4, 7, 3}, gNr}
```

Man erkennt dabei, dass eine Liste nicht gleichartige Elemente besitzen muss, sondern beliebige Arten von Eintraegen gemischt werden koennen. Insbesondere muss eine verschachtelte Liste nicht Listen gleicher Laenge enthalten.

Der Zugriff auf einzelne Elemente erfolgt aehnlich wie in anderen Programmiersprachen ueber den Index, bei verschachtelten Listen werden mehrere Indizes angegeben:

```
liste[[2]]

verschachtelteListe[[1]][[2]]

verschachtelteListe[[1, 2]] (* Kurzschreibweise *)
```

Vektoren und Matrizen sind in *Mathematica* dabei ebenfalls nur Listen, die allerdings bestimmte Bedingungen erfuellen muessen. Vektoren sollten lediglich Skalare enthalten, Matrizen nur Vektoren gleicher Laenge:

```
einVektor = {1, 4, 3, x}

eineMatrix = {{1, 4, 3}, {3, 3, 3}}
```

Hilfreich ist beim Umgang mit Matrizen und Vektoren oft, sie in der gewohnten Form darzustellen. dazu dient **MatrixForm** bzw. **TableForm**:

```
MatrixForm[einVektor]

MatrixForm[eineMatrix]

TableForm[eineMatrix, TableHeadings → {Automatic, {"ersteSpalte", "zweite Spalte"}}]
```

Mathematica - Funktionen koennen in der Regel mit Listen umgehen, das Ergebnis ist dann wiederum eine Liste:

```
Exp[einVektor]
```

Weiterhin ist es moeglich, eine Reihe von Operationen auf Listen auszufuehren, uA:

```
l1 = {1, 2}; l2 = {3, 4}; matrix1 = {{3, 5}, {4, 7}}; matrix2 = {{2, 4}, {1, 5}};
l1 - l2 (* Multiplikation *)
l1 * l2 (* Addition *)
l1.l2 (* Skalarprodukt *)
l1.matrix1 (* Matrix auf Vektor multiplzieren *)
matrix1.matrix2 (* Matrizenmultiplikation*)
Transpose[matrix1] (* Matrizentransposition *)
Inverse[matrix1] (* Inverses einer Matrix bestimmen - nur bei entsprechenden Matrizen*)
Det[matrix1] (* Determinanten bestimmen *)
Eigenvalues[matrix1] (* Eigenwerte berechnen *)
Eigenvectors[matrix1] (* Eigenvektoren betsimmen *)
Eigensystem[matrix1] (* Eigenwerte und Vektoren bestimmen *)
LinearSolve[matrix1, l1] (* Gleichungssystem loesen *)
NullSpace[matrix1] (* Basis des Kerns bestimmen *)
RowReduce[matrix1] (* Gaussalgorithmus zur Vereinfachung anwenden *)
MatrixPower[matrix1, 3] (* Liefert hier matrix1*matrix1*matrix1 *)
MatrixExp[matrix1] (* liefert Exponentialmatrix e^matrix1 *)
```

Darueber hinaus existieren verschiedene hilfreiche Funktionen fuer Listen. Allen ist gemeinsam, dass sie ie urspruenglichen Listen unveraendert lassen, und die Ergebnisse als Rueckgabewert liefern:

```
Length[eineMatrix] (* Liefert die Laenge der ersten Ebene der Liste *)
b = Flatten[a = {1, {1, 2}, {3}, {4, {2, 3}}}, n = 1] (* Eliminiert  n Ebenen der Liste*)
Partition[{1, 2, 3, 4, 5, 6, 7, 8, 9}, 3]
(* Unterteilt die Liste in Unterlisten zu je drei Elementen *)
Join[a, einVektor] (* Haengt Listen aneinander *)
Union[a, einVektor] (* Bildet Vereinigung, doppelte Eintraege werden gekuerzt *)
Intersection[a, einVektor] (* Liefert Schnittmenge *)
```

Oft ist es hilfreich, Listen automatisch mit Werten zu fuellen. Dazu dient **Table[]**:

```
Table[i, {i, 0, 4}]
Table[i, {i, 0, 4, 2}] (* Mit Angabe der Schrittweite *)
```

Auch ist es moeglich, aus einer Liste bestimmte Eintraege auszuwaehlen. Dazu kann man **Cases** (zum Suchen mittels Muster)oder **Select** (Zum Suchen mittels Auswahlfunktionen, die auf ein Element angewandt **true** oder **false** liefern) benutzen:

```
Cases[{1, 2, 3, "String", 4, "oeh"}, _Integer] (* Auswahl nach Typ *)
Cases[{v^3, 2, 3 + v, "regen"}, t_ + v_] (* Auswahl nach Muster *)
Select[{1, 4, 3, 2}, EvenQ]
```

Zuletzt kann man verschiedene Funktionen zur Manipulation von Listen verwenden:

```
RotateLeft[{t, 42, 5, 6}] (* bringt den ersten Eintrag der Liste an den Schluss *)
Reverse[{t, 42, 5, 6}] (* Kehrt die Reihenfolge er Elemente um *)
Drop[{t, 42, 5, 6}, 2] (* Schneidet die erten zwei Elemente ab*)
First[{t, 42, 5, 6}] (* Liefert erstes Element *)
Last [{t, 42, 5, 6}] (* Liefert letztes Element *)
Append[{t, 42, 5, 6}, 3] (* Haengt Elemente an die Liste an *)
Prepend[{t, 42, 5, 6}, 3] (* Fuegt am Anfang ein *)
Insert[{t, 42, 5, 6}, 3, 2] (* Fuegt an angegebener Stelle ein *)
Permutations[{t, 42, 5, 6}] (* Gibt Liste mit allen moeglichen Permutationen zurueck *)
Sort[{t, 42, 5, 6}] (* Sortiert Liste nch eingbauten Regeln*)
```

Ausgabe / Grafik

Einfache variablen, Text oder aehnliches kann man mit der Funktion Print[] ausgeben. Print fuegt automatisch Zeilenumbrueche ein, kennt aber auch Steuerzeichen wie \n (Zeilenumbruch)oder \t (Tabulator)

```
a = 42; Print["Hallo ", 23, "\n", "\t", "naechste Zeile eingerueckt ", a]; Clear[a]
```

Mittels **Plot[]** kann man Funktionen plotten:

```
Plot[Sin[x], {x, 0, 5}]
```

Es sind auch mehrere Grafen auf einmal moeglich. Dabei ist an Stelle einer Funktion einfach eine Liste aller Funktionen, deren Grafen man geplottet haben will anzugeben. Natuerlich darf die Variable, ueber die geplottet wird, nur einen Namen haben:

```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 Pi}]
```

Das Aussehen des Plots kann man veraendern, indem man Plot zusaetzliche Parameter uebergibt:

```
Plot[Sin[x], {x, 0, 2 Pi}, PlotStyle → Thickness[0.05]]
```

PlotStyle steuert dabei das Aussehen der einzelnen Kurven uA. die Art der Kurve (**Dashing**), Dicke (**Thickness**), Farbe (**RGBColor**) etc
Um einen Ueberblick ueber die Optionen einer Funktion zu erhalten, kann man **Options[]** verwenden :

```
Options[Plot]
```

Plottet man mehrere Grafen, so uebergibt man an Stelle eines Parameterwertes fuer eine Liste mit Parameterwerten, wobei jedem Graf der Parameterwert abhaengig von der Position der enspr. Funktion in der Liste der Funktionen entspricht. Bsp :

```
Plot[{Sin[x], Cos[x]}, {x, 0, Pi}, PlotStyle → {{Thickness[0.01], RGBColor[0, 0, 1]},
   {Dashing[{0.01, 0.02, 0.04}] Thickness[0.03], RGBColor[1, 0, 0]}}]
```

Parameter, welche nicht an eine spezielle Kurve gebunden sind, also bspw. die Beschirftung der Achsen, werden nicht ueber **PlotStyle**, sondern ueber spezielle andere Parameter gesetzt, zB mit
AspectRatio das Seitenverhaeltnis
AxesLabel die Beschriftung der Achsen
AxesOrigin den Ursprung des Koordinatenkreuzes
PlotLabel die Beschriftung des gesamten Plots
PlotRange den zu plottenden Bereich ...

```
Plot[{Tan[x], Sqrt[x]}, {x, 0, Pi / 4}, AxesLabel → {"x", "Tan bzw Wurzel"},
 PlotLabel → "Netter Plot", PlotRange → {{0, Pi / 8}, {0, 1 / 2}}, AxesOrigin → {0, 0.5},
 AxesStyle → {Red, Green}, Background → Blue, BaseStyle → Black (* Frueher DefaultColor *)]
```

Es istauch moeglich, dem Plot eine Legende hinzuzufuegen. Dazu wird das Paket **PlotLegends‘** benoetigt, welches mittels **Needs[]** (oder alternativ mittels <<)geladen werden kann:

```
Needs["PlotLegends‘"]
```

```
Plot[{Sin[x], Cos[x]}, {x, 0, 2 Pi}, PlotLegend → {Style["Sinus", Red], "Cosinus"}]
```

Grafiken werden als ganz normale Objekte behandelt, die auch an Variablen zugewiesen werden koennen. Ein abschliessender Strichpunkt unterdrueckt die Ausgabe bei der Zuweisung:

```
plot1 = Plot[Sin[x], {x, 0, Pi}];
```

Diese Bilder koennen mittels Show gezeichnet werden:

```
Show[plot1]
```

Auf diese Art koennen nicht nur Plots, sondern jede Art von Grafik benutzt werden:

```
Show[Graphics[Circle[{0, 0}, 0.1]]]
```

Show kennt ebenfalls Parameter:

```
Show[plot1, BaseStyle → Hue[0.56]]
```

Wichtig ist hierbei, dass immer die Parameter vom ersten angegebenen Objekt uebernommen werden, wenn also bei beiden Objekten eine Standartfarbe eingestellt wurde, wird die des ersten Objektes zur Standartfarbe der Gesamtrafik:

```
plot2 = Plot[Cos[x], {x, 0, Pi}, BaseStyle → Hue[0.56]]
```

```
kreis = Graphics[Circle[{0.3, 0.3}, 0.3], BaseStyle → Hue[0.1]]
```

```
Show[plot2, kreis]
```

Fuer Funktionen mit zwei Veraenderlichen steht die Funktion **Plot3D** zur Verfuegung:

```
Plot3D[x^2 * Sin[y 2 Pi], {x, 0, 1}, {y, 0, 1}]
```

Auch sie kennt viele, den obigen gleiche Optionen.

Oft kommt man in die Lage, Funktionen nicht analytisch, sondern nur als Vektorenlisten gegeben zu haben. Solche Listen von Punkten kann man mittels **ListPlot** plotten. Auch diese Funktion verfuegt ueber mehrere Optionen, erwaehnenswert ist hierbei Joined, welche angibt, ob die einzelnen Punkte verbunden werden sollen oder nicht:

```
liste = Table[{x, Sin[x]}, {x, 0, 1, 0.1}]
```

```
ListPlot[liste, Joined → True]
```

Es gibt noch weitere Moeglichkeiten, Funktionen grafisch darzustellen (yB. **ListPlot3D**, **ContourPlot** ,**ParametricPlot**...). Genaueres entnimmt man am besten der Onlinehilfe...

Gleichungen

Mittels des == Operators kann man Mathemica dazu bringen, Vergleiche auszufuehren. Rueckgabe ist jeweils **True** oder **False**:

```
5 == 5 (* offensichtlich wahr*)
```

Auch hier kann man natuerlich Symbole verwenden, sei es, dass man einem Symbol eine Gleichung zuweist oder in die Gleichung Symbole verbaut:

```
gl1 = 3 x == 2
```

ganz analog zu analytischen Ausdruecken kann man auch Gleichungen, die Symbole enthalten, auswerten lassen:

```
gl1 /. x → 4
```

Gleichungssysteme koennen ebenfalls angelegt werden. In *Mathematica* bestehen sie einfach aus einer Liste von Gleichungen.

```
glSys = {2 x == y, x + 5 == 3 y}
```

Es moeglich, fuer eine bestimmte Gleichung oder ein Gleichungssystem die Loesungsmenge bestimmen zu lassen, sofern die Bedingungen dafuer gegeben sind (also zB. mindenstens so viele Gleichungen wied Unbekannte zur Verfuegung stehen...). Dies geschieht mittels **Solve[] . Solve[]** liefert dabei jeweils eine Liste mit Ersetzungsregeln fuer die gesuchten Variablen zurueck.

```
loesung = Solve[glSys, {x, y}]
```

Es kann vorkommen, dass ein Gleichungssystem nur schwer exakt loesbar ist (da sich als Loesung zB. unendliche Dezimalbrueche ergaeben), was dazu fuehren kann, dass Solve kein brauchbares Ergebnis liefert. in diesem Fall bringt oft N[Solve[...]] bzw. in Kurzform NSolve[...] ein gutes numerisches Ergebnis:

```
N[Solve[x^6 - 3 x - 1 == 0, x]]
```

Sollte auch dies nicht zum gewuenschten Ergenis fuehren, kann man mit der Funktion FindRoot[] eine numerische Loesung angeben. Hierbei muss als zweiter Parameter eine Liste bestehend aus den gesuchten Variablen und einem Startwert angegeben werden. Abhaengig vom Startwert erhaelt man dann eine oder auch keine von allen moeglichen Loesungen:

```
FindRoot[E^x == 4 x + x^2, {x, 2}]
```

```
FindRoot[E^x == 4 x + x^2, {x, 1000}]
```

Es ist moeglich, aus einem System eine bestimmte Variable eliminieren zu lassen. Man erhaelt dann Bedingungen, fuer die die entsprechende Variable eliminierbar ist:

```
Eliminate[{a x + b == 0, b x == c}, x]
```

Solve[] bereitet das Problem, dass die Loesungen nicht unbedingt allgemein gueltig sind. will man die Bedingungen abfragen ,fuer die das Gleichungssystem fue alle bel. Werte fue die gesuchten Variablen loesbar ist, kann man dies mit **SolveAlways[]** tun:

```
Solve[a x + b == 0, x] (* gefundene Loesung im Fall von a = 0 nicht definiert *)
```

```
SolveAlways[{a x + b == 0}, x]
```

```
Eliminate[{a x + b == 0}, x]
```

Analysis

Ableitungen von Funktionen koennen auf verschiedene Arten berechnet werden:

```
f[x_] := x^6
∂x f[x](* mittels der Symbole auf der Palette *)
D[f[x], x] (* mittels D[] *)
f'[x]
```

Mehrfache Ableitungen koennen intuitiv gebildet werden:

```
∂x,x f[x](* die zweite Ableitung bilden *)
D[f[x], {x, 3}] (* die vierte Ableitung bilden *)
f''''[x] (* dritte Ableitung bilden *)
```

Gemischte Ableitungen koennen entsprechend berechnet werden:

```
f[x_] := a x^6
∂x,x f[x](* die Ableitung nach a und nach x bilden *)
D[f[x], {x, 3}] (* die vierte Ableitung nach x, und danach nach a bilden *)
f''''[x] (* hier gibt es keine Moeglichkeit nach a abzuleiten *)
```

Stammfunktionen koennen ebenfalls auf mehrere Arten gebildet werden:

```
∫f[x] dx (* mittels der Palette *)
Integrate[f[x], x] (* mittels Integrate[] *)
```

Integrale werden analog berechnet:

```
∫_0^1 f[x] dx
Integrate[f[x], {x, 0, 1}]
```

Integrale, die nicht analytisch gebildet werden koennen, koennen numerisch geloest werden:

```
N[Integrate[E^(-x^2), {x, 1, E}]]
```

Auch uneigentlich kann gerechnet werden:

```
Integrate[E^(-x^2), {x, 0, ∞}]
```

Limites sind wie folgt zu behandeln:

```
Limit[Sin[x] / x, x → 0]
Limit[Tan[x], x → Pi / 2, Direction → 1] (* Limes von unten *)
Limit[Tan[x], x → Pi / 2, Direction → -1] (* Limes von unten *)
```

Fuer den Umgang mit Potenzreihen steht uA. Folgendes zr Verfuegung:

```
∑_{n=1}^{10} 1 / n x^n (* endliche Potenzreihe /  Polynom *)

∑_{n=0}^{∞} x^n / n! (* unendliche Potenzreihe / Potenzreihe der Exponentialfunktion*)

Series[Exp[x], {x, x0, 5}] (* Taylorentwicklung um en Punkt x0 bis zum 5. Glied *)

Normal[Series[Exp[x], {x, x0, 5}]] (* Wie oben,
Normal[] sorgt dafuer, dass O[] nicht auftaucht *)
```

Differentialgleichungen

Differentialgleichungen werden in *Mathematica* aehnlich wie algebraische Gleichungen dargestellt:

```
dgl = f'[x] == f[x]
```

An Stelle des Striches f'[x] kann man auch D[f[x],x] oder $\partial_x f[x]$ benutzen.
Differentialgleichungen koennen selbstverstaendlich auch hoeherer Ordnung und/oder partielle Differentialgleichungen sein:

```
dgl2 = D[g[x, y], x, x] == g[x, y] - D[g[x, y], y]
```

So wie algebraische Gleichungen mittels Solve[] bzw. NSolve[] geloest werden koennen, kann man DGLen mit DSolve[] (analytische) oder NDSolve[] (numerisch) loesen lassen. Das Ergebnis ist wie bei Solve[] eine Liste mit Ersetzungsregeln.

```
DSolve[dgl, f[x], x] (* der erste Parameter ist die DGL,
der zweite die gesuchte Funktion, der dritte die Variable, nach der differenziert wird *)

DSolve[{dgl, f[0] == 1}, f[x], x] (* Bei einem Anfangswertproblem
 werden einfach mehrere Gleichungen uebergeben ( hier zus. f[0]==1 )*)
```

Bei NDsolve[] ist zu beachten, dass numerische Loesungen sich immer nur ueber endliche Intervalle berechnen lassen, weshalb NDSolve[] als letzten Parameter anstatt nur der Variable zusaetzlich einen Wertebereich erwartet, auf dem die Loesung berechnet

werden soll. Auch muss die Loesung des Problemes eine eindeutige Funktion und keine Funktionenschar sein, das Problem muss also eindeutig gestellt werden (zB. als Anfangswertaufgabe). Das Ergebnis ist ein InterpolatingFunction - Objekt, welches wie eine ganz normale Funktion behandelt werden kann, mit der Einschraenkung, dass es nur auf dem NDSolve[] uebergebenen Loesungsintervall definiert ist:

```
loesung = NDSolve[{dgl, f[0] == 2}, f[x], {x, 0, 1}]
 (* Berechnung der Loesung von f'[x]==f[x] auf dem Intervall [0;1] mit f[0]=2*)

loesFunc = f[x] /. loesung[[1]] (* Loesung aus der Liste schaelen ...*)

loesFunc /. x → 0.2  (* 0.2 liegt im Loesungsintervall [0;1], daher kein Problem *)
```

"Prozeduren"

Oft ist es sinnvoll, logisch zusammengehoerende Codepasagen zusammenzufassen. Dazu bestehen mehrere Moeglichkeiten, die teilweise auch die Verwendung lokaler Variablen gestatten. Allen diesen Moeglichkeiten ist gemeinsam, d.h. alles notwendige wird als Parameter uebergeben. Der Einsatz erfolgt als "Aufruf" entsprechend jeder anderen *Mathematica* - Funktion, wobei als erster Parameter eine Liste von lokalen Variablen und als zweiter Parameter die zusammenzufassenden Anweisungen zu uebergeben ist. Die einzelnen Anweisungen sind dabei durch Semikolon zu trennen.
Im Einzelnen sind die Moeglichkeiten:

- **With[]**: Gestattet die Zusammenfassung mehrer Anweisungen zu einem Segment und die Deklaration nur in diesem Segment gueltiger Konstanten.

```
With[{t = x - 2}, Print[t]; Print[t + t^2];]
(* t ist die einzige hier lokal deklarierte Konstante *)
```

- **Module[]**: Im Gegensatz zu With[] sind alle mit Module lokal deklarierten Variablen echte Veraenderliche, dabei wird bei jeder Ausfuehrung und jede Variable ein neues einzigartiges Symbol generiert.

In[7]:=
```
Module[{t = 1, s, u}, s = t + 1 ; t++;
 (* Die Werte von s und t zu aendern waere in einem With-Block unmoeglich *)
 Print[t, " ", s, " ", u];
]
```

- **Block[]** : Normalerweise die Methode der Wahl. Ist aehnlich wie **Module[]**, wobei fuer jede lokale Variable nur ein Symbol generiert wird. Der Unterschied wird deutlich, wenn man das obige **Module[]**-Segment und das folgende **Block[]**-Segment mehrmals aufruft und die Ausgaben fuer **u** vergleicht.

In[6]:=
```
Block[{t = 1, s, u},
 s = t + 1;
 t++;
 Print[s, " ", t, " ", u];
]
```

Eine hilfreiche Moeglichkeit, die diese Segmentanweisungen bieten, ist die Deklaration von Funktionen, die mehr als nur eine Anweisung ausfuehren (vgl. Funktionen in Java, C).

In[4]:=
```
polynom2[hvec2_] := Block[{m, x, hvec}, hvec = Table[x^m, {m, 0, Length[hvec2] - 1}]; hvec.hvec2]
```

In[5]:=
```
polynom2[{3, 2, 4, 1}]
```

Das obige Segment macht Gebrauch von der Eigenschaft der Segmentanweisungen, die Rueckgabe des letzten Befehls im Segment als Rueckgabewert des Segments zurueckzugeben, soweit der Befehl nicht mit Semikolon abschliesst. Man kann auch explizit einen Rueckgabewert angeben, indem man **Return[]** benutzt:

In[8]:=
```
hvec2 = {2, 1, 3, 5}
```

In[12]:=
```
Block[{m, x, hvec}, hvec = Table[x^m, {m, 0, Length[hvec2] - 1}]; hvec.hvec2]
(* implizite Rueckgabe *)

Block[{m, x, hvec}, hvec = Table[x^m, {m, 0, Length[hvec2] - 1}]; Return[hvec.hvec2];]
(* explizite Rueckgabe *)
```

Kontrollstrukturen

Alle aus anderen Sprachen bekannten Schleifenarten existieren auch in *Mathematica.* Wie bei den Segmentierungsanweisungen sind diese auch hier wieder als Funktionsaufrufe realisiert. Im Einzelnen sind dies:

- **Do[]** - Schleifen besitzen einen aehnlichen Aufruf wie die **Table[]** - Anweisung. Als erster Parameter erwartet sie die auszufuehrenden Anweisungen, der zweite Parameter ist eine Liste, der eine Laufvariable sowie Start- und Endwert von bzw. bis zu dem die Laufvariable hochgezaehlt werden soll, sowie uU. eine Schrittweite.

```
Do[Print[i], {i, 0, 3}]; Print["Zweiter Tei:l"];
Do[Print[i], {i, 3, 0, -1}] (* Zaehlt nun abwaerts !*)
```

- **For[]** - Schleifen wird als erster Wert die Initialisierung der Zaehlvariable, als zweiter ein zu **True** oder **False** auswertbarer Ausdruck, als dritter die an der Zaehlvariable nach jedem Durchlauf durchzufuehrende Aenderung (die Zaehlvariable wird NICHT automatisch hochgezaehlt !) sowie als letzter die in der Schleife liegenden Befehle ausgefuehrt.

In[10]:=
```
For[i = 0, i ≤ 3, i++,
 Print[i];]
```

- **While[]** - Schleifen erhalten als ersten Parameter einen zu **True** oder **False** auswertbaren Audruck, gefolgt von den Anweisungen, die solange ausgefuehrt werden, bis der Ausdruck **False** wird.

In[12]:=
```
i = 0;
```

In[13]:=
```
While[i ≤ 3,
 Print[i++];
]
```

Es besteht bei allen Schleifenarten die Moeglichkeit, die Schleife mittels **Break[]** abzubrechen, will man nur einen Schleifendurchlauf vorzeitg beenden, so benutzt man **Continue[]**.

Mathematica kennt ebenfalls bedingte Anweisungen:

- **If[]** erwartet als ersten Parameter einen zu **True** oder **False** auswertbaren Ausdruck. Im Falle von True werden die als zweiter Parameter uebergebenen Anweisungen, sonst die als dritter Parameter uebergebenen Anweisungen ausgefuehrt. Der dritte Parameter ist optional.

In[14]:=
```
maximum[x_, y_] := If[x > y, x , y]
```

- **Switch[]** erwartet als ersten Parameter einen Ausdruck A, gefolgt von einer geraden Zahl weiterer Parameter. Dabei bilden jeweils zwei aufeinanderfolgende Parameter ein Paar. Jedes derartige Paar wird wie folgt interpretiert: Ist der A gleich dem Wert, der als erster Parameter uebergeben wurde, so werden die als zweiter Parameter uebergebenen Anweisungen ausgefuehrt.

```
c[x_] := Switch[x, 0, 1, 1, 0, _, 1] (* "_" ist der Defaultfall,
 diese Funktion liefert also: 1, wenn x==0, 0 wenn x==1, sonst -1 *)
```

- **Which[]** ist **Switch[]** sehr aehnlich. Es erwartet jedoch lediglich eine geradzahlige Parameterzahl, gruppierbar in aufeinanderfolgenden Paaren, wobei der jeweils erste Parameter einen zu **True** oder **False** auswertbaren Ausdruck A darstellt, zweite Parameter alle Befehle enthaelt, die ausgefuehrt werden, wenn A **True** ist. Nach dem ersten **True** evaluiertem Ausdruck bricht die Funktion ab.

```
d[x_] := Which[x < 0, 0, x < 1, 1, x, 2, 2, True, -1] (* "True" am Ende ist der Defaultfall,
 da which nur dahin gelangt, wenn sonst alle Ausdruecke False sind. Liefert 0,
 wenn x < 0, 1, wenn x im Intervall 0<x<1 liegt, 2 wenn im Intervall 1<x<2 und -1 sonst *)
```

Dateien & Packages

Mathematica beherrscht den Umgang mit Dateien und Verzeichnissen, es kann diese anlegen, auslesen, beschreiben etc.Hier werden nur einige wenige grundlegende befehle dazu aufgefuehrt:

```
Put[3 x + 4, "hilf.txt"] (* Schreibt "3x+4" in die Datei "hilf.txt",
wenn die Datei existiert, wird sie gloescht *)
3 x + 4 >> "hilf.txt"(* Kurzschreibweise von oben *)
Get["hilf.txt"] (* liest den Inhalt der Datei "hilf.txt" wieder ein *)
<< "hilf.txt" (* kurz fuer obigen Befehl *)
PutAppend[3 x + 4, "hilf.txt"] (* Wie Put[], allerdings wird hier
 an eine bestehende Datei angehaengt anstatt diese zu ueberschreiben *)
FilePrint["hilf.txt"] (* Gibt den Inhalt von "hilf.txt" aus *)
DeleteFile["hilf.txt"] (* loescht "hilf.txt" von der Platte *)
```

Um eine Datei einlesen zu koennen, muss sie in dem Verzeichnis liegen, auf welchem *Mathematica* aktuell arbeitet. Dazu gibt es :

```
Directory[] (* liefert den Namen des aktuellen Verzeichnisses *)
SetDirectory["c:\mat"] (* Wechselt ins Verzeichnis "c:\Eigene Dateien\mat"  *)
FileNames[] (* Liefert eine Liste aller Dateien im aktuellen Verzeichnis *)

(* Beispiel, wie man eine Funktion in eine Datei speichern kann *)
maximum[x_, y_] := If[x > y, x, y]
Save["maximum.txt", maximum]
```

Natuerlich existieren daneben noch sehr viel mehr Moeglichkeiten, mit Dateien umzugehen, insbesondere durch das Konzept der Stroeme (vermutlich aus Java bekannt). Wer mehr dazu erfahren will, kann sich unter **Stream** in der Onlinehilfe schlauer machen.

Ein Package ist zunaechst nicht viel mehr als eine Datei, in welcher mehrere Funktionen usw. gesammelt liegen und die die Endung **.m** besitzt. Der grundsaetzliche Aufbau eines Packages sollte wie folgt aussehen:

```
BeginPackage["Nullstellen‘"]; (* Der Name,
unter dem das Package abgespeichert wird, sollte dem hier angegeben entsprechen *)
fpNuSt[funk_, x0_] := Block[{g, x},
  g[x_] := funk[x] + x;  (* Die Nullstelle der Funktion
   funk zu berechnen entspricht dem Berechnen des Fixpunktes von g *)
  fixPunkt[g, x0]  (* entsprechend wird als Ergebnis der Fixpunkt von g zurueckgegeben *)
 ]

Begin["Private‘"] (* Die folgenden Sachen sind nur fuer den internen Gebrauch *)

fixPunktHilf[funk_, x0_, n_] := Block[{},

  If[(Abs[x0 - funk[x0]] < diff) || (n ≥ maxn),

   Return[funk[x0]],
   Return[fixPunktHilf[funk, funk[x], n + 1]]
  ]
 ]

fixPunkt[funk_, x0_] := fixPunktHilf[funk, x0, 1]
End[]; (* Der private Teil ist zu Ende *)
EndPackage[]; (* Das Package ist zu Ende *)
```

Ein Package kann, sofern es in einem der Suchverzeichnisse von *Mathematica* oder dem aktuellen Arbeitsverzeichnis liegt, wie gewohnt mittels **Needs[]** oder << geladen werden.

```
Needs["Nullstellen‘"] (* Geht nur wenn der obige Teil
  in einer separaten Datei "Nullstellen.m" gespeichert wurde ...*)
```

Nun kann man alle oeffentlichen Funktionen aus "Nullstellen" benutzen (d.h. hier **fpNuSt[]**)

Man kann sich zu jederFunktion die Definition anschauen:

```
? fpNuSt
```