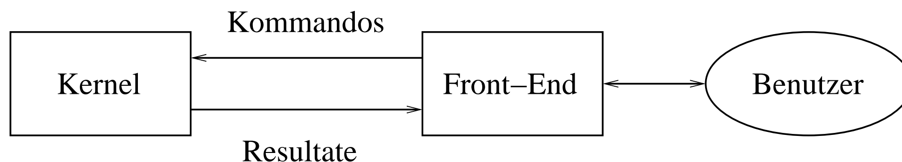
Mathematica

Mathematica ist ein weit verbreitetes Computeralgebrasystem, dessen Entwicklung in den 80iger Jahren von Stephen Wolfram initiiert und das im Jahre 1988 in einer ersten Version vorgestellt wurde. Inzwischen (2008) liegt *Mathematica* in der Version 6.0 vor und ist auf praktisch allen Computersystemen verfügbar. Laut Wolfram Research, `http://www.wolfram.com/`, ist *Mathematica* "A System for Doing Mathematics by Computer", also ein Programm, um Mathematik am Computer zu betreiben:

- Taschenrechner
- Numerische Berechnungen
- Symbolische Berechnungen
- Graphikwerkzeug
- Höhere Programmiersprache (Interpreter)
- Kommunikation mit externen C/C++ Programmen (*MathLink*)
- Schnittstelle zu Java (*J/Link*) und .NET (*.NET/Link*)

Als höhere Programmiersprache vereinigt *Mathematica* die Elemente vieler Programmiersprachen: *Prozedurale* Elemente aus Fortran, C und Pascal; *funktionale* und *regelbasierte* Elemente aus den AI-Sprachen LISP, Prolog und APL; *objektorientierte* Elemente aus Smalltalk und C++. Verwandte Systeme sind *Maple, Macsyma, Derive, Mathcad*, sowie die rein numerischen und matrix-orientierten Pakete MATLAB, *Octave, Scilab*. *Mathematica* ist besonders geeignet für eine komfortable (d.h. interaktive) Bearbeitung kleiner bis mittelgroßer Aufgaben am PC bzw. auf einer Workstation. *Mathematica* ist nicht gedacht für das effiziente Lösen sehr großer numerischer Probleme im Produktionsbetrieb. *Mathematica* besteht aus mehreren getrennten Programmen, die miteinander kommunizieren:



Kernel	:	Führt die eigentlichen Berechnungen durch.
Front-End	:	Benutzeroberfläche.
		- textbasiert
		- graphisch (*Notebook*)

Auf fast allen Systemen wird die textbasierte Version von *Mathematica* mit dem Befehl `math` von der Shell gestartet (der Befehl `mathematica` startet die Notebook-Version). *Mathematica* meldet sich mit `In[1]:=` und erwartet eine Eingabe, die mit RETURN abgeschickt wird (in der Notebook-Version mit SHIFT-RETURN bzw. mit ENTER). Der Befehl `Quit` beendet *Mathematica*, STRG-C (Notebook-Version: ALT-.) bricht eine laufende Berechnung ab, STRG-K (Notebook) vervollständigt Funktionsnamen.

Da *Mathematica* eine interpretierende Programmiersprache ist, wird die Eingabe sofort bearbeitet und das Ergebnis in der Form `Out[..]= ...` am Bildschirm ausgegeben. Dabei werden alle Ausdrücke, die vorher in derselben Sitzung definiert wurden, berücksichtigt. Alle Ein- und Ausgaben werden automatisch als `In[`*n*`]` bzw. `Out[`*n*`]` numeriert und gespeichert und können durch Angabe dieser Symbole weiterverarbeitet werden:

`%`	das letzte ausgegebene Resultat
`%%`	das vorletzte ausgegebene Resultat
`%n`, `Out[n]`	das Resultat der Ausgabezeile `Out[n]`
`In[n]`	`n`-te Eingabezeile zur Neuauswertung

Mathematica unterscheidet zwischen Groß- und Kleinschreibung. *Mathematica*-interne Symbole (wie Kommandos, Funktionen, Konstanten) beginnen generell mit einem Großbuchstaben (z.B. `Sin[x]`, `Solve[]`, `Pi`). Benutzerdefinierte Symbole sollten daher immer mit einem Kleinbuchstaben beginnen und keinen Unterstrich “`_`” enthalten (er wird von *Mathematica* zur Kennzeichnung von Mustern verwendet).

Viele Kommandos sind integraler Bestandteil von *Mathematica*. Zusätzliche Kommandos sind in *Paketen* definiert. Diese müssen geladen werden, *bevor* das erste Kommando daraus verwendet wird. Dafür existieren zwei Möglichkeiten:

`<<`*Paketname*`‘`
`Needs["`*Paketname*`‘"]`

Die zweite Möglichkeit ist besser, da hier getestet wird, ob das Paket bereits geladen ist. Beispiel: `Needs["FourierSeries‘"]`.

Hilfe zu *Mathematica* erhält man mit dem Fragezeichen:

`?Log`	Informationen zu `Log[x]`
`??Log`	mehr Informationen zu `Log[x]`
`?L*`	alle Funktionen, die mit `L` beginnen

Eine ausführliche Dokumentation (inklusive *Mathematica*-Buch) ist über das `Help`-Menü der Notebook-Version verfügbar bzw. auf `http://documents.wolfram.com/` . *Mathematica* Information Center: `http://library.wolfram.com/` .

Da in der textbasierten Version die Möglichkeiten zur Bearbeitung der Kommandozeile sehr eingeschränkt sind, empfiehlt es sich, die Kommandos mit einem Texteditor in eine Datei `file.m` zu schreiben und diese dann in *Mathematica* einzulesen und auszuführen:

```
<<file.m
```

1 Arithmetik

Die Grundrechenarten `+, -, *, /` haben die üblichen Prioritäten, runde Klammern dienen zur Gruppierung von Ausdrücken. Das Multiplikationszeichen `*` kann durch ein Leerzeichen ersetzt werden.

```
In[1]:= (3 + 4)*5
Out[1]= 35

In[2]:= (3 + 4) 5
Out[2]= 35
```

Potenzen b^a schreibt man als `b^a`.

```
In[3]:= 2^10
Out[3]= 1024
```

In *Mathematica* gibt es ganze Zahlen (`35`), rationale Zahlen (`12/5`), Gleitpunktzahlen (`3.14159`) und komplexe Zahlen (`2 + I`). Einige mathematische Konstanten sind vordefiniert: `Pi` (π), `E` (e), `I` ($\sqrt{-1}$), `Degree` ($\pi/180$), `Infinity` (∞).

```
In[4]:= (2 + I)^3
Out[4]= 2 + 11 I
```

Mathematica rechnet grundsätzlich symbolisch und liefert das exakte Resultat mit beliebiger Genauigkeit.

Sehr große (kleine) Gleitpunktzahlen, wie z.B. 6.022×10^{23}, müssen als *Zehnerpotenz* eingegeben werden, `6.022*10^23`, und nicht, wie in vielen anderen Programmiersprachen, als `6.022e+23`.

```
In[5]:= 1 + 1/3

        4
Out[5]= -
        3

In[6]:= 35!
Out[6]= 10333147966386144929\
        666651337523200000000
```

Eine numerische Auswertung (Gleitpunktapproximation) muß man explizit verlangen. Wenn nicht anders angegeben, werden numerische Berechnungen in 16-stelliger Genauigkeit ausgeführt, Ergebnisse aber mit 6 Dezimalstellen angezeigt.

```
In[7]:= N[1 + 1/3]
Out[7]= 1.33333

In[8]:= N[35!, 18]
                                    40
Out[8]= 1.03331479663861449 10
```

Mathematica-interne Namen von Kommandos, Funktionen und Konstanten beginnen mit einem Großbuchstaben. Argumente von Funktionen werden in eckige Klammern `[..]` gesetzt.

```
In[9]:= Sin[Pi/4]

           1
Out[9]= -------
        Sqrt[2]
```

`%` steht für die letzte Ausgabe, `%%` für die vorletzte, `%n` für `Out[n]`.

```
In[10]:= N[%]
Out[10]= 0.707107
```

Enthält ein Ausdruck bereits einen numerischen Wert, so erfolgt die numerische Auswertung automatisch.

```
In[11]:= Sin[Pi/4.0]
Out[11]= 0.707107
```

`N[]` kann auch als nachgestelltes Kommando (*Postfixnotation*) verwendet werden.

```
In[12]:= Sin[Pi/4] // N
Out[12]= 0.707107
```

2 Variablen und symbolische Berechnungen

Der Variablen (dem Symbol) `n` wird der Wert `4` zugewiesen (syntaktisch handelt es sich um eine *Definition* des *Symbols* `n`). Dieser Wert von `n` wird nun in einem Ausdruck verwendet.

```
In[1]:= n = 4
Out[1]= 4
In[2]:= 1 + n + n^2
Out[2]= 21
```

Während in C oder F alle Symbole *vor* ihrer Verwendung deklariert werden müssen, versucht *Mathematica*, alle Symbole mit den vorhandenen Definitionen auszuwerten. Ein Symbol existiert, sobald sein Name zum ersten Mal in einem Kommando oder in einem Ausdruck verwendet wird.

Der Wert von `x` ist ein anderer Ausdruck. Ein Strichpunkt am Zeilenende unterdrückt die nächste Ausgabe. Auch dieser Wert von `x` wird verwendet.

```
In[3]:= x = a + b;
In[4]:= x^2
                   2
Out[4]= (a + b)
```

Mit dem Befehl `Expand[]` werden Produkte und Potenzen ausmultipliziert und als Summe von Termen dargestellt.

```
In[5]:= Expand[%]
         2             2
Out[5]= a  + 2 a b + b
```

Das Symbol `x` ist noch mit dem Wert `a + b` belegt.

```
In[6]:= ?x
Global`x
x = a + b
```

Der Befehl `Clear[x]` löscht den *Inhalt* des Symbols `x`.

```
In[7]:= Clear[x]
In[8]:= ?x
Global`x
```

Der Befehl `Remove[x]` löscht das Symbol `x` selbst aus dem Benutzerkontext `Global``.

```
In[9]:= Remove[x]
In[10]:= ?x
Information::notfound:
Symbol x not found.
```

Mathematica speichert alle Symbole in sogenannten Kontexten: Vom Anwender definierte Symbole werden dem Kontext `Global`` zugeordnet, von *Mathematica* definierte Symbole dem Kontext `System``. Der Befehl `Remove["Global`*"]` entfernt sämtliche vom Benutzer definierte Symbole.

Der Befehl `Factor[`*expr*`]` reduziert den Ausdruck *expr* wieder zu einem Produkt von Faktoren. Der Befehl `Simplify[]` führt hier zum gleichen Resultat; ansonsten versucht `Simplify[]`, einen Ausruck durch die kleinste Anzahl von Termen darzustellen. Mit `FullSimplify[]` erhält man noch weitere Vereinfachungen.

```
In[11]:= Expand[ (1 + x)^3 ]
                         2    3
Out[11]= 1 + 3 x + 3 x  + x

In[12]:= Factor[%]
                   3
Out[12]= (1 + x)
```

Um einen Ausdruck für einen bestimmten Wert eines Symbols `x` auszuwerten, ohne das Symbol `x` selbst bleibend zu verändern ("Einsetzen" eines speziellen Wertes für `x`), verwendet man den **Ersetzungsoperator** `/.` zusammen mit einer **Ersetzungsregel** für `x`:

ausdruck `/.` `x ->` *wert*

Dabei wird die Ersetzungsregel `x ->` *wert* mit dem Operator `->` für vorübergehende Zuweisungen gebildet. Zwischen `/` und `.` sowie zwischen `-` und `>` dürfen keine Leerzeichen stehen! Beispielsweise liefern Kommandos wie `Solve[]` zum Lösen von Gleichungen ihre Ergebnisse als Ersetzungsregeln. Der Operator `->` wird auch zum Einstellen von Optionen verwendet, z.B. beim `NIntegrate[]`-Kommando zur numerischen Integration: `NIntegrate[f, {x, xmin, xmax}, WorkingPrecision -> 20]`.

Der Ausdruck `1 + 2 x` wird für `x = 3` ausgewertet, ohne das Symbol `x` bleibend zu verändern.

```
In[13]:= 1 + 2 x  /.  x -> 3
Out[13]= 7
```

Der Ausdruck `t` kann für verschiedene Werte von `x` ausgewertet werden (und z.B. anderen Symbolen `u` und `v` als neuer Wert zugewiesen werden).

```
In[14]:= t = 1 + x^2;
In[15]:= u = t /. x -> 2
Out[15]= 5
In[16]:= v = t /. x -> 5a
                       2
Out[16]= 1 + 25 a
```

3 Listen

Eine *Liste* dient in *Mathematica* dem Zusammenfassen von Ausdrücken, um diese als eine Einheit zu behandeln, z.B. um eine (mathematische) Funktion auf alle Listenelemente anzuwenden oder die gesamte Liste einer Variablen als Wert zuzuweisen. Listen stellen eine wichtige Struktur in *Mathematica* dar: Vektoren und Matrizen werden als Listen dargestellt und viele Kommandos erwarten Parameter in Listenform bzw. geben Ergebnisse als Listen zurück.

Eine Liste kann man durch Aufzählung ihrer Elemente erzeugen. Diese müssen durch Beistriche getrennt und in geschwungene Klammern eingeschlossen werden. Listen können beliebig tief verschachtelt werden, d.h. Elemente von Listen können wieder Listen sein.

```
In[1]:= {3, 5, 1}
Out[1]= {3, 5, 1}

In[2]:= {{a, b}, {c, d}}
Out[2]= {{a, b}, {c, d}}
```

Eine Liste kann auch mit der *Mathematica*-Funktion `Table[]` erzeugt werden.

```
In[3]:= Table[ i^2, {i, 6} ]
Out[3]= {1, 4, 9, 16, 25, 36}
```

Mit dem Kommando `Table[]` werden Listen nach einer beliebigen Vorschrift gebildet. Im ersten Parameter wird die Funktion zur Bildung der einzelnen Listenelemente angegeben, im zweiten Parameter der *Wertebereich* für die Laufvariable als sogenannter *Listeniterator* (der selbst wieder aus einer Liste mit maximal 4 Einträgen besteht):

`Table[` *funktion*`,` `{`*var, start, ende, schrittweite*`}` `]`

Listeniteratoren werden in allen Kommandos verwendet, bei denen Wertebereiche angegeben werden müssen. Für einen Listeniterator sind folgende Formen möglich:

- `{n}` erzeugt `n` identische Einträge
- `{i, n}` variiert `i` von `1,2,...,n`
- `{i, a, n}` variiert `i` von `a,a+1,...,n`
- `{i, a, n, s}` variiert `i` von `a,a+s,a+2s,...,e` mit Schrittweite `s` und `e`$\leq$`n`

`Table[]`-Kommandos mit Listeniteratoren.

```
In[4]:= Table[a, {8}]
Out[4]= {a, a, a, a, a, a, a, a}

In[5]:= Table[i^2, {i, 0, 5}]
Out[5]= {0, 1, 4, 9, 16, 25}

In[6]:= Table[i^2, {i, 0, 1, 0.3}]
Out[6]= {0, 0.09, 0.36, 0.81}
```

Die meisten Rechenoperationen können mit Listen ausgeführt werden. Viele Grundfunktionen mit nur einem Argument werden automatisch auf die Elemente von Listen angewendet, wenn das Attribut `Listable` gesetzt ist:

```
In[7]:= ??Exp
Exp[z] is the exponential function.
Attributes[Exp] = {Listable, NumericFunction, Protected}
```

Diese Listen werden elementweise subtrahiert.

```
In[8]:= {2, 4, 6} - {1, 2, 3}
Out[8]= {1, 2, 3}
```

Multiplikation mit einem Skalar.

```
In[9]:= 3 * {1, 2, 3}
Out[9]= {3, 6, 9}
```

Jedes Listenelement wird quadriert und zu jedem Resultat wird `1` addiert.

```
In[10]:= {3, 5, 1}^2 + 1
Out[10]= {10, 26, 2}
```

`Sin[]` wird elementweise angewendet.

```
In[11]:= Sin[ {1, 2, 3} ]
Out[11]= {Sin[1], Sin[2], Sin[3]}
```

Dem Symbol v wird eine Liste als Wert zugewiesen. Die mathematische Funktion Exp[] wird auf die gesamte Liste angewendet.

```
In[12]:= v = {3., 4, 5.};
In[13]:= Exp[v]
                        4
Out[13]= {20.0855, E , 148.413}
```

Einzelne Elemente einer Liste v werden durch einen Index i in doppelten eckigen Klammern, v[[i]], angesprochen.

```
In[14]:= v[[3]]
Out[14]= 5.
In[15]:= v[[3]] = 0;
In[16]:= v
Out[16]= {3., 4, 0}
```

Listen können auch mit symbolischen Ausdrücken verwendet werden.

```
In[17]:= j = {2, 3};
In[18]:= x^j - 1
                 2         3
Out[18]= {-1 + x , -1 + x }
In[19]:= % /. x -> 3
Out[19]= {8, 26}
```

Ein *Vektor* wird als Liste dargestellt.

```
In[20]:= v = {x, y, z}
Out[20]= {x, y, z}
In[21]:= TableForm[v]
Out[21]//TableForm= x
                    y
                    z
```

Eine *Matrix* wird zeilenweise als eine Liste von Listen (Matrix-Zeilen) dargestellt. i läuft über die Zeilen, j jeweils über die Spalten.

```
In[22]:= m = Table[i - j,
                  {i, 2}, {j, 2}]
Out[22]= {{0, -1}, {1, 0}}
```

Erste Teilliste (erste Zeile) der Matrix.

```
In[23]:= m[[1]]
Out[23]= {0, -1}
```

2. Element der ersten Teilliste (m_{12}).

```
In[24]:= m[[1,2]]
Out[24]= -1
```

Ausgabe der Liste m als Matrix.

```
In[25]:= MatrixForm[m]
Out[25]//MatrixForm= 0  -1
                     1   0
```

Mehrere Ersetzungsregeln werden gemeinsam angewendet, wenn sie in eine Liste zusammengefaßt werden.

```
In[26]:= (x + y)^2  /.
         {x -> a, y -> b}
                2
Out[26]= (a + b)
```

Durch *verschachtelte* Listen können gleichzeitig verschiedene Werte eingesetzt werden. Als Erbebnis erhält man eine Liste.

```
In[27]:= (x + y)^2  /.
         {{x -> a, y -> b},
          {x -> 0, y -> c}}
                 2   2
Out[27]= {(a + b) , c }
```

4 Matrizen

`Table[` f, `{`i, m`}`, `{`j, n`}` `]`	erzeugt $m \times n$ Matrix **a** mit $a_{ij} = f(i,j)$
`DiagonalMatrix[` *liste* `]`	erzeugt Diagonalmatrix aus *liste*
`IdentityMatrix[` n `]`	erzeugt $n \times n$ Einheitsmatrix
c `*` a	Multiplikation mit Skalar c (elementweise)
a `.` v	Multiplikation der Matrix **a** mit Vektor **v**
a `.` b	Multiplikation der Matrizen **a** und **b**
`Cross[` u, v `]`	Kreuzprodukt $\mathbf{u} \times \mathbf{v}$ der Vektoren **u** und **v**
`Transpose[` a `]`	Transponierte $\mathbf{a}^\mathsf{T}$ der Matrix **a**
`Inverse[` a `]`	Inverse $\mathbf{a}^{-1}$ der $n \times n$ Matrix **a**
`Det[` a `]`	Determinante der Matrix **a**
`Tr[` a `]`	Spur der Matrix **a**
`MatrixPower[` a, n `]`	n-te Potenz $\mathbf{a}^n$ der Matrix **a**
`MatrixExp[` a `]`	Exponentialfunktion $\exp(\mathbf{a})$ der Matrix **a**

```
In[1]:= u = {u1, u2};
In[2]:= v = {v1, v2};
In[3]:= mat = {{a, b}, {c, d}};

In[4]:= 3 * mat
Out[4]= {{3 a, 3 b}, {3 c, 3 d}}

In[5]:= u . v
Out[5]= u1 v1 + u2 v2

In[6]:= mat . v // MatrixForm
Out[6]//MatrixForm= a v1 + b v2

                    c v1 + d v2

In[7]:= inv = Inverse[mat]
               d               b                  c              a
Out[7]= {{------------, -(------------)}, {-(------------), ------------}}
          -(b c) + a d    -(b c) + a d       -(b c) + a d   -(b c) + a d

In[8]:= mat . inv // Simplify // MatrixForm
Out[8]//MatrixForm= 1   0

                    0   1
In[9]:= Det[mat]
Out[9]= -(b c) + a d
```

`Eigenvalues[` *a* `]`	Liste der Eigenwerte der Matrix **a**
`Eigenvectors[` *a* `]`	Liste der Eigenvektoren der Matrix **a**
`LUDecomposition[` *a* `]`	LU-Zerlegung der $n \times n$ Matrix **a**
`LUBackSubstitution[` *data*, *b* `]`	löst $\mathbf{a} \cdot \mathbf{x} = \mathbf{b}$ aus LU-Zerlegung von **a**

Als Beispiel wollen wir die *Konditionszahl* $||\mathbf{a}|| \cdot ||\mathbf{a}^{-1}||$ der 5×5 Hilbertmatrix bezüglich der 2-Norm $||\mathbf{a}||_2 = \max\{|\lambda_i|\}$ berechnen, wo λ_i die Eigenwerte der symmetrischen Matrix **a** darstellen:

```
In[1]:= h = Table[ 1.0/(i+j-1), {i, 5}, {j, 5} ];

In[2]:= MatrixForm[h]

Out[2]//MatrixForm= 1.          0.5         0.333333    0.25        0.2

                    0.5         0.333333    0.25        0.2         0.166667

                    0.333333    0.25        0.2         0.166667    0.142857

                    0.25        0.2         0.166667    0.142857    0.125

                    0.2         0.166667    0.142857    0.125       0.111111

In[3]:= u = Inverse[h];

In[4]:= MatrixForm[u]

Out[4]//MatrixForm= 25.         -300.       1050.       -1400.      630.

                    -300.       4800.       -18900.     26880.      -12600.

                    1050.       -18900.     79380.      -117600.    56700.

                    -1400.      26880.      -117600.    179200.     -88200.

                    630.        -12600.     56700.      -88200.     44100.

In[5]:= rho1 = Max[ Abs[ Eigenvalues[h] ] ]
Out[5]= 1.56705

In[6]:= rho2 = Max[ Abs[ Eigenvalues[u] ] ]
Out[6]= 304143.

In[7]:= cond = rho1 * rho2
Out[7]= 476607.
```

5 Summen und Produkte

`Sum[` f, `{`i, $imin$, $imax$`} ]`	Summe $\sum_{i=imin}^{imax} f$
`Sum[` f, `{`i, $imin$, $imax$, di`} ]`	Schrittweite di für i
`Sum[` f, `{`i, $imin$, $imax$`}, {`j, $jmin$, $jmax$`} ]`	$\sum_{i=imin}^{imax} \sum_{j=jmin}^{jmax} f$
`Product[` f, `{`i, $imin$, $imax$`} ]`	Produkt $\prod_{i=imin}^{imax} f$

Die Summe $\sum_{i=1}^{5} x^i/i$.

```
In[1]:= Sum[ x^i/i, {i, 1, 5} ]
                2    3    4    5
               x    x    x    x
Out[1]= x + -- + -- + -- + --
               2    3    4    5
```

Das Produkt $\prod_{i=1}^{3}(x+i)$.

```
In[2]:= Product[x + i, {i, 1, 3}]

Out[2]= (1 + x) (2 + x) (3 + x)
```

Mathematica liefert das exakte Resultat für diese Summe.

```
In[3]:= Sum[ 1/i^4, {i, 1, 15} ]

        18250192489014819937873
Out[3]= -----------------------
        16863445484161436160000
```

Eine numerische Näherung erhält man mit der Funktion `N[]`.

```
In[4]:= N[%]
Out[4]= 1.08223
```

Die Doppelsumme $\sum_{i=1}^{2} \sum_{j=1}^{i} x^i y^j$.

```
In[5]:= Sum[ x^i y^j, {i, 1, 2},
                      {j, 1, i}]
                2      2  2
Out[5]= x y + x  y + x  y
```

Die Summe $\sum_{i=1}^{n} i^2$ wird symbolisch ausgewertet.

```
In[6]:= Sum[ i^2, {i, n} ]

        n (1 + n) (1 + 2 n)
Out[6]= -------------------
                 6
```

Mathematica kennt auch die Exponentialreihe $\sum_{n=0}^{\infty} x^n/n!$.

```
In[10]:= Sum[ x^n / n!,
              {n, 0, Infinity} ]
          x
Out[10]= E
```

6 Grenzwerte

`Limit[` f`,` x `->` a `]`	Grenzwert $\lim_{x\to a} f(x)$
`Limit[` f`,` x `->` a`, Direction -> 1 ]`	Linksseitiger Grenzwert $\lim_{x\to a^-} f(x)$
`Limit[` f`,` x `->` a`, Direction -> -1 ]`	Rechtsseitiger Grenzwert $\lim_{x\to a^+} f(x)$
`NLimit[` f`,` x `->` a `]`	$\lim_{x\to a} f(x)$ numerisch

Der Grenzwert $\lim_{x\to 0}\frac{1}{x}$. Normalerweise wird der rechtsseitige Grenzwert berechnet.

```
In[1]:= Limit[ 1/x, x -> 0 ]
Out[1]= Infinity

In[2]:= Limit[ 1/x,
               x -> Infinity ]
Out[2]= 0
```

Die linksseitigen Grenzwerte $\lim_{x\to 0^-}\frac{1}{x}$ und $\lim_{x\to \pi/2^-} \tan x$.

```
In[3]:= Limit[ 1/x, x -> 0,
               Direction -> 1 ]
Out[3]= -Infinity

In[4]:= Limit[ Tan[x], x -> Pi/2,
               Direction -> 1 ]
Out[4]= Infinity
```

`Limit[]` arbeitet symbolisch und kann auch Argumente mit unbestimmten Parametern behandeln.

```
In[5]:= Limit[ (1 + 1/x)^x,
               x -> Infinity ]
Out[5]= E

In[6]:= Limit[ x^n / E^x,
               x -> Infinity ]
Out[6]= 0
```

Bei $\lim_{x\to\infty}(e^x/x!)$ scheiterte *Mathematica* bis zur Version 4. Ab Version 5 wird der korrekte symbolische Wert `0` zurückgegeben.

```
In[7]:= Limit[ E^x / x!,
               x -> Infinity ]
Series::esss: Essential
singularity encountered  ...
```

Jedenfalls läßt sich dieser Grenzwert numerisch berechnen, wenn das Zusatzpaket `NumericalCalculus‘` geladen wird.

```
In[8]:= Needs[
        "NumericalCalculus‘"]
In[9]:= NLimit[ E^x / x!,
                x -> Infinity ]
Out[9]= 0.
```

7 Vergleichsoperatoren und logische Operatoren

Mit *Vergleichsoperatoren* werden zwei (oder mehrere) Ausdrücke verglichen. Das Ergebnis eines solchen logischen Ausdrucks kann *wahr* oder *falsch* sein. In *Mathematica* werden dafür die Symbole `True` und `False` verwendet. Für den Fall, daß der Wahrheitswert nicht festgestellt werden kann, bleibt der gesamte logische Ausdruck unausgewertet. Vergleichsoperatoren werden in *Mathematica* so geschrieben wie in der Programmiersprache C.

ls == *rs*	Test auf Gleichheit
ls != *rs*	Ungleichheit
ls < *rs*	kleiner
ls <= *rs*	kleiner oder gleich
ls > *rs*	größer
ls >= *rs*	größer oder gleich

Dieser logische Ausdruck kann sofort ausgewertet werden.

```
In[1]:= 2 < 5
Out[1]= True
```

Auch ohne konkreten Wert von `x` muß `x` gleich sich selbst sein.

```
In[2]:= x == x
Out[2]= True
```

Da `x` und `y` noch keine Werte haben, kann der Wahrheitswert dieses Ausdrucks nicht festgestellt werden.

```
In[3]:= x == y
Out[3]= x == y
```

Logische Ausdrücke können weiter mit *logischen Operatoren* verknüpft werden, um kompliziertere logische Ausdrücke zu bilden. Auch hier ist die Notation gleich wie in der Programmiersprache C.

!*p*	logische Negation
p && *q* && ...	UND-Verknüpfung
p || *q* || ...	ODER-Verknüpfung
If[*p*, *a*, *b*]	wenn *p* dann *a*, sonst *b*

Bei der UND-Verknüpfung müssen alle Teile `True` ergeben, damit das Resultat `True` ist.

```
In[4]:= 0 <= 2 && 2 < 5
Out[4]= True
```

Bei der ODER-Verknüpfung muß mindestens ein Ausdruck `True` ergeben. Da die Ausdrücke von links nach rechts ausgewertet werden, und da `2 < 5` bereits `True` ist, kommt es gar nicht mehr auf den zweiten an.

```
In[5]:= 2 < 5 || x == y
Out[5]= True
```

Da der logische Ausdruck `5 > 4` den Wert `True` hat, wird `1` zurückgegeben.

```
In[6]:= If[ 5 > 4, 1, 2 ]
Out[6]= 1
```

8 Definition von Funktionen

Mathematica besitzt bereits eine Fülle von eingebauten Funktionen (`Abs[]`, `Exp[]`, `Sin[]`, `Expand[]`, usw.). Daneben gibt es auch die Möglichkeit, eigene Funktionen zu definieren. Zum Beispiel kann die Definition einer Funktion, welche ihr Argument quadriert, in *Mathematica* so aussehen:

```
f[x_] := x^2
```

Funktionsargumente werden in eckigen Klammern angegeben, der Operator für die Definition ist `:=` (ohne Leerzeichen zwischen `:` und `=`). Das Funktionsargument `x_` steht als Platzhalter für ein beliebiges *Muster*, d.h. für "irgend etwas", das man auf der rechten Seite mit dem Namen `x` ansprechen möchte. Im allgemeinen wird in *Mathematica* ein beliebiges Muster, d.h. ein beliebiger Ausdruck, durch einen Unterstrich `_` ausgedrückt. Die Schreibweise `x_` bedeutet, daß dieses "Irgendetwas" mit dem Namen `x` versehen wird, unter dem es dann auf der rechten Seite angesprochen werden kann.

Die Funktion `f` wird definiert und die für das Symbol `f` definierten Regeln werden angezeigt: `f` quadriert ihr Argument.

```
In[1]:= f[x_] := x^2
In[2]:= ?f
Global`f
f[x_] := x^2
```

Für Zahlen als Argumente von `f` wird das Resultat sofort ausgerechnet.

```
In[3]:= f[2]
Out[3]= 4

In[4]:= f[I]
Out[4]= -1
```

Rein symbolische Argumente werden nach der Anwendung von `f` nicht mehr weiter vereinfacht.

```
In[5]:= f[a + b]
                2
Out[5]= (a + b)

In[6]:= f[a + b] // Expand
         2            2
Out[6]= a  + 2 a b + b
```

`Integrate[]` könnte ebensogut mit `f[z]` als Integrand und `z` als Integrationsvariable geschrieben werden.

```
In[7]:= Integrate[f[t], t]
         3
        t
Out[7]= --
        3
```

Eine Funktion mit 2 Argumenten. *Mathematica* merkt sich beide Definitionen von `f` und unterscheidet sie anhand der Argumente. Eine neue Definition überschreibt eine bestehende nur, wenn die linke Seite dieselbe ist.

```
In[8]:= f[x_, y_] := Exp[-x^2-y^2]
In[9]:= f[a, b]
           2    2
         -a  - b
Out[9]= E
```

Die symbolische Auswertung des Mehrfachintegrals $\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} \mathrm{e}^{-x^2-y^2} dxdy$.

```
In[10]:= Integrate[f[x, y],
         {x, -Infinity, Infinity},
         {y, -Infinity, Infinity}]
Out[10]= Pi
```

Mathematica kennt zwei Arten von Definitionen (Zuweisungen):

ls = *rs*	sofortige Definition
ls := *rs*	verzögerte Definition

Der Unterschied besteht darin, *wann* die rechte Seite ausgewertet wird. Bei der *sofortigen Definition* wird die rechte Seite sofort beim Aufstellen (Einlesen) der Definition ausgewertet und dem Symbol auf der linken Seite zugewiesen. Bei der *verzögerten Definition* erfolgt die Auswertung der rechten Seite erst dann (und zwar jedesmal neu), wenn das Symbol auf der linken Seite verwendet wird.

Verzögerte Definition von `f`. Die rechte Seite bleibt zunächst unausgewertet.

```
In[1]:= f[x_] := Expand[(1 + x)^2]
In[2]:= ?f
Global`f
f[x_] := Expand[(1 + x)^2]
```

Sofortige Definition von `g`. Die rechte Seite wird sofort ausgewertet und das Resultat von `Expand` als Funktionsvorschrift gespeichert.

```
In[3]:= g[x_] = Expand[(1 + x)^2]
                     2
Out[3]= 1 + 2 x + x
In[4]:= ?g
Global`g
g[x_] = 1 + 2*x + x^2
```

Erst bei der Verwendung von `f` wird `Expand` ausgeführt.

```
In[5]:= f[y + 2]
                     2
Out[5]= 9 + 6 y + y
```

Bei der Funktion `g` wird das Argument in die bereits expandierte Form eingesetzt.

```
In[6]:= g[y + 2]
                                 2
Out[6]= 1 + 2 (2 + y) + (2 + y)
```

Sofortige Definitionen (`=`) und verzögerte Definitionen (`:=`) können auch für Wertzuweisungen an Variablen verwendet werden:

Die Funktion `Random[]` gibt eine Pseudozufallszahl vom Typ `Real` zwischen 0 und 1 zurück.

```
In[7]:= r1 = Random[]
Out[7]= 0.993679
In[8]:= r2 := Random[]
```

Wiederholter Aufruf von `r1` liefert immer dieselbe Zufallszahl. Bei jeder Verwendung der Variablen `r2` wird die rechte Seite ihrer Definition erneut ausgewertet und daher jedesmal eine andere Zufallszahl erzeugt.

```
In[9]:= {r1, r2}
Out[9]= {0.993679, 0.202585}

In[10]:= {r1, r2}
Out[10]= {0.993679, 0.152187}
```

Faustregeln für Definitionen: Sofortige Definitionen mit = sind dort sinnvoll, wo man ein Symbol (oder ein Muster) als Abkürzung für einen *festen Wert* definieren will. Falls auf der rechten Seite der Definition bei der Anwendung noch etwas *ausgerechnet* werden muß, so verwendet man eine verzögerte Definition mit :=. Funktionen sollten daher fast immer mit verzögerten Definitionen formuliert werden, für Konstanten können sofortige Definitionen gegeben werden.

Mathematica kennt kein eigenes Kommando, um Funktionen stückweise zu definieren. *Stückweise definierte Funktionen* werden durch mehreren Regeln mit nachgestellten Bedingungen an ihre Gültigkeit definiert. Definitionen, die nur unter bestimmten Bedingungen gelten, werden so aufgestellt:

ls := *rs* /; *lausdr*

Die Einschränkung wird mit dem Operator /; ("unter der Bedingung, daß") vorgenommen. *lausdr* ist ein logischer Ausdruck. Die Definition auf der linken Seite wird nur dann verwendet, wenn dieser Ausdruck den Wert True liefert.

So kann eine Sprungfunktion stückweise definiert werden.

```
In[1]:= step[x_] := 0 /; x < 0
In[2]:= step[x_] := 1 /; x >= 0
In[3]:= {step[-1], step[1]}
Out[3]= {0, 1}
```

Alternativ kann die obige Sprungfunktion auch so definiert werden.

```
In[4]:= step[x_] := If[x >= 0, 1, 0]
In[5]:= {step[-1], step[1]}
Out[5]= {0, 1}
```

In der *Standardform* der Funktionsanwendung in *Mathematica* stehen die Argumente in eckigen Klammern hinter dem Funktionsnamen. Bei Funktionen mit *einem* Argument sind noch zwei weitere Notationen möglich: In der *Postfixform* werden Funktionen mit dem Operator // von rechts auf einen Ausdruck angewendet. In der *Präfixform* werden Funktionen mit dem Operator @ von links auf einen Ausdruck angewendet (dabei muß der Ausdruck i.a. in runden Klammern stehen):

f[x, y]	Standardform für f[x, y]
f @ (x)	Präfixform für f[x]
x // f	Postfixform für f[x]

Anwendung der Funktion Expand[] auf das Argument (a+b)^2 in Standardform, Präfixform (Argument in runden Klammern!) und Postfixform. Alle drei Formen sind vollständig äquivalent.

```
In[1]:= Expand[(a+b)^2]
         2           2
Out[1]= a  + 2 a b + b
In[2]:= Expand @ ((a+b)^2)
         2           2
Out[2]= a  + 2 a b + b
In[3]:= (a+b)^2 // Expand
         2           2
Out[3]= a  + 2 a b + b
```

Mathematica stellt für Funktionen noch eine spezielle Konstruktion zur Verfügung, die in den meisten traditionellen Programmiersprachen fehlt, nämlich sogenannte **reine** oder **anonyme Funktionen** (**pure functions**). Die Idee ist dabei, Funktionen, die nur einmal benötigt werden, kompakt darstellen zu können, ohne sie vorher mit einem eigenen Symbol definieren zu müssen. Bei einer reinen Funktion wird nur angegeben, welche Operationen ausgeführt werden sollen und wo die Argumente stehen sollen. Reine Funktionen werden zum Beispiel als Ergebnis beim symbolischen Lösen von Differentialgleichungen mit `DSolve[]` in Form einer Liste von Ersetzungsregeln für die Lösungsfunktion zurückgegeben.

Um eine gewöhnliche Funktion verwenden zu können, muß sie zuerst definiert werden.

```
In[1]:= f[x_] := x^2
In[2]:= f[5]
Out[2]= 25
```

Reine Funktionen werden mit `Function[]` geschrieben, in den Argumenten steht zuerst der Name der Hilfsvariablen (Platzhalter), dann die Operationen, die damit ausgeführt werden sollen. Dieser Ausdruck steht für die Funktion selbst, deren Wert an der Stelle `x` gleich `x^2` ist.

```
In[3]:= Function[x, x^2]

                         2
Out[3]= Function[x, x ]
```

Wird eine reine Funktion auf ein Argument angewendet, so wird dieses für `x` eingesetzt und die Funktionsvorschrift dafür ausgewertet. Damit ist eine vorhergehende Definition nicht notwendig.

```
In[4]:= Function[x, x^2][5]

Out[4]= 25
```

Man kann auch die Ableitung dieser Funktion bestimmen. Das Ergebnis ist eine Funktion, die ihr Argument mit 2 multipliziert.

```
In[5]:= Function[x, x^2]'

Out[5]= Function[x, 2 x]
```

Der Name `x` für den Platzhalter in einer reinen Funktion ist überflüssig. Deshalb kann man ihn durch ein `#` ersetzen und nur noch die Funktionsdefinition schreiben.

```
In[6]:= Function[#^2][5]

Out[6]= 25
```

In einer noch kürzeren Schreibweise wird `Function[]` weggelassen und das Ende der reinen Funktion mit einem `&` markiert.

```
In[8]:= #^2 & [5]

Out[8]= 25
```

Der gleiche Ausdruck in Präfixnotation und in Postfixnotation.

```
In[9]:= #^2 & @ (5)
Out[9]= 25
In[10]:= 5 // #^2 &
Out[10]= 25
```

Bei reinen Funktionen mit mehreren Argumenten werden diese entweder in eine Liste gesetzt (Schreibweise mit `Function[]`) oder mit `#1`, `#2`, ... bezeichnet (Kurzschreibweise mit sogenannten *Slotvariablen*).

```
In[11]:= Function[{x, y},
          Sqrt[x^2 + y^2]][3,4]
Out[11]= 5
In[12]:= Sqrt[#1^2 + #2^2] & [3,4]
Out[12]= 5
```

Man kann natürlich auch Definitionen mit reinen Funktionen schreiben, hier am Beispiel des geometrischen Mittels dreier Zahlen.

```
In[13]:= gm1 = (#1 #2 #3)^(1/3) &;

In[14]:= gm1[a, b, c]
                1/3
Out[14]= (a b c)
```

In *Mathematica* werden viele eingebaute Funktionen automatisch auf die Elemente von Listen angewendet. Diese Funktionen besitzen das Attribut `Listable`. Benutzerdefinierte Funktionen sind im allgemeinen nicht `Listable` und werden daher nicht automatisch auf Listenelemente abgebildet.

```
In[1]:= liste = {a, b, c}
Out[1]= {a, b, c}
```

Die Funktion `Sin[]` wird automatisch auf die Elemente von `{a, b, c}` angewendet.

```
In[2]:= Sin[liste]
Out[2]= {Sin[a], Sin[b], Sin[c]}
```

Die Funktion `g` wirkt nicht separat auf jedes Listenelement. `g[liste]` ergibt 2 anstelle von `{2, 2, 2}`.

```
In[3]:= g[x_] := 2
In[4]:= g[liste]
Out[4]= 2
```

Deshalb gibt es in *Mathematica* die Möglichkeit, Funktionen mit nur *einem* Argument, die nicht `Listable` sind, auf *jedes Element* einer Liste anzuwenden:

`Map[`f`, {`a`, `b`, `c`, ... }]`	ergibt `{`f`[`a`], `f`[`b`], `f`[`c`], ... }`
f `/@ {`a`, `b`, `c`, ... }`	Kurzform in Infixnotation

`Map[]` wendet `g` auf jedes Listenelement an.

```
In[5]:= Map[g, liste]
Out[5]= {2, 2, 2}
```

Kurzschreibweise für `Map[g, liste]`.

```
In[6]:= g /@ liste
Out[6]= {2, 2, 2}
```

Manchmal ist es notwendig, die Elemente einer Liste als *Argumente* einer Funktion zu verwenden. Dafür stellt *Mathematica* die Funktion `Apply[]` zur Verfügung:

`Apply[`f`, {`a`, `b`, `c`, ... }]`	ergibt f`[`a`, `b`, `c`, ... ]`
f `@@ {`a`, `b`, `c`, ... }`	Kurzform in Infixnotation

Die Elemente der Liste `{a, b, c}` werden durch die Funktion `Apply[]` als Argumente von `f` verwendet.

```
In[7]:= Apply[f, liste]
Out[7]= f[a, b, c]
In[8]:= f @@ liste
Out[8]= f[a, b, c]
```

Die Additionsfunktion `Plus[]` wird normalerweise mit dem Operator + geschrieben. Wird sie mit `Apply[]` auf eine Liste angewendet, erhält man die Summe der Elemente.

```
In[9]:= Apply[Plus, liste]
Out[9]= a + b + c
In[10]:= Plus @@ liste
Out[10]= a + b + c
```

Analog erhält man mit der Multiplikationsfunktion `Times[]` das Produkt der Elemente.

```
In[11]:= Apply[Times, liste]
Out[11]= a b c
In[12]:= Times @@ liste
Out[12]= a b c
```

Damit läßt sich eine Variante des geometrischen Mittels als reine Funktion definieren, welche eine Liste mit *beliebiger* Länge als Argument verwendet. `Length[`*liste*`]` liefert die Anzahl der Elemente von *liste*.

```
In[13]:= gm2 =
         (Times@@#)^(1/Length[#])&;
In[14]:= gm2[{a, b, c, d, e}]
                      1/5
Out[14]= (a b c d e)
```

Eine Anmerkung zur Arbeitsweise von Mathematica

Der grundlegende (und einzige) Datentyp von *Mathematica* ist der *Ausdruck* (*expression*): Jede gültige *Mathematica*-Anweisung ist ein Ausruck. Diese Struktur ist dieselbe wie in der Programmiersprache LISP. Intern wird jeder Ausdruck dargestellt als

$$h \texttt{ [} e_1 \texttt{, } e_2 \texttt{, } \ldots \texttt{ , } e_n \texttt{] },$$

wobei h und die e_i selbst wieder Ausdrücke sind. h heißt der *Kopf* (*head*) des Ausdrucks, die e_i sind die *Elemente*.

```
In[1]:= x + 2*y + z^2 // FullForm
Out[1]//FullForm= Plus[x, Times[2, y], Power[z, 2]]
In[2]:= Head[%1]
Out[2]= Plus
In[3]:= Part[%1, 2]
Out[3]= 2 y
```

Formal ist ein Ausdruck entweder ein *Atom* oder ein *zusammengesetzter Ausdruck*. Von den Atomen gibt es nur 3 Arten: *Symbole*, *Strings* (Zeichenketten) und *Zahlen* (ganze Zahlen, rationale Zahlen, Gleitpunktzahlen und komplexe Zahlen). Atome haben ebenfalls einen Kopf, der den Typ des Atoms beschreibt:

```
In[4]:= Map[ Head, {x, "hallo", 3, 2/3, 3.14, 2 + 3*I} ]
Out[4]= {Symbol, String, Integer, Rational, Real, Complex}
```

Die grundlegende Operation von *Mathematica* ist die *Auswertung* (*evaluation*) von Ausdrücken. Dazu werden alle benutzerdefinierten und internen Definitionen (Transformationsregeln) auf einen Ausdruck in einer vorgegebenen Reihenfolge solange angewendet, bis sich der Ausdruck nicht mehr verändert:

```
In[1]:= a = 2; b = 3; a*x + b // Trace
Out[1]= {{{a, 2}, 2 x}, {b, 3}, 2 x + 3, 3 + 2 x}
```

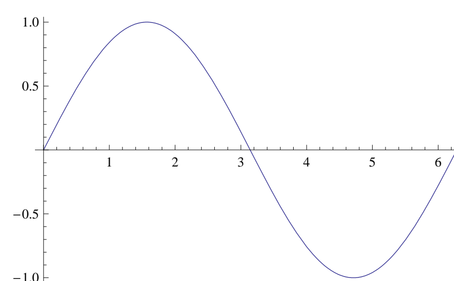
9 Graphik

9.1 Graphen von Funktionen

`Plot[`f`, {`x`, `$xmin$`, `$xmax$`}]`	zeichnet f als Funktion von x im Bereich von $xmin$ bis $xmax$
`Plot[{`f_1`, `f_2`, ... }, {`x`, `$xmin$`, `$xmax$`}]`	zeichnet mehrere Funktionen $f_1, f_2, \ldots$ gleichzeitig
`Plot[Evaluate[`f`], {`x`, `$xmin$`, `$xmax$`}]`	wertet den Ausdruck f vor dem Einsetzen von Zahlenwerten aus

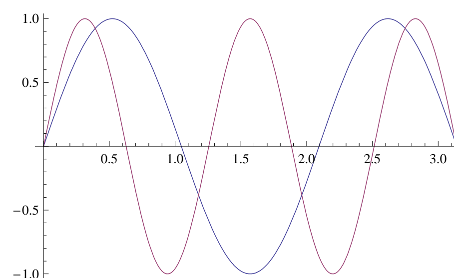
So zeichnet man den Graphen der Funktion $\sin x$ im Bereich von 0 bis 2π. Dabei wird beim Abarbeiten eines `Plot[]`-Kommandos vom Kernel als Rückgabewert eine medienunabhängige textuelle Repräsentation (ein `Graphics`-Objekt) erzeugt und an das Frontend weitergegeben. Erst dort wird dann das Graphikobjekt dargestellt.

```
In[1]:= Plot[Sin[x], {x, 0, 2Pi}]
```



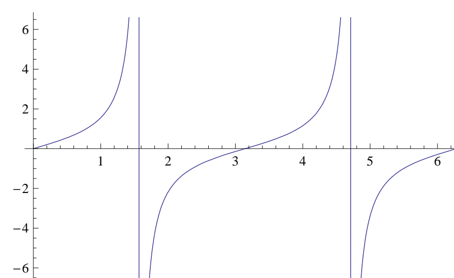
Wenn mehrere Funktionen gleichzeitig dargestellt werden sollen, müssen diese in einer Liste angegeben werden.

```
In[2]:= Plot[{Sin[3x], Sin[5x]},
             {x, 0, Pi}]
```



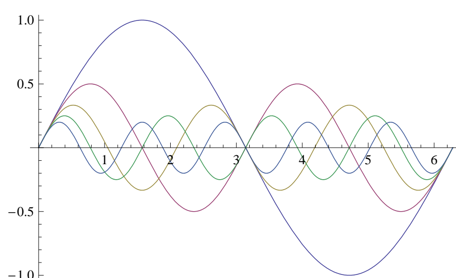
Wenn eine Funktion Singularitäten aufweist, schneidet *Mathematica* den Zeichenbereich ab und versucht nicht, alle Funktionswerte darzustellen.

```
In[3]:= Plot[Tan[x], {x, 0, 2Pi}]
```



Beim `Plot[]`-Kommando werden zunächst die Zahlenwerte für die x-Achse erzeugt, und dann erst wird jeder dieser Werte in den Ausdruck f eingesetzt, der dabei immer wieder neu ausgewertet wird. Das kann bei komplizierten Ausdrücken sehr zeitaufwendig sein oder sogar zu Fehlermeldungen und einer leeren Graphik führen, zum Beispiel wenn f ein Ausdruck ist, der eine Tabelle von Funktionen erzeugt. Deshalb sollte nach Möglichkeit das Argument f von `Plot[]` mit Hilfe von `Evaluate[]` ausgewertet werden. Dieses Kommando erzwingt die Auswertung des Ausdrucks f, *bevor* dieser an `Plot[]` weitergegeben wird:

```
In[4]:= Plot[ Evaluate[ Table[Sin[n x]/n, {n, 1, 5}] ], {x, 0, 2Pi} ];
```

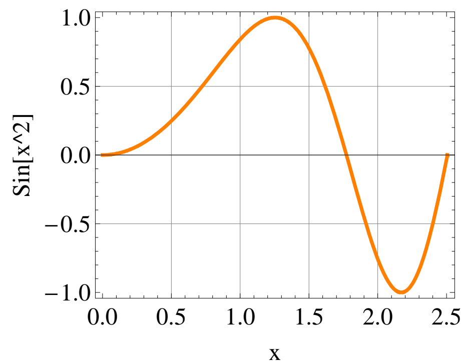


Ohne `Evaluate[]` würde jedem Datenpunkt x eine Liste mit 5 Zahlenwerten zugeordnet, und nicht eine Liste von 5 Funktionen, die *Mathematica* zeichnen könnte.

Das Aussehen einer mit dem `Plot[]`-Kommando erzeugten Graphik kann durch Angabe von *Optionen* in verschiedener Weise verändert werden. Optionen werden als Ersetzungsregeln in der Form *Optionsname* `->` *Wert* im `Plot[]`-Kommando angegeben. Für jede Option sind Vorgabewerte (Defaultwerte) eingestellt, sodaß der Entwurf einer Graphik üblicherweise sehr rasch gelingt. Hier werden exemplarisch nur die gängigsten Optionen besprochen. Eine Liste aller Optionen für das `Plot[]`-Kommando und deren Defaulteinstellungen erhält man mit `Options[Plot]`, eine genaue Beschreibung aller Optionen findet man in der elektronischen Dokumentation (im `Help`-Menü der graphischen Benutzeroberfläche unter `Documentation Center` nach Eingabe von `Plot`).

`AspectRatio`	`-> Automatic`	skaliert beide Achsen gleich
	`-> 1`	liefert eine quadratische Graphik
`Axes`	`-> False`	unterdrückt die Achsen
`AxesLabel`	`-> {"`*xlabel*`", "`*ylabel*`"}`	Achsenbeschriftungen
`DisplayFunction`	`-> Identity`	unterdrückt die Graphikausgabe
	`-> $DisplayFunction`	stellt sie wieder her
`Frame`	`-> True`	erzeugt einen Rahmen
`FrameLabel`	`-> {"`*u*`", "`*l*`", "`*o*`", "`*r*`"}`	Rahmenbeschriftungen
`GridLines`	`-> Automatic`	zeichnet ein Gitter
`PlotLabel`	`-> {"`*label*`"}`	ergibt einen Titel
`PlotPoints`	`->` n	Minimalanzahl von Stützpunkten
`PlotRange`	`-> All`	voller Wertebereich für x, y
`PlotStyle`	`-> {{`s_1`}, {`s_2`}, ... }`	Stiloptionen (z.B. Linienstärke), zyklisch bei mehreren Kurven
`LabelStyle`	`-> {`s_1`, `s_2`, ... }`	Stiloptionen für Beschriftungen

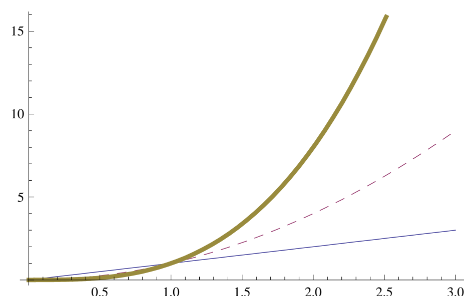
```
In[5]:= Plot[ Sin[x^2],
   {x, 0, Sqrt[2Pi]},
   AspectRatio -> Automatic,
   Frame -> True,
   FrameLabel -> {"x", "Sin[x^2]"},
   GridLines -> Automatic,
   PlotStyle -> {Orange,
                 Thickness[0.01]},
   LabelStyle -> {FontFamily -> "Times",
                  FontSize -> 16}
   ]
```



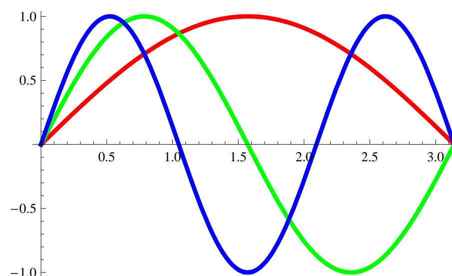
Stiloptionen: Die Darstellung von Linien und Kurven wird mit dem Kommando `Thickness[]` für die Linienbreite und `Dashing[]` für gestrichelte Linien gesteuert. Mit `Thickness[`*breite*`]` wird die Linienbreite eingestellt. `Dashing[{`l_1`, `l_2`, ... }]` gibt die Längen an, über die eine Linie durchgezogen bzw. unterbrochen werden soll. Die Angaben werden zyklisch wiederholt. Linienbreiten und Längen werden als Bruchteile der Breite der Graphik angegeben; die Defaulteinstellung für Strichbreiten ist 0.004 für 2D-Graphiken. Die beiden wichtigsten Kommandos zur Farbeinstellung lauten `RGBColor[`*rot*`, `*grün*`, `*blau*`]` und `GrayLevel[`*n*`]`, wobei der Wertebereich für die Parameter zwischen 0 und 1 liegt. Für `RGBColor[0,0,0]` bzw. `GrayLevel[0]` erhält man jeweils die Farbe schwarz. Daneben gibt es noch das Kommando `Hue[`*farbton*`, `*sättigung*`, `*helligkeit*`]` zur Farbeinstellung im HSV-Farbmodell (hue, saturation, value bzw. brightness). Für *farbton* zwischen 0 und 1 wird der Farbkreis von rot, gelb, grün, blau wieder nach rot durchlaufen. Bei der Kurzform `Hue[`*farbton*`]` sind Sättigung und Helligkeit jeweils mit 1 voreingestellt.

```
In[6]:= Plot[ {x, x^2, x^3}, {x, 0, 3},
              PlotStyle -> {{}, Dashing[{0.02}], Thickness[0.01]} ]
```

Die erste Kurve wird jetzt mit der Defaulteinstellung gezeichnet (`{}`), die zweite strichliert (`Dashing[]`), die dritte dicker als üblich (`Thickness[]`).




```
In[7]:= Plot[ {Sin[x], Sin[2x], Sin[3x]},
        {x, 0, Pi},
        PlotStyle -> {
        {Thickness[0.01], RGBColor[1,0,0]},
        {Thickness[0.01], RGBColor[0,1,0]},
        {Thickness[0.01], RGBColor[0,0,1]}}
        ]
```

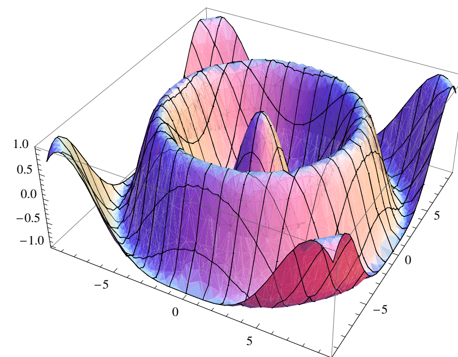


```
Plot3D[ f, {x, xmin, xmax}, {y, ymin, ymax} ]

Plot3D[ {f, farbe}, {x, xmin, xmax}, {y, ymin, ymax} ]
```

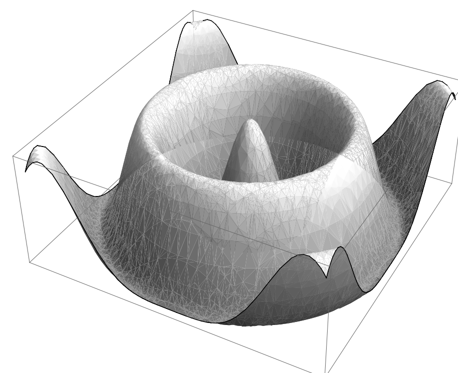
Das Kommando `Plot3D[]` stellt eine Funktion f von zwei Variablen x und y als Fläche im Raum dar. Dabei wird der rechteckige Definitionsbereich $[xmin, xmax] \times [ymin, ymax]$ als Grundfläche eines Quaders und der Wertebereich in vertikaler Richtung abgebildet. Die Farbgebung kann durch eine zweite Funktion *farbe*`[`x`, `y`]` gesteuert werden. Die Option `PlotPoints -> `n stellt die Zahl der Rasterpunkte pro Richtung ein (default: 15) und steuert damit die Auflösung. Weitere wichtige Optionen: `Boxed -> False` deaktiviert das Zeichnen eines umschreibenden Quaders (default: `True`). `Mesh -> False` unterdrückt das Zeichnen von Gitterlinien auf der Fläche (default: `True`). `BoxRatios -> {`n_x`, `n_y`, `n_z`}` bestimmt die Proportionen einer 3D-Graphik; mit der Einstellung `{1, 1, 1}` wird die Graphik in einem Würfel angezeigt, mit der Defaulteinstellung von `{1, 1, 0.4}` in einem flachen, quadratischen Quader. Mit der Einstellung `ViewPoint -> {`x`, `y`, `z`}` bestimmt man den Ort, von dem aus die Graphik betrachtet wird in einem eigenen, objektunabhängigen Koordinatensystem. Die Defaulteinstellungen sind `{1.3, -2.4, 2}`. Viele Optionen von `Plot3D[]` sind analog zu denen von `Plot[]`, wie etwa `AspectRatio`, `Axes`, `AxesLabel`, `PlotLabel`, `PlotPoints` und `PlotRange`.

```
In[8]:= Plot3D[
        Cos[Sqrt[x^2 + y^2]],
        {x, -3Pi, 3Pi},
        {y, -3Pi, 3Pi}]
```

Es wird vom Kernel ein Graphikobjet vom Typ `SurfaceGraphics` an das Frontend zur graphischen Darstellung übergeben.



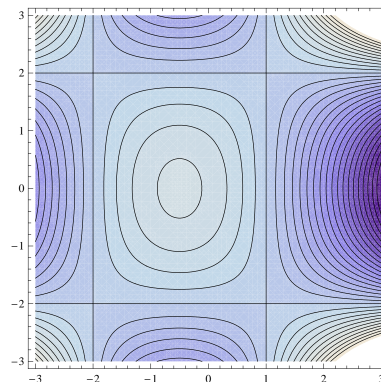

```
In[9]:=  z = Cos[Sqrt[x^2 + y^2]];
In[10]:= Plot3D[ z,
         {x, -3Pi, 3Pi},
         {y, -3Pi, 3Pi},
         Axes -> None,
         Mesh -> False,
         PlotPoints -> 30,
         ColorFunction ->
         Function[{x, y, z},
         GrayLevel[Abs[z]*0.5 + 0.4]]
         ]
```



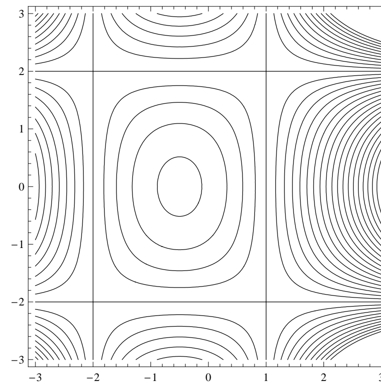
Manchmal ist eine andere Darstellung von Funktionen zweier Variablen hilfreich. Mit `ContourPlot[]` betrachtet man den rechteckigen Definitionsbereich von oben und zeichnet die Höhenschichtlinien der Funktion. Die Option `Contours ->` n gibt die Anzahl der Höhenlinien an, `ContourShading -> False` verhindert die Grauschattierung, `ContourLines -> False` unterdrückt das Zeichnen der Höhenlinien. `DensityPlot[]` bildet die Funktionswerte auf Grau- oder Farbstufen in einem quadratischen Gitter ab.

```
In[11]:= z = (x-1)(x+2)(y-2)(y+2);

In[12]:= ContourPlot[
         z,
         {x, -3, 3},
         {y, -3, 3},
         Contours -> 30,
         PlotPoints -> 40]
```

Es wird vom Kernel ein Graphikobjekt vom Typ `ContourGraphics` (2D-Rastergraphik mit Höhenschichtlinien) an das Frontend übergeben.

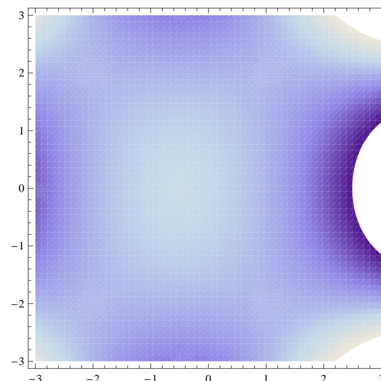


```
In[13]:= ContourPlot[
         z,
         {x, -3, 3},
         {y, -3, 3},
         Contours -> 30,
         ContourShading -> False,
         PlotPoints -> 40
         ]
```



```
In[14]:= DensityPlot[
         z,
         {x, -3, 3},
         {y, -3, 3},
         Mesh -> False,
         PlotPoints -> 60]
```

Es wird vom Kernel ein Graphikobjekt vom Typ `DensityGraphics` (2D-Rastergraphik) erzeugt.



9.2 Parametrische Kurven und Flächen

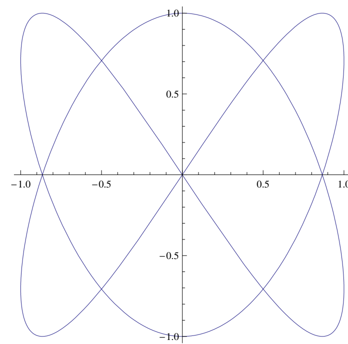
`ParametricPlot[` {f_x, f_y}, {t, *tmin*, *tmax*} `]`

`ParametricPlot3D[` {f_x, f_y, f_z}, {t, *tmin*, *tmax*} `]`

`ParametricPlot3D[` {f_x, f_y, f_z}, {u, *umin*, *umax*}, {v, *vmin*, *vmax*} `]`

Eine parametrische Kurve in der Ebene:

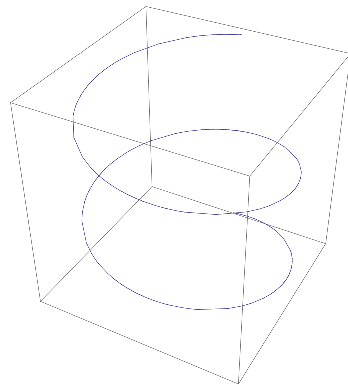
```
In[1]:= ParametricPlot[
        {Sin[2t], Sin[3t]},
        {t, 0, 2Pi},
        AspectRatio -> Automatic
        ]
```



Es können die gleichen Optionen wie bei `Plot[]` verwendet werden.

Eine parametrische Kurve im Raum:

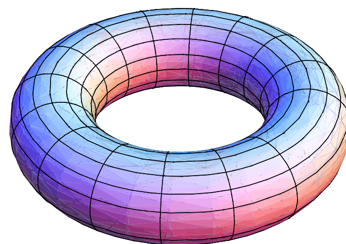
```
In[2]:= ParametricPlot3D[
        {Sin[t], Cos[t], t/6},
        {t, -2Pi, 2Pi},
        Axes -> False ]
```



Es wird vom Kernel ein Graphikobjekt vom Typ einer allgemeinen 3D-Graphik, die aus einzelnen Polygonen zusammengesetzt ist, zurückgegeben (`Graphics3D`).

Eine parametrische Fläche im Raum:

```
In[3]:= ParametricPlot3D[
        {Cos[u] (3 + Cos[v]),
         Sin[u] (3 + Cos[v]),
         Sin[v]},
        {u, 0, 2Pi},
        {v, 0, 2Pi},
        Axes -> False,
        Boxed -> False ]
```



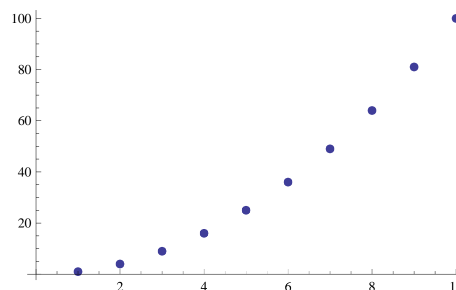
9.3 Graphische Darstellung diskreter Daten

Diskrete Daten können entweder von *Mathematica* selbst stammen (etwa aus einem `Table[]`-Kommando), oder von externen Dateien mit `ReadList[]` eingelesen worden sein. In beiden Fällen liegen die Daten als (verschachtelte) Listen vor. Zu fast allen Graphik-Kommandos existieren auch Varianten, um solche Datenlisten darzustellen:

`ListPlot[ {`y_1`, `y_2`, ... } ]`	zeichnet $y_1, y_2, \ldots$ mit x-Werten von $1, 2, \ldots$
`ListPlot[ {{`x_1`, `y_1`}, {`x_2`, `y_2`}, ... } ]`	zeichnet die Punkte (x_1, y_1), (x_2, y_2), ...
`ListPlot3D[ {{`z_{11}`, `z_{12}`, ... }, {`z_{21}`, `z_{22}`, ... }, ... } ]`	zeichnet eine 3D-Fläche mit z-Werten aus der angegebenen 2D-Liste
`ListPointPlot3D[ {{`x_1`, `y_1`, `z_1`}, {`x_2`, `y_2`, `z_2`}, ... } ]`	zeichnet eine 3D-Punktgraphik

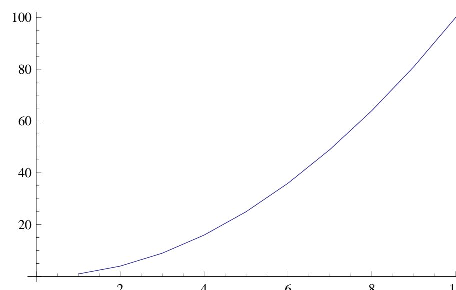
```
In[1]:= data = Table[i^2, {i, 10}];
In[2]:= ListPlot[
        data,
        PlotStyle -> PointSize[0.02]
        ]
```

Die Option `PlotStyle -> PointSize[`n`]` stellt die Punktgröße n relativ zur Gesamtgraphik ein.



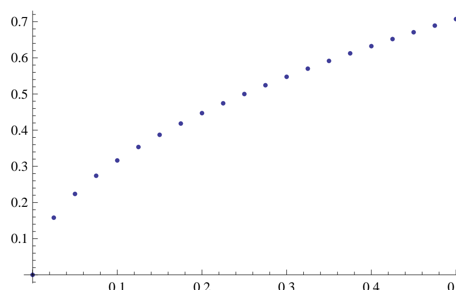

```
In[3]:= ListPlot[
        data,
        Joined -> True
        ]
```

Die Option `Joined -> True` verbindet die einzelnen Punkte durch einen Linienzug.



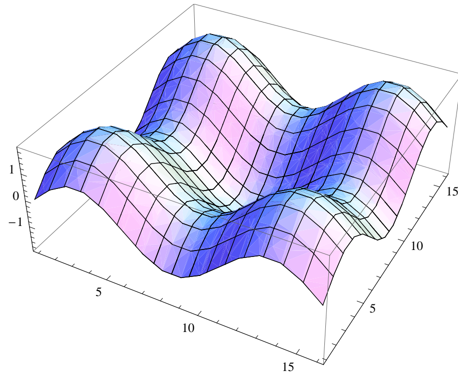

```
In[4]:= data = Table[ {i, Sqrt[i]},
                {i, 0, 0.5, 0.025} ];
In[5]:= ListPlot[data]
```

Das erste Element jeder Teilliste wird hier als x-Koordinate verwendet.

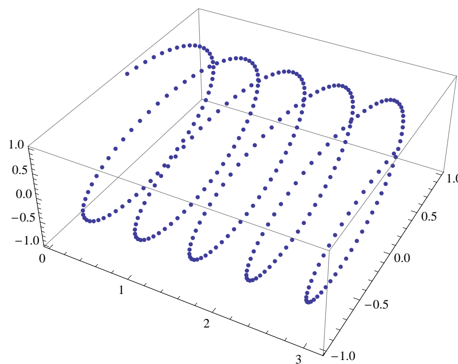


```
In[6]:= data = Table[ Sin[x] + Sin[y],
               {x, 0, 3Pi, Pi/5},
               {y, 0, 3Pi, Pi/5} ];
In[7]:= ListPlot3D[data]
```

Die Graphik wird unabhängig von der Zahl der Koordinatenpunkte in x- und y-Richtung auf einen quadratischen Bereich abgebildet.




```
In[8]:= data = Table[
               {t/10, Sin[t], Cos[t]},
               {t, 0, 10Pi, Pi/30} ];
In[9]:= ListPointPlot3D[data]
```



Das Kommando `ReadList[]` eignet sich besonders zum Lesen von Zahlenwerten aus ASCII-Dateien, die nicht von *Mathematica* stammen:

`ReadList["`*file*`", Number]`	liest Zahlen aus *file* in eindimensionale Liste
`ReadList["`*file*`", Number, RecordLists -> True]`	Daten werden zeilenweise zu Teillisten gruppiert

Das Kommando `!!`*file* zeigt die ASCII-Datei mit dem Namen *file* am Schirm an.

```
In[11]:= !!file1.dat
1 1
2 4
3 9
```

Liest die in `file1.dat` enthaltenen Zahlen, die durch Kommas, Leerzeichen oder Zeilenumbrüche getrennt sein dürfen, und gibt sie als Liste zurück.

```
In[11]:= ReadList["file1.dat", Number]
Out[11]= {1, 1, 2, 4, 3, 9}
```

Mit der Option `RecordLists -> True` werden die Daten zeilenweise zu Teillisten gruppiert.

```
In[12]:= ReadList["file1.dat", Number,
                  RecordLists -> True ]
Out[12]= {{1, 1}, {2, 4}, {3, 9}}
```

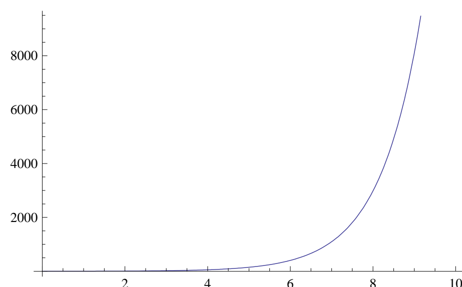
9.4 Spezialkommandos

9.4.1 Logarithmische Achsenskalierung

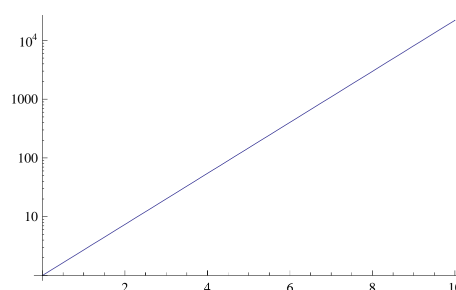
Mathematica bietet auch Kommandos zur logarithmischen Skalierung jeweils einer oder beider Koordinatenachsen. Alle diese Kommandos stehen auch in einer `ListPlot[]`-Version zur Verfügung.

`LogPlot[` f`, {`x`, `$xmin$`, `$xmax$`} ]`	y-Achse logarithmisch
`LogLinearPlot[` f`, {`x`, `$xmin$`, `$xmax$`} ]`	x-Achse logarithmisch
`LogLogPlot[` f`, {`x`, `$xmin$`, `$xmax$`} ]`	beide Achsen logarithmisch

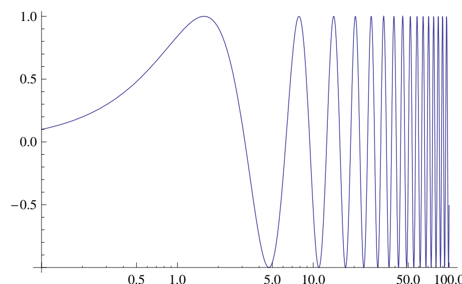
```
In[1]:= Plot[E^x, {x, 0, 10}]
```



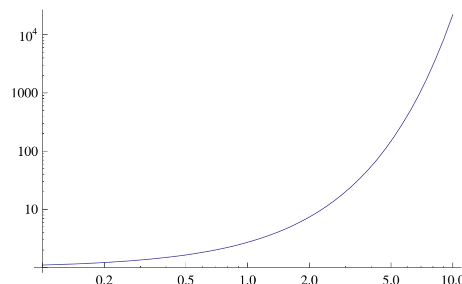

```
In[2]:= LogPlot[E^x, {x, 0, 10}]
```




```
In[3]:= LogLinearPlot[Sin[x],
        {x, 0.1, 100}]
```




```
In[4]:= LogLogPlot[E^x,
        {x, 0.1, 10}]
```

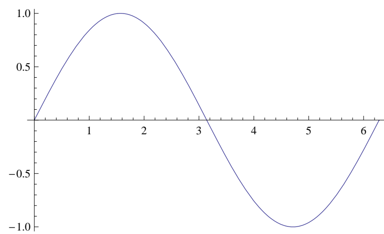


9.4.2 Graphiken kombinieren

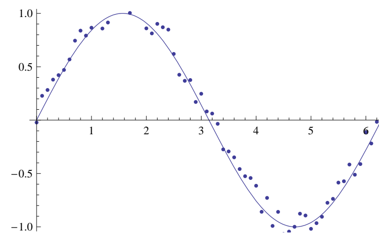
Graphiken werden *Mathematica*-intern nicht als Bilder (Rastergraphiken), sondern als eine Folge von Kommandos mit zugehörigen Daten und Optionen dargestellt. Deshalb können Graphiken, wie alle anderen *Mathematica*-Ausdrücke, in Variablen gespeichert und mit dem `Show[]`-Kommando später mit zusätzlichen bzw. veränderten Optionen nochmals angezeigt werden. Außerdem können mit dem `Show[]`-Kommando mehrere Graphiken *gleichen Typs* gemeinsam angezeigt werden; *Mathematica* paßt dabei die Wertebereiche für die Koordinatenachsen so an, daß alle vorhandenen Informationen dargestellt werden.

`Show[`g`]`	Graphik g anzeigen
`Show[`g`,` *option* `->` *value*`]`	g mit geänderten Optionen anzeigen
`Show[`g_1`,` g_2`, ... ]`	$g_1, g_2, \ldots$ gemeinsam anzeigen

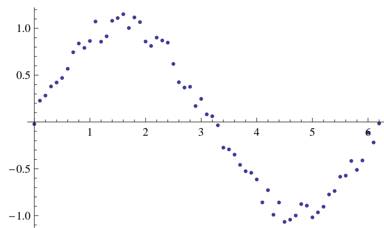
```
In[1]:= g1 = Plot[Sin[x], {x, 0, 2Pi}]
```



```
In[3]:= g3 = Show[g1, g2]
```



```
In[2]:= g2 = ListPlot[
             Table[{i, Sin[i] + Random[Real, {-0.1, 0.2}]},
                   {i, 0, 2Pi, 0.1}]
             ]
```



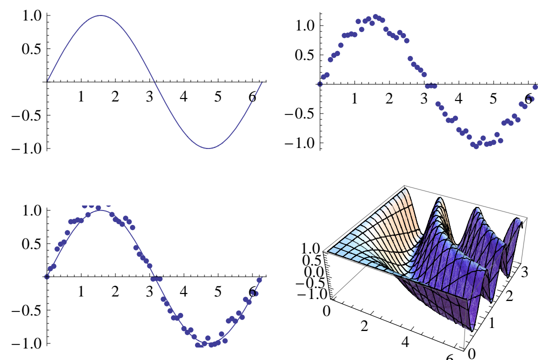
Mit dem Kommando `Show[GraphicsArray[{{`g_1`,` g_2`}, {`g_3`,` g_4`}}]]` können einzelne Graphiken auch *verschiedenen Typs* zeilenweise in einem Raster angeordnet werden. Die Einzelgraphiken werden zeilenweise in einer zweidimensionalen Liste übergeben.

```
In[4]:= g4 = Plot3D[ Cos[x y], {x, 0, 2Pi}, {y, 0, Pi},
                    DisplayFunction -> Identity ]
```

Durch die Option `DisplayFunction -> Identity` wird die Graphik zwar berechnet, aber nicht angezeigt.

```
In[5]:= Show[
        GraphicsArray[{{g1, g2},
                       {g3, g4}}]
        ]
```

Mathematica erzeugt ein Graphikobjekt vom Typ “Liste von Graphiken verschiedenen Typs”, ein sogenanntes `GraphicsArray`.



9.4.3 Vektorfelder

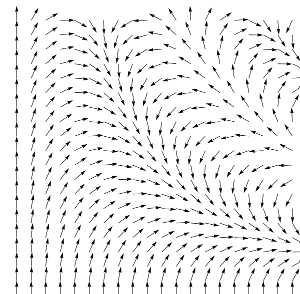
Das *Mathematica*-Paket `VectorFieldPlots‘` beinhaltet Kommandos zur Darstellung von zweidimensionalen bzw. dreidimensionalen vektoriellen Funktionen (Vektorfeldern) $(f_x(x,y), f_y(x,y))$ bzw. $(f_x(x,y,z), f_y(x,y,z), f_z(x,y,z))$.

> `VectorFieldPlot[` {f_x, f_y}, {x, $xmin$, $xmax$}, {y, $ymin$, $ymax$} `]`
>
> `VectorFieldPlot3D[` {f_x, f_y, f_z}, {x, $xmin$, $xmax$}, {y, $ymin$, $ymax$}, {z, $zmin$, $zmax$} `]`

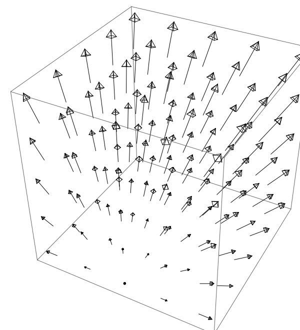
Die Komponenten der Vektorfunktion werden im ersten Parameter als Liste angegeben. Die Option `PlotPoints -> ` n bestimmt die Anzahl der Pfeile pro Koordinatenrichtung (default: 15 Pfeile in 2D, 7 Pfeile in 3D). Bei `VectorFieldPlot3D[]` werden mit `VectorHeads -> True` die Pfeile mit Spitzen versehen (default: `False`). Daneben gibt es in dem Paket noch Kommandos zum Zeichnen von Gradientenfeldern *skalarer* Funktionen $f(x,y)$ bzw. $f(x,y,z)$.

> `GradientFieldPlot[` f, {x, $xmin$, $xmax$}, {y, $ymin$, $ymax$} `]`
>
> `GradientFieldPlot3D[` f, {x, $xmin$, $xmax$}, {y, $ymin$, $ymax$}, {z, $zmin$, $zmax$} `]`

```
In[1]:= Needs["VectorFieldPlots‘"];

In[2]:= VectorFieldPlot[
        {Sin[x y], Cos[x y]},
        {x, 0, Pi},
        {y, 0, Pi},
        PlotPoints -> 20
        ]
```



```
In[3]:= Needs["VectorFieldPlots‘"];

In[4]:= VectorFieldPlot3D[ {x, y, z},
        {x, -1, 1},
        {y,  0, 2},
        {z,  0, 2},
        PlotPoints -> 5,
        VectorHeads -> True
        ]
```

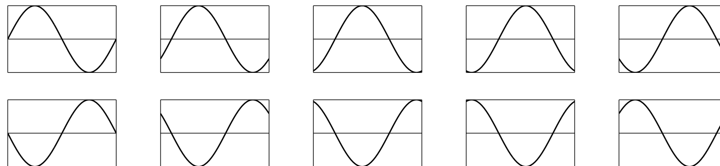


9.4.4 Animation von Graphiken

Das Prinzip von Animationen mit *Mathematica* besteht darin, daß zuerst alle Bilder einer bewegten Graphik vollständig berechnet und danach in rascher Abfolge angezeigt werden, sodaß der Eindruck einer Bewegung ensteht. Das Kommando `Animate[]` implementiert verschiedene Möglichkeiten, um solche Listen von Einzelgraphiken als bewegte Graphik darzustellen.

`Animate[` *expr*`,` `{`u_{min}`,` u_{max}`,` *du*`}` `]`	animiert den Ausdruck *expr*

```
In[1]:= Animate[ Plot[ Sin[x - n], {x, 0, 2Pi},
                       Frame -> True, FrameTicks -> None, Ticks -> False,
                       PlotPoints -> 50, PlotRange -> {{0, 2Pi}, {-1, 1}},
                       DisplayFunction -> Identity ],
                 {n, 0, 2Pi - Pi/5, Pi/5} ]
```



Anmerkung: Mit `Animate[]` können auch symbolische Ausrücke “animiert” werden, wie z.B. `Animate[Series[Exp[x], {x, 0, n}], {n, 1, 6, 1}, AnimationRunning -> False]`. Einen ähnlichen Effekt wie mit `Animate[]` erhält man mit dem Kommando `Manipulate[]`.

9.4.5 Export von Graphiken

Zum Export einer *Mathematica*-Graphik in eine Datei speichert man die Graphik zuerst in einer Variablen und übergibt diese Variable dann als zweiten Parameter an das `Export[]`-Kommando:

`Export[ "`*file*`",` *graphics*`, "`*format*`" ]`

Graphikformate (Auswahl, siehe auch `Export[]`):

`"EPS"`	Encapsulated PostScript
`"PDF"`	Adobe Acrobat portable document format
`"GIF"`	komprimiertes Bitmap-Format (max. 256 Farben)
`"TIFF"`	Bitmap-Format, keine Komprimierung

Die Option `ImageSize -> {`n`,`m`}` liefert bei Bitmap-Formaten Bilder mit $n \times m$ Pixel.

Die Graphik wird in der Variablen g gespeichert und danach in eine Datei mit dem Namen *g.eps* im Format Encapsulated PostScript und in eine Datei *g.pdf* im Format PDF exportiert.

```
In[1]:= g = Plot[Sin[x], {x, 0, 2Pi}]
In[2]:= Export["g.eps", g, "EPS"];
In[3]:= Export["g.pdf", g, "PDF"];
```

10 Differentiation

D[f, x]	partielle Ableitung von f nach x, $\frac{\partial}{\partial x} f$
D[f, {x, n}]	n-te partielle Ableitung von f nach x, $\frac{\partial^n}{\partial x^n} f$
D[f, x_1, x_2, ...]	höhere partielle Ableitungen $\frac{\partial}{\partial x_1} \frac{\partial}{\partial x_2} \cdots f$
D[f, {x_1, n_1}, {x_2, n_2}, ...]	$\frac{\partial^{n_1}}{\partial x_1^{n_1}} \frac{\partial^{n_2}}{\partial x_2^{n_2}} \cdots f$

Der erste Parameter von D[] ist der abzuleitende Ausdruck, der zweite die Variable, nach der differenziert werden soll. Bei Ableitungen höherer Ordnung muß der zweite Parameter von D[] als Liste {x, n} angegeben werden.

```
In[1]:= D[x^3 + x^2 + x, x]
                          2
Out[1]= 1 + 2 x + 3 x
In[2]:= D[x^3 + x^2 + x, {x, 2}]
Out[2]= 2 + 6 x
```

Mathematica kennt die Ableitungsfunktionen von allen elementaren Funktionen.

```
In[3]:= D[ ArcTan[x], x ]
           1
Out[3]= ------
             2
        1 + x
```

Zwei Syntaxvarianten zur Bildung der ersten Ableitung von f nach x. Dabei ist f'[x] eine Kurzschreibweise für D[f[x], x] bei Funktionen mit einer Variablen. Intern wird f'[x] durch Derivative[1][f][x] dargestellt. Analog wird auch f''[x] intern als Derivative[2][f][x] dargestellt.

```
In[4]:= f[x_] := Sin[a x]
In[5]:= D[ f[x], x ]
Out[5]= a Cos[a x]
In[6]:= f'[x]
Out[6]= a Cos[a x]
In[7]:= Derivative[1][f][x]
Out[7]= a Cos[a x]
```

Der Wert der Ableitungsfunktion von f an der Stelle 0.

```
In[8]:= {f'[0], Derivative[1][f][0]}
Out[8]= {a, a}
```

f'[x]	erste Ableitung von f nach x, wenn $f = f(x)$
f''[x]	zweite Ableitung von f nach x, wenn $f = f(x)$
Derivative[n_1, n_2, ...][f][x_1, x_2, ...]	$\frac{\partial^{n_1}}{\partial x_1^{n_1}} \frac{\partial^{n_2}}{\partial x_2^{n_2}} \cdots f$

Bei der Berechnung der partiellen Ableitung mit D[] werden alle Symbole außer den Ableitungsvariablen als Konstante behandelt.

```
In[9]:= D[ x^n, x ]
           -1 + n
Out[9]= n x
In[10]:= D[ x^2 + y^2, x ]
Out[10]= 2 x
```

Hängt y von x ab, so muß y explizit als Funktion von x in der Form y[x] formuliert werden.

```
In[11]:= D[ x^2 + y[x]^2, x ]
Out[11]= 2 x + 2 y[x] y'[x]
```

Wenn *Mathematica* keine Informationen über die abzuleitenden Funktionen hat, wird das Ergebnis in allgemeiner Form dargestellt.

```
In[12]:= Clear[f]
In[13]:= D[ f[x]g[x], x ]
Out[13]= g[x] f'[x] + f[x] g'[x]
In[14]:= D[ f[g[x]], x ]
Out[14]= f'[g[x]] g'[x]
```

11 Integration

11.1 Symbolische Integration

Mathematica kann sowohl unbestimmte als auch bestimmte Integrale mit vorgegebenen Integrationsgrenzen, die zusammen mit der Integrationsvariablen als Liste angegeben werden, berechnen:

Integrate[*f*, *x*] Integrate[*f*, {*x*, *xmin*, *xmax*}]

Ein einfaches Integral.

```
In[1]:= Integrate[x^2, x]
         3
        x
Out[1]= --
        3
```

Das unbestimmte Integral $\int \frac{dx}{x^2-1}$.

```
In[2]:= Integrate[1/(x^2 - 1), x]
        Log[-1 + x]   Log[1 + x]
Out[2]= ----------- - ----------
             2            2
```

Kontrolle: Mit D[*f*, *x*] wird die partielle Ableitung von f nach x gebildet.

```
In[3]:= D[%, x]
            1             1
Out[3]= ---------- - ---------
        2 (-1 + x)   2 (1 + x)

In[4]:= Simplify[%]

           1
Out[4]= -------
             2
        -1 + x
```

Das bestimmte Integral $\int_a^b (x^2 + x^3)\, dx$.

```
In[5]:= Integrate[x^2 + x^3,
                  {x, a, b}]
          3    4    3    4
        -a    a    b    b
Out[5]= --- - -- + -- + --
         3    4    3    4
```

Mathematica kommt auch mit schwierigeren Integralen zurecht:

```
In[6]:= Integrate[ (x^5 - 2*x^4 + 2*x^3 - x + 1) / (x^4 + x^2), x ]
                     2
          1         x                              2
Out[6]= -(-) - 2 x + -- + ArcTan[x] - Log[x] + Log[1 + x ]
          x         2
```

Integrale, die nicht durch elementare Funktionen ausgedrückt werden können, werden unausgewertet zurückgegeben:

```
In[7]:= Integrate[ Sin[x] / Log[x], x ]

                  Sin[x]
Out[7]= Integrate[------, x]
                  Log[x]
```

Anmerkung: In jedem Fall empfiehlt sich eine *Kontrolle* der Integration, z.B. durch Differenzieren des Ergebnisses und Vergleich mit dem Integranden.

11.2 Numerische Integration

Das Kommando `NIntegrate[]` zur numerischen Approximation von bestimmten Integralen hat dieselbe Syntax wie `Integrate[]`:

`NIntegrate[` f`, {`x`,` $xmin$`,` $xmax$`} ]`

Ein Näherungswert für das bestimmte Integral $\int_2^3 (x^2 + x^3)\, dx$.

```
In[1]:= NIntegrate[ x^2 + x^3,
                    {x, 2, 3} ]
Out[1]= 22.5833
```

Auch das Integral nicht elementar integrierbarer Funktionen wird numerisch approximiert.

```
In[2]:= NIntegrate[ Sin[x]/Log[x],
                    {x, 2, 3} ]
Out[2]= 0.674381
```

Das Integral $\int_1^2 (1/x^3 - 3/8) dx$ ist exakt 0. *Mathematica* liefert zwar ein richtiges Ergebnis, gibt aber eine Warnung aus, daß es sich möglicherweise um eine stark oszillierende Funktion handelt:

```
In[3]:= NIntegrate[ 1/x^3 - 3/8, {x, 1, 2} ]

NIntegrate::ploss:
    Numerical integration stopping due to loss of precision. Achieved neither
    the requested PrecisionGoal nor AccuracyGoal; suspect highly oscillatory
    integrand, or the true value of the integral is 0. If your integrand is
    oscillatory try using the option Method->Oscillatory in NIntegrate.
                  -18
Out[3]= 1.73472 10
```

Eine Erhöhung der gewünschten Zielgenauigkeit (`AccuracyGoal`) und der internen Rechengenauigkeit (`WorkingPrecision`) beseitigt diese Unklarheit:

```
In[4]:= NIntegrate[ 1/x^3 - 3/8, {x, 1, 2},
                    AccuracyGoal -> 30, WorkingPrecision -> 40 ]
             -31
Out[4]= 0. 10
```

11.3 Gauß-Legendre Quadratur

Das folgende *Mathematica*-Programm liefert die Stützstellen und Gewichte für die Gauß-Legendre Quadraturformeln.

```
(*  -------------------------------------------------------------  *)
(*  File: glq.m                                                    *)
(*  Stuetzstellen "x", Gewichte "w" fuer Gauss-Legendre Quadratur  *)
(*  Input : Anzahl "n" der Stuetzstellen                           *)
(*  Output: "gxn.dat" und "gwn.dat"                                *)
(*  -------------------------------------------------------------  *)

Clear["Global‘*"];
Needs["NumericalDifferentialEquationAnalysis‘"];

n  = Input["n : "];
xw = GaussianQuadratureWeights[ n, -1, 1, 20 ];

x = Table[ xw[[j,1]], {j, 1, n} ];
w = Table[ xw[[j,2]], {j, 1, n} ];

xfile = StringInsert[ "gx.dat", ToString[n], 3 ];
wfile = StringInsert[ "gw.dat", ToString[n], 3 ];

xout  = OpenWrite[ xfile, FormatType -> OutputForm ];
wout  = OpenWrite[ wfile, FormatType -> OutputForm ];

j = 1;
While[ j <= n,
       Write[ xout, PaddedForm[ x[[j]], {18, 15} ] ];
       Write[ wout, PaddedForm[ w[[j]], {18, 15} ] ];
       j = j + 1;
];

Close[xout];
Close[wout];
```

12 Potenzreihen

`Series[` f`, {`x`, `a`, `n`} ]`	Potenzreihe v. $f(x)$ der Ordnung n um den Entwicklungspunkt a
`SeriesCoefficient[` *reihe*`, `n` ]`	Reihenkoeffizient der Ordnung n
`Normal[` *reihe* `]`	*reihe* wird in einen gewöhnlichen math. Ausdruck umgewandelt

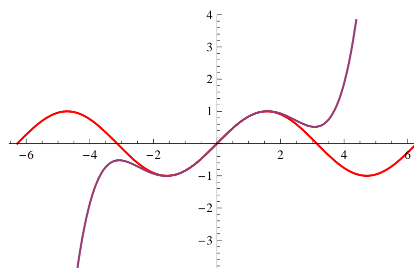
`Series[]` liefert die Taylorreihe $\sum_{k=0}^{n} \frac{f^{(k)}(a)}{k!}(x-a)^k$ bzw. die Laurentreihe $\sum_{k=-\infty}^{n} c_k(x-a)^k$ von $f(x)$ um den Entwicklungspunkt a bis zur gewünschten Ordnung n zusammen mit der Ordnungsfunktion `O[x-a]^n` :

```
In[1]:= Series[ Exp[x], {x, 0, 8} ]

                 2    3    4    5     6      7       8
                x    x    x    x     x      x       x          9
Out[1]= 1 + x + -- + -- + -- + --- + --- + ---- + ----- + O[x]
                2    6    24   120   720   5040   40320


In[2]:= Series[ Sin[z]/z^7, {z, 0, 2} ]
                                                2
         -6    1       1         1         z          3
Out[2]= z    - ---- + ------ + -(----) + ------ + O[z]
                  4        2     5040    362880
               6 z    120 z
```

`Series[]` gibt ein `SeriesData`-Objekt zurück, das zur numerischen Auswertung in einen gewöhnlichen mathematischen Ausdruck umgewandelt werden muß:

```
In[3]:= f = Series[ Sin[x], {x, 0, 5} ]

             3    5
            x    x         6
Out[3]= x - -- + --- + O[x]
            6    120

In[4]:= fn[x_] = Normal[f]

             3    5
            x    x
Out[4]= x - -- + ---
            6    120
```

```
In[5]:= Plot[ {Sin[x], fn[x]},
        {x, -2*Pi, 2*Pi},
        PlotStyle -> {{RGBColor[1,0,0],
                       Thickness[0.005]},
                       Thickness[0.005]}
        ]
```



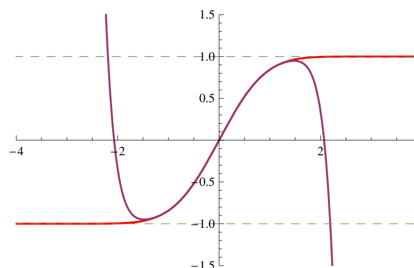
Als Anwendungsbeispiel von Potenzreihen betrachten wir die Approximation eines Integrals mit nicht elementar integrierbarem Integranden. Eine Näherungsformel für die *Fehlerfunktion* (engl.: *error function*),

$$\mathrm{erf}(x) \;:=\; \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \;\; ,$$

$\mathrm{erf}(0) = 0$, $\mathrm{erf}(\infty) = 1$, $\mathrm{erf}(-x) = -\mathrm{erf}(x)$, ergibt sich aus der gliedweisen Integration der Reihe für den Integranden $\exp(-t^2)$:

```
In[6]:= reihe1 = Series[ E^(-t^2), {t, 0, 10} ]

                  4     6     8     10
             2   t     t     t     t           11
Out[6]= 1 - t  + -- - -- + -- - --- + O[t]
                 2     6     24    120

In[7]:= reihe2 = Integrate[ reihe1, {t, 0, x} ]

             3     5     7     9      11
            x     x     x     x      x           12
Out[7]= x - -- + -- - -- + --- - ---- + O[x]
            3     10    42    216    1320
```

```
In[8]:= Plot[ Evaluate[ {Erf[x],
        (2/Sqrt[Pi])*Normal[reihe2], -1, 1} ],
        {x, -5, 5},
        PlotRange -> {{-4, 4}, {-1.5, 1.5}},
        PlotStyle -> {{RGBColor[1,0,0],
                       Thickness[0.005]},
                       Thickness[0.005],
                       Dashing[{0.02}],
                       Dashing[{0.02}]}
        ]
```



13 Fourierreihen

Zur Fourierentwicklung der Heaviside-Funktion

$$H(x) = \begin{cases} 1 & : \quad x > 0 \\ 0 & : \quad x < 0 \end{cases}$$

im Intervall $[-\frac{1}{2}, +\frac{1}{2}]$ verwenden wir die *Mathematica*-Funktion `UnitStep[]`, welche die Heaviside-Funktion implementiert.

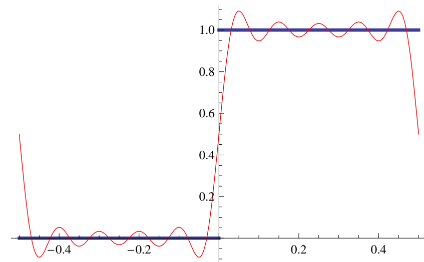
Außerdem betrachten wir nicht $H(x)$ selbst, sondern $f(x) := H(x) - \frac{1}{2}$. Diese Funktion ist ungerade, $f(-x) = -f(x)$, ihre Fourierreihe FR ist daher eine reine Sinusreihe,

$$\begin{aligned} FR(f(x)) &= \sum_{n=1}^{\infty} b_n \sin(k_n x) \\ b_n &= \frac{2}{L} \int_{-L/2}^{L/2} f(x) \sin(k_n x) \, dx \end{aligned}$$

mit $L = 1$ und $k_n = 2\pi n/L = 2\pi n$. Die Fourierkoeffizienten b_n können hier symbolisch berechnet werden:

```
In[1]:= kn = 2 Pi n;
In[2]:= b[n_] = 2 Integrate[ (UnitStep[x] - 1/2) Sin[kn x],
                             {x, -1/2, 1/2} ]  // Simplify
        1 - Cos[n Pi]
Out[2]= -------------
            n Pi

In[3]:= Table[ b[n], {n, 1, 10} ]
         2       2        2        2        2
Out[3]= {--, 0, ----, 0, ----, 0, ----, 0, ----, 0}
         Pi     3 Pi     5 Pi     7 Pi     9 Pi

In[4]:= fr[x_] = 1/2 + Sum[ b[n] Sin[kn x], {n, 1, 10} ]

        1   2 Sin[2 Pi x]   2 Sin[6 Pi x]   2 Sin[10 Pi x]
Out[4]= - + ------------- + ------------- + -------------- +
        2        Pi              3 Pi             5 Pi


        2 Sin[14 Pi x]   2 Sin[18 Pi x]
        -------------- + --------------
             7 Pi             9 Pi
```

```
In[5]:= Plot[
        {UnitStep[x], fr[x]},
        {x, -1/2, 1/2},
        PlotStyle -> {{Thickness[0.008]},
                      {RGBColor[1,0,0]}}
        ]
```



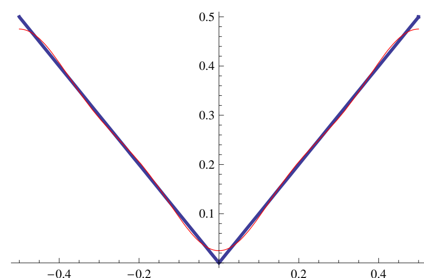
Das Paket `FourierSeries‘` enthält Kommandos, mit denen man die Fourierentwicklung periodischer Funktionen sowohl analytisch als auch numerisch ermitteln kann:

```
FourierTrigSeries[ f, x, n ]
NFourierTrigSeries[ f, x, n ]
```

`FourierTrigSeries[]` liefert die symbolische Fourierentwicklung der Ordnung n einer periodischen Funktion $f(x)$ mit dem Grundintervall $[-1/2, +1/2]$ und der Periodenlänge 1. Mit der Option `FourierParameters -> {0, b}` wird die Periodenlänge auf $1/|b|$ und das Grundintervall auf $[-1/(2\,|b|), +1/(2\,|b|)$ gesetzt.

```
In[1]:= Needs["FourierSeries‘"]

In[2]:= 1/2 + FourierTrigSeries[ UnitStep[x] - 1/2, x, 8 ]

        1   2 Sin[2 Pi x]   2 Sin[6 Pi x]   2 Sin[10 Pi x]   2 Sin[14 Pi x]
Out[2]= - + ------------- + ------------- + -------------- + --------------
        2        Pi             3 Pi             5 Pi             7 Pi

In[3]:= fr = FourierTrigSeries[ Abs[x], x, 3 ]

        1   2 Cos[2 Pi x]   2 Cos[6 Pi x]
Out[3]= - - ------------- - -------------
        4         2               2
                Pi            9 Pi
```

```
In[4]:= Plot[
        {Abs[x], fr},
        {x, -1/2, 1/2},
        PlotStyle -> {{Thickness[0.008]},
                      {RGBColor[1,0,0]}}
        ]
```



14 Differentialgleichungen

Mit den Kommandos NDSolve[] und DSolve[] kann man in *Mathematica* Differentialgleichungen bzw. Systeme von Differentialgleichungen numerisch und in einfachen Fällen auch symbolisch lösen.

14.1 Symbolische Lösung von Differentialgleichungen

Beim Kommando DSolve[] wird als erster Parameter die Differentialgleichung angegeben. Gleichungen und Differentialgleichungen werden in *Mathematica* mit doppeltem Gleichheitszeichen == geschrieben (d.h. mit dem Vergleichsoperator, der einen Test auf Gleichheit durchführt). Dabei muß die funktionale Abhängigkeit der gesuchten Lösungsfunktion von der unabhängigen Variablen immer explizit angegeben werden, also z.B. in der Form y[x] oder x[t]. Für die Ableitungen schreibt man meist x'[t] statt D[x[t],t] oder x''[t] statt D[x[t],{t,2}]. Im zweiten Parameter wird die Lösungsfunktion angegeben, die ermittelt werden soll. Im dritten Parameter muß die unabhängige Variable übergeben werden:

```
In[1]:= DSolve[ x'[t] == x[t], x[t], t ]
                  t
Out[1]= {{x[t] -> E  C[1]}}
```

Das Ergebnis wird als verschachtelte Liste von Ersetzungsregeln für die Lösungsfunktion x[t] zurückgegeben. Die Liste ist verschachtelt, da mit DSolve[] auch Systeme von Differentialgleichungen behandelt werden können (innere Liste). Außerdem kann eine Lösung mehrere Zweige besitzen (äußere Liste):

```
In[2]:= DSolve[ x[t] x'[t] == 1, x[t], t ]
Out[2]= {{x[t] -> -(Sqrt[2] Sqrt[t + C[1]])},
         {x[t] -> Sqrt[2] Sqrt[t + C[1]]}}
```

DSolve[] ermittelt die allgemeine Lösung der angegebenen Differentialgleichung mit durchnumerierten Integrationskonstanten C[*n*]. Gibt man zusätzlich Anfangsbedingungen an, dann werden die Integrationskonstanten so bestimmt, daß diese Anfangsbedingungen erfüllt sind. Die Anfangsbedingungen werden ebenfalls als Gleichungen formuliert und zusammen mit der Differentialgleichung als Liste im ersten Parameter an DSolve[] übergeben:

```
In[3]:= DSolve[ {x'[t] == x[t], x[0] == 1}, x[t], t ]
                  t
Out[3]= {{x[t] -> E }}
```

Um die von *Mathematica* gelieferten Lösungen weiterverwenden zu können (um sie z.B. graphisch darzustellen oder in einen anderen Ausdruck einzusetzen) sind noch weitere Schritte notwendig:

```
In[4]:= lsg = DSolve[ {x''[t] + x[t] == 0, x[0] == 1, x'[0] == 0}, x[t], t ]
Out[4]= {{x[t] -> Cos[t]}}
```

Das ist die Bewegungsgleichung eines harmonischen Oszillators mit Anfangsbedingungen für Ort `x[0]` und Geschwindigkeit `x'[0]`. Zuerst muß die Lösung `Cos[t]` aus der Liste der Ersetzungsregeln für `x[t]` "herausgezogen" werden:

```
In[5]:= lsg
Out[5]= {{x[t] -> Cos[t]}}
```

`lsg[[1]]` liefert das erste (und einzige) Element aus der verschachtelten Liste der Ersetzungsregeln für `x[t]`. Damit erhält man eine einfache Ersetzungsregel, die man sofort auf Ausdrücke anwenden kann.

```
In[6]:= lsg[[1]]
Out[6]= {x[t] -> Cos[t]}
```

Wendet man diese Ersetzungsregel auf `x[t]` selbst an, so erhält man unmittelbar die Lösung `Cos[t]`.

```
In[7]:= x[t] /. lsg[[1]]
Out[7]= Cos[t]
```

Dieser Ausdruck kann nun zur Definition einer neuen Funktion `x[t]` verwendet werden.

```
In[8]:= x[t_] = x[t] /. lsg[[1]]
Out[8]= Cos[t]
```

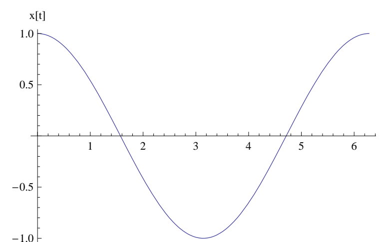
In dieser Form kann die Lösungsfunktion `x[t]` unmittelbar weiterverwendet werden. Zum Beispiel kann sie zur Probe in die gegebene Differentialgleichung eingesetzt werden,

```
In[9]:= {x''[t] + x[t] == 0, x[0] == 1, x'[0] == 0}
Out[9]= {True, True, True}
```

oder in verschiedenen Formen graphisch dargestellt werden:

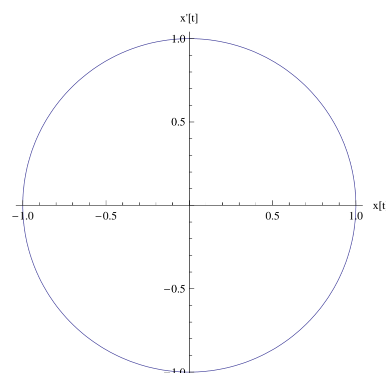
```
In[10]:= Plot[
           x[t],
           {t, 0, 2Pi},
           AxesLabel -> {"t", "x[t]"}
         ]
```

Darstellung der Lösung als x-t-Plot.




```
In[11]:= ParametricPlot[
           {x[t], x'[t]},
           {t, 0, 2Pi},
           AspectRatio -> Automatic,
           AxesLabel -> {"x[t]", "x'[t]"}
         ]
```

Darstellung der Lösung als parametrischer Plot in der Phasenebene $\{x(t), x'(t)\}$.



Anmerkung: Die obige Vorgangsweise zur Weiterverarbeitung des Resultats von `DSolve[]` hat den Nachteil, daß das Symbol `x` jetzt nicht mehr als Variable bzw. freie Funktion in `DSolve[]` verwendet werden kann. (Das ist übrigens einer der Gründe, warum die Lösungen von `DSolve[]` in *Mathematica* als temporär wirkende Ersetzungsregeln zurückgegeben werden.) Um dieses Problem zu umgehen, kann man alternativ die Lösung einer Differentialgleichung als Ersetzungsregel für `x` selbst verlangen:

```
In[1]:= lsg = DSolve[ {x''[t] + x[t] == 0, x[0] == 1, x'[0] == 0}, x, t ]
Out[1]= {{x -> Function[{t}, Cos[t]]}}
```

Das Ergebnis wird jetzt als reine Funktion zurückgegeben. Diese Variante hat den Vorteil, daß die Ersetzungsregel nicht nur für den Ausdruck `x[t]` gilt, sondern auch für `x'[t]` oder `x[0]` verwendet werden kann. Die Lösung kann damit sofort z.B. in die Ausgangsgleichung eingesetzt werden:

```
In[2]:= x'[t] /. lsg[[1]]
Out[2]= -Sin[t]
In[3]:= x[0] /. lsg[[1]]
Out[3]= 1
In[4]:= {x''[t] + x[t] == 0, x[0] == 1, x'[0] == 0} /. lsg[[1]]
Out[4]= {True, True, True}
In[5]:= Plot[ Evaluate[ x[t] /. lsg[[1]] ],  {t, 0, 2Pi} ]
```

Beim `Plot[]`-Kommando muß jetzt zusätzlich `Evaluate[]` angegeben werden, damit die Ersetzungsregel `x[t] /. lsg[[1]]` *vor* dem Zeichnen der Kurve angewendet wird.

`DSolve[` *dgl* `,` *x*`[`*t*`]` `,` *t* `]`	löst *dgl* symbolisch für *x*`[`*t*`]` mit *t* als unabhängiger Variable
`DSolve[` *dgl* `,` *x* `,` *t* `]`	Lösung für *x* als reine Funktion

Mathematica kann auch einfache Systeme von Differentialgleichungen symbolisch lösen. Im ersten Parameter von `DSolve[]` werden alle Gleichungen (inklusive Anfangsbedingungen) als Liste angegeben, im zweiten Parameter eine Liste der zu ermittelnden Lösungsfunktionen, im dritten Parameter wieder die unabhängige Variable:

```
In[1]:= DSolve[ {x'[t] == v[t], v'[t] == -x[t], x[0] == 1, v[0] == 0},
                {x[t], v[t]}, t ]
Out[1]= {{x[t] -> Cos[t], v[t] -> -Sin[t]}}
```

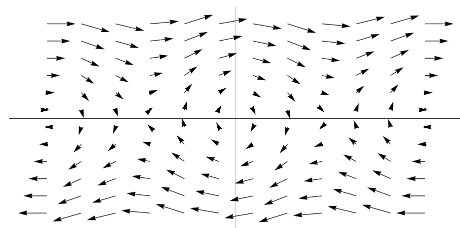
Das ist wieder die Bewegungsgleichung eines harmonischen Oszillators, diesmal geschrieben als System von gewöhnlichen Differentialgleichungen 1. Ordnung für die Auslenkung `x[t]` und die Geschwindigkeit `v[t]`.

`DSolve[ {`dgl_1`,` dgl_2`, ... }, {`x_1`[`t`],` x_2`[`t`], ... },` t `]`
`DSolve[ {`dgl_1`,` dgl_2`, ... }, {`x_1`,` x_2`, ... },` t `]`

14.2 Numerische Lösung von Differentialgleichungen

Mathematica kann zwar mit `DSolve[]` einige spezielle Typen von (gewöhnlichen) Differentialgleichungen symbolisch lösen, die meisten in der Praxis auftretenden Differentialgleichungen sind jedoch nur selten geschlossen lösbar. Man ist in diesen Fällen auf numerische Lösungen mit `NDSolve[]` angewiesen. Beispielsweise ist die Bewegungsgleichung eines mathematischen Pendels, $x''(t)+\sin x(t) = 0$, wo $x(t)$ der Winkel der Auslenkung aus der vertikalen Ruhelage ist, eine nichtlineare Differentialgleichung 2. Ordnung, vor der auch `DSolve[]` kapituliert. Immerhin kann man sich durch die Darstellung des Richtungsfeldes der Differentialgleichung eine gewisse Vorstellung über den Verlauf der Lösungen in der Phasenebene bilden:

```
In[1]:= Needs["VectorFieldPlots`"];
In[2]:= VectorFieldPlot[ {v, -Sin[x]},
         {x, -2Pi, 2Pi}, {v, -Pi, Pi},
         Axes -> True,
         PlotPoints -> 12,
         Ticks -> None,
         ScaleFactor -> 1.0 ]
```



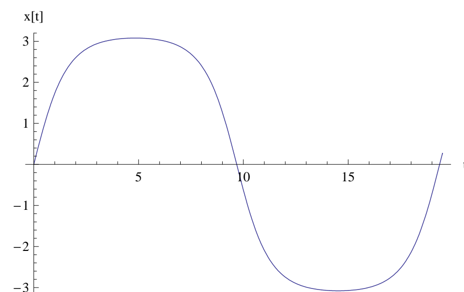
Bei der Bestimmung einer numerischen Näherungslösung mit `NDSolve[]` müssen neben der Differentialgleichung hinreichend viele Anfangsbedingungen formuliert werden, damit eine eindeutige Lösung möglich ist. Der Aufruf von `NDSolve[]` unterscheidet sich von `DSolve[]` nur darin, daß die unabhängige Variable zusammen mit dem gewünschten Wertebereich für die Lösung in einer Liste angegeben werden muß:

```
In[3]:= lsg = NDSolve[ {x''[t] + Sin[x[t]] == 0,
                        x[0] == 0, x'[0] == 2 - 0.001},
                       x, {t, 0, 19.5} ]
Out[3]= {{x -> InterpolatingFunction[{{0., 19.5}}, <>]}}
```

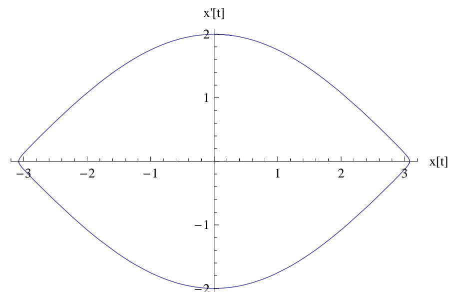
Die Lösungsfunktion wird als Ersetzungsregel für `x` in Form einer sogenannten `InterpolatingFunction[]` zurückgegeben. Dabei handelt es sich um eine Interpolationsfunktion, mit der man die Näherungslösung für jeden beliebigen Punkt im für `t` angegebenen Wertebereich berechnen kann und die man auch ableiten kann. Ansonsten wird `InterpolatingFunction[]` wie eine reine Funktion verwendet:

```
In[4]:= Plot[ Evaluate[ x[t] /. lsg[[1]] ],
              {t, 0, 19.5},
              AxesLabel -> {"t", "x[t]"}
        ]
```

Darstellung der Lösung als x-t-Plot.



```
In[5]:= ParametricPlot[
        Evaluate[ {x[t], x'[t]} /. lsg[[1]] ],
        {t, 0, 19.5},
        AspectRatio -> Automatic,
        AxesLabel -> {"x[t]", "x'[t]"}
        ]
```



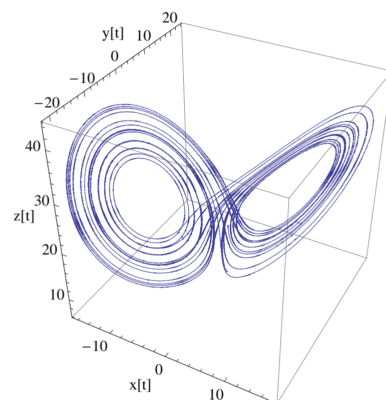
Darstellung der Lösung als parametrischer Plot in der Phasenebene $\{x(t), x'(t)\}$.

Mit `NDSolve[]` können natürlich auch Systeme von Differentialgleichungen behandelt werden. Analog zu `DSolve[]` werden alle Gleichungen zusammen mit den Anfangsbedingungen als Liste angegeben, ebenso die zu ermittelnden Lösungsfunktionen:

```
In[6]:= sigma = 10; r = 28; b = 8/3;
In[7]:= lsg = NDSolve[ {x'[t] == -sigma (x[t] - y[t]),
                        y'[t] == -x[t] z[t] + r x[t] - y[t],
                        z'[t] ==  x[t] y[t] - b z[t],
                        x[0] == 1, y[0] == 1, z[0] == 20},
                       {x, y, z}, {t, 0, 20},
                       MaxSteps -> 10000, MaxStepSize -> 0.0025 ]
Out[7]= {{x -> InterpolatingFunction[{{0., 20.}}, <>],
          y -> InterpolatingFunction[{{0., 20.}}, <>],
          z -> InterpolatingFunction[{{0., 20.}}, <>]}}
```

Das Verhalten von `NDSolve[]` kann durch zahlreiche Optionen gesteuert werden. `MaxSteps` gibt die Maximalanzahl der Integrationsschritte an (default: `1000`), `MaxStepSize` die zugehörige maximale Schrittweite. Ebenso wie bei `NIntegrate[]` können die Zielgenauigkeit (`PrecisionGoal`) und die interne Rechengenauigkeit (`WorkingPrecision`) eingestellt werden. `Method` wählt das numerische Verfahren: `Gear` oder `Adams` (default), `RungeKutta`.

```
In[8]:= ParametricPlot3D[
        Evaluate[
             {x[t], y[t], z[t]} /. lsg[[1]] ],
        {t, 0, 20},
        AxesLabel -> {"x[t]", "y[t]", "z[t]"},
        BoxRatios -> {1, 1, 1},
        PlotPoints -> 2000, PlotRange -> All,
        LabelStyle ->
        {FontFamily -> "Times-Roman",
         FontSize   -> 13}
        ]
```



Phasenporträt der Lorenz-Gleichungen.

```
NDSolve[ dgl, x, {t, tmin, tmax} ]
NDSolve[ {dgl1, dgl2, ... }, {x1, x2, ... }, {t, tmin, tmax} ]
```

14.3 Numerische Lösung partieller Differentialgleichungen

Mathematica kann mit `NDSolve[]` einige Typen von Anfangswertproblemen partieller Differentialgleichungen numerisch lösen. Für Randwertprobleme bei elliptischen Differentialgleichungen (z.B. Poisson- bzw. Laplacegleichung) hat *Mathematica* derzeit noch keine fertigen Kommandos. Die Syntax von `NDSolve[]` ist bei partiellen Differentialgleichungen im wesentlichen die gleiche wie bei gewöhnlichen Differentialgleichungen:

```
NDSolve[ dgl, u, {x, xmin, xmax}, {t, tmin, tmax} ]
```

Die 1D-Wellengleichung

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

beschreibt die Bewegung einer idealisierten Saite, wobei $u = u(x,t)$ die Auslenkung der Saite am Ort x zum Zeitpunkt t und c die Ausbreitungsgeschwindigkeit einer Störung (Welle) angibt. Mit den Anfangs- und Randbedingungen

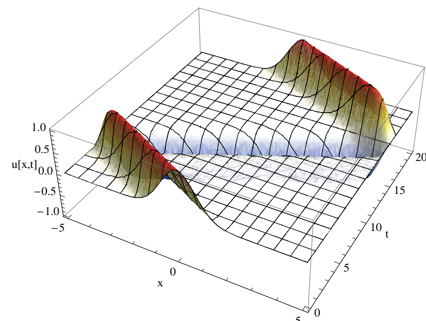
$$\begin{aligned} u(x,0) = f(x) &= \begin{cases} 0 & : \quad x = -5 \\ e^{-x^2} & : \quad -5 < x < 5 \\ 0 & : \quad x = 5 \end{cases} \\ u_t(x,0) = g(x) &= -2xe^{-x^2} \quad : \quad -5 < x < 5 \\ u(-5,t) = u(5,t) &= 0 \qquad\quad : \quad t > 0 \end{aligned}$$

wird ein nach "links" laufender Wellenzug auf einer an beiden Enden eingespannten Saite dargestellt. Für $c = 1$ wird das Problem mit `NDSolve[]` so formuliert:

```
In[1]:= f = If[ (x == -5 || x == 5), 0, Exp[-x^2] ];
In[2]:= g = -2 x Exp[-x^2];
In[3]:= lsg = NDSolve[ { D[u[x,t], t, t] == D[u[x,t], x, x],
                         u[x, 0] == f,
                         Derivative[0,1][u][x, 0] == g,
                         u[-5, t] == 0, u[5, t] == 0 },
                       u, {x, -5, 5}, {t, 0, 20} ]
Out[3]= {{u -> InterpolatingFunction[{{-5, 5.}, {0., 20.}}, <>]}}
```

Das Ergebnis ist eine zweidimensionale Interpolationsfunktion, mit der man die Lösung sofort zeichnen kann.

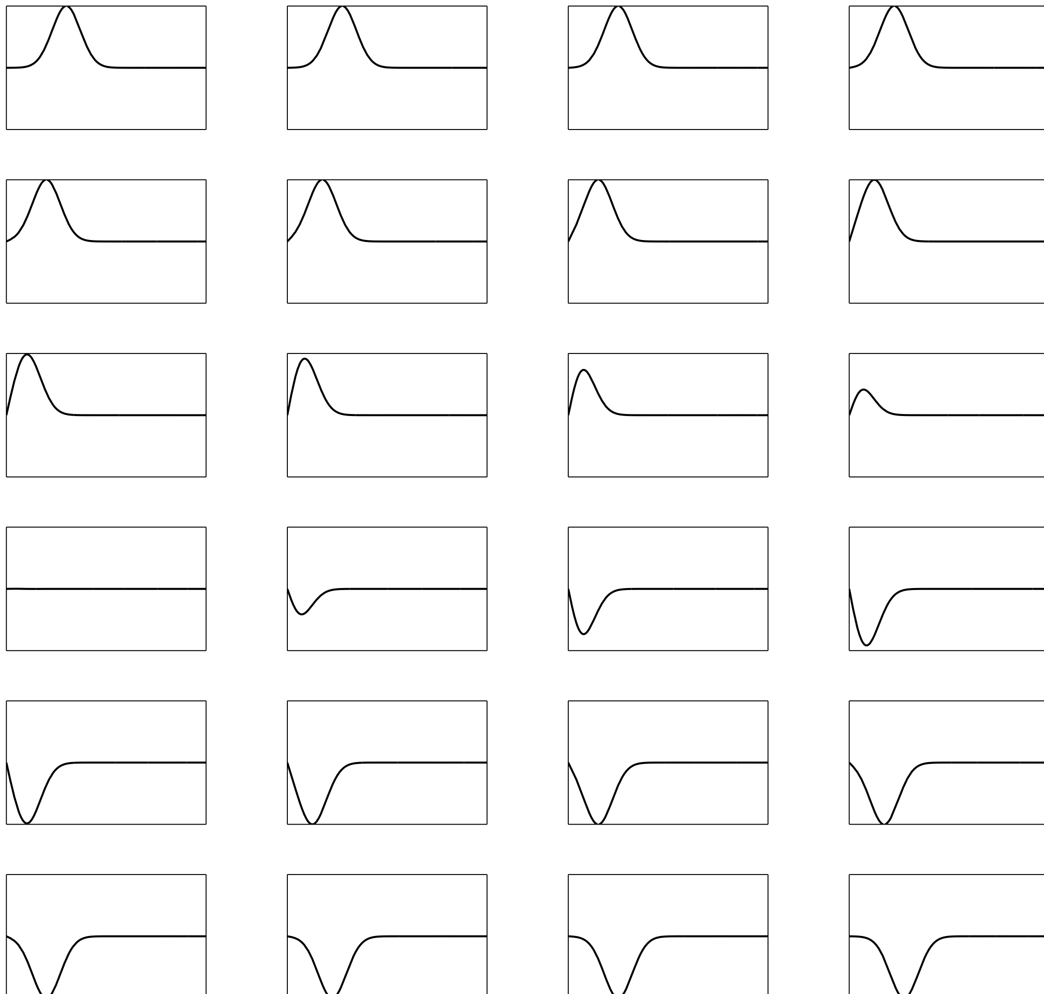
```
In[4]:= Plot3D[
        Evaluate[ u[x, t] /. lsg[[1]] ],
        {x, -5, 5}, {t, 0, 20},
        AxesLabel -> {"x", "t", "u[x,t]"},
        PlotPoints -> 30, PlotRange -> All,
        ColorFunction ->
        (ColorData["TemperatureMap"][#3] &)
        ]
```



Mit dem Kommando `Animate[]` kann die Bewegung der Saite am Bildschirm dargestellt werden. Dazu werden intern die Lösungen für aufeinanderfolgende Zeitpunkte als Einzelgraphiken berechnet, in einer Liste gespeichert, und dann abgespielt.

```
In[5]:= Animate[ Plot[ Evaluate[ u[x, t] /. lsg[[1]] ],
                       {x, -5, 5},
                       Axes -> False,
                       PlotPoints -> 25,
                       PlotRange -> {{-5, 5}, {-1, 1}},
                       PlotStyle -> RGBColor[1, 0, 0],
                       DisplayFunction -> Identity ],
                 {t, 0, 20 - 0.25, 0.25} ]
```

Die folgende Abbildung zeigt eine Sequenz aus dieser Animation:



15 Gleichungen

15.1 Symbolische Lösung von Gleichungen

Das *Mathematica*-Kommando zum Lösen von Gleichungen heißt `Solve[]`. Dem Kommando wird im ersten Parameter die zu lösende Gleichung, im zweiten Parameter die Variable, nach der aufgelöst werden soll, übergeben. Gleichungen werden wieder mit doppeltem Gleichheitszeichen == geschrieben. Bei Gleichungssystemen bzw. mehreren Unbekannten müssen Listen als Parameter verwendet werden:

`Solve[ {`gl_1`, `gl_2`, ... }, {`x_1`, `x_2`, ... } ]`

```
In[1]:= lsg = Solve[ x^2 + x == 5, x ]
                  -1 - Sqrt[21]         -1 + Sqrt[21]
Out[1]= {{x -> -------------}, {x -> -------------}}
                        2                     2
```

Als Lösung wird eine verschachtelte Liste von Ersetzungsregeln zurückgegeben (da auch mehrere Gleichungen mit mehreren Unbekannten behandelt werden können). Diese Regeln können mit dem Ersetzungsoperator `/.` vorübergehend in mathematische Ausdrücke eingesetzt werden (ohne die dabei auftretenden Variablen bleibend zu verändern):

```
In[2]:= Simplify[ x^2 + x /. lsg ]
Out[2]= {5, 5}
```

Durch `[[`n`]]` kann auf eine bestimmte Teillösung zugegriffen werden. Beispielsweise kann die erste Lösung für `x` bleibend in `x1` gespeichert werden:

```
In[3]:= x1 = x /. lsg[[1]]
        -1 - Sqrt[21]
Out[3]= -------------
              2
```

Um ein Gleichungssystem mit mehreren Unbekannten zu lösen, müssen die Gleichungen und die Unbekannten als Liste gruppiert werden:

```
In[4]:= lsg = Solve[ {2 x^2 + y == 1, x - y == 2}, {x, y} ]
                     7           3
Out[4]= {{y -> -(-), x -> -(-)}, {y -> -1, x -> 1}}
                     2           2
```

Man erhält eine Liste mit den zwei Lösungen, und jede Lösung ist selbst wieder eine Liste von Ersetzungsregeln für die beiden Unbekannten. Einsetzen der Lösung in die Ausgangsgleichungen ergibt:

```
In[5]:= Simplify[ {2 x^2 + y == 1, x - y == 2} /. lsg ]
Out[5]= {{True, True}, {True, True}}
```

`Solve[]` liefert nur die *generische* Lösung eines Gleichungssystems als Liste von Ersetzungsregeln, d.h. von gewissen Spezialfällen wird abgesehen. Die Funktion `Reduce[]` ist hier exakter und erzeugt als Lösung einen logischen Ausdruck mit Einzelgleichungen, in dem alle Spezialfälle berücksichtigt sind:

`Reduce[` {gl_1, gl_2, … }, {x_1, x_2, … }]

```
In[1]:= Solve[ a x^2 + b x + c == 0, x ]
                         2                             2
              -b - Sqrt[b  - 4 a c]         -b + Sqrt[b  - 4 a c]
Out[1]= {{x -> ---------------------}, {x -> ---------------------}}
                       2 a                           2 a
In[2]:= Reduce[ a x^2 + b x + c == 0, x ]
                       2
            -b - Sqrt[b  - 4 a c]
Out[2]= x == --------------------- && a != 0 ||
                     2 a
                       2
            -b + Sqrt[b  - 4 a c]
        x == --------------------- && a != 0 ||
                     2 a
        a == 0 && b == 0 && c == 0 ||
                          c
        a == 0 && x == -(-) && b != 0
                          b
```

Dabei werden das logische *Oder* mit `||`, das logische *Und* mit `&&` und das *Ungleich* mit `!=` geschrieben.

`Eliminate[]` ist eine Variante zu `Solve[]`, die Variablen aus einem Gleichungssystem eliminiert und die verbleibenden Gleichungen als Ergebnis liefert:

`Eliminate[` {gl_1, gl_2, … }, {x_1, x_2, … }]

```
In[1]:= Eliminate[ {x - y == a, x + y == b}, x ]
Out[1]= b - 2 y == a
```

Das Kommando `LinearSolve[]` löst *lineare* Gleichungssysteme $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, die durch Matrizen und Vektoren formuliert sind:

`LinearSolve[` A, b]

```
In[1]:= mat = { {1, 2, 3}, {4, -5, 6}, {9, 8, 7} };  b = {14, 12, 46};
In[2]:= LinearSolve[ mat, b ]
Out[2]= {1, 2, 3}
```

15.2 Numerische Lösung von Gleichungen

Die Lösungen von *algebraischen* (d.h. polynomialen) Gleichungen vom Grad > 4 lassen sich im allgemeinen nicht durch rationale Ausdrücke mit Radikalen schreiben und daher auch nicht explizit mit Hilfe von `Solve[]` darstellen. `Solve[]` liefert in diesen Fällen nur einen symbolischen Ausdruck. Mit `NSolve[]` wird dieser Ausruck gleich numerisch ausgewertet (wobei der optionale Parameter n die gewünschte Rechengenauigkeit, d.h. die Anzahl der Dezimalstellen, angibt):

> `NSolve[ {`gl_1`, `gl_2`, ... }, {`x_1`, `x_2`, ... }, `n` ]`

```
In[1]:= NSolve[ x^5 - x^2 + 1 == 0, x, 20 ]
Out[1]= {{x -> -0.80873060047939 20137},
         {x -> -0.4649122016028978543 - 1.0714738402702694092 I},
         {x -> -0.4649122016028978543 + 1.0714738402702694092 I},
         {x -> 0.8692775018425938612 - 0.3882694065997403554 I},
         {x -> 0.8692775018425938612 + 0.3882694065997403554 I}}
```

Bei *transzendenten* Gleichungen sind `Solve[]` (und damit auch `NSolve[]`) im allgemeinen überfordert. In solchen Fällen kann man mit `FindRoot[]` mit rein numerischen Verfahren eine Näherungslösung ermitteln:

> `FindRoot[ `*gleichung*`, {`x`, `x_0`} ]`
> `FindRoot[ `*gleichung*`, {`x`, `x_0`, `x_{min}`, `x_{max}`} ]`
> `FindRoot[ {`gl_1`, `gl_2`, ... }, {`x`, `x_0`}, {`y`, `y_0`}, ... ]`

`FindRoot[]` sucht eine numerische Lösung für die angegebene Gleichung unter Verwendung des Startwertes x_0. In der zweiten Variante wird die Lösung durch x_{min} und x_{max} eingegrenzt. Mit den Optionen `WorkingPrecision ->` n und `AccuracyGoal ->` n kann die gewünschte Rechen- und Zielgenauigkeit angegeben werden. Als numerisches Verfahren für die Nullstellensuche wird das *Newton-Verfahren* verwendet. `FindRoot[]` findet auch komplexe Lösungen sowie Lösungen für Gleichungssysteme.

```
In[1]:= FindRoot[ Cos[x] == x, {x, 3} ]
Out[1]= {x -> 0.739085}

In[2]:= FindRoot[ Log[x] == Cot[x], {x, 1} ]
Out[2]= {x -> 1.30828}
In[3]:= FindRoot[ Log[x] == Cot[x], {x, 4} ]
Out[3]= {x -> 3.78584}

In[5]:= FindRoot[ {x^2 + y^2 == 10, x^y == 2}, {x, 1}, {y, 1} ]
Out[5]= {x -> 1.27043, y -> 2.89586}
```

16 Anhang

16.1 Mathematische Konstanten und Funktionen

Symbol	Konstante
`Pi`	π
`E`	e
`I`	$\sqrt{-1}$
`Infinity`	∞
`Degree`	$\pi/180$
`GoldenRatio`	$(1+\sqrt{5})/2$

Symbol	Funktion
`Abs[x]`	$\|x\|$
`Sqrt[x]`	$\sqrt{x}$
`Exp[x]`	e^x
`Log[x]`	$\log x$
`Log[b, x]`	$\log_b x$
`x^y` oder `Power[x,y]`	x^y
`Sin[x]`	$\sin x$
`Cos[x]`	$\cos x$
`Tan[x]`	$\tan x$
`ArcSin[x] ...`	$\arcsin x \ldots$
`Sinh[x] ...`	$\sinh x \ldots$
`ArcSinh[x] ...`	$\operatorname{arcsinh} x \ldots$
`Round[x]`	nächste ganze Zahl an x
`Re[z]`	Realteil einer komplexen Zahl z
`Im[z]`	Imaginärteil von z
`Arg[z]`	Argument von z
`Conjugate[z]`	konjugiert komplexe Zahl von z
`Mod[n, m]`	$n \bmod m$
`Max[x, y, ... ]`	$\max(x, y, \ldots)$
`Min[x, y, ... ]`	$\min(x, y, \ldots)$
`n!` oder `Factorial[n]`	$n!$
`Random[]`	Pseudozufallszahl zwischen 0 und 1
`N[x]`	wertet `x` numerisch aus

16.2 Klammertypen

Klammer	Bedeutung	Beispiel
`()`	Gruppierung math. Ausdrücke	`(x + y)^2`
`[]`	Argumente von Funktionen	`Sin[x]`
`{}`	Listen, Vektoren, Matrizen	`{a, b, c}`
`[[]]`	Index beim Zugriff auf Listen	`liste[[2]]`
`(* *)`	Kommentar	`n! (* Fakultaet *)`

16.3 Zuweisungs- und Vergleichsoperatoren

Operator	Bedeutung	Beispiel
`=`	sofortige Zuweisung	`x = 3`
`:=`	verzögerte Zuweisung	`x := y`
`->`	vorübergehende Zuweisung	`Frame -> True`
`==`	Test auf Gleichheit	`Solve[x^2 == 3, x]`

16.4 Algebraische Umformungen

Funktion	Bedeutung
`Expand[f]`	f ausmultiplizieren (bei Brüchen nur Zähler)
`Factor[f]`	den Ausdruck f faktorisieren
`Simplify[f]`	f vereinfachen (gründlicher mit `FullSimplify[f]`)
`Collect[f, x]`	f nach Potenzen von x gruppieren (Herausheben)
`Apart[f, x]`	Partialbruchzerlegung von f für die Variable x
`Together[f]`	f auf kleinsten gemeinsamen Nenner bringen
`Cancel[f]`	einen Bruch f kürzen

16.5 Definition von Funktionen

Definition	Bedeutung
`f  = 1 + x^2`	Scheinfunktion, sofortige Auswertung
`f := 1 + x^2`	Scheinfunktion, Auswertung erst bei Anwendung
`f[x_]  = 1 + x^2`	echte Funktion, sofortige Auswertung
`f[x_] := 1 + x^2`	echte Funktion, Auswertung erst bei Anwendung