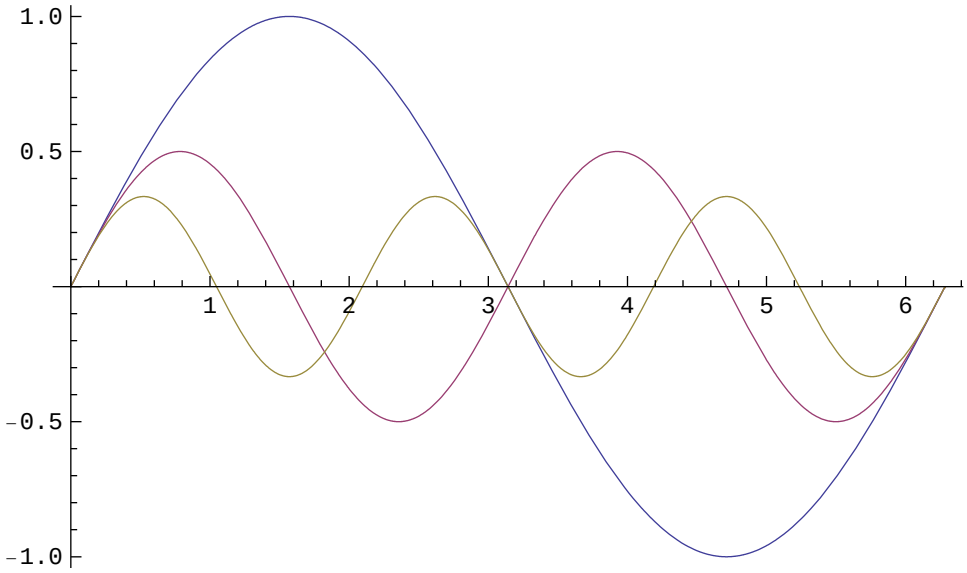
Fortgeschrittene Konzepte

Funktionen und Optionen

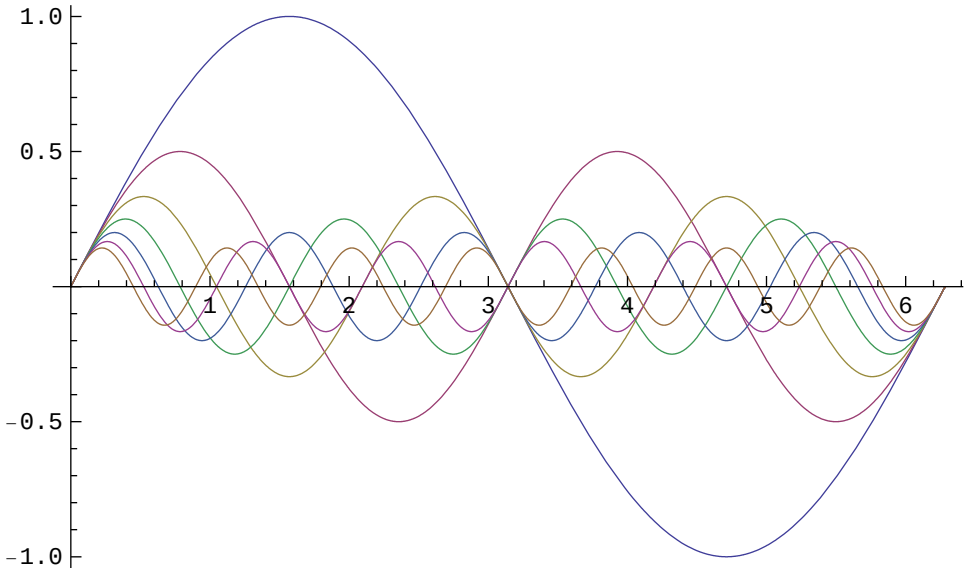
Optionen in eigenen Funktionsdefinitionen

```
ClearAll["Global`*"]
```

```
Plot[{Sin[x], Sin[2 x]/2, Sin[3 x]/3}, {x, 0, 2 π}]
```

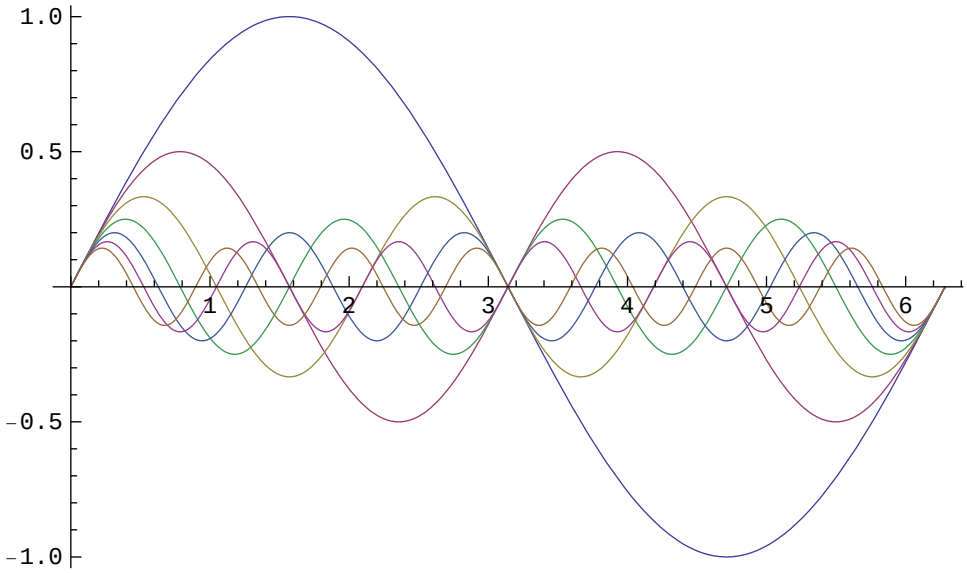


```
Plot[Evaluate[Table[Sin[n x]/n, {n, 1, 7}]], {x, 0, 2 π}]
```

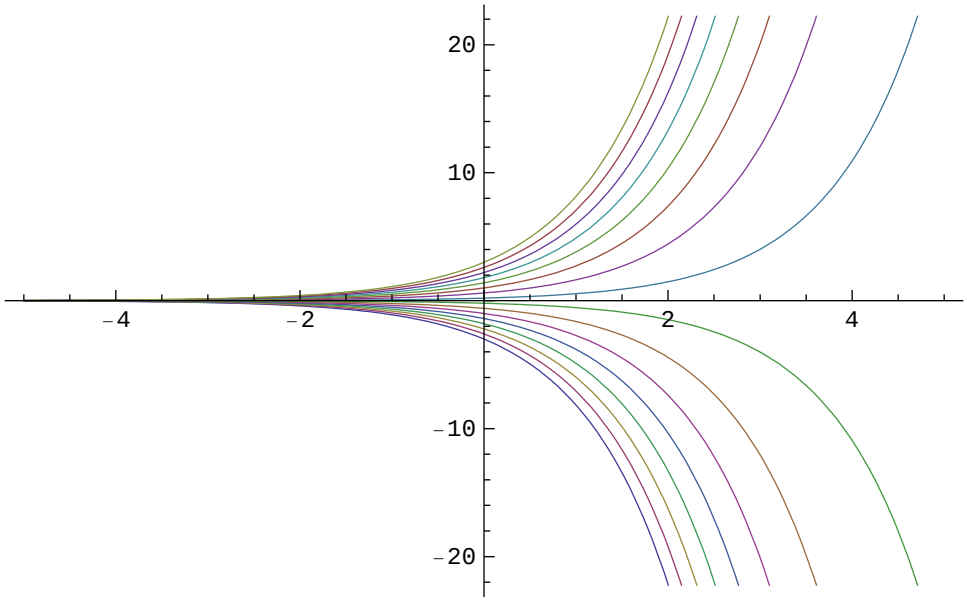



```
PlotMultiple[func_, list1_List, list2_List] :=
    Plot[Evaluate[Table[func, list1]], list2]
```

```
PlotMultiple[Sin[n x]/n, {n, 1, 7}, {x, 0, 2 π}]
```



```
PlotMultiple[c E^x, {c, -3, 3, .4}, {x, -5, 5}]
```



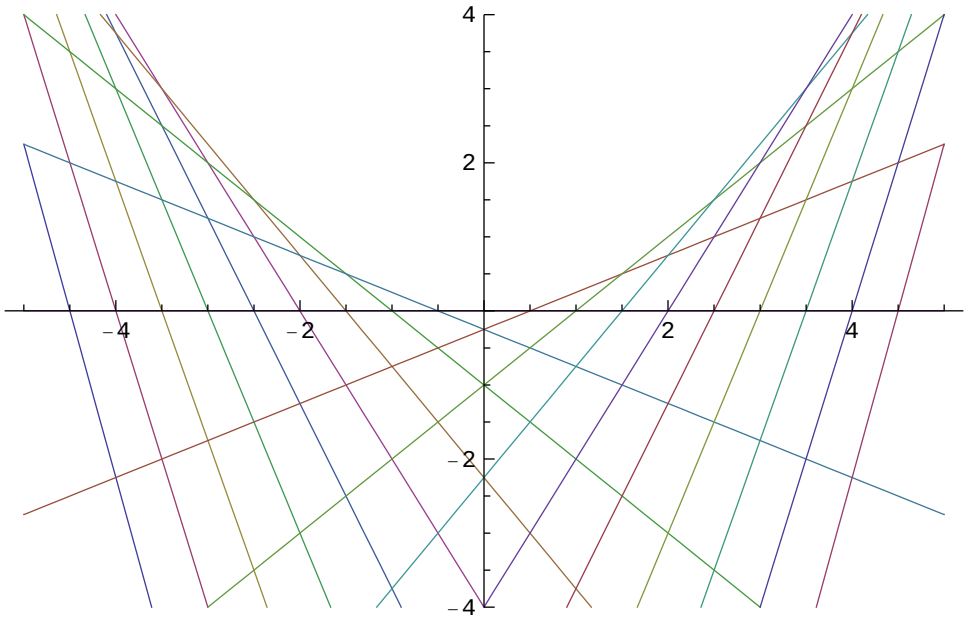

```
Clear[PlotMultiple]
PlotMultiple[func_, list1_List, list2_List, opts___?OptionQ] :=
    Plot[Evaluate[Table[func, list1]], list2, opts]
```

OptionQ ist in der Online−Hilfe noch immer nicht auffindbar

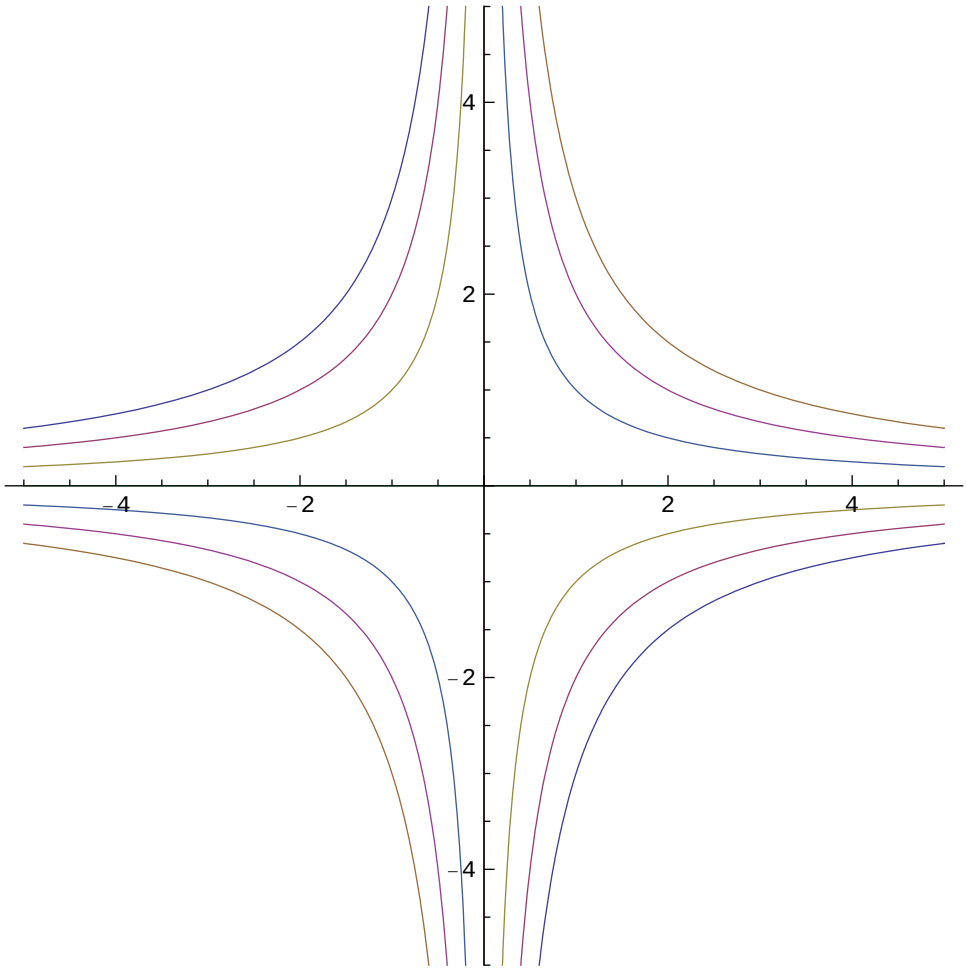
```
?OptionQ
```

```
OptionQ[e] returns True if e can be considered
  an option or list of options, and False otherwise.
```

```
PlotMultiple[c x - c^2, {c, -4.5, 4.5, 0.5}, {x, -5, 5}, PlotRange → {-4, 4}]
```



```
PlotMultiple[c/x, {c, -3, 3}, {x, -5, 5}, PlotRange → {-5, 5}, AspectRatio → 1]
```



■ Neue Optionen − Beispiel GenericMatrix

Hier sehen Sie an einem Beispiel die typische Weise Funktionen mit eigenen Optionen zu definieren.

```
clearAll["Global*'"];
genericMatrix[localOptions___?OptionQ] := myGenericMatrix[
  {x#1,#2 &, zeilen, spalten} /. {localOptions} /. {zeilen → 2, spalten → 2}]
myGenericMatrix[{f_, z_, s_}] := Table[f[i, j], {i, 1, z}, {j, 1, s}]

MatrixForm /@ {genericMatrix[], genericMatrix[spalten → 5]}
```

$$\left\{\begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix}, \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & x_{1,5} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & x_{2,5} \end{pmatrix}\right\}$$

Als Optionswerte können auch Funktionen eingesetzt werden.

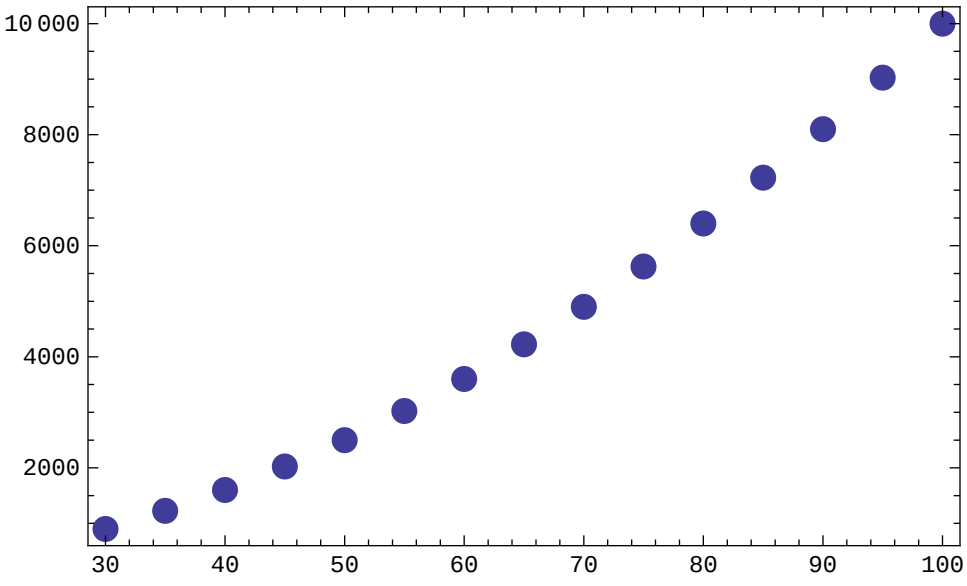
```
veryGenericMatrix[localOptions___?OptionQ] := myGenericMatrix[
  {f, zeilen, spalten} /. {localOptions} /. {f → (x#1,#2 &), zeilen → 2, spalten → 2}]

veryGenericMatrix[f → Function[{x, y}, Random[]], zeilen → 3] // MatrixForm
```

$$\begin{pmatrix} 0.646031 & 0.883221 \\ 0.651436 & 0.959997 \\ 0.118156 & 0.923297 \end{pmatrix}$$

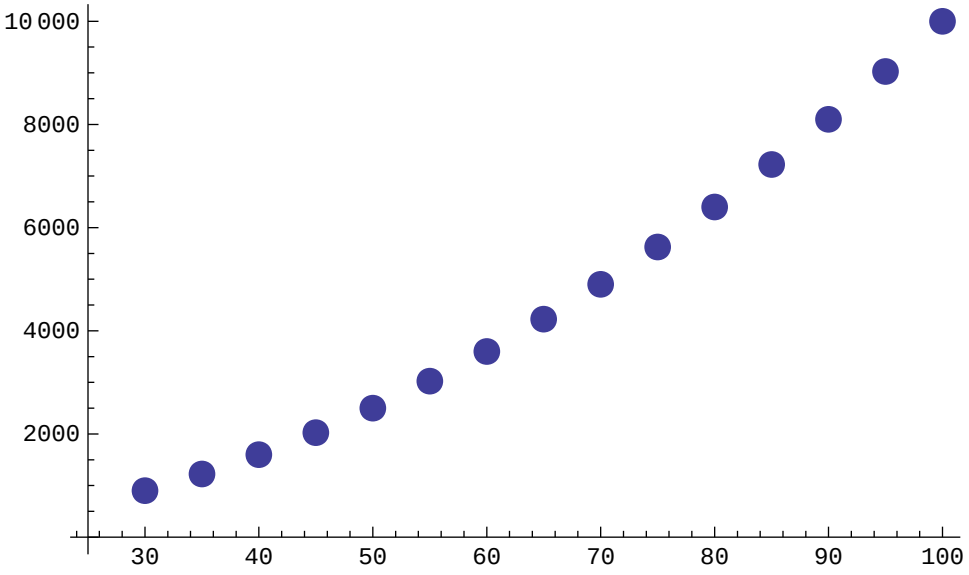
▪ Neue Optionen − Beispiel DotPlot

Und nun noch ein komplexeres Beispiel.

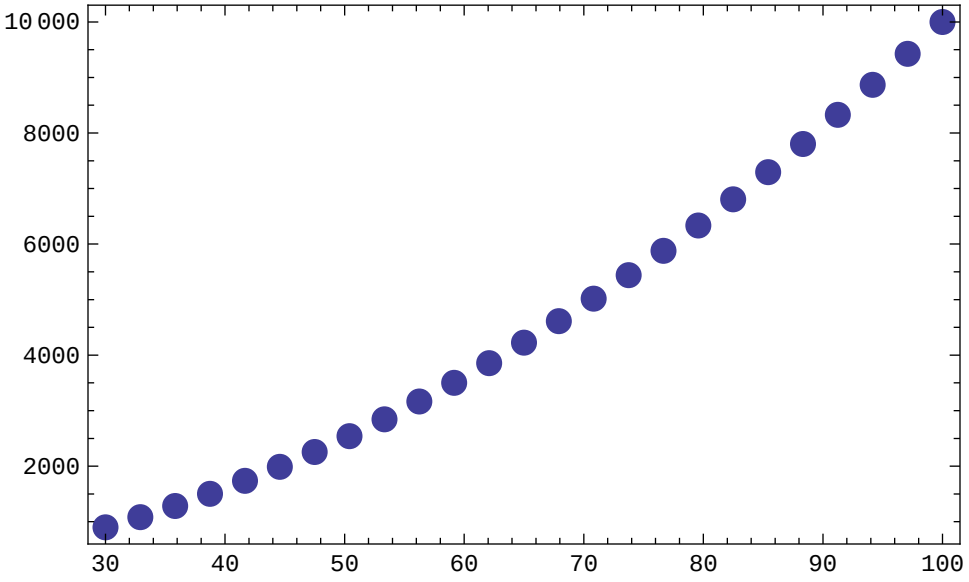
```
ClearAll[DotPlot];
Options[DotPlot] = Append[Options[ListPlot], PlotPoints → 15];
SetOptions[DotPlot, PlotRange → All, PlotStyle → PointSize[0.03],
     Frame → True, Axes → None];
DotPlot[f_, {var_, start_, end_}, opts___?OptionQ] := Module[{delta, plotpts},
    plotpts = PlotPoints /. {opts} /. Options[DotPlot];
    delta = (end - start) / (plotpts - 1);
    ListPlot[Table[{var, f}, {var, start, end, delta}],
    DeleteCases[Join[{opts}, Options[DotPlot]], PlotPoints → _]]]

DotPlot[x^2, {x, 30, 100}]
```



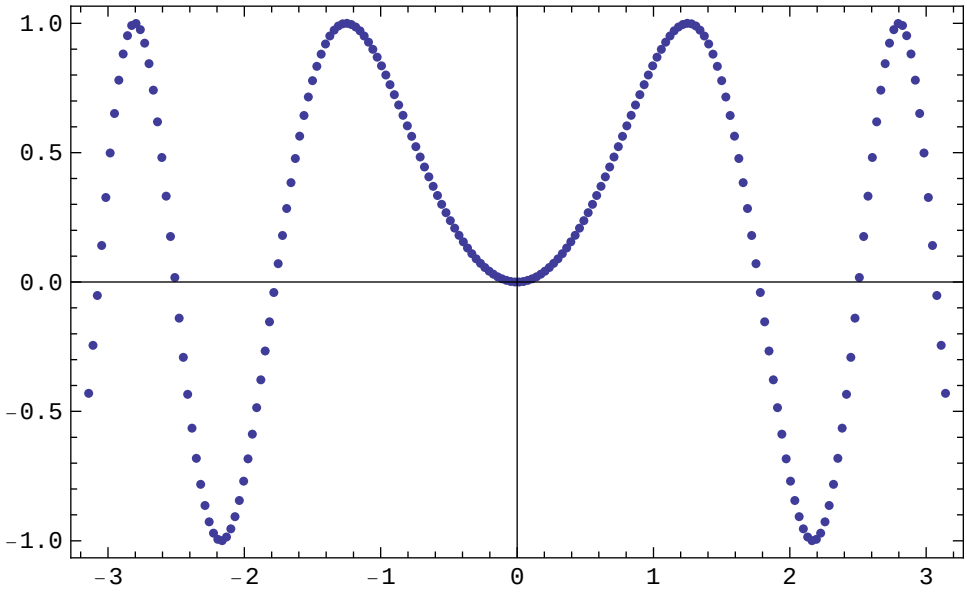
```
DotPlot[x^2, {x, 30, 100}, Frame -> False, Axes -> True, AxesOrigin -> {25, 0}]
```




```
DotPlot[x^2, {x, 30, 100}, PlotPoints → 25]
```



```
DotPlot[Sin[x^2], {x, -π, π}, PlotPoints -> 200, PlotStyle → PointSize[.01], Axes -> True]
```



■ Noch einmal Funktionen

▪ Funktionen mit optionalen Parametern

```
ClearAll["Global`*"]
```

```
potenzReihe[f_, x_: x, x0_: 0, order_: 6] := Series[f, {x, x0, order}]
```

```
potenzReihe[Sin[x], x]
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} + O[x]^7$$

```
potenzReihe[Sin[x], x, π, 3]
```

$$-(x-\pi) + \frac{1}{6}(x-\pi)^3 + O[x-\pi]^4$$

```
Clear [potenzReihe]
potenzReihe[f_, x_: x, x0_: 0, order_] := Series[f, {x, x0, order}]
```

```
potenzReihe[Sin[x], 12]
```

$$x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362\,880} - \frac{x^{11}}{39\,916\,800} + O[x]^{13}$$

```
potenzReihe[Sin[y], y, 3]
```

$$y - \frac{y^3}{6} + O[y]^4$$

Dieselbe Funktion in einer Notation mit Optionen

```
Clear[potenzReihe];
Options[potenzReihe] = {var → x, value → 0, order → 10};
potenzReihe[f_, options___?OptionQ] :=
 Series[f, {var, value, order} /. {options} /. Options[potenzReihe]]
```

```
potenzReihe[Sin[y], var → y, order → 7]
```

$$y - \frac{y^3}{6} + \frac{y^5}{120} - \frac{y^7}{5040} + O[y]^8$$

Upvalues und Downvalues

```
ClearAll["Global‘*"]
```

```
f[x_] := x^2 + x + 1;
?f
```

```
Global‘f
```

```
f[x_] := x^2 + x + 1
```

```
ClearAll[f, g]
f[g[x_]] := x^2
```

```
?f g
```

```
Global‘f
```

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

```
Global‘g
```

```
DownValues[f]
```

```
{HoldPattern[f[g[x_]]] :↦ x^2}
```

```
Dt[a, x_] := 0
```

```
SetDelayed::write : Tag Dt in Dt[a, x_] is Protected.
```

```
$Failed
```

```
Attributes[Dt]
```

```
{Protected}
```

```
ClearAll[a, x]
Dt[a, x_] ^:= 0
UpValues[a]
```

```
{HoldPattern[Dt[a, x_]] :↦ 0}
```

```
Table[Dt[a, {x, n}], {n, 1, 5}]
```

```
{0, 0, 0, 0, 0}
```

```
Dt[a, y]
```

```
0
```

```
ClearAll[a, x]
Dt[a, x] ^= 0;
UpValues /@ {a, x}
```

```
{{HoldPattern[Dt[a, x]] :↦ 0}, {HoldPattern[Dt[a, x]] :↦ 0}}
```

```
Clear[a, x]
a /: Dt[a, x] = 0;
UpValues /@ {a, x}
```

```
{{HoldPattern[Dt[a, x]] :↦ 0}, {}}
```

```
Table[Dt[a, {x, n}], {n, 1, 3}]
```

```
{0, Dt[a, {x, 2}], Dt[a, {x, 3}]}
```

```
a /: Dt[a, {x, n_Integer}] = 0;
Table[Dt[a, {x, n}], {n, 1, 3}]
```

```
{0, 0, 0}
```

▪ Zusammengesetzte Bezeichner

```
ClearAll["Global`*"]
```

```
x⃗ x⁺ xₐ
```

```
x⃗ xₐ x⁺
```

```
x = 3; a = 4;
```

```
x⃗ x⁺ xₐ
```

```
3⃗ 3₄ 3⁺
```

```
Clear[x, a]
```

```
x⃗ x⁺ xₐ // FullForm
```

```
Times[OverVector[x], Subscript[x, a], SuperPlus[x]]
```

? Global‘*

▼ **Global‘**

▮	i	plotpts$
a	j	potenzReihe
b	l	primeList
c	l1	s
clearAll	l2	sol
d	l3	spalten
delta	list1	start
delta$	list2	sublists
DotPlot	localOptions	sums
e	m	u
ec	ml	v
ec11	myGenericMatrix	value
end	n	var
f	options	veryGenericMatrix
func	opts	werteTafel
g	order	x
genericMatrix	p	x0
gleitenderDurchschnitt	p1	x$
gls	p2	y
glsys	PlotMultiple	z
h	plotpts	zeilen

```
Global‘▮
```

?? OverVector

OverVector[*expr*] displays with a right vector over *expr*.

x // SuperPlus

x^{+}

%

x^{+}

?? SuperPlus

SuperPlus[*expr*] displays as $expr^{+}$.

```
?? Subscript
```

```
Subscript[x, y] is an object that formats as x_y.
Subscript[x, y_1, y_2, ...] formats as x_{y_1,y_2,...}.
```

```
Attributes[Subscript] = {NHoldRest}
```

```
ClearAll["Global`*"]
```

```
SetAttributes[{OverVector, SuperPlus, Subscript}, HoldFirst]
```

```
x = 3; a = 4;
```

```
x⃗ x⁺ x_a
```

```
x⃗ x_4 x⁺
```

```
x_0 // Head
```

```
Subscript
```

Werte, die solchen zusammengesetzten Symbolen zugewiesen werden, sind nicht in der Symboltabelle, sondern als Funktionswerte unter dem entsprechenden Funktionssymbol gespeichert.

```
x_3 = 12; x_4 = 8;
```

```
?? Subscript
```

```
Subscript[x, y] is an object that formats as x_y.
Subscript[x, y_1, y_2, ...] formats as x_{y_1,y_2,...}.
```

```
Attributes[Subscript] = {HoldFirst, NHoldRest}
```

```
Subscript[x, 3] = 12
```

```
Subscript[x, 4] = 8
```

Sie können deshalb auch nicht mit **Clear**, sondern nur mit **Unset** (oder ...=.) gelöscht werden.

```
Clear[x_4]; x_4
```

```
Clear::ssym : x_4 is not a symbol or a string.
```

```
8
```

```
Unset[x_4]; x_4
```

```
x_4
```

```
?? Subscript
```

```
Subscript[x, y] is an object that formats as x_y.
Subscript[x, y_1, y_2, ...] formats as x_{y_1,y_2,...}.
```

```
Attributes[Subscript] = {HoldFirst, NHoldRest}
```

```
Subscript[x, 3] = 12
```

Clear kann jedoch auf das Funktionssymbol **Subscript** angewendet werden, was *alle* mit **Subscript** zusammengesetzten Symbole löscht.

```
ClearAll[Subscript]
```

```
?? Subscript
```

```
Subscript[x, y] is an object that formats as x_y.
Subscript[x, y_1, y_2, ...] formats as x_{y_1,y_2,...}.
```

Block versus Module

```
ClearAll["Global`*"]
```

```
shrink[pts_List, α_Real] := Module[{M}, M = Mean[pts] ; Map[α # + (1 - α) M &, pts]]
```

```
test = {{1, 1}, {3, 1}, {3, 2}};
GraphicsGrid[{{
   Graphics[{
     GrayLevel[.7], Polygon[test],
     GrayLevel[.5], Polygon[shrink[test, .5]]}],
   Graphics[{
     GrayLevel[.7], Polygon[test],
     GrayLevel[.5], Polygon[shrink[test, -.2]]}]
  }}]
```

Im Gegensatz zu klassischen Programmiersprachen können lokale Bezeichner, die in symbolischen Rückgabewerten vorkommen, über das Ende des Block hinaus "leben".

```
Module[{x}, x]
```

```
x$1983
```

```
f = x^2 + 3 x + 1;

Module[{x}, Print[x]; x = n; f]
x
```

```
x$1986
```

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

```
x
```

Der Inhalt des Statements wird umgebaut zu: x$123=n; f
Bei der Auswertung von f ist das dort vorkommende *x* das globale *x* und hat mit dem lokalen x$123 nichts zu tun. Das entspricht "lexical scoping", die Variable *x* ist lokal für das Code−Segment.

```
Block[{x}, Print[x]; x = n; f]
x
```

```
x
```

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

```
x
```

Der Inhalt von x wird auf dem Stack zwischengespeichert, die Zuweisung *x=n* ausgeführt und dann die Auswertung von f gestartet. Das *x* ist das globale *x*, das zur Zeit einen anderen Wert hat (nämlich n). Am Ende des Funktionsaufrufs wir der alte Wert von x wieder hergestellt.
Das entspricht "dynamical scoping"; Variablen sind nicht wirklich lokal, es wird lediglich deren Inhalt gerettet.

Eigene Operatoren definieren

```
Clear[CirclePlus]
CirclePlus[a_, b_] := (a + b)/(1 + a b)
CirclePlus[a__, b_] := CirclePlus[CirclePlus[a], b] // Together
```

```
x⊕y⊕z
```

$$\frac{x + y + z + x\,y\,z}{1 + x\,y + x\,z + y\,z}$$

```
0.4⊕0.4⊕0.4⊕0.4
```

```
0.93473
```

```
2⊕2
```

$$\frac{4}{5}$$

```
a⊕0⊕1
```

```
1
```

```
Clear[CirclePlus]
```

```
a⊕b⊗c // FullForm
```

```
CirclePlus[a, CircleTimes[b, c]]
```

```
(a⊕b)⊕c // FullForm
```

```
CirclePlus[CirclePlus[a, b], c]
```

Geschwindigkeitsoptimierung, Compile

Apfelmännchen−Beispiel ohne Compile

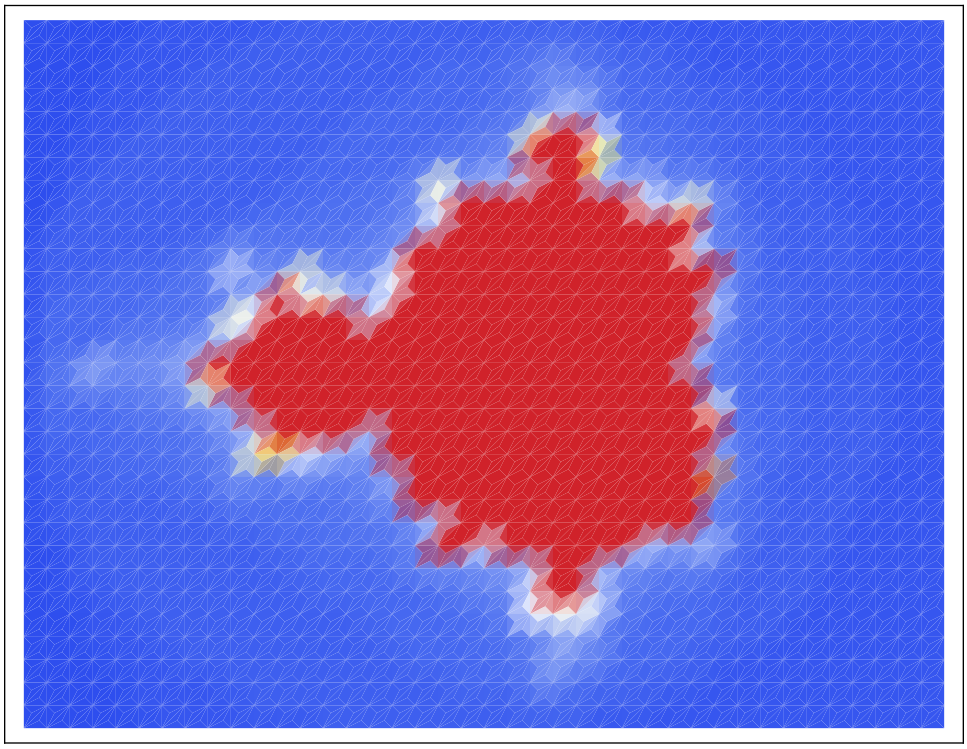
```
ApfelCalc[z0_Complex, nmax_Integer] :=
    Module[{n = 0, z = 0},
        While[Abs[z] < 2 && n < nmax, z = z^2 + z0; n++];
    n]
```

```
ApfelCalc[0.4 + 0.4 I, 20]
```

```
9
```

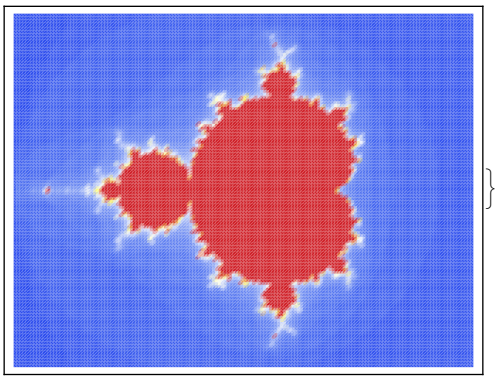
```
c = ColorData["TemperatureMap"];
ApfelPlot[z0_Complex, z1_Complex, nmax_Integer, dx_Real] :=
Module[{r0 = Re[z0], r1 = Re[z1], i0 = Im[z0], i1 = Im[z1]},
  ListDensityPlot[Table[ApfelCalc[r + I i, nmax], {i, i0, i1, dx}, {r, r0, r1, dx}],
   AspectRatio -> (i1 - i0)/(r1 - r0), FrameTicks → None, ColorFunction → c]
 ]
```

```
ApfelPlot[-2 - 1.25 I, 1.25 + 1.25 I, 50, 0.08]
```



```
Timing[ApfelPlot[-2 - 1.25 I, 1.25 + 1.25 I, 50, 0.02]]
```

{4.25,



}

... und mit Compile

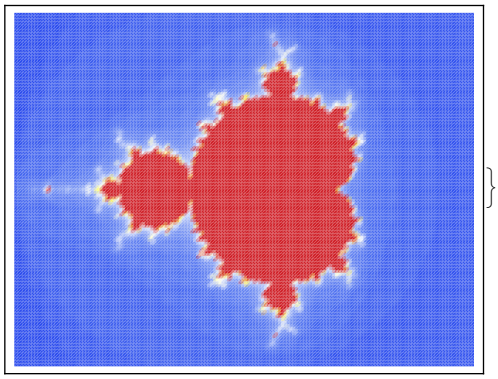
Beachten Sie, dass hier = und nicht := verwendet wird, denn die Compilierung soll ja während der Funktionsdefinition und nicht erst während des Aufrufs ausgeführt werden.

```
Clear[ApfelCalc];
ApfelCalc = Compile[{{z0, _Complex}, {nmax, _Integer}},
    Module[{n = 0, z = 0. I},
        While[Abs[z] < 2 && n < nmax,  z = z^2 + z0; n++];
    n]];
```

```
ApfelCalc[0.4 + 0.4 I, 20]
```

```
9
```

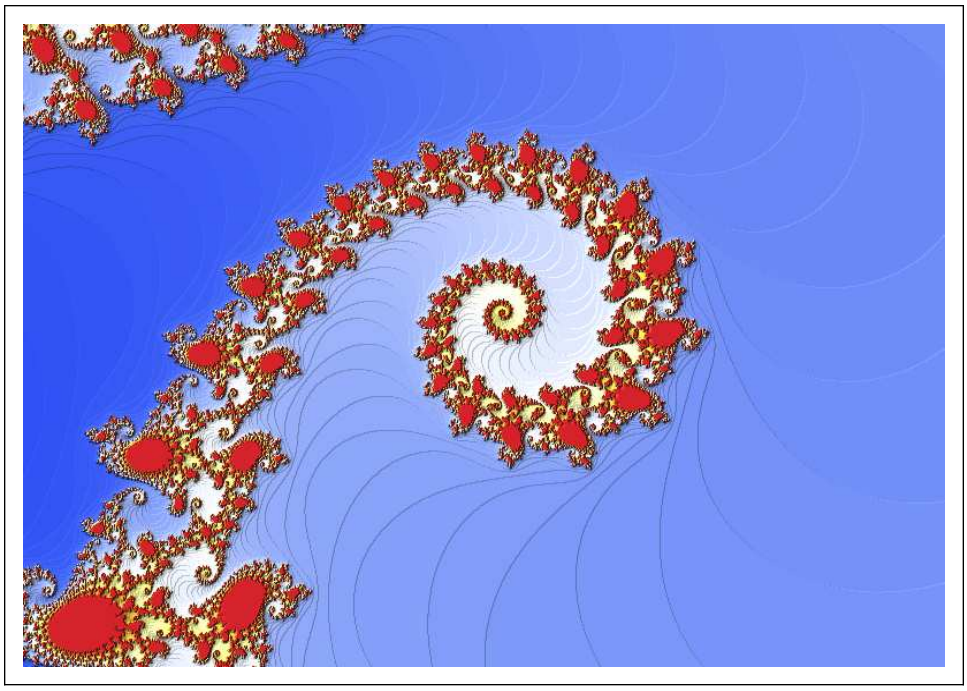
```
Timing[ApfelPlot[-2 - 1.25 I, 1.25 + 1.25 I, 50, 0.02]]
```



```
{1.4, }
```

```
ReliefApfelPlot[z0_Complex, z1_Complex, nmax_Integer, dx_Real] :=
Module[{r0 = Re[z0], r1 = Re[z1], i0 = Im[z0], i1 = Im[z1]},
  ReliefPlot[Table[ApfelCalc[r + I i, nmax], {i, i0, i1, dx}, {r, r0, r1, dx}],
   AspectRatio -> (i1 - i0)/(r1 - r0), FrameTicks → None, ColorFunction → c]
 ]
```

```
g = ReliefApfelPlot[-0.7476 + 0.0976 I, -0.7452 + 0.0993 I, 200, .3 10^-5]
```



■ Modularisierung und Pakete

```
ClearAll["Global‘*"]
```

In der folgenden Zelle ist der gesamte Code des Test−Fibonacci−Packages enthalten, der aus Teilen des Notebooks **mathcomp.nb** herauskopiert und entsprechend modifiziert wurde (insbesondere wurden alle Codeteile in die InputForm überführt, um "lesbaren" Ausgabecode zu produzieren − für *Mathematica* ist das nicht wesentlich, für "human reading" aber schon).
Markieren Sie nun diese Zelle als Initialisierungszelle (Cell|Cell Properties) und speichern Sie das Notebook ab. Es wird eine Datei **advanced.m** erzeugt, die den Package−Quellcode (und viele Kommentare) enthält.

```
(* VersuchsPackage *)
(* Zwei Definitionen zur Berechnung der Fibonaccifolge *)
BeginPackage["Test‘Fibonacci‘"]

fib::usage := "fib[n] berechnet die n-te Fibonaccizahl
mit einem iterativen Verfahren";

matfib::usage := "matfib[n] berechnet die n-te Fibonaccizahl
über Matrixpotenzen";

Begin["‘Private‘"]

fib[n_] := Module[{a = 0, b = 1}, Do[{a, b} = {b, a + b}, {n}]; a];

matfib[n_] := (MatrixPower[{{1, 1}, {1, 0}}, n] . {{1}, {0}})[[2, 1]];
End[]
EndPackage[]
```

```
Test‘Fibonacci‘

Test‘Fibonacci‘Private‘

Test‘Fibonacci‘Private‘
```

Kopieren Sie anschließend die Datei **prog−advanced.m** nach **Fib.m** und heben die Markierung der Zelle wieder auf.

Nun können Sie den Kernel neu starten und das Package mit *Needs[kontext,file]* laden.
Setzen Sie vorher mit SetDirectory[ ] das Wurzelverzeichnis.

```
SetDirectory["/home/graebe/Mma-Buch/Notebooks"]
Needs["Test‘Fibonacci‘", "Fib.m"]
```

```
/home/graebe/Mma-Buch/Notebooks
```

```
fib /@ Range[20]
```

```
{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765}
```

```
matfib /@ Range[20]
```

```
{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765}
```

```
Timing[fib[500 000];]
```

```
{8.63, Null}
```

```
Timing[matfib[500 000];]
```

```
{0.12, Null}
```