Mathematik mit Mathematica

Praktikum im Wintersemester 2021/22 an der TU Braunschweig
betreut von Prof. Dr. Michael Herrmann

Tutorium 05: Algorithmen und Programmierung

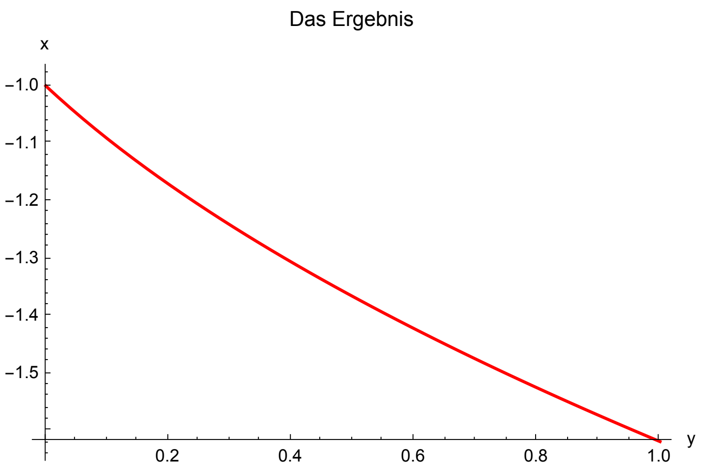
Vorbemerkung zum “guten” Programmieren

(1) Kommentare schreiben
(2) Einrückungen verwenden
(3) sinnvolle Bezeichnungen wählen (nicht zu kurz, nicht zu lang)
(4) zu komplizierte Schachtelungen vermeiden

Beispiel: zu kompakte Implementierung

In[∘]:=
```
Plot[x /. First[Solve[(x^2 + x == y * c) /. c -> 1, x]], {y, 0, 1},
 PlotLabel → "Das Ergebnis", AxesLabel → {"y", "x"}, PlotStyle → {Red}]
```

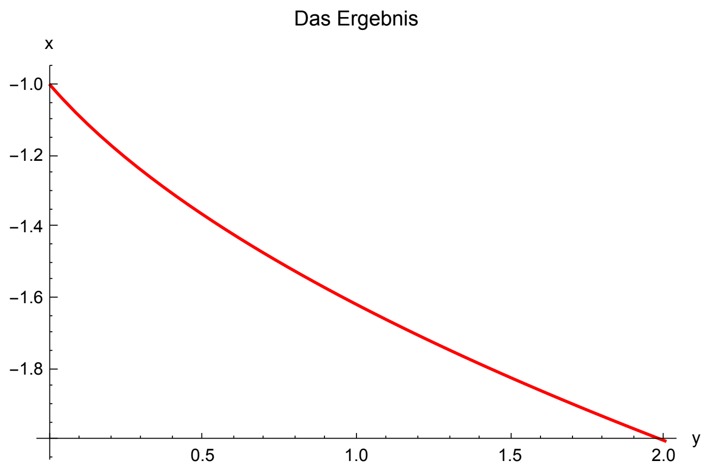
Out[∘]=



Beispiel: zu ausführliche Implementierung

In[∘]:=
```
(* Gleichung *)
Equation = x^2 + x == y * c;
(* löse die Gleichung nach x auf *)
SolutionSetRule =
  Solve[
   Equation ,  (* das wird gelöst *)
   x           (* hiernach wird aufgelöst *)
  ];
(* konvertiere die Regel in einen Ausdruck *)
SolutionSetExpression = x /. SolutionSetRule;
(* wähle den ersten Zweig *)
SolutionExpression = First[SolutionSetExpression];
(* setze den Wert des Parameters c und definiere Funktion in y *)
SolutionFunction[y_] = SolutionExpression /. c → 1;
(* plotte die Funktion über y *)
Plot[
 SolutionFunction[y],
 {y, 0, 2},
 PlotLabel → "Das Ergebnis",  (* der Titel *)
 AxesLabel → {"y", "x"},      (* das steht an den Achsen *)
 PlotStyle → {Red}            (* Graph der Funktion soll Rot sein *)
]
```

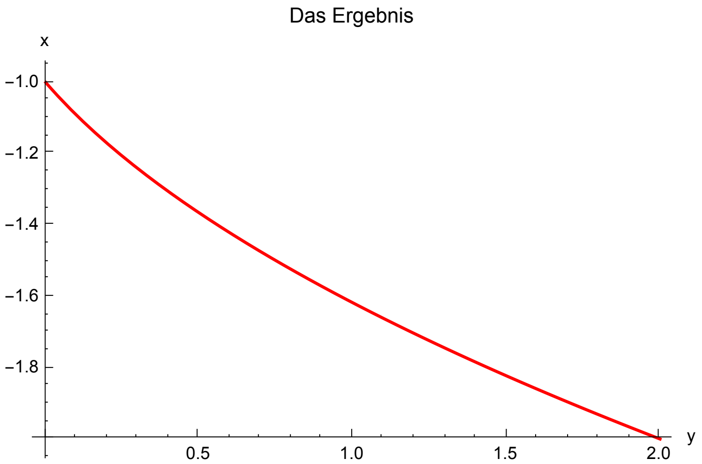
Out[∘]=



Beispiel: der goldene Mittelweg

In[∘]:=
```
(* die Gleichung *)
eqn = x^2 + x == y * c;
(* berechne x für c=1 als Funktion von y *)
sol[y_] = First[x /. Solve[eqn /. c → 1, x]];
(* plotte die Lösung *)
Plot[
 sol[y], {y, 0, 2},
 PlotLabel → "Das Ergebnis", AxesLabel → {"y", "x"}, PlotStyle → {Red}
]
```

Out[∘]=



Programmieren mit Mathematica

Verzweigungen

If

In[∘]:=
```
(* einfache binäre Entscheidung *)
num = RandomInteger[4];
If[
 (* Bedingung *)
 Mod[num, 2] == 0,
 (* wenn ja *)
 Print["gerade"],
 (* wenn nein *)
 Print["Ungerade"];
 Beep[]
]
```

```
gerade
```

Which

In[∘]:=
```
(* mehr als zwei Optionen *)
num = RandomInteger[10];
Which[
 (* Bedingung 1 *)
 Mod[num, 3] == 0,
 (* Befehl 1 *)
 Print["Fall 1"],
 (* Bedingung 2 *)
 Mod[num, 3] == 1,
 (* Befehl 2 *)
 Print["Fall 2"],
 (* Bedingung 3 *)
 Mod[num, 3] == 2,
 (* Befehl 3 *)
 Print["Fall 3"]
]
```

```
Fall 3
```

Schleifen

For

In[∘]:=
```
(*berechne die Summe 1+2+..+10 *)
sum = 0;
For[ (* 4 Argumente durch Komma getrennt *)
 (* C++-like HEADER *)
 i = 1,       (* HEADER 1: Startbefehl *)
 i ≤ 10,      (* HEADER 2: "Solange"-Bedingung *)
 i = i + 1,   (* HEADER 3: Inkrementbefehl *)
 (* BODY: eine oder mehrere Befehle, ggf. durch Semikolon getrennt *)
 sum = sum + i;
 Print["  Schritt " <> ToString[i]]
]
sum
```

```
Schritt 1
Schritt 2
Schritt 3
Schritt 4
Schritt 5
Schritt 6
Schritt 7
Schritt 8
Schritt 9
Schritt 10
```

Out[•]= 55

In[•]:=
```
(* es geht auch kompakter/komplizierter *)
(* beachte "Semikolon" versus "Komma" *)

For[
 sum = 0; i = 0,                         (* HEADER 1: zwei Startbefehle *)
 i ≤ 10,                                 (* HEADER 2: die "solange"-Bedingung *)
 sum += i; i++; Print[ToString[i]],
 (* HEADER 3: zwei Inkrementbefehle in kompakter Schreibweise und Plot *)
 (* BODY : hier machen wir gar nichts *)
]
sum
```

```
1
2
3
4
5
6
7
8
9
10
11
```

Out[•]= 55

Do

In[∘]:=
```
(* DO ersetzt FOR für den Hausgebrauch *)
sum = 0;
Do[sum = sum + i, {i, 1, 10}]
sum
```

Out[∘]=
```
55
```

While

In[∘]:=
```
step = 0;
(* eine andere Schleife,
bei der die Anzahl der Durchläufe nicht von vornherein feststeht *)
While[
 (* Bedingung *)
 step < 10,
 (* Befehle *)
 step++;
 Print["  Schritt " <> ToString[step]]
]
```

```
  Schritt 1
  Schritt 2
  Schritt 3
  Schritt 4
  Schritt 5
  Schritt 6
  Schritt 7
  Schritt 8
  Schritt 9
  Schritt 10
```

Nest

In[∘]:=
```
(* Rekursiver Aufruf einer Funktion *)
Nest[Sin, x, 15]
```

Out[∘]=
```
Sin[Sin[Sin[Sin[Sin[Sin[Sin[Sin[Sin[Sin[Sin[Sin[Sin[Sin[Sin[x]]]]]]]]]]]]]]]
```

In[∘]:=
```
(* es gibt nur eine positive Lösung der Gleichung x=Sin[x],
diese ist ein Attraktor *)
Nest[Sin, .3, 100 000]
Nest[Sin, 1.3, 100 000]
Nest[Sin, 2.3, 100 000]
```

Out[∘]= 0.00547618

Out[∘]= 0.00547699

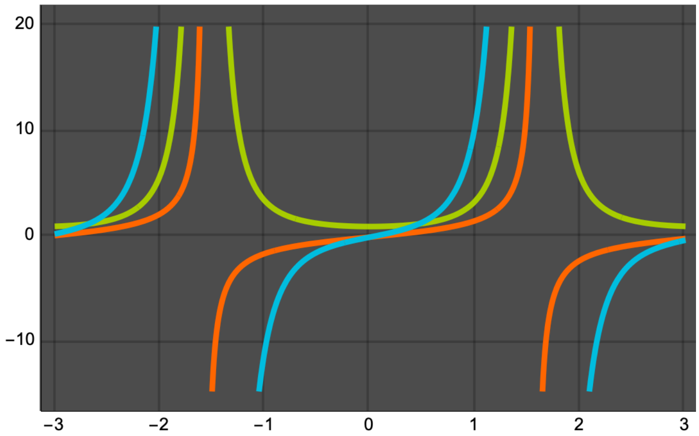
Out[∘]= 0.00547694

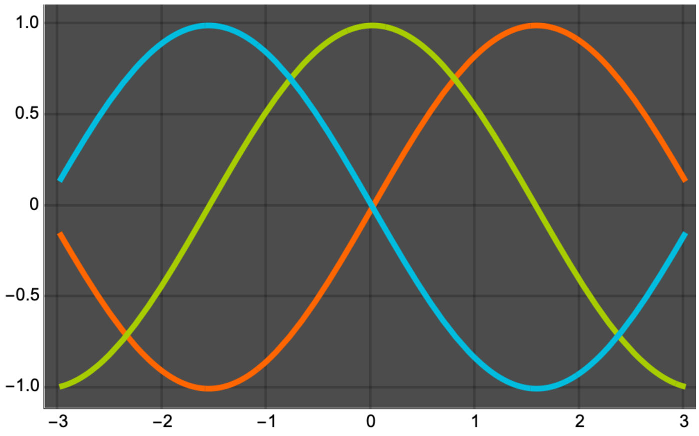
Programme

Funktionen

In[∘]:=
```
myPlot[f_] := (* ":=" ist hier besser als "=",
 weil ":=" wird erst zum Aufruf ausgeführt *)
 Plot[
  {f[x], f'[x], f''[x]},
  {x, -3, 3},
  PlotTheme -> "Marketing"
 ]
```

In[∘]:=
```
myPlot[Tan]
myPlot[Sin]
```

Out[∘]=


Out[∘]=


Module

In[•]:=
```
myProgram[x_, y_] :=        (* x und y sind Input Parameter *)
                            (* ":=" ist hier wieder besser als "=" *)
  Module[
   (* lokale Variablen *)
   {a, b, c, info},
   (* eine oder mehrere Befehle, ggf. durch Semikolon getrennt *)
   a = x;
   b = y * y;
   c = a + b;
   info =
    "  x=" <> ToString[x] <> " y=" <> ToString[y] <> "   res=" <> ToString[c];
   Print[info];
   Return[a + b]
  ];
```

In[•]:=
```
myProgram[4, 3]
```

```
x=4 y=3  res=13
```

Out[•]=
```
13
```

Rekursionen 1 : nochmal Summen

In[•]:=
```
Hyperlink["Brady Haran's Computerphile: What on Earth is Recursion?",
 "https://www.youtube.com/watch?v=Mv9NEXX1VHc"]
```

Out[•]=
```
Brady Haran's Computerphile: What on Earth is Recursion?
```

In[•]:=
```
SumOfNumbers[n_] :=
 If[
  (* teste n *)
  n == 0,
  (* n ist Null *)
  0,
  (* n ist nicht Null *)
  n + SumOfNumbers[n - 1]
 ]
```

In[•]:=
```
(* die gute alte Rechnung von Gauss,
nur mit einer Rekursion berechnet (Gauss war cleverer) *)
SumOfNumbers[100]
```

Out[•]=
```
5050
```

In[•]:=
```
SumOfNumbers[m]
```

Out[•]=
```
If[m == 0, 0, m + SumOfNumbers[m - 1]]
```

Rekursionen 2: Das Monster von Ackermann

In[∘]:=
```
Hyperlink[
 "Brady Haran's Computerphile: The Most Difficult Program to Compute?",
 "https://www.youtube.com/watch?v=i7sm9dzFtEI"]
```

Out[∘]= Brady Haran's Computerphile: The Most Difficult Program to Compute?

In[∘]:=
```
Ackermann[m_, n_] :=
 If[
  Print["  m=" <> ToString[m] <> " n=" <> ToString[n]];
  (* teste m *)
  m == 0,
  (* m ist Null *)
  n + 1,
  (* m ist nicht Null *)
  If[
   (* teste n *)
   n == 0,
   (* n ist Null *)
   Ackermann[m - 1, 1],
   (* m ist nicht Null *)
   Ackermann[m - 1, Ackermann[m, n - 1]] (* das hier ist der Monster-Befehl *)
  ]
 ]
```

In[∘]:=
```
Ackermann[1, 1]
```

```
  m=1 n=1
  m=1 n=0
  m=0 n=1
  m=0 n=2
```

Out[∘]= 3

In[∘]:=
```
Ackermann[2, 2]
```

```
m=2 n=2
m=2 n=1
m=2 n=0
m=1 n=1
m=1 n=0
m=0 n=1
m=0 n=2
m=1 n=3
m=1 n=2
m=1 n=1
m=1 n=0
m=0 n=1
m=0 n=2
m=0 n=3
m=0 n=4
m=1 n=5
m=1 n=4
m=1 n=3
m=1 n=2
m=1 n=1
m=1 n=0
m=0 n=1
m=0 n=2
m=0 n=3
m=0 n=4
m=0 n=5
m=0 n=6
```

Out[•]= 7

```
In[•]:= (* n=m=3 dauert schon ziemlich lange *)
(* für nur etwas größere Werte von n werden Sie Ackermann[n,n]
 selbst mit einem Supercomputer NICHT in der
 Ihnen verbleibenden Lebenszeit ausrechnen können *)
Ackermann[3, 3]
```

Out[•]= 61