Ableitungen und ihre Anwendungen

Ableitungen von Funktionen

Wir kennen Ableitungen von Funktionen $f : \mathbb{R} \to \mathbb{R}$ aus dem Mathematikunterricht. Sie geben uns darüber Auskunft, wie gross die Steigung der Tangente in einem bestimmten Punkt des Funktionsgraphen ist. Die Tangente stellt dabei die beste lineare Annäherung an den Funktionsgraph dar. Ableitungen beschreiben auch die lokale Änderungsrate der Funktion. Ableitungen erlauben es uns ausserdem, die Extrema und Wendepunkte einer Funktion zu bestimmen.

Die folgende Tabelle fasst die bekannten Ableitungen der Grundfunktionen zusammen.

$f(x)$	$f'(x)$	$f(x)$	$f'(x)$
x^n	$n \cdot x^n \quad (n \in \mathbb{R})$	$\sqrt{x}$	$\frac{1}{2 \cdot \sqrt{x}}$
e^x	e^x	a^x	$a^x \cdot \ln(a) \quad (a > 0, a \neq 1)$
$\ln(x)$	$\frac{1}{x}$	$\log_a(x)$	$\frac{1}{x \cdot \ln(a)} \quad (a > 0, a \neq 1)$
$\sin(x)$	$\cos(x)$	$\arcsin(x)$	$\frac{1}{\sqrt{1-x^2}}$
$\cos(x)$	$-\sin(x)$	$\arccos(x)$	$-\frac{1}{\sqrt{1-x^2}}$
$\tan(x)$	$\frac{1}{\cos^2(x)} = 1 + \tan^2(x)$	$\arctan(x)$	$\frac{1}{x^2+1}$
$\sinh(x)$	$\cosh(x)$	$\operatorname{arsinh}(x)$	$\frac{1}{\sqrt{x^2+1}}$
$\cosh(x)$	$\sinh(x)$	$\operatorname{arcosh}(x)$	$\frac{1}{\sqrt{x^2-1}}$
$\tanh(x)$	$\frac{1}{\cosh^2(x)} = 1 - \tanh^2(x)$	$\operatorname{artanh}(x)$	$-\frac{1}{x^2-1}$

Table 1: Ableitungen der Grundfunktionen

Neue Funktionen erhält man, indem man die Grundfunktionen aus Table 1 addiert, subtrahiert, multipliziert, dividiert und komponiert, d.h. Verkettungen der Form $(f \circ g)(x) = f(g(x))$ bildet. Um solche Funktionen abzuleiten, brauchen wir die Regeln aus Table 2. Mit diesen Regeln sind wir dann schon in der Lage, alle differenzierbaren Funktionen abzuleiten.

Summenregel	$\frac{d}{dx}(f(x) \pm g(x)) = f'(x) \pm g'(x)$
Produktregel	$\frac{d}{dx}(f(x) \cdot g(x)) = f'(x) \cdot g(x) + f(x) \cdot g'(x)$
Spezialfall: Faktorregel	$\frac{d}{dx}(a \cdot f(x)) = a \cdot f'(x)$
Quotientenregel	$\frac{d}{dx}\frac{f(x)}{g(x)} = \frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{g(x)^2}$
Kettenregel	$\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$

Table 2: Ableitungsregeln

An dieser Stelle sei noch angemerkt, dass sich der Begriff der Ableitung sinngemäss auf Funktionen $f : \mathbb{R}^n \to \mathbb{R}^m$ verallgemeinern lässt. Eine kurze Beschreibung der Grundidee findet sich

in @Slater2022. Weitergehende Informationen findet man z.B. in @Arens2022 oder in jedem Lehrbuch zur Analysis 2.

Programme als Funktionen

Programme, die numerische Werte einlesen und numerische Werte ausgeben, können als mathematische Funktionen betrachtet werden. Wir beschränken uns zunächst auf Programme, die nur ein Argument erhalten und nur einen Rückgabewert liefern.

Example 0.1 (Eine Funktion als Programm)**.**

```
def f(x):
    y = (2 + x) * (x - 3)
    return y

x0 = 2
print( f(x0) )
```

Diese Python-Funktion entspricht der Funktion $f : \mathbb{R} \to \mathbb{R}, x \mapsto y = (2 + x)(x - 3)$ im Sinne der Mathematik. Natürlich kann der Funktionskörper viel komplizierter aufgebaut sein und z.B. Schleifen und Bedingungen enthalten.

Um zu verstehen, wie der Computer einen Ausdruck wie `y = (2 + x) * (x - 3)` auswertet, ist es hilfreich, ihn als Baum (im Sinne der Graphentheorie) darzustellen. Ausdrucksbäume sind ein Spezialfall von so genannten *computational graphs* und werden z.B. in @Hromkovic2021 erklärt.

Wir wollen nun unsere Python-Funktion so umschreiben, dass diese Struktur auch im Funktionskörper sichtbar wird. Dazu führen wir drei Hilfsvariablen `v0, v1, v2` ein.

```
def f(x):
    v0 = x
    v1 = 2 + v0
    v2 = v0 - 3
    y = v1 * v2
    return y
```

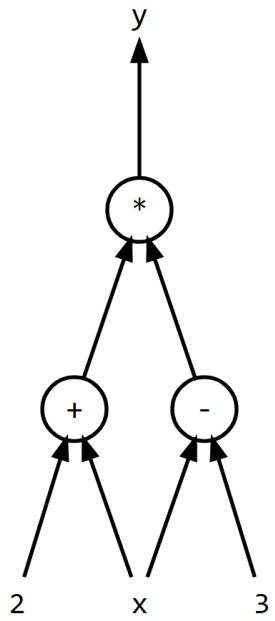

Figure 1: Ausdrucksbaum zum Ausdruck `y = (2 + x) * (x - 3)`.

Konvention

Eine Funktion berechnet aus einem Argument `x` einen Rückgabewert `y` über eine Reihe von Hilfsvariablen `v`, die mit aufsteigenden Indizes versehen sind. Dabei setzen wir am Anfang immer `v0 = x`.

Exercise 0.1 (Programm in Funktion übersetzen)**.** Schreibe die mathematische Funktion auf, die durch das folgende Programm berechnet wird.

```
import math

def f(x):
    v0 = x
    v1 = v0 ** 2
    v2 = v1 + 2
    v3 = -v1 / 2
    v4 = math.cos(v2)
    v5 = math.exp(v3)
    v6 = v4 * v5
    y = v6 + 1 / v0
    return y
```

Lösung

$$v_1 = x^2 \tag{1}$$
$$v_2 = x^2 + 2 \tag{2}$$
$$v_3 = -\frac{x^2}{2} \tag{3}$$
$$v_4 = \cos(x^2 + 2) \tag{4}$$
$$v_5 = e^{-\frac{x^2}{2}} \tag{5}$$
$$v_6 = \cos(x^2 + 2) \cdot e^{-\frac{x^2}{2}} \tag{6}$$
$$y = f(x) = \cos(x^2 + 2) \cdot e^{-\frac{x^2}{2}} + \frac{1}{x} \tag{7}$$

Exercise 0.2 (Funktion in Graph und Programm übersetzen)**.** Schreibe zur mathematischen Funktion $y = f(x) = \frac{\ln(x^2+1)}{\sqrt{x^2+1+x}}$ den Ausdrucksbaum auf. Übersetze den Ausdruck anschliessend in eine Python-Funktion gemäss der Konvention.

Lösung

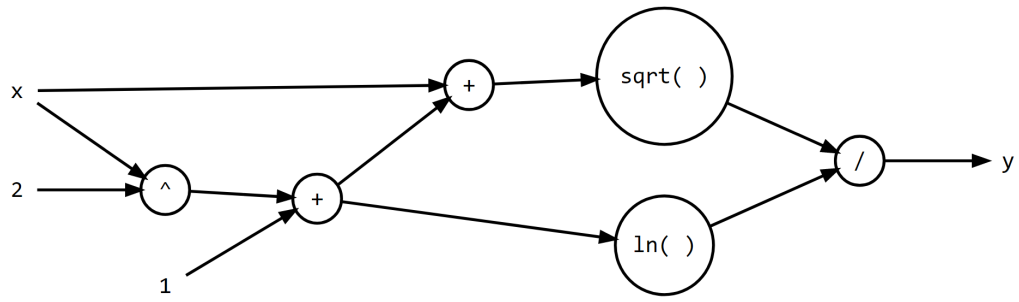

Figure 2: Computational Graph zum Ausdruck `y = ln(x^2 + 1) / sqrt(x^2 + 1 + x)`.

```
import math

def f(x):
    v0 = x
    v1 = v0 ** 2
    v2 = v1 + 1
    v3 = v2 + v0
    v4 = math.log(v2)
    v5 = math.sqrt(v3)
    y = v4 / v5
    return y
```

Natürlich hätte man z.B. `v3` und `v4` auch vertauschen können.

Exercise 0.3 (Ein Programm mit einer Schleife)**.** Betrachte das folgende Programm:

```
def f(x):
    v0 = x
```

```
    for i in range(2):
        v0 = v0 ** 2 + 1
    y = v0
    return y
```

Ersetze im Funktionskörper die Schleife durch mehrere Befehle, so dass immer noch der gleiche mathematische Ausdruck berechnet wird und unsere Konvention eingehalten wird. Welche mathematische Funktion wird durch die Python-Funktion berechnet? Was ändert sich, wenn stattdessen `for i in range(3)` oder `for i in range(4)` stehen würde?

Lösung

Für jeden Schleifendurchgang benötigen wir eine neue Hilfsvariable. Die Funktion, die dabei entsteht, kann geschrieben werden als $f(x) = (\ell \circ \ell \circ ... \circ \ell)(x)$, wobei $\ell(x) = x^2 + 1$ ist.

`range(2)`

```
def f(x):
    v0 = x
    v1 = v0 ** 2 + 1
    v2 = v1 ** 2 + 1
    y = v2
    return y
```

$$f(x) = \ell(\ell(x)) \tag{8}$$
$$= (x^2 + 1)^2 + 1 = x^4 + 2x^2 + 2 \tag{9}$$

`range(3)`

```
def f(x):
    v0 = x
    v1 = v0 ** 2 + 1
    v2 = v1 ** 2 + 1
    v3 = v2 ** 2 + 1
    y = v3
    return y
```

$$f(x) = \ell(\ell(\ell(x))) \tag{10}$$
$$= ((x^2+1)^2+1)^2+1 = x^8+4x^6+8x^4+8x^2+5 \tag{11}$$

`range(4)`

```
def f(x):
    v0 = x
    v1 = v0 ** 2 + 1
    v2 = v1 ** 2 + 1
    v3 = v2 ** 2 + 1
    v4 = v3 ** 2 + 1
    y = v4
    return y
```

$$f(x) = \ell(\ell(\ell(\ell(x)))) \tag{12}$$
$$= (((x^2+1)^2+1)^2+1)^2+1 \tag{13}$$
$$= x^{16}+8x^{14}+32x^{12}+80x^{10}+138x^8+168x^6+144x^4+80x^2+26 \tag{14}$$

Unser Ziel: Programme ableiten

Wie eingangs erwähnt wurde, haben Ableitungen viele nützliche Anwendungen.

Wir möchten nun Ableitungen von Funktionen berechnen, die durch Programme beschrieben werden, die wie oben einen numerischen Parameter `x` als Input erhalten und einen numerischen Wert `y` zurückliefern. Unser Ziel wird es sein, die Programme so zu modifizieren, dass der Funktionsaufruf `f(x0)` nicht nur den Funktionswert $f(x_0)$ zurückgibt, sondern auch den Wert der Ableitung $f'(x_0)$. Wir sind dabei nicht an einer symbolischen Ableitung interessiert, wie das z.B. GeoGebra oder Mathematica machen (s. **?@sec-ADnotSymbDiff**), sondern nur an einer punktweisen Auswertung. Natürlich wollen wir die Ableitungsfunktion auch nicht von Hand bestimmen. Wir wollen uns aber auch nicht bloss mit einer Annhäerung des Wertes der Ableitung zufrieden geben (s. **?@sec-ADnotNumDiff**), sondern den Wert von $f'(x_0)$ bis auf Maschinengenauigkeit exakt berechnen. In **?@sec-SADforOneDimFunctions** werden wir eine Methode kennen lernen, die all dies leistet und dabei die Laufzeit eines Programms nicht wesentlich erhöht. Der Name dieser Methode: Algorithmische Differentiation (AD), obwohl die Namensgebung hier nicht eindeutig ist:

> One of the obstacles in this area [of computing derivatives], which involves "symbolic" and "numerical" methods, has been a confusion in terminology [...]. There

> is not even general agreement on the best name for the field, which is frequently referred to as *automatic* or *computational differentiation* in the literature. For this book the adjective *algorithmic* seemed preferable, because much of the material emphasizes algorithmic structure, sometimes glossing over the details and pitfalls of actual implementations. (Aus dem Vorwort zu @Griewank2008EDP)

Bevor wir uns aber der Methode der algorithmischen Differentiation zuwenden, wollen wir sie durch einige Beispiele motivieren, bei denen die Berechnung von Ableitung von Funktionen und Programmen eine zentrale Rolle spielt: Das Newtonverfahren und das Gradient Descent Verfahren.

Das Newtonverfahren zur Berechnung von Nullstellen

In vielen Anwendungen steht man vor der Aufgabe, die Gleichung $f(x) = 0$ nach x aufzulösen, d.h. eine Nullstelle $\bar{x}$ der Funktion zu finden. Oft ist es aber nicht möglich, die Lösung einer solchen Gleichung in geschlossener Form darzustellen. Um dennoch eine Lösung zumindest näherungsweise berechnen zu können, kann man folgendermassen vorgehen:

1. Wähle einen Startwert x_0, der in der Nähe einer Nullstelle $\bar{x}$ von f liegt.
2. Im Kurvenpunkt $(x_0|y_0)$ wird die Tangente an die Kurve f gelegt. Deren Schnittpunkt x_1 mit der x-Achse liegt in der Regel näher bei $\bar{x}$ als x_0.
3. Nun wiederholt man das Verfahren, indem man bei x_1 die Tangente an die Kurve legt, usw. Auf diese Weise erhält man eine Folge von Näherungen $x_0, x_1, x_2, ...$, deren Grenzwert die Nullstelle $\bar{x}$ ist.

Dieser Algorithmus ist als Newtonverfahren bekannt.

Die Gleichung der Tangente im Punkt $(x_n|y_n) = (x_n|f(x_n))$ ist bekanntlich $t(x) = f(x_n) + f'(x_n) \cdot (x - x_n)$. Die Nullstelle der Tangente ist der Näherungswert x_{n+1}. Aus $t(x_{n+1}) = 0$ ergibt sich nun die Iterationsvorschrift des Newtonverfahrens:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{15}$$

Exercise 0.4 (Das Newtonverfahren programmieren)**.** Schreibe ein Programm, das mit Hilfe des Newtonverfahrens (Equation 15) eine Nullstelle der Funktion $f(x) = \frac{1}{31}x^3 - \frac{1}{20}x^2 - x + 1$ berechnet. Verwende den Startwert $x_0 = -2$. Du kannst abbrechen, wenn die Differenz $|x_{n+1} - x_n|$ kleiner als eine bestimmte Toleranz wird, z.B. kleiner als `tol = 1e-6`. Wie flexibel ist dein Programm einsetzbar? Überlege dir z.B., wie viele Änderungen du vornehmen müsstest, wenn du die Nullstelle einer anderen Funktion berechnen müsstest.

Lösung

Welche der folgenden Lösungsvorschläge kommt deinem Programm am nächsten?

Version 1

```
from math import fabs

x0 = -2
tol = 1e-6
# Erster Schritt berechnen
x1 = x0 - (1/31 * x0**3 - 1/20 * x0**2 - x0 + 1) / (3/31 * x0**2 - 1/10 * x0 - 1)
while fabs(x1 - x0) > tol:
    x0 = x1
    x1 = x0 - (1/31 * x0**3 - 1/20 * x0**2 - x0 + 1) / (3/31 * x0**2 - 1/10 * x0 - 1)
print(x1)
```

```
5.908619865450271
```

Das Newtonverfahren wird als main-Funktion (d.h. im Hauptprogramm) ausgeführt. Braucht man jedoch die Nullstelle einer anderen Funktion, dann muss ein neues Programm geschrieben werden. Die Ableitung wurde von Hand berechnet.

Version 2

```
from math import fabs

def f(x):
    y = 1/31 * x**3 - 1/20 * x**2 - x + 1
    return y

def fdot(x):
    ydot = 3/31 * x**2 - 1/10 * x - 1
    return ydot

x0 = -2
tol = 1e-6
# Erster Schritt berechnen
x1 = x0 - f(x0) / fdot(x0)
while fabs(x1 - x0) > tol:
    x0 = x1
    x1 = x0 - f(x0) / fdot(x0)
print(x1)
```

```
5.908619865450271
```

Das Newtonverfahren wird als main-Funktion (d.h. im Hauptprogramm) ausgeführt, aber die Berechnung von f und ihrer Ableitung f' wurde in zwei Funktionen `f` und `fdot` ausgelagert. Das macht das Programm übersichtlicher und flexibler. Die Ableitung wurde wieder von Hand berechnet.

Version 3

```python
from math import fabs

def newton(f, fdot, x0):
    tol = 1e-6
    # Erster Schritt berechnen
    x1 = x0 - f(x0) / fdot(x0)
    while fabs(x1 - x0) > tol:
        x0 = x1
        x1 = x0 - f(x0) / fdot(x0)
    return x1

def f(x):
    y = 1/31 * x**3 - 1/20 * x**2 - x + 1
    return y

def fdot(x):
    ydot = 3/31 * x**2 - 1/10 * x - 1
    return ydot

x0 = -2
xbar = newton(f, fdot, x0)
print(xbar)
```

```
5.908619865450271
```

Das Newtonverfahren wird als eigene Funktion `newton(f, fdot, x0)` implementiert. Dieser werden die Funktion f und ihre Ableitung f', sowie der Startwert x_0 als Argumente übergeben. Sie kann dann im Hauptprogramm aufgerufen werden. Die Ableitung wurde aber immer noch von Hand berechnet.

Version 4

```python
from math import fabs

def newton(f, x0):
    tol = 1e-6
    # Erster Schritt berechnen
    # Ableitung von f an der Stelle x0 annähern
    h = 1e-6
    ydot = ( f(x0 + h) - f(x0) ) / h
    x1 = x0 - f(x0) / ydot
    while fabs(x1 - x0) > tol:
        x0 = x1
        ydot = ( f(x0 + h) - f(x0) ) / h
        x1 = x0 - f(x0) / ydot
    return x1

def f(x):
    y = 1/31 * x**3 - 1/20 * x**2 - x + 1
    return y

x0 = -2
xbar = newton(f, x0)
print(xbar)
```

```
5.90861986545027
```

Hier wird das Newtonverfahren in einer Funktion implementiert. Die Ableitung wird nicht mehr von Hand berechnet, sondern innerhalb der Funktion mit $f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h}$ angenähert. Dabei wird einfach `h = 1e-6` gesetzt und gehofft, dass der entstehende Rundungsfehler klein genug ist. Beachte aber, dass sich der berechnete Wert von der Ausgabe in den anderen Versionen leicht unterscheidet.

Auch die Version 4 der vorgestellten Lösung ist noch nicht befriedigend. Als wir die Ableitung von Hand berechnet hatten, musste nur die Funktion `fdot` and der Stelle `x0` ausgewertet werden, um den (bis auf Maschinengenauigkeit) *exakten* Wert von $f'(x_0)$ zu erhalten. Bei der letzten Methode muss man sich mit einem Näherungswert der Ableitung zufrieden geben. Auch wenn der Wert in diesem Beispiel gut genug war [1], so haben wir doch keine Garantie, dass wir für alle Funktionen einen vernünftigen Wert erhalten. Auf die Probleme, die mit dieser Annäherung von $f'(x_0)$ auftreten, wird in **?@sec-ADnotNumDiff** näher eingegangen.

[1] Das Newton-Verfahren hat die angenehme Eigenschaft, dass kleine Rundungsfehler automatisch ausgeglichen werden. Auf andere numerische Verfahren, die die Ableitung verwenden, trifft dies aber nicht zu.

Example 0.2 (Billard auf einem runden Tisch)**.** Wir betrachten ein Beispiel aus @Gander2015. Platziere die weisse und die blaue Billardkugel auf dem runden Tisch. Das Ziel ist es, die weisse Kugel so anzustossen, dass sie die blaue Kugel trifft, nachdem sie vorher genau einmal an die Bande gespielt wurde.

Aus Symmetriegründen dürfen wir annehmen, dass der Rand des Billardtisches der Einheitskreis ist und dass die weisse Kugel auf der x-Achse liegt. Die blaue Kugel habe die Koordinaten $(x_P|y_P)$. Weiter sei X der Punkt auf dem Einheitskreis, an dem die weisse Kugel abprallt. Wir beschreiben diesen Punkt mit seinen Polarkoordinaten $X = (\cos(x)|\sin(x))$. Unser Ziel ist es, x so zu berechnen, dass die weisse Kugel die blaue trifft, nachdem sie bei X an die Bande gestossen ist. Dabei verhält sie sich so, als ob sie an der Kreistangente in X reflektiert wird. Der Tangentenvektor im Punkt X lautet $\vec{t} = \begin{pmatrix} -\sin(x) \\ \cos(x) \end{pmatrix}$.

Wir betrachten nun die Einheitsvektoren $\vec{e}_Q$ in Richtung $\overrightarrow{XQ}$ und $\vec{e}_P$ in Richtung $\overrightarrow{XP}$. Wenn die weisse Kugel die blaue treffen soll, dann müssen die Winkel zwischen der Tangente und diesen Vektoren gleich sein. Das ist genau dann der Fall, wenn $\vec{t}$ senkrecht steht auf $\vec{e}_Q + \vec{e}_P$. Wir müssen also x so bestimmen, dass $\vec{t} \cdot (\vec{e}_Q + \vec{e}_P) = 0$ ist.

Das folgende Programm berechnet das Skalarprodukt der linken Seite dieser Gleichung.

```
import math
import matplotlib.pyplot as plt

def f(x):
    # Parameter
    a = -0.8           # Position von Q = (a|0)
    px, py = 0.5, 0.5  # Position von P = (px|py)

    # Berechnung des Skalarprodukts
    v0 = x
    v1 = math.cos(v0)  # x-Koordinate von X
    v2 = math.sin(v0)  # y-Koordinate von X
    v3 = px - v1       # x-Komponente des Vektors XP
    v4 = py - v2       # y-Komponente des Vektors XP
    v5 = math.sqrt(v3**2 + v4**2)  # Länge des Vektors XP
    v6 = v3 / v5       # x-Komponente des Einheitsvektors eP
    v7 = v4 / v5       # y-Komponente des Einheitsvektors eP

    v8 = a - v1        # x-Komponente des Vektors XQ
    v9 = -v2           # y-Komponente des Vektors XQ
    v10 = math.sqrt(v8**2 + v9**2)  # Länge des Vektors XQ
    v11 = v8 / v10     # x-Komponente des Vektors eQ
```

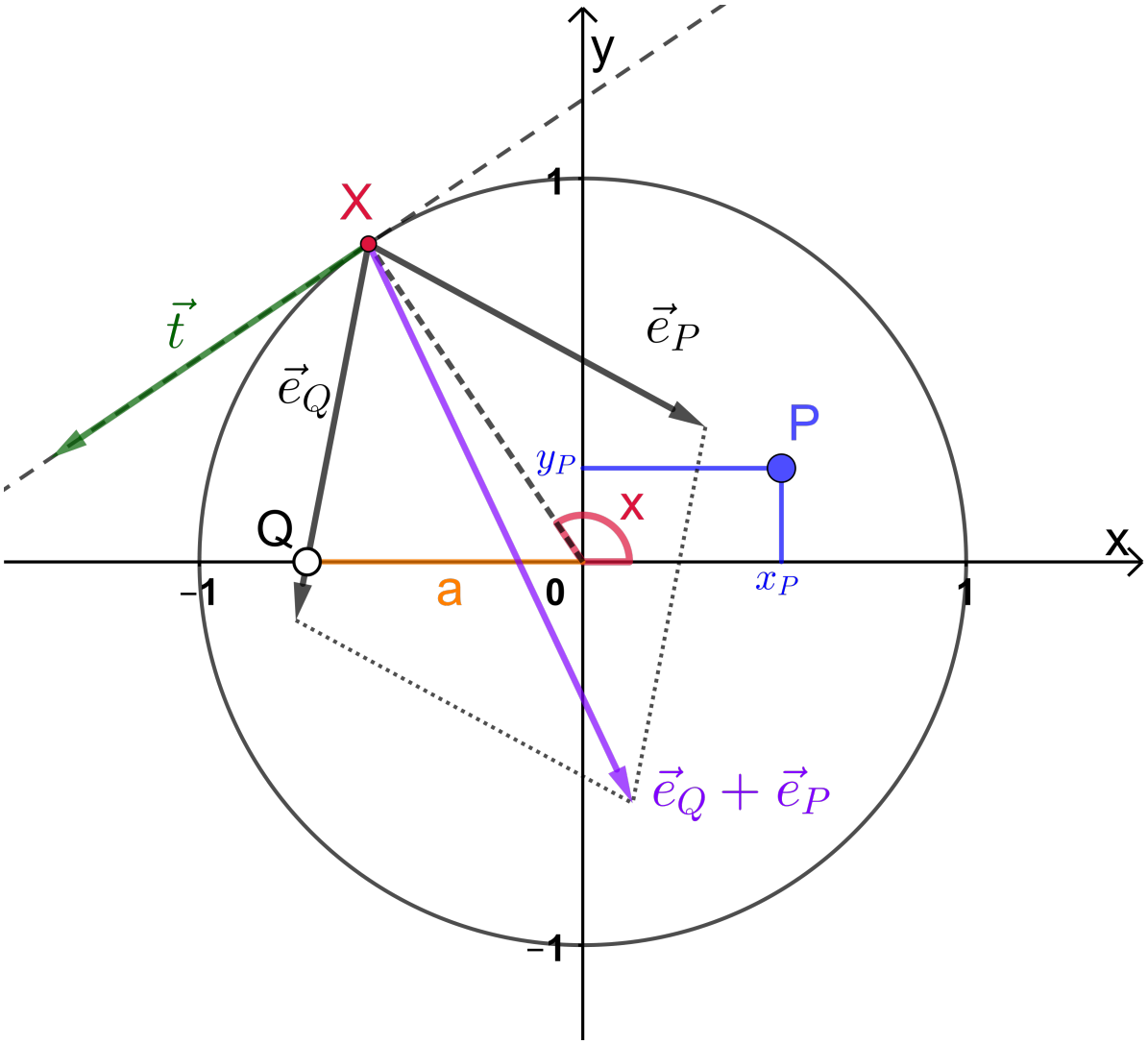

Figure 3: Billard auf einem runden Tisch

```
    v12 = v9 / v10      # y-Komponente des Vektors eQ
    y = (v6 + v11) * v2 - (v7 + v12) * v1  # Skalarprodukt
    return y

# Graph der Funktion f(x) plotten
fig = plt.figure()
ax = plt.gca()
ax.set_xlim((0,2*math.pi))
ax.set_ylim((-1.5,1.5))
X = [2*math.pi * k / 1000 for k in range(1001)]
Y = [f(x) for x in X]
plt.plot([0, 2*math.pi], [0, 0], 'k--') # x-Achse
plt.plot(X,Y)
plt.xticks([0, math.pi/2, math.pi, 3*math.pi/2, 2*math.pi],
           ['0', ' /2', ' ', '3 /2', '2 '])
plt.show()
```

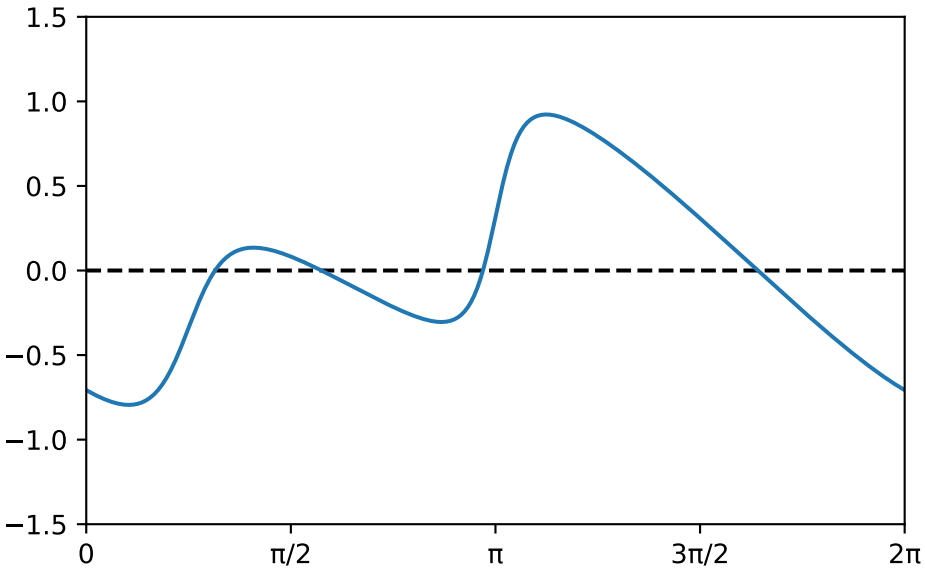

Figure 4: Graph des Skalarprodukts als Funktion des Polarwinkels x des Punktes $X = (cos(x)|sin(x))$. Die Nullstellen entsprechen den Winkeln, bei denen die weisse Kugel die blaue Kugel trifft, nachdem sie genau einmal an die Bande gespielt wurde.

Wir möchten die Nullstellen der Funktion `f(x)` mit unserer Funtion `newton` bestimmmen. Dazu müssen wir jedoch die Ableitung von `f` berechnen.

Gradient Descent zum Auffinden lokaler Minima

Eine weitere wichtige Aufgabe besteht darin, ein Minimum einer Funktion zu finden. Auch hier wollen wir mit Hilfe der Ableitung eine Folge von Näherungswerten $x_0, x_1, x_2, ...$ finden, deren Grenzwert die x-Koordinate eines (lokalen) Minimums von f ist.

Wenn $f'(x_n) > 0$ ist, dann wissen wir, dass die Funktion f an der Stelle x_0 streng monoton wachsend ist. D.h., dass die Funktionswerte links von x_n kleiner sind, als an der Stelle x_n. Analog gilt, dass wenn $f'(x_n) < 0$ ist, die Funktion monoton fallend ist und wir uns nach rechts bewegen sollten, um ein Minimum zu finden. In der Nähe eines Minimums ist ausserdem $|f'(x)|$ sehr klein und wir können entsprechend kleinere Schritte machen, um uns diesem anzunähern. Um also von x_n zu x_{n+1} zu kommen, machen wir einen Schritt, der proportional zu $-f'(x_n)$ ist. Mit dem Proporionalitätsfaktor $\lambda \in \mathbb{R}$ und einem geeignet gewählten Startwert x_0 erhalten wir die Iterationsvorschrift

$$x_{n+1} = x_n - \lambda \cdot f'(x_n) \tag{16}$$

Exercise 0.5 (Eigenschaften der Gradient Descent Methode)**.** Experimentiere mit verschiedenen Funktionen und verschiedenen Schrittweiten λ. Was passiert, wenn die Schrittweite zu klein bzw. zu gross gewählt wird? Was passiert, wenn f an der Stelle x_0 ein lokales Maximum aufweist? Was passiert in der Nähe eines Sattelpunktes?

Lösung

Ist λ zu klein, dann konvergiert das Verfahren nur sehr langsam. Ist λ dagegen zu gross, dann kann es passieren, dass die Iteration zwischen zwei oder mehr Werten hin- und herspringt oder sogar nach $\pm\infty$ divergiert.
Falls x_0 gerade mit der Stelle eines lokalen Maximums oder eines Sattelpunktes zusammenfällt, gilt auch $f'(x_0) = 0$ und damit auch $x_n = x_0$ für alle $n \in \mathbb{N}$. Maxima sind aber labile Gleichgewichtspunkte in dem Sinn, dass sich x_n von ihnen wegbewegt, wenn x_0 auch nur ein bisschen links oder rechts davon liegt. Ähnlich verhält es sich bei Sattelpunkten. Die Folge konvergiert gegen die Stelle des Sattelpunktes, wenn $f(x_0)$ grösser als der y-Wert des Sattelpunktes ist und λ nicht zu gross ist.

Exercise 0.6 (Gradient Descent programmieren)**.** Schreibe ein Programm, das mit Hilfe des Gradient Descent Verfahrens (Equation 16) ein lokales Minimums der Funktion $f(x) = \frac{1}{16}x^4 - \frac{1}{3}x^3 + \frac{1}{8}x^2 + x + 2$ berechnet. Verwende den Startwert $x_0 = 1.5$ und die Schrittweite $\lambda = 0.5$.

Du kannst abbrechen, wenn die Differenz $|x_{n+1} - x_n|$ kleiner als eine bestimmte Toleranz wird, z.B. kleiner als `tol = 1e-6`. Wie flexibel ist dein Programm einsetzbar? Überlege dir z.B., wie viele Änderungen du vornehmen müsstest, wenn du ein lokales Minimum einer anderen Funktion berechnen müsstest.

Lösung

Welche der folgenden Lösungsvorschläge kommt deinem Programm am nächsten?

Version 1

```
from math import fabs

x0 = 1.5
lam = 0.5
tol = 1e-6
# Erster Schritt berechnen
x1 = x0 - lam * (1/4 * x0**3 - x0**2 + 1/4 * x0 + 1)
while fabs(x1-x0) > tol:
    x0 = x1
    x1 = x0 - lam * (1/4 * x0**3 - x0**2 + 1/4 * x0 + 1)
print(x1)
```

```
3.3429230748530196
```

Das Gradient Descent Verfahren wird als main-Funktion (d.h. im Hauptprogramm) ausgeführt. Um das Minimum einer anderen Funktion zu bestimmen, muss ein neues Programm geschrieben werden. Die Ableitung wurde von Hand berechnet

Version 2

```python
from math import fabs

def fdot(x):
    ydot = 1/4 * x**3 - x**2 + 1/4 * x + 1
    return ydot

x0 = 1.5
lam = 0.5
tol = 1e-6
# Erster Schritt berechnen
x1 = x0 - lam * fdot(x0)
while fabs(x1-x0) > tol:
    x0 = x1
    x1 = x0 - lam * fdot(x0)
print(x1)
```

```
3.3429230748530196
```

Das Gradient Descent Verfahren wird als main-Funktion (d.h. im Hauptprogramm) ausgeführt, aber die Berechnung von f' wurde in die Funktion `fdot(x)` ausgelagert. Das macht das Programm etwas flexibler. Die Ableitung wurde wieder von Hand berechnet.

Version 3

```python
from math import fabs

def gradient_descent(fdot, x0, lam):
    tol = 1e-6
    # Erster Schritt berechnen
    x1 = x0 - lam * fdot(x0)
    while fabs(x1-x0) > tol:
        x0 = x1
        x1 = x0 - lam * fdot(x0)
    return x1

def fdot(x):
    ydot = 1/4 * x**3 - x**2 + 1/4 * x + 1
    return ydot

x0 = 1.5
lam = 0.5
xmin = gradient_descent(fdot, x0, lam)
print(xmin)
```

```
3.3429230748530196
```

Das Gradient Descent Verfahren wird als eigene Funktion `gradient_descent(fdot, x0, lam)` implementiert. Dieser Funktion werden die Ableitung f', der Startwert x_0, sowie die Schrittweite λ als Argumente übergeben. Sie kann dann im Hauptprogramm aufgerufen werden. Die Ableitung wurde aber immer noch von Hand berechnet.

Version 4

```
from math import fabs

def gradient_descent(f, x0, lam):
    tol = 1e-6
    # Erster Schritt berechnen
    # Ableitung an der Stelle x0 annähern
    h = 1e-6
    ydot = ( f(x0 + h) - f(x0) ) / h
    x1 = x0 - lam * ydot
    while fabs(x1-x0) > tol:
        x0 = x1
        ydot = ( f(x0 + h) - f(x0) ) / h
        x1 = x0 - lam * ydot
    return x1

def f(x):
    y = 1/16 * x**4 - 1/3 * x**3 + 1/8 * x**2 + x + 2
    return y

x0 = 1.5
lam = 0.5
xmin = gradient_descent(fdot, x0, lam)
print(xmin)
```

```
2.535183236464121
```

Das Gradient Descent Verfahren wird als eigene Funktion `gradient_descent(f, x0, lam)` implementiert. Dieser Funktion werden die ursprüngliche Funktion f, der Startwert x_0, sowie die Schrittweite λ als Argumente übergeben. Die Ableitung wird nicht mehr von Hand berechnet, sondern durch den Differenzenquotienten $f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h}$ angenähert. Dabei wird einfach `h = 1e-6` gesetzt und gehofft, dass der entstehende Rundungsfehler klein genug ist. Offensichtlich ist diese Annahme jedoch nicht gerechtfertigt.

Die Exercise 0.6 verdeutlicht nochmals das Problem, welches wir bereits in Exercise 0.4 gesehen haben. Wir müssen für den Algorithmus die Ableitung f' an mehreren Stellen auswerten. Wir möchten aber die Ableitung einerseits nicht von Hand berechnen und andererseits können wir uns auch nicht mit einer Approximation zufrieden geben.

Wir beschliessen dieses Kapitel mit einer praktischen Anwendung der Gradient Descent Methode.

Example 0.3 (Abstand zwischen Eillipse und Gerade)**.** Die Punkte P und Q bewegen sich auf Ellipsen im Raum. Die Position des Punktes P zur Zeit t ist gegeben durch

$$x_P(t) = 2\cos(t) - 1 \tag{17}$$
$$y_P(t) = 1.5\sin(t) \tag{18}$$
$$z_P(t) = 0 \tag{19}$$

und die Position von Q zum Zeitpunkt t lässt sich durch

$$x_Q(t) = -3\sin(2t) \tag{20}$$
$$y_Q(t) = 2\cos(2t) + 1 \tag{21}$$
$$z_Q(t) = 2\sin(2t) + 1 \tag{22}$$

bestimmen.

Der Abstand zwischen den beiden Punkten lässt sich zu jedem Zeitpunkt t berechnen durch $d = d(t) = |\overrightarrow{PQ}|$. Das folgende Programm berechnet diese Funktion und zeichnet ihren Graph.

```
import math

def d(t):
    v0 = t
    v1 = 2 * math.cos(v0) - 1    # x-Koordinate von P
    v2 = 1.5 * math.sin(v0)      # y-Koordinate von P
    v3 = 0                       # z-Koordinate von P
    v4 = -3 * math.sin(2*v0)     # x-Koordinate von Q
    v5 = 2 * math.cos(2*v0) + 1  # y-Koordinate von Q
    v6 = 2 * math.sin(2*v0) + 1  # z-Koordinate von Q
    y = math.sqrt((v4-v1)**2 + (v5-v2)**2 + (v6-v3)**2)
    return y

# Graph der Funktion d(t) plotten
fig = plt.figure()
ax = plt.gca()
ax.set_xlim((0,2*math.pi))
ax.set_ylim((0,6))
T = [2*math.pi * k / 1000 for k in range(1001)]
Y = [d(t) for t in T]
plt.plot(T,Y)
```

```
plt.xticks([0, math.pi/2, math.pi, 3*math.pi/2, 2*math.pi],
           ['0', ' /2', ' ', '3 /2', '2 '])
plt.show()
```

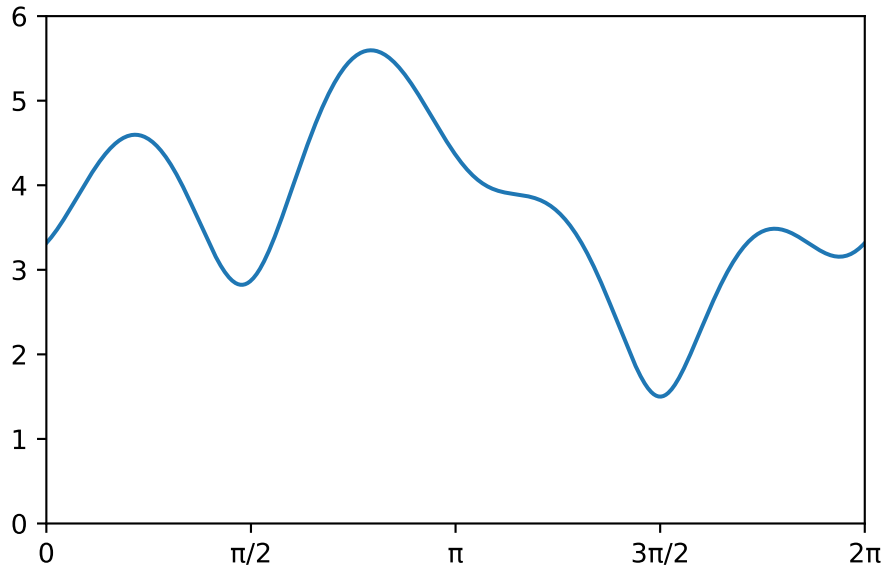

Figure 5: Graph der Abstandsfunktion $d(t)$.

Wir möchten das Minimum der Funktion $d(t)$ mit Hilfe der Gradient Descent Methode finden. Dazu müssen wir aber d ableiten können.
