# AutoDiff

**Eine Einführung in algorithmisches Differenzieren**

Michael Brand

19.10.22

# Inhaltsverzeichnis

**Vorwort** **3**

**Einleitung** **4**
Danksagung . . . . . 4

**1 Ableitungen und ihre Anwendungen** **5**
1.1 Ableitungen von Funktionen . . . . . 5
1.2 Programme als Funktionen . . . . . 6
1.3 Unser Ziel: Programme ableiten . . . . . 12
1.3.1 Das Newtonverfahren zur Berechnung von Nullstellen . . . . . 12
1.3.2 Gradient Descent zum Auffinden lokaler Minima . . . . . 20

**2 AD ist nicht ...** **27**
2.1 AD ist nicht numerisches Ableiten . . . . . 27
2.1.1 Auslöschung . . . . . 30
2.2 AD ist nicht symbolisches Ableiten . . . . . 30

**3 Standard Algorithmische Differentiation für eindimensionale Funktionen** **32**
3.1 Manuelle Implementation der SAD . . . . . 33
3.2 Implementation der SAD mit Operator Overloading . . . . . 44
3.2.1 Die Klasse `FloatSad` . . . . . 46
3.2.2 Vorzeichen . . . . . 47
3.2.3 Die Operatoren `+` und `-` . . . . . 48
3.2.4 Die Operatoren `*` und `/` . . . . . 51
3.2.5 Der Operator `**` . . . . . 53
3.2.6 Vergleichsopertoren . . . . . 55
3.3 Die Klasse `FloatSad` im Einsatz . . . . . 56
3.4 Das Modul `mathsad` . . . . . 58
3.4.1 Die Funktion `sqrt` . . . . . 59
3.4.2 Die Funktionen `exp` und `log` . . . . . 60
3.4.3 Die trigonometrischen Funktionen und ihre Umkehrfunktionen . . . . . 61
3.4.4 Die hyperbolischen Funktionen und ihre Umkehrfunktionen . . . . . 62
3.4.5 Die Betragsfunktion . . . . . 63
3.5 Das Modul `mathsad` im Einsatz . . . . . 64

**References** **69**

# Vorwort

# Einleitung

## Danksagung

# 1 Ableitungen und ihre Anwendungen

## 1.1 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.

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

Tabelle 1.1: Ableitungen der Grundfunktionen

Neue Funktionen erhält man, indem man die Grundfunktionen aus Tabelle 1.1 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 Tabelle 1.2. Mit diesen Regeln sind wir dann schon in der Lage, alle differenzierbaren Funktionen abzuleiten.

| Regel | Formel |
|---|---|
| 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* | $\frac{d}{dx}(a \cdot f(x)) = a \cdot f'(x)$ |

| Regel | Formel |
|---|---|
| Quotientenregel | $\frac{d}{dx}\frac{f(x)}{g(x)} = \frac{f'(x)\cdot g(x) - f(x)\cdot g'(x)}{g(x)^2}$ |
| Kettenregel | $\frac{d}{dx} f(g(x)) = f'(g(x)) \cdot g'(x)$ |

Tabelle 1.2: 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 Slater (2022). Weitergehende Informationen findet man z.B. in Arens u. a. (2022) oder in jedem Lehrbuch zur Analysis 2.

## 1.2 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.

**Beispiel 1.1** (Eine Funktion als Programm)**.** Betrachte das folgende 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 Hromkovic u. a. (2021) 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
```

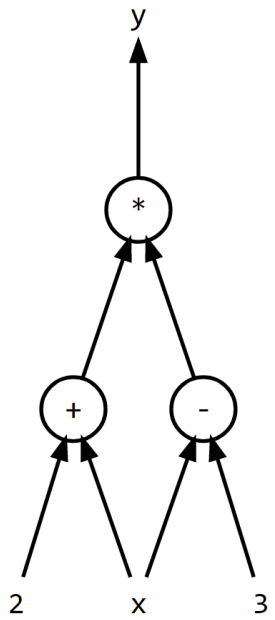


Abbildung 1.1: Ausdrucksbaum zum Ausdruck `y = (2 + x) * (x - 3)`.

```
    v1 = 2 + v0
    v2 = v0 - 3
    y = v1 * v2
    return y
```

---

> **!** 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`.

**Übungsaufgabe 1.1** (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

$$
\begin{aligned}
v_1 &= x^2 \\
v_2 &= x^2 + 2 \\
v_3 &= -\frac{x^2}{2} \\
v_4 &= \cos(x^2 + 2) \\
v_5 &= e^{-\frac{x^2}{2}} \\
v_6 &= \cos(x^2 + 2) \cdot e^{-\frac{x^2}{2}} \\
y &= f(x) = \cos(x^2 + 2) \cdot e^{-\frac{x^2}{2}} + \frac{1}{x}
\end{aligned}
$$

**Übungsaufgabe 1.2** (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

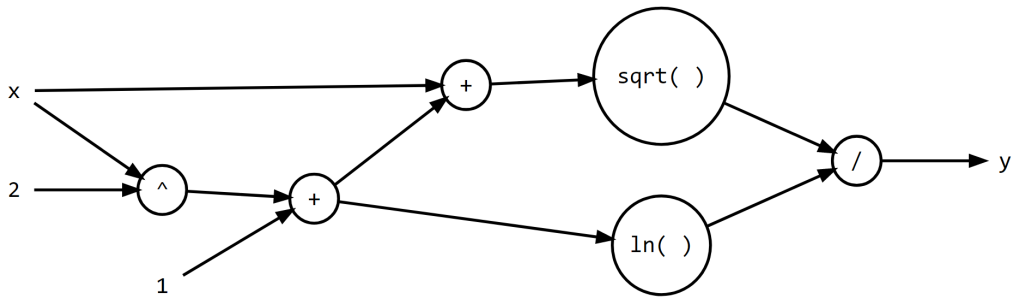


Abbildung 1.2: Computational Graph zum Ausdruck `y = ln(x^2 + 1) / sqrt(x^2 + 1 + x)`.

```python
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.

**Übungsaufgabe 1.3** (Ein Programm mit einer Schleife)**.** Betrachte das folgende Programm:

```python
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
```

$$\begin{aligned} f(x) &= \ell(\ell(x)) \\ &= (x^2+1)^2 + 1 = x^4 + 2x^2 + 2 \end{aligned}$$

**range(3)**

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

$$\begin{aligned} f(x) &= \ell(\ell(\ell(x))) \\ &= ((x^2+1)^2+1)^2 + 1 = x^8 + 4x^6 + 8x^4 + 8x^2 + 5 \end{aligned}$$

**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
```

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

## 1.3 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. Kapitel 2.2), 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. Kapitel 2.1), sondern den Wert von $f'(x_0)$ bis auf Maschinengenauigkeit exakt berechnen. In Kapitel 3 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 Griewank und Walther (2008))

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.

### 1.3.1 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:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1.1}$$

**Übungsaufgabe 1.4** (Das Newtonverfahren programmieren)**.** Schreibe ein Programm, das mit Hilfe des Newtonverfahrens (Gleichung 1.1) 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**

```
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**

```
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 Kapitel 2.1 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.

**Beispiel 1.2** (Billard auf einem runden Tisch)**.** Wir betrachten ein Beispiel aus Gander (2015). 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 $\vec{t} = \begin{pmatrix} -\sin(x) \\ \cos(x) \end{pmatrix}$.

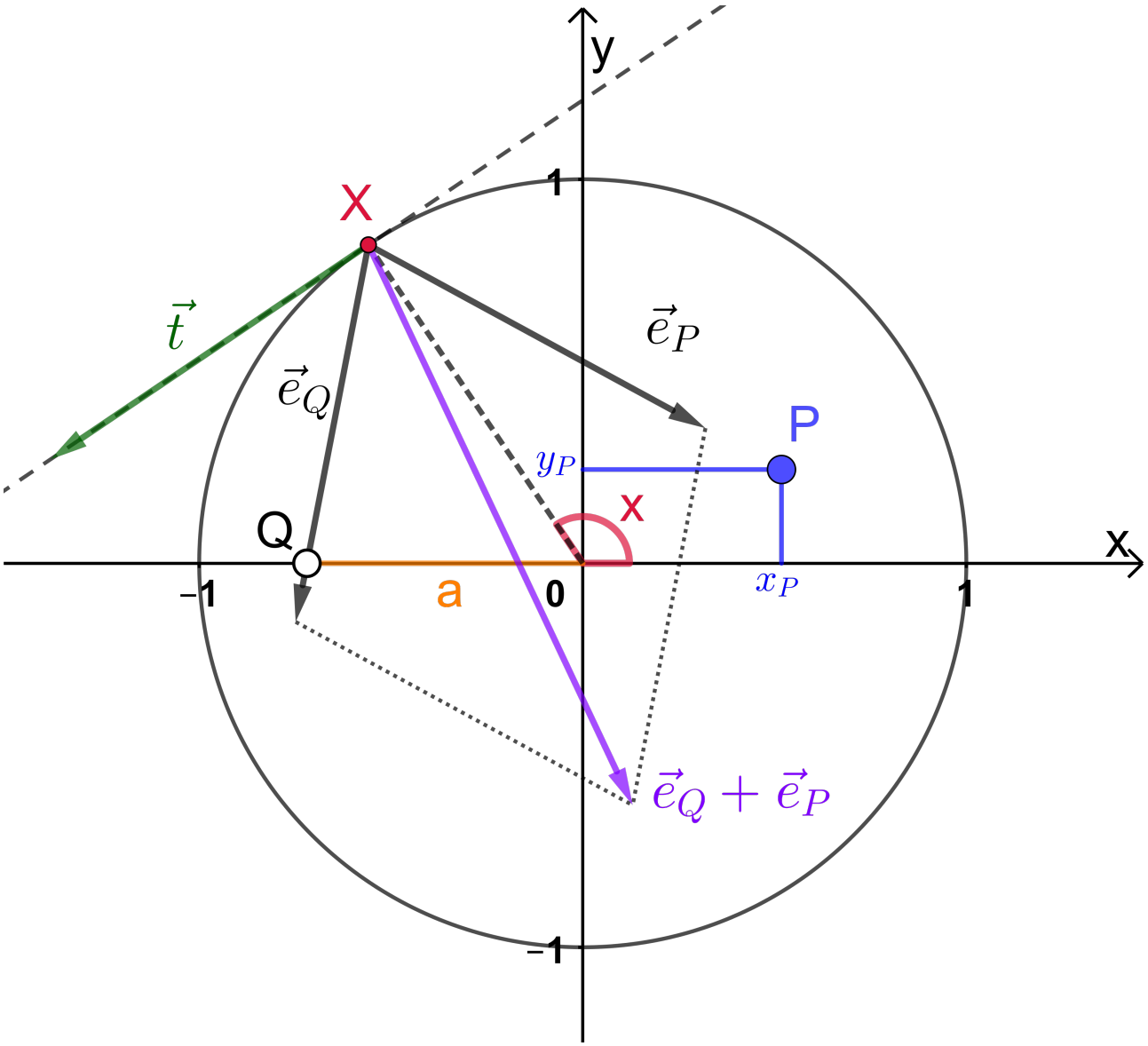


Abbildung 1.3: Billard auf einem runden Tisch

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
    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()
```

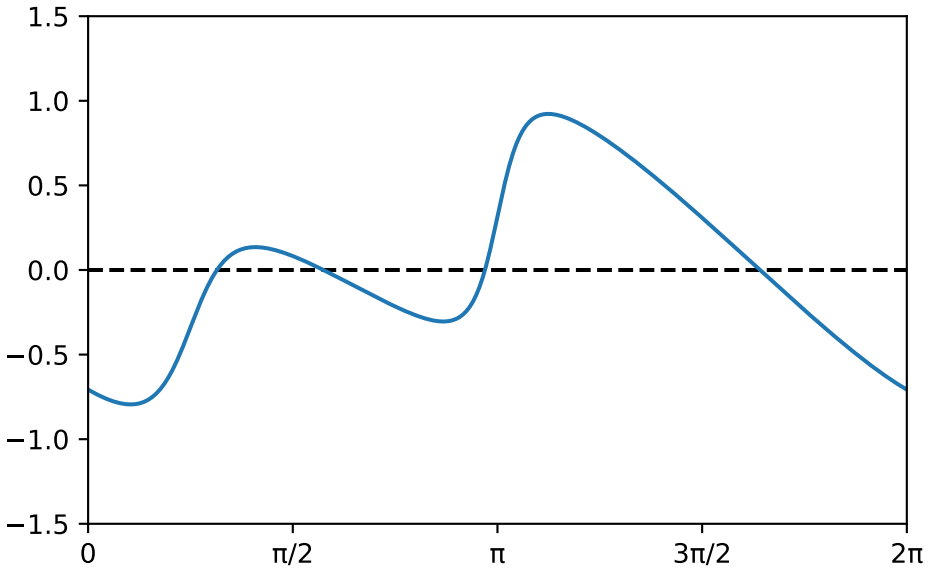


Abbildung 1.4: Graph des Skalarprodukts als Funktion des Polarwinkels $x$ des Punktes $X = (cos(x)|sin(x))$. Die Nullstellen entsprechen den Winkeln, bei denen die weisse Kugel die blaue Kugel trifft, nachdem sie genau einmal an die Bande gespielt wurde.

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

---

### 1.3.2 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

$$x_{n+1} = x_n - \lambda \cdot f'(x_n) \tag{1.2}$$

**Übungsaufgabe 1.5** (Eigenschaften der Gradient Descent Methode)**.** Experimentiere mit verschiedenen Funktionen und verschiedenen Schrittweiten $\lambda$. 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 $\lambda$ zu klein, dann konvergiert das Verfahren nur sehr langsam. Ist $\lambda$ 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 $\lambda$ nicht zu gross ist.

**Übungsaufgabe 1.6** (Gradient Descent programmieren)**.** Schreibe ein Programm, das mit Hilfe des Gradient Descent Verfahrens (Gleichung 1.2) 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**

```
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**

```
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 $\lambda$ 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 $\lambda$ 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 Übungsaufgabe 1.6 verdeutlicht nochmals das Problem, welches wir bereits in Übungsaufgabe 1.4 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.

**Beispiel 1.3** (Minimaler Abstand)**.** Die Punkte $P$ und $Q$ bewegen sich auf Ellipsen im Raum. Die Position des Punktes $P$ zur Zeit $t$ ist gegeben durch

$$\begin{aligned} x_P(t) &= 2\cos(t) - 1 \\ y_P(t) &= 1.5\sin(t) \\ z_P(t) &= 0 \end{aligned}$$

und die Position von $Q$ zum Zeitpunkt $t$ lässt sich durch

$$\begin{aligned} x_Q(t) &= -3\sin(2t) \\ y_Q(t) &= 2\cos(2t) + 1 \\ z_Q(t) &= 2\sin(2t) + 1 \end{aligned}$$

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()
```

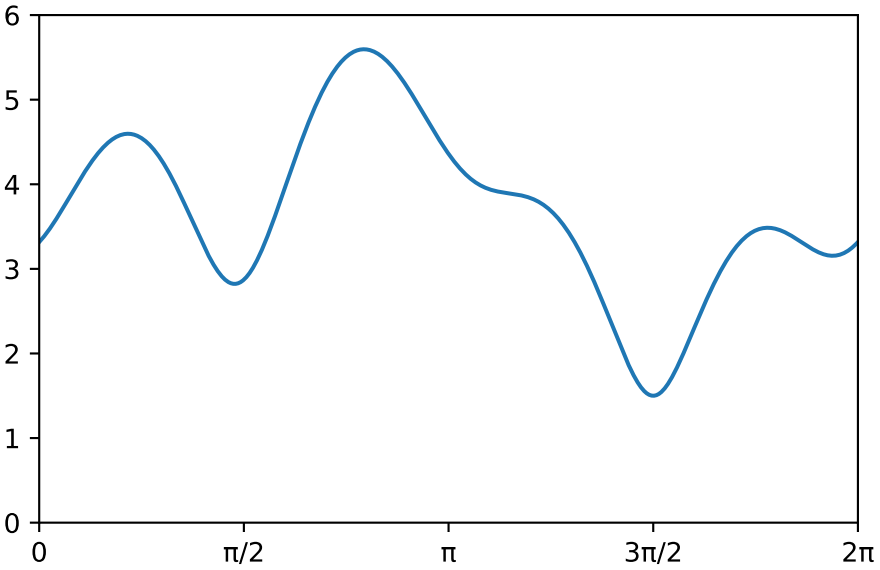


Abbildung 1.5: Graph der Abstandsfunktion $d(t)$.

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

---

# 2 AD ist nicht ...

Bevor wir uns mit den konkreten Implementationen von algorithmischer Differentiation beschäftigen, wollen wir herausstellen, was AD *nicht* ist.

## 2.1 AD ist nicht numerisches Ableiten

Eine Funktion $y = f(x)$ ist bekanntlich differenzierbar an der Stelle $x_0 \in \mathbb{D}$, wenn der Grenzwert

$$\lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

existiert. In dem Fall ist $f'(x_0)$ einfach der Wert dieses Grenzwerts.

Ein erster Ansatz zur numerischen Berechnung könnte also sein, den Differenzenquotienten für kleine $h$ auszuwerten[1].

**Beispiel 2.1** (Numerische Ableitung)**.** Leite die Funktion $f(x) = x^2$ an der Stelle $x_0 = 2$ ab.

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

def fdot(f, x0, h):
    df = (f(x0 + h) - f(x0)) / h
    return df

x0 = 0.2
H = [0.1, 0.01, 0.001, 0.0001]
for h in H:
    ydot = fdot(f, x0, h)
    print("h = " + str(h) + " \t=> f'(x0) = " + str(ydot))
```

[1] Dieser Ansatz kann verbessert werden indem man z.B. $f'(x_0) \approx \frac{f(x_0+h)-f(x_0-h)}{2h}$ verwendet. Die im Beispiel beschriebenen Probleme bleiben aber auch dann bestehen.

```
h = 0.1      => f'(x0) = 0.5000000000000001
h = 0.01     => f'(x0) = 0.4099999999999999
h = 0.001    => f'(x0) = 0.4009999999999986
h = 0.0001   => f'(x0) = 0.40009999999993107
```

Es scheint zunächst, als ob die Werte für kleiner werdende $h$ zum korrekten Wert $f'(0.2) = 0.4$ konvergieren. Wenn wir aber an sehr genauen Werten interessiert sind und entsprechend $h$ sehr klein wählen, beobachten wir folgendes:

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

def fdot(f, x0, h):
    df = (f(x0 + h) - f(x0)) / h
    return df

x0 = 0.2
H = [10 ** -8, 10 ** -9, 10 ** -10, 10 ** -11]
for h in H:
    ydot = fdot(f, x0, h)
    print("h = " + str(h) + "\t=> f'(x0) = " + str(ydot))
```

```
h = 1e-08    => f'(x0) = 0.4000000095039091
h = 1e-09    => f'(x0) = 0.3999999984016789
h = 1e-10    => f'(x0) = 0.4000000330961484
h = 1e-11    => f'(x0) = 0.3999994779846361
```

Mit kleiner werdendem $h$ scheint sich der Näherungswert für die Ableitung zu verschlechtern. Das Phänomen wird noch deutlicher, wenn wir den Fehler $E(h) = |\frac{f(x_0+h)-f(x_0)}{h} - f'(x_0)|$ als Funktion von $h$ plotten. Beachte die doppelt logarithmische Skala.

```
import matplotlib.pyplot as plt
import math

def f(x):
    y = x ** 2
    return y

def fdot(f, x0, h):
    df = (f(x0 + h) - f(x0)) / h
```

```
    return df

x0 = 0.2
H = [10**(k/100) for k in range(-1800, -300)]
E = [math.fabs(fdot(f, x0, h) - 2*x0) for h in H]

# Plot
fig = plt.figure()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
ax.set(xlim=(10**-18, 10**-3), ylim=(10**-12, 10**0))
ax.set_xscale('log')
ax.set_xlabel('h')
ax.set_yscale('log')
ax.set_ylabel('Fehler E(h)')
plt.plot(H,E)
plt.show()
```

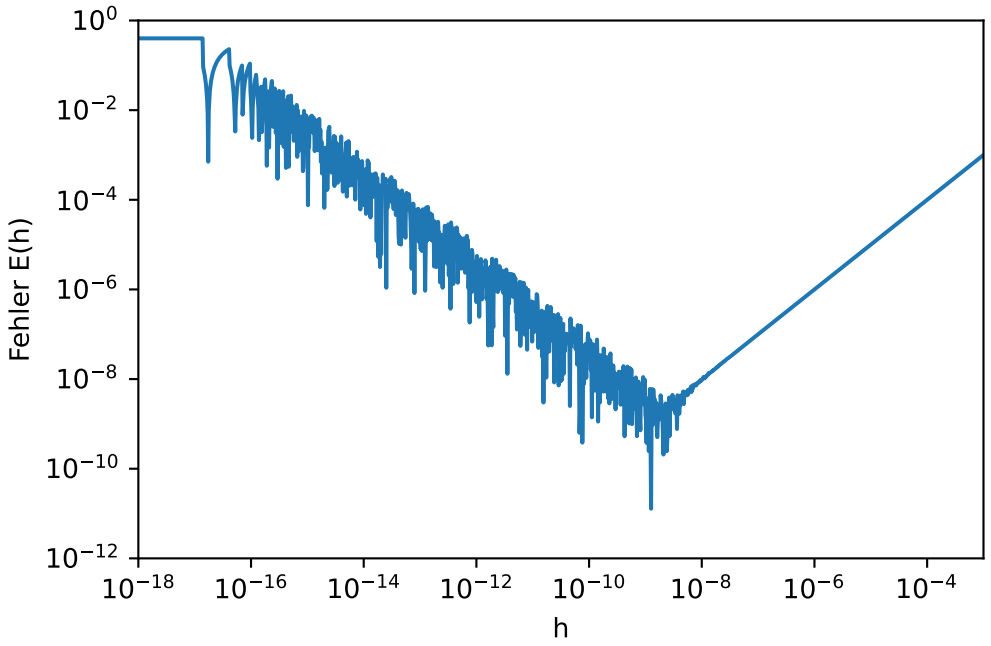


Abbildung 2.1: Grösse des Fehlers $E(h)$ als Funktion der Schrittweite $h$. Ist $h$ zu gross, dann ist der Näherungswert für $f'(x_0)$ ungenau. Bei kleiner werdendem $h$ nimmt der Fehler zunächst ab, aber ab einem gewissen Wert dominiert die Auslöschung und der Fehler nimmt wieder zu.

### 2.1.1 Auslöschung

Im vorherigen Beispiel haben wir das Phänomen der Auslöschung beobachtet. Zunächst ist dir sicher aufgefallen, dass der Näherungswert für $f'(x_0)$ mit $h = 0.01$ nicht

$$\frac{f(x_0 + h) - f(x_0)}{h} = \frac{0.21^2 - 0.2^2}{0.01} = 0.41$$

ergab, sondern $f'(x_0) \approx 0.40999....$ Das liegt daran, dass Dezimalzahlen nicht exakt als Binärzahl dargestellt werden können. Da nun die Werte von $f(x_0 + h)$ und $f(x_0)$ für kleine $h$ fast gleich sind, besteht ihre Differenz $f(x_0+h)-f(x_0)$ nur noch aus diesen Rundungsfehlern. Diese (sinnlose) Differenz ist zwar sehr klein, wird aber im nächsten Schritt mit der sehr grossen Zahl $\frac{1}{h}$ multipliziert, wodurch die Rundungsfehler die gleiche Grössenordnung annehmen, wie die ursprünglichen Funktionswerte. Mehr über Rundungsfehler und Auslöschung kann in Weitz (2021) ab S. 117 nachgelesen werden.

## 2.2 AD ist nicht symbolisches Ableiten

Computer Algebra Systeme (CAS) sind Programme zur Bearbeitung algebraischer Ausdrücke. Mit solchen Programmen lassen sich auch Ableitungen symbolisch bestimmen. Wie das funktioniert, wird in Slater (2022) kurz angedeutet. Bekannte Beispiele für CAS sind etwa Wolfram Alpha, Maxima oder Sage. Letzteres kann man hier auch online ausprobieren. Gib z.B. den folgenden Code ein, welcher die Ableitung der Funktion aus Übungsaufgabe 1.3 bestimmt:

```
l(x) = x^2 + 1
f(x) = l(l(l(x)))
fdot = diff(f,x)
expand(fdot)
```

> **Tipp**
>
> Auf der Website kannst du rechts unter `Language` auch `Maxima` auswählen und Maxima-Code ausführen. Maxima ist in Sage integriert.

Für Python gibt es die Bibliothek `sympy`, die ein CAS für Python zur Verfügung stellt. Damit können wir die Funktion aus Übungsaufgabe 1.3 direkt in Python ableiten:

**Beispiel 2.2** (Symbolische Ableitung)**.** Leite die Funktion $f(x) = l(l(l(x)))$, wobei $l(x) = x^2 + 1$ ist, an der Stelle $x_0 = 1$ ab.

```python
from sympy  import symbols, diff

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

x = symbols('x')
print("f(x) =", f(x))

df = diff(f(x),x)
print("f'(x) =", df)

x0 = 1
print("f'(" + str(x0) + ") =", df.evalf(subs={x:x0}))
```

```
f(x) = ((x**2 + 1)**2 + 1)**2 + 1
f'(x) = 8*x*(x**2 + 1)*((x**2 + 1)**2 + 1)
f'(1) = 80.0000000000000
```

---

Damit erhält man die (bis auf Maschinengenauigkeit) exakten Werte der Ableitungen. Der Grund, warum wir nicht auf symbolische Ausdrücke für Ableitungen zurückgreifen wollen, liegt darin, dass diese Methode bei komplizierten Funktionsausdrücken sehr ineffizient ist, insbesondere dann, wenn wir auch Ableitungen von Funktionen $f : \mathbb{R}^n \to \mathbb{R}^m$ berechnen wollen.

# 3 Standard Algorithmische Differentiation für eindimensionale Funktionen

In Kapitel 2 haben wir zwei Methoden für die Berechnung von Ableitungen kennen gelernt, die beide ihre Schwächen haben. Während die numerische Ableitung mit geringem Aufwand berechnet werden kann, sind ihre Näherungswerte für viele Anwendungen zu ungenau. Symbolische Ableitungen andererseits liefern zwar exakte Werte von Ableitungen, sind aber mit grossem Rechenaufwand verbunden. Die hier vorgestellte Algorithmische Differentiation (AD) vereinigt die Vorteile der beiden Methoden. Sie liefert uns (bis auf Maschinengenauigkeit) exakte Werte von Ableitungen mit nur einem geringen zusätzlichen Rechenaufwand:

> "AD as a technical term refers to a specific family of techniques that compute derivatives trhough accumulation of values during code execution to generate numerical derivative evaluations rather than derivative expressions. This allows accurate evaluation of derivatives at machine precision with only a small constant factor of overhead and ideal asymptotic efficiency." (Baydin u. a. (2018), S. 2)

In diesem Kapitel lernen wir die *Standard Algorithmische Differentiation* (SAD, auch Vorwärts-AD genannt) kennen, welche die einfachste Variante der erwähnten "family of techniques" ist. Wir beschränken uns zunächst wieder auf Funktionen $f : \mathbb{R} \to \mathbb{R}$ und werden dies später auf Funktionen $f : \mathbb{R}^n \to \mathbb{R}^m$ erweitern. Neben der Standard-AD gibt es noch die *Adjungierte Algorithmische Differentiation* (AAD, auch Rückwärts-AD genannt). Die Vorteile dieser Methode offenbaren sich jedoch erst für Funktionen in höherdimensionalen Räumen.

Gemäss unserer Konvention in Kapitel 1.2 berechnen wir eine mathematische Funktion, indem wir sie in ihre Bestandteile zerlegen, und die Zwischenergebnisse Variablen `v` zuweisen. Wie im obigen Zitat erwähnt, besteht die Grundidee der AD darin, eine Reihe von Hilfsvariablen `vdot` einzuführen, welche jeweils die Werte der Ableitungen enthalten. In diesem Kapitel machen wir dies explizit, indem wir jede Programmzeile, die ein `v` berechnet, um die Berechnung des zugehörigen `vdot` erweitern. Dies scheint zunächst umständlich zu sein, aber im nächsten Abschnitt werden wir eine Klasse schreiben, die diese Schritte für uns automatisiert. Wie Marc Henrard in seinem Buch schreibt:

> "There are as many shades of AD as there are AD users. [This chapter] provides to the user the black and the white; it is up to him to get the correct shade of grey that fits his taste and his requirements." (Henrard (2017), S. 18)

## 3.1 Manuelle Implementation der SAD

Beginnen wir mit einem Beispiel:

Wir möchten den Funktionswert und die Ableitung der Funktion $y = f(x) = \sin(x^2)$ an der Stelle $x_0 = \frac{\pi}{2}$ bestimmen. Das folgende Programm berechnet den Funktionswert.

```
import math

def f(x):
    v0 = x
    v1 = v0**2
    v2 = math.sin(v1)
    y = v2
    return y

x0 = math.pi / 2
print(f(x0))
```

```
0.6242659526396992
```

$f$ ist eine zusammengesetzte Funktion, die wir mit den Funktionen

$$\begin{aligned} v_0(x) &= x \\ v_1(v_0) &= v_0^2 \\ v_2(v_1) &= \sin(v_1) \end{aligned}$$

schreiben können als $y = f(x) = v_2(v_1(v_0(x)))$. Die Ableitung berechnet sich dann mit der Kettenregel zu

$$f'(x) = \frac{dv_2}{dv_1} \cdot \frac{dv_1}{dv_0} \cdot \frac{dv_0}{dx} = \cos(v_1) \cdot 2v_0 \cdot 1 = \cos(x^2) \cdot 2x \cdot 1$$

Wir können also die Ableitung von `f(x)` berechnen, indem wir jede Zeile des Programms gemäss den bekannten Regeln ableiten:

```
v0dot = 1
v1dot = 2 * v0 * v0dot
v2dot = math.cos(v1) * v1dot
```

Man beachte, dass durch die Konvention, dass immer `v0 = x` gesetzt wird, auch immer `v0dot = 1` ist. Nun können wir unsere Funktion so ergänzen, dass nicht nur der Funktionswert, sondern auch die Ableitung an der Stelle $x_0$ berechnet wird:

```
import math

def f(x):
    v0dot = 1
    v0 = x
    v1dot = 2 * v0 * v0dot
    v1 = v0**2
    v2dot = math.cos(v1) * v1dot
    v2 = math.sin(v1)
    ydot = v2dot
    y = v2
    return [y, ydot]

x0 = math.pi / 2
print(f(x0))
```

```
[0.6242659526396992, -2.4542495411512917]
```

Die Korrektheit des Programms können wir mit GeoGebra überprüfen, welches Ableitungen symbolisch berechnet.

Beachte, dass wir konsequent die Kettenregel verwendet haben. So wird aus `v1 = v0**2` etwa `v1dot = 2 * v0 * v0dot` oder aus `v2 = sin(v1)` wird `v2dot = cos(v1) * v1dot`.

**Übungsaufgabe 3.1** (Programm ableiten)**.** Ändere das vorherige Programm so ab, dass der Funktionswert und die Ableitung der Funktion $y = f(x) = \ln(\sin(x^2))$ an der Stelle $x_0 = \frac{\pi}{2}$ berechnet wird. Überprüfe deine Lösung mit GeoGebra.

Lösung

Es müssen lediglich zwei weitere Zeilen eingefügt werden und zwar für die Berechnung von `v3` und `v3dot`. Vergiss nicht, die richtigen Werte zurückzugeben.

```
import math

def f(x):
    v0dot = 1
    v0 = x
    v1dot = 2 * v0 * v0dot
    v1 = v0**2
    v2dot = math.cos(v1) * v1dot
    v2 = math.sin(v1)
    v3dot = 1 / v2 * v2dot
    v3 = math.log(v2)
    ydot = v3dot
    y = v3
    return [y, ydot]

x0 = math.pi / 2
print(f(x0))
```

```
[-0.4711787952593891, -3.9314166194288416]
```

**!** Konvention

Ein Programm, welches gemäss der Konvention in Kapitel 1.2 geschrieben ist, wird folgendermassen abgeleitet:

1. Für jede Variable `v` wird eine neue Variable `vdot` für den Wert der Ableitung definiert, angefangen bei `v0dot = 1`.
2. Jede Programmzeile wird gemäss den bekannten Regeln aus Tabelle 1.1 und Tabelle 1.2 abgeleitet. Dabei wird insbesondere in *jeder* Zeile die Kettenregel verwendet.
3. Die abgeleitete Anweisung wird jeweils *vor* (!) die zu ableitende Anweisung eingeschoben.

**Übungsaufgabe 3.2** (Produktregel)**.** Das folgende Programm berechnet die Funktion $y = f(x) = (2 + x)(x - 3)$:

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

```
    y = v1 * v2
    return y
```

Leite dieses Programm ab. Dein Programm soll die Gleichung der Tangente $t(x) = f(x_0) + f'(x_0) \cdot (x - x_0)$ an der Stelle $x_0 = 2$ ausgeben.

Lösung

```
def f(x):
    v0dot = 1
    v0 = x
    v1dot = v0dot
    v1 = 2 + v0
    v2dot = v0dot
    v2 = v0 - 3
    ydot = v1dot * v2 + v1 * v2dot # Produktregel
    y = v1 * v2
    return [y, ydot]

x0 = 2
[y0, y0dot] = f(x0)
print("t(x) =", y0, "+", y0dot, "* ( x -", x0, ")")
```

```
t(x) = -4 + 3 * ( x - 2 )
```

**Übungsaufgabe 3.3** (Programm ableiten)**.** Leite die Funktion aus Übungsaufgabe 1.1 ab. Gib den Funktionswert und den Wert der Ableitung an der Stelle $x_0 = -2$ aus.

Lösung

```
import math

def f(x):
    v0dot = 1
    v0 = x
    v1dot = 2 * v0 * v0dot
    v1 = v0 ** 2
    v2dot = v1dot
    v2 = v1 + 2
    v3dot = -1/2 * v1dot
    v3 = -v1 / 2
    v4dot = -math.sin(v2) * v2dot
    v4 = math.cos(v2)
    v5dot = math.exp(v3) * v3dot
    v5 = math.exp(v3)
    v6dot = v4dot * v5 + v4 * v5dot
    v6 = v4 * v5
    ydot = v6dot - 1 / v0**2 * v0dot
    y = v6 + 1 / v0
    return [y, ydot]

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

```
[-0.3700550823007931, -0.14136926695938976]
```

**Übungsaufgabe 3.4** (Programm ableiten)**.** Leite die Funktion aus Übungsaufgabe 1.2 ab.

Lösung

```
import math

def f(x):
    v0dot = 1
    v0 = x
    v1dot = 2 * v0 * v0dot
    v1 = v0 ** 2
    v2dot = v1dot
    v2 = v1 + 1
    v3dot = v2dot + v0dot
    v3 = v2 + v0
    v4dot = 1 / v2 * v2dot
    v4 = math.log(v2)
    v5dot = 1 / (2 * math.sqrt(v3)) * v3dot
    v5 = math.sqrt(v3)
    ydot = (v4dot * v5 - v4 * v5dot) / v5**2
    y = v4 / v5
    return [y, ydot]
```

Bei all diesen Beispielen könnten wir auch die Reihenfolge der Anweisungen für `vdot` und `v` vertauschen, d.h. zuerst die Variable `v` berechnen und erst danach das zugehörige `vdot`. Die folgende Übung zeigt aber, warum der 3. Punkt unserer Konvention wichtig ist.

**Übungsaufgabe 3.5** (Ein Programm mit einer Schleife)**.** Betrachte die Funktion aus Übungsaufgabe 1.3, welche aus $\ell(x) = x^2 + 1$ die Funktion $y = f(x) = \ell(\ell(\ell(x)))$ berechnet. Aus Beispiel 2.2 wissen wir, dass $f'(1) = 80$ ist. Vergleiche nun die beiden Varianten für die Ableitung des Programms:

**`vdot` vor `v`**

```
def f(x):
    v0dot = 1
    v0 = x
    for i in range(3):
        v0dot = 2 * v0 * v0dot
        v0 = v0 ** 2 + 1
    ydot = v0dot
```

```
    y = v0
    return [y, ydot]

print(f(1))
```

```
[26, 80]
```

**`v` vor `vdot`**

```
def f(x):
    v0 = x
    v0dot = 1
    for i in range(3):
        v0 = v0 ** 2 + 1
        v0dot = 2 * v0 * v0dot
    y = v0
    ydot = v0dot
    return [y, ydot]

print(f(1))
```

```
[26, 2080]
```

Warum wird bei der 2. Variante der Wert der Ableitung falsch berechnet?

> Lösung
>
> Das Problem tritt in der Schleife auf. In der 2. Variante überschreiben wir den Wert von `v0` bereits mit dem neuen Wert der Iteration. Bei der Berechnung von `v0dot` hätten wir aber noch den alten Wert gebraucht. Die Reihenfolge ist also nur in der 1. Version korrekt. Würden wir die Schleife eliminieren und dafür wie in der Lösung zu Übungsaufgabe 1.3 für jeden Schleifendurchgang fortlaufend numerierte Variablen für die `v` und `vdot` verwenden, dann wäre die Reihenfolge wieder egal.

In der nächsten Übungsaufgabe verwenden wir die Technik der AD, um das Billardproblem aus Kapitel 1.3.1 mit dem Newtonverfahren zu lösen. Da uns die Funktion `f(x)` nun nicht mehr nur der Funktionswert, sondern auch die Ableitung berechnet, können wir das Newtonverfahren ohne die Probleme aus Übungsaufgabe 1.4 implementieren.

**Übungsaufgabe 3.6** (Das Billard-Problem)**.** Leite das Programm aus Beispiel 1.2 ab. Schreibe danach eine Funktion `newton(f, x0)`, welche ausnutzt, dass der Funktionsaufruf `f(x0)` auch den exakten Wert der Ableitung zurückgibt. Stelle alle gefundenen Lösungen grafisch dar.

Lösung

```
import math
import matplotlib.pyplot as plt

def f(x):
    # Parameter werden im global space gefunden
    # Berechnung des Skalarprodukts und dessen Ableitung
    v0dot = 1
    v0 = x
    v1dot = -math.sin(v0) * v0dot  # Ableitung von ...
    v1 = math.cos(v0)  # x-Koordinate von X
    v2dot = math.cos(v0) * v0dot   # Ableitung von ...
    v2 = math.sin(v0)  # y-Koordinate von X
    v3dot = - v1dot    # Ableitung von ...
    v3 = px - v1       # x-Komponente des Vektors XP
    v4dot = - v2dot    # Ableitung von ...
    v4 = py - v2       # y-Komponente des Vektors XP
    v5dot = 1 / (2*math.sqrt(v3**2 + v4**2)) \
        * (2*v3*v3dot + 2*v4*v4dot)  # Ableitung von ...
    v5 = math.sqrt(v3**2 + v4**2)  # Länge des Vektors XP
    v6dot = (v3dot * v5 - v3 * v5dot) / v5**2  # Ableitung von ...
    v6 = v3 / v5       # x-Komponente des Einheitsvektors eP
    v7dot = (v4dot * v5 - v4 * v5dot) / v5**2  # Ableitung von ...
    v7 = v4 / v5       # y-Komponente des Einheitsvektors eP
    v8dot = -v1dot     # Ableitung von ...
    v8 = a - v1        # x-Komponente des Vektors XQ
    v9dot = -v2dot     # Ableitung von ...
    v9 = -v2           # y-Komponente des Vektors XQ
    v10dot = 1 / (2*math.sqrt(v8**2 + v9**2)) \
        * (2*v8*v8dot + 2*v9*v9dot)  # Ableitung von ...
    v10 = math.sqrt(v8**2 + v9**2)  # Länge des Vektors XQ
    v11dot = (v8dot * v10 - v8 * v10dot) / v10**2  # Ableitung von ...
    v11 = v8 / v10     # x-Komponente des Vektors eQ
    v12dot = (v9dot * v10 - v9 * v10dot) / v10**2  # Ableitung von ...
    v12 = v9 / v10     # y-Komponente des Vektors eQ
    ydot = (v6dot + v11dot) * v2 + (v6 + v11) * v2dot \
        - ( (v7dot + v12dot) * v1 + (v7 + v12) * v1dot )  # Ableitung von ...
    y = (v6 + v11) * v2 - (v7 + v12) * v1
    return [y, ydot]

def newton(f, x0):
    tol = 1e-8
    # Erster Schritt berechnen
    [y0, y0dot] = f(x0)
    x1 = x0 - y0 / y0dot
    while math.fabs(x1 - x0) > tol:
        x0 = x1
        [y0, y0dot] = f(x0)
        x1 = x0 - y0 / y0dot
    return x1


if __name__ == "__main__":
```

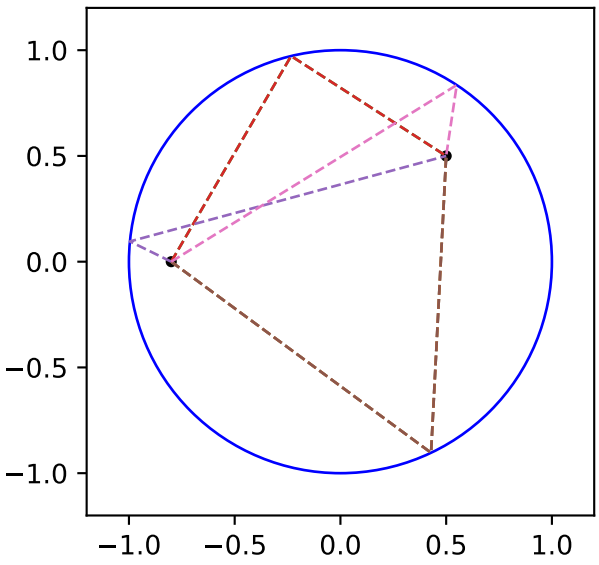


Abbildung 3.1: Lösung des Billardproblems.

**Übungsaufgabe 3.7** (Minimaler Abstand)**.** Leite das Programm aus Beispiel 1.3 ab. Schreibe danach eine Funktion `gradient_descent(f, x0, lam)`, welche ausnutzt, dass der Funktionsaufruf `f(x0)` auch den exakten Wert der Ableitung zurückgibt. Stelle die gefundene Lösung grafisch dar.

Lösung

```
import math
import matplotlib.pyplot as plt

def d(t):
    v0dot = 1
    v0 = t
    v1dot = -2 * math.sin(v0) * v0dot   # Ableitung von...
    v1 = 2 * math.cos(v0) - 1    # x-Koordinate von P
    v2dot = 1.5 * math.cos(v0) * v0dot  # Ableitung von...
    v2 = 1.5 * math.sin(v0)      # y-Koordinate von P
    v3dot = 0                    # Ableitung von...
    v3 = 0                       # z-Koordinate von P
    v4dot = -6 * math.cos(2*v0) * v0dot # Ableitung von...
    v4 = -3 * math.sin(2*v0)     # x-Koordinate von Q
    v5dot = -4 * math.sin(2*v0) * v0dot # Ableitung von...
    v5 = 2 * math.cos(2*v0) + 1  # y-Koordinate von Q
    v6dot = 4 * math.cos(2*v0) * v0dot  # Ableitung von...
    v6 = 2 * math.sin(2*v0) + 1  # z-Koordinate von Q
    ydot = (2*(v4-v1)*(v4dot-v1dot) + 2*(v5-v2)*(v5dot-v2dot) + 2*(v6-v3)*(v6dot-v3dot)) \
         / (2 * math.sqrt((v4-v1)**2 + (v5-v2)**2 + (v6-v3)**2))
    y = math.sqrt((v4-v1)**2 + (v5-v2)**2 + (v6-v3)**2)
    return [y, ydot]

def gradient_descent(f, x0, lam):
    tol = 1e-9
    # Erster Schritt berechnen
    [y0, y0dot] = f(x0)
    x1 = x0 - lam * y0dot
    while math.fabs(x1-x0) > tol:
        x0 = x1
        [y0, y0dot] = f(x0)
        x1 = x0 - lam * y0dot
    return x1

if __name__ == "__main__":
    t0 = 3
    tmin = gradient_descent(d, t0, 0.01)
    [dmin, dmindot] = d(tmin)
    print("Minimum bei (", tmin, dmin, ")")

    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)[0] for t in T]  # nur Funktionswert plotten
    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.plot(tmin,dmin,color='r', marker='o')
    plt.show()
```

```
Minimum bei ( 4.712388977478413 1.5 )
```

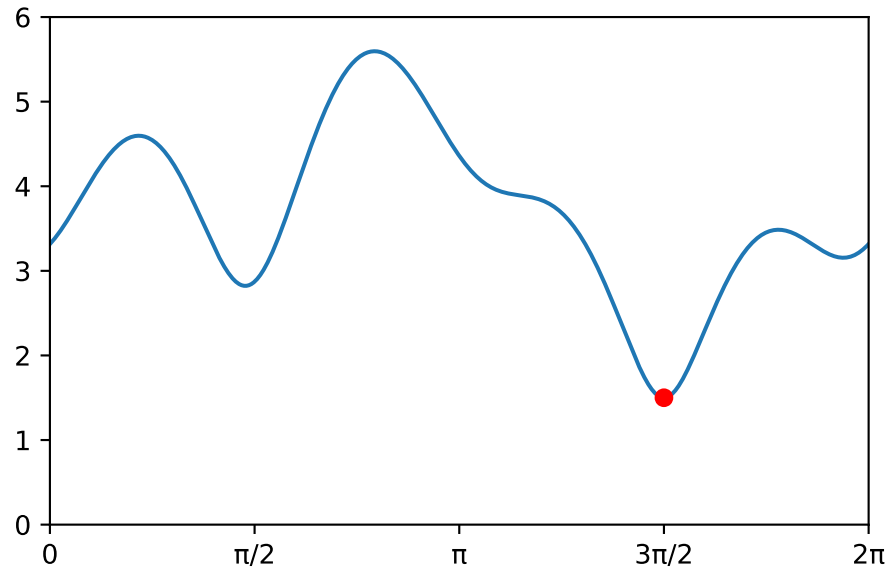


Abbildung 3.2: Kürzester Abstand mit Gradient Descent.

Beachte, dass beim Zeichnen des Funktionsgraphen neu `Y = [d(t)[0] for t in T]` steht. Der Grund dafür ist, dass `d(t)` nun eine Liste mit zwei Elementen ist (Funktionswert und Ableitung) und wir nur den Funktionswert zeichnen wollen. Schreibt man stattdessen `Y = [d(t) for t in T]`, dann wird zusätzlich auch der Graph der Ableitung gezeichnet.

## 3.2 Implementation der SAD mit Operator Overloading

Nach dem letzten Abschnitt könnte man einwenden, dass wir die Ableitungen der Funktionen ja doch von Hand berechnet haben, denn wir haben jede Programmzeile, in der eine Variable `v` berechnet wird, um eine weitere Zeile ergänzt, in der wir `vdot` nach den bekannten Ableitungsregeln berechnet haben. Dieser Einwand ist auch berechtigt - oder wie es Henrard ausdrückt:

> "The bad news is that it [calculating the derivatives] has to be done; it will not appear magically. It is not only a figure of speech that 'something has to be done' but that to have it working everything has to be done". (Henrard 2017, 17)

Die gute Nachricht ist, dass wir diesen Prozess weiter automatisieren können. Wir kennen die Ableitungsregeln für die elementaren Operationen (`+,-,*,/`), sowie für die Grundfunktionen. In diesem Abschnitt werden wir eine Klasse `FloatSad` schreiben, deren Instanzen Funktionswerte und

Werte der Ableitung speichern. Da solche Werte in der Regel vom Typ `float` sind und wir die Standard-AD implementieren, nennen wir die Klasse `FloatSad`. Die Arbeit besteht dann darin, die Ableitungsregeln richtig in den Operatoren dieser Klasse zu kodieren. Da Python *Operator Overloading* kennt, werden wir dann nach getaner Arbeit die Ableitungen wirklich ohne zusätzlichen Programmieraufwand erhalten.

Der Grundstein für unsere Klasse wurde bereits im 19. Jahrhundert gelegt, wie die folgende Infobox zeigt:

**i** Hintergrund: Duale Zahlen

Duale Zahlen wurden 1873 durch William Clifford eingeführt und sind ähnlich definiert, wie komplexe Zahlen. Zur Erinnerung: Eine komplexe Zahl ist eine Zahl der Form $a+bi$,wobei $a, b \in \mathbb{R}$ sind und $i$ die Eigenschaft $i^2 = -1$ hat. Eine duale Zahl ist eine Zahl der Form $a+a'\epsilon$, wobei wieder $a, a' \in \mathbb{R}$ gilt, aber $\epsilon$ die Eigenschaft $\epsilon^2 = 0$ hat. Nun kann man nach dem Permanenzprinzip die folgenden Operationen für duale Zahlen definieren:

$$
\begin{aligned}
&\text{Addition:} && (a + a'\epsilon) + (b + b'\epsilon) &&= (a + b) + (a' + b')\epsilon \\
&\text{Subtraktion:} && (a + a'\epsilon) - (b + b'\epsilon) &&= (a - b) + (a' - b')\epsilon \\
&\text{Multiplikation:} && (a + a'\epsilon) \cdot (b + b'\epsilon) &&= ab + a'b\epsilon + ab'\epsilon + a'b'\epsilon^2 \\
& && && = ab + (a'b + ab')\epsilon \\
&\text{Division:} && \text{(für } b \neq 0\text{)} \quad \frac{a + a'\epsilon}{b + b'\epsilon} &&= \frac{(a + a'\epsilon)(b - b'\epsilon)}{(b + b'\epsilon)(b - b'\epsilon)} \\
& && && = \frac{ab + a'b\epsilon - ab'\epsilon - a'b'\epsilon^2}{b^2 - (b')^2\epsilon^2} \\
& && && = \frac{ab + (a'b - ab')\epsilon}{b^2} \\
& && && = \frac{a}{b} + \frac{a'b - ab'}{b^2}\epsilon
\end{aligned}
$$

Wir sehen, dass der reelle Teil den Wert der Operation und der duale Teil den Wert der zugehörigen Ableitung enthält. Dies gilt auch für Potenzen, wie man unter Anwendung des binomischen Satzes sieht:

$$
\begin{aligned}
(a + a'\epsilon)^n &= \sum_{k=0}^{n} \binom{n}{k} a^{n-k} (a'\epsilon)^k \\
&= a^n + n \cdot a^{n-1} \cdot a'\epsilon + (\text{Terme mit } \epsilon^2) \\
&= a^n + n \cdot a^{n-1} \cdot a'\epsilon
\end{aligned}
$$

Im dualen Teil erkennen wir die Kettenregel $(a^n)' = n \cdot a^{n-1} \cdot a'$. Damit können wir duale Zahlen auch in Polynome $p(x) = p_0 + p_1 x + p_2 x^2 + ... + p_n x^n$ einsetzen. Wir erhalten dann

$$
\begin{aligned}
p(a + a'\epsilon) &= p_0 + p_1(a + a'\epsilon) + p_2(a + a'\epsilon)^2 + ... + p_n(a + a'\epsilon)^n \\
&= p_0 + p_1 a + p_1 a'\epsilon + p_2 a^2 + p_2 \cdot 2aa'\epsilon + ... + p_n a^n + p_n \cdot n a^{n-1} a'\epsilon \\
&= p_0 + p_1 a + p_2 a^2 + ... p_n a^n + (p_1 + p_2 \cdot 2a + ... + p_n \cdot n a^{n-1}) \cdot a'\epsilon \\
&= p(a) + p'(a) \cdot a'\epsilon
\end{aligned}
$$

Dieses Resultat lässt sich auf allgemeine Funktionen $f$ verallgemeinern (für den Beweis entwickelt man $f$ in eine Taylorreihe und macht die gleichen Überlegungen wie für ein Polynom):

$$
f(a + a'\epsilon) = f(a) + f'(a) \cdot a'\epsilon
$$

(Wikipedia: Dual number und Slater (2022))

### 3.2.1 Die Klasse `FloatSad`

Beginnen wir nun mit der Implementation unserer Klasse `FloatSad`. Analog zu den dualen Zahlen enthält jedes `FloatSad`-Objekt zwei Attribute. Das Attribut `value` speichert den Funktionswert und das Attribut `derivative` speichert den Wert der Ableitung. Im Konstruktor der Klasse setzen wir für `derivative` den Standardwert `1`. Damit können wir eine gewöhnliche `Float`-Zahl korrekt in ein `FloatSad` umwandeln. Dies wird im `main` Programm demonstriert.

```
import math

class FloatSad:

    def __init__(self, value, derivative = 1.0):
        self.value = float(value)
        self.derivative = derivative


if __name__ == '__main__':
```

```
def f(x):
    v0 = FloatSad(x)
    y = v0
    return y

x0 = 2
resultat = f(x0)
print("Funktionswert:", resultat.value)
print("Ableitung:", resultat.derivative)
```

```
Funktionswert: 2.0
Ableitung: 1.0
```

In der Funktion `f` haben wir nun unsere Konvention, dass `v0 = x` sein soll, dazu verwendet, den Zahlenwert `x` in ein `FloatSad`-Objekt umzuwandeln. Die Konvention `v0dot = 1` ist im Konstruktor kodiert. Von nun an machen wir also die folgende Konvention:

> **!** Konvention
>
> Eine Funktion berechnet aus einem Argument `x` vom Typ `float` oder `int` einen Rückgabewert `y` vom Typ `FloatSad` über eine Reihe von Hilfsvariablen `v`, die alle vom Typ `FloatSad` sind. Insbesondere setzen wir am Anfang immer `v0 = FloatSad(x)`.

Das obige Programm berechnet also den Funktionswert und den Wert der Ableitung von $f(x) = x$ an der Stelle $x_0 = 2$.

Um die Ausgabe etwas einfacher zu gestalten implementieren wir als nächstes die `print` Methode für unsere Klasse. Wir geben ein `FloatSad`-Objekt einfach in der Form `< value ; derivative >` aus.

```
def __repr__(self):
        return "< " + str(self.value) + " ; " + str(self.derivative) + " >"
```

Da wir nun die Funktion $f(x) = x$ programmieren können, wollen wir als nächstes auch die Funktion $f(x) = -x$ programmieren können. Wir müssen unsere `FloatSad`-Objekte also mit Vorzeichen versehen.

### 3.2.2 Vorzeichen

Natürlich wollen wir nicht nur das negative Vorzeichen, sondern auch das positive Vorzeichen implementieren, damit wir in unseren Programmen z.B. `v1 = +v0` oder `v2 = -v0` schreiben können.

Beim positiven Vorzeichen müssen wir nichts machen, wir geben also ein `FloatSad`-Objekt mit den gleichen Attributen zurück. Beim negativen Vorzeichen ändern beide Attribute ihr Vorzeichen.

```
def __pos__(self):
        return FloatSad(self.value, self.derivative)

def __neg__(self):
    newValue = -self.value
    newDerivative = -self.derivative
    return FloatSad(newValue, newDerivative)
```

Nun gehen wir daran, die Grundoperationen für `FloatSad`-Objekte zu implementieren.

### 3.2.3 Die Operatoren + und -

Wir möchten in unseren Programmen Anweisungen wie `v2 = v0 + v1` verwenden können. Gemäss der Summenregel können wir dazu einfach die Funktionswerte und auch die Werte der Ableitungen addieren.

```
def __add__(self, other):
        newValue = self.value + other.value
        newDerivative = self.derivative + other.derivative
        return FloatSad(newValue, newDerivative)
```

Nun können wir zwei `FloatSad`-Objekte miteinander addieren. Manchmal möchten wir aber auch ein `float`- oder `int`-Wert zu einem `FloatSad`-Objekt addieren, z.B. `v1 = v0 + 2`. Dazu machen wir eine Typabfrage und passen den Wert der Ableitung entsprechend der Konstantenregel an:

```
def __add__(self, other):
    if type(other) in (float, int):
        newValue = self.value + other
        newDerivative = self.derivative + 0.0
    else:
        newValue = self.value + other.value
        newDerivative = self.derivative + other.derivative
    return FloatSad(newValue, newDerivative)
```

Jetzt funktioniert zwar die Anweisung `v1 = v0 + 2`, aber die Anweisung `v1 = 2 + v0` erzeugt immer noch eine Fehlermeldung. Um dieses Problem zu beheben, müssen wir als nächstes den *reverse-add*-Operator implementieren.

```
def __radd__(self, other):
    if type(other) in (float, int):
        newValue = other + self.value
        newDerivative = 0.0 + self.derivative
    else:
        newValue = other.value + self.value
        newDerivative = other.derivative + self.derivative
    return FloatSad(newValue, newDerivative)
```

Hier ist die bisher implementierte Klasse zusammen mit einem kleinen Testprogramm.

```
import math

class FloatSad:

    def __init__(self, value, derivative = 1.0):
        self.value = float(value)
        self.derivative = derivative

    def __repr__(self):
        return "< " + str(self.value) + " ; " + str(self.derivative) + " >"


    # unäre Operatoren

    def __pos__(self):
        return FloatSad(self.value, self.derivative)

    def __neg__(self):
        newValue = -self.value
        newDerivative = -self.derivative
        return FloatSad(newValue, newDerivative)


    # binäre Operatoren

    def __add__(self, other):
        if type(other) in (float, int):
            newValue = self.value + other
            newDerivative = self.derivative + 0.0
        else:
            newValue = self.value + other.value
```

```
            newDerivative = self.derivative + other.derivative
        return FloatSad(newValue, newDerivative)

    def __radd__(self, other):
        if type(other) in (float, int):
            newValue = other + self.value
            newDerivative = 0.0 + self.derivative
        else:
            newValue = other.value + self.value
            newDerivative = other.derivative + self.derivative
        return FloatSad(newValue, newDerivative)



if __name__ == '__main__':

    def f(x):
        v0 = FloatSad(x)
        v1 = -v0
        v2 = 3 + v1
        v3 = v2 + v1
        y  = +v3
        return y

    resultat = f(2)
    print(resultat)
```

```
< -1.0 ; -2.0 >
```

**Übungsaufgabe 3.8** (Korrektheit überprüfen)**.** Welche Funktion berechnet `f` im `main` Programm? Stimmt die Ausgabe?

Lösung

Es handelt sich um die Funktion $f(x) = 3 - 2x$. Die Ausgabe $f(2) = -1, f'(2) = -2$ ist also korrekt.

Für die nächste Übung musst du das obige Programm kopieren und in einer Datei mit dem Namen `floatsad.py` abspeichern. Speichere die Datei im gleichen Ordner wie die anderen Dateien.

**Übungsaufgabe 3.9** (Den Operator `-` implementieren)**.** Implementiere auf die gleiche Weise den `-`-Operator. Die Methoden lauten `__sub__(self, other)` bzw. `__rsub__(self, other)`. Schreibe auch eine Testfunktion `f`, welche die neuen Operatoren verwendet.

Lösung

```
def __sub__(self, other):
    if type(other) in (float, int):
        newValue = self.value - other
        newDerivative = self.derivative - 0.0
    else:
        newValue = self.value - other.value
        newDerivative = self.derivative - other.derivative
    return FloatSad(newValue, newDerivative)

def __rsub__(self, other):
    if type(other) in (float, int):
        newValue = other - self.value
        newDerivative = 0.0 - self.derivative
    else:
        newValue = other.value - self.value
        newDerivative = other.derivative - self.derivative
    return FloatSad(newValue, newDerivative)
```

### 3.2.4 Die Operatoren `*` und `/`

Als nächstes wollen wir die Multiplikation implementieren, um Anweisungen der Form `v2 = v0 * v1` ausführen zu können. Dazu müssen wir die Produktregel verwenden. Wie bei der Addition und der Subtraktion soll unser `*`-Operator aber auch Anweisungen der Form `v1 = v0 * 2` oder `v1 = -3 * v0` richtig auswerten, bei denen die Faktorregel angewendet wird. Dazu ist wieder eine Typabfrage nötig.

**Übungsaufgabe 3.10** (Den Operator `*` implementieren)**.** Ergänze die Datei `floatsad.py` mit den Methoden `__mul__(self, other)` und `__rmul__(self, other)`. Überlege dir verschiedene Testfälle und überzeuge dich von der Korrektheit deines Programms.

Lösung

```
def __mul__(self, other):
    if type(other) in (float, int):
        newValue = self.value * other
        newDerivative = self.derivative * other
    else:
        newValue = self.value * other.value
        newDerivative = self.derivative * other.value + self.value * other.derivative
    return FloatSad(newValue, newDerivative)

def __rmul__(self, other):
    if type(other) in (float, int):
        newValue = other * self.value
        newDerivative = other * self.derivative
    else:
        newValue = other.value * self.value
        newDerivative =  other.derivative * self.value + other.value * self.derivative
    return FloatSad(newValue, newDerivative)
```

Es fehlt noch der Divisionsoperator, damit wir Anweisungen wie `v2 = v1 / v0` verwenden können. Da wir es bei differenzierbaren Funktionen immer mit `float`- bzw. `FloatSad`-Objekten zu tun haben, implementieren wir nur den `/`-Operator, also die Funktion `__truediv__(self, other)` und nicht den `//`-Operator. Wir wollen aber wieder die Fallunterscheidung nach den Typen machen, so dass auch Anweisungen wie `v1 = v0 / 4` verwendet werden können. Dabei benötigen wir nur die Faktorregel und nicht die Quotientenregel. Um schliesslich auch noch `v1 = -4 / v0` zu ermöglichen, muss noch `__rtruediv__(self, other)` implementiert werden. Bei letzterem darf nicht vergessen werden, dass auch die Kettenregel benutzt werden muss, denn $\frac{dv_1}{dx} = \frac{dv_1}{dv_0} \cdot \frac{dv_0}{dx} = \frac{4}{v_0^2} \cdot v_0'$. Quadrate kann man mit `math.pow(value, 2)` berechnen. Bei der Implementierung müssen wir uns übrigens nicht um Fehlerbehandlungen, wie das Abfangen einer Division durch Null, kümmern, weil diese bereits im `/`-Operator, den wir verwenden, implementiert sind.

**Übungsaufgabe 3.11** (Den Operator `/` implementieren)**.** Ergänze die Datei `floatsad.py` mit den Methoden `__truediv__(self, other)` und `__rtruediv__(self, other)`. Überlege dir auch wieder verschiedene Testfälle und überzeuge dich von der Korrektheit deines Programms.

Lösung

```
def __truediv__(self, other):
    if type(other) in (float, int):
        newValue = self.value / other
        newDerivative = self.derivative / other
    else:
        newValue = self.value / other.value
        newDerivative = (self.derivative * other.value - self.value * other.derivative) /
    return FloatSad(newValue, newDerivative)

def __rtruediv__(self, other):
    if type(other) in (float, int):
        newValue = other / self.value
        newDerivative = - other / math.pow(self.value, 2) * self.derivative
    else:
        newValue = other.value / self.value
        newDerivative = (other.derivative * self.value - other.value * self.derivative) /
    return FloatSad(newValue, newDerivative)
```

### 3.2.5 Der Operator **

Interessant ist nun die Implementation des Potenzoperators. Hier sind mehrere Fallunterscheidungen nötig.

Betrachten wir zuerst den Fall, `type(other) in (float, int)`, d.h. wir haben einen Ausdruck der Form `v1 = v0 ** 3`. In diesem Fall wenden wir die Potenzregel zusammen mit der Kettenregel an.

Im zweiten Fall haben wir einen Ausdruck wie `v3 = v1 ** v2`. Wir müssen uns also zuerst überlegen, wie wir einen Ausdruck der Form $v_3(x) = v_1(x)^{v_2(x)}$ überhaupt ableiten. Offenbar muss dazu $v_1(x) > 0$ gelten. Um die Ableitung zu finden wenden wir den Trick an, dass wir die Funktion zuerst logarithmieren,

$$\ln(v_3(x)) = \ln(v_1(x)^{v_2(x)}) = v_2(x) \cdot \ln(v_1(x))$$

und danach beide Seiten ableiten, wobei wir auf der rechten Seite die Produktregel anwenden:

$$\frac{d}{dx}(\ln(v_3(x))) = v_2'(x) \cdot \ln(v_1(x)) + v_2(x) \cdot \frac{1}{v_1(x)} \cdot v_1'(x)$$

Die linke Seite ergibt andererseits $\frac{d}{dx}(\ln(v_3(x))) = \frac{1}{v_3(x)} \cdot v_3'(x)$, so dass wir nun nach $v_3'(x)$ auflösen können:

$$
\begin{aligned}
v_3'(x) &= v_3(x) \cdot \left( v_2'(x) \cdot \ln(v_1(x)) + \frac{v_2(x)}{v_1(x)} \cdot v_1'(x) \right) \\
&= v_1(x)^{v_2(x)} \cdot \left( \ln(v_1(x)) \cdot v_2'(x) + \frac{v_2(x)}{v_1(x)} \cdot v_1'(x) \right)
\end{aligned}
$$

Auch hier sind alle nötigen Fehlerbehandlungen bereits in `math.pow` implementiert.

**Übungsaufgabe 3.12** (Den Operator `**` implementieren - Teil 1)**.** Ergänze die Datei `floatsad.py` mit der Methode `__pow__(self, other)`. Dabei übernimmt `self` die Rolle von $v_1$ in der obigen Herleitung und `other` entspricht $v_2$. Die Funktion ln(...) heisst in Python `math.log()`. Teste dein Programm an verschiedenen Funktionen.

Lösung

```
def __pow__(self, other):
    if type(other) in (float, int):
        newValue = math.pow(self.value, other)
        newDerivative = other * math.pow(self.value, other - 1) * self.derivative
    else:
        newValue = math.pow(self.value, other.value)
        newDerivative = math.pow(self.value, other.value) * \
            (other.derivative * math.log(self.value) + other.value * self.derivative / sel
    return FloatSad(newValue, newDerivative)
```

Nun implementieren wir auch noch die Methode `__rpow__(self, other)`. Im Fall, dass `type(other) in (float, int)` ist, handelt es sich hierbei um eine Exponentialfunktion. `math.pow` stellt dann sicher, dass die Basis, also `other`, eine positive Zahl ist. Falls `other` ebenfalls ein `FloatSad` ist, dann kann die Ableitung gleich wie oben berechnet werden, ausser, dass jetzt `self` und `other` ihre Rollen tauschen.

**Übungsaufgabe 3.13** (Den Operator `**` implementieren - Teil 2)**.** Ergänze die Datei `floatsad.py` mit der Methode `__rpow__(self, other)`. Teste dein Programm an verschiedenen Funktionen.

Lösung

```
def __rpow__(self, other):
    if type(other) in (float, int):
        newValue = math.pow(other, self.value)
        newDerivative = math.pow(other, self.value) * math.log(other) * self.derivative
    else:
        newValue = math.pow(other.value, self.value)
        newDerivative = math.pow(other.value, self.value) * \
            (self.derivative * math.log(other.value) + self.value * other.derivative / oth
    return FloatSad(newValue, newDerivative)
```

### 3.2.6 Vergleichsopertoren

Es könnte sein, dass wir `FloatSad`-Objekte auch miteinander vergleichen wollen, also eine der Abfragen aus Tabelle 3.1 machen wollen.

| Operator | Methode |
|---|---|
| `<` | `__lt__(self, other)` |
| `<=` | `__le__(self, other)` |
| `==` | `__eq__(self, other)` |
| `!=` | `__ne__(self, other)` |
| `>` | `__gt__(self, other)` |
| `>=` | `__ge__(self, other)` |

Tabelle 3.1: Vergleichsoperatoren

Dazu vergleichen wir jeweils nur die Funktionswerte. Die Implementation sieht dann folgendermassen aus:

```
# Vergleichsoperatoren

def __lt__(self, other):
    if type(other) in (float, int):
        return self.value < other
    else:
        return self.value < other.value

def __le__(self, other):
```

```
    if type(other) in (float, int):
        return self.value <= other
    else:
        return self.value <= other.value

def __eq__(self, other):
    if type(other) in (float, int):
        return self.value == other
    else:
        return self.value == other.value

def __ne__(self, other):
    if type(other) in (float, int):
        return self.value != other
    else:
        return self.value != other.value

def __gt__(self, other):
    if type(other) in (float, int):
        return self.value > other
    else:
        return self.value > other.value

def __ge__(self, other):
    if type(other) in (float, int):
        return self.value >= other
    else:
        return self.value >= other.value
```

## 3.3 Die Klasse `FloatSad` im Einsatz

Falls im letzten Abschnitt etwas nicht geklappt haben sollte, kann die fertige Klasse `FloatSad` von hier kopiert werden.

Um unsere Klasse zu verwenden müssen wir sie jeweils am Anfang mit

```
from floatsad import FloatSad
```

einbinden.

**Beispiel 3.1** (Ein Programm mit FloatSad)**.** Betrachte die Funktion aus Beispiel 1.1. Wir übernehmen das Programm und passen lediglich die erste Zeile der Funktion gemäss der Konvention aus Kapitel 3.2.1 an.

```
from floatsad import FloatSad

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

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

```
< -4.0 ; 3.0 >
```

Da nun alle Ableitungsregeln in den verwendeten Operatoren integriert sind, können wir nun sogar auf die Zwischenschritte mit den `v` verzichten:

```
from floatsad import FloatSad

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

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

```
< -4.0 ; 3.0 >
```

---

Wir sehen, dass wir also alle unsere Konventionen, die dazu dienten, komplizierte Funktionsausdrücke in ihre Bestandteile zu zerlegen und diese mit den elementaren Ableitungsregeln zu differenzieren, wieder aufgeben können! Der einzige Zusatzaufwand, den wir bei der Programmierung haben, ist das Umwandeln des Arguments `x` in ein `FloatSad`-Objekt.

**Übungsaufgabe 3.14** (`FloatSad` anwenden)**.** Vereinfache die Lösung von Übungsaufgabe 3.5 mit Hilfe der Klasse `FloatSad`. Überzeuge dich davon, dass die Ableitungen auch für Programme mit Schleifen korrekt berechnet werden.

Lösung

```
from floatsad import FloatSad

def l(x):
    y = x**2 + 1
    return y

def f(x):
    x = FloatSad(x)
    for i in range(3):
        x = l(x)
    return x

print(f(1))
```

```
< 26.0 ; 80.0 >
```

## 3.4 Das Modul `mathsad`

Mit der Klasse `FloatSad` können wir Funktionswerte und Ableitungen von algebraischen Funktionen bilden. Wir können aber unsere `FloatSad`-Objekte noch nicht mit den Funktionen aus dem Python-Modul `math` verwenden, z.B. mit `exp` oder `sin`. In diesem Abschnitt wollen wir ein eigenes Modul `mathsad` schreiben, in dem wir die Funktionen aus Tabelle 3.2 so implementieren, dass wir sie auf `FloatSad`-Objekte anwenden können.

| | | |
|---|---|---|
| sqrt | exp | log |
| sin | cos | tan |
| asin | acos | atan |
| sinh | cosh | tanh |
| asinh | acosh | atanh |
| fabs | | |

Tabelle 3.2: Funktionen des Moduls `math` (Auswahl)

Gemäss der Python-Dokumentation liefert die Funktion `math.exp(x)` präzisere Werte als `math.e ** x` oder `math.pow(math.e, x)`. Die Funktion `math.log(x)` berechnet den Logarithmus zur Basis $e$, man kann ihr aber als zweites Argument auch eine andere Basis übergeben, z.B. `math.log(x,b)`, was dann mit `math.log(x)/math.log(b)` berechnet wird. Die Funktion `math.fabs(x)` schliesslich berechnet den Absolutbetrag $|x|$. Ihre Ableitung ist

```
(math.fabs(v)).derivative = v.derivative if v>=0 else -v.derivative
```

Die Funktion $y = |x|$ ist an der Stelle $x = 0$ eigentlich nicht differenzierbar. Da wir aber nicht Ableitungsfunktionen, sondern nur Werte von Ableitungen an einer bestimmten Stelle berechnen, reicht es, den rechts- oder linksseitigen Grenzwert zurückzugeben, siehe Gander (1992). Wir müssen es dem Benutzer überlassen, das Ergebnis im jeweiligen Kontext korrekt zu interpretieren.

### 3.4.1 Die Funktion `sqrt`

Beginnen wir mit der Implementierung der Wurzelfunktion.

```
import math
from floatsad import FloatSad

def sqrt(x):
    newValue = math.sqrt(x.value)
    newDerivative = 1/(2*math.sqrt(x.value)) * x.derivative
    return FloatSad(newValue, newDerivative)

if __name__ == '__main__':

    def f(x):
        x = FloatSad(x)
        y = 1 / sqrt(x**2 + 1)
        return y

    x0 = -1
    print(f(x0))
```

```
< 0.7071067811865475 ; 0.3535533905932737 >
```

Wir gehen davon aus, dass `x` ein `FloatSad`-Objekt ist. Für den Wert von `sqrt(x)` verwenden wir einfach die Funktion `math.sqrt`. Diese enthält auch die nötige Fehlerbehandlung. Zusätzlich berechnen wir aber noch den Wert der Ableitung mit Hilfe der bekannten Ableitungsregel und wie zuvor wenden wir immer die Kettenregel an. Das Programm enthält auch ein Testprogramm, welches die Ableitung der

Funktion $f(x) = \frac{1}{\sqrt{x^2+1}}$ an der Stelle $x_0 = -1$ berechnet. Zur Kontrolle kann die GeoGebra-Vorlage zu Beginn von Kapitel 3.1 verwendet werden.

### 3.4.2 Die Funktionen `exp` und `log`

**Übungsaufgabe 3.15** (Exponentialfunktion)**.** Kopiere den obigen Code und speichere ihn in einer Datei mit dem Namen `mathsad.py`. Speichere die Datei im gleichen Ordner wie die anderen Dateien. Ergänze die Datei danach mit der Funktion `exp`. Wähle eine neue Testfunktion im `main`, um dich von der Richtigkeit deiner Lösung zu überzeugen.

Lösung

```
def exp(x):
    newValue = math.exp(x.value)
    newDerivative = math.exp(x.value) * x.derivative
    return FloatSad(newValue, newDerivative)
```

Für die Logarithmusfunktion müssen wir uns wieder etwas mehr Gedanken machen. Mit `def log(x, b = math.e)` kann man der Basis $b$ wie oben beschrieben den Standardwert $b = e$ geben. Solange `b` vom Typ `int` oder `float` ist, kann man einfach die bekannte Ableitungsregel anwenden. Wenn aber `b` ein `FloatSad`-Objekt ist, wie z.B. in `v3 = math.log(v1, v2)`, dann müssen wir den Basiswechselsatz

$$v_3(x) = \log_{v_2(x)}(v_1(x)) = \frac{\ln(v_1(x))}{\ln(v_2(x))}$$

verwenden und mit der Quotientenregel ableiten.

**Übungsaufgabe 3.16** (Logarithmusfunktion)**.** Überlege dir, wie die Ableitung von $v_3(x)$ aussieht. Ergänze danach die Datei `mathsad.py` mit der Implementation der Logarithmusfunktion. Überzeuge dich mit einer Testfunktion von der Richtigkeit deines Programms.

Lösung

Die Ableitung lautet

$$\frac{d}{dx}v_3(x) = \frac{\frac{v_1'(x)}{v_1(x)} \cdot \ln(v_2(x)) - \ln(v_1(x)) \cdot \frac{v_2'(x)}{v_2(x)}}{\ln^2(v_2(x))}$$

```
def log(x, b = math.e):
    if type(b) in (float, int):
        newValue = math.log(x.value, b)
        newDerivative = 1 / (x.value * math.log(b)) * x.derivative
    else:
        newValue = math.log(x.value, b.value)
        newDerivative = (x.derivative/x.value * math.log(b.value) - math.log(x.value) * b.
            / math.pow(math.log(b.value), 2)
    return FloatSad(newValue, newDerivative)
```

### 3.4.3 Die trigonometrischen Funktionen und ihre Umkehrfunktionen

Bei den trigonometrischen Funktionen und den Arcus Funktionen können wir einfach die bekannten Ableitungsregeln verwenden.

**Übungsaufgabe 3.17** (Trigonometrische Funktionen)**.** Ergänze die Datei `mathsad.py` mit den Funktionen `sin`, `cos` und `tan`, sowie den Funktionen `asin`, `acos` und `atan`.

Lösung

Beachte, dass man für `tan` einfach $\tan(x) = \frac{\sin(x)}{\cos(x)}$ verwenden kann, wenn `sin` und `cos` bereits implementiert sind.

```
def sin(x):
    newValue = math.sin(x.value)
    newDerivative = math.cos(x.value) * x.derivative
    return FloatSad(newValue, newDerivative)

def cos(x):
    newValue = math.cos(x.value)
    newDerivative = -math.sin(x.value) * x.derivative
    return FloatSad(newValue, newDerivative)

def tan(x):
    return sin(x) / cos(x)

def asin(x):
    newValue = math.asin(x.value)
    newDerivative = 1/math.sqrt( 1 - math.pow(x.value, 2)) * x.derivative
    return FloatSad(newValue, newDerivative)

def acos(x):
    newValue = math.acos(x.value)
    newDerivative = -1/math.sqrt( 1 - math.pow(x.value, 2)) * x.derivative
    return FloatSad(newValue, newDerivative)

def atan(x):
    newValue = math.atan(x.value)
    newDerivative = 1/(math.pow(x.value, 2) + 1) * x.derivative
    return FloatSad(newValue, newDerivative)
```

### 3.4.4 Die hyperbolischen Funktionen und ihre Umkehrfunktionen

Auch bei den hyperbolischen Funktionen und den Area Funktionen verwenden wir die bekannten Ableitungsregeln.

**Übungsaufgabe 3.18** (Hyperbolische Funktionen)**.** Ergänze die Datei `mathsad.py` mit den Funktionen `sinh`, `cosh` und `tanh`, sowie den Funktionen `asinh`, `acosh` und `atanh`.

Lösung

Wie bei den trigonometrischen Funktionen gilt auch hier $\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$.

```
def sinh(x):
    newValue = math.sinh(x.value)
    newDerivative = math.cosh(x.value) * x.derivative
    return FloatSad(newValue, newDerivative)

def cosh(x):
    newValue = math.cosh(x.value)
    newDerivative = math.sinh(x.value) * x.derivative
    return FloatSad(newValue, newDerivative)

def tanh(x):
    return sinh(x) / cosh(x)

def asinh(x):
    newValue = math.asinh(x.value)
    newDerivative = 1/math.sqrt(math.pow(x.value, 2) + 1) * x.derivative
    return FloatSad(newValue, newDerivative)

def acosh(x):
    newValue = math.acosh(x.value)
    newDerivative = 1/math.sqrt(math.pow(x.value, 2) - 1) * x.derivative
    return FloatSad(newValue, newDerivative)

def atanh(x):
    newValue = math.atanh(x.value)
    newDerivative = -1/(math.pow(x.value, 2) - 1) * x.derivative
    return FloatSad(newValue, newDerivative)
```

### 3.4.5 Die Betragsfunktion

Schliesslich ergänzen wir die Datei `mathsad.py` noch mit der Funktion `fabs` wie oben beschrieben:

```
def fabs(x):
    newValue = math.fabs(x.value)
    newDerivative = x.derivative if x>=0 else -x.derivative
    return FloatSad(newValue, newDerivative)
```

Das fertige Modul kann auch von hier kopiert werden.

## 3.5 Das Modul `mathsad` im Einsatz

Nun können wir unser Modul mit

```
import mathsad
```

einbinden und verwenden. Die Funktion aus Übungsaufgabe 3.3 beispielsweise können wir nun direkt hinschreiben:

```
from floatsad import FloatSad
import mathsad

def f(x):
    x = FloatSad(x)
    y = mathsad.cos(x**2 + 2) * mathsad.exp(-1/2 * x**2) + 1/x
    return y

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

```
< -0.3700550823007931 ; -0.14136926695938976 >
```

**Übungsaufgabe 3.19** (Verwendung von `mathsad`)**.** Verwende das Modul `mathsad`, um die Lösung von Übungsaufgabe 3.4 zu vereinfachen. Bestimme die Ableitung von $f$ an der Stelle $x_0 = \sqrt{2}$.

Lösung

```
from floatsad import FloatSad
import math
import mathsad

def f(x):
    x = FloatSad(x)
    u = x**2 + 1
    y = mathsad.log(u) / mathsad.sqrt(u + x)
    return y

x0 = math.sqrt(2)
f0 = f(x0)
print(f0.derivative)
```

```
0.22198842685304976
```

**Übungsaufgabe 3.20** (Billard-Problem mit `mathsad`)**.** Verwende das Modul `mathsad`, um die Lösung des Billard-Problems aus Übungsaufgabe 3.6 zu vereinfachen. Programmiere dazu nochmals die Funktion `f(x)`, aber verwende aussagekräftigere Variablen. Weil `f` nun `FloatSad`-Objekte zurückgibt, muss auch die Funktion `newton(f, x0)` angepasst werden. Die Funktion `main` kann aus der obigen Lösung kopiert werden.

Lösung

In der Funktion `newton(f, x0)` muss lediglich die Berechnung des neuen Näherungswertes angepasst werden durch `x1 = x0 - y0.value / y0.derivative`.

```
from floatsad import FloatSad
import math
import mathsad
import matplotlib.pyplot as plt

def f(x):
    # Parameter a, px, py werden im global space gefunden
    x = FloatSad(x)
    Xx, Xy = mathsad.cos(x), mathsad.sin(x) # Koordinaten von X
    tx, ty = -Xy, Xx                        # Komponenten des Tangentialvektors
    XPx, XPy = px - Xx, py - Xy             # Komponenten des Vektors XP
    lXP = mathsad.sqrt(XPx**2 + XPy**2)     # Länge des Vektors XP
    ePx, ePy = XPx / lXP, XPy / lXP         # Komponenten des Einheitsvektors in Richtung
    XQx, XQy = a - Xx, -Xy                  # Komponenten des Vektors XQ
    lXQ = mathsad.sqrt(XQx**2 + XQy**2)     # Länge des Vektors XQ
    eQx, eQy = XQx / lXQ, XQy / lXQ         # Komponenten des Einheitsvektors in Richtung
    y = (ePx + eQx) * tx + (ePy + eQy) * ty # Skalarprodukt
    return y

def newton(f, x0):
    tol = 1e-8
    y0 = f(x0)
    x1 = x0 - y0.value / y0.derivative
    while math.fabs(x1 - x0) > tol:
        x0 = x1
        y0 = f(x0)
        x1 = x0 - y0.value / y0.derivative
    return x1

if __name__ == "__main__":

    # Parameter definieren
    a = -0.5          # Position von Q = (a|0)
    px, py = 0.2, 0.6 # Position von P = (px|py)

    # Lösung des Billardproblems berechnen
    sol = set({}) # leere Menge, in der die gefundenen Lösungen gespeichert werden
    X = [2*math.pi * k / 10 for k in range(10)]  # Liste der Startwerte für Newton
    for x0 in X:
        x = newton(f, x0)
        sol.add(x)

    # Lösungen grafisch darstellen
    fig = plt.figure()
    ax = plt.gca()
    ax.set_xlim((-1.2, 1.2))
    ax.set_ylim((-1.2, 1.2))
    ax.set_aspect('equal')
    circle = plt.Circle((0,0), 1, color='b', fill=False)
    qBall = plt.Circle((a,0), 0.02, color='k')
    pBall = plt.Circle([px, py], 0.02, color='k')
    ax.add_patch(circle)
    ax.add_patch(qBall)
    ax.add_patch(pBall)
    for x in sol:
```

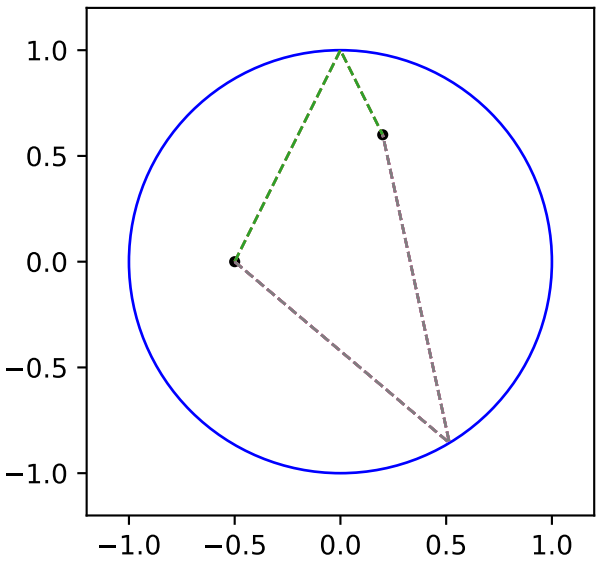


Abbildung 3.3: Lösung des Billardproblems mit anderen Startwerten.

**Übungsaufgabe 3.21** (Kürzeste Distanz mit `mathsad`)**.** Verwende das Modul `mathsad`, um die Lösung von Übungsaufgabe 3.7 zu vereinfachen. Weil `d` nun `FloatSad`-Objekte zurückgibt, muss auch die Funktion `gradient_descent(f, x0, lam)` angepasst werden.

Lösung

In der Funktion `gradient_descent(f, x0, lam)` muss lediglich die Berechnung des neuen Näherungswertes angepasst werden. Statt des Graphen wird hier nur das globale Minimum als Punkt ausgegeben.

```
from floatsad import FloatSad
import math
import mathsad

def d(t):
    t = FloatSad(t)
    Px = 2 * mathsad.cos(t) - 1     # x-Koordinate von P
    Py = 1.5 * mathsad.sin(t)       # y-Koordinate von P
    Pz = 0                          # z-Koordinate von P
    Qx = -3 * mathsad.sin(2*t)      # x-Koordinate von Q
    Qy = 2 * mathsad.cos(2*t) + 1   # y-Koordinate von Q
    Qz = 2 * mathsad.sin(2*t) + 1   # z-Koordinate von Q
    y = mathsad.sqrt((Px-Qx)**2 + (Py-Qy)**2 + (Pz-Qz)**2)
    return y

def gradient_descent(f, x0, lam):
    tol = 1e-9
    # Erster Schritt berechnen
    y0 = f(x0)
    x1 = x0 - lam * y0.derivative
    while math.fabs(x1-x0) > tol:
        x0 = x1
        y0 = f(x0)
        x1 = x0 - lam * y0.derivative
    return x1

if __name__ == "__main__":
    t0 = 3
    tmin = gradient_descent(d, t0, 0.01)
    dmin = d(tmin)
    print("Minimum bei (", tmin, dmin.value, ")")
```

```
Minimum bei ( 4.712388977478413 1.5 )
```

Abbildung 3.4: **?(caption)**

# References


Arens, Tilo, Frank Hettlich, Christian Karpfinger, Ulrich Kockelkorn, Klaus Lichtenegger, und Hellmuth Stachel. 2022. *Mathematik*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Baydin, Atilim Gunes, Barak A. Pearlmutter, Alexey Andreyevich Radul, und Jeffrey Mark Siskind. 2018. „Automatic Differentiation in Machine Learning: a Survey“. *Journal of Machine Learning Research* 18 (153): 1–43. http://jmlr.org/papers/v18/17-468.html.

Gander, Walter. 1992. *Computermathematik*. Birkhäuser.

———. 2015. *Learning MATLAB: A Problem Solving Approach*. 1. Aufl. UNITEXT. Cham, Switzerland: Springer International Publishing.

Griewank, Andreas, und Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd Aufl. Other Titles in Applied Mathematics 105. Philadelphia, PA: SIAM. http://bookstore.siam.org/ot105/.

Henrard, Marc. 2017. *Algorithmic Differentiation in Finance Explained*. Financial Engineering Explained. Cham: Palgrave Macmillan. https://doi.org/10.1007/978-3-319-53979-9.

Hromkovic, Juraj, Jarka Arnold, Cédric Donner, Urs Hauser, Matthias Hauswirth, Tobias Kohn, Dennis Komm, David Maletinsky, und Nicole Roth. 2021. *INFORMATIK, Programmieren und Robotik: Grundlagen der Informatik für Schweizer Maturitätsschulen*.

Slater, Max. 2022. „Differentiable programming from scratch“. Juli 2022. https://thenumb.at/Autodiff/.

Weitz, Edmund. 2021. *Konkrete Mathematik (nicht nur) für Informatiker*. 2. Aufl. Wiesbaden, Germany: Springer Spektrum.