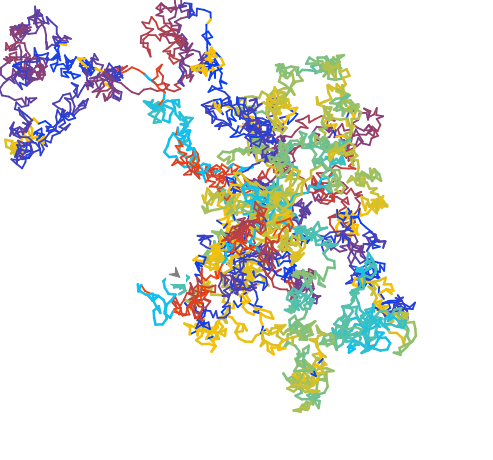
Brown'sche Bewegung als Simulation

**Der zufällige Nachhauseweg[[1]](#footnote-1)**

**Kurz vor der Prüfungssession hast Du in der Schule so viel gelernt, dass Dein Kopf und Dein Gleichgewichtssinn nicht mehr richtig funktionieren. Statt auf direktem Wege nach hause laufen zu können, fällst Du jeweils mit gleicher Wahrscheinlichkeit einen Meter weit in eine beliebige Richtung. Da Du eine robuste Person bist, stehst Du gleich wieder auf... und fällst gleich wieder um. Wie viele Male musst Du im Durchschnitt stürzen, um nach hause zu kommen?**

Du lernst in diesem Modul

* reale Fragestellungen zu abstrahieren und zu modellieren
* sinnvolle Annahmen/Vereinfachungen zu treffen
* manuelle Prozesse in einen Computercode umzuschreiben
* Simulationen durchzuführen und einfache Auswertungen zu machen
* grundlegenden Programmierblöcke selber anzuwenden

Physikalische Phänomene lassen sich oft mit Hilfe eines Computers veranschaulichen und/oder einfacher berechnen und approximieren. Im Unterricht haben wir die Bewegung von Pollen in Wasser betrachtet: Die zufällig erscheinende Bewegung von Pollen in Wasser lässt sich mit Hilfe der Kollisionen, welche durch die Zitterbewegung der die Pollen umgebenden Teilchen entstehen, erklären. Die Frage in der Einleitung lässt sich folgendermassen umformulieren: wie lange geht es, bis ein Pollen eine gewisse Strecke in der Lösung zurückgelegt hat?

Das Ziel dieser drei Lektionen ist es, das Verhalten der Pollen auf vereinfachte Art und Weise mit Hilfe von Simulationen nach zu bauen und gewisse Grössen des Vorgangs zu messen.

Um das Problem überhaupt angehen zu können, schauen wir uns zuerst ein System an, welches durch ein paar Vereinfachungen überschaubar wird.[[2]](#footnote-2)

**1. Vereinfachungen**

Nachfolgende Vereinfachungen erlauben es uns, einen speziellen Fall der Dynamik zu programmieren:

* Die Pollen, welche auf der Oberfläche einer Flüssigkeit schwimmen, bewegen sich (idealisiert) in zwei Dimensionen. Die **erste Vereinfachung** besteht darin, dass wir uns zuerst auf eine Dimension beschränken. Dh der Pollen kann sich nur nach links oder rechts bewegen.
* Die Stösse, welche die Pollen erfahren, sind im realen Fall unterschiedlich gross. Die **zweite Vereinfachung** besteht darin, alle Stösse gleich zu behandeln.
* Die **dritte Vereinfachung** besteht darin, dass wir annehmen, die Stösse der Teilchen in der Flüssigkeit seien von allen Seiten gleich wahrscheinlich.
* Die Zeit ist ereignisgesteuert d.h. jeder Schritt nach links oder rechts erfolgt nach derselben Zeit.

Diese Vereinfachungen lassen uns den Weg der Pollen wie Deinen Nachhauseweg simulieren: statt alle Teilchenkollisionen zu berechnen, schauen wir uns direkt die Dynamik des Pollens an.

**A1**. Was bedeuten diese Vereinfachungen für die Simulation des Nachhauseweges? Diskutiere mit Deinem Teampartner, wie sich die Vereinfachungen auf Deinen Nachhauseweg auswirken.

**2. Von Hand Programmieren**

Bevor wir uns der Umsetzung auf dem Computer widmen, fragen wir uns, ob wir bereits mit einem einfachen Experiment einzelne Aspekte genauer anschauen können.

Du kannst den Fortschritt auf dem Nachhauseweg ohne Computer wie folgt veranschaulichen:

**A2**. Nimm zwei Münzen und lege die erste Münze in der Mitte auf den Tisch. Wirf die zweite Münze: bei Kopf verschiebst Du die erste Münze eine Handbreite nach links, bei Zahl eine Handbreite nach rechts.

**A3**. Überlege Dir, wo sich die Münze nach 6 bzw. 100 mal Werfen am wahrscheinlichsten aufhalten wird.

**A4**. Schätze, wie weit die Münze nach 100 Würfen im Schnitt vom Ursprung wegwandert.

**A5**. Wie könntest Du den Weg in A2 mit Hilfe eines Blattes und eines Stiftes noch besser veranschaulichen?

**3. Konzepte des Programmierens**

Alle nachfolgenden Programmierbausteine kannst Du online ausführen. Wir schreiben unsere Programme in der Sprache *Python* und nutzen ein Onlinetool, um unser Programm laufen zu lassen (dies aus dem Grund, dass wir keine Software auf den Computern installieren müssen).

* Öffne einen Browser und navigiere zur folgenden Website: https://repl.it
* Gibt im Suchfeld ('Search for a language...') *Python* ein und wähle *Python3* aus
* Im Fenster das sich öffnet, gibst Du jeweils auf der linken Seite Dein Programm ein und hast auf der rechten Seite (je nach Programm) eine Ausgabe. Die linke Seite nennen wir *Editor* und die rechte Seite *Terminal*.
* Im *Editor* wirst Du den Code schreiben und/oder anpassen
* Mit *Run* (oberhalb des Editors) führst Du den Code aus
* Die Ausgabe Deines Programmes wird im Terminal ersichtlich sein (oder später mal als Bild gespeichert).

Wir wollen nun Schritt für Schritt das Verhalten der Münze, des Pollens, Deines Nachhauseweges nachbauen.

**3.0 Variabeln**

Variabeln sind nicht nur in der Physik und der Mathematik nützlich, sondern auch essentieller Bestandteil des Programmierens. Um der Variablen a den Wert 7 zuzuweisen, schreibst Du einfach in den Editor

a = 7

Drücke oben auf *Run*, was Dein Programm (die einzelne Zuweisung) ausführt. Um zu überprüfen, ob die Zuweisung funktioniert hat, tippe a ins Terminal und *Enter* (dies sagt dem Terminal "gib mir den Wert von a aus").[[3]](#footnote-3)

**A6.** Weise der Variablen q den Wert 3 zu, der Variable p den Wert -17 und der Variablen t die Summe von p und q**.**

**Gelernt**

Du weisst, was Variablen sind und dass Du mit ihnen Zuweisungen ausführen und ihren Wert ändern kannst und wie Du sie im Terminal abrufst.

**3.1 Speichern der Position**

Als erstes brauchen wir eine Struktur, welche uns den Ort nach einer gewissen Anzahl Schritten speichert.

Eine mögliche Zahlenfolge, welche unseren Ort für die ersten 15 Schritte angibt könnte folgendermassen aussehen:

0, 1, 2, 1, 2, 3, 2, 1, 0, -1, 0, -1, -2, -1, -2,...

0 ist dabei der Startort, 1 der Ort nach dem ersten Schritt, 2 der Ort nach dem zweiten Schritt, 1 der Ort nach dem dritten Schritt, ...

**A7**. Gib an, ob folgende Zahlenfolgen für unseren Weg möglich sind. Begründe Deine Antwort.

0, 1, 2, 3, 2, 3, 4, 2, 1, 0, 1, 0, ...

0, -1, -2, -3, -4, -5, -6, -7, -8, -9, ...

1, 0, -1, -2, -1, 0, -1, 0, -1, 0, ...

Um diese Zahlenfolgen zu speichern, nutzen wir die sogenannten *Listen*:

Die Zuweisung

Ort = [0]

sagt, dass die Grösse Ort eine Liste ist (das wird durch die eckigen Klammern [...] angezeigt) mit erstem Element gleich 0. Du kannst das überprüfen, indem Du im Terminal selber obigen Code ausführst und dann Ort eingibst und *Return* drückst.

**A8**. Erstelle eine neue Grösse Position mit den ersten zwei Werten 1 und 4.

Wir können Elemente zu einer Liste hinzufügen mit Hilfe des Befehles append(). Dazu müssen wir der Liste Ort sagen, dass sie ein neues Element (hier 5) sich selber hinzufügen soll.

Ort.append(5)

**A9**. Füge Position die Werte 8, 5\*\*2, -0.999, 1/7 und 'a' hinzu.

**A10**. Was ist 5\*\*2?

Mit

Ort[3]

greifst Du auf das vierte Element der Liste zu (Listen beginnen mit dem Index 0).

**A11**. Gib das 2-te Element der untenstehenden Liste p aus und ersetze das 5-te Element durch die Zahl -8.

p = [1, 2, 4, 8, 16, 32, 64]

**Gelernt**

Du weisst nun, wie Werte in Variabeln oder Listen gespeichert werden. Das ist wichtig, um z.B. die Position auf dem Weg nach hause zu speichern.

**3.2 Ausführen von vielen Schritten desselben Mechanismus**

Um einen Code-Block mehrmals hintereinander auszuführen (z.B. Wurf der Münze), nutzt man die sogenannten *Schlaufen*. Eine Schlaufe führt einen vordefinierten Block eine gewisse Anzahl mal aus.

**A12**. Kannst Du bereits nachvollziehen, was folgender Code-Block macht?

i = 0  
 while i<10:  
 print(i)  
 i = i + 1

Wir nutzen die sogenannten *while*-Schlaufen, die einen Code-Block so lange ausführen, bis die Bedingung (im obigen Beispiel i < 10) nicht mehr erfüllt ist (bis sie nicht mehr True sondern False ist). Lass den Code nun laufen. Du wirst bemerken, dass Ausgaben im Terminal erscheinen. Dies geschieht mittels der print() Funktion.

Nehmen wir an, dass unsere Münze sich bei der Position 0 befindet und sich nur nach rechts verschieben kann (d.h. die Münze zeigt immer Zahl an). Wir würden gerne die Liste Ort mit den Positionen der Münze füllen.

**A13**. Schreibe einen *while*-Schlaufe, die unserer Liste Ort mit den Zahlen von 5 bis 15 füllt. Nimm dazu untenstehenden Code und erweitere ihn entsprechend.

i =   
 Ort = []  
 while ...:  
 ...

Achtung: Dass der Computer weiss, was alles zur *while*-Schlaufe gehört, wird jede dazugehörige Zeile mit einem *Tabulator* (→|) eingerückt.

Wenn die *while*-Schlaufe richtig ausgeführt wurde, sollte bei Abfrage von Ort im *Terminal* folgender Output auf dem Bildschirm erscheinen:

**Ort** => [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

**A14**. Schreibe eine Schleife die alle ungeraden Zahlen zwischen 0 und 50 in eine Liste mit dem Namen Ungerade\_Zahlen schreibt.

**Gelernt**

Du kannst nun sich wiederholende Anweisungen (z.B. einen Schritt weiter zu gehen) mit Hilfe einer Schlaufe programmieren.

**3.3 Steuerungsstrukturen**

Da Du nicht nur nach rechts, sondern auch nach links umfallen sollst, wollen wir eine Struktur einbauen, welche Entscheidungen fällt. Eine dieser Strukturen ist die *if … then …, else … then …*-Struktur. Wir wollen diese Entscheidungsstruktur nutzen, um zu entscheiden, ob wir jeweiligen Schritt nach links oder nach rechts fallen.

**A15**. Kannst Du bereits erraten, was folgende Struktur macht?

i = -10  
 if i < 30:  
 print('Kleine Zahl')  
 else:  
 print('Grosse Zahl')

**A16**. Was wird die Ausgabe im folgenden Programm sein? Und wie gross ist i am Ende? Versuche die Antwort zu finden, ohne den Code laufen zu lassen.

i = 3\*\*2  
 while i < 100:  
 i = 2\*i + 1  
 if i < 30:  
 print('Kleine Zahl')  
 elif i >=30 & i<40:  
 print('Grosse Zahl')  
 else:  
 print('Sehr grosse Zahl')

Bisher konnten wir nur nach rechts wandern (die Einträge in der Liste Ort in der Aufgabe **A13** wurden jeweils um eins grösser, was einem Schritt nach rechts entspricht). Wir wollen ein kleines Programm schreiben, welches bei jeder geraden Zahl eins nach rechts geht (also: eins addiert) und bei jeder ungeraden Zahl zwei nach links geht (also: zwei subtrahiert).

Dazu brauchen wir aber noch einen neuen Operator:

**A17**. Wie kannst Du entscheiden, ob eine Zahl x gerade oder ungerade ist? Nutze Google um die Antwort auf diese Frage zu finden. Du wirst einen bestimmten *Operator* (mathematischen Operator) finden. Welches Zeichen wird für den Operator genutzt?

**A18**. Wie gross ist 19\_3? 20\_2? 100\_49? ("\_" steht hier für den Operator, welchen Du in Aufgabe A14 finden sollst)

**A19**. Wie kannst Du nun überprüfen, ob es sich um eine gerade oder eine ungerade Zahl handelt?

Wir wollen nun einen Weg finden, wie wir den neuen Ort (wo wir hinfallen) bestimmen können, mit dem Wissen, wo wir uns aktuell befinden.

Allgemein formuliert: Der Ort, an welchem wir uns nach dem i-ten Schritt befinden, finden wir, indem wir den vorherigen Ort nehmen (i-1) und je nach dem eins hinzuzählen (Schritt nach rechts) oder zwei abzählen (Schritt nach links).

Im Code erhälst Du den (i-1)-ten Wert der Ort-Liste folgendermassen:

Ort[i-1]

**A20**. Wie würde der Code aussehen, wenn Du dem i-ten Wert die Summe der zwei vorhergehenden Werte zuordnen möchtest? Schreibe ein Programm, welches jeweils die Summe der zwei vorhergehenden Zahlen der Liste Zahl anfügt, wenn die Liste mit 1 und 2 beginnt. Dann wäre das nächste Element 1 + 2 = 3, das nächste 2 + 3 = 5 usw. Kennst Du diese Zahlenfolge?

Nach diesem Beispiel wollen wir nun einen weitere Schritt in unserem Nachhauseweg-Beispiel nehmen:

**A21**. Erweitere Deinen Code aus A13 so, dass Du mit Hilfe einer *if… then… else… then*... Struktur das gewünschte Verhalten (links/rechts) repliziert wird. In der Ort-Liste sei der Startpunkt (Null) bereits gespeichert.

i = 0

Ort = [0]

while ...:  
 ...  
 if ....:  
 ....  
 else:  
 ....

Die Ausgabe sollte wie folgt aussehen, wenn Du Ort eingibst und *Enter* drückst:

**Ort** => [0, 1, -1, 0, -2, -1, -3, -2, -4, -3, -5, -4, -6, -5, -7, -6, -8]

Das sieht zwar schon zufällig aus, ist aber ein ziemlich einfach zu durchschauendes Muster.

**A22**. Skizziere diesen Weg wie in **A5**. Ist er so zufällig?

**Gelernt**

Du kannst nun Programme mittels Steuerungsstrukturen wie if... else... gezielter einsetzen, kannst zwischen geraden und ungeraden Zahlen unterscheiden und hast dieses Wissen in eine while()-Schlaufe eingebaut. Damit haben wir einen Grossteil der notwendigen Strukturen bereits geschaffen.

**3.4 Der Zufall kommt ins Spiel**

'Richtigen' Zufall gibt es in der Welt der Informatik nicht[[4]](#footnote-4). Da wir das weder können noch wollen, begnügen wir uns mit sogenannten *Pseudo*-Random-Number-Generators (P-RNGs). Die sind zwar nicht richtig zufällig, aber für unser Vorhaben reicht das aus.[[5]](#footnote-5)

Zuerst müssen wir der Programmierumgebung mitteilen, dass wir gerne Zufallszahlen generieren würden. Die Programmierumgebung ist nur mit den wichtigsten Programmierelementen ausgestattet (Schlaufen, Zuweisungen, einfache mathematische Operatoren) und muss explizit erweitert werden. Das machen wir, indem wir eine sogenannte *Bibliothek* importieren. Das geschieht immer zuoeberst im Programm (je nach Sprache auch auf andere Art und Weise).

import random

Damit erweitern wird die elementaren Fähigkeiten von Python um den Umgang mit Zufallszahlen.

Mit folgendem Code generierst Du 10 Zufallszahlen und gibst sie gleich noch aus:

import random

for i in range(0,10):

print(random.random())

Du hast soeben die zweite Art von Schlaufe genutzt: die *for-Schlaufe*. Sie durchläuft alle Elemente in einer Liste (in diesem Falle die Liste von 0 bis 10-1 = 9) und der print()-Befehl gibt das Argument über das Terminal aus. random.random() generiert eine Zufallszahl [0,1).

**A23**. Generiere 5 Zufallszahlen in [5, 15).

**A24**. Generiere 3 Zufallszahlen in [-1,1).

**Gelernt**

Du kannst Zufallszahlen in einem beliebigen Intervall generieren und weisst, wie man die Programmierumgebung mit Bibliotheken erweitert.

**4. Die Brown'sche Bewegung**

Nun wollen wir den ersten vollständigen Zufallspfad (Random Walk) generieren und in der Liste Ort[] speichern. Er soll sich über 100 Schritte erstrecken und mit Wahrscheinlichkeit 50% nach links bzw. rechts einen Schritt ausführen.

**A25**. Vervollständige das untenstehende Programm so, dass 100 Schritte der Brown'schen Bewegung ausgeführt werden, wobei p\_r die Wahrscheinlichkeit angibt, nach rechts zu gehen (**p**robability **r**ight). In einem ersten Schritt soll sie 50% betragen. (Wieso ist keine Angabe zu p\_l gegeben?)

import random  
  
 p\_r = ...  
  
 Ort = [0]  
  
 *(Schlaufe)*

r = random.random()  
 if r >= p\_r:

*(Neuen Wert in Ort[] einfüllen)*  
 else:  
 *(Neuen Wert in Ort[] einfüllen)*

Falls Du die Aufgabe korrekt ausgeführt hast, sollte Deine Ausgabe (bei Eingabe von Ort und *Enter* im Terminal) ähnlich wie folgender Output aussehen:

**Ort** => [0, -1, -2, -1, 0, -1, -2, -3, -2, -1, 0, 1, 0, -1, 0, -1, -2, -3, -2, -1, -2, -3, -2, -1, 0, 1, 2, 1, 0, -1, -2, -1, 0, -1, -2, -3, -4, -5, -4, -5, -6, -5, -4, -5, -6, -7, -8, -7, -6, -7, -8, -9, -8, -7, -6, -5, -6, -7, -6, -5, -6, -7, -8, -9, -8, -9, -8, -7, -8, -7, -6, -5, -6, -7, -8, -9, -8, -7, -8, -7, -6, -5, -6, -5, -4, -5, -6, -7, -6, -5, -6, -7, -6, -7, -6, -5, -6, -5, -6, -5]

**A26**. Die letzte Zahl (-5) gibt an, wo wir uns nach 100 Schritten befinden. Mit Ort[i] fragen wir das i-1-te Element der Liste Ort ab. Wenn len(Ort) die Länge der Liste Ort ist, wie könntest Du mit Hilfe der Länge das letzte Element abfragen?

**Gelernt**

Du kannst einen Weg von beliebiger Länge generieren und das letzte Element einer Liste ausgeben.

**5. Visualisierung**

Um Grafiken zu generieren, musst Du zuerst die Bibliothek *mathlibplot.pyplot[[6]](#footnote-6)* importieren.

import matplotlib as mpl

mpl.use('Agg')

import matplotlib.pyplot as plt

Füge ein neues File in das Projekt auf repl.it ein und nenne es 'graph.png'.

Füge dann untenstehenden Codeblock unterhalb Deines Programmes ein und lass es laufen. Nach einiger Zeit (es kann bis zu einer Minute dauern), erkennst Du oben über dem Editor einen zweiten Tab mit dem Namen *'random\_walk1.png'*.

plt.plot(Ort)

plt.show(Ort)

plt.savefig('graph.png')

**A27**. Erkennst Du, was auf der Grafik dargestellt wird? Kannst Du die Achsen benennen?

**6. Simulationen**

**A28**. Was erwartest Du für die Enddistanz, wenn die Wahrscheinlichkeit nach links zu wandern 20% und die Wahrscheinlichkeit nach rechts zu wandern 80% beträgt?

**A29**. Programmiere Deinen Code um, so dass er diese Wahrscheinlichkeiten abbildet. Überprüfe Deine Idee, indem Du die Simulation 10x durchführst.

**A30**. Was erwartest Du für die Enddistanz, wenn Du doppelt so viele Schritte machst? (Wieder mit 50%)

**A31**. Führe die Simulation je 5x mit 1000 Schritten, 5x mit 2000 Schritten und 5x mit 4000 Schritten durch. Zeichne von Hand den dazugehörigen Graphen (Schritte vs. Distanz). Was stellst Du für die durchschnittliche Distanz vom Ursprung fest? Versuche, den gefundenen Zusammenhang mit 5 Simulationen mit je 10000 Schritten zu festigen.

**A32**. Wie lange würde es demnach dauern, bis Du von der Schule bis nach hause (3000m) gefallen bist, wenn Du pro Sekunde einen Meter umfällst?

**7. Durchschnittswerte**

Statt die Simulationen von Hand einzeln auszuführen, können wir uns an Kapitel*3.2 Ausführen von vielen Schritten desselben Mechanismus*zurückerinnern.

**A32**. Schreibe ein Programm, dass eine Anzahl Simulationen (z.B. numb\_sim) und die Anzahl Schritte (z.B. steps) als Eingabe hat und nach erfolgreichem Durchlauf die durchschnittliche Distanz vom Startpunkt ausgibt.

**8. Erweiterungen**

Möglich Erweiterungen sind:

* Simulationen in zwei bzw. drei Dimensionen
* Pfade unterschiedlich einzufärben
* Nicht nur ganz-zahlige Schritte, sondern gemäss einer Verteilung
* Darstellung der durchschnittlichen Distanz in Abhängigkeit der Anzahl Schritte in einer Grafik
* ... und viele mehr...

1. Bildquelle: http://nilesjohnson.net/teaching/rand\_walk.png [↑](#footnote-ref-1)
2. Diese Vorgehensweise ist weit verbreitet in der Physik: wir nehmen ein komplexes System, vereinfachen es so weit, dass wir es modellieren können und fügen die komplexen Elemente zu einem späteren Zeitpunkt wieder hinzu. [↑](#footnote-ref-2)
3. Du kannst auch direkt im Terminal die Zuweisung machen. Der Sinn des Editors ist es, ein 'Drehbuch' zu schreiben, welches der Computer dann abarbeitet. Sonst darfst Du alle Zuweisungen jedes einzelne Mal wieder eintippen... [↑](#footnote-ref-3)
4. Ausser man misst z.B. die Zerfallsrate eines radioaktiven Nuklids mit dem Computer. [↑](#footnote-ref-4)
5. 'Gute' Zufallsgeneratoren sind schwer zu programmieren und oft einfach zu hacken. [↑](#footnote-ref-5)
6. Du kannst den Graphen auf unendliche Weise anpassen: Achsen beschriften, skalieren, einfärben, ... Such einfach nach *Pyplot Library* und schau Dir die Beschreibung und Beispiele der Bibliothek an. [↑](#footnote-ref-6)