PHBern

Vertiefungsarbeit

Simulierte Physik

Entwicklung von Programmiermodulen im Physikunterricht

Autor:	**Thomas Bisig** Hönggerstrasse 27 8037 Zürich T.: 078 725 85 31 E-Mail: tbisig@gmail.com
Matrikelnummer:	01-924-406
Gutachter:	Prof. Dr. Marc Eyer
Abgabetermin:	25. Juni 2018

Kurzfassung

Die vorliegende Arbeit beschreibt die Entwicklung und die dabei entstandenen Produkte einer Einführung in das Programmieren im Kontext des Physikunterrichts. Die zwei Halbtagesmodule sind als Partnerarbeit konzipiert und richten sich an GYM3/4 Klassen, welche die Themen *Wärme* und *Elektrostatik* im Physikunterricht behandeln. Das Ziel des interdisziplinären Inputs ist das Lesen und Verstehen von bestehendem Code und Codestrukturen, das Schreiben erster Codezeilen, das Abstrahieren von physikalisch-mathematischen Problemstellungen und das Erkennen der Sinnhaftigkeit des Einsatzes von Computern.

Diese Module wurden am Campus Muristalden im Sommersemester 2018 mit einer GYM3 Klasse durchgeführt. Alle Produkte (Dokumentation und Code) stehen frei (unter der MIT License[i]) unter https://github.com/ThomasBisig/Physik zur Verfügung.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Interdisziplinarität (auch: fächerübergreifende Kompetenzen) wird seit einigen Jahren im Schweizer Schulunterricht gefordert und gefördert[ii]. Fachschaften an Gymnasien werden dazu ermutigt, in speziellen Lektionen ihr Fachgebiet mit anderen Fachgebieten sinnvoll zu vernetzen, um so den Schülerinnen und Schülern (SuS) eine integrale Sicht auf den Schulstoff vermitteln zu können. Das Gymnasium Campus Muristalden[iii] formuliert die Vermittlung der überfachlichen Kompetenzen in einem Konzeptpapier *Konzept Überfachliche Kompetenzen Gymnasium Muristalden*[iv].

Durch diese Entwicklung motiviert, wurde eine für zwei Halbtage durch die SuS selbstständig zu bearbeitende Dokumentation (im Folgenden *Module* bzw. *Modul I* und *Modul II* genannt) konzipiert, welche den SuS im gymnasialen Physikunterricht einen Einstieg in das Programmieren geben sollen. Computerunterstützte Berechnungen und Simulationen werden auch an anderen Gymnasien oder Institutionen eingesetzt, z.B. im Fachbereich Mathematik[v], in Simulationen im Fachbereich der Biologie oder der Geographie[vi] oder als Mix verschiedener Fachbereiche sowie Vertiefungen[vii].

Die angestrebten Ziele dieser Einführung ins Programmieren sind:

- Den SuS den ersten Schritt ins Programmieren durch das Einbinden in ein bereits bekanntes Fach zu vereinfachen.
- Den SuS durch diese Einbindung aufzuzeigen, dass eine computerunterstützte Simulation oder Berechnung sinnvoll sein kann.
- Den SuS die grundlegenden Strukturen eines Codes und das Lesen desselben aufzuzeigen.
- Den SuS aufzuzeigen, wie sie physikalische Problemstellungen abstrahieren können, um diese Abstraktion in der Folge zu mathematisieren und letztlich zu programmieren.
- Visualisierungen zu erstellen, mithilfe derer physikalische Systeme aufgezeigt und erklärt werden können.
- Im Internet nach technischen Antworten zu suchen und die gelieferten Antworten für die eigene Lösung zu verwenden.

Das Erlernte wird im Unterricht anhand von Multiple-Choice-Fragen und Prüfungsfragen überprüft. Das Feedback der SuS wird durch einen Umfragebogen eingeholt und ausgewertet.

Die vorliegende Arbeit beginnt im Kapitel 2 mit dem Ursprungskonzept und erklärt die groben Ideen und Überlegungen vor der Erarbeitung der Module. Kapitel 3 behandelt den Inhalt der Module und geht dabei bereits auf einige Details ein, Kapitel 4 fasst die Auswertung durch Prüfungs- und Kontrollfragen, sowie direktes Feedback zusammen. Im Kapitel 5 werden die Erfahrungen zusammengefasst, Verbesserungen diskutiert und auf die Zielerreichung eingegangen.

2 Konzeption und Idee

Das Grobkonzept besteht darin eine im Physikunterricht bereits erarbeitete (mathematische und physikalische) Grundlage, mithilfe von Computersimulationen zu erweitern. Als Inhalt dient im Thema *Wärme* die Generierung einer Brown'schen Bewegung (Abbildung 1) und im Thema der *Elektrostatik* die Visualisierung des elektrostatischen Feldes einer beliebigen Ladungsverteilung (Abbildung 2).

2.1 Abgedeckte Themengebiete und genutzte Tools

Als Programmiersprache wird Python[viii] genutzt und um eine computersystemunabhängige Durchführung zu ermöglichen, wird der Code auf einer Onlineplattform[ix] ausgeführt.

Programmierthemen:	Variablen, Listen, Zuweisungen, mathematische Operationen, Inkrementierungen, Schlaufen (while, for), Steuerungsstrukturen (if... else...), Zufall, Bibliotheken, Konsolen und graphische Ausgaben, Funktionen, Konstanten
Mathematische Themen:	Grundoperationen, Modulo, Winkelfunktionen, Pythagoras, Vektorgeometrie, Zufall, Abstandsberechnung, Diagramme
Physikalische Themen:	Brown'sche Bewegung, (elektrische) Felder, Ladungen, Coulombgesetz, Kräfte und Kräfteaddition
Weiteres:	Vereinfachungen, Abstrahierungen, Verallgemeinerungen, Internetsuche

2.2 Konzept der Module

Die Idee des ersten Moduls, welches sich über drei (oder mehr) Lektionen erstreckt, besteht darin, an einem zufallsgenerierten, eindimensionalen Nachhauseweges die ersten Programmierschritte (Zuweisung von Variablen, mathematische Operationen, Schlaufen, Listen, Zufall und Bibliotheken) kennenzulernen. Der Nachhauseweg soll beispielhaft für die Zitterbewegungen von Pollen stehen, in deren Ursprung die Entdeckung der Brown'schen Bewegung liegt.

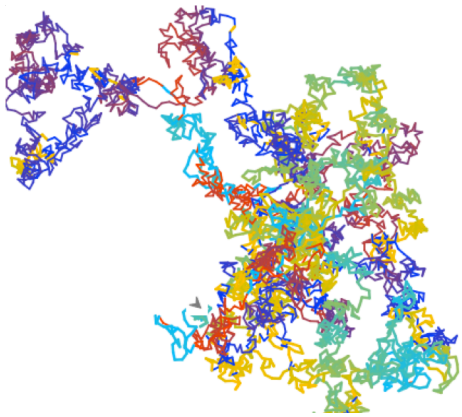


Abbildung 1: Brown'sche Bewegung

Die SuS generieren zuerst mit mehreren Münzwürfen einen Zufallsprozess und zeichnen diesen (Kopf: ein Schritt nach links, Zahl: ein Schritt nach rechts). Dabei werden die SuS zur Frage geführt, wieviele Münzwürfe es im Durchschnitt braucht, bis ein Nachhauseweg von z.B. 1000 Metern zurückgelegt wird. Mit dieser Frage im Kopf begeben sich die SuS an die Computer, wo sie mithilfe der Online Plattform *repl.it* Code im Browser

eingeben und ausführen. Somit kann dieses Modul auf jedem Schulcomputer durchgeführt werden[1]. Durch einfache Beispiele und direktes Ausprobieren wird der langsame Aufbau zu einem voll funktionsfähigen Zufallsweggenerator erarbeitet.

Dieses erste Modul beinhaltet insgesammt 32 Aufgaben, die die SuS zu zweit lösen.

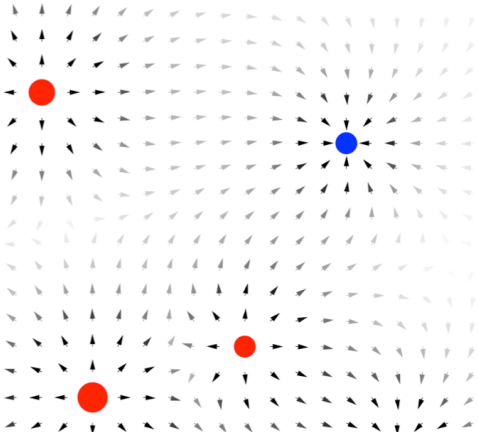


Abbildung 2: Elektrostatisches Feld

Die Idee des zweiten Moduls, welches sich über zwei (oder mehr) Lektionen erstreckt, besteht darin, für eine vorgegebene Ladungskonfiguration das elektrostatische Feld zu berechnen und zu visualisieren. Zusätzlich zu einem Input zur Vektorrechnung und einem Input über Vereinfachungen in der Physik, besteht dieses Modul hauptsächlich im Verstehen eines vorgegebenen Codes (der bereits läuft) und im korrekten Abändern von Codeblöcken. Den SuS wird ein funktionierender Code vorgelegt, welcher für eine beliebige Ladungskonfiguration die Feldvektoren des elektrostatischen Feldes darstellen soll. Sobald die SuS diesen laufen lassen, stellen sie fest, dass die Feldvektoren zufällig ("durcheinander") generiert sind (siehe Abbildung 3). Ihre Aufgabe besteht darin, den Code so anzupassen, dass die Feldvektoren richtig berechnet und dargestellt werden. Durch Abstraktion des Problems (Reduktion auf eine einzelne Quellladung) wird es möglich, an jedem Ort im Raum die Komponenten des Feldvektors der Quellladung in zwei Dimensionen zu berechnen. Dabei werden Themen wie Pythagoras, Umkehrfunktion des Tangens sowie die komponentenweise Addition von Vektoren behandelt. Am Ende des Moduls schreiben die SuS mithilfe einer Schlaufenstruktur den Algorithmus so um, dass eine Generalisierung auf beliebig viele Quellladungen entsteht.

Dieses zweite Modul beinhaltet insgesammt 22 Aufgaben, die die SuS zu zweit lösen.

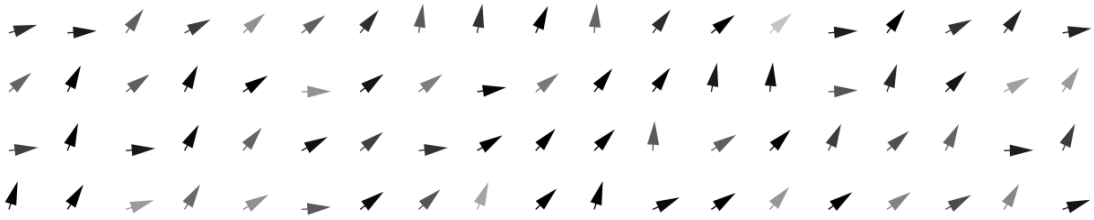


Abbildung 3: Zufällig generierte Feldvektoren

[1] Wobei einige alte Browser gewisse JS Komponenten nicht oder fehlerhaft unterstützen. Empfohlen wird Firefox

3 Produkt[2]

Die Produkte bestehen in beiden Modulen aus einer Dokumentation (siehe Anhang) und vorgefertigten Codestrukturen (siehe github-Repository).

3.1 Modul I: Der zufällige Nachhauseweg

3.1.1 Inhalt

Das Ziel dieses Moduls ist es, die Grundelemente einer Programmiersprache und deren Syntax anhand der Brown'schen Bewegung zu verstehen und umzusetzen.

Im ersten Modul werden die nachfolgenden Themengebiete abgedeckt:

Programmierthemen:	Variablen, Listen, Zuweisungen, mathematische Operationen, Inkrementierungen, Schlaufen (while, for), Steuerungsstrukturen (if... else...), Zufallfunktion, Bibliotheken, Konsolen- und graphische Ausgaben
Mathematische Themen:	Grundoperationen, Modulo, Zufall, Diagramme, Koordinaten
Physikalische Themen:	Brown'sche Bewegung
Weiteres:	Internetsuche

3.1.2 Aufbau

Die SuS werden in der Einführung an die bereits bekannte Beobachtung der Brown'schen Bewegung wie folgt herangeführt (Seite 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?

Diese Einführung soll den SuS einen neuen Blickwinkel darauf geben, für was ein Zufallsprozess, wie die Brown'sche Bewegung, genutzt werden kann.

Um das Problem in sinnvoller Zeit auf den Computer übertragen zu können, entdecken die SuS in *Kapitel 1* die notwendigen *Vereinfachungen* (gekürzt, Seite 3):

> Die erste Vereinfachung besteht darin, dass wir uns zuerst auf eine Dimension beschränken. D.h. 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.

[2] alle Blockzitate finden sich in vollständiger Form in der Dokumentation.

> 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.

Die Grundidee und die Vereinfachungen sind somit geklärt. Jedoch verstehen die SuS manchmal nicht, wieso ein Computer sehr hilfreich sein kann, um die Fragestellung zu klären. Aus diesem Grund sollen die SuS in *Kapitel 2 Von Hand Programmieren* (gekürzt, Seite 4):

> Nimm zwei Münzen und lege die erste Münze in die 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.
>
> [...]
>
> Wie könntest du den Weg in [der vorherigen Aufgabe] mit Hilfe eines Blattes und eines Stiftes noch besser veranschaulichen?

Im Anschluss an das manuelle Programmieren, lernen die SuS in *Kapitel 3 Konzepte des Programmierens* zuerst die Programmierumgebung *repl.it* kennen (was ist der Editor, wie führe ich ein Programm aus, wo sehe ich die Ausgabe des Programmes, etc.), um danach Schritt für Schritt den Algorithmus der Brown'schen Bewegung selber zu erarbeiten. Die einzelnen Schritte sind im Folgenden kurz beschrieben.

Um einen Weg bzw. die Positionen eines Weges (eines Pollens, eines Nachhauseweges, einer Münze) später darstellen oder analysieren zu können, lernen die SuS das Speichern von Zahlen kennen. Um eine Liste von Positionen zu speichern, lernen die SuS zuerst, wie eine Variablen- und dann eine Listenzuweisung funktioniert.

Der Code ist in der Dokumentation zur Erhöhung der Lesbarkeit in grauen Kästchen eingebunden und in einer Festbreitenschrift geschrieben. Nachfolgen wird beispielhaft ein 'Weg' der ersten 7 Quadratzahlen in einer sogenannten Liste *p* gespeichert (Seite 8):

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

Nachdem die SuS Listen erstellen, erweitern und ausgeben können, werden sie darauf aufmerksam gemacht, dass der Computer den repetitiven Teil der Arbeit übernehmen könnte: die Generierung eines Zufallsweges erfordert das wiederholte Ausführen eines (zufälligen) Schrittes nach links bzw. nach rechts und das Speichern des neu 'generierten' Ortes. Nachdem die SuS die Position des Objektes speichern können, lernen sie dazu die erste Schlaufe kennen (Seite 9):

> 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.

Die SuS schreiben am Ende dieses Unterkapitels einen Algorithmus, welcher alle ungeraden Zahlen von 0 bis 50 ausgibt. Dabei lernen sie neben der Nutzung von Schlaufen auch, dass das Inkrement (Schrittgrösse) in einer Schlaufe nicht nur *eins* betragen kann.

Da jedoch die Schlaufe bis anhin nur in eine Richtung geht (Münze springt immer nach rechts), entdecken die SuS im Kapitel *Der Zufall kommt ins Spiel* die *if*-Bedingung und einen (Psuedo)Zufallsgenerator (Seite 14).

> 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 den jeweiligen Schritt nach links oder nach rechts fallen.
>
> [...]

```
import random
for i in range(0,10):
   print(random.random())
```

Mit all diesen Werkzeugen können die SuS letztlich in *Kapitel 4 Die Brown'sche Bewegung* Ausgaben wie die Folgende generieren (Seite 15):

```
   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]
```

Die *Visualisierungen* (exemplarisch in Abbildung 4) in *Kapitel 5* zeigen den eindimensionalen Weg auf, den die SuS generiert haben und schliessen den Pflichtteil damit ab.

Für ambitionierte und/oder schnelle SuS sind im *Kapitel 6 Simulationen*, *Kapitel 7 Durchschnittswerte* und *Kapitel 8 Erweiterungen* vertiefende Aufgaben aufgelistet.

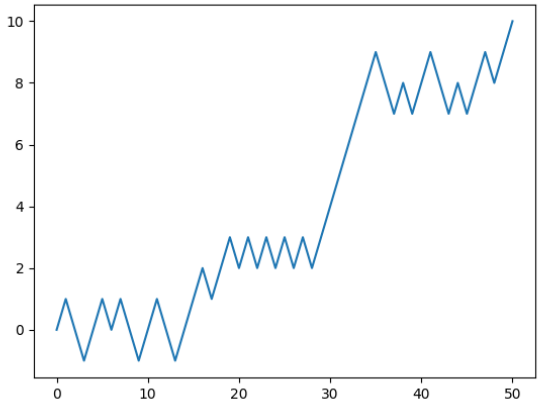


Abbildung 4: Zufallsweg in 50 Schritten

3.2 Modul II: Elektrische Felder beliebiger Ladungsverteilungen

3.2.1 Inhalt

Das Ziel dieses Moduls ist am Beispiel der Visualisierung eines elektrischen Feldes, das erarbeitete Programmierwissen einzusetzen und zu erweitern, Code zu lesen und verstehen zu lernen, mathematische Konzepte in Code zu übersetzen und graphischen Output zu interpretieren.

Im zweiten Modul werden die nachfolgenden Themengebiete abgedeckt:

Programmierthemen:	alle Themen aus Modul I, Funktionen, Konstanten
Mathematische Themen:	Winkelfunktionen, Pythagoras, Vektorgeometrie, Abstandsberechnung, Diagramme
Physikalische Themen:	(elektrische) Felder, Ladungen, Coulombgesetz, Kräfte und Kräfteaddition
Weiteres:	Vereinfachungen, Abstrahierungen, Verallgemeinerungen, Internetsuche

3.2.2 Aufbau

Vor dem Hintergrund der Erfahrungen aus dem ersten Modul, war für die Entwicklung des zweiten Moduls klar, dass der graphische Output an erster Stelle stehen musste (siehe *Kapitel 6 Diskussion*). Deswegen werden die SuS im zweiten Modul direkt nach der Einführung aufgefordert, einen bereits existierenden Code laufen zu lassen (Seite 1/2):

> Im Unterricht hast du qualitativ einige elektrische Felder verschiedener Ladungskonfigurationen gesehen. Qualitativ deshalb, weil es von Hand sehr aufwendig ist, das elektrische Feld von Ladungskonfigurationen von mehr als einer Ladung zu berechnen. In diesem Modul sollst du einen vorgefertigten Computercode so anpassen, dass er diese rechenintensive Aufgabe für (fast) jede beliebige Ladungskonfiguration berechnet.
>
> Geh zuerst auf https://repl.it/@ThomasBisig/Electric-Field-Lines und lass den Code laufen ('Run' drücken). Wenn der Code fertig ausgeführt wurde, schau dir das generierte Bild 'field_vectors.png' an.

Der zweite Unterschied zum ersten Modul besteht darin, dass ein bereits bestehender Code abgeändert, korrigiert und erweitert werden muss.

In einem ersten Schritt frischen die SuS ihr Wissen über das Coulombgesetz sowie über die Addition von Vektoren auf. Damit sind die Grundsteine gelegt, das elektrische Feld zu berechnen bzw. die vielen elektrischen Feldkomponenten zu addieren.

Im *Kapitel 3 Reduktion auf eine Ladung* wird den SuS nahegelegt, zuerst das Problem nur für eine einzelne Ladung zu betrachten. Die Überlegungen werden mithilfe einer Skizze gemacht (Seite 5/6):

Zeichne zum untenstehenden Bild Achsen und beschrifte folgende Dinge:

- die Ladung c (für *charge*) sei durch den ungefüllten Kreis dargestellt und der Punkt im Raum, welcher uns interessiert, mit p (für *point*),
- die Abstände in *x*- und *y*-Richtung (schau im Code nach, wie sie beschrieben werden),
- den absoluten Abstand von c und p (im Code zu finden),
- sowie den Feld-Vektor f (wenn *Q* eine positive Ladung ist) in Komponenten zerlegt (die wiederum im Code zu finden sind).

Wie bereits in vorhergehenden Aufgaben, lernen die SuS im Code nachzusehen, wie welche Grössen genannt werden. Dadurch lernen sie sich in bestehendem Code zu orientieren und nach hilfreichen Angaben zu suchen. Darüberhinaus wird offensichtlich, wie wertvoll sinnvolle Listen- und Variablennamen sind.

Die elektrischen Feldvektoren sind bis anhin zufällig generiert. Die Aufgabe der SuS besteht darin, diesen Teil neu zu programmieren, so dass die korrekten Komponenten des elektrischen Feldvektors berechnet werden.

Zuerst werden die SuS aufgefordert, anhand der Skizze die Komponenten des Feldvektors zu bestimmen. Die Hauptüberlegungen die sie sich dabei machen, sind:

- der elektrische Feldvektor zeigt in Richtung der Verbindungslinie zwischen Quell- und Probeladung
- diese Richtung bestimmt einen Winkel
- dieser Winkel kann wieder genutzt werden, um die Komponenten des Vektors zu bestimmen.

Die gesuchte Winkelfunktion (*Arcustangens* bzw. in der *Math-Library* von *Python atan2()* genannt) kann mithilfe einer Suchmaschine oder der Formelsammlung gefunden werden (Seite 6/7):

> Welche zwei Grössen in Kombination mit welcher Winkelfunktion geben somit den Winkel an, mit welchem der Feldvektor f von p weg/hinzeigt? Suche im Internet nach der Funktion in Kombination mit "*Python Math Library*" und lass Dein Resultat durch die Lehrperson prüfen. Weise der Variablen angle auf Zeile 58 das Resultat der Winkelfunktion (= einen Winkel) zu.

Nachdem die Berechnung für den vereinfachten Fall einer einzelnen Ladung durch die SuS umgesetzt wurde, führt *Kapitel 4 Verallgemeinerung auf mehrere Ladungen* die SuS zur Idee der Addition von Feldvektoren (Seite 8).

> Wie du aus der Vektorgeometrie weisst, kann man Vektoren komponentenweise addieren. D.h. das 'Aneinanderhängen' von Vektoren (grafisch), wird rechnerisch über das Addieren der einzelnen Komponenten gelöst. Genau das wollen wir auch in unserem Beispiel anwenden.

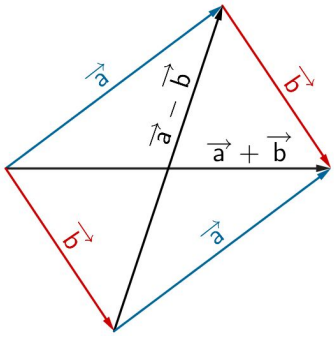


Abbildung 5: Vektoraddition

Die Änderung, welche im Code erforderlich ist, ist minimal, die Überlegungen und das notwendige Verständnis des Codes jedoch alles andere als trivial.

Zum Abschluss sollen die SuS einige beliebige Ladungskonfigurationen vorgeben und die jeweiligen Visualisierungen generieren. Manche SuS stossen dabei an die Grenzen der Umsetzung (Skalierung, Abmessung des Plots, etc).

Als Herausforderung für ambitionierte SuS erlaubt die Abschlussfrage *Kannst du das Beispiel so umprogrammieren, dass die Länge der Pfeile der Stärke der Pfeile entspricht?* ein tieferes Eintauchen in den Code und die Syntax.

4 Auswertung

4.1 Prüfungs- und Kontrollfragen

4.1.1 Modul I

In der Prüfungsfrage zum ersten Modul geht es um das Lesen und Verstehen eines Codeblocks, welcher unterschiedliche Steuerungsstrukturen miteinander verbindet. Die Aufgabe bezieht sich nicht auf das physikalische Problem der Wärme, sondern auf das Grundverständnis einer Codestruktur.

Schreibe auf, was folgender Codeblock für (eine) Ausgabe(n) generiert.

```
i = 5%2
print(i)
while i < 15:
   i = 3*i - 1
   if i < 5:
      print('i<5')
   else:
      print('i>=5')
```

Zeile zwei soll falsche Moduloberechnungen in Zeile eins zeigen, so dass die Schlaufe mit Folgefehler bewertet werden kann.

Auswertung: Von 1.5 Punkten wurden im Schnitt 0.6 (40%) Punkte erreicht, wobei nur eine Person die vollständig korrekte Ausgabe notierte.

4.1.2 Modul II

In der Prüfungsfrage zum zweiten Modul geht es darum, den erarbeiteten Teilalgorithmus wieder zu erkennen und diesen zu korrigieren und zu erklären:

Unten ist der Teil des Codes abgebildet, welchen Du an einem Nachmittag selber erarbeitet hast (zumindest Zeilen 10 bis 17). Wir haben damit das E-Feld einer beliebigen Ladungsverteilung gezeichnet.

a) Finde den Fehler im Code in Zeile 13 [0.5]

b) Erkläre mithilfe einer Skizze, was in Zeilen 16/17 berechnet wird. [1]

c) Nenne einen Grund, warum im vorliegenden Fall (Feld einer beliebigen Ladungsverteilung) die Zuhilfenahme eines Computers sinnvoll ist. [0.5]

```
[1]  def getEFieldComponents(ch,p):
[2]
[3]    field_x = 0.0
[4]    field_y = 0.0
[5]
[6]    # loop over all charges c[x_pos, y_pos, charge]
[7]    for c in ch:
[8]
[9]      ####### -- for the students -- #######
[10]     d_x = p[0] - c[0]
[11]     d_y = p[1] - c[1]
[12]     r = sqrt(d_x**2+d_y**2)
[13]     f = 1/(4*pi*epsilon_0)*c[2]/r*2
[14]     angle = atan2(d_y,d_x)
[15]
[16]     field_x += f*cos(angle)
[17]     field_y += f*sin(angle)
[18]
[19]     ####### ---------------------- #######
[20]   return [field_x, field_y]
```

Auswertung: Von 2 Punkten wurden im Schnitt 0.6 Punkte (30%) erreicht, wobei keine Person die Aufgabe korrekt lösen konnte. Die Punktzahlen reichten von 0 (5x) bis 1.75 (1x) Punkten. Aufgabenteil *c* wurde von fast allen korrekt gelöst (0.35 von 0.5 Punkten).

4.1.3 Kontrollfragen

Die Kontrollfragen werden mithilfe von Kahoot^{x} in die Klasse gebracht. Das Quiz besteht aus acht Fragen mit jeweils vier möglichen Antworten. Die SuS wählen die jeweils richtige Antwort. 43% der Antworten waren korrekt, wobei die Spannweite von 11% bis 83% korrekten Anworten reichte. Fragen, welche Zusammenhänge von mathematischen Konzepten und Programmiercode verbanden, erreichten die tiefsten Prozentsätze.

4.2 Feedback der SuS

Die Auswertung der Fragebogen ergab folgende Mittelwerte (*Avg*) und Standardabweichungen (*Std*), wobei die Bewertung von 0 (keine Zustimmung) bis 4 (volle Zustimmung) reichte. Das Feedback wurde anonym durchgeführt:

	Frage	Avg	Std
1	Ich habe einen Einblick in die Welt des Programmierens erhalten	2.2	1.3
2	Ich habe in diesem Modul von den grundlegenden Programmiereinheiten gehört (Zuweisungen, Operationen, Schlaufen, Funktionen, Bibliotheken, ...)	2.4	1.1
3	Ich verstehe, wieso es sinnvoll sein kann, Computer als Hilfsmittel zu verwenden	3.5	0.8
4	Ich konnte mein Wissen über das Programmieren erweitern	1.5	1.2
5	Es hat mir Spass gemacht	1.6	1.3
6	Ich kann mir vorstellen, mich (irgendwann) mehr mit der Materie zu beschäftigen	1.4	1.3
7	Die Dokumentation war klar geschrieben und es war ein Aufbau erkennbar	2.8	1.1
8	Die Lehrperson hat mich (bei Bedarf) gut unterstützt	3.7	0.8

Abbildung 6: Feedback der SuS

5 Diskussion

5.1 Zielerreichung

Die Erarbeitung, Durchführung und Überprüfung haben klar aufgezeigt, welche der ursprünglich angedachten Ziele aus *Kapitel 1* erreicht wurden und welche nicht.

- Den SuS den ersten Schritt ins Programmieren durch das Einbinden in ein bereits bekanntes Fach zu vereinfachen.

Der erste Schritt ins Programmieren konnte den SuS ermöglicht werden. Vereinfacht wurde es den SuS nur zum Teil, da manchmal das Wissen der Materie nicht genug verankert war oder die Motivation, sich in eine neues Thema einzuarbeiten, zu klein war.

- Den SuS durch diese Einbindung aufzuzeigen, dass eine computerunterstützte Simulation oder Berechnung sinnvoll sein kann.

Laut dem Feedback der SuS wurde dieser Punkt erfüllt. Auch die Antworten zur zweiten Prüfungsfrage Teil b) deuten auf das Verständnis der Sinnhaftigkeit hin.

- Den SuS die grundlegenden Strukturen eines Codes und das Lesen desselben aufzuzeigen.

Obwohl Frage zwei und Frage vier in Abbildung 6 auf ähnliche Antworten abzielen, war das Feedback sehr unterschiedlich. Subjektiv beurteilt konnte ein Grossteil der SuS nach den zwei Modulen einfachen Code verstehen und nachvollziehen.

- Den SuS aufzuzeigen, wie sie physikalische Problemstellungen abstrahieren können, um diese Abstraktion in der Folge zu mathematisieren und letztlich zu programmieren.

Diese Überlegungen wurden im zweiten Modul aufgezeigt und durch Rückfragen der Lehrperson bei den SuS bestätigt. Da diese Methode der rechengestützten Physik jedoch viel Übung braucht, kann nicht davon ausgegangen werden, dass die SuS dieses Wissen ohne Wiederholung behalten können oder auf neue Probleme anwenden können.

- Visualisierungen zu erstellen, mithilfe welcher physikalische Systeme aufgezeigt und erklärt werden können.

Vor allem im zweiten Teil wurde den SuS klar, wie der Computer zum Erstellen von Visualisierungen genutzt werden kann und wie dies zum besseren Verständnis beitragen kann.

- Im Internet nach technischen Antworten zu suchen und die gelieferten Antworten für die eigene Lösung umzusetzen.

Das Onlinesuch- und verstehverhalten der SuS war sehr unterschiedlich. Einige wussten sogleich, wie Antworten zu verstehen waren, während andere exakte Antworten nicht als solche erkannten.

5.2 Lessons Learned

Mehr Visualisierungen (Ziel: Motivation, Persistenz): Das erste Modul setzt den Fokus auf die Grundstrukturen des Programmierens. Bei der Durchführung wurde schnell klar, dass zu den ersten Schritten des Programmierens in der Physik ein direktes visuelles Feedback/Resultat gehört. Statt Ausgaben auf der Konsole zu interpretieren (wie in Modul I), sollte die Dokumentation so überarbeitet werden, dass in Modul I von Beginn weg eine Visualisierung generiert werden kann. Eine anfängliche Visualisierung wurde bei der Erarbeitung des zweiten Moduls berücksichtigt.

Spielerischer Einstieg (Ziel: Motivation, Hürden abbauen): Für viele der SuS ist es eine Herausforderung, neben den schon anspruchsvollen Themen der Physik noch ein zusätzliches Abstraktionslevel zu meistern. Als neues Modul 0 könnte eine spielerische Einführung ins Programmieren mithilfe einer Sprache wie Scratch[xi] dienen. Die SuS müssen so keinen eigenen Code schreiben und werden nicht abgeschreckt durch den vielen "Text", sondern können per Drag-n-Drop einfache Algorithmen nachbauen.

Internetsuche (Ziel: Recherche im Internet): Viele SuS wissen zwar, wie sie im Internet navigieren können, haben jedoch bisher keine Methodik entwickelt, wie sie optimiert Suchen und wie sie Antworten (Suchresultate) lesen können, um eigene Probleme zu lösen. Diese Kompetenz kann in allen Fächern am Gymnasium (und davor) geschult werden.

5.3 Weitere Beobachtungen

Also notwendige Voraussetzung kristallisierte sich klar heraus, dass die Durchführung das volle Engagement der Lehrperson braucht. Diese benötigt demnach eine gewisse Sicherheit in der Thematik des Programmierens (zumindest was die Grundlagen angeht). Viele Unklarheiten können von den SuS noch nicht selbstständig geklärt werden ("Suche nach der Antwort im Internet"), sondern benötigen ein Begleiten der Lehrperson.

Der unerwartet tiefe Wert von 1.6 Punkten von 4 möglichen Punkten in Frage 4 im Feedbackbogen ("*Ich konnte mein Wissen über das Programmieren erweitern"*) ist vermutlich darauf zurückzuführen, dass die SuS nicht merken, was und dass sie lernen, da das Lernen der Grundstrukturen der Programmierung, sobald man sie verstanden hat, als trivial erscheinen kann. Ein anderer Grund kann in der Formulierung der Frage liegen, da das Wort *Erweiterung* auf ein bestehendes Basiswissen abzielt.

5.4 Fazit

Die grösstenteils erfolgreiche Zielerreichung (*Kapitel 5.1*) ermutigt dazu, die Module zu verbessern (*Kapitel 5.2*) und in Folgeklassen wieder einzusetzen. Interessierte und versierte (*Kapitel 5.3*) Lehrkräfte können die Module aus dem Repository herunterladen, verändern und in eigenen Klassen einsetzen.

Anhänge

Alle Dokumente sind frei zugänglich unter https://github.com/ThomasBisig/Physik und beide Dokumentationen werden dieser Arbeit am Ende angehängt: Modul I (*Der zufällige Nachhauseweg*) und Modul II (*Elektrische Felder beliebiger Ladungsverteilungen*).

Verweise

[i] https://github.com/ThomasBisig/Physik/blob/master/LICENSE

[ii] https://files.eric.ed.gov/fulltext/EJ1101113.pdf

[iii] http://www.muristalden.ch

[iv] Kann beim Autor bezogen werden.

[v] https://www.ksh.edu/schule/faecher/mathematik.html

[vi] https://www.learner.org/courses/envsci/interactives/disease/index.php

[vii] http://www.python-exemplarisch.ch/

[viii] https://www.python.org

[ix] https://www. repl.it/

[x] https://create.kahoot.it/details/elektrostatik-4/9c95efdc-e521-4bb3-a316-e74da940d880

[xi] https://scratch.mit.edu

Institut Sekundarstufe II
Fabrikstrasse 8, CH-3012 Bern
T +41 31 309 21 15, contactdesk@phbern.ch, www.phbern.ch

PHBern
Pädagogische Hochschule

Lehrdiplom für Maturitätsschulen
Eigenständigkeitserklärung zur Vertiefungsarbeit

Ich bestätige hiermit, dass ich die Vertiefungsarbeit mit dem Titel

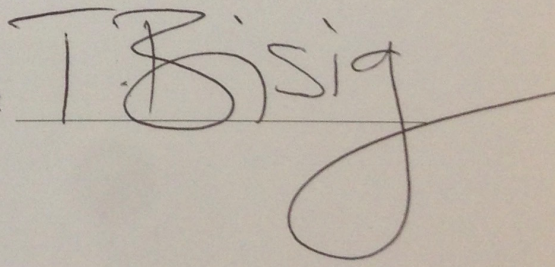
Simulierte Physik - Entwicklung von Programmiermodulen im Physikunterricht

unter der Leitung von

Prof. Dr. Marc Eyer

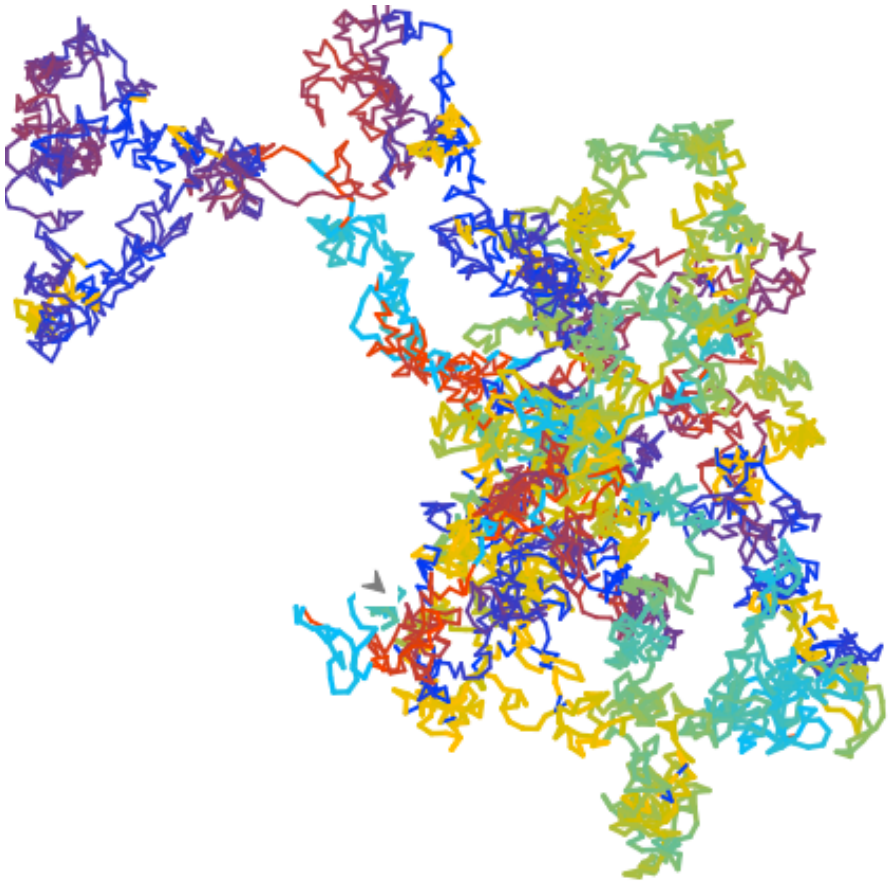
eigenständig und ohne unerlaubte Mittel verfasst habe.

Ich nehme zur Kenntnis, dass gemäss Artikel 59b der Verordnung vom 13. April 2005 über die deutschsprachige pädagogische Hochschule (PHV; BSG 436.911, Stand am 30.10.2013) ein Verstoss gegen den Grundsatz der Lauterkeit des wissenschaftlichen Arbeitens rechtliche und disziplinarische Konsequenzen nach sich ziehen kann.

Ort, Datum 31. Juni 2018 Unterschrift 

Brown'sche Bewegung als Simulation

Der zufällige Nachhauseweg[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

[1] Bildquelle: http://nilesjohnson.net/teaching/rand_walk.png

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]

[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.

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.

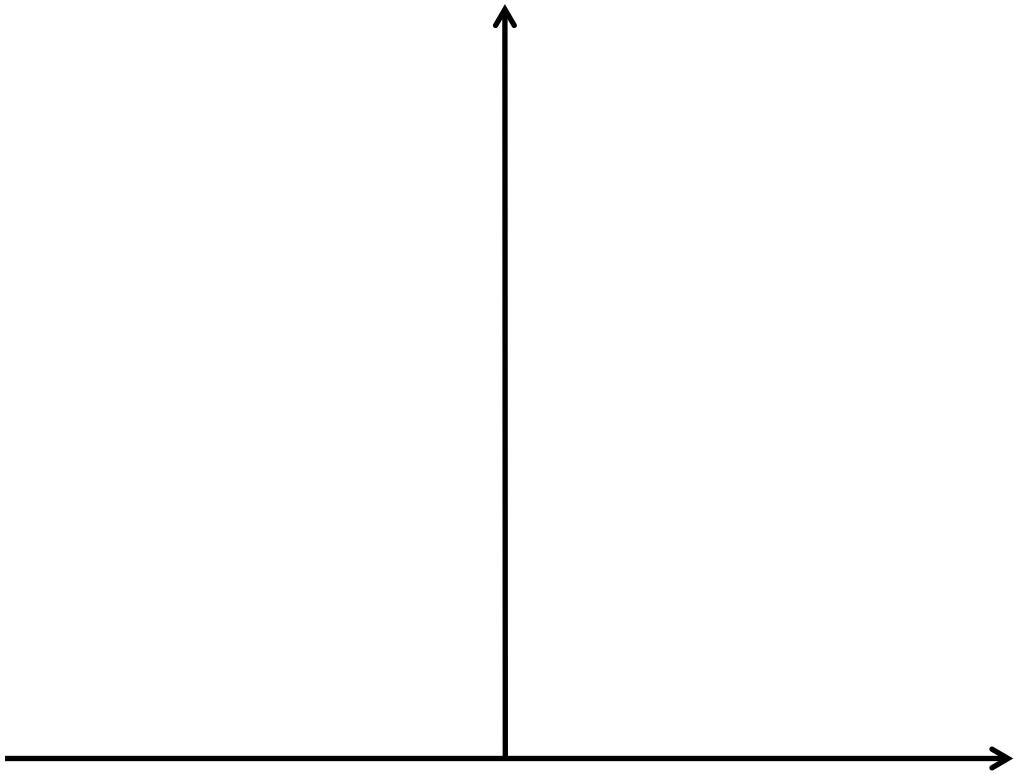
Du kannst den Fortschritt auf dem Nachhauseweg ohne Computer wie folgt veranschaulichen:

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]

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] 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...

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* (→I) 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]. 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]

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 zuoberst 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] Ausser man misst z.B. die Zerfallsrate eines radioaktiven Nuklids mit dem Computer.
[5] 'Gute' Zufallsgeneratoren sind schwer zu programmieren und oft einfach zu hacken.

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] 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] 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.

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.

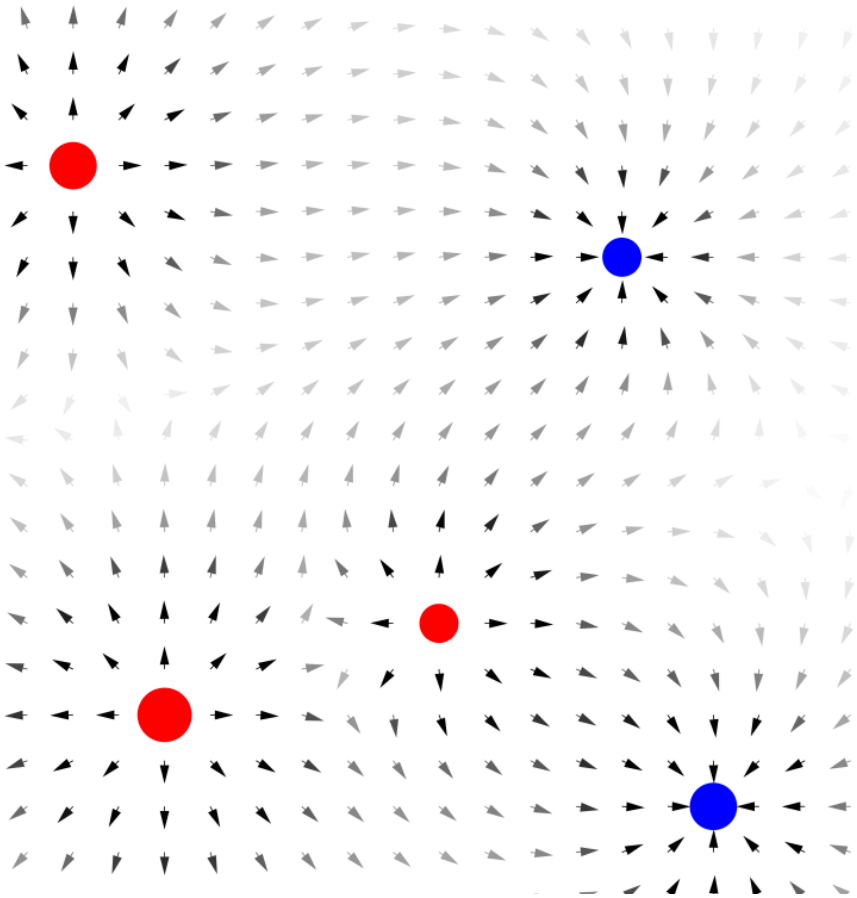
8. Erweiterungen

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...

Graphische Darstellung von elektrischen Feldern

Elektrische Felder beliebiger Ladungsverteilungen



Im Unterricht hast du qualitativ einige elektrische Felder verschiedener Ladungskonfigurationen gesehen. Qualitativ deshalb, weil es von Hand sehr aufwendig ist, das elektrische Feld von Ladungskonfigurationen von mehr als einer Ladung zu berechnen. In diesem Modul sollst du einen vorgefertigten Computercode so anpassen, dass er diese rechenintensive Aufgabe für (fast) jede beliebige Ladungskonfiguration berechnet.

Du lernst in diesem Modul

- Dich in einem bestehenden Code zu orientieren
- eine Fragestellung zu vereinfachen, zu lösen und wieder zu verallgemeinern
- eine Modellierung auf Papier in Code zu übersetzen
- Antworten im Internet zu finden
- Wissen aus der Mathematik praktisch anzuwenden

Geh zuerst auf https://repl.it/@ThomasBisig/Electric-Field-Lines und lass den Code laufen ('Run' drücken). Wenn der Code fertig ausgeführt wurde, schau dir das generierte Bild 'field_vectors.png' an.

A1. Was könnte auf diesem Bild dargestellt sein? Welche Teile erkennst Du? Was kann nicht korrekt sein?

1. Physikalische Grundlagen: Coulombgesetz

Aus dem Physikunterricht wissen wir, welche Kraft zwischen zwei Ladungen Q_1 und Q_2 besteht.

A2. Such in der Formelsammlung nach dem **Coulombgesetz** und notiere dir das Gesetz mit den zwei Ladungen Q_1 und Q_2 im Abstand r.

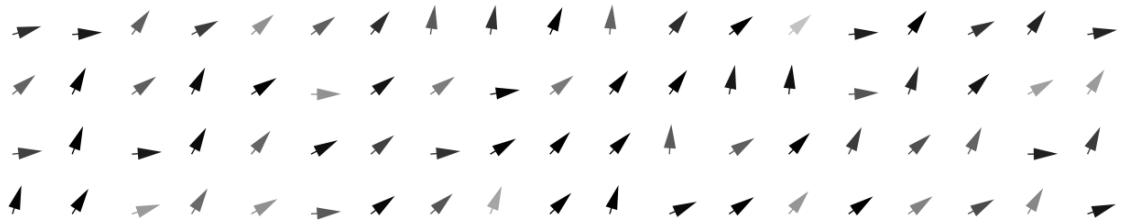
Wir haben im Unterricht gesehen, dass wir statt der Kraft einer Ladung Q_1 auf eine zweite Ladung Q_2 auch das sogenannte **elektrische Feld** einer (Quell)ladung Q betrachten können. Dieses elektrische Feld ergibt sich, wenn wir die Coulombkraft aufteilen. Sie gibt an, wie stark und in welche Richtung eine Probeladung q im Raum verschoben wird (hier haben wir stillschweigend Q_2 durch q ausgetauscht – weil q eine sehr kleine Ladung sein soll).

A3. Schau im Physikskript nach, wie wir das elektrische Feld definiert haben und notiere dir die Herleitung.

Stellen wir uns den Raum den wir betrachten wollen als zweidimensionale Fläche vor, dann können wir den Raum als *xy*-Koordinatensystem abstrahieren (die dritte Dimension kannst du als herausfordernde Aufgabe selber implementieren).

Gelernt
Du hast das Wissen über elektrische Felder und Kräfte repetiert und verstehst, wieso diese Konzepte zentral sind, um unser Visualisierungsproblem zu lösen.

2. Feldvektoren in der Physik



Elektrische Felder – wie auch Kräfte – werden als Vektoren beschrieben.

A4. Durch welche zwei Grössen ist ein Vektor bestimmt?

Die Stärke der Kraft wird normalerweise mit der Länge des Vektors dargestellt. In unserem Beispiel ist dies aber nicht sehr praktisch, da sich (so wie der Code programmiert ist) die Pfeile überschneiden würden.

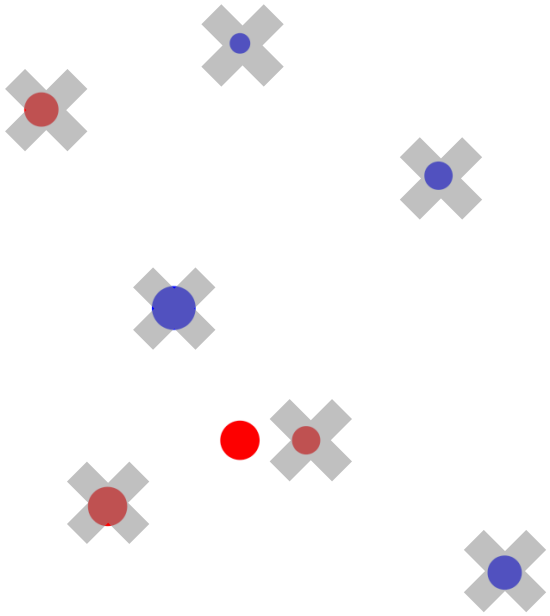
A5. Anstelle der Länge des Vektors, was wurde zur Darstellung der Stärke im Titelbild verwendet?

A6. Zeichne ein Koordinatensystem mit nur positiven x- und y-Achsen, beide Achsen skaliert von 0 – 10. Zeichne dann an einem beliebigen Punkt im Koordinatensystem eine Ladung positive Ladung ein (ein Punkt). Wähle zwei weitere Punkt und überlege dir qualitativ, wie die Vektoren des elektrischen Feldes aussehen müssten. Zeichne diese zwei Vektoren ein.

Gelernt
Du hast Dein Wissen über Vektoren aufgefrischt und kannst dieses mathematische Konzept auf das Beispiel der Kräfte zwischen zwei Ladungen anwenden.

3. Reduktion auf eine Ladung

Im Titelbild siehst du das elektrische Feld von unterschiedlichen Ladungen. Wie wir gesehen haben, wäre das von-Hand-rechnen für dieses Beispiel sehr aufwendig. Statt mit mehreren Ladungen, beginnen wir deshalb mit nur einer Ladung. Wenn wir diese Aufgabe gelöst haben (und überprüft haben, dass das Vektorfeld korrekt ist), können wir weitere Ladungen hinzufügen und uns überlegen, was wir anpassen müssen.



A7. Finde im Code die Zuweisung der Ladungen (Tipp: Englisch) und überlege Dir, wie die du die Angaben (d.h. so wie die Ladungen gegeben sind) interpretieren könntest.
Tipp: Welche Dinge musst du über die einzelne Ladung wissen? Lass Deine Antwort von der Lehrperson überprüfen.

A8. Verändere die Liste der Ladungen so, dass nur noch eine Ladung mit Stärke 1e-9 C im Punkt P(0.4/0.4) bleibt. Lass das Programm laufen.
Hinweis: Wie du gleich sehen wirst, ist im bestehenden Code der Faktor "e-9" ausgelagert in die Variable *charge_scale*. D.h. du musst Dich nur um die "ganzen Zahlen" kümmern.

A9. Wenn du das Bild aus A8 anschaust: was hättest du erwartet bzw. was ist anders als erwartet?

Die Richtung und Stärke der Pfeile werden (noch) zufällig generiert. D.h. dass an jedem Punkt, an welchem du einen Pfeil siehst, nicht die richtige Berechnung des elektrischen Feldes ausgeführt wird, sondern zwei zufällige Grössen

```
field_x = max_field*random.random()
filed_y = max_field*random.random()
```

zugeordnet werden. *random.random()* generiert eine Zufallszahl und *max_field* ist ein Skalierungsfaktor.[1]

A10. Auf was könnte sich die Grösse `field_x` und `field_y` beziehen?

Du siehst, dass zwischen den Zeilen 53 und 63 einiges an Code fehlt. Deine Aufgabe ist es, diesen Teil (d.h. die Berechnung der Komponenten des elektrischen Feldes) zu programmieren.

Zuerst ist es hilfreich, eine Zeichnung zu machen und die einzelnen Punkte zu beschriften:

A11. Zeichne zum untenstehenden Bild Achsen und beschrifte folgende Dinge:

- die Ladung `c` (für charge) sei der ungefüllte Kreis,
- den Punkt im Raum, welcher uns interessiert, mit `p` (point),
- die Abstände in x- und y-Richtung (schau im Code nach wie sie beschrieben werden),
- den absoluten Abstand von c und p (im Code zu finden),
- sowie den E-Feld-Vektor `f` (wenn Q eine positive Ladung ist) in Komponenten zerlegt (die wiederum im Code zu finden sind).

[1] Siehe Kapitel 3.4 im letzten Skript

Wenn du nun zurückblätterst zu **A3**, kannst du feststellen, welche Grössen wir – um die Formel anwenden zu können – berechnen müssen.

- Die Ladung(en) sind bereits vorgegeben => keine Berechnung
- ε_0 und Pi können wir aus den Bibliotheken importieren (siehe später) => keine Berechnung
- Der Abstand `r` zwischen der Ladung und dem Ort, an welchem wir das Feld berechnen wollen, fehlt => Berechnung notwendig

Um den Abstand `r` zu berechnen gehen wir wie folgt vor:

A12. Übertrage – mit Hilfe der Grössen in `c` und `p` – die Abstände von der Quellladung `c` zum Ort des Interesses `p` in x und y Richtung in den Code. (Abstände werden immer von der Ladung weg gemessen).
Achtung: bei den Grössen `c` und `p` handelt sich um Listen (bzw. Arrays) für jedes `c` wie auch jedes `p`. Schau im Skript zum ersten Modul nach, was das heisst oder nutze die Suchmaschine Deiner Wahl um Dich zu informieren.

A13. Wie kannst du nun den absoluten Abstand `r` mit Hilfe der zwei vorher definierten Grössen finden? Schau dir die obige Skizze an und wende einen der fundamentalen mathematischen Sätze der Geometrie an. Was für eine Funktion brauchst Du, um das zu berechnen? Nutze das Internet, um die notwendige Hilfe zu erhalten.

A14. Mit Hilfe der Formel aus **A3** kannst du nun die Berechnung für die Feldstärke `f` in den Code übernehmen.
Tipp: du kannst `Pi` als Dezimalzahl eingeben oder die vorgespeicherte Grösse `Pi` aus der *math* Bibliothek benutzen: `pi`

Du kannst den Code laufen lassen, jedoch wird den zwei Komponenten `field_x` und `field_y` immer noch die zwei Zufallszahlen zugewiesen (Zeilen 65/66). Wir wollen nun – mit Hilfe der vorher berechneten Feldstärke `f` – die Komponenten berechnen, die du in der Skizze von **A11** oben eingezeichnet hast.

A15. Bevor du im Skript weiterliest: diskutiere und finde mögliche Wege, die Komponenten `field_x` und `field_y` zu finden. Nutze dazu Deine Skizze von vorhin und Dein Wissen über Winkel(sätze). Lass Deine Ideen von der Lehrperson überprüfen.

A16. Falls du nicht mindestens 5min selber versucht hast, eine Lösung zu **A15** zu finden, gehe nochmals zurück und nutze das Internet, um eine Lösung zu finden.

Wie können wir die zwei Komponenten (bzw. auch Vektoren) finden, die vektoraddiert unseren Feldstärkevektor erzeugen?

Angenommen wir hätten den Steigungswinkel des Dreiecks `f`, `field_x` und `field_y` (siehe Deine Skizze oben oder mach dir eine neue), könnten wir mit Hilfe der Trigonometrie die Grössen berechnen. Aber wie finden wir den Winkel? In welche Richtung zeigt der Vektor?

Wenn wir zurück ans Coulombgesetz denken – mit zwei Ladungen Q und q – wird der elektrische Feldvektor in der Verbindungslinie von Q und q liegen. Damit liegt der elektrische Feldvektor auch auf der Verbindungsgerade von c und p. (In welche Richtung zeigt der Vektor `f` in Deiner Skizze in **A11**?)

A17. Welche zwei Grössen in Kombination mit welcher Winkelfunktion gibt somit den Winkel an, mit welchem der Feldvektor `f` von `p` weg/hinzeigt? Such im Internet nach der Funktion in Kombination mit *"Python Math Library"* und lass Dein Resultat durch die Lehrperson prüfen. Weise der Variablen *angle* auf Zeile 58 das Resultat der Winkelfunktion (= einen Winkel) zu.

A18. Gegeben sei der Winkel (wie oben berechnet) im Dreieck `f`, `field_x` und `field_y`, wie auch `f`. Wie findest du nun die zwei Katheten `field_x` und `field_y`?
Tipp: Nutze weitere zwei Winkelsätze.

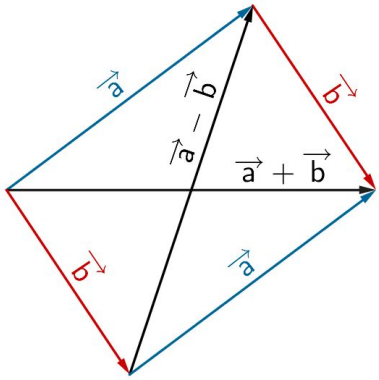
Schreibe alle hergeleiteten Berechnungen in den Code und kommentiere die Zeilen 65/66 aus (schreibe ein "#" zuforderst in die Zeile). Lass dann den Code laufen. Ergibt das Bild Sinn?

A19. Wenn du eine weitere Ladung hinzufügst, was geschieht (oder eben nicht)?

Gelernt
Du hast Dein Wissen aus dem Physikunterricht über Vektoren, Kräfte und das Coulombgesetz angewendet. du hast die Winkelfunktionen in einem rechtwinkligen Dreieck angewedet, um die Katheten bzw. die Winkel zu berechnen.

3. Verallgemeinerung auf mehrere Ladungen

A20. Es seien zwei Vektoren gegeben: `a = (3/4)` und `b = (5/-3)`. Wie gross ist a + b?



Wie du aus der Vektorgeometrie weisst, kann man Vektoren komponentenweise addieren. D.h. das 'Aneinanderhängen' von Vektoren (grafisch), wird rechnerisch über das Addieren der einzelnen Komponenten gelöst. Genau das wollen wir auch in unserem Beispiel anwenden.

Wenn mehrere Ladungen anwesend sind (passe die Ladungen gegebenenfalls an) wirken sich alle Ladungen auf das elektrische Feld in einem gewissen Punkt im Raum aus. D.h. wir müssen (für die Berechnung des "Nettofeldes") die Feldvektoren von allen Ladungen berechnen und diese addieren. Im Algorithmus heisst das, dass wir über alle Ladungen iterieren und ihren Feldvektor berechnen. Dann können wir komponentenweise addieren.

A21. Wie könntest du das im Code umsetzen? (Die Iteration = Schlaufe/Loop ist bereits auf Zeile 49 gegeben).
Tipp: Wenn du einer bestehenden Grösse `x` sich selber plus eine Zahl zuweisen willst (zum Beispiel pro Iteration die Zahl um eins erhöhen – zum Zählen) kann ich das folgendermassen machen: `x += 1`. Dies ist genau dasselbe wie `x = x + 1`.[2]

A22. Generiere nun mindesten 5 Bilder mit unterschiedlichen Ladungskonfigurationen.

Fragen

- Funktioniert unser Code auch mit negativen Ladungen?
- Wie ist die Grösse der einzelnen Ladungen dargestellt? Findest du den Ort im Code, wo die Zuweisung passiert?
- Herausforderung: Kannst du das Beispiel so umprogrammieren, dass die Länge der Pfeile der Stärke der Pfeile entspricht?

[2] Siehe A12 im letzten Skript.