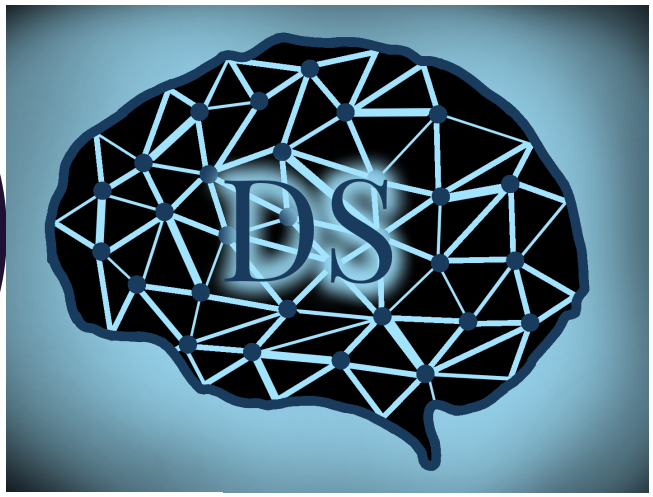
16. Woche der MODELLIERUNG mit Mathematik



Dokumentationsbroschüre 8.2. – 14.2.2020

WOCHE DER MODELLIERUNG MIT MATHEMATIK

LEIBNITZ, 8.2.–14.2.2020

WEITERE INFORMATIONEN:

https://imsc.uni-graz.at/modellwoche/2020/

ORGANISATOREN UND SPONSOREN

Bildungsdirektion
Steiermark

KOORDINATION

Mag. DDr. Patrick-Michel Frühmann

Mag. Vanessa Peinhart

Alexander Sekkas

Vorwort

Viele Wissenschaften erleben zurzeit einen ungeheuren Schub der Mathematisierung. Mathematische Modelle, die vor wenigen Jahrzehnten noch rein akademischen Wert hatten, können heute mit Hilfe von Computern vollständig durchgerechnet werden und liefern praktische Vorhersagen, die helfen, Phänomene zu verstehen, Vorgänge zu planen, Kosten einzusparen. Damit unsere Gesellschaft auch in Zukunft mit der technologischen Entwicklung schritthält, ist es wichtig, bereits junge Leute für diese Art mathematischen Denkens zu begeistern und in der Gesellschaft das Bewusstsein für den Nutzen angewandter Mathematik zu heben. Dies war für uns einer der Gründe, die Woche der Modellierung mit Mathematik zu veranstalten.

Nun ist leider für viele Menschen Mathematik ein Schulfach, mit dem sie eher unangenehme Erinnerungen verbinden. Umso erstaunlicher erscheint es, dass SchülerInnen sich freiwillig melden, um eine ganze Woche lang mathematische Probleme zu wälzen - und dabei auch noch Spaß haben. Sie erleben hier offensichtlich die Mathematik auf eine Art und Weise, wie sie der Schulunterricht nicht vermitteln kann. Die jungen Leute arbeiten und forschen in kleinen Gruppen mit WissenschaftlerInnen an realen Problemen aus den verschiedensten Bereichen und versuchen, mit Hilfe mathematischer Modelle neue Erkenntnisse zu gewinnen. Sie arbeiten ohne Leistungsdruck, dafür mit Eifer und Enthusiasmus, rechnen, diskutieren, recherchieren, oft auch noch am späten Abend, in einer entspannten und kreativen Umgebung, die den SchülerInnen und betreuenden WissenschaftlerInnen gleichermaßen Spaß macht. Die Projektbetreuer konnten auch in diesem Jahr wieder erleben, wie eigenes Entdecken und Selbstmotivation das Verhalten der SchülerInnen während der ganzen Modellierungswoche bestimmen. Sie lernen eine Arbeitsmethode kennen, die in beinahe allen Details den Arbeitsmethoden einer Forschergruppe entspricht. Bei keiner anderen Gelegenheit erfahren SchülerInnen so viel über Forschung wie bei so einer Veranstaltung.

Modellierungswochen gab bzw. gibt es zum Beispiel auch in den USA, in Deutschland oder in Italien. Wir verdanken Herrn Univ.-Prof. Dr. Stephen Keeling den Vorschlag, auch durch die Universität Graz so eine Woche zu veranstalten, und seiner unermüdlichen Organisationsarbeit das tatsächliche Zustandekommen. Er leitet nun bereits zum 16. Mal diese inzwischen zur Institution gewordene Veranstaltung. Ihm sei an dieser Stelle noch einmal ausdrücklich und herzlich gedankt. Besonders wichtig war in den vergangenen Jahren auch die Unterstützung durch den langjährigen Mentor der Modellierungswoche, Herrn o.Univ.-Prof. Dr. Franz Kappel, der oft auch eine eigene Gruppe mit interessanten Problemstellungen betreut hat.

Wir danken der Bildungsdirektion für Steiermark, und hier insbesondere Frau FI^{in} Mag^{a}. Michaela Kraker, für die Hilfe bei der Organisation und die kontinuierliche Unterstützung der Idee einer Modellierungswoche.

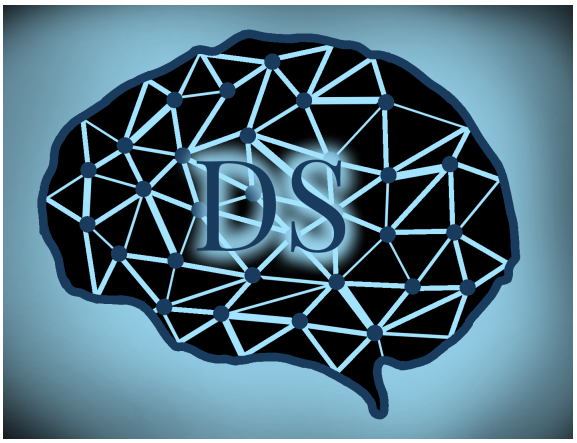
Finanzielle Unterstützung erhielten wir von der Karl-Franzens-Universität Graz durch Rektor Univ.-Prof. Dr. Martin Polaschek und Dekan Assoz. Prof. Dipl.-Ing. Dr. techn. Karl Lohner, vom regionalen Fachdidaktikzentrum für Mathematik und vom Forschungsmanagement der Uni Graz.

Ohne den idealistischen, unentgeltlichen und engagierten Einsatz der direkten Projektbetreuer Gernot Holler, BSc MSc, Dipl.-Ing. Richard Huber, BSc, Florian Thaler, BSc, Dr. Robert Beinert, BSc MSc – Institut für Mathematik und Wissenschaftliches Rechnen – hätte diese Modellierungswoche nicht stattfinden können.

Besonderer Dank gebührt ferner Herrn Mag. DDr. Patrick-Michel Frühmann, der die gesamte Veranstaltung organisiert und betreut hat und auch die Gestaltung dieses Berichtes übernommen hat, Frau Mag. Vanessa Peinhart für die tatkräftige Hilfe bei der organisatorischen Vorbereitung und Herrn Alexander Sekkas für die Hilfe bei der Betreuung der Hard- und Software.

Leibnitz, am 14. Februar 2020

Bernd Thaller
Institut für Mathematik und Wissenschaftliches Rechnen
Karl-Franzens-Universität Graz



Projekt Data Science

Objekterkennung mittels künstlicher neuronaler Netzwerke

David Georg Steiner Eva Zarschenas Larissa Marbler Laura List Raphael W. Koller
Selina Golz Betreuer : Gernot Holler

Februar 14, 2020

Inhaltsverzeichnis

1 Einführung

Machine Learning ist ein Prinzip für die synthetische Generierung von Wissen aus Erfahrung, die durch Trial-and-Error gewonnen wird. Einem Algorithmus wird eine bestimmte Aufgabe gestellt, woraufhin dieser versucht, Muster und Gesetzmäßigkeiten in Lerndaten zu erkennen und damit generalisierte Richtlinien aufzustellen, mit denen die Aufgabe bestmöglich gelöst werden kann. Wenn die selbst aufgestellten Regeln auch bei dem System unbekannten Daten gut anwendbar sind, wird das Modell beibehalten. Sollte es jedoch zum "Overfitting" kommen (Das Modell ist den Trainingsdaten zu sehr angepasst. Oft bei zu geringer Menge von Trainingsdaten der Fall), oder schneidet es allgemein schlecht ab, wird es verworfen, der Algorithmus leicht verändert und ein neues Modell generiert, mit dem hoffentlich bessere Resultate erzielt werden. Der entscheidende Punkt hierbei ist die automatische Optimierung des Algorithmus, da das manuelle Bestimmen derartig vieler Regeln viel zu viel Zeit und Aufwand benötigen würde. Man unterscheidet man grundsätzlich zwischen zwei Arten des maschinellen Lernens:

1. Das überwachte Lernen - Trainingsergebnisse werden mit den schon bekannten, richtigen Ergebnissen verglichen.
2. Das unüberwachte Lernen - Es gibt keine bereits bekannten Ergebnisse.

Wir haben uns ausschließlich mit dem überwachten Lernen beschäftigt. Die Bandbreite der möglichen Anwendungsgebiete beinhaltet:

- medizinische Diagnostik
- Analyse des Aktienmarktes
- Text- und Spracherkennung

Deep Learning ist eine Methode des Machine Learnings, welche künstliche neuronale Netze als Algorithmus zum Lernen verwendet. Diese Netzwerke sind lose angelehnt an die Funktionsweise des Gehirns. Im Gehirn haben manche Verbindungen zwischen den Nervenzellen, den so genannten Neuronen, eine größere Auswirkung als andere. Nach dem gleichen Prinzip haben die Neuronen eines künstlichen neuronalen Netzwerkes bedeutsamere und weniger bedeutsame Verbindungen untereinander. Die "Stärke" der Verbindung zwischen den Einzelnen Neuronen wird in unserem Algorithmus mit den "Weight" - Parametern angegeben. Die Neuronen im KNN (=Künstliches neuronales Netzwerk) werden in "Layer" (Schichten) organisiert. Die Anzahl der Neuronen im ersten Layer ist gleich der Anzahl der Informationswerte, die dem System gefüttert werden. So hat ein KNN, das ein 28 x 28 Pixel großes Bild verarbeitet, im ersten Layer 784 Neuronen. Danach gibt es beliebig viele "Hidden Layers" mit einer beliebigen Anzahl (kleiner als die Ausgangsanzahl) von Neuronen und zum Schluss einen Layer, wo die Anzahl der Neuronen gleich der gewünschten Anzahl der Outputs ist. Jedes Neuron einer Schicht ist mit allen Neuronen der vorherigen und der folgenden Schicht verbunden, wobei der Wert, der einem Neuron zugeordnet wird, von allen Neuronen-Werten der vorherigen Schicht, den Weights der Verbindungen und einem anderen Parameter, dem Bias, abhängt. Grundlegend besteht ein KNN also aus einer äußerst komplexen Funktion mit unzählbar vielen Parametern, die sich aus sehr vielen, fundamental einfachen Funktionen zusammensetzt. Diese Parameter werden vom Algorithmus optimiert (=lernen). Wir haben im Laufe unseres Projektes mithilfe von Google Colab bzw. Python und der Programm Bibliothek Tensorflow“ künstliche neuronale Netzwerke programmiert und sie mit unseren eigens zusammengestellten Datensets dazu trainiert, gewisse Aufgaben zu lösen.

2 Punktklassifizierung

Importieren der notwendigen Programmbibliotheken

```
try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass

import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
```

In der ersten Code-Section haben wir verschiedene Programmbibliotheken importiert, um zusätzliche Funktionen wie zum Beispiel Ergebnisse "plotten" beziehungsweise grafisch darstellen zu können, zu haben. Diese Biliotheken werden unter einem einfachen Kürzel gespeichert, damit man sie schneller aufrufen kann.

Generieren von Datenpunkten.

```
input_dim = 2
train_size = 200
test_size = 150

train_inputs = np.random.rand(train_size,input_dim)   # generieren der Tra
train_labels = np.zeros((train_size,1))*1
train_labels[np.tan(10*train_inputs[:,0] +np.exp(train_inputs[:,1]))<0.5,0

test_inputs = np.random.rand(test_size,input_dim)     #generieren der Test
test_labels = np.zeros((test_size,1))
test_labels[np.tan(10*test_inputs[:,0] +np.exp(test_inputs[:,1]))<0.5,0] =
```

Zuallererst wird eine bestimmte Anzahl zufälliger Werte zwischen 0 und 1 pro Datenset generiert. Wir haben 2 Datensets: unsere Trainingdaten (`train_inputs`), mit denen wir unseren Netzwerk lehren die Daten korrekt in 2 Kategorien einzuteilen und die Testdaten (`test_inputs`) um zu testen wie gut unser Netzwerk mit nicht zuvor gesehenen Daten ist.

Die `train-` beziehungsweise `test_ inputs` beinhalten eine Liste mit den Koordinaten der einzelnen Punkte. Die `train-` beziehugsweise `test_labels` hingegen sind eine Liste in der jedem Punkt eine Kategorie zugeordet wird.

Die Aufteilung der Punkte in die 2 Kategorien (man kann die Aufteilung in der nächsten Grafik anhand von unterschiedlichen Farben erkennen) erfolgt durch die Übergebung der zufälligen Werte an eine Funktion. In unserem Falle lautet diese Funktion:

$$f(x) = \tan(10x + e^x)$$

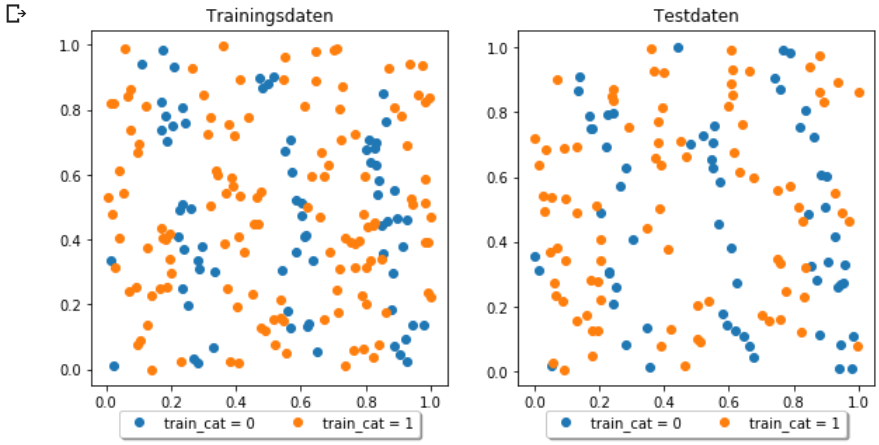
Wenn die Werte, die für die Inputs bei der Funktion herrauskommen größer als 0,5 sind, gehören sie zu Kategorie 0 und wenn sie kleiner als 0,5 sind zu Kategorie 1.

Plot der Datenpunkte

Die folgenden Zeilen sind nur dazu da, unsere Punkte grafisch darzustellen / zu visualisieren und diese Grafik anschaulich zu gestalten.

```
# Darstellen der train_inputs
ax = plt.figure(figsize=(10,5))
ax = plt.subplot(1,2,1)
ax.plot(train_inputs[train_labels[:,0] == 0, 0], train_inputs[train_labels
        label = 'train_cat = 0', marker = 'o', linestyle = ' ', color = '#
ax.plot(train_inputs[train_labels[:,0] == 1, 0], train_inputs[train_labels
        label = 'train_cat = 1', marker = 'o', linestyle = ' ', color = '#
# Aufhübschen des Plots / der Grafik
box = ax.get_position()
ax.set_position([box.x0, box.y0,
              box.width, box.height ])
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05),
      fancybox=True, shadow=True, ncol=4)
```

```
ax.axis('square')
plt.title('Trainingsdaten')

# Darstellen der test_inputs
ax = plt.subplot(1,2,2)
ax.plot(test_inputs[test_labels[:,0] == 0, 0], test_inputs[test_labels[:,0
        label = 'train_cat = 0', marker = 'o', linestyle = ' ', color = '#
ax.plot(test_inputs[test_labels[:,0] == 1, 0], test_inputs[test_labels[:,0
        label = 'train_cat = 1', marker = 'o', linestyle = ' ', color = '#
# Aufhübschen des Plots
box = ax.get_position()
ax.set_position([box.x0, box.y0,
             box.width, box.height ])
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05),
      fancybox=True, shadow=True, ncol=4)

ax.axis('square')
plt.title('Testdaten')
plt.show()
```



Erlerne eine Funktion die jedem Punkt die zugehörige Kategorie zuordnet.

Ein neuronales Netzwerk besteht wie schon erwähnt aus einzelnen Layern. Ein Netzwerk mit nur einem Layer wäre dazu fähig unsere Daten nach einer Trenn-Gerade einzuteilen. Aber bei unseren Datensätzen ist es etwas schwer mit nur einer Gerade alles einzuteilen. Deshalb braucht man mehrere Layer. Es gilt, je mehr Layer ein Netzwerk hat, desto komplitziertere Trennlinien kann es ermitteln.

Ein Layer besteht aus einer linearen Funktion mit der Form $z(x) = a * x + b$ in Kombination mit einer Aktivierungsfunktion $\sigma(z)$.

In der linearen Funktion $z(x) = a * x + b$ haben wir a und b, W und b genannt. Das W steht für weight und das b für bias.

Mehrere Layer bestehen also aus mehreren linearen Funktion in Kombination mit der Aktivierungsfunktion. Diese Verkettungen kann man sich vorstellen indem man quasi die Ergebnisse des vorherigen Layers / der inneren Funktion in die nächste einsetzt. Somit sind viel kompliziertere Trennlinien möglich, wie es für unsere Datenpunkte von nöten ist.

$$Funktion = \ldots h_2(h_1(\sigma(Wx + b)))$$

wobei $h_0(x) = \sigma(Wx + b)$

Für die Aktivierungsfunktion σ verwenden wir die voreingestellten Funktionen mit dem Namen "relu". Diese Funktionen sind im Code definiert durch `activation = 'relu'` (relu = Rectified Linear Unit)

Jeder Layer hat einen Output, den er erstellen kann. Diesen Output nennt man auch Neuronen, von denen unsere Layer jeweils 50 haben. Das besondere an dem letzten Layer ist, dass seine Neuronenzahl beziehungsweise sein Output der Anzahl an Kategorien entsprechen muss, wie in unserem Fall 2. Für die Aktivierungsfunktion verwenden wir im letzten Layer die voreingestellte Funktion mit dem Namen `softmax`. Sie sorgt nur dafür, dass wir als Output eine prozentuelle Wahrscheinlichkeit pro Kategorie erhalten.

Der Befehl `model.compile` führt mehrere Funktionen (auswerten & optimieren) aus.

- Die erste Funktion ist der `optimizer`, der dafür zuständig ist, dass das Netzwerk sich verbessern kann beziehungsweise bessere Werte für seine verkettete Funktion findet. Unser Optimierungsverfahren ist wieder ein voreingestelltes und trägt den Namen `adam`. Diese Funktion ist also dafür dar, ein Minimum der loss - Funktion zu finden.
- Die `loss` - Funktion beschreibt, wie ähnlich die vom Netzwerk generierten Werte für die Kategorien der Daten, den "Lösungen" also den labels sind. Je höher die loss - Funktion ist, desto ungenauer sind die Ergebnisse. Geht die loss - Funktion gegen 0 ist die Genauigkeit am höchsten. Das heißt, die loss - Funktion soll möglichst gering sein.

- Die letzte Funktion ist die `accuracy`. Sie gibt einfach in Prozent an wieviele Punkte der richtigen Kategorie / wieviele Inputs richtig gelabelt wurden.

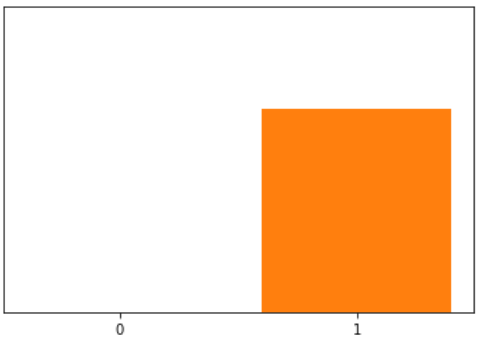
```
model = keras.Sequential([
    keras.layers.Dense(50,activation='relu'),    # 1. Hidden Layer
    keras.layers.Dense(50,activation='relu'),
    keras.layers.Dense(50,activation='relu'),
    keras.layers.Dense(50,activation='relu'),
    keras.layers.Dense(2, activation='softmax')  # letzter Layer
])

model.compile(optimizer='adam',                        #Optimierungsverfa
              loss='sparse_categorical_crossentropy',  #beste loss Funkti
              metrics=['accuracy'])                    #Accuracy = Anzahl

model.fit(train_inputs, train_labels, epochs=1000, verbose = 0)
```

```
<tensorflow.python.keras.callbacks.History at 0x7f12c0820198>
```

Visualisierung der Ergebnisse

Im nächsten Absatz wird der Befehl `prediction` definiert. Er sagt mithilfe der verketteten Funktion vorraus, zu welcher Kategorie ein Datenpunkt gehört. Als nächstes nehmen wir einen Datenpunkt heraus und definieren diesen als k (als eine Zahl zwischen 0 und 199, wobei sowohl 0 als auch 199 enthalten sind, da insgesamt 200 Datenpunkte dem train_input zugehören) und stellen die Wahrscheinlichkeiten, zu welcher Kategorie dieser Datenpunkt gehört, grafisch dar. Die Wahrscheinlichkeit zu Kategorie 0 und die zu Kategorie 1 zu gehören sind mit Farben gekennzeichnet und summieren sich auf 1.

```
prediction = model.predict(train_inputs)

k=30
predicted_label =np.argmax(prediction[k,:])

# Visualisierung der Wahrscheinlichkeiten
plt.figure()
plt.xticks(range(2))
plt.yticks([])
plt.ylim([0, 1.5])
thisplot= plt.bar(0,prediction[k,0])
thisplot= plt.bar(1,prediction[k,1])
```



Testen des neuronalen Netzwerks

Um zu erfahren, wie gut unser Netzwerk gelernt hat, welche Daten zu welchen Kategorien gehören, testen wir das Netzwerk nochmal an Testdaten (`test_inputs`), welche sich von den Trainingsdaten unterscheiden.

Die Accuracy von den Testdaten ist niedriger als die, der Trainingsdaten, da unser Netzwerk die "Auflösungen" der Trainingsdaten bekommt und immer wieder mit ihnen übt. Die Kritierien, die es an den Trainingsdaten feststellt, sind möglicherweise nur auf den Trainingsdaten festzustellen und somit wäre unser Netzwerk nicht in der Lage auch andere Daten zuzuordnen. Doch dies ist bei uns nicht festzustellen, da die Accuracy auf den Testdaten der Accuracy auf den Trainingsdaten recht nahe kommt.

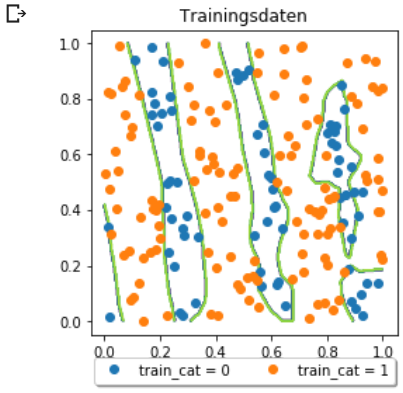
```
test_loss, test_acc = model.evaluate(test_inputs,  test_labels, verbose=2)

print('\nTest accuracy:', test_acc)
```

```
150/150 - 0s - loss: 0.4694 - accuracy: 0.8467

Test accuracy: 0.8466667
```

Grenze zwischen Punktarten angeben lassen

Um darzustellen, was unser Netzwerk erreicht hat, visualisieren wir noch einmal alle unserer Datenpunkten und die Trennlinie, die erzeugt wurde. An dieser Trennlinie wurde von unserem Netzwerk entschieden, ob ein Datenpunkt zu Kategorie 0 oder zu Kategorie 1 gehört. Wie man sieht ist unsere Trennlinie recht komplex, sie besteht nicht nur aus einer Gerade sondern auch aus Kurven und komplizierteren Formen und teilt unsere Grafik in mehrere Abschnitte ein. Das alles ist nur durch die verschiedenen Layer unseres Netzwerkes möglich.

```
# Darstellen des train_inputs
ax = plt.subplot(111)
ax.plot(train_inputs[train_labels[:,0] == 0, 0], train_inputs[train_labels
        label = 'train_cat = 0', marker = 'o', linestyle = ' ', color = '#
ax.plot(train_inputs[train_labels[:,0] == 1, 0], train_inputs[train_labels
        label = 'train_cat = 1', marker = 'o', linestyle = ' ', color = '#
# Aufhübschen des Plots
box = ax.get_position()
ax.set_position([box.x0, box.y0,
             box.width, box.height ])
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05),
      fancybox=True, shadow=True, ncol=4)
ax.axis('square')
plt.title('Trainingsdaten')

# Einzeichnung der Trennlinie
n_gridpoints = 300
x = np.linspace(0,1,n_gridpoints)
mesh = np.zeros([n_gridpoints,n_gridpoints,2])

mesh[:,:,0], mesh[:,:,1] = np.meshgrid(x,x)
result = model.predict(np.reshape(mesh,(n_gridpoints * n_gridpoints,2)))
result = np.reshape(result, (n_gridpoints, n_gridpoints,2))
plt.contour(mesh[:,:,0], mesh[:,:,1], np.argmax(result, axis=2))
plt.show()
```



3 Logistische Regression

Bisher haben wir immer bereits definierte Funktionen aus der `keras` bzw. der `tensorflow` Bibliothek verwendet, um passende Werte für die Gewichte und den Bias zu finden. Um selbst auch die Prozesse, die hinter den ausgeführten Befehlen stecken, zu verstehen, haben wir nun auch ein Programm geschrieben, in dem alle für den Trainingsprozess benötigten Funktionen von uns manuell definiert werden.

In diesem Notebook beschränken wir uns auf eine logistische Regression. Eine logistische Regressionsfunktion ist ein Spezialfall eines neuronalen Netzwerks ohne verstecke Schichten (hidden layers).

Importieren der notwendigen Programmbibliotheken

```
try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass

import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from matplotlib import rc
```

Gleich wie in den vorherigen Notebooks importieren wir die Bibliotheken, die wir in unserem Programm brauchen.

Generieren von Datenpunkten

Wir generieren zwei Klassen von Datenpunkten auf einem Quadrat. Unsere Daten haben die folgende Form:

- `train_inputs, test_inputs` : Arrays mit zweidimensionalen Koordinaten der Trainings- bzw. Testpunkte.
- `train_labels, test_labels` : Arrays mit Kategorien der Trainings- bzw. Testpunkte. Der Wert 0 steht für Kategorie 0, der Wert 1 steht für Kategorie 1.

```
# Unsere Inputs sind zweidimensional, d.h.
# jeder Input besteht aus 2 Daten - den x- und y-Koordinaten des Punktes
input_dim = 2
train_size = 300 # Anzahl der Trainingsdaten
test_size = 100  # Anzahl der Testdaten

train_inputs = np.random.rand(train_size,input_dim)
train_cat = np.zeros((train_size,2))
# den Inputs werden Kategorien zugewiesen
train_cat[np.sin(3*train_inputs[:,0]**2) + train_inputs[:,1]**2 < 0.5,0] = 1
train_cat[:,1] = 1 - train_cat[:,0]

test_inputs = np.random.rand(test_size,input_dim)
test_cat = np.zeros((test_size,2))
# den Inputs werden Kategorien zugewiesen
test_cat[np.sin(3*test_inputs[:,0]**2) + test_inputs[:,1]**2 < 0.5,0] = 1
test_cat[:,1] = 1 - test_cat[:,0]

# Matrizen mit den Kategorienzuordnungen werden umformatiert
train_labels = train_cat[:,1].reshape(len(train_cat))
test_labels = test_cat[:,1].reshape(len(test_cat))
```

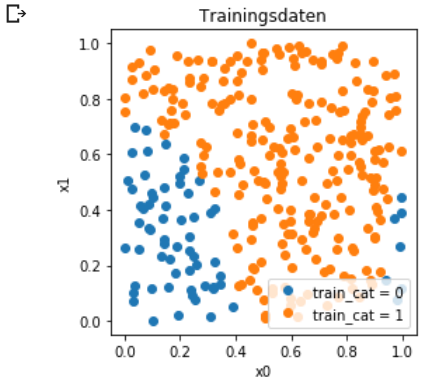
Gleich wie im Notebook "Classify Points" generieren wir zuerst eine bestimmte Anzahl an Trainings- und Testdaten (`train_inputs` und `test_inputs`). Denen wird in weiterer Folge je nach Wert einer bestimmten Funktion eine Kategorie (0 oder 1) zugewiesen. In unserem Falle lautet diese Funktion

$$f(x_0, x_1) = \sin(3x_0^2) + x_1^2$$

Bei uns hängt die Kategorie des Punktes (x_0, x_1) davon ab, ob der Funktionswert $f(x_0, x_1)$ größer oder kleiner als 0.5 ist. In unserem Programm erhält man die Koordinaten des i-ten Datenpunktes mittels `train_inputs[i]` . Genauso erhält man die Kategorie des i-ten Datenpunktes mittels `train_labels[i]` .

```
def plot_datapoints(inputs,labels,plot_title) :
```

```
    plt.subplot(111)
    plt.plot(inputs[labels[:,0] == 1, 0], inputs[labels[:,0] == 1, 1], \
        label = 'train_cat = 0', marker = 'o', linestyle = ' ', color = '#1f77b4')
    plt.plot(inputs[labels[:,1] == 1, 0], inputs[labels[:,1] == 1, 1], \
        label = 'train_cat = 1', marker = 'o', linestyle = ' ', color = '#ff7f0e')
    plt.title(plot_title)
    plt.xlim(0,1)
    plt.ylim(0,1)
    plt.axis('square')

plot_datapoints(train_inputs,train_cat,'Trainingsdaten')
plt.xlabel('x0')
plt.ylabel('x1')
plt.legend()
plt.show()
```



Hier definieren wir eine Funktion `plot_datapoints`, die uns alle Punkte eines Datensets, je nach Kategorie unterschiedlich gefärbt, in einer Grafik ausgibt. Hier sind die oben generierten Trainingsdaten `train_inputs` dargestellt, wobei alle blauen Punkte der Kategorie 0 und alle orangen der Kategorie 1 angehören.

Logistische Regression

Nun beschreiben wir die Bausteine unserer logistischen Regressionsfunktion. Der erste Baustein ist eine affin lineare Funktion der Form

$$x \mapsto Wx + b$$

Hier ist

- $W = \begin{pmatrix} w_{0,0} & w_{0,1} \end{pmatrix} \in \mathbb{R}^{1,2}$ eine Matrix mit 1 Zeile und 2 Spalten, da unsere Daten zwei Koordinaten besitzen. Wir nennen die Bestandteile dieser Matrix Gewichte/Weights und in weiterer Folge wird die Matrix W im Programm als `weights` bezeichnet.
- $b \in \mathbb{R}$ eine Zahl. Diese wird auch als "Bias" bezeichnet und kommt im Programm auch als `bias` vor.
- Wir verwenden oftmals die Schreibweise $x = (x_0, x_1)^\top$. In dem Fall sind x_0 und x_1 die Koordinaten von x.

Die Gewichte und der Bias werden später im Laufe des Trainingsprozesses vom Programm so bestimmt, dass eine möglichst akkurate Trenngerade zwischen den beiden Punktekategorien geschaffen wird.

```
input_dim = 2
output_dim = 1

weights = 2*np.random.rand(output_dim,input_dim) # zufällig ausgewählt
bias = np.random.rand(output_dim) # zufällig ausgewählt
```

Anfänglich wählen wir Werte der Einträge von `weights` und `bias` zufällig durch die Funktion `np.random.rand`.

Später werden diese Variablen ledeglich als Member der Klasse `NeuralNetwork` auftauchen. In dieser Klasse tauchen alle Funktionen auf die wir in früheren Programmen durch `model=keras.Sequential` sowie `model.compile` und `model.fit` aufgerufen haben.

Logistische Regressionsfunktion

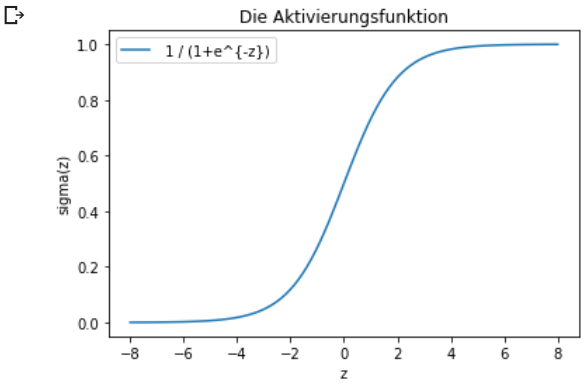
Die logistische Regressionsfunktion erhält man nun, wenn man eine affin lineare und eine sogenannte logistische Funktion (später Aktivierungsfunktion genannt) hintereinander ausführt. Wir bezeichnen die logistische Regression folgendermaßen:

$$x \mapsto \sigma(Wx + b)$$

Dieser grundsätzliche Funktionsaufbau einer solchen Funktion ist auch schon in dem Kapitel "Classify Points" zu finden. Hier ist die Aktivierungsfunktion $\sigma\colon \mathbb{R} \to \mathbb{R}$ gegeben durch

$$\sigma(z) = \frac{1}{1+e^{-z}}.$$

```
z_vec = np.linspace(-8,8,100) # wähle 100 Werte für z im intervall (-8,8)

def sigma(z):
# definition der aktivierungsfunktion
 return 1 / (1 + np.exp(-z))

sigma_z_vec = sigma(z_vec)

plt.plot(z_vec,sigma_z_vec, label = ' 1 / (1+e^{-z})')
plt.xlabel('z')
plt.ylabel('sigma(z)')
plt.legend()
plt.title('Die Aktivierungsfunktion')
plt.show()
```



Zur besseren Veranschaulichung dieser Funktion plotten wir zunächst einmal die Aktivierungsfunktion. Wie man sieht ist diese Funktion für alle Werte $z < -2$ nahe 0, steigt rund um den z-Wert 0 stark an und nimmt dann für alle $z > 2$ Werte nahe 1 an.

Wahrscheinlichkeiten

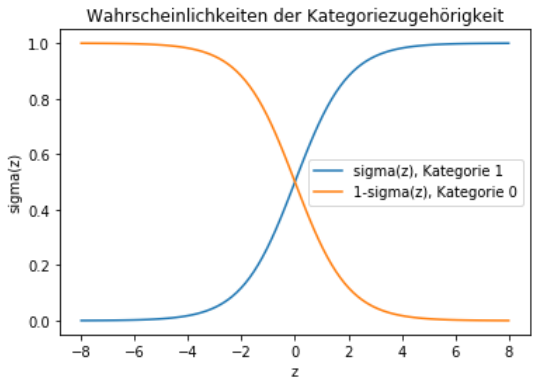
Eigentlich berechnen wir keine Wahrscheinlichkeiten im klassischen Sinne. Es ist aber dennoch oft nützlich einige Werte als "Wahrscheinlichkeiten" zu interpretieren. Wir machen das folgendermaßen:

- Den Wert von $\sigma(z)$ interpretieren wir später als Wahrscheinlichkeit der Zugehörigkeit zur Kategorie 1.
- Den Wert von $1 - \sigma(z)$ interpretieren wir später als Wahrscheinlichkeit der Zugehörigkeit zur Kategorie 0.

Somit summieren sich die Wahrscheinlichkeiten für Kategorie 0 und Kategorie 1 immer zum Wert 1 auf.

Wir plotten nun die Wahrscheinlichkeiten der Kategoriezugehörigkeit 1 bzw. 0 in Abhängigkeit vom Wert der affin linearen Funktion $z(x) = Wx + b$. Hier ist z eine reelle Zahl.

```
plt.plot(z_vec,sigma_z_vec, label = 'sigma(z), Kategorie 1')
plt.plot(z_vec,1-sigma_z_vec, label = '1-sigma(z), Kategorie 0')
plt.xlabel('z')
plt.ylabel('sigma(z)')
plt.legend()
plt.title('Wahrscheinlichkeiten der Kategoriezugehörigkeit')
plt.show()
```



Wie man sieht erhält man für $1 - \sigma(z)$ die an der Gerade $y = 0$ gespiegelte Funktion $\sigma(z)$. Hier kann man den Schnittpunkt der beiden Funktionen als unsere Trenngerade ansehen. Auf der einen Seite dieser Gerade entscheidet sich das Programm dafür den Punkt der Kategorie mit der überwiegenden Wahrscheinlichkeit zuzuweisen. Auf der anderen Seite der anderen Kategorie, die in diesem Berreich den höheren σ-Funktionswert hat.

Das Erstellen unseres neuronalen Netzwerkes

Nun kommen wir zur Definition der Klasse `NeuralNetwork`, in der die benötigten Funktionen für das Training definiert werden. Wenn wir ein Objekt der Klasse übergeben, können wir alle in ihr definierten Funktionen mit der übergabe von passenden Daten auf dieses Objekt anwenden. Die Herleitung und der Zweck der einzelnen Funktionen werden nach der Code-Sektion ausführlicher erklärt.

```
class NeuralNetwork() :

    def __init__(self, dimensions):
    # Initialisiert ein Element der Klasse NeuralNetwork

        self.weights = 2 * np.random.rand(1,2) - 1   # Wähle das Anfangsgewicht zufällig
        self.bias = np.random.rand(1)   # Wähle den Anfangsbias zufällig

    def think(self,input): #rechnet den Wert unseres Layers aus
      #affin lineare Abbildung Wx + b
      z = np.tensordot(input,self.weights, axes = (1,1)).reshape(len(input)) + self.bias

      thought = self.activation(z) # Aktivierungsfunktion \sigma

      return thought

    def loss(self,pred_label,true_label):
      # Berechnet den aktuellen Wert der loss Funktion
      fun_value = -(1-true_label) *  np.log(1-pred_label)  - (true_label * np.log(pred_label))

      fun_value = np.sum(fun_value) / len(fun_value)

      return fun_value

    def deriv_loss(self,pred_label,true_label):
      # Ableitung der loss Funktion
      deriv_value = (1-true_label) / (1- pred_label) - (true_label / pred_label)

      return deriv_value / len(pred_label)

    def activation(self,z):
      # Aktivierungsfunktion \sigma

      return 1 / (1 + np.exp(-z))

    def deriv_activation(self,z):
      # Die Ableitung von \sigma ausgewertet in \sigma^{-1}(z)

      return z * (1-z)

    #das eigentliche Trainieren des Netzwerkes
    def train(self, inputs, labels, stepsize = 0.01, epochs = 100, batch_size = None):
      # Gradientenverfahren
      # stepsize ... Schrittweite
      # epochs   ... Momentan: Anzahl der Gradientenschritte
      if batch_size == None:
```

```
        batch_size = len(inputs)


    indices = np.arange(len(inputs))

    n_batches = int(len(inputs) / batch_size)

    for k in range(epochs):

      np.random.shuffle(indices)

      for i in range(n_batches):

          inputs_b = inputs[indices[i*batch_size:(i+1)*batch_size]]
          labels_b = labels[indices[i*batch_size:(i+1)*batch_size]]

          y_pred = self.think(inputs_b) # Berechnet \sigma (W inputs_b + b)

          # Berechnet loss'(y_pred, labels_b)*\sigma'(\sigma^{-1}(y_pred))
          delta = self.deriv_loss(y_pred,labels_b) * self.deriv_activation(y_pred)

          #Berechnet die Ableitung nach W
          grad_weights = np.tensordot(delta, inputs_b, axes=(0,0))
          # Berechnet der Ableitung nach b: Summieren über alle Punkte des Batches
          grad_bias = np.sum(delta, axis = 0)

          self.weights = self.weights - stepsize * grad_weights # Gradientenschritt
          self.bias = self.bias - stepsize * grad_bias # Gradientenschritt

    return
```

Logistische Regressionsfunktion

Die vorhin bereits erwähnte Form unserer Funktion für die das Netzwerk gute Werte für W, b sucht, wird hier nun entgültig als

$$f(x) = \sigma(Wx + b)$$

beziehungsweise `think(x)` definiert.

Nach der Berechnung von $z := Wx + b$ wird der Wert von z an die klasseneigene Aktivierungsfunktion, d.h. unsere bereits graphisch dargestellte logistische Funktion

$$\sigma(z) = \frac{1}{1 + e^{-z}},$$

übergeben. Diese σ-Funktion kommt als `activation(z)` im Programm vor.

Die Werte von `thought` (entsprechen der Ausgabe von `think`) geben somit die vom System berechneten Kategorien aller Punkte aus. Das System hat somit eine Einteilung der Punkte in 2 Kategorien vollbracht, jedoch noch keine Möglichkeit zu überprüfen wie richtig oder falsch es mit seinen Einschätzungen liegt.

Konstruktion der Verlustfunktion

Um dem System nun einen Weg zu geben die Richtigkeit seiner Werte auszuwerten, definieren wir die Verlustfunktion bzw die `loss`-Funktion. Wir übergeben dieser 2 Werte. Als ersten Wert verwenden wir den Output von $\sigma(Wx + b)$ bzw `thought`, den Return-Wert der `think`-Funktion. Wir bezeichnen diese mit $\hat{y}$.

Für die Vorhersage $\hat{y}$, die `loss` übergeben wird, gilt:

- $\hat{y}$ ist die Wahrscheinlichkeit dass $\hat{y}$ zur Kategorie 1 gehört. (Dies entspricht dem vorherigen $\sigma(z)$.)
- $1 - \hat{y}$ ist die Wahrscheinlichkeit dass $\hat{y}$ zur Kategorie 0 gehört. (Entspricht dem $1 - \sigma(z)$.)

Für das exakte Label y, das als zweiter Wert der Funktion übergeben wird, gilt:

- $y = 0$ falls Objekt gehört zur Kategorie 0
- $y = 1$ falls Objekt gehört zur Kategorie 1

Diese Verlustfunktion wird für einen Datenpunkt folgendermaßen definiert:

$$\text{loss}(\hat{y}, y) = -(1 - y)\ln(1 - \hat{y}) - y\ln(\hat{y})$$

Betrachten wir ein paar Fälle für einen Datenpunkt:

- Falls $y = 1$, dann gilt
  $$\text{loss}(\hat{y}, y) = -\ln(\hat{y})$$
  Falls $\hat{y}$ nahe 1, erfolgt keine Bestrafung. Wenn $\hat{y}$ gegen 0 strebt, dann geht $\text{loss}(\hat{y}, 1)$ gegen ∞.

- Falls $y = 0$, dann gilt
$$\text{loss}(\hat{y}, y) = -\ln(1 - \hat{y})$$
Falls $\hat{y}$ nahe 0, erfolgt keine Bestrafung. Wenn $\hat{y}$ gegen 1 strebt, dann geht $\text{loss}(\hat{y}, 0)$ gegen ∞.

Hätten wir nur einen Datenpunkt x^i, y^i, dann verwenden wir als Verlustfunktion:
$$L_i(W, b) = \text{loss}(\sigma(Wx^i + b), y^i)$$

Da wir aber viele Datenpunkte haben und alle evaluieren wollen, bilden wir die Summe aller `loss`-Werte der einzelnen Datenpunkte. Also erhält man dafür:
$$L(W, b) = \sum_{i=0}^{N} L_i(W, b)$$

Die Zahl, die wir als Ergebnis dieser Verlustfunktion erhalten ist ein Wert der die Richtigkeit der Werte unseres Netzwerks anzeigt. Je höher dieser Wert ist desto ungenauer und schlechter ist unser Netzwerk.

Unser Optimierungsverfahren

Wir wollen nun, da wir die Genauigkeit der errechneten Kategoriezuteilungen berechnen können, dem Netzwerk auch eine Möglichkeit geben die Werte der Gewichte W und dem Bias b zu verbessern.

Wir haben bereits festgestellt, dass ein großer Wert als Ergebnis unserer `loss`-Funktion schlecht ist. Somit ist ein Minimalwert dieses Ergebnisses der Verlustfunktion anzustreben.

Um die Extremwerte (d.h. Minimal- oder Maximalwerte) einer Funktion $f(x)$ zu berechnen, bedient man sich normalerweise der Ableitung der Funktion und setzt diese 0, also $f'(x) = 0$.

Wir wollen also eigentlich die Ableitung unserer Loss-Funktion berechnen. Da diese aber zweidimensional und recht komplex ist, können wir die Optimalwerte für die Minimalstelle nicht exakt mit einer mathematischen Gleichung ausrechnen. Daher verwenden wir ein Annäherungsverfahren, mit dem wir Schrittweise immer näher an die benötigten Werte des Minimums kommen. Das Verfahren, das wir verwenden nennt sich Gradientenverfahren.

Um genauer zu sein das Gradientenverfahren mit fixer Schrittweite.

Beim Gradientenverfahren bedient man sich dem Gradienten. Dieser ist ein Richtungsvektor, bestehend aus den Ableitungen einer zweidimensionalen Funktion. Für $f(x_0, x_1)$ wäre dieser:
$$\nabla f = \begin{pmatrix} \frac{\mathrm{d}f}{\mathrm{d}x_0} \\ \frac{\mathrm{d}f}{\mathrm{d}x_1} \end{pmatrix}$$

Dieser Vektor beschreibt bei einer zweidimensionalen Funktion die Richtung des stäksten Anstiegs. Die Funktionalität des Gradientenverfahrens beruht auf Folgendem: Wenn man von einem beliebigen Punkt auf dieser Funktion ausgeht und den Gradienten berechnet, geht man einen Schritt in diese Richtung des stärksten Anstiegs. Durch diese Bewegung befindet man sich an einem neuen Punkt und der Vorgang wird wiederholt. Wenn man diesen Vorgang oft genug wiederholt, hofft man, dass man sich schlussendlich an der höchsten Stelle des Graphens wiederfindet, da man kontinuierlich "hinaufgegangen" ist.

Wir aber wollen aber nicht an der höchsten Stelle (dem Maximum), sondern an der niedrigsten Stelle unserer Funktion landen. Daher benötigen wir den negativen Gradienten, der die Richtung des stärksten Abstiegs beschreibt und uns schlussendlich in die Nähe des Minimums unserer Verlustfunktion bringen wird.

Grundsätzlich ist der Algorithmus, den wir in unserem Programm unter `train` implementiert haben, folgender:

- Schritt 0: Wähle W^0 und b^0 zufällig, wählen eine fixe Schrittweite $\tau > 0$ und setzen $k = 0$.
- Schritt 1: Berechne $\frac{\mathrm{d}}{\mathrm{d}W} L(W^k, b^k)$.
- Schritt 2: Berechne $\frac{\mathrm{d}}{\mathrm{d}b} L(W^k, b^k)$.

Nun führen wir die Gradientenschritte aufgeteilt in W- und b-Richtung aus:

- Schritte 3: Wenn W_{k+1}, der nächste Wert für W ist, der näher am Minimum liegen sollte, setzen wir $W_{k+1} = W_k - \tau \frac{\mathrm{d}}{\mathrm{d}W} L(W^k, b^k)$. Also gehen wir einen Schritt mit Schrittweite τ in die Richtung des stärksten Abstieges.
- Schritte 4: Analog setzen wir $b_{k+1} = b_k - \tau \frac{\mathrm{d}}{\mathrm{d}b} L(W^k, b^k)$. Also gehen wir ebenfalls einen Schritt vom ursprünglichen Wert für b_k aus in die Richtung des stärksten Abstiegs, also der negativen Ableitung.
- Wir setzen dann $k = k + 1$, gehen zurück zu Schritt 1 und wiederholen den Vorgang mit neuen Anfangswerten für W^k, b^k.

Der Wert `epochs`, der der Funktion `train` übergeben wird, beschreibt wie oft dieses Verfahren durchlaufen wird.

Da wir nun wissen, wie unser Netzwerk seine Werte akkurater machen kann, bleibt uns nur mehr eine Hürde übrig. Nämlich die Berechnung der Ableitungen unserer Loss-Funktion. Diese werden in den definierten Funktionen `deriv_loss` und `deriv_activation` berechnet.

Jedoch stellen auch diese schlussendlich kein großes Problem dar:

Wir betrachten in weiterer Folge immer die Berechnung der Verlustfunktion für nur einen Datenpunkt (x^i, y^i). Durch die Summenregel gilt dannach immer für alle Datenpunkte, dass:

$$L'(W,b) = \sum_{i=0}^{N} L'_i(W,b)$$

Also können wir die Ableitungen für unsere einzelnen Datenpunkte einfach aufsummieren und somit die richtige Ableitung erhalten.

Hier gilt unter Verwendung der Kettenregel dann für die Ableitung von einem Datenpunkt:

$$\frac{\mathrm{d}}{\mathrm{d}W} L_0(W,b) = \mathrm{loss}'(\sigma(Wx^0+b), y^0))\sigma'(Wx^0+b)(x^0)^\top$$

$$\frac{\mathrm{d}}{\mathrm{d}b} L_0(W,b) = \mathrm{loss}'(\sigma(Wx^0+b), y^0))\sigma'(Wx^0+b)$$

Nun bleibt noch die Ableitung der Loss-Funktion zu berechnen. Diese ist gegeben durch:

$$\mathrm{loss}(\hat{y}, y) = -(1-y)\ln(1-\hat{y}) - y\ln(\hat{y})$$

Nachdem $\frac{\mathrm{d}}{\mathrm{d}x}\ln x = \frac{1}{x}$ gilt, erhalten wir für die Ableitung unserer Loss-Funktion nach $\hat{y}$:

$$\frac{\mathrm{d}}{\mathrm{d}\hat{y}}\mathrm{loss}(y, \hat{y}) = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}$$

Dies ist dann auch der exakte Inhalt unserer Funktion `deriv_loss`.

Wir müssen noch $\sigma'(Wx+b)$ berechnen.

Dazu verwenden wir folgenden Trick. Man bemerke, dass

$$\sigma'(Wx+b) = \sigma'(\sigma^{-1}(\sigma(Wx+b)))$$

Hier ist σ^{-1} die Umkehrfunktion von σ.

Die Definition einer Umkehrfunktion lautet wie folgt:

Angenommen für alle $y \in (a,b)$ existiert genau ein $x \in \mathbb{R}$ sodass gilt $f(x) = y$. Dann definiert man die Umkehrfunktion durch $f^{-1}(y) = x$ für $y \in (a,b)$ und das eindeutig bestimmte $x \in \mathbb{R}$ sodass $f(x) = y$.

In unserem Fall ist $z = \sigma(Wx+b)$ gleich dem Output unserer `think`-Funktion, daher können wir die Ableitung als $\sigma'(\sigma^{-1}(z)) = z(1-z)$ berechnen. Dieser Teil der gesamten Ableitung der Loss-Funktion steht in unserem Code in der Funktion `deriv_activation`.

In unserer `train`-Funktion definieren wir:

$$\delta = \mathrm{loss}'(\sigma(Wx^0+b), y^0))\sigma'(Wx^0+b)$$

Im Code wird $\mathrm{loss}'(\sigma(Wx^0+b), y^0))$ durch `deriv_loss` und $\sigma'(Wx^0+b)$ durch `deriv_activation` übergeben.

Somit ergibt sich für unsere Ableitungen lediglich:

$$\frac{\mathrm{d}}{\mathrm{d}W} L_0(W,b) = \delta(x^0)^\top$$

$$\frac{\mathrm{d}}{\mathrm{d}b} L_0(W,b) = \delta$$

wobei $(x^0)^\top$ die errechneten Werte unserer `think`-Funktion sind, die im Code als `y_pred` bezeichnet werden.

Da wir nun endlich alle Komponenten für den Gradientenschritt bereit haben, wird nun im letzten Teil der `train`-Funktion der Gradientenschritt mit den berechneten Werten für W und b durchgeführt.

Stochastisches Gradientenverfahren

Nun bleibt noch eine letzte Optimierung zu beschreiben, die wir in unserem Programm noch hinzugefügt haben. Wir betrachten nun unsere Kostenfunktion in reskalierter Form

$$L(W,b) = \frac{1}{N+1}\sum_{i=0}^{N} L_i(W,b)$$

Es gilt

$$\frac{\mathrm{d}}{\mathrm{d}W} L(W,b) = \frac{1}{N+1}\sum_{i=0}^{N} \frac{\mathrm{d}}{\mathrm{d}W} L_i(W,b)$$

Wir nehmen B der Länge K und approximieren unseren Gradienten wie folgt

$$\frac{\mathrm{d}}{\mathrm{d}W} L(W,b) \approx \frac{1}{K}\sum_{i \in B} \frac{\mathrm{d}}{\mathrm{d}W} L_i(W,b)$$

Somit bekommen wir einen approximierten Gradienten, der weniger Rechenzeit benötigt und sich dem Gradienten über alle Daten recht gut annähert.

Angenommen wir verwenden eine `batch_size` der Größe `n_inputs / n_batches`. Dann zerteilen wir unsere Trainingsset in eine Anzahl von `n_batches` Mengen $B_0, \dots B_{\mathrm{n_{batches}}}$ der Länge `batch_size`

Mit diesen Teilmengen unserer Inputs verwenden wir das stochastische Gradientenverfahren:

1. Wir wählen W_0, b_0 und die Schrittweite $\tau > 0$
2. Für $k = 0, \dots$ `n_epochs`
3. Wir wählen $B_0, \dots B_{\mathrm{n_{batches}}}$
4. Für $i = 0, \dots \mathrm{n_{batches}}$
   - Wir berechnen $\frac{\mathrm{d}}{\mathrm{d}W} L_{B_i}(W, b)$, $\frac{\mathrm{d}}{\mathrm{d}b} L_{B_i}(W, b)$
   - Dann setzen wir $W_k = W_k - \tau \frac{\mathrm{d}}{\mathrm{d}W} L_{B_i}(W, b)$
   - und analog für unsere Bias-Werte $b_k = b_k - \tau \frac{\mathrm{d}}{\mathrm{d}b} L_{B_i}(W, b)$
5. Dann wählen wir $W_{k+1} = W_k, b_{k+1} = b_k$ und kehren zu Schritt 2 zurück.

Obwohl die Berechnung der Gradientenapproximation billiger ist, ist der Rechenaufwand für eine Epoche trotzdem ungefähr gleich. Simultan machen wir aber auch mehr als einen Gradientenschritt pro Epoche, was das stochastische Gradientenverfahren von der Rechenleistung günstiger macht.

Trainingsfunktion

Nun definieren wir unser Netzwerk (`mein_netzwerk`).

```
dimensions = [output_dim,input_dim]

mein_netzwerk = NeuralNetwork(dimensions)

n_iterations = 500 #wie oft das Programm die for-Schleife durchläuft

my_loss = np.zeros(n_iterations)      #Werte unserer Loss-Funktion
my_loss_test = np.zeros(n_iterations) #Werte der Loss-Funktion für die Testdaten
acc = np.zeros(n_iterations)          #Genauigkeit
acc_test = np.zeros(n_iterations)     #Genauigkeit

for i in range(n_iterations):
  # Trainieren des Netzwerks
  mein_netzwerk.train(train_inputs,train_labels, stepsize =0.05, epochs = 1, batch_size=50)
  # Vorhersagen treffen
  predictions = mein_netzwerk.think(train_inputs) # input -> sigma ( W input + bias )

  predictions_test = mein_netzwerk.think(test_inputs) # input -> sigma ( W input + bias )

  # Loss Funktion auswerten
  my_loss[i] = mein_netzwerk.loss(predictions, train_labels)
  my_loss_test[i] = mein_netzwerk.loss(predictions_test, test_labels)

  # Anzahl der korrekt kategorisierten Punkte / Gesamtpunkte
  acc[i] = np.sum((predictions >= 1/2) == train_labels) / len(predictions)
  acc_test[i] = np.sum((predictions_test >= 1/2) == test_labels) / len(predictions_test)
```

Nach dem Trainieren unseres Netzwerks berechnen wir die vorhergesagten Kategorien unserer Trainings- und Testdaten (`predictions` und `predictions_test`). Diese sind einfach die Ergebnisse der `think`-Funktion für die jeweiligen Datensets. Wir speichern für jeden Durchlauf auch die Werte der Loss-Funktion `my_loss` (bzw `my_loss_test` für die Testdaten) und die Genauigkeit unseres Netzwerkes. Diese Genauigkeit wird berechnet in dem wir die Anzahl der korrekt kategorisierten Daten durch die Gesamtanzahl der Daten dividieren. Somit erhalten wir den relativen Anteil der korrekten Ergebnisse, die wir für unsere Trainingsdaten im Array `acc` (vom Englischen "accuracy" für Genauigkeit) und für unsere Testsdaten im Array `acc_test` speichern.

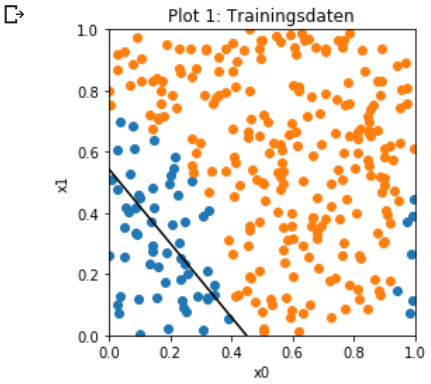
Im Anschluss plotten wir unsere Trenngerade (Plot 1). Diese Gerade zeigt, dass das Programm das Koordinatensystem in 2 Bereiche einteilt. Das Programm teilt alle Punkte im linken unteren Dreieck der Kategorie 0 zu. Dem Rest wurde die Kategorie 1 eingeteilt. Wie man sieht stimmt es nicht zu 100%, jedoch ist trotzdem ein Großteil der Punkte richtig kategorisiert, was auch unser Ziel war.

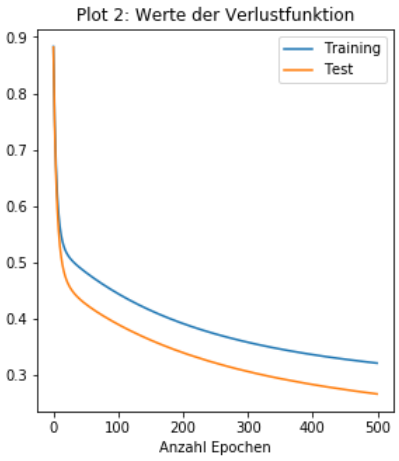
Im Plot 2 sieht man das Kleinerwerden der Verlustfunktionen im Laufe der Epochen. Also werden hier die Werte von `my_loss` und `my_test_loss` abgebildet.

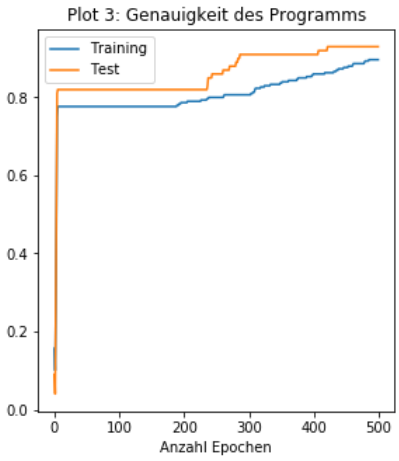
Im dritten und letzten Plot wird die Genauigkeit des Systems aufgezeichnet. Wir plotten hier die Werte der Arrays `acc` und `acc_test`.

```
weights = mein_netzwerk.weights
bias = mein_netzwerk.bias
```

```
x0_values = np.linspace(0,1,100)
x1_values = -(weights[0,0] / weights[0,1]) * x0_values - bias / weights[0,1]

plot_datapoints(train_inputs,train_cat,'Plot 1: Trainingsdaten')
plt.plot(x0_values, x1_values, color = 'black')
plt.xlim(0,1)
plt.ylim(0,1)
plt.xlabel('x0')
plt.ylabel('x1')
plt.show()

plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.plot(my_loss, label = 'Training')
plt.plot(my_loss_test, label = 'Test')
plt.title('Plot 2: Werte der Verlustfunktion')
plt.xlabel('Anzahl Epochen')
plt.legend()

plt.subplot(1,2,2)
plt.title('Plot 3: Genauigkeit des Programms')
plt.plot(acc, label = 'Training')
plt.plot(acc_test, label = 'Test')
plt.legend()
plt.xlabel('Anzahl Epochen')
plt.show()
```







4 Lernen mit unseren Daten

Erkennen unserer Gesichter

Einführung

In diesem Notebook wollen wir ein neuronales Netzwek kreiiren, das in der Lage ist 7 Personen von einander zu unterscheiden. Zusätzlich sollen auch 4 verschiedene Gesichtsausdrücke erkannt werden. Um dieses zu erzeugen und zu trainieren verwenden wir tf.keras als Schnittstelle zu tenserflow.

Information zu Beginn

In den Textzellen finden Sie kurze Informationen zu der Vorgangsweise. Genauer werden unsere Schritte in Kommentaren im Code erklärt.

Zuerst wechseln wir in den Ordner in dem sich unsere Daten befineden. Danach werden die im nachhinein benötigten Programmbibliotheken importiert.

```
from google.colab import drive
# Importieren des Moduls drive von google.colab
drive.mount('/content/drive/')
# Einbinden des default Google Drive Ordners
# Mittels cd (= change directory) wechseln wir nun in unseren drive Ordner
%cd '/content/drive/My Drive/'
!pwd
# zeigt an in welchem ordner wir uns befinden (pwd = print working directory)
!mkdir MW
# erzeugt den Ordner MW
%cd 'MW/converted6464/'
```

```
Drive already mounted at /content/drive/; to attempt to forcibly remount, call drive.mount("/content/drive/", force_remount=
/content/drive/My Drive
/content/drive/My Drive
mkdir: cannot create directory ‘MW’: File exists
/content/drive/My Drive/MW/converted6464
```

```
try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass

import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
```

Importieren der Fotos

Zum Trainieren und Evaluieren eines neuronalen Netwerks benötigt man in vielen Fällen eine große Menge an Daten. In unserem Fall wollten wir erreichen, dass das Programm die 7 Mitglieder unserer Gruppe erkennen und den Bildern auch die Kategorie "glücklich", "traurig", "wütend" oder "überrascht" zuordnen kann. Aus diesem Grund schossen wir von jedem jeweils 80 verschiedenen Bilder. Das ergab 20 Bilder pro Kategorie (der Gesichtsausdrücke/ Emotionen) pro Person. Um die Sache noch ein wenig zu erschweren, veränderten wir gelegentlich auch unsere Frisuren. Anschließenden konvertierten wir die Bilder auf die Pixelanzahl 64 x 64 und skallierten sie grau. Neben einer Liste von Bildern gibt es auch eine Liste von Labels, deren i-ter Eintrag die auf dem i-ten Bild abgebildete Person ist.

```
import os
import matplotlib.image as mpimg

lst = os.listdir()    #ladet Liste der Namen aller files in dem aktuellen Folder
lst.sort()            #sortiert die files nach Name

n_images = len(lst)   #Bilderanzahl
```

```
p = 64                #Bilder sollten aus 64 x 64 pixeln bestehen

data = np.zeros((n_images, p, p)) #erstellt ein Array

i = 0                 #Schleifenvariable/ Startwert
for filename in lst:
                      #if  filename.endswith("g"):
    aux = mpimg.imread(filename)

    data[i,:,:] = np.rot90(aux,k=-1, axes=(0, 1))

    i = i+1
```

Hier kann man die Daten testen, indem man die Matrix und 7 Bilder ausgeben lässt.

```
#print(data.shape) #Array wird ausgegeben

#n_rows = 3        #Reihenanzahl
#n_columns = 3     #Spaltenanzahl
#n_fig = 7         #Blideranzahl

#plt.figure()      #Ausgabe
#for i in range(n_fig):
#    plt.subplot(n_rows,n_columns,i+1)
#    plt.imshow(data[i*80], cmap=plt.cm.gray) #Ausgabe
```

Generieren von Labels für unsere Daten

Wir erstellen zwei Listen der Namen aller Files (eine mit dem Namen der Personen und eine mit den Emotionen) und sortieren diese auch danach. Wir definieren die Anzahl der Bilder mit Hilfe der Länge der Liste. Als nächstes ordnen wir unseren Daten die richtigen Labels(Beschriftungen) zu und definieren einige Variablen, die später benötigt werden.In diesem Fall erstellen wir 7 Klassen mit unseren Namen, die wir in Bilder nachstellen. Nacher definieren wir 4/5 der gesammten Bilder mit den dazugehörigen Labels als Trainingsdaten und die restlichen 1/5 der Daten als Testdaten.

```
class_names_person = ['Selina', 'Eva', 'Larissa', 'David', 'Raphael', 'Gernot', 'Laura']
class_names_expression = ['happy', 'sad', 'mad', 'surprised']

n_persons = len(class_names_person) #Personenanzahl
n_expressions = len(class_names_expression) #Gesichtsausdrucksanzahl

persons = np.zeros((n_images,7))
expressions = np.zeros((n_images,4))

pic_pp = int(n_images / n_persons)   #Bilderanzahl pro Person
ex_pp = int(pic_pp / n_expressions)  #Anzahl der Gesichtsausdrücke pro Person


for i in range(7):
    persons[i*pic_pp:(i+1)*pic_pp,i] = 1

for i in range(7):
    for j in range(4):
        expressions[i*pic_pp + j*ex_pp : i*pic_pp+(j+1)*ex_pp,j] = 1

n_train = int (4/5 * n_images)
n_test = n_images - n_train

train_input = np.zeros((n_train, p, p))
train_expressions = np.zeros((n_train,n_expressions),dtype=int)
train_persons = np.zeros((n_train,n_persons),dtype=int)
#Array für die Trainingsdaten der Personen

test_input = np.zeros((n_test, p, p))
#Array für die die Testdaten
test_expressions = np.zeros((n_test,n_expressions),dtype=int)
#Array für Gesichtsausdrücke
test_persons = np.zeros((n_test,n_persons),dtype=int)
#Bereich für die Testpersonen


for i in range(28):
    train_input[(i*16):(i*16+16),:,:] = data[(i*20):(i*20+16),:,:]
    train_expressions[(i*16):(i*16+16),:] = expressions[(i*20):(i*20+16),:]
    train_persons[(i*16):(i*16+16),:] = persons[(i*20):(i*20+16),:]
```

```
    test_input[i*4:(i*4+4),:,:] = data[(i*20+16):(i*20+16+4),:,:]
    test_expressions[i*4:i*4+4,:] = expressions[(i*20+16):(i*20+16+4),:]
    test_persons[i*4:i*4+4,:] = persons[(i*20+16):(i*20+16+4),:]



train_labels = np.zeros(448,dtype=int) #Array mit den Trainingslabels
test_labels = np.zeros(112,dtype=int) #Array mit den Testlables
train_labels_exp = np.zeros(448,dtype=int) #Array mit den Trainingslables (exp)
test_labels_exp = np.zeros(112,dtype=int) #Array mit den Testlables (exp)
for i in range(448):
    for j in range(7):
        train_labels[i] = train_labels[i] + j*train_persons[i,j]

for i in range(448):
    for j in range(4):

        train_labels_exp[i] = train_labels_exp[i] + j*train_expressions[i,j]

for i in range(112):
    for j in range(7):

        test_labels[i] = test_labels[i] + j*test_persons[i,j]

for i in range(112):
    for j in range(4):

        test_labels_exp[i] = test_labels_exp[i] + j*test_expressions[i,j]
```

Wir erstellen den eindimensionale Array "train_lables" und füllen ihn mit den Zahlen 0,1,2,3,4,5,6 (eine Zahl pro Bild). Dabei steht jede der 7 Zahlen für eine Person.

Ebenso erstellen wir den eindimensionale Array "train_labels_exp" und füllen ihn mit den Zahlen 0,1,2,3 (eine Zahl pro Bild). Jede der 4 Zahlen steht für eine "expression".

Nach demselben Prinzip erstellen wir auch zwei Arrays für die Testdaten und füllen den Bereich test_labels wieder mit 0,1,2,3,4,5,6 und den Bereich test_lables mit den Zahlen 0,1,2,3.

Skalieren der Daten

Damit ein gutes und erfolgreiches Ergebnis wahrscheinlicher ist, müssen wir sicher stellen, dass die Daten in unseren `train_input` und `test_input` im Bereich zwischen 0 und 1 liegen und nicht zwischen 0 und 255. Aus diesem Grund müssen wir die beiden Datensets durch 255 dividieren.

```
train_input = train_input / 255
test_input = test_input / 255
```

Das Programm erlernt unsere Gesichter zu erkennen

Im folgenden Teil erstellen wir 2 Layers, die aus mehreren "Neuronen" bestehen. Dabei wählen wir für die beiden Hidden-Layers eine Neuronenanzahl von 100. Der Output-Layer hat 7 "Neuronen", da es auch nur 7 verschiedene Ergebnissmöglichkeiten gibt. Als Aktivierungsfunktion verwenden wir "relu" und das Obtimierungsprogramm "adam".

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(64, 64)),
    keras.layers.Dense(100, activation = 'relu'),
    keras.layers.Dense(7, activation='softmax')
    # h_1 mit output dimension 10 (entspricht zehn Ziffern)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])# accuracy = Anteil korrekt gelabelter Bilder
```

Wir lassen das Programm mit Hilfe der Trainings-Daten und den Trainings-Labels trainieren, sodass im bestmöglichen Fall die acurracy, also der Anteil von richtig gelabelten Bildern, steigt.

```
model.fit(train_input, train_labels, epochs=40, verbose = 0) #Trainieren unseres Programms
```

```
<tensorflow.python.keras.callbacks.History at 0x7f6077fb9160>
```

```
model.evaluate(test_input, test_labels) #Testgenauigkeit
```

```
112/112 [==============================] - 0s 720us/sample - loss: 0.0410 - accuracy: 1.0000
[0.04102227623973574, 1.0]
```

Zum Schluss lassen wir das Programm sich selbst testen. Dies tut es, indem es versucht die Test-Bilder den Test-Labels zuzuordnen.

```
def plot_image(i, predictions_array, true_label, img,class_names): #Ausgabe
  predictions_array, true_label, img = predictions_array, true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  plt.imshow(img, cmap=plt.cm.gray)

  predicted_label = np.argmax(predictions_array)
  if predicted_label == true_label:
    color = 'blue'
  else:
    color = 'red'

  plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label,class_names):
  predictions_array, true_label = predictions_array, true_label[i]
  plt.grid(False)
  plt.xticks(range(len(class_names)))
  plt.yticks([])
  thisplot = plt.bar(range(len(class_names)),predictions_array, color="#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)

  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
```

Hier kann man die korrekten Vorhersagen in blau und falsche Vorhersagen in rot plotten.

```
#predictions = model.predict(test_input)

#num_rows = 4 #Reihen
#num_cols = 4 #Säulen
#num_images = 14 #Bilderanzahl
#plt.figure(figsize=(2*2*num_cols, 2*num_rows)) #Ausgabe
#for i in range(num_images):
#  plt.subplot(num_rows, 2*num_cols, 2*i+1)
#  plot_image(i*8, predictions[i*8], test_labels, test_input,class_names_person)
#  plt.subplot(num_rows, 2*num_cols, 2*i+2)
#  plot_value_array(i*8, predictions[i*8], test_labels,class_names_person)
```

```
model_exp = keras.Sequential([
    keras.layers.Flatten(input_shape=(64, 64)),
    keras.layers.Dense(50, activation = 'relu'),
    keras.layers.Dense(50, activation = 'relu'),
    keras.layers.Dense(30, activation = 'relu'),
    keras.layers.Dense(4, activation='softmax')])

model_exp.compile(optimizer='adam', loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

```
model_exp.fit(train_input, train_labels_exp, epochs=100, verbose = 0) #Trainieren
                                                        #100 Epochen
```

```
<tensorflow.python.keras.callbacks.History at 0x7f60756486d8>
```

Evaluieren des neuronalen Netzwerkes

```
model_exp.evaluate(test_input, test_labels_exp) #Genauigkeit
```

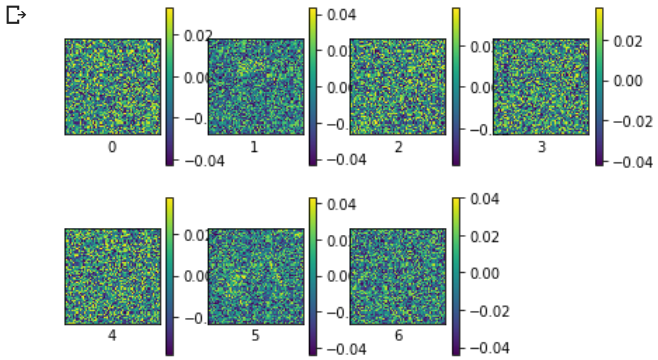
```
112/112 [==============================] - 0s 841us/sample - loss: 1.2733 - accuracy: 0.6607
[1.273307638508933, 0.66071427]
```

Wir plotten die korrekten Vorhersagen in blau und falsche Vorhersagen in rot.

```
predictions_exp = model_exp.predict(test_input)

#num_rows = 37 #Reihen
#num_cols = 4 #Spalten
#num_images = num_rows*num_cols #Bilderanzahl
#plt.figure(figsize=(2*2*num_cols, 2*num_rows)) #Ausgabe
#for i in range(num_images):
#  plt.subplot(num_rows, 2*num_cols, 2*i+1)
#  plot_image(i, predictions_exp[i], test_labels_exp, test_input,class_names_expression)
#  plt.subplot(num_rows, 2*num_cols, 2*i+2)
#  plot_value_array(i, predictions_exp[i], test_labels_exp,class_names_expression)
#plt.tight_layout()
#plt.show()
```

Visualisierung der Gewichte

```
weights = model.layers[1].get_weights()[0]


plt.figure(figsize=(7,7))
for i in range(7):
    plt.subplot(3,4,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(weights[:,i].reshape(64,64))
    plt.colorbar()
    plt.xlabel(i)
plt.show()
```



Erkennen unserere Handschriften

Ein ähnliches Notebook

Als weiteren Versuch programmierten wir auch das Notebook "Recognize our Handwriting" Hierbei verfolgten wir dasselbe Prizip wie bei dem Notebook "Erkenne unsere Gesichter". Der Unterschied zur Gesichterkennung lag darin, dass 6 Testpersonen ein bestimmtes Wort geschrieben haben und unser Datenset somit aus Handschriften bestand. Genauer schrieb jede Person 40 Wörter.

5 Recognizing our faces in the wild

Einführung

Das Ziel dieses Notebooks ist es die Gesichter von 6 Testpersonen in verschienen Umgebungen zu finden. Diese sollen mit einem Rahmen gekennzeichnet werden

Information zum Beginn

Da dieser Code ähnlich zum Code des Notebooks "Lernen mit unseren Daten" ist, können Sie dort genauere Erklärungen nachlesen.

Ordner wechseln und Einbinder von Programmbibliotheken

```
from google.colab import drive        # Importieren des Moduls drive von google.colab
drive.mount('/content/drive/')        # Einbinden des default Google Drive Ordners
# Mittels cd (= change directory) wechseln wir nun in unseren drive Ordner
%cd '/content/drive/My Drive/MW/datasetpeople_128/'
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc

Enter your authorization code:
··········
Mounted at /content/drive/
/content/drive/My Drive/MW/datasetpeople_128
```

```
%tensorflow_version 2.x
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))

import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
```

```
TensorFlow 2.x selected.
Found GPU at: /device:GPU:0
```

Laden der Fotos aus dem aktuellen Ordner

Wir laden die Foto wie bei den vorherigen Notebooks aus dem Ordner, konvertierten die Bilder auf die Pixelanzahl 128 x 128 und skallierten sie grau.

```
import os
import matplotlib.image as mpimg

lst = os.listdir()
#ladet die Liste der Namen aller files in dem aktuellen Folder
lst.sort()
# sortiert die files nach Name

n_images = len(lst)
# len() gibt mir die Anzahl der Einträge, die in der Liste enthalten sind
p = 128
# Bilder sollten aus 64 x 64 pixeln bestehen

data = np.zeros((n_images, p, p))

i = 0
for filename in lst:
    #if  filename.endswith("g"):
    aux = mpimg.imread(filename)
```

```
    data[i,:,:] = np.rot90(aux,k=-1, axes=(0, 1))

    i = i+1
```

```
%cd '/content/drive/My Drive/MW'
labels = np.loadtxt("dataset_labels_2.csv", delimiter=",")

indices = np.arange(len(labels))
np.random.seed(1)
np.random.shuffle(indices)

labels = labels[indices,:]
data = data[indices,:]

aux1 = (labels[:,1].copy() + labels[:,0].copy()) / 2
aux2 = (labels[:,1].copy() - labels[:,0].copy()) / 2
aux3 = (labels[:,3].copy() + labels[:,2].copy()) / 2
aux4 = (labels[:,3].copy() - labels[:,2].copy()) / 2

labels[:,0] = aux1
labels[:,1] = aux2
labels[:,2] = aux3
labels[:,3] = aux4

print(labels)
```

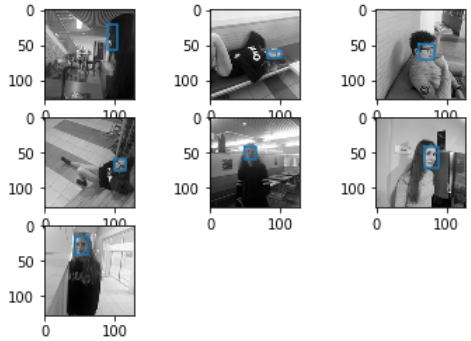
```
/content/drive/My Drive/MW
[[97.   7.  38.5 17.5]
 [91.5  9.5 63.5  4.5]
 [71.5 11.5 60.  11. ]
 ...
 [43.5  4.5 57.  13. ]
 [61.   6.  39.   7. ]
 [42.   6.  35.5  6.5]]
```

In unseren Daten befinden sich die Koordinaten der linken unteren Ecke und der rechten oberen Ecke. Diese rechnen wir in die folgenden Labelwerte um: die x-Koordinate des Boxmittelpunktes, die Hälfte der Breite der Box, die y Koordinate des Boxmittelpunktes und die Hälfte der Höhe der Box.

Wir legen die Lables für die oben angegebenen Daten fest.

```
print(data.shape)

n_rows = 3 #Reihen
n_columns = 3 #Spalten
n_fig = 7 #Bilderanzahl

plt.figure()
for i in range(n_fig):
    plt.subplot(n_rows,n_columns,i+1)
    plt.imshow(data[i], cmap=plt.cm.gray)

    xmin = labels[i,0] - labels[i,1]
    xmax = labels[i,0] + labels[i,1]
    ymin = labels[i,2] - labels[i,3]
    ymax = labels[i,2] + labels[i,3]

    plt.plot([xmin, xmax, xmax, xmin, xmin], [ymin, ymin, ymax, ymax, ymin])
```

```
(600, 128, 128)
```



Wir geben 7 Bilder zur Überprüfung aus.

```
print(n_images)
```

```
600
```

Wir geben die Anzahl der Bilder aus.

Generieren von Labels für unsere Daten

Hier teilen wir unseren Daten passende Labels zu und definieren einige Variablen, die später benötigt werden. Hier erstellen wir 6 Klassen mit unseren Namen. Ebenso definieren wir wieder 4/5 der Bilder mit den dazugehörigen Labels als Trainingsdaten und die restlichen 1/5 der Daten als Testdaten.

```
# class_names_person = ['Eva', 'Laura', 'David', 'Larissa', 'Selina', 'Raphael']

class_names_person = ['x0', 'x1', 'x2', 'x3']

n_persons = len(class_names_person)

pic_pp = int(n_images / n_persons)

n_train = int (4/5 * n_images)
n_test = n_images - n_train

train_input = np.zeros((n_train, p, p))
test_input = np.zeros((n_test, p, p))

train_labels = np.zeros((n_train,4))
test_labels = np.zeros((n_test,4))

train_pp = int (4/5 * pic_pp)
test_pp = int (1/5 * pic_pp)

for i in range(n_persons):

    train_input[i*train_pp:(i+1)*train_pp,:,:] = data[i*pic_pp:i*pic_pp+train_pp,:,:]
    test_input[i*test_pp:(i+1)*test_pp,:,:] = data[i*pic_pp+train_pp:(i)*pic_pp+train_pp+test_pp,:,:]
    train_labels[i*train_pp:(i+1)*train_pp,:] = labels[i*pic_pp:(i)*pic_pp+train_pp,:]
    test_labels[i*test_pp:(i+1)*test_pp,:] = labels[i*pic_pp+train_pp:(i)*pic_pp+train_pp+test_pp,:]
```

Skalieren der Daten

Um den Erfolg unseres Projektes zu versichern stellen wir sicher, dass die Daten in unseren Listen `train_input` und `test_input` im Bereich von 0 bis 1 liegen, statt zwischen 0 bis 255.

```
train_input = train_input / 255
test_input = test_input / 255
```

Das Programm erlernt unsere Gesichter zu erkennen

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(128, 128)),
    keras.layers.Dense(90, activation = 'relu'),
    keras.layers.Dense(80, activation = 'relu'),
    keras.layers.Dense(70, activation = 'relu'),
    keras.layers.Dense(60, activation = 'relu'),
    keras.layers.Dense(50, activation = 'relu'),
    keras.layers.Dense(40, activation = 'relu'),
    keras.layers.Dense(30, activation = 'relu'),
    keras.layers.Dense(20, activation = 'relu'),
    keras.layers.Dense(20, activation = 'relu'),
    keras.layers.Dense(20, activation = 'relu'),
    keras.layers.Dense(20, activation = 'relu'),
    keras.layers.Dense(20, activation = 'relu'),
    keras.layers.Dense(20, activation = 'relu'),
    keras.layers.Dense(20, activation = 'relu'),
    keras.layers.Dense(20, activation = 'relu'),
    keras.layers.Dense(20, activation = 'relu'),
```

```
    keras.layers.Dense(20, activation = 'relu'),
    keras.layers.Dense(4, activation=None)   # h_1 mit output dimension 4
])

model.compile(optimizer='adam',
              loss='mse',
              metrics=['accuracy'])# accuracy = Anteil korrekt gelabelter Bilder
```

Wir lassen das Programm mit Hilfe der Trainings-Daten und den Trainings-Labels trainieren, sodass im bestmöglichen Fall die acurracy, also der Anteil von richtig gelabelten Bildern, steigt.

```
model.fit(train_input, train_labels, epochs=5000, batch_size = 50, verbose=0 )
```

```
<tensorflow.python.keras.callbacks.History at 0x7f50a066c6a0>
```

```
model.evaluate(test_input, test_labels)
```

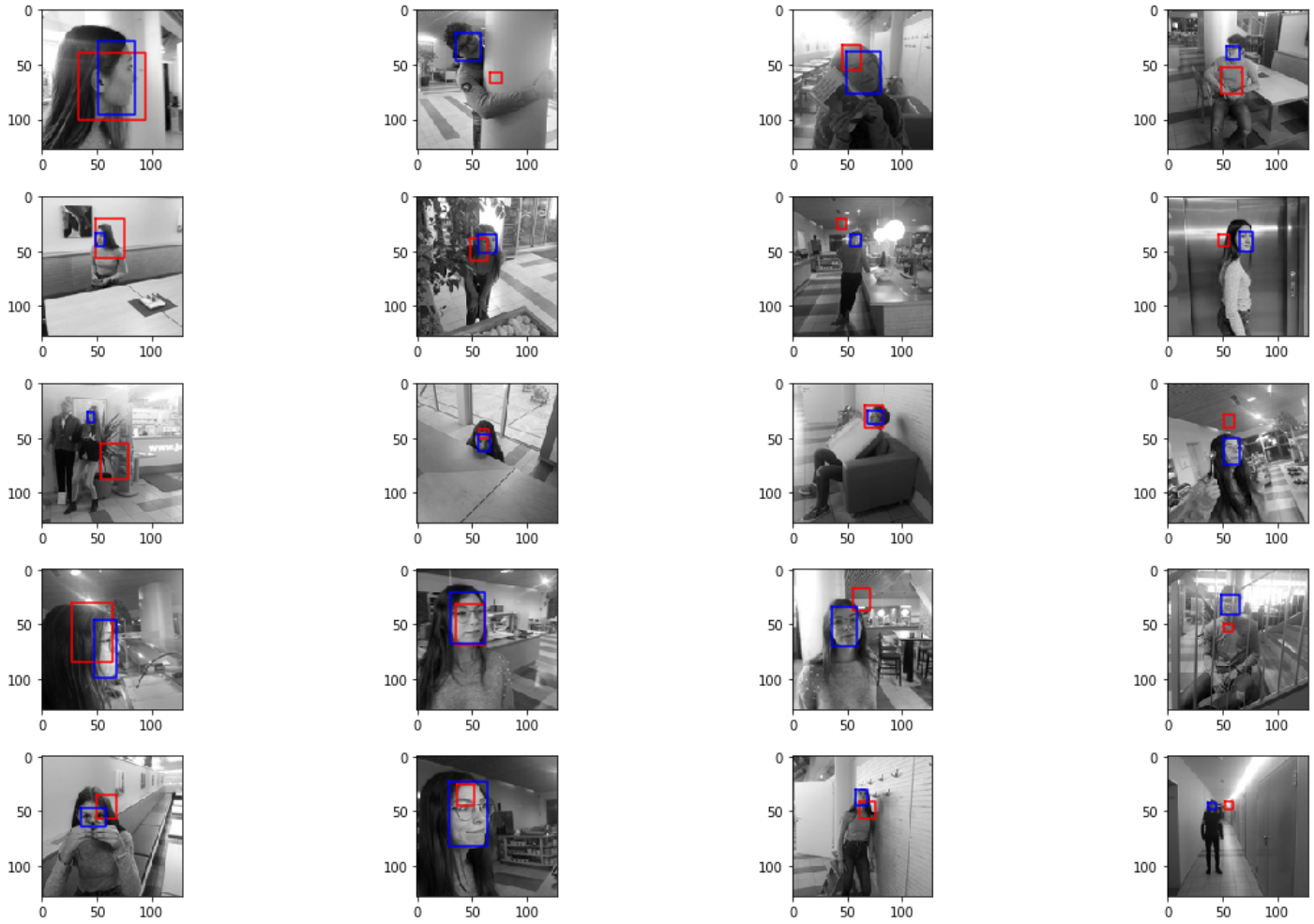
```
120/120 [==============================] - 0s 1ms/sample - loss: 179.7209 - accuracy: 0.6833
[179.72087910970052, 0.68333334]
```

Ergebnis

Nun testen wir unser Programm indem wir die vom Computer generierten Koordinaten (die unser Gesicht finden sollen) in Form von roten Boxen anzeigen lassen. Zusätzlich vergleichen wir diese mit von uns generierten blauen Boxen. So wird veraunschaulicht wie gut der Computer unsere Gesichter in verschiedensten Umgebungen erkannt hat.

```
predictions = model.predict(test_input)
print(predictions.shape)

num_rows = 5
num_cols = 4
len(predictions)

num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)

  plt.imshow(test_input[i], cmap=plt.cm.gray)


  xmin = predictions[i,0] - predictions[i,1]
  xmax = predictions[i,0] + predictions[i,1]
  ymin = predictions[i,2] - predictions[i,3]
  ymax = predictions[i,2] + predictions[i,3]

  plt.plot([xmin, xmax, xmax, xmin, xmin], [ymin, ymin, ymax, ymax, ymin], color ='red')

  xmin = test_labels[i,0] - test_labels[i,1]
  xmax = test_labels[i,0] + test_labels[i,1]
  ymin = test_labels[i,2] - test_labels[i,3]
  ymax = test_labels[i,2] + test_labels[i,3]

  plt.plot([xmin, xmax, xmax, xmin, xmin], [ymin, ymin, ymax, ymax, ymin], color ='blue')

plt.tight_layout()
plt.show()
```

```
(120, 4)
```



Ein ähnliches Notebook

Als weiteren Versuch programmierten wir das Notebook "Traffic signs" Hierbei verfolgten wir dasselbe Prizip wie bei dem Notebook "Recognizing us in the wild". In diesem Beispiel haben wir anders als bei dem vorherigen Notebook Verkehrsschilder in der Stadt fotografiert und ließen vom Programm erkennen.

Im Bann der Kräfte

Gruppe Dynamik

8.-14. Februar 2020

Betreuer: Richard Huber
Mitglieder: Elisa Scheucher, Marie Fürst, Matthias Schloffer, Jakob Irmler, Tobias Grandits und Luisa Vortisch

Inhaltsverzeichnis

1 Aufgabenstellung

Der Begriff Dynamik ist definiert als die Lehre des Einflusses von Kräften auf die Bewegungsvorgänge von Körpern. Dieser Einfluss kann beispielsweise beim Fall eines Steines von einem Turm beobachtet werden: Ohne die Schwerkraft würde der Stein beispielsweise nicht nach unten fallen. Der Luftwiderstand ist im Gegensatz dazu jene Kraft, die der Schwerkraft gewissermaßen entgegenwirkt und den Fall des Steines abbremst. Doch dieses Beispiel vom Fall eines Steines erscheint fast schon nichtig, wenn man bedenkt, dass sogar die Flugbahnen der Planeten in hohem Maße von Kräften abhängig sind. In unserer Gruppe wurden wir vor die Aufgabe gestellt, die Dynamik mithilfe der Software „Matlab“ zu untersuchen und in unterschiedlichen Beispielen möglichst realitätsnah darzustellen. Als endgültiges Ziel setzten wir es uns schließlich, die Umlaufbahnen der Planeten zu programmieren.

1.1 Umsetzung

Die Physik die hinter unserer Aufgabenstellung steckt, basiert großteils auf den Prinzipien der klassischen Mechanik, die wir mit Hilfe verschiedenster mathematischer Vorgänge beschrieben haben und somit simulieren konnten.
Um uns möglichst effizient die Grundlagen, die benötigt werden, um ein brauchbares Skript mit Matlab zu schreiben, anzueignen, haben wir uns insbesondere zu Beginn unseres Aufenthalts mit dem Verständnis der Vorgänge hinter der einzelnen Befehle beschäftigt.

2 Mechanische Grundlagen

Die mechanischen Grundideen nach Sir Isaac Newton haben bis heute in der klassischen Mechanik elementare Gültigkeit und tragen enorm zu jeglicher Forschung bei. Hinter Zusammenhängen, die im täglichem Leben vollkommen selbstverständlich scheinen, stecken komplizierte und hochinteressante Überlegungen, die helfen, das Universum ein wenig besser zu verstehen.

2.1 Die Newtonschen Axiome

Eben jene erwähnten Zusammenhänge bildeten die Grundlage für das Umsetzen der Ideen unserer Gruppe. Die Newtonschen Axiome wiederum, sind die Basis des Verstehens eines jeden mechanisch fungierenden physikalischen Systems. Sie lauten:

1. Materie ist träge. Wirken keinerlei Kräfte auf einen Körper, verbleibt er in seiner Postion oder in geradliniger, konstanter Geschwindigkeit.

2. Kraft ist Masse mal Beschleunigung, oder kurz gesagt:

$$\vec{F} = m \cdot \vec{a} \tag{1}$$

3. Actio est Reactio, Kraft gleicht Gegenkraft. Das bedeutet, dass eine jede Kraft einer Gegenkraft entspricht, die zwar in die entgegengesetzte Richtung zeigt, allerdings den selben Betrag hat. Also gilt:

$$\vec{F}_{A\to B} = -\vec{F}_{B\to A} \tag{2}$$

2.2 Zusammenhänge und Gesetzmäßigkeiten

Weitere der bereits genannten Gesetzmäßigkeiten lassen sich aus dem Verständnis klassischer Mechanik schließen. Einer der einfachsten Zusammenhänge lautet:
Geschwindigkeit gleicht der Änderung der Position in einer Gewissen Zeitspanne, also
Geschwindigkeit = Weg pro Zeitspanne

$$\vec{v} = \frac{\Delta s}{\Delta t} \tag{3}$$

Ebenso bekannt und einfach scheint der dazugehörige Zusammenhang zur Beschleunigung. Beschleunigung entspricht der Änderung der Geschindigkeit in einer gewissen Zeitspanne, also:
Beschleunigung = Änderung der Geschwindigkeit pro Zeitspanne
Beschleunigung = Weg pro Zeitspanne pro Zeitspanne

$$\vec{a} = \frac{\Delta v}{\Delta t} = \frac{\frac{\Delta s}{\Delta t}}{\Delta t} = \frac{\Delta s}{\Delta t^2} \tag{4}$$

Wird die Masse als Faktor hinzugezogen ergibt sich die bereits bei Newtons Axiomen erwähnte Formel für Kraft:
Kraft = Masse mal Beschleunigug

$$\vec{F} = m \cdot \vec{a} \tag{5}$$

Wirkt nun wiederum die Kraft längs eines Weges entsteht Arbeit, als das Skalarprodukt der Kraft und des Wegs.
Arbeit = Kraft mal Weg

$$W = \vec{F} \cdot \vec{s} \tag{6}$$

Wenn Arbeit in einer gewissen Zeispanne verrichtet wird, definiert man Leistung. Was bedeutet:
Leistung = Arbeit pro Zeitspanne = Kraft mal Geschwindigkeit

$$P = \frac{W}{\Delta t} = \vec{F} \cdot \vec{v} \tag{7}$$

2.3 Würfe und Fallvorgänge

Werden Gegenstände geworfen fallen sie auf Grund der Gravitation bekanntermaßen zu irgendeinem Zeitpunkt wieder auf den Boden. Dabei spielt die Erdbeschleunigung g eine wichtige Rolle ($g \approx 9,81 \frac{m}{s^2}$). Die Formel für die Position des geworfenen Gegenstandes, welche durch Integration herzuleiten ist, lautet:

$$s(t) = g \cdot \frac{t^2}{2} + v_0 \cdot t + s_0 \tag{8}$$

Die Richtung, also die Vorzeichen, sind sehr wichtig für das Erzielen eines akkuraten Ergebnisses.

2.4 Gravitationsfelder

In der klassischen Mechank ist das Gravitationsfeld das Kraftfeld, das durch die Gravitation von Massen hervorgerufen wird. Die Feldstärke des Gravitationsfeldes gibt für jeden Ort den durch Gravitaion verursachten Teil der Fallbeschleunigung g an. Planet A hat die Masse mA. Ein fixer Planet (Sonne) S wobei S im den Koordinatenursprung liegt und A die Position:

$$x(t) = \begin{pmatrix} x1(t) \\ x2(t) \end{pmatrix} \tag{9}$$

und die Geschwindigkeit:

$$v(t) = \begin{pmatrix} v1(t) \\ v2(t) \end{pmatrix} \tag{10}$$

Formel:

$$F_{AS} = \frac{-m_A \cdot m_S \cdot (x-0)}{\|x-0\|^3} \cdot g \tag{11}$$

$$x'' = \frac{-m_S \cdot (x-0)}{\|x-0\|^3} \cdot g \tag{12}$$

wobei [0,0] die Position der Sonne beschreibt.

3 Mathematische Grundlagen

Bevor wir mit der Modellierung physikalischer Systeme beginnen können, möchten wir im Folgenden einige mathematische Grundlagen und Techniken diskutieren, die in unserem Projekt Anwendung finden.

3.1 Infinitesimalrechnung

Die Differentialrechnung ist ein wesentlicher Bestandteil der Analysis. Sie ist eng verwandt mit der Integralrechnung, mit der sie gemeinsam unter der Bezeichnung Infinitesimalrechnung zusammengefasst wird. Zentrales Thema der Differentialrechnung ist die Berechnung lokaler Veränderungen von Funktionen. Um diese zu bestimmen, können Differenzenquotienten im Punkt x_0 mit Schrittweite h bestimmt werden:

$$\frac{\Delta f}{\Delta_x}(x_0, h) = \frac{f(x_0+h) - f(x_0)}{h}, \tag{13}$$

also dem Verhältnis von Funktionsänderung zu Positionsänderung. Lässt man dabei die Schrittweite h gegen 0 laufen, und existiert der zugehörige Grenzwert, so nennt man diesen

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0+h) - f(x_0)}{h}. \tag{14}$$

Insbesondere deckt sich dieser Begriff mit der Anschauung der Steigung einer Tangente an die Funktion, also die lokale Steigung bzw Änderung der Funktion. Ist die Funktion f überall differenzierbar, so nennt man die Funktion f' die Ableitung (oder differential) von f, welche sich für Standardfunktionen mittels wohlbekannten Rechenregeln berechnen lässt.

Differenzieren ist also das Ableiten von Funktionen, wodurch der Anstieg der Tangente, an verschiedenen Stellen des Graphs, ermittelt werden kann. Dadurch ist es möglich Aussagen über die momentane Änderungsrate einer Funktion an einzelnen Stellen des Graphen zu machen. Durch Ableitung einer Funktion erhält man wiederum eine Funktion, die die Steigung der Stammfunktion beschreibt.

In gewissem Sinne bildet das Integrieren die Umkehrfunktion zum Differenzieren. Das Integral einer Funktion auf einem Gebiet $[a, b]$ – geschrieben $\int_a^b f(t)\mathbf{d}t$ – repräsentiert dabei die Fläche unterhalb einer Funktion f. Insbesonders gilt dabei der Zusammenhang, dass für die Funktion $F(x) = \int_a^x f(t)\mathbf{d}t$ gilt $F'(x) = f(x)$. Diese Formel lässt sich für differenzierbare Funktionen g weiter verallgemeinern zu

$$g(b) - g(a) = \int_a^b g'(t)t, \tag{15}$$

das heißt, das Integral einer Funktion entspricht der Stammfunktion der oberen Stammfunktion, minus der Stammfunktion der unteren Schranke.

3.2 Kurven im Raum

Um höher dimensionale Bewegungen darstellen zu können muss die Theorie der Kurven eingeführt werden. Eine Kurve ist eine Funktion $X\colon \mathbb{R} \to \mathbb{R}^d$, das heißt für feste Zeit t hat $X(t) = (X_1(t), \ldots, X_d(t))$ insgesamt d komponentenfunktionen. Man kann sich dabei vorstellen, dass $X(t)$ die Position eines Körper darstellt, der sich mit der Zeit t durch den Raum bewegt. Im späteren Kontext von Differentialgleichungen ist aber auch abstraktere Interpretationen, beispielsweise stellt die erste Komponente den zurückgelegten Weg und die Zweite die Geschwindigkeit dar.

Man kann auch Kurven analog zu Funktionen ableiten, indem man einfach jede Komponentenfunktion differenziert, um den Tangentialvektor (oder Geschwindigkeitsvektor) zu errechnen. Dieser zeigt eine Vektor tangential zur Bewegungskurve in der momentanen Position, und beschreibt damit die Richtung in welche sich die Bahn bewegt.

In analogem Sinne ließe sich eine zweite Ableitung der Kurve, der Beschleunigungsvektor als Ableitung des Geschwindigkeitsvektors berechnen. Dieser wird insbesondere im Zusammenhang mit den Newtonschen Gesetzen interesant.

Als einfaches Beispiel für eine Kurve betrachteten wir die Funktion

$$X(t) = \begin{pmatrix} \cos(t) \\ \sin(t) \\ 0.1t \end{pmatrix} \qquad \text{für } t \in [0, 4\pi] \tag{16}$$

und errechneten die Trajektorie und erzeugten eine Animation einer Kugel die dieser Bahn folgt.

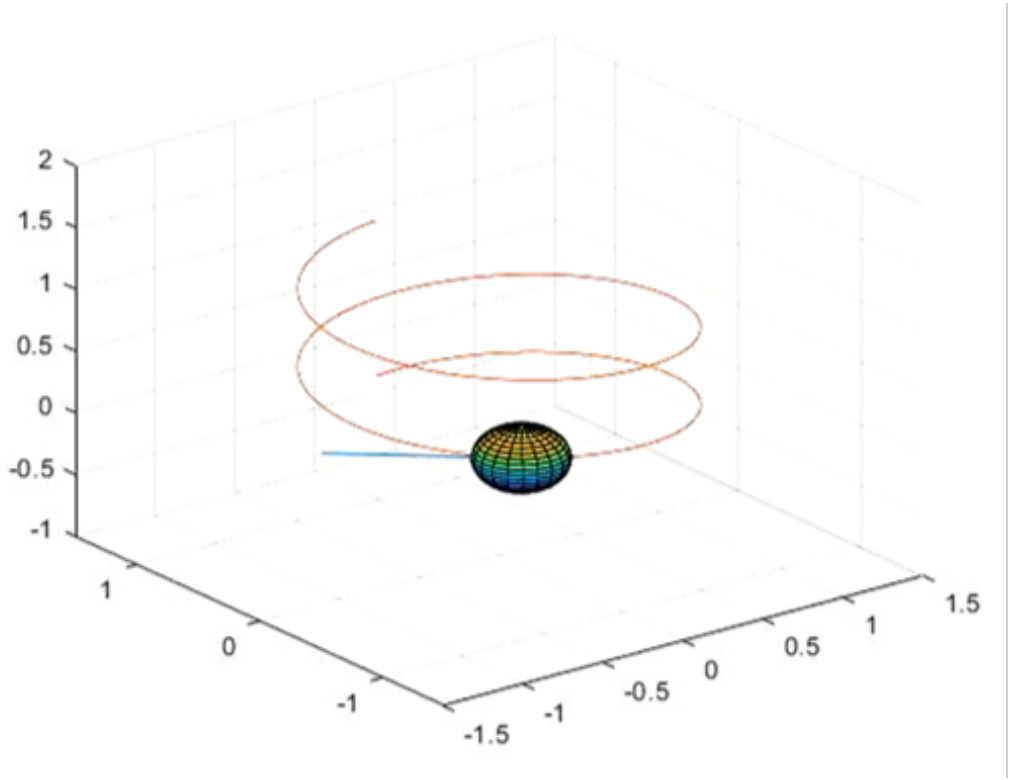
3.3 Differentialgleichungen

Eine Differentialgleichung ist eine Gleichung, in der sowohl die gesuchte Funktion, als auch eine oder mehrere ihrer Ableitungen vorkommen. Viele nätürliche, physikalische Phänomene können mit Differentialgleichungen beschrieben werden, somit sind sie ein enorm wichtiger Bestandteil jeglicher mathematischen Modellierung, wie auch wir sie betrieben haben.

Hängt die gesuchte Funktion lediglich von einer Variable ab, so spricht man von einer gewöhnlichen Differentialgleichung (Ordinary Differential Equation). Es kommen lediglich gewöhnliche Ableitungen nach der einen Veränderlichen vor. Ist die Differentialgleichung nach der höchsten Ableiung aufgelöst, das heißst wenn gilt:

$$y^{(k)}(t) = f(t, y, y', \ldots, y^{(k-1)}) \tag{17}$$

so nennt man die gewöhnliche Differentialgleichung explizit. Im Folgenden sprechen wir stets von expliziten Differentialgleichungen. Die höchste vorkommende Ableitungsordnung n wird Ordnung der Differentialgleichung genannt. Zusätzlich zu der Gleichung werden Anfangswerte $(y(0), y'(0), \ldots, y^{(k-1)}(0))$ angegeben, um sicher zu stellen, dass eindeutige Lösungen existieren, man spricht auch von einem Anfangswertproblem k.ter Ordnung.



3.3.1 Differentialgleichungen erster Ordnung

Eine Differentialgleichnung der Ordnung eins ist wird als Differentialgleichung erster Ordnung bezeichnet, das heißt

$$y'(t) = f(t, y(t)). \tag{18}$$

Diese Differentialgleichungen stellt also einen Zusammenhang zwischen der Ableitung und der Funktion her, also beispielsweise die Geschwindigkeit/Wachstum abhängig von der Funktion.

Solche Problemnstellungen umfassen den Trivialenfall $y' = g(t)$ (also rechte Seite hängt nicht unmittelbar von y ab) oder die Wohlbekannte Gleichung $y' = \lambda y$ welche exponentiellen Zerfall oder Wachstum repräsentiert.

Jedoch ist selbst das Lösen solcher Differentialgleichungen erster Ordnung (die gewissermaßen den einfachsten Fall darstellen) eine mathematische Herausforderung die oft mancherlei Kniffe erfordert, weshalb oft auf numerische Näherungsverfahren zurückgegriffen wird, Siehe 3.3.3. Insbesondere sind die dort beschriebenen Vorgangsweisen auf Systeme erster Ordnung zugeschnitten, wodurch sich die Frage stellt, wie sich diese auf Systeme höherer Ordnung verallgemeinern lassen.

3.3.2 Differentialgleichungen höherer Ordnung und Systeme erster Ordnung

Differentialgleichungen von Ordnung k tretten dann auf, wenn der Zusammenhang zwischen den Ableitungen zu komplex ist, um nur von einer Ableitung beschrieben zu werden. Jedoch ist es so, dass sich jede Differentialgleichung k.ter Ordnung in ein äquivalentes System erster Ordnung umschreiben lässt, das heißt man kann die Ordnung reduzieren, zum Preis das vektorwertig Funktionen berücksichtigt werden müssen. Genauer gilt für ein Anfangswertproblem k.ter Ordnung ein äquivalentes System $z = (z_1, \dots, z_k)^T : \mathbb{R} \to \mathbb{R}^k$ sodass

$$z'(t) = F(t, z(t)), \qquad \text{mit} \qquad F(t, z(t)) = \begin{pmatrix} z_2 \\ z_3 \\ \vdots \\ z_k \\ f(t, z1, z_2, \dots, z_k) \end{pmatrix} \qquad z(0) = \begin{pmatrix} y(0) \\ y'(0) \\ \vdots \\ y^{(k)}(0) \end{pmatrix} \tag{19}$$

welches durch die assoziation $z_1 = y$, $z_2 = y'$,...,$z_k = y^{(k-1)}$ äquivalent ist. Insbesondere bedeutet dies, dass das numerische Lösen einer Differentialgleichung stets auf das Lösen eines Systems erster Ordnung zurückgeführt werden kann, wodurch diese Methoden auch für Differenzialgleichungen höherer Ordnung zum Einsatz kommen.

Insbesonder lassen sich auch Differenzialgleichungen höherer Ordnung von Raumkurven als Systeme erster Ordnung schreiben, indem man sich ähnliche Strategien zunutze macht.

3.3.3 Approximative Lösungswege

Differentialgleichungen haben als Lösung Funktionen, die Bedingungen an ihre Ableitungen erfüllen. Eine Approximation geschieht meist, indem Raum und Zeit durch ein Rechengitter in endlich viele Teile zerlegt werden (Diskretisierung). Die Ableitungen werden dann nicht mehr durch einen Grenzwert dargestellt, sondern durch Differenzen approximiert. In der numerischen Mathematik wird der dadurch entstandene Fehler analysiert und möglichst gut abgeschätzt.
Die diskretisierte Differentialgleichung enthält keine Ableitungen mehr, sondern nur noch rein algebraische Ausdrücke. Damit ergibt sich entweder eine direkte Lösungsvorschrift oder ein lineares oder nichtlineares Gleichungssystem, welches dann mittels numerischer Verfahren gelöst werden kann.

Das explizite **Euler-Verfahren** dient zur approximativen Lösung eines Anfangswertproblems (siehe Kapitel Differenzialgleichungen) erster Ordnung, also einem Problem

$$y'(t) = f(t, y(t)) \quad \text{für } t \in [0, T] \qquad y(0) = y_0 \tag{20}$$

mit gegebenem Anfangswert y_0. Man merke dass das Verfahren und dessen Rechtfertigung für Systeme erster Ordnung exakt analog funktioniert, wir der Einfachheit halber aber nur skalare Differenzialgleichungen diskutieren.

Für den Zeitparameter $t \in [0, T]$ betrachtet man eine Diskretisierung der Zeit $(t_0, \dots, t_N)$ sodass $t_k = T \cdot k/N$ gilt, also eine Zerlegung der Zeit in gleich große Intervalle, jedoch sind auch algemeinerer Wahlen möglich.

Dank der Integrationsformel (b.z.w. Taylorformel) ließe sich für zwei Zeitpunkte t_k und t_{k+1} der Zusammenhang

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} y'(t)\mathbf{d}x = y(t_k) + \int_{t_k}^{t_{k+1}} f(t, y(t))\mathbf{d}t \tag{21}$$

herstellen, wobei man die Differentialgleichung (20) ausnutzt. Wenn wir annehmen, dass wir $y(t_k)$ kennen würden, erlaubte uns dies jedoch dennoch nicht die Berechnung von $y(t_{k+1})$ da die Formel implizit $y(t)$ enthält. Daher wird die Quadraturformel zur Auswertung eines Integrals an dem linken Punkt angewandt, um

$$y(t_{k+1}) \approx y(t_k) + \int_{t_k}^{t_{k+1}} f(t_k, y(t_k))\mathbf{d}t = y(t_k) + f(t_k, y(t_k)) \cdot (t_{k+1} - t_k) \tag{22}$$

zu erhalten. Insbesondere kann man, falls $y(t_k)$ und t_k bekannt sind die rechte Seite der Gleichung explizit ausrechnen.

Nun kennt man aber natürlich im allgemeinen auch $y(t_k)$ nicht exakt, jedoch könnte man dieses wiederum näherungsweise mit der Formel aus (22) für $k - 1$ ausrechnen. Durch iteratives Vorgehen ergibt sich folglich

$$\begin{cases} y_{k+1} = y_k + f(t_k, y_k)(t_{k+1} - t_k) & \text{für } k \in \{0, \dots, N-1\}, \\ y_0 = y_0, \end{cases} \tag{23}$$

wobei jeweils y_k eine Approximation zu $y(t_k)$ darstellt.

Folglich lassen sich mit diesem Verfahren näherungsweiße die Lösung der Differentialgleichung y darstellen, wobei bei jedem Schritt ein Fehler gemacht wird, da die tatsächliche Steigung der Funktion im Intervall $[t_k, t_{k+1}]$ durch die Steigung zum Zeitpunkt t_k und Zustand y_k ersetzt wird. Allerdings lässt sich dieser Fehler durch größeres N – kleiner Zeitschritte– reduzieren um für hinreichend großes N eine vernünftige Rekonstruktion zu erhalten.

3.4 Matrizen und Vektoren

Unter einer Matrix versteht man grundsätzlich die rechteckige Anordnung von meist mathematischen Elementen. Sie sind die Essenz der linearen Algebra, kommen allerdings in den meisten Bereichen der Mathematik vor. Man kann sie sich auch als eine Auflistung von errechneten oder gegebenen Werten vorstellen, so werden sie auch von Matlab gehandhabt.
Allgemein haben $m \times n$ Matrizen folgende Form:

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \tag{24}$$

Matrizen werden in Matlab folgendermaßen geschrieben:

```
A =[1,2,3;2,3,4;4,5,6]
```

demnach gelte

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 4 & 5 & 6 \end{pmatrix} \tag{25}$$

Auch Vektoren sind Matrizen, die eben entweder nur eine Zeile, oder eine Spalte haben. Ein Vektor kann als Position im Raum der entsprechenden Dimension vorgestellt werden, beziehungsweiße als Pfeil der den Koordinatenursprung 0 mit der Position verbindet.

Spaltenvektoren haben allgemein folgende Form:

$$\vec{v}_s = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \tag{26}$$

Zeilenvektoren dahingegen:

$$\vec{v}_z = \begin{pmatrix} v_1 & v_2 & \cdots & v_n \end{pmatrix} \tag{27}$$

Vektoren werden in Matlab forgendermaßen geschrieben:

4 Matlab

4.1 Grundlagen

Matlab ist eine 1970 entwickelte Software, die zur Lösung und graphischen Darstellung von mathematischen Problemen dient. Im Folgenden wollen wir einen kurzen und prägnanten Überblick über die Grundfunktionen von Matlab geben.
Da Matlab zu Beginn der Modellierungswoche für uns alle noch sehr neu war, mussten wir zunächst die Grundlagen dieser Software erarbeiten. Nachfolgend beispielweise der Lösungsweg einiger Grundrechnungsarten:
$x = [1, 2]$ (Zeilenvektor mit den eintzrägen 1 und 2)
$y = [3, 4]$ (Zeilenvektor mit den eintzrägen 3 und 4)
$a = x + y$ (Addition zweier Zeilenvektoren)
$b = x - y$ (Subtraktion zweier Zeilenvektoren)
$c = x \cdot y'$(Zeilenvektor x wird mit Spaltenvektor y multipliziert, der Apostroph transponiert "verwandelt". Es ist nicht möglich, in Matlab zwei Vektoren in gleicher Schreibweise zu multiplizieren.)

In der Command-Zeile von Matlab wird daraufhin sofort die Lösung dieser Gleichungen angezeigt. Die folgenden Zeilen des Skripts stellen eine if-Funktion dar (auf Deutsch also eine sogenannte wenn-Funktion). Wenn der absolute Wert von c größer ist als das Produkt der beiden Vektoren x und y, dann soll in der Command-Zeile „yes“ geschrieben werden. (Betrag von x = $\sqrt{5}$, Betrag von y=$\sqrt{5}$, c=11. $5 < 11$ =“yes“)

```
if(abs(c)<=norm(x)*norm(y))
display("yes")
end
```

Als nächsten Schritt näherten wir uns dann langsam der höheren Mathematik an und haben uns mit dem Thema Matrizen beschäftigt. Eine Matrix kann man sich als eine Art Tabelle vorstellen, in der eine bestimmte Menge von Zahlen systematisch angeordnet ist. Im Folgenden die Erstellung einer 10×10-Matrix (also eine Matrix mit 10 Zeilen und 10 Spalten). Dabei soll die Matrix mit einer Zahlenreihe von Brüchen ausgefüllt werden, die mit 1 beginnt und zu deren Nenner in jedem Schritt 1 addiert wird. Diese Art von Matrix wird auch als Hilbert-Matrix bezeichnet.

```
H=zeros(10,10)

for zeile=1:10
    for spalte=1:10
   H(zeile,spalte)=1/(zeile+spalte-1)
    end
end
```

Das Endresultat sieht dann so aus:

$$H = \begin{pmatrix} 1.0000 & 0.5000 & 0.3333 & 0.2500 & 0.2000 & 0.1667 & 0.1429 & 0.1250 & 0.1111 & 0.1000 \\ 0.5000 & 0.3333 & 0.2500 & 0.2000 & 0.1667 & 0.1429 & 0.1250 & 0.1111 & 0.1000 & 0.0909 \\ 0.3333 & 0.2500 & 0.2000 & 0.1667 & 0.1429 & 0.1250 & 0.1111 & 0.1000 & 0.0909 & 0.0833 \\ 0.2500 & 0.2000 & 0.1667 & 0.1429 & 0.1250 & 0.1111 & 0.1000 & 0.0909 & 0.0833 & 0.0769 \\ 0.2000 & 0.1667 & 0.1429 & 0.1250 & 0.1111 & 0.1000 & 0.0909 & 0.0833 & 0.0769 & 0.0714 \\ 0.1667 & 0.1429 & 0.1250 & 0.1111 & 0.1000 & 0.0909 & 0.0833 & 0.0769 & 0.0714 & 0.0667 \\ 0.1429 & 0.1250 & 0.1111 & 0.1000 & 0.0909 & 0.0833 & 0.0769 & 0.0714 & 0.0667 & 0.0625 \\ 0.1250 & 0.1111 & 0.1000 & 0.0909 & 0.0833 & 0.0769 & 0.0714 & 0.0667 & 0.0625 & 0.0588 \\ 0.1111 & 0.1000 & 0.0909 & 0.0833 & 0.0769 & 0.0714 & 0.0667 & 0.0625 & 0.0588 & 0.0556 \\ 0.1000 & 0.0909 & 0.0833 & 0.0769 & 0.0714 & 0.0667 & 0.0625 & 0.0588 & 0.0556 & 0.0526 \end{pmatrix} \tag{28}$$

Mit den sogenannten For-Schleifen ist es außerdem möglich, in Matlab interessante Animationen zu unterschiedlichen Vorgängen zu erzeugen. Unsere erste For-Schleife haben wir erstellt, um das schon vorhin erwähnte Beispiel vom Fall eines Steines von einem Turm nachzustellen. Dabei haben wir zunächst eine $N \times 1$-Matrix erstellt, die den Weg S in Abhängigkeit von der Zeit darstellen soll. Der Fall dauert insgesamt T=2 Sekunden und jeder N=1000ste Schritt des Vorganges soll in der Animation gezeigt werden. Es entsteht ein Plot bestehend aus einer schnellen Aufeinanderfolge von 1000 einzelnen Ausschnitten dieses Prozesses. Die For-Schleife läuft folgendermaßen ab: k beschreibt die Zahlenmenge zwischen 1 und N. Die Zeit t hängt von k, T und N ab. Sobald t in der Schleife festgelegt ist, wird daraus mithilfe der zuvor festgelegten Zeit-Weg-Funktion S(k)=myfunction der Weg berechnet und gleich darauf in einer Graphik abgebildet. Diesen Vorgang bezeichnet man als plotten. Wenn der Wert dann geplottet wurde, läuft die For-Schleife noch einmal von vorne ab, nur dass k jetzt um einen Wert höher ist und in der Folge der Weg zum nächstgrößten Zeitabschnitt berechnet und geplottet.

```
function[S]=Wurf2(h,v0)
T=2
```

```
N=1000
g=-9.81
S=zeros(N,1);

for k=1:N
    tk(k)=(k-1)*T/N;
   S(k)=myfunction(h,v0,g,tk(k));
end
plot(tk,S,'r+','LineWidth',10)
end
```

4.2 Animationen

Will man einen bestimmten Vorgang, z.B. wie ein von einer Feder hängendes Gewicht auf und ab springt, in Maltlab animiert simulieren, muss man zuerst alle Werte der Simulation ausrechnen, bevor die Animation beginnt. Im Falle unseres Beispiels wollten wir einen Ball simulieren, wie er in einem zweidimensionalen Koordinatensystem auf- und abspringt. Dazu mussten wir sämtliche Höhen, die der Ball im Laufe des Zeitintervalls einnimmt, ausrechnen. Dann wurden alle Positionen, die der Ball einnimmt, nacheinander geplottet und als Bild gespeichert. Spielt man alle diese Bilder hintereinander ab, erhält man eine Videoanimation.
Dasselbe gilt für andere Simulationen. In einem anderen Beispiel folgt eine Sphäre einer dreidimensionalen Bahn. Wir rechnen alle Werte der Bahn aus, um dann den Mittelpunkt der Sphäre nacheinander an der Bahn entlang zu platzieren. Die einzelnen Bilder werden zusammen in einer Videodatei gespeichert. Zusätzlich zur Sphäre sieht man noch die Trajektorie sowie den Richtungsvektor zu jedem Zeitpunkt.

5 Das Masse-Feder

In dem Masse-Feder System betrachtet man unter idealisierten Bedingungen die Bewegung der eines Gewichts (Metalballes) das an einer Feder hängt und auf das die Schwerkraft wirkt. Folglich sind die relevanten Kräfte die Schwerkraft sowie die Federkraft, welche zusammen die Gesamtkraft darstellen. Wir nehmen daher ein 1-dimensionales Sytem x an, indem der Abstand von dem Aufhängepunkt dargestellt wird wobei die Achse nach unten zeigt. Daher lassen sich die Kräfte in diesem System als

$$F = F_G + F_F = gm - cx, \tag{29}$$

wobei $F_g = gm$ der Schwerkraft (Erdbeschleunigung mal Masse) entspricht, während $-cx$ die Federkraft wiederspieglet, die Abhängig von dem Abstand zur Aufhängung ab- oder zunimmt.

Gemäss dem zweiten Newtonschen Gesetz gilt dann

$$mx'' = F = gm - cx \qquad \Leftrightarrow \qquad x'' = g - \frac{c}{m}x \tag{30}$$

also eine Differenzialgleichung zweiter Ordnung. Bevor wir nun das besprochene explizite Eulerverfahren anwenden können um eine näherungsweise Lösung zu erhalten, müssen wir das Problem zunächst in ein System erster Ordnung umschreiben. Dem Vorgehen in obigen Kapiteln folgend betrachten wir $z = (z_1, z_2)\colon \mathbb{R} \to \mathbb{R}^2$ wobei $z_1 = x$ und $z_2 = x'$ wiederspiegelt und der Differenzialgleichung

$$z'(t) = F(t, z(t)), \qquad \text{mit } F(t,z) = \left(z_2, g - \tfrac{c}{m}z_1\right) \tag{31}$$

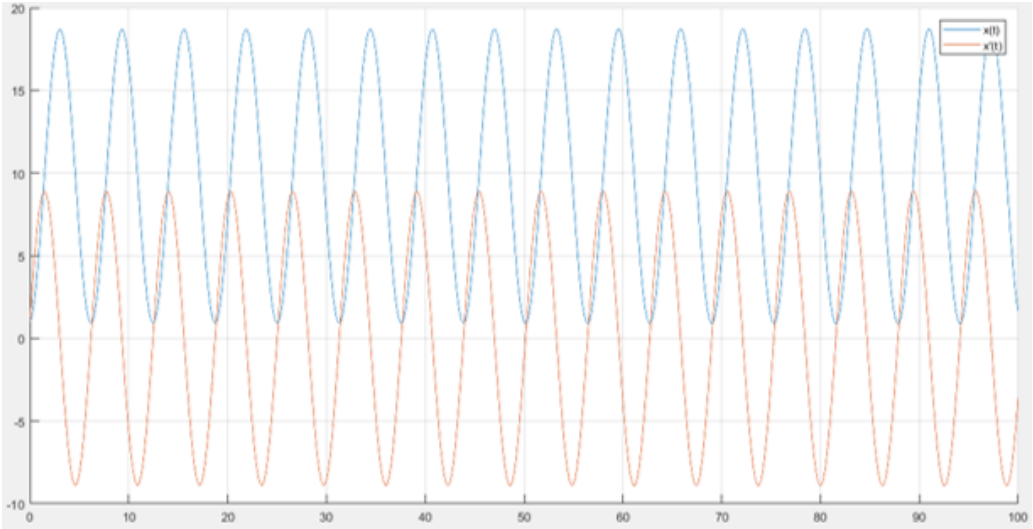
genügt.

Anschließend lösten wir das Problem numerisch mit dem expliziten Eulerverfahren gemäß der Iteration

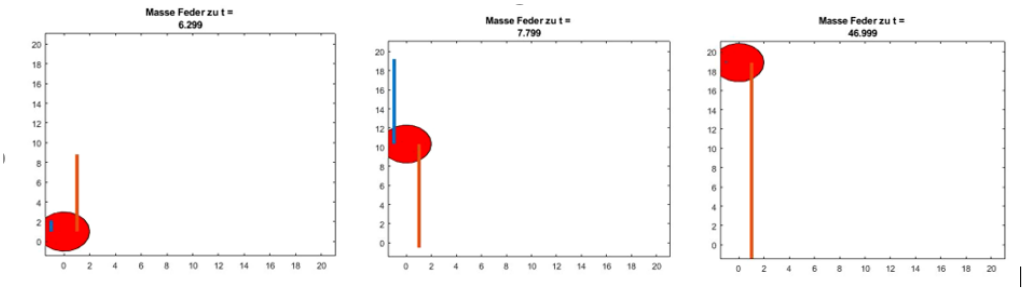
$$\begin{cases} z_{k+1} = z_k + (t_{k+1} - t_k)F(t_k, z_k) \\ z_0 = (x_0, v_0) \end{cases} \tag{32}$$

1000000 = Iterationen, wobei 1 = die Anfangsposition und 1 = die Anfangsgeschwindigkeit wiedergibt.

Wie aus dem physikalischen Verständnis zu erwarten beschreiben sowohl die Position als auch die Geschwindigkeit sinusförmige Funktionen bezüglich der Zeit, und bei Betrachtung des Systems in animiertem Zustand beobachtet man in der Tat dass die Kugel sich auf und nieder



Damit kann man insgesamt festhalten, dass die Simulation des Modells gut mit der Realität übereinstimmt. Allerdings würde die Bewegung unendlichlange weiterlaufen, und sogar auf Grund numerischer Fehler zunehmen, da keine Dämpfung oder Widerstandskräfte berücksichtigt wurden. In dieser Hinsicht ist das Modell noch imperfekt, aber man könnte auch solche Faktoren in unser Modell einbeziehen und ohne wesentlichen Mehraufwand Simulationen durchführen.



6 Planetensysteme

6.1 Beschreibung

Um so eine komplexe Systematik wie die unseres Sonnensystems darstellen zu können, braucht es zunächst ein allgemeines Grundverständnis über die wichtigsten Gesetzmäßigkeiten und Gegebenheiten, die im Universum vorzufinden sind. Das Sonnensystem umfasst insgesamt 8 Planeten: Merkur, Venus, Erde Mars, Jupiter, Saturn, Uranus und Neptun. Der Zwergplanet Pluto wurde bis zur Neudefinition des Begriffes Planet im Jahre 2006 noch als Planet bezeichnet und galt als der äußerste und kleinste Planet des Planetensystems. Aufgrund seiner kleinen Größe von gerade einmal 2374 km Radius wurde ihm aber die Bezeichnung als Planet schließlich abgesprochen.
Das Zentrum des Sonnensystems bildet klarerweise die Sonne. Mit einem Durchmesser von 1 392 684 km und einer Masse von $1,9884 \cdot 10^{30}$ kg macht die Sonne, die übrigens ein Stern ist, etwa 99.98 % der Masse des Sonnensystems aus. Die Sonnenstrahlung, die eine Grundvoraussetzung für das Leben auf der Erde ist, kommt durch Fusionen von Helium und Wasserstoff im Kern der Sonne zustande.
Die Planeten umkreisen die Sonne in ellipsenförmigen Flugbahnen. Der Planet Merkur, welcher der Sonne am nächsten ist, hat eine Entfernung von gerade einmal 46 072 000 km zur Sonne und umkreist diese somit am schnellsten, da er am meisten von der Anziehungskraft der Sonne beeinflusst wird. Der Saturn hat hingegen einen Abstand von etwa 1 430 000 000 km zur Sonne und braucht für eine Umkreisung somit ganze 29 Jahre und 166 Tage.
Aufgrund der unterschiedlichen Abstände der Planeten zur Sonne, war es für uns nicht möglich, alle Planeten gleichzeitig zu plotten und in einer Graphik sichtbar zu machen. Daher haben wir den Saturn und den Uranus weggelassen. Außerdem haben wir die Größe der Planeten optisch viel größer festgelegt und jener der Sonne dafür übertrieben klein. In einem realitätsnahen Größenverhältnis wäre bei der Simulation nahezu gar nichts herauslesbar.
Neben der Anziehungskraft der Sonne müssen allerdings auch die Kräfte zwischen den einzelnen Planeten untereinander in die Simulation miteinberechnet werden.

In der nachfolgenden Beschreibung unserer Programmierung des Planetensystems wird auf die Auswirkungen dieses Kräfteverhältnisses näher eingegangen werden.

6.2 Simulation

Unser Ziel war es, die Planeten auf ihren Bahnen um die Sonne via Matlab zu animieren. Dazu nutzten wir die Erkenntnisse aller unserer vorigen Übungen und Überlegungen. Nötig waren vor allem ein Verständnis der Kepler‘schen Gesetze, der Newton‘schen Axiome, des Expliziten Eulers und der Arbeitsweise von Matlab. Wir gingen von der Sonne als unbeweglichen Mittelpunkt aus, um die wir die Planeten kreisen ließen. Um die Simulation erstellen zu können, brauchten wir die folgenden Werte:

1.: Die Positionen der Planeten, die sie zu Beginn im dreidimensionalen
Koordinatensystem einnehmen
2.: Die Startgeschwindigkeiten der Planeten
3.: Die Massen der Planeten und der Sonne

Da wir von der Sonne als Mittelpunkt ausgingen, konnten wir ihr die Koordinaten (0 / 0 / 0) zuweisen. Ausgehend davon erhält man die Koordinaten der Planeten mithilfe deren Abstand von der Sonne (z.B. Erde: (149597870700 / 0 / 0)). Die Startgeschwindigkeiten der Planeten werden ebenso in Vektorform angegeben (z.B. Erde: (0 / 29780 / 0)). Die Massen der Planeten wurden einfach in Variablen gespeichert. Mit allen Grundwerten gespeichert, ging es dann ans Berechnen. Zuerst waren alle Gravitationskräfte, mit welchen die Himmelskörper sich beeinflussen, auszurechen. Mit den Gravitationskräften berechnet, konnten wir den expliziten Euler anwenden, um die Positionen der Planeten in der nächsten Zeiteinheit auszurechnen. Dies taten wir dann mithilfe einer Schleife so oft, bis wir die Werte aller Planeten im gesamten Zeitraum berechnet hatten.

Um dieses System zu Lösen müssen wir uns zunächst überlegen welche Kräfte auf jeden der beobachteten Planeten wirken, um anschließend anhand der Newtonschen Gesetze eine Bewegungsgleichung herzuleiten.

Gemäss obigen Beobachtungen gilt, dass die Kraft die Aufgrund von Gravitation zwischen zwei Körpern mit Positionen $A, B \in \mathbb{R}$ gegeben ist durch

$$F_{AB} = m_A m_B G \frac{(B-A)}{\|B-A\|^3}. \tag{33}$$

Nun muss man aber alle relevanten Körper und deren Gravitation berücksichtigen um die gesamt wirkende Kraft zu erhalten. Folglich Gilt falls die Positionen der Planeten mit $X_p \in \mathbb{R}^3$ bezeichnet werden und P die Menge aller Planeten darstellt dass auf den Planeten q die Kraft

$$F_q = m_S m_q G \frac{(S-X_q)}{\|S-X_q\|^3} + \sum_{p \neq q, p \in P} m_p m_q G \frac{(X_p - X_q)}{\|X_p - X_q\|^3}, \tag{34}$$

wobei m_p die zugehörigen Massen, und S die Position der Sonne bezeichnet, von der wir annehmen dass sie fest und unbeweglich ist.

Mittels der Newtonschen Gesetze ergibt sich dann

$$X_q''(t) = m_S G \frac{(S-X_q)}{\|S-X_q\|^3} + \sum_{p \neq q, p \in P} m_p G \frac{(X_p - X_q)}{\|X_p - X_q\|^3}. \tag{35}$$

Diese lässt sich für jeden Planeten q in ein 6-dimensionales System erster Ordnung verwandeln, indem zusätzlich die Geschwindigkeit V_q betrachtet wird, d.h.

$$\begin{pmatrix} X_q \\ V_q \end{pmatrix}' = \begin{pmatrix} V_q \\ m_S G \frac{(S-X_q)}{\|S-X_q\|^3} + \sum_{p \neq q, p \in P} m_p G \frac{(X_p - X_q)}{\|X_p - X_q\|^3} \end{pmatrix}. \tag{36}$$

Dieses System lässt sich nun abermals mit explizitem Euler lösen, wobei man nun alternierend die Position und Geschwindigkeit aller Planeten updated.

Dann ging es ums Formatieren. Den Planeten werden nicht maßstabsgetreue Radien zugewiesen, damit sie auf der Simulation sichtbar sind. Die Sonne ist unverhältnismäßig klein, alle anderen unverhältnismäßig groß. Wir wiesen einem jeden Planeten eine Farbe zu und erzeugten zufallsmäßig Sterne, um sie in den Hintergrund zu setzen. Ein weiteres Problem waren die Achsen. Wenn man diese nicht korrekt festlegte, konnte man die Planeten entweder überhaupt nicht sehen oder die Sphären waren so verzerrt, dass sie wie Striche aussahen. Zum plotten der Planeten verwendeten wir eine Schleife. Alle Planeten wurden an einem gewissen Zeitpunkt in ihrer Umlaufbahn geplottet, dann wurde das Program pausiert und das Bild in eine Datei gespeichert. Wenn man nun jeden Zeitpunkt im Zeitintervall durchgeht und dann alle einzelnen Bilder hintereinander abspielt, erhält man eine Videosimulation.

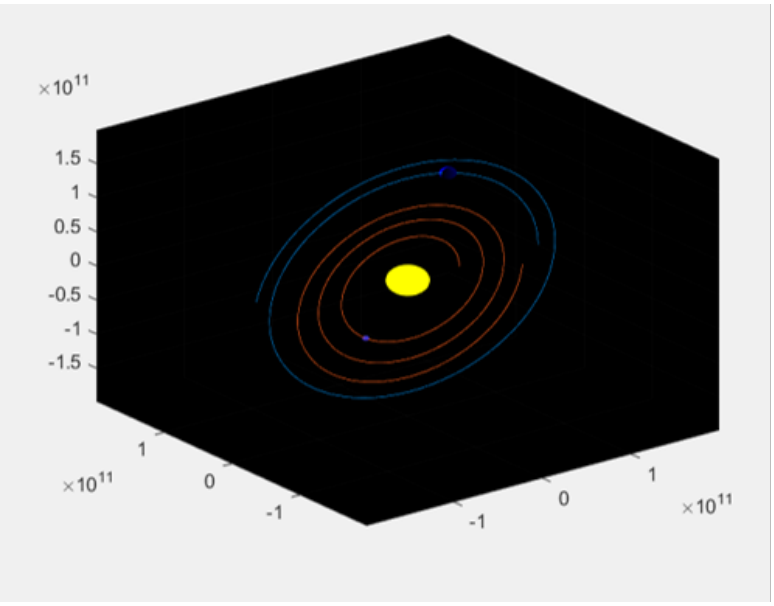


Abbildung 1: Erster Versuch nur mit Sonne, Merkur und Erde, ohne Berücksichtigung der Kräfte zwischen den Planeten

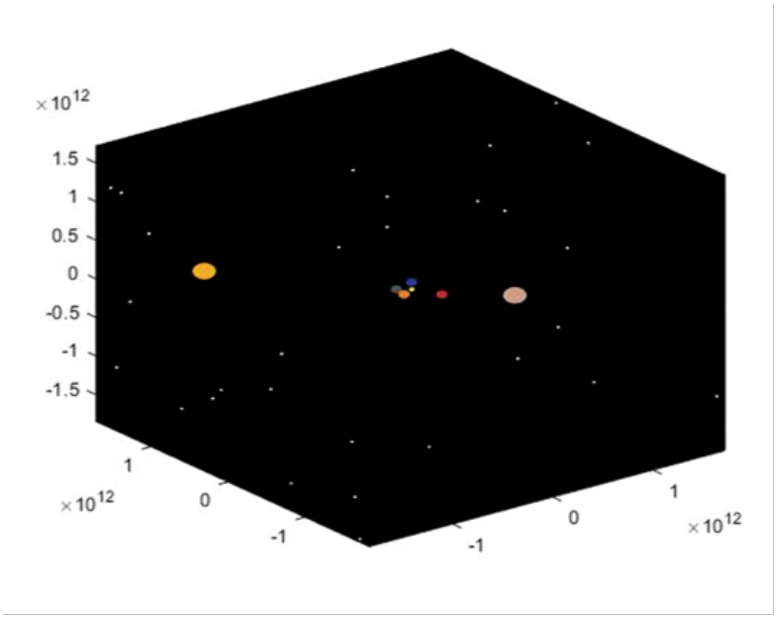


Abbildung 2: Mehrere Planeten, mit Berücksichtigung der Kräfte zwischen ihnen

6.2.1 Der Code

Dies ist unser Programm:

```
%Zuerst waren verschieden Variablen und Matrizen zu definieren:
T=1928886400
„T“ ist die Zeit in Sekunden, die wir simulieren wollen. In diesem Fall ist es die
% Umlaufzeit des Saturns um die Sonne.
N=1000000
%N“ bestimmt, dass die Simulation in N Schritte aufgeteilt wird, um sie einzeln zu
% berechen. Je größer N, desto genauer wird die Simulation.
MS=1.989*(10^30)
%„mS“ ist die Masse der Sonne
G=6.674*(10^(-11))
%„G“ ist die Gravitationskonstante, sie wird benötigt, um die Gravitationskräfte
% zwischen den Himmelskörpern zu berechnen.
P=6
%„P“ beschreibt die Anzahl der zu simulierenden Planeten.
S=[0;0;0]
%„S“ bestimmt die Koordinaten der Sonne.
AE=149597870700
%„AE“ steht für Astronomische Einheit und beschreibt den Abstand der Erde zur Sonne.
mass=zeros(P,1);
%In der Matrix „mass“ sollen später die Massen der Planeten gespeichert werden.
%Sie hat P Zeilen und eine Spalte. „zeros“ erstellt die Matrix und füllt sie mit Nullen,
%damit wir sie später mit den richtigen Werten füllen können.
t=linspace(0,T,N);
%„t“ ist der Zeitraum in unserer Funktion. t geht von 0 bis T und hat N Einträge.
Z=zeros(6,N,P);
%„Z“ ist eine dreidimensionale Matrix, die die Positionen und Geschwindigkeit
% der jeweiligen Planeten an einem Zeitpunkt im Zeitintervall t zuordnet.

%Nun sind die Anfangswerte der Planeten zu definieren:
Z(1,1,1)=0.466*AE;
%= Die x-Koordinate am 1.Punkt des Zeitintervalls des 1.Planeten
Z(2,1,1)=0;
%= Die y-Koordinate am 1.Punkt des Zeitintervalls des 1.Planeten
Z(3,1,1)=0;
%= Die z-Koordinate am 1.Punkt des Zeitintervalls des 1.Planeten
Z(4,1,1)=0;
%= Der x-Wert des Geschwindigkeitsvektors am 1.Punkt des Zeitintervalls des 1.Planeten
Z(5,1,1)=47360;
%=Der y-Wert des Geschwindigkeitsvektors am 1.Punkt des Zeitintervalls des 1.Planeten
Z(6,1,1)=0;
%=Der z-Wert des Geschwindigkeitsvektors am 1.Punkt des Zeitintervalls des 1.Planeten

Z(1,1,2)=0.728*AE;
%=Die x-Koordinate am 1.Punkt des Zeitintervalls des 2.Planeten
Z(2,1,2)=0;
Z(3,1,2)=0;
Z(4,1,2)=0;
Z(5,1,2)=35020;
Z(6,1,2)=0;

Z(1,1,3)=1.017*AE
```

```
Z(2,1,3)=0
.
.
.
Z(6,1,6)=0

%Nun werden die Planetenmassen festgelegt:
mass(1)=5.97*10^24
mass(2)=3.3*10^23
.
.

%Nun wird alles berechnet
for k=1:N-1
%Eine Schleife wird erstellt, k gibt an,
%an welcher Stelle des Zeitintervalls man sich befindet
    gravity=zeros(3,P);
%gravity ist eine Matrix, die für jeden Planteten die Gravitationskräfte speichert,
% die auf ihn einwirken.
    for pl1=1:P
%pl1 nimmt nacheinander die Werte aller Planeten an.
        othpl=1:P, othpl(pl1)=[];
%Die anderen Planeten werden in othpl gespeichert

    gravity(:,pl1)=gravity(:,pl1)-mS*G*((Z(1:3,k,pl1)-S)/(norm(Z(1:3,k,pl1)-S)^3));
%Die Gravitationskraft zwischen dem Planeten und der Sonne wird berechnet.
        for pl2=othpl
%pl2 nimmt nacheinander die Werte der anderen Planeten an
  current=Z(1:3,k,pl1)-Z(1:3,k,pl2)
           gravity(:,pl1)=gravity(:,pl1)-mass(pl2)*G*(current)/ (norm(current)^3));
%Die Gravitationskraft zwischen pl1 und pl2 wird berechnet.
        end
        Z(1:3,k+1,pl1)=Z(1:3,k,pl1)+Z(4:6,k,pl1)*(t(k+1)-t(k));
%Mittels expliziten Euler werden die nächsten Koordinaten von pl1 berechnet
        Z(4:6,k+1,pl1)=Z(4:6,k,pl1)+gravity(:,pl1)*(t(k+1)-t(k));
%Mittels expliziten Euler werden die nächsten Geschwindigkeitswerte von pl1 berechnet
    end
end

%Nun sind alle Positionen der Planeten im Zeitraum t berechnet.
%Jetzt geht es noch darum, die Planeten um die Sonne kreisen zu lassen.
r=zeros(P+1,1);
%Eine leere Liste für die Radien der Planeten P und und der Sonne(P+1) wird erstellt
r(3)=93710000000*0.5
%Der Erdradius wird festgelegt.
r(1)=r(3)
r(2)=r(3)
r.
.
r(P+1)=r(3)*0.5
%Die Radien der anderen Himmelskörper werden festgelegt. Bei den Radien ging es nicht darum,
%ob sie Maßstabsgetreu sind, sondern darum, dass sie am Ende gut erkennbar sind.
```

```
zeros(21,21,3,P)
csun=zeros(21,21,3)
CP(:,:,:,1)=color(85,85,85)
CP(:,:,:,2)=color(214,126,44)
CP(:,:,:,3)=color(56,061,150)
CP(:,:,:,4)=color(175,54,60)
CP(:,:,:,5)=color(194,150,130)
CP(:,:,:,6)=color(224,163,48)
csun(:,:,:)=color(231,199,31)
%Die Farben der Himmelskörper werden festgelegt.

uschranke=min(min(min(Z(1:3,:,1:P))))-r(P)
%Der niedrigste Wert den irgendein Planet an irgendeinem Zeitpunkt annimmt
% wird in diese Variable gespeichert.
oschranke=max(max(max(Z(1:3,:,1:P))))+r(P)
%Der höchste Wert den irgendein Planet an irgendeinem Zeitpunkt annimmt
%wird in diese Variable gespeichert.
%Die Variablen dienen später dazu, die Länge der x-, y- und z-Achsen festzulegen.

[a,b,c]=sphere()
%Erzeugt die Oberflächenkoordinaten einer Späre,
% deren Mittelpunkt auf dem Ursprung liegt.

Nstar=10
.
.
cstar(:,:,3)=ones(21);
%Dient zur Erzeugung der Sterne im Hintergrund der Graphik. Die Position wird zufällig
% berechnet, die Farbe ist weiß.

v=VideoWriter('umlaufbahnen.avi');
open(v);
%Die Speicherdatei wird geöffnet

for i=round([1:N/1000]*1000)
%Jeder tausendste Schritt wird animiert.
    aS=r(P+1)*a;
    bS=r(P+1)*b;
    cS=r(P+1)*c;
%Die Oberflächenkoordinaten der Sonne werden berechnet: Sonnenradius*Ursprungskoordinaten
    surf(aS,bS,cS,csun)
%Die Sonne wird dargestellt
    hold on
%Ohne diesen Befehl wäre immer nur ein Planet gleichzeitig zu sehen.
    for k=1:P
%k geht alle Planeten nacheinander durch
        aP=r(k)*a;
        bP=r(k)*b;
        cP=r(k)*c;
%Die Sphäre wird um den Radius des Planeten vergrößert
        aa=aP+Z(1,i,k);
        bb=bP+Z(2,i,k);
        cc=cP+Z(3,i,k);
```

```
%Die Sphäre wird auf die Position des Planeten verschoben
        surf(aa,bb,cc,CP(:,:,:,k))
%Der Planet wird dargestellt
    end
    for j=1:3*Nstar
        starx=r(1)*0.3*a+starposi(1,j);
        stary=r(1)*0.3*b+starposi(2,j);
        starz=r(1)*0.3*c+starposi(3,j);
        surf(starx,stary,starz,cstar)
    end
%Die Sterne werden im Hintergrund dargestellt
    axis([uschranke oschranke uschranke oschranke uschranke oschranke])
%Die x-, y- und z-Achsen werden festgelegt, um sicherzustellen,
% dass die Planeten nicht verzerrt dargestellt werden.
    shading interp
%Ohne diesen Befehl wären schwarze Linien auf den Sphären zu sehen
    set(gca,'Color','k')
%Der Hintergrund wird schwarz eingefärbt
    hold off
%Ohne diesen Befehl würden alle Positionen,
%die die Planeten einnehmen gleichzeitig angezeigt werden.
    frame=getframe(gcf);
    writeVideo(v,frame);
%Das aktuelle Bild wird in die Videofile gespeichert.
    pause(0.00001)
%Das aktuelle Bild wird angezeigt. Sonst würde der Rechner die ganze Simulation durchspielen,
%ohne sie anzuzeigen
end
close(v)
%Die Speicherdatei wird geschlossen.
```

Kontrolltheorie

Autonome Steuerung einer Mondlandefähre

David Kumpusch, Carolin Promitzer, Maximilian Hack, Maximilian Gepp, Jan Handler, Cora Handler

Betreut von: Florian Thaler, BSc

14. Februar 2020

Inhaltsverzeichnis

Kapitel 1

1D-Modell

1.1 Voraussetzungen

Bei dem ersten und einfachsten Modell wurde der Treibstoffverbrauch der Mondlandefähre nicht berücksichtigt, die Masse bleibt also zu allen Zeitpunkten konstant. Die gedachte Landefähre soll mithilfe einer Steuerung der Schubkraft des unteren Triebwerks sanft auf ebenem Boden landen. Sie bewegt sich dabei ausschließlich in vertikale Richtung und besitzt nur ein Antriebsmodul. Hierbei wirkt die Schubkraft der Mondanziehung (m*g) entgegen (Siehe Abb. 1.1).

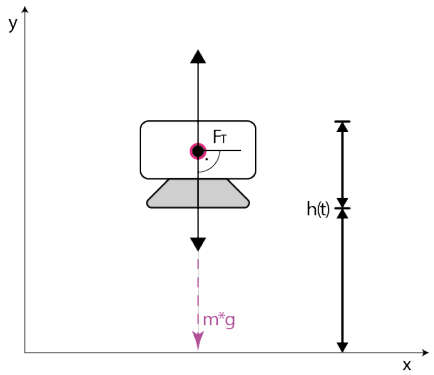


Abbildung 1.1: 1D-Modell

1.2 Ziele

Die Mondlandefähre soll mit einer möglichst niedrigen Geschwindigkeit in einer geringen Höhe (idealerweise 0) aufsetzen. Dazu musste eine Kontrollva-

riable u als Faktor der Schubkraft definiert werden, die die Antriebskraft mit abnehmender Höhe so steuert, dass die Landefähre möglichst langsam am Boden ankommt. Um unser Projekt an echte Begebenheiten so gut wie möglich anzupassen, stand uns nur dieser Faktor zur Regulierung der Schubkraft zur Verfügung. u ist hier vergleichbar mit einem Gaspedal beim Autofahren.

1.3 Wirkende Kräfte

Damit notwendige Bewegungsgleichungen aufstellbar sind, muss man sich die, auf unsere Mondlandefähre wirkenden Kräfte vor Augen führen. Auf dieses vereinfachte Modell wirkt, wie auf alle nachfolgenden ausgearbeiteten Modelle die Mondbeschleunigung mit einer Kraft von $1,62 m/s^2$ in die negative y-Richtung. Der Mondbeschleunigung kann durch Einschalten des Antriebs entgegengewirkt werden. Die Antriebskraft bezeichneten wir mit 45000 Newton, der tatsächlichen Antriebskraft des „Apollo 11 Lunar Module“.

Die von uns definierte Kontrollvariable trägt als Faktor der Schubkraft Aussage darüber, wie sehr das Potenzial des Antriebsmodul genutzt wird. In diesem Modell liegt der Wert von u im Intervall $[0, 1]$, was als 0 und 100% übersetzt werden kann. Die Kräftebilanz setzt sich folgenderweise zusammen:

$$-mg + u(t) * FT = m * a(t) \tag{1.1}$$

FT... Antriebskraft (Thrustforce) = 45000N

Die Geschwindigkeit der Landefähre zu verschiedenen Zeitpunkten:

$$v(t + \Delta t) = a(t) * \Delta t + v(t), v(t0) = v0 \tag{1.2}$$

Die Höhe, in der sich die Landefähre zu verschiedenen Zeitpunkten befindet:

$$h(t + \Delta t) = h(t) + v(t) * \Delta t, h(t0) = h0 \tag{1.3}$$

1.4 Lösungsansätze ohne PID-Control

Die Kontrollvariable u wurde in diesem Fall nach dem Prinzip entworfen, dass die negative Geschwindigkeit zunächst steigen und sich anschließend wieder dem Nullwert annähern sollte (könnte beispielsweise eine Parabelform sein). Die Höhe sollte entsprechend der negativen Geschwindigkeit zunächst stark abnehmen und sich anschließend ebenfalls langsam der x-Achse annähern, da die Geschwindigkeit weniger wird.

1.4.1 Kontrolle mithilfe der if-Abfrage

Eine simple Überlegung war es, den Landevorgang unserer Mondfähre in mehrere Zeitabschnitte, beziehungsweise Höhenabschnitte zu unterteilen, in denen das u jeweils um einen kleinen Teil verändert wird. Ein Nachteil dieser Methode war aber, dass das erzeugte Modell hier bei Veränderung der Ausgangshöhe nicht mehr funktionierte. Im Endeffekt erhält man annähernd optimale Graphen in der sich Höhe und Geschwindigkeit jeweils dem Nullwert annähern. Mithilfe von „if-clauses“, in der die Höhe jeweils auf einen Zeitabschnitt beschränkt wurde veränderten wir für jeden Abschnitt unsere Kontrollvariable $u(t)$. Ein Nachteil dieser Methode ist aber, dass die Ausgangshöhe ($h0$) hier nicht änderbar ist, da die Werte der „if“-Abfragen dann nicht mehr übereinstimmen.

Sourcecode:

```
1  import numpy as np
2  from matplotlib import pyplot as plt
3
4  m = 16400
5  g = 1.62
6
7  thrstFrc = 45000
8
9  t0 = 0

10 v0 = -1
11 h0 = 10000
12
13 dt = 1

14 i = 1
15
16 iMax  = 10000
17
18 hList =[]
19 vList = []
20 tList = []
21
22 hList.append(h0)
23 vList.append(v0)
```

```
tList.append(t0)
#h_alt = h0

#v_alt = v0

u=0.59


stop = False
while stop == False:
    a = (1 / m) * (-m * g + u * thrstFrc)

    tList.append(t0 + i * dt)
    vList.append(vList[i - 1] + a * dt)
    hList.append(hList[i - 1] + vList[i - 1] * dt)
    i = i + 1
    #if (i % 2) == 0:
        # u = 0.0
    if hList[i - 1] < 10000:
        u =0
    if hList[i - 1] < 9500:
        u =0
    if hList[i-1]<8000:
        u=0
    if hList[i - 1] < 8500:
        u = 0.2
    if hList[i-1]<8000:
        u=0.3
    if hList[i-1]<7500:
        u=0.4
    if hList[i-1]<7000:
        u=0.45
    if hList[i-1]<6000:
        u=0.55
    if hList[i-1]<5000:
        u=0.8
    if hList[i-1]<4000:
        u=0.88
    if hList[i-1]<3000:
        u=0.9
    if hList[i-1]<2000:
```

```
65            u=0.97
66        if hList[i-1]<1000:
67            u=0.9785
68        if hList[i-1]<1.2:
69            u=0
70            g=0
71            stop = True
72
73        print(i)
74
75    """
76        Visualisierung der Ergebnisse
77    """
78    fig1 = plt.figure(1)
79    plt.plot(tList, hList)
80    plt.xlabel('t')
81    plt.ylabel('h')
82    plt.grid(True)
83    plt.title('Hoehe in Relation zur Zeit')
84    fig2 = plt.figure(2)
85    plt.xlabel('t')
86    plt.ylabel('v')
87    plt.plot(tList, vList)
88    plt.grid(True)
89    plt.title('Geschwindigkeit in Relation zur Zeit')
90    plt.show()
```

Graphen und Ergebnisse:

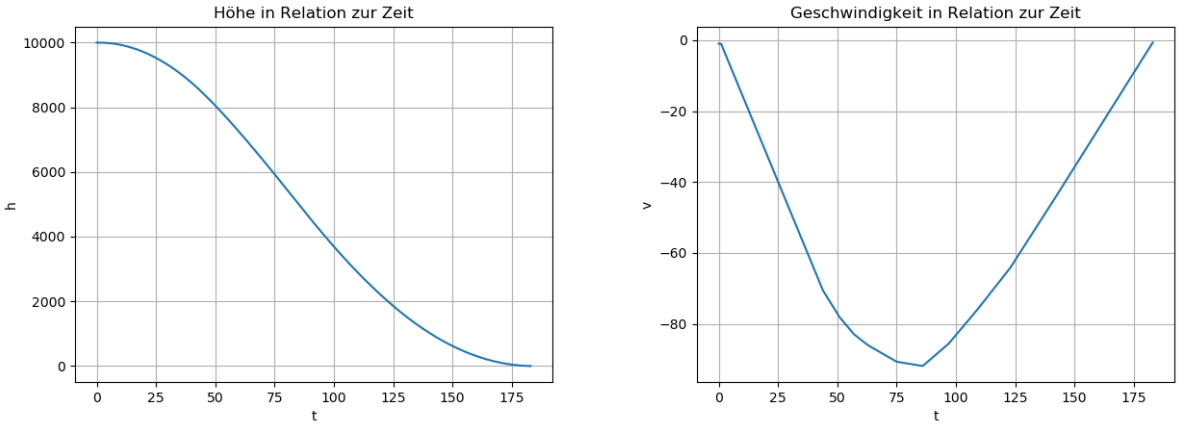
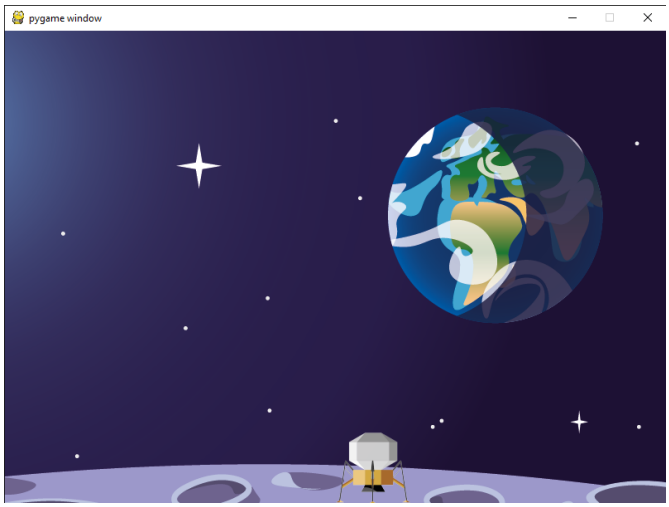


Abbildung 1.2: Kontrollgröße geregelt mit einer if-Abfrage

Mithilfe dieser Kontrollstrategie würde die Mondlandefähre mit einer Geschwindigkeit von $0,7m/s$ auf einer Höhe von $-0,1m$ landen.

1.4.2 Erweiterung mit Animation in pygame

Neben der Ausgabe von Graphen lassen sich diese Modelle in der Umgebung pygame grafisch visualisieren. Im unten angeführten Beispiel wird der Fall der Mondlandefähre durch einen PID-Regler reguliert.



Sourcecode:

```
1 import pygame
2 import time
3 import numpy as np
4
5 ACCELERATION_MOON = 1.62
6
7 FPS = 33  # Anzahl der Aktualisierungen bzw. Frames pro
      Sekunde
8 COLOR_BLACK = (0, 0, 0)
9 SCREEN_WIDTH, SCREEN_HEIGHT = 800, 600
10 LANDER_X = SCREEN_WIDTH / 2
11
12 backgroundImg = pygame.image.load('Background.png')
13
14 class LunarLander(pygame.sprite.Sprite):
15     def __init__(self):  # Konstruktor (
          Initialisierungsroutine)
16         super().__init__()  # Initialisierung der
              Basisklasse
17         self.image = pygame.transform.smoothscale(
              pygame.image.load('Sonde.png'), (90,90))
18         self.rect = self.image.get_rect()
19         self.rect.topleft = (LANDER_X, 20)
20         self.velocity = 0
21         self.posY = 20
22         self.pos0 = 20
23
24     def update(self):
25         self.rect.top += 1
26         self.physics_update()
27         self.rect.top = self.posY
28         #print(self.velocity)
29
30     def physics_update(self):
31         dt = 0.1
32         mass = 16400
33         thrust = 45000
34
35         #Bremsung des Falls durch PID-Regler
```

```
36          Kp = 30# 5
37          Ki = 40#1.4
38          Kd = 0 # 935.1
39          vTarged = -1
40          SumFehler = 0
41          FehelerAlt = 0
42
43          e = vTarged - self.velocity
44          SumFehler += e * dt
45
46          if self.posY < 200:
47              u = 0
48          else:
49              u = Kp*e + Ki*SumFehler + Kd*(e-FehelerAlt)
50              FehelerAlt = e
51
52          if u > 1:
53              u = 1
54          elif u < 0:
55              u = 0
56          print(self.velocity)
57          acceleration = (1/mass) * (-mass *
                ACCELERATION_MOON + u * thrust)
58          self.velocity += dt * acceleration
59          self.posY -= dt * self.velocity
60
61  def main():
62      screen = pygame.display.set_mode((SCREEN_WIDTH,
            SCREEN_HEIGHT))
63      fpsClock = pygame.time.Clock()
64      lander = LunarLander()
65      spritesAll = pygame.sprite.Group(lander)
66      while True:
67          if pygame.event.get(pygame.QUIT):
68              return  # Programm beenden!
69          if lander.rect.bottom < SCREEN_HEIGHT-30:
70              spritesAll.update()
71
72          #print(lander.rect.bottom)
73          screen.fill(COLOR_BLACK)  # schwarzer
                Hintergrund
```

```
74            screen.blit(backgroundImg, (0, 0))
75            spritesAll.draw(screen)
76            pygame.display.update()  # Bildschirm
                  aktualisieren
77            fpsClock.tick(FPS)
78
79    if __name__ == '__main__':  # wurde Python-Datei als
          Skript gestartet?
80        main()
```

1.4.3 Verhältnis zwischen Geschwindigkeit zum Zeitpunkt [i-1] und relativer Höhe aufstellen.

Eine weitere Kontrollstrategie läuft über die Definition der Kontrollvariable $u(t)$ als ein Verhältnis der Geschwindigkeit eines vorherigen Zeitpunkts zu dem der relativen Höhe. (Vergleiche Sourcecode Zeile 52)

```
52 u = 0.009*np.abs(vList[i-1])/np.abs(0.9*hList[i-1]/h0)
```

Sourcecode:

```
1  import numpy as np
       importieren
2  from matplotlib import pyplot as plt
3
4  m = 16400
5  g = 1.62
6
7  thrstFrc = 45000
8
9  t0 = 0
10 v0 = 0
11 h0 = 10000
12
13 dt = 1
14
15 i = 1
16 base = 0
17
18 numTimeSteps = 100
19
20 hList =[]
21 vList = []
22 tList = []
23
24 hList.append(h0)
25 vList.append(v0)
26 tList.append(t0)
27 uList =[]
28
29 u = 0
30 uList.append(u)
```

```
31 vsoll = -10
32
33 stop = False
34
35 while stop == False:
36
37     a = (1 / m) * (-m * g + u * thrstFrc)
38
39     tList.append(t0 + i * dt)
40     vList.append(vList[i - 1] + a * dt)
41     hList.append(hList[i - 1] + vList[i - 1] * dt)
42
43     i = i + 1
44     print(i)
45
46     u = 0.009*np.abs(vList[i-1])/np.abs(0.9*hList[i-1]/
       h0)
47
48     uList.append(u)
49
50
51     if(hList[i-1]<150) or i > 1000000:
52         g=0
53         u=0
54         a=0
55         vList[i-1]=0
56         stop = True
57
58 figure, axs = plt.subplots(4)
59 axs[0]. plot(tList, vList)
60 axs[0].set_title('Geschwindigkeit in Abhaengigkeit zur
    Zeit')
61 axs[0].set(xlabel='Zeit', ylabel= 'Geschwindigkeit')
62
63 axs[1]. plot(tList, hList, color= "brown", linewidth =
    3, linestyle = '--')
64 axs[1].set_title('Hoehe in Abhaengigkeit zur Zeit')
65 axs[1].set(xlabel= 'Zeit', ylabel = 'Hoehe')
66
67 axs[2].plot(hList, vList)
68 axs[2].set_title ('Geschwindigkeit in Abhaengigkeit zur
```

```
␣Hoehe')
69
axs[3].plot(tList, uList)
axs[3].set_title('U␣in␣Abhaengigkeit␣der␣Zeit')
plt.show()
```

Graphen und Ergebnisse:

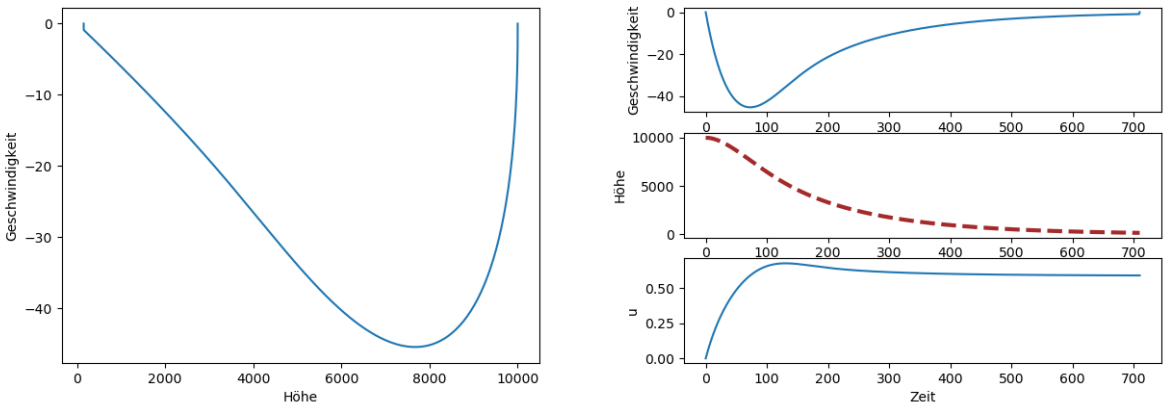


Abbildung 1.3: Kontrollgröße geregelt durch Verhältnisse

Mithilfe dieser Kontrollstrategie würde die Mondlandefähre mit einer Geschwindigkeit von 0.007 m/s in einer Höhe von 150 m landen.

1.4.4 u(t) als relative Änderung der Höhe

Wenn man u als die relative Änderung der Höhe definiert gleicht sich der Verlauf der Höhe, die die Mondlandefähre zu verschiedenen Zeitpunkten annimmt, einer Parabel an. Mit dieser Methode läuft der Landevorgang für die Insassen am angenehmsten ab, da es keine starken Beschleunigungen gibt und somit auch kein plötzliches Abfallen oder Ansteigen der Höhe.

Wesentliches Prinzip hinter dieser Methode:

Im Großen und Ganzen beschränkt sich die Kontrollstrategie dieses Modells auf zwei Zeilen im Code.

$$\Delta h = np.abs(hList[i-1] - h0)/h0 \tag{1.4}$$

Hier wird die relative Änderung der Höhe berechnet, wobei hList[i-1] der aktuellen Höhe und h0 der Ausgangshöhe entspricht. Am Anfang ist dieser

Wert 0, jedoch wird er immer größer bis er 1 erreicht. Dadurch wird die, nach unten wirkende, Geschwindigkeit immer mehr abgefedert, um eine möglichst sanfte Landung zu ermöglichen (Siehe Abb. 4). Je stärker sich die Höhe ändert bzw. je schneller man im Vergleich zur Ausgangshöhe fällt, desto mehr Schub wird entgegen diese Fallgeschwindigkeit gegeben

```
59 u = 0.10065 + \Delta h
```

Nachdem Δh als Wert zwischen 0 und 1 berechnet wurde, wird u mit dieser Variable belegt. Zusätzlich dazu addierten wir einen weiteren Betrag zu Δh, um die Schubkraft stärker zu machen. Dass u nicht über 1 steigt, wird weiter unten im Code implementiert:

Da nach der Visualisierung der bisherigen Befehle eine menschenunfreundliche Landehöhe bzw. Geschwindigkeit erzeugt wurde, musste man eine weitere Abfrage verfassen:

```
61  if hList[i-1] > h0 * 0.975:
62          u = 0.145
```

Sourcecode:

```
1 from typing import List, Any, Union
2
3 import numpy as np
4 from matplotlib import pyplot as plt
5
6 m = 16400                                          #Masse
      Apollo 11 Lunar Module
7 g = 1.62  # m/s^2         #Mondbeschleunigung
8
9
10
11 thrstFrc = 45000         #Schubkraft unteres
      Antriebsmodul
12
13 t0 = 0

14 v0 = 0
15 h0 = 10000
16
17 dt = 1             # point as comma / Schrittweite
18
19 i = 1
```

```
20 j = 0
21 numTimeSteps = 100
22
23 hList = []
24 vList = []
25 tList = []
26 uList = []
27 aList = []
28
29 hList.append(h0)
30 vList.append(v0)
31 tList.append(t0)
32
33 u = 0
34 uList.append(u)
35
36 while hList[i - 1] >= 0:
37
38     a = (1 / m) * (-m * g + u * thrstFrc)
39     aList.append(a)
40     #print(a)
41     tList.append(t0 + i * dt)
42     vList.append(vList[i - 1] + a * dt)
43     hList.append(hList[i - 1] + vList[i - 1] * dt)
44     i = i + 1
45
46     if hList[i - 1] < 1:          #Mondboden wird
       implementiert
47         g = 0
48         u = 0
49         vList[i - 1] = 0
50
51     dh = np.abs(hList[i - 1] - h0)/h0
52 u = 0.10065 + dh
53
54     if hList[i-1] > h0 * 0.975:
55         u = 0.145
56     if u > 1:
57         u = 1.0
58
59     uList.append(u)
```

```
60
61 figure , axs = plt.subplots(4)
62 axs[0].plot(tList , vList)
63 axs[0].set_title('Geschwindigkeit in Abhaengigkeit zur
       Zeit')
64 axs[0].set(xlabel='Zeit', ylabel='Geschwindigkeit')
65
66 axs[1].plot(tList , hList , color="brown", linewidth=3,
       linestyle='--')
67 axs[1].set_title('Hoehe in Abhaengigkeit zur Zeit')
68 axs[1].set(xlabel='Zeit', ylabel='Hoehe')
69
70 axs[2].plot(hList , vList)
71 axs[2].set_title('Geschwindigkeit in Abhaengigkeit zur
       Hoehe')
72 axs[2].set(xlabel='Hoehe', ylabel='Geschwindigkeit')
73
74 axs[3].plot(tList , uList)
75 axs[3].set_title('U in Abh. zur Zeit')
76 axs[3].set(xlabel='Zeit', ylabel='u')
77
78 plt.show()
```

Graphen und Ergebnisse:

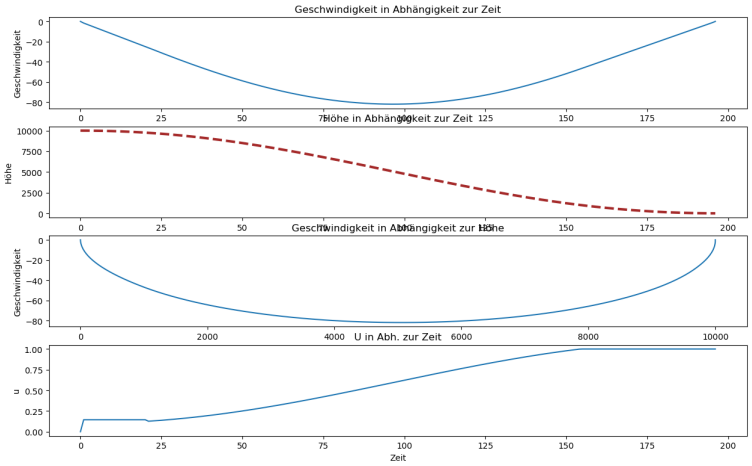


Abbildung 1.4: Kontrollgröße geregelt durch die relative Änderung der Höhe

Die Mondlandefähre würde in diesem Fall mit einer Geschwindigkeit von $0.02m/s$ in einer Höhe von $2m$ an der Mondoberfläche ankommen.

1.5 PID-Controller

Weitere Lösungsansätze basieren auf dem Prinzip von sogenannten „Proportional-Integral-Derivative controllern“, welche abgekürzt auch PID – Controller genannt werden. Darunter versteht man einen Regler, der die Kontrollvariable u(t) aus grundsätzlich 3 Teilen berechnet: Dem P-, dem I- und dem P – Teil. Weiters wird jeder dieser Teile mit einer Konstante berechnet, welche allerdings selbst eingestellt und angepasst werden muss. Im Folgenden bezeichnen wir diese Konstanten als Kp, Ki und Kd.

Das Ziel dieser Kontrollstrategie ist es, Abweichungen von einem gewünschten Wert durch Schwingungen auszugleichen. Der Graph eines typischen PID – Controllers verläuft folgendermaßen:

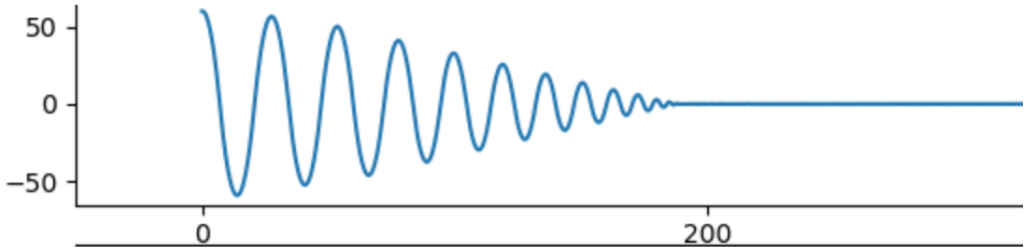


Abbildung 1.5: P-Regler

Nachdem der Graph den Wunschwert mehrmals übertritt, wird die Abweichung immer geringer, bis er den Idealwert genau beschreibt.

1.5.1 P-Teil

Prinzipiell lässt sich sagen, dass der P – Teil des Reglers die Werte, welche im aktuellen Zeitschritt passieren, beobachtet und mit dem Zielwert (In unserem Fall häufig Null, da gelandet wird) vergleicht. Für die Berechnung dieses Wertes bringt man den Unterschied zwischen diesen zwei Größen ständig in Erfahrung (Bezeichnung als Fehler für jede Zeiteinheit, also $e(t)$). Abschließend wird dieser Fehler mit der Konstante für den P – Teil multipliziert.

Hinsichtlich des Graphen steuerten wir mit dem P – Teil auf möglichst gleichbleibende Schwingungen hin.

$$p(t) = Kp * e(t) \tag{1.5}$$

1.5.2 I-Teil

Im Vergleich zum P-Wert kann man sich den I-Teil des Reglers so vorstellen, dass er die Summe vergangenen Fehlerwerte betrachtet und sicherstellt, dass diese möglichst gering sind. In Verbindung mit der Konstante Ki und dem Zeitschritt Δdt beschreibt jene Summe aller vergangenen und aktuellen Fehler den I Anteil des Controllers.

Grafisch betrachtet sorgt der I-Teil typischerweise für gedämpftere Wellen.

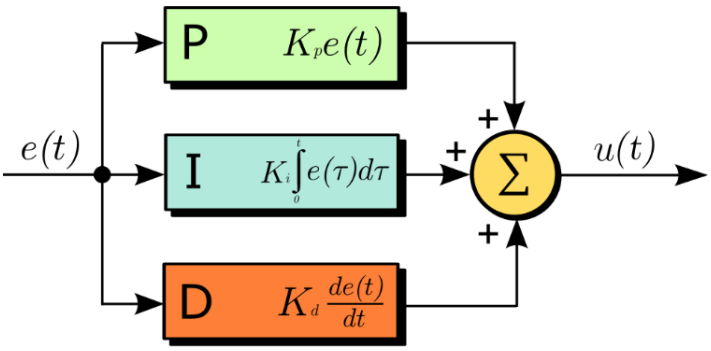
$$I(t) = Ki * (e(t0) + e(t1) + \ldots + e(t)) * \Delta t \tag{1.6}$$

1.5.3 D-Teil

Der Derivative-Teil des Controllers beeinflusst abschließend die Differenz zwischen dem vergangenen und aktuellen Fehler ($e(t-1)$–$e(t)$). Der resultierende Wert wird zusätzlich mit Kd (oft kleiner als Null) multipliziert.

Mit diesem Teil als Abschluss verlieren die vorhin gedämpften Wellen stark an Ausschlag und nähern sich wie in obiger Abbildung zu sehen dem jeweiligen Idealwert an.

$$d(t) = Kd(e(t-1) - e(t) \tag{1.7}$$



Kapitel 2

2D-Modell

2.1 Voraussetzungen

Das 2D-Modell der Mondlandefähre soll in y-/ und x-Richtung steuerbar sein. Es besitzt drei Triebwerke, wovon eines am Boden der Mondlandefähre befestigt ist und zwei an den Seiten derselben. Dieses Modell ist drehbar gelagert. Der Drehpunkt befindet sich im Schwerpunkt des Objektes. Die Masse des Objektes ist konstant, der Treibstoffverbrauch wird nicht berücksichtigt.

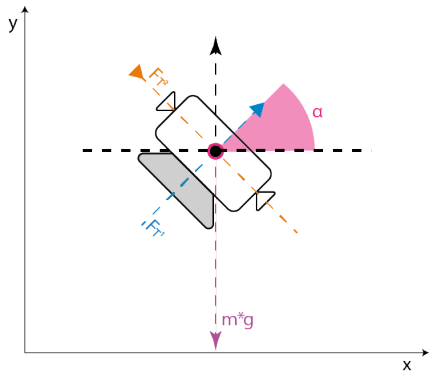


Abbildung 2.1: Gedrehte Mondlandefähre

Das drehbar gelagerte Objekt soll sich von einer geneigten Ausgangsposition so gerade ausrichten können, um bei der Landung stabil zu stehen. Das Objekt soll vertikal und horizontal steuerbar sein und an einem fixen Punkt in x-Richtung ankommen. Dafür benötigen wir zwei Kontrollvariablen „u1“ und „u2“. „u1“ steuert die vertikale Schubkraft, „u2“ steuert die horizontale. Während der Wert von u1(t) sich zwischen dem Intervall von $[0, 1]$ bewegt kann $u2(t)$ Werte im Intervall $[-1, 1]$ annehmen. Wie beim 1D-Modell ist das

Hauptziel, die Mondlandefähre mit möglichst kleiner Höhe und Geschwindigkeit zu landen.

2.2 Wirkende Kräfte

Auf das 2D-Modell wirkt die Mondbeschleunigung, die das Objekt nach unten zieht. Zusätzlich wirkt auch noch die vertikale Schubkraft (FVer) und die horizontale Schubkraft (FHor), die sich aus den vertikalen und horizontalen Kraftanteilen der Antriebsmodule zusammensetzen. Das Triebwerk an der Unterseite hat eine maximale Schubkraft von $45000N$, die seitlichen Triebwerke eine Kraft von $500N$.

$$Ft1 = 45000 * u1 \tag{2.1}$$
$$Ft2 = 500 * u2 \tag{2.2}$$
$$FHor = Ft2 * sin(\alpha) + Ft1 * cos(\alpha) \tag{2.3}$$
$$FVer = Ft2 * cos(\alpha) – Ft1 * sin(\alpha) \tag{2.4}$$

In weiterer Folge erhält man jeweils eine Beschleunigung und Geschwindigkeit in die horizontale und vertikale Richtung sowie die Winkelbeschleunigung.

$$a1(t) = 1/m * (Ft2 * sin(\alpha) + Ft1 * cos(\alpha)) \tag{2.5}$$
$$a2(t) = 1/m * ((Ft2 * cos(\alpha) – Ft1 * sin(\alpha)) \tag{2.6}$$
$$aw = (1/J) * (l * Ft2) \tag{2.7}$$

Die horizontale und vertikale Kraft ist abhängig vom Winkel α.

Im Sourcecode unten haben wir die Winkelbeschleunigung entfernt bzw. auskommentiert, da es für uns nicht umsetzbar war, die vertikale und horizontale PID-Regelung mit der Regelung des Winkels zu verbinden. Das rührt daher, dass sich beide PID-Regelungen überschneiden, wenn horizontale Position und Winkelregelung aufeinandertreffen. Die Winkeländerung und die Horizontalkraft beeinflussen sich nämlich gegenseitig, was zu einer erheblichen Verkomplizierung führt.

Um die Kontrolle des Winkels trotzdem zu verwenden, schrieben wir ein Programm, welches nur die Winkelkontrolle beinhaltete, wobei in horizontale Richtung nichts geregelt wurde.

2.3 Steuerung mit PID-Regler

Als nächsten Schritt gilt es, die Landefähre auf eine kleinere Höhe zu bringen. Dazu wird die Kontrollgröße u1 definiert. Dazu wurden verschiedene Zielwerte erstellt, die zu unterschiedlichen Höhenstufen zum Einsatz kommen.

2.3.1 Kontrolle der vertikalen Position

```
80 y_target1 = 70
81 y_target2 = 50
82 y_target3 = 20
83 y_target4 = 0
84 evy = y_target - yList[i]
85 evySum += evy * dt
86 u1 = VP * evy + VI * evySum + (evy - evyAlt) * VD
```

2.3.2 Kontrolle der horizontalen Position:

Um unseren Zielwert in horizontaler Richtung zu erhalten mussten wir unsere Kontrollvariable u2 mithilfe eines weiteren PID-Reglers definieren.

$$x_target = 40 evx = x_target - xList[i-1] evxSum+ = evx * dtu2 = VxP * evx + VxI * evxSum$$

VxP, VxI und VxD nehmen, wie im Kapitel der PID-Regelung beschreiben, konstante Werte an, welche man manuell und oftmals durch Probieren finden muss.

2.3.3 Winkelkontrolle

Um unseren Winkel auszutarieren setzten wir unseren Zielwert als $\pi/2$ fest, also $90 Grad$ Der Fehler wird als Differenz zwischen Soll- und Istwert deklariert.

```
1 e = (np.pi / 2) - wList[i-1]
2 eSum += e * dt
3 u2 = KP * e + KI * eSum + (e - eAlt) * KD
4 eAlt = e
```

Sourcecode:

```
1 import numpy as np
2 import array as arr
3 from matplotlib import pyplot as plt
```

```
4
5 m = 15000
6 g = 1.62
7
8 l = 5
9
10 J = 1000          #Winkelgeschwindigkeit
11
12 u1 = 1                    # Kontrollschub vertikal
13 u2 = 0            # Kontrollschub horizontal
14
15 Ft1 = u1 * 45000

    # Schubkraft vertikal
16 Ft2 = u2 * 500

    # Schubkraft horizontal
17 FG = m * g

    # Anziehungskraft
18
19 t0 = 0
20
21 vx0 = 0.1
22 vy0 = 0
23 vw0 = 0
24
25 KP = 150
26 KI = 1
27 KD = 600
28
29 #VP = 1
30 #VI = 1
31 #VD = 500
32
33 eSum = 0
34 eAlt = 0
35
36 evySum = 0
37 evyAlt = 0
38
```

```
39 evxSum = 0
40 evxAlt = 0
41
42 x0 = 0
43 y0 = 100
44 w0 = np.pi / 2
                #Ausgangswinkel
45
46 numTimeSteps = 100000                                  #Zeit
47 dt = 0.1
                        #Zeitschritte
48
49 tList = []
50
51 vxList = []
52 vyList = []
53 vwList = []
54
55 axList = []
56 ayList = []
57 awList = []
58
59 xList = []
60 yList = []
61 wList = []
62
63 u1List = []
64 u2List = []
65
66 u1List.append(u1)
67 u2List.append(u2)
68
69 wList.append(w0)
70 yList.append(y0)
71 xList.append(x0)
72 alphaList = []
73
74 tList.append(t0)
75
76 vxList.append(vx0)
77 vyList.append(vy0)
```

```
78 vwList.append(vw0)
79
80 y_target1 = 70
81 y_target2 = 50
82 y_target3 = 20
83 y_target4 = 0
84
85 y_target = y_target1
86
87 x_target = 40
88 numTimeSteps = 26700
89
90 #cntr1 = 0
91
92 for i in range(1, numTimeSteps):
93
94     fHor = Ft2 * np.sin(wList[i - 1]) + Ft1 * np.cos(
       wList[i - 1])
95     fVer = Ft1 * np.sin(wList[i - 1]) - Ft2 * np.cos(
       wList[i - 1])
96
97
98     ax = (1/m) * fHor
99     ay = (1/m) * (-FG + fVer)
100    #aw = (1/J) * (l*Ft2)
101    aw = 0
102    #ay = 0
103    axList.append(ax)
104    ayList.append(ay)
105    awList.append(aw)
106
107    tList.append(t0 + i * dt)
108
109    vxList.append(vxList[i-1] + ax * dt)
110    vyList.append(vyList[i-1] + ay * dt)
111    vwList.append(vwList[i-1] + aw * dt)
112
113    xList.append(xList[i-1] + vxList[i-1] * dt)
114    yList.append(yList[i-1] + vyList[i-1] * dt)
115    wList.append(wList[i-1] + vwList[i-1] * dt)
116
```

```
117
118     #
            ##################################################################
119     # Versuche Raumschiff gerade zu stellen mit
            Winkelkontrolle
120     #
            ##################################################################
121
122
123     """
124     #Winkelkontrolle
125     e = (np.pi / 2) - wList[i-1]
126
127     eSum += e * dt
128     u2 = KP * e + KI * eSum + (e - eAlt) * KD
129     eAlt = e
130
131
132     if u2 > 1:
133         u2 = 1
134
135     if u2 < -1:
136         u2 = -1
137
138     Ft2 = u2 * 500
139     #print(Ft2)
140     """
141
142     #if yList[i] <= 0:
143     if 1 == 2:
144         g = 0
145         yList[i] = 0
146         vyList[i] = 0
147         vxList[i] = 0  #Annahme ohne am Boden
                weiterrutschen
148
149
150     ### Update Kontrolle vertikal
151     # Stufenweise Kontrolle der y Position
```

```
152
153     if i > 150:          # Hier wird nach einer gewissen
            Anzahl von Zeitschritten (i) ein Wert N
            festgelegt, um den man in die Vergangenheit
            schaut
154         N = 140
155
156
157
158         if np.all(np.abs(y_target1 - np.asarray(yList[i
                - 1 - N : i])) < 2.5) and np.all(np.abs(
                x_target - np.asarray(xList[i - 1 - N : i]))
                < 0.03):
159             y_target = y_target2
160             #cntr1 += 1
161
162         if np.all(np.abs(y_target2 - np.asarray(yList[i
                - 1 - N : i])) < 0.5) and np.all(np.abs(
                x_target - np.asarray(xList[i - 1 - N : i]))
                < 0.03):  #0.05
163             y_target = y_target3
164
165         if np.all(np.abs(y_target3 - np.asarray(yList[i
                - 1 - N : i])) < 0.6) and np.all(np.abs(
                x_target - np.asarray(xList[i - 1 - N : i]))
                < 0.03):  #0.05
166             y_target = y_target4
167
168     #if cntr1 > 1000:
169     #    y_target = 20
170
171     """
172     #PID Vertikal
173     """
174     evy = y_target - yList[i]
175
176     evySum += evy * dt
177
178     # Einstellung der Werte, herumprobieren
179     VP = 0.073
                #0.043
```

```
180     VI = 0.000519
                #0.0056
181     VD = 1.9    #1.9
182     u1 = VP * evy + VI * evySum + (evy - evyAlt) * VD
183
184     evyAlt = evy
185     #print(yList[i-1])
186     # PID Regelung Horizontalposition x, analog zu
            vorher
187
188     evx = x_target - xList[i - 1]
189
190     evxSum += evx * dt
191
192     VxP = 0.00004
193     VxI = 0.000000001
194     VxD = 0.056
195
196     u2 = VxP * evx + VxI * evxSum + (evx - evxAlt) *
            VxD
197
198     evxAlt = evx
199
200
201     # ### Update Kontrolle horizontal
202
203
204     if u1 > 1:
205         u1 = 1
206     if u1 < 0:
207         u1 = 0
208
209     if u2 > 1:
210         u2 = 1
211
212     if u2 < - 1:
213         u2 = -1
214
215     u1List.append(u1)
216     u2List.append(u2)
217
```

```
218         Ft1 = u1 * 45000
219         Ft2 = u2 * 25000
220 
221 
222 figure, axs = plt.subplots(4)
223 axs[0].plot(np.arange(0, numTimeSteps) * dt, wList)
224 axs[0].set_title('Winkel in Abh. zur Zeit')
225 axs[0].set(xlabel='Zeit', ylabel='Winkel')
226 
227 axs[1].plot(xList, yList)
228 axs[1].set_title('Position im Raum')
229 axs[1].set(xlabel='posx', ylabel='posy')
230 
231 axs[2].plot(np.arange(0, numTimeSteps), vyList)
232 axs[2].set_title('Vertikale Geschw.')
233 axs[2].set(xlabel='Time', ylabel='Vy')
234 
235 axs[3].plot(np.arange(0, numTimeSteps), vxList)
236 axs[3].set_title('Horizontale Geschw.')
237 axs[3].set(xlabel = 'time', ylabel = 'Vx')
238 
239 figure, axs = plt.subplots(2)
240 axs[0].plot(np.arange(0, numTimeSteps) * dt, u1List)
241 axs[0].set_title('u1')
242 axs[0].set(xlabel='Zeit', ylabel='posy')
243 
244 axs[1].plot(np.arange(0, numTimeSteps) * dt, u2List)
245 axs[1].set_title('u2')
246 axs[1].set(xlabel='Zeit', ylabel='posx')
247 
248 figure, axs = plt.subplots(2)
249 axs[0].plot(np.arange(0, numTimeSteps) * dt, yList)
250 axs[0].set_title('Vertikale Position in Abh. zur Zeit')
251 axs[0].set(xlabel='Zeit', ylabel='posy')
252 
253 axs[1].plot(np.arange(0, numTimeSteps) * dt, xList)
254 axs[1].set_title('Horizontale Position in Abh. zur Zeit
    ')
255 axs[1].set(xlabel='Zeit', ylabel='posx')
256 
257 plt.show()
```

Kapitel 3

Optimalsteuerung

Die Idee, die hinter dieser Kontrolltheorie steckt, ist es, verschiedene Kosten für bestimmte Größen, die kontrolliert werden sollen, einzuführen. Wir nehmen zum Beispiel an, eine hohe Geschwindigkeit sorge für einen hohen Treibstoffverbrauch, ist also sehr teuer. Eine große Höhe kostet uns ebenfalls viel. Aktionen, wie das Steuern des Raumfahrzeugs sollten in diesem Fall jedoch nicht zu teuer sein, da die ideale Lösung für unser Problem nur durch ein regelmäßiges Ändern unserer Kontrollgröße möglich ist.

Die Kosten werden bei der Bewertung der Zustände in laufende Kosten und Endkosten unterteilt, die variabel verstellbar sind und in einem bestimmten Verhältnis stehen müssen um das gewünschte Ergebnis zu bekommen.

Die Aufgabe der Optimalsteuerung besteht in weiterer Folge darin, die Kosten für den Landevorgang zu minimieren, also den für uns günstigsten Weg zu finden. Die beste Steuerung über die ganze Zeitspanne erhält man nur, wenn man zu jedem Zeitpunkt optimal steuert um Kostenanstiege zu umgehen.

Wir beschränken uns der Allgemeinheit halber auf sogenannte linear quadratische Optimalsteuerungsprobleme. Diese sind für reellwertige Kontrollen wie folgt charakterisiert:

Dynamik des Systems:

$$s_{i+1} = A * s_i + B * u_i, \tag{3.1}$$

wobei s_i den Zustand des Systems (Geschwindigkeit und Höhe) zum Zeitpunkt i, u_i die Kontrolle zum Zeitpunkt i bezeichnet. A bezeichnet eine Matrix und B einen Vektor.

Kosten:

$$LaufendeKosten : (1/2) * s_i^T * Q * s_i + (1/2) * u_i^2 * R \tag{3.2}$$

$$Endkosten : (1/2) * s_N^T * S * s_N \tag{3.3}$$

Dabei bezeichnen Q, S Matrizen und R eine Zahl. Die Matrizen Q, S bewerten den aktuellen Zustand und R beschreibt die Kosten eine bestimmte Kontrolle auszuführen.

Die Kosten werden durch die sogenannte Wertefunktion evaluiert, die jeden Zustand bewertet. Sie berechnet die Kosten, die sich an jedem Startzeitpunkt und jedem Anfangswert ergeben wenn bis dato bereits optimal gesteuert wurde. Anhand der Wertefunktion wird die Kontrolle gewählt die in weiterer Folge am wenigsten Kosten verursacht.

Die Optimalsteuerung wurde, nach seinem Entdecker, Richard Bellman als „Bellman-Prinzip“ benannt. Diese Steuerung formuliert bei einer bekannten Wertefunktion eine, an jedem Moment optimale Kontrolle. Für ein linear quadratisches Optimalsteuerungsproblem lässt sich eine geschlossene Formel für diese Wertefunktion angeben – sie genügt der sog. Riccati- Gleichung.

Wir erlauben uns an dieser Stelle nicht weiter auf die Darstellung der Wertefunktion einzugehen. In unseren Simulationen haben wir einen Code-Abschnitts unseres Betreuers übernommen welcher die Berechnung dieser gewährleistet.

Ergebnisse:

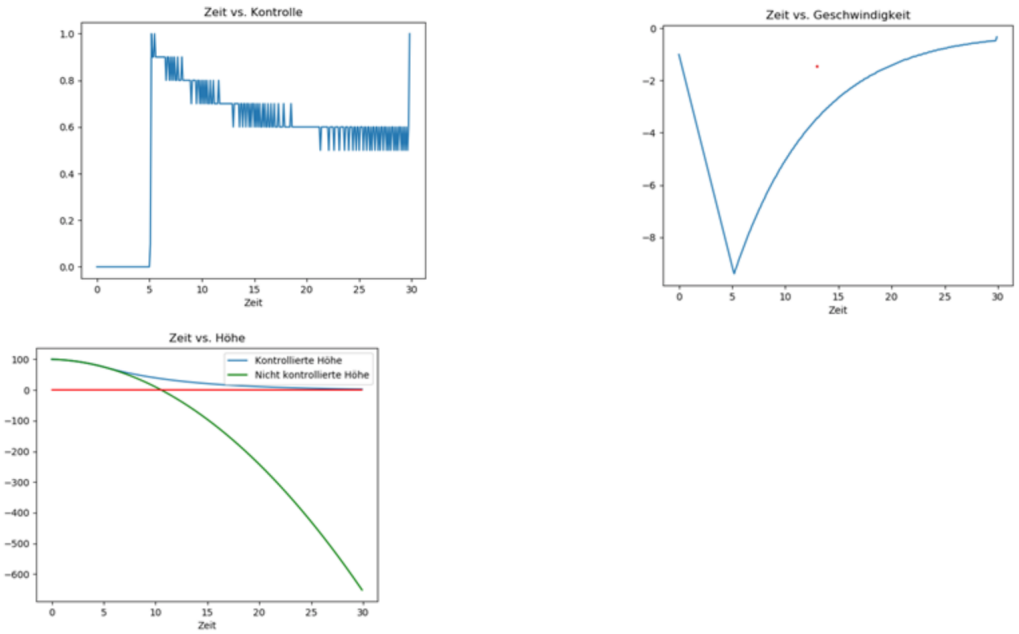


Abbildung 3.1: Änderung der Kontrollgröße mittels Optimalsteuerung

Sourcecode:

```
1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 frc = 45000
5 m = 15000
6 #
    ##################################################################

7
8 dt = 0.1
9
10 #
    ##################################################################

11
12 A = np.zeros((3, 3))
13 A[0, 0] = 1
14 A[1, 1] = 1
15 A[2, 2] = 1
16
17
18 A[0, 1] = dt
19 A[1, 2] = -dt
20
21 B = np.zeros(3)
22 B[1] = dt * frc / m
23
24
25
26 #
    ##################################################################

27
28 h0 = 100
29 v0 = -1
30 g0 = 1.62
31
32 #
    ##################################################################
```

```
33
34 # terminal cost
35 W = np.zeros((3, 3))
36 W[0, 0] = 2
37 W[1, 1] = 1                                    # beeinflusst
       Geschwindigkeit
38
39 #
       ##################################################################

40
41 # state cost
42 Q = np.eye(3)
43 Q[0, 0] = 0.02
44 Q[1, 1] = 1                                    # beeinflusst
       Geschwindigkeit
45
46 # control cost
47 R = 0.001
48
49 #
       ##################################################################

50
51 # list of matrices - upside down in time!
52
53 numTimeSteps = 300
54
55 valFuncMtrcs = np.zeros((numTimeSteps, 3, 3))
56 valFuncMtrcs[-1, :, :] = W
57
58 # Berechnung der Wertefunktion
59
60 for i in range(numTimeSteps - 2, -1, -1):
61
62 W = valFuncMtrcs[i + 1, :, :]
63
64 M = R + np.dot(B.transpose(), np.dot(W, B))
65
66 invM = 1.0 / M
```

```
67
68 tmp = np.dot(B.transpose(), np.dot(W, A)).reshape(1,
       -1)
69 valFuncMtrcs[i, :, :] = Q + np.dot(A.transpose(), np.
       dot(W, A)) - invM * np.dot(tmp.transpose(), tmp)
70
71 controlList = []
72
73
74 #
       ##################################################################
75
76 # now iterate
77
78 # Erlaube nur 11 Stufen der Kontrollen: 0.0, 0.1, 0.2,
       ..., 1.0
79 N = 11
80 uList = np.arange(0, N) * 0.1
81
82 stateList = np.zeros((3, numTimeSteps))
83 stateList[:, 0] = np.asarray([h0, v0, g0])
84
85 # states with zero control - 'free fall'
86 stateList0 = np.zeros((3, numTimeSteps))
87 stateList0[:, 0] = np.asarray([h0, v0, g0])
88
89 for i in range(0, numTimeSteps - 1):
90
91 tmpVec = np.zeros(N)
92 xx = np.zeros(3)
93
94
95 # Anwendung des Bellman- Prinzips:
96 for j in range(0, N):
97
98
99 xx = np.dot(A, stateList[:, i]) + uList[j] * B
100
101 stgCost = 0.5 * np.dot(xx.transpose(), np.dot(Q, xx)) +
        0.5 * uList[j] * (R ** 2)
```

```
102
103 tmpVec[j] = stgCost * dt + 0.5 * np.dot(xx.transpose(),
        np.dot(valFuncMtrcs[i + 1], xx))
104
105
106 idx = tmpVec.argmin(axis = 0)
107
108 u = uList[idx]
109
110 # W = valFuncMtrcs[i + 1, :, :]
111 # M = R + np.dot(B.transpose(), np.dot(W, B))
112
113 # u = -(1 / M) * np.dot(B.transpose(), np.dot(
        valFuncMtrcs[i + 1], np.dot(A, stateList[:, i])))
114
115
116 # now determine the 'best' state
117 controlList.append(u)
118 stateList[:, i + 1] = np.dot(A, stateList[:, i]) + u *
       B
119
120 # Vergleich: freier Fall
121 stateList0[:, i + 1] = np.dot(A, stateList0[:, i])
122
123 fig = plt.figure(1)
124 plt.plot(np.arange(0, numTimeSteps) * dt, stateList[0,
       :], label = 'Kontrollierte Hoehe')
125 plt.plot(np.arange(0, numTimeSteps) * dt, stateList0[0,
        :], 'g', label = 'Nicht kontrollierte Hoehe')
126 plt.plot(np.arange(0, numTimeSteps) * dt, np.zeros(
       numTimeSteps), 'r')
127 plt.xlabel('Zeit')
128 plt.title('Zeit vs. Hoehe')
129 plt.legend()
130
131
132 fig = plt.figure(2)
133 plt.plot(np.arange(0, numTimeSteps) * dt, stateList[1,
       :])
134 plt.xlabel('Zeit')
135 plt.title('Zeit vs. Geschwindigkeit')
```

```
136
137
138 fig = plt.figure(3)
139 plt.plot(np.arange(0, numTimeSteps - 1) * dt,
        controlList)
140 plt.xlabel('Zeit')
141 plt.title('Zeit vs. Kontrolle')
142
143 plt.show()
```

Kapitel 4

Fazit

Lässt man die vergangene Woche Revue passieren blicken wir zurück auf viele Stunden, in denen wir selbstständig an verschiedensten Kontrollmechanismen und Theorien gearbeitet haben. Durch die viele Praxis, die wir im Programmieren im Laufe der Zeit bekommen haben, erhielten wir einen Einblick in die Programmiersprache "Python" und verschiedene bestehende Kontrolltheorien, die auch in der Realität verwendet werden, wie den PID-Regler als einen der klassischen Reglerarten und wir bekamen eine Einführung in die Theorie der Optimalsteuerung. Dadurch, dass uns beim Ausarbeiten der Kontrolltheorien meistens viel Freiheit gelassen wurde, kam es hin und wieder zu Problemen und Fehlern, die wir mit der Hilfe unseres Betreuers jedoch in den meisten Fällen schnell lösen konnten. Auch konnten wir manche Steuerelemente nicht miteinander kombinieren, jedoch haben wir im Endeffekt für alle Problemstellungen eine passende Lösung gefunden.

Mechanik und Statik

Brücken und Türme
Kräfteverteilung in Fachwerken

Emil Čavkić[1], Clemens Frizberg[1], Anna Günther[2], Johanna Kubassa[2], Simone Lemesch[3], Ida Stettner[4], MingHan Zhang[4]

[1]Bundesrealgymnasium/Bundesgymnasium Leibnitz
[2]Graz International Bilingual School
[3]Bundes- und Bundesrealgymnasium Weiz
[4]Akademisches Gymnasium Graz

Dr. **Robert Beinert[5]** MSc BSc

[5]Institut für Mathematik und Wissenschaftliches Rechnen
Karl-Franzens-Universität Graz

Inhaltsverzeichnis

1 Einleitung

Am Tag der feierlichen Eröffnung, dem 31. März 1989, bestieg Gustave Eiffel mit deiner Delegation die Spitze des gerade fertiggestellten Turmes und hisst die französische Trikolore. Selten waren die Proteste gegen ein Bauwerk größer, selten die Begeisterung und der Stolz. Als monumentales Eingangsportal und Aussichtsturm der zehnten Weltausstellung in Paris ist das damals mit Abstand höchste Gebäude der Welt nicht mehr aus dem Stadtbild und dem französischen Nationalbewusstsein wegzudenken. Mit seinen 18.038 Einzelteilen, zusammengehalten von über 2,5 Millionen Nieten trotz der riesige Eisenfachwerkturm, Wahrzeichen der Industrialisierung und Ingenieurskunst, Wind und Wetter.

Wie können filigrane Türme, Brücken, Fassaden, Kuppeln, Fahrzeugrahmen, bestehend aus einem mehr oder weniger komplizierten Geflecht aus Stahlstreben, den enormen physischen Kräften, denen sie täglich ausgesetzt sind, widerstehen? Um Fragen wie diese zu beantworten, werde wir diese Strukturen als sogenannte Fachwerke modellieren. Bei einer Belastung durch äußere Kräfte beginnen sich Stahlkonstruktionen zu verformen. Sind die Kräfte nicht zu groß, kehrt das Stabwerk bei Wegnahme der Belastung wieder in den Ursprungszustand zurück. Das Material verhält sich also elastisch. Obwohl bei Bauwerken die Verformung meistens nicht mit dem bloßen Auge wahrgenommen werden kann, ist diese für die Berechnung der Kräfteverteilung unerlässlich. Werden die Kräfte in einem einzelnen Stab zu groß, verliert die Stahlkonstruktion ihre Stabilität und bricht auseinander.

Im Laufe der Woche waren es unsere Ziele zuerst die Kräfte in einem einzelnen Stab zu untersuchen, dann ein lineares Modell zu entwickeln und dieses auf Fachwerkkonstrukte anzuwenden. Weiters wollten wir das ganze noch numerisch und auch kreativ umsetzen, folglich haben wir mit Python ein Programm zur graphischen Ausgabe unserer Ergebnisse erstellt.

2 Voraussetzungen

Für unser Modell gehen wir davon aus, dass die Stäbe bzw. das Fachwerkkonstrukt folgende Eigenschaften besitzt:

- Stäbe sind elastisch
- Stäbe sind sehr dünn
- Drehmomente in den Verbindungen können vernachlässigt werden
- Bei Belastungen verformt sich die Struktur
- Bei Wegnahme der Kraft kehren die Stäbe in ihre Ursprungspostion zurück

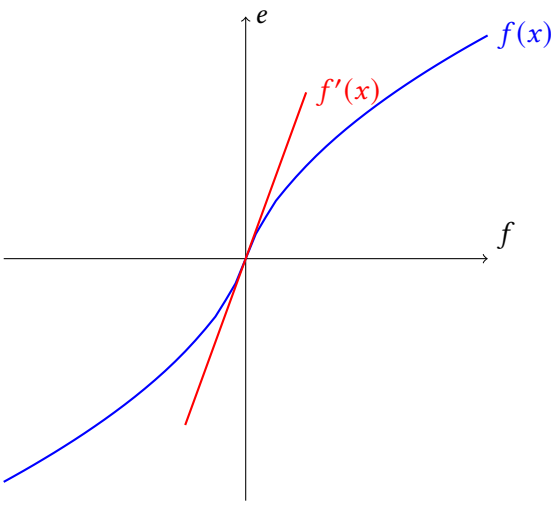


Abbildung 1: Angenommenes Verhältnis zwischen Längenänderung und Krafteinwirkung in der Wirklichkeit

3 Eigenschaften eines Stabes

Um ein größeres Bauwerk modellieren und simulieren zu können, benötigen wir zuerst ein Modell für einen einzelnen Stab. Ein Stab wird hier immer über seine Endpunkte definiert.

3.1 Kräfte und Längenänderungen Wenn eine Kraft f von außen auf einen Stab wirkt, wird er gestreckt oder gestaucht. Umgekehrt hat ein Stab mit veränderter Länge eine innere Kraft g, eine Spannung – er will in seine Ursprungslage zurückkehren. Diese innere und äußere Kraft haben denselben Betrag, wirken allerdings in gegensätzliche Richtungen. Das lässt sich mit Newtons dritten Axiom erklären:

$$Actio = Reactio$$

$$g = -f$$

In der Wirklichkeit wird die benötigte Kraft, um den Stab zu strecken oder zu stauchen größer, je mehr der Stab schon verformt ist. Dieses Verhältnis könnte ungefähr wie in Abbildung 1 aussehen.

Da uns nur minimale Streckungen und Stauchungen interessieren, können wir dieses Modell linearisieren. Sehen wir uns die nähere Umgebung der Stelle 0 an, erkennen wir, dass die benötigte Kraft f dort annähernd direkt proportional zu der relativen oder absoluten Längenänderung e ist. Daher setzten wir

$$f = -k \cdot e \quad \text{und} \quad g = k \cdot e,$$

wobei wir k als Federkonstante bezeichnen.

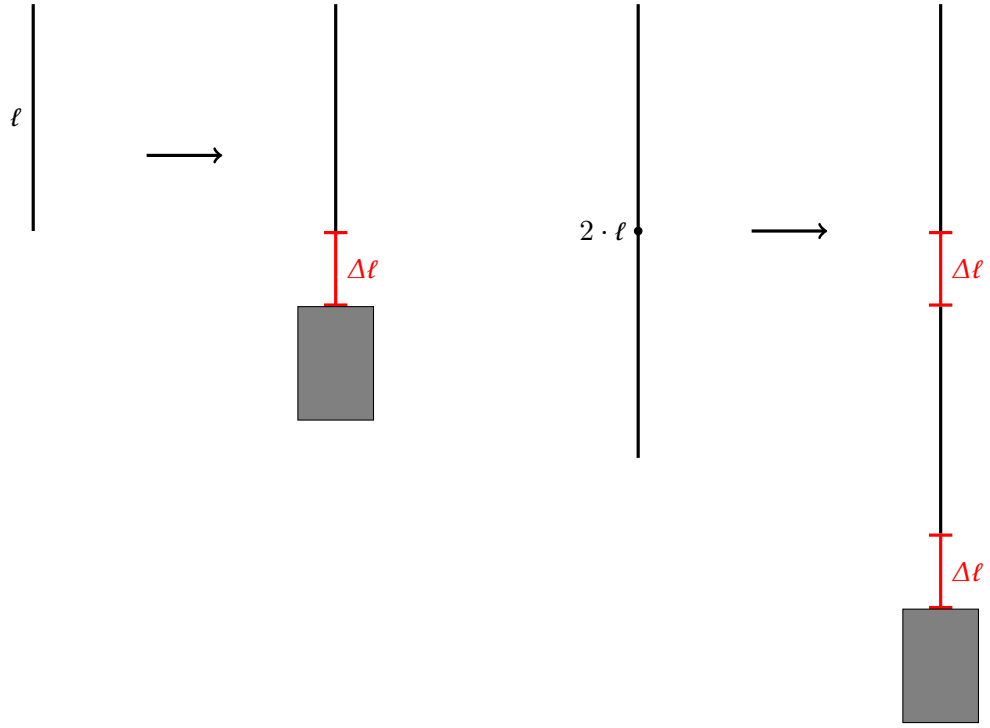


Abbildung 2: Wenn ein Gummiband um $\Delta\ell$ gestreckt wird, werden zwei um $2\Delta\ell$ gestreckt

3.1.1 Relative oder absolute Längenänderung? Es stellt sich noch die Frage, ob die Längenänderung e relativ zur gesamten Stablänge oder absolut ist. Dazu folgendes Gedankenexperiment:

Wenn an ein Gummiband mit der Länge ℓ ein Gewicht gehängt wird, so dehnt sich das Gummiband um eine Länge $\Delta\ell$ aus. Bei zwei solchen miteinander verbundenen Gummibändern, die ebenfalls jeweils die Länge ℓ haben, wirkt die Gewichtskraft auf beide. Ihre Längen verändern sich jeweils um $\Delta\ell$. Insgesamt ändert sich die Länge also um $2\Delta\ell$. Bei einem Gummiband mit der Länge 2ℓ ändert sich die Länge bei der gleichen Gewichtskraft also um $2\Delta\ell$. (Siehe Abbildung 2) Daraus folgern wir, dass die Längenänderung bei einer wirkenden Kraft relativ zu der Länge des Gummibandes (oder Stabes) ist.

3.2 Relative Längenänderung aus Punkten und Änderungsvektoren

3.2.1 Länge eines Stabes Um die relative Längenänderung zu bekommen, müssen wir zuerst die Länge eines Stabes anhand seiner Punkte $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ bestimmen. Mithilfe des Satzes von Pythagoras kann man die Länge eines Vektors bestimmen wie der Abbildung 3 zu entnehmen.

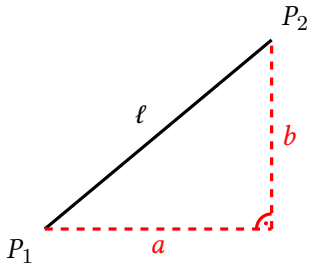


Abbildung 3: Anwendung des Satzes von Pythagoras

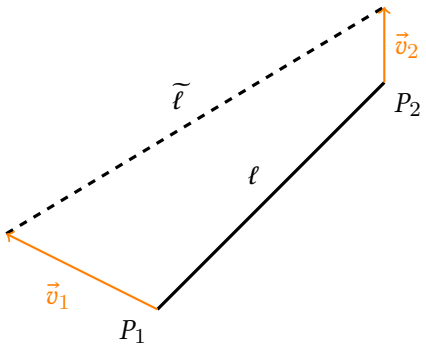


Abbildung 4: Die Verschiebung der Punkte

Wenn wir nun den Vektor $\overrightarrow{\boldsymbol{p}_1\boldsymbol{p}_2}$ bestimmen, können wir daraus die Länge des Stabes berechnen:

$$\ell_{1,2} = |\overrightarrow{\boldsymbol{p}_1\boldsymbol{p}_2}| = |\boldsymbol{p}_2 - \boldsymbol{p}_1| = \left|\begin{pmatrix} \boldsymbol{p}_2^x - \boldsymbol{p}_1^x \\ \boldsymbol{p}_2^y - \boldsymbol{p}_1^y \end{pmatrix}\right| = \sqrt{(\boldsymbol{p}_2^x - \boldsymbol{p}_1^x)^2 + (\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)^2}$$

3.2.2 LÄNGE DES GESTRECKTEN/GESTAUCHTEN STABES Wir versuchen, die Länge eines verschobenen Stabes anhand der ursprünglichen Punkte $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ und der Verschiebungen $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ zu definieren. Die neuen, verschobenen Punkte sind

$$\boldsymbol{p}_1 + \boldsymbol{v}_1 = \begin{pmatrix} \boldsymbol{p}_1^x + \boldsymbol{v}_1^x \\ \boldsymbol{p}_1^y + \boldsymbol{v}_1^y \end{pmatrix}$$

$$\boldsymbol{p}^2 + \boldsymbol{v}^2 = \begin{pmatrix} \boldsymbol{p}_2^x + \boldsymbol{v}_2^x \\ \boldsymbol{p}_2^y + \boldsymbol{v}_2^y \end{pmatrix}$$

Aus diesen Punkten ergibt sich die Länge des gestreckten bzw. gestauchten Stabes:

$$\widetilde{\ell}^{1,2} = \sqrt{((\boldsymbol{p}_2^x + \boldsymbol{v}_2^x) - (\boldsymbol{p}_1^x + \boldsymbol{v}_1^x))^2 + ((\boldsymbol{p}_2^y + \boldsymbol{v}_2^y) - (\boldsymbol{p}_1^y + \boldsymbol{v}_1^y))^2}$$

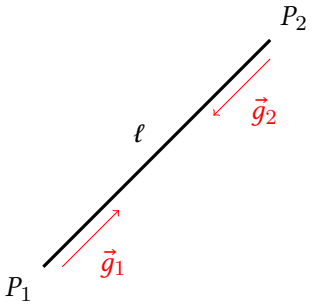


Abbildung 5: Die Kräfte, die innerhalb des Stabes wirken

3.2.3 Relative Längenänderung Die relative Längenänderung lässt sich aus der absoluten Längenänderung geteilt durch die ursprüngliche Länge berechnen. Die absolute Längenänderung ist die Differenz zwischen neuer Länge $\widetilde{\ell}_{1,2}$ und alter Länge $\ell_{1,2}$.

$$\begin{aligned}\frac{\widetilde{\ell}_{1,2} - \ell_{1,2}}{\ell_{1,2}} &= \frac{\widetilde{\ell}_{1,2}}{\ell_{1,2}} - \frac{\ell_{1,2}}{\ell_{1,2}} = \frac{\widetilde{\ell}_{1,2}}{\ell_{1,2}} - 1 \\ &= \frac{\sqrt{((\boldsymbol{p}_2^x + \boldsymbol{v}_2^x) - (\boldsymbol{p}_1^x + \boldsymbol{v}_1^x))^2 + ((\boldsymbol{p}_2^y + \boldsymbol{v}_2^y) - (\boldsymbol{p}_1^y + \boldsymbol{v}_1^y))^2}}{\sqrt{(\boldsymbol{p}_2^x + \boldsymbol{p}_1^x)^2 + (\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)^2}} - 1\end{aligned}$$

3.3 Die innere Kraft als Vektor Als nächstes stellen wir die Spannung g innerhalb eines einzelnen Stabes als zweidimensionalen Vektor dar, um ihn später in Relation zu den äußeren Kräften zu setzten. Die Richtung der Spannung entspricht hierbei der Richtung des Stabes, also der Richtung des Vektors wie in Abbildung 5 zu sehen.
$\overrightarrow{\boldsymbol{p}_1\boldsymbol{p}_2} = (\boldsymbol{p}_2 - \boldsymbol{p}_1)$.

Für die Darstellung der inneren Kraft müssen wir diesen Vektor nur mit der richtigen Länge und Ausrichtung (Vorzeichen) versehen. Hierfür müssen wir den Vektor also entsprechen skalieren.

Sei $\boldsymbol{w}$ ein beliebiger zweidimensionaler Vektor. Die neue Länge nach dem Skalieren ist

$$|t \cdot \boldsymbol{w}| = |t| \cdot |\boldsymbol{w}|,$$

wobei t eine reelle Zahl ist.

Die x- und y-Koordinaten des Vektor $\boldsymbol{w}$ bezeichnen wir mit $\boldsymbol{w}^x$ und $\boldsymbol{w}^y$. Die zu zeigende Aussage folgt aus der Umformung der Berechnung der Länge, also

$$\begin{aligned}|t \cdot \boldsymbol{w}| &= \sqrt{(t \cdot w^x)^2 + (t \cdot w^y)^2} \\ &= \sqrt{t^2 \cdot (w^x)^2 + t^2 \cdot (w^y)^2} \\ &= \sqrt{t^2 \cdot \left((w^x)^2 + (w^y)^2\right)}\end{aligned}$$

$$= \sqrt{t^2} \cdot \sqrt{(w^x)^2 + (w^y)^2}$$
$$= |t| \cdot |\boldsymbol{w}|.$$

Dies ist möglich, da die Wurzel aus t^2 gleich $|t|$.

Um unsere innere Kraft richtig darzustellen müssen wir $\overrightarrow{\boldsymbol{p}_1\boldsymbol{p}_2}$ nur richtig skalieren, also mit dem richtigen t multiplizieren. Das heißt wir müssen unser t so wählen, dass $|t| \cdot \ell = g$ gilt. Durch Umformen dieser Gleichung erhalten wir

$$|t| = \frac{g}{\ell} \qquad \text{und} \qquad |t| = \frac{k \cdot e}{\ell}.$$

Nun können wir die innere Kraft als Vektor $\boldsymbol{g}$ darstellen. Genauer haben wir

$$\boldsymbol{g} = \frac{k \cdot e}{\ell} \cdot (\boldsymbol{p}_2 - \boldsymbol{p}_1),$$

Hierbei haben wir für die Variable t das berechnete $\frac{k \cdot e}{\ell}$ eingesetzt. Das Vorzeichen von $\boldsymbol{g}$ richtet sich nach dem Vektor.

Da die Spannung g an beiden Enden des Stabes zur Hälfte wirkt, teilen wir sie noch durch zwei. Wenn wir nun e und ℓ einsetzen, bekommen wir folgende Formel:

$$\boldsymbol{g} = \frac{k \cdot \frac{\sqrt{((\boldsymbol{p}_2^x + \boldsymbol{v}_2^x) - (\boldsymbol{p}_1^x \boldsymbol{v}_1^x))^2 + ((\boldsymbol{p}_2^y + \boldsymbol{v}_2^y) - (\boldsymbol{p}_1^y + \boldsymbol{v}_1^y))^2}}{\sqrt{(\boldsymbol{p}_2^x - \boldsymbol{p}_1^x)^2 + (\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)^2}} - 1}{2 \cdot \sqrt{(\boldsymbol{p}_2^x - \boldsymbol{p}_1^x)^2 + (\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)^2}} \begin{pmatrix} \boldsymbol{p}_2^x - \boldsymbol{p}_1^x \\ \boldsymbol{p}_2^y - \boldsymbol{p}_1^y \end{pmatrix}$$

4 Gleichungssystem für die Verschiebungen

Unser Ziel ist es, bei gegebenen Punkten, auf die eine gegebene Kraft wirkt, die einzelnen Verschiebungen der Punkte auszurechnen. Zuletzt haben wir die innere Kraft g ausgerechnet.

Setzen wir nun teilweise die Formelzeichen für die Länge und für den Stab-Vektor wieder ein, ersetzen g durch $-f$ und schreiben die Brüche um, damit das Ganze etwas übersichtlicher wird, erhalten wir:

$$+\frac{k}{2\ell} \cdot (\ell^{-1} \sqrt{((\boldsymbol{p}_2^x + \boldsymbol{v}_2^x) - (\boldsymbol{p}_1^x + \boldsymbol{v}_1^x))^2 + ((\boldsymbol{p}_2^y + \boldsymbol{v}_2^y) - (\boldsymbol{p}_1^y + \boldsymbol{v}_1^y))^2} - 1) \cdot \overrightarrow{\boldsymbol{p}_1\boldsymbol{p}_2} = -\boldsymbol{f}$$

Innerhalb der großen Wurzel können wir jetzt die $\boldsymbol{p}$s und $\boldsymbol{v}$s ordnen:

$$+\frac{k}{2\ell}(\ell^{-1} \sqrt{((\boldsymbol{p}_2^x + \boldsymbol{p}_1^x) + (\boldsymbol{v}_2^x - \boldsymbol{v}_1^x))^2 + ((\boldsymbol{p}_2^y - \boldsymbol{p}_1^y) + (\boldsymbol{v}_2^y - \boldsymbol{v}_1^y))^2} - 1)\overrightarrow{\boldsymbol{p}_1\boldsymbol{p}_2} = -\boldsymbol{f}$$

Als Funktion, die von den x- und y-Verschiebungen $\boldsymbol{v}_1^x$, $\boldsymbol{v}_1^y$ des Punkts 1 und von den Verschiebungen $\boldsymbol{v}_2^x, \boldsymbol{v}_2^y$ des Punkts 2 abhängt und wir die Konstante $+\frac{k}{2\ell}$ vernachlässigen, sieht der Teil der Gleichung, den wir mit dem Vektor $\overrightarrow{\boldsymbol{p}^1\boldsymbol{p}^2}$ multiplizieren, so aus:

$$h(\boldsymbol{v}_1^x, \boldsymbol{v}_2^x, \boldsymbol{v}_1^y, \boldsymbol{v}_2^y) = \ell^{-1}\sqrt{((\boldsymbol{p}_2^x + \boldsymbol{p}_1^x) + (\boldsymbol{v}_2^x - \boldsymbol{v}_1^x))^2 + ((\boldsymbol{p}_2^y - \boldsymbol{p}_1^y) + (\boldsymbol{v}_2^y - \boldsymbol{v}_1^y))^2} - 1$$

Setzen wir alle Verschiebungen bis auf $\boldsymbol{v}_1^x$ auf 0, erhalten wir die Funktion $h(v_1^x, 0, 0, 0)$. Das heißt, wir verschieben nur einen Punkt lediglich um die x-Koordinate, die anderen Punkte verschieben sich nicht.

$$h(\boldsymbol{v}_1^x, 0, 0, 0) = \ell^{-1} \cdot \sqrt{((\boldsymbol{p}_2^x - \boldsymbol{p}_1^x) - \boldsymbol{v}_1^x)^2 + (\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)^2} - 1$$

Jetzt vereinfachen wir die Gleichungen und setzten statt $(\boldsymbol{p}_2^x - \boldsymbol{p}_1^x)$ der Einfachheit halber a und statt $(\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)^2$ einfach b. Als $\boldsymbol{v}_1^x$ setzen wir t. Dadurch erhalten wir die neue Gleichung:

$$\begin{aligned} h(t) &= \ell^{-1}\sqrt{(a-t)^2 + b} - 1 \\ &= \frac{1}{l}\sqrt{(a-t)^2 + b} - 1 \end{aligned}$$

4.1 Linearisierung der Funktion Um uns diese Funktion vorstellen zu können, nehmen wir die Punkte $\boldsymbol{p}_1 = (1, 2)$ und $\boldsymbol{p}_2 = (2, 3)$, die durch einen Stab verbunden sind. Im Koordinatensystem in Abbildung 6 wird die Situation nochmals verdeutlicht. Die Graph der Funktion mit diesen Punkten sieht so aus, wie in Abbildung 7 dargestellt.

Wir erkennen, dass die Funktion um die Stelle 0 annähernd linear verläuft. Deswegen wollen wir sie linearisieren. Dazu bilden wir die erste Ableitung der Funktion h, um Steigung der Funktion an der Stelle 0 herauszufinden. Dann finden wir den Funktionswert an der Stelle 0 und infolgedessen eine lineare Funktion aufzustellen. Dazu nützen wir die Taylor-Entwicklung aus:

$$h(t) \approx h(t_0) + h'(t_0) \cdot (t - t_0)$$

Die Ableitung der Funktion $h(t)$

$$\begin{aligned} h(t) &= \ell^{-1} \cdot \sqrt{(a-t)^2 + b} - 1 \\ h'(t) &= \ell^{-1} \cdot \frac{1}{2} \cdot ((a-t)^2 + b)^{-\frac{1}{2}} \cdot 2 \cdot (a-t) \cdot (-1) \\ h'(t) &= \frac{2 \cdot (a-t) \cdot (-1)}{2 \cdot \ell \cdot \sqrt{(a-t)^2 + b}} \end{aligned}$$

$$h'(t) = \frac{t-a}{\ell \cdot \sqrt{(a-t)^2 + b}}$$

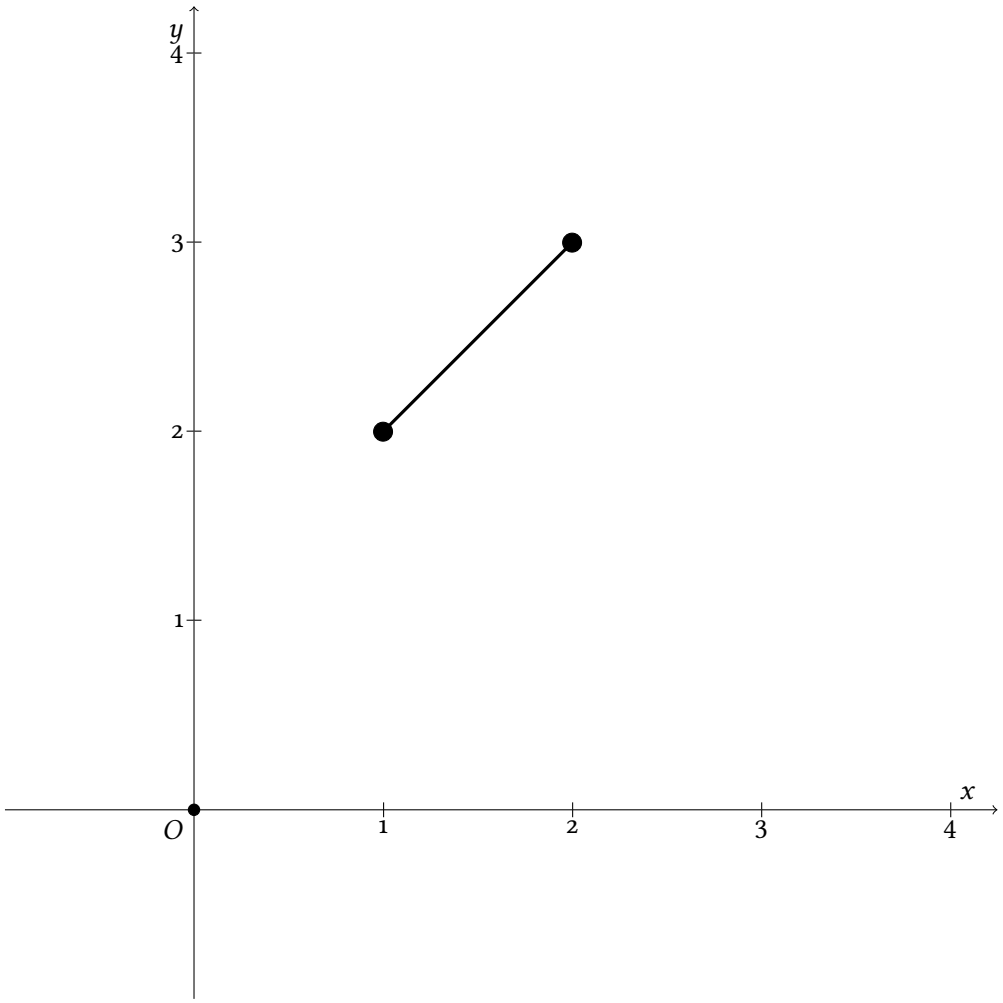


Abbildung 6: Darstellung zweier angenommener Punkte im Koordinatensystem

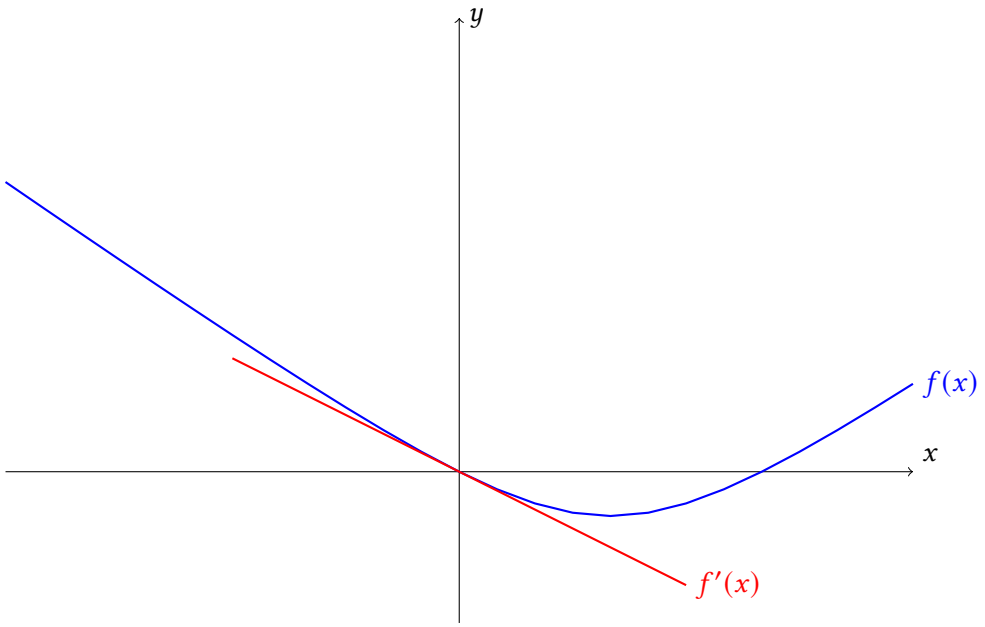


Abbildung 7: Graphische Veranschaulichung der Funktion

Die Ableitung an der Stelle 0 ist

$$h'(0) = \frac{-a}{\ell \cdot \sqrt{a^2 + b}}.$$

Der Funktionswert an der Stelle 0 ist

$$h(0) = \ell^{-1} \cdot \sqrt{a^2 + b} - 1.$$

Also ist die linearisierte Funktion

$$f(t) = (\frac{1}{\ell} \cdot \sqrt{a^2 + b} - 1) - (\frac{1}{\ell} \cdot \frac{a}{\sqrt{a^2 + b}}) \cdot t.$$

Hier können wir folgendes einsetzen:

$$\ell = \sqrt{a^2 + b},\ a = (\boldsymbol{p}_{\boldsymbol{x}}^2 - \boldsymbol{p}_{\boldsymbol{x}}^1),\ b = (\boldsymbol{p}_{\boldsymbol{y}}^2 - \boldsymbol{p}_{\boldsymbol{y}}^1)^2,\ \text{und}\ \ell = \sqrt{(\boldsymbol{p}_{\boldsymbol{x}}^2 - \boldsymbol{p}_{\boldsymbol{x}}^1)^2 + (\boldsymbol{p}_{\boldsymbol{y}}^2 - \boldsymbol{p}_{\boldsymbol{y}}^1)^2}$$

Wenn wir die Gleichung vereinfachen, erhalten wir:

$$f(t) = -\frac{a}{\ell^2} \cdot t$$

oder – mit eingesetztem a:

$$f(t) = -\frac{(\boldsymbol{p}_{\boldsymbol{x}}^2 - \boldsymbol{p}_{\boldsymbol{x}}^1)}{\ell^2} \cdot t$$

Das ist also die linearisierte Funktion, die uns für eine gegebene x-Verschiebung eines einzelnen Punktes die für die Verschiebung benötigte Kraft gibt.

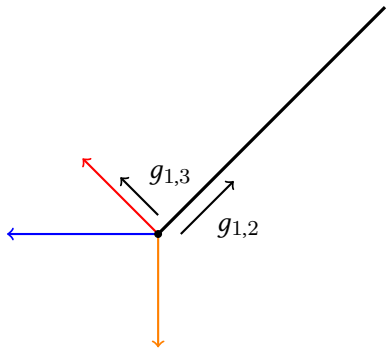


Abbildung 8: Die Wirkung der Kräfte

4.2 DIE KRAFT BEI ALLEN VERSCHIEBUNGEN Die Kraft hängt bei einem Stab nun nicht nur von einer Verschiebung, sondern von vier Verschiebungen – zwei x-Verschiebungen und zwei y-Verschiebungen ab. Hier noch einmal die Gleichung:

$$+\frac{k}{2\ell}(\ell^{-1}\sqrt{((\boldsymbol{p}_2^x-\boldsymbol{p}_1^x)+(\boldsymbol{v}_2^x-\boldsymbol{v}_1^x))^2+((\boldsymbol{p}_2^y-\boldsymbol{p}_1^y)+(\boldsymbol{v}_2^y-\boldsymbol{v}_1^y))^2}-1)\cdot(\boldsymbol{p}_2-\boldsymbol{p}_1)=-\boldsymbol{f}_1$$

Um die Funktion, die von allen Verschiebungen abhängt, zu bekommen, leiten wir die Funktion zuerst nicht nur nach einem v, sondern von allen vs ab und finden die Ableitung an der Stelle 0:

$$\frac{1}{\ell^2}\cdot(\boldsymbol{p}_2^x-\boldsymbol{p}_1^x)=f'_{\boldsymbol{v}_2^x}(0,0,0,0) \qquad \frac{1}{\ell^2}\cdot(\boldsymbol{p}_1^x-\boldsymbol{p}_2^x)=f'_{\boldsymbol{v}_1^x}(0,0,0,0)$$
$$\frac{1}{\ell^2}\cdot(\boldsymbol{p}_2^y-\boldsymbol{p}_1^y)=f'_{\boldsymbol{v}_2^y}(0,0,0,0) \qquad \frac{1}{\ell^2}\cdot(\boldsymbol{p}_1^y-\boldsymbol{p}_2^y)=f'_{\boldsymbol{v}_2^y}(0,0,0,0)$$

Die Funktion, die von allen vs abhängt, erhalten wir, wenn wir den Funktionswert an der Stelle 0 ausrechnen und dann die Ableitungen nach allen vs jeweils an der Stelle 0 addieren.

$$\begin{aligned}f(v_2^x,v_1^x,v_2^y,v_1^y)\approx f(0,0,0,0)&+f'_{v_2^x}(0,0,0,0)\cdot v_2^x+f'_{v_1^x}(0,0,0,0)\cdot v_1^x\\&+f'_{v_2^y}(0,0,0,0)\cdot v_2^y+f'_{v_1^y}(0,0,0,0)\cdot v_1^y\end{aligned}$$

Wir setzten die Ableitungen ein. Nun haben wir den linearisierten Faktor, mit dem wir den Vektor $\overrightarrow{\boldsymbol{p}^1\boldsymbol{p}^2}$ multiplizieren können, um $-f$ zu bekommen. Wenn wir nur den x-Wert von $-f$ wollen, müssen wir auch nur mit den x-Werten des Vektors multiplizieren. Daraus ergibt sich folgende Gleichung:

$$\begin{aligned}\frac{k}{2\ell^3}(\boldsymbol{p}_2^x-\boldsymbol{p}_1^x)^2\cdot\boldsymbol{v}_2^x+\frac{k}{2\ell^3}(\boldsymbol{p}_1^x-\boldsymbol{p}_2^x)(\boldsymbol{p}_2^x-\boldsymbol{p}_1^x)\cdot\boldsymbol{v}_1^x+\frac{k}{2\ell^3}(\boldsymbol{p}_2^y-\boldsymbol{p}_1^y)(\boldsymbol{p}_2^x-\boldsymbol{p}_1^x)\cdot\boldsymbol{v}_2^y\\+\frac{k}{2\ell^3}(\boldsymbol{p}_1^y-\boldsymbol{p}_2^y)(\boldsymbol{p}_2^x-\boldsymbol{p}_1^x)\cdot\boldsymbol{v}_1^y=-f_1^x\end{aligned}$$

Heben wir $(\boldsymbol{p}_2^x - \boldsymbol{p}_1^x)$ bzw. auch $(\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)$ heraus, erhalten wir für $-f_1^x$:

$$\frac{k}{2\ell^3}(\boldsymbol{p}_2^x - \boldsymbol{p}_1^x)^2(\boldsymbol{v}_2^x - \boldsymbol{v}_1^x) + \frac{k}{2\ell^3}(\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)(\boldsymbol{p}_2^x - \boldsymbol{p}_1^x)(\boldsymbol{v}_2^y - \boldsymbol{v}_1^y) = -f_1^x$$

Schon herausgehoben sieht die Gleichung für $-f_1^y$ also so aus:

$$\frac{k}{2\ell^3}(\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)(\boldsymbol{p}_2^x - \boldsymbol{p}_1^x)(\boldsymbol{v}_2^x - \boldsymbol{v}_1^x) + \frac{k}{2l\ell^3}(\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)^2(\boldsymbol{v}_2^y - \boldsymbol{v}_1^y) = -f_1^y$$

Wenn wir auch die x-Koordinate der Kraft f_2^x im zweiten Punkt berechnen wollen, müssen wir nur die Einser und Zweier vertauschen. Ohne Herausheben sieht es so aus:

$$-f_2^x = \frac{k}{2\ell^3}(\boldsymbol{p}_1^x - \boldsymbol{p}_2^x)^2 \cdot \boldsymbol{v}_1^x + \frac{k}{2\ell^3}(\boldsymbol{p}_2^x - \boldsymbol{p}_1^x)(\boldsymbol{p}_1^x - \boldsymbol{p}_2^x) \cdot \boldsymbol{v}_2^x +$$
$$\frac{k}{2\ell^3}(\boldsymbol{p}_1^y - \boldsymbol{p}_2^y)(\boldsymbol{p}_1^y - \boldsymbol{p}_2^y) \cdot \boldsymbol{v}_1^y + \frac{k}{2\ell^3}(\boldsymbol{p}_2^y - \boldsymbol{p}_1^y)(\boldsymbol{p}_1^x - \boldsymbol{p}_2^x) \cdot \boldsymbol{v}_2^y$$

4.3 GLEICHUNGSSYSTEM MIT MATRITZEN AUFSTELLEN Um bei gegebenen Kräften f die Verschiebungen v ausrechnen zu können, werden wir ein Gleichungssystem aufstellen müssen. Dazu werden wir Matritzen verwenden. Wir zeigen zuerst anhand des folgenden Beispiels, wie man Matritzen zum Lösen von Gleichungen verwenden kann:

Beispiel 4.1. Gesucht sind x_1 und x_2. Es gilt:

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

Wenn wir die Matrix A mit dem Vektor x multiplizieren, erhalten wir den Vektor b.

$$A \cdot x = b$$

Nun setzen wir für A, x und b die richtigen Werte ein, also

$$A = \begin{pmatrix} a_{11}a_{12} \\ a_{21}a_{22} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad \text{und} \quad b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

So kann dieses Gleichungssystem mithilfe von Matritzen bzw. Vektoren aufgestellt werden.

In unserem Fall stellen wir das Gleichungssystem für einen Stab (zwei Punkte) so auf:

$$A \cdot \begin{pmatrix} \boldsymbol{v}_1^x \\ \boldsymbol{v}_2^x \\ \boldsymbol{v}_1^y \\ \boldsymbol{v}_2^y \end{pmatrix} = \begin{pmatrix} -f_1^x \\ -f_2^x \\ -f_1^y \\ -f_2^y \end{pmatrix}$$

Wobei A die folgende Matrix ist:

$$A = -\frac{k}{2\ell^3} \cdot \begin{pmatrix} (p_1^x - p_2^x)(p_1^x - p_2^x) & (p_2^x - p_1^x)(p_1^x - p_2^x) & (p_1^y - p_2^y)(p_1^x - p_2^x) & (p_2^y - p_1^y)(p_1^x - p_2^x) \\ (p_1^x - p_2^x)(p_2^x - p_1^x) & (p_2^x - p_1^x)(p_2^x - p_1^x) & (p_1^y - p_2^y)(p_2^x - p_1^x) & (p_2^y - p_1^y)(p_2^x - p_1^x) \\ (p_1^x - p_2^x)(p_1^y - p_2^y) & (p_2^x - p_1^x)(p_1^y - p_2^y) & (p_1^y - p_2^y)(p_1^y - p_2^y) & (p_2^y - p_1^y)(p_1^y - p_2^y) \\ (p_1^x - p_2^x)(p_2^y - p_1^y) & (p_2^x - p_1^x)(p_2^y - p_1^y) & (p_1^y - p_2^y)(p_2^y - p_1^y) & (p_2^y - p_1^y)(p_2^y - p_1^y) \end{pmatrix}$$

Wir erkennen: die Koordinaten und die Zahlen des einen Faktors hängen jeweils von den zugehörigen vs ab, die Koordinaten und Zahlen des anderen Faktors von den zugehörigen fs ab. (Die erste Zahl ist jeweils der Index des Punktes, auf den sich v oder f beziehen.)

Nun versuchen wir, die Matrix A nicht nur für zwei Punkte (einen Stab) aufzustellen, sondern für beliebig viele Punkte. Wir stellen also eine Matrix für drei Punkte auf. Bei drei Punkten müssen wir sechs Verschiebungskoordinaten v pro Kraftkoordinate berücksichtigen. Für f_1^x sieht das so aus:

$$\begin{aligned} &-\tfrac{k_{1,2}}{2\ell^3}(p_1^x - p_2^x)(p_1^x - p_2^x)v_1^x - \tfrac{k_{1,2}}{2\ell^3}(p_2^x - p_1^x)(p_1^x - p_2^x)v_2^x \\ &-\tfrac{k_{1,2}}{2\ell^3}(p_1^y - p_2^y)(p_1^x - p_2^x)v_1^y - \tfrac{k_{1,2}}{2\ell^3}(p_2^y - p_1^y)(p_1^x - p_2^x)v_2^y \\ &-\tfrac{k_{1,3}}{2\ell^3}(p_1^x - p_2^x)(p_1^x - p_2^x)v_1^x - \tfrac{k_{1,3}}{2\ell^3}(p_2^x - p_1^x)(p_1^x - p_2^x)v_3^x \\ &-\tfrac{k_{1,3}}{2\ell^3}(p_1^y - p_2^y)(p_1^x - p_2^x)v_1^y - \tfrac{k_{1,3}}{2\ell^3}(p_2^y - p_1^y)(p_1^x - p_2^x)v_3^y = -f_1^x \end{aligned}$$

Wenn wir die Matrix aufstellen, müssen wir die orangenen Faktoren an den orangenen Stellen einfügen – den Faktor zu v_2^x links, den zu v_2^y rechts. Dasselbe gilt für die blauen Faktoren.

Um die Faktoren an den violetten Stellen zu erhalten, müssen wir mehrere Elemente miteinander addieren - für die linke violette Stelle sind das die erste und die dritte Zeile des violetten Blocks, für die rechte violette Stelle die zweite und vierte Zeile. Wenn wir diese Faktoren genauer betrachten, erkennen wir, dass die einzelnen Summanten auch jeweils die orangen und blauen Faktoren der eigenen Hälfte sein können.

$$\left(\begin{array}{ccc|ccc} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{array}\right) \cdot \begin{pmatrix} v_1^x \\ v_2^x \\ v_3^x \\ \hline v_1^y \\ v_2^y \\ v_3^y \end{pmatrix} = \begin{pmatrix} -f_1^x \\ -f_2^x \\ -f_3^x \\ \hline -f_1^y \\ -f_2^y \\ -f_3^y \end{pmatrix}$$

Mit diesem Wissen können wir nun eine Matrix mit N Punkten aufstellen. n und m sind zwei beliebige Zahlen zwischen 1 und der doppelten Punktanzahl N. k_{nm} ist die Federkonstante des Stabes zwischen den Punkten n und m. Existiert zwischen n und m kein Stab, so ist die Federkonstante 0. In der Realität sind die Stäbe S_{nm} und S_{mn} ein und derselbe Stab, der in der Matrix allerdings doppelt angeschrieben wird.

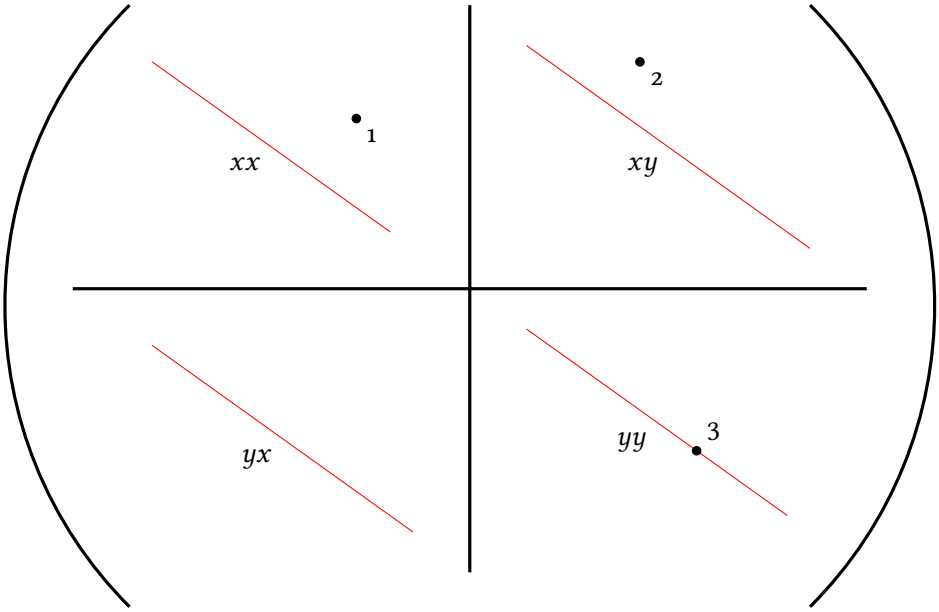


Abbildung 9: Das SSystem"hinter der Matrix

Der Punkt 1 ist ein Punkt n, m, bei dem sowohl n als auch m kleiner sind als die Gesamtpunktanzahl. Der Eintrag hier ist:

$$\frac{-k_{nm}}{2 \cdot \ell^3} \cdot (\boldsymbol{p}_n^x - \boldsymbol{p}_m^x) \cdot (\boldsymbol{p}_m^x - \boldsymbol{p}_n^x)$$

Der Eintrag beim Punkt 2, welcher ein Punkt n, m ist, bei dem n kleiner als die Gesamtpunktanzahl N ist und m größer, sieht folgendermaßen aus:

$$\frac{-k_{nm}}{2 \cdot \ell^3} \cdot (\boldsymbol{p}_m^y - \boldsymbol{p}_n^y) \cdot (\boldsymbol{p}_n^x - \boldsymbol{p}_m^x)$$

Punkte, die links oder rechts unten wären, hätten ein n, das größer als die Gesamtpunktanzahl N ist. Links wäre m kleiner, rechts größer als N. Einträge für unten links:

$$\frac{-k_{nm}}{2 \cdot \ell^3} \cdot (\boldsymbol{p}_m^x - \boldsymbol{p}_n^x) \cdot (\boldsymbol{p}_n^y - \boldsymbol{p}_m^y)$$

Für unten rechts:

$$\frac{-k_{nm}}{2 \cdot \ell^3} \cdot (\boldsymbol{p}_n^y - \boldsymbol{p}_m^y) \cdot (\boldsymbol{p}_m^y - \boldsymbol{p}_n^y)$$

Für Punkte, die auf den roten Diagonalen liegen, gilt: $n = m$. Wie bei der violetten Stelle vorhin können wir die Werte der Zeile, die im selben Viertel liegen, summieren, um den Wert zu bekommen.

$$\sum_{m=1}^{N} \frac{k_n}{2\ell^3} (\boldsymbol{p}_n^y - \boldsymbol{p}_m^y)^2 = -\sum_{m=1}^{N} a_{nm}$$

Wenn wir diese Matrix aufstellen und den f-Vektor festlegen, so können wir das Gleichungssystem lösen und den v-Vektor, also die Verschiebungen der einzelnen Punkte, bekommen.

5 Erstellung einer graphischen Darstellung mit Python

Um all unsere theoretischen Berechnungen anwenden zu können, haben wir in Python ein Programm erstellt, das anhand von verschiedenen kleinen miteinander verknüpften Funktionen, ein Fachwerk darstellt und wenn eine Kraft darauf angewendet wird, sich dementsprechend verhält, sodass die Stäbe entweder gesteckt oder gestaucht werden. Das haben wir mit Farben veranschaulicht, so sind zum Beispiel grüne Punkte beweglich während schwarze fest sind. Bei den Linien bedeutet blau eine Streckung und rot eine Stauchung. Prinzipiell haben wir ein Gleichungssystem erstellt, welches wir mit Numpy gelöst haben und die grahphische Darstellung erfolgte mit pygames.

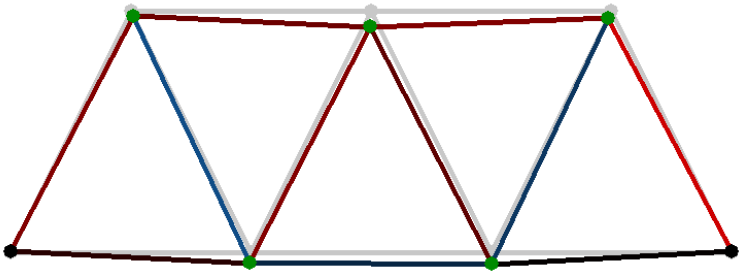


Abbildung 10: Verschiebung einer Fachwerk-Brücke unter Einfluss von Kräften

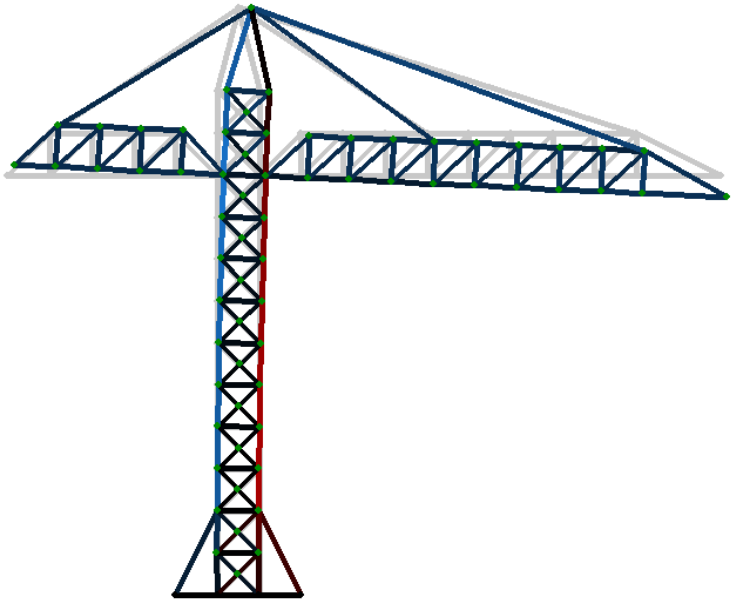


Abbildung 11: Wirkungen eines Gewichts auf den Kran

6 Danksagung

Diese Woche der Modellierung mit Mathematik im Jufa Leibnitz hat uns allen die Welt der Mathematik und wie damit eindrucksvoll die Realität ein Stück weit angenähert und anhand von Modellen simuliert werden kann, veranschaulicht. Mit einer Engelsgeduld hat unser Betreuer Dr. Robert Beinert, MSc BSc uns bei ausnahmslos allem unterstützt und damit einen erheblichen Teil zum Gelingen dieses Projekts in seiner Form beigetragen hat.

Lieber Robert, vielen Dank dafür! Ohne dich hätten wir das wohl nie geschafft!

Projekt:
Probabilistische Physik
Die Entstehung von Determinismus aus Indeterminismus

Florian Wedenig, Sebastian Andritsch, Mike Jesenik, Nik Kummert, Johannes Trinkaus, Valentin Posch

Betreut von
Mag. Dr. Stephen Keeling
a.k.a. Mister Modellierungswoche

Inhaltsverzeichnis

Einleitung

Ist Zufall nur das Resultat mangelnder Information? Fragen wie diese waren für viele von uns der Ansporn, das Thema der Probabilistischen Physik zu wählen. Egal ob man das Thema nun aus philosophischem oder mathematischem Interesse gewählt hat, man kam immer auf seine Kosten. Denn sowohl das mathematische Modellieren verschiedener Problemstellungen als auch das Hinterfragen von für uns selbstverständlich scheinenden Systeme und Modelle haben ein endlos scheinendes Potential, das dementsprechend unsere Faszination geweckt hat. Wir hätten niemals erwartet, in welche Dimensionen wir mit diesem Gedankengut kommen konnten. Von einfachen Münzwürfen haben wir unser Wissen bis zur Quantenmechanik aufgebaut. Selbst wenn wir nicht die Chance hatten, alles bis aufs kleinste Detail durchzuarbeiten und zu verstehen, hat uns dieser Einblick doch geprägt. In nur ein paar Intensiven Arbeitstagen kamen wir bis zum Ende des Projekts und hatten noch Zeit, alles zu verdauen. All das hätte nie ohne die gezielte und erfolgversprechende Führung von unserem Betreuer Stephen Keeling funktioniert. Wir wollen Steve gleich am Anfang von diesem Bericht unseren Dank für diese sehr spannende und informative Woche aussprechen. Ohne ihn wäre dieses Projekt nie so aufgegangen und wir hätten niemals so viel mitgenommen. Er war immer bereit, auf individuelle Interessen einzugehen und hat diese immer voller Elan unterstützt.

Einführung in die Wahrscheinlichkeiten

Unser erster Schritt in die Welt der Wahrscheinlichkeiten war, Begriffe wie Zufall, Wahrscheinlichkeiten und Zustand zu klären, um darauf vorbereitete Fragestellungen beantworten zu können. Fragen wie "*Was ist mit "Natürliches Gesetz" gemeint?*" und "*Was ist ein Zufallsgenerator*?" waren in der Gruppe schnell beantwortet. Für uns ist das "Natürliche Gesetz" eine quasi bekannte und erwartete Norm. Um es einfach zu sagen, "der Himmel war gestern blau, also wird er morgen auch blau sein." und ein Zufallsgenerator ist, trotz des verführerischen Namens, nicht zufällig. Ein Zufallsgenerator arbeitet immer nach einem Algorithmus. Sobald ein Algorithmus dabei ist, gibt es keinen Zufall. Also ist er lediglich ein Programm, das Zahlen ausspuckt. Als nächstes behandelten wir die Themen Determinismus und Indeterminismus. Zuerst musste uns erklärt werden, was die Begriffe bedeuten. Determinismus ist eine Weltansicht, bei der man jeden Schritt als vorhergesehen sieht. Also gibt es keine Zufälle, sondern nur fixe Zustände. Indeterminismus ist die gegenteilige Ansicht. Jeder Zustand ist aus einem Zufall entstanden und jeder Schritt ist willkürlich.

Wir kamen zum Schluss, dass die mikroskopische Welt sehr indeterministisch ist und die Makroskopische Welt eher deterministisch. Also kann man nicht genau sagen, welche die korrekte Ansicht ist. Allerdings fanden wir auch heraus, dass die Wahrscheinlichkeit im Grunde dafür verwendet werden kann, um reale Zufälle bestmöglich beschreiben zu können.

In der Mathematik ist die Wahrscheinlichkeit der Anteil der eingetroffenen Zustände inzwischen von vielen anderen möglichen Zuständen. Zum Beispiel, bei dem Münzenwurf kann man Kopf oder Zahl herausbekommen. Daher tritt bei einer hinreichend großen Menge an Versuchen bei der Hälfte der Würfe Zahl und bei der anderen Hälfte Kopf ein. Allerdings gibt es noch zwei wichtige Begriffe in der Mathematik zu klären.

Kommentiert [Ga1]: formulierung

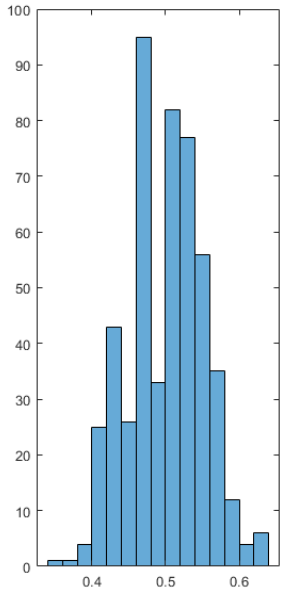
Diese sind der Erwartungswert und die Varianz. Wenn wir wieder den Münzwurf heranziehen, aber bei jedem Kopf 1€ bekommt und bei Zahl 0€ erhalten, erwarten wir 1€*0.5 + 0*0.5 Euro (50 Cent). Wenn wir dieses Experiment zehn Mal durchführen, erwarten wir also 10*0.5Euro=5€

Gewinn. Allerdings könnte in zehn Würfen mehr Kopf als Zahl bzw. andersrum vorkommen. Dann würden wir weniger bzw. Mehr als 5€ gewinnen. Diese Abweichungen ergeben dann die Varianz, also die Streuung um den Erwartungswert.

Ersten Beispiele

a.) Münzen

Unser erstes konkretes Beispiel, welches wir selbst und eigenständig programmierten, waren die Wahrscheinlichkeiten beim Münzwurf. Dieser stellt das einfachste Beispiel für einen Zufallsversuch dar, da es nur zwei Möglichkeiten als Ergebnis gibt und die Wahrscheinlichkeiten konstant bleiben. Zur Simulierung des Zufalls benutzten wir die in Matlab vorhandene Funktion randi(). Diese wählt aus dem vorgegebenen Bereich (in diesem Fall 0 bis 1) eine pseudozufällige, ganze Zahl aus. Dabei entspricht 0 Kopf und 1 Zahl. Dies wird n-mal durchgeführt. Die Summe dieser Ergebnisse wird in y gespeichert und letztendlich als Histogramm dargestellt. Das sieht dann so aus:



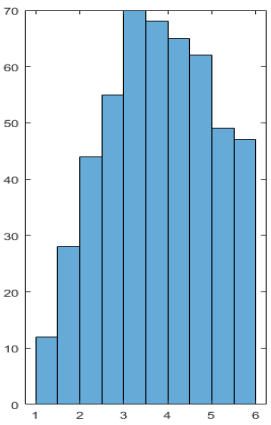
Dies zeigt die Verteilung der Resultate der einzelnen Versuche. Beispielsweise ist dem Histogramm zu entnehmen, dass bei einem der 500 Versuche nur etwa

35 Mal Zahl herauskam. Etwa zehn Mal betrug das Ergebnis mehr als 60 von 100. Eine so große Abweichung bei 100 Münzwürfen steht für eine große Varianz.

```
n = 100; %Anzahl der Münzwürfe, die in den Versuch einfließen
repetitions = 500; %Anzahl der Versuche
y = zeros(1, repetitions); %Array zur Speicherung der Versuchsresultate
% for-Schleife zur Simulation der Versuchsresultate
for i=1:repetitions
    x = randi([0 1], 1, n); % Generierung der Zufallswerte, 0 oder 1  % [1
6] für Würfel
    y(i) = sum(x)/n; % Speicherung der Zufallswerte eines Versuchs
end

mean(y) %Mittelwert aller Versuche
histogram(y) %Histogramm der Versuchsdurchschnitte
```

b.) Würfel

Das zweite Beispiel hatte den Sinn, zu überprüfen ob wir die Problemstellung vom ersten Thema verstanden haben. Während wir bei dem ersten Experiment die Möglichkeiten von Kopf und Zahl beziehungsweise 0 und 1 hatten, haben wir hier pro geworfenen Würfel eine Chance auf 1 bis 6. Wenn man den Code von dem vorherigen Beispiel (Münzwurf) anschaut, dann fällt einem auf, dass sie grundsätzlich genau gleich sind. Der wesentliche Unterschied ist, dass die Wahrscheinlichkeit einer Zahl nicht mehr 50% beträgt, sondern nun 1/6. Der Erwartungswert von einmal Würfeln beträgt 3,5. Da wir in Zeile 8 dieses „/n" haben, ändert sich der Erwartungswert nicht, auch wenn wir die Anzahl der Würfel erhöhen. Auf dem folgenden Histogramm kann man dann den Erwartungswert und die Varianz erkennen.




```
n =2;    %Anzahl der Würfel
repetitions = 500;    %Anzahl der Versuche

y = zeros(1, repetitions);      %Array zur Speicherung der Versuchsresultate

for i=1:repetitions           %Schleife die so oft wiederholt wird, so groß
"repetitions" ist
    x = randi([1 6], 1, n);      %zufällige Zahl zwischen 1 und 6
    y(i) = sum(x)/n;         % Speicherung der Zufallswerte eines Versuchs
end

mean(y)      %Mittelwert aller Versuche
subplot(1,2,1)       %unwichtig
histogram(y)       %Dieser Befehl erstellt ein Histogramm mit den errechneten
Werten
```

Geleebohnenversuch

Unser Interesse an diesem Versuch hat mit dem Video bei der Themenvorstellung begonnen. Zuerst war es unklar, wie dieser Versuch funktionieren konnte, aber gleich am ersten Tag lernten wir, dass er auf dem Prinzip des Grenzwertsatzes basiert. Zusammengefasst sagt er, wenn man sehr, sehr viele Daten, beziehungsweise Schätzungen zur Verfügung hat, wird der Durchschnitt der Schätzungen dem Erwartungswert immer näherkommen. Also selbst wenn jemand eine Zahl schätzt, die komplett unlogisch ist, braucht man nur genug Ausgleichsdaten und der Durchschnitt wird sich wieder anpassen.

Wir konnten nicht wiederstehen und führten diesen Versuch selber durch. Zuerst befüllten wir einen Krug mit 825 Skittles und führten dann zwei Befragungen durch. Die Erste erfolgte nur mit den Teilnehmern und Betreuern der Modellierungswoche, eine Zweite wurde bei verschiedensten Personen aus Leibnitz durchgeführt.

Bei der ersten Befragung bekamen wir insgesamt 36 verschiedene Schätzungen. Von 120 bis 2000 Skittles war alles dabei. Der Erwartungswert von 825 wurde leider nicht erreicht. Die durchschnittliche Schätzung lag bei 676.2917 Bohnen. Daraufhin entschieden wir uns dazu, noch mehr Personen zu befragen denn umso mehr Daten wir hatten, umso eher würden wir den Grenzwertsatz erfüllen.

Bei der zweiten Befragung bekamen wir 284 verschiedene Schätzungen. Wie erwartet kamen wir viel näher an den Erwartungswert. Die Durchschnittliche Schätzung lag bei **826** Skittles. Das entspricht der Abweichung einer einzelnen Bohne. Durch eine Berechnung mithilfe eines Konfidenzintervalls erfuhren wir, dass die Wahrscheinlichkeit für eine solche Genauigkeit bei dieser Anzahl an Probanden bei 0,3 % lag. Wir waren sehr erstaunt, aber auch zufrieden gestellt.

Wahrscheinlichkeiten und Teilchen

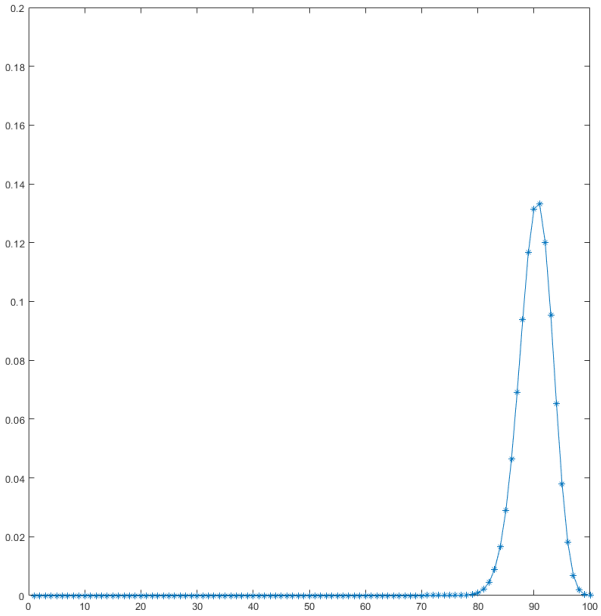
a.) Teilchenzerfall

Beim Teilchenzerfall ging es darum, einen passenden Code zu erstellen, mit dem die Zerfallswahrscheinlichkeit eines Teilchens mit Bezug zu Zeit modelliert werden kann. Dabei soll ein Teilchen in einem Zeitintervall von *0* bis *t* eine gewisse Wahrscheinlichkeit haben zu zerfallen. Hierzu wird eine Zufallsvariable *X(t)* mit *X(t) = 1*, wenn das Teilchen innerhalb einer vorgesehenen Zeit zerfällt, und *X(t) = 0*, wenn dies nicht passiert, eingeführt. Es ergibt sich bei genauerer Betrachtung die Zerfallsfunktion $P(X(t)=1) = e^{-\lambda t}$ und folglich $P(X(t)=0)=1-e^{-\lambda t}$. Für den Erwartungswert erhält man $\mu X(t)=E(X(t)) = P(X(t)=0)\cdot 0+P(X(t)=1)\cdot 1=e^{-\lambda t}$ und für die Varianz $V(X(t)) = E((X(t)-\mu X(t))^2) = \sigma^2 X(t) = P(X(t)=0)(0-e^{-\lambda t})^2+P(X(t) =1)(1-e^{-\lambda t})^2 = e^{-\lambda t}(1-e^{-\lambda t})$. Für die Wahrscheinlichkeit des Zerfalls mehrerer Teilchen werden die Zufallsvariablen der einzelnen Teilchen mit einander multipliziert.

Der Code dafür sieht aus wie folgt:

```
n = 100; % Teilchenanzahl

p = zeros(1,n+1); % Array mit Wschlkeiten

lambda = 0.01; % x - Koeffizient der e^x - fkt. MUSS: positiv sein; Je größer, desto schnellere Abnahme

%x = linspace(0,n,n+1);


tmax=100; %Maximale Zeit


% for - Schleife zur Berechnung der Wahrscheinlichkeit zu jedem zeitschritt

for t=0:0.1:tmax


    % for - Schleife zur Berechnung der Wahrscheinlichkeit, dass nach der

    % Zeit t eine bestimmte Teilchenzahl noch existiert

    for i=1:n
```

```
        p(i)    =    nchoosek(n,    i)    *    (exp(-lambda*t)).^i*(1-exp(-
lambda*t)).^(n-i);
    end
    plot(p) % Erzeugung der Grafik
    axis([0 n 0 0.2]) % Festlegung der Achsen
    drawnow % nötig, da sonst innerhalb einer Schleife nichts gezeichnet
wird

    sum(p) % Kontrollsumme, ob die Summe aller Wahrscheinlichkeiten 1 ist.

end
```



Hier sieht man das Ergebnis des Codes nach 10 Zeitschritten. Auf der X-Achse ist die Teilchenzahl aufgetragen, auf der Y-Achse ist die Wahrscheinlichkeit für jede einzelne Teilchenzahl aufgetragen. Nach 10 Zeitschritten liegt die Wahrscheinlichkeit, dass 90 von 100 Teilchen noch existieren, also bei etwa 0.13. Dies ist abhängig von dem gewählten `lambda`, welches in diesem Fall bei 0.01 liegt.

b.) Teilchenverteilung im Raum

Es befinden sich eine Vielzahl von Teilchen in einem Behälter der in Abschnitte, “Bins”, unterteilt ist. Dann wird davon ausgegangen, dass es gleich wahrscheinlich ist, dass sich ein beliebiges Teilchen in einem beliebigen Bin befindet. Man betrachtet die absoluten Anzahlen von den Teilchen in den Bins, wobei es natürlich nicht mehr Bins als Teilchen geben soll. Die Schlussfolgerung ist, dass je größer die Anzahl der Teilchen ist, desto wahrscheinlicher ist die gleichmäßige Verteilung dieser unter den Bins. Analog dazu kann man sich vorstellen, wie unwahrscheinlich es ist, dass sich alle Stick - und Sauerstoffmoleküle in einer Ecke eines geschlossenen Raumes sammeln.

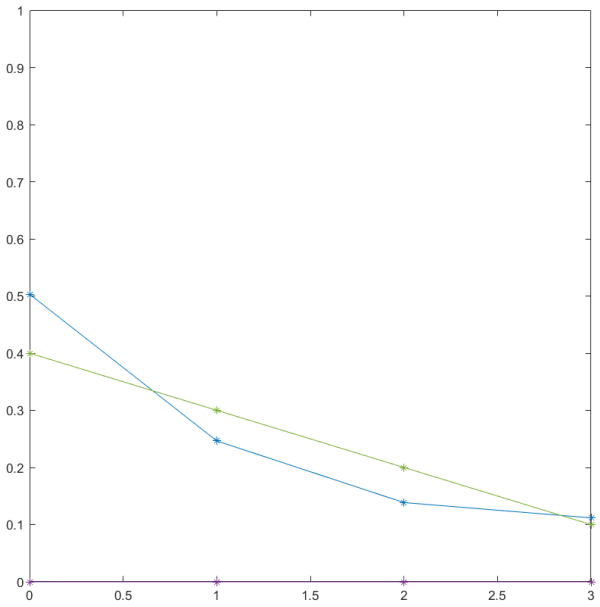
c.) sich bewegende Teilchen im Raum

Zuerst haben wir uns nur mit stationären Teilchen beschäftigt. Aber bald schon kam die Frage, ob wir auch Simulationen mit sich bewegenden Teilchen Modellieren konnten. Also überlegten wir. Mit der Hilfe von Steve kamen wir zum Schluss, dass wir die Bewegung von Teilchen auch als ihre Energie sehen konnten und nur eine bestimmte Menge an Energie im Raum zu Verfügung stehe. Also sahen wir ihre unterschiedlichen Energien als Kisten. Im Endeffekt hatten wir eine lange Ordnung an Kisten. In der ersten Kiste sind Teilchen mit keiner Energie also keiner Bewegung und in der letzten Kiste Teilchen mit der höchsten Energie, also der schnellsten Bewegung. Wir sahen ihre Energien wie Bereiche im Raum. Schnelle Teilchen fliegen quasi weiter weg als langsame oder sich nicht bewegende Teilchen. Der einzige große Unterschied war, dass die Teilchen nichtmehr alle unbeeinflusst von einander waren. Wenn das 10 von 100 Teilchen bereits das Energiemaximum erreicht hatte, konnten die anderen Teilchen nur noch in die Kiste mit keiner Energie.

```
n = 3; % Teilchenzahl


nOfEnivs = 4; % Anzahl der Energieniveaus(0 bis nOfEnivs-1)
```

```
nOfCurrentEnivs = nOfEnivs; % Zur Energieverteilung benötigte
Größe


E = 3; % Gesamtenergie des Systems


Enivs = zeros(1,nOfEnivs); % Array zur Speicherung der
Teilchenzahl pro Energieniveau


x = linspace(0,nOfEnivs-1, nOfEnivs); % x-Achse (für jeden Wert
in x wird ein Punkt geplottet)


sampleSize = 10000; % Versuchsanzahl


samples = zeros(nOfEnivs); % Array zur Speicherung der
Versuchsergebnisse (= Teilchenzahl pro Energieniveau)


% for - Schleife zur Generierung der Samples
for s=1:sampleSize
    % Variablen werden zurückgesetzt, da sie in der nächsten
Schleife gebraucht werden
    Esum = 0; % momentane Gesamtenergie des Systems
    nOfCurrentEnivs = nOfEnivs;
    currentNs = 0;
    Enivs = zeros(1,nOfEnivs);


    % while - Schleife zur Verteilung der Teilchen auf
Energieniveaus
    % Läuft, solange nicht die gesamte Energie vergeben und alle
Teilchen verteilt wurden
    while(Esum <= E && currentNs < n)


        index = randi([0 nOfCurrentEnivs-1]); % zufällig
zugeteiltes Energieniveau für ein Teilchen
        Esum = 0; % wird später benötigt


        for j=1:nOfEnivs
            Esum = Esum + (j-1)*Enivs(j); % Gesamtenergie des
Systems wird berechnet
```

```
        end


        % Überprüfung, ob genug Energie für das zufällige
Energieniveau
        % vorhanden ist
        if index <= (E - Esum)
            Enivs(index+1) = Enivs(index+1) + 1; % dem
entsprechenden Energieniveau wird ein Teilchen hinzugefügt
            currentNs = currentNs + 1; % Die Anzahl der
verteilten Teilchen wird erhöht
        else
            nOfCurrentEnivs = nOfCurrentEnivs - (E - Esum); % Ist
zu wenig Energie vorhanden, werden die zufällig generierten
Energieniveaus dementsprechend angepasst
        end
    end


    samples(1:nOfEnivs) = samples(1:nOfEnivs) +
Enivs(1:nOfEnivs); % Das Ergebnis der Teilchenverteilung wird
gespeichert
%     disp('ende'); % Textausgabe für debugging
end


erwartungsvektor = [12, 9, 6, 3]/10; %permutierter
Erwartungsvektor von Steve7
% plottet den tatsächlichen und vorhergesagten Erwartungsvektor
% stellt Linien und Punkte dar
plot(x, samples/sampleSize/n, '*-', x, erwartungsvektor/n, '*-')
axis([0 nOfEnivs-1 0 1]) % Achsenmaße, für jedes Energieniveau
einen Punkt auf x und 0 bis 1 auf y aufgrund des relativen
Anteils
```



Dieses Diagramm zeigt in grün den von Steve permutierten Erwartungsvektor. In blau wird die über 10000 Versuche ermittelte mittlere Verteilung der Teilchen für jedes Energieniveau gezeigt.

Brown'sche Bewegung

Die Brown'sche Bewegung beschreibt die ruckartige und unregelmäßige Bewegung von kleinsten Teilchen wie Staub in Wasser. Diese wird durch partielle Differentialgleichungen beschrieben. Die Berechnung der Brown'schen Bewegung war als Hinleitung auf unser finales Ziel, die Quantenphysik, konzipiert. Die Schrödingergleichung und die Bewegungsgleichung für die Brown'sche Bewegung basieren nämlich auf den gleichen mathematischen Prinzipien.

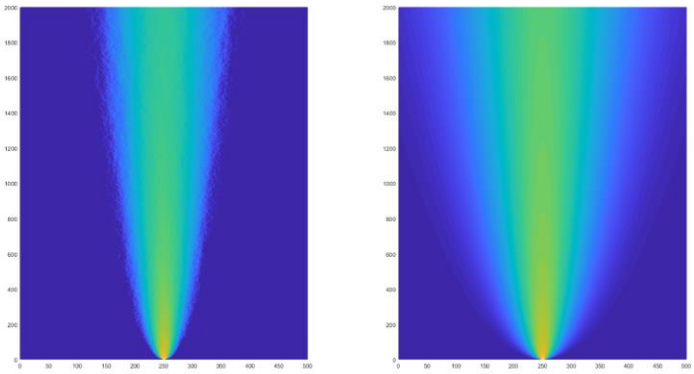
In unserem Programm wurde die Teilchenverteilung einerseits zufällig generiert und andererseits mithilfe einer Normalverteilung angenähert. Bei beiden Varianten wurde der Raum zwecks einfacherer Berechnung diskretisiert, also in `binN` Einheiten unterteilt.

```
n = 10000; % Teilchenzahl

binN = 100; % Anzahl der diskreten Einheiten, in die der Raum unterteilt
wurde

tvar = 4; % Dauer als vielfaches der breite

moveVar = 0.4; % Wahrscheinlichkeit der Bewegung nach rechts bzw. links
bins = zeros(1,binN);

bins = zeros(1,binN); % Der diskretisierte 1D - Raum wird erstellt

binsPrev = bins; %  Zur Generierung neuer Binwerte werden die alten
benötigt.

bins(round (binN/2)) = n;

tmax = binN*tvar; % Generierung der maximalen Zeit

distrOverTime = zeros(tmax, binN); % Hier wird der Ort jedes Teilchens
zu jedem Zeitabschnitt gespeichert

% for - Schleife zur Berechnung der binwerte für jede Zeiteinheit

for t=1:tmax


binsPrev = bins;  %  Zur Generierung neuer Binwerte werden die alten
benötigt.

    bins = zeros(1,binN); % die alten binwerte werden entfernt


    % for - Schleife zur Berechnung der einzelnen Teilchenzahlen in den
Bins

    for i=2:(binN-1)
```

```
        % Variablen, wie viele Teilchen in den rechten und linken Bin
        % gesetzt werden und wie viele verbleiben
        Lbin = 0;
        Cbin = 0;
        Rbin = 0;

        % for - Schleife, in der für jedes Teilchen entschieden wird, ob
und wie es sich bewegt
        for j=1:binsPrev(i)

            move = rand; % Zufallszahl zwischen 0 und 1

            % entscheidet anhand vpn move, in welchen bin das teilchen
            % versetzt wird
            % Die Wahrscheinlichkeit für öinks und rechts ist dabei
            % konstant, die für den Verbleib entspricht der
            % restwahrscheinlichkeit
            if move < moveVar
                Lbin = Lbin + 1;
            elseif move >= 1-moveVar
                Rbin = Rbin + 1;
            else
                Cbin = Cbin + 1;
            end

        end

        % j - te Bin wird entsprechend der zuvor generierten Verteilung
auf die Nachbarbins aufgeteilt
        bins(i - 1) = bins(i - 1) + Lbin;
        bins(i + 1) = bins(i + 1) + Rbin;
        bins(i) = bins(i) + Cbin;

    end
```

```
    distrOverTime(t, 1:binN) = bins; % Die Verteilung dieses
Zeitschritts wird gespeichert


end


gaussOverTime = zeros(tmax, binN); % Array für die Verteilung der
simulierenden Normalverteilung


% for - Schleife, die für jeden Zeitschritt die entsprechende

% Normalverteilung berechnet

for t=1:tmax

    gaussOverTime(t, 1:binN) = normpdf(1:binN, binN/2,
sqrt(t*2*moveVar));

end



c = log(distrOverTime); % Farbschema für distrOverTime, Logarithmus zur
besseren Sichtbarkeit


cGauss = (sqrt(sqrt(sqrt(gaussOverTime)))); % Farbschema für die
Normalverteilung, 8 - te Wurzel zur besseren Vergleichbarkeit mit den
empirischen Resultaten


subplotN = 3; % Anzahl der ausgegebenen Plots


subplot(1,subplotN,1) %  Erste Position von links

mesh(distrOverTime/n, c ) % Karte des relativen Anteils an den gesamten
Teilchen mit Farbe(c)

distrOverTime;


subplot(1,subplotN,2) % Zweite Position von links

mesh(gaussOverTime, cGauss) % Karte der theoretischen Normalverteilung
mit Farbe


subplot(1,subplotN,3) % Dritte Position von links

mesh(distrOverTime/n-gaussOverTime) % Karte der Differenz zwischen
Vorhersage und empirischen Resultaten


% % Debugging der Farbverteilung

% subplot(1, subplotN, 4)
```

```
% mesh(c-cGauss)
% (c-cGauss)
% max(max(c))
n - sum(bins) % "Verlust" = Teilchen, die hinaus diffundieren
```

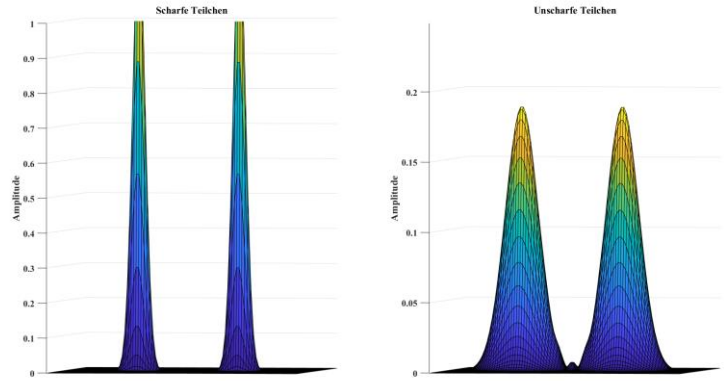


*

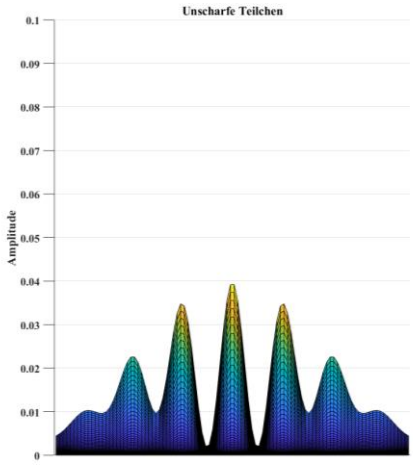
Links ist die simulierte Verteilung durch die zufällige Auswahl der Bewegungsrichtung der Teilchen sichtbar. Rechts ist die Normalverteilung mit μ als Zentrum der Normalverteilung und Sigma als Quadratwurzel aus 2*t*p, wobei t die Zeit und p die Wahrscheinlichkeit der einzelnen Teilchen, nach rechts oder links zu diffundieren, darstellt. Wie eine Überprüfung ergab, beträgt die Differenz zwischen Normalverteilung und simuliert empirischen Ergebnissen bei ausreichend großen Teilchenzahlen weniger als fünf Prozent. Somit ist diese von unserem Betreuer hergeleitete Formel eine sehr gute Annäherung an die tatsächlichen, wenn auch simulierten Ergebnisse.

Von Quanten- bis zur klassischen Materie

Wir erkannten, mithilfe von Steve Ähnlichkeiten zwischen dem Quantenmodell und der Brown'schen Bewegung. Unsere Gruppe hatte nämlich als letztes großes Ziel die Simulation des Vorgangs, wie aus der Quantenmechanik klassische Materie entsteht. Dies beinhaltet, wie die unscharfe Materie, also Teilchen ohne bestimmbaren Aufenthaltsort, "scharf" und dadurch zu klassischen Teilchen wird. Obwohl wir Probleme mit diesem komplexen Thema hatten, haben wir mit einem Programm von Steve eine laufende Simulation programmiert.



Hier sieht man den Unterschied zwischen der Interferenz zwischen simulierten Teilchen. Die Teilchen links beeinflussen sich nicht gegenseitig, da sie aufgrund ihrer hohen Masse wie klassische Materie funktionieren. Beim rechten Diagramm erkennt man, dass die beiden Teilchen aufgrund ihrer geringen Masse einerseits eine viel geringere Aufenthaltswahrscheinlichkeit an einem bestimmten Ort haben. Andererseits erkennt man bereits den Beginn eines Interferenzmusters an dem kleinen, mittigen Wahrscheinlichkeitshügel.



Schreitet die Zeit weiter fort, entwickelt sich das Interferenzmuster weiter. Das obige Bild entspricht in etwa dem Interferenzmuster eines Doppelspaltexperiments. Bei diesem werden kleine Teilchen wie Elektronen auf eine Wand mit zwei parallelen Spalten geschossen. Da sie aufgrund ihrer geringen Massen und der daraus resultierenden Quanteneffekte miteinander interferieren, treffen sie mit einer Wahrscheinlichkeit ähnlich der obigen auf einer hinter den Spalten liegende Wand auf.

Persönlicher Eindruck

Wir haben uns dazu entschieden, alle einen individuellen Eindruck zu schreiben, da wir alle Unterschiedliche Kompetenzen aus dieser Woche mitnehmen konnten.

- Sebastian

  Mir hat diese Woche einen Einblick in die Welt der Mathematik gezeigt. Obwohl wir in nur einen kleinen Bereich geschnuppert haben, gab es sehr viel zu entdecken und erforschen. Dazu hat mir diese Woche zum ersten Mal die Chance gegeben, mich selber für ein Thema einzusetzen und mit einem Team an motivierten Gleichaltrigen daran zu forschen. Natürlich war nicht jeder gleich erfolgreich mit den unterschiedlichen Problemstellungen. Aber trotzdem hat unser Leistungsdurchschnitt den Erwartungswert überboten. Allerdings hat diese Woche nicht nur mein Wissen beeinflusst. Ich bekam auch die Chance andere Mathematikinteressierte in meinem Alter kennen zu lernen und habe auch viele neue Freundschaften mit diesen geschlossen.

- Florian

  Trotz meines Stresses mit der VWA entschied ich mich, auch dieses Jahr an der Modellierungswoche teilzunehmen. Die Erfahrung, mit anderen Interessierten an, von der Schule unbekannten, Problemstellungen zu arbeiten, fand ich sehr interessant und anregend. Ich wählte diese Gruppenarbeit, da ich sowohl die philosophischen Überlegungen als auch die dahinterstehende Mathematik sehr interessant fand. Und ich wurde nicht enttäuscht. Unser Betreuer, Mr. Modellierungswoche, ging auch auf unsere Interessen individuell ein und bemühte sich inständig um ein tieferes Verständnis der hinter der Quantenphysik stehenden Mathematik, was mir persönlich sehr weiterhalf. Alles in allem war es eine sehr wertvolle Erfahrung, die mir zu tieferen, mathematisch unterstützter Erkenntnis in sehr vielen verschiedenen Bereichen.

- Valentin

  Diese Woche hat mir persönlich sehr weiter geholfen. Durch den großen Niveauunterschied zu herkömmlichen Schulproblemen habe ich gelernt

beim Lösen von Problemstellungen, die mir auf den ersten Blick zu schwer scheinen nicht aufzugeben, sondern zu versuchen diese mit meinen begrenzten Möglichkeiten zu lösen. Ursprünglich hat mich außerdem die Philosophie an diesem Thema sehr interessiert und ich und demenentsprechend philosophiert. Letztendlich lässt sich sagen, dass die Woche nicht nur mathematisch ein Erfolg war, da man das Herantasten an schweren Problemstellungen eine Kompetenz ist, die im späteren Leben überall gebraucht wird.

- Mike

  Diese Woche hat mein Wissen, meine Ansichten und sogar meine Persönlichkeit beeinflusst. Allein in den ersten 3 Tagen bekam ich einen sehr genauen Einblick in die Wahrscheinlichkeitstheorie. Das ist viel schneller als es in der Schule überhaupt möglich wäre. Die Arbeiten mit Wahrscheinlichkeiten bauten für mich ein gutes Fundament, um schlussendlich die komplexe, höhere Mathematik hinter der Quantenmechanik zu verstehen. Die Analogie dieser Arbeiten machten das Lernen und vor allem verstehen dieser Aufgaben zu einem regelrechten Klacks. Aufgrund des immer steigenden Niveaus wurde mir auch nie langweilig, und immer gefordert. Dieses Projekt hat auch meine Programmierkünste verbessert und mein Interesse für Schach gestärkt.

- Nik

  Hilfe! Mein Gehirn macht nicht das, was es soll. Letztes Jahr habe ich noch behauptet, dass die Modellierungswoche für die Ober-Nerds sei und dieses Jahr sitze ich selber hier. Ich habe mich dafür entschieden, weil mein Interesse für angewandte Mathematik im letzten Jahr unglaublich gestiegen ist. Diese Woche hat mir nicht nur geholfen, die Mathematik der Wahrscheinlichkeit tiefer und besser zu verstehen, sondern auch simple Sachen, die ich aus unverständlichen Gründen wieder vergessen habe, wie zum Beispiel Expotentialfunktionen nochmal zu wiederholen und dahinterzukommen. Allerdings haben wir zum Großteil auch programmiert und da ich auf ein Gymnasium mit dem Kurssystem gehe, habe ich, nicht so wie manch andere aus meiner Gruppe, nicht programmieren können. Das Programm Matlab und alle anderen

Programme mit der ,,Programmiersprache" ergeben mittlerweile Sinn in meinem Kopf.

- Johannes

Während man in der Modellierungswoche kein "sicheres" Verständnis für das Gelernte hat, anders als für mich in der Schule, sondern eine Ahnung, wie ein Teil mit dem anderen zusammenhängen könnte, ist es trotzdem so, dass man sehr viele Eindrücke gewinnt und man sich an das Programmieren und an die wissenschaftliche Denkweise gewöhnt. So habe ich bemerkt, dass mir das Arbeiten mit Matlab leichter gefallen ist als bei der letztjährigen Modellierungswoche.