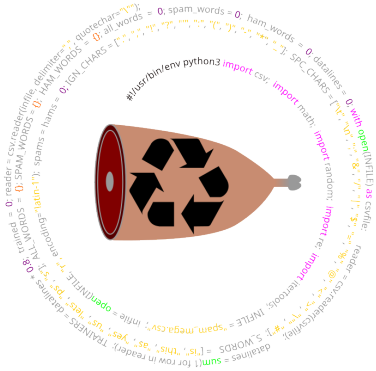
Steirische
Modellierungswoche

18. Woche der Modellierung

mit Mathematik



Dokumentationsbroschüre 6.1.-12.1.2024

Woche der Modellierung mit Mathematik

JUFA Leibnitz, 6.1.-12.1.2024

Weitere Informationen:

https://imsc.uni-graz.at/modellwoche/2024/

Sponsoren und Organisatoren

Koordination

Mag. DDr. Patrick-Michel Frühmann

Mag. Silvia Lebosi

Alexander Sekkas

Vorwort

Mathematik hat eine einzigartige Doppelrolle als Jahrtausende alte Kulturleistung und als Schlüsseltechnologie für die modernsten Entwicklungen der heutigen Zeit. Darüberhinaus kommt ihre Entwicklung ohne komplexe Apparate und nur mit Papier und Bleistift aus. Trotzdem empfinden die meisten Menschen heutzutage Mathematik als etwas unzugängliches und für ihr Leben wenig relevantes; dies beginnt bereits in der Schule, wo trotz engagierter Lehrkräfte die Zeit fehlt, die Vielfalt, Schönheit, und Nützlichkeit dieser Disziplin begreifbar zu machen.

Daher soll als Ergänzung ses Schulunterrichts die Woche der Modellierung mit Mathematik Schüler:innen die Möglichkeit geben, Mathematik als lebendiges Fach zu erfahren: Die jungen Leute arbeiten und forschen in kleinen Gruppen mit Wissenschafter:innen an realen Problemen aus den verschiedensten Bereichen und versuchen, mit Hilfe mathematischer Modelle neue Erkenntnisse zu gewinnen. Sie arbeiten freiwillig ohne Leistungsdruck, dafür mit Eifer und Enthusiasmus, rechnen, diskutieren, recherchieren -- oft auch noch am späten Abend -- in einer entspannten und kreativen Umgebung, die den Schüler:innen und betreuenden Wissenschafter:innen gleichermaßen Spaß macht. Die Projektbetreuer konnten auch in diesem Jahr wieder erleben, wie eigenes Entdecken und Selbstmotivation das Verhalten der Schüler:innen während der ganzen Modellierungswoche bestimmen. Sie lernen dadurch eine Arbeitsmethode kennen, die dem tatsächlichen Forschungsleben näher kommt, als das im Rahmen des Schulunterrichts möglich wäre.

Ähnliche 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 -- trotz wohlverdienten Ruhestand -- zum 17. 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. Leider verstarb Prof. Kappel Anfang 2020, so dass er nicht erleben konnte, wie die Woche der Modellierung mit Mathematik nach der Pandemie-bedingten Pause wieder mit großem Zuspruch starten konnte.

Wir danken der Bildungsdirektion für Steiermark, und hier insbesondere Frau FIin Maga. 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 Vizerektorin Univ.-Prof. Dr. Catherine Walter-Laager und Dekan Univ.-Prof. Dipl.-Ing. Dr.techn. Klemens Fellner, vom Fachdidaktikzentrums

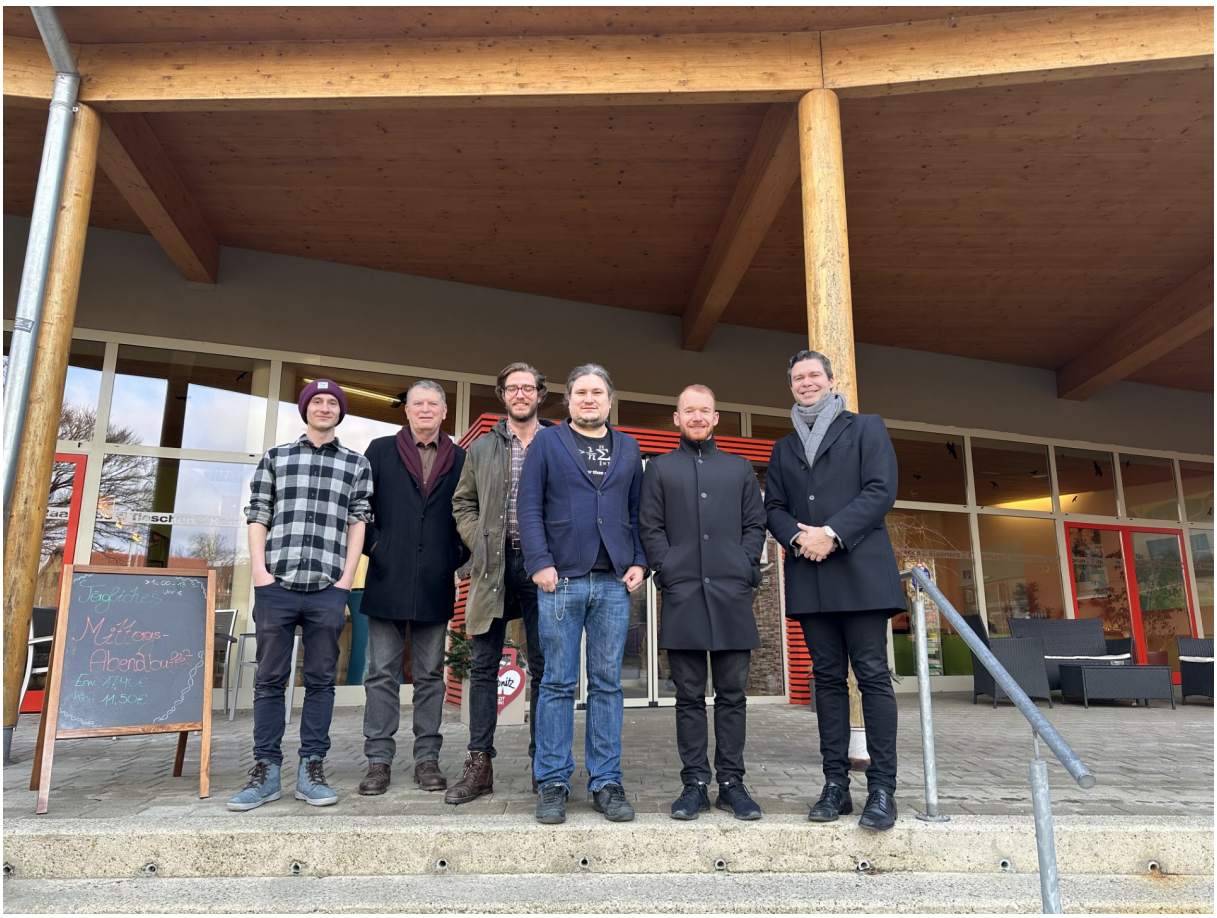
für Mathematik und Geometrie der Uni Graz, Leiterin Ass.-Prof. Dr. Christina Krause, vom Forschungsmanagement der Uni Graz, und vom Forschungszentrum "Virtual Vehicle".

Ohne den idealistischen, unentgeltlichen und engagierten Einsatz der direkten Projektbetreuer, Dr. Stefan Reiterer, Matthias Höfler, Gabriel Pichlbauer und Dr. Michael Fischer hätte diese Modellierungswoche nicht stattfinden können.

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

Leibnitz, 18. Februar 2024

Univ.-Prof. Dr. Christian Clason
Institut für Mathematik und Wissenschaftliches Rechnen
Karl-Franzens-Universität Graz



FLÜSSE & NETZWERKE

Betreuer: Stefan Reiterer Teilnehmer: Valentin Löw, Tim Leipold, Simon Reindl, Julia Reuter

Contents

Flüsse

Was sind Flüsse?

Definition: Eine Flussgröße ist die Ableitung einer Erhaltungsgröße.

Definition: Erhaltungsgrößen bleiben in Summe immer konstant. (z.B. Energien, Ladung , Masse, Impuls, Drehimpuls)

Proposition:

Seien $q_1, \dots q_n$ zeitabhängige Erhaltungsgrössen und es seien $\frac{dq_i}{dt} = f_i$ unsere Flussgrößen, dann gilt $f_1 + \dots + f_n = 0$.

Beweis:

Es gilt da $q_1, \dots q_n$Erhaltungsgrössen sind $q_1 + \dots q_n = const.$
Daher folgt durch Ableiten nach der Zeit

$$0 = \frac{d}{dt} const = \frac{d}{dt}(q_1 + \dots q_n) = \frac{dq_1}{dt} + \dots \frac{dq_n}{dt} = f_1 + \dots + f_n.$$

Auf dieser Grundidee basieren unsere weiteren Beobachtungen. Durch diese Flussbedingung erhalten wir eine zusätzliche Gleichung um Modelle durch verschiedene Interfaces zu verbinden. Diese werden in der Sprache Modelica so definiert (Beispiel eines elektrischen Pins)

Flussgrösse

```
1   connector PinMinus "Negative pin of an electric component"
2     Modelica.Units.SI.Voltage v "Potential at the pin";
3     flow Modelica.Units.SI.Current i "Current flowing into the pin";
4 >   annotation ( ...);
30  end PinMinus;
31
```

In diesem Fall ist die Flußgröße der Strom der die Ableitung der Ladung ist. Wir haben einige Flussgrößen kennengelernt (Kraft, Strom, Volumstrom, „Biomasse“, Moment, Einheiten ...)

ROTATIONAL MODEL

Rotational Modell erstellen mit: Trägheit (Inertia), Feder (Spring), Dämpfer (Damper) und Mechanischen Boden (mechanical ground).

Two Flange:

Für die zwei Flanges wird ein **partial model** erstellt.

```
partial model TwoFlange
 "Definition of a partial rotational component with two flanges"
```

Unter equation werden die Formel definiert:

```
phi_rel = flange_a.phi-flange_b.phi;
```

Der Relative Drehwinkel gibt an, wie groß der Unterschied zwischen den Drehwinkeln der Flanges ist.

Inertia:

Danach ein **model** namens Inertia erstellen. Als Parameter für die Trägheit setzt man die SI-Einheit J.

```
model Inertia "Rotational inertia without inheritance"
  parameter Modelica.Units.SI.Inertia J;
```

Nachdem das Inertia model auf einem Modell mit zwei Flanges basiert, braucht man hier die Definition der Flanges (mithilfe von `extends TwoFlange;`).

Man fügt zwei weitere Größen hinzu, die Winkelgeschwindigkeit w, und den Drehwinkel phi und das Drehmoment tau, die unter equation definiert werden:

```
phi = flange_a.phi;
  w = der(flange_a.phi) "velocity of inertia";
  phi_rel = 0 "inertia is rigid";
  J*der(w) = flange_a.tau + flange_b.tau
    "Conservation of angular momentum with storage";
```

Die Formel: `J*der(w) = flange_a.tau + flange_b.tau` drückt im Grunde die Tatsache aus, dass die Zunahme des in der Trägheit gespeicherten Impulses gleich der Summe der auf die Trägheit ausgeübten Drehmomente ist.

Compliant:

Um nicht bei jedem Bauteil angeben zu müssen, dass das Drehmoment Konstant bleibt, wird ein partial model erstellt.

```
equation
  tau = flange_a.tau;
  flange_a.tau + flange_b.tau = 0
    "Conservation of angular momentum (no storage)";
```

Spring:

Jetzt werden die Federn erstellt. Die Federkonstante c, wird eingeführt und das Hooksche Gesetz unter equation beschrieben:

```
model Spring1 "A rotational spring component"
  parameter Modelica.Units.SI.RotationalSpringConstant c;
  extends Compliant;
equation
  tau = c*phi_rel "Hooke's Law";
end Spring1;
```

Damper:

Anschließend wird der Damper erstellt. Die rotatorische Dämpfkonstante d wird eingeführt, die in folgender Formel definiert wird:

```
tau = d*der(phi_rel) "Damping relationship";
```

Mechanical Ground:

Jetzt muss nurmehr der mechanical ground. Da sich der Boden nicht bewegen soll, wird phi gleich null gesetzt.

```
flange_a.phi = 0;
```

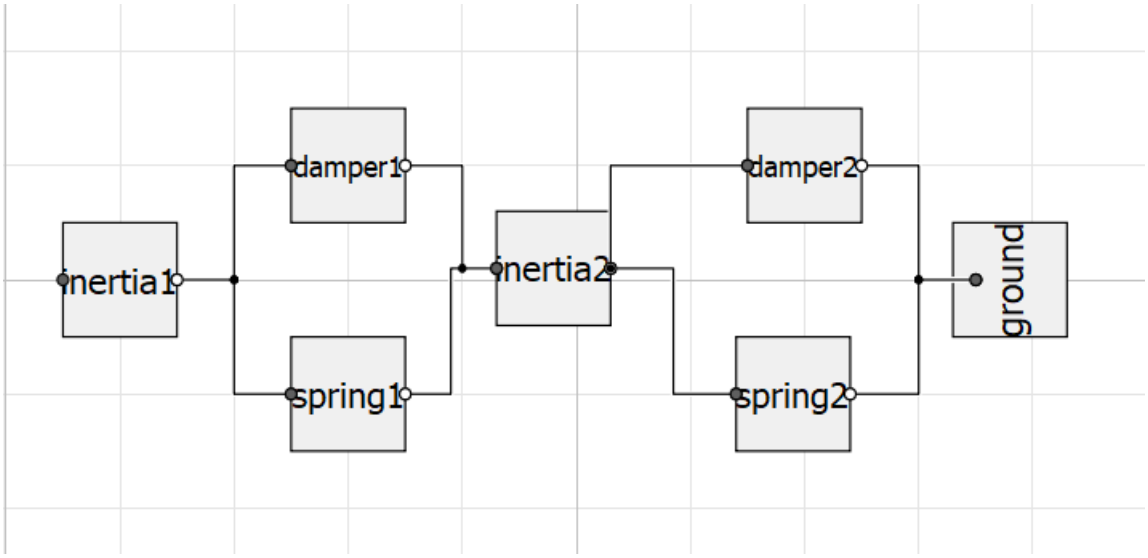
Rotational System:

Schließlich müssen nurmehr alle Bauteile miteinander verbunden werden. Die einzelnen Komponenten müssen angegeben werden (Deklaration der Variablen).

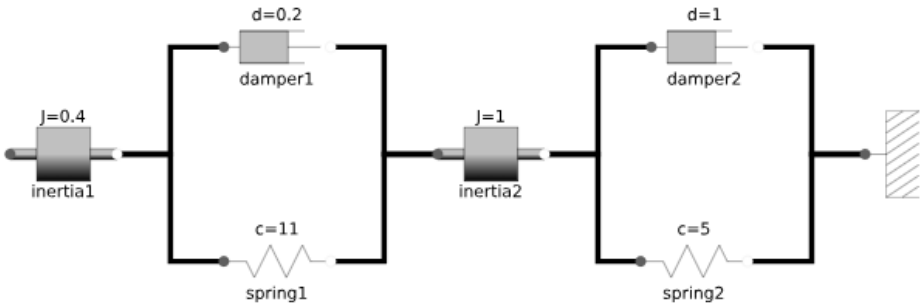
```
Components.Spring spring2(c=5)
```

Zusätzlich wird für die Federkonstante c=5 definiert.

Mit dem Befehl `connect` verbindet man die Teile miteinander.



The diagram for this model, when rendered, looks like this:

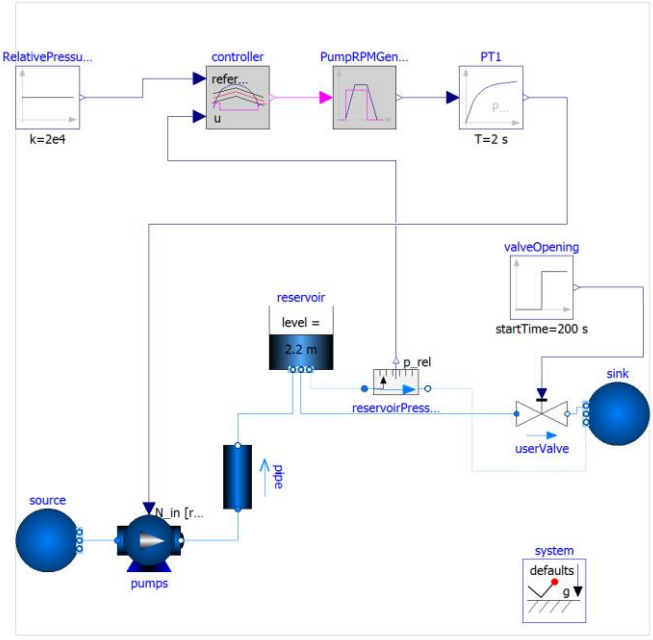


Die Flussgröße in diesem Modell ist der Impuls und die Flanges fungieren als connector.

Pumpen System

Flussmodelle

Um was geht es?



Gezeigt wird ein Flusssystem in Form von Wasser dabei ist die Wassermenge die Erhaltungsgröße.

Es geht in diesem Modell (siehe Aufbau rechts)

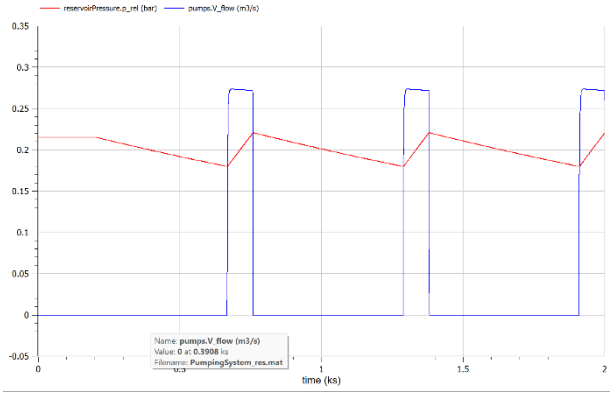
um eine Modellierung, wie Wasser aus einer Quelle bis zu einem Waschbecken „Sink“ geleitet wird.

Dabei ist ein Reservoir dazwischen geschalten um zu garantieren, dass Wasser immer abrufbar ist. Es wird zugleich der Druck im Reservoir gemessen, um der Pumpe sagen zu können, wann sie wieder zum Pumpen anfangen, soll um neues Wasser nachzuholen. Und zu guter Letzt gibt es vor dem Waschbecken eine Steuerbare Öffnung, die hier nach 200s Wasser hindurch lassen soll, wie der Wasserhahn.

Bauteile

- Quelle
- Pumpe

- Rohre
- Reservoir und Drucksensor
- Ventil
- Waschbecken
  Die Bauteile sind in dieser Reihenfolge verbunden worden



In diesem Graph erkennt man nochmal die Wirkungsart des Drucksensors im Reservoir und der Korrelation mit der Pumpe.

Blau = Gepumptes Wasser pro Zeit (m3/s
Rot = Druck im Reservoir (bar)

Stromkreis

Zusammenhang zwischen Ladung und Stromstärke:

Im Stromkreis bleibt die Ladung immer konstant da alle eingehende Kräfte gleich alle ausgehenden Kräfte sind (Laut Kirchhoff´schen gesetz). In diesem Fall der Zusammenhang.
Stromstärke ist die Ableitung von der Ladung nach der Zeit.

zwischen Stromstärke und der Ladung: $\text{Stromstärke} = \frac{\text{Ladung}}{\text{Messzeit}} \rightarrow I = \frac{\Delta Q}{\Delta t}$

Im Stromkreis wirken folgende Komponente:

Im partial Model TwoPin werden PinMinus und PinPlus zusammengeführt und es werden zwei variablen i und v (Stromstärke und Stromspannung) zwischen beiden Pins definiert.

Capacitor 1&2, Inductor und Resistor wurden anhand des DRY Prinzips erstellt (Don´t repeat yourself) und dafür benötigen wir ein partial Model.
Beim Resistor wird dadurch das er ins TwoPin Model extended wurde, die Variablen i und v implementiert. Der Parameter für die Resistance wurde ebenfalls erstellt, was darin resultiert das man nun das Ohm`sches Gesetz verwenden kann.
Beim Inductor und Capacitor wurde ebenfalls ein neuer Parameter erstellt und durch das extenden in das TwoPin Model ist es möglich beim Inductor das Induktionsgesetz einzubauen und beim Capacitor das Kapazitätsgesetz.

Im StepVoltage Model werden noch weiter Parameter erstellt wie V0 (Stromspannung zum Zeitpunkt 0) Vf (Stromspannung zum Zeitpunkt f) und stepTime (eine Zeit konstante), außerdem wird das stepTime Model in das TwoPin Model extendet um auf die Variablen in diesem zuzugreifen.

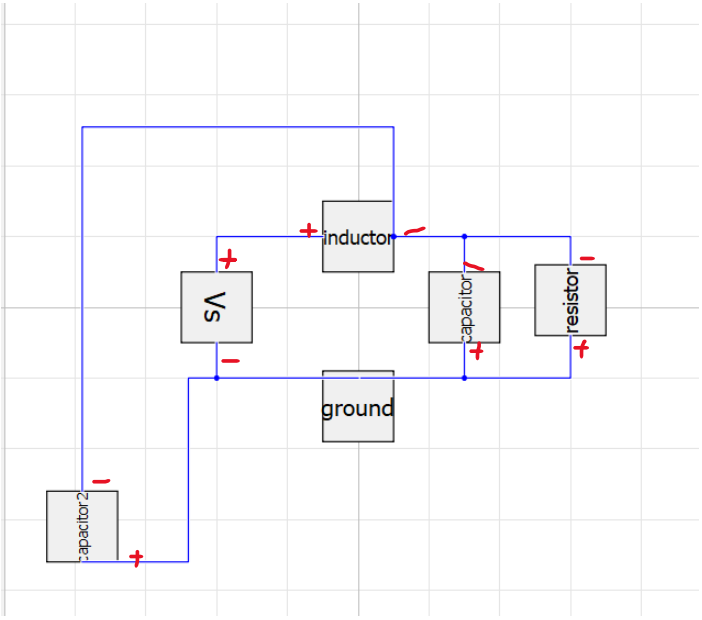
Der Ground ist die Erdung welche überflüssige Spannung ableitet. Im Model wird ein Positiver Pin erstellt, wobei jede Spannung, die in diesen geleitet wird, gleich 0 wird.

Zusammengeführt wird alles im Model „MyCircuit“. In diesem werden auch alle benötigten Werte für die Parameter bestimmt, welche wir zuvor nur erstellt haben. Es wird wie folgt verbunden:

n= Negativ, p=Positiv, Vs=StepVolage

inductor.n ➜ resistor.n
capacitor.n ➜ inductor.n
inductor.p ➜ Vs.p
capacitor.p ➜ ground
resistor.p ➜ ground
Vs.n ➜ ground
capacitor2.n ➜ inductor.n
capacitor2.p ➜ ground

Das ist der fertige Strom Kreis.



Lokta Volterra Equations

Species Connector

Erstellt Variablen Population und Fluss zur Interaktion zwischen Modellen. Die Flussgröße ist die „rate“ die man sich intuitiv wie die Biomasse vorstellen kann (Tiere fließen nicht …)

Regional Population Model

Setzt Startpopulation einer Spezies und definiert den Fluss als Ableitung nach der Population

Partielle Modelle

Definieren Variablen Wachstum/Abnahme als Änderung der Flussvariable zur Weiterverwendung

Reproduktion

Definiert Variable Alpha als Wachstumskonstante proportionell zur Population

Wachstum = Alpha*Population

Verhungern

Definiert die Geschwindigkeit Gamma mit der eine Spezies ohne Nahrung Ausstirbt auch proportionell zur Population

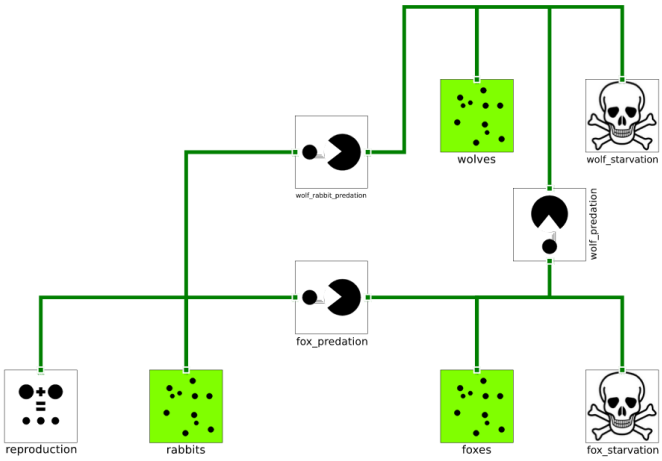
Abnahme = Gamma*Population

Prädation

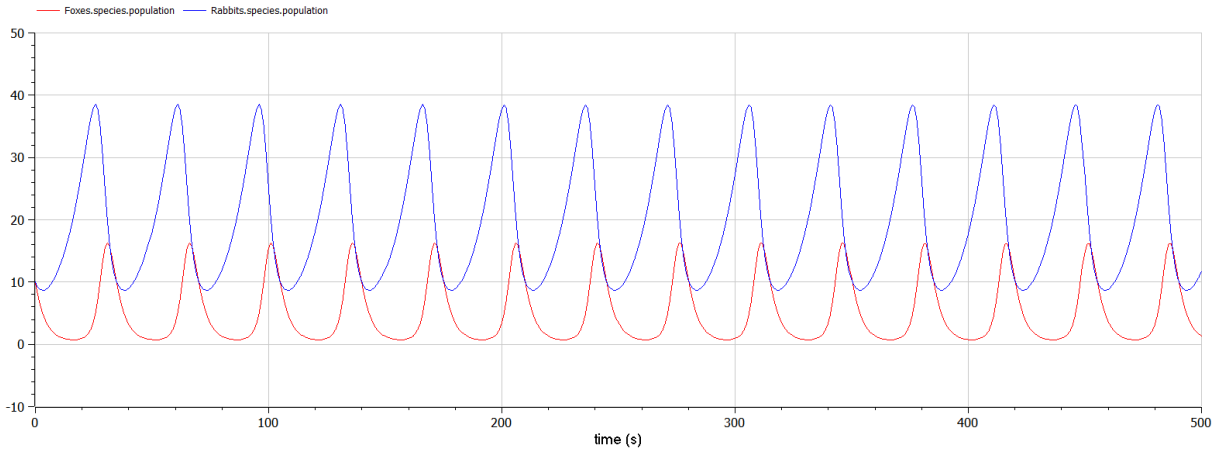
Definiert wie viele Beutetiere ein Raubtier frisst als Beta und welcher eigene Populationszuwachs dabei entsteht als Delta

Raubtierwachstum = Delta*Population-Beutetier*Population-Raubtier
Abnahme-Beutetier = Beta*Population-Beutetier*Population-Raubtier

Hier das Schema mit 3 Spezies (rabbits, foxes wolves) aus dem „Modelica by Example“ Tutorial (https://mbe.modelica.university/components/components/population/ retrieved 11.01.2024)



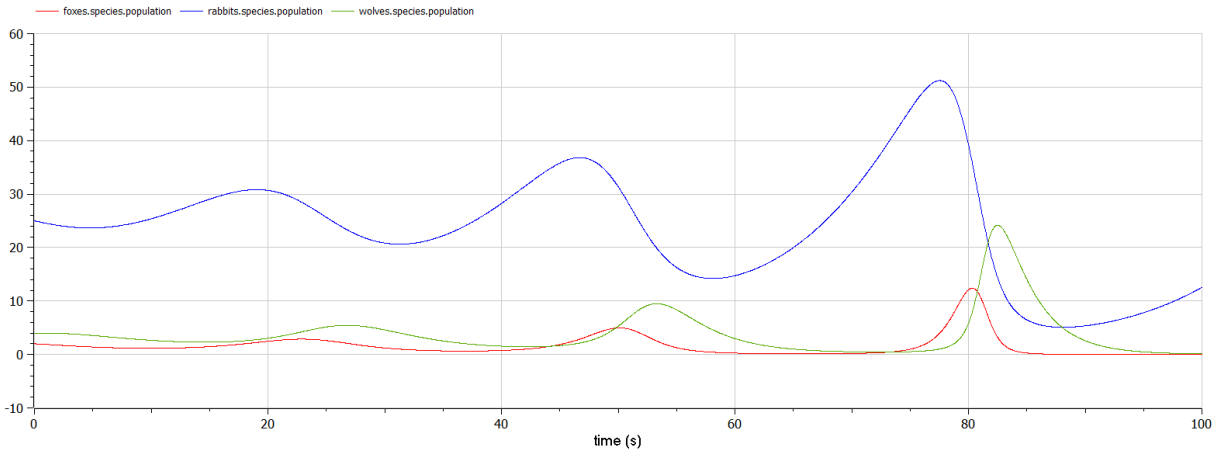
In Folgendem Graphen werden die Interaktionen zweier Spezies über einen gewissen Zeitraum dargestellt. Die Fuchspopulation verringert sich mit Abnahme der Futterspezies, weniger Raubtiere führen wiederum zu mehr Beutetieren. Die mittlere Populationsrate bleibt gleich.



Parameter:

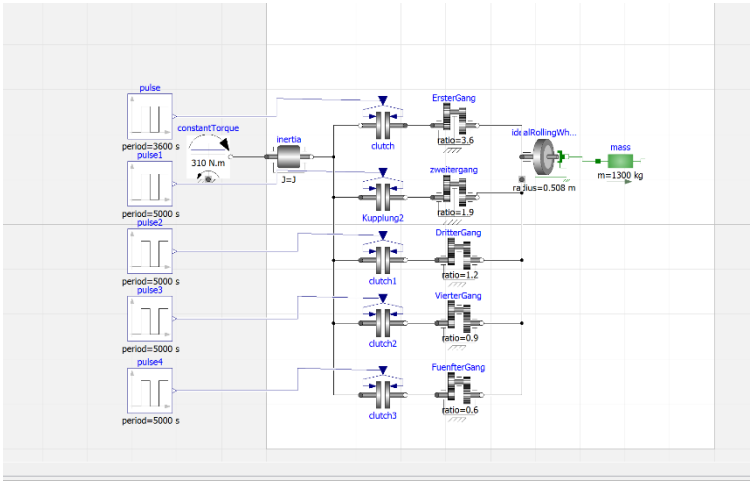
Füchse Ausgangspopulation: 10
Hasen Ausgangspopulation: 10
Hasen Reproduktionsalpha: 0.1
Füchse Sterberate Gamma: 0.4
Füchse Prädation Beta: 0.02
Füchse Prädation Delta: 0.02

Nach Integrierung einer weiteren Spezies in das modellierte Ökosystem, sind die Auswirkungen am Graphen ablesbar. Bei Einstellungen unterschiedlicher Parameter, sind unterschiedliche Ergebnisse zu erreichen. Teils endet die Simulation mit dem Aussterben einer Spezies, teils bleibt es stabil.



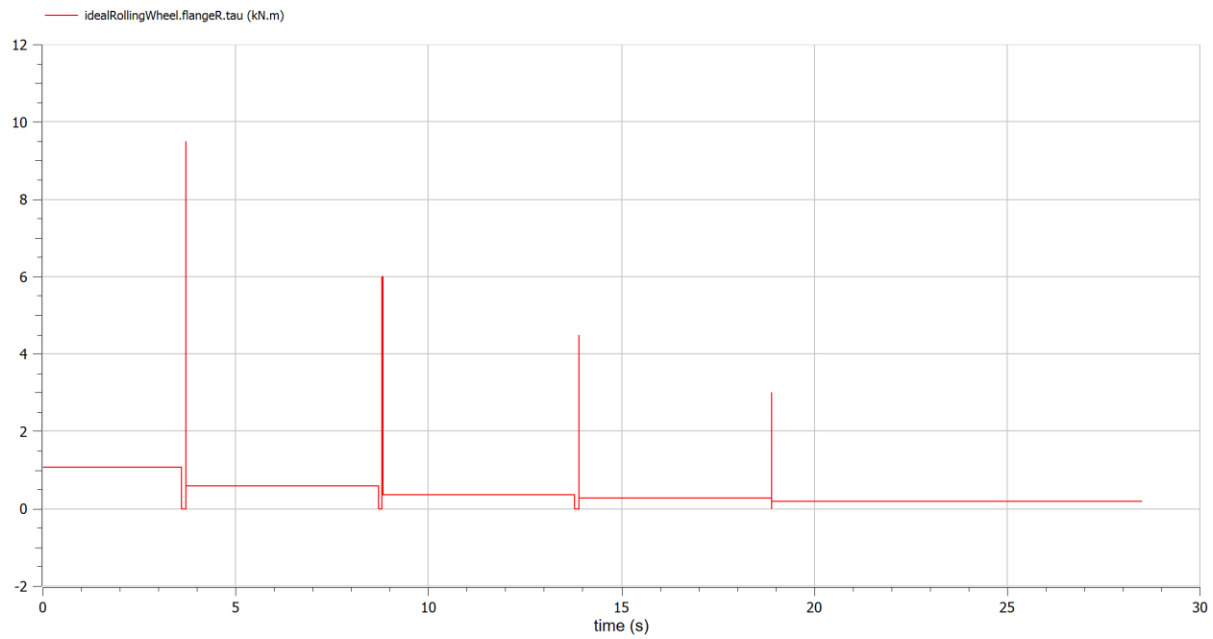
5 Gang Getriebe in Open-Modelica

In diesem Open Modelica-Modell wurde erfolgreich die Simulation eines Fünf-Gang-Getriebes durchgeführt. Das Modell besteht aus einer Drehmomentsquelle, die jeden Gang des Getriebes antreibt. Die Steuerung der Gänge erfolgt durch Kupplungen, die es ermöglichen, den gewünschten Gang zu wählen. Dabei wurde dieselbe Drehmomentsquelle für jeden Gang verwendet, um realitätsnahe Ergebnisse zu erzielen.



Die Simulation wurde auf die Beschleunigung einer Masse von 1.3 Tonnen ausgerichtet, repräsentierend ein Auto. Die graphische Darstellung zeigt, wie sich die Beschleunigung des Fahrzeugs über die Zeit entwickelt. Durch die Anpassung der Gänge und die gezielte Steuerung der Kupplungen konnte die Leistungsfähigkeit des Getriebes in Bezug auf die Beschleunigung der Fahrzeugmasse analysiert werden.

Die Ergebnisse dieser Simulation bieten Einblicke in die Leistungsfähigkeit des Fünf-Gang-Getriebes unter verschiedenen Bedingungen und ermöglichen es, die Wechselwirkungen zwischen den Gängen und der Fahrzeugdynamik zu verstehen.



Hier oben sieht man die einzelnen Schaltvorgänge und ihr Drehmoment in kN.m.

Die Abstufung ist: 1. = 3.6

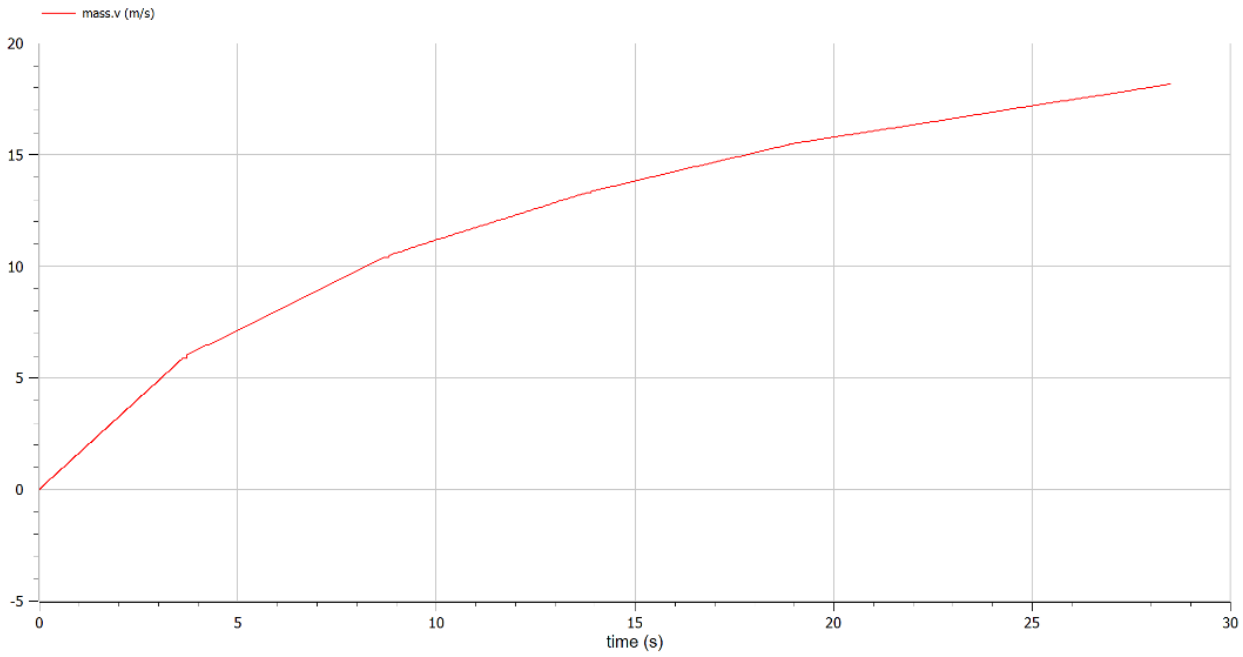
2. = 1.9

3. = 1.2

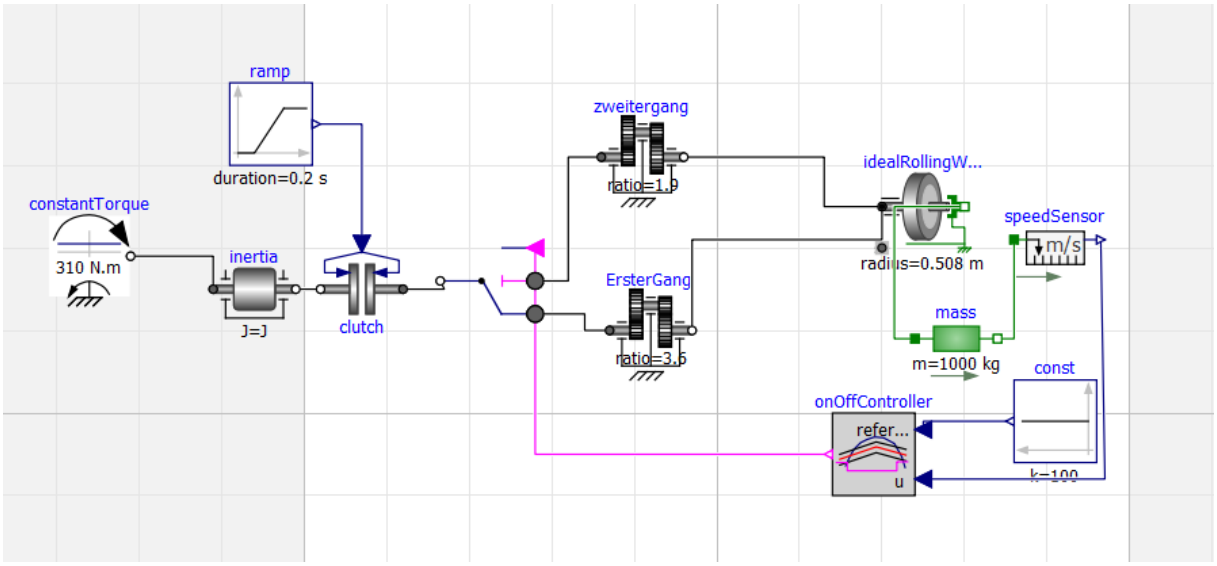
4. = 0.9

5. =0.6

Hier unten sieht man dann die Geschwindigkeit des Fahrzeugs. Das Fahrzeug hat eine Masse von 1.3 Tonne und wird von einem Motor angetrieben der konstant 310 N.m ausgiebt

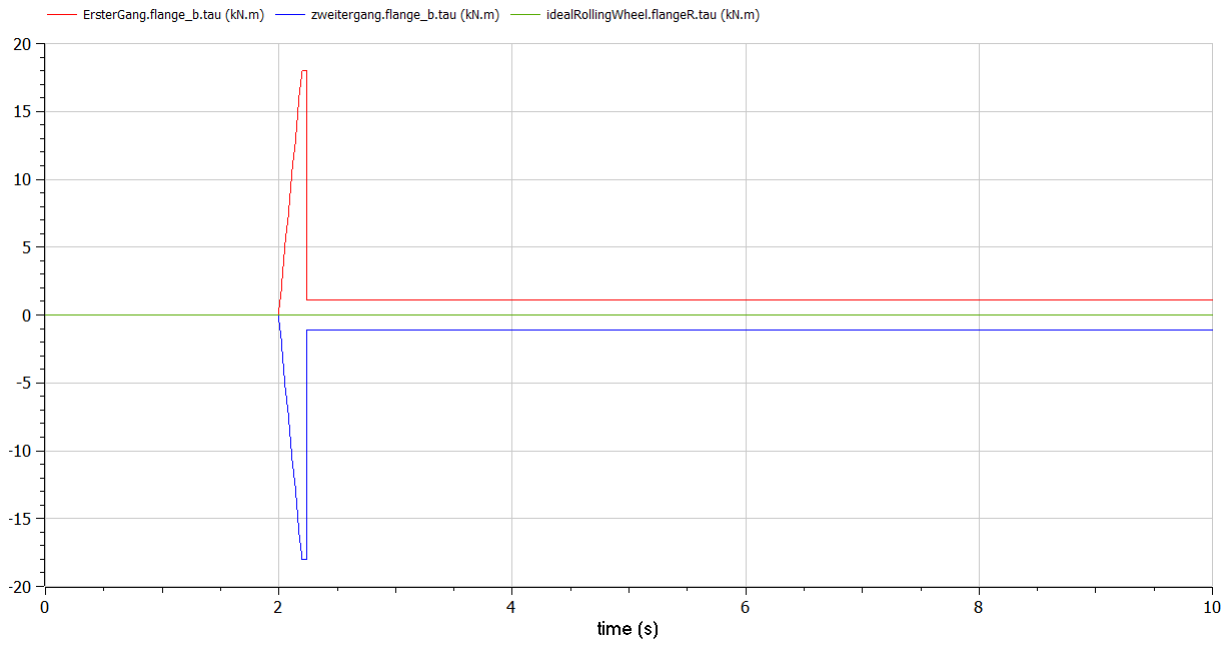


Im unteren Modell sieht man den Versuch das 5-Gang-Getriebe zu automatisieren. Dabei haben wir einen Speed-Sensor an die Masse angeschlossen, welcher das Signal an einen sogenannten OnOffController weitergibt. Zusätzlich wurde eine Konstante an den OnOffController angeschlossen damit dieser weiß ab welcher Geschwindigkeit das Signal weitergereicht werden muss. Dieses wird dann an einen selbstgebauten Switch geleitet, welcher zwischen ersten und zweiten Gang wechseln sollte. Das Problem ist, das der Switch aus einem unbekannten Grund nicht zwischen den Gängen wechselt und auch keine Kraft (Drehmoment) weitergibt.



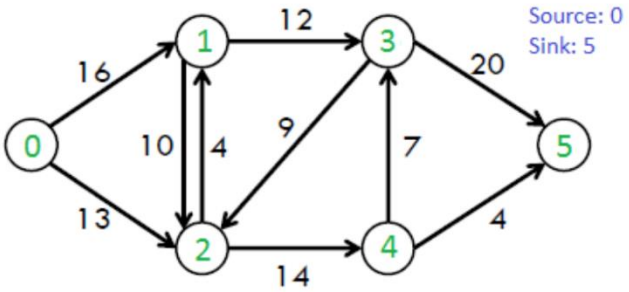
In der Simulation sieht man wie die Kupplung schließt und das Drehmoment auf die Gänge übertragen wird.

Durch den Fehler im Selbstgebauten Switch wird das Drehmoment auf beide Gänge übertragen. Der Output der Gänge ist allerdings fehlerhaft da beide Gänge gleichzeitig Drehmoment ausgeben aber dieses nicht auf das Rad (idealRollingWheel) übertragen. getrieben mit konstanten 310 N.m



Maximum Flow Rate

Ein Netzwerk besteht aus Knoten (nodes) und Kanten (edges), wobei die edges immer jeweils zwei nodes verbinden und gewichtet sind. In jedem dieser Netzwerke gibt es einen Startpunkt (Source) und einen Endpunkt (Sink), wo der flow initiiert wird und ankommt. So ein Netzwerk kann zum Beispiel so aussehen:



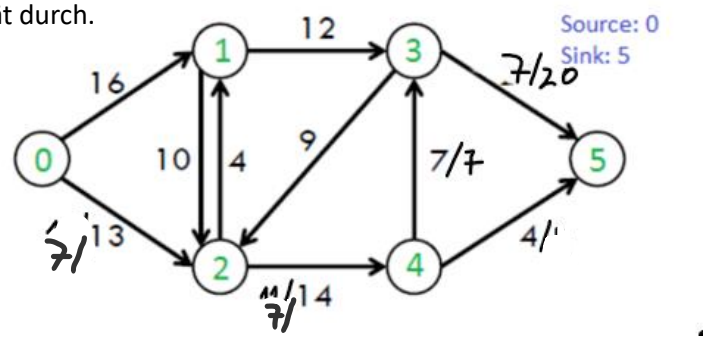
Ein klassisches Beispiel für ein Netzwerk wäre unter anderem das Abwassersystem. In dem Beispiel wären die nodes die Häuser, source die Wasseraufbereitung, sink die Kläranlage. Diese Knoten werden durch edges, also die Rohrleitungen, verbunden. Die Gewichtung stellt zum Beispiel den Durchmesser der Rohre da.

Um jetzt die maximale flowrate dieses Netzwerkes zu berechnen, gibt es mehrere Ansätze, die bei einfacheren Systemen noch mit Hand durchgeführt werden können, bei komplexeren aber mithilfe von Algorithmen berechnet werden müssen.

<u>Für obiges Netzwerk war der erste analoge Lösungsansatz folgender:</u>

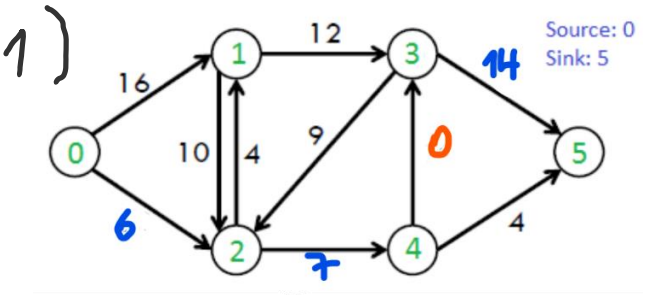
Beispiel 1)

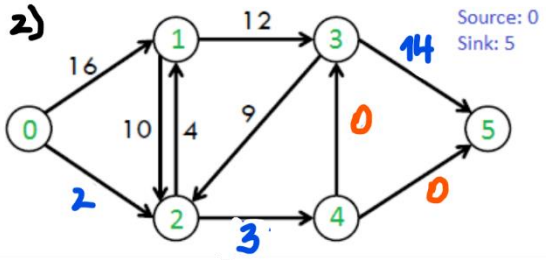
Zuerst sucht man den sogenannten bottleneck, also den Engpass. Bei diesem Ansatz ist das der node, der am wenigsten Weiterleitungskapazität hat. Danach sucht man sich einen beliebigen path, der die source mit dem sink verbindet und schickt einen flow, mit dem Wert der geringsten Weiterleitungskapazität durch.

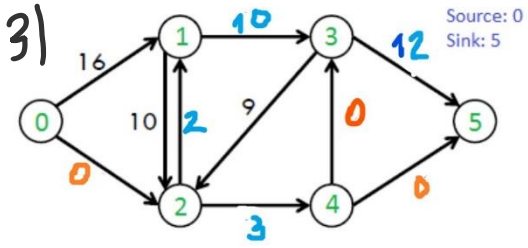


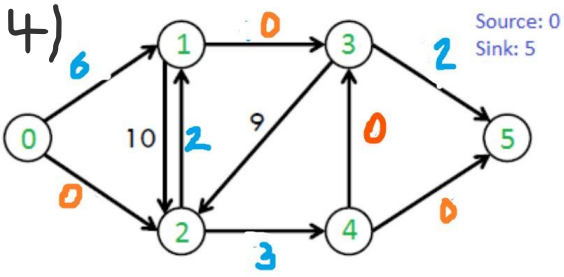
Nun errechnet man die Differenzen, indem man die flowrate von den Kapazitäten subtrahiert. Jetzt muss man wieder einen neuen path finden.

Dies führt man weiter fort, bis man irgendwann einmal keine paths mehr finden kann.





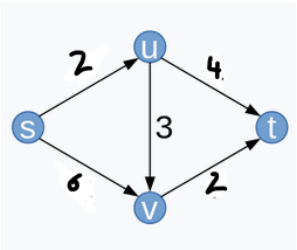




Jetzt muss man nurmehr alle flows die man durchgeschickt hat addieren: 7+4+2+10 =23
Und somit hat man die maximum flowrate gefunden. Dieses Konzept ist für viele flownetwork einsetzbar, jedoch gibt es auch Netzwerke, die nicht so gelöst werden können.

Beispiel 2)

Man nehme dieses Netzwerk als Exempel:

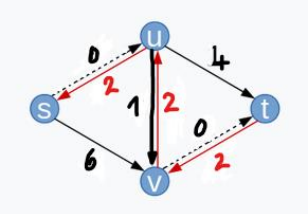


Würde man den Pfad *s-u-v-t* abgehen. Wäre die maximale flowrate mit dem vorherigen Lösungsweg 2. Die maximale Flussrate ist jedoch 4. Bei solchen kleinen Netzwerken kann man schnell erkennen, wenn man einen ineffektiveren Pfad eingeschlagen hat, bei größeren ist das schwer. Deshalb funktioniert die vorherige Methode nur bedingt. Man sieht, dass man nicht jeden beliebigen Pfad wählen kann. Um den echten maximalen flow auch immer beschreiben zu können, muss man den Rückfluss innerhalb von bereits befüllten edges Betracht ziehen.

Solche Networks können trotzdem auch noch mit der Hand ausgerechnet werden, jedoch gestaltet sich dieses Procedere als mühselig. Im Folgenden dennoch eine Anleitung:

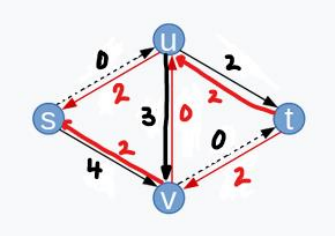
1. Man wählt einen beliebigen Pfad, beispielsweise den Pfad *s-u-v-t.* Wie im Beispiel 1 legt man den flow mit der Rate c =2, über den Pfad.

2. Jetzt den Rückfluss graphen einzeichnen. Hierfür dreht man die Pfeile einfach um und schreibt die übergebliebenen Kapazitäten zu den „richtigen“ Pfeilen und den flow zu den Umgedrehten. Das sieht in unserem Beispiel dann wie folgt aus:



3. Jetzt nimmt man einen neuen Pfad, zum Beispiel *s-v-u-t.* Man kann diesen Pfad wählen, weil wir durch den Rückfluss von vorherigen flow eine Rückkante bei *v-u* entstanden ist. Man schickt wieder einen flow durch, in diesem Fall mit der Rate =2.

4. Erneut Rückkanten einzeichnen:
   Der Flow bei der Kante *v-u* betragt jetzt Null, weil sich die zwei entgegengesetzten flows durch die Rechnung:

   Residualkapazität(uv) = flow1(uv) – flow2(uv) = 2 – 2 = 0

5. Jetzt gibt es keinen Pfad mehr, der source mit sink verbindet. Deshalb werden alle durchgeschickten flows addiert:
   max flow = 2+2 = 4

Praktische Implementation des Ford Fulkerson Algorithmus in Python unter Verwendung von NetworkX:

Ford Fulkerson Algorithmus:

Step 0: Initialisiere Fluss bei 0.

Step 1 Pathfinder – find any augmenting Path .

Step 2 Fill said Path to max capacity .
Account for Backflow

Our program starts by declaring a source and a sink to our network, then defining the edges with according weights, in this case representing the capacity or the maximum flow rate of a given connection, and adding them to our network “netz”.
Afterwards a flow grid is created containing the same number of nodes as our initial network, as well as all edges with a weight / flowrate of 0. Now we determine our maximum flow using the following function:

```
def determine_maxflow(netz, source, sink):
    flow = init_flow(netz)
    max_flowrate= 0

    naive = False

    while (True):
        if naive is True:
            new_network = flow_difference_naive(netz,flow)
        else:
            new_network = flow_difference(netz,flow)
        path = DiGraph.find_path(new_network, source,sink)
        #path = find_shortest_path(new_network,source,sink)
        if (path == []):
            break
        bottleneck = find_bottleneck(path,new_network)
        max_flowrate += bottleneck
        flow = fill_flow(new_network,flow,path,bottleneck)
    #print("MaxFlowrate:" + str(max_flowrate))
    return max_flowrate
```

The main algorithm first creates a residual network. In our first approach this was done by simply creating a new network, subtracting the weights of our flow grid from the initial grid and only adding edges with a capacity greater than zero. However, this algorithm did not account for possible backflow within pipes and in some cases leads to wrong outcomes. To solve this, we create additional edges in the opposite direction whose capacity we set to the current flow, which will then subtract from our flow grid as they fill up. The resulting function looks like this:

```
def flow_difference(initial_network, flow_network):

    netz_new = DiGraph()
    for u,v in initial_network.edges:
        initial_weight = initial_network.get_edge_data(u,v)['weight']
        flow_weight = flow_network.get_edge_data(u,v)['weight']
        difference = initial_weight - flow_weight
        if difference != 0:
            netz_new.add_edge(u,v,weight = difference)
        if flow_weight != 0:
            netz_new.add_edge(v,u,weight = flow_weight)
    return netz_new
```

Within this network any pathfinding algorithm - in our case we could use the inbuilt pathfinder or define our own - tries to find any augmenting path between out sink and source nodes, an augmenting path being a path without cycles only containing edges with positive capacities. In case there is no connection available the maximum capacity of the entire network has already been reached and our algorithm ends. Otherwise, the maximum capacity of our connection is determined by finding the smallest capacity of any edge that our path contains - the so called "bottleneck" - using the following function:

```
def find_bottleneck(path,netz):
    weights = []
    for i in range(0,len(path)-1):
        weights.append(netz.get_edge_data(path[i],path[i+1])['weight'])

    return min(weights)
```

The flow network is then updated along the found path by increasing the flow of all its edges by the value of our bottleneck using our function "fill_flow" which looks like this:

```
def fill_flow(diff_graph, flow_network,path,bottleneck):
    for i in range(0,len(path)-1):
        if flow_network.has_edge(path[i],path[i+1]):
            old_weight = flow_network.get_edge_data(path[i],path[i+1])['weight']
            flow_network.remove_edge(path[i],path[i+1])
            flow_network.add_edge(path[i],path[i+1], weight = bottleneck + old_weight)
        else:
            old_weight = diff_graph.get_edge_data(path[i],path[i+1])['weight']
            flow_network.remove_edge(path[i+1],path[i])
            flow_network.add_edge(path[i+1],path[i], weight = old_weight - bottleneck)

    return flow network
```

This function also accounts for our implementation of possible backflow within our pipes by updating the flow network accordingly, that being by adding to existing paths and subtracting from a given path when going in reverse.

Then our program repeats by creating another residual network using the updated flow.

Expanding on our existing algorithm we can tackle some modest optimization problems. Say we want to increase our flow by incrementing a single edge's capacity by one. Here is a simple algorithm to determine our optimal candidates sorted by smallest capacity edge to increase:

```
maxflow = determine_maxflow(netz, source, sink)
weightlist = []

for u,v in netz.edges:
    weightlist.append(netz.get_edge_data(u,v)["weight"])
weightlist = list(dict.fromkeys(weightlist))
weightlist.sort()

candidates = []
for minweight in weightlist:
    minweight_edges = []
    for u,v in netz.edges:
        if (netz.get_edge_data(u,v)["weight"] == minweight):
            minweight_edges.append([u,v])

    for minedge in minweight_edges:
        new_graph = DiGraph()
        for edge in netz.edges:
            origweight = netz.get_edge_data(edge[0],edge[1])["weight"]
            if (edge[0] == minedge[0]) and (edge[1] == minedge[1]):
                origweight+=1
            new_graph.add_edge(edge[0],edge[1],weight = origweight)
        new_maxflow = determine_maxflow(new_graph, source, sink)

        if (new_maxflow > maxflow):
            candidates.append(minedge)
print(candidates)
```

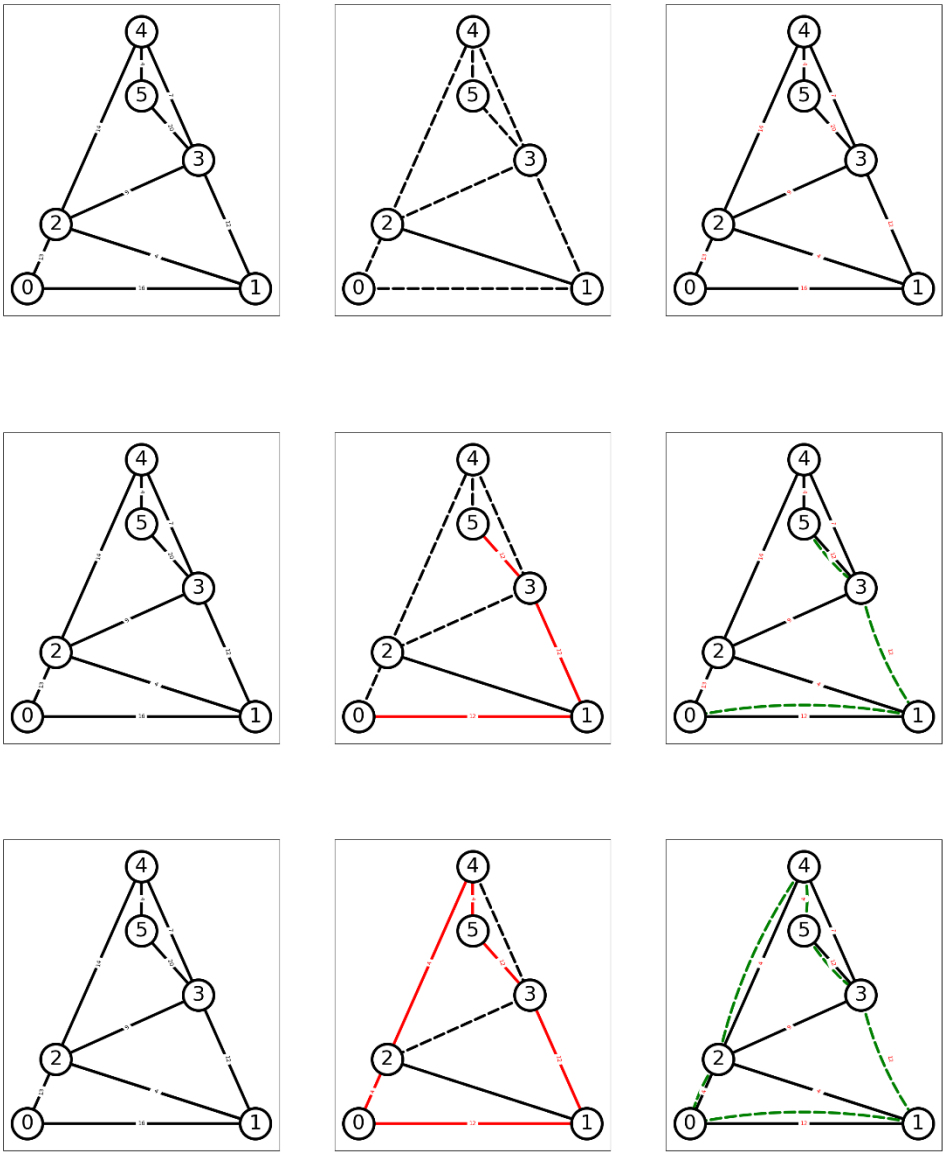
Additionally, we implemented a minimal cost algorithm by expanding our edges by another variable named cost and determining our path by smallest cost using the Dijkstra-algorithm for pathfinding instead of choosing a random path.

Verlauf Ford Fulkerson an einem Beispiel (implementiert in Python)

Links: Original Netzwerk

Mitte Fluss Netzwerk

Rechts: REsidualnetzwerk



Quellen

Modelica by Example: https://mbe.modelica.university/ (abgerufen 06.01.2024)

Wikipedia (Ford Fulkerson): https://de.wikipedia.org/wiki/Algorithmus_von_Ford_und_Fulkerson ((abgerufen 08.01.2024)

FFTs Blog (min cost max flow): https://codeforces.com/blog/entry/105330 (abgerufen 09.01.2024)

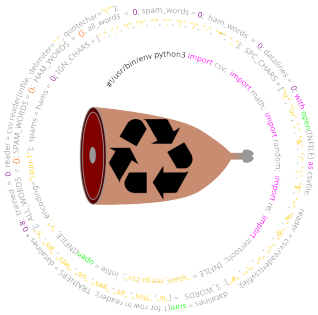
Korte, Bernhard H., et al. *Combinatorial optimization*. Vol. 1. Berlin: Springer, 2011.

Kommunikation: Spam-Mail Classification

Olivia Maria Ringhofer Felix Fromm Alexander List
Navid Pavel Milo Giuliani Alexander Goritschnig

Betreuer: Matthias Höfler

12. Januar 2024



Inhaltsverzeichnis

1 Einleitung

Als Monty Python ihren Sketch “Spam“ veröffentlichten, war ihnen offenbar noch nicht bewusst, was sie alles damit auslösen würden. In den 90ern wurde er zur Basis des Terms “Spam”, als Begriff für unerwünschte, lästige Nachrichten, die wir nicht in unseren Posteingängen sehen wollen. Um dem gerecht zu werden, haben wir uns damit beschäftigt, ebenjene Nachrichten automatisch zu erkennen und zu filtern. Selbstverständlich haben wir den eigentlichen Filter in der Programmiersprache Python geschrieben, die ebenfalls nach der britischen Comedy-Gruppe benannt wurde. Jedoch sagt dies natürlich nichts über die Qualität der Sprache aus. Oder? Also, zumindest sagt es nichts über die Qualität unseres Spam-Klassifizierers aus.

2 Features

2.1 Definition

Ein Feature ist ein Indikator oder Kennzeichen für eine bestimmte Katgorie (in diesem Fall: Spam oder Ham). Am Beginn der Modellierungswoche haben wir uns mögliche Beispiele überlegt:

- **Worthäufigkeiten:**
  Kommt ein bestimmtes Wort vor, bzw. wie oft kommt es vor?
- **Links:**
  Ist ein Link vorhanden?
- **Nachrichtenlänge:**
  Wie viele Wörter/Zeichen besitzt die Nachricht?
- **Buchstabenhäufigkeiten:**
  Kommen bestimmte Buchstaben vor, bzw. wie oft kommen sie vor?

2.2 Evaluierung der Features

Im Laufe der Woche stellten sich einige dieser Beispiele als gute Features heraus, und andere als weniger nützlich - denn je häufiger ein Feature in Spamnachrichten als im Verlgeich zu Hamnachrichten vorkommt, desto besser ist dieses, um Spam bzw. Ham zu identifizieren. Mithilfe von Python kann man berechnen, wie gut ein Feature ist. Anhand dessen wählt man (manuell od. dynamisch) jene Features, welche man zur Klassifizierung verwenden möchte, aus. Wichtig dabei ist, dass man nicht zu viele Features verwendet (overfitting), da dadurch der Klassifizierer abhängig von dem Trainingsdatensatz wird und somit das Ergebnis für andere Testdaten nicht mehr aussagekräftig ist.

Beispiel: Berechnung der Features

Für die Berechnung, welche Features geeignet sind haben wir uns die Verteilung/Aufteilung der einzelnen Features in Spamnachrichten bzw. in Hamnachrichten angeschaut.

$$\frac{\#\text{ Feature in Spammails}}{\#\text{ Feature in Gesamt}} \; bzw. \; \frac{\#\text{ Feature in Hammails}}{\#\text{ Feature in Gesamt}}$$

Anschließend wird bestimmt ob das ausgerechnete Verhältnis über der im Vorherein festgelegten Verteilungsgrenze (bei uns: 0.95) liegt, damit nur die jeweils ausschlaggebensten Features (kommen in einer Kategorie viel häufiger als in der anderen vor) ausgesucht werden.
Danach wird das berechnete Verhältnis mit der Anzahl des Wortes in der Kategorie multipliziert, um die Relevanz der Wörter besser zu ermitteln können. Beispiel: Kommt ein Wort einmal in Spammails vor und nie in Hammails, ist zwar das Verhältnis über der Verteilungsgrenze, das Wort selbst hat jedoch wenig Aussagekraft. Weiteres dividiert man

dieses Ergebnis durch die Anzahl der gesamten Emails der jeweiligen Kategorie, um eine ungleiche Aufteilung der Spammails und Hammails in einem Datensatz auszugleichen.

$$\frac{\text{Verhältnis } * \#\text{ des Wortes in Kategorie}}{\#\text{ der gesamten Emails in Kategorie}}$$

Das Ergebnis ist als Tauglichkeits des Features zu interpretieren. Anhanddessen können dann weitere Berechnungen zur Klassifizierung unternommen werden.

```
def featureVerteilung(data, indicator):
#Berechnung der Verteilung der Woerter fuer Gewichtung
#durch indicator auch fuer ham anwendbar
    #wordCount["free"] / gesamtWordCount["free"] = wie viel Prozent des Wortes "free" in spam ist
    wordCount = {}
    gesamtWordCount  = {}
    wordVerteilung = {}

    #wenn Wort mehr als verteilungsGrenze Prozent nur in spam vorkommt,
    #dann wird es mit seiner Anzahl multipliziert und in ueVerteilungsGrenzeWordCount gespeichert
    verteilungsGrenze = 0.95
    ueVerteilungsGrenzeWordCount = {}

    #Liste der  eigene features welche keine Woerter sind ("0", "!", "?", etc)
    ownFeatures = ["0","!", "?", "http","$"]

    #durchgehen der einzelnen Emails
    for zeile in data:
        #zaehlen der eigenen Features, muss vorm Herausziehen der Woerter passieren, weil dort viel geloescht wird
        for feature in ownFeatures:
            if feature in zeile[1]:
                if zeile[0] == indicator:
                    if feature in wordCount:
                        #wenn das Wort im Text (zeile[1]) vorkommt, die Email ein spam ist, und eh schon in wordCount dann wird die Anzahl des features dazuaddiert
                        wordCount[feature]+=zeile[1].count(feature)
                    else:
                        #wenn das Wort im Text  (zeile[1]) vorkommt, und noch nicht in wordCount dann wird die Anzahl des features als counter gesetzt
                        wordCount[feature]=zeile[1].count(feature)
                if feature in gesamtWordCount:
                    #wenn das feature schon in gesamtWordCount ist, wird dieser count um den counter des features erhoeht
                    gesamtWordCount[feature]+=zeile[1].count(feature)
                else:
                    #wenn das Wort noch nicht in gesamtWordCount ist, wird dieser count auf den counter des features gesetzt
                    gesamtWordCount[feature]=zeile[1].count(feature)
        #Herausziehen der Woerter
        punctuation = ".,:;?!&/*#()-_$"
        for character in punctuation:
            if character in zeile[1]:
                zeile[1] = zeile[1].replace(character, "")
        words = zeile[1].split(" ")
```

```
41          #durchgehen  der einzelnen Woerter dieser einen Email
42          for word in words:
43              if zeile[0] == indicator:
44                  #wenn Email ein spam ist und das Wort schon in wordCount
     ist, wird dieser counter um Eins erhoeht
45                  if word in wordCount:
46                      wordCount[word]+=1
47                  #wenn Email ein spam ist und das Eort noch nicht in
    wordCount ist, wird dieser counter auf Eins gesetzt
48                  else:
49                      wordCount[word]=1
50              #wenn das Wort schon in gesamtWordCount ist, wird dieser
    counter um Eins erhoeht
51              if word in gesamtWordCount:
52                  gesamtWordCount[word]+=1
53              #wenn das Wort noch nicht in gesamtWordCount ist, wird
    dieser counter auf Eins  gesetzt
54              else:
55                  gesamtWordCount[word]=1
56
57      for spamyWord in wordCount:
58          #Berechnung der Verteilung
59          wordVerteilung[spamyWord] = wordCount[spamyWord] /
    gesamtWordCount[spamyWord]
60          #schauen ob wort mehr als verteilungsGrenze Prozent nur in
    indikator vorkommt
61          if wordVerteilung[spamyWord] > verteilungsGrenze:
62              #mit seiner Anzahl multipliziert und in
    ueVerteilungsGrenzeWordCount gespeichert
63                  #Frage: warum wird hier multipliziert?
64                  #weil wir dann sehen wie oft woerter welche zu 95% in
    spam vorkommen, ueberhaupt in spam vorkommen und somit deren Relevanz
     besser ermitteln koennen
65                  #UND damit wir dann mit der anzahl an spamEmails
    dividieren koennen um das Ergebnis zu relativieren
66                  #weil Problem: die Funktion mit indikator="ham"
    aufgerufen gibt hoehere Ergebnise weil ham 8 mal mehr Mails hat ->
    abhaengig von Datensatz
67              ueVerteilungsGrenzeWordCount[spamyWord] = (wordCount[
    spamyWord] * wordVerteilung[spamyWord])/len(wordCount)
68      ueVerteilungsGrenzeWordCount_absteigend = sorted(
    ueVerteilungsGrenzeWordCount.items(), key=lambda x: x[1], reverse=
    True)
69
70      return ueVerteilungsGrenzeWordCount_absteigend
```

2.3 Worthäufigkeiten

Hauptsächlich haben wir als Features Worthäufigkeiten - sowohl in Spammails als auch in Hammails, und das Vorkommen von Links benutzt, da diese zu einem zufriedenstellenden Ergebnis geführt haben. Features wie die Nachrichtenlänge oder die Buchstabenhäufigkeit waren nicht wirklich aussagekräftig, da diese in Spammails und Hammails gleichermaßen oft vorkommen. Mithilfe von Python haben wir die Wörter in einem Email gezählt.

Python-Code zur Berechnung der Worthäufigkeiten

```
for word in words:
    if zeile[0] == indicator:
    #wenn Email ein spam ist und das Wort schon in spamWordCount ist,
   wird dieser counter um Eins erhoeht
        if word in spamWordCount:
            spamWordCount[word]+=1
            #wenn Email ein spam ist und das Eort noch nicht in
   spamWordCount ist, wird dieser counter auf Eins gesetzt
            else:
                spamWordCount[word]=1
            #wenn das Wort schon in gesamtWordCount ist, wird dieser
   counter um Eins erhoeht
    if word in gesamtWordCount:
        gesamtWordCount[word]+=1
        #wenn das Wort noch nicht in gesamtWordCount ist, wird dieser
   counter auf Eins  gesetzt
    else:
        gesamtWordCount[word]=1
```

Nach den Berechnungen hat sich gezeigt, dass Wörter wie “FREE“ oder “claim“ viel häufiger in Spamnachrichten vorkommen als in Hamnachrichten.

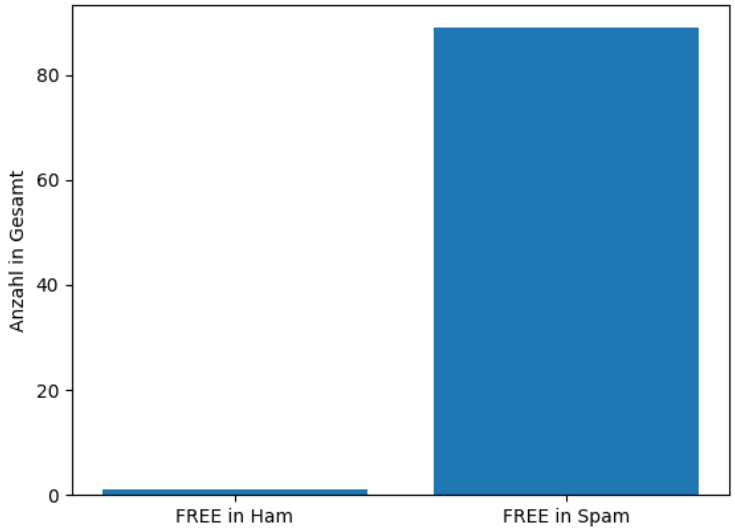


Abbildung 1: Anzahl des Wortes “FREE“ in Hammails im Vergleich zu Spammails

Betrachtet man die Worthäufigkeiten ausschließlich bei Spamnachrichten, ergibt sich folgendes Ergebnis:

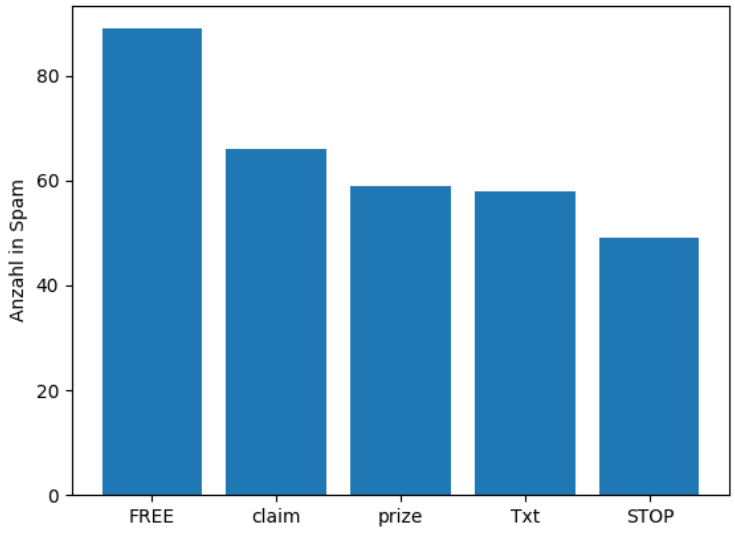


Abbildung 2: Vergleich der Häufigkeiten der einzelnen Wörter in Spammails

Jene gelisteten Wörter sind aufgrund ihrer Tauglichkeit als Features zur Klassifizierung verwendet worden.

2.4 Behandlung von Features:

Je nach Art der Features lassen sie sich anders behandeln. Beispielsweise haben wir anfangs die Worthäufigkeit ausschließlich als Bernoulli-Feature benutzt. Hierbei bestimmt man die Anzahl jener Emails, in denen das Feature vorkommt (ja/nein). Eine weitere Methode, welche wir erst im Laufe der Woche angewendet haben, ist es, auch die Anzahl eines bestimmten Wortes, die in einem Email vorkommen, zu berücksichtigen. Wie sinnvoll eine Methode ist, hängt vom Anwendegebiet ab und muss mit jedem neuen Fall neu abgeschätzt werden.

Beispiel:

"FREE NOKIA Or Motorola with upto 12mths 1/2price linerental, 500 FREE x-net mins&100txt/mth FREE B'tooth*. Call Mobileupd8 on 08001950382 or call 2optout/D3WV"

Abbildung 3: Beispiel-Email (Spam): nach der Bernoulli-Methode würde 1 Punkt vergeben werden, da das Feature "FREE" vorkommt (0 = nicht enthalten; 1 = enthalten). Mit der zweiten Methode würden 3 Punkte vergeben werden, da das Feature "FREE" dreimal vorkommt.

3 Bernoulli-Verteilung

Die Bernoulli-Verteilung war zur Wertung einiger unserer Features zu Beginn der Woche essenziell. Folgendes Kapitel dient als Erklärung dieser Verteilung.

3.1 Erklärung der Bernoulli Verteilung

Die Bernoulli-Verteilung ist eine Wahrscheinlichkeitsverteilung einer Zufallvariable X die die Werte 1 oder 0 annehmen kann. Sie beschreibt Zufallsereignisse, für diese es nur zwei Ausgänge gibt. Dabei beschreibt $X = 1$ den Fall, dass ein Ereignis erfolgreich ist und $X = 0$ den Fall, dass ein Ereignis nicht erfolgreich ist.
Die Wahrscheinlichkeit p eines erfolgreichen Ausgangs kann aus folgender Formel geschätzt werden:

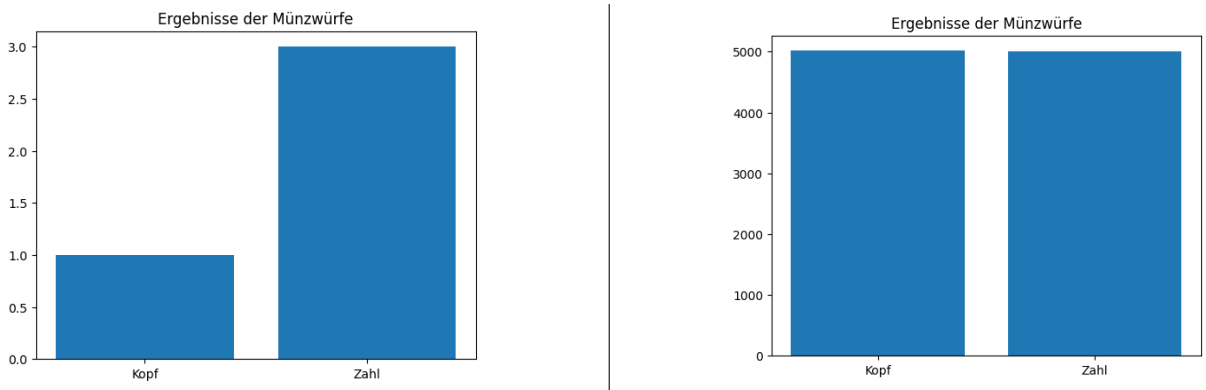
$$p = \frac{N_1}{N_0 + N_1} \tag{1}$$

wobei N_1 die Anzahl der Erfolge und N_0 die Anazhl Misserfolge entspricht.

3.1.1 Gesetz der großen Zahlen mit dem Beispiel Münzwurf

Eine mögliche Anwendung der Bernoulli-Verteilung liegt bei dem Münzwurf vor. Anfangs legen wir das Ergebnis $Kopf$ als Erfolg fest und das Ergebnis $Zahl$ als Misserfolg.

Um die Wahrscheinlichkeit des Ergebnisses $Kopf$ nun zu ermitteln muss nun die Anzahl der $Kopf$-Würfe mit der Anzahl der Münzwürfe dividiert werden. Ausschlaggebend für die Genauigkeit dieser Abschätzung ist die Anzahl Münzwürfe.
Das wird durch die folgenden Graphiken gezeigt:



(a) Ergebnisse bei 5 Münzwürfen

(b) Ergebnisse bei 10 000 Münzwürfen

Abbildung 4: Vergleich zwischen 5 und 10 000 Münzwürfen

Die Ergebnisse der Graphiken wurden durch eine Python-Library mit folgendem Code simuliert:

```
n = 10000    #Anzahl der Wuerfe
p = 0.5      #Theoretische Wahrscheinlichkeit
np.random.binomial(n,p)
```

Allgemein bedeutet das, dass die Abschätzung der Wahrscheinlichkeit genauer wird, je mehr Versuche es gibt. Dieses Prinzip wird auch das Gesetz der großen Zahlen genannt.

3.2 Beweis der Formel

Die Formel für die Schätzung der Wahrscheinlichkeit p (vgl. 1) kann folgenderweise hergeleitet werden:

Die Wahrscheinlichkeit des Falles $X = 1$ wird mit p bezeichnet. Die Wahrscheinlichkeit des Falles $X = 0$ ist durch die Gegenwahrscheinlichkeit $1 - p$ definiert. Das heißt:

$$\begin{aligned} \mathbb{P}(X = 1) &= p \\ \mathbb{P}(X = 0) &= 1 - p \end{aligned} \tag{2}$$

Daraus folgt die Wahrscheinlichkeitsfunktion $f(y)$ mit der Zufallsgröße X, die die Werte 0 oder 1 annehmen kann:

$$f(y) = \mathbb{P}(X = y) = p^y * (1 - p)^{1-y} \tag{3}$$

Um p aus einer Liste von Beobachtungen abzuschätzen kann diese nach der Annahme der Unabhängigkeit der Beobachtungen in folgendes Produkt umgeschrieben werden.

$$\begin{aligned} &\mathbb{P}(X_1 = y_1, ..., X_n = y_n) \\ &= \mathbb{P}(X_1 = y_1) * ... * \mathbb{P}(X_n = y_n) \end{aligned} \tag{4}$$

Nach dem Einsetzen der Wahrscheinlichkeitsfunktion lässt sich die Funktion $L(p)$ wie folgt beschreiben:

$$L(p) = \prod_{i=1}^{n} f(y_i) \tag{5}$$

Anschließend lässt sich durch Logarithmieren die Funktion in eine Summe umschreiben:

$$\log L(p) = \sum_{i=1}^{n} \log f(y_i) \tag{6}$$

Um die abgeschätzte Wahrscheinlichkeit p zu berechnen ist das Nullsetzen der 1. Ableitung der Funktion $L(p)$ notwendig, um die Funktion zu maximieren. Durch Umschreiben der Wahrscheinlichkeitsfunktion in die Form $p_i^y * (1 - p)^{1-y_i}$ und Ableiten dieser kann die Summe vereinfacht werden:

$$\sum_{i=1}^{n} \log f(y_i) = \sum_{i=1}^{n} \log \left(p_i^y * (1 - p)^{1-y_i} \right) \tag{7}$$

Durch Ableiten:

$$\begin{aligned}
&\frac{d}{dp}\sum_{i=1}^{n}\log f(y_i) \\
&=\sum_{i=1}^{n}\left[\frac{y_i}{p}+\frac{1-y_i}{1-p}*(-1)\right] \\
&=\sum_{i=1}^{n}\left[\frac{y_i-y_i*p-p+y_i*p}{p*(1-p)}\right] \\
&=\sum_{i=1}^{n}\left[\frac{y_i-p}{p*(1-p)}\right]
\end{aligned} \tag{8}$$

Da beim Nullsetzen des Terms nur der Zähler betrachtet werden muss ergibt sich schließlich:

$$\sum_{i=1}^{n}[y_i-p]=0 \tag{9}$$

Die Summe entspricht also:

$$y_1+...+y_i-p*n=0 \tag{10}$$

Durch umformen nach p ergibt sich also:

$$p=\frac{y_1+...+y_i}{n} \tag{11}$$

Da die Summe aller y_i der Anzahl der Erfolge und n die Anzahl aller Versuche gleicht lässt sich daraus der Extremwert p folgenderweise beschreiben:

$$p^*=\frac{N_1}{N_0+N_1} \tag{12}$$

4 Textverarbeitung

Um den Text, in unserem Fall vorwiegend den Inhalt aus Mails, leichter als Spam und Ham zu klassifizieren, muss der Text in ein Format gebracht werden, welches eine einfachere Handhabung erlaubt.

4.1 In Wörter aufteilen

Da unsere Features hauptsächlich auf die Anzahl des Vorkommens eines Wortes schauen, war es für uns unerlässlich den Text in seine einzelnen Wörter aufzuteilen.
Die Methode welche einem zuerst erscheint, ist es, im Vorhinein Satzzeichen zu entfernen und darauffolgend den Text anhand seiner übrig gebliebenen Leerzeichen auseinanderzutrennen.

Daraus ergibt sich folgende Konstellation:

'Hallo! Das ist ein Test.' - 'Hallo Das ist ein Test' - ['Hallo','Das', 'ist'...]

Diese Methodik funktioniert auf den ersten Blick, stoßt aber bei dem Arbeiten mit großen Datensätzen auf ein Problem: Es berücksichtigt nicht die Möglichkeit, dass zwei Wörter nur durch ein Satzzeichen, also ohne Leerzeichen, getrennt werden.
Dies ist nicht nur der Fall wenn der Text fehlerhaft ist, weil er die allgemeine Regel, dass nach gängigen Satzzeichen immer ein Leerzeichen folgt, nicht einhält, sondern auch bei einigen Sonderfällen. Häufig vorkommend ist der Fall, dass zwei Wörter nur durch einen Zeilenumbruch ("\n") getrennt werden.

Dieses Problem wird mit der Verwendung von Regex vermieden. Folgender Python-Code zeigt genau dies:

```
1 import re
2 text = "hallo  test !?\n\ttesting\thello"
3 words = re.split(r'[ !?\.,:;_"&/\n//\t/]+', text)
```

words = ['hallo', 'test', 'testing', 'hello']

Durch das Regex "[!?\.,:;_"&/\n//\t/]+"wird des Weiteren auch bei Vorkommen mehrerer Satzzeichen ein passendes Ergebnis geliefert.

4.2 Wörter aufarbeiten

Das aufarbeiten der einzelnen Wörter hilft uns unteranderem performanter zu sein, durch zum Beispiel das Entfernen unnötiger Wörter wie Stopwords, andererseits hilft es uns auch, Indikatoren zu finden und anzuwenden.

4.2.1 Tippfehler

Zwar kann die Anzahl an Tippfehlern als ein Indikator für Spam beziehungsweise Ham gesehen werden, andererseits sollten diese Tippfehler ausgebessert werden um die Bedeutung hinter falsch geschriebener Wörter leichter zu evaluieren. Die Python-Bibliohtek pyspellchecker hilft einem dabei enorm falsche Wörter zu finden und auszubessern. Folgender Python-Code gilt als Beispiel der Anwendung von pyspellchecker:

```
from spellchecker import SpellChecker
words = ["This", "Teext", "ies", "veri", "wong"]
for i in range(len(words)):
        word[i] = spell.correction(word[i])
```

4.2.2 Stemming

Stemming, auf Deutsch “entstammen “, sorgt dafür, die Stammform eines Wortes zu bekommen. Das ist besonders wichtig um den eigentlichen Sinn eines Wortes zu erheben.
Dabei hilft uns NLTK, stehend für Natural Language Toolkit (Deutsch: Toolkit für natürliche Sprache) beim besagten Finden von Stammformen der Wörter. Folgender Python-Code verwendet die nltk Bibliothek:

```
from nltk.stem import PorterStemmer
porter_stemmer = PorterStemmer()
words = ["running", "jumps", "easily", "fairly", "happily", "happiness"]
stemmed_words = [porter_stemmer.stem(word) for word in words]
```

4.2.3 Stopwords

Stopwords, auf Deutsch Stoppwörter, stehen für Wörter, welche bei weiterer Verarbeitung als irrelevant angesehen werden. Stopwords sind unter anderem “der“, “die“, “das“, “sein“, “haben“ etc.
Diese Wörter haben keine Relevanz bei der Evaluierung einer Mail, weil diese keine Bedeutung mit sich führen. Eine Möglichkeit, stopwords zu entfernen bringt wieder die Python-Bibliothek nltk mit sich. Die Anwendung zeigt folgender Python-Code:

```
from nltk.corpus import stopwords
import nltk
words = ["it", "is", "free"]
stopWords = set(stopwords.words("english"))
wordsWithoutStopWords = [word for word in words if word.lower() not in
    stop_words]
```

5 Datensatz für Ham und Spam Mails

Einen Großteil unserer Datensätze haben wir von Kaggle gezogen. Kaggle ist eine Online-Plattform für maschinelles Lernen und Datenwissenschaft und bietet eine umfangreiche Sammlung von öffentlichen Datensammlungen. Beim Wählen eines Datasets war uns wichtig, dass die Mails im vorhinein bereits nach ham und spam kategorisiert sind. Zum Einem dient uns dies zum Trainieren unseres Models aber zum Anderen auch zum Testen.

In den von uns am Anfang benutzten Datensätzen waren ∼5500 englische Emails vertreten, in einem Verhältnis von 13% Spam zu 87% Ham.
Weiters ist es wichtig zu betonen, dass man nie mit den selben Daten das Model trainieren und testen sollte. Zwar würde das Testergebnis besser werden, andererseits ist das Ergebnis als verfälscht und als nicht mehr realitätsgetreu anzusehen.

Im Laufe der Modellierungswoche wurden wir dann auch von unserem Betreuer mit einem Test- und Train- Datensatz versorgt. Dies half uns die Ergebnisse innerhalb der Gruppe zu vereinheitlichen und somit besser zu vergleichen.

6 Naive Bayes Annahme

Dieses Kapitel beschäftigt sich mit unserer ersten Methode, die E-Mails wirklich zu klassifizieren. Dabei wird näher auf den mathematischen Hintergrund dieser Methode eingegangen und dann auch das Programmieren eines solchen Algorithmus anhand von Beispielen dokumentiert.

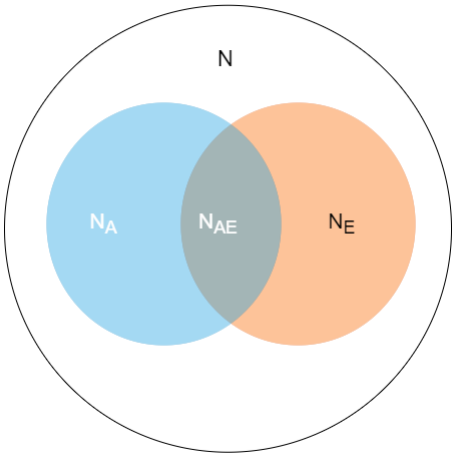
6.1 Satz von Bayes

Als Grundlage für diese Annahme dient der Satz von Bayes, mit dem bedingte Wahrscheinlichkeiten beschrieben werden können. Umgemünzt auf die Problemstellung der Klassifizierung von E-Mails in Spam- und Ham-Nachrichten ergibt sich daraus folgende Gleichung:

$$\mathbb{P}(Spam|F) = \frac{\mathbb{P}(F|Spam) * \mathbb{P}(Spam)}{\mathbb{P}(F)} \tag{13}$$

Der Satz von Bayes lässt sich dabei mithilfe eines Venn-Diagramms beweisen. Dabei definieren wir folgende vier Mengen und Anzahlen, deren Zusammenhang im Diagramm dargestellt ist:

$$N, N_A, N_E \text{ und } N_{AE} \tag{14}$$



Dafür lassen sich nun folgende Wahrscheinlichkeiten bestimmen:

$$\mathbb{P}(A) = \frac{(N_A)}{(N)} \tag{15}$$

$$\mathbb{P}(E) = \frac{(N_E)}{(N)} \tag{16}$$

$$\mathbb{P}(E|A) = \frac{(N_{AE})}{(N_A)} \tag{17}$$

$$\mathbb{P}(A|E) = \frac{(N_{AE})}{(N_E)} \tag{18}$$

Daraus ergibt sich schließlich der Satz von Bayes, der im Zusammenhang dieser Mengen so aussieht:

$$\mathbb{P}(E|A) = \frac{\mathbb{P}(E|A) * \mathbb{P}(E)}{\mathbb{P}(A)} \tag{19}$$

Durch diesen Satz berechnen wir die Wahrscheinlichkeit dafür, dass ein Mail Spam ist, wenn ein Feature darin vorkommt. Da mehrere Features nötig sind, um die richtige Klassifizierung von Nachrichten zu erleichtern, muss dies in der Gleichung berücksichtigt werden.

6.2 Unabhängigkeit

Um dies zu erleichtern, nehmen wir Unabhängigkeit an. Das heißt, wir gehen davon aus, das die unterschiedlichen Features keinen Einfluss aufeinander haben. Daraus folgt, dass man die Wahrscheinlichkeiten für die Features in Spam-Nachrichten multiplizieren darf. Auch wenn diese Annahme meist unwahr und somit unrealistisch ist, können Wahrscheinlichkeiten so beschrieben werden und sie ist somit auch für unsere Klassifizierung nützlich. Wenn wir also zum Beispiel von 2 Features ausgehen, ergibt sich folgende Gleichung:

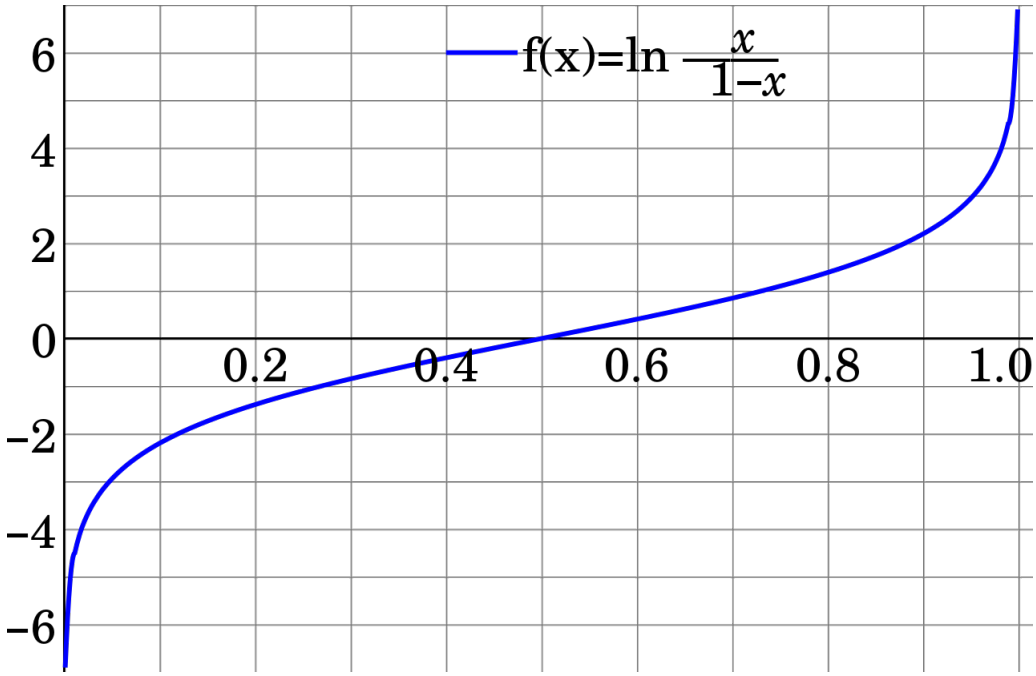
$$\mathbb{P}(Spam|F_1, F_2) = \frac{\mathbb{P}(F_1|Spam) * \mathbb{P}(F_2|Spam) * \mathbb{P}(Spam)}{\mathbb{P}(F_1, F_2)} \tag{20}$$

6.3 Logit-Funktion

Jetzt sehen wir uns die Logit-Funktion an und versuche sie für unsere Problemstellung zu nutzen. Die Logit-Funktion ist folgendermaßen definiert:

$$logit(x) = \ln\left(\frac{x}{1-x}\right) \tag{21}$$

Aufgrund von dieser Definition ist diese Funktion gut für unser Klassifizierungsverfahren geeignet. Wegen des Schnittpunkts des Graphen der Funktion an der x-Achse an der Stelle 0.5, können Wörter und in weiterer Folge auch Nachrichten gut als Spam oder Ham eingestuft werden. Ist nämlich die Wahrscheinlichkeit, dass eine Nachricht Spam ist, wenn ein Feature vorkommt, größer als 0.5, so nimmt die Logit-Funktion einen positiven Wert an. Ist die Wahrscheinlichkeit jedoch kleiner als 0.5, nimmt sie einen negativen Wert an.



Mithilfe von folgenden Schritten lässt sich zeigen, wie die Funktion für gegebenen Sachzusammenhang genau genutzt werden kann:
1) Umschreiben von bedingten Wahrscheinlichkeiten durch den Satz von Bayes unter Annahme von Unabhängigkeit:

$$\mathbb{P}(Spam|F_1, F_2) = \frac{\mathbb{P}(F_1|Spam) * \mathbb{P}(F_2|Spam) * \mathbb{P}(Spam)}{\mathbb{P}(F_1, F_2)} \tag{22}$$

$$\mathbb{P}(Ham|F_1, F_2) = \frac{\mathbb{P}(F_1|Ham) * \mathbb{P}(F_2|Ham) * \mathbb{P}(Ham)}{\mathbb{P}(F_1, F_2)} \tag{23}$$

2) Verhältnis zwischen den bedingten Wahrscheinlichkeiten bilden. Dadurch können die schwer bestimmbaren Gesamtwahrscheinlichkeiten der Features aus der Gleichung gekürzt werden. So ergibt sich folgende Form:

$$\frac{\mathbb{P}(Spam|F_1, F2)}{\mathbb{P}(Ham|F_1, F_2)} = \frac{\mathbb{P}(F_1|Spam) * \mathbb{P}(F_2|Spam) * \mathbb{P}(Spam)}{\mathbb{P}(F_1|Ham) * \mathbb{P}(F_2|Ham) * \mathbb{P}(Ham)} \tag{24}$$

Wenn man nun die Gegenwahrscheinlichkeit der Wahrscheinlichkeit für Ham, wenn ein Feature gegeben ist, berechnet und die entstandene Gleichung mithilfe des ln logarithmiert, entsteht die Form der Logit-Funktion. Somit ist bewiesen, dass Folgendes gilt:

$$logit(\mathbb{P}(Spam|F_1, F2, ..., F_l) = \log \frac{\mathbb{P}(Spam)}{\mathbb{P}(Ham)} + \sum_{i=1}^{n} \log \frac{\mathbb{P}(F_i|Spam)}{\mathbb{P}(F_i|Ham)} \tag{25}$$

7 Algorithmus

7.1 Naive Bayes Classifier

Mithilfe der letzten Formel kann nun mit dem Programmieren eines Algorithmus, der Nachrichten den Kategorien Ham und Spam zuordnen kann, begonnen werden. Nach den ersten Textverarbeitungsschritten können die Wörter, die sowohl in Ham als auch Spam-Nachrichten vorkommen, in die Formel eingesetzt werden.

Beispiel: Einsetzung der Formel in Python

```
logs={}
for word in worddict_spam:
  if word in worddict_ham:
    logs[word]=math.log(worddict_spam[word]/worddict_ham[word])+math.log
   (count_spam/count_ham)
```

Mithilfe der Werte, die aus der Formel-Berechnung entstehen, können nun Spam- und Ham-Indikatoren unterschieden werden:

Beispiel: Aufteilung der Indikatoren

```
keywords_spam={}
keywords_ham={}
for word in logs:
  if logs[word]>0:
    keywords_spam[word]=logs[word]
  if logs[word]<0:
    keywords_ham[word]=logs[word]
```

Jetzt kann die Funktion zur Klassifizierung der Nachrichten definiert werden.

Beispiel: Definieren des Algorithmus

```
# nimmt einen Text als Input entgegen und als Rueckgabewert ein Label
   mit "ham" oder "spam" aus
def naive_bayes_classifier(text):
  words=normalize(text)
  # entfernt Satzzeichen aus den E-Mails
  score=0
  for word in words:
    if word in keywords_spam:
      score+=1
    if word in keywords_ham:
      score=1

  if score >0:
    label="spam"
  else:
    label="ham"

  return label
```

Nun können die Nachrichten ausgewertet werden.

Beispiel: Auswertung der Nachrichten

```
score=0
for row in dataset:
  mylabel=naive_bayes_classifier(row[1])
  label=row[0]
  if mylabel==label:
    score+=1

print(dataset)
print(score)
print(score/count_dataset*100)
# Wie viel Prozent der Nachrichten werden richtig erkannt?
```

In diesem Fall erzielen wir bei der Klassifizierung eine Richtigkeit von ca. 89

7.2 Score

Um ein Overfitting, also das Verwenden zu vieler Features, zu vermeiden, kann noch ein Score berechnet werden. Dieser wird folgendermaßen berchnet:

$$R = \frac{\text{richtig klassifizierte Mails}}{\text{Anzahl aller Mails}} \tag{26}$$

$$F = \text{Anzahl der Features} \tag{27}$$

$$Score = -2\log(R) + F\log(\text{Anzahl aller Mails}) \tag{28}$$

Allgemein sollte der Score einen Wert von ca. 100 annehmen, um noch immer eine gute Auswertung der Mails zu garantieren, aber das Verwenden zu vieler Features zu vermeiden. Wir erreichten einen Score von etwa 120.

8 Logistische Regression

Während Naive-Bayes-Klassifizierung Bernoulli-Variablen (Wahr/Falsch) verwendet, können mittels der logistischen Regression auch Features aus $\mathbb{R}$ verwendet werden. Beispielsweise kann nun anstatt des binären Features "Befindet sich das Wort 'FREE' im Satz?" das Feature "Wie oft befindet sich das Wort 'FREE' im Satz?" eingesetzt werden. Somit können mehr und genauere Daten an das Modell übergeben werden, was es in der Regel dynamischer und vielseitiger macht. Mit den richtigen Indikatoren (Wortanzahlen sowie Doppelziffer-RegEx) konnte sogar ca. 98.5% Test-Genauigkeit erzielt werden. Allgemein wird von einer Funktion der Form

$$\mathbb{P}(Spam|F_1, F_2, ..., F_n) = f(w_1F_1 + w_2F_2 + ... + C)$$

ausgegangen, wobei $a, b, ...$ als Gewichtungen (Kalibrierungswerte, engl. "Weights") der einzelnen Features und C als Bias-Term gesehen werden können. f muss als Wahrscheinlichkeit auf $0 \leq f(x) \leq 1$ begrenzt sein. Außerdem muss $f(x)$ zweimal differenzierbar sein, um später das Newton-Verfahren einsetzen zu können, sowie streng monoton sein, unter anderem, um einen einzelnen Entscheidungspunkt (generell bei $f(0) = 0.5$) zu gewährleisten.

8.1 Herleitung von $f(x)$ bei einem einzigen Indikator

Nehmen wir einen einfachen Fall mit einem einzigen Indikator an:

$$\mathbb{P}(Spam|F) = f(aF + b)$$

Durch Umformung lässt sich eine Ähnlichkeit zu Naive-Bayes erkennen:

$$f^{-1}(\mathbb{P}(Spam|F)) = b + aF = bias + weight \cdot F$$

$$logit(\mathbb{P}(Spam|F)) = bias + score \cdot F$$

$logit^{-1}$ scheint also ein mögliches f zu sein. Allgemein wird dies als Sigmoid-Funktion $\sigma(x)$ verwendet:

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

8.1.1 Beweis der Sigmoid-Umkehrfunktion

Wie vorhin angenommen gleicht die Umkehrfunktion der Sigmoid-Funktion dem *logit*. Das lässt sich wie folgt beweisen:

Die Sigmoid-Funktion wird mit y gleichgesetzt und anschließend die Umkehrfunktion der beiden gebildet:

$$y = \frac{1}{1 + e^{-x}}$$

$$y^{-1} = \left(\frac{1}{1 + e^{-x}}\right)^{-1} \tag{29}$$

Durch umformen auf x ergibt sich:

$$
\begin{aligned}
x &= -\ln\left(\frac{1}{y} - 1\right) \\
x &= \ln\left(\left(\frac{1}{y} - 1\right)^{-1}\right) \\
x &= \ln\left(\frac{1}{\frac{1}{y} - 1}\right) \\
x &= \ln\left(\frac{y}{1-y}\right) \qquad (30)
\end{aligned}
$$

Dadurch ist bewiesen, dass die Umkehrfunktion der Sigmoid-Funktion dem *logit* gleicht.

8.1.2 Alternativen für $f(x)$ anstatt $\sigma(x)$

Funktionen, die $\sigma(x)$ ähnlich sind, wie z.B.

$$f(x) = \frac{tan^{-1}(x)}{\pi} + \frac{1}{2}$$

können auch als $f(x)$ eingesetzt werden, liefern aber generell leicht (wenige Prozent) schlechtere Ergebnisse.

8.2 Bedeutung/Notwendigkeit des C-Terms

Der Bias-Term C "verschiebt" $f(x)$. Wären also beispielsweise alle Features $F_n = 0$, würde er "alleine" über das Ergebnis entscheiden. Aus Implementierungssicht ist es jedoch einfacher, C als Feature F_C mit konstantem Wert $F_C = 1$ sowie einer Gewichtung $w_C = C$ einzubauen. Da bei der Kalibrierung diese Gewichtungen angepasst werden, können so auch Algorithmen (siehe unten) "selbst" entscheiden, welchen Wert C annehmen soll. In der Praxis stellt sich oft heraus, dass Algorithmen F_C eine sehr geringe Gewichtung zuordnen ($w_C < 0.05$, während viele andere Features bei $w_F > 10$ liegen), d.h., es als unbedeutsam sehen, sofern passende Feature-Offsets gegeben sind (siehe 8.4).

8.2.1 Händische Berechnung von C

Für die Annahme das nur ein Feature und eine Beobachtung berechnet wird, kann auch händisch auf den Bias-Term C umgeformt werden. Dafür gehen wir von folgender Form aus:

$$\mathbb{P}(Spam|F) = \sigma(F + C)$$

Diese Form lässt sich ähnlich wie im vorherigen Beweis (siehe. 3.2) lösen. Er unterscheidet sich in

$$\mathbb{P}(x = 1) = p$$
$$\mathbb{P}(x = 0) = 1 - p$$

was nun wie folgt berechnet wird:

$$\mathbb{P}(x = 1) = \sigma(F + b)$$
$$\mathbb{P}(x = 0) = 1 - \sigma(F + b)$$

berechnet wird.

Dadurch lässt der Term durch Ableiten nach C wie folgt darstellen:

$$y_1 - \sigma(F + C) = 0$$

Daraus folgt:

$$C = \sigma^{-1}(y_1) - F$$

bzw.

$$C = logit(y_1) - F$$

8.3 Interpretation von Gewichtungen

Allgemein gilt für Features (mit $F > 0$): Positive Gewichtungen ziehen den Ausgabewert nach oben, negative den Ausgabewert nach unten. Bei $w = 0$ wird ein Feature völlig ignoriert. Ein Beispiel: Durch iterative Anpassung wurde dem Feature "Wie oft kommt 'CLAIM' im Satz vor? - 1" eine Gewichtung von ca. 66 zugeordnet (Spam-Indiz), dem selben Feature, jedoch wurde mit dem Wort "ME" eine Gewichtung von ca. -98 (Ham-Indiz) zugeordnet.

8.4 Feature-Offsets

Man gehe von einem Wortanzahl-Feature F ("Wie oft kommt das Wort X im Satz vor?") aus. Kommt das Wort "PRIZE" in einem Satz nicht vor, lässt sich dies als Ham-Indiz sehen. Im Modell jedoch würde, da $F = 0$, dieses Indiz nicht berücksichtigt werden. Solche Probleme können bei jedem F auftreten, dass nur positiv oder negative (und Nullwerte) Werte annimmt. Eine mögliche sowie einfach einzubauende Lösung ist es, diesem Feature eine Konstante zu addieren bzw. subtrahieren: $F_{total} = F_{actual} + k$ Bei einem Wortanzahl-Feature eines Spam-Worts mit $k = -1$ würde es also bedeuten, dass $F_{actual} = 0$ Vorkommen als Ham-Indiz ($F_{total} = -1$), $F_{actual} \geq 2$ als Spam-Indiz ($F_{total} \geq 1$) vom Modell interpretiert werden. In unserer Implementierung wurden in solchen Fälen hauptsächlich $k \in \{-2, -1, -\frac{1}{2}, 0, 1\}$ verwendet. k kann auch als zusätzlicher C-Term gesehen werden:

$$\mathbb{P}(Spam|F_1, ...) = f(w_1F_1 + ... + C) = f(w_1 \cdot (F_{1_{actual}} + k) + ... + C) \quad (31)$$
$$= f(w_1F_{1_{actual}} + ... + (C + w_1k)) \quad (32)$$

8.5 Algorithmen zur Gewichtung

Durch die Verwandschaft zum Naive-Bayes können die Indikatoren-Scores von ähnlichen Indikatoren fast direkt in die Gewichtungen übertragen werden, und durchaus passable, wenn auch suboptimale Ergebnisse liefern. Durch ein grundlegendes Verständnis des Systems können die Gewichtungen auch (theoretisch) per Hand angepasst werden, was teils Ergebnisse über 88% liefern konnte. Dies liefert jedoch schlechtere Ergebnisse als algorithmisch Gewichtungs-Angepassung.

8.5.1 Perceptron-Algorithmus

Das Perceptron (bereits 1943 von Warren McCulloch und Walter Pitts erfunden) bietet einen Algorithmus zur automatischen Gewichtung. Der Einfachkeit halber werden die n-dimensionalen Vektoren F (Features) und w (Gewichtungen) definiert.

$$F = \begin{pmatrix} F_1 \\ F_2 \\ \dots \\ F_n \end{pmatrix} \qquad w = \begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_n \end{pmatrix}$$

Zum Anpassen ("trainieren") der Gewichtungen wird nun Folgendes für jede Trainings-Datenzeile mit dem erwünschten Ergebnis $\mathbb{P}_{goal}$ (z.B. bei Ham $\mathbb{P}_{goal} = 0$, bei Spam $\mathbb{P}_{goal} = 1$) ausgeführt. Eine Datenzeile wird auf die oben beschriebene Art (hier mit Skalarprodukt) auf die Wahrscheinlichkeit, dass das gesuchte Attribut (z.B. Ham/Spam) zutrifft:

$$\mathbb{P}_{out} = \sigma(F \cdot w)$$

Der Unterschied ("Fehler", engl. "error") zwischen tatsächlich erreichtem und erwünschten Ergebnis, den es zu minimieren gilt, wird berechnet:

$$error = \mathbb{P}_{goal} - \mathbb{P}_{out}$$

Die neuen Gewichtungen w_{next}, welche in der nächsten Iteration eingesetzt werden, werden folgendermaßen angepasst:

$$w_{next} = w + F \cdot error \cdot r$$

Hier ist r ein "Lernfaktor" (engl. "learning rate"), der die Anpassung verlangsamt, um Überanpassung zu vermeiden, da diese beispielsweise unerwünschte Oszillationen zur Folge haben kann. In unserem Fall haben wir $r = 0.1$ verwendet. Der Anpassungsalgorithmus kann auch intuitiv erklärt werden: Bei großen Fehlern wird stark angepasst, bei kleinen nur leicht und ist der berechnete Wert zu klein, müssen die Gewichtungen erhöht werden (und umgekehrt), wobei aber auch das Vorzeichen von F_n beachtet werden muss.

Hier die verwendete Python-Implementierung:

```
1  class Perceptron():
2      inputs  = [] # Features (als Funktionen/Lambdas)
3      weights = [] # Gewichtungen
4      bias    = 0  # C-Term (ist ein Feature "F_C = lambda x: 1" vorhanden
       , kann dies ausgelassen werden)
5      rate    = 0.1 # Learning Rate
6
7      def __init__(self, inputs):
8          self.inputs = copy.copy(inputs)
9          self.weights = np.zeros(len(self.inputs)) # Alternativ: np.
       random.rand
10
11     def output(self, content):
12         # Einfaches Skalarprodukt
```

```
        s = self.bias + sum([i(content) * w for i, w in zip(self.inputs,
    self.weights)])

        return sigmoid(s)

    def train_once(self, content, expected):
        inps = []
        for i in self.inputs:
            inps.append(i(content))

        real = sigmoid(np.dot(inps, self.weights))

        error = expected - real

        for idx, i in enumerate(inps):
            self.weights[idx] += error * i * self.rate

        return real
```

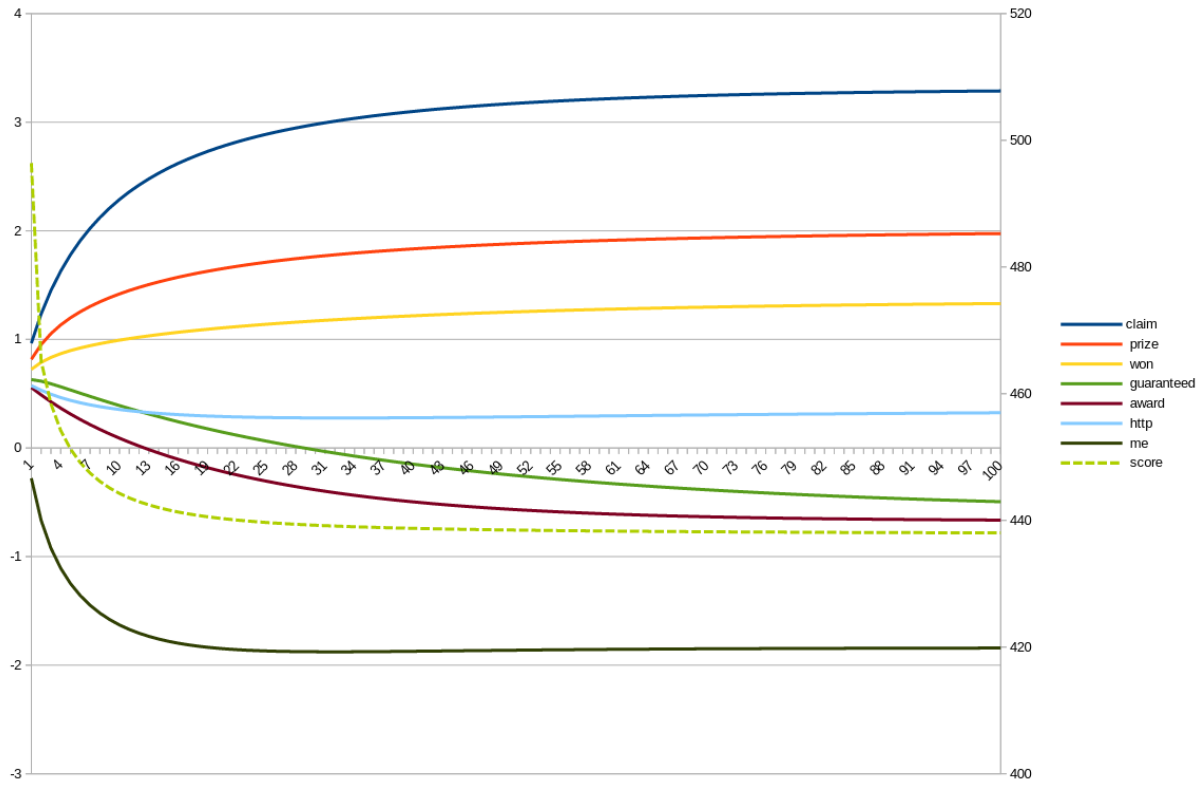


Abbildung 5: Anpassung von Wortanzahl-Feature-Weights über 100 Iterationen mit niedrigem $r = 0.01$, sowie Summe der quadrierten Fehler (rechte Skala)

8.6 Neuronale Netzwerke

Die soeben erwähnten Inhalte bilden die Grundlage für sogenannte neuronale Netzwerke, in welchen Neuronen über Gewichtungen miteinander verbunden sind, und das selbe Skalarprodukt verwendet wird, um die sogenannte Aktivierung eines Neurons auszurechnen. Diese Aktivierung kann dann als Eingabe (Feature) für einen weiteren Neuronen verwendet werden. Es können so also mehrere Neuronen beliebig vernetzt werden, wodurch

für Eingaben beliebig komplexe Ausgaben erreicht werden können. Solche Netze können für Mustererkennung (Bilderkennung, auch Spam-Erkennung!), aber auch für Bildgenerierung verwendet werden. Hierbei sind jedoch auch komplexere Trainings-Algorithmen notwendig.

9 Fake News

Zur Klassifizierung von Fake-News wurde ebenfalls der Naive-Bayes Algorithmus verwendet. Wichtige Features waren in diesem Fall die Wörter “21wire”, “quot” und “rohingya”. Beim Analysieren von Trainingsdaten für den Fake-News-Klassifizierer ergaben sich zudem einige Probleme. In den frei zugänglichen Datasets, welche wir ausgesucht hatten, waren die Artikel oftmals länger als die maximale Zellengröße. Zwar konnte dieses Problem in Python mithilfe kleinerer Anpassungen schnell behoben werden, jedoch blieb das Problem, dass das Programm lange Rechenzeiten benötigte, bestehen.

	Naive Bayes	Logistische Regression
Benötigte Zeit	56s	77s

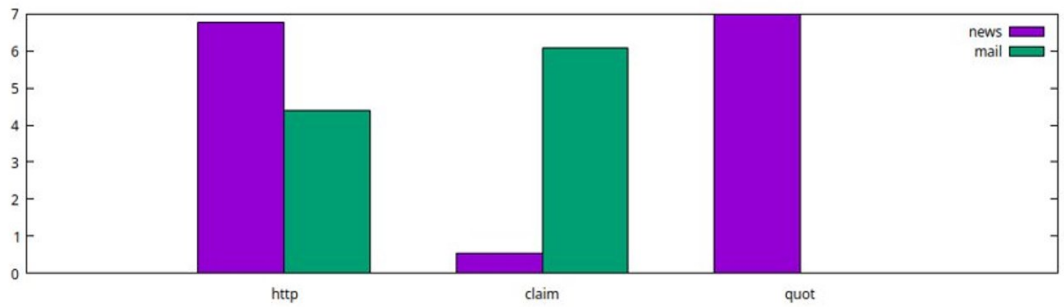
9.1 Kompatibilität mit dem Spam-Mail Klassifizierer

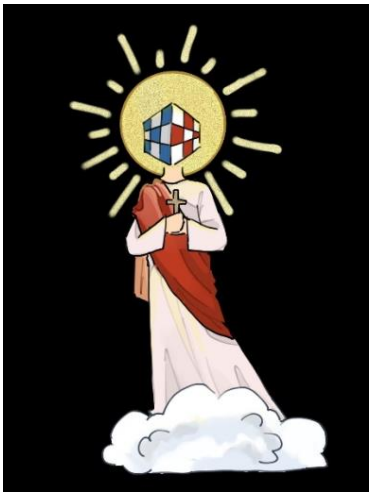
Einige weitere Feststellungen konnten wir auch machen, als wir versuchten, Spam-Filter und Fake-News-Filter zu kombinieren. Der Naive-Bayes Algorithmus war zwar in der Lage, Spam- und Ham-Mails, sowie Fake- und normale News beziehungsweise News und Mails generell zu unterscheiden, jedoch sank die Erfolgsrate beim Einteilen von “erwünschten” und “unerwünschten” Texten (mit “unerwünscht” sind in diesem Fall Fake-News sowie Spam-Mails gemeint) fast auf das Niveau einer reinen Zufallsentscheidung. Das liegt daran, dass sich News und Textnachrichten kaum Indikatoren teilen. Der einzige, zumindest teilweise aussagekräftige Indikator war der Term “http”, welcher auf das Vorhandensein eines Weblinks hinweist. Jedoch war dieser Indikator nicht aussagekräftig genug, um Texte sicher in “erwünscht” und “unerwünscht” einzuteilen.

Die folgende Grafik zeigt, dass die Top-Indikatoren aus Mails nicht gut für News-Klassifizierung und umgekehrt geeignet sind. Die Aussagekraft der Indikatoren wurde mit der Formel

$$|\ln \frac{\mathbb{P}(F|Spam)}{\mathbb{P}(F|Ham)}|$$

berechnet. Man kann sehen, dass die beiden Top-Indikatoren (“claim” im Fall von Mails und “quot” im Fall von News) für die jeweils andere Kategorie nicht gut geeignet sind. Tatsächlich kommen die besten Indikatoren aus dem News-Datenset (“rohingya”, ‘21wire”, “tco” und “rakhine”) im Mail-Datensatz überhaupt nicht vor.





Kombinatorik & Algorithmik

(Optimales) Lösen von Puzzles
(Der Gottes-Algorithmus)

Inhaltsverzeichnis

Einleitung

Einen Rubik's Cube zu lösen ist die eine Kunst, ihn mit der geringstmöglichen Anzahl an Zügen lösen, die Andere. Um dies auch möglichst effizient zu tun, muss man sich gute Strategien überlegen. Ein 3x3 Rubik's Cube hat so viele verschiedene Stellungen, die man durch Drehungen erreichen kann, dass ein Programm, welches ihn naiv probiert auf die schnellste Art und Weise zu lösen, den Speicher und die Rechnungsleistung unserer einfachen Rechner schnell überschreitet. Deshalb war es für unser Vorhaben notwendig, eine Stufe kleiner zu denken: Jede Position eines 2x2 Rubik's Cube mit so wenig Zügen wie möglich zu lösen. Zuerst mit einem vorgegebenen Datensatz und dann ohne. Um dies zu Bewerkstelligen bietet es sich an auf bekannte Algorithmen der Graphentheorie zurückzugreifen. Wenn man aber keinen Computer zu Hand hat, gibt es eine Alternative, zu der man greifen kann: Die Gruppentheorie. Mithilfe von Konjugationen und Kommutatoren kann man sich selbst mit Stift und einem Blatt Papier Lösungsstrategien ausdenken. Kein leichtes Unterfangen, wie wir nach der Theorie gemerkt haben.

1 Der Gottes-Algorithmus eines Rubiks Cubes

Ein 2x2 Rubik's Cube besitzt 8 Steine und 3 verschiedene Rotationsachsen, womit sich bereits eine Vielzahl möglicher Stellungen ergibt. Wie viele mögliche Positionen es genau sind, werden wir später aufzeigen. Ein Algorithmus, der jede Position mit der geringstmöglichen Anzahl an Zügen lösen kann, wird als Gottes-Algorithmus bezeichnet, und die Gottes-Zahl gibt an, wie viele Züge (=Drehungen) maximal gebraucht werden, um jede beliebige Stellung des Würfels lösen zu können.

1.1 Die Idee für die Implementation

Der erste Schritt zur Entwicklung eines Gottes-Algorithmus, ist es sich zu überlegen, wie viele verschiedene mögliche Stellungen es bei einem Zauberwürfel überhaupt gibt und wie man diese darstellt. Ein 2x2 Rubik's Cube besteht aus 8 verschiedenen (Eck-)Steinen. Die Eigenheit dieses Würfels ist, dass man einen Stein immer als gelöst ansehen kann, da es keine fixierten Mittelsteine gibt. In Folge dessen können nur mehr 7 Steine positioniert werden. Einer von diesen kann auf 7 verschiedenen Stellen positioniert werden, der nächste nur mehr auf 6 und so weiter. In der Mathematik kann man das mit 7! = 7x6x5x4x3x2x1 darstellen. Zusätzlich besitzen diese Steine eine Orientierung. Um diese zu beschreiben, betrachteten wir die gelben und weißen Seiten der einzelnen Steine und definierten oben und unten. Zeigt die gelbe oder weiße Seite eines Steins oben beziehungsweise unten, ist ihre Orientierung 0. Ist die Seite im Uhrzeigersinn um eins weiter gedreht ist die Orientierung 1. Mit demselben Prinzip folgt Orientierung 2 und dann geht es wieder von vorne los. Man könnte nun meinen, dass die Steine 3^7 verschiedene Möglichkeiten bei den Orientierungen besitzen. Das Problem ist aber, dass die Orientierung der Steine immer eine durch drei teilbare Zahl sein muss. Deshalb sind nur 36 verschiedene Orientierungen möglich und der eine Stein, der überbleibt muss so angepasst werden, dass die Orientierung durch drei teilbar ist. Also ist die Anzahl möglicher Stellungen bei einem 2x2 Rubik's Cube 7! * 3^6.

```
function res = solutionList()
  position = [1:8, repelem(0, 8)]

  max = factorial(7) * 3^6;
  res = ones(1, max) .* (-1);

  res(1) = 0;
  set = 1;

  current = [position];

  depth = 1;
  while (set < max)
    depth
    newcurrent = [];
    for a = 1:9
      c = current;
      new = performMove(c, a);
      val = positionToIndex(new);

      new = new(res(val) == -1, :);
      val = val(res(val) == -1);

      newcurrent = [newcurrent; new];

      res(val) = depth;
      set += length(val);
    endfor
    current = newcurrent;
    ++depth;
  endwhile
endfunction
```

1.2 Datensätze

Der nötige Datensatz mit den Tiefen aller Stellungen für den 2x2 Rubik's Cube kann leicht mit Hilfe einer Breitensuche[1] erstellt werden. Dafür geht man von der gelösten Position aus bis alle Stellungen einmal erreicht wurden. Der Code rechts generiert eine solche Liste.

[1] Siehe Kapitel 3.2.1.

1.3 Speicherbedarf

Möchte man die nötige Tiefe jeder Stellung eines 2x2 Würfels abspeichern, so muss für jede der $7! * 3^6 = 3674160$ eine Zahl zwischen 0 und 11 abgespeichert werden, da die Gottes-Zahl für den 2x2 Würfel 11 beträgt. Jede solche Zahl kann mit 4 Bits gespeichert werden, somit ergibt sich insgesamt ein Speicheraufwand von 3674160*4 Bits = 14696640 Bits, was ca. 1.83 Megabyte entspricht. Für den üblichen 3x3 Zauberwürfel lautet die Gotteszahl 20 (mit 5 Bits speicherbar), und wie wir uns überlegt haben gibt es $8! * 3^7 + 12! * 2^{10} = 4.32 * 10^{19}$ mögliche Stellungen. Diese Zahl ergibt sich aus den 8! Permutationen der Ecksteine, den $12!$ Permutationen der Ecksteine, den 3^8 Möglichkeiten eine Ecke zu orientieren und den 2^{12} Möglichkeiten eine Kante zu orientieren. Allerdings sind durch die normalen Drehungen des Zauberwürfels nur ein Drittel der Orientierungen für die Ecken erreichbar, nur die Hälfte aller Kantenorientierungen und nur die Hälfte aller möglichen Permutationen.

Daraus würde sich ein Speicheraufwand von $2.16 * 10^{20}$ Bits ergeben, was umgerechnet ca. 27 Millionen TB entspricht. Aus diesen Überlegungen wird bereits klar, dass unsere Lösungsstrategie für den leichteren 2x2 Rubik's Cube sich nicht auf den 3x3 Rubik's Cube übertragen lassen kann.

1.4 Konvertierung der Positionen in einen Index

Wir haben zuerst einen 16-stelligen Code eingeführt, der jede mögliche Position beschreiben kann. Im nächsten Schritt wurde mithilfe dieses Codes jeder Würfelstellung eine eindeutige Zahl zwischen 1 und $7! * 3^6$ zugeordnet, womit unter anderem der Speicheraufwand reduziert werden kann. Diese Zahl wurde weiteres benötigt für den Gebrauch einer Tabelle, die sowohl die umgewandelte Zahl, als auch die genaue Anzahl der verbleibenden Züge beinhaltete.

1.4.1 Positionen für den Computer leserlich machen

Um den Index zu berechnen, muss man zunächst verstehen, wie er aufgebaut ist. Die ersten acht Zahlen dienen der allgemeinen Stellung, in der sich der Würfel befindet (Permutationen der Steine). Jede der acht Zahlen steht für eine der acht kleineren Steine, in die ein 2x2 Würfel unterteilt ist und jede der acht Positionen in dem 16-stelligen Code dient einer Position die so ein kleinerer Stein einnehmen kann. Man zählt diese Steine beginnend von links hinten oben im Urzeigersinn ab und startet dann an der links hinteren unteren Position wieder. So würden die ersten 8 Ziffern der gelösten Stellung so aussehen: [1,2,3,4,5,6,7,8]. Wenn Stein 1 und Stein 3 vertauscht sind würde der dementsprechende Code dann so aussehen: [3,2,1,4,5,6,7,8].

Der letzte Teil des 16-stelligen Codes dient der Beschreibung der Rotation wie in Kapitel 1.1 bereits beschrieben wurde (Orientierung der Steine), und besteht somit aus 8 Zahlen zwischen 0 und 2.

Um einen dieser Codes in eine einzelne Zahl um zurechnen muss man aber noch die Reihenfolge verstehen. Man startet bei der Ausgangsposition (dem gelösten Würfel) diese hat den Code [1,2,3,4,5,6,7,8,0,0,0,0,0,0,0,0]. Danach geht man alle möglichen

Rotationen ab vom letzten Würfel startend. Danach verändert man die ersten 8 Zahlen so, dass sie die kleinstmögliche aber immer noch größere Zahl ergibt und dann startet alles von vorne.

Wie im Kapitel 1.1 bereits beschrieben, kann man sowohl die 5. als auch die 13. Position weg lassen weil der Würfel an dieser Ecke fixiert ist, wenn man die richtigen Drehungen ausführt. Danach hat man nur mehr 7 Zahlen, die etwas über die Position der Steine aussagen und 7 die etwas über die Drehung der einzelnen Steine aussagen.

1.4.2 Berechnung für den Index

Um den Index zu berechnen, muss man zunächst den Code wieder in 1. Und 2. Teil aufteilen.

Die ersten 7 Zahlen werden umgerechnet, indem man sich überlegt, wie viele Zahlenkombinationen von 7 Zahlen existieren, die kleiner sind als die zu Betrachtende. Wie aus unseren Überlegungen folgte, nimmt man dazu alle 7 Zahlen her, und überlegt sich für jede davon, wieviele Zahlen rechts im Code kleiner sind, und multipliziert diese mit der Fakultät der Anzahl an Zahlen die noch rechts im Code kommen (für erste Stelle im Code mit 6!, weil es soviele Permutation der rechts liegenden Zahlen gibt, für die zweite Stelle mit 5!, usw.).

Beispiel: 6547132 → (5*6! + 4*5! + 3*4! + 3*3! + 0*2! + 1*1!) = 4171

Dazu nimmt man die erste Zahl verringert sie um 1 und auch die Anzahl der Zahlen die kleiner sind als die Jeweilige und schon in einer früheren Position vorgekommen ist. Danach multipliziert man sie mit n!, wobei n die Anzahl der Zahlen ist, die nach der jeweiligen kommen. Wenn man das für jede Zahl gemacht hat, muss man diese Ergebnisse addieren und zum S. Das endgültige Ergebnis nennen wir den Index der Permutation.

Der zweite Teil wird um einiges einfacher berechnet. Nämlich nimmt man die letzte Zahl des Codes und multipliziert sie mit 3^0 dann die vorletzte mit 3^1 und immer so weiter bis man den 2. Teil des Codes abgeschlossen hat. All diese Zahlen die man für den 2. Teil bekommt addiert man wieder. Diese Zahl bezeichnen wir als Index der Orientierung. Ein möglicher Code dazu sieht so aus:

```
function index = position_to_index(position)

  permutations = position(1:8);
  permutations(5) = [];
  permutations(permutations>5) = permutations(permutations>5) - 1;

  smaller_numbers = 0:7;

  indexp = 0;
  for i = 1:7
    indexp = indexp + factorial(7-i)*smaller_numbers(permutations(i));
    smaller_numbers((permutations(i)+1):8) = smaller_numbers((permutations(i)+1):8) - 1;
  endfor
  indexo = 0;

  position([13,16]) = [];
  for i = 9:14
    indexo = indexo + 3^(14-i)*position(i);
  endfor
  index = 1 + indexo + 3^6*indexp;
```

Um aus dem Index der Permutation und dem Index der Orientierung einen einzelnen Index zu berechnen wurde der Index der Permutation mit 3^6 multipliziert, weil es so viele Möglichkeiten der Rotationen der kleineren Würfel gibt, und danach mit dem Index der Orientierung addiert. Schließlich wurde der Index um 1 erhöht, um die ersten Code der gelösten Stellung nicht mit 0, sondern mit 1 identifizieren zu können.

1.5 Permutationen und drehen eines Würfels

Einen gewöhnlichen 2x2 Zauberwürfel kann man auf 3 verschiedene Arten drehen: vorne(f), rechts(r) und oben(u). Mit diesen 3 Zügen kommt man wirklich auf alle Stellungen des Würfels, aber wenn man zum Beispiel rechts gegen den Uhrzeigersinn drehen will, müsste man 3x rechts drehen, deswegen hat man dafür eigene Zeichen eingeführt: Für vorne in die andere Richtung (f´), 2x vorne (2f) und analog für die anderen Drehungen.

Wenn man sich überlegt, was so ein Zug zu der Stellung eigentlich macht, dann bemerkt man schnell spezielle Muster. Zum Beispiel bei einem „f“ Zug vertauscht man Stein 3,4,7,8 und die Rotation der Steine ändert sich. Jeder dieser Züge hat eine Zahl von 1-9 bekommen.

f	→	[1,2,4,8,5,6,3,7,0,0,1,2,0,0,2,1]	=	1
f´	→	[1,2,7,3,5,6,8,4,0,0,1,2,0,0,2,1]	=	2
2f	→	[1,2,8,7,5,6,4,3,0,0,0,0,0,0,0,0]	=	7
u	→	[4,1,2,3,5,6,7,8,0,0,0,0,0,0,0,0]	=	5
u´	→	[2,3,4,1,5,6,7,8,0,0,0,0,0,0,0,0]	=	6
2u	→	[3,4,1,2,5,6,7,8,0,0,0,0,0,0,0,0]	=	9
r	→	[1,3,7,4,5,2,6,8,0,1,2,0,0,2,1,0]	=	3

r´ → [1,6,2,4,5,7,3,8,0,1,2,0,0,2,1,0] = 4

2r → [1,7,6,4,5,3,2,8,0,0,0,0,0,0,0,0] = 8

Mit den Informationen kann man ein Programm schreiben, das einen eingegebenen Zug zu einer speziellen Stellung hinzufügt und die neue Stellung mit dem angewandten Zug berechnet. Jedoch kann es sein, dass man bei den Rotationen eine größere Zahl als 2 bekommt, wenn man die zahlen addiert, deswegen muss man immer modulo 3 rechnen, sodass nur der Rest übrig bleibt wenn man durch 3 dividiert. Ein entsprechendes Programm kann so aussehen:

```
function position_output = performCubemove(position_input, move)
  position_output = position_input;
  a = position_input(1);
  b = position_input(2);
  c = position_input(3);
  d = position_input(4);
  e = position_input(5);
  f = position_input(6);
  g = position_input(7);
  h = position_input(8);
  i = position_input(9);
  j = position_input(10);
  k = position_input(11);
  l = position_input(12);
  m = position_input(13);
  n = position_input(14);
  o = position_input(15);
  p = position_input(16);

  if(move == 1)
    position_output = [a,b,d,h,e,f,c,g,i,j,l+1,p+2,m,n,k+2,o+1];
    position_output(9:16) = mod(position_output(9:16),3);
  endif
  if(move == 2)
    position_output = [a,b,g,c,e,f,h,d,i,j,o+1, k+2, m, n, p+2,l+1];
    position_output(9:16) = mod(position_output(9:16),3);
  endif
```

```
28  if(move == 3)
29    position_output = [a,c,g,d,e,b,f,h,i,k+1,o+2,l,m,j+2,n+1,p];
30    position_output(9:16) = mod(position_output(9:16),3);
31  endif
32  if(move == 4)
33    position_output = [a,f,b,d,e,g,c,h,i,n+1,j+2,l,m,o+2,k+1,p];
34    position_output(9:16) = mod(position_output(9:16),3);
35  endif
36  if(move == 5)
37    position_output = [d,a,b,c,e,f,g,h,l,i,j,k,m,n,o,p];
38    endif
39  if(move == 6)
40    position_output = [b,c,d,a,e,f,g,h,j,k,l,i,m,n,o,p];
41  endif
42  if (move == 7)
43    position_output = [a,b,h,g,e,f,d,c,i,j,p,o,m,n,l,k];
44  endif
45  if (move == 8)
46    position_output = [a,g,f,d,e,c,b,h,i,o,n,l,m,k,j,p];
47  endif
48  if (move == 9)
49    position_output = [c,d,a,b,e,f,g,h,k,l,i,j,m,n,o,p];
50  endif
51 endfunction
```

1.6 Vollständiger Algorithmus

Wenn man die Programme geschrieben hat, in denen man die Codes der Position in einen Index umrechnen kann, man Züge zu einer Position hinzufügen kann und die vollständige Liste mit Indexposition und Anzahl der noch zu machenden Züge vorhanden ist, steht einem nichts mehr im Wege, wenn man einen funktionsfähigen optimalen 2x2 Cube Löser schreiben will. Der entsprechende Datensatz wurde dazu im ersten Schritt vorgegeben.

Die Idee ist, dass man die zu lösende Position eingibt und das Programm diese Position umrechnet in einen Index. Mit Hilfe dieses Indexes und der Liste kann man die Anzahl der Züge herausfinden, die noch zu machen sind um den Würfel zu lösen. Danach wendet das Programm jede Drehung an und wandelt diese Permutationen in Indexe um und schaut, ob die Anzahl der Züge , die noch zu machen sind um eines kleiner geworden ist. Wenn das der Fall ist, dann merkt sich das Programm den Zug und schreibt ihn in die Lösung. Danach fängt alles ausgehend von der neu berechneten Position wieder von vorne an.

Am Ende zeigt das Programm die genauen Züge, die zu Machen sind, um den Würfel zu lösen. Das Programm sieht so aus:

```
function solution = solveCube(position, optVec)
  k = 0;
  depth0 = optVec(position_to_index(position));
  solution = "";
  depth = depth0;

  for i = 1:depth0
    optVec(position_to_index(position))
    if (optVec(position_to_index(performCubemove(position, 1))) == (optVec(position_to_index(position)) -1))
      position = performCubemove(position, 1);
      solution = [solution, "f"];
    elseif (optVec(position_to_index(performCubemove(position, 2))) == (optVec(position_to_index(position)) -1))
      position = performCubemove(position, 2);
      solution = [solution, "f'"];
    elseif(optVec(position_to_index(performCubemove(position, 3))) == (optVec(position_to_index(position)) -1))
      position = performCubemove(position, 3);
      solution = [solution, "r"];
    elseif (optVec(position_to_index(performCubemove(position, 4))) == (optVec(position_to_index(position)) -1))
      position = performCubemove(position, 4);
      solution = [solution, "r'"];
    elseif (optVec(position_to_index(performCubemove(position, 5))) == (optVec(position_to_index(position)) -1))
      position = performCubemove(position, 5);
      solution = [solution, "u"];
    elseif (optVec(position_to_index(performCubemove(position, 6))) == (optVec(position_to_index(position)) -1))
      position = performCubemove(position, 6);
      solution = [solution, "u'"];
    elseif (optVec(position_to_index(performCubemove(position, 7))) == (optVec(position_to_index(position)) -1))
      position = performCubemove(position, 7);
      solution = [solution, "ff"];
    elseif (optVec(position_to_index(performCubemove(position, 8))) == (optVec(position_to_index(position)) -1))
      position = performCubemove(position, 8);
      solution = [solution, "rr"];
    elseif (optVec(position_to_index(performCubemove(position, 9))) == (optVec(position_to_index(position)) -1))
      position = performCubemove(position, 9);
      solution = [solution, "uu"];
    endif

  endfor
endfunction
```

2 Der Cube als Gruppe

Den Zauberwürfel können wir als Gruppe ansehen. In dieser Gruppe sind alle Möglichen Zugkombinationen aufgelistet. Mit Hilfe von Gruppen können wir Algorithmen ableiten und selbst kreieren.

2.1 Gruppen

Was sind eigentlich Gruppen?
Sei G eine beliebige Menge und ° eine Operation °: $GxG \rightarrow G$.
Wir nennen $(G, °)$ eine Gruppe falls folgende drei Eigenschaften erfüllt sind:

1. Assoziativität:

$$\text{Für alle } a, b, c \in G$$
$$(a°b)°c = a°(b°c)$$

2. Es existiert ein Neutrales Element:

$$\text{Es gibt ein } e \in G \text{ sodass für alle } a \in G$$
$$a°e = e°a = a$$

3. Für alle $a \in G$ existiert ein Inverses Element:

$$\text{Für alle } a \in G \text{ gibt es ein } a^{-1} \in G \text{ sodass}$$
$$a°a^{-1} = a^{-1}°a = e$$

Eine Gruppe ist endlich, wenn $|G| < unendlich$ und abel'sch, wenn die Operation ° kommutativ ist (i.e. für alle $a, b \in G \vee a°b = b°a$).

Beispiele für Gruppen:

	unendlich	**endlich**
abel'sch	$(Z, +)$, $(Q\backslash\{0\}, *)$ $(R\backslash\{0\}, *)$	$(1, *), (0, +), (Z/nZ, *)$
nicht abel'sch	$(f: R \rightarrow R\ invertierbar,$ $hintereinander$ $ausführen)$	$Permutationen\ von\ der\ Menge$ $1 \rightarrow n, hintereinander$ $verknüpfen)$

Für Gruppen lassen sich viele allgemein gültige Aussagen treffen, ein paar Beispiele die wir kennengelernt haben:

1. Es existiert genau ein neutrales Element.
2. Für jedes $a \in G$ ist $a^{-1} \in G$ eindeutig bestimmt.
3. Wenn $|G| = p$ mit p ist eine Primzahl, dann ist G eine abel'sche Gruppe.
4. Wenn für jedes $a \in G$ gilt, dass $a°a = e$, dann ist G eine abel'sche Gruppe.
5. Für jedes $a \in G$ gilt $a^{ord(G)} = e$, wobei $ord(G) = G \vee$.

Für den Zauberwürfel haben wir also folgendes bestimmt:

$$G = MengeallerZugkombinationen.$$

Zugkombinationen, die die gleiche Endstellung des Zauberwürfels bewirken, fassen wir als das gleiche Element von G auf, um eine endliche Gruppe zu erhalten.
Das Neutrale Element $e = id$, wobei id die Zugkombination ist, bei der man keine Bewegung macht.
Die Operation ° ist die Ausführung dieser Zugkombinationen hintereinander.
G ist dann eine endliche Gruppe, da es nicht unendlich viele Stellungen des Würfels gibt.
G ist keine abel'sche Gruppe, da es wichtig ist, in welcher Reihenfolge zwei Zugfolgen ausgeführt werden.[2]

2.2 Untergruppen/Nebenklassen

Sei $(G, °)$ eine Gruppe und $H \subset G$, dann heißt H Untergruppe von G, falls $(H, °)$ eine Gruppe ist.

→ Satz von Lagrange: $ord(H)$ ist immer ein Teiler von $ord(G)$.

Beispiele für Untergruppen:

Gruppe	**Untergruppe**
$(Z, +)$	$(\{0\}, +), (\{2Z\}, +), (\{kZ\}, +)$
$(Q\backslash\{0\}, *)$	$(\{1\}, *), (\{2^n \mid n \in Z\}, *),$ $(\{p^n \mid n \in \mathrm{Z}\}, +),$ $(\{g \mid g \in \mathrm{Q} \,\&\, \mathrm{g} > 0\}, *)$
(2x2 Zauberwürfel, °)	$(\{id\}, °), (\{\mathrm{R}, \mathrm{R2}, \mathrm{R}', id\}, °),$ $(\{id, \mathrm{m}\}, °)$ bei $\mathrm{m} ° \mathrm{m} = id,$ $(\{id, m, m^2\}, °)$ bei $\mathrm{m} ° \mathrm{m} ° \mathrm{m} = id$

Sei $(G, °)$ eine Gruppe und $(H, °)$ eine Untergruppe von G und $a \in G$ beliebig, dann nennen wir $aH = \{b \in G \vee b = a°h, h \in H\}$ eine Nebenklasse.[3]

Beispiel: $(G, °) = (\mathrm{Z}, +)$ 1. Nebenklasse: $3Z + 1$

$(\mathrm{H}, °) = (3\mathrm{Z}, +)$ 2. Nebenklasse: $3Z + 2$

2 Beweis: Sie können es leicht selbst austesten, z.B. ist R U nicht das gleiche wie U R.
3 Genau genommen handelt es sich um eine linke Nebenklasse, nur bei abel'schen Gruppen ist diese Unterscheidung von rechter und linker Nebenklasse nicht mehr notwendig.

Zwei Nebenklassen sind entweder ident oder haben kein einziges gleiches Element.
Jedes Element aus der Gruppe G ist entweder in der Untergruppe H oder in einer Nebenklasse aH.

Beim Zauberwürfel haben wir es so gesehen, dass wenn G die Menge aller Stellungen des Würfels ist, dann ist die Menge aller Permutationen jeder Stellung des Würfels eine Untergruppe von G und die Menge aller Orientierungen jeder Stellung eine Untergruppe von G ist.

2.3 Konjugation

Wenn man einen Algorithmus hat, der zum Beispiel zwei gegenüberliegende Seitenkanten (3x3 Zauberwürfel) vertauscht, aber die Steine die man vertauschen will, nicht in richtiger Position sind, kann man diese vorerst in Position bringen (Setup-Move), um dann den Algorithmus anzuwenden. Nach dem Algorithmus muss man den Stein wieder in die Ursprungsposition zurückbringen. Die Zugfolge die den Stein in Position bringt nenne wir g, den Algorithmus den wir anwenden h und die Zugfolge die den Stein zurück bringt g^{-1}.

Sei $g, h \in G$:

$g \circ h \circ g^{-1}$ („h konjugiert mit g“)

Dadurch kann man mit einem gegebenem Algorithmus neue herleiten und damit nicht nur ein kleines Teilproblem lösen, sondern gegebenenfalls gleich mehrere.

2.4 Kommutator

Der Kommutator gibt in gewisser Weise an, wie kommutativ zwei Zugfolgen sind.

Sei $g, h \in G$:

$$[g, h] = g \circ h \circ g^{-1} \circ h^{-1}$$

Wenn $[g, h] = e$, dann sind g und h kommutativ:

$$[g, h] = e = g \circ h \circ g^{-1} \circ h^{-1} \Leftrightarrow e \circ h = h = g \circ h \circ g^{-1}$$
$$\Leftrightarrow h \circ g = g \circ h$$

Damit kann man versuchen, Zugfolgen zu finden, die als Kommutator sehr nahe am neutralem Element sind und nur wenig am Würfel verändern.

Beispiel: Wir wollen in der oberen Ebene Steine austauschen. Wir positionieren die Steine so, dass in der oberen Ebene alle Steine gleich bleiben bis auf die, die wir neu positioniert haben und tauschen wollen – die unteren zwei Ebenen sind sehr wahrscheinlich zerstört, aber das macht nichts, denn nun verwenden wir eine Zugfolge die nur die obere Ebene beeinflusst und bringen danach die Steine wieder in die Ursprungsposition zurück – es ist ähnlich wie bei der Konjugation, nur dass wir Zugfolgen benutzen die annähernd kommutativ sind.

3 Der Cube als Graph

Den Zauberwürfel können wir auch als Graphen ansehen. In diesem Graph sind alle Möglichen Stellungen aufgelistet. Mit Hilfe von Graphen können wir auch Algorithmen für den Computer erstellen, die ohne Datensatz oder zumindest mit weniger Daten auskommen als die ursprüngliche Variante des Lösers.

3.1 Graphen

Ein Graph $G = (V, E)$ besteht aus der Menge von Knoten (V) und Kanten (E). Jede Kante in E ist ein Paar aus zwei Knoten das Graphen G.
Kanten *ungerichteter Graphen* sind lediglich Verbindungen $[1 - 2]$ während *gerichtete Graphen* eine zusätzliche Richtungsinformation $[1 \rightarrow 2]$ aufweisen.

Ungerichteter Graph:

$$V = \{1,2,2,3,4,5\}$$

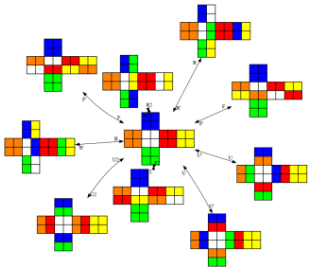
$$E = \{(1,4), (2,1), (3,4), (5,2), (3,1)\}$$

Ein *Pfad von A nach B* ist eine Folge von Knoten A, $V_1, V_2, V_3, \ldots, V_{n-1}, V_n$, B sodass $(A, V_1) \in E \wedge (V_1, V_2) \in E \wedge (V_2, V_3) \in E \wedge \ldots \wedge (V_{n-1}, V_n) \in E \wedge (V_n, B) \in E$ gilt.

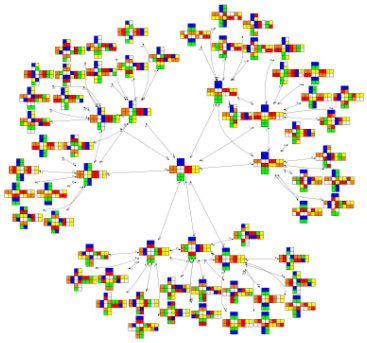
Ein Kreis in einem Graph beschreibt einen Pfad von A nach A ohne wiederholte Kanten.
z.B. $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$, Graphen ohne Kreise werden *Bäume* genannt.

Um den Rubik's-Cube als Graph darzustellen, werden als Knoten (V) die einzelnen Positionen und als Kanten (E) die möglichen Züge $\{R, R2, R', F, F2, F', U, U2, U'\}$ verwendet. Dieser Graph kann zum Lösen vom Rubik's-Cube abgewandert werden, wobei ein Pfad von der ungelösten zur gelösten Position gesucht wird.
Die Anzahl an Knoten steigt mit zunehmender Anzahl an Zügen / Tiefe exponentiell an.

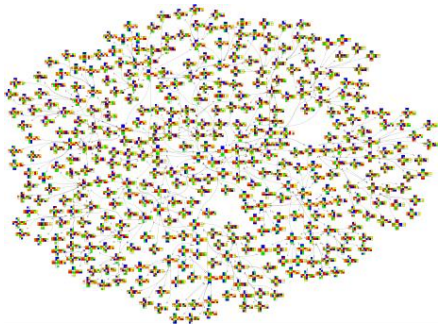
1 Zug Tiefe
(9 Knoten)



2 Züge Tiefe
(64 Knoten)



3 Züge Tiefe
(385 Knoten):



3.2 Suche mit Algorithmen

Gegeben sei ein zusammenhängender, ungerichteter und ungewichteter Graph (V, E), gesucht ein Pfad von einem Startknoten V_1 zu einem anderen Knoten V_2. Nun gibt es mehrere Ansätze diesen zu finden.

3.2.1 Breitensuche (Breadth-First Search)

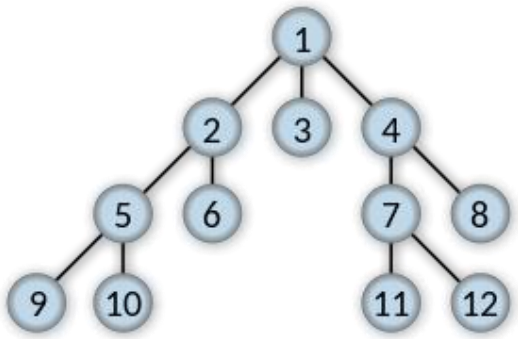
Hier werden zuerst von V_1 (Tiefe 0) alle Nachbarknoten, also diejenigen, die damit eine Kante teilen, besucht und gespeichert. Dafür kann sich oft die Datenstruktur einer Queue als nützlich erweisen, in die man Elemente "hinten" einfügt und "vorne" ausliest.

Sobald man also alle Knoten der Tiefe 1 besucht hat, geht es weiter: Ausgehend von jenen, wird wiederum jeweils jeder Nachbar besucht, und in die Datenstruktur hinzugefügt, währenddessen werden die vorherigen Knoten entfernt, sobald jeder der Angrenzenden gespeichert ist.

Nun sind also alle Pfade der Länge 2 von V_1 aus abgegangen worden. Dieser Prozess wiederholt sich (meist iterativ), als Nächstes sind die "Tiefe 3"-Knoten an der Reihe.

Da man von V_1 aus zuerst alle Knoten der Tiefe 0, dann 1, 2, 3, ... besucht, kennt man, sobald man auf V_2 stößt, sicher auch den kürzesten Pfad dorthin, beziehungsweise einen der kürzesten Pfade sollte es mehrere verschiedene Zugfolgen geben. (Gäbe es einen von geringerer Länge, hätte man ihn bereits gefunden).

Als ein weiterer Vorteil der Breitensuche gilt die Einfachheit der Implementierung. Das Kontra ist aber der große Speicherbedarf, da man sich in jedem Schritt eine Vielzahl an Knoten merken muss. Je nach den spezifischen Bedingungen kann man diesen Algorithmus noch verbessern, indem man gleichzeitig von V_1 und von V_2 aus eine BFS durchführt bis diese sich “in der Mitte treffen”. (Wie bei dem Meet-in-the-middle-Angriff aus der Kryptographie)



https://en.wikipedia.org/wiki/File:Breadth-first-tree.svg

3.2.2 Tiefensuche (Depth-First Search):

Im Gegensatz zur Breitensuche wird der Graph hier in erster Linie möglichst tief durchlaufen. Zuerst wird ein beliebiger Nachbar von V_1 besucht, dann von diesem Knoten aus (rekursiv) eine weitere Tiefensuche ausgeführt. Sobald alle Nachbarn dieses Knotens besucht worden sind (und bei diesen erneut eine DFS aufgerufen wurde) geht es zurück zum Startknoten.

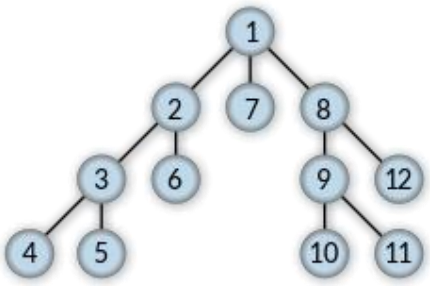
Von hier aus wird der nächste Nachbar von V_1 aufgerufen, bei diesem geschieht dasselbe. Das Ganze wiederholt sich, bis man auf den gesuchten Knoten stößt oder sämtliche Nachbarn des Startknotens besucht worden sind.

Dieser Algorithmus kann sich oft als sehr nützlich erweisen, handelt es sich beim Graphen allerdings nicht um einen Baum (zusammenhängender Graph mit genau |V| - 1 Kanten) können Probleme auftreten:

Gibt es mehrere Pfade von irgendeinem Knoten A zu Knoten B, ist es möglich, dass man in einen Kreis gerät, weshalb oft eine Maximaltiefe mitgegeben wird, bei der der Algorithmus, anstatt weiterzusuchen, abbricht und wieder zurückgeht. Außerdem findet man hier nicht mit Sicherheit zuerst den kürzesten Pfad, da es mehrere Wege geben kann, über die man von V_1 zu V_2 gelangt, und man sie nicht wie bei der Breitensuche der Tiefe nach durchläuft.

Der Vorteil daran ist allerdings, dass dieser Algorithmus in den meisten Fällen im Gegensatz zu Obigem nicht allzu vielen Speichers bedarf, da man nicht die gesamten Knoten einer gewissen Tiefe abspeichern muss, sondern nur den Pfad, auf dem man sich momentan befindet.

```
function [solution, target_found] = TreeSearch(knot_s, depth, knot_t, lastM = "0")
  moves = ["F"; "F2"; "F'"; "R"; "R2"; "R'"; "U"; "U2"; "U'"];
  target_found = 0;
  solution = "";

  if isequal(knot_s, knot_t)
    target_found = 1;
  else
    if (depth != 0)
      for m = 1:size(moves)(1)
        if (floor((lastM - 1)/3) == floor((m - 1)/3))
          continue;
        endif

        [position, orientation] = perform_move(knot_s(1:8), knot_s(9:end), moves(m,:));
        knot_new = [position, orientation];

        [solution_, target_found_] = TreeSearch(knot_new, depth - 1, knot_t, m);

        if !isequal(solution_, "") || target_found_ == 1
          solution = [moves(m,:), " ", solution_];
        endif

        if (target_found_ == 1)
          target_found = 1;
          break;
        endif

      endfor
    endif
  endif
endfunction
```



https://en.wikipedia.org/wiki/Depth-first_search

3.2.3 Iterative Tiefensuche (Iterative Deepening Search)

Bei diesem Algorithmus werden die Vorteile von Breiten- und Tiefensuche kombiniert, allerdings ist er etwas zeitaufwändiger als die beiden Anderen. Zuerst wird eine Tiefensuche mit Maximaltiefe 1 ausgeführt, dann 2, usw.

Im Gegensatz zur Breitensuche muss man hier nicht jeweils alle Knoten einer bestimmten Tiefe abspeichern, da der Algorithmus mit jedem Schritt wieder beim Startknoten anfängt, was ihn aber auch verlangsamt. Dafür findet dieser Algorithmus wie bei der Breitensuche mit Sicherheit zuerst den (beziehungsweise einen) kürzesten Pfad.

Diese Ideen können zum Beispiel von Nutzen sein, wenn man den Rubiks-Cube so als Graphen betrachtet, dass V_1 die aktuelle Permutation ist und V_2 die Ausgangsstellung. Dann helfen sie, eine bzw. die optimale Folge an Kanten (Züge am Würfel) zu finden, um V_2 zu erreichen (den Würfel zu lösen).

Sind zusätzlich Heuristiken bekannt, wie viele Züge man von einer gegebenen Stellung aus noch mindestens benötigt, um zur Zielstellung zu gelangen, kann der Algorithmus signifikant beschleunigt werden. Man spricht dann von dem sogenannten Iterative Deepening A* Algorithmus.

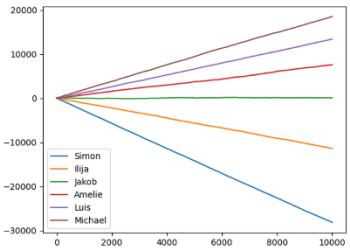
Rating Systeme

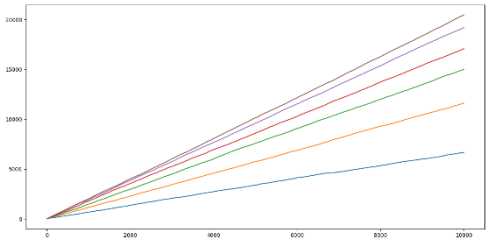
PROTOKOLL

Amelie Lerch, Luis Nievoll, Simon König, Jakob Tegshee, Ilija Novak, Michael Fischer

Sonntag, 07.01.2024

- In Python ein Skript geschrieben
  - Skillpunkte basierend, Charakterliste
    - 1-6
  - Jeder spielt gegen jeden - Ausgänge computergeneriert
    - Loopen lassen, gesehen, dass es mit Wahrscheinlichkeiten zusammenpasst
- Problem: Stärke ist nicht Rating
  - Skill nicht updaten, Rating schon -> aufsplitten
- Mathematische Formel aufstellen
  - Generalisieren – machbar machen für andere Beispiele, Zahlen
- Während Mittagspause: 10000 Durchgänge machen lassen





- 1,0;1,1;1,2;1,3 etc. anschauen vergleichen
  - Draufgekommen, dass Winkel sich nichts bringen
  - Geplottet in Diagramme
- Neues Ziel--> Rating mit 100 Leuten (Gegner mit max. 100P. Unterschied)
  - Bearbeitung Datenbank (10)
  - Überarbeiteter Code
  - Namen Leute Zahlen von 0 bis 10 Zufallsgenerator (Skill) – 100 verschiedene

Montag, 08.01.2024

- Mathematische Formeln aufstellen

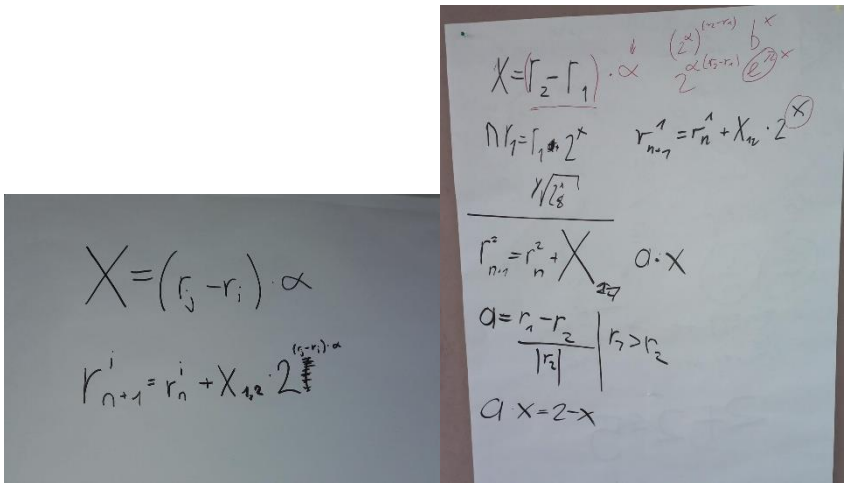
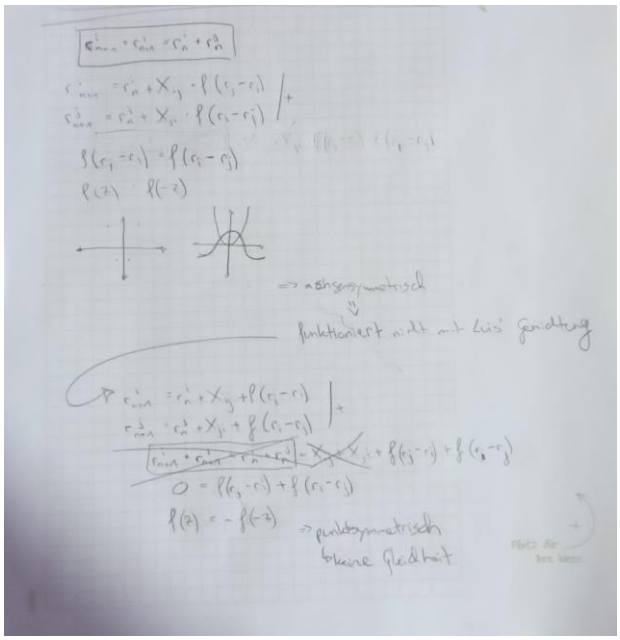
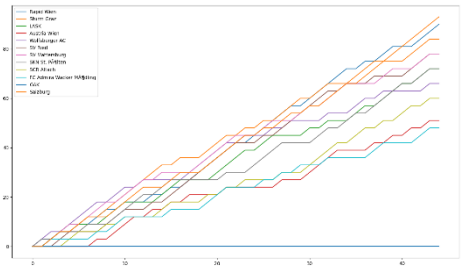


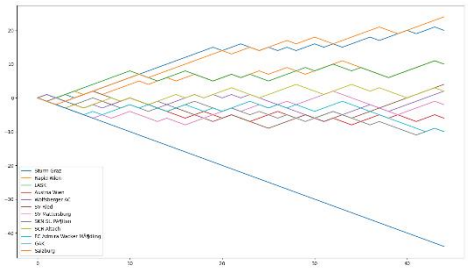
Bild 1 (Formel Luis) vs. Bild 2 (Formel Jakob)

- Bei 1 kommen neue Rating Punkte ins System dazu, bei 2 nicht -> bei 2 funktioniert der 2. Teil aber nicht
- 1: beim Gewinner = Winner minus Looser; beim Verlierer = Looser minus Winner
- 2. System aufgegeben
- Bei originaler Idee, wo Originalwerte die gleichen sind wie die nach einem Spieltag -> kann mit Formel 1 nicht funktionieren, mit mal und achsensymmetrisch
    - Deshalb mit Addition => f(z) = - f(-z) => punktsymmetrisch

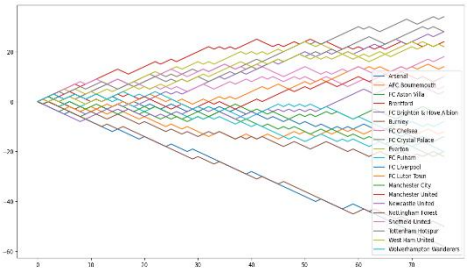


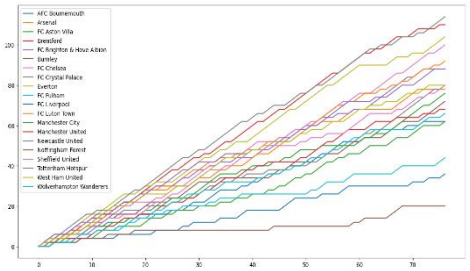
- Problem mit Graphen
    - Brauchen Funktion zwischen –1 und 1 -> Tangens Hyperbolicus
- 1. Vorarbeit für die Präsentation: verschiedene Datensets
    - **Österreich-Fußballliga** mit 12 Mannschaften, 2x Hin- und Rückrunden = 4 Runden, 4 Spieltage – Skills [1, 10[von Ilija und Luis fixiert worden
        - Alle gegen alle (0 oder 3 Punkt) & (-1 oder 1 Punkte)

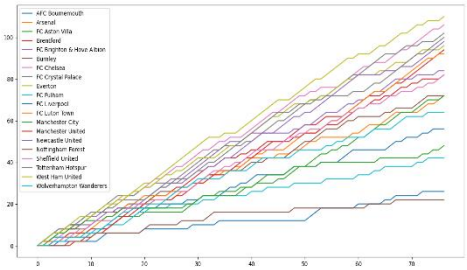




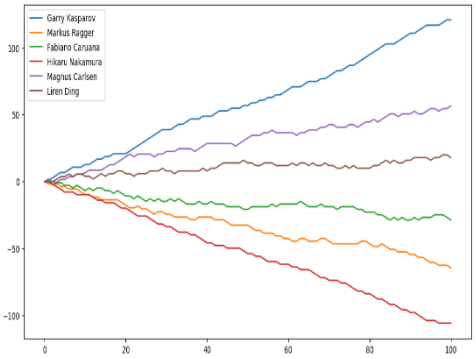
    - **England-Fußballliga**
        - 20 Teams, 2 Runden, alle gegen alle
          0/2 und 0/3 und -1/1 Punkte bei Sieg/Niederlage - verschiedene tries
          Skills [1,10] auf die 20 Teams shuffeln


Arsenal
AFC Bournemouth
FC Aston Villa
Brentford
FC Brighton & Hove Albion
Burnley
FC Chelsea
FC Crystal Palace
Everton
FC Fulham
FC Liverpool
FC Luton Town
Manchester City
Manchester United
Newcastle United
Nottingham Forest
Sheffield United
Tottenham Hotspur
West Ham United
Wolverhampton Wanderers

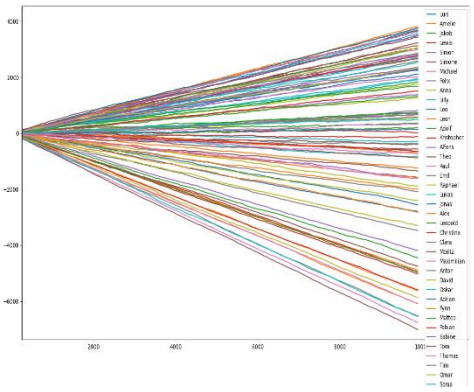

AFC Bournemouth
Arsenal
FC Aston Villa
Brentford
FC Brighton & Hove Albion
Burnley
FC Chelsea
FC Crystal Palace
Everton
FC Fulham
FC Liverpool
FC Luton Town
Manchester City
Manchester United
Newcastle United
Nottingham Forest
Sheffield United
Tottenham Hotspur
West Ham United
Wolverhampton Wanderers


AFC Bournemouth
Arsenal
FC Aston Villa
Brentford
FC Brighton & Hove Albion
Burnley
FC Chelsea
FC Crystal Palace
Everton
FC Fulham
FC Liverpool
FC Luton Town
Manchester City
Manchester United
Newcastle United
Nottingham Forest
Sheffield United
Tottenham Hotspur
West Ham United
Wolverhampton Wanderers

- **Schachliga**
  - 6 Spieler: innen, wie zuvor
    beide Ratingsysteme verglichen, 100 Spiele



  - 100 Spieler: innen, wie zuvor (-1/1)
    Skills [1,10] auf die 100 Spieler: innen shufflen
    beide Ratingsysteme verglichen, 100 Spiele



Dienstag, 09.01.2024

- Code umgeändert, optimiert - schneller, 500 000 in 3 Sekunden, können größere Graphen schnell plotten

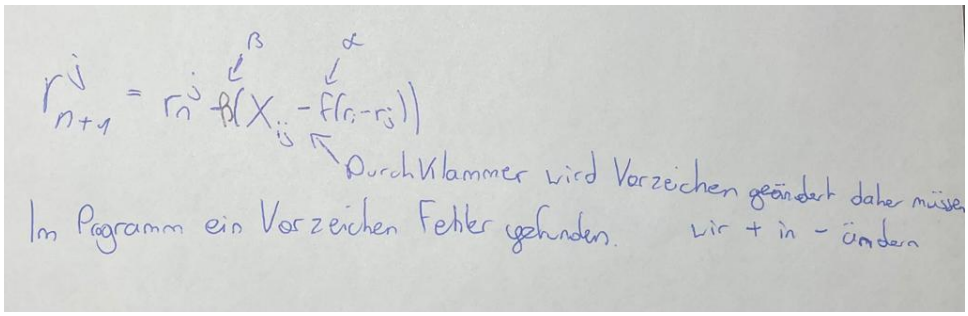
Zusammenfassung

- (0/1) Punkteverteilung ist einfach; (-1/1) besser bei Graphen zu plotten und erkennen, System verliert keinen punkte, bleiben gleich
  - Geht immer weiter auseinander
- Luis'sche Gewichtung, multiplikativ gewichtet mit den verschiedenen Spielern, mit variablem Alpha auch noch ändern
  - Einer haut immer ab, weil punkte dazu entstehen im Gesamtsystem
- Mit Formel bewiesen, dass Luis'sche Gewichtung, mit mal achsensymmetrisch ist, aber nicht funktioniert, weil die Gesamtpunkte sich verändern
- Deshalb mit Addition – wird dann punktsymmetrisch und funktioniert

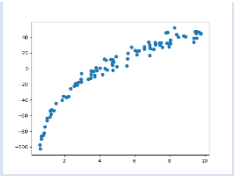
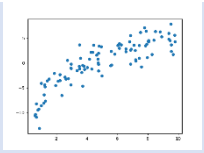
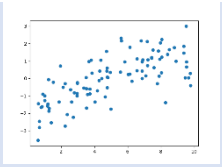
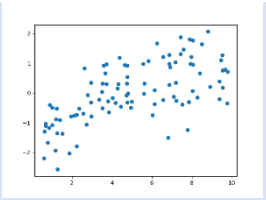
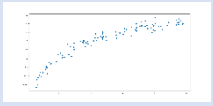
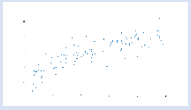
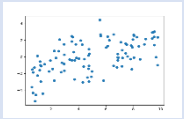
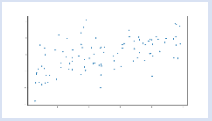
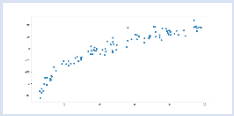
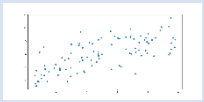
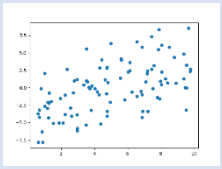
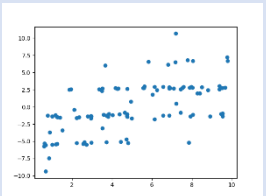
- Funktion muss zwischen –1 und 1 sein, punktsymmetrisch, muss sich an –1 und 1 nur annähern, nicht überschreiten, Funktion muss immer größer werden = streng monoton steigend – deshalb funktioniert Sinus nicht – Tangens hyperbolicus

- Kalibrierung
  - Alpha und Beta sensibles Verhältnis miteinander bei der Formel
- X-Achse Zeit <-> y-Achse Rating (i) | x-Achse skill <-> y-Achse Rating (ende)
- Alpha und Beta sind abhängig voneinander

- Vorzeichen-Fehler gefunden:



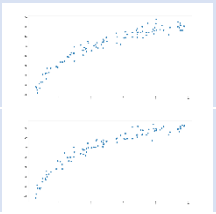
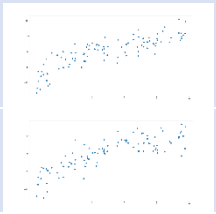
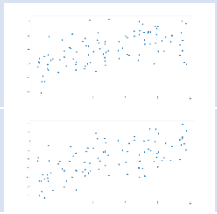
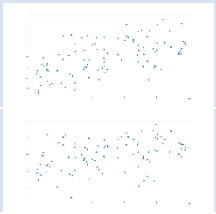
100 Schachspieler, 200 Tage -> Luis' Gewichtung mit Tangens

Alpha ->	0,01	0,1	0,5	1
Beta=0,5				
Beta=1				
Beta=2				

- Je kleiner Alpha ist und kleiner Beta ist, desto besser das Produkt
- Bauchige kurve (von 9 auf 10 weniger unterschied als bei 1 bis 2)
- Bei 0 => perfekte kurve, aber gehen immer weiter auseinander = Streuung insgesamt wird schlechter

- 0,1 = 😟 und 0,01 = 😊 aber zu große Streuung => wollen minimale gesamte Streuung müssen eine gute Mischung von 0,1 und 0,01 finden
    - Beta = 1

Gleich wie vorher aber Beta immer 1

Alpha ->	0,01	0,1	0,5	1
400 Tage				
800 Tage				

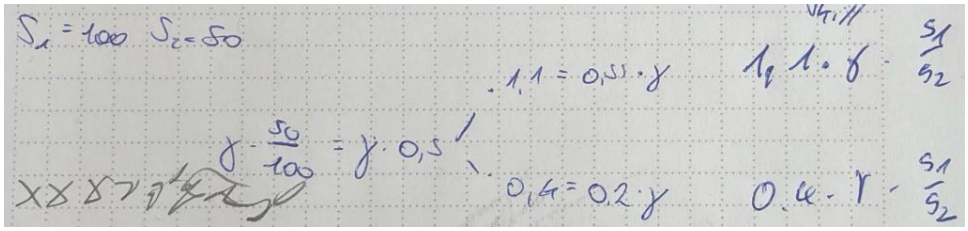
1. Brauchen viel weniger spiele bei manchen parameterformen
    a. Luis, Ilija -> Rundungsfehler gefunden
2. Suchen noch zwischen 0,1 und 0,01 Beta und die Anzahl der Spiele (s)
3. Beta, Alpha und Anzahl der Spiele in der Legende (im Bild)

Zwischenstand vom Vergleich

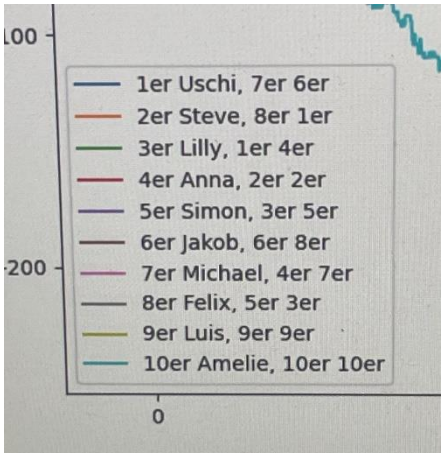
- Abgemacht mit 10-100 Spieltage ist optimal
    - Darüber kaum unterschied

- Neuer Plan
    - N=10 (Spieler) -> deshalb 45 Matches
    - Vorher: 1 Spieltag entspricht 45 Matches
    - Jetzt: 1 Spieltag entspricht 1 Match für 45 Spieltage
- Toleranzrate womit zwei Spieler miteinander gewürfelt werden
    - Haben 2 ausgewählt aus Übungszwecken
    - Programmiert, dass der Toleranzbereich mit der Differenz vom Rating von i und j im Betrag kleiner gleich zwei sein muss
        - Dass Spieler, die gegeneinander spielen nicht allzu großen Ratingunterschied haben
    - Wenn zufällig gewählter Spieler keinen Gegner, dann Toleranzbereich verändert
        - Nach Spiel: reset
- Toleranzgrenzenveränderung wieder rausgehaut
- Neue Idee: Skills updaten

Mittwoch, 10.01.2024



- Diese Formel funktioniert im Programm sehr gut
- Unterschied mit verschiedenen Gammas



Zum Beispiel: 1er vor Uschi ist der Skill am Anfang

Uschi, 7er ist der Skill am Ende

6er bei Uschi ist das Elo Rating am Ende

Parameter verändern (Alpha, Beta, Gamma, toleranzrate)

Alpha – die Zahl, die im Tanh ist; dass Tanh gestreckt wird; wenn Alpha erhöht, Punkte niedriger die bekommt bei großer Differenz = höheres Alpha: höheres Punishment

Beta – die Punkteverteilung; Skalierung von den Punkten, die man bekommen/verlieren würde; indirekt proportional

Gamma – Variable, die macht, dass die Lerneffekte nicht extrem sind = zu viel, wenig

- 10 Spieler, 10 000 Spieltage
- Alpha: statisch – 0,2
- Beta: statisch – 2
- Gamma: wird geändert
- Toleranzrate: statisch – 100

Gamma->	**0,001**	**0,01**	**0,1**
Durchgang 1			Zu schwach

	Zu wenig Veränderung von Skills	Die Obersten und Untersten duellieren sich eher nur untereinander (2), lernen nicht Also mehr im Mittelfeld	

Zwischen 0,001 und 0,01 sieht es am besten aus – Mittelding finden

Alpha = 0,001 ->Vermutung: Rating funktioniert einigermaßen

- Die, die nie spielen, keine skillverbesserung
- Fixes Gamma

- Andere Varianten des Modells
  - K.O. System
    - Personen mit niedrigen Skills können ins Finale kommen
    - Zweitbeste könnten direkt in erster Runde ausscheiden

Wahrnehmungspsychologie

Autostereogramme

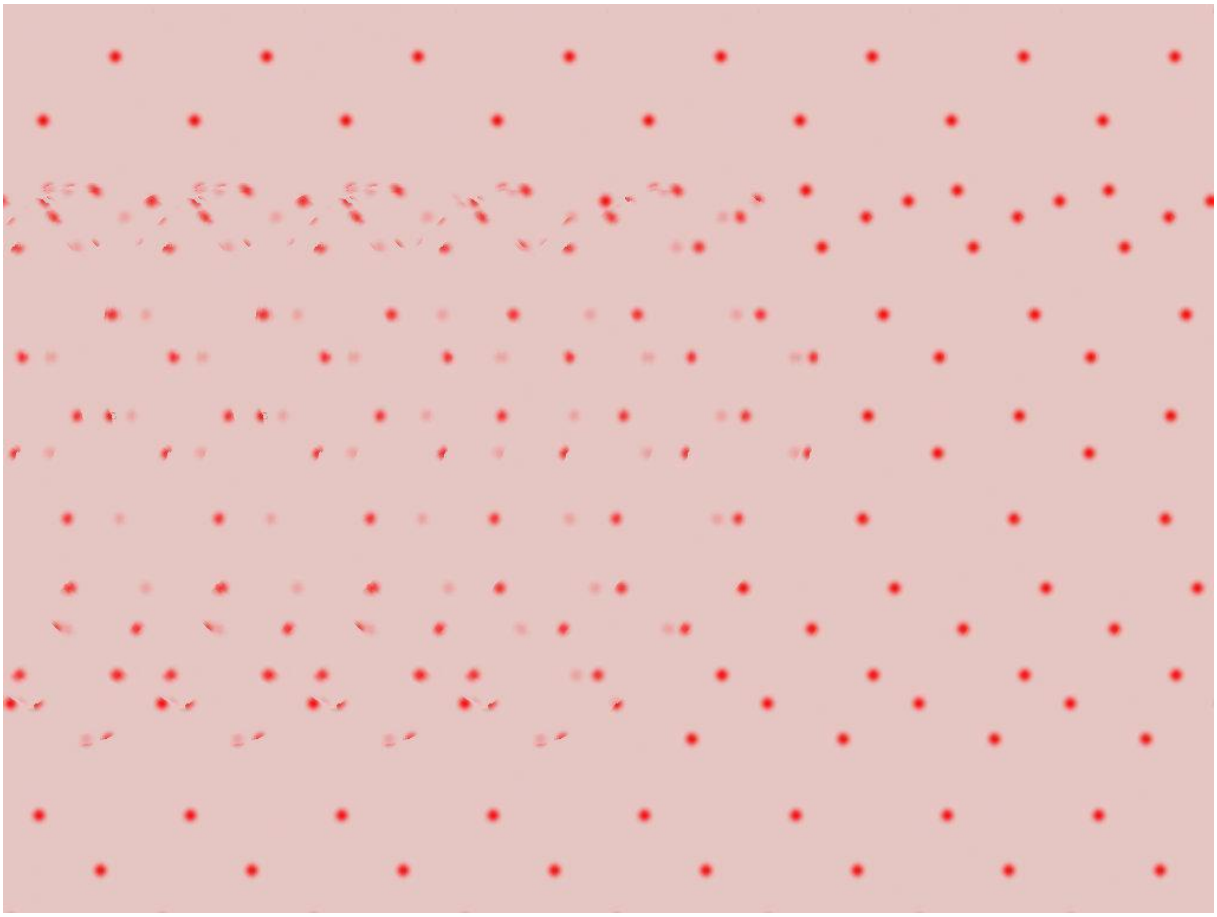
Sophia Neukirchner
Hannah Grabner
Eileen Schmieger
Maya Muriel Katschnig
Miriam Neukirchner
Noreen Garea

Inhalt

1 Autostereogramme

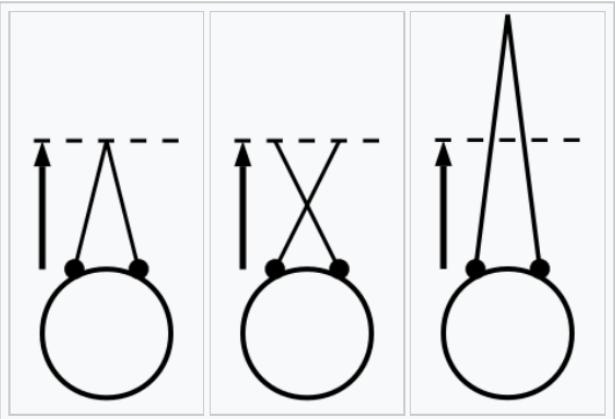
Autostereogramme sind zweidimensionale Bilder, die dem Betrachter mit der richtigen Methode als dreidimensional erscheinen können. Dies ist Teil einer optischen Täuschung, die durch die Verschiebung der Periodizität erzeugt wird. Durch methodisches Anschauen eines Autostereogramms wird die Tiefenwahrnehmung getäuscht und das jeweilige Bild wird als dreidimensional wahrgenommen, was bedeutet, dass der Betrachter mehrere Ebenen erkennen kann. In der Regel gibt es drei verschiedene Techniken, mit denen man Autostereogramme betrachten kann.

1



1.2 Methoden zur Betrachtung von Autostereogrammen

Für die Betrachtung von Autostereogrammen gibt es zwei Möglichkeiten. Zum einen bietet sich die sogenannte „cross-eyed“-Technik an, die im Deutschen große Ähnlichkeit mit dem „Schielen“ hat und demnach simpler für das Verständnis jener ist, die zum ersten Mal ein Autostereogramm betrachten. Eine andere Methode wäre die „wall-eyed“-Technik, bei der der Fokus auf ein Objekt auf der Ebene hinter dem Bild gerichtet wird.



Aligned vergence and accommodation. How one usually views objects.[1]

Cross-eyed vergence (⨉).[1] Arrow indicates accommodation.

Wall-eyed ("parallel") convergence (⊔).[1]

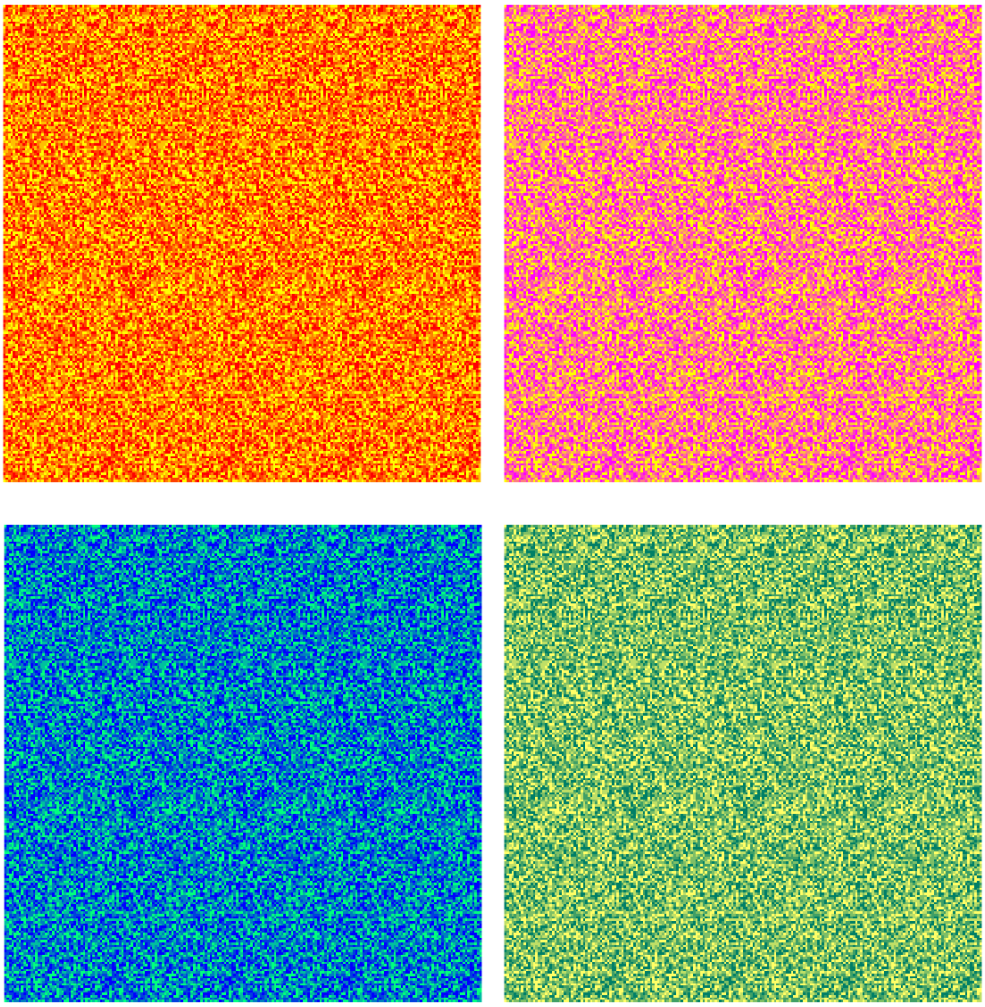
1.3 Bedingungen zum Betrachten von Autostereogrammen

Es gibt einige Aspekte, die beim Betrachten eines Autostereogramms in Abhängigkeit zur Qualität stehen. Zunächst beeinflusst die Pixelanzahl den Zustand des Autostereogramms. Diese sollten allgemein weder zu übertrieben groß noch zu übertrieben klein sein. Grundsätzlich kann man jedoch sagen, dass ein Autostereogramm deutlicher zu erkennen ist, wenn die Anzahl der Pixel geringer ist.

Weiters sollte auch die Verschiebung nicht zu groß sein und den Wert 30 nicht überschreiten. Eine Verschiebung von eins bis fünf passt in den meisten Fällen und als optimal erweist sich eine Verschiebung von drei bis zehn.

Darüber hinaus beeinflusst auch die Periodizität die Beschaffenheit des Autostereogramms. Diese hängt in der Regel vom Bild selbst ab, je nach Motiv und Größe variiert diese. Die Wiederholung sollte jedoch üblicherweise zwischen vier und sechs liegen.

Letztlich spielt die Farbe des Autostereogramms ebenfalls eine große Rolle beim Betrachten. Jede Person, die sich ein Stereogramm ansieht, hat gewisse Präferenzen gegenüber den Farben des Autostereogramms und Bedingungen, unter denen sie das im Autostereogramm versteckte Bild besser oder schlechter erkennen kann. Unten angeführt sind vier Autostereogramme, die dasselbe Bild darstellen, jedoch in verschiedenen Farben.



1.4 Abstand vom Bild zu den Augen bei verschiedenen Periodizitäten und Verschiebungen

Sophia (mit Handy): Periodizität – 150

Verschiebung – 5
Abstand zum Bild: 8,5cm

Period. – 50
Verschiebung – 5
Abstand: 9cm

P. – 250
V. – 5
A: 9cm

P. – 150
V. – 3
A: 9,5cm

P. – 50
V. – 3
A: 9cm

P. – 250
V. – 3
A: 9cm

P. – 150
V. – 7
A: 9cm

P. – 50
V. – 7
A: 9cm

P. – 250
V. – 7
A: 9cm

Fazit: minimale Veränderung, allerdings nicht auffällig
Fehleranalyse: ungenaue Messung möglich

Miriam (mit Laptop): Periodizität – 50

Verschiebung – 3
Mindestabstand zum Bild: 7,5cm

P. – 100
V. – 3
Mindesta: 8,5cm

P. – 50
V. – 5
Mindesta: 9cm

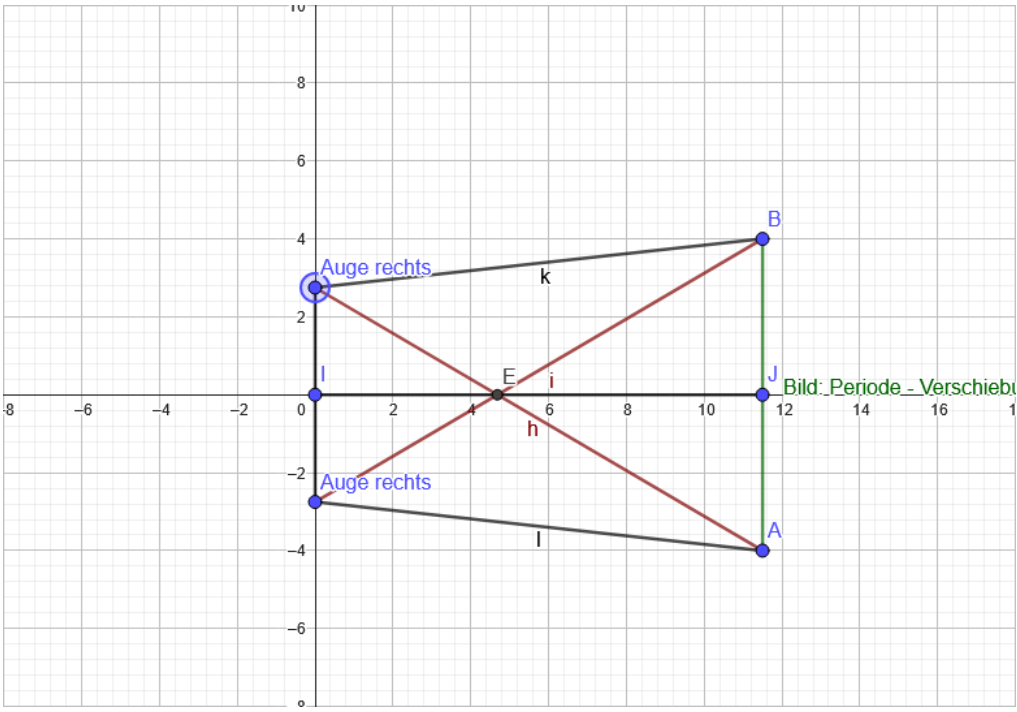
P. – 100
V. – 5
Mindesta: 8,5cm

P. – 50
V. – 7
Mindesta: 8,5cm

P. – 100
V. – 7
Mindesta: 7cm

Fazit: kleine Veränderung ersichtlich, bei Verschiebung 5 ist ein wenig größerer Mindestabstand notwendig
Fehleranalyse: ungenaue Messung möglich

1.5 Berechnungen mit Geogebra



A = (11.5, -4)

B = (11.5, 4)

f = Strecke(A, B)
= 8

C = (0, 2.75)

D = (0, -2.75)

g = Strecke(C, D)
= 5.5

I = (0, 0)

J = (11.5, 0)

j = Strecke(I, J)
= 11.5

i = Strecke(D, B)
= 13.33

h = Strecke(C, A)

Erklärung:

A, B: Punkte auf einem Bild, entspricht selben Punkt aber mit einer Verschiebung.

C, D: Augen

g Strecke (C, D): Abstand der Augen voneinander

h Strecke (C, A): linkes Auge schaut zu Punkt A

i Strecke (D, B): rechtes Auge schaut zu Punkt B

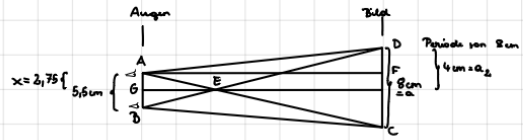
j Strecke (I, J): Abstand der Augen zum Bild

E = Schnittpunkt(i, h)
= (4.69, 0)

k = Strecke(C, B)
= 11.57

l = Strecke(D, A)
= 11.57

Beispiel zur Stereopaar-Kreuzübung 8 cm Periode



$AC = s \quad BD = s \quad AF = e$

$b = a_2 + x$

$s_1 = \sqrt{e^2 + b^2}$

$s_1 = \sqrt{11,5^2 + 6,75} = 13,33 \text{ cm}$

$AB = w, AC = w$
$y = a_2 + x$

$w_1 = \sqrt{e^2 + y}$

$w_1 = \sqrt{11,5^2 + 1,25^2} = 11,567 \text{ cm}$

$\frac{s}{\sin\delta} = \frac{w}{\sin\delta_1} \quad | \cdot \sin\delta \quad | \cdot \sin\delta_1$

$s \cdot \sin\delta_1 = w \cdot \sin\delta \quad | : s \quad | \sin^{-1}$

$\delta_1 = \sin^{-1}\left(\frac{w \cdot \sin\delta}{s}\right)$

$\delta_1 + \delta_2 \qquad \tan = \frac{G}{A} \quad /A$

$\tan\delta_1 \cdot A = G$

$\tan 59,617 \cdot 4 = 6,82 \text{ cm}$

$\sphericalangle LCB = \gamma \qquad \tan\gamma = \frac{1,25}{11,5} = 6,12°$

$\sphericalangle CBL = \delta \qquad \delta = 90 - 6,02 = \underline{83,8°}$

$\sphericalangle DCA = \delta_1$

$\delta_1 = \sin^{-1}\left(\frac{11,567 \cdot \sin 83,8}{13,33}\right)$

$\delta_1 = 59,617°$

$\sphericalangle DEC = \beta \qquad \beta = 180 - (2 \cdot 59,617) = 60,765°$

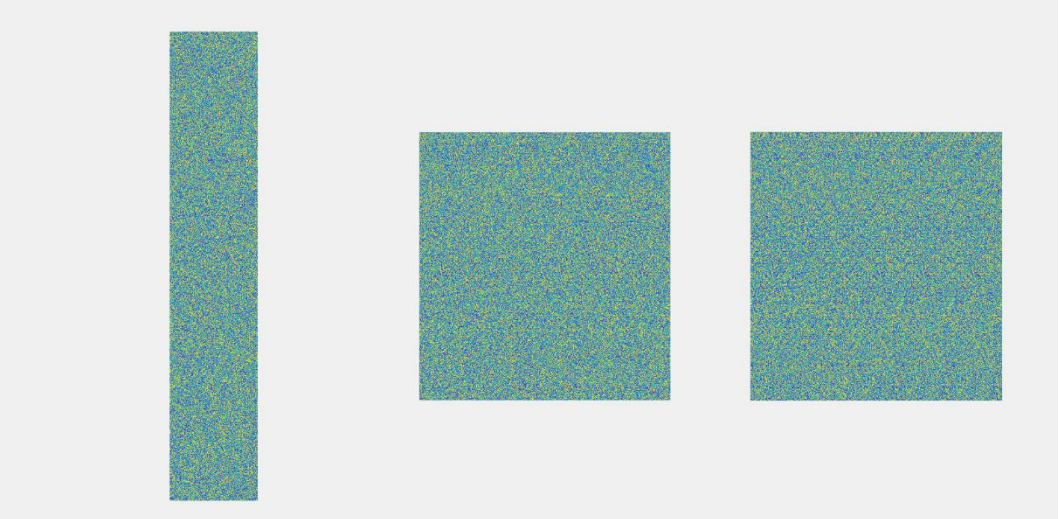
$GE = e \qquad e = 11,5 - 6,82 = \underline{4,68 \text{ cm}}$

2 Darstellungen

2.1 Darstellung eines Vierecks in einem Autostereogramm

```
clc;
p=rand(800,150);
imagesc(p); axis image; axis off;

q=repmat(p,1,5);
imagesc(q); axis image; axis off;

d=zeros(800,150*5);
d(200:400,200:400)=30;
imagesc(d); axis image; axis off;

subplot(1,3,1);
imagesc(p); axis image; axis off;
subplot(1,3,2);
imagesc(q); axis image; axis off;
subplot(1,3,3);
imagesc(d); axis image; axis off;
```

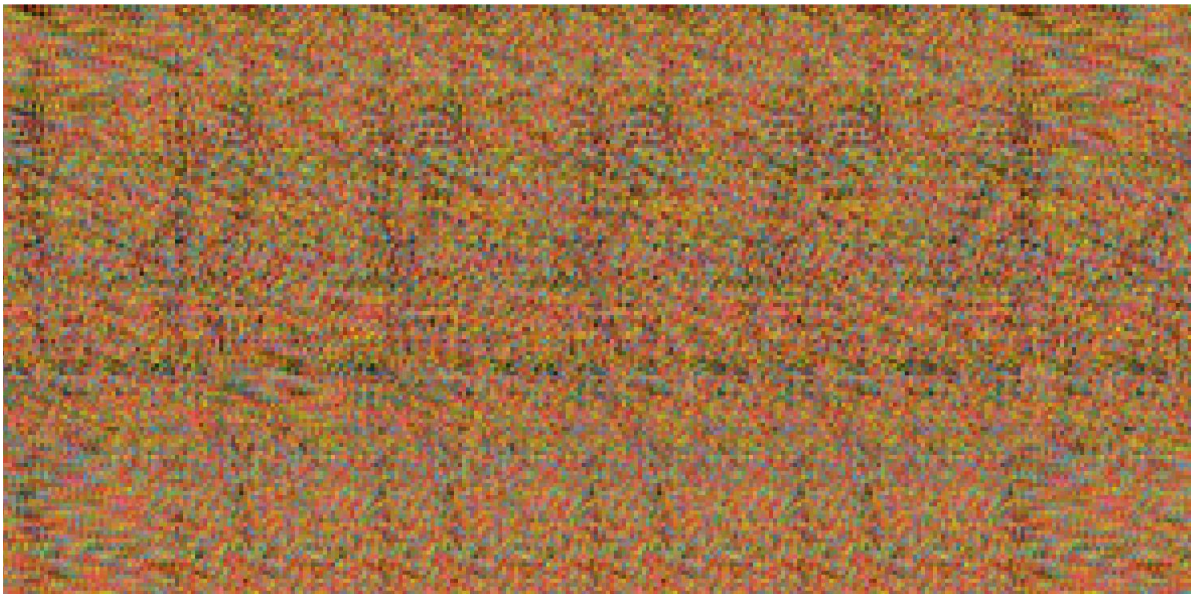
```
 r=q;
for i=1:800
 for j=81:(5*80)
   r(i,j)=r(i,j-80+d(i,j));
 end
end
```



2.2 Bewegte Darstellung des Autostereogramms „Shark“

```
s=imread('shark.png');
for i=0:100
  s1=double(circshift(s,i,2));
  s2=double(circshift(s,-i,2));
  imagesc(uint8((s1+s2)/2)); axis image; axis off;
  drawnow;
end
```

Diese bewegte Darstellung war eine der ersten Versuche unsererseits das zweidimensionale Bild, das der Betrachter erkennen kann, in 3D darzustellen, um die Ansicht für alle möglich zu machen, die das Bild anders nicht erkennen können.



2.3 Verbesserte bewegte Darstellung des Autostereogramms „Shark“

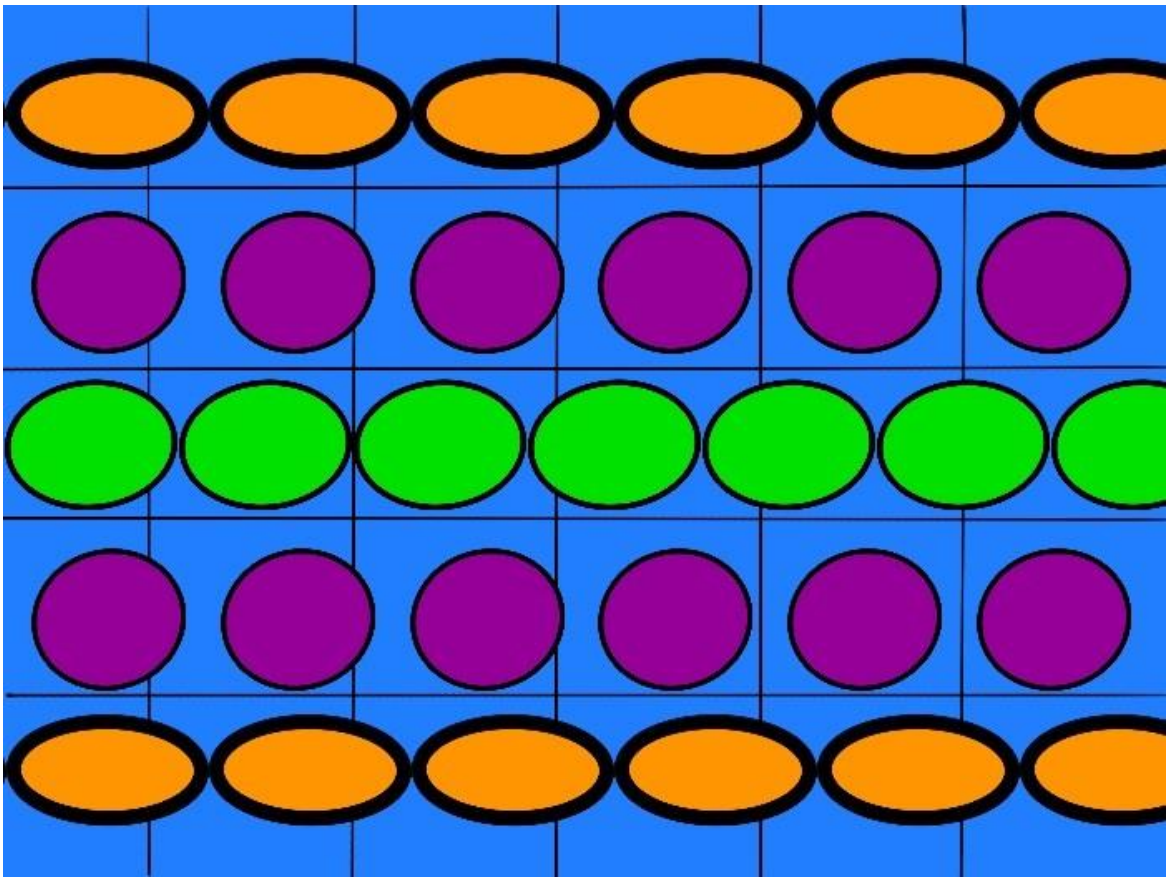
```
s  = imread('shark.png'); l=106;
s  = imread('Shark.png'); l=140;
m  = size(s,1); n = size(s,2);
l2 = round(l/2);

h1 = figure(1); close(h1); h1 = figure(1);
set(h1,'Position',[10 40 1024 512]);

s0 = double(circshift(s,l2,2));

w  = 1;
i1 = l2-w;
i2 = l2+w;
```

```
for k=1:100
    for i=i1:i2
        s1 = double(circshift(s,i,2));
        s2 = double(circshift(s,-i,2));
        sa = uint8(s1+s2-s0);
        imagesc(sa); axis image; axis off; grid off;
        drawnow;
    end
    for i=i2:-1:i1
        s1 = double(circshift(s,i,2));
        s2 = double(circshift(s,-i,2));
        sa = uint8(s1+s2-s0);
        imagesc(sa); axis image; axis off; grid off;
        drawnow;
    end
end
```

3 Eigene Autostereogramme

3.1 Kästchen Autostereogramm

Bereits am zweiten Tag gelang es Maya Muriel Katschnig ein Autostereogramm ohne jegliche Programmierung zu zeichnen.

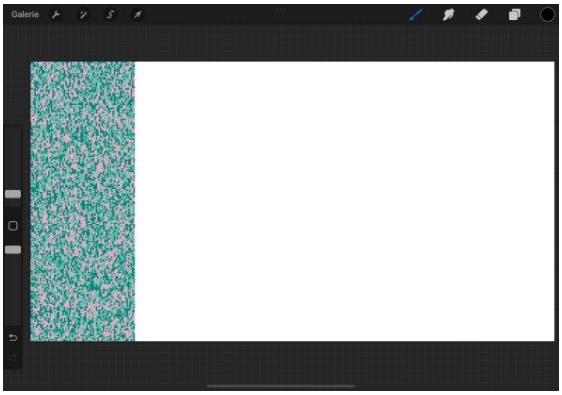


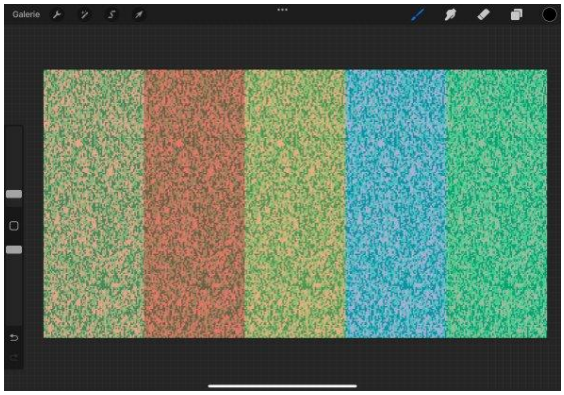
3.2 Autostereogramme mit Procreate

Maya Muriel Katschnig hat im Laufe der Woche 20 Autostereogramme und 5 Stereogramme erstellt, unter anderem, das einer Katze, das man vergleichsweise sehr gut erkennen kann.

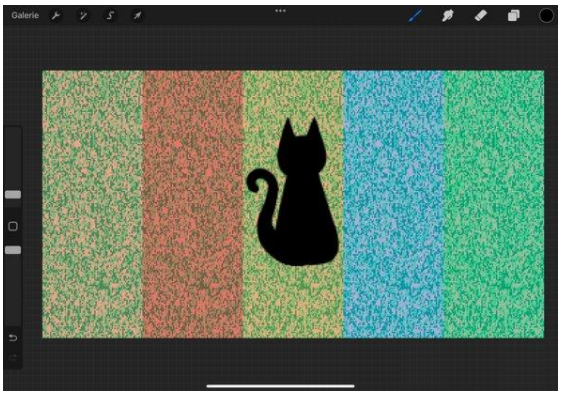
Um ein Pixel Autostereogramm mit Procreate zu erstellen, hat sie eine einfach nachzumachende Weise, um dies zu tun entwickelt. Um diese auszuführen, kann man folgende Schritte befolgen:

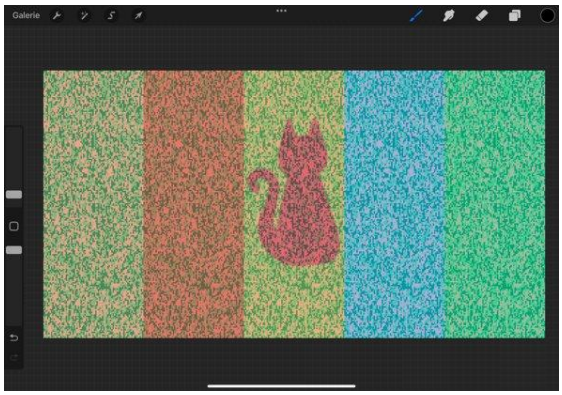
- Schritt 1: Man fügt einen zufallsgenerierten Pixelstreifen in einer Breite von mindestens 30 Pixeln in das Zeichenprogramm der Wahl ein und dupliziert diesen etwa 4–5-mal. In dem angefügten Bild haben die einzelnen Streifen unterschiedliche Farben, um sie einfacher erkennbar zu machen.
- Schritt 2: Man rückt die duplizierten Pixelstreifen nebeneinander ein.



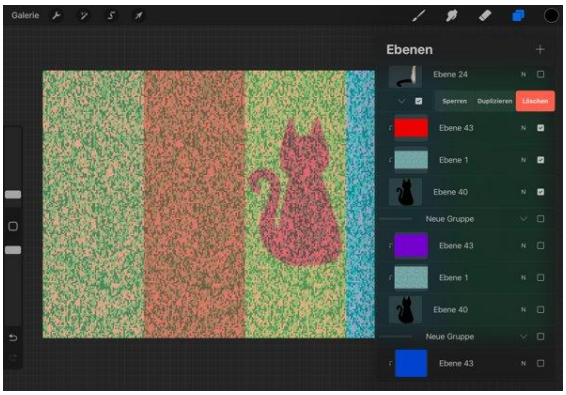


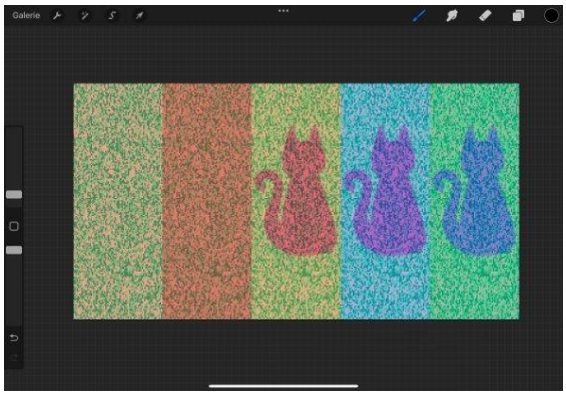
- Schritt 3: Man fügt die Form, die man später herausstechen sehen will ein.
- Schritt 4: Man dupliziert alle Streifen nochmal und fügt diese in ein Bild zusammen. Dann stellt man diese als Clipping-Maske für die Form ein.



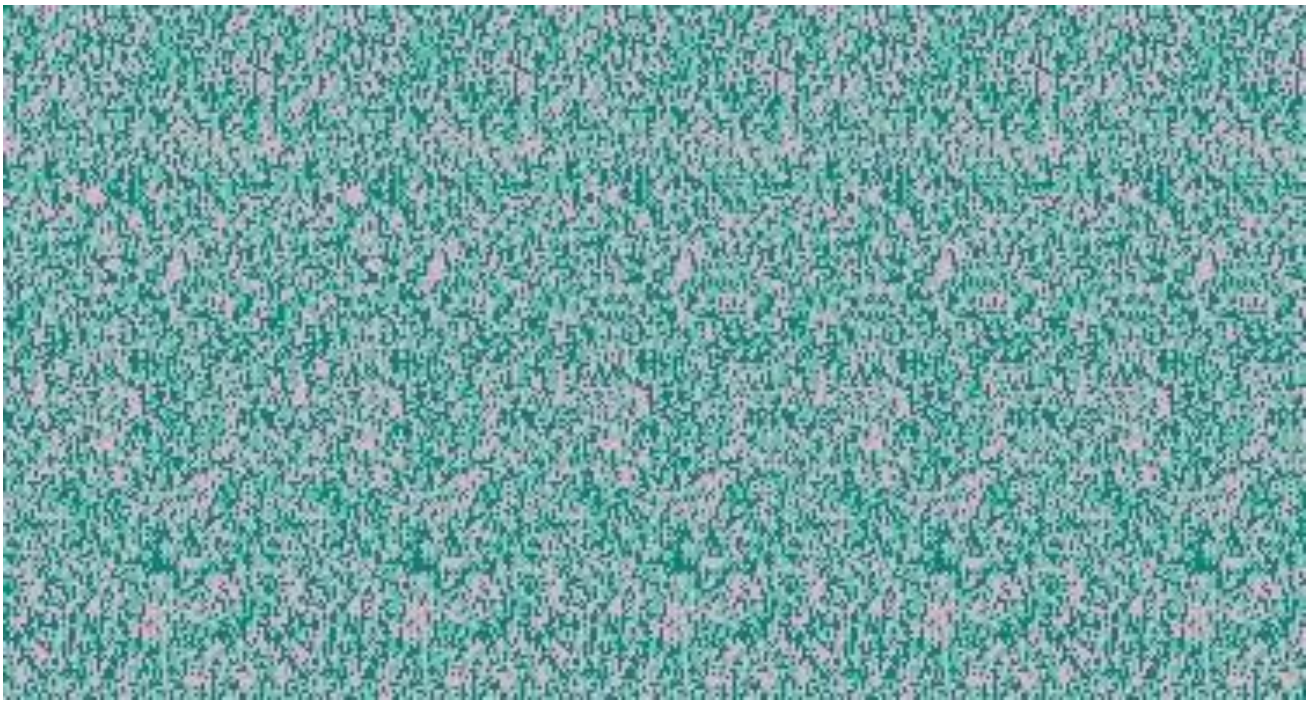


- Schritt 5: Man gibt die Clipping-Maske und die Form zusammen in eine Gruppe und verrückt diese um etwa 2-10 Pixel nach links um sie später herausstechen zu lassen. Um es so erscheinen zu lassen, als wäre die Form hinter dem Rest des Bildes verrückt man sie nach rechts.
- Schritt 6: Man dupliziert die Gruppe und rückt sie an die gleichen Stellen in den restlichen Streifen an der rechten Seite der Form ein.





Fertiges Bild:



Um dieses Autostereogramm für alle Leserinnen und Leser sichtbar zu machen, hat ihre Kollegin Sophia Neukirchner das gezeichnete Bild auf Matlab eingelesen und akzentuiert.

```
s  = imread('katze.jpg'); l=75;
m  = size(s,1); n = size(s,2);
l2 = round(l/2);

h1 = figure(1); close(h1); h1 = figure(1);
set(h1,'Position',[10 40 1024 512]);

s0 = double(circshift(s,l2,2));

w  = 1;
i1 = l2-w;
i2 = l2+w;
```

Scan me!

```
for k=1:100
    for i=i1:i2
        s1 = double(circshift(s,i,2));
        s2 = double(circshift(s,-i,2));
        sa = uint8(s1+s2-s0);
        imagesc(sa); axis image; axis off; grid off;
        drawnow;
    end
    for i=i2:-1:i1
        s1 = double(circshift(s,i,2));
        s2 = double(circshift(s,-i,2));
        sa = uint8(s1+s2-s0);
        imagesc(sa); axis image; axis off; grid off;
        drawnow;
    end
end
```

3.3 Autostereogramme mit Matlab

3.3.1 Autostereogramm Rechteck

```
p=rand(1000,100);
imagesc(p); axis image; axis off;
q=repmat(p,1,10);
imagesc(q); axis image; axis off;
d=zeros(1000,100*10);
d(50:500,50:400)=6;
```

Die Variable „p“ beschreibt die Pixelgröße eines einzelnen Streifens (1000 ist dabei die Höhe und 100 die Breite und damit auch die Periodizität in Pixel). Durch den Befehl „rand“ entsteht dann das zufällige Muster des Autostereogramms. „q“ gibt an, wie oft dieser Streifen „p“ nach unten und rechts wiederholt wird und der Befehl „repmat“ führt diese Wiederholung aus(1x waagrecht und 10x nach rechts). „d“ ist unsere Depth Map (in diesem Fall ein Rechteck) und 6 ist die Verschiebung in Pixel. Eine Depth Map ist die Darstellung des Bildes eines Autostereogramms, das man mit der richtigen Sehtechnik erkennen kann. Die Werte in der Klammer nach dem Befehl „zeros“ ergeben die Größe der Depth Map in Pixel und die Zahlen in der nächsten Klammer geben an, von wo bis wo das Bild zu sehen sein wird (ab Pixel 50:50 bis hin zu Pixel 500:400).

```
  subplot(1,3,1);
  imagesc(p); axis image; axis off;
  subplot(1,3,2);
  imagesc(q); axis image; axis off;
  subplot(1,3,3);
  imagesc(d); axis image; axis off;

r=q;

for i=1:1000
for j=101:(10*100)
r(i,j)=r(i,j-100+d(i,j));
end
end
imagesc(r); axis image; axis off;
```

„subplot“ ist zur Veranschaulichung der einzelnen Variablen nebeneinander. „r“ zeigt dann das fertige Autostereogramm durch die „for“-Schleifen. Der Befehl „imagesc“ veranschaulicht die einzelnen Variablen.

3.3.2 Autostereogramm eines Kreises

Sophia Neukirchner

```
h1 = figure(1); close(h1); h1 = figure(1);
set(h1,'Position',[10 40 1024 1024]);

m = 1000; n = 750; l = 150; l2=round(l/2);  % l = periodizitaet

x = linspace(0,1,m)';                       % spaltenvektor
y = linspace(0,1,n);                        % zeilenvektor
xx = kron(x,0*y+1);                         % x-array fuer jedes pixel
yy = kron(0*x+1,y);                         % y-array fuer jedes pixel

v  = 5;                                     % verschiebung
d = v*double((xx-0.5).^2 ...
          + (yy-0.5).^2 < 1/20);            % depth map
p = rand(m,l);                              % streifen
q = repmat(p,1,5);                          % ausgangsbild
```

Die Variable „m“ beschreibt die Anzahl der Pixel für die Höhe eines Streifens, die Variable „n“ die Anzahl der Pixel für die Breite des gesamten Bildes. „l“ ist die Variable für die Anzahl der Pixel eines einzelnen Streifens, auch Periodizität genannt. „x“ ist die Variable für einen Spaltenvektor, „y“ für einen Zeilenvektor. „xx“ ist die Reihe von links nach rechts für jedes

Pixel. Der Begriff „yy“ für die Reihe von unten nach oben für jedes Pixel. „v“ beschreibt die Verschiebung der einzelnen Streifen in Pixel. Die Variable „d“ ist eine Kreisgleichung, die die depth map ist. „p“ beschreibt die zufällig-ausgewählten Streifen mit der Höhe „m“ und der Breite „l“ in Pixel, „q“ beschreibt die Darstellung des Streifens „p“ einmal in der Höhe und fünf Mal in der Breite.

```
r = q;                                          % ...mit verschiebungen
for i=1:m
    for j=(l+1):n
        r(i,j)=r(i,j-l+d(i,j));
    end
end

subplot(2,2,1);
imagesc(p); axis image; axis off;
subplot(2,2,2);
imagesc(q); axis image; axis off;
subplot(2,2,3);
imagesc(d); axis image; axis off;
subplot(2,2,4);
imagesc(r); axis image; axis off;

input('continue? ')



subplot(1,1,1)

% r0 = double(circshift(r,l2,2));           % bis halbe periode verschoben

w  = 1;
i1 = l2-w;
i2 = l2+w;
s  = 1;    % skalierung
```

```
for k=1:100
    for i=i1:i2
        r1 = double(circshift(r,i,2));      % nach rechts verschoben
        r2 = double(circshift(r,-i,2));     % nach links verschoben
%       ra = uint8(r1+r2-r0);               % waehle Deine Lieblings
        ra = s*uint8(r1+r2);                %   Formel aus!
        imagesc(ra); axis image; axis off; grid off;
        drawnow;
    end
    for i=i2:-1:i1
        r1 = double(circshift(r,i,2));
        r2 = double(circshift(r,-i,2));
%       ra = uint8(r1+r2-r0);
        ra = s*uint8(r1+r2);
        imagesc(ra); axis image; axis off; grid off;
        drawnow;
    end
end
```

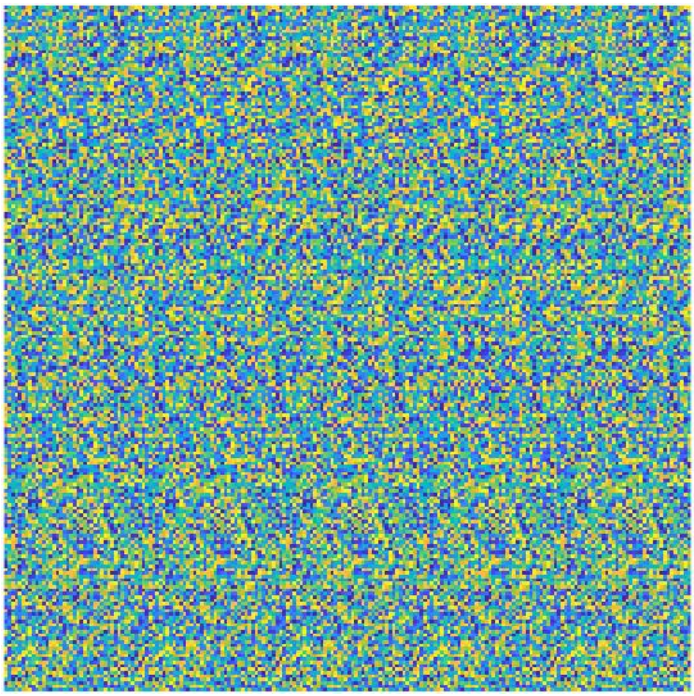
Zuerst wird eine Schleife erstellt, die das Ausgangsbild „q“ durch die depth map „d“ verschiebt. Der Begriff „subplot“ beschreibt die Darstellung der Bilder, die „p“, „q“, „d“ und „r“ beschreiben.

Danach werden Schleifen erstellt, die die zwei Ebenen des Ausgangsbildes, das verschoben worden ist, hin- und herschieben.

Hier finden Sie eine weiteres Autostereogramm eines Kreises von Miriam Neukirchner:

```
h1 = figure(1); close(h1); h1 = figure(1);
set(h1,'Position',[10 40 1024 1024]);

m = 500; n = 500; l = 100; l2=round(l/2);  % l = periodizitaet

x = linspace(0,1,m)';                      % spaltenvektor
y = linspace(0,1,n);                       % zeilenvektor
xx = kron(x,0*y+1);                        % x-array fuer jedes pixel
yy = kron(0*x+1,y);                        % y-array fuer jedes pixel

v  = 5;                                    % verschiebung
d = v*double((xx-0.5).^2 ...
          + (yy-0.5).^2 < 1/20);           % depth map
p = rand(m,l);                             % streifen
q = repmat(p,1,5);                         % ausgangsbild


r = q;                                       % ...mit verschiebungen
for i=1:m
    for j=(l+1):n
        r(i,j)=r(i,j-l+d(i,j));
    end
end

subplot(2,2,1);
imagesc(p); axis image; axis off;
subplot(2,2,2);
imagesc(q); axis image; axis off;
subplot(2,2,3);
imagesc(d); axis image; axis off;
subplot(2,2,4);
imagesc(r); axis image; axis off;

input('continue? ')
```

```
subplot(1,1,1)

% r0 = double(circshift(r,l2,2));           % bis halbe periode verschoben

w  = 1;
i1 = l2-w;
i2 = l2+w;
s  = 1;    % skalierung



for k=1:100
    for i=i1:i2
        r1 = double(circshift(r,i,2));      % nach rechts verschoben
        r2 = double(circshift(r,-i,2));     % nach links verschoben
%       ra = uint8(r1+r2-r0);               % waehle Deine Lieblings
        ra = s*uint8(r1+r2);                %   Formel aus!
        imagesc(ra); axis image; axis off; grid off;
        drawnow;
    end
    for i=i2:-1:i1
        r1 = double(circshift(r,i,2));
        r2 = double(circshift(r,-i,2));
%       ra = uint8(r1+r2-r0);
        ra = s*uint8(r1+r2);
        imagesc(ra); axis image; axis off; grid off;
        drawnow;
    end
end
```



Scan me!

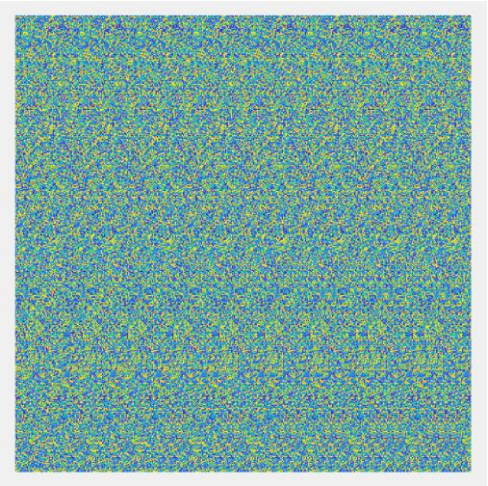
3.3.3 Autostereogramm Herz

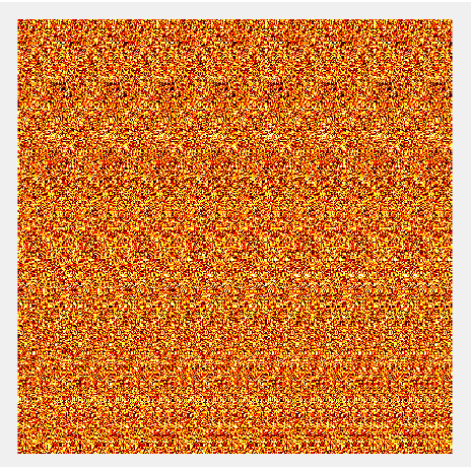
```
n0=400; m0=3*n0/2;
x = linspace(-1,2,m0)';
y = linspace(-1,1,n0);

xx=kron(x(m0:-1:1),ones(1,n0));
yy=kron(ones(m0,1),y);

h=double((xx - nthroot(yy.^2,3)).^2 + yy.^2 <= 1);
z=100;
n=n0+2*z;
m=m0+2*z;
d=zeros(m,n);
d(z+1:z+m0,z+1:z+n0) = 10*h;
imagesc(d); axis image; axis off;

l=50;
p=rand(m,l);
q=repmat(p,1,8);



r=q;
for i=1:m
for j=l+1:n
  r(i,j)=r(i,j-l+d(i,j));
end
end
%subplot(2,2,4)
imagesc(r); drawnow; axis image; axis off; colormap(hot); drawnow;

%l=80
%p=10
%f=rand(m,l);
%q=repmat(f,1,p);



%r=f;
%for i=1:m
    %for j=(l+1):(l*p)
%r(i,j)=r(i,j-l+d(i,j));
   % end
%end

%h1 = figure(1); close(h1); h1 = figure(1);
%set(h1,'Position',[10 40 900 300]);

%subplot(1,3,1);
%imagesc(f); axis image; axis off;
%subplot(1,3,2);
%imagesc(q); axis image; axis off;
%subplot(1,3,3);
%imagesc(d); axis image; axis off;
```

```
r=imread('stereogram_1.jpg');
imagesc(r); axis image; axis off;
h=rot90(r);
g=rot90(h);
l=rot90(g);

s=imread('stereogram_2.jpg');
imagesc(s); axis image; axis off;
k=rot90(s);
m=rot90(k);
n=rot90(m);

subplot (1,3,1);
imagesc(l); axis image; axis off;
subplot(1,3,2);
imagesc(n); axis image; axis off;

e=imadd(l,n);
imagesc(e); axis image; axis off;
```

3.3.4 Autostereogramm Regenbogen

```
m=400; n=400;
x=linspace(0,2,m)';
y=linspace(-1,1,n);

xx=kron(x(m:-1:1),ones(1,n));
yy=kron(ones(m,1),y);

d=10*double(((1/3)^2 <= 0.5*xx.^2 + yy.^2) & (0.5*xx.^2 + yy.^2 <= (2/3)^2));
imagesc(s); axis image; axis off;

l=50;
p=rand(m,l);
q=repmat(p,1,8);
```





```
  r=q;
for i=1:m
for j=l+1:n
  r(i,j)=r(i,j-l+d(i,j));
end
end
imagesc(r); drawnow; axis image; axis off; drawnow;
```

Erklärung

Für dieses Autostereogramm, das einen Regenbogen darstellen soll, definieren „m“ die Pixel von oben nach unten und „n“ die Anzahl der Pixel von links nach rechts. Mittels *x=linspace(0,2,m)'* und *y=linspace(-1,1,n)* werden der Spaltenvektor und der Zeilenvektor angegeben, wodurch die Größe des Bildes definiert wird. Einen Regenbogen kann man wie zwei Halbkreise darstellen, deren Zwischenbereich – also die Fläche des Regenbogens – bestimmt wird, wenn zum Beispiel eine solche Regel eingehalten wird: $(1/3)^2 <= x^2 + y^2 <= (2/3)^2$. Diese Bedingungen werden in dem Depthmap „d“ erfüllt, wo „d“ nicht Null ist. Das Depthmap wird in einem Ausgangsbild „d“ versteckt, sichtbar nur durch Überlagerungen von Verschiebungen von „r“, die der Periodizität „l“ des Ausgangsbildes entsprechen.
Der Befehl „imagesc“ stellt die einzelnen Variablen grafisch dar. „l“ gibt die Periodizität von dem Ausgangsbild „q“ an, das aus Kopien des Streifenbilds „p“ besteht. „p“ beschreibt die Pixelgröße eines einzelnen Streifens (Höhe 400 und Breite 50), das hingegen mit dem Befehl „rand“ das zufällige Muster des Autostereogramms erstellt und „q“ bestimmt die Wiederholungen des Streifens „p“ nach unten und rechts. „repmat“ führt diese Wiederholungen aus.
Das fertige Autostereogramm wird mit „r“ durch die „for“-Schleifen gezeigt.

Bei der Eingabe *colormap('hot')* ins Command Window verändern sich die Farben und es ist aufgrund der größeren Farbunterschiede leichter zu erkennen.

3.3.5 Autostereogramm Stern

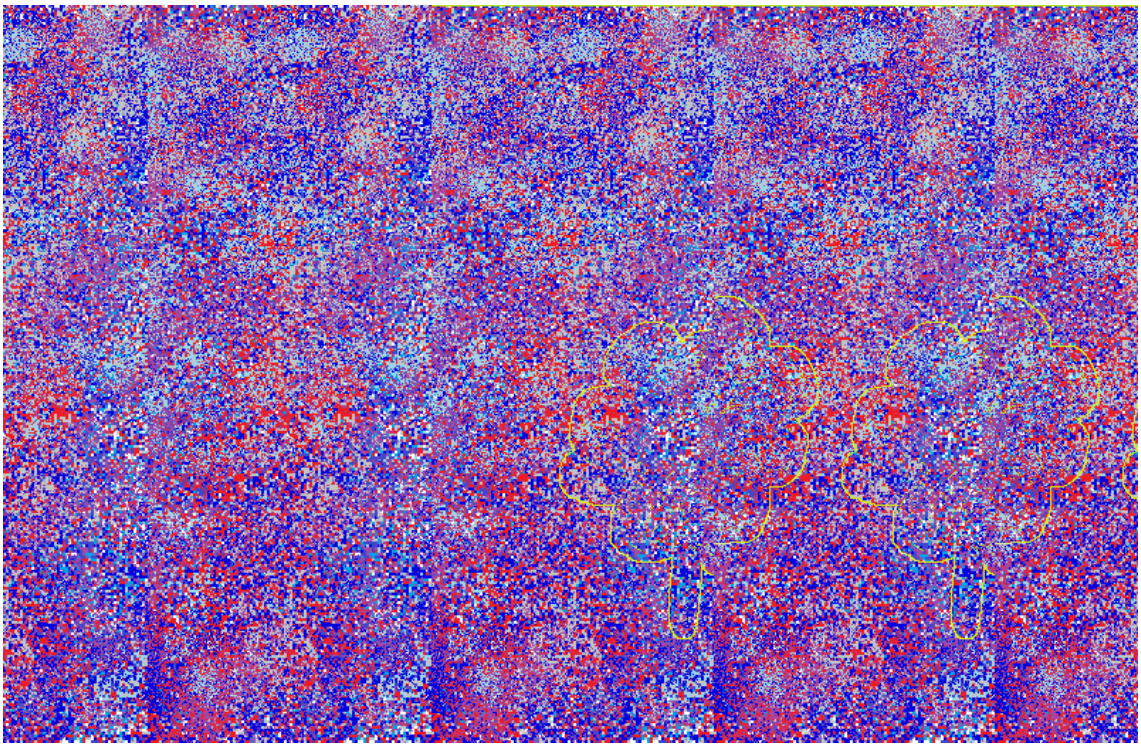
```
h1 = figure(1); close(h1); h1 = figure(1);
set(h1,'Position',[10 40 1024 1024]);

m = 1000; n = 750; l = 50; l2=round(l/2);  % l = periodizitaet

x = linspace(-2,2,m)';                      % spaltenvektor
y = linspace(-2,2,n);                       % zeilenvektor
xx = kron(x,0*y+1);                        % x-array fuer jedes pixel
yy = kron(0*x+1,y);                        % y-array fuer jedes pixel

g = 0.55;
r=80;
(abs(x).^g)+(abs(y).^g)==r.^g;

v  = 10;
d = v*double(((abs(x).^g)+(abs(y).^g)).^g) <= r.^g;
p = rand(m,l);                              % streifen
q = repmat(p,1,(n/l));                      % ausgangsbild



r = q;                                      % ...mit verschiebungen
for i=1:m
    for j=(l+1):n
        r(i,j)=r(i,j-l+d(i,j));
    end
end

subplot(2,2,1);
imagesc(p); axis image; axis off; colormap(bone);
subplot(2,2,2);
imagesc(q); axis image; axis off; colormap(bone);
subplot(2,2,3);
imagesc(d); axis image; axis off; colormap(bone);
subplot(2,2,4);
imagesc(r); axis image; axis off; colormap(bone);

input('continue? ')
```

```
subplot(1,1,1)

w  = 1;
i1 = l2-w;
i2 = l2+w;
s  = 5;    % skalierung
```

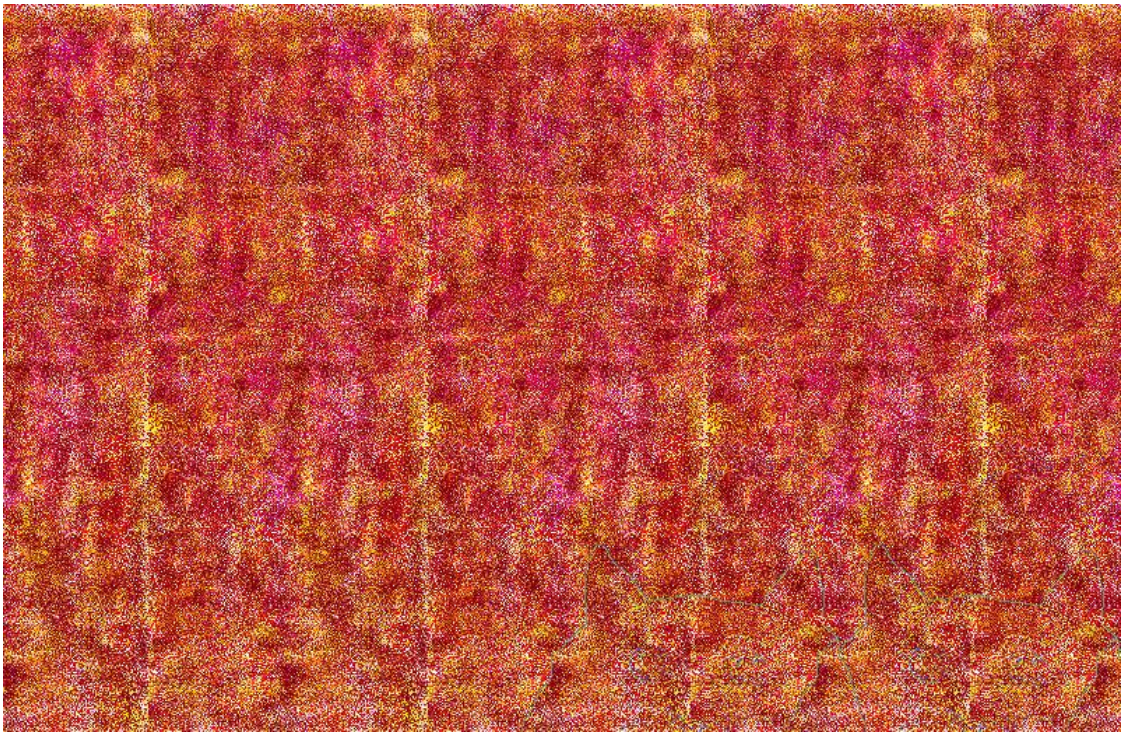
```
for k=1:100
    for i=i1:i2
        r1 = double(circshift(r,i,2));      % nach rechts verschoben
        r2 = double(circshift(r,-i,2));     % nach links verschoben
%       ra = uint8(r1+r2-r0);               % waehle Deine Lieblings
        ra = s*uint8(r1+r2);                %   Formel aus!
        imagesc(ra); axis image; axis off; grid off; colormap(bone);
        drawnow;
    end
    for i=i2:-1:i1
        r1 = double(circshift(r,i,2));
        r2 = double(circshift(r,-i,2));
%       ra = uint8(r1+r2-r0);
        ra = s*uint8(r1+r2);
        imagesc(ra); axis image; axis off; grid off; colormap(bone);
        drawnow;
    end
end
```

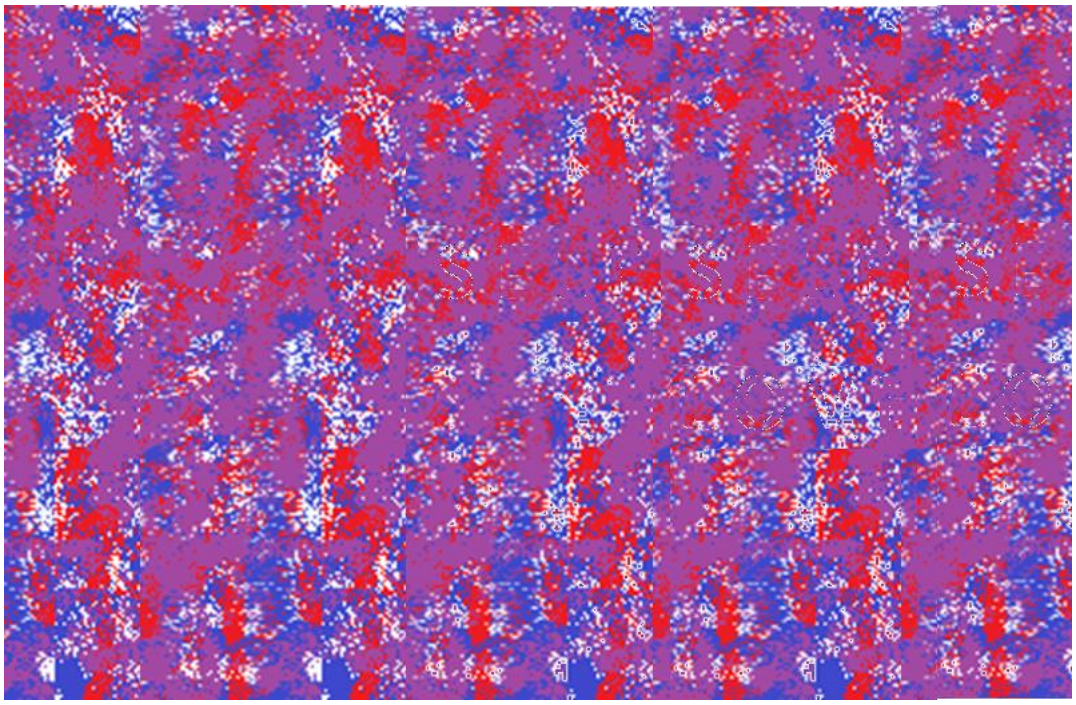
3.4 Autostereogramme mit Paint

Am darauffolgenden Tag gelang es Eileen Schmieger mit der Grafiksoftware „Paint“ ein Autostereogramm zu konstruieren.



Hier finden Sie einen verbesserten Versuch von Eileen Schmieger auf Paint ein Autostereogramm zu erstellen.





4.Stereogramme

4.1 fotografierte Stereogramme

Der entscheidende Unterschied zwischen Stereogrammen und Autostereogrammen ist, dass Stereogramme aus zwei verschiedenen Bildern bestehen, die man mit der „cross-eyed“ Technik betrachten muss, um das eigentliche Bild zu erkennen.

Sophia Neukirchner hat am dritten Tag eine Mandarine in einem Stereogramm in Szene gesetzt.


Daraufhin hat sie ihre Technik verbessert, als sie mit einem Spielzeugauto experimentiert hat.

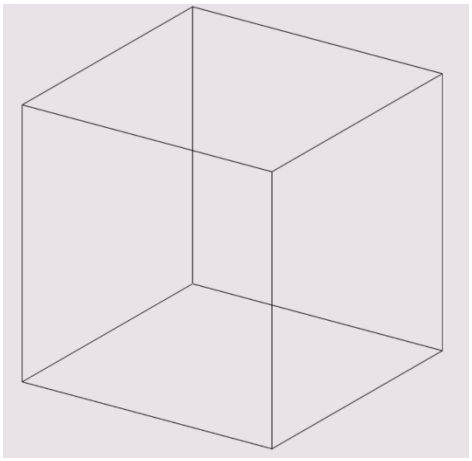

Sie hat die beiden Bilder mit Hilfe von Matlab mit der „wiggle stereoscopy“-Methode dargestellt, die dazu beitragen soll, die Tiefe des Bildes hervorzubringen, dargestellt. Um diesen Effekt zu erzielen, sollten die Bilder allmählich ident sein, jedoch ist das linke Bild eher weiter unten und das rechte eher weiter oben.

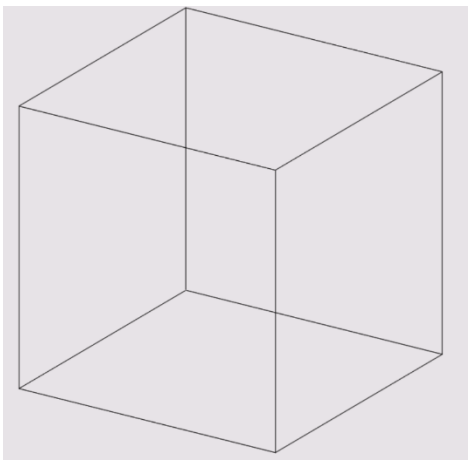
```
r=imread('zug4.jpg');
imagesc(r); axis image; axis off;
h=rot90(r);
g=rot90(h);
y=rot90(g);
imagesc(y); axis image; axis off;

s=imread('zug3.jpg');
imagesc(s); axis image; axis off;
k=rot90(s);
m=rot90(k);
n=rot90(m);
imagesc(n); axis image; axis off;

for k=1:100
    imagesc(y); axis image; axis off; drawnow;
    imagesc(n); axis image; axis off; drawnow;
end
```

Scan me!

4.2 grafisch dargestellte Stereogramme

4.2.1 Würfel





```
r=imread('Würfel1.jpg');
imagesc(r); axis image; axis off;
h=rot90(r);
g=rot90(h);
y=rot90(g);
imagesc(y); axis image; axis off;

s=imread('Würfel3.jpg');
imagesc(s); axis image; axis off;
k=rot90(s);
m=rot90(k);
n=rot90(m);
imagesc(n); axis image; axis off;

for k=1:100
    imagesc(y); axis image; axis off; drawnow;
    imagesc(n); axis image; axis off; drawnow;
end
```

Scan me!

4.2.2 Digital Gezeichnete Landschaft

Um selbst ein Stereogramm Paar zu zeichnen, fügt man erst Zeichnungen oder Bilder, die in verschiedenen Abständen zum Betrachter stehen auf unterschiedlichen Ebenen ein. Diese Ebenen gibt man in eine Gruppe und dupliziert sie. In der duplizierten Gruppe verrückt man alle Gegenstände verhältnismäßig zu ihrem Abstand zum Betrachter in eine Richtung. Objekte die näher am Betrachter sind werden mehr verrückt als Objekte, die sich weiter weg befinden.

Scan me!

Scan me!

Scan me!

Scan me!

```
h1 = figure(1); close(h1); h1 = figure(1);
set(h1,'Position',[10 40 1024 512]);

B = imread('bella.png');
m = size(B,1); n = size(B,2); l = size(B,3);

B = double(B);
M = (B(:,:,3) < 250);
B(:,:,1) = B(:,:,1).*M;
B(:,:,2) = B(:,:,2).*M;
B(:,:,3) = B(:,:,3).*M;

d = B(:,:,1);
d = round(20*(d - min(d(:)))/(max(d(:)) - min(d(:))));
```

Scan me!

```
x1 = B(:,:,1);
for i=1:m
    for j=1:n
        x1(i,j) = x1(i,mod(j-1+d(i,j),n)+1);
    end
end
x2 = B(:,:,1);
for i=1:m
    for j=n:-1:1
        x2(i,j) = x2(i,mod(j-1-d(i,j),n)+1);
    end
end

y1 = B(:,:,2);
for i=1:m
    for j=1:n
        y1(i,j) = y1(i,mod(j-1+d(i,j),n)+1);
    end
end
```

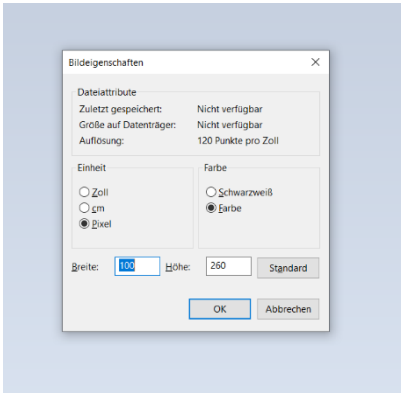
```
y2 = B(:,:,2);
for i=1:m
    for j=n:-1:1
        y2(i,j) = y2(i,mod(j-1-d(i,j),n)+1);
    end
end

z1 = B(:,:,3);
for i=1:m
    for j=1:n
        z1(i,j) = z1(i,mod(j-1+d(i,j),n)+1);
    end
end
z2 = B(:,:,3);
for i=1:m
    for j=n:-1:1
        z2(i,j) = z2(i,mod(j-1-d(i,j),n)+1);
    end
end
```

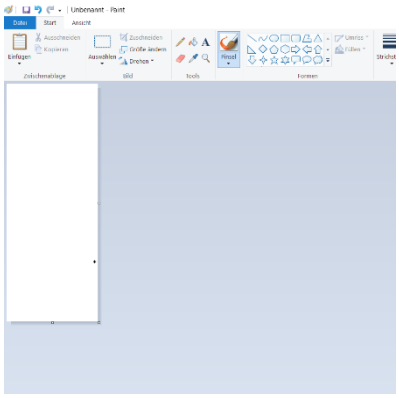
```
X = zeros(m,n,3);
X(:,:,1) = x1; X(:,:,2) = y1; X(:,:,3) = z1;
X = 255*(X - min(X(:)))/(max(X(:)) - min(X(:)));
X = uint8(X);

Y = zeros(m,n,3);
Y(:,:,1) = x2; Y(:,:,2) = y2; Y(:,:,3) = z2;
Y = 255*(Y - min(Y(:)))/(max(Y(:)) - min(Y(:)));
Y = uint8(Y);

subplot(1,2,1)
imagesc(X); axis image; axis off;
subplot(1,2,2)
imagesc(Y); axis image; axis off;
```

5.Tutorial

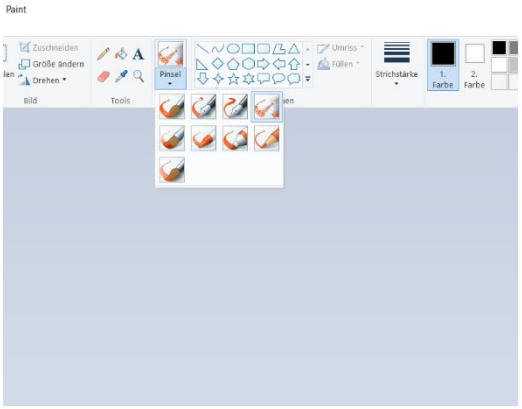
Damit es auch Ihnen möglich ist selbst ein Autostereogramm zu erstellen, haben wir hier eine Anleitung erfasst, in der Schritt für Schritt erklärt wird, wie Sie ein Autostereogramm anfertigen können.

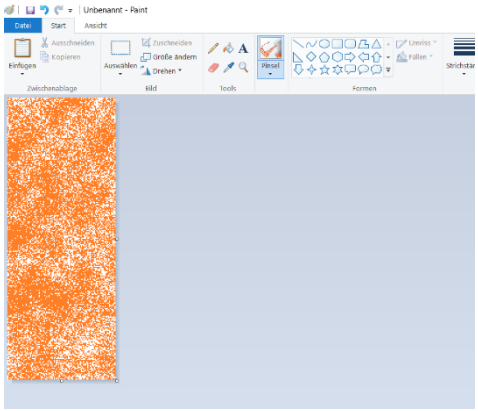
1. Öffnen Sie auf Paint ein leeres, weißes Blatt Papier und ändern Sie dessen Größe auf 100x260 Pixel (Das können Sie mit Hilfe des Shortcuts STRG + E).



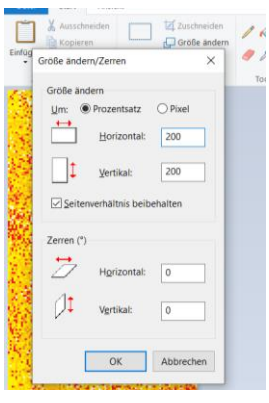


2. Verwenden Sie nun das Tool „Airbrush“ in einer Farbe, die nicht weiß ist, um den typischen Autostereogramm Look zu erzeugen.

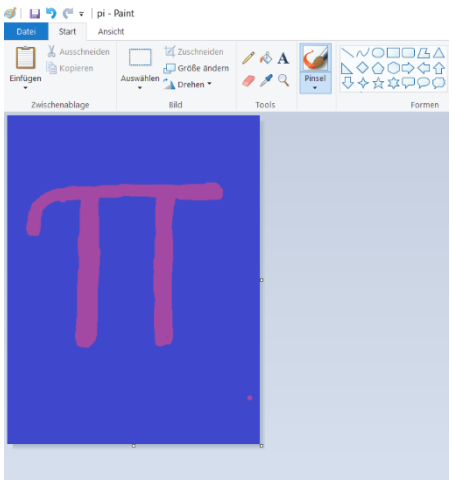




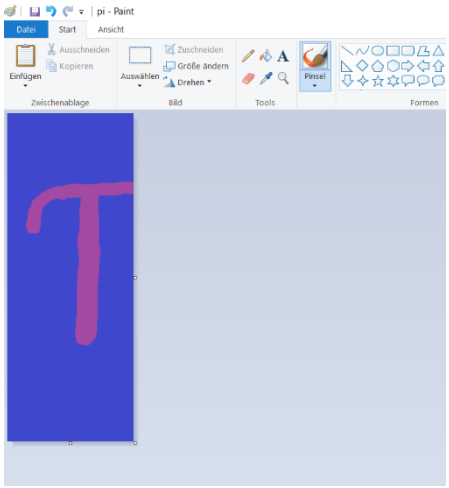
3. Wiederholen Sie das mit drei oder mehr Farben. Verdoppeln Sie nun die Größe (Dies können Sie mit dem Shortcut STRG + W) und bemalen Sie es erneut mit den Farben.

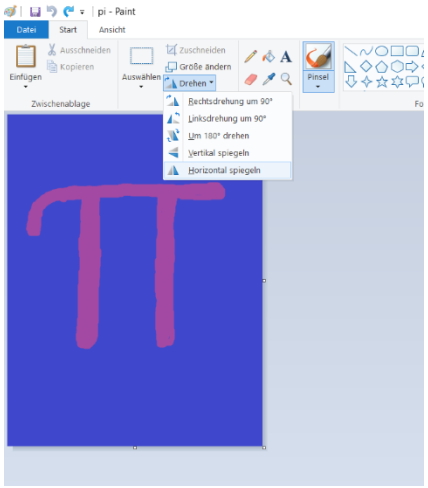


4. Speichern Sie dieses Bild nun als „Streifen1.png“. Öffnen Sie nun ein neues Paint Fenster und ändern Sie die Größe des Bildes auf 400x520 Pixel. Zeichnen Sie dann die Figur, die Sie im Autostereogramm sehen möchten. Verwenden Sie dafür eine Farbe für den Hintergrund und eine für den Vordergrund. Beide dürfen in den von ihnen zuvor verwendeten Farben nicht dabei sein. Damit die Verarbeitung später funktioniert, ist essenziell, dass Sie ausschließlich das „Pinsel“ Tool verwenden.

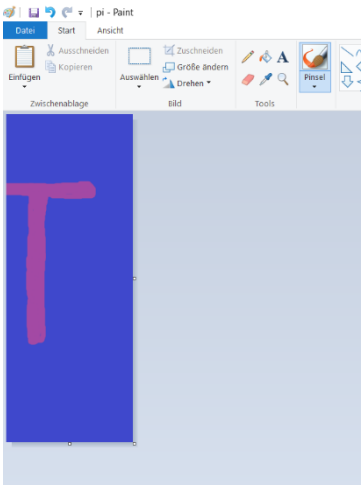


5. Öffnen Sie das gezeichnete in einem neuen Fenster und ändern Sie die Breite auf 200 Pixel. Dadurch wird die rechte Hälfte abgeschnitten. Nun gehen Sie zurück zur originalen Zeichnung und drehen Sie diese horizontal.

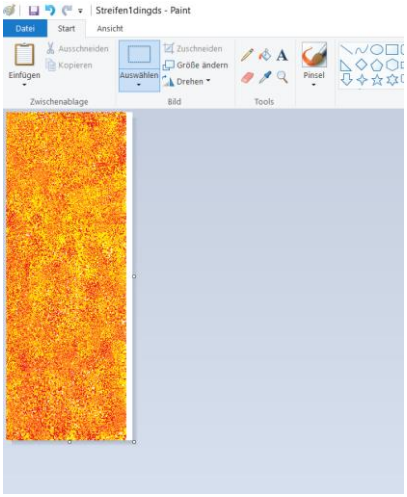


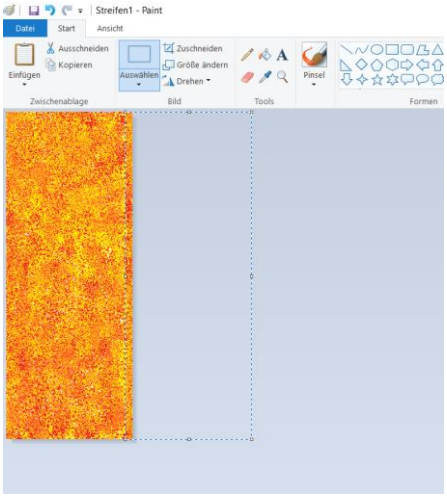


6. Nun ändern Sie auch bei der originalen Zeichnung, die Sie gerade gedreht haben, die Breite auf 200. Drehen Sie diese daraufhin erneut.

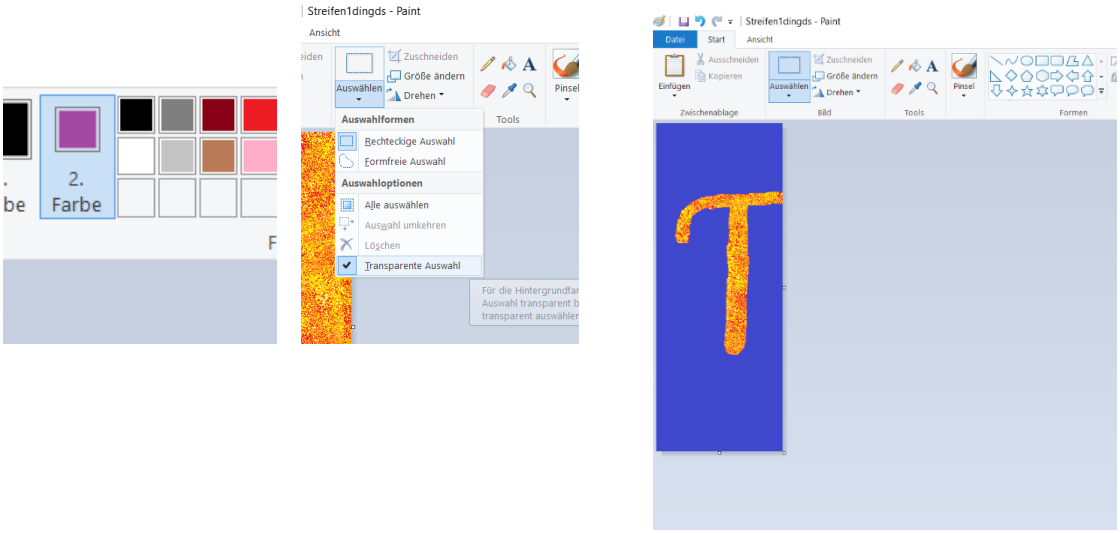


7. Öffnen Sie nun ein neues Paint Fenster. Die Größe bei diesem und auch bei den kommenden neuen Fenstern sollte immer 200x520 Pixel betragen, außer es wird explizit anders erwähnt. Kopieren Sie „Streifen1.png“ in dieses Fenster und bewegen Sie das gesamte Bild um 10 Pixel nach links. Dieser Schritt erzeugt später den Effekt des Autostereogramms. Fügen Sie danach an die rechte Seite des Bildes ein zweites Exemplar der Datei „Streifen1.png“ an, um die 10 Pixel weite Lücke zu schließen.



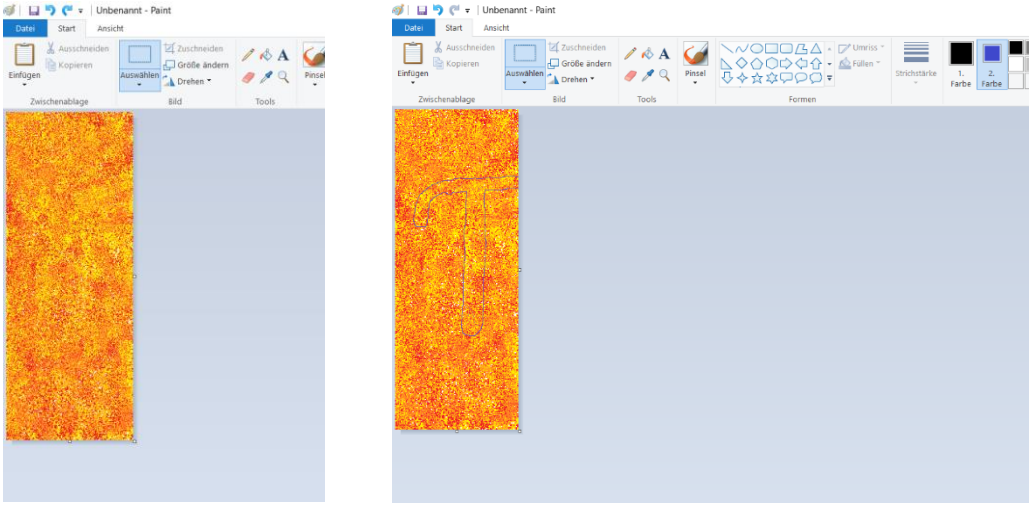


8. Für den nächsten Schritt müssen Sie zunächst das Feld „Farbe 2“ auf die Farbe der Umrisse ihrer Zeichnung ändern (In diesem Fall wäre es violett). Wählen Sie danach unter „Auswählen“ die „Transparente Auswahl“ aus und fügen Sie die linke Hälfte Ihres Bildes ein. Wichtig ist dabei, dass Sie es nicht als gespeichertes Dokument einfügen, sondern die in Paint geöffnete Zeichnung zur Gänze kopieren und im anderen Paint Fenster einfügen.

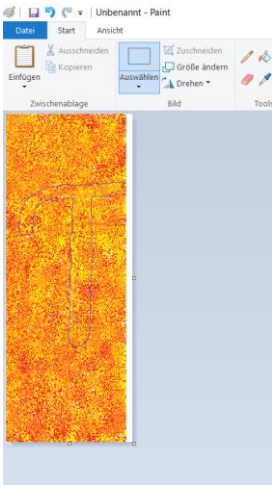
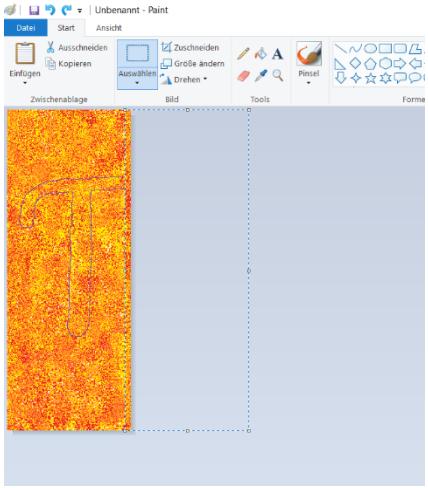
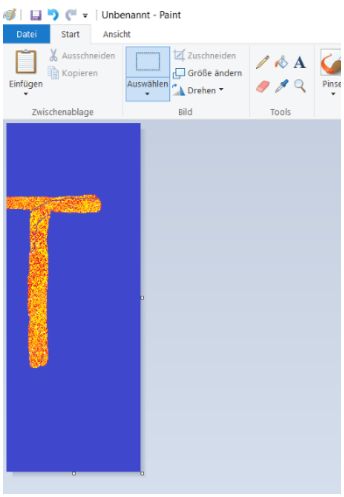


links Pi transparent

9. Öffnen Sie nun ein neues Paint Fenster und fügen Sie den Inhalt der Datei „Streifen1.png“ ein. Ändern Sie nun das Feld „Farbe 2“ auf die Hintergrundfarbe Ihrer Zeichnung (In diesem Fall wäre das blau) und wählen Sie erneut „Transparente Auswahl“ aus. Kopieren Sie nun die bereits transparente linke Hälfte ihrer Zeichnung (in diesem Fall die Abbildung „links Pi transparent) und fügen Sie diese in das neue Fenster zu „Streifen1.png“ ein. Speichern Sie dieses Bild als „Streifen2.png“.

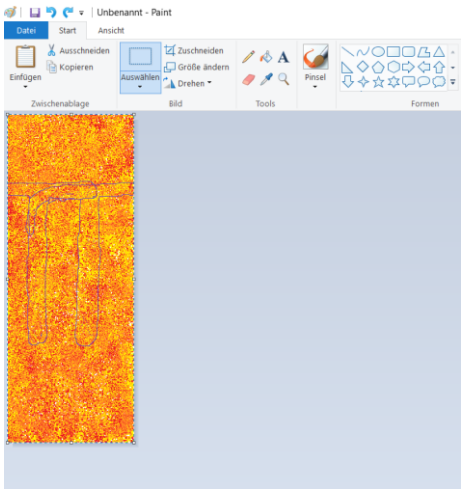


10. Öffnen Sie nun ein neues Paint Fenster und kopieren Sie den Inhalt der Datei „Streifen2.png“ ein und bewegen Sie es 10 Pixel nach links und fügen Sie direkt daneben ein weiteres Exemplar der Datei „Streifen2.png“ ein und füllen Sie die Lücke zwischen den beiden. Wählen Sie nun die Farbe Ihrer Zeichnung im Feld „Farbe 2“ aus (In diesem Fall violett) und wählen Sie erneut „Transparente Auswahl“ aus. Fügen Sie dann die rechte Hälfte Ihrer Zeichnung ein.

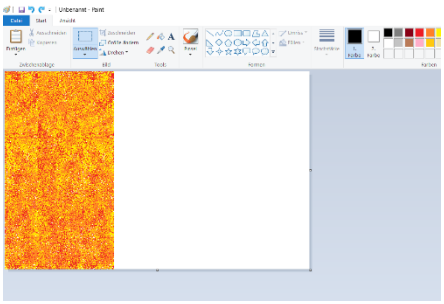




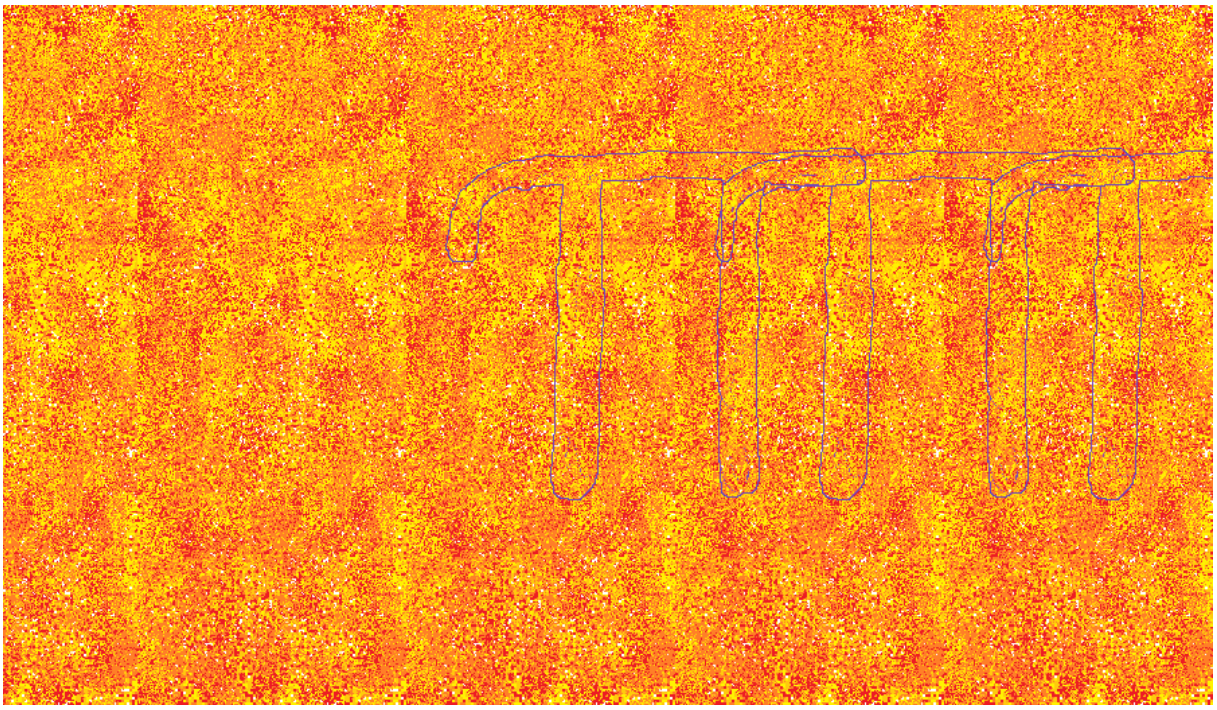
rechte Seite pi

11. Öffnen Sie nun ein neues Paint Fenster und fügen Sie den Inhalt der Datei „Streifen2.png“ ein. Ändern Sie das Feld „Farbe 2“ auf die Hintergrundfarbe Ihrer Zeichnung (in diesem Fall blau) und wählen Sie „Transparente Auswahl“ aus. Fügen Sie darüber den gesamten Inhalt des zuvor bearbeiteten Fensters (Abbildung „rechte Seite Pi“) ein. Speichern Sie dieses als „Streifen3_4.png“.

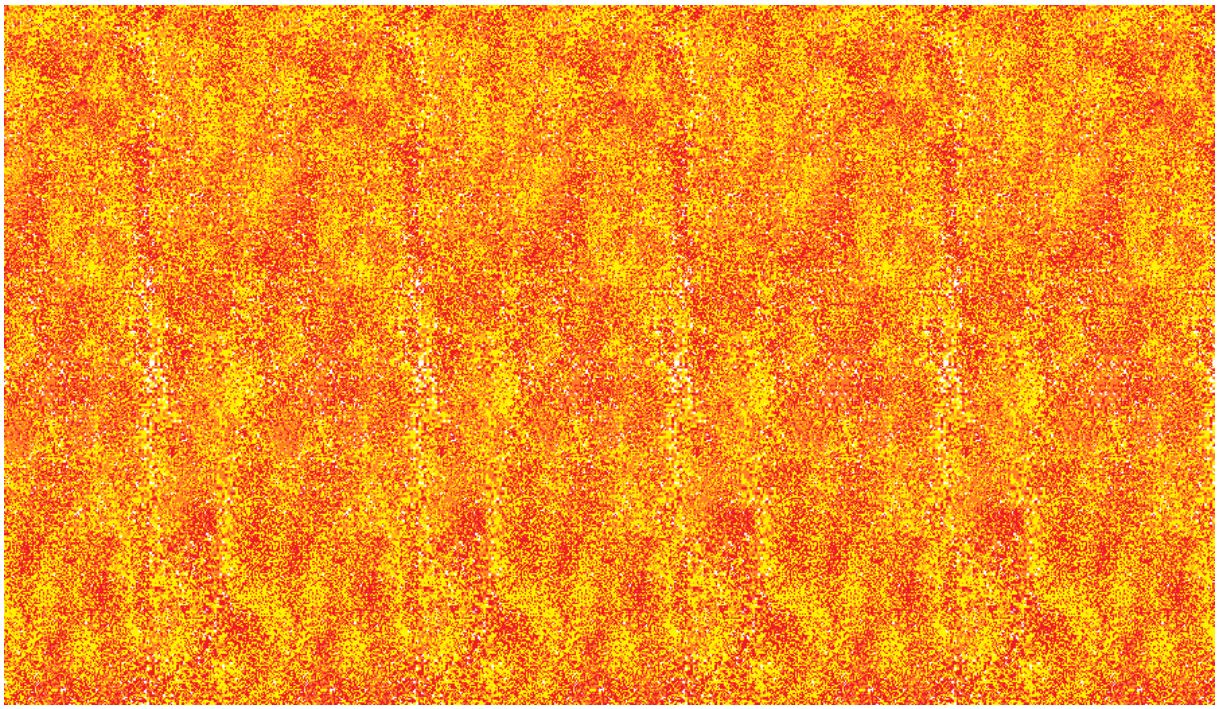


12. Öffnen Sie ein letztes Paint Fenster und ändern Sie die Größe auf 900x520 Pixel. Fügen Sie ganz links „Streifen1.png“ ein und verschieben Sie diese nach links bis circa die Hälfte davon zu sehen ist. Fügen Sie direkt daneben ein weiteres Exemplar der Datei „Streifen1.png“ ein. Daneben Streifen2.png“ und letztlich zwei Mal nebeneinander die Datei „Streifen3_4.png“.





Sollten Sie für Ihre Zeichnung eine Farbe gewählt haben, die sich stark von den Hintergrundfarben unterscheidet, könnte es passiert, dass wie in diesem Fall ein leichter Rand entsteht, der das Bild unsauber macht. Sie sollen zwar Farben nehmen, die sich unterscheiden, sie sollten jedoch nicht zu unterschiedlich sein. Verwendet man ähnlichere Fragen, sieht das so aus:



Ein großes Dankeschön an unseren Betreuer Steve für die interessante Zusammenarbeit! Vielen Dank für deine Erklärungen und dass du immer verschiedene Wege gesucht und gefunden hast, damit wir alle die (Auto)-Stereogramme sehen und erstellen können! Es war sehr spannend, mit dir an diesem Thema zu forschen und wir nehmen im Alltag auch immer wieder „Autostereogramme“ wahr – Stichwort Decke.

Wir möchten uns auch bei Herrn Professor Frühmann und allen anderen für die Organisation dieser Modellierungswoche bedanken!