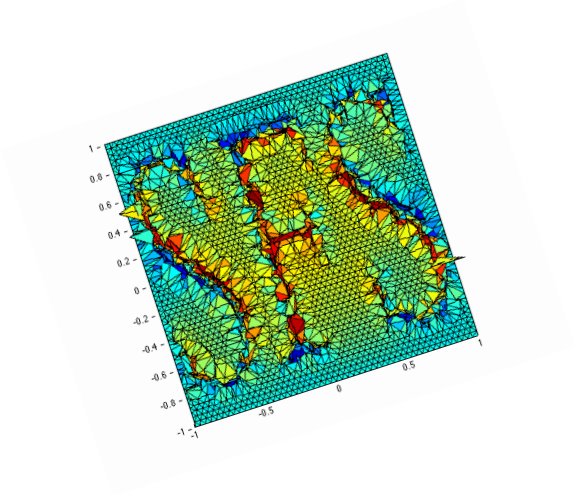
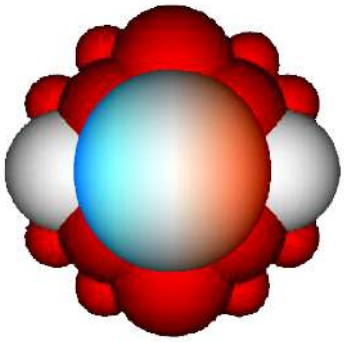
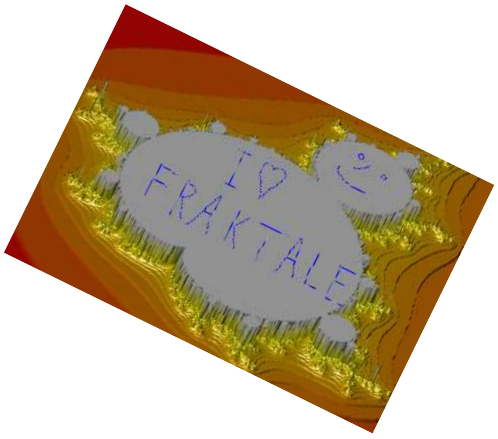




11. WOCHE DER MODELLIERUNG MIT MATHEMATIK



Dokumentationsbroschüre 7.2. – 13.2.2015

WOCHE DER MODELLIERUNG MIT MATHEMATIK

PÖLLAU, 7.2.–13.2.2015

WEITERE INFORMATIONEN:

HTTP://MATH.UNI-GRAZ.AT/MODELLWOCHE/2015/

ORGANISATOREN UND SPONSOREN

KOORDINATION

Mag. Michaela Seiwald

Alexander Sekkas

Mag. DDr. Patrick-Michel Frühmann

Vorwort

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

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

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

Wir danken dem Landesschulrat für Steiermark, und hier insbesondere Herrn Landesschulinspektor Mag. Gerhard Sihorsch, für die Hilfe bei der Organisation und seine kontinuierliche Unterstützung der Idee einer

Modellierungswoche. Ohne den idealistischen, unentgeltlichen und engagierten Einsatz der direkten Projektbetreuer Dr. Dipl.-Math.techn. Tobias Breiten, Dr. Dipl.-Math.techn. Florian Kruse, Michael Kniely, BSc BSc MSc MSc und Dr. Laurent Pfeiffer – Institut für Mathematik und Wissenschaftliches Rechnen – 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. Michaela Seiwald für die tatkräftige Hilfe bei der organisatorischen Vorbereitung, und Herrn Alexander Sekkas für die Hilfe bei der Betreuung der Hard- und Software.

Finanzielle Unterstützung erhielten wir von der Karl-Franzens-Universität Graz durch Vizerektor Prof. Dr. Martin Polaschek und Dekan Prof. Dr. Karl Crailsheim, vom regionalen Fachdidaktikzentrum für Mathematik und von Comfortplan.

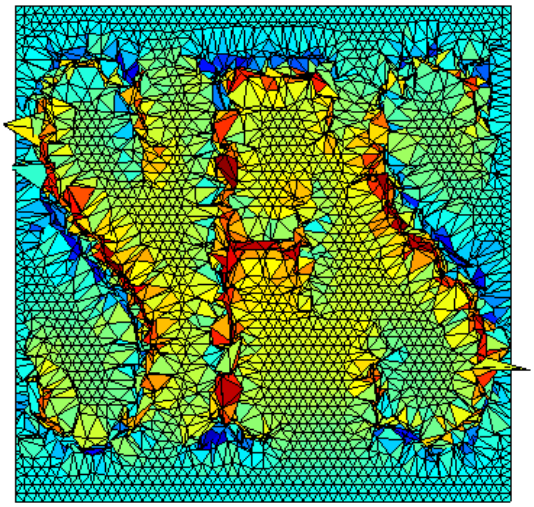
Pöllau, am 13. Februar 2015

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

Inverse Probleme

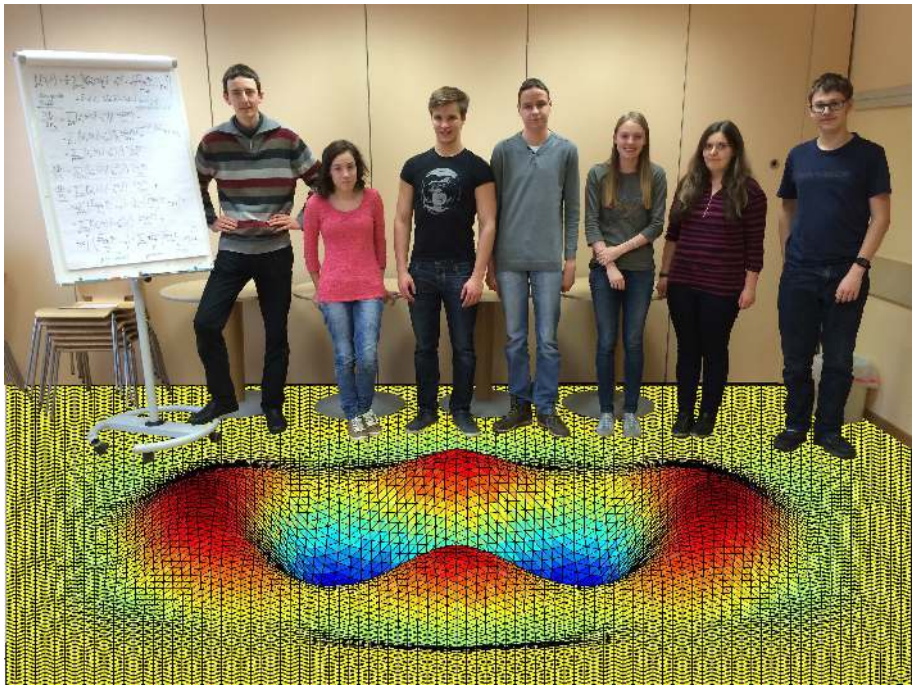
Shape From Shading — Vom Schattenbild zur Oberfläche

Tobias Dünser, Nina Maria Lampl, Alexander Rührnößl, Julia Stadlmann, Michael Karl Steinbauer, Victoria Zeiler



Betreuer: Michael Kniely BSc BSc MSc MSc

Inhaltsverzeichnis



1 Shape-From-Shading Problem

Das Shape-From-Shading Problem zählt zu den inversen Problemen, welche sich mit der Ursache einer beobachteten Wirkung auseinandersetzen. Das bedeutet, dass man von gegebenen Output-Werten auf die zugrundeliegenden Input-Werte schließt. Bei der Shape-From-Shading Problemstellung soll aus einem Schattenbild die Oberfläche rekonstruiert werden. Für diese Aufgabenstellung bietet sich MATLAB an, um die gesuchte Oberfläche zu generieren. Am besten eignen sich dafür triangulierte Rekonstruktionen. Eine Triangulierung besteht aus Dreiecken, die die Oberfläche darstellen. Dreiecke werden aufgrund der Tatsache, dass sie bei komplexen Oberflächen effizienter zu bearbeiten sind als Polygone, verwendet. Abb. 1 zeigt zwei Beispiele für mögliche Triangulierungen.

Abbildung 1: Triangulierungen eines Rechtecks und eines Kreises

1.1 Funktionen zur Minimierung

Um ein angemessenes Ergebnis zu erhalten, sollten die aktuellen Schattenwerte möglichst genau mit den gegebenen übereinstimmen. Die unten näher besprochene Funktion f_1 berücksichtigt auch die z-Komponenten der einzelnen Punkte, damit Zacken in den rekonstruierten Oberflächen unterdrückt werden (siehe Abb. 2). Weniger gezackten Rekonstruktionen werden durch f_1 also kleinere Werte zugewiesen.

Anders als bei der Funktion f_1 widmet sich eine zweite Funktion f_2 nicht den Differenzen von z-Komponenten, sondern summiert — neben der Summe über die Schattenwertdifferenzen — über die Differenzen aller benachbarten Normalvektoren. Dies liefert ähnliche Ergebnisse, wobei geringe Abweichungen festzustellen sind. Bei einer optimalen Rekonstruktion wäre demnach die Summe über die Differenz der Schattenwerte Null, jedoch ist dies in der Realität nicht umsetzbar. Werte zwischen 0.3 und 0.7 weisen bei den von uns betrachteten Problemen auf eine gut rekonstruierte Oberfläche hin und sind auch realisierbar.

Die Gewichtung der Schattenwerte in Bezug auf die Differenzen der z-Komponenten bzw. die Differenzen der Normalvektoren hängt von einer Variablen α ab, die je nach Problem größer oder kleiner gewählt werden kann. Demnach werden verschiedene α-Werte getestet, um einen passenden Wert zu erhalten. Je kleiner die Variable ist, desto mehr liegt der Fokus auf den Schattenwerten.

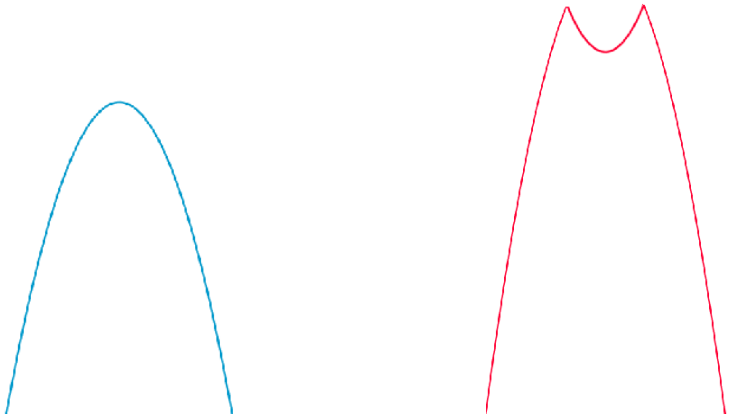


Abbildung 2: „Gute“ und „schlechte“ Funktionen

1.2 Schattenwerte berechnen

Um die Schattenwerte berechnen zu können wird ein normierter Normalvektor der Oberfläche benötigt. Sein Skalarprodukt mit dem Lichtvektor ergibt die Schattenwerte. Der Lichtvektor steht orthogonal auf die zugrunde liegende Ebene und wird als konstanter Vektor betrachtet (siehe Abb. 3). Auch die Beleuchtung des Objekts mit einer schrägen Lichtquelle wäre theoretisch realisierbar, jedoch wurde das Augenmerk auf andere Schwerpunkte gelegt und somit diesem Aspekt keine große Bedeutung beigemessen.

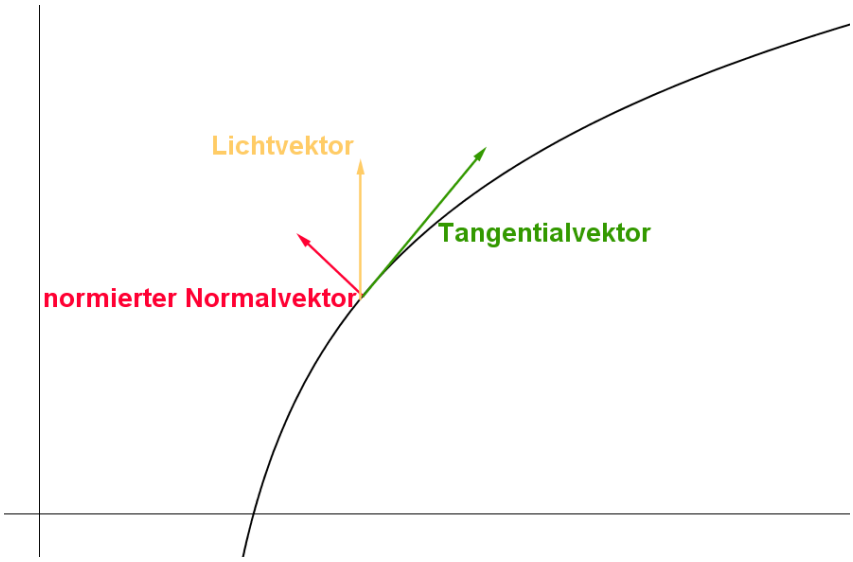


Abbildung 3: Lage von Tangential-, Normal- und Lichtvektor auf einer Oberfläche

Da die Punkte der triangulierten Oberfläche jedoch nicht immer auf dem gegebenen Punktenetz der Schattenwerte liegen, wird hierfür der Mittelwert der vier umliegenden Punkte gewählt. In Abb. 4 kann man einen in rot gezeichneten Punkt sehen, an welchem man den Schattenwert berechnen kann, indem man den Mittelwert der Schattenwerte an den benachbarten, blauen Punkten berechnet.

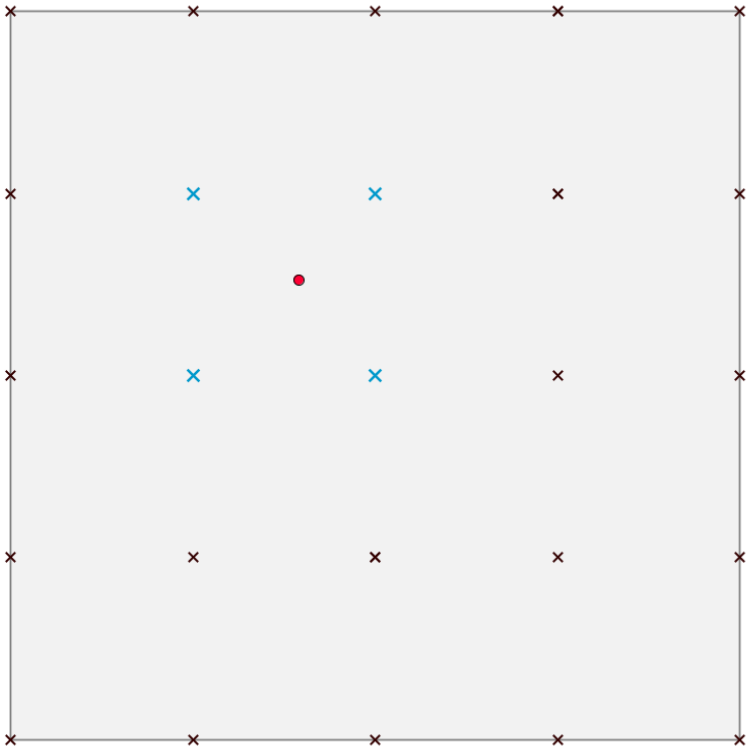


Abbildung 4: Gitter der gegebenen Schattenwerten

1.3 Weitere Triangulierungen

Mit den gegebenen Parametern lassen sich nicht nur eine sondern auch mehrere Triangulierungen herstellen, sodass die Oberfläche am Ende weniger Zacken aufweist. Dazu werden die neuen z-Komponenten der feiner triangulierten Oberfläche aus den alten z-Komponenten der gröber triangulierten Oberfläche durch Mittelung bestimmt. Dadurch entsteht ein noch feineres Netz aus Dreiecken. Bei den beiden Bildern in Abb. 5 erkennt man den Unterschied zwischen einer gröberen und einer feineren Triangulierung. Anschaulich gesprochen wird die feinere über die gröbere Triangulierung gelegt, um Zacken auszugleichen und die Darstellung der Rekonstruktion zu glätten.

2 Gradientenverfahren

Der Gradient einer Funktion f, die triangulierte Oberflächen auf reelle Zahlen abbildet, ist der Vektor bestehend aus den partiellen Ableitungen von f nach x_1, y_1, z_1 bis x_N, y_N, z_N, wobei N die Anzahl an Punkten auf der Oberfläche ist. Dieser zeigt in die Richtung des steilsten Anstiegs der Funktion f. Da aber das globale Minimum oder zumindest ein passendes lokales Minimum gesucht wird, benötigt man den negativen Gradienten, der die Oberfläche dahingehend modifiziert, dass die Funktionswerte kleiner werden.

Abbildung 5: Eine grobe und eine feine Triangulierung eines Rechtecks

2.1 Algorithmus zum Gradientenverfahren

Der hier verwendete Algorithmus ist das Verfahren des steilsten Abstiegs. Zunächst wird ein Anfangspunkt gewählt, an dessen Stelle der Gradient berechnet wird. Es folgt die Bestimmung der passenden Schrittweite in Richtung des negativen Gradienten. Den nächsten Punkt erhält man, indem man vom aktuellen Punkt mit der eben bestimmten Schrittweite in Richtung des negativen Gradienten geht. An diesem Punkt wird der neue Gradient berechnet. Dieses Verfahren wird so lange wiederholt bis der Betrag des Gradienten einen gewissen Wert unterschreitet oder ein Maximum an Iterationen erreicht ist. In diesem Fall folgt der Abbruch der Berechnungen und im besten Fall ist das globale Minimum lokalisiert worden. Je nach Wahl des Startpunktes wird also entweder das globale oder ein lokales Minimum berechnet. Dies wird in Abb. 6 veranschaulicht.

2.2 Passende Schrittweite

Die Ermittlung der passenden Schrittweite erfolgt durch ein Liniensuchverfahren. Damit das Gradientenverfahren effektiv ist, sollen die Armijo- und die Goldstein-Bedingung erfüllt sein: Der Funktionswert von f am nächsten Punkt der Iteration muss zwischen den beiden unten dargestellten Geraden liegen (siehe Abb. 7). Eine Variable, die im Allgemeinen mit dem griechischen Buchstaben σ bezeichnet wird, beschreibt wie stark die Armijo-Gerade fällt, während eine zweite Variable, meist als μ deklariert, die negative Steigung der Goldstein-Gerade bestimmt.

Die Erfüllung dieser Bedingungen wird folgendermaßen erreicht: Zuerst werden zwei Variablen α_1 und α_2 die Werte 0 beziehungsweise 1 zugewiesen. Dann wird α_2 so lange vergrößert bis die Goldstein-Bedingung erfüllt ist. Danach folgt die Ermittlung der potentiellen Schrittweite $\alpha = (\alpha_1 + \alpha_2)/2$. Liegt dieser Wert zwischen den Geraden, wurde eine passende Schrittweite berechnet. Wenn α die Armijo-Bedingung nicht erfüllt, wird α_2 auf α gesetzt. Falls α jedoch die Goldstein-Bedingung nicht erfüllt, wird α_1 auf α gesetzt. Nun wird α wieder wie oben berechnet. Dies wird so lange wiederholt bis beide Bedingungen erfüllt sind.

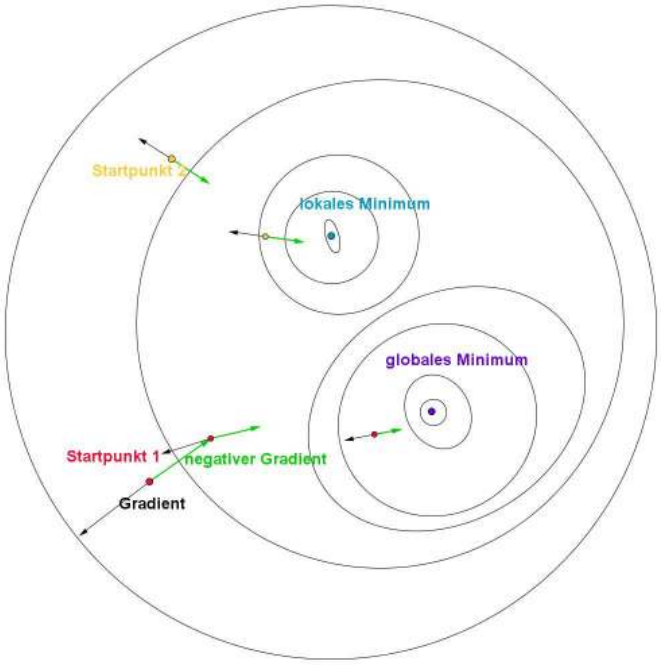


Abbildung 6: Abhängig vom Startpunkt kann das Verfahren des steilsten Abstiegs unterschiedliche lokale Minima erreichen.

3 Minimierungsfunktionen und deren Gradienten

Die folgenden Formeln werden verwendet, um der jeweiligen triangulierten Fläche einen Wert zuzuweisen, der bei hoher Übereinstimmung mit dem Schattenbild möglichst niedrig ist. Die beiden Varianten unterscheiden sich in der Weise, wie sie die Beschaffenheit der Oberfläche berücksichtigen.

- Erste Variante:

$$f_1(P) = \frac{1}{2} \sum_{p \in P} \left((s_p - s_p^*)^2 + \frac{\alpha}{2} \left(\frac{\sum_{q \in N(p) \setminus p} z_q}{m_p} - z_p \right)^2 \right)$$

- Zweite Variante:

$$f_2(P) = \frac{1}{2} \sum_{p \in P} \left((s_p - s_p^*)^2 + \frac{\alpha}{2} \left(\sum_{q \in N(p) \setminus p} |\overrightarrow{n_p}(P) - \overrightarrow{n_q}(P)|^2 \right) \right)$$

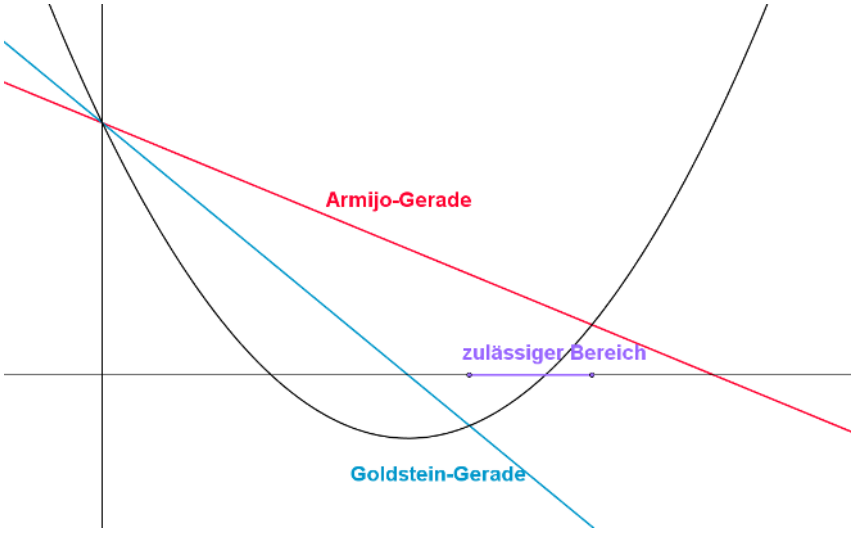


Abbildung 7: Zur Definition von Armijo- und Goldsteinbedingung

3.1 Diskussion der Funktionen

Bei beiden Varianten werden bestimmte positive Größen über alle Punkte der trianguliertenOberfläche summiert. Der erste Teil berücksichtigt dabei den Unterschied zwischen den gegebenen Schattenwerten und jenen der aktuellen Rekonstruktion, während der zweite die Beschaffenheit der Oberfläche bewertet. Im Folgenden betrachten wir die einzelnen Terme näher und berechnen deren Ableitung nach den Koordinaten der einzelnen Punkte.

(a) $\hat{f}(P) = \frac{1}{2}\sum_{p\in P}(s_p - s_p^*)^2$

Der erste Teil der Funktion ist für beide Varianten gleich. s_p gibt hier den Schattenwert der triangulierten Oberfläche an. Dieser wird aus dem Skalarprodukt zwischen dem Normalvektor zur Oberfläche und der Richtung des einfallenden Lichts berechnet, $s_p = \overrightarrow{n_p} \cdot \overrightarrow{l}$. s_p^* ergibt sich aus dem korrelierenden Wert des vorgegebenen Schattenbildes.

Im Idealfall verschwindet die Differenz der beiden Werte. Ist dies nicht der Fall, wird die Differenz quadriert um einen positiven Wert zu garantieren. Wir verwenden hierbei nicht den Betrag, da dies zu späteren Komplikationen bei der Ableitung der Funktion führen könnte. Der Faktor $1/2$ vor der Summe fällt nach dem Ableiten weg.

(b) $\tilde{f}_1(P) = \frac{1}{2}\sum_{p\in P}\frac{\alpha}{2}\left(\frac{\sum_{q\in N(p)\setminus p} z_q}{m_p} - z_p\right)^2$

Dieser Teil hingegen wird nur von der ersten Variante verwendet. Um zu vermeiden, dass die Rekonstruktion der Oberfläche des Schattenbildes zu viele Zacken und Falten aufweist, wird eine glatte Oberfläche besser bewertet. Hier geschieht dies, indem die Höhenkoordinate eines Punktes mit denen seiner Nachbarn verglichen wird.

z_q ist die z-Koordinate eines Nachbarpunktes. Die Summe aller z_q wird durch m_p dividiert, welches die Anzahl der an den jeweiligen Punkt angrenzenden Nachbarpunkte darstellt. z_p, die z-Koordintate des betroffenen Punktes, wird von der Summe abgezogen und die Differenz wird quadriert. α, eine reelle Zahl, ist hierbei ausschlaggebend, wie stark geglättet wird.

(c) $$\tilde{f}_2(P) = \frac{1}{2} \sum_{p \in P} \left(\frac{\alpha}{2} \sum_{q \in N(p) \setminus p} |\overrightarrow{n_p}(P) - \overrightarrow{n_q}(P)|^2 \right)$$

Bei der zweiten Variante werden Normalvektoren anstelle von z-Koordinaten zur Bewertung der Oberfläche herangezogen. Ganz gleich wie bei der ersten Variante dient dieser Teil der Funktion dazu die Oberfläche abzuglätten.

$\overrightarrow{n_p}$ ist der Normalvektor auf den jeweiligen Punkt, während $\overrightarrow{n_q}$ den Normalvektor auf einen benachbarten Punkt darstellt. Die Summe der Betragsquadrate von Differenzen benachbarter Normalvektoren wird in dieser Variante der Funktion dazu verwendet um festzustellen, wie glatt die rekonstruierte Oberfläche ist.

Als Code in MATLAB implementiert sieht die erste Variante der Minimierungsfunktion, f_1, folgendermaßen aus:

```
function val = f1(box, neibtri, edge, u, ...
                  light, shadepts, alpha)

npts = length(u) / 3;
val = 0;

for i = 1:npts
    val = val + (light' * nvec(neibtri, u, i) -...
                 shades(box, shadepts, u(3 * i - 2:3 * i)))^2;

    mwz = 0;
    for j = 1:2:length(neibtri{i})
        mwz = mwz + u(3 * neibtri{i}(j));
    end
    mwz = 2 * mwz / length(neibtri{i});

    val = val + alpha * (mwz - u(3 * i))^2;
end

val = val / 2;
```

3.2 Berechnung der Gradienten

Um den Gradienten für die Anwendung des bereits beschriebenen Gradientenverfahrens zu erhalten, müssen die beiden Varianten der Funktion abgeleitet werden. Der Gradient ist die partielle Ableitung nach $x_1, y_1, z_1, ..., x_N, y_N, z_N$. Der Schattenwert eines Punktes, der zuvor mit s_p bezeichnet wurde, ist das Skalarprodukt des Normalvektors $\overrightarrow{n_p}(P)$ und des Lichtvektors $\overrightarrow{l}$.

(a) Für die Ableitung von $\hat{f}(P)$ ergibt sich

$$\begin{aligned}\frac{\partial \hat{f}(P)}{\partial r_k} &= \frac{\partial}{\partial r_k}\left(\frac{1}{2}\sum_{p\in P}\left(\overrightarrow{n_p}(P)\cdot\overrightarrow{l}-s_p^*\right)^2\right)\\ &= \sum_{p\in P}\left(\overrightarrow{n_p}(P)\cdot\overrightarrow{l}-s_p^*\right)\cdot\left(\overrightarrow{l}\cdot\frac{\partial\overrightarrow{n_p}(P)}{\partial r_k}\right).\end{aligned}$$

Die Ableitung von $\hat{f}$ tritt sowohl in der Ableitung von f_1 als auch in jener von f_2 auf. Zuerst wurde die Kettenregel angewandt. r_k ist die Variable, nach der abgeleitet wird, wobei r_1 eine x-Komponente, r_2 eine y-Komponente und r_3 eine z-Komponente ist. Nun wird nach r_k abgeleitet. n_p wird aus der Summe der Kreuzprodukte der Vektoren in Richtung der Nachbarpunkte berechnet. Ist ein Punkt p also kein Nachbar von r oder r selbst, so ist n_p unabhängig von r_k und die Ableitung somit Null. Zusammen mit der Menge $N(r)$ bestehend aus r und den Nachbarn von r folgt nun

$$\frac{\partial \hat{f}(P)}{\partial r_k} = \sum_{p\in N(r)}\left(\overrightarrow{n_p}(P)\cdot\overrightarrow{l}-s_p^*\right)\cdot\left(\overrightarrow{l}\cdot\frac{\partial\overrightarrow{n_p}(P)}{\partial r_k}\right).$$

(b) Da $\tilde{f}_1(P)$, der zweite Teil der Funktion f_1, ausschließlich die z-Koordinaten berücksichtigt, ist

$$\frac{\partial \tilde{f}_1(P)}{\partial r_1} = \frac{\partial \tilde{f}_1(P)}{\partial r_2} = 0$$

für alle $r \in P$. Für die Ableitung nach $r_3 \equiv z_r$ müssen die folgenden drei Fälle bzgl. $p \in P$ unterschieden werden.

Fall 1: $p = r$: Wird nach z_p abgeleitet, so gilt

$$\frac{\partial}{\partial r_3}\left(\frac{\alpha}{2}\left(\frac{\sum_{q\in N(p)\setminus p} z_q}{m_p}-z_p\right)^2\right) = (-1)\cdot\alpha\cdot\left(\frac{\sum_{q\in N(r)\setminus r} z_q}{m_r}-z_r\right).$$

Fall 2: $p \in N(r)\setminus r$: Wird nach der z-Koordinate eines Nachbarpunktes von p abgeleitet, so ist die Ableitung nach z_r von z_p gleich Null, ebenso wie für z_q, falls q ein Nachbar von p ungleich r ist. Dies liefert

$$\frac{\partial}{\partial r_3}\left(\frac{\alpha}{2}\left(\frac{\sum_{q\in N(p)\setminus p} z_q}{m_p}-z_p\right)^2\right) = \frac{\alpha}{m_p}\cdot\left(\frac{\sum_{q\in N(p)\setminus p} z_q}{m_p}-z_p\right).$$

Fall 3: $p \notin N(r)$: Ist r kein Nachbar von p, so ist

$$\frac{\partial}{\partial r_3}\left(\frac{\alpha}{2}\left(\frac{\sum_{q\in N(p)\setminus p} z_q}{m_p}-z_p\right)^2\right) = 0.$$

In Summe ergibt sich daher

$$\begin{aligned}\frac{\partial \tilde{f}_1(P)}{\partial r_3} &= \frac{\partial}{\partial r_3}\left(\frac{1}{2}\sum_{p\in P}\frac{\alpha}{2}\left(\frac{\sum_{q\in N(p)\setminus p} z_q}{m_p} - z_p\right)^2\right) \\ &= -\alpha\cdot\left(\frac{\sum_{q\in N(r)\setminus r} z_q}{m_r} - z_r\right) + \sum_{p\in N(r)\setminus r}\frac{\alpha}{m_p}\cdot\left(\frac{\sum_{q\in N(p)\setminus p} z_q}{m_p} - z_p\right).\end{aligned}$$

(c) Die Ableitung von $\tilde{f}_2(P)$ nach r_k ist gegeben durch

$$\begin{aligned}\frac{\partial \tilde{f}_2(P)}{\partial r_k} &= \frac{\partial}{\partial r_k}\left(\frac{1}{2}\sum_{p\in P}\left(\frac{\alpha}{2}\sum_{q\in N(p)\setminus p}|\overrightarrow{n_p}(P) - \overrightarrow{n_q}(P)|^2\right)\right) \\ &= \frac{\alpha}{2}\sum_{p\in P}\sum_{q\in N(p)\setminus p}\left(\overrightarrow{n_p}(P) - \overrightarrow{n_q}(P)\right)\cdot\left(\frac{\partial\overrightarrow{n_p}(P)}{\partial r_k} - \frac{\partial\overrightarrow{n_q}(P)}{\partial r_k}\right).\end{aligned}$$

Der wesentliche Punkt ist nun die Berechnung der Ableitung eines beliebigen Normalvektors $\overrightarrow{n_p}$ nach r_k mit $r \in P$ und $k \in \{1,2,3\}$ fest gewählt. Ähnlich wie bei der Funktion $\tilde{f}_1$ hängen Normalvektoren nur von den Koordinaten benachbarter Punkte ab. Es gilt daher

$$\frac{\partial\overrightarrow{n_p}(P)}{\partial r_k} = 0$$

falls p kein Nachbarpunkt von r ist. Da die Berechnung der Ableitung in den restlichen Fällen sehr technisch ist, genauso wie die Implementierung in MATLAB, konnten wir hier auf bereits bestehende MATLAB-Funktionen unseres Betreuers zurückgreifen.

4 Ergebnisse

4.1 Einfluss der Parameter

Der Optimierungsprozess der Oberfläche hängt nicht nur von den einzelnen Funktionen f_1 und f_2 ab, sondern mindestens gleich stark von dessen Parametern, wie α, σ, μ, der Auflösung und der Zahl an Schritten. Die Möglichkeit einer allgemeinen Aussage über die Qualität der erzeugten Oberfläche besteht nicht, deshalb ist es notwendig für jedes einzelne Schattenbild die Parameter neu zu wählen um eine Optimierung des Ergebnisses zu erhalten. Ein paar allgemeine Aussagen über die Auswirkung von Änderungen der Parameterwerte auf den Optimierungsprozess kann man dennoch treffen.

4.1.1 Glättungsparameter α

Wählt man ein höheres α, so erhält man eine glattere Oberfläche, sprich die Differenz zwischen den einzelnen z-Koordinaten (bei f_1) oder Normalvektoren (bei f_2) wird kleiner. Dennoch wird damit auch der Effekt erzielt, dass die Schattenwerte an Priorität verlieren. Somit entsteht bei einem zu hoch gewählten α eine zu glatte Oberfläche, welche sehr einer

Ebene ähneln kann. Ist das α nur etwas zu hoch gewählt, so erhält man eine Oberfläche, bei der die Aus- und Einbuchtungen nicht rund, sondern abgeflacht sind. Bei einem zu niedrig gewählten α erhält man eine zerknüllte Oberfläche, sprich eine Oberfläche mit großer Differenz zwischen z-Koordinaten bzw. Normalvektoren, jedoch passen hier die eingelesenen Werte des Schattenbildes sehr gut. Ist das α etwas zu niedrig gewählt, sieht man einige Unebenheiten und Zacken in der generierten Oberfläche. Das optimale α impliziert eine relativ glatte Oberfläche, welche zu einem hohen Grad mit den Werten des eingelesenen Schattenbildes übereinstimmt. Da die Funktion f_1 mit demselben α wie f_2 hohe Differenzen aufweist, müssen die α-Werte für die Funktion f_1 meist höher gewählt werden. Bei der Funktion f_1 erwiesen sich Werte für α in der Nähe von 50 als gut, bei der Funktion f_2 resultierten Werte nahe an 0.1 in guten Rekonstruktionen der Schattenbilder. Somit erweist sich bei f_1 meist ein α-Wert, welcher 500-mal höher ist als bei f_2, als gut.

4.1.2 Armijo-Parameter σ

Erhöht man den Wert für σ in der Armijo-Bedingung, so schränkt man typischerweise die Schrittweite ein. Ausgangspunkt ist bei unserem Beispiel ein σ von 0.1. Erhöht man dieses σ benötigt man meist mehr Iterationen um dieselbe Änderung der Oberfläche zu erhalten, da die Schrittweite eingeschränkt wird. Hier ist es notwendig den richtigen Wert zu finden um nicht zu wenig Änderung an der Oberfläche zu erhalten und um nicht zu viel Zeit mit vielen Iterationen zu verlieren, da die Schritte bei einem hohen σ relativ klein sind. Aber σ sollte auch nicht zu klein sein, um die Oberfläche nicht durch zu große Schrittweiten zu ungenau zu rekonstruieren.

4.1.3 Goldstein-Parameter μ

Ein kleineres μ liefert zumeist eine größere Schrittweite. Somit sind weniger Iterationen notwendig um dieselbe Änderung an der Oberfläche hervorzurufen. Andererseits kann man auch kleinere Schrittweiten und somit schwächere Änderungen an der Oberfläche erlauben, falls ein größeres μ gewählt wird. Wie bei σ ist es hier notwendig den richtigen Wert zu finden, da sonst die Oberfläche kaum deformiert wird bzw. der hohe Deformationsgrad nicht produktiv ist und nicht zu einer optimalen Oberfläche führt. Weiters ist es notwendig den μ-Wert größer als den σ-Wert zu wählen, da sonst die Armijo-Goldstein-Bedingungen nicht erfüllt werden können.

4.1.4 Auflösung

Ändert man die Auflösung, so erhält man eine höhere oder niedrigere Dichte an Dreiecken in der Triangulation. Somit wird bei einer höheren Auflösung eine höhere Rechenleistung, und daher auch mehr Zeit benötigt, da die Koordinaten für mehrere Dreiecke berechnet werden müssen. Zuerst ist es ratsam mit einer Auflösung von 0.1 zu rechnen, damit nicht zu viel Rechenzeit benötigt wird um eine erste grobe Übersicht für die Parameter zu bekommen und diese anzupassen. Falls man nun bereits in dieser groben Auflösung erkennt, dass die restlichen Parameter nicht passen, kann man das Programm bereits abbrechen, bevor man weitere Rechenzeit mit einer größeren Auflösung verschwendet. Danach sollte man die Auflösung in kleineren Schritten anpassen, jedoch entstehen mit einer höheren Auflösung auch mehrere unnatürliche Plateaus. Um nicht ein zu hohes α wählen zu

müssen und somit die Relevanz der Schattenwerte zu verlieren, ist es ratsam im Zuge der Verfeinerung der Triangulierung eine Mittelung über z-Komponenten von Punkten in einer gewissen Umgebung in den Code zu implementieren. Im Folgenden ist der Code des Hauptprogramms dargestellt.

```
clear, clc

%% Definition der Parameter
alpha = 10;
sigma = 0.1;
my = 0.5;
light = [0; 0; 1]; light = light/norm(light);
resol = [.1,.07,0.05,0.04];
npoints = zeros(1,length(resol));

%% Berechnen bzw. Einlesen von Schattenbildern
a = [-1:.01:1];
b = a;
[aa, bb] = meshgrid(a, b);

zz = zeros(size(aa));
shadepts = ones(size(zz));
synth = @(x, y) (1+4*(x^2+y^2)-5*(x^2+y^2)^2)/10;

for i = 1:size(zz, 1)
    for j = 1:size(zz, 2)
        x = aa(i, j);
        y = bb(i, j);

        if (x^2+y^2<=1)
            zz(i, j) = synth(x, y);
            fx = (4*x-10*x*(x^2+y^2))/5;
            fy = (4*y-10*y*(x^2+y^2))/5;
            shadepts(i, j) = [-fx, -fy, 1]*...
                             light/sqrt(fx^2 + fy^2 + 1);
        end

    end
end

figure(2), clf,
surf(aa, bb, zz, 64*shadepts + 1);
shading interp, colormap gray, axis equal, view ([10 20]);

%% for-Schleife ueber die verschiedenen Aufloesungen
for h = 1:length(resol)

%% Erzeugen der Triangulierung
figure(1), clf
fd = inline('dcircle(p, 0, 0, 1)','p');
box = [-1, -1; 1, 1];
fix = [-1, -1; 1, -1; 1, 1; -1, 1];
[pts, tri] = distmesh2d(fd, @huniform, resol(h), box, fix);
[pts, tri] = fixmesh(pts, tri);
```

```
50 npoints(h)=size(pts,1);
51
52 %% Berechnen der Nachbarschaftsbeziehungen der Punkte
53 npts = size(pts, 1);
54 ntri = size(tri, 1);
55 boundpts = zeros(npts, 1);
56 boundpts(unique(boundedges(pts,tri))) = 1;
57
58 neibtri = cell(npts, 1);
59 neibpts = cell(npts, 1);
60 edge = cell(npts,1);
61
62 k = ones(npts, 1);
63 for i = 1:ntri
64     for j = 1:3
65         l = tri(i, j);
66         neibtri{l}(k(l)) = tri(i,mod(j,3)+1);
67         neibtri{l}(k(l)+1) = tri(i,mod(j+1,3)+1);
68         k(l) = k(l) + 2;
69     end
70 end
71
72 for i = 1:npts
73     neibpts{i} = unique(neibtri{i});
74     edge{i} = neibpts{i}(neibpts{i}>i);
75 end
76
77 %% Berechnen der initialen bzw. verfeinerten Oberflaechen
78 if (h == 1)
79     rep = 15
80     u = zeros(3*npts, 1);
81     u(1:3:end) = pts(:, 1);
82     u(2:3:end) = pts(:, 2);
83     u(3:3:end) = (1 - (pts(:,1).^2+pts(:,2).^2))/10;
84 else
85     rep = 30
86     alpha = 15
87     u0 = zeros(npts,1);
88
89     for i = 1:npts
90         if boundpts(i) == 0
91             temp = 0;
92             n = 0;
93             for j = 1:npoints(h-1)
94                 if norm(pts(i,:) - u(3*j-2:3*j -1)') < 3*resol(h)
95                     temp = temp + u(3*j);
96                     n = n+1;
97                 end
98             end
99             u0(i) = temp/n;
100        end
101    end
102
103    u = zeros(3*npts,1);
104    u(1:3:3*npts) = pts(:,1);
105    u(2:3:3*npts) = pts(:,2);
```

```
106     for i = 1:npts
107         if boundpts(i) == 0
108             temp = 0;
109             n = 0;
110             pneib = neibpts{i};
111             nneib = length(pneib);
112             for j = 1:nneib
113                 temp = temp+u0(pneib(j));
114             end
115             u(3*i) = temp/nneib;
116         end
117     end
118
119 end
120
121 %% Definition anonymer Funktionen fuer f und gradf.
122 ff = @(u) f1(box, neibtri, edge, u, light, shadepts, alpha);
123 gradff = @(u) gradf1(box, boundpts, neibtri, neibpts, u, ...
124                      light, shadepts, alpha);
125
126 u = abstieg(ff, gradff, box, tri, boundpts, neibtri, neibpts, light,...
127             shadepts, alpha, u, rep, 1e-3);
128 end
```

4.1.5 Minimierungsfunktionen f_1 und f_2

Die Funktion f_1 benötigt eine geringere Rechenleistung als die Funktion f_2, da sie nur die z-Koordinaten der einzelnen Punkte benötigt, während die Funktion f_2 länger braucht. Diese berechnet nämlich für jeden Punkt den Normalvektor zur Oberfläche aus den Normalvektoren auf die benachbarten Dreiecke. Der erste Ausdruck der Funktion von f_1 und f_2 ist identisch, somit unterscheiden sie sich nur durch den zweiten Teil. Da das Berechnen der Normalvektoren weit mehr Zeit benötigt als über die benachbarten z-Komponenten zu mitteln, ist es mit der Funktion f_1 möglich bei derselben Rechenzeit eine um ein Vielfaches höhere Auflösung zu erzielen, da mehr Iterationsschritte möglich sind. Obwohl die Funktion f_2 meist glattere Oberflächen liefert, erzeugt auch die Funktion f_1 manchmal bessere Resultate.

4.1.6 Iterationsschritte

Da bei einer Rekonstruktion der Oberfläche wie bei unserer der Gradient fast nie einen Wert erreichen wird, der sehr nahe an Null ist, ist die Anzahl der maximalen Iterationsschritte meist ausschlaggebend dafür, wie oft man sich in Richtung des negativen Gradienten bewegt. Im Idealfall sollte jeder Schritt näher zur ursprünglichen Oberfläche führen, in der Praxis ist dies jedoch nicht immer der Fall. Meistens resultieren diese Schritte darin, dass man sich einem lokalen Minimum nähert, manchmal allerdings wird die ursprüngliche Oberfläche mit einem weiteren Schritt weniger gut rekonstruiert als zuvor. Natürlich benötigt man für eine höhere Iterationsanzahl auch mehr Rechenzeit.

4.1.7 Minimaler Betrag des Gradienten

Wird der Gradient zu klein, kommt es zu keinem nennenswerten Fortschritt bei der Rekonstruktion der Oberfläche. Daher muss in den Code eine while-Schleife mit der Bedingung eingebaut werden, dass der Gradient einen bestimmten Wert nicht unterschreitet. Die im Folgenden dargestellte MATLAB-Funktion führt die Schleife über die verschiedenen Iterationen durch.

```
function u = abstieg(ff, gradff, box, tri, neibtri,...
                     light, shadepts, alpha, u0, maxit, maxgrad)

npts = length (u0/3);

u = u0;
h = 1;
s = 1;

gradientf = gradff(u)';

while (norm(gradientf) > maxgrad) && (h <= maxit) && (s >0)
    s = armgold(ff, box, neibtri, light, shadepts, alpha,...
                gradientf, u, -gradientf, .1, .5);
    u = u - s*gradientf;
    gradientf = gradff(u)';
    h = h + 1;
    plotshape(tri, u, 4, [10 20]);
end
```

4.2 Rekonstruierte Oberflächen

4.2.1 Beispiel 1

In dem folgenden Beispiel kann man erkennen, dass die Oberfläche durch das höhere α glatter ist und sich einer Ebene annähert. Nach mehreren Tests kamen wir zum Ergebnis, dass man mit $\alpha = 70$ und der Funktion f_1 die besten Ergebnisse erzielen kann. Bei der Rekonstruktion in Abb. 12 haben wir außerdem hintereinander sieben unterschiedliche Auflösungen verwendet, was zu diesem sehr nahe am Original liegenden Ergebnis geführt hat.

4.2.2 Beispiel 2

Bei dieser Oberfläche hat sich $\alpha = 0.1$ als ideal erwiesen; im Gegensatz zu den rekonstruierten Funktionen von zuvor wurde hier aber auch die Funktion f_2 verwendet, welche viel geringere α-Werte zulässt. Die Auflösung der Rekonstruktion in Abb. 14 ist — wie unschwer zu erkennen — geringer als bei unserer idealen Rekonstruktion von der ersten Oberfläche.

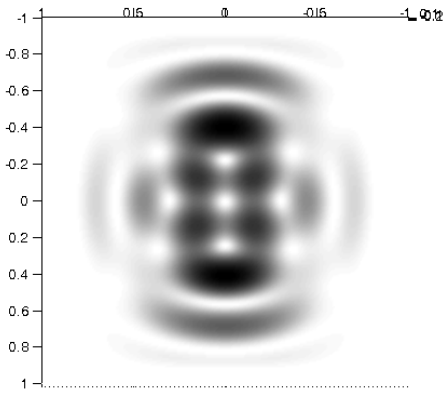


Abbildung 8: Schattenbild der Oberfläche 1 von oben

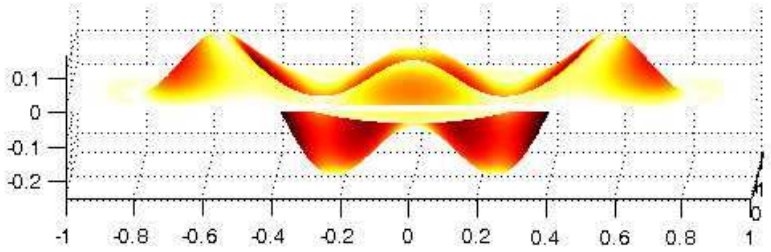


Abbildung 9: Oberfläche 1 von der Seite

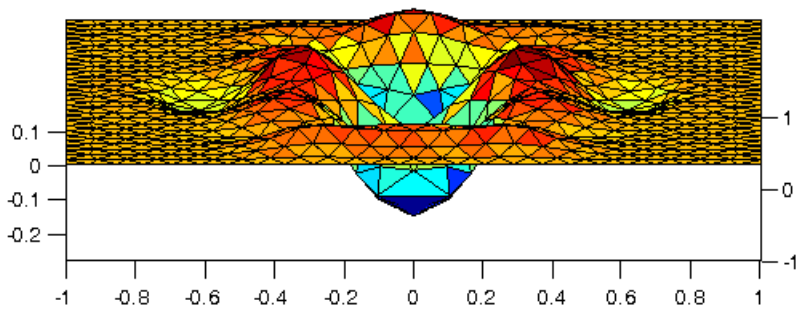


Abbildung 10: Oberfläche 1 mit f_1, $\alpha = 20$, $\sigma = 0.01$, $\mu = 0.5$. Schritte: 20, Auflösungen: 0.1, 0.08, 0.05

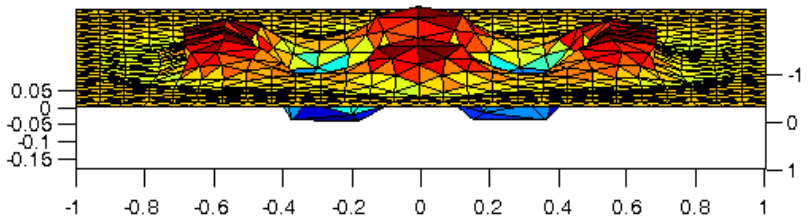


Abbildung 11: Oberfläche 1 mit f_1, $\alpha = 80$, $\sigma = 0.01$, $\mu = 0.5$. Schritte: 20, Auflösungen: 0.1, 0.08, 0.05

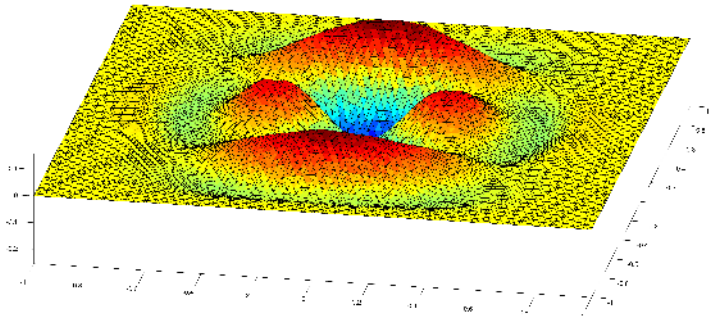


Abbildung 12: Oberfläche 1 mit f_1, $\alpha = 70$, $\sigma = 0.01$, $\mu = 0.5$. Schritte: 20, Auflösungen: 0.1, 0.08, 0.06, 0.05, 0.04, 0.033, 0.025

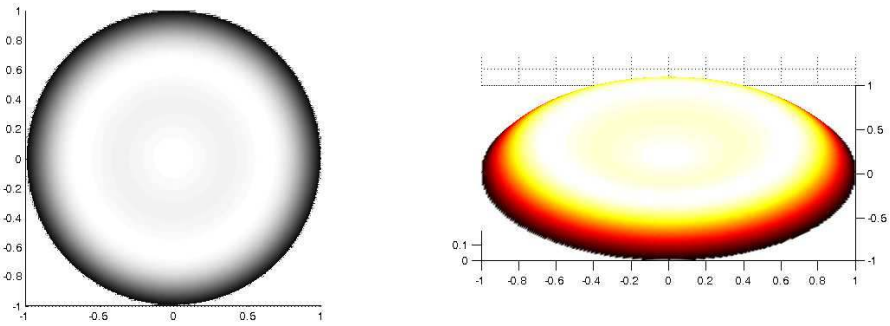


Abbildung 13: Schattenbild der Oberfläche 2 von oben

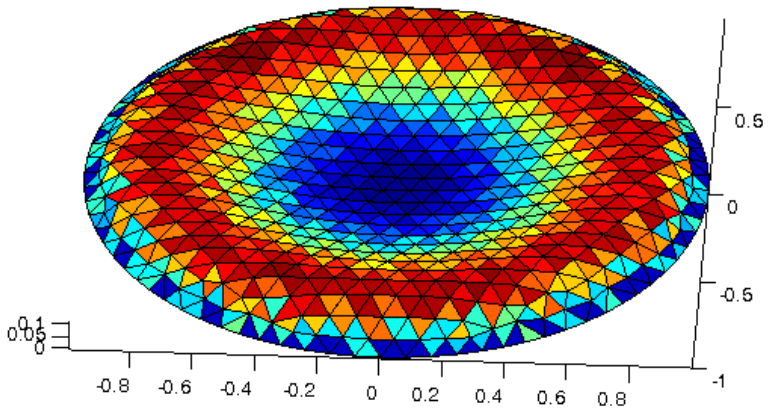


Abbildung 14: Oberfläche 2 mit f_2, $\alpha = 0.1$, $\sigma = 0.01$, $\mu = 0.05$. Schritte: 18, Auflösungen: 0.1, 0.075

Ingenieurstechnik:

Wie(so) funktioniert Segwayfahren?

Betreuer

Dipl.-Math. techn. Dr. Tobias Breiten

Team

David Haberl, Armin Lachini, Christoph Pfeifer, Maximilian Steiner, Matthias Wallner, Valentin Zsilavecz

07.-13. Februar 2015

Inhaltsverzeichnis

Problemstellung

Über die letzten Jahre ist der Segway-Personal-Transporter zu einem immer beliebteren Fortbewegungsmittel geworden. Gerade im Freizeitbereich, zum Beispiel bei touristischen Stadtführungen, erfährt er besondere Beachtung. Ganz vereinfacht dargestellt funktioniert die Steuerung eines Segways über Körperbewegungen der beförderten Person. Dabei stellt sich natürlich die grundlegende Frage, warum man beim Segwayfahren eigentlich nicht umfällt.

Ziele des Projekts:

- Lösen einfacherer Differentialgleichungen anhand der Modellierung einer Federschwingung mit verschiedenen Methoden
- Erstellen eines vereinfachten Modells, das es uns erlaubt, die Dynamik des Systems „Segway“ am Computer zu simulieren
- Auffinden der verschiedenen Ruhelagen
- Überprüfung der Stabilität der gefundenen Ruhelagen mithilfe der Eigenwerte von Matrizen
- Erstellen einer Simulation, die es ermöglicht, die Ruhelagen automatisiert anzusteuern

Numerisches Lösen von Differentialgleichungen

Um die Bewegung eines Segways überhaupt erst beschreiben zu können, ist es zunächst nötig, die Grundlagen von Differentialgleichungen zu verstehen. Grundsätzlich ist eine Differentialgleichung eine Gleichung, die sowohl eine Funktion, als auch deren Ableitung(en) enthält. Als Lösung erhält man keinen spezifischen Wert, sondern die Funktion selbst.

Wir beginnen mit einer einfachen Differentialgleichung erster Ordnung, die wir mithilfe von MATLAB numerisch lösen. Hierzu sind allerdings ein Startwert und ein Intervall für die x-Werte notwendig:

$$f'(x) = f(x)$$

$$f(0) = 3$$

$$x \in [0; 10]$$

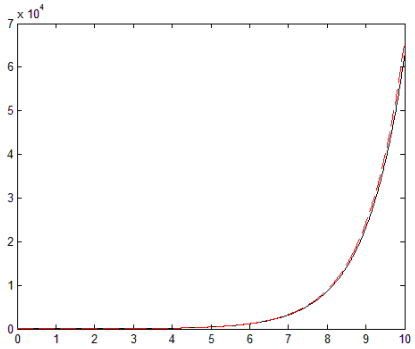
Mithilfe der Definition des Differenzenquotienten erhalten wir folgende Gleichungen:

$$\frac{f(t + \Delta t) - f(t)}{\Delta t} \approx f(t)$$

$$f(t + \Delta t) \approx f(t) * \Delta t + f(t)$$

Mit der letzten Gleichung können wir jeweils das nächste Folgenglied von f berechnen. Je kleiner der Wert von Δt ist, umso genauer wird auch die Näherungsfunktion. Implementiert man diese Überlegungen in Matlab, erhält man:

```
t=linspace(0,10,1000)
f(1)=3;
delta_t=t(2)-t(1)
for i=2:length(t);
  f(i)=f(i-1)*delta_t+f(i-1);
end
plot(t,f,'k')
hold on
plot(t,3*exp(t),'r--')
```



Unser Ergebnis stellt eine Annäherung an die Funktion $f(x) = 3e^x$. Das bedeutet, dass die Ableitung der Exponentialfunktion gleich ist wie die ursprüngliche Funktion.

Modellierung einer Federschwingung

Schwingung eines Federpendels ohne Dämpfung

Als nächstes versuchen wir, die Bewegungsgleichungen eines vorerst ungedämpften Federpendels aufzustellen. Wir stützen uns dabei auf das Hooke'sche Gesetz und das Newton'sche Gesetz als physikalische Grundlagen:

$$F = -y * k$$

$$F = m * a$$

mit:

- y = momentane Auslenkung
- k = Federkonstante
- m = Masse des Pendels
- a = Beschleunigung des Pendels
- F = Kraft

Des Weiteren gilt, dass die 2. Ableitung der Auslenkung des Pendels gleich dessen Beschleunigung und der 1. Ableitung seiner Geschwindigkeit ist:

$$y''(t) = v'(t) = a(t)$$

Durch Gleichsetzen und Umformen erhält man folgende Differentialgleichung 2. Ordnung:

$$m * y''(t) = -y(t) * k$$

Legt man k und m willkürlich auf den Wert „1" fest, ergibt sich:

$$y''(t) = -y(t)$$

Um die Gleichung numerisch lösen zu können, legen wir folgende Anfangsbedingungen fest:

$$x(0) = 10$$

$$x'(0) = v(0) = 0$$

Wie bei der Differentialgleichung 1. Ordnung nutzen wir wieder den Differenzenquotienten, um eine Lösung zu erhalten:

$$v'(t) = -x(t) \approx \frac{v(t+\Delta t) - v(t)}{\Delta t}$$

$$x'(t) = v(t) \approx \frac{x(t+\Delta t) - x(t)}{\Delta t}$$

Durch Umformen ergibt sich:

$$v(t+\Delta t) \approx -x(t) * \Delta t + v(t)$$

$$x(t+\Delta t) \approx v(t) * \Delta t + x(t)$$

Diese beiden Gleichungen können wir direkt in MATLAB implementieren und erhalten somit eine relativ gute Näherung.

Schwingung eines Federpendels mit Dämpfung:

Es ist zu beachten, dass die vorige Betrachtung relativ ungenau ist, da in der Realität immer auch eine Dämpfung des Pendels auftritt und somit miteinbezogen werden muss. Das erreichen wir dadurch, indem wir die Dämpfungskonstante d einführen, sodass die Kraft proportional zur Geschwindigkeit des Pendels ist. Infolgedessen lautet die neue Gleichung für die Pendelschwingung:

$$m * x''(t) + k * x(t) + d * x'(t) = 0$$

bzw.

$$m * v'(t) + d * v(t) + k * x(t) = 0$$

Durch Umformungen und neuerlichen Überlegungen bezüglich des Differenzenquotienten gelangen wir zu den beiden allgemeinen Gleichungen:

$$v(t+\Delta t) = \frac{\Delta t * (-d * v(t) - k * x(t))}{m} + v(t)$$

$$x(t+\Delta t) = v(t) * \Delta t + x(t)$$

Vergleich zweier Schwingungen mit bzw. ohne Dämpfung

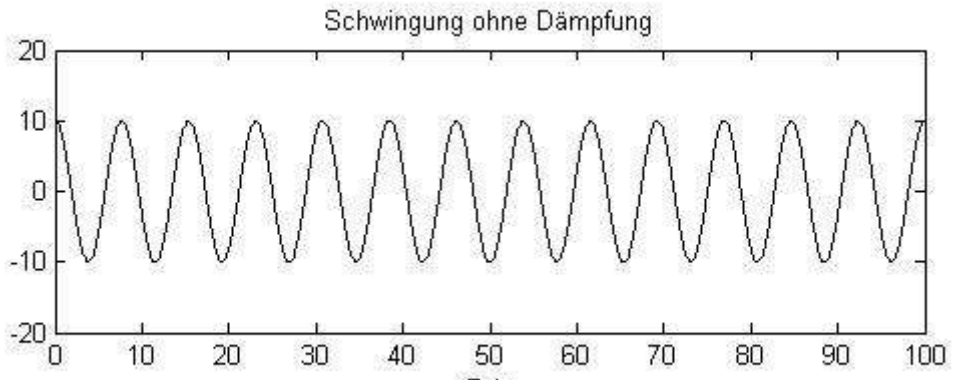


Abbildung 1

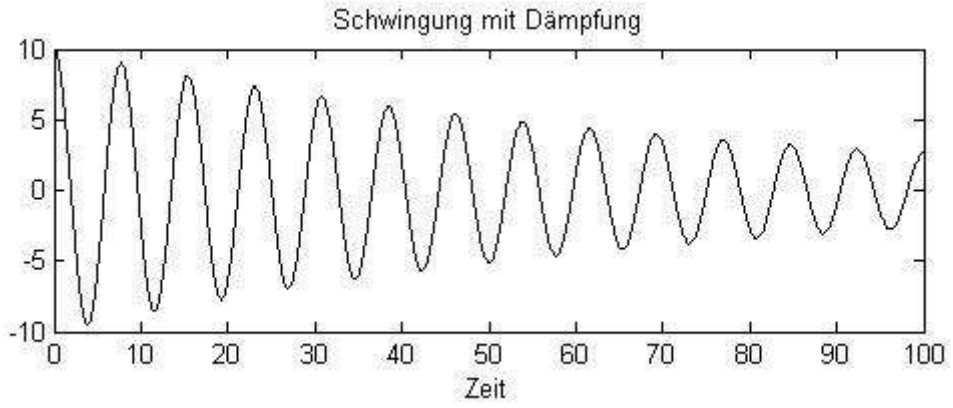


Abbildung 2

In beiden Abbildungen 1 und 2 wird die Auslenkung des Federpendels im Zeitintervall [0; 100] gezeigt, wobei die Federkonstante k mit 2 und die Masse m mit 3 angenommen werden. In Abbildung 2 wird zusätzlich eine Dämpfung d mit 0,1 angenommen. In der ersten Grafik zeigt sich, dass die Amplitude konstant bleibt, wobei sie bei der zweiten Grafik bei gleichbleibender Frequenz abnimmt und sich 0 annähert.

Matrixschreibweise

Für den vorigen Löser haben wir 2 Gleichungen benötigt. Allerdings ist es auch möglich, dieselbe Methode in Matrixdarstellung mit einer 2x2-Matrix darzustellen. Allgemein gilt für alle linearen Differentialgleichungen:

$$y' = Ay$$

In unserem Fall ist y ein 2x1-Vektor:

$$y = \begin{bmatrix} x \\ v \end{bmatrix}$$

Gemäß der Bewegungsgleichung für das Fadenpendel lautet die Matrix A:

$$A = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{d}{m} \end{bmatrix}$$

Natürlich muss auch hier ein Vektor mit den Anfangswerten vorhanden sein:

$$y(0) = \begin{bmatrix} x(0) \\ v(0) \end{bmatrix} = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$

Erneute Verwendung des Differenzenquotienten und anschließendes Umformen ergibt:

$$y(t + \Delta t) = Ay(t) * \Delta t + y(t)$$

Damit haben wir nur eine Gleichung, welche problemlos in MATLAB implementiert werden kann.

Implizites Euler-Verfahren

Nachdem wir anfangs immer unsere Differentialgleichungen mit dem expliziten Euler-Verfahren gelöst haben und schlussendlich feststellen mussten, dass die damit errechneten Ergebnisse zu ungenau waren, entschieden wir uns ein neues Lösungsverfahren zu verwenden.

Dabei handelt es sich um das implizite Euler-Verfahren, mit dem wir eine numerische Lösung für unsere Differentialgleichungen erhalten. Da hier ein implizites Verfahren angewandt wird, werden in jeder Iteration im Normalfall nicht-lineare Gleichungen gelöst.

Im Allgemeinen beschreibt man das implizite Euler-Verfahren auf diese Weise:

$$\frac{y_i - y_{i-1}}{\Delta t} = A * y_i$$

In folgender Gleichung

$$m * x_i'' + k * x_i + d * x_i' = 0$$

wird x'' durch v' ersetzt und anschließend nach v' umgeformt:

$$v_i' = -\frac{k}{m} * x_i - \frac{d}{m} * v_i$$

Jetzt werden die ursprüngliche Funktion sowie deren gerade besprochene Ableitung in Matrixschreibweise dargestellt:

$$\begin{bmatrix} \frac{x_i - x_{i-1}}{\Delta t} \\ \frac{v_i - v_{i-1}}{\Delta t} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{d}{m} \end{bmatrix} * \begin{bmatrix} x_i \\ v_i \end{bmatrix}$$

Durch Ausführen des Matrix-Vektor-Produktes erhalten wir die folgenden zwei Differentialgleichungen erster Ordnung:

$$\frac{x_i - x_{i-1}}{\Delta t} = v_i$$

$$\frac{v_i - v_{i-1}}{\Delta t} = -\frac{k}{m} * x_i - \frac{d}{m} * v_i$$

Zunächst formen wir die erste Gleichung nach x_i um und setzen die Umformung in die zweite Gleichung für x_i ein:

$$\frac{v_i - v_{i-1}}{\Delta t} = -\frac{k}{m} * (v_i * \Delta t + x_{i-1}) - \frac{d}{m} * v_i$$

Nach diversen Umformungsschritten kommen wir auf folgendes Ergebnis:

$$v_i = \frac{v_{i-1} * m - k * x_{i-1} * \Delta t}{m + k * \Delta t^2 + d * \Delta t}$$

Um unsere zweite Gleichung zu erhalten, müssen wir nun v_i in die ursprüngliche Gleichung von x_i einsetzen:

$$x_i = \frac{v_{i-1} * m - k * x_{i-1}}{\frac{m}{\Delta t} + k * \Delta t + d} + x_{i-1}$$

Diese zwei Gleichungen können wir nun in MATLAB eingeben, um sie zu lösen. Trotzdem ist diese Methode noch immer nicht optimal. Um sie zu optimieren, kombinieren wir das explizite und das implizite Euler-Verfahren zu einer neuen Methode namens Trapez-Verfahren.

Trapez-Verfahren

Aus den beiden vorhergegangenen Methoden entwickeln wir nun eine neue Variante der Gleichungslösung:

$$\frac{y_i - y_{i-1}}{\Delta t} = A * \frac{y_i + y_{i-1}}{2}$$

Damit MATLAB die Gleichung interpretieren kann, formen wir diese nach y_i um.

$$y_i = \left(2 * \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \Delta t * A\right)^{-1} * (\Delta t * A * y_{i-1} + 2 * y_{i-1})$$

Das Trapez-Verfahren hat die Konvergenzordnung 2, das heißt bei Verdopplung der Stützstellen wird die Genauigkeit der Approximation um das 4-fache erhöht. Im Gegensatz dazu haben das implizite und das explizite Euler-Verfahren die Konvergenzordnung 1, woraus sich ein weniger starker Anstieg der Genauigkeit ergibt.

Ruhelagen

Ruhelagen eines Fadenpendels

Ein Fadenpendel besteht aus einer fixen Achse, um die sich eine masselose Pendelstange dreht, auf der die Masse m angebracht ist. Der Winkel φ beschreibt die Auslenkung des Pendels von der Ruhelage. Die allgemeine Gleichung für die Bewegung eines Fadenpendels lautet, wie folgt:

$$\varphi'' + \frac{g}{l} * \sin(\varphi) = 0$$

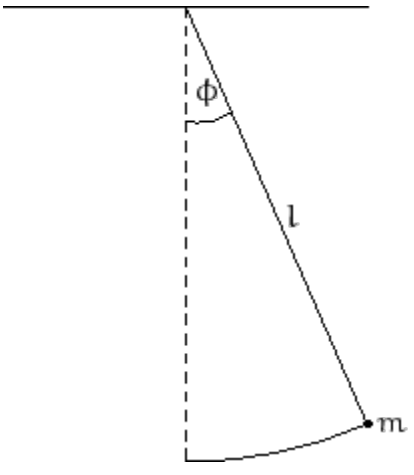


Abbildung 3

Ein Pendel ist dann in einer Ruhelage, wenn alle Funktionswerte konstant bleiben und deren Ableitungen gleich 0 sind. In unserem Fall liegen diese Ruhelagen bei $\varphi = 0$ und bei $\varphi = \pi$, weil bei diesen Werten Obiges zutrifft. Die untere Ruhelage liegt bei $\varphi = 0$ und ist eine stabile Ruhelage, wohingegen die obere Ruhelage, die bei $\varphi = \pi$ liegt, eine instabile ist.

Wie wir auf diese Vermutungen gekommen sind, wird später im Kapitel „Bestimmung der Stabilität einer Ruhelage“ nachzulesen sein.

Aufstellen der Gleichung für ein Pendel

Als kleine Vorübung für das spätere Aufstellen der Bewegungsgleichungen für den Segway versuchen wir zuerst auf die Gleichung für das Fadenpendel zu kommen. Als Modellierungsvariable legen wir φ fest und stellen zunächst die Gleichungen für die kinetische Energie T und potentielle Energie V auf. Daraus lassen sich folgende Gleichungen für die Masse m, die Länge des Fadenpendels l und die Winkelgeschwindigkeit ω herleiten:

$$T = \frac{1}{2} * m * l^2 * \omega^2$$

$$V = m * g * \cos(\varphi) * l$$

Wenn man nun die Differenz dieser beiden Energien bildet, ergibt sich daraus die Lagrange-Funktion, die, wie folgt, aussieht:

$$L = T - V$$

Um von dieser Gleichung auf die Bewegung eines Fadenpendels zu kommen, müssen wir nun die entsprechende Euler-Lagrange-Gleichung aufstellen:

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = 0\,.$$

1. Schritt: Ableiten von L nach φ'

$$L'_{(\omega)} = m * l^2 * \omega$$

2. Schritt: Ableiten des aus Schritt 1 erhaltenen Ergebnisses nach der Zeit t

$$\frac{d}{dt} L'_{(\omega)} = m * l^2 * \varphi''$$

3. Schritt: Ableiten der Grundfunktion L nach φ

$$L'_{(\varphi)} = m * g * \sin(\varphi) * l$$

4. Schritt: Differenz von Schritt 2 und Schritt 3

$$m * l^2 * \varphi'' - m * g * \sin(\varphi) * l = 0$$

Durch Befolgen dieser Schritte und Vereinfachen jener Gleichung kommt man auf dieses Ergebnis:

$$\varphi'' - \frac{g}{l} * \sin(\varphi) = 0$$

Wenn man unser eigenes Ergebnis mit der wirklichen Gleichung für die Bewegung des Fadenpendels vergleicht, erkennt man, dass unsere Gleichung bis auf das Vorzeichen identisch ist. Das Vorzeichen ergibt sich durch die Position des Pendels, an der die Auslenkung gleich 0 ist.

Bestimmung der Stabilität einer Ruhelage

Um feststellen zu können, ob eine Ruhelage stabil oder instabil ist, muss die Bewegungsgleichung für das Fadenpendel linearisiert werden. Erst dann ist es möglich, die Eigenwerte der Matrix zu berechnen.

Die allgemeine Lösung der linearen Differentialgleichungen vom Grad n benötigen wir zwar nicht direkt, sie hilft uns aber, zu verstehen, warum bestimmte Ruhelagen stabil bzw. instabil sind.

$$x(t) = A_1 * e^{\lambda_1 * t} + A_2 * e^{\lambda_2 * t} + \cdots + A_n * e^{\lambda_n * t}$$

In diesem Fall stehen λ_1 bis λ_n für die Eigenwerte der Matrix, die die lineare Differentialgleichung beschreibt. Für den Fall, dass der Realteil von einem dieser Werte positiv ist, wächst die Funktion über alle Grenzen. Wenn dieser Fall eintritt, ist die von uns errechnete Ruhelage instabil. Umgekehrt ist die Ruhelage stabil, wenn die Realteile sämtlicher Eigenwerte negativ sind.

Wie vorhin erwähnt, muss die Bewegungsgleichung für das Fadenpendel zuerst linearisiert werden. Wir beginnen mit der Ruhelage $\varphi = 0$:

$$\varphi'' + \frac{g}{l} * \sin(\varphi) = 0$$

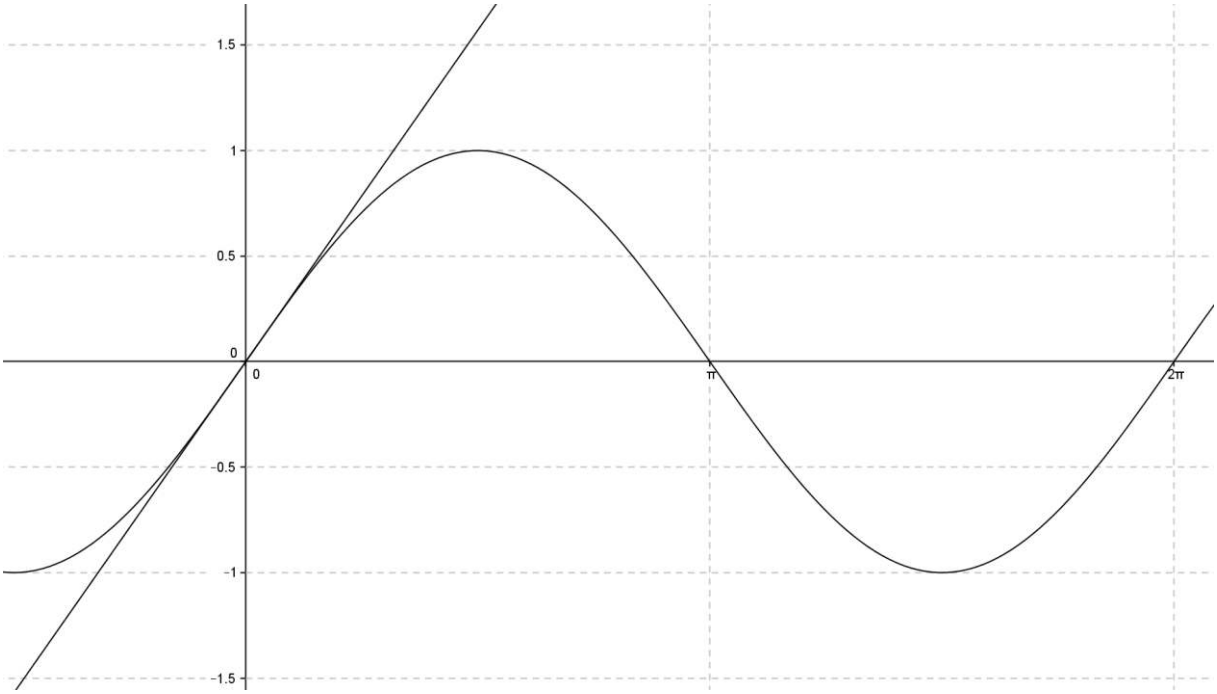


Abbildung 4

Abbildung 3 verdeutlicht, dass $\sin(\varphi)$ bei kleinen Winkeln in etwa φ entspricht. Daher können wir statt der obigen Gleichung auch schreiben:

$$\varphi'' + \frac{g}{l} * \varphi = 0$$

Die Lösung einer linearen Differentialgleichung ist immer eine Linearkombination von Exponentialfunktionen. Daher können wir statt dieser Gleichung auch schreiben:

$$\lambda^2 * e^{\lambda * t} + \frac{g}{l} * e^{\lambda * t} = 0$$

Für λ erhält man durch Umformungen:

$$\lambda_{1,2} = \sqrt{-\frac{g}{l}}$$

Beide Eigenwerte sind imaginäre Zahlen. Das bedeutet, dass die Realteile gleich 0 sind und $\varphi = 0$ eine stabile Ruhelage darstellt.

Analog dazu betrachten wir nun den Fall $\varphi = \pi$:

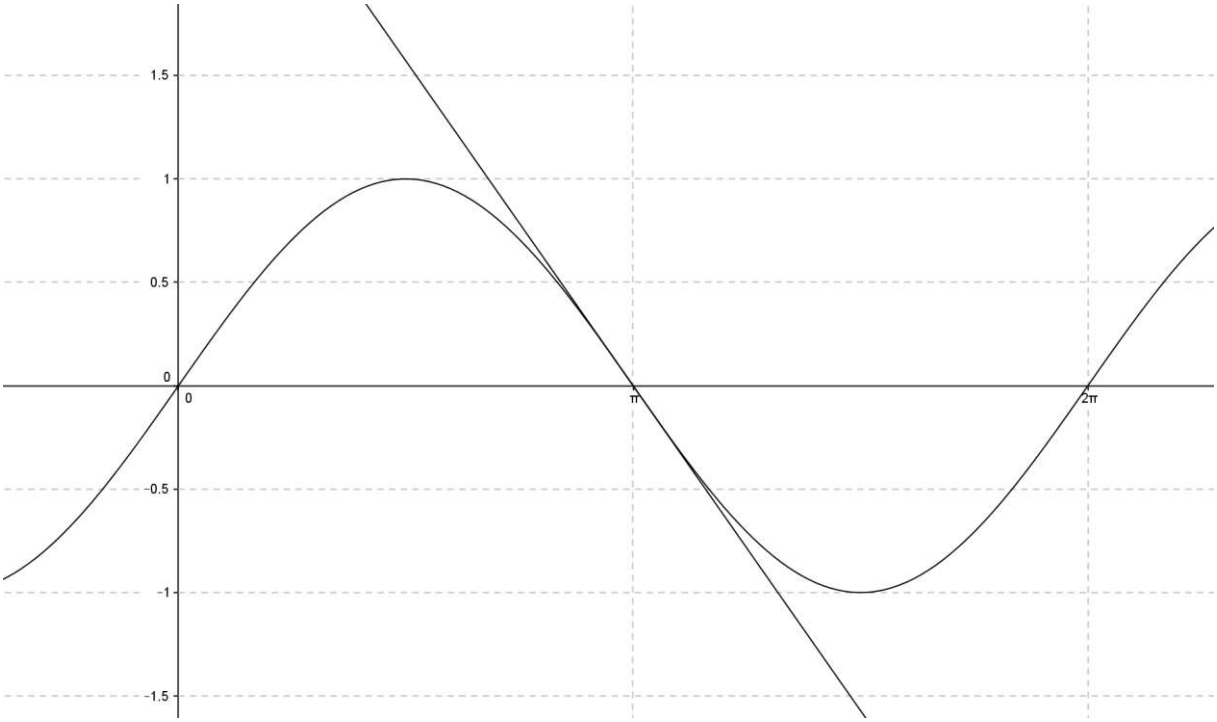


Abbildung 5

An der Stelle π lässt sich $\sin(\varphi)$ durch $-\varphi$ annähern. Einsetzen und Ausmultiplizieren führt zu:

$$\varphi'' - \frac{g}{l} * \varphi = 0$$

Wie beim vorherigen Fall können wir diese Gleichung umschreiben:

$$\lambda^2 * e^{\lambda * t} - \frac{g}{l} * e^{\lambda * t} = 0$$

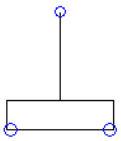
Nach einigen weiteren Umformen gelangt man zu:

$$\lambda_{1,2} = \pm\sqrt{\frac{g}{l}}$$

Es zeigt sich, dass einer der beiden Eigenwerte positiv und die Ruhelage somit instabil ist.

Modellierung des Segways

Die Vorstellung des Segways als inverses Pendel auf einem Wagen



Anfangs legen wir unseren Segway als Wagen mit aufgesetztem inversen Pendel fest. Dabei nehmen wir den Verbindungspunkt von Pendel und Wagen als Startpunkt an. Danach können wir die X- und Y-Werte der Eckpunkte sowie des Pendelkopfs und der Reifen berechnen und damit ist es uns möglich, das oben gezeigte Modell in MATLAB zu erstellen.

Aufstellen der „Segway-Gleichungen“

Der nächste Schritt ist es, das Pendel zu stabilisieren. Dabei gehen wir wieder von der Euler-Lagrange Gleichung aus:

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = 0\,.$$

Das L haben wir in unserem Beispiel folgend definiert:

$L = T_1 + T_2 - V$

T_1 stellt die kinetische Energie des Pendels dar, T_2 die kinetische Energie des Wagens. V ist die potenzielle Energie des Pendels. Damit können wir folgende Gleichung erstellen:

$L = \frac{1}{2} * m_1 * l^2 * \omega^2 + \frac{1}{2} * m_2 * v^2 - m_1 * g * \cos(\varphi) * l$

Die Variablen in der Gleichung setzen sich zusammen aus:

- m_1 Masse des Pendels
- m_2 Masse des Wagens
- l Länge des Pendels

- g Erdbeschleunigung
- φ Auslenkungswinkel des Pendels, dabei ist der Winkel 0, wenn das Pendel wie in der Abbildung senkrecht nach oben zeigt
- ω Winkelbeschleunigung
- v Geschwindigkeit des Wagens

Durch Umformungen und Verwendung des Differentialquotienten kommen wir zu:

$$L = m_1 * \cos(\varphi) * \omega * l * \mathrm{v} + \frac{1}{2}(m_1 + m_2) * \mathrm{v}^2 + \frac{1}{2} m_1 * \omega^2 * l^2 - m_1 * g * \cos(\varphi) * l$$

Um die Euler-Lagrange-Gleichung zu vervollständigen, rechnen wir:

$$\frac{d}{dt}\big(L'(\omega)\big) - L'(\varphi) = 0$$

Durch Einsetzen und Kürzen kommt man auf die Form:

$$\cos(\varphi) * \mathrm{v}' + l * \omega' - g * \sin(\varphi) = 0$$

Diese Gleichung stellt die erste der vier „Segway-Gleichungen“ dar. Sie ist die Gleichung für ω.

Die weiteren Gleichungen sind:

Für φ:$\varphi' - \omega = 0$

Für x: $x' - v = 0$

Für v: $\frac{d}{dt} L'(t) - \frac{d}{dx} L'(x) = m_1 * l * (-\sin(\varphi) * w^2 + \cos(\varphi) * \omega') + (m_1 + m_2) * \mathrm{v}' = 0$

Um diese Gleichungen in MATLAB verwenden zu können, definieren wir 2 Vektoren:

$$\text{var1}=\begin{bmatrix} x \\ v \\ \varphi \\ \omega \end{bmatrix} \quad \text{var2}=\begin{bmatrix} x' \\ v' \\ \varphi' \\ \omega' \end{bmatrix}$$

Die Gleichungen für v und ω formen wir anschließend so um, dass nur mehr eine Ableitung pro Gleichung besteht. Zusätzlich bauen wir die Kraft F ein. Sie soll die Geschwindigkeit v des Wagens, sowie die Winkelgeschwindigkeit ω verändern, damit das Pendel in weiterer Folge stabilisiert wird.

Die neuen Gleichungen sehen wie folgt aus:

Für v:

$$-m_1 * l * \sin(\varphi) * \omega^2 + m_1 * \cos(\varphi) * [g * \sin(\varphi) - \cos(\varphi) * v'] + (m_1 + m_2) * v' - F = 0$$

Für ω:

$$\cos(\varphi) * [m_1 * l * \sin(\varphi) * \omega^2 - m_1 * l * \cos(\varphi) * \omega + F] + l * (m_1 + m_2) * \omega' - (m_1 + m_2) * g * \sin(\varphi) = 0$$

Einbauen eines Störfaktors

In MATLAB können wir nun auch einen Störfaktor einbauen, um dem Segway einen „Stoß“ mitzugeben. Dazu muss man nur eine neue Kraft, welche in einer For-Schleife definiert wird, in die Formeln einbauen. Diese Schleife sieht folgendermaßen aus:

```
if(t>=9.9 && t<=10.1 || t>=15.9 && t<=16.1)
    F2=2000;
else
    F2=0;
end
```

Abbildung 6

Linearisierung der Gleichungen

Die Gleichungen müssen nun linearisiert werden, um die Eigenwerte und in weiterer Folge die Ausgleichskraft berechnen zu können. Dazu müssen alle trigonometrischen Funktionen und Polynome entfernt werden. Wir nehmen daher für eine Linearisierung um den Punkt 0 Folgendes an:

$$\sin(\varphi) = \varphi$$

$$\cos(\varphi) = 1$$

$$\omega^2 = 0$$

Als nächsten Schritt formen wir die Gleichungen um und bauen sie in die Matrix $y' = Ay$ ein.

$$v_w' = \frac{-m_1 * g * \varphi}{m_2} + \frac{F}{m_2}$$

$$\omega' = -\frac{F}{m_2 * l} + \frac{(m_1 + m_2) * g * \varphi}{m_2 * l}$$

$$\begin{bmatrix} x' \\ v' \\ \varphi' \\ \omega' \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-m_1 * g}{m_2} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{(m_1+m_2)*g}{m_2 * l} & 0 \end{bmatrix} \begin{bmatrix} x \\ v \\ \varphi \\ \omega \end{bmatrix} + F \begin{bmatrix} 0 \\ \frac{1}{m_2} \\ 0 \\ \frac{-1}{m_2 * l} \end{bmatrix}$$

Einbauen der Parameter von F zur Stabilisierung des Segways

Die Kraft F, die auf den Segway wirken soll, hat die Form:

$$F = a * x + b * v + c * \varphi + d * \omega$$

Setzt man F in die Matrix ein, so bekommt man die neue Matrix A die wir auch für die Programmierung in MATLAB benutzen. Die Parameter a, b, c und d werden dabei so gewählt, dass die Realteile der Eigenwerte von A negativ sind und dadurch das System stabil wird.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{a}{m_2} & \frac{b}{m_2} & \frac{c - m_1 * g}{m_2} & \frac{d}{m_2} \\ 0 & 0 & 0 & 1 \\ \frac{-a}{m_2 * l} & \frac{-b}{m_2 * l} & \frac{(m_1 + m_2) * g - c}{m_2 * l} & \frac{-d}{m_2 * l} \end{bmatrix}$$

In folgender Abbildung sind die Werte für $a, b, c,$ und d mit einer Schleife zufällig gewählt, dabei gilt jedoch noch immer die Bedingung, dass die Realteile der Eigenwerte negativ sein müssen. Der Wagen mit dem Pendel verhält sich in diesem Fall so:

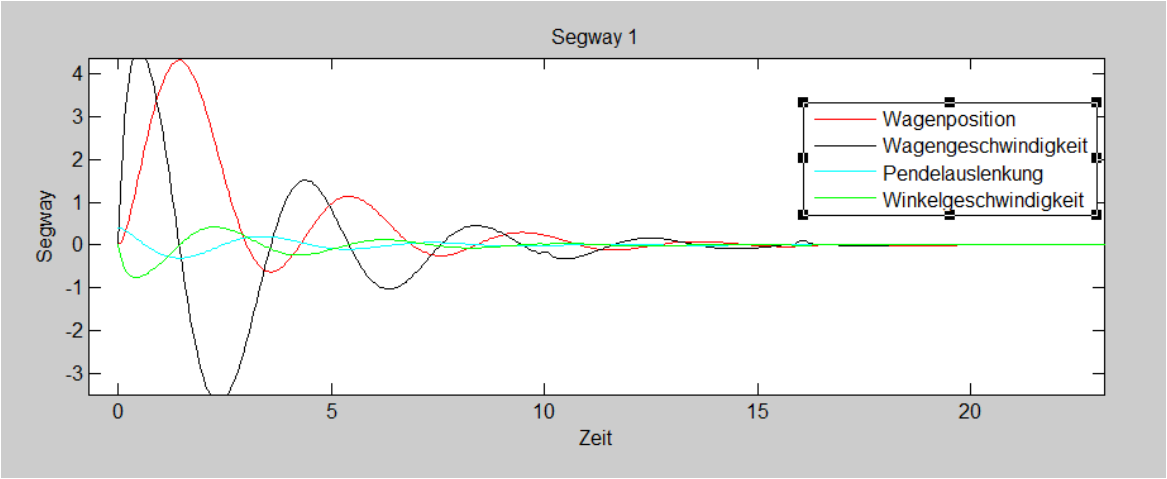


Abbildung 7

Durch Verwendung bereits in MATLAB implementierter Funktionen können wir die optimalen Werte für a, b, c und d berechnen.

Hier ist die Ausgleichskraft sehr groß, während die anderen Graphen kaum Auslenkungen haben.

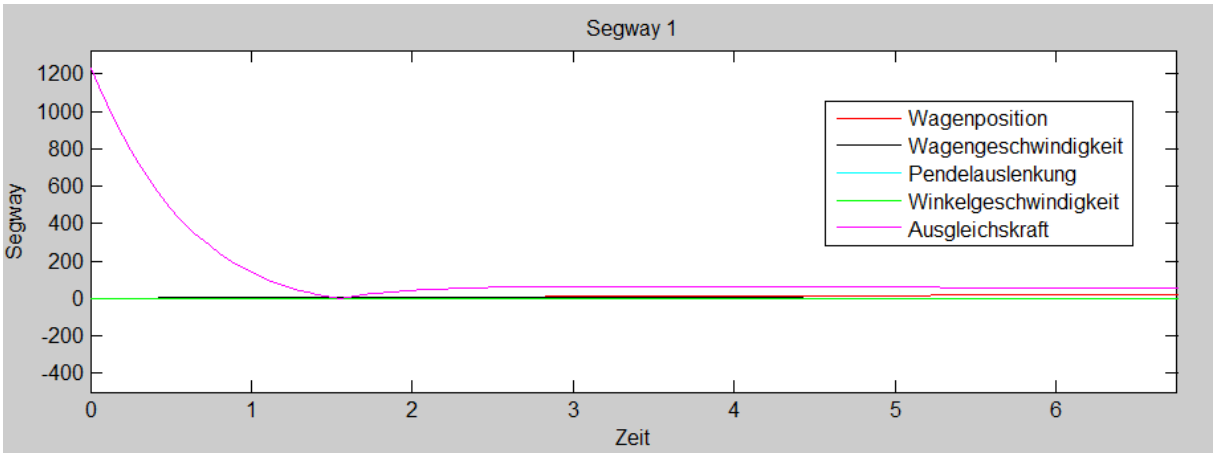


Abbildung 8

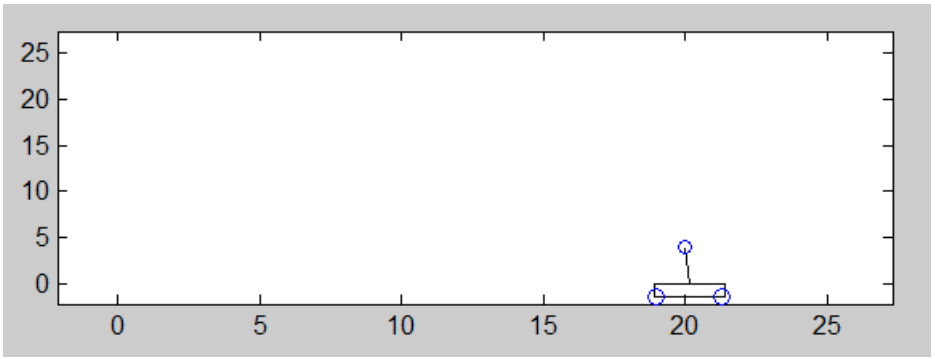


Abbildung 9

In dieser Abbildung bewegt sich der Wagen in Richtung Nullpunkt zurück. Wenn der Wagen diesen Punkt erreicht hat ist die Auslenkung des Pendels wieder 0 und das System befindet sich in Ruhe.

Zusammenfassung

Schon nach den ersten Stunden hat sich herausgestellt, dass die Modellierung eines Segways komplizierter ist, als ursprünglich gedacht. Wir wurden mit für uns vollkommen neuen

Teilbereichen der Mathematik konfrontiert und mussten erst die richtigen Verfahren herleiten. Doch schließlich haben wir es geschafft, eine Simulation zu erstellen, in der sich der von uns modellierte Segway selbst stabilisiert. Unser Hauptziel haben wir also auf alle Fälle erreicht. Trotzdem ist es schade, dass wir es aus Zeitgründen nicht mehr geschafft haben, eine ähnliche Simulation für ein Doppelpendel zu erstellen.

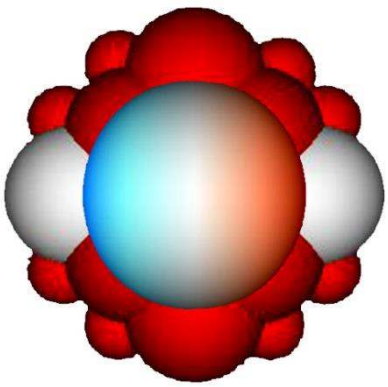
Computergrafik

Betreuer:

Ass. Prof. Dr. Laurent Pfeiffer

Team:

Duy Du Quoc, Samuel Rabensteiner, Jakob Krasser, Christian Schlegl, Maximilian Michl, Stefan Krickl



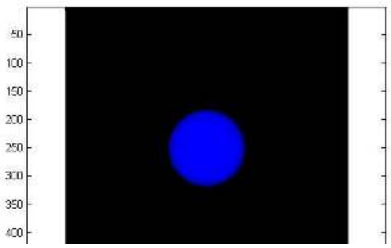
07.02. – 13.02.2015

1 Inhalt

1 Hintergründe

Mit dem Begriff „Computergrafik“ definiert man ein Teilgebiet zwischen Informatik und angewandter Mathematik, die sich mit der Erzeugung von realistischen Bildern beschäftigt. Die Anwendungsmöglichkeiten von Computergrafiken reichen von der Videospielindustrie über die Produktion von animierten Filmen (z.B. Baymax) bis hin zu der Medizin sowie der Darstellung von Architekturprojekten am Computer.

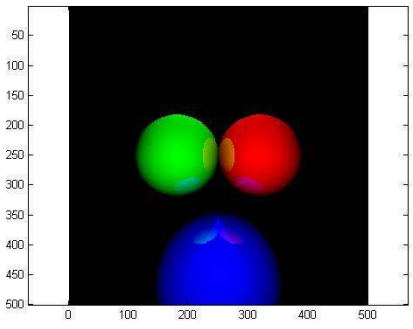
Mit einem einfach geschriebenen Programm kann man mit kleinen Veränderungen der Werte das Ergebnis beliebig verändern. Infolgedessen ist es möglich die Position des Beobachters, sowie die Position des Lichtes und auch die Position des Gegenstandes anders zu bestimmen. Die Möglichkeit besteht mehrere Objekte bzw. Lichter einzufügen. Nichtsdestotrotz wirken die Ergebnisse realistisch. So kann man mit relativ wenig Aufwand, verschiedene Bildern erzeugen, diese analysieren oder zur Erstellung komplizierterer Figuren verwenden. Aus diesem Grund ist Computergrafik heute in vielen Bereichen nicht mehr wegzudenken.



1 blaue Kugel mit weißer Beleuchtung von vorne; betrachtet von vorne

Werte für das blaue Dreieck

```
l=3;
L=3;
m=500;
n=500;
d=2;
C=[10; 0; 0];
r=[2];
pBeleuchtung=[0 0 0];
fBeleuchtung=[1 1 1];
Fk=[0; 0; 1 ];
```



3 Kugeln (grün, rot, blau); 2 Beleuchtung (vorne, oben)

Änderung der Werte C, r, pBeleuchtung, fBeleuchtung, Fk

```
l=3;
L=3;
m=500;
n=500;
d=2;
C=[10 10 10; 2 -2 0; 0 0 -6];
r=[2, 2, 3];
pBeleuchtung=[0 10; 0 0; 0 10];
fBeleuchtung=[1 1; 1 1; 1 1];
Fk=[1 0 0; 0 1 0; 0 0 1];
```

2 Benötigte Formeln

2.1 Grundlagen der Vektorrechnung

2.1.1 Raum R²

Rechenregeln für Vektorrechnung für $R^2 = \{(x; y), (x \in R; y \in R)\}$.

2.1.1.1 Euklidische Norm

Die euklidische Norm ist die Länge eines Vektors $u = (ux; uy)$: $\|u\|^2 = ux^2 + uy^2 = u * u$

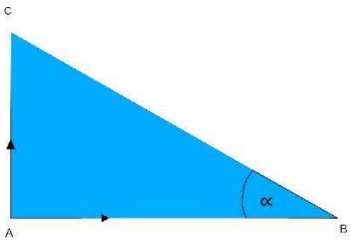
2.1.1.2 Skalarprodukt

Für 2 Vektoren $u = (ux; uy)$ und $v = (vx; vy)$ definieren wir das Skalarprodukt als:

$u * v = (ux * vx + uy * vy)$ Das Ergebnis ist eine Zahl.

Wenn das Skalarprodukt gleich Null ist; sind die Vektoren orthogonal. Nun ist auch der Satz des Pythagoras anwendbar. Der Vektor AB sei orthogonal auf den Vektor AC:

$AB^2 + AC^2 = BC^2$



Auch die Winkel können somit berechnet werden:

$$sin\beta = \frac{\|AC\|}{\|BC\|}$$

$$cos\beta = \frac{\|BA\|}{\|BC\|} \qquad \|BA\| = \|BC\| * \cos(\beta)$$

$BA * BC = \|BA\| * \|BC\| * \cos(\beta)$ $BA * BC = \|BA\|^2$

2.1.1.3 Projektion

Die Projektion ist in R^2 und in R^3 möglich. Zur Berechnung der Spiegelung ist die Projektion enorm wichtig. Ohne dieses Hilfsmittel wären komplizierte Berechnungen von Einfalls- und Austrittswinkel von Nöten gewesen. Wir projektieren den Punkt M auf die Gerade (AB).

$$M = A + AM$$

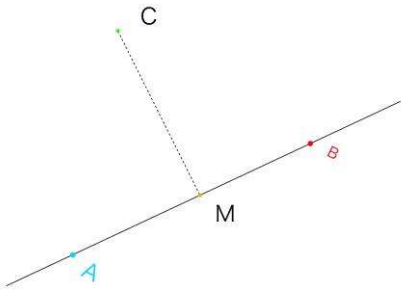
$$AB * MC = 0$$

$$AM = \frac{\|AM\|}{\|AB\|} * AB$$

$$AB * AC = \|AB\| * \|AC\| * cos\alpha = \|AB\| * \|AM\|$$

$$\|AM\| = \frac{(AB * AC)}{\|AB\|}$$

$$AM = \frac{AB * AC}{AB^2} * AB$$



Projektion

2.1.2 3-D-Raum R^3

$$R^3 = \{(x; y; z);\ (x \in R;\ y \in R;\ z \in R\}$$

2.1.2.1 Kreuzprodukt

Um in R^3 den Normalvektor n bestimmen zu können braucht man das Kreuzprodukt zweier Vektoren u und v. Dieses beschreibt dann einen neuen Vektor, der auf u und auf v normal ist. Das Kreuzprodukt von u und v ist immer orthogonal zu u und v.

$$u \times v = \begin{pmatrix} uy * vz - vy * uz \\ uz * vx - vz * ux \\ ux * vy - vx * uy \end{pmatrix}$$

2.1.3 Geraden

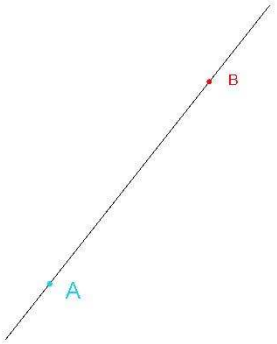
Wir parametrisieren eine Gerade mit einem Parameter s.

$M \in (AB)$

$M = A + s * AB$

$M = A + s * (B - A)$

$M = (1 - s) * A + s * B$

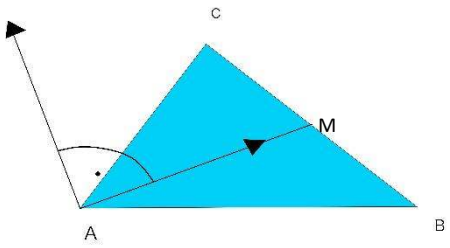


Punkte A und B auf Gerade AB

2.1.4 Ebenen

Wir parametrisieren eine Ebene mit zwei Parametern s und t

$M = A + s * AB + t * AC$



Ebene ABC im 3D-Raum

2.2 Basen

Die Vektoren u, v und w bilden eine Basis, wenn für alle Vektoren r = $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ a, b und c existieren, so dass: $r = a * u + b * v + c * w$

2.3 Schnitt von Geraden und Ebenen

Um bestimmten Punkten auf unserem Bildschirm bestimmte Farben zuordnen zu können, mussten wir zuerst feststellen, ob es einen Schnittpunkt F der Gerade Beobachter(A)-Bildschirmpunkt(B) mit der Ebene (C; D; E) gibt.

F: $A + s * AB = C + t * CD + p * CE$

$$s = \frac{(AC \times CD) * CE}{(AB \times CD) * CE}$$

$F = A + s * AB$

2.4 Kombination der Farben

Die Farbe eines Gegenstandes setzt sich zusammen aus vielen Faktoren: verschiedenfarbige Lichtquellen (f1, f2), Spiegelungen und der Farbe der Objekte (fK) und auch der Winkel des Auftreffen des Lichtes. Der Winkel zwischen dem Normalvektor und dem Vektor zwischen Schnittpunkt und Beleuchtung ist $\gamma 1\ und\ \gamma 2$.

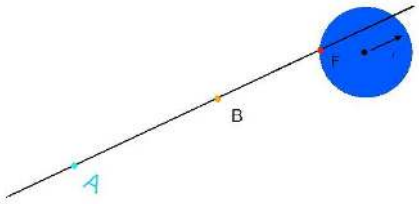
$$f = (f1 * cos(\gamma 1) + f2 * cos(\gamma 2) - (f1 * cos\ (\gamma 1)\ ) * (f2 * cos(\gamma 2))\ * fk$$

2.5 Schnitt von Ebenen und Kugeln

Um Kugeln darzustellen, also einen Schnittpunkt der Gerade AB mit der Oberfläche einer Kugel mit dem Mittelpunkt C und dem Radius r zu finden brauchten wir eine weitere Formel.

$0 = a * s^2 + b * s + c$

$F = A + s * AB$



$r^2 = \|F - C\|^2$

$r^2 = \|CA + s * AB\|^2$

$0 = \|AB\|^2 * s^2 + 2 * AB * CA * s + \|CA\|^2 - r^2$

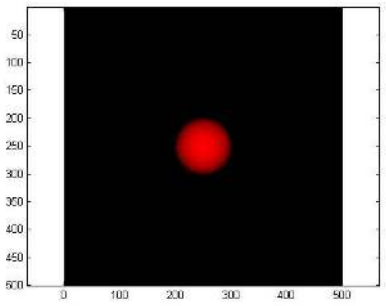
$a = \|AB\|^2 * s^2 \qquad b = 2 * AB * CA * s \qquad c = \|CA\|^2 - r^2$

$$s = \frac{-b - \sqrt{b^2 - 4 * a * c}}{2a}$$

3 Was sieht der Beobachter?

3.1 Definition des Problems

Ein Beobachter stellt sich in den Raum und schaut in Richtung eines Objektes. Dieses Objekt hat eine beliebige Farbe, zum Beispiel rot, und wird beleuchtet. Die Position der Lichtquelle ist frei wählbar. Um die Sache jedoch zu vereinfachen, haben wir angenommen, dass die Position des weißen Lichtes die gleiche Position wie die des Beobachters hat.

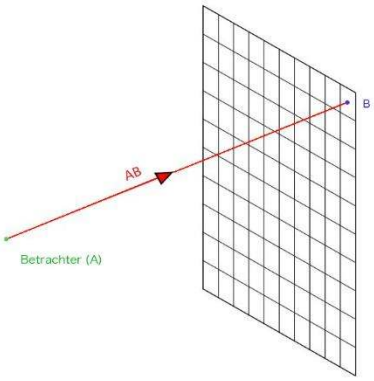


Beobachter (0|0|0), Lichtquelle (0|0|0), rote Kugel (10|0|0)

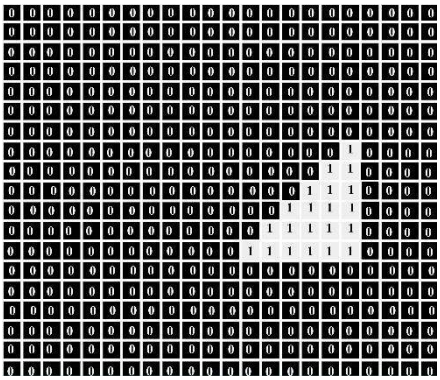
Was sieht der Beobachter nun? – Die Grafik rechts beschreibt die Lösung grafisch. – Der Beobachter sieht eine rote Kugel. – Daraufhin haben wir unsere zweite Frage wie folgend formuliert: „Warum sieht der Beobachter eine rote Kugel?“

3.2 Erste Überlegungen

Der Beobachter sendet Sehstrahlen in Richtung des Objektes. Die Strahlen werden auf dem Weg dorthin durch Punkte von einem imaginären Bildschirm geschickt. Diese Punkte werden mit einer Matrix beschrieben. Daraufhin treffen die Strahlen auf den Gegenstand oder gehen daran vorbei. Falls ein Strahl das Objekt treffen sollte, wird dem Punkt der Matrix, durch dem der Strahl geschickt worden ist, die Zahl 1 zugeordnet. Sollte dieser Fall nicht eintreten, so werden die Punkte mit 0 bezeichnet. Für das Programm bedeutet 1 die Farbe Weiß und 0 die Farbe Schwarz, daraus ergibt sich das unten gezeigte Bild.



Strahl ausgehend von Betrachter; Bildschirm; Objekt



0 = Sehstrahlen treffen das Objekt nicht;
1 = Sehstrahlen treffen das Objekt

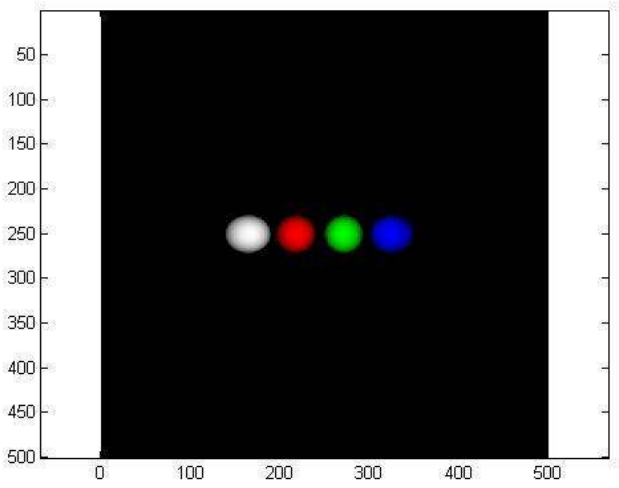
3.3 Bildschirm

Unser Bildschirm wird einfach in Spalten und Zeilen aufgeteilt. In jeder Zeile und jeder Spalte gibt es eine von uns definierte Anzahl von Pixeln. Der Bildschirm bestimmt die Auflösung des Bildes. Je größer der Bildschirm, desto größer die Anzahl der Pixel, desto besser die Auflösung.

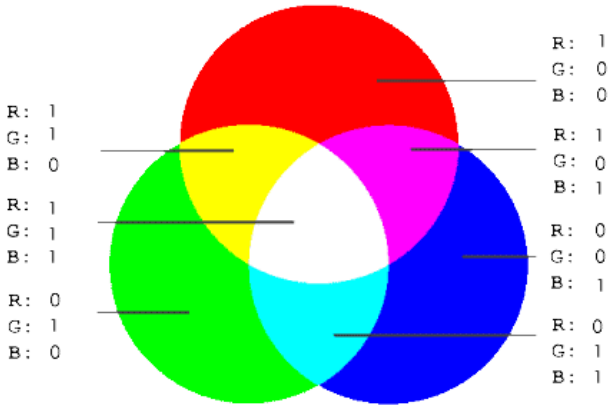
3.4 Farbe

Die verschiedensten Farben werden mit einem sogenannten RGB-Farbraum beschrieben, wofür man 3 Zahlen zur Definition benötigt. Die Grundfarben des Systems sind Rot, Grün und Blau, deswegen die Abkürzung „RGB-Farbraum“. Durch das Mischen dieser Farben ist es möglich einen großen Teil der Farbpalette abzudecken. Die Zahlen liegen im Intervall [0; 1].

Der Gegenstand ist schwarz, wenn der Vektor der Farbe des Gegenstandes $n = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ ist. Für die Farbe Weiß ist $n = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$, für Rot ist $n = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, für Grün ist $n = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ und die Farbe Blau erhält man, wenn $n = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ ist. Um die weiteren Farben kreieren zu können, muss man nur die obengenannten Farben miteinander mischen, dies folgt denselben Regeln wie die Vektorrechnung, z.B.: $Grün \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + Blau \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = Cyan \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$



Grundfarben des RGB-Farbraums;
4 Kugeln mit weißem Licht beleuchtet

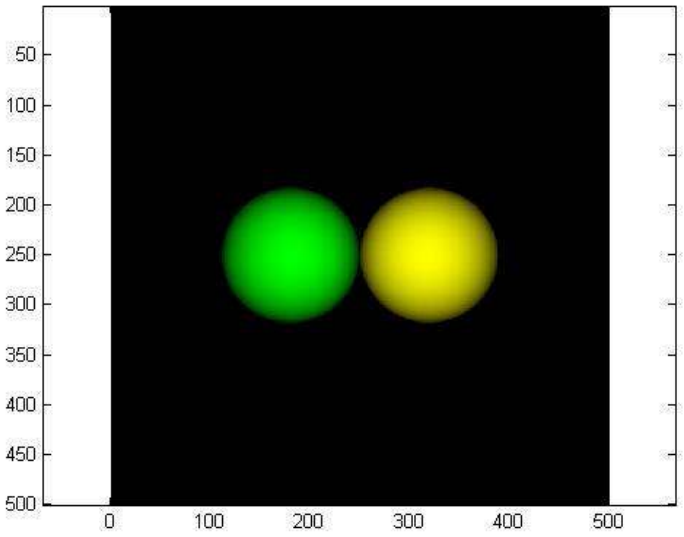


RGB Farbraum

3.4.1 Beleuchtung mit Farbe

Bisher hat man die Gegenstände nur mit weißem Licht beleuchtet. Ein Problem ergibt sich, wenn das Licht, welches den Gegenstand beleuchtet, nicht weiß, sondern einen anderen Farbvektor hat, z.B.: $Gelb = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$. Daraus zieht man den Schluss, dass das Ergebnis eine andere Farbe haben muss, als sein Farbvektor. – Unsere Überlegung waren wie folgt: wenn eine rote Kugel mit weißem Licht beleuchtet wird, absorbiert diese die Grün-, sowie Blaufarbwerte des Lichtes, indessen sie den Rotfarbwert reflektiert. Das reflektierende Licht ist jetzt rot und wird vom Betrachter als solches wahrgenommen. – Wir haben daraufhin eine Formel entwickelt, mit der man die Farbe der beleuchten Kugel berechnen kann:

$$Farbvektor\ vom\ Ergebnis = \begin{pmatrix} Rotfarbwert\ von\ Kugel * rW\ vom\ Licht \\ gW\ von\ K * gW\ vom\ L \\ bW\ von\ K * gW\ vom\ L \end{pmatrix}$$

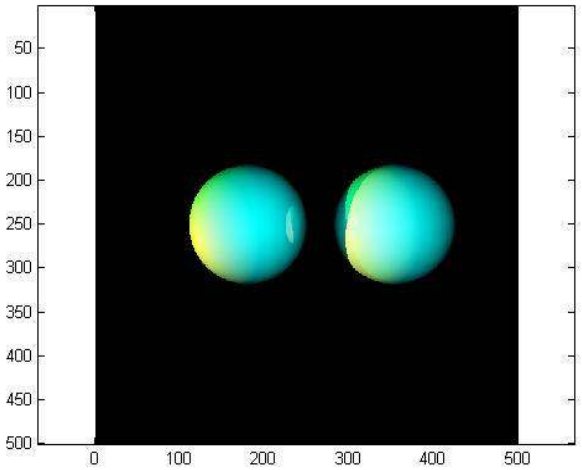


2 Kugeln mit gelbem Licht beleuchtet;
links: grüne Kugel
rechts: weiße Kugel

3.4.2 Andere Faktoren

Aber für mehrere sich gegenseitig beeinflussenden, verschiedenfarbigen Lichtquellen (f1, f2) und Spiegelungen und der Farbe der Objekte (fk) musste für alle möglichen Schnittpunkte eine Formel für das im Endeffekt auf dem Bildschirm dargestellte Licht (f) entwickelt werden, wobei auch der Winkel des Auftreffen des Lichts berücksichtigt wurde. Mit der unteren Formel kann man diese Faktoren miteinander in einem Zusammenhang bringen.

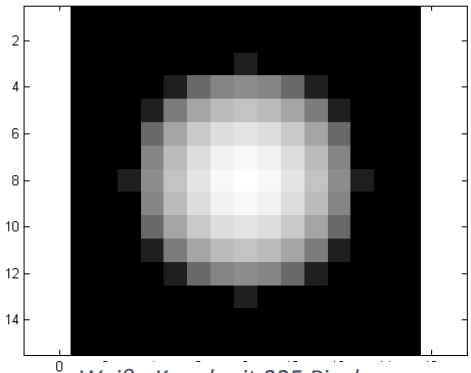
$$f = (f1 * cos(\theta) + f2 * cos(\theta) - (f1 * cos(\theta)) * (f2 * cos(\theta)) * fk$$



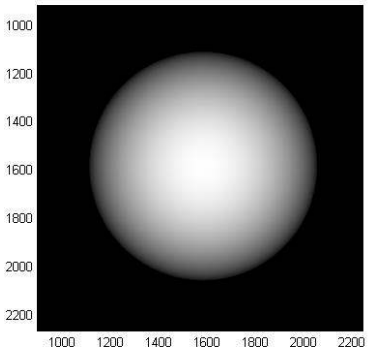
2 weiße Kugeln;
beleuchtet von vorne mit weiß; links unten mit gelb; links oben mit grün;

3.5 Auflösung

Die Auflösung des Bildes wird durch die die Größe des Bildschirmes definiert. Je mehr Pixel der Bildschirm hat desto besser ist die Auflösung des Bildes. Meistens nutzten wir eine Auflösung von 100x100 Pixeln bis die Funktion wirklich richtig funktionierte. Dann nahmen wir eine Auflösung von 1000x1000 Pixeln, die beste Auflösung die unsere PC´s verarbeiten konnte, also 1 Million Pixel.



Weiße Kugel mit 225 Pixel

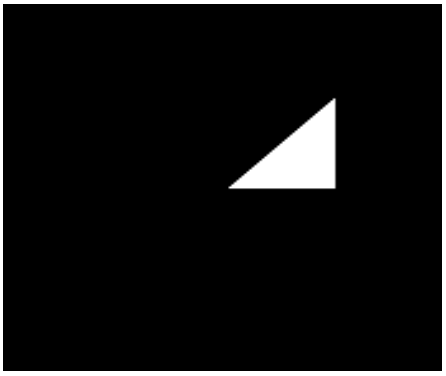


Weiße Kugel mit 10 Million Pixel

4 Dreiecke

Die erste Figur, die wir modelliert haben, war ein Dreieck. Vom Beobachter werden Strahlen durch einen Bildschirm gesendet. Treffen diese auf das Dreieck sehen wir die Pixel in weißer Farbe, treffen diese das Dreieck nicht, sehen wir schwarze Pixeln. Wir fingen mit einfachen Funktionen für die Position des Dreiecks und den Schnitten mit den Strahlen des Beobachters und der Lichtquelle an. Die Lichtquelle hatte dieselbe Position wie der Beobachter, sodass das Dreieck gleichmäßig von vorne beleuchtet wurde. Am Anfang berechneten wir dieses Dreieck nur mit geringer Pixelanzahl und es sah noch eher nach einer Stiege aus. Mit steigender Pixelanzahl wurde das Bild einem Dreieck immer ähnlicher.

Dreieck mit geringer Auflösung

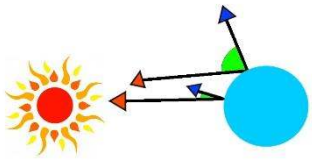


Dreieck mit hoher Auflösung

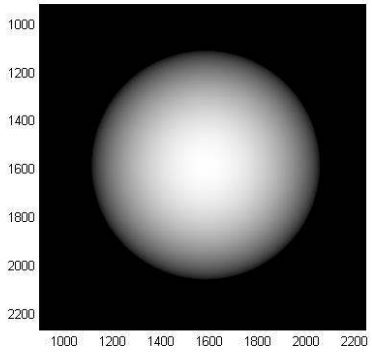
Der nächste Schritt war es, mehrere Dreiecke in einem Bild darzustellen. Dazu benötigten eine Schleife, die es uns ermöglichte, mehrere Werte aus einer Matrix, der Reihe nach zu berechnen. Dann wurden die Werte aller Schnittpunkte am Bildschirm kombiniert. Wir mussten aber beachten, dass es für viele Schnittpunkte mehrere Werte (1 oder 0) gab und in diesem Faktor unserer Funktion berücksichtigen, dass, wenn es für einen Punkt beide Möglichkeiten 1 und 0 gab, immer 1, also ein Schnittpunkt gewählt wurde. Dafür verwendeten wir folgende Regel: $f(Wert\ der\ Beleuchtung) = \max(f, 0)$

5 Kugel

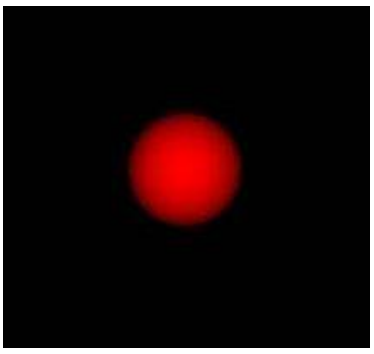
Nachdem wir alle ein Programm für ein Dreieck geschrieben hatten, begannen wir mit dem Modellieren einer Kugel. Unser Grundprogramm blieb gleich und wir fügten nur Funktionen hinzu, die die Schnittpunkte der Gerade AB mit der Kugel im Raum bestimmte. Wir begannen damit, dass wir den Mittelpunkt der Kugel im Raum darstellten und eine Funktion schrieben, um die Kugel rund um den Punkt zu modellieren. Zu diesem Zeitpunkt gab es noch keine farbige Beleuchtung, sondern nur eine weiße Lichtquelle beim Beobachter, die bei einem Schnitt mit der Kugel den Punkt weiß färbte. Damit nicht die ganze Kugel Weiß berechnet den Cosinus des Winkels zwischen den einfallenden Lichtstrahl und dem Normalvektor des Schnittpunktes. Wenn der Winkel ist, also der Strahl 90° Winkel auf die Kugel auftrifft, erhalten wir den Wert 1, der der Reflexion des ganzen Lichtes entspricht. Je mehr sich der Winkel 90° annähert, also je weiter der Punkt von der Lichtquelle aus gesehen entfernt ist, desto schwächer und dunkler wird das Licht. Bei einem Winkel von exakt 90°, wenn also nur mehr genau der Rand der Kugel getroffen wird, ist der Cosinus gleich 0 und der Punkt ist nicht beleuchtet.



Berechnung der Lichtstärke



Weiße Kugel



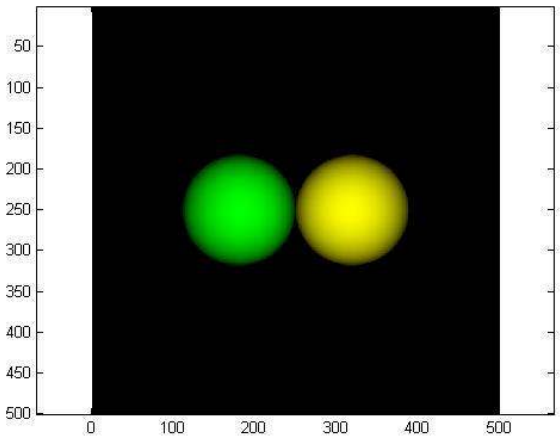
Kugel mit RGB-Beleuchtung

Beleuchtung

Nach der Modellierung einer schwarz/weiß-Kugel wollten wir dieser Farben geben. Formeln an Wie bereits oben beschrieben, wandten wir das RGB – Farbsystem auf unsere.

Mehrere Kugeln:

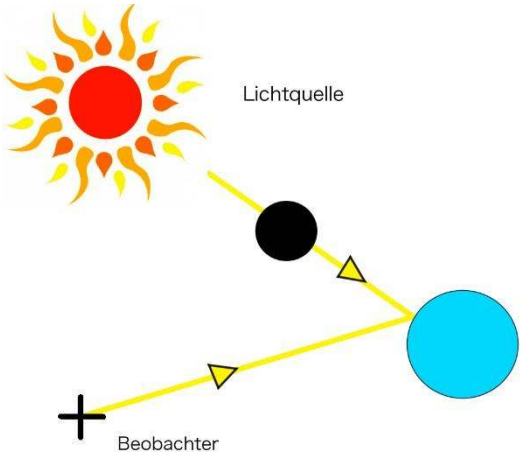
Nachdem wir eine Kugel beleuchtet hatten, wurden wir vor die nächste Aufgabe gestellt: Aus einer Kugel mehrere Kugeln zu machen. Dafür nahmen wir eine kleine Veränderung an unserer bestehenden Formel vor, die es uns erlaubte an Stelle des Vektors, für die Position des Mittelpunktes der Kugel, eine Matrix einzusetzen. Diese kleine Veränderung ermöglichte uns nun beliebig viel Kugeln im Raum zu verteilen.



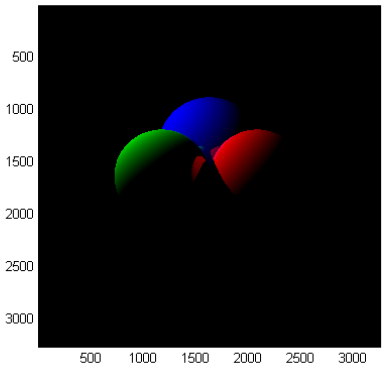
Farbe:

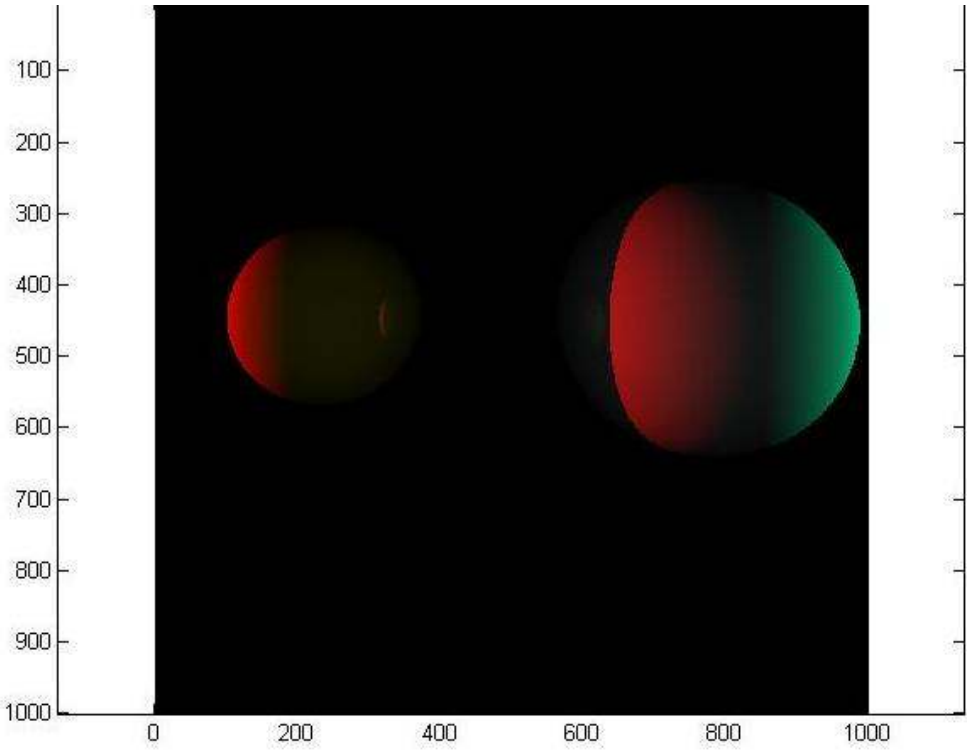
Da wir mehrere Kugeln in Farbe beleuchten konnten wollten wir jetzt auch noch den Kugeln selbst Farbe geben. Durch Vereinfachen unserer Beleuchtungs-Funktion und Einfügen neuer Variablen konnten wir dies auch tun.

6 Schatten



Als nächstes stellten wir uns die Frage, wie unser Modell Schatten bekommen könnte. Dafür veränderten wir die Funktion dahingehend, dass die Strahlen der Lichtquelle, die schon einmal einen Schnittpunkt mit einer Kugel hatten, nicht mehr eine Kugel dahinter beleuchteten, indem wir von jedem Schnittpunkt zu den Lichtquellen aussandten und mit diesem Strahl nach Schnittpunkten suchten.

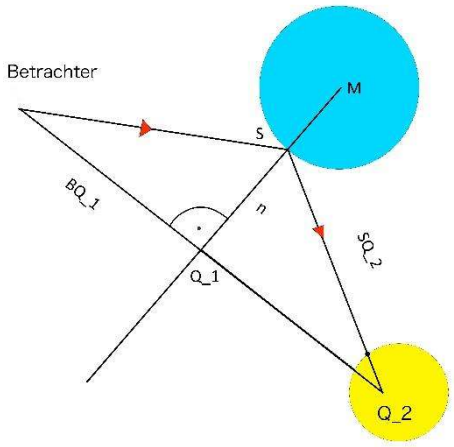




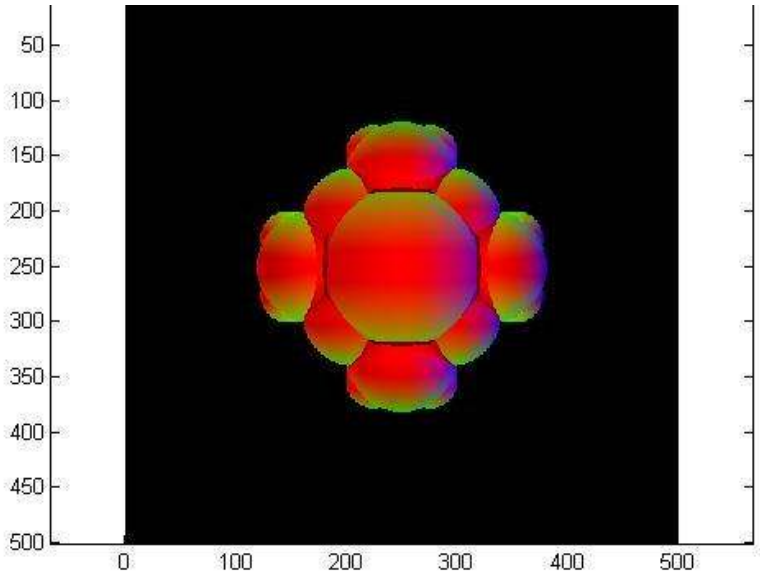
2 Kugeln beleuchtet von vorne, links und rechts (verschiedene Farben);

7 Spiegelung

Nachdem unsere verschieden farbige Kugeln im Raum Schatten warfen, stellten wir nun eine Funktion auf, die gleichzeitig unsere Beleuchtungsfunktion weiter vereinfachte und auch ermöglichte, dass sich Kugeln ineinander spiegelten. So konnten wir schlussendlich viele weiße Kugeln im Raum verteilen und sie von unterschiedlichen Positionen aus, verschiedenfärbig beleuchten. Diese Spiegelung funktioniert so, dass vom Schnittpunkt der Kugel mit einem Austrittswinkel der gleich dem Einfallswinkel des Lichtes ist, ein neuer Strahl ausgesandt wird. Mit diesem wird untersucht, ob ein Objekt vorhanden ist, welches gespielt werden muss. Ferner werden die Beleuchtungen beider Objekte kombiniert und man erhält am ersten Objekt eine Spiegelung des zweiten.



Spiegelung

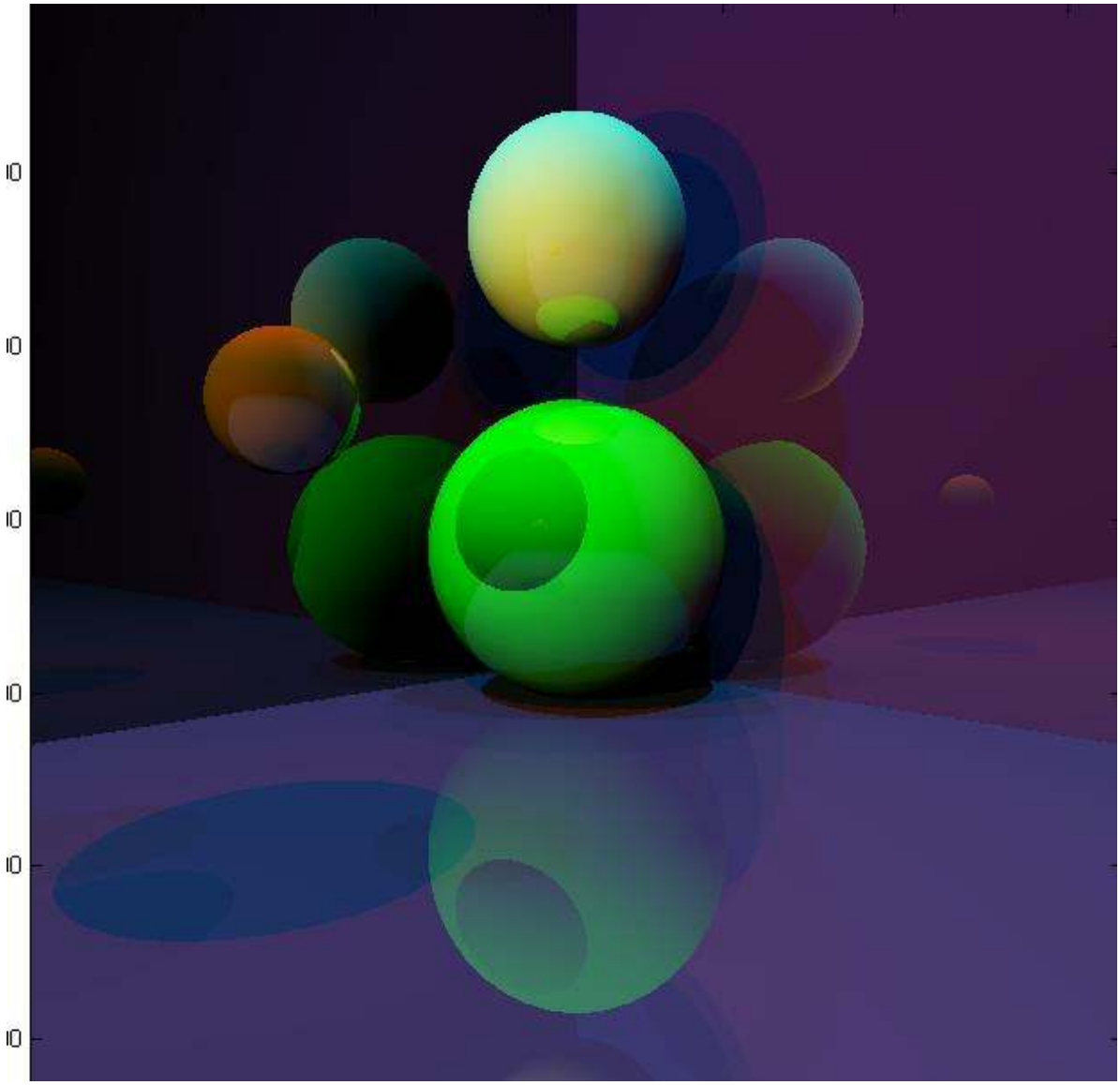


8 Resüme

In unserem Projekt „Computergrafik“ haben wir uns mit der Voraussage einer Beobachtung durch Strahlenverfolgung beschäftigt. Wir befassten uns vor allem mit dem Strahlenverfolgungs-Algorithmus, durch den wir letztendlich eine Kugel in einem dreidimensionalen Raum modellieren, die einen Schatten wirft und sich selbst in einer anderen Kugel spiegelt. Dieses Projekt führten wir im Lauf der Modellierungswoche für Mathematik durch. An der Woche nehmen Schüler aus der ganzen Steiermark teil. Wir hatten am Anfang der Woche die Wahl zwischen 5 Projekten. Wie schon oben erwähnt haben wir 6 uns für das Projekt „Voraussage einer Beobachtung durch Strahlenverfolgung“ gewählt. Unser Hauptgrund dieses Projekt zu wählen war, dass wir wissen wollten wie die grundliegenden Programme hinter Videospielen funktionieren und diese auch selbst anwenden. Das Programm, das für die Film- und Videospielindustrie essentiell ist, ist der Strahlenverfolgungs-Algorithmus. Diesen konnten wir in einfacher Form am Ende der Woche selbst programmieren. Durch die Verwendung des Programm Matlab ist es uns gelungen, realistische Objekte am Computer darzustellen.

Wir erwarteten uns einen kleinen Einblick in die Welt der Computergrafik, da es sich anfangs noch ziemlich leicht anhörte, den Strahlenverfolgungs-Algorithmus zu programmieren. Diese Erwartungshaltung wurde uns zwischenzeitlich genommen, da nur 2 Personen aus unserer Gruppe Vorkenntnisse im Programmieren hatten. Am Anfang fiel es uns relativ schwer die Funktionsweise des Computers nachzuvollziehen, aber innerhalb eines Tages, an dem wir programmierten, waren wir alle ziemlich sicher, bei dem was wir taten. Bis zuletzt kannten wir uns gut genug aus um zu wissen wie der Strahlenverfolgungs-Algorithmus, den wir programmiert hatten, funktioniert. Die Schwierigkeiten der Gruppe lagen, mit zunehmender Länge des Programmes, die Fehler in der Funktion zu finden. Zur Arbeitsweise bleibt nur zu sagen, dass unser Projektleiter uns die grundsätzliche Aufgabenstellung in Teilen erklärte und wir selbstständig die Lösung dafür finden mussten. Am zweiten Tag an dem wir an unseren Programmen arbeiteten war es schon notwendig eine neue Funktion zu erstellen, da jeder von uns unterschiedliche Funktionen hatte, die aber alle das gleiche Ergebnis erzielten.

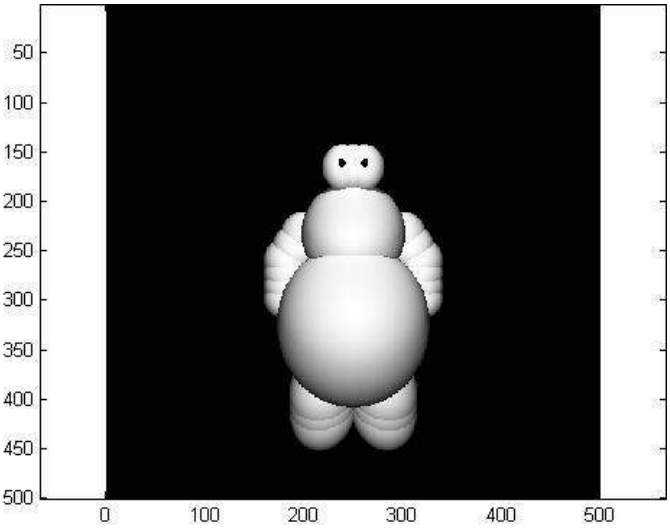
9 Best of



Mehrere Kugeln; mehrere Ebenen; Spiegelungen; Schatten; Farben; Licht; alles kombiniert

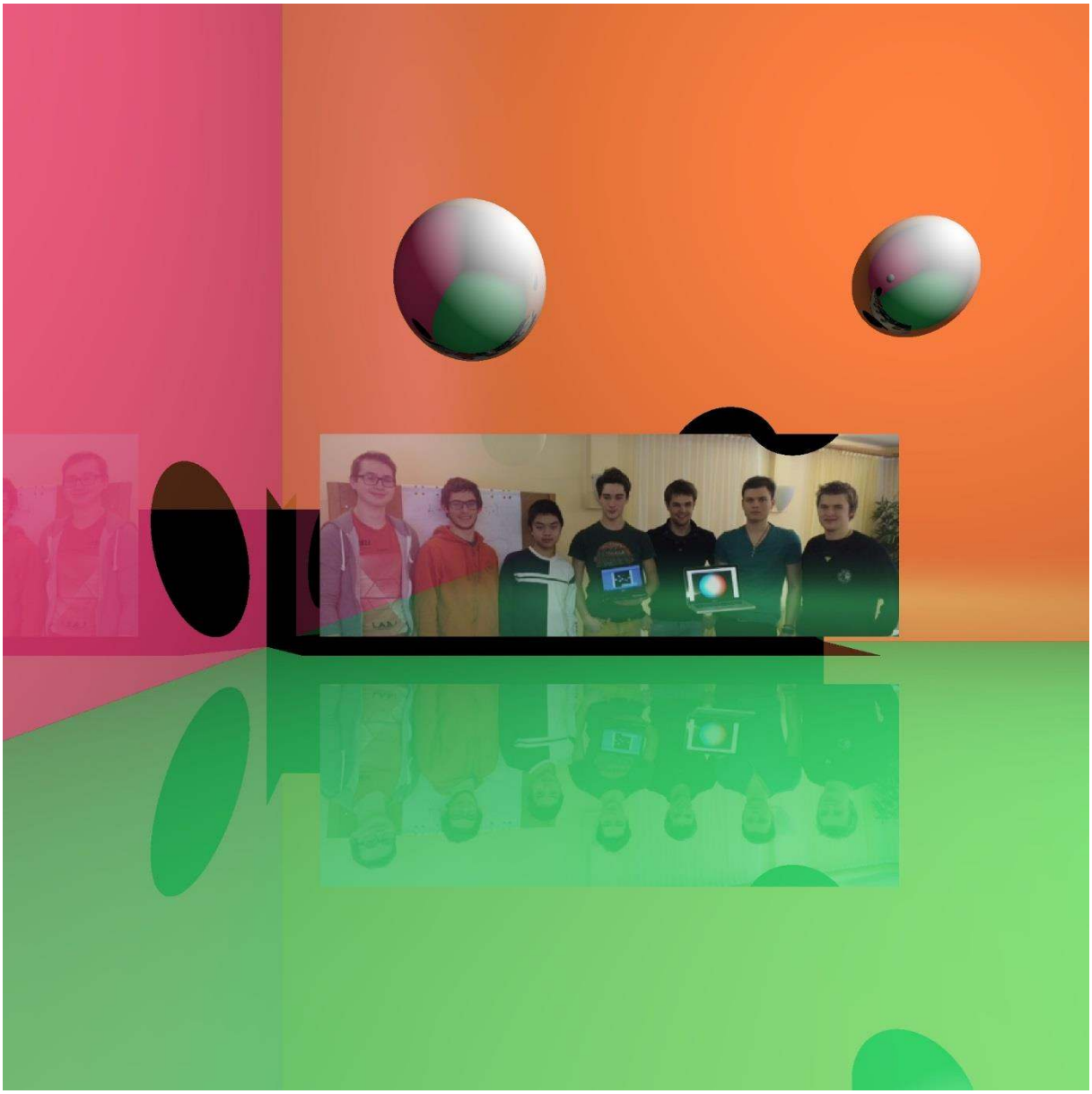
Baymax

http://cinemaforever.blog.de/2015/01/09/baymax-riesiges-robowabohu-usa-2014-kritik-walt-disney-animationsschmiede-empfaengt-marvel-superhelden-ueberdruss-19949610/



Modellierter Baymax mit Kugeln

PS: das ist kein Schneemann



Projekt Dynamische Systeme

Chaos und fraktale Geometrie

Niklas Tscheppe, Fuad Farajov, Sebastian Kölbl, Johannes Droschl,
Gerda Prach, Felix Breuß, Florian Kruse (Betreuer)

13. Februar 2015

Inhaltsverzeichnis

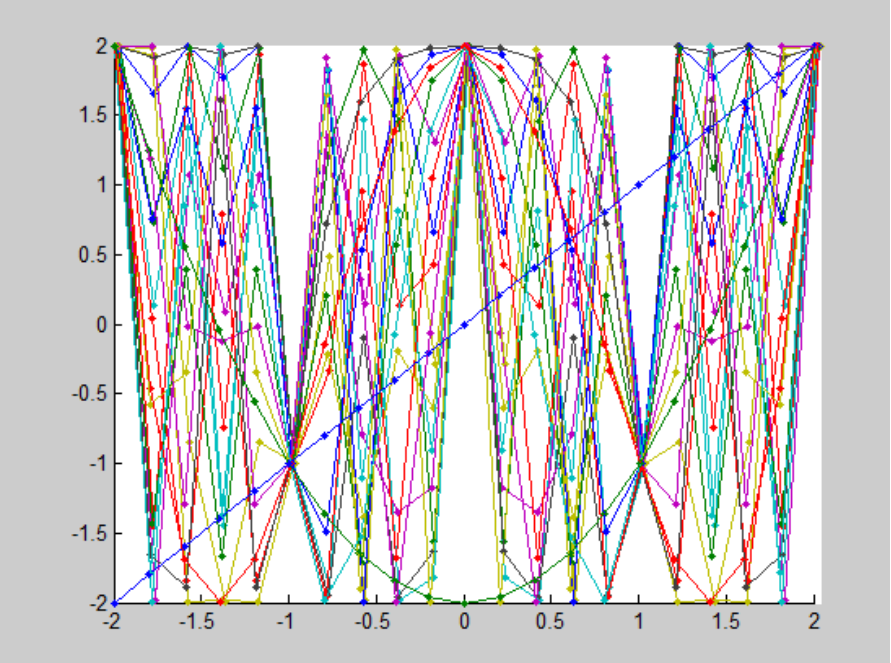
1 Folge

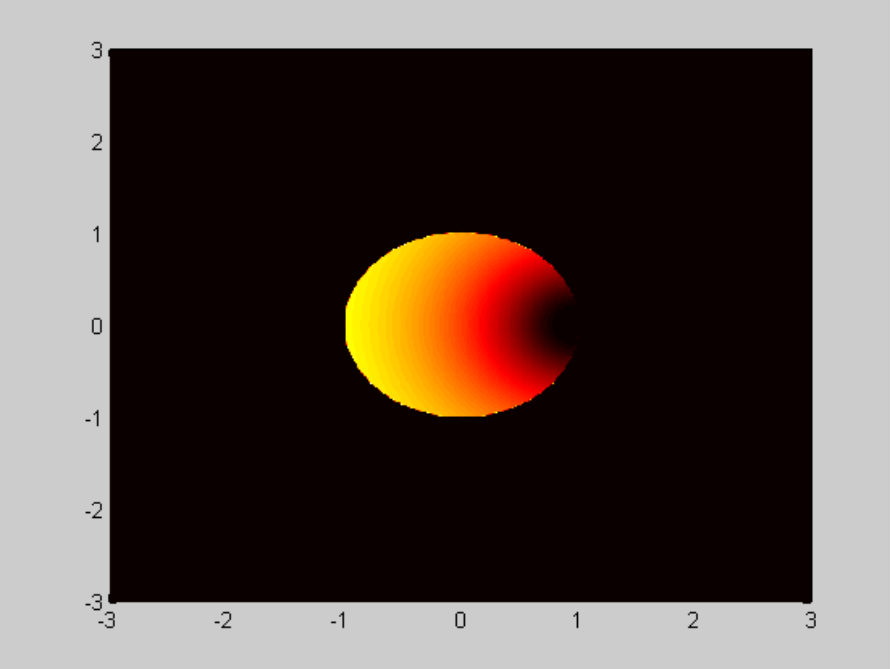
Als Beispiel für eine Folge betrachteten wir $a_{n+1} = a_n^2 - 2$. Nun galt für uns herauszufinden, ob die Folge gegen einen Grenzwert konvergiert oder „wegläuft“. Nach reichlicher Überlegung fanden wir heraus, dass die Werte der Folge im Bereich kleiner -2 und größer 2 „weglaufen“ würden und in dem Bereich dazwischen Werte liegen, die sich als Fixpunkte herausstellten sollten, d.h., dass sie auch nach einer beliebigen Anzahl an Iterationen noch immer denselben Wert haben. Jedoch war es für uns zu aufwendig viele Iterationen (wiederholtes Rechnen einer Folge mit bereits berechneten Zwischenwerten) mit vielen Werten mit kleinem Abstand zu berechnen, und so schrieben wir ein Programm, das das für uns erledigen sollte. Bei zu vielen Iterationen entstanden Rundungsfehler, weil durch das Quadrieren der Zahlen irgendwann zu viele Nachkommastellen entstanden mit denen der Computer nicht mehr rechnen kann. Die besten Beispiele für so eine Situation sind Zahlen mit unendlich vielen Nachkommastellen wie z.B. Wurzel von zwei; wenn man das Ergebnis anschließend quadriert, erhält man nicht mehr exakt zwei. Diese Rundungsfehler haben jedoch nur für einzelne Werte schlimme Auswirkungen, wie wir herausgefunden haben, so ergibt die Folge mit dem Startwert Wurzel von zwei nach mehr als 10^8 Iterationen zwei, weil der Computer auf zwei rundet, bei Wurzel von drei als Startwert ergibt die Folge jedoch schon nach 102 Iterationen einen ganz anderen Wert als den erwarteten. Mit der expliziten Darstellung der Formel $y_n = 2 \cdot \cos(2^n \cdot \pi \cdot \tau_0)$ kann man diesen Rechenfehler der rekursiven Darstellung umgehen. Des Weiteren erkannten wir, dass wenn τ_0 zwischen null und eins liegt, ein Bruch mit ganzen Zahlen und kein Vielfache von π ist, ist die Folge periodisch, andernfalls ist die Folge ein Chaos. Dem entsprechend gibt es unendlich viele Stellen, an denen die Folge periodisch und unendlich viele Stellen an denen die Folge nicht periodisch ist. Beispielsweise für $\tau = 1/3$ ist die Folge periodisch, da immer wieder die gleichen Reste bei der Division durch 2π entstehen. Für $\tau = 1/\pi$ ist die Folge nicht periodisch, also ein Chaos, weil sich das π aus der Rechnung kürzt und dadurch keine Wiederholung zustande kommt.

In der Grafik unten sieht man, dass -1 und 2 Fixpunkte sind. Dies erkennt man daran, dass nur ein Wert auf der y-Achse eingezeichnet ist, da die Folge nach beliebig vielen Iterationen immer noch denselben Wert hat.

2 Matlab-Code beschleunigen

Eines unserer größten Probleme war, dass das Programm zu lange gedauert hat. Wir haben uns mithilfe einer implementierten Matlabfunktion angeschaut, in welcher Zeile der Code am längsten braucht. Dadurch entdeckten wir, dass das Zeichnen sehr viel Zeit in Anspruch nahm. Und so suchten und probierten wir verschiedene Zeichenfunktionen um den Code zu beschleunigen, was dazu führte, dass das Zeichnen nicht mehr bis zu 16 Sekunden dauerte, sondern innerhalb einer Sekunde abgeschlossen war. Eine sehr nützliche Funktion von Matlab ist, dass es sehr schnell mit Vektoren und Matrizen rechnen kann. Dies machten wir uns zu Nutzen und rechneten daher nicht alle Punkte extra, daher in Schleifen und hintereinander, sondern in Matrizen, was den Vorgang erheblich beschleunigte, da alle Punkte auf einmal berechnet werden. Bei der Mandelbrotmenge berechneten wir gleichzeitig zwei Iterationen um den Code doppelt so schnell zu machen.





3 Potenzreihen

Zu Beginn hatten wir die Reihe $1+z+z^2+\ldots+z^n = \sum_{i=0}^{n} z^i = S$ gegeben. Durch Umformen bekamen wir $S = \frac{1-z^{n+1}}{1-z}; z \neq 1$ und daraus wiederum $\frac{1}{1-z} - \frac{1}{1-z} \cdot \lim_{n\to\infty} z^n; z \neq 1$. Wir betrachteten nur den Fall $|z| < 1$. Daraus Folgt die „geometrische Reihe“ $f(x) = \frac{1}{1-z}; z \in \mathbb{C}$.

Mit Matlab dargestellt (siehe Abbildung oben) sahen wir die Startwerte. Je heller die Punkte dargestellt werden, desto kleiner ist der Betrag der Summe S der Folge nach n

Iterationen.

Alle Punkte mit Ausnahme von null konvergieren früher oder später zu unendlich, null bleibt nach einer Iteration bei eins stehen. Des Weiteren entdeckten wir eine besondere Eigenschaft der Reihe, und zwar, dass die Reihe auch als die Summe $\sum_{i=0}^{\infty} a_i \cdot z^i$, also $R(z) = a_0 \cdot z^0 + a_1 \cdot z^1 + a_2 \cdot z^2 + \dots$ dargestellt werden kann. So konnten wir durch Ableiten an der Stelle z = 0 die Werte von a berechnen. Diese berechneten wir mithilfe von $R^{(n)} = n! \cdot a_n$.

4 Komplexe Zahlen

4.1 Allgemeines

Komplexe Zahlen erweitern den Zahlenbereich der reellen Zahlen. Somit ist die Gleichung

$$z^2 + 1 = 0$$

lösbar. Dabei wird eine neue Zahl **i** eingeführt, die die Eigenschaft $i^2 = -1$ hat. Diese Zahl **i** wird als imaginäre Einheit bezeichnet.

Komplexe Zahlen können in der Form

$$a + b \cdot i$$

dargestellt werden, wobei a und b reelle Zahlen sind und **i** die imaginäre Einheit ist. Auf die so dargestellten komplexen Zahlen lassen sich die üblichen Rechenregeln für reelle Zahlen anwenden, wobei i^2 stets durch -1 ersetzt werden kann. Für die Menge der komplexen Zahlen wird das Symbol C verwendet.

Der so konstruierte Zahlenbereich der komplexen Zahlen bildet einen Erweiterungskörper der reellen Zahlen und hat eine Reihe vorteilhafter Eigenschaften, die sich in vielen Bereichen der Natur- und Ingenieurwissenschaften als äußerst nützlich erwiesen haben.

4.2 Grundrechnungsarten bei komplexen Zahlen

4.2.1 Addition

Verläuft wie bei reellen Zahlen

$$(a + bi) + (c + di) = a + bi + c + di = (a + c) + (b + d)i$$

4.2.2 Subtraktion

Analog zur Addition gilt auch die Subtraktion

$$(a + bi) - (c + di) = a + bi - c - di = (a - c) + (b - d)i$$

4.2.3 Multiplikation

Für die Multiplikation gilt

$$(a + bi) \cdot (c + di) = ac + adi + bic + bdi^2 = (ac - bd) + (ad + bc)i$$

4.2.4 Division

Bei der Division erweitert man den Nenner mit der zum Nenner konjugierten komplexen Zahl

$$\frac{(a + bi)}{(c + di)} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{ac - adi + bic - bdi^2}{c^2 - d^2i^2} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2} \cdot i$$

4.3 Der Betrag

Der Betrag $|z|$ einer komplexen Zahl ist eine reelle Zahl

$$|a + bi| = \sqrt{a^2 + b^2}$$

Der Betrag ist die Länge eines Vektors in der Zahlenebene vom Ursprung aus.

4.4 Gleichungen und Wurzeln

Quadratische Polynome löst man auch mit den quadratischen Lösungsformeln

$$z_{1,2} = -\frac{p}{2} + -\sqrt{(\frac{p}{2})^2 - q}$$

Dabei erhaltet man immer 2 Lösungen, manche sind reell, andere nicht. Ob eine Lösung reell ist oder nicht kann man unter der Diskriminante (Wurzel) einsehen. z.B.

$$z^2 + 1 = 0$$

$$z = + - \sqrt{-1} = + - i$$

4.5 Realteil und Imaginärteil einer komplexen Zahl

Ist $z = x + y \cdot i$, so bezeichnet man

$$Re(z) = x$$

als Realteil von z und

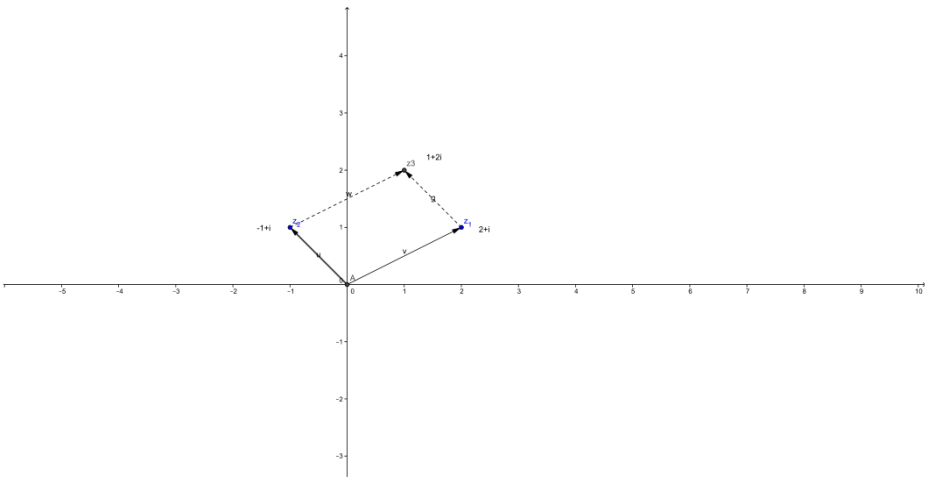
$$Im(z) = y$$

als Imaginärteil von z.

4.6 Komplexe Zahlenebene

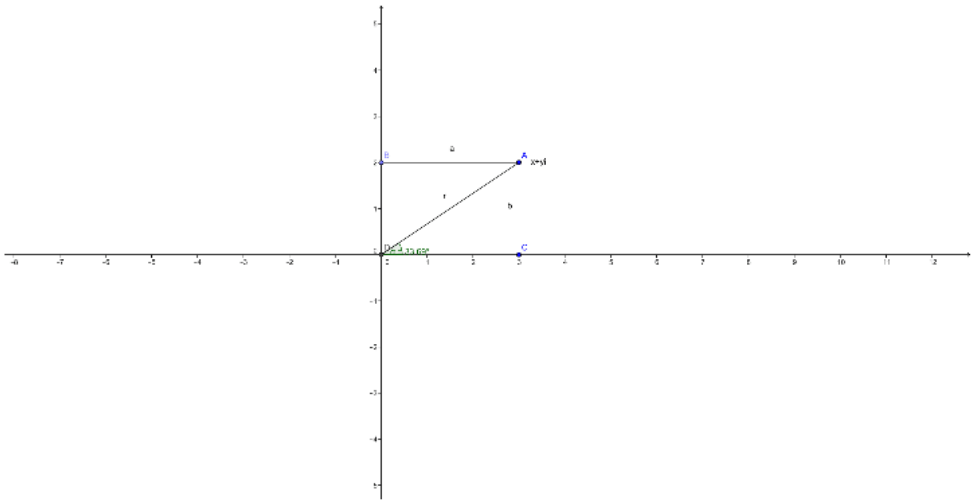
Die komplexe Zahlenebene wird auch die Gauß'sche Ebene genannt. Die Ebene ist ein Koordinatensystem. Auf die x-Achse wird der Realteil einer komplexen Zahl z und auf die y-Achse wird der Imaginärteil der Zahl z aufgetragen.

4.6.1 Geometrische Bedeutung von komplexer Addition



Bei diesem Beispiel werden zwei komplexe Zahlen addiert und eine neue Zahl entsteht in der Ebene.

4.6.2 Geometrische Bedeutung von komplexer Multiplikation



Bei diesem Beispiel ist z mithilfe von Polarkoordinaten dargestellt.

Um den Punkt, durch den eine komplexe Zahl $z = x + yi$ in der komplexen Ebene dargestellt wird, anzugeben, können wir anstelle von x und y auch dessen Abstand r vom Ursprung und den im Gegenuhrzeigersinn gemessenen Winkel φ , den der Ortsvektor mit der positiven reellen Achse einschließt, verwenden.

Somit ist

$$x = r \cdot \cos \varphi$$

und

$$y = r \cdot \sin \varphi$$

r heißt Radialkoordinate, φ heißt Polarwinkel. Eine komplexe Zahl kann deswegen als

$$z = r \cdot (\cos \varphi + i \sin \varphi)$$

angeschrieben werden. Die Radialkoordinate r gibt den Betrag von z an.

$$|z| = r$$

Auch die Eulersche Formel zu den Polarkoordinaten ist wichtig zur Bestimmung der Positionen von komplexen Zahlen in der Ebene.

$$r \cdot e^{i\varphi} = r \cdot (\cos \varphi + i \sin \varphi) = z$$

Die Eulersche Formel haben wir nicht direkt hergeleitet, sondern nur verwendet. Die einfachste Methode, dies zu beweisen, ist über die **Taylorreihe**. Man bildet dann eine unendliche Reihe der Exponentialfunktion und trennt den Realteil vom Imaginärteil. Die beiden Reihen, die hier als Real- und Imaginärteil auftreten, sind genau die Taylorreihen der Cosinus- und der Sinusfunktion. Somit gilt die Gleichung $r \cdot e^{i\varphi} = r \cdot (\cos \varphi + i \sin \varphi) = z$.

Eine Multiplikation von Polarkoordinaten:

$$z_1 \cdot z_2 = r_1(\cos \varphi_1 + i \sin \varphi_1) r_2(\cos \varphi_2 + i \sin \varphi_2)$$

$$z_1 \cdot z_2 = r_1 r_2(\cos \varphi_1 \cos \varphi_2 - \sin \varphi_1 \sin \varphi_2 + i(\cos \varphi_1 \sin \varphi_2 + \sin \varphi_1 \cos \varphi_2))$$

Die beiden Kombinationen von Winkelfunktionen, die hier auftreten, haben eine einfache Bedeutung:

$$\cos\varphi_1 \cos\varphi_2 - \sin\varphi_1 \sin\varphi_2 = \cos(\varphi_1 + \varphi_2)$$

$$\cos\varphi_1 \sin\varphi_2 + \sin\varphi_1 \cos\varphi_2 = \sin(\varphi_1 + \varphi_2)$$

Daraus kann man den Summensatz herleiten:

$$z_1 \cdot z_2 = r_1 r_2(\cos(\varphi_1 + \varphi_2) + i\sin(\varphi_1 + \varphi_2))$$

Beweis für Summensatz: Multipliziert man $e^{i\varphi_1} = (\cos\varphi_1 + i\sin\varphi_1)$ mit $e^{i\varphi_2} = (\cos\varphi_2 + i\sin\varphi_2)$, so erhält man

$$\cos\varphi_1 \cos\varphi_2 - \sin\varphi_1 \sin\varphi_2 + i(\cos\varphi_1 \sin\varphi_2 + \sin\varphi_1 \cos\varphi_2)$$

Andererseits ist $e^{i\varphi_1} e^{i\varphi_2}$ gleich $e^{i(\varphi_1+\varphi_2)}$. Somit ist

$$\cos(\varphi_1 + \varphi_2) + i\sin(\varphi_1 + \varphi_2)$$

Jetzt ist der Summensatz bewiesen.

4.7 Lösen von Polynomen mit komplexen Zahlen

Bei der Gleichung $z^3 - 1 = 0$ folgt $z^3 = 1$. Da diese Gleichung ein Polynom mit dritter Potenz ist, gibt es drei Lösungen z_1, z_2, z_3.

$$z_1 = a_1 + ib_1$$

$$z_2 = a_2 + ib_2$$

$$z_3 = a_3 + ib_3$$

Wenn man jetzt die dritte Wurzel zieht, erhält man eine Nullstelle

$$z_1 = 1$$

Nun verwendet man die Polynomdivision. Dabei nimmt man die konjugierte Zahl der Nullstelle, also aus der Nullstelle 1 wird -1. Durch geschicktes Herausheben erhält man eine quadratische Gleichung.

$$(z - 1)(z^2 + z + 1) = 0$$

Man erhält die zwei weiteren Nullstellen:

$$z_2 = \frac{-1 + \sqrt{3}i}{2}$$

$$z_3 = \frac{-1 - \sqrt{3}i}{2}$$

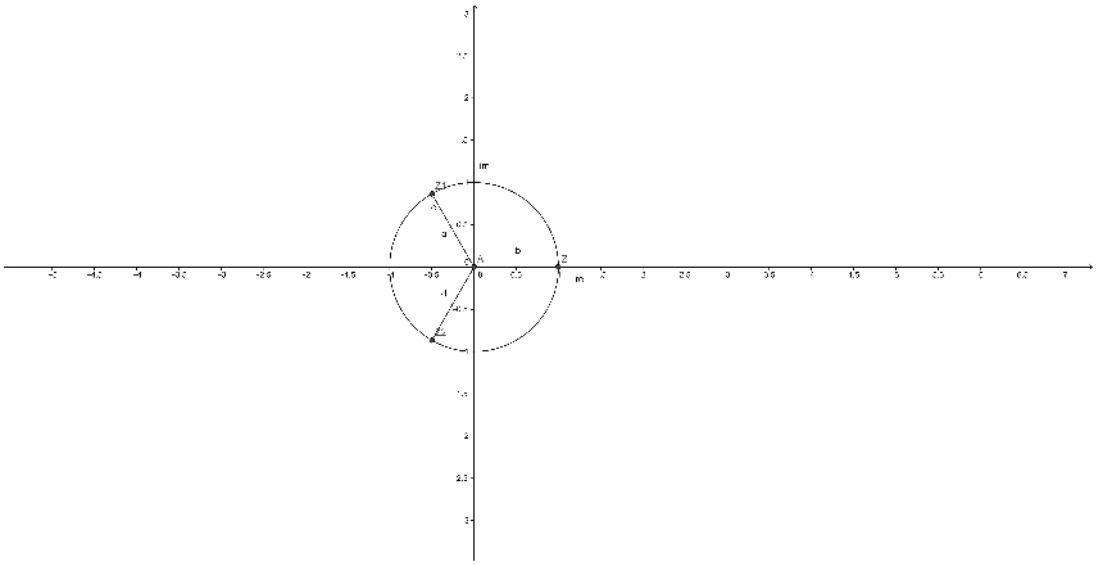
Man kann das Ganze aber auch anders darstellen, nämlich mit Polarkoordinaten.

$$z_1 = r_1 \cdot e^{i\varphi_1}$$

$$z_2 = r_2 \cdot e^{i\varphi_2}$$

$$z_3 = r_3 \cdot e^{i\varphi_3}$$

Da $|z| = r$ gilt, kann man die Polarwinkel berechnen. Da $z^3 = r^3 \cdot e^{3\varphi i}$, muss r^3 1 sein und $e^{3\varphi i}$ auch 1 sein. e^{3xi} kann nur 1 sein, wenn $e^{3\cdot 0i}$ ist. Das heißt $3\varphi = 0$. Also $\varphi = \frac{0}{3}$. Aber nicht nur 0 als Winkel ist möglich, sondern auch $\frac{2\varphi}{3}$ und $\frac{4\varphi}{3}$ und so weiter. Bei z_1 ergibt sich der Winkel 0, bei z_2 der Winkel $2/3 \cdot \pi$ und bei z_3 der Winkel $4/3 \cdot \pi$.



Man sieht die drei Lösungen, die zusammen ein gleichseitiges Dreieck bilden, der Gleichung $z^3 - 1 = 0$ in der Zahlenebene.

4.8 Ableitungen

Eine Ableitung beschreibt die Steigung in einem bestimmten Punkt in einer Funktion. Die Ableitungen im Komplexen verlaufen gleich wie im Reellen und sie werden mit Apostrophen gekennzeichnet. z.B.:

$$f(z) = z^3 + z^2 + 3z + 1$$

$$f'(z) = 3z^2 + 2z + 3$$

oder:

$$h(z) = sin(2x)$$

$$h'(z) = 2 \cdot \cos(2x)$$

$$h''(z) = 4 \cdot (-\sin(2x))$$

oder:

$$k(z) = e^z$$

$$k'(z) = e^z$$

5 Das Newtonverfahren

Das Newtonverfahren ist ein iteratives Verfahren zum Bestimmen von Nullstellen einer Funktion. Bei einer Funktion wie $f(x) = log(x) + sin(x)$ kann man mit herkömmlichen Lösungsverfahren keine Nullstellen bestimmen. Mit dem Newtonverfahren kann man zumindest eine Annäherung finden, die mit zunehmender Iterationsanzahl immer genauer wird.

5.1 Verfahren

Das Newtonverfahren ist ein Iteratives Verfahren, das heißt, dass immer wieder der gleiche Vorgang durchgeführt wird und bei jeder Iteration das Ergebnis genauer wird. Im ersten Schritt wählt man irgendeinen Punkt auf der x-Achse aus für den auch ein f(x) existiert, je näher an der vermuteten Nullstelle desto besser, weil man dann oft weniger Iterationen für einen halbwegs vernünftigen Wert braucht. Diesen ersten Wert nennt man x_0. Dann setzt man x_0 in die Funktion ein, bestimmt die Tangente in diesem Punkt und rechnet den Schnittpunkt der Tangente mit der x-Achse aus. Der neue Punkt auf der x-Achse heißt x_1. Nach n Schritten hat man dann x_n. Mathematisch formuliert schaut das dann so aus:

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

5.1.1 Herleitung

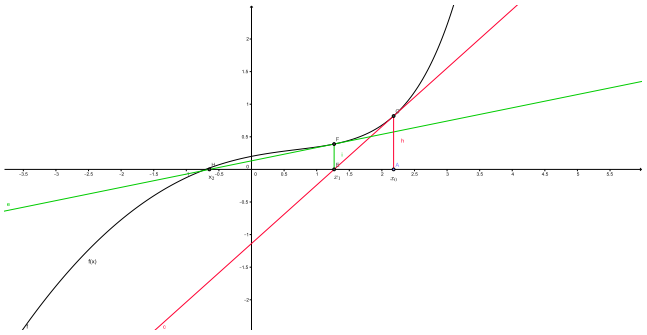
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

$$x_{n+1} f'(x_n) = x_n f'(x_n) - f(x_n)$$

$$(x_{n+1} - x_n) f'(x_n) + f(x_n) = 0$$

Jetzt kann man $(x_{n+1} - x_n)f'(x_n) + f(x_n)$ als Funktion $g(x)$ auffassen und $(x_{n+1} - x_n)f'(x_n) + f(x_n) = 0$ als $g(x) = 0$. Und das besagt, dass $k * x = -d$ was für die Nullstelle jeder konstanter Funktion gilt.

Graphisch kann man das so darstellen:



5.2 Anwendung

Wir haben uns die Funktion $f(z) = z^3 + 1$ für $z \in \mathbb{C}$ angeschaut. Genauer haben wir den Fall $f(z) = 0$ betrachtet, bei dem sich das Newtonverfahren anbietet. Wir wollten alle Zahlen in einer gleichen Farbe färben, die gegen eine der drei Nullstellen oder gar nicht konvergieren. Nach 100 Iterationen und 4 Million Punkte zwischen $10 + 10i$ und $-10 + -10i$. Dann wurden die Zahlen, die nach den Iterationen in der Nähe von 1 sind in grün dargestellt, die in der Nähe von $\frac{1+\sqrt{3}i}{2}$ sind cyan und die in der Nähe von $\frac{1-\sqrt{3}i}{2}$ sind blau. Das Ergebnis sieht so aus:

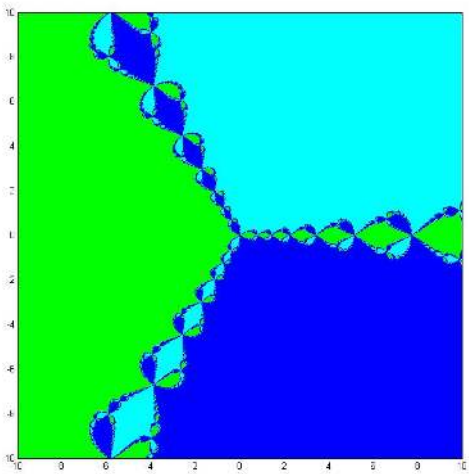


Abbildung 1: Attraktionsgebiete von $z^3 + 1 = 0$

6 Mandelbrot-Menge

6.1 Allgemein

Die Mandelbrot-Menge ist eine fraktal erscheinende Menge, die eine bedeutende Rolle in der Chaosforschung spielt. Dabei ist das Gebilde nach den strengeren Definitionen gar kein Fraktal, weil ihm die wesentliche Eigenschaft der Selbstähnlichkeit fehlt. Schaut man sich den Rand der Mandelbrot- Menge unter immer stärkerer Vergrößerung an, so entdeckt man eben nicht immer dieselben Strukturen, was Selbstähnlichkeit wäre, sondern immer neue Strukturen.

Die Mandelbrot-Menge ist die Menge (M) aller komplexen Zahlen (c), für welche die rekursiv definierte Folge komplexer Zahlen

$$z_0, z_1, z_2, ...$$

mit dem Bildungsgesetz

$$z_{n+1} = z_n^2 + c$$

und dem Anfangsglied

$$z_0 = 0$$

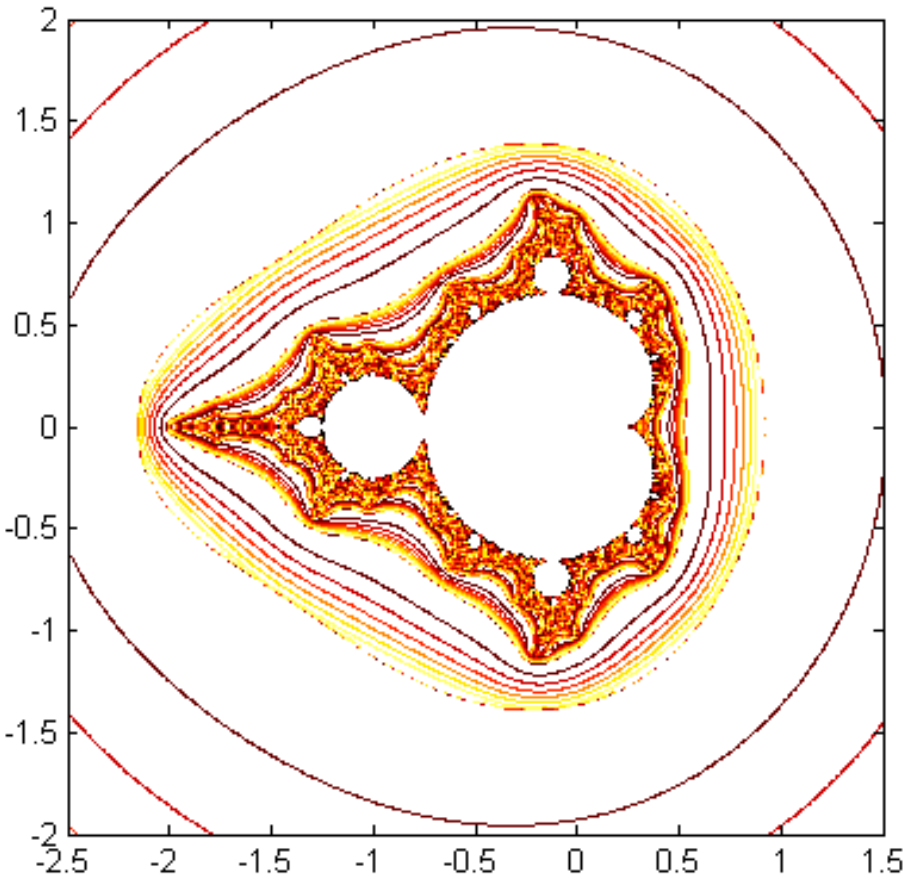
beschränkt bleibt, das heißt, der Betrag der Folgenglieder wächst nicht über alle Grenzen. Die grafische Darstellung dieser Menge erfolgt in der komplexen Ebene. Die Punkte der Menge werden dabei in der Regel schwarz dargestellt und der Rest farbig, wobei die Farbe eines Punktes den Grad der Divergenz der zugehörigen Folge widerspiegelt.

6.2 2D Darstellung

Um die Mandelbrotmenge in Matlab zweidimensional darzustellen, haben wir damit begonnen, die m-te Iteration der rekursive Funktion der Menge mit einer `for`-Schleife zu berechnen. Anschließend haben wir für die Startwerte (c) eine Matrix aus zwei Vektoren genommen. Den einen Vektor (x) haben wir mit den Werten -2 bis 2 in m Unterteilungen und den anderen (y) von -2i bis 2i in genauso vielen Unterteilungen. Von diesen Startwerten (c) haben wir anfangs alle Punkte eingefärbt, deren Betrag einen gewissen Wert (z.B. 4) überschritten hat und diese in einem Plot ausgegeben. Später haben wir mit Hilfe der `contour`-Funktion die Punkte verlaufend nach ihren Betrag eingefärbt.

```
ita = 100;
m = 501;
lim=4;
x = linspace(-2.5,1.5,m);
y = linspace(-2,2,m);
[x,y] = meshgrid(x,y);
z = x+1i*y;
c = z;
for n = 1:ita
II = boolean( 1-(abs(z)>lim) );
z(II) = (z(II).^2+c(II)).^2 + c(II);
end
figure(1);
```

```
contourf(x,y,sqrt(abs(z)));
axis image
```



6.2.1 Zoom

Da die integrierte Zoomfunktion die Punkte nicht neu berechnen kann, haben wir einen Zoom geschrieben, der das kann. Dazu haben wir einfach die `getrect`-Funktion verwendet um an dem ausgegebenen Apfelmännchen ein Quadrat zu markieren und haben die x und y-Werte hergenommen um ein neues Quadrat in der Größe des ausgewählten zu berechnen.

6.3 Mandelbrot-Menge-3d

Es gibt verschiedene Ansätze, eine dreidimensionale Darstellung der Mandelbrot-Form zu erreichen. Zuvor gab es z.B. Ansätze, einfach die klassische zweidimensionale Form zu extrudieren.

Extrusion bezeichnet in der Geometrie eine Dimensionserhöhung eines Elementes durch Parallelverschieben im Raum. Extrudiert man eine Linie oder Kurve erhält man eine Fläche. Dabei wird die Kurve entlang einer Richtung gezogen. Die Richtung kann auch

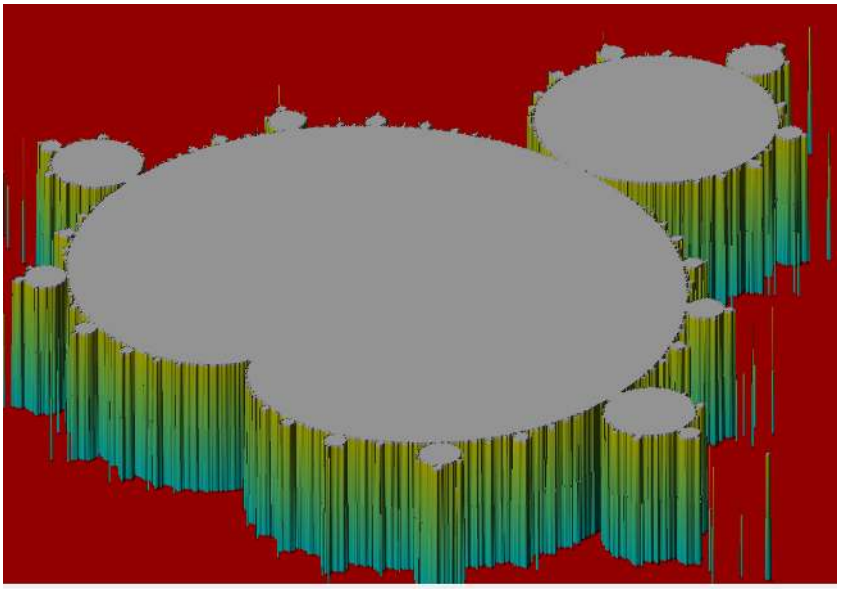
wiederum durch eine Kurve beschrieben werden. Durch Extrusion einer Fläche erhält man einen Körper mit dem Querschnitt der Fläche.

Erst 2007 kam von Daniel White der Ansatz, der eine raumgreifende Form entwickelt, die ihren Formenreichtum von Anfang an in alle drei Dimensionen entwickelt und den typischen unbeschränkten Detailreichtum in allen drei Dimensionen aufweist.

Wir aber haben die Mandelbrot-Menge zuerst in 2d programmiert, und danach noch die 3-dimensionale Form einprogrammiert. Zuerst einmal haben wir mit der der Folge begonnen, welche die Mandelbrot-Menge beschreibt. Diese haben wir in eine Schleife geschrieben, um die Mandelbrot-Menge als Fraktale darzustellen.

```
for n=1:ita
z = z.^2+c;
u = boolean(abs(z)>4);
r(u & r==0) = n;
end
```

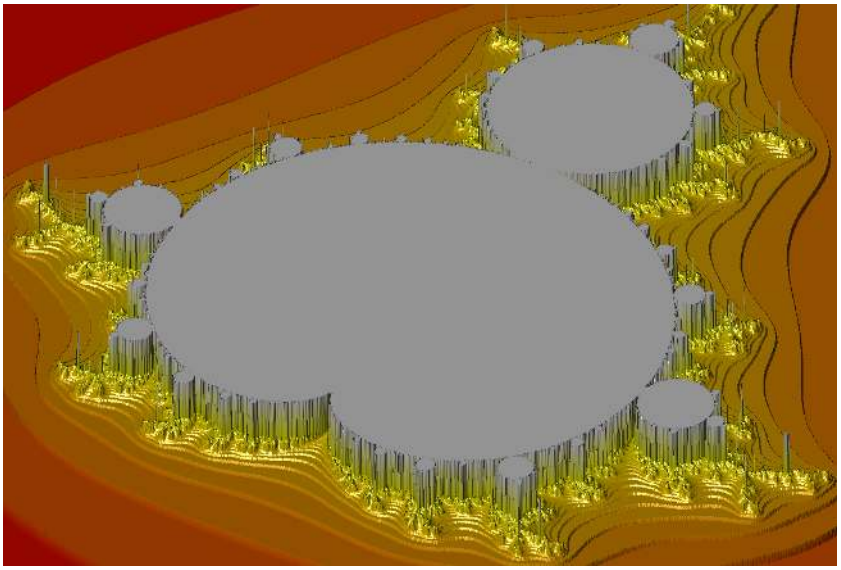
Um die Mandelbrot-Menge überhaupt 2d darstellen zu können, braucht man linspace für die x und y-Achsen. Diese Eingabe erstellt Punkte auf der x-Achse und auf der y-Achse, die auf einer beliebigen Länge platziert werden. Dazu braucht man auch noch meshgrid, welches die Punkte der x-Achse mit der y-Achse multipliziert. So entsteht eine Fläche von Punkten die ein Bild erzeugen kann. Wir haben meistens 500*500 Punkte genommen, damit wir ein schnelles Bild bekamen. Die Idee für die 3-dimensionale Form der Mandelbrot-Menge war z.B, die Beträge von z zu nehmen, und alle Punkte zeichnen zu lassen, welche die Werte >4 und <4 haben. Und so haben wir das dann auch programmiert. Das Ergebnis hat dann so ausgesehen:



Doch da das einfach eine parallele Erhöhung der Ebene darstellte, dachten wir uns aus, wie wir eine nicht abrupte Verbindung mit dem Boden erstellen könnten. Deshalb schauten wir uns die Beträge von z an, und erkannten, dass manche Beträge von z nach vielen Iterationen größer als 4 wurden, und manche Beträge schon nach einigen Iterationen, größer als 4 waren. Dann kam uns diese Idee:

```
u = boolean(abs(z)>4);
r(u & r==0) = n;
```

Diese Eingabe beschreibt wie schnell oder wie langsam der Betrag von z größer als 4 wird. Und diese Werte haben wir als Punkte vergeben. Und mit einigen Detailveränderungen, schaute es schließlich so aus:



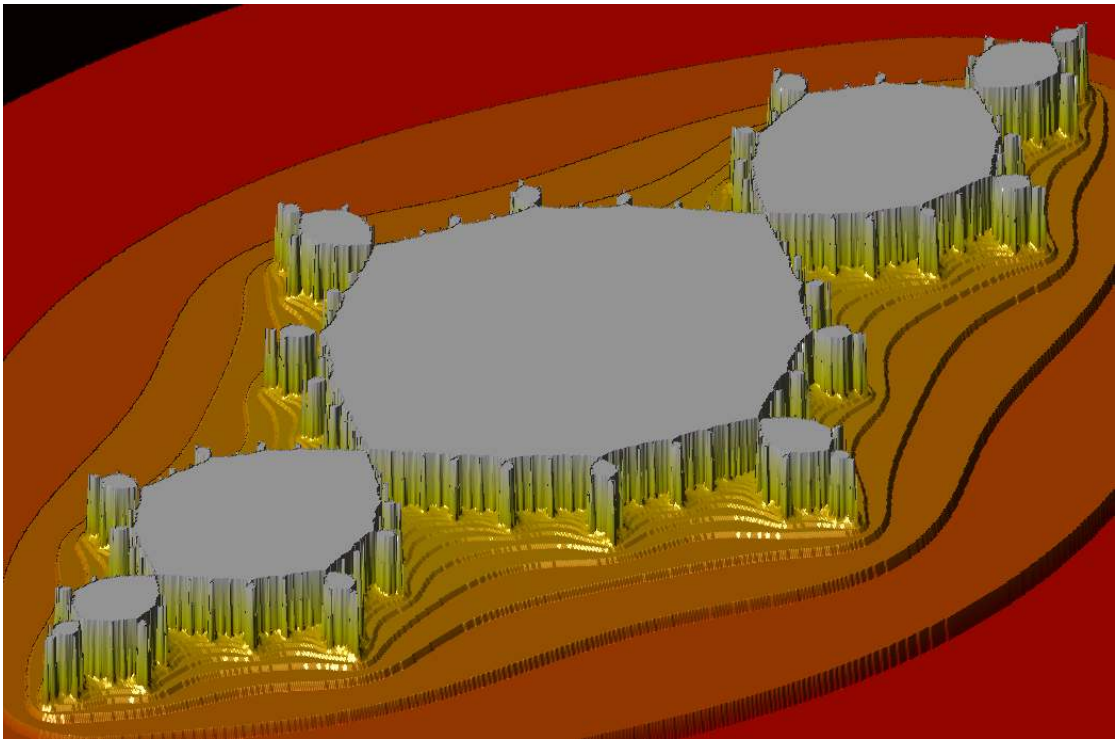
6.4 Julia-Mengen

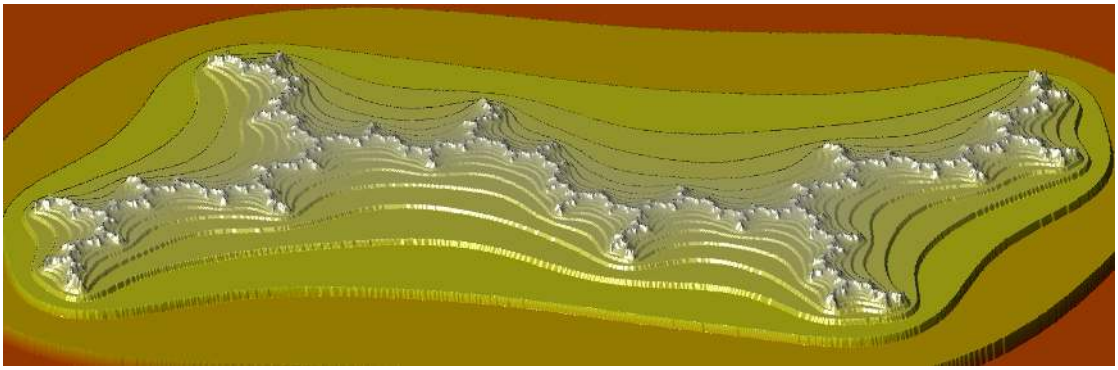
Julia-Mengen haben das gleiche Bildungsgesetz wie die Mandelbrot-Menge Sie werden dargestellt indem man statt c, eine Zahl einsetzt. Zwei Beispiele wären i, oder -1. Oft sind die Julia-Mengen fraktale Mengen. Abhängig vom Startwert c kann diese Folge zwei grundlegend verschiedene Verhalten zeigen:

1. Eine kleine Änderung des Startwertes führt zu praktisch der gleichen Folge.

2. Eine noch so kleine Änderung des Startwertes führt zu einem komplett anderen Verhalten der Folge. Die Dynamik hängt "chaotisch"vom Startwert ab. Der Startwert gehört zur Julia-Menge.

Wir haben die Startwerte -1 (oberes Bild) und i (unteres Bild) verwendet. Die Ergbenisse haben dann so ausgesehen:





7 Potenzreihen

Eine Potenzreihe ist eine Reihe der Form

$$a_0 + a_1 z + a_2 z^2 + a_3 z^3 + ...a_n z^n$$

für ein endliches n kann mithilfe verschiedener Umformung die Formel

$$S = \frac{1 - z^{n+1}}{1 - z}$$

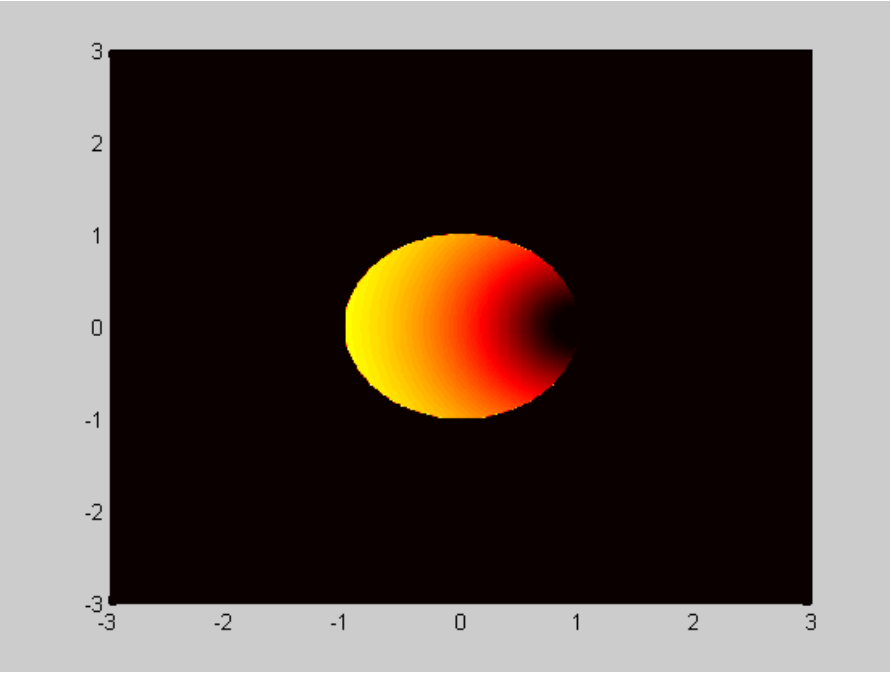
für $z \in \mathbb{C}$ für $z \neq 1$ bekommen. Für ein unendliches n ergibt sich

$$S = \frac{1}{1 - z} - \frac{1}{1 - z} \lim_{n \to \infty} z^n$$

Genauer angeschaut haben wir uns den Fall $|z| < 1$, da für alle anderen Zahlen S unendlich wird.

$$S = \frac{1}{1 - z}$$

In der Abbildung sind man komplexe Zahlen, deren Helligkeit vom Betrag nach n Iterationen abhängt.



Solche Summen lassen sich auch als

$$\sum_{k=0}^{\infty} a_k z^k$$

darstellen. Man kann zu dieser Formel kommen, indem man $z = 0$ setzt und um a_n auszurechnen, bildet man die nte Ableitung der Summe.

$$R(z) = a_0 + a_1 z + a_2 + z^2 + a_3 + z^3 + ...a_n + z^n$$

$$a_n = n! R^{(n)}(0)$$

Mithilfe dieser Notation kann man auch eine Reihendarstellung von π ausrechnen.

$$\tan(\frac{\pi}{4}) = 0$$

$$\arctan(0) = \frac{\pi}{4}$$

Da uns die Ableitung des *arctan* und deren Reihendarstellung bekannt ist

$$[\arctan(z)]' = \frac{1}{1+z^2} = \sum_{k=o}^{\infty} (-z^2)^k$$

müssen wir nur noch einen Weg finden, die a_k vom arctan durch die vom arctan$'$ auszudrücken. Das geht folgendermaßen

$$a_{n+1} = \frac{b_{n+1}}{n+1}$$

Eingesetzt ergibt es

$$\arctan(z) = \sum_{k=0}^{\infty} (-1)^{k+1} \frac{z^k}{k}$$

für $z = 1$ ergibt sich wegen $\arctan(1) = \frac{\pi}{4}$

$$\frac{\pi}{4} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} ...$$

Dokumentation der Modellierungswoche vom 7. - 13. Februar 2015

Projekt: Sozialwissenschaften

Betreuer: Mag. Dr. Stephen Keeling

Titel: Optimierung des Verkehrsflusses

Wer mit dem Auto fährt, kennt Staus, auf der Autobahn und in der Stadt. Manchmal erlebt man einen Stau, ohne bestimmten Grund dafür zu erkennen! Man glaubt, es müsste möglich sein, den Verkehrsfluss besser zu planen. Bei diesem Projekt hat man die Gelegenheit, diese Intuition zu testen. Der Verkehrsfluss wird modelliert; beeinflussende Faktoren werden untersucht, um die wichtigsten zu identifizieren. Anhand dieser Ergebnisse wird versucht den Fluss zu steuern und ihn bezüglich gewisser Kriterien zu optimieren.

Teilnehmer: Benedikt Andritsch

Konstantin Andritsch

Doris Prach

Lisa Minkowitz

Lisa Putzl

FranziskaHarich

Inhaltsverzeichnis

Erste Schritte mit NetLogo

NetLogo ist eine Mutli-Agenten-Programmierumgebung, mit der unter anderem auch sehr komplexe wissenschaftliche Systeme einfach grafisch darstellgestellt werden können. Der Umgang mit diesem Programm ist schnell erlernt, es wird jedoch auch den Anforderungen von fortgeschrittenen Programmierern gerecht.

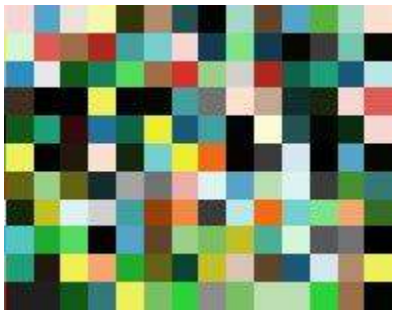
Um mit NetLogo vertraut zu werden, beschäftigten wir uns am Anfang mit den verschiedenen Tutorien zu diesem Programm. Die Oberfläche von diesem Programm ist leicht überschaubar und gliedert sich in das Interface, in dem die Befehle ausgeführt werden, und aus dem Skript, in dem der Code mit den verschiedenen Befehlen geschrieben werden. Anhand von Beispielen aus der Models Library (= eine Sammlung von vielen NetLogo-Codes) wird im ersten Tutorium der Programmierer mit der Oberfläche vertraut gemacht. Das zweite Tutorium beschäftigt sich vor allem mit dem Command Center im Interface. Das Command Center bietet viele verschiedene Möglichkeiten die Agenten, die sogenannten Turtles, beliebig zu verändern und die Grafik nach Belieben zu gestalten. Jedoch wird die Änderung nach dem Zurücksetzen mit dem Setup-Button wieder rückgängig gemacht, weshalb es praktischer ist, diese Veränderungen direkt in den Code einzubauen. Wie ein solcher Code aufgebaut ist, wird im dritten Tutorium mit einem Beispiel erklärt. Dabei bauten wir schrittweise ein kleines Programm auf, mit dem wir auf spielerische Art und Weise mit den einzelnen Befehlen vertraut wurden.

Definitionen

Um das Programm NetLogo einfacher zu verstehen, müssen zuerst ein paar Begriffe erklärt werden. Als **Turtles** werden in diesem Programm die Agenten genannt, die der Programmierer beliebig im Grafikfenster steuern kann. Es kann Form, Anzahl, Farbe, und Position eines jeden Turtles genau bestimmt werden.

Turtles mit verschiedenen Farben und Formen

Die **Patches** stellen die Raumeinheiten im Grafikfenster dar, welche ebenfalls vom Programmierer beliebig verändert werden können.



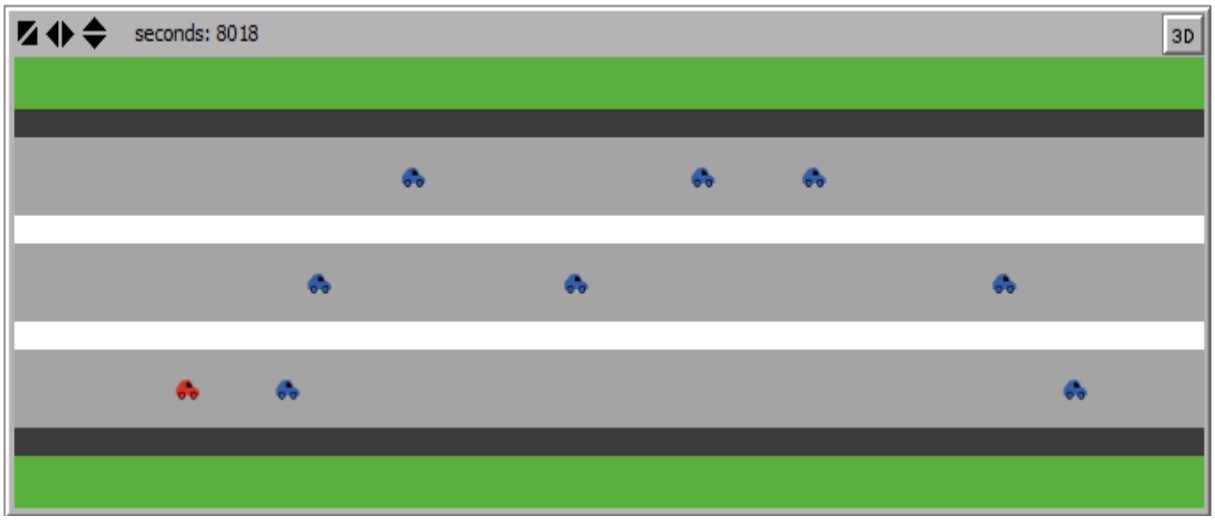
Patches mit verschiedenen Farben

Das **Grafikfenster** selbst ist das Koordinatensystem, in dem die Patches liegen. Außerdem besteht das Programm aus Stammfunktionen den sogenannten primitives, die vom Programmierer nicht definiert werden müssen, und aus Befehlen, welche er selbst in das Programm einbauen muss. Die **Ticks** ist die diskrete Zeiteinheit in NetLogo.

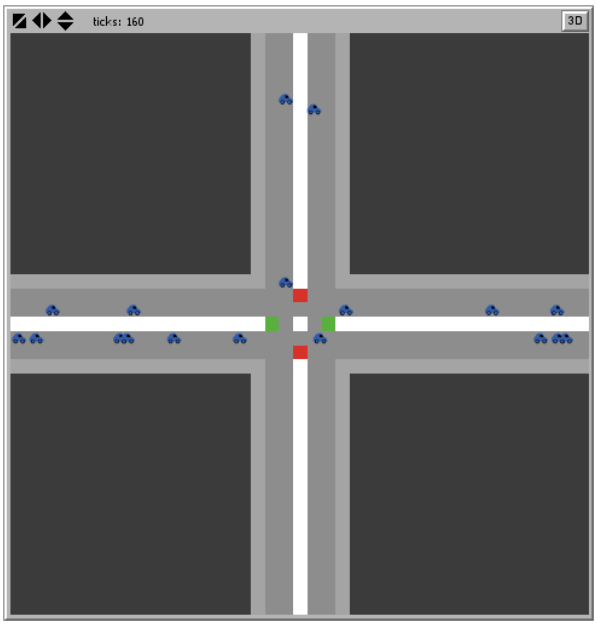
Arbeiten mit NetLogo

Um dieses Programm für unsere Arbeit zu verwenden, haben wir uns zunächst einzelne Programme aus der Models Library herausgesucht, um diese für unseren Zweck zu verwenden. So wurde zum Beispiel das Programm Traffic Basic zum Grundbaustein von zahlreichen selbst programmierten Codes. Zu sehen ist eine einspurige Straße auf der nacheinander Autos fahren. Durch die Veränderung der Patches wurde der Hintergrund beliebig verändert und so wurden aus einer einzelnen Straße schon bald Kreuzungen mit Ampeln oder Vorrangstraßen, Kreisverkehre, Autobahnen mit mehreren Spuren oder sogar ein ganzes Straßennetz. Als die grafische Modellierung eines jeden Projektes abgeschlossen war, wurden diverse Plots am Interface eingefügt, um verschiedene Daten auszuwerten. Im Plot haben wir einerseits die Geschwindigkeiten und Anzahlen der Turtles, aber auch andere wichtige Ergebnisse unserer Berechnungen grafisch dargestellt. NetLogo hat uns ermöglicht, sehr komplexe Verkehrsmodelle zu erstellen und diese dann durch zahlreiche Plots miteinander zu vergleichen.

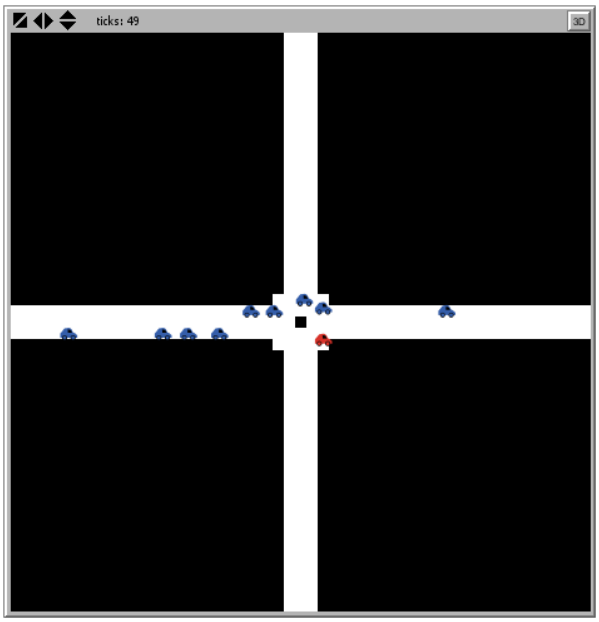
Autobahn mit 3 Spuren:



Kreuzung:



Kreisverkehr:



Erste Ideen zum Verkehr

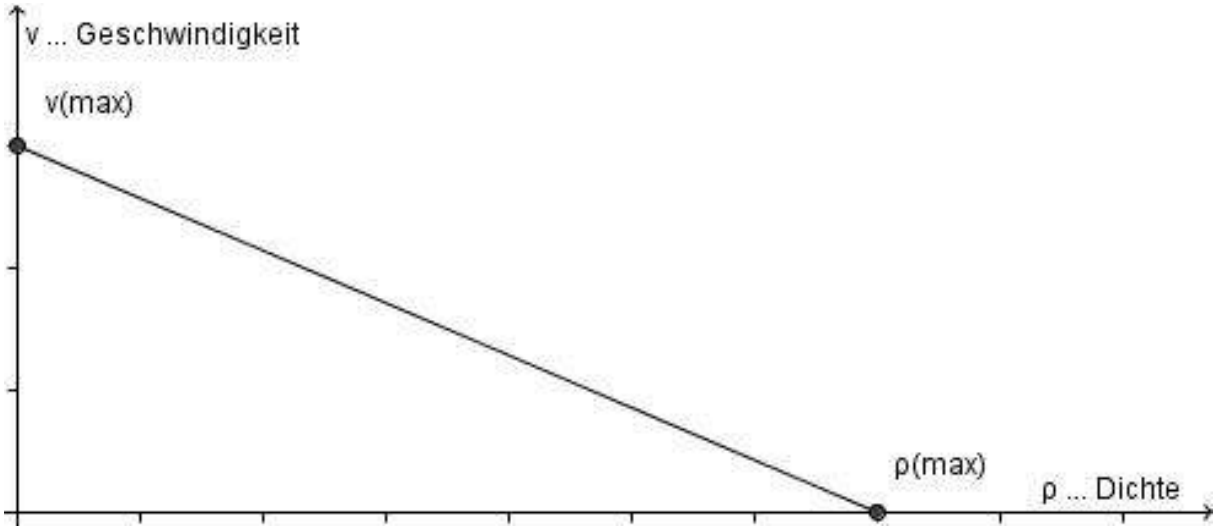
Zwei Faktoren bestimmen den Verkehr:

1) <u>die Dichte ρ</u>

Die Dichte ist bestimmt durch die Anzahl an Autos A die sich auf der Strecke befinden und durch die Länge L der ausgewählten Strecke.
Daraus ergibt sich folgende Formel: $\rho = \frac{A}{L}$

2) <u>die Geschwindigkeit v</u>

Die Größe der Geschwindigkeit ist Länge pro Zeiteinheit: L/t
Wenn die Dichte das Maximum ρ_{max} erreicht, ist die Geschwindigkeit 0, da sich kein Auto mehr bewegen kann. Geht die Dichte gegen 0 erreicht die Geschwindigkeit ihr Maximumv_{max}.
Als erstes Modell nahmen wir an, dass die Geschwindigkeit linear von der Dichte abhängt.



Aus oben genannten Beziehungen ergibt sich die Formel $v = v_{max} * (1 - \frac{\rho}{\rho_{max}})$

Der Verkehrsfluss F hat die Größe Autos pro Zeit, $\frac{A}{t} => F = \rho * v$

$\Leftrightarrow F = \rho * v_{max} * (1 - \frac{\rho}{\rho_{max}})$

$\Leftrightarrow F = v_{max} * (\rho - \frac{\rho^2}{\rho_{max}})$

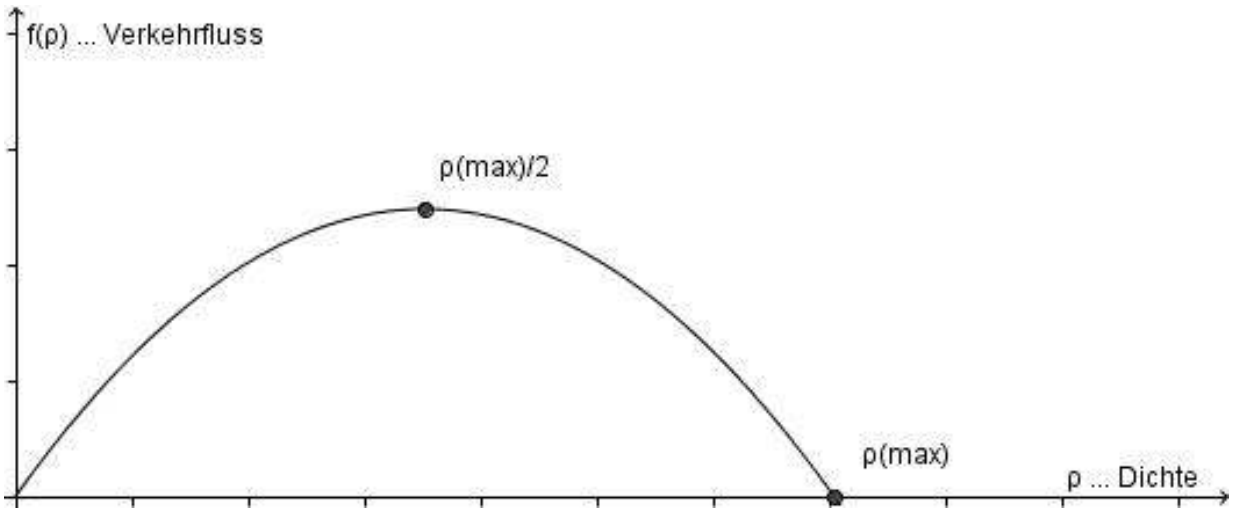
$\Leftrightarrow F' = v_{max} * (1 - \frac{2*\rho}{\rho_{max}})$

$\Rightarrow F'\left(\frac{\rho_{max}}{2}\right) = 0, F(0) = F(\rho_{max}) = 0$

⇨ F hat ein globales Maximum im Intervall $\rho \in [0, \rho_{max}]$ in $\rho = \frac{\rho_{max}}{2}$

Der Verkehrsfluss in diesem Modell ist also in $\rho = \frac{\rho_{max}}{2}$ maximal.

⇨ $F_{max} = F\left(\frac{\rho_{max}}{2}\right) = \frac{\rho_{max} * v_{max}}{4}$

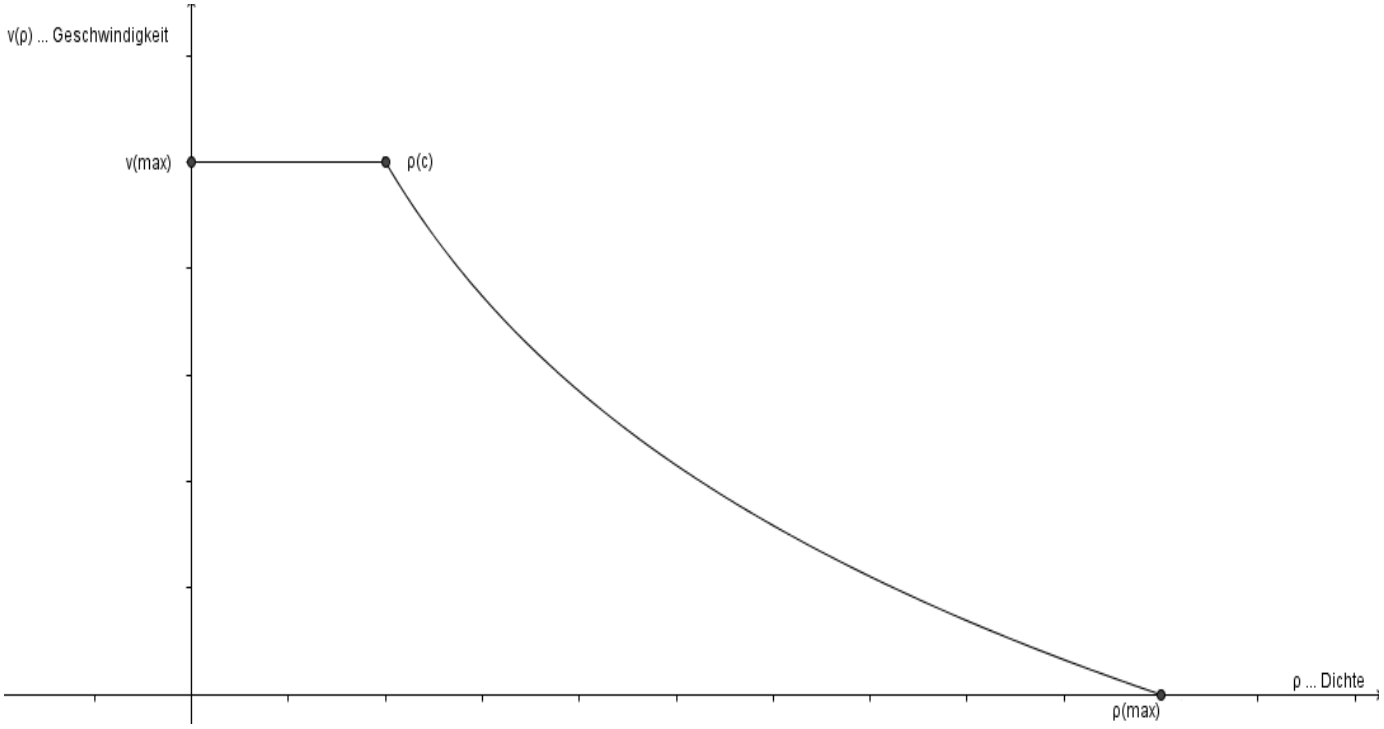


Erstes Modell (mit Netlogo)

Traffic Basic: Die Autos befinden sich auf einer Straße mit einer Spur, diese Straße ist wie ein Kreis aufgebaut, Autos, die auf der einen Seite hinausfahren, kommen auf der anderen Seite wieder heraus.
Diese Autos werden als Individuen betrachtet mit einfachen Regeln: Wenn sich ein anderes Auto in dem Feld davor befindet, wird die Geschwindigkeit des Autos reduziert.
Befindet sich jedoch kein Auto im Feld davor wird die Geschwindigkeit vergrößert bis sie ihr Maximum erreicht.

Beobachtung: Die Geschwindigkeit v ist **nicht** linear abhängig

Die Formel dieses Graphen lautet:



$$v(\rho) = v_{max} \qquad \text{für } 0 \leq \rho \leq \rho_c$$

$$v(\rho) = \frac{v_{max}}{\ln[\frac{\rho_{max}}{\rho_{max}}] * \ln[\frac{\rho_{max}}{\rho}]} \qquad \text{für } \rho_c < \rho < \rho_{max}$$

$$v(\rho) = 0 \qquad \text{für } \rho = \rho_{max}$$

Die mathematische Herleitung dieser Funktion:

Das Newton'sche Gesetz besagt:

$$-M * x_i''(t) = A * \frac{x_i'(t) - x_{i+1}'(t)}{x_{i+1}(t) - x_i(t)} = -A(\frac{d}{dt})\ln[x_{i+1}(t) - x_i(t)]$$

Mit M = Masse, A = Konstante, rechte Seite = Bremskraft, i = Autoindex, t = Zeit
Wir setzen: $\lambda = \frac{A}{M}$
Dynamisches Modell:

$x_i'(t) = \lambda * \ln[x_i(t) - x_i(t)] + \alpha_i$
$x_i' \dots v$
$x_{i+1} - x_i = \frac{1}{\rho}$
$\alpha_i = \alpha$

$$\Rightarrow \quad v = \lambda * \ln\frac{1}{\rho} + \alpha$$

$$v(\rho) = \begin{cases} 0 < \rho \leq \rho_c \rightarrow v = v_{max} \\ \rho_c < \rho < \rho_{max} \rightarrow v = ? \\ \rho = \rho_{max} \rightarrow v = 0 \end{cases}$$

$$v(\rho_{max}) = 0 = \lambda * \ln\frac{1}{\rho_{max}} + \alpha$$
$$\Rightarrow \quad \alpha = -\lambda * \ln\frac{1}{\rho_{max}} = \lambda * \ln[\rho_{max}]$$
$$v(\rho_c) = v_{max} = \lambda * \ln\frac{1}{\rho_c} + \lambda * \ln[\rho_{max}] = \lambda * \ln\frac{\rho_{max}}{\rho_c}$$
$$\Rightarrow \quad \lambda = \lambda * v_{max} / ln\frac{\rho_{max}}{\rho_c}$$

Durch Einsetzen erhalten wir oben genannte Formeln.

Da wir nun wissen, dass die Geschwindigkeit von der Dichte abhängt, gilt für die Flussformel:
$F(\rho) = \rho * v(\rho)$

Ersetzen von v:

1) $0 \leq \rho \leq \rho_c$

$F(\rho) = \rho * v_{max}$
$F'(\rho) = v_{max}$

Globales Maximum im Intervall $[0, \rho_c]$:

$$F(\rho_{\max}) = \rho_c * v_{max}$$

2) $\rho_c < \rho < \rho_{max}$

$$F(\rho) = \rho * (\lambda * \ln \frac{\rho_{max}}{\rho})$$

// Ableitungsregeln:
// $[p(x) * q(x)]' = p'(x) * q(x) + p(x) * q'(x)$
// $[ln[p(x)]]' = \frac{p'(x)}{p(x)}$

$$F'(\rho) = \lambda * \ln[\frac{\rho_{max}}{\rho}] + \lambda * \rho * (-\frac{\rho_{max}}{\rho^2})/(\frac{\rho_{max}}{\rho})$$
$$F'(\rho) = \lambda * \ln[\frac{\rho_{max}}{\rho}] - 1$$

Für das globale Maximum setzen wir jetzt $F'(\rho) = 0$

$$F'(\rho) = 0 = \lambda * \left(\ln\left[\frac{\rho_{max}}{\rho}\right] - 1\right)$$

Da $\lambda = v_{max}/\ln[\frac{\rho_{max}}{\rho_c}]$ eine Konstante ist und ungleich 0, muss gelten:

$$\ln\left[\frac{\rho_{max}}{\rho}\right] = 1$$
// $ln[e] = 1$
$$\frac{\rho_{max}}{\rho} = e$$
$$\rho = \frac{\rho_{max}}{e}$$

Globales Maximum im Intervall $[\rho_c; \rho_{max}]$:

$$F\left(\frac{\rho_{max}}{e}\right) = \frac{\rho_{max}}{e} * \frac{v_{max}}{\ln\left[\frac{\rho_{max}}{\rho_c}\right]} * \ln[\rho_{max}/\frac{\rho_{max}}{e}]$$
$$F\left(\frac{\rho_{max}}{e}\right) = \frac{\rho_{max}}{e} * \frac{v_{max}}{\ln\left[\frac{\rho_{max}}{\rho_c}\right]}$$

3) $\rho = \rho_{max}$:

$$F(\rho) = \rho * 0$$

$$F(\rho) = 0$$

Für das Maximum der Flussfunktion gibt es nun 2 Möglichkeiten:

1) $wenn\ \rho_c > (\frac{\rho_{max}}{e}) => F(\rho) = F_{max} = v_{max} * \rho_c$
2) $wenn\ \rho_c < \left(\frac{\rho_{max}}{e}\right) => F\left(\frac{\rho_{max}}{e}\right) = F_{max} = \frac{\rho_{max}}{e} * v_{max} / \ln[\frac{\rho_{max}}{\rho_c}]$

In unseren Simulationen haben wir keinen eindeutigen Faktor erkannt, der bestimmt in welchem Intervall das Maximum von F liegt.
Es kamen immer wieder unterschiedliche Ergebnisse heraus. Häufig war die Reaktionszeit ausschlaggebend dafür, dass das Maximum in $\frac{\rho_{max}}{e}$ lag.

Realitätsgetreuere Simulation des Verkehrsflusses durch Einbinden weiterer Faktoren

In weiterer Folge untersuchten wir den Einfluss von Be- und Entschleunigung und Reaktionszeit auf den Verkehrsfluss.

Wir haben 1 Meter näherungsweise in $1/5$ Patches und 1 Sekunde (je nach Programmdauer) in ungefähr 10 Ticks angegeben.

Damit war es uns nun möglich die Geschwindigkeit in km/h anzugeben. Die Umrechnungsvariable u ist leicht zu bestimmen:

$$u = 1\,{}^{km}/_{h} \Leftrightarrow u = \frac{1\,{}^{m}/_{s}}{3{,}6}$$
$$\Leftrightarrow u = \frac{1}{5}/(10 * 3{,}6) \text{ Patches/Ticks}$$

somit sind: $100 * u = 27{,}8\,{}^{m}/_{s}$

Beschleunigung

Des Weiteren ergibt sich für die Beschleunigung a durch die Formel $a = 1\,{}^{m}/_{s^2}$ folgender Wert:

$$a = \frac{1}{5}/10^2 => \frac{1}{500} \text{ Patches/Ticks}$$

Da aber jedes Auto eine unterschiedliche Beschleunigung hat, gaben wir jedem Turtle mit einer random Funktion seine eigene Beschleunigung zwischen 7 und 10 Sekunden auf 100 km/h.

Bsp. Ein Turtle kommt von 0 auf 100 km/h in 8 Sekunden

⇨ $\frac{100^{km}/_{h}}{8}$
⇔ $\frac{27.8^{m}/_{s}}{8}$
⇔ $3{,}48\ ^{m}/_{s^2}$
⇔ $\frac{3{,}48}{500}$ Patches/Ticks
⇔ 0.0069 Patches/Ticks

Also erhöht sich die Geschwindigkeit dieses Autos um 0.0069 Patches/Tick pro Tick.

Entschleunigung

Die Entschleunigung *d* hängt in unserem Modell vom vorderen Auto ab. Die Faktoren hierbei sind die Distanz zum und die Geschwindigkeit des vorderen Autos.

-> Ist die Distanz zu dem Auto groß, ist die Entschleunigung klein
-> Ist die Distanz klein, ist die Entschleunigung groß

-> Ist die Geschwindigkeit des Autos groß, ist die Entschleunigung klein
-> Ist die Geschwindigkeit klein, ist die Entschleunigung groß

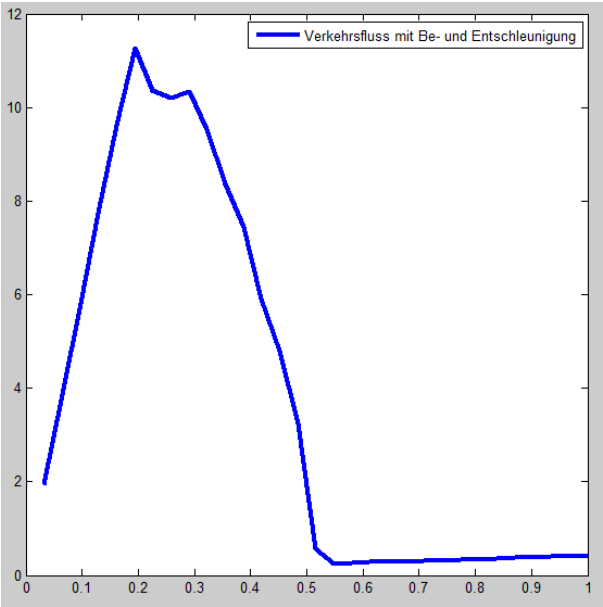
Durch Recherche kamen wir auf die Formel:

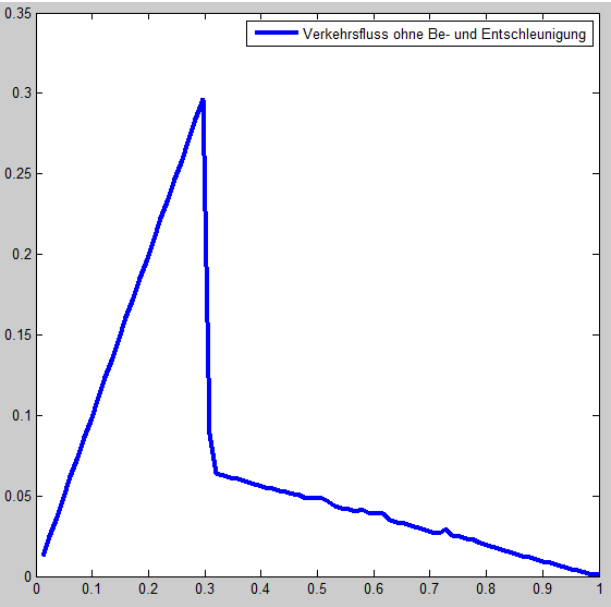
x_i ist die position des i-ten Autos
$d = \frac{(x_i\prime)^2 - (x_{i+1}\prime)^2}{2*(x_{i+1} - x_i)}$

Wir zogen allerdings nur Werte für *d* in Betracht, die größer als 0 sind.

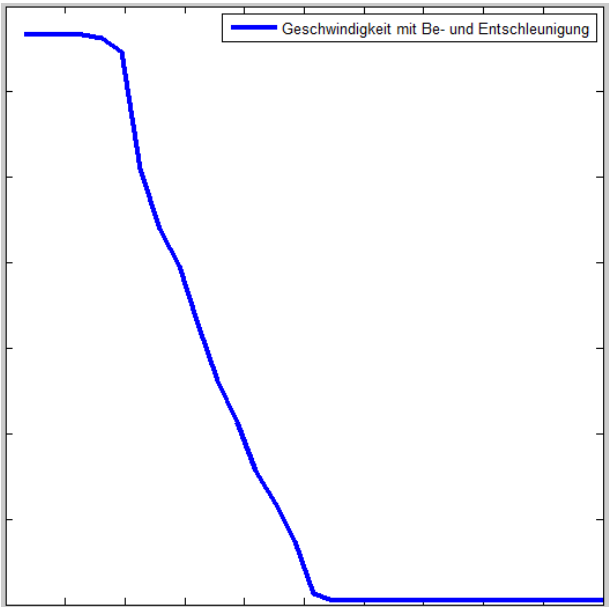
Reaktionszeit

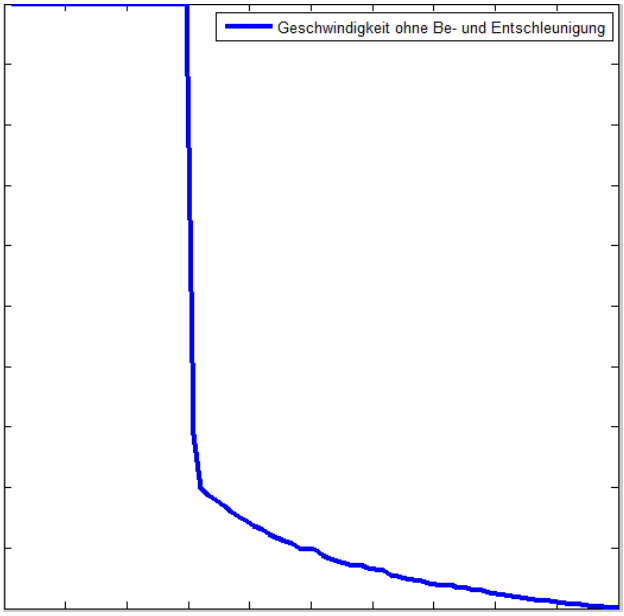
Da jeder Mensch auch eine gewisse Reaktionsverzögerung im Bereich von 0.2 bis 0.3 Sekunden hat (vgl. Wiki), haben wir auch diesen Faktor in die Simulation eingebunden.
Mit dem Wert für eine Sekunde in Ticks haben wir diese Zeit in Ticks umgerechnet und die Veränderung der Geschwindigkeit jedes Autos um diesen Wert verzögert.





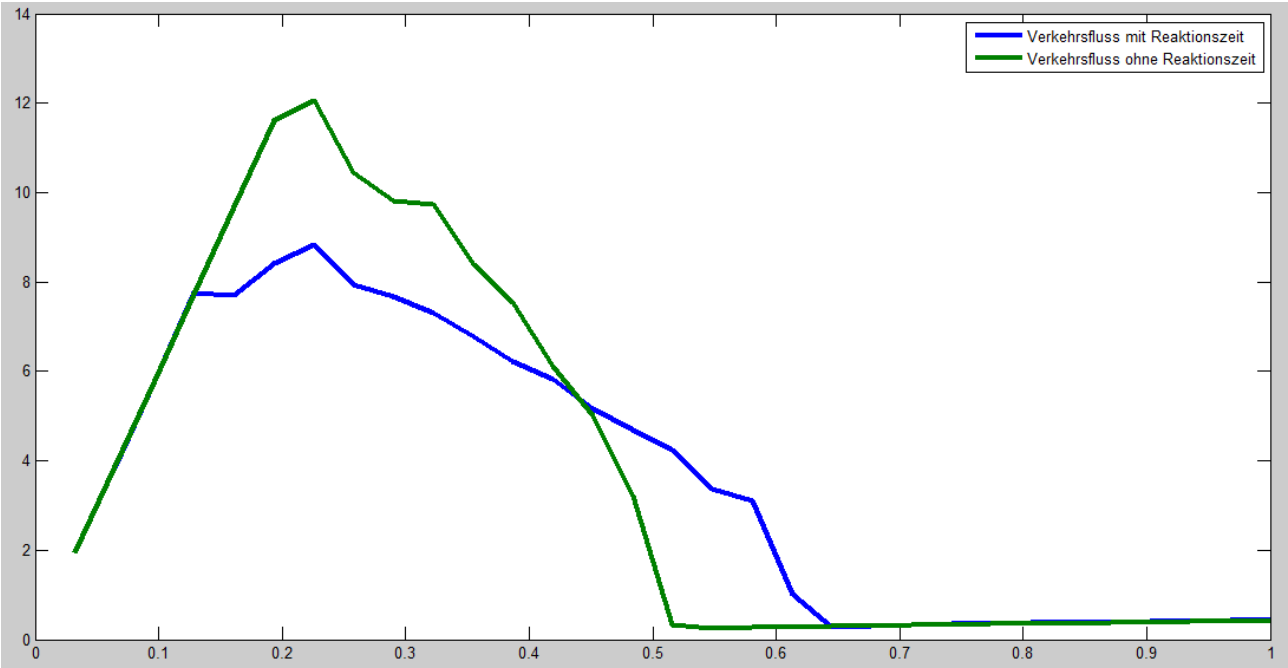
In den Bildern ist der Verkehrsfluss (y-Achse) abhängig von der Dichte (x-Achse) zu sehen. Links mit, und rechts ohne Be- und Entschleunigung.
Da sich bei geringer Anzahl an Autos kein Stau bildet, ist der erste Teil der Graphen jeweils linear steigend. Danach fällt der linke Graph sehr schnell, da die Autos nicht nur kurz bremsen, wenn sich jemand vor ihnen befindet, sondern sie bleiben sofort stehen.





Diese beiden Grafiken zeigen die maximale Geschwindigkeit (y-Achse) abhängig von der Dichte (x-Achse), wieder rechts ohne und links mit Be- und Entschleunigung.

Rechts entsteht ab einer gewissen Dichte ein Stau, da die Autos sofort stehen bleiben und der erste Autofahrer schon wieder am Ende des Staus angekommen ist, bevor sich dieser lösen kann. Links wird das durch den Faktor Bremsen besser gelöst.



Diese Grafik stellt den Unterschied zwischen dem Modell mit und ohne Reaktionszeit dar.
Die blaue Linie zeigt den Verkehrsfluss mit und die grüne den Verkehrsfluss ohne Reaktionszeit.
Deutlich zu sehen ist, dass die Reaktionszeit den Fluss geringer hält, was daran liegt, dass sich Stau nicht so leicht wieder lösen kann.

Mehrspurige Straße

Da auch in der Realität ein- und mehrspurige Straßen vorkommen, wollten wir untersuchen, welches Verhältnis zwischen dem Verkehrsfluss einspuriger und mehrspuriger Straßen besteht.

Zur Simulation haben wir wieder das Programm Netlogo verwendet.
Anstatt zu Bremsen versuchen Autos zuerst die Spur zu wechseln, um die hohe Geschwindigkeit beizubehalten. Als Beispiel verwendeten wir eine dreispurige Straße.

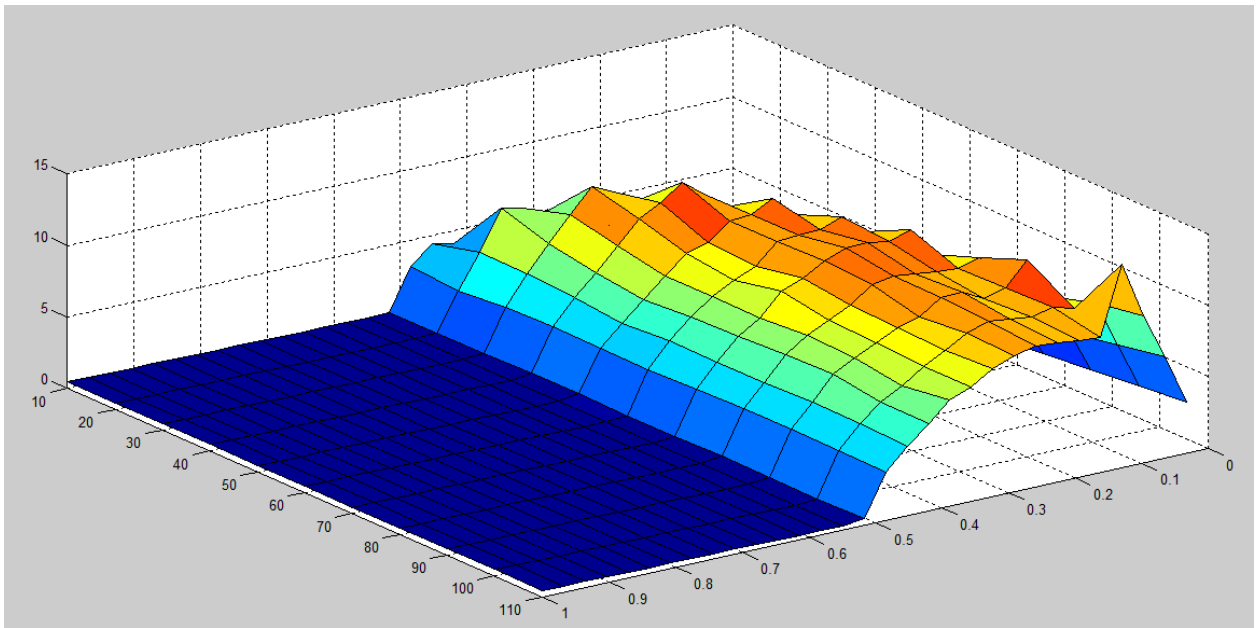
Die Dichte ρ wird nun anders berechnet:

$$\rho = \frac{Anzahl\ der\ Autos}{3*(Länge\ der\ Strecke)}$$

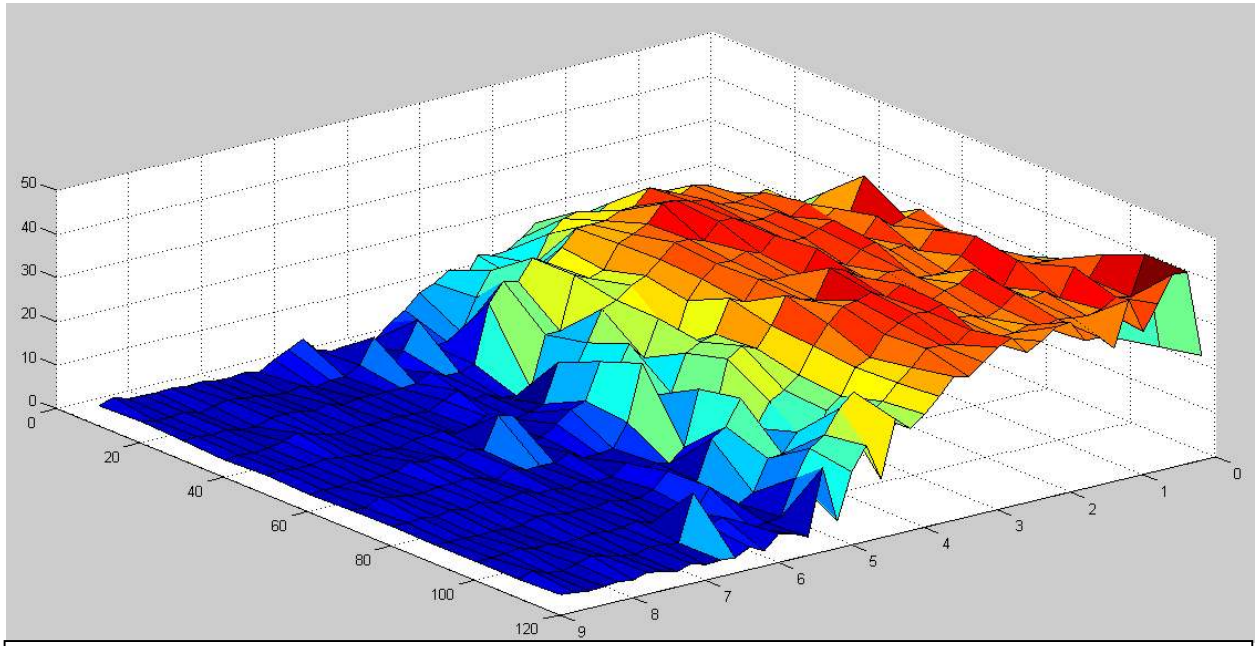
Bei der Geschwindigkeit beobachteten wir einen ähnlichen Abfall wie bei der einspurigen Simulation.
Jedoch waren beim Fluss Veränderungen zu sehen:

1) Das Maximum des Flusses war ungefähr proportional zu der Anzahl an Spuren
2) Das Ergebnis war robuster, d.h. der Fluss fiel also um das Maximum nicht so schnell wieder stark ab, sondern fiel langsamer.

Fazit: Mehrspurige Straßen sind nicht so anfällig auf Staubildungen, da sie einen größeren Spielraum der Dichte um den maximalen Verkehrsfluss aufweisen. Außerdem ist die Effizienz, also der Verkehrsfluss, direkt proportional zur Anzahl der Spuren, wie man in den folgenden Abbildungen sieht.



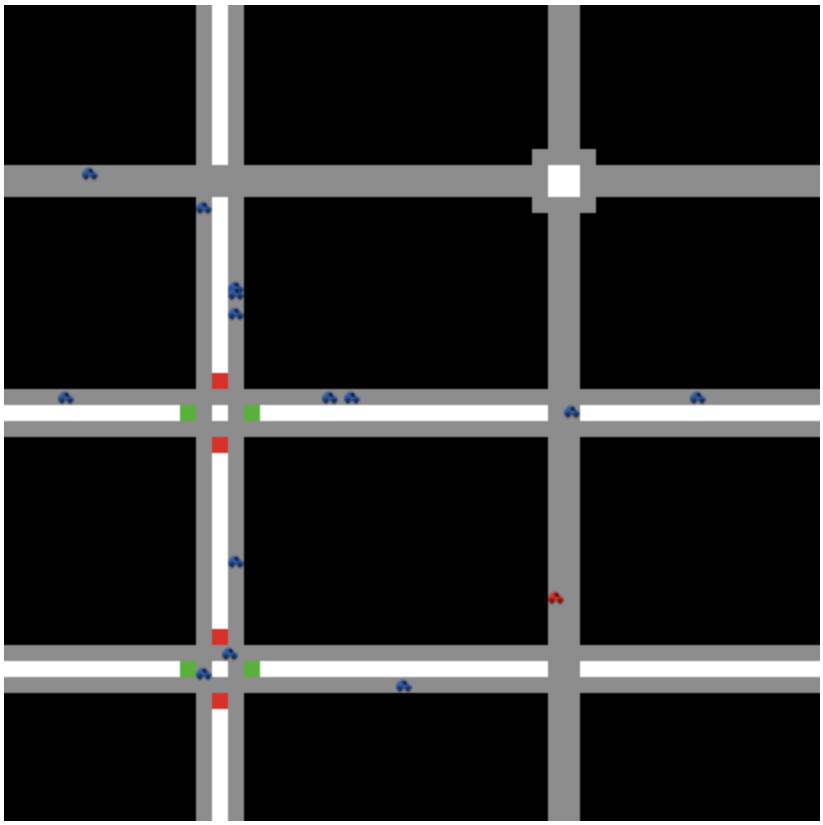
Einspurig (x-Achse: Geschwindigkeit, y-Achse: Dichte, z-Achse: Fluss)



Dreispurig (x-Achse: Geschwindigkeit, y-Achse: Dichte, z-Achse: Fluss)

Anwendung auf ein Straßennetz mit Ampel, Kreuzung und Kreisverkehr

Wie bereits im vorherigen Modell beschrieben hängt der Verkehrsfluss sehr stark mit den Faktoren Beschleunigung, Entschleunigung und Reaktionszeit zusammen. Allerdings muss ein Auto auf einer mehrspurigen Straße nur dann bremsen, wenn der Abstand zum vorderen Auto zu gering ist. Im Modell des Straßennetzes einer Stadt gibt es jedoch noch einen wichtigen Faktor, warum ein Auto sein Tempo reduzieren muss, und zwar Kreuzungen.
Zu diesem Zweck haben wir in NetLogo ein Straßennetz mit verschiedenen Arten von Kreuzungen modelliert. Einerseits gibt es Kreuzungen mit Ampelschaltung, andererseits Kreuzungen, die dadurch geregelt sind, dass eine der beiden eine Vorrangstraße ist. Zusätzlich gibt es noch einen Kreisverkehr. Die Autos wurden so programmiert, dass sie vor einer roten Ampel stehen bleiben, erst dann in eine Kreuzung fahren, wenn sich auf der Vorrangstraße kein Auto mehr nähert und auch den Autos im Kreisverkehr Vorrang geben.
Durch Beobachtungen erkannten wir, dass bei geringerer Autodichte wesentlich mehr Autos vor Kreuzungen mit Ampeln warten, als bei durch Vorrangstraßen geregelten Kreuzungen. Bei einer hohen Autodichte bildet sich vor den Kreuzungen ohne Ampeln viel Stau, deutlich mehr als bei den durch Ampeln geregelten Kreuzungen. In unserem Modell ist der Kreisverkehr die beste Möglichkeit, Stau zu vermeiden. Unabhängig von der Autodichte bildet sich dort immer am wenigsten Stau.



Kreisverkehr:

Einen längeren Aufenthalt gab es auch bei dem Kreisverkehr, während gleichzeitig der Bau einer dreispurigen Autobahn forciert wurde. Der Kreisverkehr konnte aufgrund der Unterteilung der Grundfläche in Patches leider nicht rund werden und musste daher auf eine etwas verpixelt wirkende, eckige Struktur ausweichen. Dies wurde jedoch nicht als Hindernis genommen, und einfach die Ecken verwendet, um die Autos ihre „Kurven“ fahren zu lassen. Die Turtles kommen hierbei in den Kreisverkehr und müssen zwangsläufig eine Ecke hinter sich bringen. Danach können sie sich entscheiden, ob sie weiterhin im Kreisverkehr bleiben, oder daraus ausfahren, wollen. Bleiben sie weiter im Kreisverkehr, können sie sich an jeder weiteren Ausfahrt wieder entscheiden, was sie machen wollen. Die Wahrscheinlichkeit die erste Ausfahrt zu nehmen ist also am höchsten. Dies könnte jemand auch in der Grafik beobachten. Aufgrund dessen mussten alle Ecken einzeln definiert werden, und es ging leider Gottes ein Gutteil an Zeit für die Fehlersuche drauf.
Die Überlegung war ursprünglich, dass ein Kreisverkehr eigentlich effizienter als eine einfache Kreuzung sein müsste. Aufgrund der Modellierung eines Kreisverkehrs und einer Kreuzung konnte eine bessere Effizienz des Kreisverkehrs nachgewiesen werden. Genauere Untersuchungen wurden durch einen Mangel an Zeit verhindert.

Zweites Modell (mit Matlab)

Das zweite Modell betrachtet die Autos auch als Individuen, deren Verhalten aber nur vom vorderen Auto abhängt. Nur das vorderste Auto hat eine fixe Bewegung.

Zur Untersuchung des zweiten Modells verwendeten wir Matlab.

Die Position eines Autos zur Zeit t wird durch die Funktion $x_i(t)$ gegeben und die Geschwindigkeit ist die erste Ableitung also $x_i'(t)$

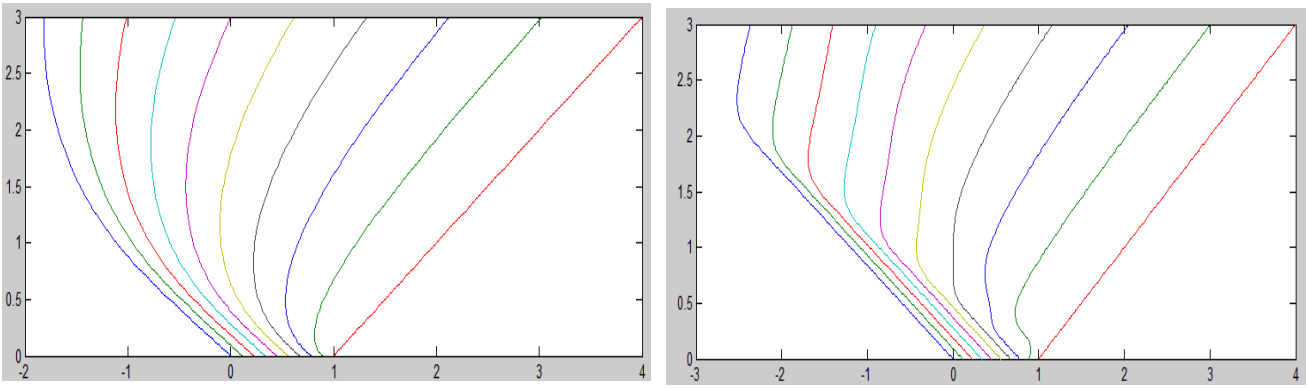
In einer Kolonne von Autos mit dem vordersten Auto X_N hängt nun die Geschwindigkeit des i-ten Autos von der Position und der Geschwindigkeit des (i+1)-ten Autos ab.

Auch hier verwendeten wir das Newton'sche Gesetz:

$$-M * x_i''(t+\tau) = A * \frac{x_i'(t) - x_{i+1}'(t)}{x_{i+1}(t) - x_i(t)} = -A(\frac{d}{dt})\ln[x_{i+1}(t) - x_i(t)]$$

Mit $\lambda = \frac{A}{M}$, wobei x_N gegeben ist und τ die Reaktionszeit darstellt

$$X'_{i(t+\tau)} = \lambda \ln[x_{(i+1)}(t) - x_i(t)] + \alpha i \quad \text{für } i = 1, \ldots, N-1$$



Die x-Achsen der beiden Grafiken zeigen die Position der Autos zum Zeitpunkt t auf der y-Achse an.
Im rechten Bild ist eine Reaktionszeit von 0.2 Sekunden zu beobachten. Dadurch entsteht eine verspätete Reaktion auf eine Änderung der Situation.
Das erkennt man an den Schwankungen der einzelnen Linien.
Während die Autos in der ersten Grafik früher auf den freien Platz vor ihnen mit einer Vorwärtsbewegung reagieren, geschieht dieses im rechten Bild verzögert.

Drittes Modell (mit Matlab)

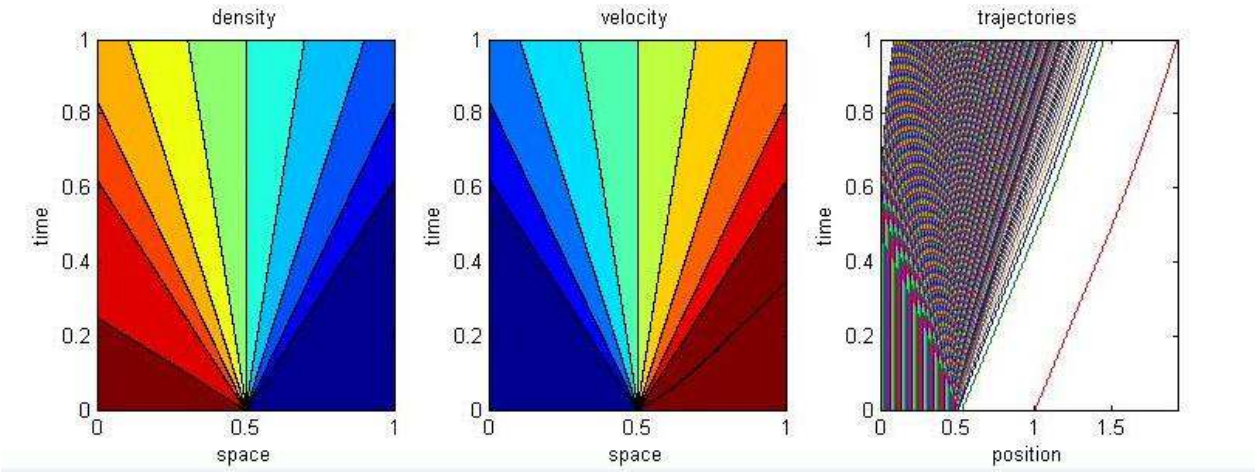
Das dritte Modell betrachtet nicht mehr jedes Auto spezifisch, sondern sieht alles als großes Ganzes, etwa vergleichbar mit einer Staubwolke.

Jede Bewegung ist abhängig von einer Dichteungleichheit im „Kontinuum“. Dadurch entsteht auch ein potentielles Geschwindigkeitsfeld. Die lokale Geschwindigkeit steigt, wenn die lokale Dichte gering ist, und sinkt, bei einer hohen lokalen Dichte.

Durch diese Dichte- und Geschwindigkeitsverteilung ergeben sich Autopositionen.

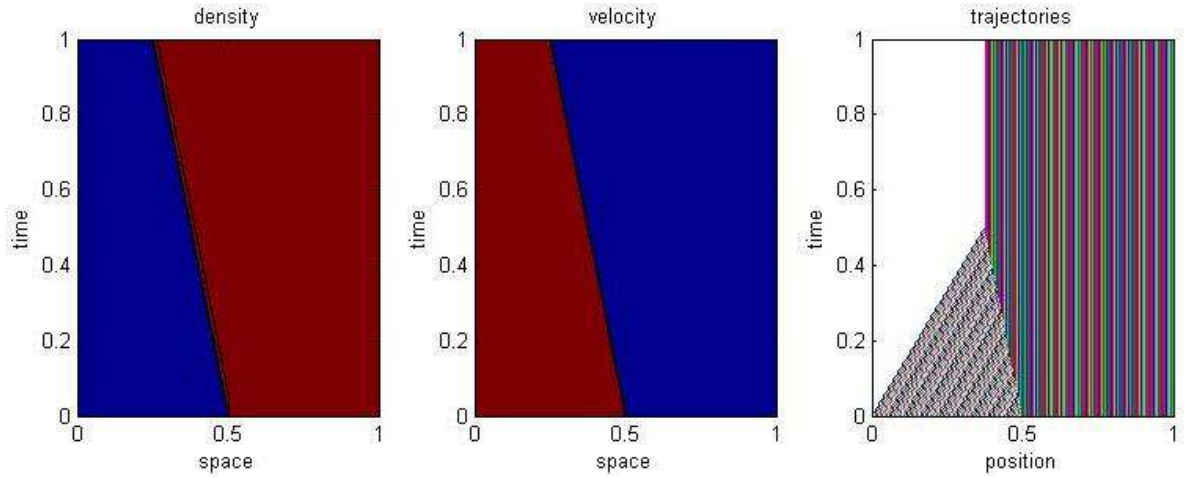
In diesem Modell wird also anders als in den beiden vorhergehenden die Autopositionen aufgrund von einem Dichtefeld und einem darauf basierendem Geschwindigkeitsfeld bestimmt.

Dieses Modell wurde in Matlab mit partieller Differentialgleichungen gelöst.



Das obere Bild zeigt die drei verschiedenen Aspekte dieses Modells.
Das linke Diagramm zeigt die lokalen Dichteverteilungen zum Zeitpunkt t.
Rot steht für hohe Dichte, dort sind also viele Autos konzentriert, und blau steht für geringe Zahl bis gar keine Autos.
In der Mitte sieht man die potenziellen lokalen Geschwindigkeiten zum Zeitpunkt t. Wenn es also in einem dunkelroten Feld ein Auto gibt, kann es sehr schnell fahren, in einer blauen Fläche können die Autos nur langsam bis gar nicht fahren.
Schlussendlich werden diese Daten im rechten Diagramm ausgewertet und in Autopositionen umgerechnet.

Zwischen den Punkten 0 und 0.5 (dunkelrot im Dichtediagramm) starten sehr viele Autos in einem kleinen Raum. Da aber ab 0.5 fast keine Autos mehr fahren, können sich die Autos in den Freiraum (blau im Dichtediagramm) bewegen. Es entsteht eine **Verdünnungswelle**.



In dieser Grafik ist zuerst eine Verdünnungswelle zu erkennen, da der mittlere Raum frei von Autos ist. Die Autos die sich in diesen Raum bewegen, werden dann aber von der großen Zahl an Autos im letzten Drittel (rot im Dichtediagramm) gestoppt. Das nennt man eine **Schockwelle**.