15. WocheE der MODELLIERUNG

mit Mathematik

(T)RICHTERLOS

well[en]done

Dokumentationsbroschüre 9.2. – 15.2.2019

WOCHE DER MODELLIERUNG MIT MATHEMATIK

JUFA LEIBNITZ, 9.2.–15.2.2019

WEITERE INFORMATIONEN:

https://imsc.uni-graz.at/modellwoche/2019/

ORGANISATOREN UND SPONSOREN

KOORDINATION

Mag. DDr. Patrick-Michel Frühmann

Mag. Vanessa Peinhart

Alexander Sekkas

Vorwort

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

Nun ist leider für viele Menschen Mathematik ein Schulfach, mit dem sie eher unangenehme Erinnerungen verbinden. Umso erstaunlicher erscheint es, dass Schülerinnen 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 Univ.-Prof. Dr. Stephen Keeling den Vorschlag, auch durch die Universität Graz so eine Woche zu veranstalten, und seiner unermüdlichen Organisationsarbeit das tatsächliche Zustandekommen. Er leitet nun bereits zum 15. Mal diese inzwischen zur Institution gewordene Veranstaltung. Ihm sei an dieser Stelle noch einmal ausdrücklich und herzlich gedankt. Besonders wichtig war in den vergangenen Jahren auch die Unterstützung durch den langjährigen Mentor der Modellierungswoche, Herrn o.Univ.-Prof. Dr. Franz Kappel, der oft auch eine eigene Gruppe mit interessanten Problemstellungen betreut hat.

Wir danken der Bildungsdirektion für Steiermark, und hier insbesondere Frau HR Mag. Christa Horn und Frau Mag. Michaela Kraker, für die Hilfe bei der Organisation und die kontinuierliche Unterstützung der Idee einer Modellierungswoche.

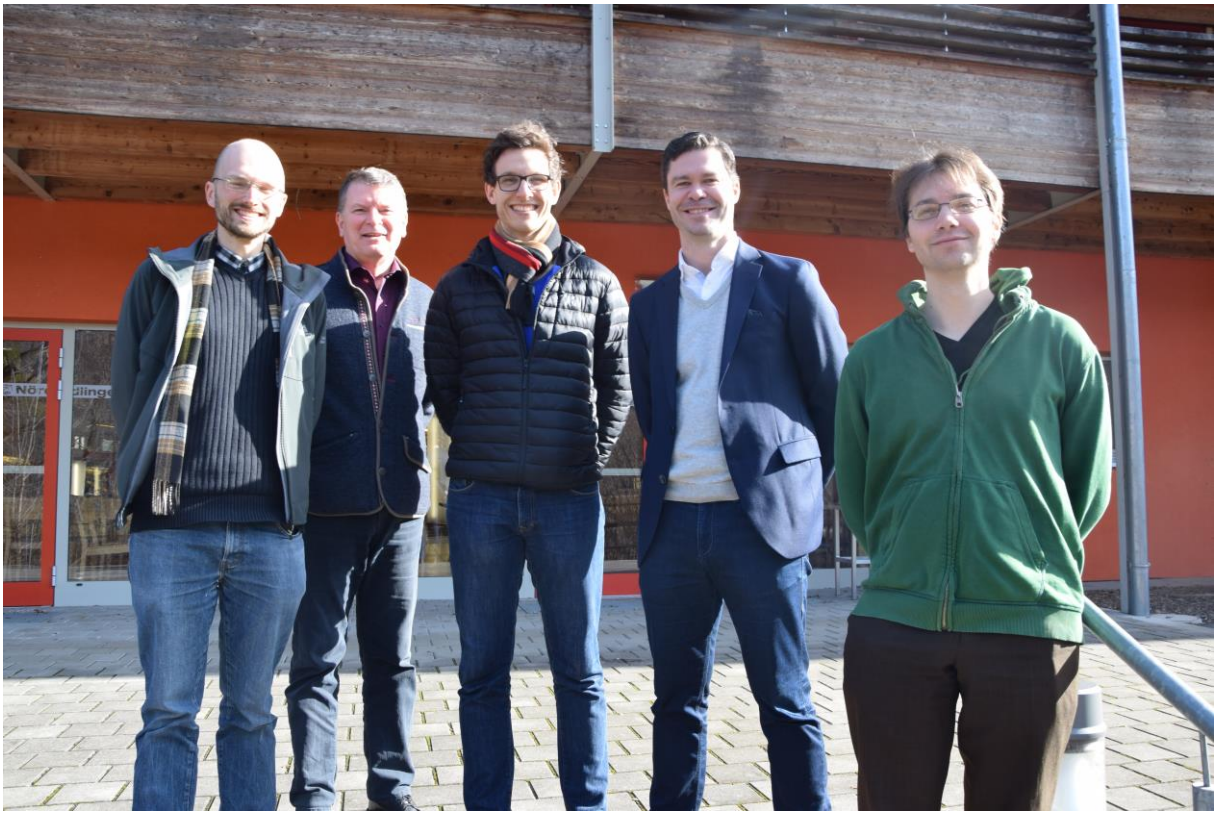
Finanzielle Unterstützung erhielten wir von der Karl-Franzens-Universität Graz durch Rektor Univ.-Prof. Dr. Martin Polaschek und Dekan Prof. Dr. Christof Gattringer, vom regionalen Fachdidaktikzentrum für Mathematik und von Comfortplan.

Ohne den idealistischen, unentgeltlichen und engagierten Einsatz der direkten ProjektbetreuerInnen Dr. Ana Garcia Elsener, Clemens Schiffer, BSc MSc, Florian Thaler, BSc, Dr. Robert Beinert, BSc MSc – Institut für Mathematik und Wissenschaftliches Rechnen – hätte diese Modellierungswoche nicht stattfinden können.

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

Leibnitz, am 15. Februar 2019

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



Projekt Biologie
Schwarmintelligenz

Elodie Ahn, Kristian Hatz, Jakob Hernler, Felix Udier,
Florian Wedenig

February 2019

betreut von
Clemens Schiffer

Inhaltsverzeichnis

1 Einleitung

Unser Projekt beschäftigt sich mit dem Phänomen der Schwarmintelligenz. Dieses beschreibt das Schwarm-Verhalten von Tieren, wie etwa Termiten, und besagt, dass Individuen eine kollektive Intelligenz besitzen, aber nicht selbst denken. Durch diese entwickeln sie als Gruppe ein komplexes Verhalten und Sozialleben. Besonders dabei ist, dass es keine zentrale Führung gibt, die die jeweiligen Aufgaben vorgibt und verteilt, sondern, dass jedes Tier, nach einem einfachen, auf seinen Instinkten basierendes, System handelt. Konkret behandeln wir dieses Verhalten am Beispiel von Termiten. Die Kommunikation zwischen den Tieren funktioniert mittels Pheromonen (Duftstoffe). Mit Hilfe dieser Pheromone werden Futterwege, oder Strecken zu Baumaterial markiert. Neben den Pheromonen werden auch Faktoren, wie z.B. Wind und Feuchtigkeit beim Errichten des Baus berücksichtigt. Als Inspiration haben wir das Paper [2] zu lesen bekommen Das Diffusionsgleicungs-Modell haben wir von [1].

2 Der Erste Schritt: Testphase der Pheromonablagerungen

Der erste Schritt war es, ein funktionierendes Kommunikationssystem unter den Termiten aufzubauen. Termiten geben Duftstoffe ab, sogenannte Pheromone.

Auf diese reagieren andere Termiten, indem sie sich dorthin bewegen oder sich entfernen. In unserer Modellierung wollten wir zu Beginn Termiten erschaffen, die sich von Pheromonen angezogen fühlen. Dazu haben wir als erstes einen neuen Abschnitt der Karte namens Map.ph (für Pheromone), definiert. Die von den Termiten abgegebenen Pheromone werden in dieser Instanz von Map gespeichert. Sie sollen sich dort gleichmäßig verteilen. Das erreicht man mit einer sogenannten Faltung im Bereich von 1 (die Termite gibt in dem Moment den Duftstoff ab) und 0.

Um die Faltung zu testen, programmierten wir zuerst eine Testfunktion. Die Faltung wird als mit Nullen gefüllte 7x7 Matrix erstellt. Sie wird konkret im Bereich von 2 bis 6 entstehen. Der Befehl `mesh (PN)` gibt alle Kombinationen von x und y Werten in einem gewissen Bereich, dort kann dann ein Wert für den entsprechen Punkt (x, y) Festgelegt werden. Diese schaut dann dreidimensional dargestellt so aus: `function Map = pheromone_diff(Map)`

```
1  P=zeros(7); %Fläche der Phermonenspur
2  Map.ph = min(Map.ph, 1);
3  P(2:6,2:6) = 4;
4
5  s = sum(sum(P));
6  F = 0.8;
7  PN=F*P/s;
8  conv2(Map.ph,PN,'same');
9  Map.ph = conv2(Map.ph,PN,'same');
```

Durch eine an den Code angehängte Schleife sieht man, wie sich die Pheromone bilden, und dann verteilen. Auf dem Bild haben sich die Pheromone schon so stark verteilt, dass der höchste Wert schon bei 0.16 (statt 1) liegt.

Die entsprechende Zeitschleife schaut dann folgendermaßen aus:

```
10 for n=1:50  %50x wiederholen
11     ph = conv2(ph,PN,'same'); %Ausbreitung
12     imagesc(ph)
13     colorbar;
14     pause(0.5)
15 end
```

Wir wollten aber nicht, dass ewig Pheromon-Reste existieren. Sonst würde es zu einer Reizüberlagerung bei den Termiten kommen. Damit dies nicht passiert, haben wir bei der Summe ‘s‘ einen Ausbreitungs- und Zerfall-Faktor eingefügt. Je höher der Faktor, desto aggressiver breiten sich die Duftstoffe aus und desto schneller verschwinden sie wieder.

```
6      s = sum(sum(P))*5; %Ausbreitungs und Zerfallfaktor
           100
```

Das sieht dann graphisch so aus:

Um einen Überblick zu schaffen, findet sich hier der gesamte Test-Code für die Pheromon-Ablagerungen.

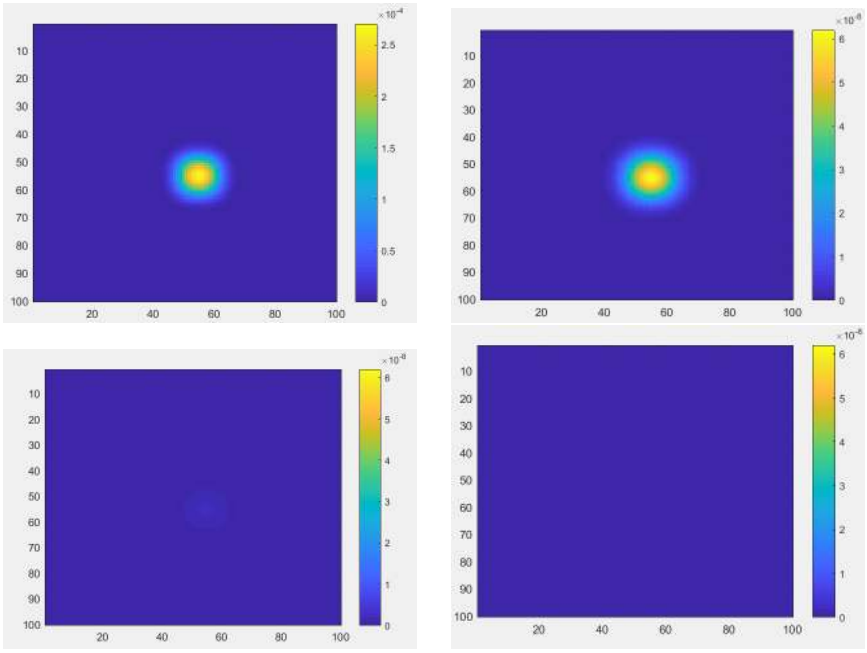


Abbildung 1: Die Pheromone entstehen und dehnen sich aus. Gleichzeitig werden ihre Werte kleiner, dann verschwinden sie wieder und die komplette Map besteht wieder nur aus Nullen.

```
P=zeros(7); %7*7 zeros
ph = zeros( 100,100); %Testfeld
ph(50:60,50:60)=1; %Position des Zentrums (zwischen
    50 und 60 in der Breite, same in der Länge)

P(2:6,2:6) = 4; %der Raum in dem sich die Faltung
    befindet (zwischen 2&6 in der Länge...)
s = sum(sum(P))*5; %Ausbreitungs und Zerfallfaktor
    100
PN=P/s;
mesh(PN)
conv2(ph,PN,'same')
for n=1:50  %50x wiederholen
    ph = conv2(ph,PN,'same'); %Ausbreitung
    imagesc(ph)
    colorbar;
    pause(0.5)
end
```

3 Der zweite Schritt: die Einarbeitung in die ‚walk ‘Funktion

Nun war es an der Zeit, die Funktion für die Geh-Funktionen umzuschreiben und zuzuschneiden. Die Karte Map.ph ist in der main mit allen anderen Karten definiert:

```
h = 100; %Höhe
b = 100; %Breite
Map.ter = zeros(h, b); %Termitenkarte

Map.ph = zeros(h, b); %Pheromonenkarte

Forbidden = zeros(h, b);
Forbidden([1,h], :) = 2; %Rand von der Hauptkarte
Forbidden(:, [1,b]) = 2;
```

```
function Map = pheromone_diff(Map)
    P=zeros(7); %Fläche der Phermonenspur

    P(2:6,2:6) = 4; %der Raum in dem sich die Faltung
        befindet (zwischen 2 und 6 in der Länge...)
    s = sum(sum(P))*5; %Ausbreitungs und Zerfallfaktor 5
    PN=P/s;
    mesh(PN)
    conv2(Map.ph,PN,'same')

    for n=1:50 %50x wiederholen
        Map.ph = conv2(Map.ph,PN,'same'); %Ausbreitung
    end
end
```

Pheromone sollen allerdings die Termiten leiten. Dafür haben wir die ‚walk ‘Funktion etwas modifiziert. Die alte ‚walk ‘schickte die Termiten nach einem Zufallssystem in eine der vier Himmelsrichtungen. Dabei hatte jede Richtung die gleiche Wahrscheinlichkeit. Nun sollen die Termiten dahin gehen, wo es Pheromone gibt. Dafür mussten wir in der ‚walk ‘erst einige Sachen neu definieren.

```
    p1 = Map.ph (A.i,A.j-1);
    p2 = Map.ph (A.i-1,A.j);
    p3 = Map.ph (A.i,A.j+1);
    p4 = Map.ph (A.i+1,A.j);

    p_sum = (p1+p2+p3+p4);

    p1N = p1/p_sum;             %Wichtigkeit der Pheromone
        runter stellen
```

```
11      p2N = p2/p_sum;             %Wahrscheinlichkeit muss
            zwischen 0 und 1 sein!
12      p3N = p3/p_sum;
13      p4N = p4/p_sum;
```

Die Zeilen 3-6 geben hierbei die Wahrscheinlichkeit an, dass die Termite in die jeweilige Richtung wandert. Die Wahrscheinlichkeit ist abhängig davon, wie viele Pheromone sich auf den an die Termite anschließenden Feldern befindet. Dabei ist die Wahrscheinlichkeit, dass die Termite in Richtung der meisten Pheromone geht, die größte. In Zeile 8 werden als Zwischenschritt die Summe aller Wahrscheinlichkeiten zusammengerechnet, um dann in den Zeilen 10-13 dem Zufall noch etwas Platz zu geben. Zudem wird kontrolliert, dass die Summe der Wahrscheinlichkeiten nicht über 1 ist. Sollten keine Pheromone im Umkreis sein, ist die Wahrscheinlichkeit für jede Richtung 0.25.

```
15      if p_sum == 0 %wenn Wahrscheinlichkei_sum 0 ist, dann
            ist die Wahrscheinlichkeit überall gleich groß
16          p1N = 0.25;
17          p2N = 0.25;
18          p3N = 0.25;
19          p4N = 0.25;
20      end
```

Um dann die Richtung zu bestimmen, haben wir eine Zeile geschrieben, die in Abhängigkeit der Wahrscheinlichkeiten die Richtung bestimmt.

```
22    random = randsample([1,2,3,4],1,true,[p1N p2N p3N p4N])
          ; %Zufallsgenerator mit
```

Hierbei sind die Outputs [1, 2, 3, 4] die möglichen Richtungen und [p1N, p2N, p3N, p4N] die Wahrscheinlichkeiten. In Abhängigkeit des Outputs wird dann die dementsprechende Richtung ausgewählt:

```
25                                  %die jeweilige Richtung
26      if      random == 1
27          iN = A.i;
28          jN = A.j - 1;
29      elseif random == 2
30          iN = A.i - 1;
31          jN = A.j;
32      elseif random == 3
33          iN = A.i;
34          jN = A.j + 1;
35      elseif random == 4
36          iN = A.i + 1;
37          jN = A.j;
38      end
```

Dann werden die alten Koordinaten (A.i, A.j) verändert und in neue (iN, jN) umgewandelt. Zum Schluss werden alte und neue Koordinaten überschrieben

und die Termite bewegt sich. Zusätzlich wird in Zeile 49 eine Pheromonspur hinterlassen:

```
1      Map.ter(A.i, A.j) = Map.ter(A.i, A.j) - 1; %die
           Termite bewegt sich
2      A.i = mod(iN-1, h) + 1;
3      A.j = mod(jN-1, b) + 1;
4      Map.ter(A.i, A.j) = Map.ter(A.i, A.j) + 1;
5
6      Map.ph (A.i, A.j) = Map.ph (A.i, A.j) + 0.1;
```

Hier die gesamte ‚walk ‘- Funktion.

```
1  function [A, Map] = walk(A, Map, Forbidden, h, b)
2
3      p1 = Map.ph (A.i,A.j-1); %Wahrscheinlichkeit für eine
           Richtung
4      p2 = Map.ph (A.i-1,A.j);
5      p3 = Map.ph (A.i,A.j+1);
6      p4 = Map.ph (A.i+1,A.j);
7
8      p_sum = (p1+p2+p3+p4);
9
10     p1N = p1/p_sum;           %Wichtigkeit der Pheromone
           runter stellen
11     p2N = p2/p_sum;           %Wahrscheinlichkeit muss
           zwischen 0 und 1 sein!
12     p3N = p3/p_sum;
13     p4N = p4/p_sum;
14
15     if p_sum == 0 %wenn Wahrscheinlichkei_sum 0 ist, dann
           ist die Wahrscheinlichkeit überall gleich groß
16         p1N = 0.25;
17         p2N = 0.25;
18         p3N = 0.25;
19         p4N = 0.25;
20     end
21
22      random = randsample([1,2,3,4],1,true,[p1N p2N p3N p4N
           ]); %Zufallsgenerator mit Wahrscheinlichkeiten
23
24                               %die jeweilige Richtung
25      if      random == 1
26          iN = A.i;
27          jN = A.j - 1;
28      elseif random == 2
29          iN = A.i - 1;
```

```
30          jN = A.j;
31      elseif random == 3
32          iN = A.i;
33          jN = A.j + 1;
34      elseif random == 4
35          iN = A.i + 1;
36          jN = A.j;
37      end
38
39      if(Forbidden(iN, jN)) %wenn eine Termite an den Rand
            stößt bleibt sie stehen
40          iN = A.i;
41          jN = A.j;
42      end
43
44      Map.ter(A.i, A.j) = Map.ter(A.i, A.j) - 1; %die
            Termite bewegt sich
45      A.i = mod(iN-1, h) + 1;
46      A.j = mod(jN-1, b) + 1;
47      Map.ter(A.i, A.j) = Map.ter(A.i, A.j) + 1;
48
49      Map.ph (A.i, A.j) = Map.ph (A.i, A.j) + 0.1;
50
51  end
```

4 Einfluss von Wind und Feuchtigkeit

Bei diesem Modell wird das Bauverhalten unter dem Einfluss von Wind und Feuchtigkeit simuliert. Als Baumaterial steht nasses Holz in einem abgegrenzten Bereich zur Verfügung. Dieser Bereich musste als erstes definiert werden. Da sich das Baumaterial unter der Erde befindet, brauchte man eine Linie, die die Oberfläche darstellt. Außerdem wurde ein Rand benötigt, um unsere fiktive Welt zu begrenzen. Dieser und die Oberfläche können nicht von den Termiten überschritten werden. An der Oberfläche gibt es einen offenen Bereich, wo die Termiten hinausklettern können. Der Rahmen beschreibt den nicht betretbaren Rand rund die Map. Die Oberfläche ist eine Linie auf 2/3 der Höhe. Diese Linie reicht vom linken Rand bis zu 2/5 der Breite und von 3/5 der Breite bis zum rechten Rand. In der Mitte wird 1/5 der Breite als Ausgang für die Termiten freigelassen.

```
1  map.forbidden = zeros(h, b);      %Rand
2  map.forbidden([1,h], :) = 1;
3  map.forbidden(:, [1,b]) = 1;
4
5  i0 = floor(2/3*h);
```

```
6
7 map.forbidden(i0,1:(2/5*b)) = 1;    %Grenze links
8 map.forbidden(i0,(3/5*b):end) = 1; %Grenze rechts
```

Nachdem der komplette Rahmen definiert war, wurde der untere Bereich mit Baumaterial gefüllt. Dieses sollte anfangs nass sein. Der Bereich, wo Material liegt, bekam den Wert 1. Die Feuchtigkeit wurde im gleichen Bereich auf einen Anfangswert von 1 gesetzt.

```
1 map.wood((i0+1):(end-1),2:(end-1)) = 1; %wo Holz liegt
2 map.nass((i0+1):(end-1),2:(end-1)) = 1; %wo nasses Holz
    liegt
```

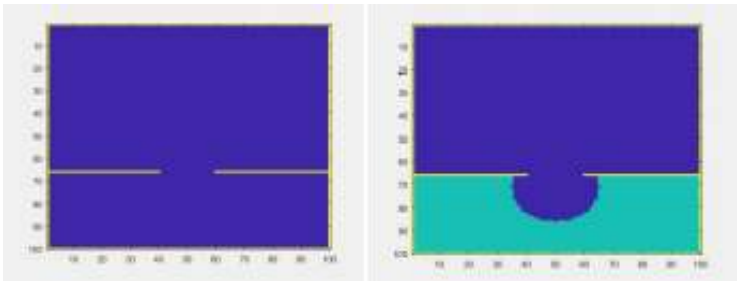


Abbildung 2: Rand und Baumaterial

Um eine Art Nest darzustellen, musste aus dem mit Baumaterial gefüllten Bereich ein Teil herausgeschnitten werden. In diesem sollte sich kein Material befinden. Ein Kreis war dafür am Besten geeignet. Für diesen brauchte man eine Gleichung und eine fixe Position. An der Position des Kreises werden die Werte auf 0 gesetzt, wodurch sich dort kein nasses Material mehr befindet.

```
1  i = i0+5; %Koordinaten des Mittelpunkts
2  j = b/2;
3  a = 1:h;      %von wo X genommen wird
4  b = 1:b;      %von wo Y genommen wird
5  r = 10;
6  [X,Y] = meshgrid(a,b);
7  cp = (X-j).^2+(Y-i).^2 <= r.^2; %Kreisgleichung
8
9  map.wood(cp) = 0;
10 map.nass(cp) = 0;
```

Nun sollten die Termiten dieses Material aufnehmen. Dafür wurden zwei Bedingungen festgelegt. Einerseits kann eine Termite nur Material aufnehmen, wenn sie nicht schon etwas trägt. Andererseits nimmt sie nur Material auf, das einen bestimmten Feuchtigkeitswert (>0,8) hat. Da Termiten beim Aufnehmen von Material häufig darin enthaltenes Wasser und Wasser aus der Umgebung trinken, wird es dort trockener. Um diesen Effekt zu simulieren, verringert sich

beim Aufnehmen die Feuchtigkeit des Materials und auch in einem kleinen Bereich rundherum. Dadurch wird Material durch mehrfaches Aufnehmen immer weniger nass, bis das Material schließlich zu trocken ist, um erneut verwendet zu werden. Um zu verhindern, dass diese Werte negativ werden, gilt 0 als Grenzwert, der komplette Trockenheit beschreibt.

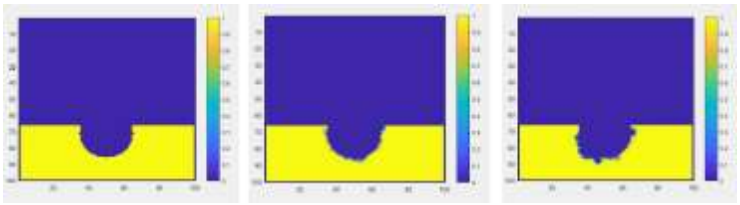


Abbildung 3: Änderung der Feuchtigkeit des Baumaterials

```
function [A, Map] = pickupNass(A, Map)
%Termiten nehmen nur Holz auf, wenn es nass ist

    surWood = Map.wood((A.i-1):(A.i+1), (A.j-1):(A.j+1));
    surNass = Map.nass((A.i-1):(A.i+1), (A.j-1):(A.j+1));

    Wood = [];

    for i=1:3
        for j=1:3
            if(surWood(i, j) == 1 && surNass(i,j) > 0.8)
                W.i = i;
                W.j = j;
                Wood = [Wood, W];
            end
        end
    end

    M = Map.nass((A.i-1):(A.i+1), (A.j-1):(A.j+1));

    if(~isempty(Wood) && A.wood == 0) %Es gibt Holz und
        trägt nicht
        r = randi([1, length(Wood)]);
        Map.wood(A.i + Wood(r).i - 2, A.j + Wood(r).j -
            2) = 0;
        A.wood = 11; % 11 == trägt etwas
        Map.nass((A.i-1):(A.i+1), (A.j-1):(A.j+1)) = max(
            Map.nass((A.i-1):(A.i+1), (A.j-1):(A.j+1))
            -0.1,0); %wird weniger nass (um-0.1)
        A.woodHum = max(max(M));
```

```
27
28      end
29
30  end
```

Zu der Feuchtigkeit kam nun Wind als zweiter Faktor, der das Bauverhalten der Termiten beeinflusst. Um Wind, der von links kommt, zu simulieren, werden in die linke Spalte Einser geschrieben. Diese werden weiter von links nach rechts in jedes Feld geschrieben, bis sie auf ein Hindernis stoßen. In diesem Fall werden Nullen geschrieben. Für Wind von rechts passiert das Gleiche von rechts nach links.

```
1   function [Map] = wind(Map)
2
3       [h,b] = size(Map.ter);
4       h0 = Map.h0;
5       windL = zeros(h,b); %Wind von links
6       windL(1:h0,1) = 1;
7       for j = 2:b
8           for i = 1:floor(2/3*h)
9               windL(i,j) = windL(i,j-1);
10              if(Map.wood(i,j)==1 || Map.forbidden(i,j)==1)
11                  windL(i,j) = 0;
12              end
13          end
14      end
15
16      windR = zeros(h, b); %Wind von rechts
17      windR(1:h0,end) = 1;
18      imagesc(windR);
19      for k = 1:(b-1)
20          j=b-k;
21          for i = 1:floor(2/3*h)
22              windR(i,j) = windR(i,j+1);
23              if(Map.wood(i,j)==1 || Map.forbidden(i,j)==1)
24                  windR(i,j) = 0;
25              end
26          end
27      end
```

Um den Wind zu testen, erstellten wir einen Prototyp von einem Bau. Dieser besteht aus zwei Linien. Außerdem sollte unter der Erde bzw. unter der Oberfläche kein Wind gehen, weswegen der Bereich des Windes durch eine Zeile auf Höhe der Oberfläche begrenzt wird.

```
1   h = 100;
2   b = 100;
3
```

```
h0 = floor(2/3*h); %Zeile, die Wind begrenzt

j0 = 1/4*b;
j1 = 3/4*b;

Bau = zeros(h,b);
Bau(floor(2/4*h):100,j0) = 1; %Grenze links
Bau(floor(2/4*h:100),j1) = 1; %Grenze rechts


[windL,windR] = wind(Bau,h0);
windGes = windL+ windR; %gesamter Wind
```

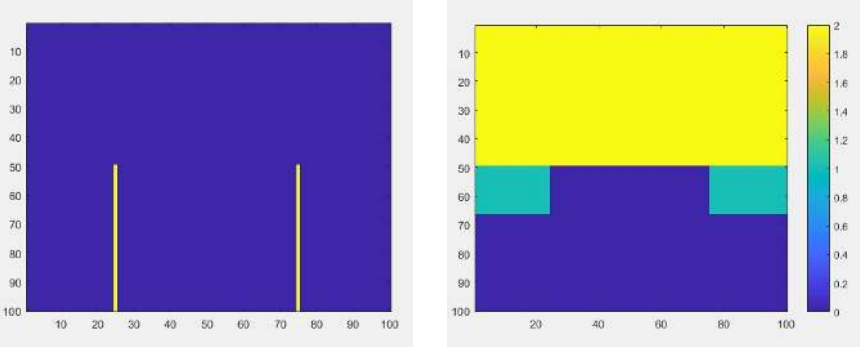


Abbildung 4: Vereinfachter Bau und Wind

Man kann sehen, dass die Werte unter der Oberfläche und zwischen den beiden Linien null sind. Hier ist es windstill. Außerhalb liegen sie entweder bei 1 oder 2. Bei einem Wert von 1 kommt der Wind nur aus einer Richtung, bei 2 kommt Wind von links und rechts und die Werte werden addiert.

Der nächste Schritt war es den Wind mit dem Bauverhalten der Termiten zu kombinieren. Diese tragen das gesammelte Baumaterial zu einem Ort, der relativ trocken und nicht windstill ist. Diese Bedingungen wurden in der Build-Funkton festgelegt.

```
i0 = A.i + i - 2;
j0 = A.j + j - 2;
cond1 = Map.nass(i0, j0) < humid   ;
cond2 = Map.wind(i0, j0) >= 1;
```

Trifft Wind auf Baumaterial so wird er, wie bei dem Test mit dem Bau, gestoppt. Die Werte werden wieder auf 0 gesetzt. So bauen die Termiten mit der Zeit weiter hinauf, da manche Stellen bereits windstill sind

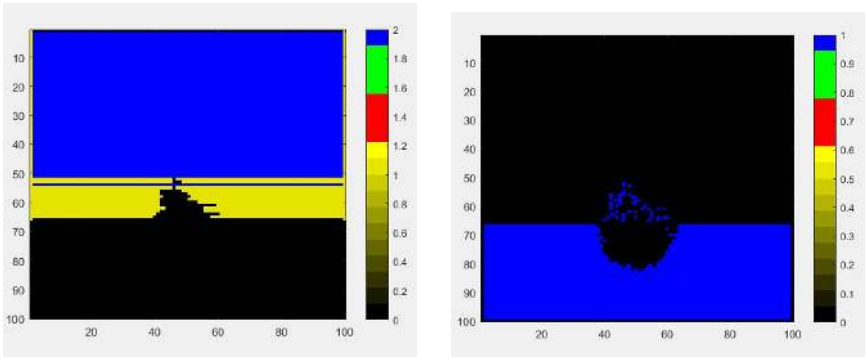


Abbildung 5: Wind(links) und Verteilung des Baumaterials(rechts)

5 Die Home Funktion

Der ursprüngliche Gedanke der Home.m Funktion war, einen Bereich zu definieren, in den die Termiten gerne nach dem Bauprozess zurückkehren. Um diesen Bereich attraktiv zu machen, benutzten wir die Matlab Funktion „Meshgrid“. Diese Funktion kann man am besten mit dem Kreuzprodukt vergleichen, wobei im Fall des Meshgrids alle Kombination der x- und y-Werte und nicht zwei Vektoren kombiniert werden. Wir stießen auf erste Probleme mit dieser Funktion, als wir bemerkten, dass die Abstände der Werte auf den Achsen der Funktion nicht gleich groß waren. Dies wurde behoben, indem wir x und y mit „linspace(-1,1,h/b)“ bezeichneten. Hierbei steht „-1,1“ für den vorgesehenen Wertebereich, in diesem Fall von -1 als tiefster und +1 als höchster Wert. Die Werte h und b stehen hier für die gewünschte Anzahl von gleich großen Werten. Hier sind b und h beide jeweils 100. Nun war es an der Zeit, zu definieren, wie unsere Funktion aussehen soll. Für diesem Fall wählten wir die Form einer Exponentialfunktion (siehe unten). Wir wählten eine Funktion mit Hochpunkt, um das sogenannte „Heimkehrpotenzial“ besser darstellen und umsetzen zu können. Wir haben es mit Pheromonen verglichen, welche die Termiten wahrnehmen und zu denen diese dann im Umkehrschluss auch gerne zurückgehen. Das nächste Problem war, dass wir die Funktion nicht nach Koordinaten, sondern nach Spalten darstellen wollten. Dies stellte für uns eine genauere Methode zur Auslegung dar. Da wir jedoch in der Ursprungsfunktion von Koordinaten und nicht von Spalten ausgingen mussten wir auch diese abändern. Der letzte Feinschliff passierte bei der Ausrichtung. So war das Zentrum der Funktion bei den Werten 50/50 in der linken oberen Ecke, jedoch sollte es genau in der Mitte liegen. Behoben haben wir es indem wir die Werte x0 und y0 so modifizierten, dass Der ursprüngliche Wert geteilt wird. Dies bewirkte, dass besagtes Zentrum nun exakt mittig liegt. Als die Funktion aufgestellt war musste sie nur noch implementiert werden. Dies geschah in der spreadHome.m.

Um die Übersicht zu verbessern, hier nochmal die ganze Funktion:

```
1 function F = Home (h,b,i,j)
```

```
2
3  x0= 1/h*2*(h/2-j);
4  y0= 1/b*2*(b/2-i);
5  e=4 ;
6  x = linspace(-1,1,h);
7  y = linspace(-1,1,b);
8
9  y=y-y0 ;
10 x=x-x0 ;
11 [X,Y] = meshgrid(x,y);
12 X.^2 + Y.^2;
13 F =-exp((-X.^2-Y.^2)*e);
14 surf(X,Y,F)
15 end
```

6 Build Funktion

Die Build Funktion war mit der Pickup Funktion, in welcher die Termiten Holz erkennen und aufheben, die erste Erweiterung zu den laufenden Termiten. Dadurch war es möglich, das Verhalten durch Änderung der Parameter und Optimierung der main.m Funktion beim Bau in die eine oder die andere Richtung zu beeinflussen. Dies stellte die erste größere Errungenschaft nach dem Erstellen der main.m Datei dar. Die Funktion enthält mehrere Parameter. Die wichtigsten sind „A.wood“, „empty“ und „Map.wood“. „A.wood“ speichert, ob Holz von der Termite getragen wird oder nicht. „empty“ wiederum lässt die Termite erkennen, ob sich um sie herum leere oder mit Holz gefüllte Felder befinden. „Map.wood“ ist die von uns benutzte Map, in welcher sich Holz befindet.

```
1  function [ A, Map ] = build( A, Map)
2      sur = Map.nextToWood((A.i-1):(A.i+1), (A.j-1):(A.j+1)
           );
3
4      if(A.wood == 11)
5          empty = 0;y
6          for i=1:3
7              for j=1:3
8                  if(Map.nextToWood(A.i + i - 2, A.j + j -
                       2) > empty && i ~= 2 && j ~= 2 && Map.
                       forbidden(A.i + i - 2, A.j + j - 2) ~=
                        1)
9                      empty = Map.nextToWood(A.i + i - 2, A
                           .j + j - 2);
10                     E.i = A.i + i - 2;
11                     E.j = A.j + j - 2;
12                 end
```

```
13                  end
14              end
15              if(empty ~= 0)
16                  Map.wood(E.i  , E.j) = 1;
17                  A.wood = 10;
18              end
19          elseif(0 < A.wood && A.wood <= 10)
20
21              A.wood = A.wood - 1;
22          end;
23
24  end
```

7 Mauerbau

Das Ziel bei unserem Mauerbau - Experiment war, dass die Termiten mit dem Holz der Umgebung eine kreisförmige Mauer um einen vorgegebenen Mittelpunkt errichten. Nach dem Prinzip der Multiagentensimulation sollten die Termiten dabei aber nicht nach einem vorgegebenen Bauplan, sondern nach den Informationen aus ihrer direkten Umgebung arbeiten. Daher entschieden wir uns am Beginn dazu, einen neuen Pheromontyp einzuführen. Dieses "Home Pheromone" wird am Anfang auf der gesamten Map verteilt und kann danach nicht verändert werden. Die Termiten bauen nur auf Felder, deren "Home Pheromone" - Level in einem bestimmten Bereich liegt.

Natürlich stellte sich bald die Frage, wie diese Pheromone verteilt werden können. Damit der Prozess skalierbar ist, wird dazu eine mathematische Funktion benutzt.

Dabei nehmen die Pheromone in einem bestimmten Bereich einen Wert über null ein. Dort bringen die Termiten ihr Baumaterial hin. Jedoch gibt es dabei zwei große Probleme. Einerseits bauen die Termiten statt einer zwei Mauern, die sich am inneren bzw. äußeren Rand des möglichen Baubereichs befinden. Die Erklärung dafür liegt in der Programmierung der Termiten. Sie können sich durch das Baumaterial nicht bewegen und bauen, sobald alle Bedingungen dazu erfüllt sind. Zudem bewegen sie sich zufällig.

Andererseits sehen die Mauern auch nicht sehr organisch aus. Die Grenzen sind klar definiert, außerdem entsprechen sie ziemlich genau einem perfekten Kreis.

7.1 Die Exponentialfunktion

Zu diesem Zweck entwickelten wir eine $e^(x^2)$ - Funktion. Das dadurch erzeugt Pheromonlevel nimmt von der Mitte nach außen ab und eignet sich damit zur Navigation der Termiten. Damit die Termiten eine ringförmige Mauer bauen, musste jedoch der Code zum Bauen modifiziert werden. Bei der Exponentialfunktion sollten die Termiten ja nicht mehr bei allen positiven Werten bauen.

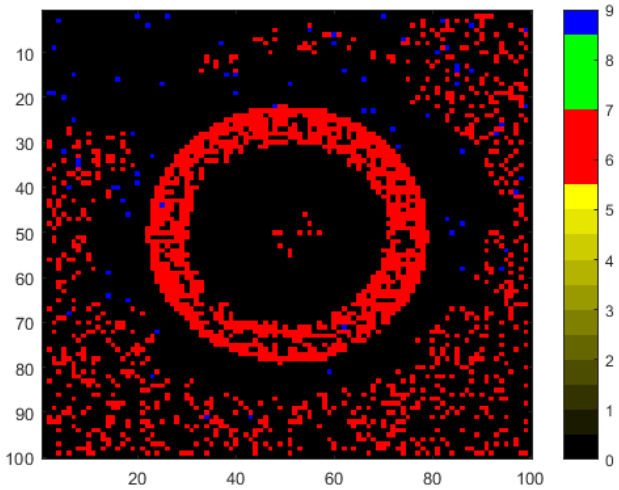


Abbildung 6: Die klar definierte Mauer. Da die innere und äußere Mauer nah aneinander gerückt wurden, sind sie nur schwer unterscheidbar.

Der Bereich, in dem Bauen erlaubt ist, ist nun nicht mehr nur nach unten, sondern auch nach oben begrenzt ist. Sonst würden die Termiten auch in der Mitte bauen, die ja leer bleiben sollte. Daher erstellten wir die Funktion $e^(-(x-x0)^2)$. Diese Funktion wird von den Termiten auf jedes Feld angewendet. Dabei wird ermittelt, ob auf das entsprechende Feld gebaut werden darf. x0 ist dabei der bevorzugte Wert.

BILD EXP F(X)

Das Ergebnis lässt sich zwar besser für andere Skripte verwenden und ist etwas realistischer, aber es sind noch immer zwei kreisähnliche Mauern sichtbar.

```
1  function [ A, Map ] = buildWall( A, Map)
2
3      wallBuildingCondition = 0.9;
4      homePh0 = 0.4;
5
6      if(A.wood == 11) %Es gibt Holz und trägt
7          empty = 0;
8          for i=1:3
9              for j=1:3
10                 i0 = A.i + i - 2;        %global, i =
                       lokal
11                 j0 = A.j + j - 2;        %global, j =
                       lokal
12                 cond1 = Map.nextToWood(i0 , j0) > empty;
```

```
13                      nichtrand = Map.forbidden(i0  , j0) ~= 1;
14
15  %                     wallProbability = rand * exp(-(Map.
        homePh(i0,j0) - homePh0)^2);
16                      wallProbability = exp(-(Map.homePh(i0,j0)
                            - homePh0)^2);
17
18                      if( cond1 && i ~= 2 && j ~= 2 &&
                            nichtrand && wallProbability >
                            wallBuildingCondition)
19                          empty = Map.nextToWood(i0 , j0);
20                          E.i = i0;
21                          E.j = j0;
22                      end
23                  end
24              end
25              if(empty ~= 0)
26                  Map.wood(E.i  , E.j) = 1;
27                  A.wood = 10;
28              end
29          elseif(0 < A.wood && A.wood <= 10)
30              A.wood = A.wood - 1;
31          end
32
33  end
```

7.2 Zufall

Um die Struktur der Mauer organischer zu machen, fügten wir einen Zufallsfaktor hinzu. Um dessen Funktion zu verstehen, muss man jedoch tiefer auf die buildWall - Funktion eingehen. Sie ist eine um eine Bedingung erweiterte build-Funktion. Während build auf alle Felder mit Holz in der Nähe baut, wird bei buildWall auch der "Home - Pheromone Wert miteibezogen. Aus diesem wird dann die wallProbability durch $e^(- (x - x0)^2)$ berechnet. Liegt diese über einem bestimmten Wert, darf die Termite auf dieses Feld bauen. Der Zufallsfaktor modifiziert die wallProbabililty. Dadurch wird die Grenze zwischen bebaubaren und nicht bebaubaren Feldern verwischt.

Nach einigem finetuning kann sich das Ergebnis durchaus sehen lassen.

8 Futtersuche

```
1  clear all
2
3  h = 100; %Höhe
```

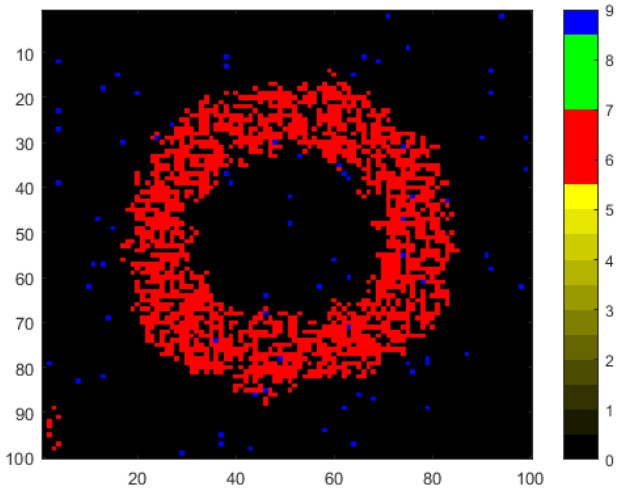


Abbildung 7: Eine organischere Mauer.

```
b = 100; %Breite
    Map.h = h;
    Map.b = b;

Map.ter = zeros(h, b); %Hauptkarte
Map.ph = Map.ter; %Pheromonkarte

%
Map.homePh = zeros(h,b);
Map.homePh = spreadHomePh(h,b,h/2,b/2);
%

Map.forbidden = zeros(h, b); %Rand und Begrenzungen
Map.forbidden([1,h], :) = 1;
Map.forbidden(:, [1,b]) = 1;
    i0 = floor(2/3*h);
    Map.forbidden(i0,1:(2/5*b)) = 2; %Grenze links
    Map.forbidden(i0,(3/5*b):end) = 2; %Grenze rechts

Map = woodGenerator(Map);
Map = woodScanner(Map);

run ColorData;
colormap(mymap);
figure (1)
```

```
29
30 explorers = 0; %Anzahl Endeckertermiten
31 gatherers = 75;  %Anzahl Transporttermiten
32
33 Explorer = [];       %1. Entdeckertermite
34 Gatherer = [];       %1. Trägertermite
35
36 for i=1:explorers
37     E.i = 3;
38     E.j = i + 1;
39     E.wood = 0;
40     Map.ter(3,i+1)=1;
41     E.b = 4;
42     Explorer = [Explorer, E];
43 end
44
45 for i=1:gatherers
46     G.i = 2;
47     G.j = i + 2;
48     G.wood = 0;
49     Map.ter(2,i+2)=1;
50     G.b = 4;
51     Gatherer = [Gatherer, G];
52 end
53 %tic
54 for k=1:100000
55
56     Map = woodScanner(Map);
57
58     for i=1:explorers
59         [Explorer(i), Map] = walke(Explorer(i), Map); %
               Termite bewegt sich
60         [Explorer(i), Map] = buildWall(Explorer(i), Map);
               %Termite baut Holz
61         [Explorer(i), Map] = pickup(Explorer(i), Map); %
               Termite nimmt Holz
62     end
63
64     for i=1:gatherers
65         r = rand(gatherers,1);
66         [Gatherer(i), Map] = walkg(Gatherer(i), Map);
67         [Gatherer(i), Map] = buildWall(Gatherer(i), Map);
68         tic
69         [Gatherer(i), Map] = pickup(Gatherer(i), Map);
70         toc
71     end
```

```
72
73      Map = pheromone_diff(Map);
74
75      if(mod(k,1000)==0)
76          %toc
77          %tic
78          k
79          imagesc(dispMap(Map,1))
80          colorbar;
81          drawnow
82      end
83
84
85      %figure(1)
86      %imagesc(Map.ter + Map.wood);
87      %figure(2)
88      %imagesc(Map.ph);
89
90      %pause(0.01);
91  end
92
93  figure(1)
94  imagesc(Map.ter + Map.wood);
95  colorbar;
96  figure(2)
97  imagesc(Map.ph);
98  colorbar;
```

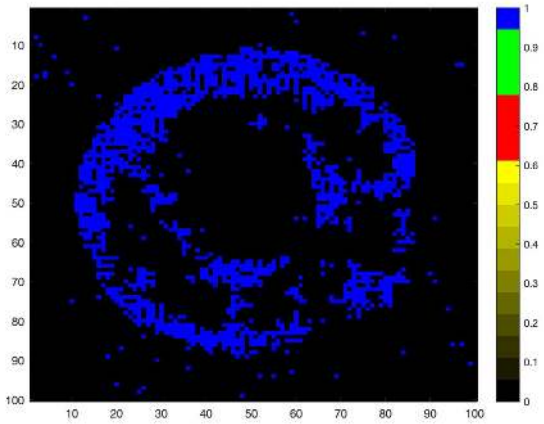


Abbildung 8: Eine Doppelseitige Mauer.

Nach den bisherigen Versuchen fehlte uns allerdings noch ein häufiges Experiment zur Schwarmintelligenz: ein sogenannter Ameisenalgorithmus. Dabei suchen Agenten (Ämeisen") nach Futter. Sobald sie dieses gefunden haben, setzen sie Pheromone frei. Diese locken andere Agenten an. Dadurch wird eine Futterquelle sehr schnell abgebaut, sobald sie entdeckt wurde.

Das Ergebnis war nach einiger Arbeit sehr anschaulich. Die blauen Explorer - Termiten bewegen sich zufällig vom Zentrum weg. Stoßen sie auf Baumaterial, tragen sie dieses ins Zentrum und geben Pheromone ab. Die Pheromone locken grüne Gatherer - Termiten an, welche dann mehr und mehr Baumaterial ins Zentrum tragen.

Bei dieser Simulation wird das Baumaterial allerdings nicht gesetzt, sondern als Futter behandelt. Deshalb werden in der Mitte keine Blöcke gesetzt.

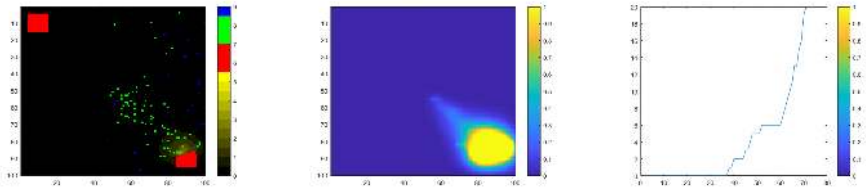


Abbildung 9: Ameisen finden das Futter

9 Diffusionsgleichungen

9.1 Wärmeleitungsgleichung

Die Verteilung von Wärme in einem Raum wird durch eine Funktion $u(t, x)$ zur Zeit t am Ort x beschrieben. Die Wärme am Anfang im $t = 0$ ist durch eine fixe Verteilung zum Start $u(0, x) = u_0(x)$ beschrieben. Die Verteilung entwickelt sich nach der Differenzialgleichung:

$$\partial_t u = \partial_{xx} u \tag{1}$$

Die Wärme fließt von heißen zu kälteren Orten. Am Rand kann die Wärme nicht hinaus. Die Funktion u wird diskretisiert, d.h. Werte gibt es nur an einzelnen Punkten. Wir halten uns an den Programmcode von Stephen Keeling und studieren ihn. Die Raum-Ableitungen ∂_x werden mit finiten Differenzen, so wie im Differenzenquotienten, angenähert. Die ersten Ableitungen erfolgen durch Vorwärtsdifferenzen, also $\partial f_i = 1/h(f_{i+1} - f_i)$, wobei h der Abstand zwischen zwei Punkten im Raum ist. Die zweiten Ableitungen werden durch Rückwärtsdifferenzen der ersten Ableitungen angenähert. So entsteht die Matrix Dx, sodass $\partial_{xx} \approx -Dx^T Dx$ wobei die Transponierte Dx^T die Rückwärtsdifferenzen enthält, nur mit einem Minus. Wir approximieren die Lösung von $\partial_t u = -Lu + f$. Ist die Lösung u^t zum Zeitpunkt t bekannt berechnen wir mittels impliziten

Eulerschritt die Lösung u^{t+1} zum nächsten Zeitpunkt:

$$\frac{u^{t+1} - u^t}{dt} = -Lu^{t+1} + f \tag{2}$$
$$u^{t+1} - u^t = -dtLu^{t+1} + dtf \tag{3}$$
$$u^{t+1} + dtLu^{t+1} = u^t + dtf \tag{4}$$
$$(I + dtL)u^{t+1} = u^t + dtf \tag{5}$$
$$u^{t+1} = (I + dtL)^{-1}(u^t + dtf) \tag{6}$$
$$\tag{7}$$

9.2 Diffusionsmodell für Termiten

Ähnlich wie die Diffusion modellieren wir auch die Termiten als eine Masse, genau wie die Pheromone und das Baumaterial:

$$\partial_t H = k_1 P - k_2 H + D_h \partial_{xx} H \tag{8}$$
$$\partial_t C = \Phi - k_3 C + D_c \partial_{xx} C - \gamma \partial_x (C \partial_x H) \tag{9}$$
$$\partial_t P = k_3 C - k_4 P \tag{10}$$

Dann ist die gesuchte Funktion u die Komponenten $u = [H, C, P]$. Die Konstanten f, die die Quellen der Komponenten angeben, sind dann $f = [0; \Phi; 0]$, d.h. es gibt einen konstanten Strom an Termiten der Hinzukommt.

Die zeitliche Entwicklung der Funktionen wird jeweils durch mehrere Parameter beeinflusst, indem sie die Wechselwirkung untereinander festlegen. So hängt die Verteilung H der Pheromone direkt von deren Abgabe k_1 durch das Baumaterial P ab, miteinberechnet wird die Zerstreung durch den Diffusionkoeffizienten D_h und der Zerfall der Pheromonspuren k_2. Die Verteilung C der Termiten, die Baumaterial transportieren, wird ebenfalls durch einen Diffusionskoeffizenten D_c sowie durch die Affinität der Termiten gegenüber den Pheromonen bestimmt und hat sowohl einen konstanten Zuwachs von Φ pro Zeitpunkt wie auch einen Rückgang, verursacht durch die Abgabe k_3 des Baumaterials. Die Verteilung des Baumaterials P wächst einerseits durch die dessen Abgabe k_3 von den Termiten, sinkt jedoch durch den eigenen Zerfall k_4. In unserem müssen wir die Matrix L berechnen:

$$L = \begin{bmatrix} L_{HH} & L_{HC} & L_{HP} \\ L_{CH} & L_{CC} & L_{CP} \\ L_{PH} & L_{PC} & L_{PP} \end{bmatrix}$$

Wir erhalten für die einzelnen Teilmatrizen:

$$L = \begin{bmatrix} D_h Dx^T * Dx + k_2 I & 0 & -k1I \\ -\gamma Dx^T * diag(C) * Dx & D_h Dx^T * Dx & 0 \\ 0 & -k_3 I & k_4 I \end{bmatrix}$$

Dann lösen wir das diskrete Problem mit dem impliziten Eulerverfahren so wie die Wärmeleitungsgleichung.

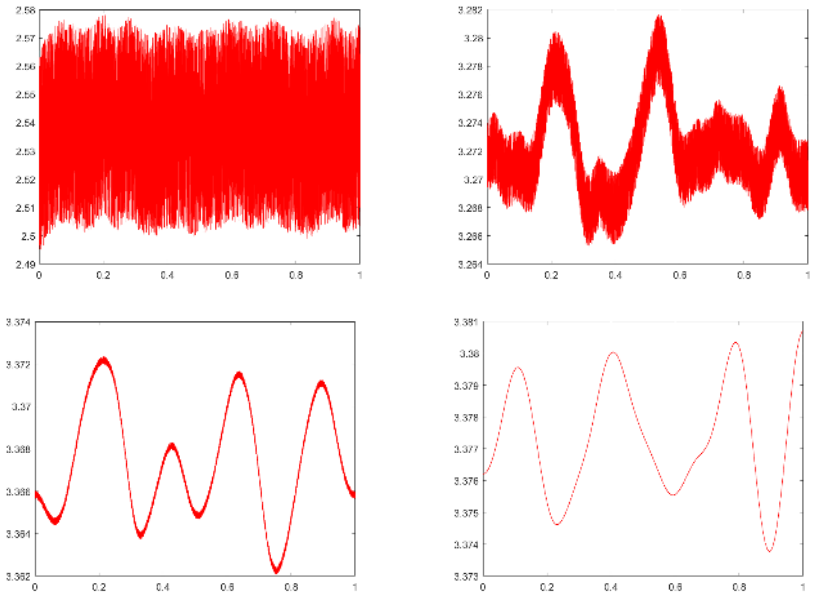


Abbildung 10: Lösung der gekoppelten Diffusionsgleichungen zu vier Zeitpunkten

Literatur

[1] Eric Bonabeau, Guy Theraulaz, J Deneubourg, Nigel R Franks, Oliver Rafelsberger, J Joly, and Stephane Blanco. A model for the emergence of pillars, walls and royal chambers in termite nests. *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, 353(1375):1561–1576, 1998.

[2] J Scott Turner. Termites as models of swarm cognition. *Swarm Intelligence*, 5(1):19–43, 2011.

Woche der Modellierung

Projekt: Automatisiertes Fahren

STRATEGIEN DER AUTOMATISIERTEN FAHRZEUGSTEUERUNG

Markus Frohmann, Alexander Heidinger, Mike Jesernik, Alexandra Kahr, Laura List, Leonie Namesnig, Leonie Rubin, Bennet Wichmann

Betreuer: Florian Thaler, BSc

Contents

1 Einleitung

Heutzutage verfügen immer mehr Autos über Assistenzsysteme. Automatisiertes Fahren wird die Zukunft revolutionieren. Zahlreiche Automobilkonzerne widmen sich diesem Thema und betreiben intensive Forschung, um diese Systeme genauer zu untersuchen und weiterzuentwickeln. Doch worauf beruhen diese Konzepte eigentlich?
Mit dieser Frage haben wir uns im Sinne der diesjährigen mathematischen Modellierungswoche in Leibnitz genauer auseinandergesetzt. Unser Ziel war es, herauszufinden, wie sich ein autonom fahrendes Auto verhält, wenn es auf ein Hindernis trifft. Hierzu haben wir versucht, das Problem mit MATLAB zu modellieren. Für unsere Simulationen haben wir uns am „Artificial Potential Field Algorithm“ bedient. Diese Methode beruht auf künstlich erzeugten Potentialfeldern und dem zweiten Newton'schen Gesetz.
Wir haben Objekte simuliert, die sich von einem Startpunkt zum Ziel bewegen und haben anschließend verschiedene Hindernisse eingebaut, um die daraus resultierende Bewegung des Objekts zu lenken. Des Weiteren haben wir uns auch mit der Bewegung des Objekts bei einem dynamischen Ziel auseinandergesetzt.

2 Theorie

Bevor wir beginnen konnten, unsere Codes zu erstellen, war es essenziell, sich mit der Materie des autonomen Fahrens auseinanderzusetzen. Zuerst war es wichtig zu wissen, wie wir unsere Problemstellung am besten modellieren. Die Basis unseres Modells bildet ein sogenanntes ,,Potential Field“. Diese kann man sich als eine Gebirgslandschaft vorstellen, worauf sich eine Kugel befindet, die sich aufgrund der Gravitationskraft talwärts bewegt und durch Hindernisse in Form von ,,Bergen“ lokal von ihrer Bahn abgelenkt wird. Unser Modell erzeugt global eine künstliche Kraft, die stets entlang der Richtung des steilsten Abstiegs wirkt. Diese Kraft zeigt in die Richtung des tiefsten Tals in unserem plastischen Modell. Im Gegensatz dazu wurden Potentiale erzeugt, die lokal eine abstoßende Wirkung auf das Objekt haben, um Hindernisse in der Realität zu symbolisieren.

3 Potentialfelder

3.1 Attraktives Potential

Die Funktionsformel für ein attraktives Potential, beziehungsweise für ein,,Tal“ in unserer Analogie mit einer Gebirgslandschaft, ergibt sich wie folgt:

$$U_{attr}(x, y) = \alpha \cdot ((x - x_0)^2 + (y - y_0)^2)^{n/2}$$

α, n = Parameter zur Anpassung des Potentials
x,y = Momentane x- und y-Koordinaten des Objekts
x_0, y_0 = Koordinaten des Zielpunkts

Wobei dieses Potential eine globale Wirkung hat.

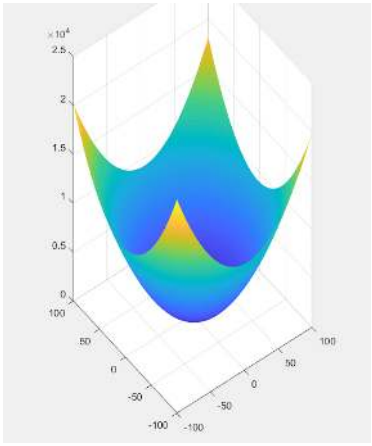


Figure 1: Ein Beispiel für ein sogenanntes ,,Tal“

3.2 Abstoßendes Potential

Die Funktionsformel für ein abstoßendes Potential, beziehungsweise für einen ,,Berg“ in unserer Analogie mit einer Gebirgslandschaft, ergibt sich wie folgt:

$$U_{rep}(x,y) = \beta \cdot \left[\frac{1}{w + ((x-a)^2 + (y-b)^2)^{\frac{q}{2}}} - \frac{1}{w + \rho_0^q} \right]$$

β, w, q	= Parameter zur Anpassung des Potentials
ρ	= Bestimmung des Wirkungsbereiches der Abstoßreaktion
a,b	= Koordinaten der Spitze des Potentials

Im Gegensatz zum attraktiven Potential wirkt diese Funktion nur lokal in gewissen Bereichen.

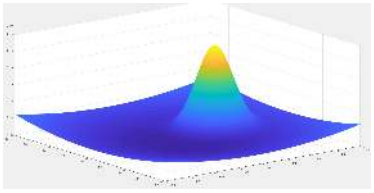


Figure 2: Ein Beispiel für einen sogenannten ,,Berg“

Durch einige Simulationen fanden wir heraus, dass das Objekt unsere Berge aber ignoriert, da diese zu hoch und spitz sind. Die Berge waren so spitz, dass sie einer Nadel glichen und unser Objekt sie somit missachtete. Wie man in der Grafik sieht, lief unser Objekt (rot gefärbt) am Weg zum Ziel (pink gefärbtes Kreuz) durch eine Mauer von einigen Bergen (türkis gefärbt) einfach durch. Auch das Erstellen einer Mauer mit mehreren Bergen, wie man es bei den Gebirgsketten x=7 oder y=5 erkennen kann, hatte keine abstoßende Wirkung auf unser Objekt. MATLAB war somit für diese Simulation leider nicht optimal geeignet.

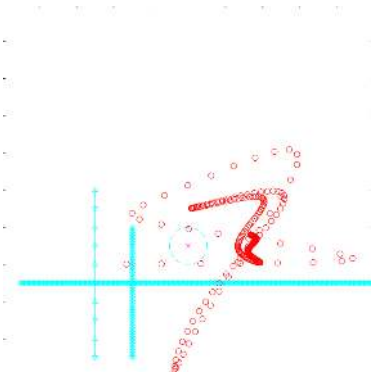


Figure 3: Das Objekt missachtet die Berge

Wenn sich das Objekt (grün) und auch das Ziel (blau) allerdings in einer Kreisbahn bewegt und nur ein Hindernis (rot) vorhanden ist, wird das Objekt sichtbar von seiner Bahn abgelenkt und umrundet das Hindernis, um sich danach dem Ziel wieder anzunähern.

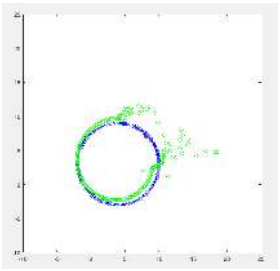


Figure 4: Das Objekt folgt dem bewegten Ziel, wird aber von einem Hindernis abgelenkt

3.2.1 Berge durch Gauß'sche Glockenkurve

Um unser Problem von vorhin zu beheben, haben wir versucht, unsere Berge anders zu modellieren. Dies haben wir mit Hilfe einer Gauß'schen Glockenkurve gemacht. Lässt man den Graph jener Kurve um die y-Achse rotieren, so ergibt sich ein hutförmiger Rotationskörper, der besser für die Modellierung der Bewegung unseres Objekts beim Treffen auf ein abstoßendes Potential geeignet ist.

$$U(x,y) = \beta \cdot e^{-\frac{(x-a)^2}{\sigma_1} - \frac{(y-b)^2}{\sigma_2}}$$

Der neue Berg lässt sich durch die oben genannte Formel beschreiben. σ_1 bzw. σ_2 beschreiben die Standardabweichungen vom Erwartungswert. Variiert man jene, so ergeben sich Hügel mit verschiedenen „Längen“ beziehungsweise „Breiten“. Dies haben wir umgesetzt und konnten nun beobachten, dass unser Objekt von den Hügeln abgelenkt wird.

4 Bewegung des Objekts

Dazu implementieren wir zunächst die wichtigste Komponente unseres Programms, das eigentliche Objekt, das sich in die Richtung unseres Zielpunkts, dem tiefsten Punkt des Potentialfeldes bewegen soll. Wir wollen also eine Kraft einführen, die das Objekt zu diesem Punkt hinbewegt. Wir haben uns dabei dem mathematischen Faktor des Gradienten bedient, ein Richtungsvektor, der im Falle eines dreidimensionalen Funktionsgraphen die zwei partiellen Ableitungen als Elemente hat. Dieser Gradient zeigt die Richtung des größten Anstiegs der Funktion im Punkt (x,y) auf dem Funktionsgraphen an.

$$\nabla F(x,y) = \begin{pmatrix} \partial_x F(x,y) \\ \partial_y F(x,y) \end{pmatrix}$$

Jedoch wird in unserem Modell die konträre Eigenschaft benötigt. Der negative Gradient beschreibt insofern die Richtung des steilsten Abstiegs, also die Eigenschaft, die wir, um den Pfad des Objektes von einem Punkt aus zu berechnen, bestimmen müssen. Unsere Kraft die somit dem negativen Gradienten entspricht ist somit eine konservative Kraft:

$$F = -\nabla U$$

In dieser Gleichung kennzeichnet U das Potential und das Zeichen ∇ den Gradienten.
Um Newton treu zu bleiben, beschreiben wir die Kraft, die auf die Position p zum Zeitpunkt t auf unser Objekt mit Masse m wirkt, wie folgt:

$$m \cdot \ddot{p}(t) = F$$

Wobei $\ddot{p}$ der zweiten Ableitung des Ortes, der Beschleunigung entspricht, die wir in jedem Punkt bestimmen wollen. Jede Beschleunigung unterteilt sich in einen horizontalen und vertikalen Anteil, weshalb wir unsere Beschleunigung weiter in Beschleunigungen in x- und y-Richtung, unterteilen. Daraus ergibt sich ein gekoppeltes System von Differenzialgleichungen zweiter Ordnung.

$$m \cdot \begin{pmatrix} \ddot{x}(t) \\ \ddot{y}(t) \end{pmatrix} = F$$

Da MATLAB aber nur Differenzialgleichungen erster Ordnung lösen kann, müssen wir eine Ordnungsreduktion unseres Gleichungssystems durchführen. Dies führen wir durch folgende Substitutionen durch:

$$x_2(t) = \dot{x}(t) \qquad\qquad y_2(t) = \dot{y}(t)$$

Wir definieren $x_2(t)$ beziehungsweise $y_2(t)$ als Geschwindigkeit des Objekts in x- und y-Richtung zum Zeitpunkt t, also der ersten Ableitung $\dot{x}$(t) und $\dot{y}$(t). Weiters werden in den folgenden Umformungen die x- und y-Position als $x_1(t)$ und $y_1(t)$ bezeichnet.

$$\dot{x_1}(t) = x_2(t) \qquad \dot{y_1}(t) = y_2(t)$$
$$\dot{x_2}(t) = \frac{F_1}{m} \qquad \dot{y_2}(t) = \frac{F_2}{m}$$

Laut der Definition ist die erste Ableitung von $x_1(t)$ gleich $x_2(t)$. Durch einfaches Umformen des oben angeführten zweiten Newton'schen Axioms bekommt man die zweite Gleichung. Natürlich ergeben sich auch die identen Gleichungen für $y_1(t)$ und $y_2(t)$.

4.1 Implementierung

```
1  n = 3;
2  alpha = 0.1;
3  x0 = 0;
4  y0 = 0;
```

Im oberen Code-Ausschnitt werden die Parameter für das attraktive Potential festgesetzt, wobei x0 und y0 die Koordinaten des Zielpunktes sind.

```
1  m = 3;
2  t0 = 0;
3  T = 1000;
4  N = 5000;
5  tt = linspace(t0,T,N);
```

Es wird die Masse des Objekts (m) deklariert und es wird das Intervall [t0;T] in N Zeitabschnitte unterteilt. Die daraus resultierenden Zeitpunkte werden im Vektor tt abgelegt.

```
1  x0i = 4;
2  y0i = 8;
3  vx0i = 1;
4  vy0i=-1;
```

Die Koordinaten des Startpunktes (x0i,y0i) und die Anfangsgeschwindigkeiten (vx0i, vy0i) werden gewaehlt.

```
1  for i=1:(N-1)
2  [tVeci,zi]=ode45(@(t,u)Func(t,u,m,n,alpha,x0,y0),[tt(i),tt(
       i+1)],[x0i,vx0i,y0i,vy0i]);
3  x0i=zi(end,1);
4  vx0i=zi(end,2);
5  y0i=zi(end,3);
6  vy0i=zi(end,4);
7  end
```

Wir werden dieses Differentialgleichungssystem numerisch beziehungsweise approximativ lösen. Dazu verwenden wir eine bereits in MATLAB vorhandene

Funktion „ode45“. In dieser Funktion werden die Ableitungen der Differentialgleichungen durch entsprechende Differenzenquotienten angenähert. Ein sehr einfaches Verfahren um dies zu erreichen ist das Vorwärts-Euler-Verfahren, das wie folgt funktioniert:
Für eine kleine Schrittweite $\Delta \cdot t$ ist

$$\begin{aligned}
\frac{x_1(t+\Delta t) - x_1(t)}{\Delta t} &\approx \dot{x_1}(t) = x_2(t) \\
\frac{x_2(t+\Delta t) - x_2(t)}{\Delta t} &\approx \dot{x_2}(t) = \frac{1}{m} \cdot F_1\left(x_1(t), y_1(t)\right) \\
\frac{y_1(t+\Delta t) - y_1(t)}{\Delta t} &\approx \dot{y_1}(t) = y_2(t) \\
\frac{y_2(t+\Delta t) - y_2(t)}{\Delta t} &\approx \dot{y_2}(t) = \frac{1}{m} \cdot F_2\left(x_1(t), y_1(t)\right)
\end{aligned}$$

Durch Umformen erhält man die Beziehungen:

$$\begin{aligned}
x_1(t+\Delta t) &= x_1(t) + \Delta t \cdot x_2(t) \\
x_2(t+\Delta t) &= x_2(t) + \frac{\Delta t}{m} \cdot F_1\left(t, x_1(t), y_1(t)\right) \\
y_1(t+\Delta t) &= y_1(t) + \Delta t \cdot x_2(t) \\
y_2(t+\Delta t) &= y_2(t) + \frac{\Delta t}{m} \cdot F_2\left(t, x_1(t), y_1(t)\right)
\end{aligned}$$

So lassen sich iterativ Näherungslösungen für Position und Geschwindigkeit in den Zeitpunkten $t + i \cdot \Delta t$ für gewählte Anfangspositionen und Anfangsgeschwindigkeiten berechnen. Wir übergeben der Funktion auch das Zeitintervall, auf dem eine Näherungslösung berechnet wird, und die abgeleiteten Startwerte, die extern berechnet werden.

Für diese externe Berechnung haben wir eigens eine Funktion erstellt (Func), die die abgeleiteten Werte wie folgend berechnet:

Bezeichnet $U_{ges} = U_{attr} + \sum_{i=0}^{n} U_{rep,i}$ das Gesamtpotenzial als Summe aus anziehendem Potenzial U_{attr} und den abstoßenden Potenzialen $U_{rep,i}$, so erhalten wir durch partielles Ableiten die Gleichungen:

$$\begin{aligned}
\dot{x_1}(t) &= x_2(t) \\
\dot{x_2}(t) &= \partial_x U_{ges}(x_1(t), y_1(t)) \\
\dot{y_1}(t) &= y_2(t) \\
\dot{y_2}(t) &= \partial_y U_{ges}(x_1(t), y_1(t))
\end{aligned}$$

Wobei gilt:

$$\partial_x U_{attr}(x,y) = \alpha \cdot n \cdot (x - x_0) \cdot ((x-x_0)^2 + (y-y_0)^2)^{\frac{n-2}{2}}$$

$$\partial_y U_{attr}(x,y) = \alpha \cdot n \cdot (y - y_0) \cdot ((x-x_0)^2 + (y-y_0)^2)^{\frac{n-2}{2}}$$

$$\partial_x U_{rep}(x,y) = \beta \cdot \left[(-1) \cdot \left[w + ((x-a)^2 + (y-b)^2)^{\frac{q}{2}} \right]^{-2} \cdot q \cdot \left[(x-a)^2 + (y-b)^2 \right]^{\frac{q-2}{2}} \cdot (x-a) \right]$$

$$\partial_y U_{rep}(x,y) = \beta \cdot \left[(-1) \cdot \left[w + ((x-a)^2 + (y-b)^2)^{\frac{q}{2}} \right]^{-2} \cdot q \cdot \left[(x-a)^2 + (y-b)^2 \right]^{\frac{q-2}{2}} \cdot (y-a) \right]$$

Diesen Ablauf schrieben wir in MATLAB wie folgt nieder, wobei in diesem Fall keine abstoßende Kraft berücksichtigt wird:

```
1  function y = Func(t,u,m,n,alpha,x0,y0)
2  y = zeros(4,1);
3  y(1)=u(2);
4  y(2)= (-n/m*alpha*(norm([u(1)-x0;u(3)-y0],2))^(n-2)*(u(1)-
       x0));
5  y(3)= u(4);
6  y(4)= (-n/m*alpha*(norm([u(1)-x0;u(3)-y0],2))^(n-2)*(u(3)-
       y0));
7  end
```

Die Lösungen, die ode45 für Position und Geschwindigkeit des Objekts ausgibt, werden im weiteren Verlauf der Schleife für die nächste Iteration als Ausgangswerte gesetzt.

5 Hindernisse

Wir erstellten sogenannte Hindernislisten, welche die Befehle kurz und knapp zusammenfassen, ohne viel Schreibaufwand. Diese Listen enthalten neben der Position der Hindernisse auch Parameter, welche dazu verwendet werden, um diese Hindernisse mathematisch darzustellen.

```
M = 3;
hindernisListe = zeros(M, 5);

hindernisListe(1, 1) = 10;
hindernisListe(1, 2) = 10;
hindernisListe(1, 3) = 4;
hindernisListe(1, 4) = 4;
hindernisListe(1, 5) = 100000;

hindernisListe(2, 1) = 5;
hindernisListe(2, 2) = 0;
hindernisListe(2, 3) = 2;
hindernisListe(2, 4) = 2;
hindernisListe(2, 5) = 100000;

hindernisListe(3, 1) = 0;
hindernisListe(3, 2) = 5;
hindernisListe(3, 3) = 3;
hindernisListe(3, 4) = 10;
hindernisListe(3, 5) = 100000;

plot(hindernisListe(1, 1), hindernisListe(1, 2), '-gx', '
    MarkerSize', 15);
hold on;
plot(hindernisListe(2, 1), hindernisListe(2, 2), '-gx', '
    MarkerSize', 15);
hold on;
plot(hindernisListe(3, 1), hindernisListe(3, 2), '-gx', '
    MarkerSize', 15)
```

Danach wird noch der dazugehörige Plot dargestellt. Dieser enthält die Eigenschaften, die essentiell für die Grafik sind. Zuerst werden die Koordinaten des Hindernisses angegeben, danach die Farbe (in diesem Fall grün), danach die Größe des Hindernisses („MarkerSize“, 15).

```
for i=1:N-1
[tVeci,zi] = ode45(@(t,u)my_attr_force(t,u,1, n, alpha, xtrg,
     ytrg, hindernisListe, M),[tt(i),tt(i+1)],[x0i,vx0i,y0i,
    vy0i]);
end
```

Danach muss die Hindernisliste noch an die Funktion ode45 übergeben werden. In die Variablen [tVeci,zi] werden dann die Näherungslösungen geschrieben.

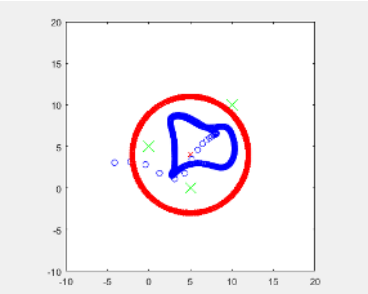


Figure 5: Blau ist die Bewegung des Objekts; rot die Kreisbahn, auf der sich das Ziel bewegt. Die grünen Kreuze zeigen die Hindernisse, die das Objekt ablenken.

6 Modellierung des Bremsvorganges

Nachdem es uns gelungen ist mithilfe unserer eingeführten Kraft das Objekt in Richtung Ziel zu bewegen, haben wir versucht die Geschwindigkeit des Objekts in der Nähe des Ziels zu verringern. Dies ermöglichte uns schlussendlich, dass sich das Objekt nicht stets am Ziel vorbei bewegte, sondern sich mit der Zeit nur noch knapp vom Ziel entfernt aufhielt, ohne noch vollkommen stehen zu bleiben.

6.1 Manuelle Geschwindigkeitsreduktion

Unsere erste Idee bestand darin, dass wir in einem Radius ϵ um den Zielpunkt, die Geschwindigkeit um einen gewissen Faktor verringern. Dafür haben wir zuerst die Zeitschritte manuell gesteuert. Von unserem Startpunkt ausgehend haben wir unser Objekt pro Zeiteinheit um den Richtungsvektor multipliziert mit der Höchstgeschwindigkeit bewegen lassen. Dadurch sind immer wieder neue Punkte entstanden, von denen wiederum neue Punkte berechnet wurden, die sich mit der Zeit immer weiter dem Ziel annäherten, bis sie sich so nah am Ziel befanden, dass sie im Kreis um den Zielpunkt mit dem vorher bestimmten Radius ϵ lagen. Ab diesem Zeitpunkt wurde die Geschwindigkeit laufend verringert, sodass das Objekt immer langsamer wurde, je näher es dem Ziel kam.

```
1          w = [zi(end,1)- x0i; zi(end,3)- y0i];
2          wpos = maxVel*(w/norm(w,2));
3
4          xynew = x0iy0i + wpos; %neuerPunkt
5          if norm(x0y0 - xynew,2)<eps
6          maxVel = 0.5
7          else
8          maxVel = 3;
9          end
```

6.2 Reibungskraft

Unsere zweite Idee basiert auf der Reibungskraft, die bislang missachtet wurde. Ab einem gewissen Abstand um den Zielpunkt wirkt diese Reibungskraft auf das Objekt, wodurch die Geschwindigkeit ebenso wie in der ersten Idee verringert wird. Um diesen Zustand zu erreichen, haben wir eine Kraft eingeführt, die der ursprünglichen Anziehungskraft mit einem Faktor (k) entgegenwirkt und direkt proportional zur Momentangeschwindigkeit ist.

```
1        function v = myAttrPoFunc (t,u,m,alpha,beta,rho0,n,x0,y0,a,
             b)
2        tmp = norm([u(1)-x0;u(3)-y0],2);
3        tmp1 = sqrt((u(1)-a)^2+(u(3)-b)^2);
4
5        v = zeros(4,1);
6        v(1) = u(2);
7        v(2) = -n/m*alpha*tmp^(n-2)*(u(1)-x0)+(1/m)*(beta*(tmp1
             ^(-3)*(u(1)-a)))*(tmp1<=rho0);
8        v(3) = u(4);
9        v(4) = -n/m*alpha*tmp^(n-2)*(u(3)-y0)+(1/m)*(beta*(tmp1
             ^(-3)*(u(3)-b)))*(tmp1<=rho0);
10
11       r=2;
12       k=10;
13       if tmp <= r
14       v(2) = v(2) - k/m * u(2);
15       v(4) = v(4) - k/m * u(4);
```

6.3 Modellierung des Anhaltens

Im nächsten Schritt versuchten wir das Objekt am Zielpunkt beziehungsweise in der Nähe des Zielpunktes zum Stillstand zu bringen. Zuallererst führten wir einen Toleranzbereich (ϕ) um den Zielpunkt ein. Sobald sich das Objekt in diesem Bereich befand, ließen wir das Programm die Berechnung neuer Punkte stoppen, sodass das Objekt an diesem Punkt stehen blieb.

```
1        while norm([x0i-x0;y0i-y0],2)>phi && (i < N -1)
2        end
```

7 Dynamische Ziele

Um für Abwechslung zu sorgen, erstellten wir einen beweglichen Zielpunkt, der im Folgenden von dem Objekt verfolgt wurde. Dieses Modell ist zwar für unsere Ausgangssituation des autonomen Fahrens nicht unbedingt übertragbar, da sich der Ort, zu dem wir mit dem Auto fahren, üblicherweise nicht von der Stelle bewegt. Jedoch gibt es einige andere Anwendungsmöglichkeiten in der Realität, wie zum Beispiel ein selbstfahrender Koffer der Firma Piaggio, der einer Person (in unserem Fall dem Ziel) folgt. Im Folgenden werden unterschiedliche Verläufe eines Ziels und dessen Programmierung erläutert.

7.1 Zielpunkte als Kreisbahn

Um eine Situation für einen rotierenden Zielpunkt modellieren zu können, ließen wir das Programm bei jedem Durchlauf einen neuen Zielpunkt berechnen, indem vom Kreismittelpunkt ausgehend in einem gewissen Radius die x-Koordinate mit der Cosinusfunktion beziehungsweise die y-Koordinate mit der Sinusfunktion bestimmt wurden. In dieser Grafik werden zwei „Linien“ dargestellt, eine rote (Ziel) und eine blaue (Objekt). Das Objekt folgt dem Ziel und weicht währenddessen den grünen Kreuzen (Hindernisse) aus.
Der dazugehörige Code:

```
1        omega = 4;
2        trgPathRad = 7;
3        trgPathMdPnt = [5; 4];
4        N = 1000;
5
6        psi = linspace(0, 4 * pi, N);
7        xtrg = trgPathMdPnt(1) + trgPathRad * cos(omega * psi(1));
8        ytrg = trgPathMdPnt(2) + trgPathRad * sin(omega * psi(1));
```

Wir definieren zuerst die Winkelgeschwindigkeit (ω), sowie den Radius (trgPathRad) und die Koordinaten des Mittelpunkts (trgPathMdPnt) der Kreisbahn. Der Winkel wird in kleinere Teilstücke (ψ), die Anzahl dieser wird im Code als N angegeben, unterteilt und noch bevor wir in der Schleife beginnen die Bewegungen unseres Objekts zu bestimmen, werden die Startkoordinaten des Ziels berechnet.

```
1        for i=1:N-1
2        xtrg = trgPathMdPnt(1) + trgPathRad * cos(omega * psi(i +
             1));
3        ytrg = trgPathMdPnt(2) + trgPathRad * sin(omega * psi(i +
             1));
4        end
```

Nun werden die Koordinaten des Ziels je einmal pro Zeiteinheit neu berechnet, in dem der zur Rechnung verwendete Winkel um eine Einheit (ψ) erhöht wird.

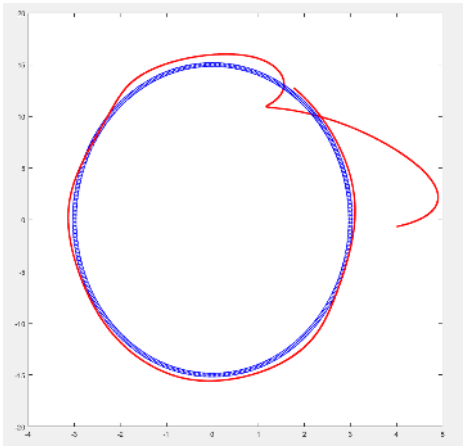


Figure 6: Man sieht wie das Objekt (rot) dem sich im Kreis bewegenden Ziel (blau) folgt.

7.2 Zielpunkte als Ellipsenbahn

Weiteres wurde von uns ein Zielpunkt, der sich in einer Ellipse bewegt, erstellt, indem den x- und y-Koordinaten verschieden große Radien (trgradx und trgrady) zugeteilt wurden.

```
1        trgM= [0;0];
2        trgradx=5;
3        trgrady=15;
4        x0= trgM(1,1) + trgradx* cos(omega*psi(i));
5        y0= trgM(2,1) + trgrady* sin(omega*psi(i));
6        while norm([x0i-x0;y0i-y0],2)>phi && (i < N -1)
7        x0= trgM(1,1) + trgradx* cos(omega*psi(i+1));
8        y0= trgM(2,1) + trgrady* sin(omega*psi(i+1));
9        end
```

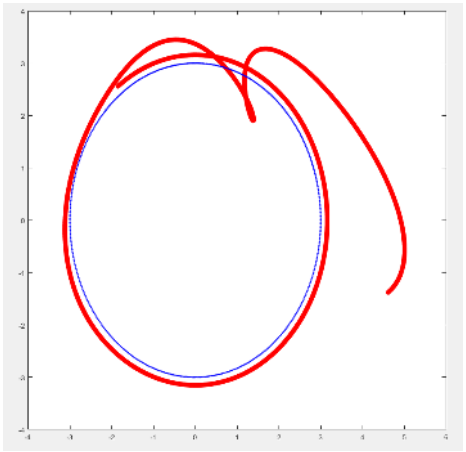


Figure 7: Das Objekt (rot) folgt wiederum dem Ziel (blau), das sich diesmal auf einer Ellipsenbahn bewegt.

7.3 Zielpunkte mit zufälligen Koordinaten

Zusätzlich haben wir auch die Koordinaten des Zielpunktes in keinem bestimmten Muster laufen lassen, d. h. die Punkte wurden zufällig bestimmt. Das erreichten wir durch einen Random-Operator (rand) und einem gewählten Parameter r, der bestimmt, in welchem Bereich die Punkte auftreten sollen.

```
1          r = 7;
2          while norm([x0i-x0;y0i-y0],2)>phi && (i < N -1)
3          x0=rand*r;
4          y0=rand*r;
5          end
```

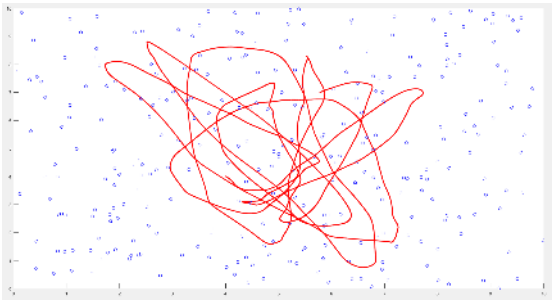


Figure 8: Zielpunkte (blau) werden zufällig gewählt und Objekt (rot) folgt dem jeweiligen Punkt.

8 Fazit

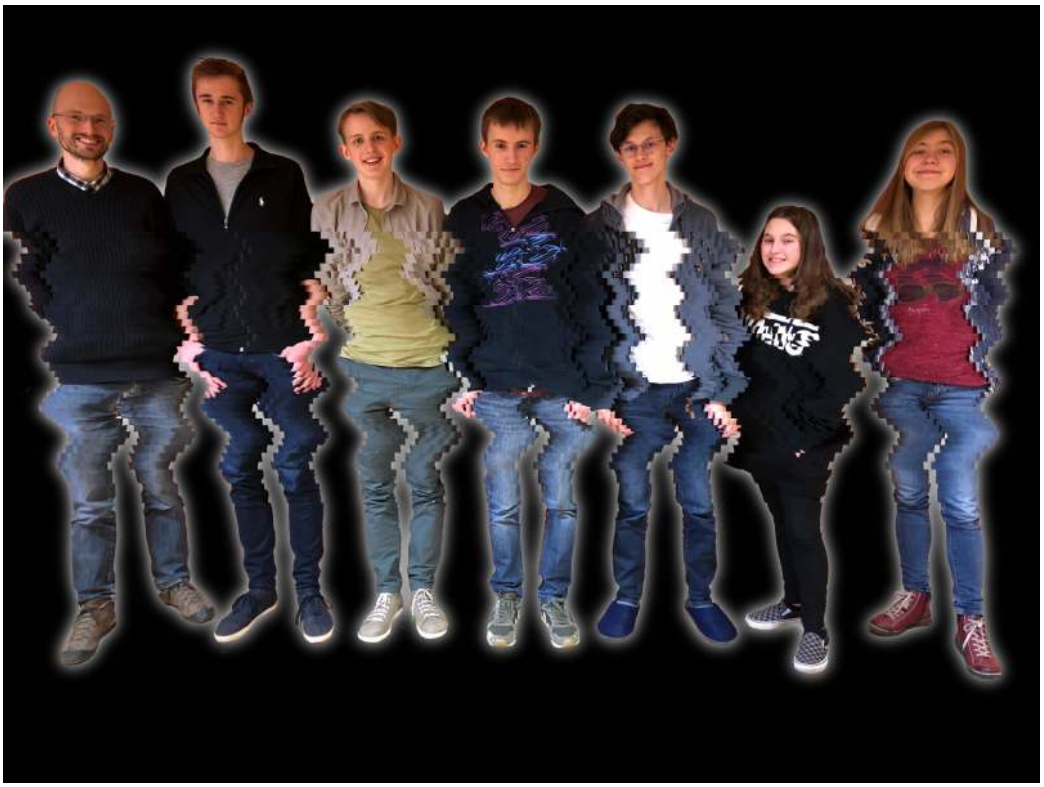
Anfangs war MATLAB für unsere Problemstellung gut geeignet, da wir es für unkomplizierte Modelle benutzen konnten. Zum Beispiel konnten wir problemlos die Bewegung eines Objekts auf ein Ziel simulieren. Auch Simulationen mit nur einem Hindernis verliefen gut. Als wir allerdings versuchten sogenannte „Gebirgsketten“ beziehungsweise Mauern einzubauen, sind wir auf Probleme gestoßen. Diese Probleme konnten wir aber dann mit einer anderen Methode, der Gauß'schen Glockenkurve, umgehen. Im Allgemeinen lässt sich sagen, dass wir durch viel Ausprobieren immer zu unseren gewünschten Ergebnissen gekommen sind. In dieser Woche hatten wir die Chance, in die Welt des Programmierens einzutauchen, wobei auch Mathematik und Physik nicht vernachlässigt wurden.

Das Unsichtbare sichtbar machen:

Algebraische Rekonstruktionstechniken in der Computertomographie

Projekt: Medizinische Bildgebung

Elias Krainer, Klara Balic, Lukas Ertl, Max Schüßler, Mira Zuser, Nico Leonhard
Gruppenbetreuer: Dr. Robert Beinert, BSc MSc



Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Um einen Körper von außen zu untersuchen, hat Wilhelm Conrad Röntgen die Röntgendiagnostik entwickelt. Dadurch, dass die Bilder nicht aus verschiedenen Winkeln aufgenommen werden, wird die Interpretation der Bilder erschwert, da sich hintereinanderliegende Objekte überschneiden. Wegen dieser Überschneidung erhält man nur einen Schattenriss. Um das zu verhindern, galt es, eine andere Methode zur Diagnose zu entwickeln, nämlich die Computertomographie(CT). Damit man Operationen präziser geplant werden konnten und detailliertere Einblicke möglich wurden, musste man die Überschneidungen umgehen. Um das zu erreichen, werden Röntgenbilder aus mehreren Winkeln aufgenommen. Aus den Bildern kann man den Querschnitt des durchleuchteten Gegenstandes zwei- oder dreidimensional rekonstruieren. Dies löste Johann Radon 1917 theoretisch. Es konnte jedoch bis zur Entwicklung leistungsstarker Rechner in der Praxis keine Anwendung finden. 1979 wurde der Nobelpreis an Hounsfield und Cormack verliehen, da sie erste Untersuchungen mit einem CT möglich machten.

In unserem Projekt haben wir zunächst ein Modell entwickelt um den Zusammenhang zwischen Querschnitten und Röntgenstrahlen herzustellen. Grundlage für unser Modell war das Lamberd-Beersche Gesetz. Danach wurde dieses Modell diskretisiert, um die Aufgabe numerisch zu betrachten. Bei der Diskretisierung entstand ein großes Gleichungssystem, welches nicht in den Arbeitsspeicher passte. Wir lösten es daher durch Projektion näherungsweise. Dann konnten wir den Querschnitt rekonstruieren.

2 Kontinuierliches Modell

2.1 Lambert-Beersches Gesetz

Während unseres Arbeitsprozesses haben wir das Lambert-Beersche Gesetz als Gegebenheit angenommen.

$$I_0(\vec{\theta}, s) = I_1(\vec{\theta}, s) \cdot \exp\left(\int_\infty^\infty f\left(t \cdot \theta + s \cdot \theta^\perp\right)\right)$$

Dieses beschreibt eine exponentielle Abnahme der Emissionsintensität I_0 und der gemessenen Intensität I_1 von Röntgenstrahlen im Zusammenhang mit der Verschiebung s zwischen parallelen Röntgenstrahlen sowie ihrer Richtung θ. Durch Umkehren und Logarithmieren dieses bringen wir die Intensitäten, die durch unsere Daten dargestellt werden, auf eine Seite, und das Integral auf die andere. Durch Umformung erhalten wir

$$\begin{aligned}
I_0(\vec{\theta}, s) &= I_1(\vec{\theta}, s) \cdot \exp\left(\int_\infty^\infty f\left(t \cdot \vec{\theta} + s \cdot \vec{\theta}^\perp\right)\right) \\
\frac{I_0(\vec{\theta}, s)}{I_1(\vec{\theta}, s)} &= \exp\left(\int_\infty^\infty f\left(t \cdot \vec{\theta} + s \cdot \vec{\theta}^\perp\right)\right) \\
\ln\left(\frac{I_0(\vec{\theta}, s)}{I_1(\vec{\theta}, s)}\right) &= \int_\infty^\infty f\left(t \cdot \vec{\theta} + s \cdot \vec{\theta}^\perp\right) \\
\ln\left(I_0(\vec{\theta}, s)\right) - \ln\left(I_1(\vec{\theta}, s)\right) &= \int_\infty^\infty f\left(t \cdot \vec{\theta} + s \cdot \vec{\theta}^\perp\right)
\end{aligned}$$

Der Zusammenhang in der letzten Zeile ist auch bekannt als die Radontransformation und kann auf folgende Weise angeschrieben werden.

$$\ln\left(I_0(\vec{\theta}, s)\right) - \ln\left(I_1(\vec{\theta}, s)\right) = \mathbf{R}[f](\vec{\theta}, s)$$

2.2 Radontransformation

Die Radon-Transformation

$$\mathbf{R}[f](\vec{\theta}, s) = \int_\infty^\infty f\left(t \cdot \vec{\theta} + s \cdot \vec{\theta}^\perp\right)$$

hat die folgenden Eigenschaften.

Satz. Die Radontransformation ist linear. Das bedeutet, dass

$$\boldsymbol{R}[\alpha f + \beta g](\vec{\theta}, s) = \alpha \boldsymbol{R}[f](\vec{\theta}, s) + \beta \boldsymbol{R}[g](\vec{\theta}, s)$$

Beweis.

$$\begin{aligned}
\boldsymbol{R}[\alpha f + \beta g](\vec{\theta}, s) &= \int_{-\infty}^{\infty} [\alpha f + \beta g](s\vec{\theta} - t\vec{\theta}^{\perp}) dt \\
&= \int_{-\infty}^{\infty} \alpha f(s\vec{\theta} - t\vec{\theta}^{\perp}) + \beta g(s\vec{\theta} - t\vec{\theta}^{\perp}) dt \\
&= \alpha \int_{-\infty}^{\infty} f(s\vec{\theta} - t\vec{\theta}^{\perp}) dt + \beta \int_{-\infty}^{\infty} g(s\vec{\theta} - t\vec{\theta}^{\perp}) dt \\
&= \alpha \boldsymbol{R}[f](\vec{\theta}, s) + \beta \boldsymbol{R}[g](\vec{\theta}, s) \qquad \square
\end{aligned}$$

Satz. Die Radontransformation ist symmetrisch. Das bedeutet, dass

$$\boldsymbol{R}[f](\vec{\theta}, s) = \boldsymbol{R}[f](-\vec{\theta}, -s)$$

Beweis. Aus der Definition der Radon-Transformation folgt

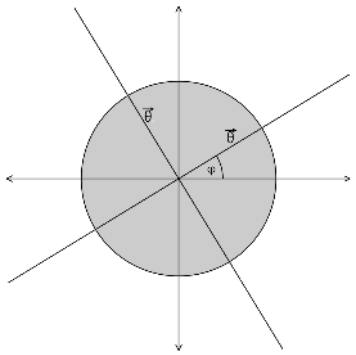
$$\boldsymbol{R}[f](-\vec{\theta}, -s) = \int_{-\infty}^{\infty} f(-t\vec{\theta} + s\vec{\theta}^{\perp}) dt.$$

Durch substituieren von $-t$ mit t' und dt mit $-dt'$ erhält man

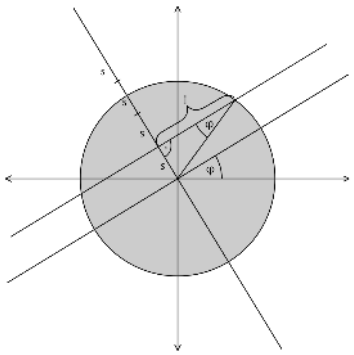
$$\boldsymbol{R}[f](\vec{\theta}, s) = -\int_{\infty}^{-\infty} f(t\vec{\theta} + s\vec{\theta}^{\perp}) dt = \int_{-\infty}^{\infty} f(t\vec{\theta} + s\vec{\theta}^{\perp}) dt = \boldsymbol{R}[f](\vec{\theta}, s) \quad \square$$

2.3 Kreis und Quadrat

Die Gruppe hat sich mit der Radontransformation der basischen zweidimensionalen Formen eines Kreises und eines Quadrates beschäftigt. Durch den Richtungsvektor $\vec{\theta}$ der Röntgenstrahlen sowie der Distanz s zwischen den Röntgenstrahlen kann mithilfe von grundlegender Trigonometrie die zurückgelegte Distanz, beziehungsweise das Gewicht (engl. weight), eines bestimmten Strahls $\left(s \cdot \theta^{\perp} + t \cdot \theta\right)$ in einem Objekt berechnet werden.



(a) Einheitskreis mit Richtungsvektoren $\vec{\theta}$ und $\vec{\theta}^{\perp}$ und Winkel ϕ eingezeichnet

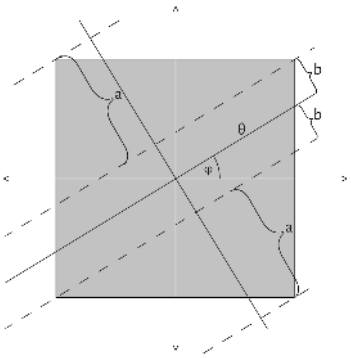


(b) Einheitskreis mit möglicher Verschiebung s und resultierender Länge l zwischen $\vec{\theta}^{\perp}$ und dem Umkreis

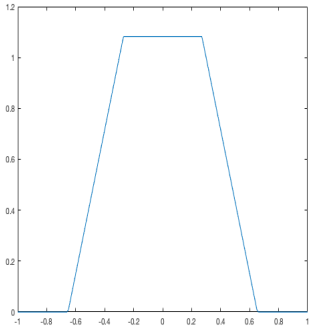
Abbildung 1: Gewichte im Einheitskreis

Für einen Kreis ist die Ermittlung der Länge in erster Linie trivial. Sie kann mithilfe der Illustrationen 1a und 1b dargestellt werden. Man nehme an, der Kreis ist ein Einheitskreis, sprich ein Kreis mit dem Radius gleich 1. Die Länge des Strahles, der durch den Mittelpunkt geht, ist somit gleich 2. Geht der Röntgenstrahl durch den Kreis mit einer Verschiebung s vom Mittelpunkt, so können wir den Satz von Pythagoras verwenden um die Länge l zu berechnen. Aus diesem schließen wir $l = \sqrt{1 - s^2}$ und somit $\boldsymbol{R}[f](\vec{\theta}, s) = 2\sqrt{1 - s^2}$ für $s \leq 1$ und $\boldsymbol{R}[f](\vec{\theta}, s) = 0$ sonst.

Das Gewicht eines Röntgenstrahls in einem Quadrat mit Seitenlänge 1, welches wir hier χ nennen, kann auch mithilfe von Winkelrechnung und trigonometrischen Zusammenhängen ermittelt werden, dies illustrieren wir mithilfe der Abbildungen.

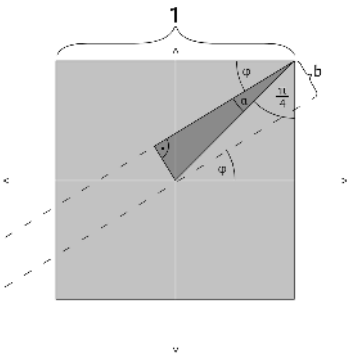


(a) Einheitsquadrat χ mit Bereichen a und b, in denen verschiedene Merkmale auf das Gewicht zutreffen

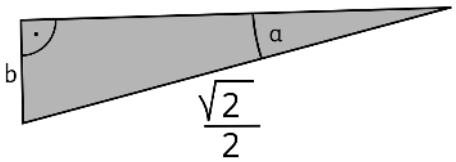


(b) x-Achse: Distanz zum Mittelpunkt s, y-Achse: Länge des Strahls im Quadrat

Abbildung 2: Gewichte und Unterteilungen im Einheitsquadrat χ



(a) χ mit eingezeichnetem rechtwinkeligem Dreieck, Winkel α und Gegenkathete b



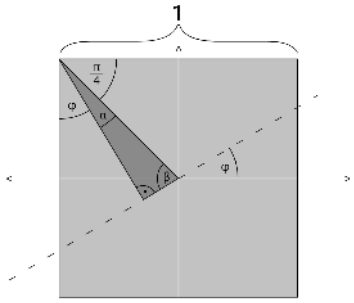
(b) Rechtwinkeliges Dreieck, dessen Hypotenuse die halbe Diagonale von χ ist

Abbildung 3: Berechnung von b

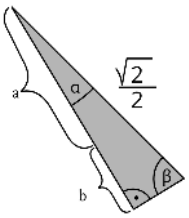
Das Gewicht des Strahls in χ ist interessant, da es je nach der Distanz vom Mittelpunkt entweder konstant bleibt oder mit größerer Distanz linear abnimmt. Diese Bereiche sind in der Abbildung 2a jeweils mit b und a gekennzeichnet, und beide dieser Längen hängen vom Winkel ϕ ab. Für den Winkel $\phi = \frac{\pi}{8}$ stellt der Graph in 2b diese Linearität dar. Im Bereich $s \leq b$ ist das Gewicht konstant ähnlich wie beim Kreis. Um die Länge zu berechnen benutzen wir den Zusammenhang zwischen Kosinus und dem Verhältnis zwischen Ankathete und Hypotenuse. Für $s \leq b$ erhalten wir in Abhängigkeit vom Winkel

$\boldsymbol{R}[\chi](\vec{\theta}, s) = 1/\cos(\phi)$. Die Ermittlung der Länge b kann näher erläutert werden mithilfe von 3a und 3b. Die zwei gestrichelten Röntgenstrahlen stehen zueinander parallel und schließen mit der x-Achse und der oberen Seite des Quadrats jeweils den Winkel ϕ ein. Wir können ein rechtwinkeliges Dreieck bilden, dessen Hypotenuse die halbe Diagonale von χ ist, und in welchem die Länge b die Gegenkathete des neuen Winkels α ist. Mit dem Satz von Pythagoras legen wir die Hypotenuse des Dreiecks als $\frac{\sqrt{2}}{2}$ fest, und wir wissen, dass der Sinus eines Winkels in einem rechtwinkeligen Dreieck das Verhältnis zwischen seiner Gegenkathete und der Hypotenuse beschreibt. Zusammenhang zwischen b, α und ϕ:

$$\begin{aligned}
\frac{\pi}{2} &= \phi + \alpha + \frac{\pi}{4} \\
\alpha &= \frac{\pi}{4} - \phi \\
b &= \frac{\sqrt{2}}{2} \cdot \sin(\alpha) = \frac{\sqrt{2}}{2} \cdot \sin\left(\frac{\pi}{4} - \phi\right)
\end{aligned}$$



(a) χ mit eingezeichnetem rechtwinkeligem Dreieck Winkel β und Gegenkathete $a + b$



(b) Rechtwinkeliges Dreieck, ähnlich zu Dreieck in 3b, Winkel β und Gegenkathete $a + b$

Abbildung 4: Berechnung des Bereichs a in χ

Mit dieser Information kommen wir auch leicht zur Länge a. In der Abbildung 2a kann ein ähnliches Dreieck zu jenem in 3b eingezeichnet werden. Dieses Dreieck, abgebildet in 3b besteht aus den Winkeln α, dem daraus herleitbaren Winkel β, der Hypotenuse $\frac{\sqrt{2}}{2}$, und Gegenkathete zu β, welches

aus den Längen a und b besteht. Zusammenhang zwischen β und ϕ:

$$\beta = \frac{\pi}{4} - \alpha = \frac{\pi}{4} + \phi$$
$$a + b = \frac{\sqrt{2}}{2} \cdot \sin(\beta) = \frac{\sqrt{2}}{2} \cdot \sin\left(\frac{\pi}{4} + \phi\right)$$
$$a = \frac{\sqrt{2}}{2} \cdot \sin\left(\frac{\pi}{4} + \phi\right) - b$$

Wie oben schon demonstriert, ist im Bereich b das Gewicht gleich $\frac{1}{\cos(\phi)}$. Im Bereich a nimmt es linear ab, bis es null beträgt. Diese Linearität erweist sich durch die Anwendung des Strahlensatzes, siehe 5.

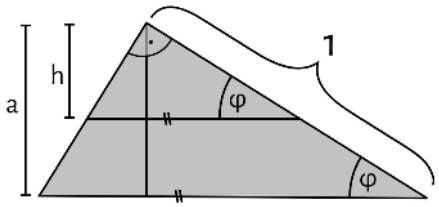


Abbildung 5: Zwei ähnliche rechtwinkelige Dreiecke; größeres mit dem längsten Gewicht in χ als Hypotenuse und Höhe a, und kleineres mit Höhe h

Die zwei Hypotenusen verlaufen parallel zueinander, und der rechte Winkel sowie ϕ bleiben erhalten, also sind die zwei eingezeichneten Dreiecke ähnlich. Wird die Höhe des rechten Winkels auf die Hypotenuse von a auf h reduziert, so beschreibt das Verhältnis $\frac{a}{h}$ das Verhältnis aller Seiten des größeren Dreiecks zu den entsprechenden Seiten des kleineren Dreiecks. Zudem haben wir, weil Quadrate vier Symmetrieachsen haben, uns auf einen Definitionsbereich $\left[0,\frac{\pi}{4}\right]$ geeinigt für den Winkel ϕ. Für Winkel größer als $\frac{\pi}{4}$ gehen wir von äquivalenten Ergebnissen zu jenen im Bereich $\left[0,\frac{\pi}{4}\right]$ aus; für negative Winkel entspricht der Wert des Gewichts demjenigen, den das Gewicht für den Absolutbetrag desselben Winkels annehmen würde.

2.4 Sinogramm

Die Perspektivengrenzen eines einfachen Röntgenbildes kann man umgehen, indem man mehrere Röntgenbilder eines Objektes aus verschiedenen Winkeln zusammenfügt. Eine Art, diese darzustellen, ist ein Sinogramm. Für

das Sinogramm werden aus unendlich vielen verschiedenen Winkeln um eine Drehachse herum Röntgenbilder aufgenommen. Pro Aufnahme wird dann jeweils eine Reihe auf derselben Höhe im Bild entnommen, und diese Reihen werden zusammengefügt in ein kontinuierliches Bild. Das bedeutet also, dass ein Sinogramm eine vollständige „Drehtischaufnahme“ eines Objektes auf einer bestimmten Höhe darstellt.

2.5 Inversion der Radontransformation

Als nächstes wollen wir eine Inversionsformel für die kontinuierliche Radontransformation herleiten. Hierfür verwenden wir die eindimensionale

$$\boldsymbol{F}[g](\omega) = \int_{-\infty}^{\infty} g(t)e^{-i\omega t} \quad dt$$

und die zweidimensionale Fourier-Transformation

$$\boldsymbol{F}[g](\omega_1, \omega_2) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(t_1, t_2)e^{-i\omega_1 t_1}e^{-i\omega_2 t_2} \quad dt_1 dt_2$$

für ein- und zweidimensionale Funktionen g mit kompakten Trägern.

Projektionssatz. Für die Radontransformation gilt

$$\widehat{\boldsymbol{R}}[g](\vec{\theta}, \sigma) = \widehat{g}(\vec{\theta}, \sigma),$$

wobei links die Fourier-Transformation bezüglich der Verschiebung s und rechts die zweidimensionale Fourier-Transformation verwendet wird.

Beweis. Wir beginnen mit der eindimensionalen Fourier-Transformation Radontransformation bezüglich s, welche durch

$$\widehat{\boldsymbol{R}}[g](\vec{\theta}, \sigma) = \int_{-\infty}^{\infty}\left[\int_{-\infty}^{\infty} g(t\vec{\theta} + s\vec{\theta^{\perp}})dt\right]e^{-i\sigma s}ds \tag{1}$$

gegeben ist. Bildlich integrieren wir die Funktion g entlang den Koordinaten in Richtung θ und $\theta^{\perp}$.

Mithilfe des Substitutionssatzes wollen wir die Funktion g entlang der kartesischen Koordinaten integrieren. Für eine beliebige Funktion f und Koordinatentransforamtion κ hat der Substitutionssatz die folgende Form

$$\int\int f(\kappa(s,t))dsdt = \int\int f(\kappa(\kappa^{-1}(t_1,t_2)))|detD_{\kappa^{-1}}|dt_1dt_2$$

$$= \int\int f(t_1,t_2)\frac{1}{|detD_{\kappa}(t_1,t_2)|}dt_1dt_2.$$

In unserem Fall sind die kartesischen Koordinaten durch die Abbildung κ mit

$$\begin{aligned}\kappa(s,t) &= t\theta + s\theta^{\perp} \\ &= t\begin{pmatrix}\cos(\phi)\\ \sin(\phi)\end{pmatrix} + s\begin{pmatrix}-\sin(\phi)\\ \cos(\phi)\end{pmatrix} \\ &= \begin{pmatrix}t\cdot\cos(\phi) - s\cdot\sin(\phi)\\ t\cdot\sin(\phi) + s\cdot\cos(\phi)\end{pmatrix} = \begin{pmatrix}t_1\\ t_2\end{pmatrix}.\end{aligned}$$

Die Jacobimatrix, die Matrix der partiellen Ableitung, ist für diese Koordinatentransformation gegeben durch

$$D_\kappa = \begin{pmatrix}\cos(\phi) & -\sin(\phi)\\ \sin(\phi) & \cos(\phi)\end{pmatrix}.$$

Die Determinante dieser Matrix ist

$$\det D_\kappa = (\cos(\phi))^2 + (\sin(\phi))^2 = 1.$$

Um den Substitutionssatz anzuwenden benötigen wir eine Darstellung der Variable s durch t_1 und t_2. Hierfür lösen wir das Gleichungssystem nach s auf, indem wir die zweite Gleichung mit $\cos(\phi)/\sin(\phi)$ multiplizieren und von der ersten Gleichung abziehen. Auf diese Weise erhalten wir die Gleichungen

$$t_1 - \frac{\cos(\phi)}{\sin(\phi)}\cdot t_2 = -s\sin(\phi) - s\frac{(\cos(\phi))^2}{\sin(\phi)}$$

beziehungsweise

$$\sin(\phi)\cdot t_1 - \cos(\phi)\cdot t_2 = -s(\sin(\phi))^2 - s(\cos(\phi))^2.$$

Somit können wir s darstellen als $s = \cos(\phi)\cdot t_2 - \sin(\phi)\cdot t_1$. Nach anwenden des Substitutionssatzes können wir das Integral (1) als

$$\widehat{\boldsymbol{R}}[g](\vec{\theta},\sigma) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(t_1,t_2)\, e^{-i(\sigma\vec{\theta}\bullet\vec{t})}\, dt_1 dt_2 = \widehat{g}(\sigma\vec{\theta})$$

mit $\vec{t} = (t_1,t_2)^T$, was zu beweisen war. □

Mit Hilfe des Projektionssatzes und der inversen Fourier-Transformation können wir aus einem gegebenen Sinogramm wieder den ursprüngliche Querschnitt rekonstruieren.

Korollar. Die Radontransformation ist injektiv (linkseindeutig).

3 Diskretes Modell

3.1 Scanner-Geometrie

Wir haben bei unserem Projekt Daten mit zwei verschiedenen Scanner-Geometrien verwendet. Wir selbst haben den Datensatz eines Parallelscanners generiert. Das Bedeutet, dass die Strahlen parallel aus der Quelle kommen und dann in die verschiedenen Winkel gedreht werden. Zum Vergleich wurden fertige Daten von den Aufnahmen einer Walnuss mittels eines Fächerscanners verwendet. Hier kommen die Strahlen in Fächerform aus der Quelle und werden dann ebenfalls in die verschiedenen Winkel gedreht. Bei der Berechnung unterscheidet sich die beiden Geometrien kaum, da die beiden Matrizen ähnlich berechnet werden können. Die Zeilen in den Matrizen unterscheiden sich nur durch ihre Anordnung. Da unsere Projektionsalgorithmen beliebige Gleichungssysteme lösen können, können beliebige Scanner-Geometrien verwendet werden. Wir konnten also problemlos Bilder, aus unserem Datensatz und dem eines Fächerscanners, rekonstruieren.

3.2 Diskretisierung des Models

Da ein Computer nur diskrete Daten berechnen kann, muss das Bild in Pixel aufgeteilt werden. Die Querschnitte werden also in lauter gleich große Quadrate aufgeteilt wobei jedes mit der zugehörigen Absorbtionskonstante multipliziert wird. Mathemtisch können wir den Querschnitt somit als Superposition

$$f(\cdot,\cdot) = \sum_{n=-N}^{N} \sum_{m=-N}^{N} f_{nm} \cdot \chi(\cdot - n, \cdot - m)$$

von kleinen Quadraten beschreiben, wobei f_{nm} dem Absorptionskoeffizienten im Quadrat n, m entspricht und $\chi(x_1, x_2)$ die Funktion für ein Quadrat ist. Nimmt man davon die Radontransformation erhält man:

$$R[f](\theta, s) = \sum_{n=-N}^{N} \sum_{m=-N}^{N} f_{nm} \cdot R[\chi(\cdot - n, \cdot - m)](\theta, s)$$

Diese Gleichung kann für jeden Winkel und für jede Verschiebung als folgende Gleichung umgeschrieben werden:

$$(w_1, ..., w_{(2N+1)^2}) \cdot \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{(2N+1)^2} \end{pmatrix} = b_{n,m}$$

Wo der Vektor $\vec{f} = (f_1, \ldots, f_{(2N+1)^2})^T$ die Vektorisierung der Helligkeiten $f_{n,m}$ ist und $w_1, \ldots, w_{(2N+1)^2}$ die entsprechenden Gewichte bezeichnen. Für die verschiedenen Winkel und Verschiebungen kann man das anschreiben als:

$$\mathbf{W} \cdot \vec{f} = \vec{b}$$

Wobei $\mathbf{W}$ die aus $R[\chi(\cdot - n, \cdot - m)]$ erhaltene Matrix ist (jedem Strahl entspricht eine Zeile), $\vec{f}$ den Vektor $(f_1, f_2, ..., f_{(2N+1)^2})$ bezeichnet und $\vec{b}$ der Vektor der erhaltenen Daten für jeden Strahl ist.

3.3 Weight Matrix

Für jeden Röntgenstrahl, aus jeder Richtung $\vec{\theta}$, mit jeder Verschiebung s, kann eine Reihe in einer Matrix hergestellt werden, in dem jede Spalte einem Pixel entspricht. Anders gesagt bedeutet das, dass jedem Pixel die Distanz, die jeder einzelne Röntgenstrahl darin zurücklegt, zugeordnet wird. In dieser Matrix entspricht die Zeilenzahl der Anzahl an Röntgenstrahlen mal die Anzahl an Winkeln, und die Spaltenzahl der Menge an Pixeln im Querschnitt. Diese Matrix bezeichnen wir als die „Gewichtsmatrix“. Dazu haben wir noch einen Vektor $vecf$ erstellt, in der jede Reihe bzw. jedes Element für den Helligkeitswert eines Pixels steht. Wenn man die Gewichtsmatrix $\boldsymbol{W}$ mit dem Vektor $\vec{f}$ multipliziert, erhalten wir ein Gleichungssystem, aus dem letztendlich ein Vektor mit dem Integral über jeden möglichen Röntgenstrahl in einem Querschnitt.

4 Algebraische Rekonstruktion

4.1 Hyperebenen

Eine Hyperebene ist ein Objekt, das einen Raum in zwei Teile teilt. Sie hat eine Dimension weniger als der Raum, den sie teilt. Eine Hyperebene im drei-dreidimensionalen Raum ist eine Ebene. Im Zweidimensionalen ist die Hyperebene eine Gerade. Eine Gleichung mit n Unbekannten lässt sich im n-dimensionalen Raum als Hyperebene darstellen. Die Lösung eines Gleichungssystems ist der Schnittpunkt dieser Ebenen. Findet man diesen, hat man das System gelöst.

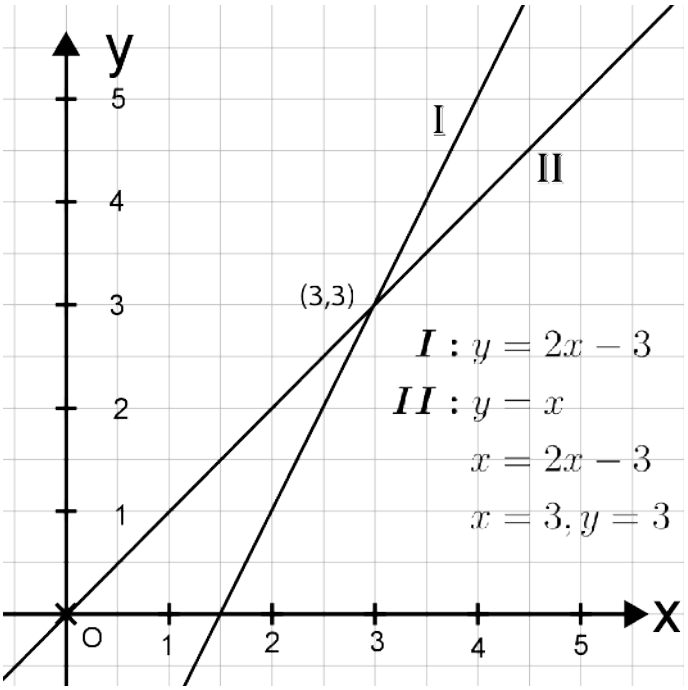


Abbildung 6: Lösung eines LGS mit zwei Unbekannten x und y.

In der Abbildung 6 ist zu erkennen, wie sich ein LGS alleine durch das Schneiden der korrespondierenden Hyperebenen lösen lässt. Das ist der Grund, weshalb Hyperebenen für die Problemstellung so interessant sind. Im nächsten Schritt gilt es den Schnittpunkt in wenigen Schritten möglichst schnell und recheneffizient zu finden.

4.2 Projektion

Die Projektion ist ein Ansatz solche Systeme effizient zu lösen. Orthogonale Projektionen können folgendermaßen berechnet werden.

Satz. Die orthogonale Projektion des Punktes $\vec{x}$ auf die Hyperebene orthogonal zu dem Vektor $\vec{a}$ ist gegeben durch

$$P_{\vec{a}^\perp}(\vec{x}) = \vec{x} - \frac{(\vec{a} \bullet \vec{x})}{|\vec{a}|^2}\,\vec{a}.$$

Beweis. Die Projektion auf die Hyperebene, welche orthogonal zum Vektor $\vec{a}$ ist, ist durch ein Minimierungsproblem gegeben. Setzt man nun in die Zielfunktion $|x-y|^2$ für das Argument y die vermutete Projektion $P_{\vec{a}^\perp}(\vec{x})$ ein, erhält man

$$\left| x - x + \frac{(\vec{a} \bullet x)}{|\vec{a}|^2} \cdot \vec{a} \right|^2 = \frac{(\vec{a} \bullet x)^2}{|\vec{a}|^4} \cdot |\vec{a}|^2 = \frac{(\vec{a} \bullet x)^2}{|\vec{a}|^2}$$

Werten wir die Zielfunktion an einem anderen Punkt der Form $P_{\vec{a}^\perp}(\vec{x}) + \vec{a}^\perp$ der Hyperebene aus, wobei $\vec{a}^\perp$ mit $(\vec{a} \bullet \vec{a}^\perp)$ ein Vektor innerhalb der Hyperebene ist, erhalten wir

$$\begin{aligned}
\left| \frac{(\vec{a} \bullet x)^2}{|\vec{a}|^2} \cdot \vec{a} + \vec{a}^\perp \right|^2 &= \left(\frac{(\vec{a} \bullet x)^2}{|\vec{a}|^2} \cdot \vec{a} + \vec{a}^\perp \right) \bullet \left(\frac{(\vec{a} \bullet x)^2}{|\vec{a}|^2} \cdot \vec{a} + \vec{a}^\perp \right) \\
&= \frac{(\vec{a} \bullet x)^2}{|\vec{a}|^4} \cdot (\vec{a} \bullet \vec{a}) + 2\frac{(\vec{a} \bullet x)}{|\vec{a}|^2} \cdot (\vec{a} \bullet \vec{a}^\perp) + (\vec{a}^\perp \bullet \vec{a}^\perp) \\
&= \frac{(\vec{a} \bullet x)^2}{|\vec{a}|^2} + |\vec{a}|^2 \\
&\geq \frac{(\vec{a} \bullet x)^2}{|\vec{a}|^2}
\end{aligned}$$

Somit ist der Zielfunktionswert an jeder anderen Stelle größer als der Zielfunktionswert am Punkt $P_{\vec{a}^\perp}(\vec{x})$, was die Behauptung zeigt. □

Es gibt zwei Sorten von Projektionsalgorithmen, über die im Folgenden zu lesen sein wird. Die alternierende und die gemittelte Projektion.

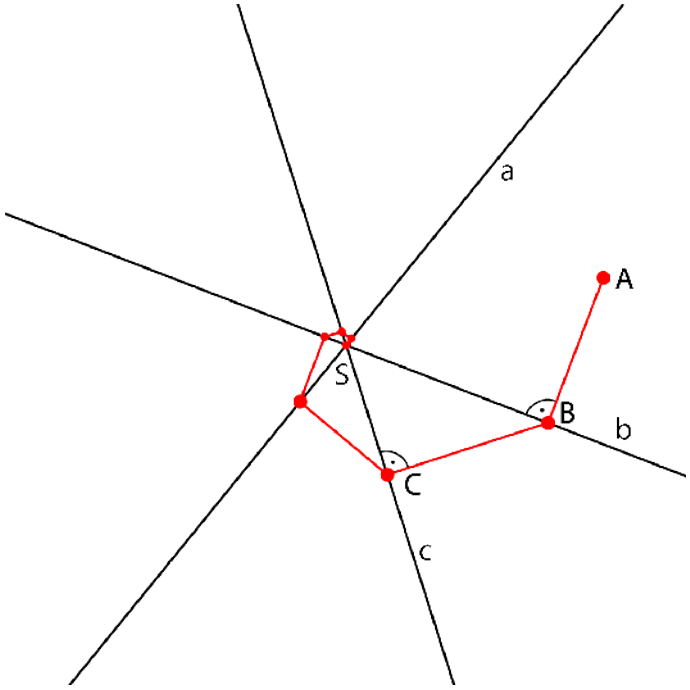


Abbildung 7: Den Schnittpunkt von drei Geraden durch alternierende Projektion finden.

Wie in Abbildung 7 dargestellt, funktioniert der alternierende Projektionalgorithmus wie folgt. Mit einem beliebigen Punkt beginnend projiziert man diesen Punkt auf eine Gerade. In der Abb. Punkt A, der als Projektion auf die Gerade b den Punkt B ergibt. Dieser Punkt wird erneut projiziert. Schneiden sich alle Geraden in einem Punkt S findet das Verfahren immer diesen.

Die gemittelte Projektion projiziert im Unterschied dazu auf jede Gerade und wählt den Durchschnitt all dieser Punkte als neuen Punkt.

4.3 Relaxation

Die Relaxation ist eine Verallgemeinerung der alternierende Projektionalgorithmen zu verstehen. Doch während der neue Punkt der alternierende Projektion immer Element der Gerade ist, ist dies bei der Relaxation nicht der Fall. Während der Punkt B bei der Projektion durch das Schneiden der Gerade b mit der Gerade $\vec{b^{\perp}}t + A$ wobei gilt $t \in \mathbb{R}$, gilt bei der Relaxation folgendes. Sei $\vec{v}$ der Vektor $A\vec{B}_{alt}$ wobei B_{alt} der Schnittpunkt der alternierenden Projektion ist. So gilt für B_{rel}, $B_{rel} = \lambda\vec{v} + A$ und $\lambda \in]0, 2[$. Der Neue Punkt liegt also irgendwo zwischen dem alten und dem über die Gerade

gespiegelten Punkt.

4.4 Konvergenz

Man kann zeigen, dass die Projektionsalgorithmen für Gleichungssysteme mit eindeutiger Lösung konvergieren.

Satz. Wenn das zugehörige Gleichungssystem eine eindeutige Lösung besitzt, konvergiert der alternierende Projektionsalgorithmus zu dieser.

Beweis. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass die Lösung des Gleichungssystems, also der eindeutige Schnittpunkt der Hyperebenen, im Ursprung ist. Ansonsten könnte man alles verschieben. Weiters sei $\vec{w}$ der aus der vorigen Projektion erhaltene Punkt, $\vec{u}$ der durch die Projektion von $\vec{w}$ auf eine andere Hyperebene erhaltene Punkt (da es einen eindeutigen Schnittpunkt gibt können die Ebenen nicht parallel liegen) und $\vec{v}$ der Vektor von $\vec{u}$ nach $\vec{w}$, wie auch in der Abbildung gezeigt. Somit gilt

$$\vec{w} = \vec{u} + \vec{v} \quad \text{mit} \quad \vec{u} \bullet \vec{v} = 0$$

da $\vec{u}$ und $\vec{v}$ orthogonal zuneinander sind. Durch Umformen erhält man

$$|\vec{w}|^2 = |\vec{u}|^2 + |\vec{v}|^2 + 2\,\vec{u} \bullet \vec{v} \qquad \text{und} \qquad |\vec{w}|^2 - |\vec{v}|^2 = |\vec{u}|^2.$$

Wir sehen also, dass sich die Norm nach der Projektion nicht vergrößern kann. Genauer haben wir $|\vec{w}|^2 \geq |\vec{u}|^2$. Gleichheit gilt wenn $\vec{v}$ der Nullvektor ist, also wenn nicht projeziert wird, da zwei Gleichungen äquivalent sind. Da wir einen eindeutigen Schnittpunkt haben, können jedoch nicht alle Gleichungen äquivalent sein, also ist die Norm nach einer vollständigen Iteration auf jeden Fall kleiner. Insgesamt konvergiert der Algorithmus somit zum Ursprung, welcher die Lösung des Gleichungssystems ist. □

Satz. Wenn das zugehörige Gleichungssystem eine eindeutige Lösung besitzt, konvergiert der gemittelte Projektionsalgorithmus zu dieser.

Beweis. Sei x der Ursprüngliche Punkt, $x_1, x_2, ..., x_n$ die Projektionen von x. Nach dem obrigen Beweis sind alle projezierten Punkte mindestends so nahe an der Lösung wie der Ursprüngliche Punkt manche sogar näher. Man kann also eine Mehrdimensionale Kugel mit Zentrum bei der Lösung so zeichnen, dass der Ursprüngliche Punkt auf der Kugel liegt und alle projezierten Punkte auf oder innerhalb des Kugel. Daher gilt

$$|x_1| \leq |x|, |x_2| \leq |x|, ..., |x_n| \leq |x|.$$

Für den Betrag der gemittelten Projektion gilt aufgrund der Dreiecksungleichung

$$|1/n \cdot (x_1 + x_2 + ... + x_n)| \leq 1/n \cdot (|x_1| + |x_2| + ... + |x_n|) \leq |x|.$$

Da nicht alle Hyperebenen identisch sind, ist die Norm von x sogar echt kleiner als die Norm vom Durchschnitt der projezierten Punkte, also

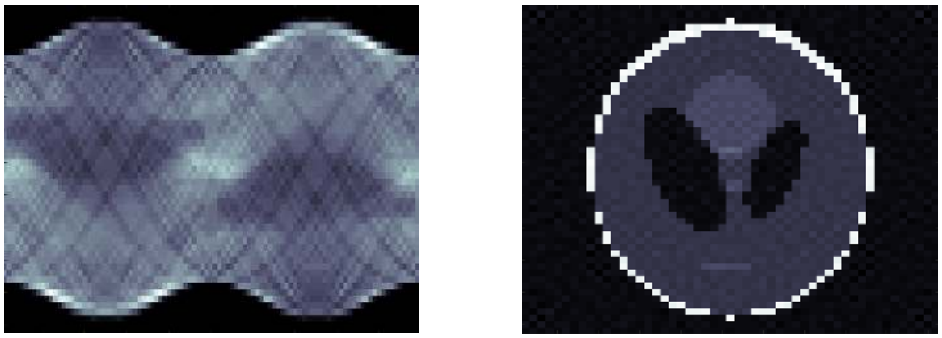
$$|x| > 1/n \cdot |(x_1 + x_2 + ... + x_n)|.$$

Das bedeutet, dass der Punkt nach jeder Projektion näher an den Schnittpunkt der Hyperebenen gelangt und damit zu diesem konvergiert. □

5 Numerische Simulation

5.1 Shepp-Logan.Phantom

Um die Theorie in die Praxis umsetzen zu können, nutzten wir die Entwicklungsumgebung Matlab. Zunächst erstellten wir ein Sinogramm des Shepp-Logan-Phantoms, welches mittlerweile ein Standardtestbild für Probleme dieser Art geworden ist, und Rekonstruierten es mithilfe eines alternierenden Projektionsalgorithmus.

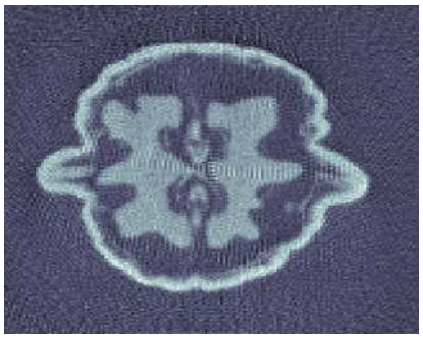


(a) Sinogramm (b) Rekonstruktion

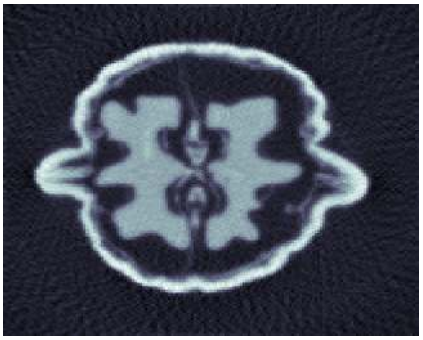
Abbildung 8: Sinogramm und Rekonstruktion

5.2 CT einer Wallnuss

Dann versuchten wir diesen Algorithmus auch auf gemessene CT-Daten einer Walnuss anzuwenden, mit weniger gutem Ergebnis. Da aufgrund der vorhanden Fehler, kein eindeutiger Schnittpunkt der Hyperebenen existiert. Im Gegensatz dazu erzeugte der gemittelte Projektionsalgorithmus deutlich bessere Daten, da er einen guten Näherungswert des überbestimmten Gleichungsystems findet.



(a) Walnuss, alternierende Projektion



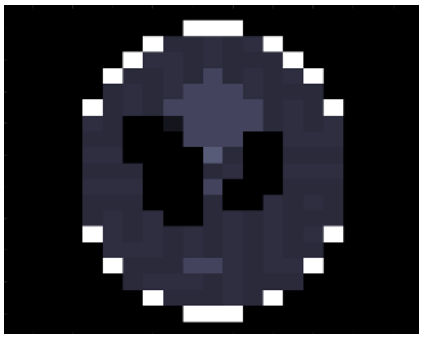
(b) Walnuss, gemittelte Projektion

Abbildung 9: Walnüsse durch unterschiedliche Projektionen Rekonstruiert

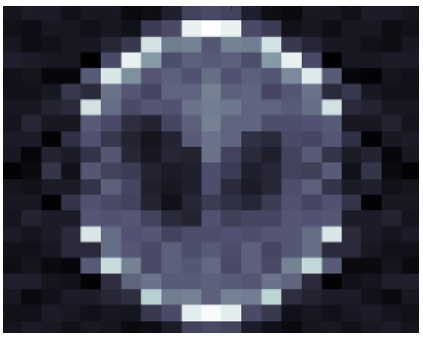
5.3 Vergleich des alternierenden und gemittelten Projektionsalgorithmus

Um dieses Phänomen zu verdeutlichen haben wir das Shepp-Logan-Phantom einmal mit und einmal ohne Fehler mit alternierenden sowie gemittelten Projektionsalgorithmus rekonstruiert.

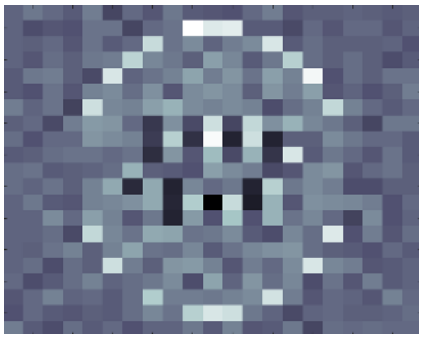
Auf folgenden Graphen ist das Verhältnis der Anzahl der Anwendungen eines Projektionsalgorithmus zu der Genauigkeit des resultierenden Bildes dargestellt. Bei dem Graphen ohne Fehler sieht man, dass der alternierende Projektionsalgorithmus wesentlich schneller konvergiert als der gemittelte Projektionsalgorithmus. Bei der Anwendung auf Fehler behaftete Daten verändert sich das Konvergenzverhalten. Der gemittelte Projektionsalgorithmus scheint zu einer sinnvollen Lösung zu konvergieren, wie man anhand der rekonstruierten Bilder erkennen kann. Der alternierende Projektionsalgorithmus konvergiert nicht, was auch nicht verwunderlich ist, da das Zugehörige lineare Gleichungssystem aufgrund der Fehler in der rechten Seite keine eindeutige Lösung besitzt. Im Graphen lässt sich dieses Phänomen erkennen, da der Rekonstruktionsfehler wächst.



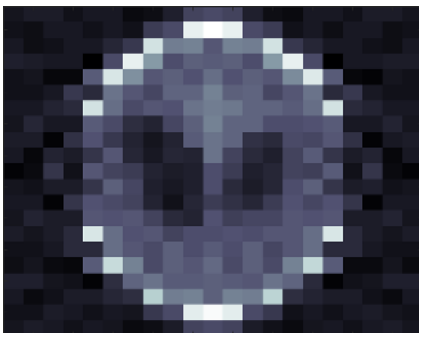
(a) Phantom, alternierend ohne Fehler



(b) Phantom, gemittelt ohne Fehler

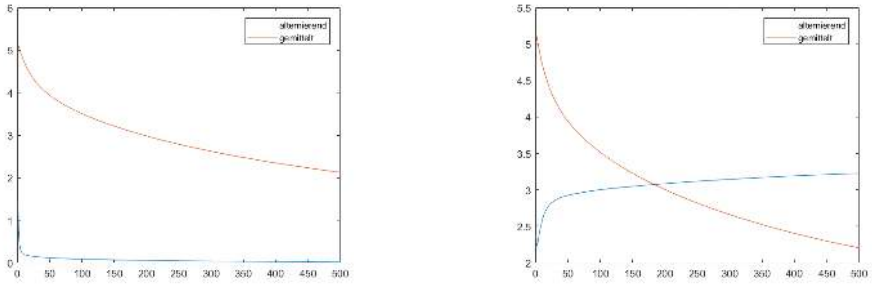


(c) Phantom, alternierend mit Fehler



(d) Phantom, gemittelt mit Fehler

Abbildung 10: Walnüsse durch unterschiedliche Projektionen Rekonstruiert



(a) Vergleich ohne Fehler (b) Vergleich mit Fehler

Abbildung 11: Walnüsse durch unterschiedliche Projektionen Rekonstruiert

5.4 Numerische Umsetzung der Verfahren

Da die Gewichtsmatrix sehr groß werden kann und hauptsächlich aus Nullen besteht, nutzten wir ein Objekt mit dem Namen „Sparse-Matrix“ (engl. „sparse matrix“), welche alle Werte der Matrix, mit Ausnahme von Nullen, in einer Liste mit ihren jeweiligen Indizes speichert. Diese Maßnahme reduziert zwar deutlich den verwendeten Arbeitsspeicher, aber macht es für den Computer umständlicher und zeitintensiver aus dieser Matrix einzelne Zeilen rauszunehmen, was für die Rekonstruktion einen erheblichen Nachteil hat. Deshalb haben wir jede einzelne Zeile der Matrix gesondert in ein Cell-Array gespeichert. Das Cell-Array lässt sich als Ansammlung verschiedener „Container“ visualisieren, welche jeweils mit einer Zeile der Sparse-Matrix gefüllt sind.

Der alternierende Projektionsalgorithmus kann durch eine Optimierung der Reihenfolge der Hyperebenen, auf welche projiziert wird, deutlich optimiert werden. Die größte Verbesserung an der Näherungslösung bringen heuristisch Projektionen zwischen orthogonalen Hyperebenen. In der Matrix für die parallele Scannergeometrie sind die Gleichungen für parallelen Geraden mit einem festen Winkel hintereinander angeordnet. Dadurch kann es passieren, dass die Röntgenstrahlen für aufeinander folgende Gleichungen dieselben Pixel durchqueren und die zugehörigen Hyperebenen sich in einen sehr kleinen Winkel schneiden. Um die zu verhindern durchlaufen wir die Gleichung in einer Reihenfolge, so dass die zugehörigen Röntgenstrahlen keine gemeinsamen Pixel kreuzen.

Psychologie

Die Wirkung von Meditation und ähnlichen Therapien

Maximilian Droschl, Maximilian Gepp, Jan Handler, Sabrina Hörmann, Georg Mandl, Mathias Sommerbauer, Johannes Trinkaus, Jakob Tscheppe, Jana Waxenegger

Betreuer: a.o. Univ.-Prof. Mag. Dr. Stephen Keeling

well[en]done

Inhaltsverzeichnis

1 Einleitung

Meditation ist eine Praxis, bei der durch Konzentrationsübungen tiefe Entspannung erzielt werden soll. Obwohl Meditation in vielen Religionen praktiziert wird und sie oft als spirituell beschrieben ist, existieren wissenschaftliche Belege für ihre Wirkung.

Trotz vieler Pseudowissenschaftler, die sich mit dem Thema Gehirnwellen und Meditation beschäftigen, liegen also einige Ergebnisse in der rigorosen Wissenschaft vor. Meditation und ähnliche Praktiken haben nachweislich einen Einfluss auf die Frequenz der Gehirnströme. So ist die Frequenz während der Meditation ähnlich derer im Schlafzustand. Außerdem wurde der Praktik eine Senkung des Blutdrucks und des Cholesterinspiegels sowie die Förderung des Wohlbefindens zugeschrieben.

Es gibt verschiedene Arten von Meditation:
1) Entspannungsmeditation
2) Einsichtsmeditation: Das Verweilen in kognitiver Dissonanz
3) Tiefe Meditation

Neuronale Oszillationen oder auch Gehirnwellen sind rhythmische und sich wiederholende Muster im zentralen Nervensystem (ZNS). Solche Aktivitäten geschehen im Inneren von Neuronen und zwischen Neuronen. In einzelnen Neuronen können die Impulse als Membran- oder als Aktionspotenzial auftreten. Die Interaktion zwischen einer großen Anzahl an Nervenzellen kann als Oszillation in einem Elektroenzephalogramm dargestellt werden.

Auch spielt in unserem Projekt der Unterschied zwischen linker und rechter Gehirnhälfte eine erhebliche Rolle wie wir im weiteren Verlauf sehen werden.

In unserem Projekt beschäftigen wir uns mit der Simulation von Wellen und den mathematischen Parametern und Werkzeugen, um deren verschiedene Frequenzen zu beschreiben. So hat ein Teil unserer Gruppe die Reizweiterleitung von rechter zu linker Gehirnhemisphäre behandelt, ein weiterer Teil hielt sich mit harmonischen Klängen auf und wieder ein anderer Zweig unserer Gruppe beschäftigte sich mit Neuroplastizität, was später noch genauer erörtert wird. Der Prozess des Aufwachens wurde ebenfalls in Form von Änderungen von Frequenz beschrieben.

2 Basis der Modellierung

Wir untersuchen wellenartige Phänomene im Gehirn mit Betonung auf Unterschieden zwischen meditativen und nicht meditativen Zuständen. Die Modellierung wird vom mechanischen Paradigma inspiriert, in dem die Wechselwirkungen in einem System, in dem Massen mit Federn verbunden sind, leicht verständlich sind. Die Massen stellen Pakete von Ionen dar, und die elastische Energie in den Federn stellt die elektrische Energie dar.

2.1 Einfachste Masse-Feder System

Das einfachste mechanische System sieht man ab Seite 73 im Skriptum,
https://imsc.uni-graz.at/keeling/skripten/modein.pdf
Die Auslenkung u einer Masse m wird mit dem Newtonschen Gesetz (Masse $\times$ Beschleunigung = Summe der wirkenden Kräfte) modelliert,

$$mu''(t) = -ku(t) + f, \quad t \geq 0, \quad u(0) = u_0, \quad u'(0) = u_1.$$

Hier ist $u''(t)$ die Beschleunigung. Die elastischen Kräfte der Feder werden durch die möglichst einfache lineare Abhängigkeit von der Auslenkung modelliert,

$$f^{\text{elas}} = -ku$$

wobei $k > 0$ die Feder-Konstante ist. Die externen Kräfte, wie z.B. Schwerkraft, werden mit f bezeichnet. Die rechte Seite der gewöhnlichen Differentialgleichung ist die Summe der wirkenden Kräfte. Die Anfangsauslenkung ist u_0 und die Anfangsgeschwindigkeit ist u_1. Ohne Reibungskräfte weißt die Lösung ewige ungedämpfte Schwingungen auf. Reibungskräfte in den Federn werden durch die möglichst einfache lineare Abhängigkeit von der Geschwindigkeit modelliert,

$$f^{\text{reib}} = -ku'(t)$$

wobei $c > 0$ der Dämpfungsparameter ist. Mit dieser zusätzlichen Kraft ist das Modell der Auslenkung gegeben durch

$$mu''(t) + cu'(t) = -ku(t) + f, \quad t \geq 0, \quad u(0) = u_0, \quad u'(0) = u_1.$$

2.2 Umformen und Programmieren

Da die in Matlab eingebauten Funktionen zur Lösung einer gewöhnlichen Differentialgleichung von der folgenden Form ausgehen,

$$\boldsymbol{y}'(t) = \boldsymbol{f}(t, \boldsymbol{y}(t)), \quad t \geq 0, \quad \boldsymbol{y}(0) = \boldsymbol{y}_0$$

muss unsere Differentialgleichung zweiter Ordnung in erste Ordnung umgeformt werden. Dies erfolgt folgendermaßen:

$$\begin{bmatrix} 1 & 0 \\ 0 & m \end{bmatrix} \begin{bmatrix} u(t) \\ u'(t) \end{bmatrix}' = \begin{bmatrix} 0 & 1 \\ -k & -c \end{bmatrix} \begin{bmatrix} u(t) \\ u'(t) \end{bmatrix} + \begin{bmatrix} 0 \\ f \end{bmatrix}, \quad \begin{bmatrix} u(0) \\ u'(0) \end{bmatrix} = \begin{bmatrix} u_0 \\ u_1 \end{bmatrix}$$

oder

$$\boldsymbol{U}'(t) = A\boldsymbol{U}(t) + \boldsymbol{F}, \quad \boldsymbol{U}(0) = \boldsymbol{U}_0$$

wobei

$$\boldsymbol{U}(t) = \begin{bmatrix} u(t) \\ u'(t) \end{bmatrix}, \quad A = \begin{bmatrix} 1 & 0 \\ 0 & m \end{bmatrix}^{-1} \begin{bmatrix} 0 & 1 \\ -k & -c \end{bmatrix}, \quad \boldsymbol{F} = \begin{bmatrix} 1 & 0 \\ 0 & m \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ f \end{bmatrix}, \quad \boldsymbol{U}_0 = \begin{bmatrix} u_0 \\ u_1 \end{bmatrix}$$

Ein Matlab-Code zur Lösung dieses Problems ist:

```
function MasseFeder0
    global A F

    m = 1;
    k = 1;
    c = 0.1;
    f = 0.0;

    M = [1,0;0,m];
    K = [0,1;-k,0];
    C = [0,0;0,-c];
    A = M\(K+C);
    F = M\[0;f];

    U0 = [0.5;0];
```

```
    [t,U] = ode45(@Kraefte0,[0,100],U0);
    plot(t,U(:,1))
end
function dU = Kraefte0(t,U)
    global A F

    dU = A*U + F;
end
```

Hier wird die in Matlab eingebaute Funktion `ode45` aufgerufen, die wiederum die eigene Funktion `Kraefte0` für die rechte Seite der Differentialgleichung aufruft. Da diese Funktion nicht immer wie erwünscht eingestellt werden kann, ist der folgende Matlab-Code ohne `ode45` geschrieben worden:

```
    nt = 1000; dt = 0.1; th = 1.0;

    m = 1;
    k = 1;
    c = 0.1;
    f = 0.0;

    M = [1,0;0,m];
    K = [0,1;-k,0];
    C = [0,0;0,-c];
    A = M\(K+C);
    F = M\[0;f];

    U  = [0.5;0];
    uv = U(1);
    for i=1:nt
        U = (I2 - th*dt*A) \ (U + dt*((1-th)*A*U + F));
        uv = [uv,U(1)];
        plot(uv)
    end
```

2.3 Zeitschritte

Die Zeitschritte in der `for`-Schleife basieren auf folgenden Überlegungen. Die zeitliche Ableitung $\boldsymbol{U}'$ in der Differentialgleichung $\boldsymbol{U}' = A\boldsymbol{U} + \boldsymbol{F}$ lässt sich mit einem Differenzenquotienten approximieren,

$$\frac{\boldsymbol{U}^{k+1} - \boldsymbol{U}^k}{\Delta t} = A\boldsymbol{U}^{\hat{k}=?} + \boldsymbol{F}$$

wobei die berechneten Werte $\boldsymbol{U}^k \approx \boldsymbol{U}(t_k)$ die exakte Lösung $\boldsymbol{U}(t_k)$ an der Stelle t_k approximieren, und der zeitliche Bereich $[0, n_t\Delta t]$ wird mit dem Gitter $t_k = k\Delta t$, $k = 0, \ldots, n_t$ approximiert. Mit $\hat{k} =?$ bleibt noch die Frage offen, wie $\boldsymbol{U}$ auf der rechten Seite ausgewertet werden soll. Mit $\hat{k} = k$ ist die Methode *explizit*, weil die unbekannte $\boldsymbol{U}^{k+1}$ explizit bezüglich der zur Zeit t^k bekannten Information so dargestellt werden kann:

$$\boldsymbol{U}^{k+1} = \boldsymbol{U}^k + \Delta t \cdot (A\boldsymbol{U}^k + \boldsymbol{F})$$

Mit $\hat{k} = k{+}1$ ist die Methode *implizit*, weil die unbekannte $\boldsymbol{U}^{k+1}$ nur implizit in dieser Gleichung definiert ist, d.h. $\boldsymbol{U}^{k+1}$ steht auf beiden Seiten. Man muss das folgende Gleichungssystem lösen, um $\boldsymbol{U}^{k+1}$ zu bestimmen:

$$[I - \Delta t \cdot A]\boldsymbol{U}^{k+1} = \boldsymbol{U}^k + \Delta t \cdot \boldsymbol{F}$$

Mit $\boldsymbol{U}^{\hat{k}} = \theta \boldsymbol{U}^{k+1} + (1-\theta)\boldsymbol{U}^k$ ist die Methode eine Kombination der rein expliziten Methode und der rein impliziten Methode,

$$[I - \theta \Delta t \cdot A]\boldsymbol{U}^{k+1} = \boldsymbol{U}^k + \Delta t \cdot ((1-\theta)A\boldsymbol{U}^k + \boldsymbol{F}).$$

Dies ist die Formulierung im obigen Matlab-Code ohne `ode45`. Für $\theta \in [0.5, 1.0]$ ist die Methode stabiler, und zwar dissipativ für $\theta \approx 1$ und dispersiv für $\theta \approx \frac{1}{2}$. Eine Methode ist dissipativ, wenn Komponenten der Lösung mit hohen Frequenzen gedämpft werden. Eine Methode ist dispersiv, wenn Komponenten der Lösung mit unterschiedlichen Wellenlängen mit unterschiedlichen Geschwindigkeiten fortgepflanzt werden. Für $\theta \in [0.0, 0.5]$ hängt die Stabilität der Methode von einem ausreichend kleinen Δt ab. Weitere einführende Details über die Numerik für gewöhnliche Differentialgleichungen können im folgenden Skriptum gefunden werden:

https://imsc.uni-graz.at/keeling/skripten/numerik.pdf

2.4 Zwei Massen

Seien nun *zwei* hängende Massen gegeben, wobei diese die Gleichgewichtspositionen (d.h. ohne Störungen und ohne externe Kräfte) u_1^{e} bzw. u_2^{e} haben. $u_1(t)$ bzw. $u_2(t)$ seien die Auslenkungen der zwei Massen zur Zeit t. Analog zum Modell mit einer Masse ergibt sich das folgende System von gewöhnlichen Differentialgleichungen für das System mit einer zusätlichen Masse,

$$\begin{array}{rcll} m_1 u_1''(t) + c_1 u_1'(t) & = & -k_1 u_1(t) \quad -k_2(u_1(t) - u_2(t)) + f_1, & t \geq 0, \quad u_1(0) = u_1^0, \quad u_1'(0) = u_1^1 \\ m_2 u_2''(t) + c_2 u_2'(t) & = & \quad -k_2(u_2(t) - u_1(t)) + f_2, & t \geq 0, \quad u_2(0) = u_2^0, \quad u_2'(0) = u_2^1 \end{array}$$

Hier sind m_1 und m_2 die Masse der zwei Massen, c_1 und c_2 die jeweiligen Dämpfungsparameter, f_1 und f_2 die jeweiligen äußeren Kräfte und k_1 und k_2 sind die Feder-Konstanten der zwei Federn. Mit der Lösung dieses Systems sind die Positionen der Massen gegeben durch $u_1^{\mathrm{e}} + u_1(t)$ bzw. $u_2^{\mathrm{e}} + u_2(t)$. Dieses System lässt sich in erste Ordnung umschreiben,

$$\boldsymbol{U}'(t) = A\boldsymbol{U}(t) + \boldsymbol{F}, \quad \boldsymbol{U}(0) = \boldsymbol{U}_0$$

wobei

$$\boldsymbol{U}(t) = \begin{bmatrix} u_1(t) \\ u_2(t) \\ u_1'(t) \\ u_2'(t) \end{bmatrix}, \quad \boldsymbol{U}_0 = \begin{bmatrix} u_1^0 \\ u_2^0 \\ u_1^1 \\ u_2^1 \end{bmatrix}, \quad A = M^{-1} \begin{bmatrix} Z & I \\ K & C \end{bmatrix}, \quad M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & m_1 & 0 \\ 0 & 0 & 0 & m_2 \end{bmatrix},$$

$$\boldsymbol{F} = M^{-1} \begin{bmatrix} 0 \\ 0 \\ f_1 \\ f_2 \end{bmatrix}, \quad Z = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} -c_1 & 0 \\ 0 & -c_2 \end{bmatrix}, \quad K = \begin{bmatrix} -(k_1 + k_2) & k_2 \\ k_1 & -k_2 \end{bmatrix}$$

Nun seien zwei Massen gegeben, die über eine Feder miteinander verbunden sind, aber auch jeweils über eine Feder mit einer Wand befestigt sind. u_1^{e} bzw. u_2^{e} seien wieder die Gleichgewichtspositionen der zwei Massen. $u_1(t)$ bzw. $u_2(t)$ seien die Auslenkungen der zwei Massen zur Zeit t. Diese Auslenkungen werden mit dem folgenden System von gewöhnlichen Differentialgleichungen bestimmt:

$$\begin{array}{rcll} m_1 u_1''(t) + c_1 u_1'(t) & = & -k_1 u_1(t) \quad -k_2(u_1(t) - u_2(t)) + f_1, & t \geq 0, \quad u_1(0) = u_1^0, \quad u_1'(0) = u_1^1 \\ m_2 u_2''(t) + c_2 u_2'(t) & = & -k_3 u_2(t) \quad -k_2(u_2(t) - u_1(t)) + f_2, & t \geq 0, \quad u_2(0) = u_2^0, \quad u_2'(0) = u_2^1 \end{array}$$

Hier sind m_1 und m_2 die Masse der zwei Massen, c_1 und c_2 die jeweiligen Dämpfungsparameter, f_1 und f_2 die jeweiligen äußeren Kräfte und k_1, k_2 und k_3 sind die Feder-Konstanten der drei Federn. Mit der Lösung dieses Systems sind die Positionen der Massen gegeben durch

$u_1^{\mathrm{e}} + u_1(t)$ bzw. $u_2^{\mathrm{e}} + u_2(t)$. Dieses System lässt sich in eine Differentialgleichung erster Ordnung umschreiben,

$$\boldsymbol{U}'(t) = A\boldsymbol{U}(t) + \boldsymbol{F}, \quad \boldsymbol{U}(0) = \boldsymbol{U}_0$$

wobei die meisten Objekte für das vorherige Beispiel schon definiert worden sind. Hier gilt aber:

$$K = \begin{bmatrix} -(k_1 + k_2) & k_2 \\ k_2 & -(k_2 + k_3) \end{bmatrix}$$

Nun seien zwei Massen gegeben, die über eine Feder miteinander verbunden sind. Sonst sind sie an keiner Wand befestigt, d.h. die Enden sind frei. u_1^{e} bzw. u_2^{e} seien wieder die Gleichgewichtspositionen der zwei Massen. $u_1(t)$ bzw. $u_2(t)$ seien die Auslenkungen der zwei Massen zur Zeit t. Diese Auslenkungen werden mit dem folgenden System von gewöhnlichen Differentialgleichungen bestimmt:

$$\begin{array}{rcl} m_1 u_1''(t) + c_1 u_1'(t) & = & -k_1(u_1(t) - u_2(t)) + f_1, \quad t \geq 0, \quad u_1(0) = u_1^0, \quad u_1'(0) = u_1^1 \\ m_2 u_2''(t) + c_2 u_2'(t) & = & -k_1(u_2(t) - u_1(t)) + f_2, \quad t \geq 0, \quad u_2(0) = u_2^0, \quad u_2'(0) = u_2^1 \end{array}$$

Hier sind m_1 und m_2 die Masse der zwei Massen, c_1 und c_2 die jeweiligen Dämpfungsparameter, f_1 und f_2 die jeweiligen äußeren Kräfte und k_1 ist die Feder-Konstante der einen Feder. Mit der Lösung dieses Systems sind die Positionen der Massen gegeben durch $u_1^{\mathrm{e}} + u_1(t)$ bzw. $u_2^{\mathrm{e}} + u_2(t)$. Dieses System lässt sich in erste Ordnung umschreiben,

$$\boldsymbol{U}'(t) = A\boldsymbol{U}(t) + \boldsymbol{F}, \quad \boldsymbol{U}(0) = \boldsymbol{U}_0$$

wobei die meisten Objekte schon definiert worden sind. Hier gilt aber:

$$K = \begin{bmatrix} -k_1 & k_1 \\ k_2 & -k_2 \end{bmatrix}$$

2.5 Mehrere Massen

Nun seien $n \geq 3$ Massen gegeben, jeweils mit Massen $m_i = m$ (d.h. zur Vereinfachung sind alle gleich), Auslenkungen $u_i(t)$, Dämpfungskonstanten c_i und externen Kräften f_i, $i = 1, \ldots, n$, und

$$\boldsymbol{u}(t) = \begin{bmatrix} u_1(t) \\ \vdots \\ u_n(t) \end{bmatrix} \quad \boldsymbol{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} \quad \boldsymbol{f} = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}.$$

Seien wieder u_i^{e}, $i = 1, \ldots, n$ die Gleichgewichtspositionen der n Massen. Diese Auslenkungen werden mit dem folgenden System von gewöhnlichen Differentialgleichungen bestimmt:

$$\boldsymbol{U}'(t) = A\boldsymbol{U}(t) + \boldsymbol{F}, \quad t \geq 0, \quad \boldsymbol{U}(0) = \boldsymbol{U}_0$$

wobei

$$\boldsymbol{U}(t) = \begin{bmatrix} \boldsymbol{u}(t) \\ \boldsymbol{u}'(t) \end{bmatrix}, \quad A = \begin{bmatrix} Z & I \\ K & C \end{bmatrix}, \quad m\boldsymbol{F} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{f} \end{bmatrix}$$

$$Z = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \quad I = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix} \quad mC = \mathrm{diag}(\boldsymbol{c}) \quad \text{und} \quad mK = -D^\top \mathrm{diag}(\boldsymbol{k}) D.$$

Die Anzahl der Federn hängt von der Konfiguration ab:

Fall 1: Die erste und die letzte Masse sind frei, also mit keiner Wand verbunden. Es gibt dann $n-1$ Federn und

$$\boldsymbol{k} = \begin{bmatrix} k_1 \\ \vdots \\ k_{n-1} \end{bmatrix} \quad D = \begin{bmatrix} -1 & +1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & -1 & +1 & 0 \\ 0 & \cdots & 0 & -1 & +1 \end{bmatrix} \quad (n-1 \text{ Zeilen, } n \text{ Spalten}).$$

Fall 2: Die erste und die letzte Masse sind jeweils über eine Feder an einer Wand befestigt. Es gibt dann $n+1$ Federn und

$$\boldsymbol{k} = \begin{bmatrix} k_1 \\ \vdots \\ k_{n+1} \end{bmatrix} \quad D = \begin{bmatrix} +1 & 0 & \cdots & 0 \\ -1 & +1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ & & -1 & +1 \\ 0 & \cdots & 0 & -1 \end{bmatrix} \quad (n+1 \text{ Zeilen, } n \text{ Spalten}).$$

Fall 3: Die erste und die letzte Masse sind über eine Feder verbunden. Es gibt dann n Federn und

$$\boldsymbol{k} = \begin{bmatrix} k_1 \\ \vdots \\ k_n \end{bmatrix} \quad D = \begin{bmatrix} +1 & 0 & \cdots & -1 \\ -1 & +1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & -1 & +1 \end{bmatrix} \quad (n \text{ Zeilen, } n \text{ Spalten}).$$

2.6 Programmieren

Ein Matlab-Code zur Lösung des obigen Systems ist:

```
    h1 = figure(1);
    close(h1);
    h1 = figure(1);
    set(h1,'Position',[100 10 1000 500]);

    m = 1;

    beispiel = 1;
    switch beispiel
        case 1
% haengende Massen
            nx = 10; nt = 1000; dt = 0.1; th = 1.0;
            ue = (1:nx)';
            u0 = zeros(nx,1);
            u1 = zeros(nx,1);
            f = 1;
            D = spdiags(ones(nx,1), 0,nx,nx) ...
              - spdiags(ones(nx,1),-1,nx,nx);
            k = ones(nx,1);
            K = -D'*spdiags(k,0,nx,nx)*D;
            c = zeros(nx,1);
        case 2
```

```
% auf zwei Seiten befestigte Massen
            nx = 100; nt = 1000; dt = 0.1; th = 0.75;
            ue = (1:nx)';
%           u0 = 0.1*randn(nx,1);
            u0 = zeros(nx,1);
            u0(round(3*nx/8):round(5*nx/8)) = 1;
            u1 = zeros(nx,1);
            f = 0.0;
            D = spdiags(ones(nx,1), 0,nx+1,nx) ...
              - spdiags(ones(nx,1),-1,nx+1,nx);
            k = ones(nx+1,1);
            K = -D'*spdiags(k,0,nx+1,nx+1)*D;
            c = zeros(nx,1);
        case 3
% Massen auf einem Ring
            nx = 20; nt = 1000; dt = 0.1; th = 0.75;
            ue = (1:nx)';
            u0 = 0.2*randn(nx,1);
            u1 = zeros(nx,1);
            f = 0.0;
            D = spdiags(ones(nx,1), 0,nx,nx) ...
              - spdiags(ones(nx,1),-1,nx,nx);
            D(1,nx) = -1;
            k = ones(nx,1);
            K = -D'*spdiags(k,0,nx,nx)*D;
            c = zeros(nx,1);
        case 4
% freie Massen
            nx = 100; nt = 1000; dt = 0.1; th = 0.75;
            ue = (1:nx)';
            u0 = zeros(nx,1);
            u0(round(3*nx/8):round(5*nx/8)) = 1;
            u1 = zeros(nx,1);
            f = 0.0;
            D = spdiags(ones(nx-1,1),1,nx-1,nx) ...
              - spdiags(ones(nx-1,1),0,nx-1,nx);
            k = ones(nx-1,1); k=2*k;
            K = -D'*spdiags(k,0,nx-1,nx-1)*D;
            c = zeros(nx,1); c(1) = sqrt(k(1)); c(nx) = sqrt(k(nx-1));
    end

    Z  = sparse(nx,nx);
    I  = speye(nx);
    C  = spdiags(-c,0,nx,nx);
    A  = [Z,I;K/m,C/m];
    I2 = speye(2*nx);
    F  = (f/m)*[zeros(nx,1);ones(nx,1)];

    uv = ue+u0;
    U  = [u0;u1];

    for i=1:nt
```

```
    U = (I2 - th*dt*A) \ (U + dt*((1-th)*A*U + F));

    switch beispiel
        case 1
            u  = U(1:nx) + ue;
            uv = [uv,u];
            subplot(1,2,1)
            plot(0:i,uv')
            axis tight
            subplot(1,2,2)
            plot(0,-u,'*')
            axis([-1 1 -max(uv(:)) 0])
            axis off
            drawnow;
        case 2
            u  = U(1:nx);
            uv = [uv,u+ue];
            subplot(1,2,1)
            plot(0:i,uv')
            axis tight
            subplot(1,2,2)
            plot(0:nx+1,[0;u;0])
            axis([0 nx+1 -1.5 1.5])
            axis off
            drawnow;
        case 3
            u  = U(1:nx) + ue;
            uv = [uv,u];
            subplot(1,2,1)
            plot(0:i,uv')
            axis tight
            subplot(1,2,2)
            v = [u;u(1)]; v = 2*pi*v/nx;
            plot(cos(v),sin(v),'-*')
            axis([-1.5 1.5 -1.5 1.5])
            pbaspect([1 1 1]);
            axis off
            drawnow;
        case 4
            u = U(1:nx);
            uv = [uv,u+ue];
            subplot(1,2,1)
            plot(0:i,uv')
            axis tight
            subplot(1,2,2)
            plot(1:nx,u)
            axis([0 nx+1 -2 2])
            axis off
            drawnow;
    end

end
```

2.7 Grafische Darstellungen

Anbei befinden sich die grafischen Darstellungen der (einfacheren) Modelle.

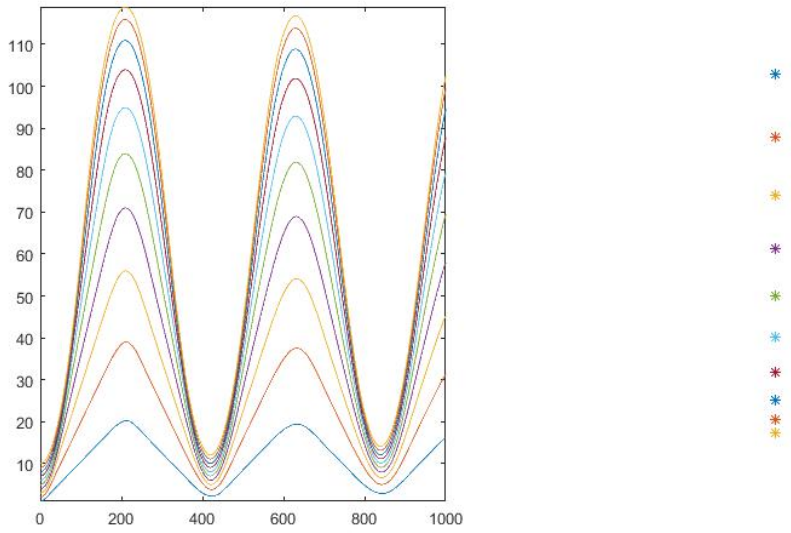


Abbildung 1: hängende Massen 1D

Die Graphen im linken Bereich der obigen Abbildung visualisieren die Auslenkungen der einzelnen Massen. Auf der rechten Seite sind die hängenden Massen als Stern dargestellt.

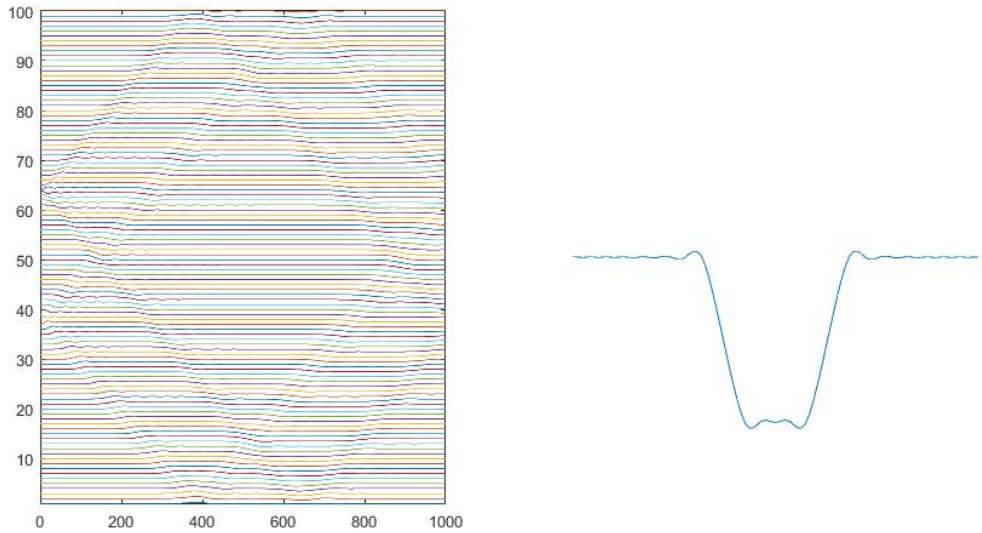


Abbildung 2: auf zwei Seiten befestigte Massen 1D

Im linken Bereich der obigen Abbildung lässt sich gut der zeitliche Verlauf der Auslenkungen der Massen erkennen. An den Rändern werden die Wellen reflektiert.

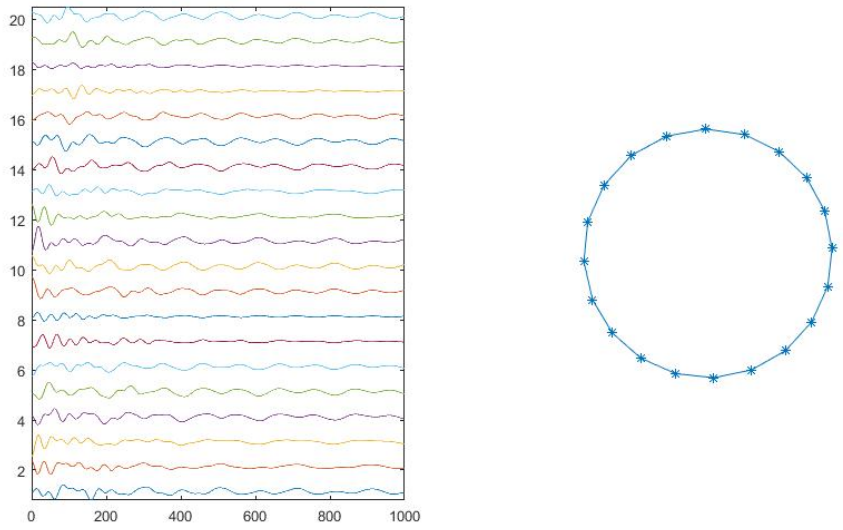


Abbildung 3: Massen auf einem Kreis 1D

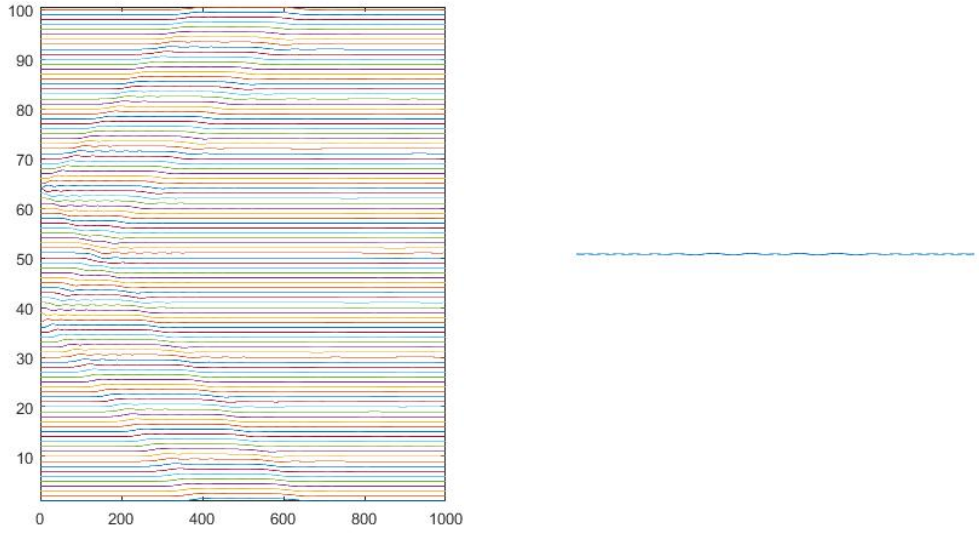


Abbildung 4: freie Massen 1D

In der obigen Abbildung ist erkennbar, dass das Wellenmuster am Rand versiegt, da die Massen am Rand frei vorliegen.

Nun folgen grafische Darstellungen der bereits genannten Fällen in zwei Dimensionen.

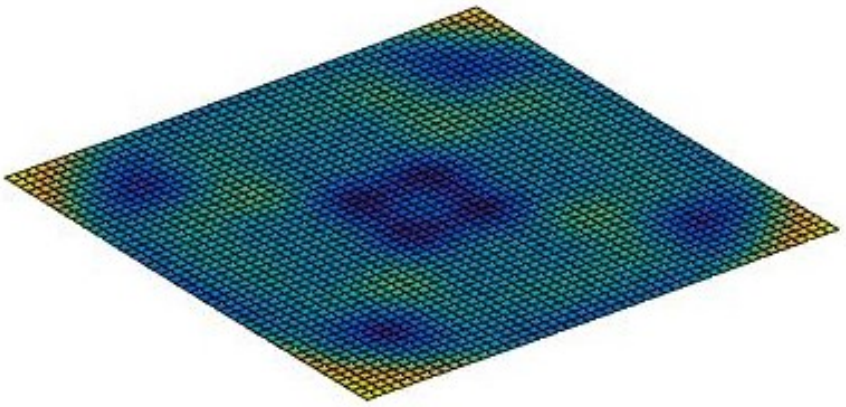


Abbildung 5: freie Massen 2D

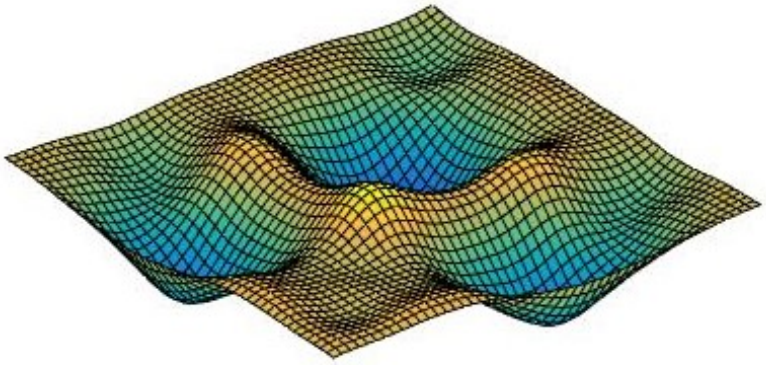


Abbildung 6: am Rand befestigte Massen 2D

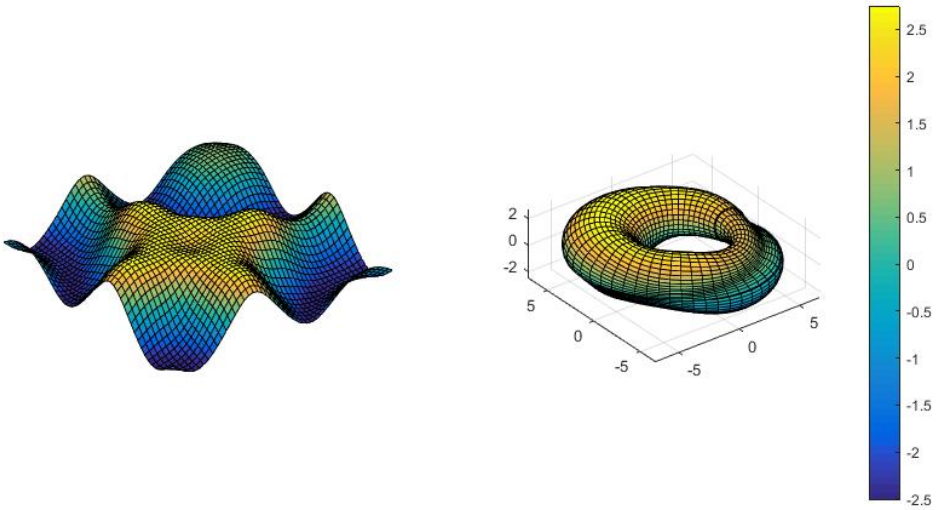


Abbildung 7: Massen auf einem Torus 3D

In der obigen Darstellung wurde die quadratische Fläche auf einen Torus abgebildet. Der Torus ist der erste Schritt, um die Wellen auf einem Gehirn abzubilden.

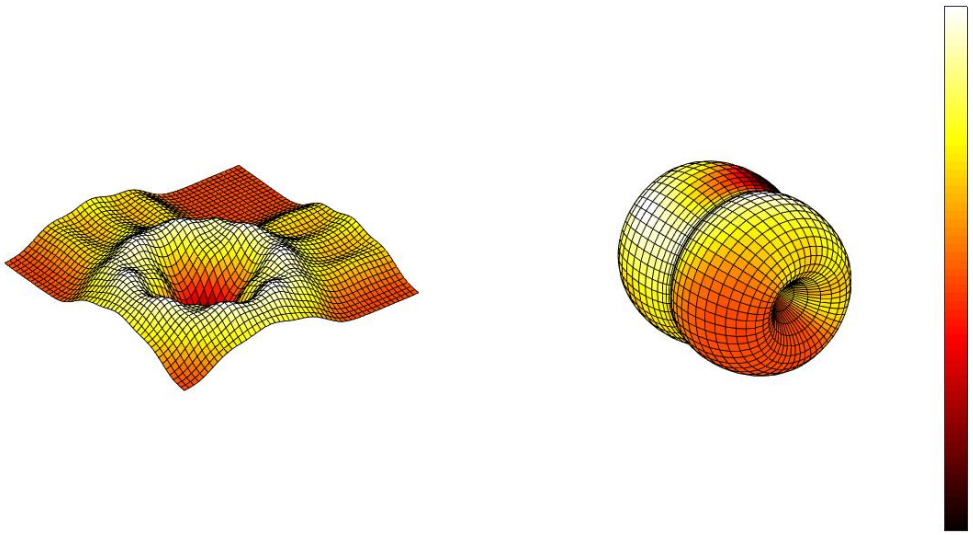


Abbildung 8: Gehirn 3D

In dieser Abbildung setzt sich das Gehirn aus zwei Tori, die miteinander verschmolzen sind, zusammen. Hierbei bildet die links dargestellte Fläche die Oberfläche des 3D Gehirns. Je ein Torus steht für eine Gehirnhälfte.

2.8 Partielle Differentialgleichung in einer räumlichen Dimension

Wir sind von einem diskreten System von Massen und Federn ausgegangen. Die Basis des entsprechenden Systems von gewöhnlichen Differentialgleichungen ist in der folgenden partiellen Differentialgleichung zu sehen:

$$m(x)u_{tt}(x,t) + c(x)u_t(x,t) = (\kappa(x)u_x(x,t))_x + f(x)$$

Hier bezeichnet ein tiefgestellter Buchstabe eine Ableitung nach der entsprechenden Variable, z.B. u_t bezeichnet die Ableitung nach t, während x fixiert bleibt. Die Beziehung zwischen der partiellen Differentialgleichung und der gewöhnlichen Differentialgleichung ergibt sich durch räumliche Diskretisierung. Dafür seien $x_i = i \cdot \Delta x$, $i = 0, \ldots, n$, die Knoten eines Gitters im räumlichen Gebiet, in dem die Auslenkungen stattfinden. Dann wird der Term $(\kappa(x)u_x(x,t))_x$ durch Differenzenquotienten so approximiert,

$$\begin{aligned}
(\kappa(x)u_x(x,t))_x &\approx \frac{\kappa(x+\frac{\Delta x}{2})u_x(x+\frac{\Delta x}{2},t) - \kappa(x-\frac{\Delta x}{2})u_x(x-\frac{\Delta x}{2},t)}{\Delta x} \\
&\approx \frac{\kappa(x+\frac{\Delta x}{2})\dfrac{u(x+\Delta x,t)-u(x,t)}{\Delta x} - \kappa(x-\frac{\Delta x}{2})\dfrac{u(x,t)-u(x-\Delta x,t)}{\Delta x}}{\Delta x} \\
&= \left[\frac{\kappa(x+\frac{\Delta x}{2})}{\Delta x^2}\right] u(x+\Delta x,t) - 2\left[\frac{\kappa(x+\frac{\Delta x}{2}) + \kappa(x-\frac{\Delta x}{2})}{\Delta x^2}\right] u(x,t) + \left[\frac{\kappa(x-\frac{\Delta x}{2})}{\Delta x^2}\right] u(x-\Delta x,t).
\end{aligned}$$

Also ist die Beziehung zwischen dem Koeffizienten κ in der partiellen Differentialgleichung und dem Koeffizienten k in der gewöhnlichen Differentialgleichung gegeben durch

$$k = \frac{\kappa}{\Delta x^2}.$$

2.9 Zwei räumliche Dimensionen und die partielle Differentialgleichung

Die bisherigen Systeme von Massen und Federn in einer ein-dimensionalen Kette können zu einem zwei-dimensionalen Netzwerk ergänzt werden. Hier sind zwei Massen in (x_i, y_j) und (x_{i+1}, y_j) mit Auslenkungen $u_{i,j}(t)$ und $u_{i+1,j}(t)$ zur Zeit t durch eine Feder mit der Feder-Konstante $k^x_{i+\frac{1}{2},j}$ verbunden. Analog sind zwei Massen in (x_i, y_j) und (x_i, y_{j+1}) mit Auslenkungen $u_{i,j}(t)$ und $u_{i,j+1}(t)$ zur Zeit t durch eine Feder mit der Feder-Konstante $k^x_{i,j+\frac{1}{2}}$ verbunden. Das entsprechende System der gewöhnlichen Differentialgleichungen ist

$$\boldsymbol{U}'(t) = A\boldsymbol{U}(t) + \boldsymbol{F}, \quad t \geq 0, \quad \boldsymbol{U}(0) = \boldsymbol{U}_0,$$

wobei

$$\boldsymbol{U}(t) = \begin{bmatrix} \boldsymbol{u}(t) \\ \boldsymbol{u}'(t) \end{bmatrix}, \quad A = \begin{bmatrix} Z & I \\ K & C \end{bmatrix}, \quad m\boldsymbol{F} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{f} \end{bmatrix}$$

$$\text{alle } n_x n_y \times n_x n_y \quad \begin{cases} Z = \operatorname{diag}(\boldsymbol{0}), \quad I = \operatorname{diag}(\boldsymbol{1}), \quad mC = \operatorname{diag}(\boldsymbol{c}), \\ \\ mK = -D_x^\top \operatorname{diag}(\boldsymbol{k}^x) D_x - D_y^\top \operatorname{diag}(\boldsymbol{k}^y) D_y \end{cases}$$

$$\boldsymbol{u} = [u_{1,1}, u_{2,1}, \cdots, u_{n_x,1}, u_{1,2}, u_{2,2}, \cdots, u_{n_x,1}, \cdots, u_{1,n_y}, u_{2,n_y}, \cdots u_{n_x,n_y}]^\top$$

$$\boldsymbol{c} = [c_{1,1}, c_{2,1}, \cdots, c_{n_x,1}, c_{1,2}, c_{2,2}, \cdots, c_{n_x,1}, \cdots, c_{1,n_y}, c_{2,n_y}, \cdots c_{n_x,n_y}]^\top$$

Die Anzahl der Federn und der Feder-Konstanten und die Form von D_x und D_y hängen von der Konfiguration ab. Diese Details lassen sich am leichtesten aus dem folgenden Matlab-Code entnehmen.

```
    h1 = figure(1);
    close(h1);
    h1 = figure(1);
    set(h1,'Position',[100 10 500 500]);

    m  = 1;

    beispiel = 1;
    switch(beispiel)
        case 1
% am Rand freie Massen
            nx = 50; ny = 50; nt = 1000; dt = 0.1; th = 0.9;
            f  = 0;
            u0 = zeros(nx,ny);
            u0(round(3*nx/8):round(5*nx/8),round(3*nx/8):round(5*nx/8)) = 1.0;
            u1 = zeros(nx,ny);
            dx = spdiags(ones(nx-1,1),1,nx-1,nx) ...
               - spdiags(ones(nx-1,1),0,nx-1,nx);
            iy = speye(ny);
            Dx = kron(iy,dx);
            dy = spdiags(ones(ny-1,1),1,ny-1,ny) ...
               - spdiags(ones(ny-1,1),0,ny-1,ny);
            ix = speye(nx);
            Dy = kron(dy,ix);
            kx = ones(nx-1,ny);
            ky = ones(nx,ny-1);
            K = -Dx'*spdiags(kx(:),0,(nx-1)*ny,(nx-1)*ny)*Dx ...
                -Dy'*spdiags(ky(:),0,(ny-1)*nx,(ny-1)*nx)*Dy;
            c  = zeros(nx,ny);
            c( 1,:) = sqrt(kx(1,:));
            c(nx,:) = sqrt(kx(nx-1,:));
            c(:, 1) = sqrt(ky(:,1));
            c(:,ny) = sqrt(ky(:,ny-1));
        case 2
% am Rand befestigte Massen
            nx = 50; ny = 50; nt = 1000; dt = 0.1; th = 0.9;
            f  = 0;
            u0 = zeros(nx,ny);
            u0(round(nx/4):round(nx/2),round(nx/4):round(nx/2)) = 1.0;
%           u0 = 0.2*randn(nx,ny);
            u1 = zeros(nx,ny);
            dx = spdiags(ones(nx,1), 0,nx+1,nx) ...
               - spdiags(ones(nx,1),-1,nx+1,nx);
            iy = speye(ny);
            Dx = kron(iy,dx);
            dy = spdiags(ones(ny,1), 0,ny+1,ny) ...
               - spdiags(ones(ny,1),-1,ny+1,ny);
            ix = speye(nx);
            Dy = kron(dy,ix);
            kx = ones(nx+1,ny);
            ky = ones(ny,ny+1);
            K = -Dx'*spdiags(kx(:),0,(nx+1)*ny,(nx+1)*ny)*Dx ...
```

```
                -Dy'*spdiags(ky(:),0,nx*(ny+1),nx*(ny+1))*Dy;
            c  = zeros(nx,ny);
        case 3
% Massen auf einem Torus
            nx = 50; ny = 50; nt = 1000; dt = 0.1; th = 0.9;
            f  = 0;
            u0 = zeros(nx,ny);
            u0(round(nx/4):round(nx/2),round(nx/4):round(nx/2)) = 1.0;
%           u0 = 0.2*randn(nx,ny);
            u1 = zeros(nx,ny);
            dx = spdiags(ones(nx,1), 0,nx,nx) ...
               - spdiags(ones(nx,1),-1,nx,nx);
            dx(1,nx) = -1;
            iy = speye(ny);
            Dx = kron(iy,dx);
            dy = spdiags(ones(ny,1), 0,ny,ny) ...
               - spdiags(ones(ny,1),-1,ny,ny);
            dy(1,nx) = -1;
            ix = speye(nx);
            Dy = kron(dy,ix);
            kx = ones(nx,ny);
            ky = ones(nx,ny);
            K = -Dx'*spdiags(kx(:),0,nx*ny,nx*ny)*Dx ...
                -Dy'*spdiags(ky(:),0,nx*ny,nx*ny)*Dy;
            c  = zeros(nx,ny);
    end

    Z  = sparse(nx*ny,nx*ny);
    I  = speye(nx*ny);
    C  = spdiags(-c(:),0,nx*ny,nx*ny);
    A  = [Z,I;K/m,C/m];
    F  = (f/m)*[zeros(nx*ny,1);ones(nx*ny,1)];
    I2 = speye(2*nx*ny,2*nx*ny);

    U0 = [u0(:);u1(:)];

    i = 0;
        U = U0;
        u = reshape(U(1:nx*ny),nx,ny);
        surf(u);
        axis([0 nx+1 0 ny+1 -1 +1])
        axis off
        drawnow;

    for i=1:nt
        U = (I2 - th*dt*A) \ (U + (1-th)*dt*(A*U+F));
        u = reshape(U(1:nx*ny),nx,ny);
        surf(u);
        axis([0 nx+1 0 ny+1 -1 +1])
        axis off
        drawnow;
    end
```

Die Basis dieses Systems von gewöhnlichen Differentialgleichungen ist in der folgenden partiellen Differentialgleichung zu sehen:

$$m(x,y)u_{tt}(x,y,t) + c(x,y)u_t(x,y,t) = (\kappa^x(x,y)u_x(x,y,t))_x + (\kappa^y(x,y)u_y(x,y,t))_y + f(x,y)$$

Die Beziehung zwischen der partiellen Differentialgleichung und der gewöhnlichen Differentialgleichung ergibt sich durch räumliche Diskretisierung. Weitere fortgeschrittene Details über die Numerik für partielle Differentialgleichungen können im folgenden Skriptum gefunden werden,

https://imsc.uni-graz.at/keeling/skripten/numpde.pdf

3 Natürliche Rhythmen

3.1 Natürliche Rhythmen - partielle Differentialgleichung

Gesucht werden die Wellenformen, die durch geeignete Erregung natürlich entstehen und stabil bleiben können. Solche Wellenformen sind Fließgleichgewichte für unsere partielle Differentialgleichung,

$$\begin{cases} m(x,y)u_{tt}(x,y,t) + c(x,y)u_t(x,y,t) = (\kappa^x(x,y)u_x(x,y,t))_x + (\kappa^y(x,y)u_y(x,y,t))_y + f(x,y) \\ \qquad\qquad 0 \le x \le x_{\max}, \quad 0 \le y \le y_{\max}, \quad 0 \le t \\ u(0,y,t) = u(x_{\max},y,t) = u(x,0,t) = u(x,y_{\max},t) = 0, \quad 0 \le t \\ u(x,y,0) = u_0(x,y), \quad u_t(x,y,0) = u_1(x,y) \end{cases}$$

hier beispielsweise mit Befestigung am Rand des Gebiets $\{(x,y) : 0 \le x \le x_{\max}, 0 \le y \le y_{\max}\}$. Ein Fließgleichgewicht ist zeitunabhängig, und eine dynamische Lösung pendelt sich diesem Fließgleichgewicht ein. Zur Vereinfachung nehmen wir zuerst an, dass die Evolution nur in einer räumlichen Dimension abläuft,

$$\begin{cases} mu_{tt}(x,t) + cu_t(x,t) = \kappa u_{xx}(x,t) + f(x), 0 \le x \le x_{\max}, \quad 0 \le t \\ u(0,t) = u(x_{\max},t) = 0, \quad 0 \le t \\ u(x,0) = u_0(x), \quad u_t(x,0) = u_1(x) \end{cases}$$

und die Parameter m, c und κ sind alle konstant. Da ein Fließgleichgewicht u_∞ zeitunabhängig ist, sind alle zeitlichen Ableitungen auf der linken Seite der partiellen Differentialgleichung Null, und es gilt

$$0 = \kappa u''_\infty(x) + f(x), \quad 0 \le x \le x_{\max}.$$

Eine zielführende Erregung ist das Vielfache $f = -\lambda u_\infty/\kappa$,

$$0 = u''_\infty(x) - \lambda u_\infty(x), \quad 0 \le x \le x_{\max}, \quad u_\infty(0) = u_\infty(x_{\max}) = 0.$$

Mit $x_{\max} = \pi$ gibt es folgende einfache Lösungen,

$$u_\infty(x) \to u_n(x) = \sin(nx), \quad \lambda = -n^2, \quad n = 1, 2, \ldots$$

Die Funktionen $u_n(x) = \sin(nx)$ sind die sogenannten *Eigenfunktionen* des räumlichen Differentialoperators in der partiellen Differentialgleichung, und die Werte $\lambda = -n^2$ sind die entsprechenden *Eigenwerte*.

Wenn die Lösung unserer partiellen Differentialgleichung mit Anfangsbedingungen $u(x,0) = \sin(nx)$, $n \in \mathbb{N}$, und $u_t(x,0) = 0$ gestartet wird, und eine ständige Erregung $f(x) = n^2 \sin(nx)/\kappa$ eingeführt wird, dann bleibt die Lösung bei der Wellenform $u(x,t) = \sin(nx)$ für alle Zeiten! Wenn die Anfangsbedingungen anders ausgewählt werden, aber das gleiche f verwendet wird, dann pendelt sich die Lösung dem Fließgleichgewicht $\sin(nx)$ ein!

3.2 Natürliche Rhythmen - gewöhnliche Differentialgleichung

Diese Eigenschaften des Modells zeigen sich auch durch die Diskretisierung der partiellen Differentialgleichung. Dafür sei $x_i = i\pi/n_x$, $i = 0, \ldots, n_x$, ein Gitter für das räumliche Gebiet $[0, \pi]$. Wegen der Befestigung auf beiden Seiten des räumlichen Gebiets gibt es $n_x + 1$ Feder an den Stellen $x_{i+\frac{1}{2}}$, $i = 0, \ldots, n_x$. Die Auslenkung der $i$ten Masse ist $u_i(t)$, der Dämpfungsparameter der $i$ten Masse ist c_i, die äußere Kraft auf der $i$ten Masse ist f_i und die Federkonstante zwischen den $i$ten und $(i+1)$ten Massen ist $k_{i+\frac{1}{2}}$. So seien die folgenden Vektoren definiert,

$$\boldsymbol{u}(t) = \begin{bmatrix} u_1(t) \\ \vdots \\ u_n(t) \end{bmatrix}, \quad \boldsymbol{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}, \quad \boldsymbol{f} = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}, \quad \boldsymbol{k} = \begin{bmatrix} k_{1-\frac{1}{2}} \\ \vdots \\ k_{n_x+\frac{1}{2}} \end{bmatrix}.$$

Mit

$$\boldsymbol{U}(t) = \begin{bmatrix} \boldsymbol{u}(t) \\ \boldsymbol{u}'(t) \end{bmatrix} \quad A = \begin{bmatrix} Z & I \\ K & C \end{bmatrix} \quad m\boldsymbol{F} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{f} \end{bmatrix}$$

$$Z = \operatorname{diag}(\boldsymbol{0}), \quad I = \operatorname{diag}(\boldsymbol{1}), \quad mC = \operatorname{diag}(\boldsymbol{c}), \quad mK = -D^\top \operatorname{diag}(\boldsymbol{k}) D$$

$$D = \begin{bmatrix} +1 & 0 & \cdots & 0 \\ -1 & +1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ & & -1 & +1 \\ 0 & \cdots & 0 & -1 \end{bmatrix} \in \mathbb{R}^{(n_x+1)\times n_x}$$

ist die Diskretisierung durch das folgende System von gewöhnlichen Differentialgleichungen gegeben,

$$\boldsymbol{U}'(t) = A\boldsymbol{U} + \boldsymbol{F}, \quad t \geq 0, \quad \boldsymbol{U}(0) = \boldsymbol{U}_0.$$

Da ein Fließgleichgewicht $\boldsymbol{U}_\infty$ zeitunabhängig ist, sind alle zeitlichen Ableitungen auf der linken Seite der gewöhnlichen Differentialgleichung Null, und es gilt

$$0 = A\boldsymbol{U}_\infty + \boldsymbol{F}.$$

Eine zielführende Erregung ist das Vielfache $\boldsymbol{F} = -\mu W \boldsymbol{U}_\infty$,

$$0 = A\boldsymbol{U}_\infty - \mu W \boldsymbol{U}_\infty, \quad \text{wobei} \quad W = \begin{bmatrix} 0 & 0 \\ I & 0 \end{bmatrix}.$$

Es gibt genau n_x Vektoren $\boldsymbol{U}_n$ und Werte μ_n, die diese Gleichung lösen,

$$\boldsymbol{U}_\infty \to \boldsymbol{U}_n : \quad A\boldsymbol{U}_n = \mu_n W \boldsymbol{U}_n, \quad n = 1, \ldots, n_x.$$

Eine Charakterisierung der Lösungen der letzten Gleichung wird durch die Zerlegung erleichtert,

$$\boldsymbol{U}_n = \begin{bmatrix} \boldsymbol{u}_n \\ \boldsymbol{v}_n \end{bmatrix}, \quad \text{d.h.} \quad A\boldsymbol{U}_n = \mu_n W \boldsymbol{U}_n \quad \text{wenn} \quad \begin{bmatrix} Z & I \\ K & C \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_n \\ \boldsymbol{v}_n \end{bmatrix} = \mu_n \begin{bmatrix} 0 \\ \boldsymbol{u}_n \end{bmatrix}.$$

Die letzte Gleichung wird gelöst durch $\boldsymbol{v}_n = \boldsymbol{0}$ und

$$K\boldsymbol{u}_n = \mu_n \boldsymbol{u}_n, \quad n = 1, \ldots, n_x.$$

Die Vektoren $\boldsymbol{u}_n$ sind die sogenannten *Eigenvektoren* der Matrix K und die Werte μ_n sind die entsprechenden *Eigenwerte*. Schönerweise sind hier die diskreten Lösungen $\boldsymbol{u}_n$ Approximationen der Eigenfunktionen im kontinuierlichen Kontext,

$$(\boldsymbol{u}_n)_i \approx u_n(x_i) = \sin(nx_i), \quad \mu_n \approx \lambda_n.$$

Wenn die Lösung unserer gewöhnlichen Differentialgleichung mit Anfangsbedingungen $\boldsymbol{U}_0 = [\mathbf{0} \;\; \boldsymbol{u}_n]^\top$, $n \in \mathbb{N}$ gestartet wird, und eine ständige Erregung $\boldsymbol{F} = [\mathbf{0} \;\; -\mu_n \boldsymbol{u}_n]^\top$ eingeführt wird, dann bleibt die Lösung bei $\boldsymbol{U}(t) = \boldsymbol{U}_0$ für alle Zeiten! Wenn die Anfangsbedingungen anders ausgewählt werden, aber das gleiche $\boldsymbol{F}$ verwendet wird, dann pendelt sich die Lösung dem Fließgleichgewicht $\boldsymbol{U}_0$ ein!

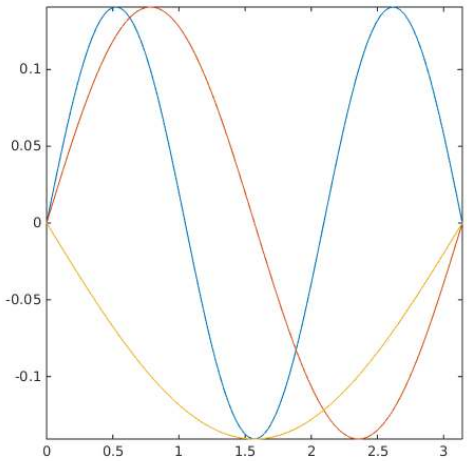
3.3 Natürliche Rhythmen - Programmieren in 1D

Das System der gewöhnlichen Differentialgleichungen mit natürlichen Rhythmen kann für eine räumliche Dimension mit dem folgenden Matlab-Code gelöst werden.

```
    h1 = figure(1);
    close(h1);
    h1 = figure(1);
    set(h1,'Position',[100 10 500 500]);

% auf zwei Seiten befestigte Massen
    nx = 100; nt = 100; dt = 0.1; th = 0.5;
    x = linspace(0,pi,nx+2); x = x(2:(nx+1))';
    h = x(2)-x(1);
    D = spdiags(ones(nx,1), 0,nx+1,nx) ...
      - spdiags(ones(nx,1),-1,nx+1,nx);

% gleichmaessige Federkonstanten
    k = ones(nx+1,1)/h^2;
% mit Bruch
%   k(round(nx/2)) = 0;
% zufaellige Federkonstanten
%   k = rand(nx+1,1)/h^2;

    K = -D'*spdiags(k,0,nx+1,nx+1)*D;

% Eigenraum-Zerlegung der Matrix K
    [V,L] = eig(full(K));
% wobei V(:,1) der ite Eigenvektor ist, und L(i,i) ist der ite Eigenwert
% Fliessgleichgewicht
    ug = sin(x);   % kontinuierlich
    ig = nx;
    ug = V(:,ig);  % diskret

% Daempfung zur schnellen Konvergenz
    c  = zeros(nx,1) + 1;
    Z  = sparse(nx,nx);
    I  = speye(nx);
    C  = spdiags(-c,0,nx,nx);
    A  = [Z,I;K,C];
    I2 = speye(2*nx);
    F  = zeros(2*nx,1);
% staendige natuerliche Erregung
    F  = [zeros(nx,1);-L(ig,ig)*ug];

% mit Null startend
    u0 = zeros(nx,1);
```

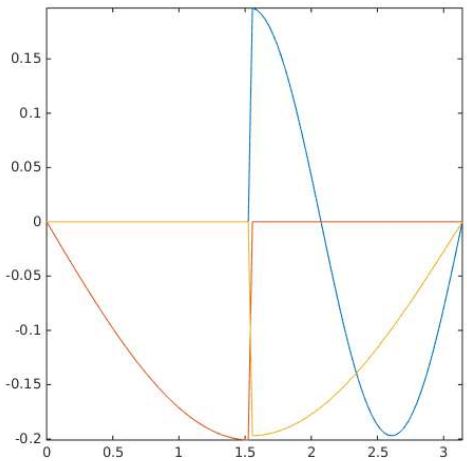
```
% mit Fliessgleichgewicht startend
%   u0 = ug;
    u1 = zeros(nx,1);
    U  = [u0;u1];

    for i=1:nt
        U = (I2 - th*dt*A) \ (U + dt*((1-th)*A*U + F));

        u = U(1:nx);
        plot([0;x;pi],[0;u;0])
        axis([0 pi -0.25 0.25])
        drawnow;
    end
```

Die sanftesten Eigenvektoren der Matrix K mit Federkonstanten $\boldsymbol{k} = \mathbf{1}/n_x^2$ sind hier grafisch dargestellt,



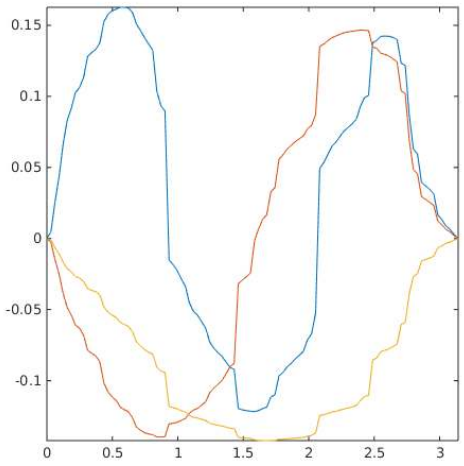
Diese sind natürliche Fließgleichgewichte für die gegebenen Federkonstanten.

Die sanftesten Eigenvektoren der Matrix K mit Federkonstanten $\boldsymbol{k} = \mathbf{1}/n_x^2$, $(\boldsymbol{k})_{n_x/2} = 0$, sind hier grafisch dargestellt,



Diese sind natürliche Fließgleichgewichte für die gegebenen Federkonstanten.

Die sanftesten Eigenvektoren der Matrix K mit Federkonstanten $\boldsymbol{k}$ mit zufälligen Werten zwischen 0 und $1/n_x^2$ sind hier grafisch dargestellt,



Diese sind natürliche Fließgleichgewichte für die gegebenen Federkonstanten.

Es ist auffällig, dass die natürlichen Rhythmen für gleichmäßige Federkonstanten stimmiger sind als für sonstige Federkonstanten.

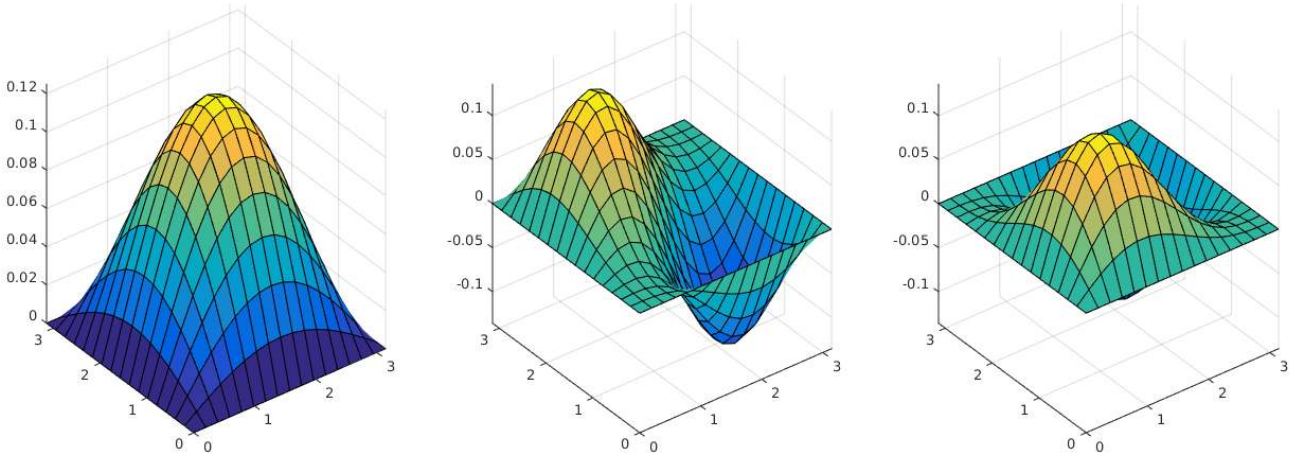
3.4 Natürliche Rhythmen - Programmieren in 2D

Das System der gewöhnlichen Differentialgleichungen mit natürlichen Rhythmen kann für zwei räumliche Dimensionen mit dem folgenden Matlab-Code gelöst werden.

```
    h1 = figure(1);
    close(h1);
    h1 = figure(1);
    set(h1,'Position',[100 10 500 500]);

% am Rand befestigte Massen
    nx = 15; ny = 15; nt = 100; dt = 0.1; th = 0.5;
    x  = linspace(0,pi,nx+2); x = x(2:(nx+1));
    y  = linspace(0,pi,ny+2); y = y(2:(ny+1));
    hx = x(2)-x(1);
    hy = y(2)-y(1);
    dx = spdiags(ones(nx,1), 0,nx+1,nx) ...
       - spdiags(ones(nx,1),-1,nx+1,nx);
    iy = speye(ny);
    Dx = kron(iy,dx);
    dy = spdiags(ones(ny,1), 0,ny+1,ny) ...
       - spdiags(ones(ny,1),-1,ny+1,ny);
    ix = speye(nx);
    Dy = kron(dy,ix);

% gleichmaessige Federkonstanten
    kx = ones(nx+1,ny)/hx^2;
    ky = ones(ny,ny+1)/hy^2;
% mit Bruch
%   kx(round(nx/2),:) = 0;
```

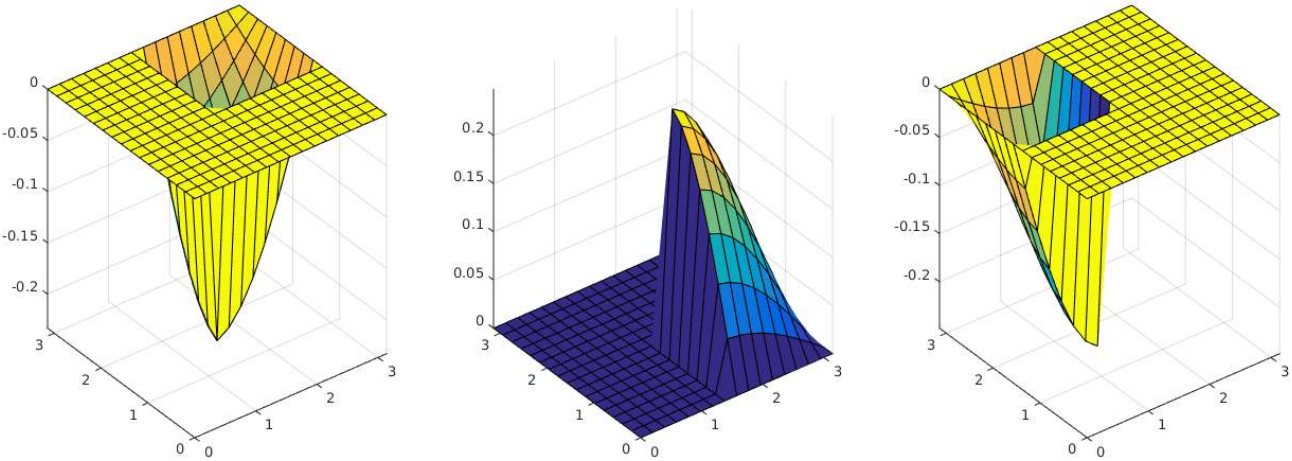
```
%   ky(:,round(ny/2)) = 0;
% zufaellige Federkonstanten
%   kx = rand(nx+1,ny)/hx^2;
%   ky = rand(ny,ny+1)/hy^2;

    K = -Dx'*spdiags(kx(:),0,(nx+1)*ny,(nx+1)*ny)*Dx ...
        -Dy'*spdiags(ky(:),0,nx*(ny+1),nx*(ny+1))*Dy;

% Eigenraum-Zerlegung der Matrix K
    [V,L] = eig(full(K));
% wobei V(:,1) der ite Eigenvektor ist, und L(i,i) ist der ite Eigenwert
% Fliessgleichgewicht
%    ug = 0.25*sin(x')*sin(y);   % kontinuierlich
     ig = nx*ny;
     ug = V(:,ig);               % diskret

% Daempfung zur schnellen Konvergenz
    c  = zeros(nx,ny) + 1;

    Z  = sparse(nx*ny,nx*ny);
    I  = speye(nx*ny);
    C  = spdiags(-c(:),0,nx*ny,nx*ny);
    A  = [Z,I;K,C];
    I2 = speye(2*nx*ny,2*nx*ny);
    F  = zeros(2*nx*ny,1);
% staendige natuerliche Erregung
%   F  = [zeros(nx*ny,1);ug(:)];            % kontinuierlich
    F  = [zeros(nx*ny,1);-L(ig,ig)*ug(:)];  % diskret

% mit Null startend
    u0 = zeros(nx,ny);
% mit Fliessgleichgewicht startend
    u0 = ug;
    u1 = zeros(nx,ny);
    U  = [u0(:);u1(:)];

    for i=1:nt
        U = (I2 - th*dt*A) \ (U + (1-th)*dt*(A*U+F));
        u = reshape(U(1:nx*ny),nx,ny);
        u = [zeros(nx,1),u,zeros(nx,1)];
        u = [zeros(1,ny+2);u;zeros(1,ny+2)];
        surf([0;x(:);pi],[0;y(:);pi],u);
        axis([0 pi 0 pi -0.1 +0.1])
        drawnow;
    end
```

Die sanftesten Eigenvektoren der Matrix K mit Federkonstanten $\boldsymbol{k}^x = \mathbf{1}/n_x^2$ und $\boldsymbol{k}^y = \mathbf{1}/n_y^2$ sind hier grafisch dargestellt,



Diese sind natürliche Fließgleichgewichte für die gegebenen Federkonstanten.

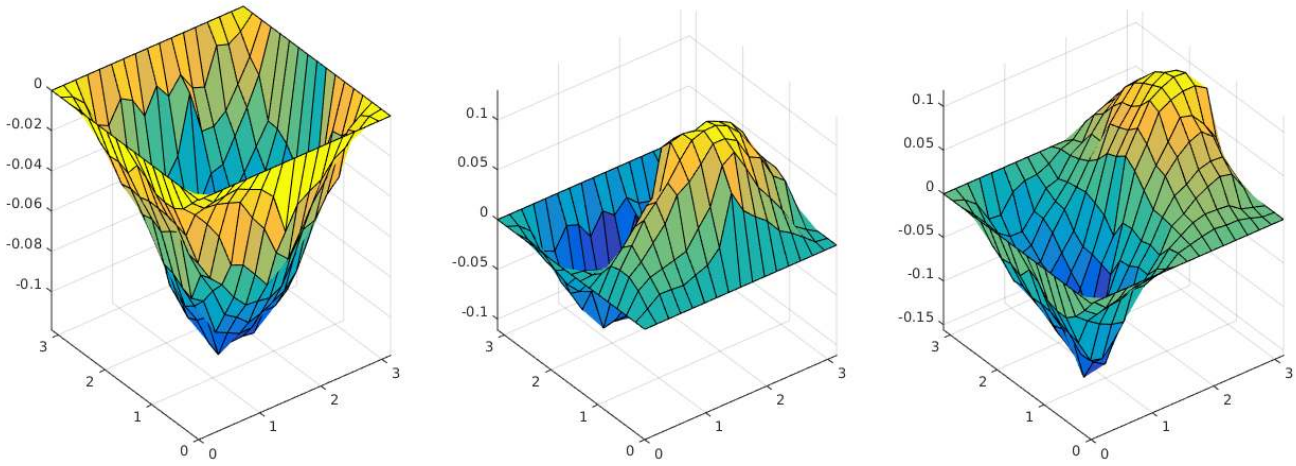
Die sanftesten Eigenvektoren der Matrix K mit Federkonstanten $\boldsymbol{k}^x = \mathbf{1}/n_x^2$, $(\boldsymbol{k}^x)_{n_x/2,:} = 0$, $\boldsymbol{k}^y = \mathbf{1}/n_y^2$, $(\boldsymbol{k}^y)_{:,n_y/2} = 0$, sind hier grafisch dargestellt,



Diese sind natürliche Fließgleichgewichte für die gegebenen Federkonstanten.

Die sanftesten Eigenvektoren der Matrix K mit Federkonstanten $\boldsymbol{k}^x$ mit zufälligen Werten zwischen 0 und $1/n_x^2$ und Federkonstanten $\boldsymbol{k}^y$ mit zufälligen Werten zwischen 0 und $1/n_y^2$ sind

hier grafisch dargestellt,



Diese sind natürliche Fließgleichgewichte für die gegebenen Federkonstanten.

Es ist auffällig, dass die natürlichen Rhythmen für gleichmäßige Federkonstanten stimmiger sind als für sonstige Federkonstanten.

4 Harmonische Klänge mit Sinus- und Cosinus-Kurven im Eindimensionalen

Fall Zwei des Programmcodes, der eine Anzahl von n Federn gespannt zwischen zwei Fixpunkten darstellt, wird hier in Form einer Sinus- Kurve in eine Graphik umgesetzt. In diesem Fall konzentrierten wir uns vor allem auf die Darstellung von einem Klang in Form der Sinus- und Cosinus-Funktionen.

Zusätzlich werden diese Wellen durch Turbulenzen gestört, die allerdings nur in den Bereichen der Funktion auftauchen, in denen die y-Werte unter einen einstellbaren Schwellenwert fallen. Bei der unten eingesendeten Grafik wird alle 20 Zeiteinheiten (Zeile 34 des Codes) die Sinus-/Cosinuswelle durch einen Impuls aufrechterhalten, so dass die y-Werte nicht unter einen Schwellenwert von 0.2 fallen (Zeile 43 des Codes). Wenn dies jedoch geschieht, wird automatisch alle 5 Sekunden ein Störungsimpuls ausgesendet.

Der folgende Programmcode stellt die Grafik *Harmonie mit Turbulenzen* dar:

```
1     h1 = figure(1);
2     close(h1);
3     h1 = figure(1);
4     set(h1,'Position',[100 10 1000 500]);
5
6  %  auf zwei Seiten befestigte Massen
7     nx = 100; nt = 10000; dt = 0.1; th = 1.0;
8     x = linspace(0,2*pi,nx+2); x = x(2:(nx+1))';
9     h = x(2)-x(1);
```

```
10    ue = (1:nx)';
11    ug = sin(x);
12    u0 = zeros(nx,1); % u0 = ug;
13    u1 = zeros(nx,1);
14    D = spdiags(ones(nx,1), 0,nx+1,nx) ...
15      - spdiags(ones(nx,1),-1,nx+1,nx);
16    k = ones(nx+1,1)/h^2;
17    K = -D'*spdiags(k,0,nx+1,nx+1)*D;
18    c = zeros(nx,1)+0.01;
19    Z  = sparse(nx,nx);
20    I  = speye(nx);
21    C  = spdiags(-c,0,nx,nx);
22    A  = [Z,I;K,C];
23    I2 = speye(2*nx);
24    F  = zeros(2*nx,1); % F = [zeros(nx,1);ug];
25
26    uv = ue+u0;
27    U  = [u0;u1];
28
29    ir = [];
30    for i=1:nt
31        U = (I2 - th*dt*A) \ (U + dt*((1-th)*A*U + F));
32
33 % stabilisierend:
34        if (mod(i,20) == 0)
35            F = [zeros(nx,1);5*ug];
36        else
37            F = zeros(2*nx,1);
38        end
39
40 % turbulenzen:
41        if (mod(i,2) == 0)
42            ir = randi(nx);
43            if (abs(U(ir)) < 0.2)
44                U(ir) = U(ir) + (-1)^ir;
45            end
46        end
47
48 % grafische Darstellung
49        u  = U(1:nx);
50        uv = [uv,u+ue];
51        subplot(1,2,1)
52        plot(0:i,uv')
53        axis tight
54        subplot(1,2,2)
55        plot(0:nx+1,[0;u;0])
56        axis([0 nx+1 -1.5 1.5])
57 %       axis off
58        drawnow;
59    end
```

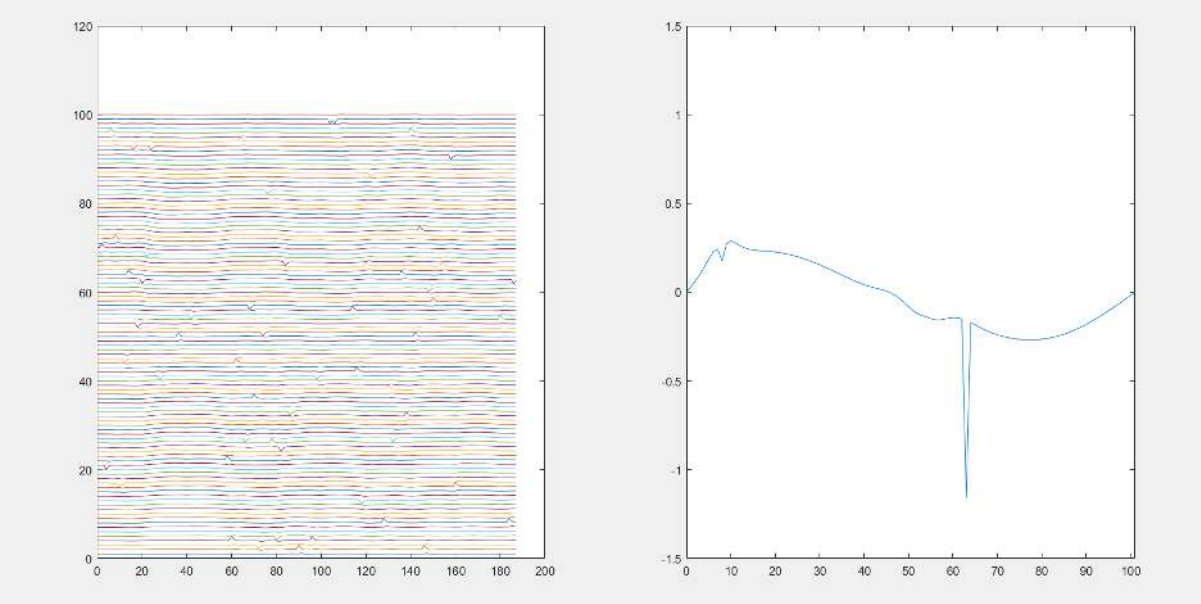


Abbildung 9: Harmonie mit Turbulenzen

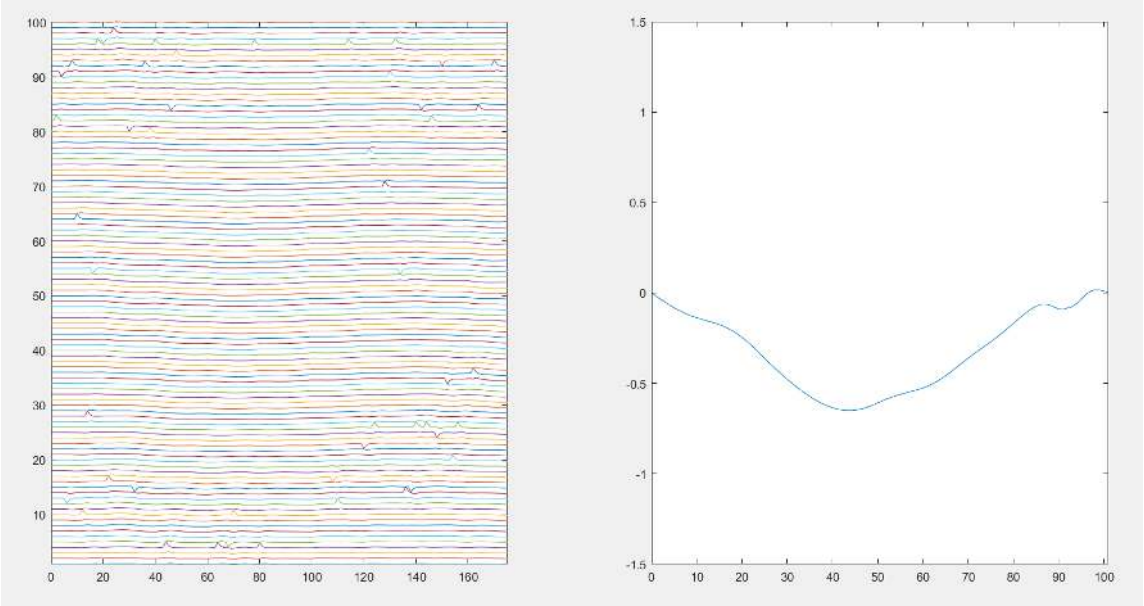


Abbildung 10: Harmonische Klänge in Form einer Cosinus-Kurve

5 Harmonische Klänge mit Sinus- und Cosinus-Kurven im Zweidimensionalen

Ähnlich zum Programmcode der eindimensionalen Grafik funktioniert auch die zweidimensionale Darstellung. Näher werden die Unterschiede im Kapitel „Zweiräumliche Dimensionen und die partielle Differentialgleichung“ beschrieben. Hier wird neben der Sinusfunktion auch auf die Cosinusfunktion eingegangen. Um zwischen den beiden Funktionen zu wechseln, muss lediglich in

Zeile 12 des Programmcodes der Befehl „sin“ mit „cos“ ausgetauscht werden. Gleich wie im Eindimensionalen treten Turbulenzen auf und verändern, wie oben erklärt, die Welle. Grundsätzlich stellen diese Störungen, in unserer Interpretation der Funktion, die unbeeinflussbaren, zufällig auftretenden Probleme des Alltags dar. Je höher der Wert der Zahl in Zeile 51 ist, desto geringer sind die Störungen, da die Zeitintervalle zwischen den einzelnen Impulsen der Turbulenzen vergrößert wurden. Ein Mensch, dessen Zeitintervall sehr gering ist, beispielsweise wie im unten angeführten Code mit 2, besitzt sehr viele Alltagsprobleme. Wenn sehr wenige Probleme auftreten und die Anzahl der Impulse für die Sinusfunktion, welche in Zeile 45 einstellbar ist, gering ist, dann gehen wir von einer Person aus, die in ihrer Freizeit regelmäßig Meditation und/oder Yoga betreibt, da die Funktion sehr ruhig und regelmäßig abläuft. Wir würden diese Schwingung als harmonisch bezeichnen.

Der folgende Programmcode stellt die folgende Grafik dar:

```
1     h1 = figure(1);
2     close(h1);
3     h1 = figure(1);
4     set(h1,'Position',[100 10 1000 500]);
5
6     % auf zwei Seiten befestigte Massen
7     nx = 60; ny = 60; nt = 10000; dt = 0.1; th = 1.0;
8     x = linspace(0,2*pi,nx+2); x = x(2:(nx+1));
9     y = linspace(0,2*pi,ny+2); y = y(2:(ny+1));
10    hx = x(2)-x(1);
11    hy = y(2)-y(1);
12    ug = sin(x')*sin(y);
13    u0 = zeros(nx,ny);
14    % u0 = ug;
15    u1 = zeros(nx,ny);
16    dx = spdiags(ones(nx,1), 0,nx+1,nx) ...
17       - spdiags(ones(nx,1),-1,nx+1,nx);
18    iy = speye(ny);
19    Dx = kron(iy,dx);
20    dy = spdiags(ones(ny,1), 0,ny+1,ny) ...
21       - spdiags(ones(ny,1),-1,ny+1,ny);
22    ix = speye(nx);
23    Dy = kron(dy,ix);
24    kx = ones(nx+1,ny)/hx^2;
25    ky = ones(ny,ny+1)/hy^2;
26    K = -Dx'*spdiags(kx(:),0,(nx+1)*ny,(nx+1)*ny)*Dx ...
27        -Dy'*spdiags(ky(:),0,nx*(ny+1),nx*(ny+1))*Dy;
28    c = zeros(nx,ny)+0.01;
29
30    Z  = sparse(nx*ny,nx*ny);
31    I  = speye(nx*ny);
32    C  = spdiags(-c(:),0,nx*ny,nx*ny);
33    A  = [Z,I;K,C];
34    I2 = speye(2*nx*ny);
35    F = zeros(2*nx*ny,1); % F=[zeros(nx*ny,1);2*ug(:)];
36
37    %uv = ue+u0;
38    U  = [u0(:);u1(:)];
```

```
39
40      ir = [];
41      for i=1:nt
42          U = (I2 - th*dt*A) \ (U + dt*((1-th)*A*U + F));
43
44      % stabilisierend:
45          if (mod(i,30) == 0)
46              F = [zeros(nx*ny,1);10*ug(:)];
47          else
48              F = zeros(2*nx*ny,1);
49          end
50      % turbulenzen:
51          if (mod(i,2) == 0)
52              ir = randi(nx*ny);
53            if (abs(U(ir)) < 0.3)
54                  U(ir) = U(ir) + (-1)^ir;
55              end
56          end
57
58      % grafische Darstellung
59
60          u  = reshape(U(1:nx*ny),nx,ny);
61          surf (u);
62          axis([0 nx+1 0 ny-1 -1 1.5])
63          axis on
64          drawnow;
65
66      end
```

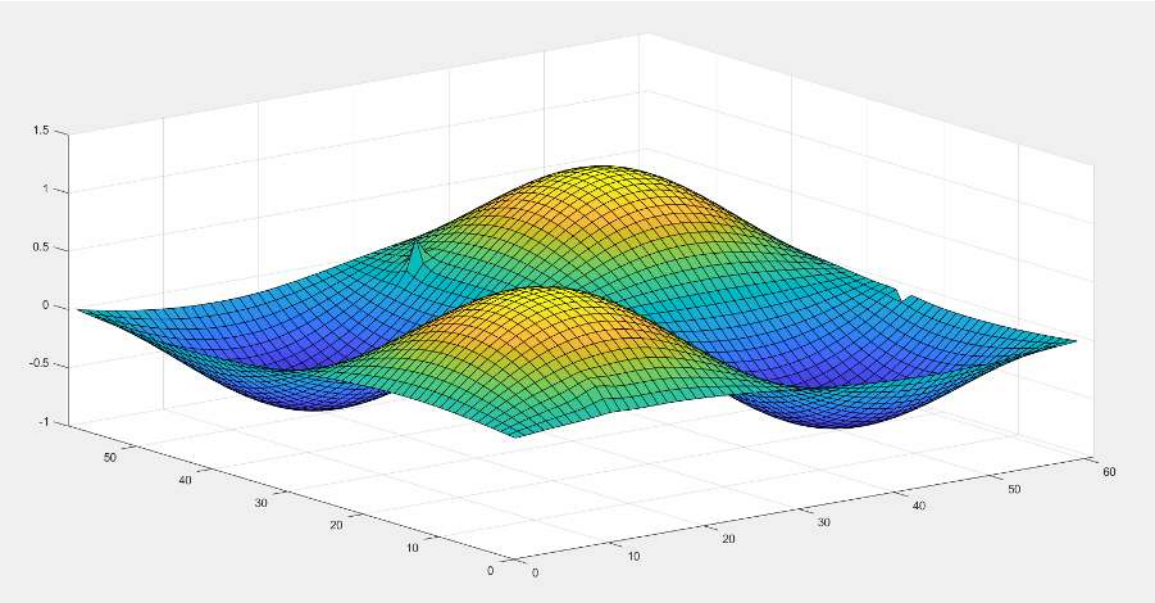


Abbildung 11:

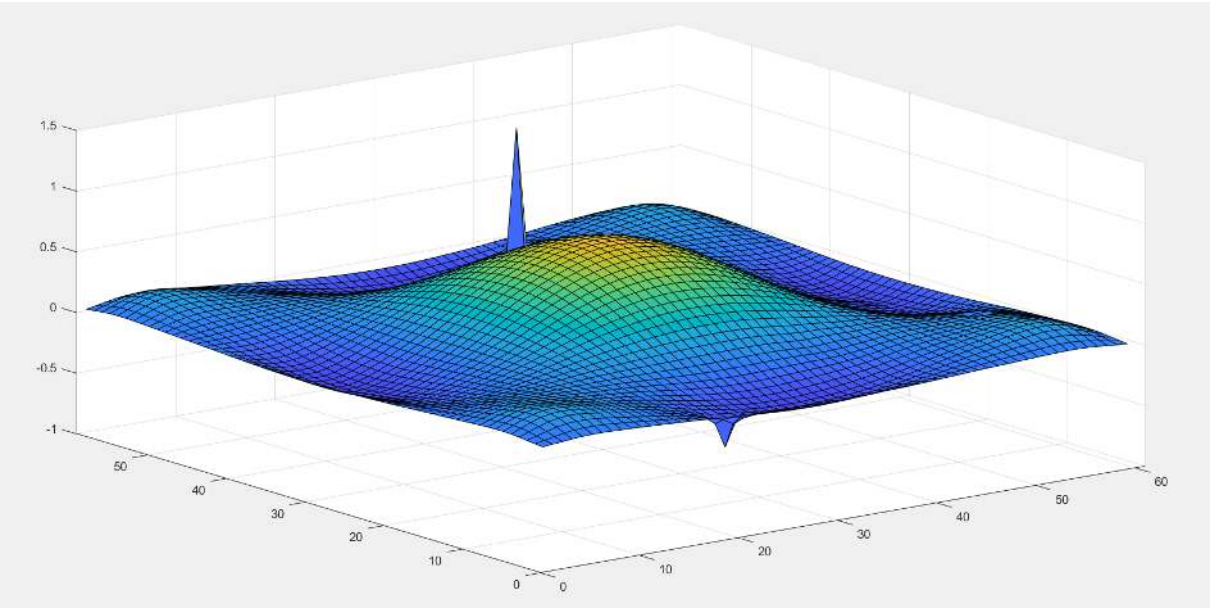


Abbildung 12: harmonischer Klang in Form einer Cosinus-Kurve in 2D

6 Reizweiterleitung im Gehirn

Zur Veranschaulichung der unterschiedlichen Arbeitsweisen der beiden Gehirnhälften erstellten wir ein Modell, das den Durchlauf eines Reizes durch beide Gehirnhälften simuliert.
Die zwei Gehirnhälften: Das menschliche Gehirn besteht aus der rechten und der linken Gehirnhälfte, welche durch den Gehirnbalken verbunden sind. Somit ist eine Interaktion zwischen den beiden Gehirnhälften nur über den Gehirnbalken möglich.

Abbildung 13: rechte und linke Gehirnhälfte

Vergleich zwischen linker und rechter Gehirnhälfte

Linke Gehirnhälfte	Rechte Gehirnhälfte
Verarbeitung rationaler, sprachlicher, analytischer, zeitlich linearer und logischer Prozesse	Ganzheitlich, bildhaft, musisch, kreativ, zeitlos, intuitiv, räumlich, emotional, körperorientiert

Die Grundidee von diesem Modell bildet auch die Basis für Verfahren psychologischer Marktforschung.

Unser Modell

Ein Reiz (z.B.: ein Gedanke) wird in unserem Modell als (neuronale) Welle dargestellt.

Dabei entsteht zu Beginn eine Welle in der rechten Gehirnhälfte. Diese bahnt sich ihren Weg zur Mitte des Gehirns. Dort wird entschieden, ob der Reiz in die linke Hemisphäre weitergeleitet wird. Die Weiterleitung findet jedoch nur unter der Bedingung statt, dass sich in der linken Hälfte keine Gedankenwelle befindet.

```
h1 = figure(1);
set(h1,'Position',[100 10 1000 500]);
m = 1;      a = 0;       nx = 100;            nt = 2000;         f  = 0.0;
dt = 0.5;   th = 0.75;   nh = round(nx*0.5);  ue = (1:nx)';
u0 = zeros(nx,1);        u0(0.75*nx) = 8;     u1 = zeros(nx,1);
D  = spdiags(ones(nx-1,1),1,nx-1,nx) ...
- spdiags(ones(nx-1,1),0,nx-1,nx);
k  = ones(nx-1,1);    k(nh)=0;          K  = -D'*spdiags(k,0,nx-1,nx-1)*D;
c  = zeros(nx,1);     c(0.5*nx+1) = sqrt(k(nx-1));                    c(nx) = sqrt(k(nx-1)); c(1) = sqrt(k(1));     %Reibung
Z  = sparse(nx,nx);   I  = speye(nx);   C  = spdiags(-c,0,nx,nx);     A  = [Z,I;K/m,C/m];    I2 = speye(2*nx);
F  = (f/m)*[zeros(nx,1);ones(nx,1)];    uv = ue+u0;                   U  = [u0;u1];
for i=1:nt
U = (I2 - th*dt*A) \ (U + dt*((1-th)*A*U + F));
u = U(1:nx);
if abs(U(nh+1))>0.1 && a==0                         %Übergang vom rechten ins linke Hirn
U(nh) = U(nh+1)*10*2;   a=1;
end
if abs(U(2))>0.1                                    %Übergang vom linken ins rechte Hirn
U(nx-1) = U(1)*2;   a=0;
end
if U(0.5*nx:nx)<0.05                                %Wiederholung des Impulses im rechten Hirn
U(0.75*nx) = 8;
end
uv = [uv,u+ue];
subplot(1,2,1)
plot(0:i,uv');           axis tight
     subplot(1,2,2)
     plot(1,0,'.k');          hold on
     plot(nx,0,'.k');         hold on
     plot(nx*0.5,0,'.k');     hold on
     plot(nx*0.75,0,'.k');    hold on
     plot(1:nx,u);            hold off
     axis([0 nx+1 -3 3]);     axis off;
     text(round(nx/8),-0.3,'linke Gehirnhälfte');
     text(nh+round(nx/8),-0.3,'rechte Gehirnhälfte');
     drawnow;
     end
```

Abbildung 14: Code

In unserem Code wird ein Impuls in der rechten Gehirnhälfte erzeugt (Zeile 5) und als Welle grafisch dargestellt. Sie breitet sich in Richtung linker Gehirnhälfte aus. Am Gehirnbalken angekommen wird überprüft, ob sich eine Welle in der linken Gehirnhälfte befindet (Zeile 15). Ist dies der Fall, wird der Gedanke in der rechten Hemisphäre abgeschwächt und erlischt. Ist jedoch kein Impuls in der linken Gehirnhälfte, so wird er ins linke Gehirn weitergeleitet. Dort wandert es durch die linke Hemisphäre und wird in gleichförmiger Geschwindigkeit zum rechten Teil geschickt (Zeile 16).

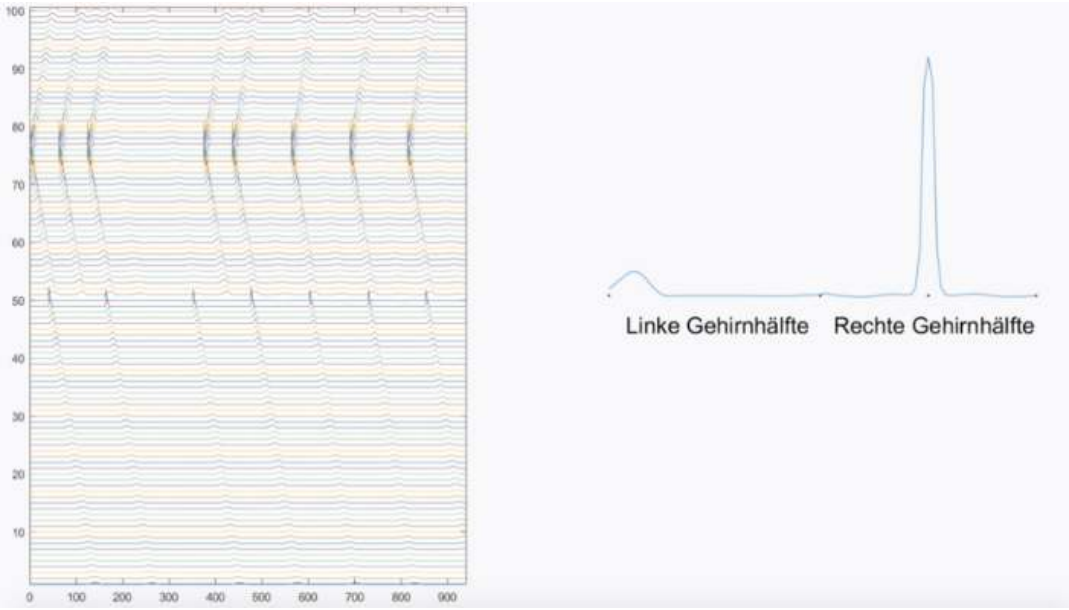


Abbildung 15: Erzeugung des Impulses

Abbildung 16: Impuls am Gehirnbalken

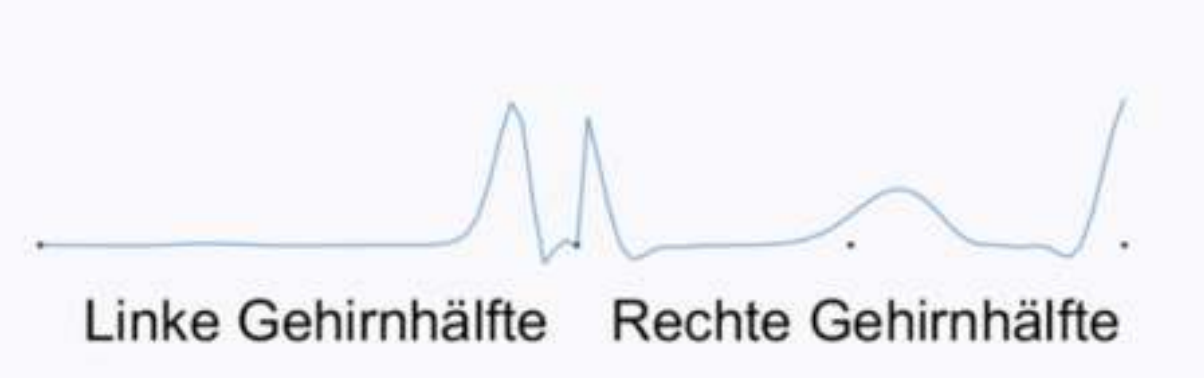


Abbildung 17: Reizweiterleitung in die linke Hemisphäre

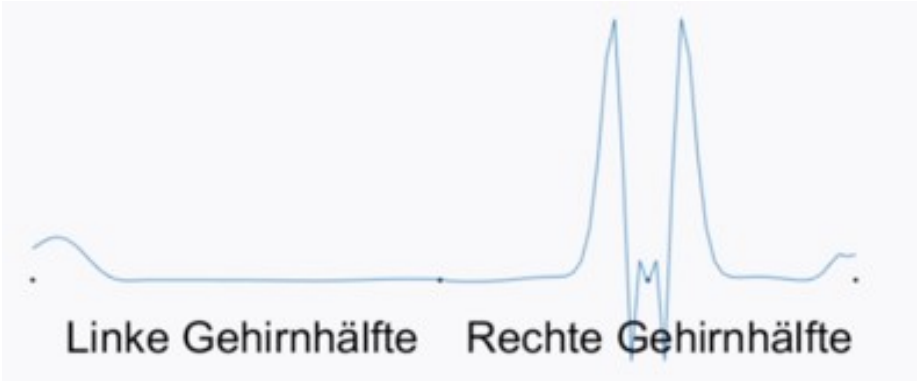


Abbildung 18: Impuls springt auf die rechte Gehirnhälfte um

Abbildung 19: Impuls wird am Gehirnbalken aufgehalten und abgeschwächt

Dabei kam es auch zu einem Spezialfall. Die ins linke Gehirnareal weitergeleitete Welle gelangte wieder in die rechte Seite und hemmte somit den weiteren Impuls (Weiterer Impuls: Zeile 21). Dieses Phänomen ist schon seit langem von Neurophysiologen entdeckt worden. Wenn also eine Gehirnhälfte aktiv ist, unterdrückt sie dabei die andere. Dabei scheint es so, als hätten beide Hemisphären verschiedene Interessen und Kommunizieren nur minimalst. Die Verschiedenheit hat jedoch auch seine Vorteile. Durch diesen Kontrast ergänzen sich beide Gehirnhälften und harmonieren.

Abbildung 20: Unterdrückung des Impulses der rechten Hemisphäre

Unser Fazit ist, dass sich die linke und rechte Hemisphäre gegenseitig stark beeinflussen. Dass unsere linke Gehirnhälfte nur einen Impuls nach dem anderen verarbeiten kann und somit strukturiert und logische, rationale Probleme bewältigt. Die rechte Seite beschäftigt sich mit den räumlich, emotionalen Dinge. Zusammen bewältigen wir Probleme in unserem Alltag.

7 Dämpfung durch Meditation

Meditation soll nachweislich beruhigend wirken und gilt in vielen Ländern als Entspannungstechnik. Dabei nehmen wir an, dass die hohe Amplitude der Wellen durch Meditation (immens) gedämpft wird. Wir wollen ein Modell erstellen, welches die Visualisierung dieses Phänomens gut ermöglicht.

Um die Amplitude der Wellen in unseren bereits bestehenden Modellen gezielt verringern zu können, wird die Federkonstante k geändert. Da wir dieses Phänomen gleich in einem zweidimensionalen Modell visualisieren wollen, müssen die einzelnen Parametern vom Masse-Feder-System in x-Richtung sowie in y-Richtung betrachtet werden. Somit werden k^x und k^y verändert, woraus sich ein neues K, welches von k^x und k^y beeinflusst wird, mit anderen Werten ergibt.

Da wir den Übergang von ungeordneten Wellen mit großer Amplitude zu gemäßigten mit einer kleineren Amplitude darstellen wollen, müssen die Federkonstanten so verändert werden, dass Bereiche mit verschiedenen Federkonstanten k entstehen. Dabei kommt es zu Grenzbereichen, an denen Bereiche mit zwei verschiedenen Federkonstanten aneinanderstoßen. Dazu wurde die zwei-dimensionale Fläche in 64 gleichgroße Teilbereiche geteilt. Dadurch kann ein Ring, der innen sowie außen mit Bereichen, die eine höhere Federkonstante aufweisen als der Ring selbst, umgeben ist, erstellt werden. Dabei wird für die Teilbereiche jeweils eine eigene Federkonstante definiert. Die Aufteilung der 64 Teilbereiche ist in nachstehender Abbildung leicht erkennbar. Im grünen Ring ist die Federkonstante geringer als in den blau eingefärbten Bereichen.

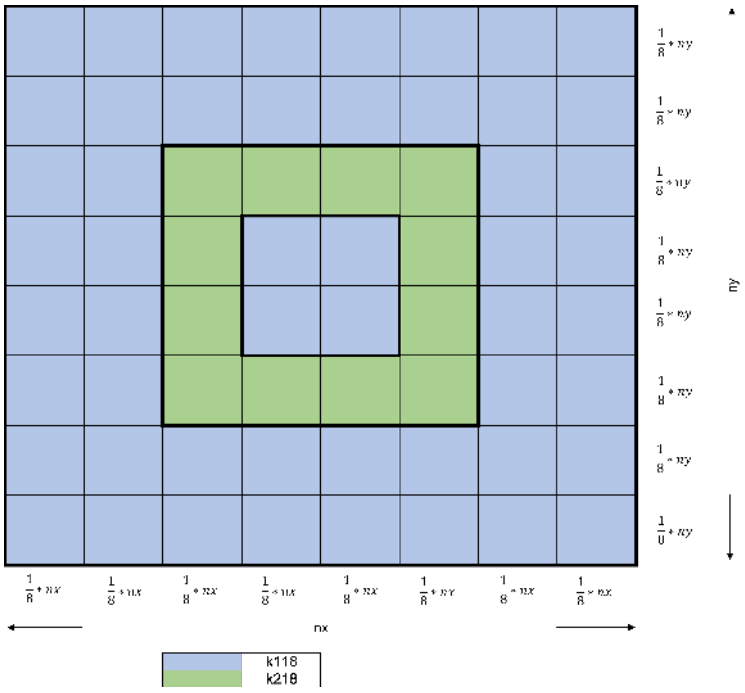


Abbildung 21: Veranschaulichung der k^x, k^y Matrix

In Matlab wurde vom zweidimensionalen Code Massen auf einem Torusäusgegangen. Anschließend musste die ursprüngliche Definition von k^x und k^y auskommentiert werden. Nun werden $k118$ und $k218$ definiert. Hierbei handelt es sich um Federkonstanten-Matrizen, die eine Größe von 1/64 der ursprünglichen Matrizengröße aufweisen. Nun werden k^x und k^y mithilfe zuvor definiertem $k118$ und $k218$ erzeugt. Dabei hat $k118$ den Wert 1 und $k218$ den Wert 0.1.

```
h1 = figure(1);
    close(h1);
    h1 = figure(1);
    set(h1,'Position',[100 10 500 500]);

    m  = 1;
% Massen auf einem Torus
            nx = 8*10; ny = 8*10; nt = 1000; dt = 0.1; th = 0.9;
            f  = 0;
            u0 = zeros(nx,ny);
      u0(round(nx/2-nx/64):round(nx/2+nx/364),round(nx/2-nx/64):round(nx/2+nx/64))=1.0;
            u1 = zeros(nx,ny);
            dx = spdiags(ones(nx,1), 0,nx,nx) ...
               - spdiags(ones(nx,1),-1,nx,nx);
            dx(1,nx) = -1;
            iy = speye(ny);
            Dx = kron(iy,dx);
            dy = spdiags(ones(ny,1), 0,ny,ny) ...
               - spdiags(ones(ny,1),-1,ny,ny);
            dy(1,nx) = -1;
            ix = speye(nx);
            Dy = kron(dy,ix);
            a = 0.1;
            b = 1.0;
            k118 = b*ones(nx/8,ny/8);
            k218 = a*ones(nx/8,ny/8);
            kx = [k118,k118,k118,k118,k118,k118,k118,k118; ...
                  k118,k118,k118,k118,k118,k118,k118,k118; ...
                  k118,k118,k218,k218,k218,k218,k118,k118; ...
                  k118,k118,k218,k118,k118,k218,k118,k118; ...
                  k118,k118,k218,k118,k118,k218,k118,k118; ...
                  k118,k118,k218,k218,k218,k218,k118,k118; ...
                  k118,k118,k118,k118,k118,k118,k118,k118; ...
                  k118,k118,k118,k118,k118,k118,k118,k118;];
            ky = [k118,k118,k118,k118,k118,k118,k118,k118; ...
                  k118,k118,k118,k118,k118,k118,k118,k118; ...
                  k118,k118,k218,k218,k218,k218,k118,k118; ...
                  k118,k118,k218,k118,k118,k218,k118,k118; ...
                  k118,k118,k218,k118,k118,k218,k118,k118; ...
                  k118,k118,k218,k218,k218,k218,k118,k118; ...
                  k118,k118,k118,k118,k118,k118,k118,k118; ...
                  k118,k118,k118,k118,k118,k118,k118,k118;];
            K = -Dx'*spdiags(kx(:),0,nx*ny,nx*ny)*Dx ...
                -Dy'*spdiags(ky(:),0,nx*ny,nx*ny)*Dy;
            r = 0.0;
            c2 = r*ones(nx,ny);
          end

    Z  = sparse(nx*ny,nx*ny);
    I  = speye(nx*ny);
    C  = spdiags(-c2(:),0,nx*ny,nx*ny);
    A  = [Z,I;K/m,C/m];
```

```
F  = (f/m)*[zeros(nx*ny,1);ones(nx*ny,1)];
I2 = speye(2*nx*ny,2*nx*ny);

U0 = [u0(:);u1(:)];

i = 0;
    U = U0;
    u = reshape(U(1:nx*ny),nx,ny);
    surf(u);
    axis([0 nx+1 0 ny+1 -1 +1])
    axis off
    drawnow;

for i=1:nt
    U = (I2 - th*dt*A) \ (U + (1-th)*dt*(A*U+F));
    u = reshape(U(1:nx*ny),nx,ny);
     if (mod(i,200) == 0)
      u(round(nx/2-nx/64):round(nx/2+nx/64),round(nx/2-nx/64):round(nx/2+nx/64)) = 1.0;
         U(1:nx*ny) = u(:);
     end
    w = 2;     %Verstärkung der Auslenkung in der Grafik
    surf(w*u);
    axis([0 nx+1 0 ny+1 -1 +1])
    axis off
    drawnow;
end
```

Mit den Parametern a und b lässt sich die Federkonstante für den "grünen Ring" und den restlichen "blauen Bereich" einstellen (grün - a; blau - b). Zusätzlich wurde der Code so abgeändert, dass nach gewissen Zeitabschnitten eine erneute Erregung in der Mitte der Fläche stattfindet.

```
if (mod(i,200) == 0)
     u(round(nx/2-nx/64):round(nx/2+nx/64),round(nx/2-nx/64):round(nx/2+nx/64)) = 1.0;
     U(1:nx*ny) = u(:);
end
```

Diese ständigen Erregungen können zum Beispiel Konflikte oder ärgerliche Gedanken, die einem immer wieder hochkommen und zu einer Belastung werden können, symbolisieren. Es lässt sich klar ein Ring in der Grafik erkennen. Hierbei werden die Wellen derartig gedämpft, dass sie nur mit einer geringen Amplitude in den äußeren Bereich eindringen. Dabei verlaufen die Wellen in diesem Bereich ruhiger als im Zentrum. Der Bereich, in dem die Wellen gedämpft und abgeschwächt werden, soll die Meditation verkörpern. Dadurch gelingt es mithilfe der Meditation, ärgerliche Gedanken in angenehme und ertragbare Gedanken umzuwandeln.

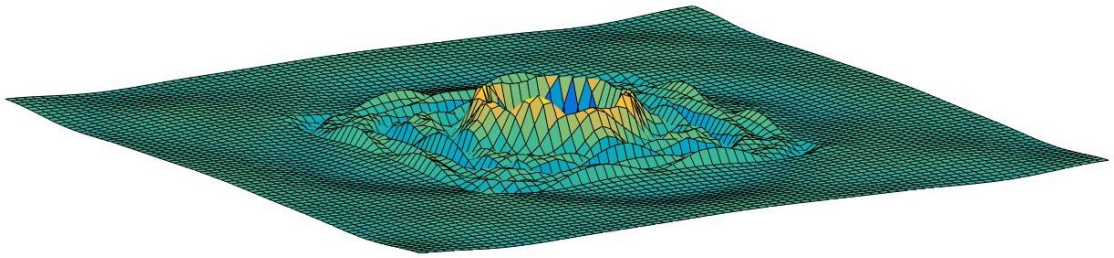


Abbildung 22: Dämpfung durch Meditation

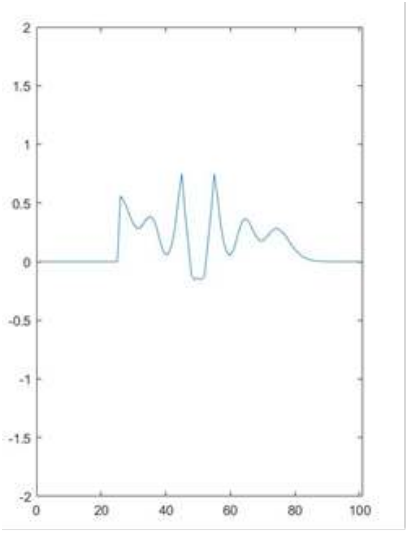
Der Übergang von einem Bereich mit geringer Federkonstante in einen Bereich mit größerer Federkonstante sorgt hauptsächlich für die Dämpfung (Meditation). Die Federkonstante k ist folgendermaßen definiert:

$$k = \frac{F}{\Delta L}$$

Dabei steht F für die Kraft und ΔL für die Auslenkung der Feder. Wenn somit die Federkonstante verkleinert wird, vergrößert sich die Auslenkung. Beim Übergang vom grünen zum blauen Bereich steigt die Federkonstante. Wenn dabei die Kraft F konstant bleibt, verringert sich die Auslenkung, wodurch ruhige Wellenbewegungen entstehen.

8 Neuroplastizität

Wir haben mit einem Modell von Wellen die Neuroplastizität des Gehirns dargestellt. Dafür haben wir durch einen Impuls ausgelöste Wellen eine programmierte Störung überwinden lassen. Unseren Impuls haben wir nicht durch die von außen wirkenden Kräfte, sondern durch die Auslenkung definiert und erschaffen. Wir haben mit dem bereits oben angegeben Code für am Rand freie Massen gearbeitet. Zuerst haben wir unser Modell in der ersten Dimension programmiert. Wenn der Impuls hoch genug ist, wird die Störung überwunden. Ist er es nicht, reflektiert das Hindernis die Welle.



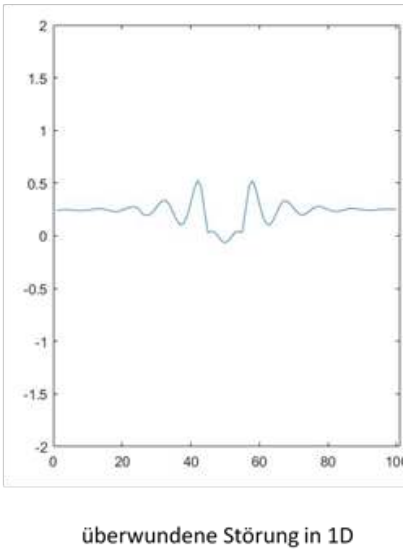


Abbildung 23: Störung 1D

Um das Modell für zwei Dimensionen umzuformen, haben wir die Anfangsauslenkung, also den Anfangsimpuls, wie folgt programmiert:

```
u0(nsx-2:nsx+2,nsy-2:nsy+2)=1;
```

Um den Impuls regelmäßig zu machen, haben wir eine for Schleife programmiert. Des Weiteren haben wir den Impuls auf ein Intervall begrenzt, somit konnten die Wellen in alle Richtungen strömen. Außerdem konnten wir mit diesem Code die Geschwindigkeit der Impulse regulieren, indem wir ein Intervall eingegeben haben. Erhöht man die Auslenkung des Impulses, werden die darauffolgenden Wellen stärker.

```
for i=1:nt
    U = (I2 - th*dt*A) \ (U + (1-th)*dt*(A*U+F));
    u = reshape(U(1:nx*ny),nx,ny);
    surf(u);
    axis([0 nx+1 0 ny+1 -1 +1])

    if (mod(i,20)==0)
        u(nsx-2:nsx+2,nsy-2:nsy+2)=8;
    else
        u(nsx-2:nsx+2,nsy-2:nsy+2)=0;
end
```

Um die Störung zu erzeugen haben wir die Federkonstante (k) an der Stelle

```
x(nbx-2:nbx+1,nby-2:nby+2)
y(nbx-2:nbx+2,nby-2:nby+1)
```

gleich 0 gesetzt. Da wir in 2D arbeiten, haben wir in diesem Code die Werte der zweiten Dimension angepasst, die wir am Anfang des Codes definiert haben. Grundsätzlich setzten wir die Federkonstante, mit Ausnahme des besagten Intervalls, gleich 1, indem sie in diesem Teil jedoch gleich 0 war, erzeugten wir eine künstliche Störung, die die Wellen am Ausbreiten hindern

sollte.
Wir haben die Störung mithilfe einer for Schleife auf ein Intervall begrenzt. Mit der ersten Zeile des Codes haben wir die Stärke der Störung regulieren können. Wir haben die Störung mit 0.2 sehr niedrig eingestellt, daher musste die Auslenkung nicht sehr hoch sein um diese zu überwinden.

```
if (abs(u(nbx+3,nby-2:nby+2))>0.2)
    j1 = nby-3; j2 = nby+3; i=nbx-2:nbx+1;
    for j=nby-2:nby+2
        kx(i,j) = (kx(i,j) + (kx(i,j1)+kx(i,j2))/2)/2;
    end
    i1 = nbx-3; i2 = nbx+3; j=nby-2:nby+1;
    for i=nbx-2:nbx+2
        ky(i,j) = (ky(i,j) + (ky(i1,j)+ky(i2,j))/2)/2;
    end
    K = -Dx'*spdiags(kx(:),0,(nx-1)*ny,(nx-1)*ny)*Dx ...
        -Dy'*spdiags(ky(:),0,(ny-1)*nx,(ny-1)*nx)*Dy;
    A  = [Z,I;K,C];
end
```

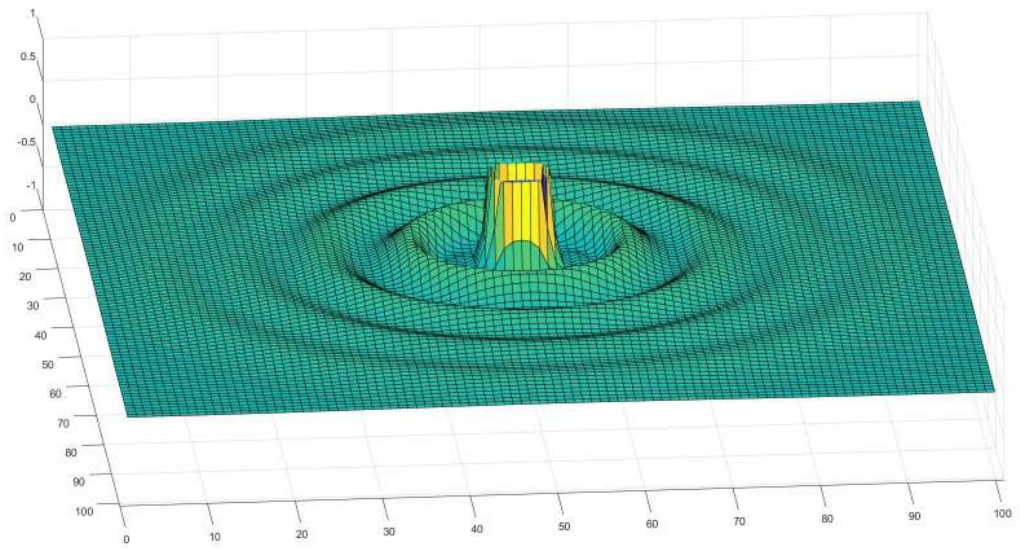


Abbildung 24: blockierte Störung in 2D

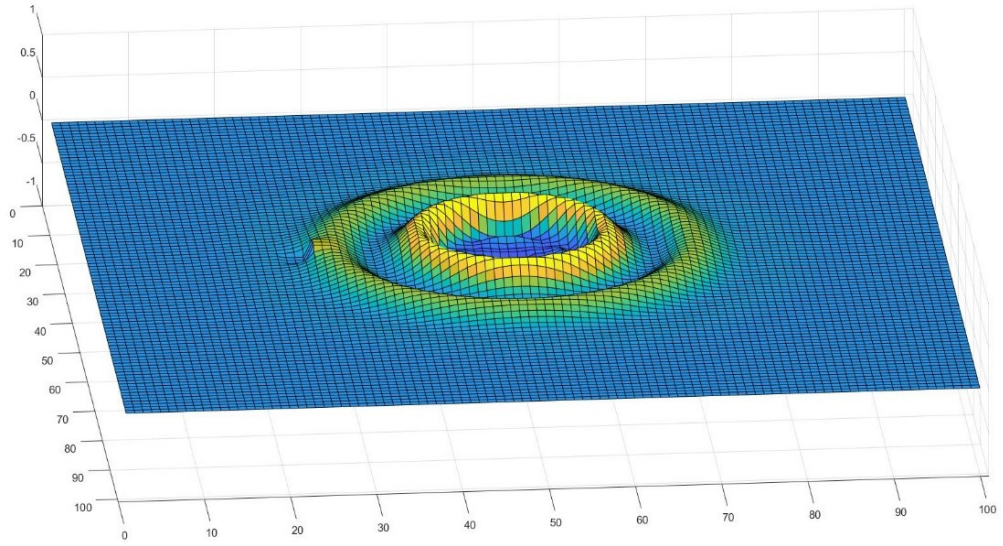


Abbildung 25: überwundene Störung in 2D

Unser Fazit zu diesem Projekt ist, dass man durch Yoga oder Meditation gewisse Störungen im Gehirn überwinden kann. Wenn der Impuls, also die Kraft der Meditation stark genug ist, kann man damit Störungen bzw. alltägliche Probleme im Gehirn überwinden und somit können alle Hirnareale wieder erreicht werden. Dadurch soll es laut Studien gelingen, Stress und negative Gefühle zu reduzieren. Es gilt darüber hinaus, Beobachterin oder Beobachter der eigenen Gedanken und Gefühle zu werden und sie wie Wolken am Himmel zu betrachten. Meditationstechniken zielen darauf ab, schlechte Gedanken und Gefühle als momentanen Zustand zu sehen, der auch wieder vergeht. Man kontrolliert die Verbindung zwischen Reiz und Reaktion.

9 Schlaf- und Wachzustand

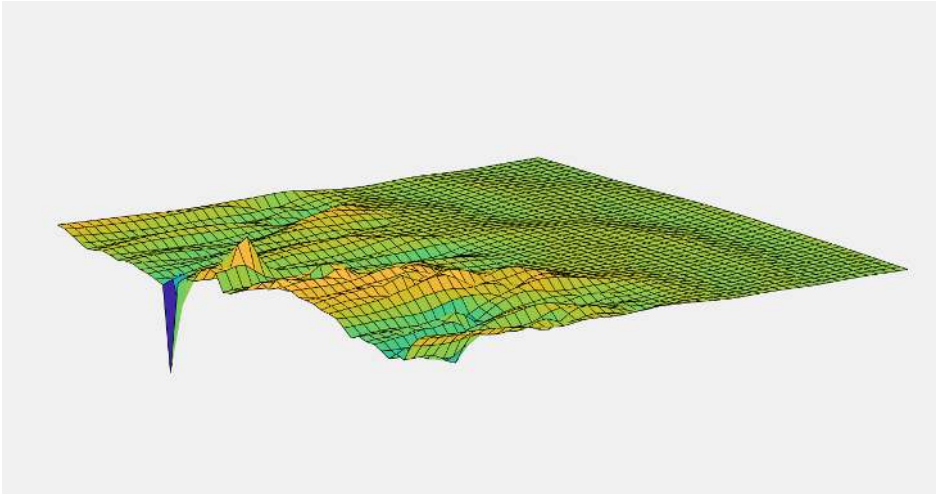


Abbildung 26: Schlaf- und Wachzustand

Das obere Bild stellt den Unterschied zwischen neuronalen Oszillationen im Schlaf- und Wachzustand da. Man kann sich den Übergang zwischen Schlaf- und Wachphase auch als Aufwachen vorstellen. Dabei stellt der aktive linke Teil der Simulation den Wachzustand dar.

```
for i=1:nt
U = (I2 - th*dt*A) \ (U + (1-th)*dt*(A*U+F));
u = reshape(U(1:nx*ny),nx,ny);
% turbulenzen:
if (mod(i,20) == 0)
ir = randi(nx);
if (abs(U(ir)) &lt; 0.1)
U(ir) = U(ir) + (-1)^ir;
end
end
surf(u);
axis([0 nx+1 0 ny+1 -1 +1])
axis off
drawnow;
end
```

Der oben zu sehende Ausschnitt des Codes, der die Simulation generiert, ist teils für die Turbulenzen im linken Abschnitt der Darstellung verantwortlich. Dabei werden die Turbulenzen zufällig auf der Ebene erzeugt und sollen die Störungen in der Wachphase darstellen. Außerdem wird solch ein Impuls alle 20 Zeitintervalle ausgelöst.

10 Fazit

Durch dieses Projekt nehmen wir eine positive Einstellung bezüglich Meditation mit. Während der Woche meditierten wir sogar selbst. Kurze Meditationseinheiten können unseren Alltagsstress mindern. Wenn wir im Stress sind, beschäftigen uns viele Probleme, die eigentlich keine wirklichen sind. Durch Meditation ist man mehr im Einklang mit sich selbst sowie entspannter. Des Weiteren wird in der Schule vorgegeben, was wir herausfinden müssen und wie wir zu unserem Ergebnis kommen. In dieser Woche konnten wir selbst die Richtung unserer Bestreben wählen. Somit wurde trainiert, dass es nicht immer nur einen Lösungsweg gibt. So mussten wir (selbst) Lösungen suchen, die zu unseren Problemen passten. Es war eine andere Art des Arbeitens. Wir arbeiteten sehr selbständig und sammelten viele neue (Lebens-)Erfahrungen. Matlab führte uns in ein neues Gebiet der Mathematik - Numerik - ein.

Außerdem möchten wir Steve herzlich für sein Engagement und seine Geduld während der gesamten Modellierungswoche (egal ob Tag oder Nacht) danken!