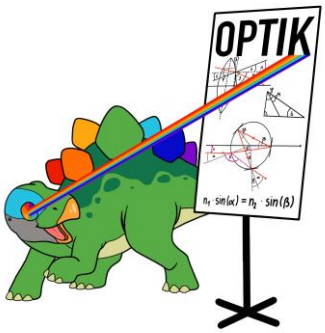
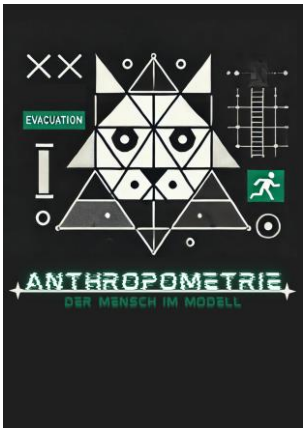


Steirische
Modellierungswoche

20. Woche der Modellierung

mit Mathematik



Dokumentationsbroschüre 8.2.-14.2.2025

Woche der Modellierung mit Mathematik

JUFA Leibnitz, 8.2.-14.2.2025

Weitere Informationen:

https://mathematik.uni-graz.at/de/studienservice/modellierungswoche/

Sponsoren und Organisatoren

Koordination

Mag. DDr. Patrick-Michel Frühmann

Mag. Silvia Lebosi

Dr. Benjamin Hackl

Vorwort

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

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

Ähnliche Modellierungswochen gab bzw. gibt es zum Beispiel auch in den USA, in Deutschland oder in Italien. Wir verdanken Herrn Univ.-Prof. Dr. Stephen Keeling den Vorschlag, auch durch die Universität Graz so eine Woche zu veranstalten, und seiner unermüdlichen Organisationsarbeit das tatsächliche Zustandekommen. Es freut mich besonders, in diesem Jahr das zwanzigjährige Jubiläum unter seiner Ägide feiern zu können; wir hoffen, dass wir nach seinem wohlverdienten Ruhestand die Modellierungswoche ähnlich erfolgreich weiterführen können!

Besonders wichtig war in den vergangenen Jahren auch die Unterstützung durch den langjährigen Mentor der Modellierungswoche, Herrn o.Univ.-Prof. Dr. Franz Kappel, der oft auch eine eigene Gruppe mit interessanten Problemstellungen betreut hat. Leider verstarb Prof. Kappel Anfang 2020, so dass er nicht erleben konnte, wie die Woche der Modellierung mit Mathematik nach der Pandemie-bedingten Pause wieder mit großem Zuspruch starten konnte.

Wir danken der Bildungsdirektion für Steiermark, und hier insbesondere Frau FIin Maga. Michaela Kraker, für die Hilfe bei der Organisation und die kontinuierliche Unterstützung der Idee einer Modellierungswoche. Finanzielle Unterstützung erhielten wir von der Karl-Franzens-Universität Graz durch Vizerektorin Univ.-Prof. Dr. Catherine Walter-Laager und Dekan

Univ.-Prof. Dipl.-Ing. Dr.techn. Klemens Fellner, vom Fachdidaktikzentrums für Mathematik und Geometrie der Uni Graz, Leiterin Ass.-Prof. Dr. Christina Krause, von NAWI Graz, vom Forschungsmanagement der Uni Graz, und vom Forschungszentrum "Virtual Vehicle". Ohne diese stetigen großzügigen Zusagen wäre die Modellierungswoche in dieser Form nicht machbar.

Ohne den idealistischen, unentgeltlichen und engagierten Einsatz der direkten Projektbetreuer, Dr. Stefan Reiterer, Florian Thaler, Gabriel Pichlbauer und Dr. Michael Fischer hätte diese Modellierungswoche nicht stattfinden können. Hier sei auch ausdrücklich dem “Virtual Vehicle” und der TU Graz gedankt, die ihren Mitarbeitern diese Teilnahme möglich gemacht haben.

Besonderer Dank gebührt Herrn Mag. DDr. Patrick-Michel Frühmann, der die ganze Veranstaltung betreut und auch die Gestaltung dieses Berichtes übernommen hat. Auch für ihn ist dieses Jahr ein (zehnjähriges) Jubiläum und damit eine Gelegenheit, seinen steten Einsatz für Modellierungswoche zu würdigen, der wesentlich für den Erfolg verantwortlich ist. Wir wünschen ihm für seine zukünftigen, wichtigen, Tätigkeiten alles Gute!

Schließlich gilt unser Dank Frau mag. Dott.ssa Silvia Lebosi und Herrn Dipl-Ing. Dr. techn. Benjamin Hackl für die tatkräftige Hilfe bei der organisatorischen Vorbereitung und Begleitung.

Graz, 28. Februar 2025

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

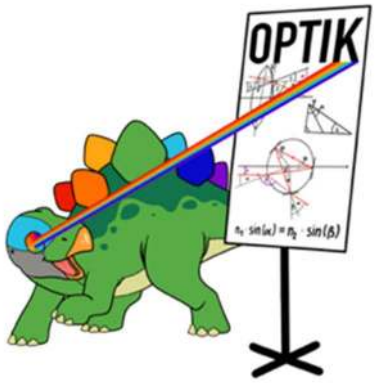
„Do you know what

EXCITEMENT

means?”

(Univ.-Prof. Dr. Stephen Keeling, Deutschlandsberg, 2014)

Modellierungswoche 2025 – Protokoll zum Thema „Optik“



Sara Johary

Julia Leber

Leonor Heidinger

Maria Sudy

Theresa Wölfler

Johanna Wahrbichler

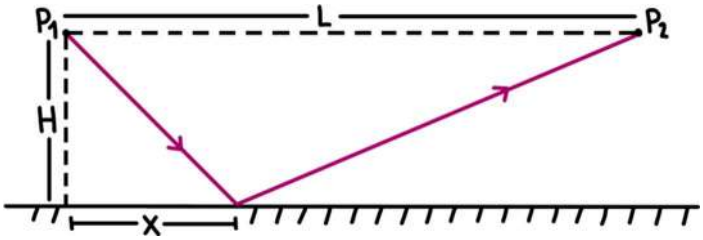
Betreuer: Gabriel Pichlbauer

Inhaltsverzeichnis

1. Reflexions – und Brechungsgesetz

1.1. Reflexion und Winkel

Die erste Frage, die wir uns zum Thema Reflexion stellten, war, warum der Einfallswinkel dem Ausfallswinkel gleicht. Anhand folgenden Beispiels konnten wir ermitteln, dass der Strahl, der von dem Punkt P1 zu dem Punkt P2 reflektiert wird, den kürzesten, beziehungsweise schnellsten, Weg nimmt. Bei besagtem Weg berührt der Strahl das Objekt also genau mittig zwischen P1 und P2.



$$T(x) = \frac{L_1}{c} + \frac{L_2}{c}$$

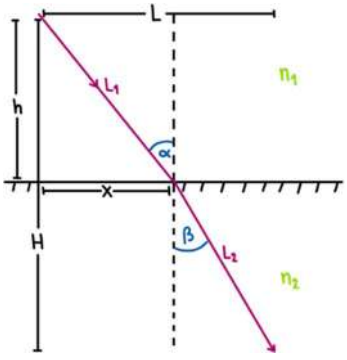
$$T(x) = \frac{\sqrt{H^2 + x^2}}{c} + \frac{\sqrt{H^2 + (L-x)^2}}{c}$$

$$T'(x) = 0 \longrightarrow x = \frac{L}{2}$$

Durch das Aufstellen einer Formel für die Laufzeit des Strahles und das Ableiten und 0 setzen von dieser konnten wir herausfinden, dass in diesem Szenario x=L/2 ist, also wie gedacht die Hälfte der Länge.

1.2. Das Snell´sche Brechungsgesetz

Das Snell´sche Brechungsgesetz stellt eine Beziehung zwischen den Winkeln Alpha und Beta dar. Es besagt, dass $n_1 \cdot sin(\alpha) = n_2 \cdot sin(\beta)$ und war eine der von uns am meisten benutzten Formeln.

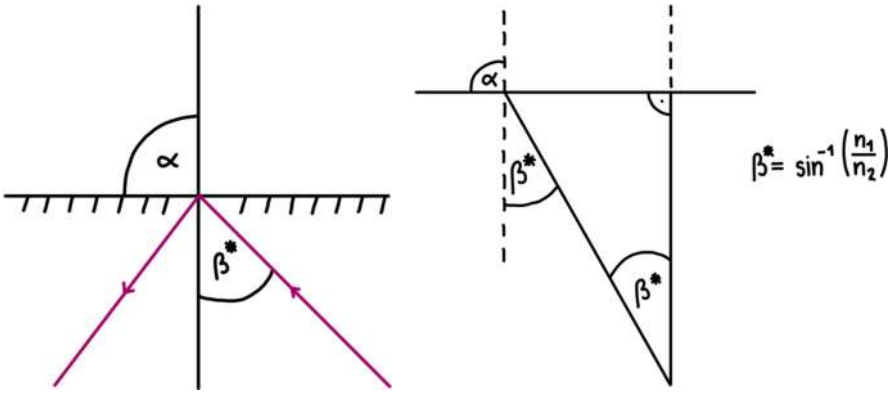


Der Brechungsindex ”n” in einem Medium wird folgendermaßen definiert: $n_{Med} = \frac{C_{vak}}{C_{Med}}$

Hierbei beschreibt C_{Vak} die Lichtgeschwindigkeit im Vakuum und C_{Med} die Lichtgeschwindigkeit im Medium. Wie beim Reflexionsgesetz haben wir wieder die Laufzeit des Strahls berechnet und dessen Minimum bestimmt, und konnten so das Snell´sche Brechungsgesetz nachvollziehbar modellieren.

Wir haben also hergeleitet, dass $T(x) = \frac{\sqrt{h^2+x^2}}{\frac{C_{Vak}}{n_1}} + \frac{\sqrt{(L-x)^2+(H-h)^2}}{\frac{C_{Vak}}{n_2}}$. Wenn wir diese Formel ableiten und mit 0 gleichsetzen, also $T'(x) = 0$, und die Längenverhältnisse mit den Winkelfunktionen ersetzen, erhalten wir das Snell’sche Brechungsgesetz.

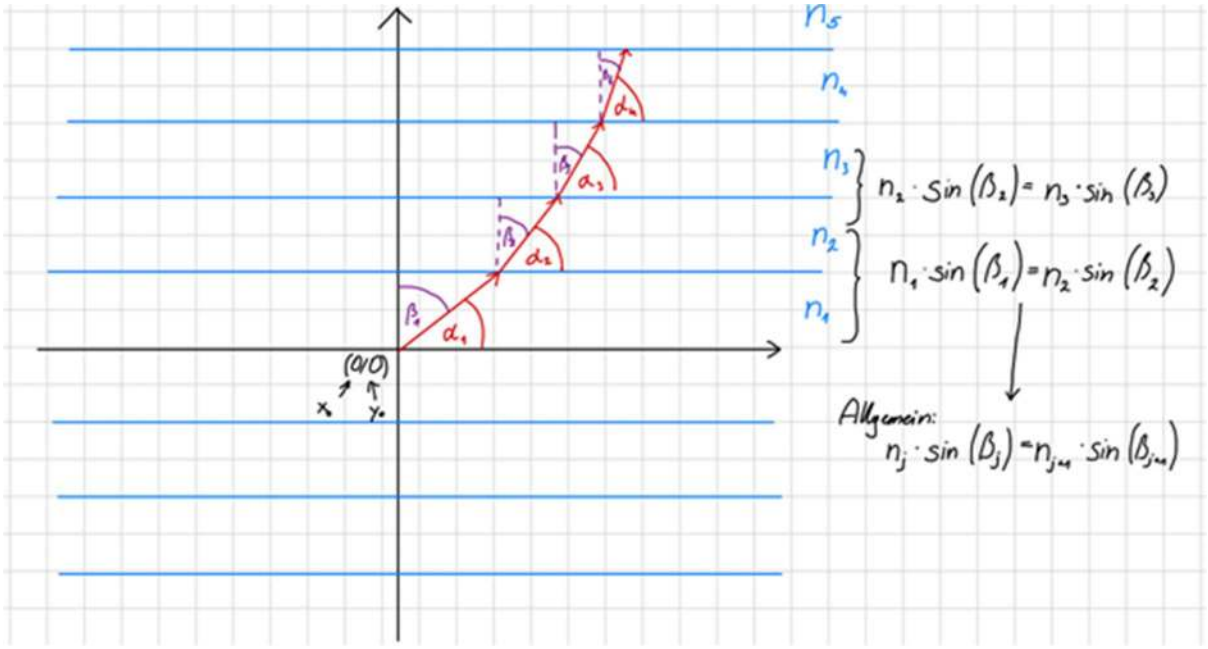
1.3. Totalreflexion



Totalreflexion bedeutet, dass der Lichtstrahl in einem Grenzwinkel der Totalreflexion auf das Medium trifft, so flach ist, dass er nicht mehr gebrochen werden kann, sondern nur mehr reflektiert wird. Er wird also dem Reflexionsgesetz nach im selben Winkel reflektiert, in dem er auch eingetroffen ist. Wenn also β > β* dann findet nur mehr Reflexion statt. Das heißt wenn ein Lichtstrahl in einem größeren Winkel auf ein Medium auftrifft, als der in der Skizze abgebildete Winkel β*, wird eine Totalreflexion stattfinden. Allerdings trifft dies nur bei einem Übergang von einem optisch dichten zu einem optisch dünnen Material.

1.4. Kontinuierlicher Brechungsindex

Grundidee: Approximation durch viele diskrete Schichten

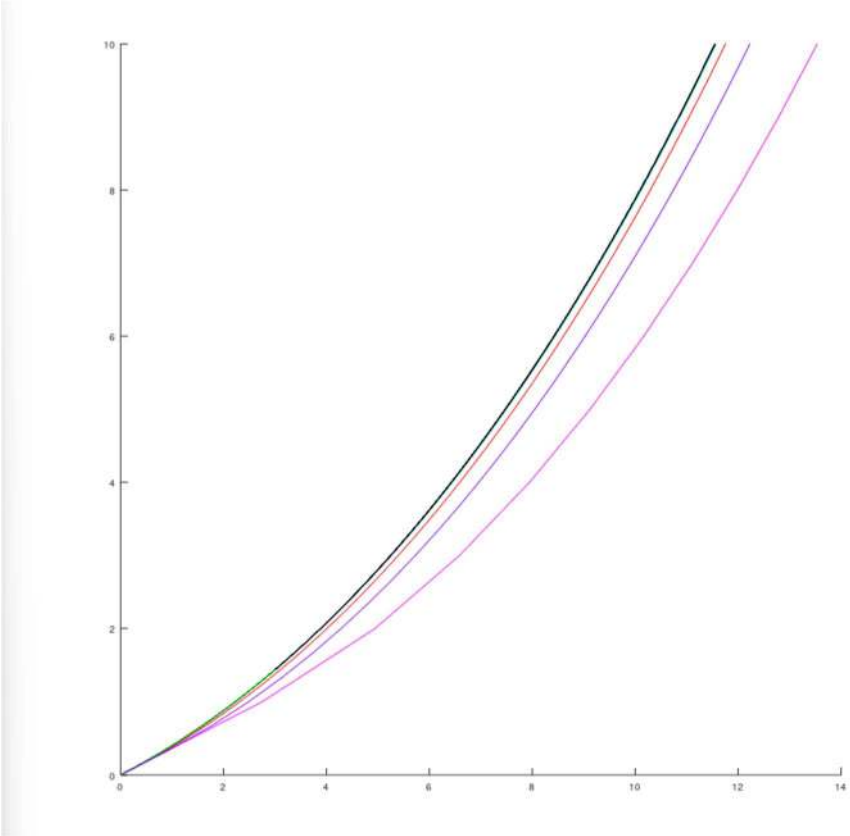


n wird mit der Höhe immer größer —> n5 > n4 > n3 > n2 > n1

Je größer n wird, desto größer wird auch α.

Um unsere Theorie zu überprüfen, haben wir unsere Funktion in Octave eingegeben.

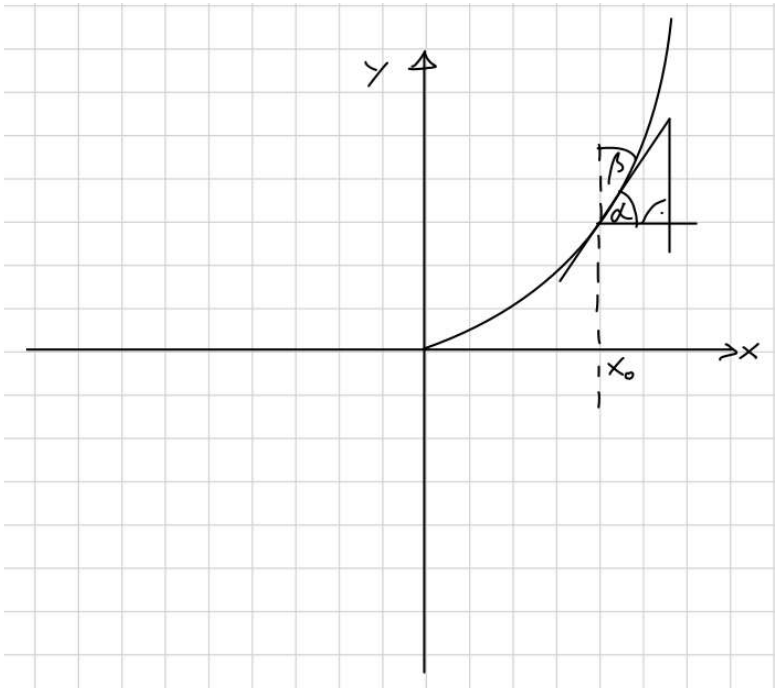
Wir haben schließlich verschiedene Werte für N (die Anzahl der Schichten) eingegeben, und dabei die Schichtdicke kleiner werden lassen. Dabei ist herausgekommen, dass wenn N groß genug wird, sich die Kurve fast nicht mehr ändert. Man sieht dies besonders gut an der schwarzen und an der türkisen Kurve. Bei der schwarzen Kurve haben wir für N = 100 000 eingesetzt, bei der türkisen nur N = 1 000. Für die rote Kurve haben wir N = 100, für die blaue N = 30 und für die pinke N = 10.



```
clear; clc; clf;


x0 = 0;
y0 = 0;
alpha = 20;
alpha = alpha*pi/180;
n0 = 15;
k = 1;
N = 100;   #Anzahl an Schichten
D = 10;
d = D/N;
beta = pi/2 - alpha;

n = @(y)(n0 + k*y);


x = zeros(N + 1,1);
y = zeros(N + 1,1);
x(1) = x0
y(1) = y0

for j = 1:N
  x(j + 1) = (d/tan(alpha)) + x(j);
  y(j + 1) = y(j) + d;
  nj = n(y(j)+d/2);
  beta_new = asin(n0*sin(beta)/nj);
  alpha_new =  pi/2 - beta_new;

 ## if (n0*sin(beta)/nj >= 1)
   ## break;
  ##endif


  alpha = alpha_new;
endfor

hold on
plot(x,y, "color", "r");
```

Wie finden wir exakte Kurven?

Wir haben hier statt β & α, β(x) & α(x) genommen. Die Steigung im Punkt x_0 ist $f'(x_0)$.



Grundformel:
$f'(x_0) = \tan(\alpha(x_0))$

Unsere Idee:

Für diskrete Schichten:

Allgemein: $n_1 * \sin(\beta_1) = n_j * \sin(\beta_j)$
für alle j = 1, 2, 3, …

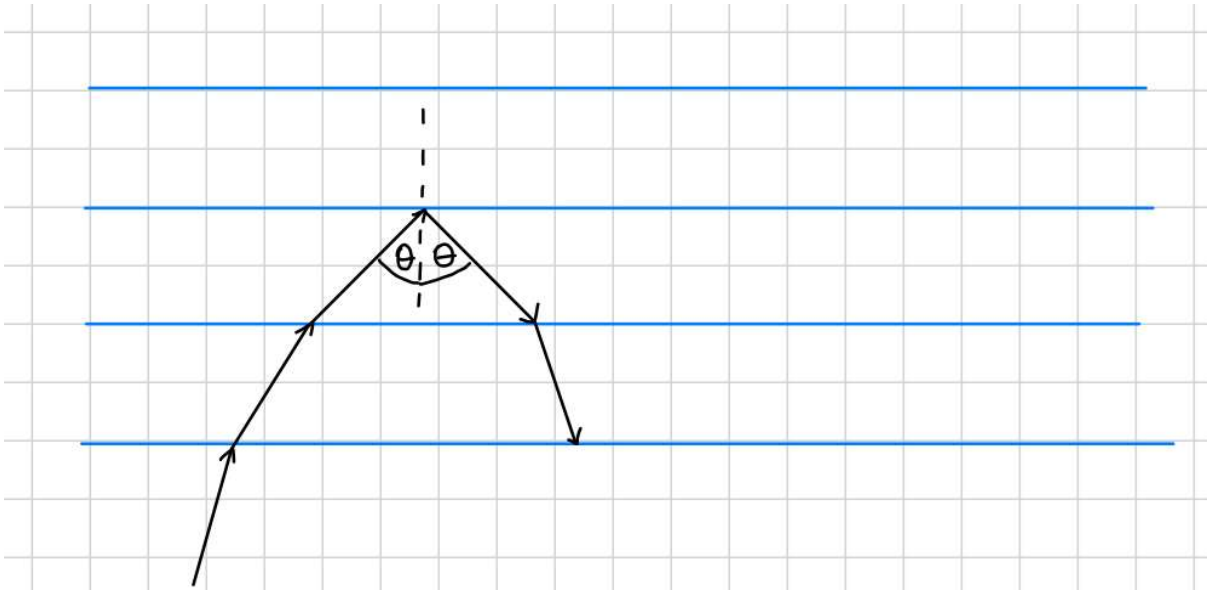
Für kontinuierliche Schichten:

$n_0 * \sin(\beta_0) = n(f(x_0)) * \sin(\beta(x_0))$
n_0 … Index in Punkt (x_0, y_0)

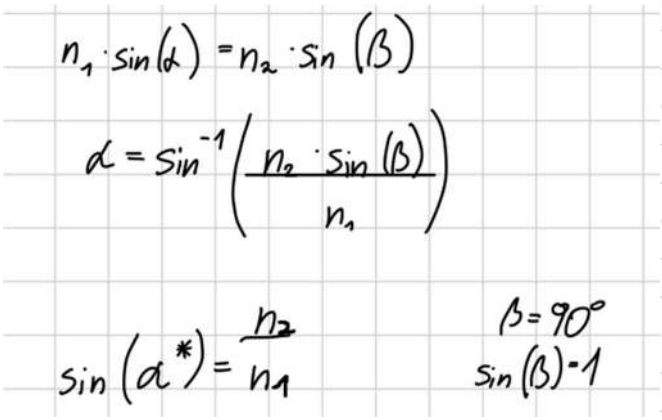
Neue Formel --> Differentialgleichung (Mit Octave & Wolfram Alpha weiter damit gearbeitet):

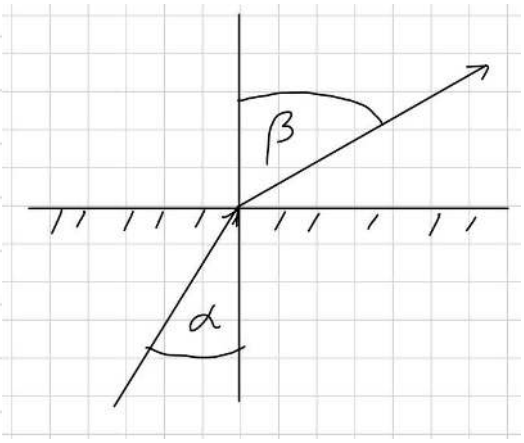
$$f'(x) = \sqrt{\left(\frac{n(f(x_0))}{s_0}\right)^2 - 1}$$ **(Lösung ist die grüne Kurve in Octave)**

Wie überprüft man Totalreflexion?



n wird hier mit der Höhe kleiner, statt größer.





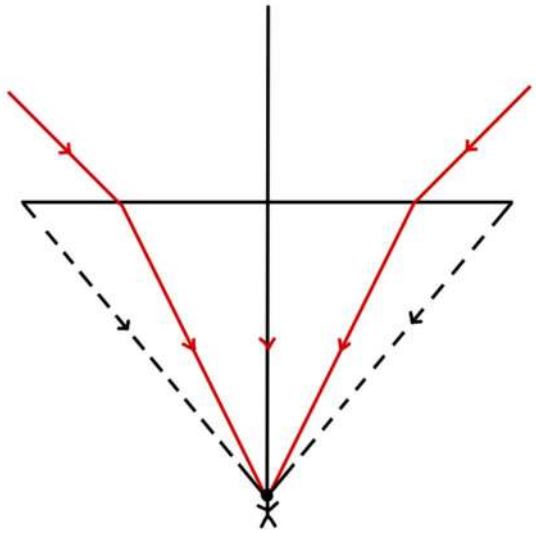
α* wenn β= 90°

Wenn α kleiner als α* wird, kommt es zur Brechung und auch zur Reflexion.

Wenn α größer als α* wird kommt es zur Totalreflexion, was bedeutet, dass es nur zur Reflexion kommt und nicht zur Lichtbrechung.

Wir haben schließlich auch in unserem Code auf Octave k, also die Steigung, negativ gemacht. Hierbei konnten wir dann die Totalreflexion sehen.

1.5 Das Snell´sche Fenster



Das “Snell´s Window” beziehungsweise Snell´sche Fenster ist ein Phänomen, das unter Tauchern sehr bekannt ist. Wenn man unter Wasser nach oben schaut, sieht man alles über der Wasseroberfläche durch ein kreisrundes Loch, während alles rundherum dunkel erscheint. Dieses Phänomen entsteht als Konsequenz der Totalreflexion, denn ab einem gewissen Winkel verlaufen die Strahlen so knapp am Wasser entlang, dass sie uns unter Wasser nicht mehr erreichen können. Dadurch entsteht das Sichtfenster auf der Wasseroberfläche.

2. Licht, Wellennatur des Lichts

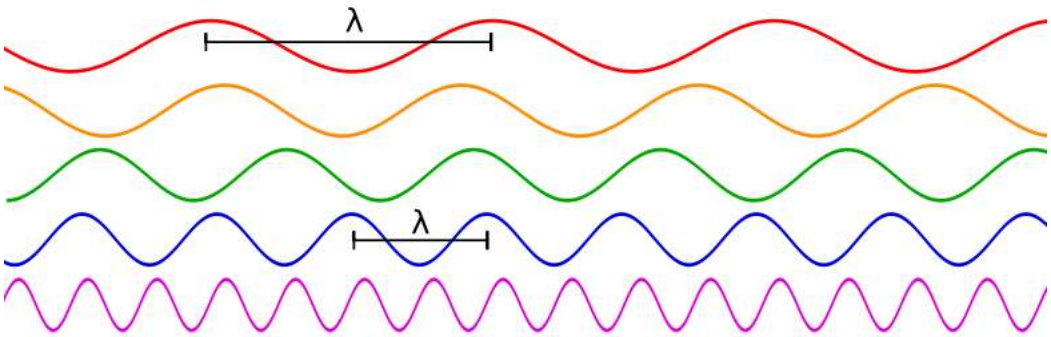
2.1. Was ist Licht?

Eine bewegte elektrische Ladung erzeugt elektromagnetische Wellen. Diese bestehen aus einem schwingenden elektrischen und einem dazu senkrechten magnetischen Feld, und diese Welle bezeichnen wir als “Licht”.

2.1.1. Was sind Farben?

Wenn weißes Licht in seine Bestandteile zerlegt wird, entstehen die verschiedenen Farben, die wir kennen. Diese Spaltung ist allerdings nur möglich, weil jede Farbe eine eigene Wellenlänge bzw. Frequenz besitzt.

Betrachten wir diese Wellen genauer, lassen sich einige spannende Beobachtungen machen:



https://pp.iqo.unihannover.de/doku.php?id=d_optikundatomphysik:elektromagnetische_wellen

Je höher die Wellenamplitude, desto heller erscheint uns das Licht. Wie bereits erwähnt haben Farben unterschiedliche Frequenzen; zum Beispiel schwingt blaues Licht über eine bestimmte Distanz häufiger als rotes Licht.

2.2. Licht Spaltung: Dispersion

Die Wellenabhängigkeit von “n” nennt man Dispersion.

Für die Spaltung von Licht darf der Strahl nicht normal und nicht parallel zu dem Medium verlaufen, sondern muss in einem Winkel auf das Medium treffen.

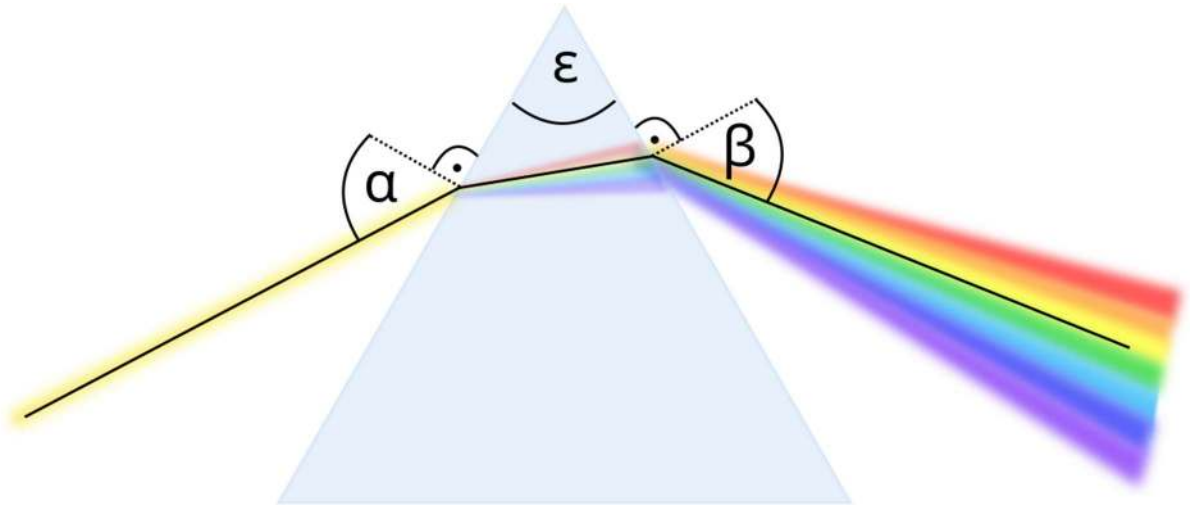
In diesem Medium sorgt der Brechungsindex dann dafür, dass die Lichtwellen gebrochen werden.

Die Farben haben unterschiedliche Brechungsindexe da jede Farbe mit den Frequenzen der Wellen variiert.

Nehmen wir an, ein Lichtstrahl gelangt in ein Medium. Dieser Strahl wird wegen den unterschiedlichen Brechungsindexen schon minimal gespalten. Wenn er das Medium

wieder verlässt, wird er nochmal gespalten und weil wir vorher schon unterschiedliche Spaltungen hatten, gehen die verschiedenen Wellen dann nochmal intensiver auseinander. Das Ergebnis, das wir dann vorfinden ist ein gespaltener Lichtstrahl und deswegen können wir die Farben einzeln wahrnehmen. Würde man die Strahlen wieder zusammenführen, hätte man wieder weißes Licht.

In dieser Abbildung ist dieses Phänomen dargestellt:



https://de.wikipedia.org/wiki/Winkeldispersion

3. Optische Phänomene

3.1. Regentropfen

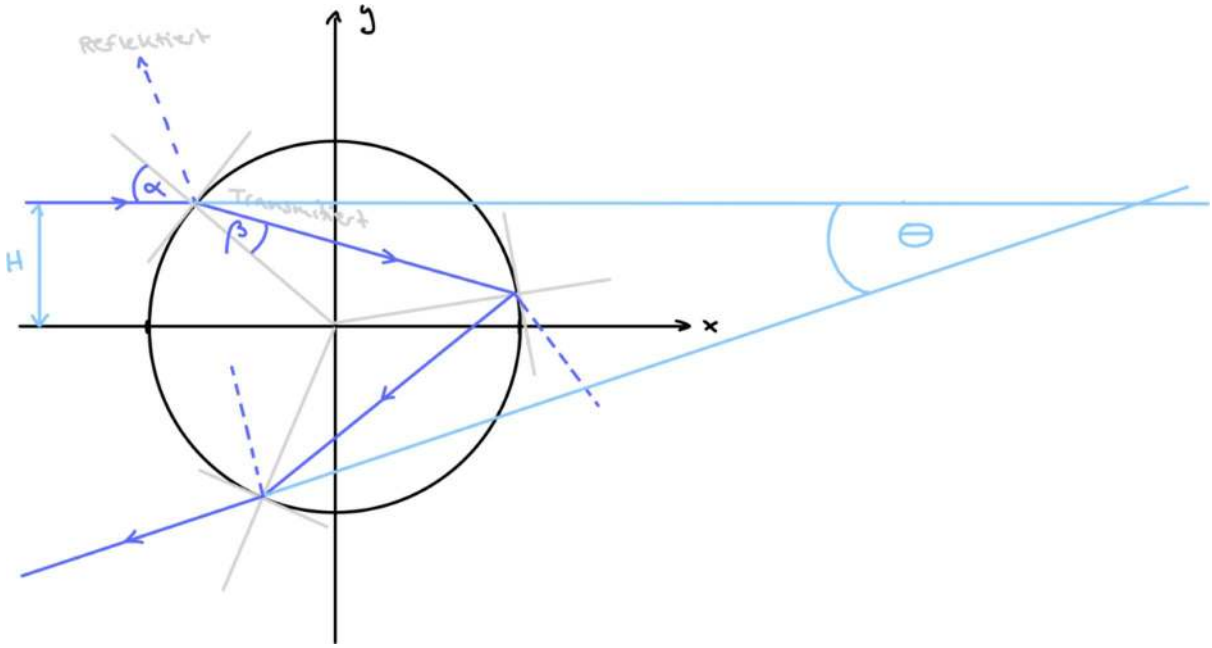
Wir haben uns auch viel mit dem Regentropfen beschäftigt. Wir sind von einem perfekt kugelförmigen Regentropfen ausgegangen, wie in der Skizze zu sehen ist. Von einer gewissen Höhe “H” trifft ein Lichtstrahl auf die Oberfläche des Regentropfens. Ein kleiner Teil wird gemäß dem Reflexionsgesetz reflektiert. Der größere Teil des Strahls, der uns in dem Fall mehr interessiert, wird transmittiert. Der Strahl trifft dann auf der anderen Innenseite von dem Regentropfen auf und wird wieder teilweise reflektiert und teilweise transmittiert. Der reflektierte Teil verläuft dann weiter, bis er wieder auf die Innenseite des Tropfens trifft und anschließend nochmals reflektiert und transmittiert wird.

Unser Modell behandelt nur maximal zwei Reflexionen innerhalb des Regentropfens, in der Realität würde der Strahl aber noch weiter reflektiert und transmittiert werden, bis er immer schwächer wird und an Intensität verliert. Dadurch ist der Lichtstrahl irgendwann fast gar nicht mehr sichtbar, denn immer mehr davon geht durch Transmission an den Grenzflächen verloren.

Für uns interessant waren vor allem diese Winkel:

Alpha(α): zwischen Strahl und Flächennormale

Beta(β): zwischen Flächennormale und erstmals gebrochenen Stahls



Theta(θ): Der entstandene Winkel zwischen der weitergeführten Linie des ursprünglichen Lichtstrahls, und der nach hinten vervollständigten Linie des letzten Ausgangsstrahles.

Unser Ziel war es mit mathematischen Zusammenhängen der Winkel, des Radius und der Höhe, eine Formel für Theta aufzustellen, die wir für späteres Programmieren verwenden können.

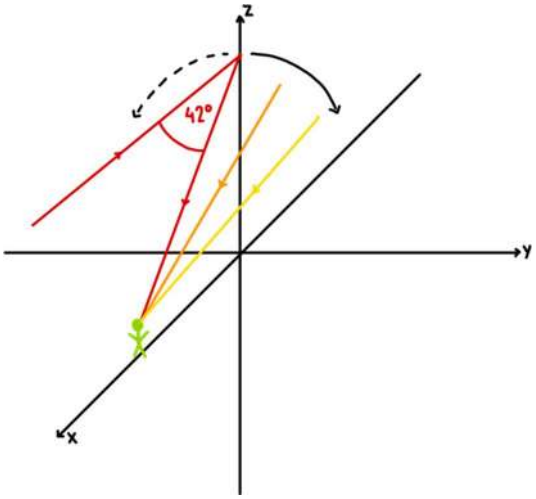
Um den Weg ein bisschen einfacher zu gestalten, haben wir zuerst einmal Alpha in Abhängigkeit der Höhe H bestimmt. Da $\sin(\alpha)=H/R$, konnten wir Alpha herausfinden. Im nächsten Schritt wollten wir die Relation zwischen Alpha und Beta herausfinden. Die haben wir mithilfe des Snell´schem Brechungsgesetzt dargestellt. Der letzte und etwas schwierigere Schritt war dann Theta nur mit Alpha und Beta auszudrücken. Das haben wir mithilfe der inneren gleichschenkligen Dreiecke und der Winkelsumme 180° von Dreiecken herausgefunden.

Schlussendlich sind wir auf die folgende Erkenntnis gekommen: $\theta = 4 * \beta - 2 * \alpha$

3.2. Regenbögen

Im Alltag behaupten wir meistens, dass Regenbögen entstehen, wenn Regen und Sonne aufeinandertreffen. Da steckt allerdings noch viel mehr dahinter und das wollten wir beweisen und modellieren. Wie Licht durch einen Tropfen reflektiert und gebrochen wird haben wir bereits mithilfe von Skizzen und Formeln hergeleitet, jetzt fehlt also nur mehr der tatsächliche Regenbogen. Wenn wir also an einen von der Sonne beleuchteten Regenschauer denken können wir uns vorstellen, dass in jedem einzelnen Tropfen das Licht in alle Farben gespalten wird. Da aber jeden Menschen andere reflektierte Lichtstrahlen treffen, ist es klar, dass keiner den gleichen Regenbogen sehen kann. Außerdem ist die Helligkeit eines Strahles winkelabhängig, denn wenn wir uns das ganze als Graph ansehen gilt: je flacher die Steigung der Kurve an der Stelle, an der der Strahl auftrifft, desto schmäler der Winkel und desto heller die Farbe. Rot wird mit einem Winkel von 42° reflektiert, alle weiteren Farben in einem immer minimal kleinerem Winkel.

Aber wie entsteht überhaupt die Form des Regenbogens?



Wir haben uns das Ganze mit folgender Skizze vorgestellt. Das Licht trifft den Regentropfen in einem 42° Winkel und wird zu unserem Auge reflektiert. Ein weiterer Tropfen weiter rechts muss nun aber auch weiter unten sein, um unser Auge im richtigen Winkel tatsächlich zu treffen. Der nächste Tropfen rechts muss wieder weiter unten sein und so weiter. So entsteht dann der tatsächliche Bogen des Regenbogens. In der Skizze kann man dies auch besser erkennen. Die Farben entsprechen allerdings nicht den Farben des Regenbogens, sondern sind nur zur Veranschaulichung gedacht.

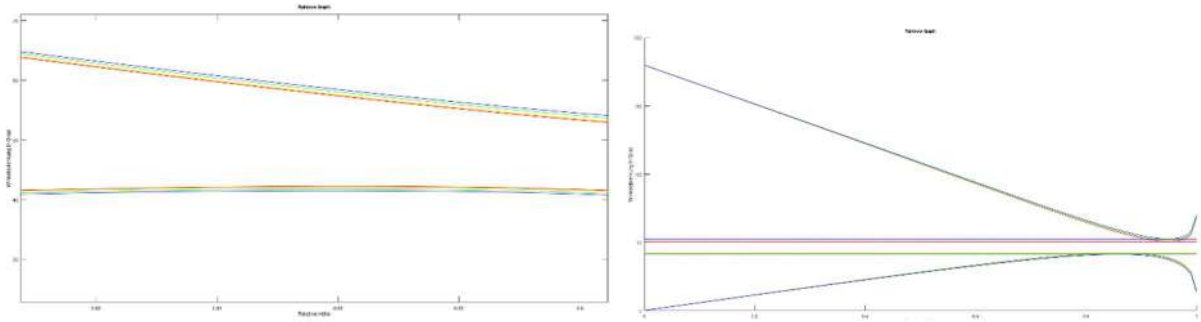
3.3. Programmieren eines Regenbogens in Octave

Nachdem wir in der Theorie also verstanden haben wie Regenbögen funktionieren, wollten wir versuchen, anhand der erstellten Formeln eine Art Regenbogen zu programmieren. Dafür haben wir das Programm Octave benutzt.

Durch das oben bereits erklärte Tropfenbeispiel konnten wir herausfinden, dass $\phi(x) = 4 \cdot sin^{-1}\left(\frac{x}{n}\right) - 2 \cdot sin^{-1}(x)$. Mithilfe dessen konnten wir also folgende Parameter für unser Programm aufstellen.

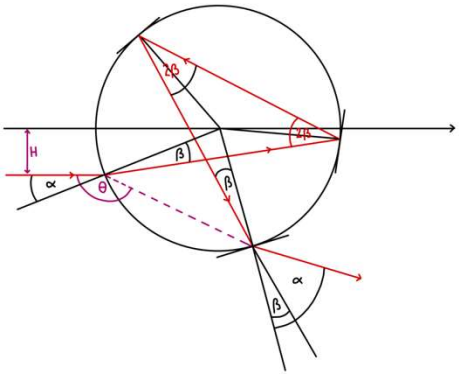
```
1  clear; clc;
2  n = 1.333;
3  x=linspace(0,1,100);
4  y = (4*asin(x/n)- 2*asin(x))*180/pi;
5  plot(x,y);
6  disp(max(y));
7
```

Durch das Herausfinden der verschiedenen Wellenlängen von Farben und deren Brechungsindex in Wasser konnten wir pro Farbe eine eigene Kurve erstellen. Mithilfe der Formel für einen doppelten Regenbogen, $\phi(x) = 2 * sin^{-1}(x) - 6 \cdot sin^{-1}\left(\frac{x}{n}\right)$, konnten wir zusätzlich zu den normalen Regenbogenkurven auch die des Bogens zweiter Ordnung plotten. Das Problem: obwohl wir jeweils nur etwa fünf der sieben Farben des Regenbogens erstellen wollten, war niemand in der Lage, eine der Linien orange zu machen. Denn während man für eine rote Linie einfach nur das Kürzel “r” schreiben muss, braucht es für orange die Kombination “color”,[1,0.6,0], also den RGB code. Nachdem wir es allerdings geschafft hatten, das herauszufinden, konnten wir letztendlich unseren Regenbogen vervollständigen.



Der Einfachheit halber ist hier nur ein kleiner Teil der Kurven abgebildet, damit man die Farben besser erkennen kann. Hier kann man nun zwei Dinge erkennen: einerseits die invertierten Farben des Doppelregenbogens und andererseits das sogennante Alexander´s Band.

3.4. Doppelregenbogen und Alexander´s Band



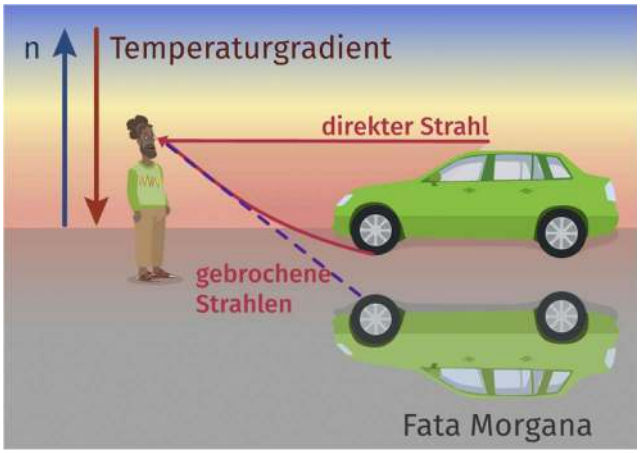
In dieser Skizze sehen wir einen Tropfen, in dem der Lichtstrahl doppelt reflektiert wird bevor er den Regentropfen endgültig verlässt. Wir konnten wie beim ersten Regenbogen ermitteln, dass bei doppelter Reflexion $\phi(x) = 2 \cdot sin^{-1}(x) - 6 \cdot sin^{-1}\left(\frac{x}{n}\right)$ gilt. Bei einem Doppelregenbogen wir also wie in der Skizze das Licht doppelt reflektiert, was im Endeffekt zu invertierten Farben des zweiten Regenbogens führt.

Alexander´s Band ist bekannt als der dunkle Streifen zwischen dem ersten und dem zweiten Regenbogen. Wie man anhand des obigen Plots erkennen kann, ergibt sich dieses aus der Tatsache, dass zwischen 42° und 50° überhaupt kein Licht rückreflektiert wird. Das könnte erst für Strahlen geschehen, die innerhalb des Tropfens öfters als zweimal reflektiert werden, diese Strahlen erscheinen aber bereits weit dunkler als bei einmaliger oder doppelter Reflexion.

3.5. Fata Morgana

Eine Fata Morgana ist ein, durch Ablenkung des Lichts an unterschiedlich warmen Luftschichten auf dem Fermat’schen Prinzip basierender optischer Effekt, den wir mithilfe der hergeleiteten gekrümmten Strahlengängen und Totalreflexion an Luftschichten erklärt werden kann.

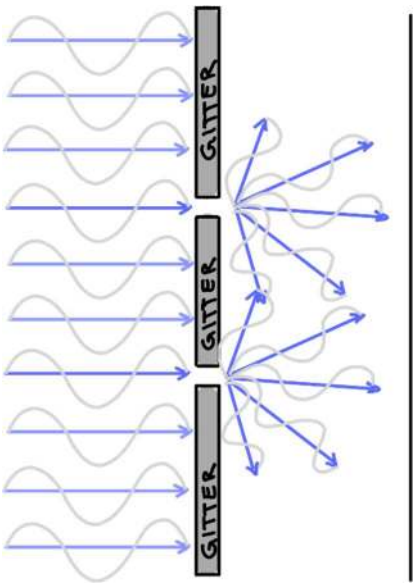
Ist die Temperatur höhenabhängig wie etwa bei einer sehr heißen Straße, so ist auch der Brechungsindex höhenabhängig. Er wird wie in der Abbildung dargestellt mit der Höhe größer. Anhand unserer Überlegungen zu inhomogenen Medien ist klar, dass sich die Lichtstrahlen zu krümmen beginnen

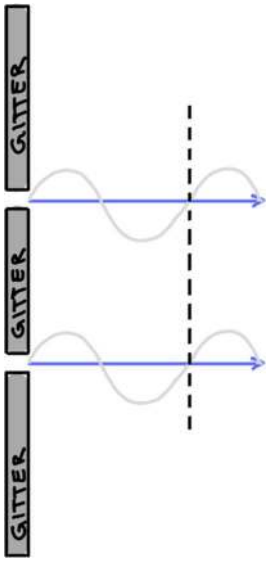


Da unser visuelles System aber die Bilder immer so wahrnimmt, als würden die Strahlen geradlinig verlaufen, ergibt sich die Illusion einer Spieglung an der Straße. In diesem Fall spricht man von einer „hot air mirage“, wir haben aber auch Beispiele von „cold air mirages“ gesehen, die zum Beispiel auf hoher See beobachtet werden können, und bei denen die Temperatur mit der Höhe steigt und n daher sinkt.

3.6. Beugungsgitter

Ein Beugungsgitter ist ein Gitter mit einer gewissen Anzahl an Rillen pro Millimeter, wobei die Distanz zweier Gitterstriche in der Größenordnung der Wellenlänge sein muss um eine Farbaufspaltung wahrnehmen zu können. Um dieses Gitter zu vereinfachen, gehen wir jetzt allerdings nur von einem Gitter mit zwei Rillen aus, wie unten in der Skizze zu sehen ist. Nehmen wir jetzt an, dass von der einen Seite auf so ein Gitter Licht auftrifft. Das Licht wird Großteils wieder abprallen und reflektiert oder absorbiert werden und nur durch die zwei Spalten kommen die Strahlen zur anderen Seite durch. Diese Strahlen gehen in alle Richtungen und treffen auf der Wand dahinter auf.





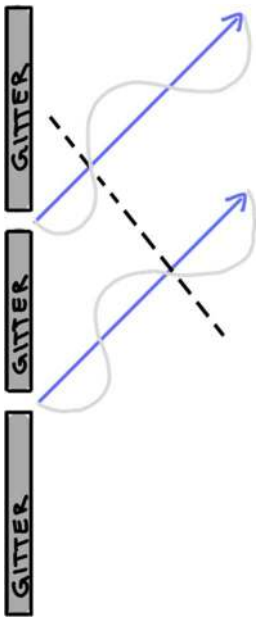
Wenn wir jetzt davon ausgehen, dass der Abstand zwischen Gitter und Wand sehr groß ist, können wir davon ausgehen, dass wenn zwei Wellen von den unterschiedlichen Öffnungen aufeinandertreffen, fast parallel sind.

Hier sind zwei Optionen was nun passieren könnte:

1. Wellen sind parallel und schwingen genau gleich: konstruktive Interferenz

Hergeleitete Formel: $m * \lambda = d * \sin(\alpha)$, (m ist ganze Zahl)

Diese Wellen werden dann vereint und weil sie genau die gleiche Wellenfrequenz haben, addieren sie sich zusammen und ergeben dann eine gemeinsame doppelt so große Welleamplitude. Diese erscheint für uns heller und intensiver.



2. Wellen sind parallel, schwingen gleich aber versetzt um genau eine halbe Wellenlänge: destruktive Interferenz

Hergeleitete Formel: $m * \lambda + \lambda /2 = d * \sin(\alpha)$, (m ist ganze Zahl)

Diese zwei Wellen werden sich genau eliminieren da sie genau gegenteilig verlaufen. Wenn Welle1 ein Maximum hat, ist Welle2 genau auf ihrem Minimum und vice versa. Auf der Wand wird kein Licht zu sehen sein wegen der Eliminierung.

Zwischen den Punkten konstruktiver und destruktiver Interferenz steigt beziehungsweise fällt die Intensität in Form einer kontinuierlichen Kurve.

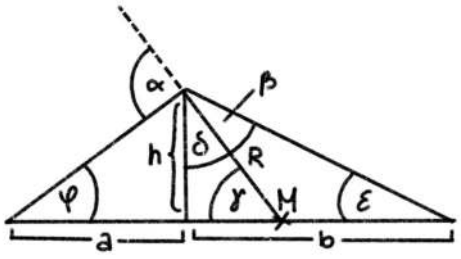
Das Verteilen der Farben kann man begründen durch die unterschiedlichen Winkel die von den Wellenlängen abhängen. Das sieht man in den zwei hergeleiteten Formeln.

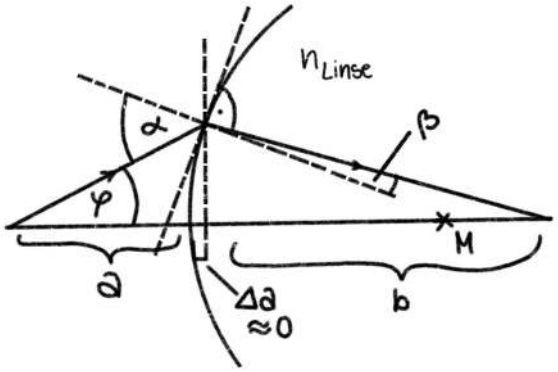
4. Die Linse

4.1. Die Funktionsweise einer Linse

Die graphische Darstellung einer Linse entsteht durch den Schnitt von 2 Kreisen mit zwei verschiedenen oder denselben Radien, und unterschiedlichem Kreismittelpunkt (R oder R1 & R2).

Das erste Ziel war es, eine Relation zwischen der sogenannten Objektweite “a”, welche die Entfernung eines Objektes zur Linse beschreibt, und der Bildweite “b”, welche die Entfernung des entstehenden Bildes zur Linse beschreibt, zu finden.





4.2. Die Flächennormale

Bei Brechung durch gekrümmte Flächen,
wie z.B. eine Linse, wird immer der Winkel von der Flächennormale zum gebrochenen Strahl gemessen, damit auch in diesem Szenario das Snell’sche Brechungsgesetz angewendet werden kann.

4.3. Die Brennweite

Mithilfe geometrischer Überlegungen haben wir hergeleitet, dass die Brennweite einer Linse sich aus den folgenden Parametern berechnen lässt: Krümmungsradien (R1, R2) und Brechungsindex (n).

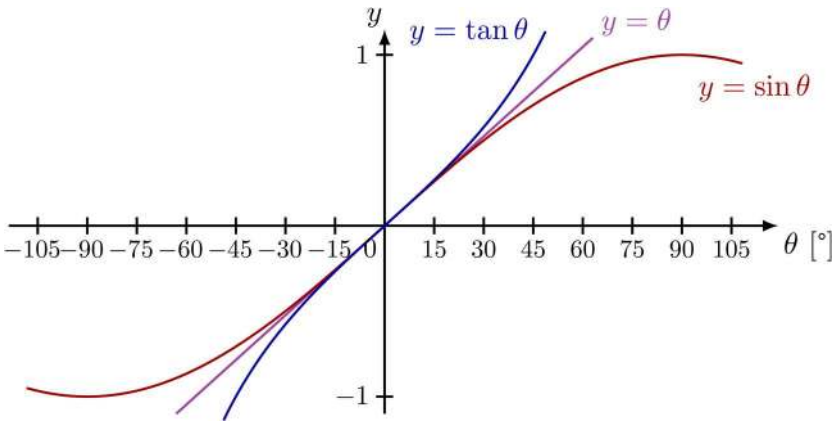
$$f = \frac{1}{(n-1)} \cdot \frac{R1 \cdot R2}{(R1 + R2)}$$

Die Brennweite ist der Abstand des Brennpunkts von der Linsenmitte. Der Brennpunkt ist der Punkt, in dem sich parallel zur optischen Achse verlaufende Strahlen nach der Brechung durch die Linse auf der optischen Achse schneiden.

4.3.1. Die Kleinwinkelannäherung

Der Einfachheit halber nehmen wir an, dass:

$\alpha \cong sin\,(\alpha) \cong tan\,(\alpha)$



(Dies wird anhand der Tayler Entwicklung bewiesen.

$f(x) \cong f(x0) + (x - x0) \cdot f'(x0)$

Wenn man für F(x) entweder x, sin(x), tan(x). Und x_0 = 0 wählt)

Je dicker die Linse jedoch wird, desto ungenauer werden unsere Ergebnisse weil wir diese Kleinwinkelannäherung benutzten.

4.4. Konstruktion der Bilder, Abbildungsgleichung

Bei der Abbildung eines Gegenstandes durch eine Sammellinse hängen Lage und Größe des Bildes von der Entfernung des Gegenstands zur Linse und von deren Brennweite (f) ab.

$$\frac{1}{f} = \frac{1}{a} + \frac{1}{b}$$

4.4.1. Abbildungsmaßstab

Der Abbildungsmaßstab gibt an, wie sehr ein Objekt durch eine Linse vergrößert wird, und ob das Bild richtig orientiert oder um 180° verkehrt herum abgebildet wird. Diesen Abbildungsmaßstab konnten wir leicht mithilfe von ähnlichen Dreiecken herleiten.

$M = \frac{b}{a} = \frac{f}{a-f}$

Wenn $f < a < 2f$

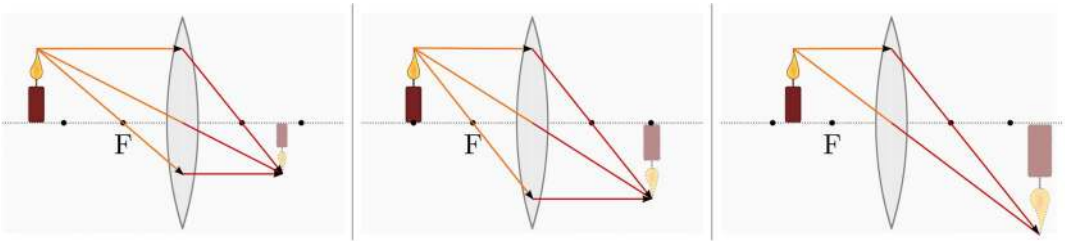
Dann wird ein reelles Bild umgedreht und größer auf der anderen Seite der Linse wieder gespiegelt.

Wenn $a = 2f$

In diesem Fall wird ein reelles Bild generiert welches umgedreht auf der anderen Seite der Linse aber in derselben Größe wie das Objekt erscheint.

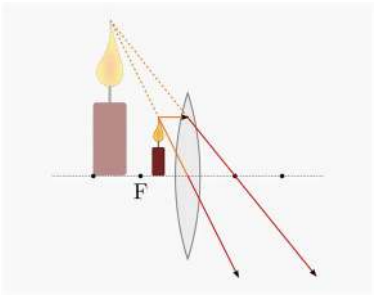
Wenn $2f < a$

Dann wird ebenfalls ein reelles Bild umgedreht auf der anderen Seite der Linse wieder gespiegelt, dieses ist die diesem Fall jedoch kleiner.



Wenn $0 < a < f$

In diesem Fall handelt es sich um ein virtuelles Bild, welches hinter dem Objekt wieder gespiegelt wird. Es ist gleich orientiert und größer als das Objekt.



Bildquellen:

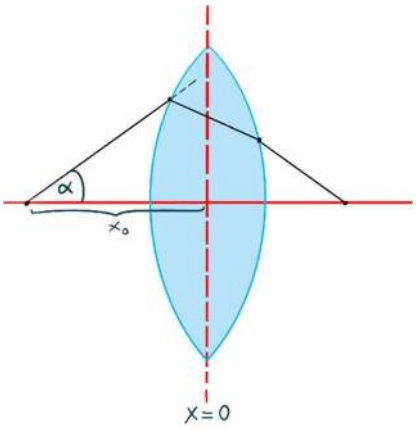
https://www.grund-wissen.de/physik/optik/lichtbrechung.html

https://tikz.net/small_angle_approximation/

4.5. Die Brechung des Lichts - Simulation mit Octave

Um den Weg eines Strahls durch eine Linse zu bestimmen (wie in der untenstehenden Abbildung), haben wir die Software octave.org verwendet.

4.5.1. Das Zeichnen einer Linse



Um eine Linse zu zeichnen, werden zwei Krümmungsradien R1 und R2 definiert, wobei der zweite Kreis um einen bestimmten Wert nach links verschoben wird. Der schneidende Teil der beiden Kreise ist die Linse. Dafür wird folgender Code geschrieben. Die Variablen x_{L1} und x_{L2} bestimmen, wie dünn oder dick die Linse ist (Mittelpunkt der Teilkreise). x1 und x2 beinhalten eine Liste von Werten zu plotten, die innerhalb einer bestimmten Grenze sind, und die ypos1 und ypos2 bzw. yneg1 und yneg2 speichern die entsprechende positive bzw. negative y-Werte von x1 und x2. Am Ende werden diese Koordinaten der Kreise auf einem Graph dargestellt.

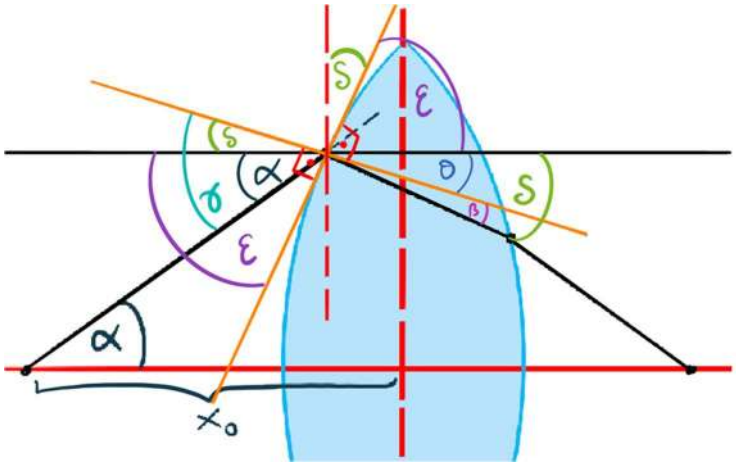
```
clear; clc; clf;

R1 = 10; # Radius des ersten Kreises
R2 = 10; # Radius des zweiten Kreises

m = 100; # Anzahl der Punkte fürs Plotten;

xL1 = 5; # Mittelwert für erste Linse
xL2 = -5 # Mittelwert für zweite Linse

# erster Kreis
x1 = linspace(xL1 - R1, 0); # x-Werte für den Plot
ypos1 = sqrt(R1^2 - (x1 - xL1).^2); # positive y-Werte
yneg1 = -ypos1; # negative y-Werte

# zweiter Kreis
x2 = linspace(0, xL2 + R2, m);
ypos2 = sqrt(R1^2 - (x2 - xL2).^2); # positive y-Werte
yneg2 = -ypos2; # negative y-Werte

hold on
plot(x1,ypos1, "k");
plot(x1, yneg1, "k");

plot(x2,ypos2, "k");
plot(x2, yneg2, "k");
axis([-15, 15, -15, 15])
```

Als Nächstes trifft der Strahl von dem Startpunkt ausgehend auf der Linse. Die Variable start_x speichert einen beliebigen Wert, der die x-Koordinate vom Startpunkt angibt.

Die Steigung des Strahls wird von dem Winkel α berechnet, der beliebig geändert werden kann. Danach wird der Schnittpunkt von dem Strahl und der Linse berechnet und die Koordinaten ermittelt. Der Strahl wird schließlich vom Startpunkt bis zum ersten Schnittpunkt gezeichnet.

```
start_x = -20; # x-Koordinate vom Startpunkt
alpha = 10; # Winkel (in Grad), mit dem der Strahl auf der Linse trifft
alpha = alpha*pi/180; # in Radian umgewandelt

####### erste Linie #######
# Funktion der ersten Linie
k1 = tan(alpha);
d1 = tan(alpha) * abs(start_x);

# Schnittpunkt 1: erste Linie und linke Seite der Linse
s1 = @(x) (k1*x + d1 - sqrt(R1^2 - (x - xL1).^2)); # Funktion des Schnittpunkts
my_zero = real(fsolve(s1, start_x)); # findet die x-Koordinate von dem Schnittpunkt
my_fval = k1 * my_zero + d1; # y-Koordinate von dem Schnittpunkt
line([start_x, my_zero], [0, my_fval], "color", [1, 0.6 ,0]); # zeichnet die erste Linie

```

Danach wird die Steigung nach dem Brechen des Lichtstrahls berechnet. Hierfür wird die Steigung der Tangente an dem ersten Schnittpunkt durch die Ableitung der Funktion k1 berechnet. Mithilfe des Winkels ε werden andere Winkel bestimmt und schließlich θ herausgefunden.



$$k_1(x) = \sqrt{R_1^2 - (x - x_{L1})^2}$$

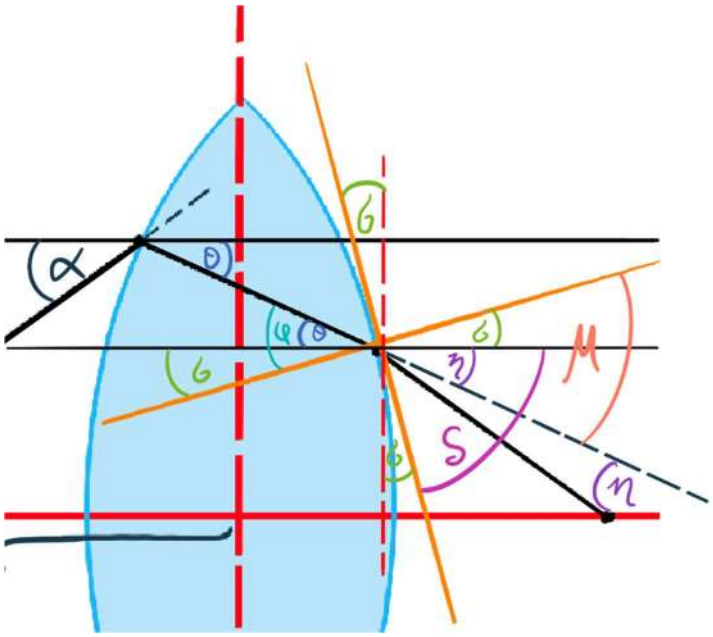
$$k_1'(x) = \frac{(x - x_{L1})}{\sqrt{R_1^2 - (x - x_{L1})^2}}$$

$$\epsilon = arctan(\frac{x_{L1}}{R_1 - x_{L1}})$$

Der Strahl wird mit Brechungsindex n = 1,6 gebrochen. Es wird ein weiterer Schnittpunkt durch die Funktion s2 ausgedrückt und die Linie vom ersten bis zum zweiten Schnittpunkt wird gezeichnet.

```
46  ####### zweite Linie #######
47  n = 1.6; # Brechungsindex n
48  dk1 = @(x) ((xL1 - x)/sqrt(R1^2 - (x - xL1)^2)); # Ableitung der Funktion des Kreises
49  k_tan1 = dk1(my_zero); # Steigung an dem Schnittpunkt (my_zero, my_fval)
50
51  # Rechnen des Winkels theta (zwischen der zweiten Linie und die x-Achse)
52  epsilon = atan(k_tan1);
53  delta = pi/2 - epsilon; # 90° = pi/2
54  gamma = alpha + delta;
55  beta = asin(sin(gamma)/n);
56  theta = beta - delta;
57  theta = -theta;
58
59  # Funktion der zweiten Linie
60  k2 = tan(theta);
61  d2 = my_fval - k2 * my_zero;
62
63  s2 = @(x) (k2*x + d2 - sqrt(R2^2 - (x - xL2).^2)); # Funktion des Schnittpunkts
64  my_zero2 = real(fsolve(s2, 10)); # findet die x-Koordinate vom Schnittpunkt
65  my_fval2 = k2 * my_zero2 + d2; # y-Koordinste vom Schnittpunkt
66  line([my_zero, my_zero2], [my_fval, my_fval2], "color", [1, 0.6 ,0]); # zeichnet die zweite Linie
67
```

Um den Strahl vom dem zweiten Schnittpunkt bis zur x-Achse darzustellen, wird die Steigung der Winkel θ berechnet, womit eine lineare Funktion gestellt wird. Diese Funktion wird mit der x-Achse geschnitten und eine Linie wird dementsprechend gezeichnet.



$$k_2(x) = \sqrt{R_2^2 - (x - x_{L2})^2}$$

$$\rho = -arctan(f_2'(myzero2))$$

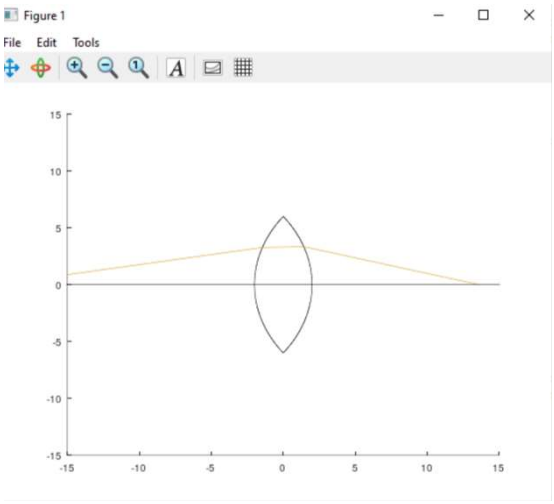
k3 ist die Steigung der Linie und s3 repräsentiert den dritten Schnittpunkt. Der Endpunkt auf der x-Achse hat dann die Koordinaten (my_zero3, my_fval3).

```
69  ####### dritte Linie #######
70  dk2 = @(x) ((xL2 - x)/sqrt(R2^2 - (x - xL2)^2)); # Ableitung der Funktion des zweiten Kreises
71  k_tan2 = dk2(my_zero2); #  Steigung an dem zweiten Schnittpunkt
72
73  # Rechnen des Winkels eta
74  rho = -atan(k_tan2);
75  sigma = pi/2 - rho;
76  phi = sigma + theta;
77  mu = asin(n * sin(phi));
78  eta = mu - sigma;
79
80  k3 = -tan(eta); # Steigung der dritten Linie
81  d3 = my_fval2 - k3 * my_zero2;
82
83  # Schnittpunkt mit der x-Achse
84  s3 = @(x) (k3*x + d3); # eine lineare Funktion, deren Nullstelle gesucht wird
85  my_zero3 = real(fsolve(s3, 0)); # findet die x-Koordinate vom Schnittpunkt
86  my_fval3 = 0;
87  line([my_zero2, my_zero3], [my_fval2, my_fval3], "color", [1, 0.6 ,0]); # zeichnet die dritte Linie
88
```

Zusätzlich wird eine Linie durch die x-Achse gezogen.

```
89  hold on
90  line([-30, 30], [0, 0], "color", "k"); # Linie der x-Achse
91  hold off
92
```

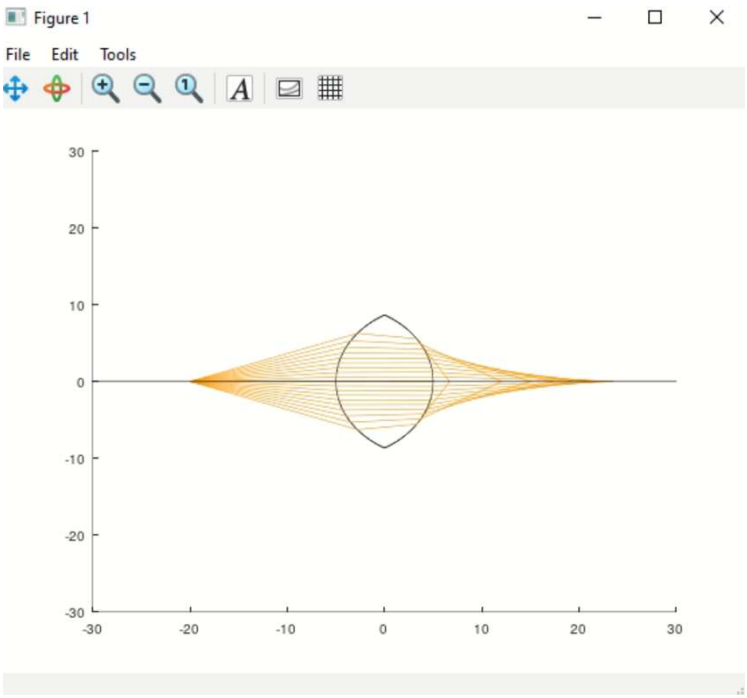
Ergebnis:



Es wird eine Schleife hinzugefügt, die mit unterschiedlichen Winkeln Strahlen aussendet.

```
31  alpha_max = 20;
32  num_angles = 10;
33  angles = linspace(0, alpha_max, num_angles);
34
35  for i = 1: num_angles
36    alpha = angles(i);
```

Darüber hinaus werden die Strahlen im sowohl positiven als auch negativen Bereich dargestellt. Dies ergibt folgende Ausgabe:



4.6. Beispiele einer simulierten Linse

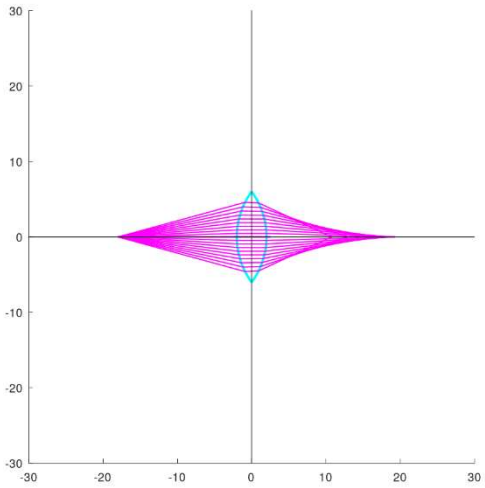
Wir haben zuerst die Brennweite bestimmt um im Programm Octave den eingezeichneten plot mit den Abbildungsgleichung vergleichen zu können.

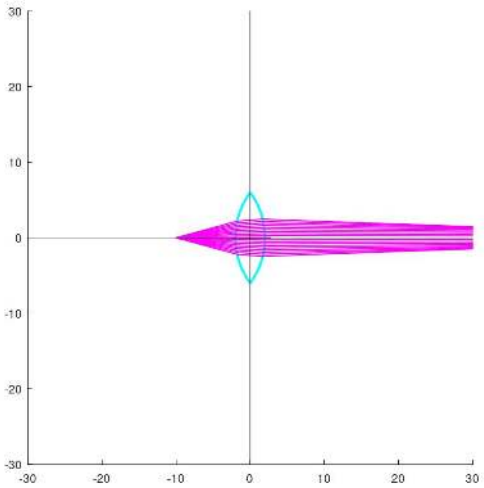
Unsere Formel für die Brennweite war :

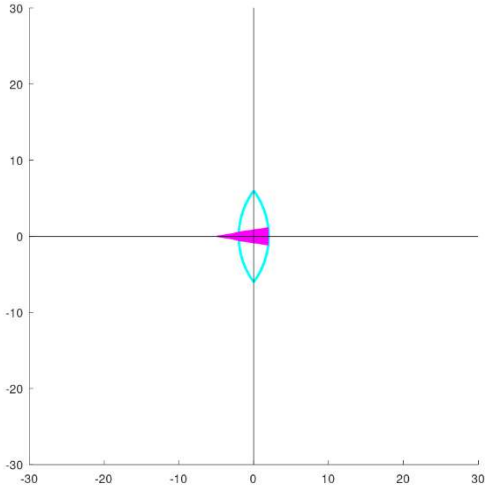
$$f = \frac{1}{(n-1)} \cdot \frac{R_1 \cdot R_2}{(R_1+R_2)} = \frac{R}{2 \cdot (n-1)} \qquad \text{falls } R = R_1 = R_2$$

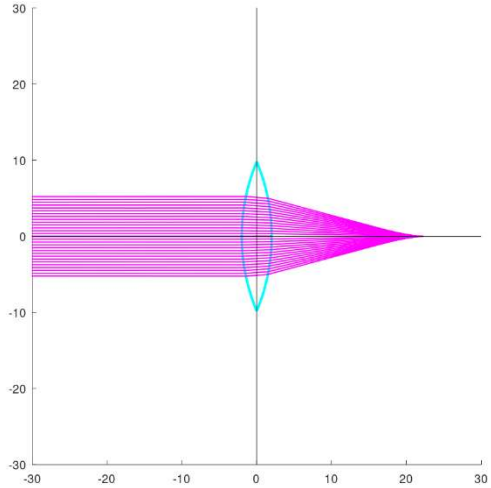
Als Experiment haben wir noch unseren Parameter a geändert.

1) Mit einer sehr dünnen Linse (mit f = 8,3334 —> Brennweite)
   A) a > 2f (Bsp.: mit a = 18)
   B) 2f > a > f (Bsp.: mit a = 10)
   C) f > a > 0 (Bsp.: mit a = 4), Simulation nicht ausgereift genug
   D) a sehr sehr groß (Bsp.: mit a = 10 000)

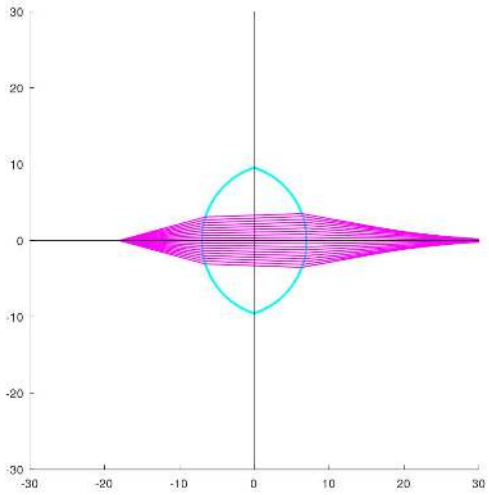


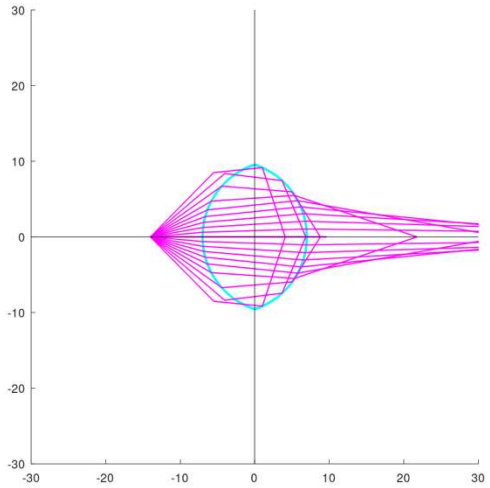


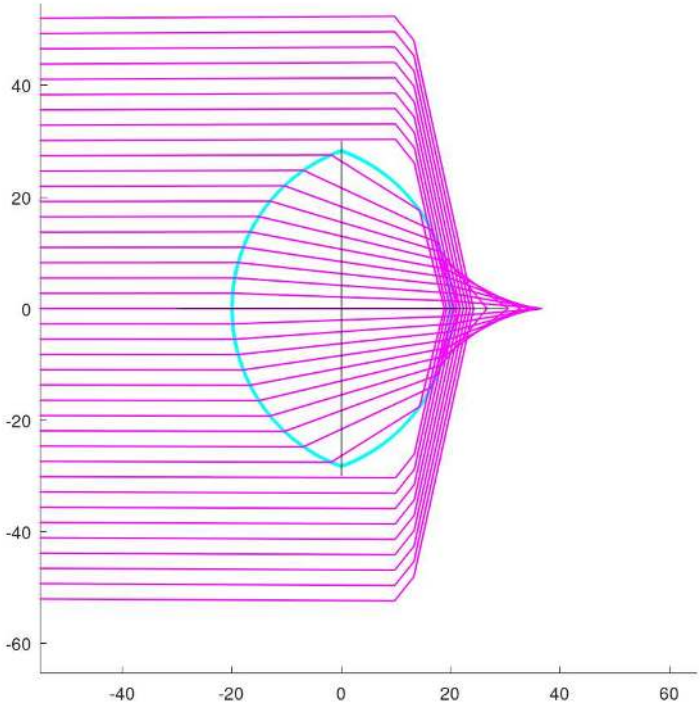




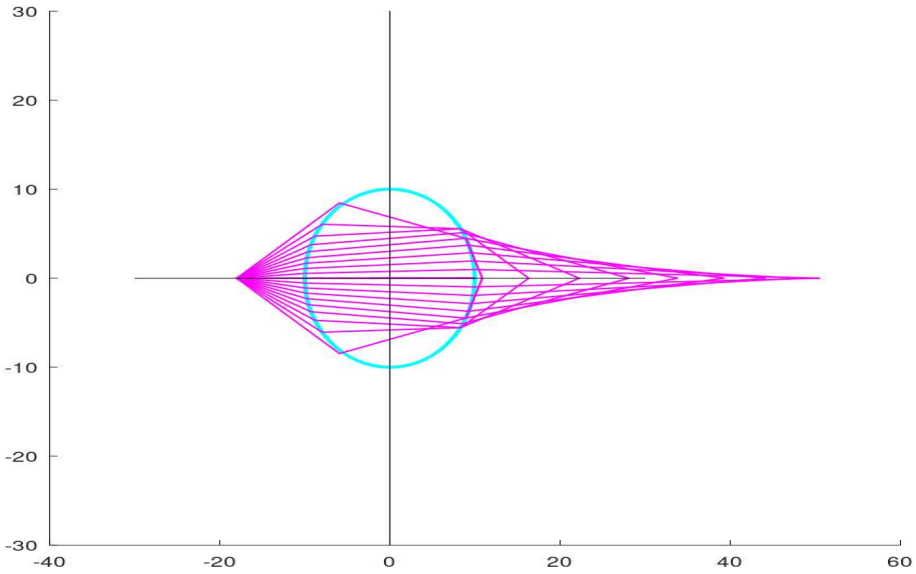
2) Mit einer dickeren Linse (Strahlen mit zu großem Winkel unphysikalisch und Artefakt von zu einfachen Simulationsstrategien)
   A) a > 2f (Bsp.: mit a = 18)
   B) 2f > a > f (Bsp.: mit a = 12)
   C) a sehr sehr groß (Bsp.: mit a = 10 000)

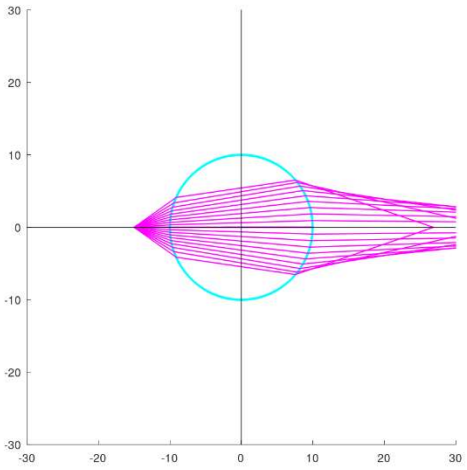


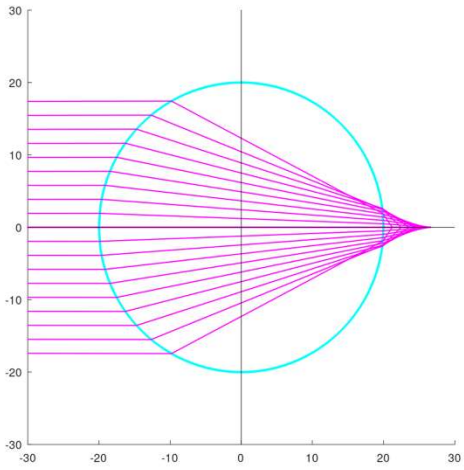




3) Mit einer sphärischen Linse (Kreis), wieder Artefakte wegen zu einfacher Simulationsstrategien.
   a) a > 2f (Bsp.: mit a = 18)
   b) 2f > a > f (Bsp.: mit a = 15)
   c) a sehr sehr groß (Bsp.: mit a = 10 000)







Als die Linse dicker wurde, haben unsere hergeleiteten Formeln schlechter funktioniert. Dies liegt daran, dass in unserer Herleitung der Linsengleichung die Linse keine Dicke hatte, was speziell für Kugellinsen nicht der physikalischen Wahrheit entspricht.

4.7. Abbildungsfehler

Im Alltag sieht man Phänomene, die nur anhand der Optik erklärt werden können. Theoretisch gesehen verlaufen alle parallel einfallenden Lichtstrahlen durch eine Linse durch den sogenannten Brennpunkt, aber in der Wirklichkeit kommt es zu Abweichungen oder die Strahlen treffen sich schräg auf der Linse, sodass das Bild unscharf wird. Es gibt viele Arten von Abbildungsfehlern. Davon haben wir die chromatische und sphärische Aberration besprochen.

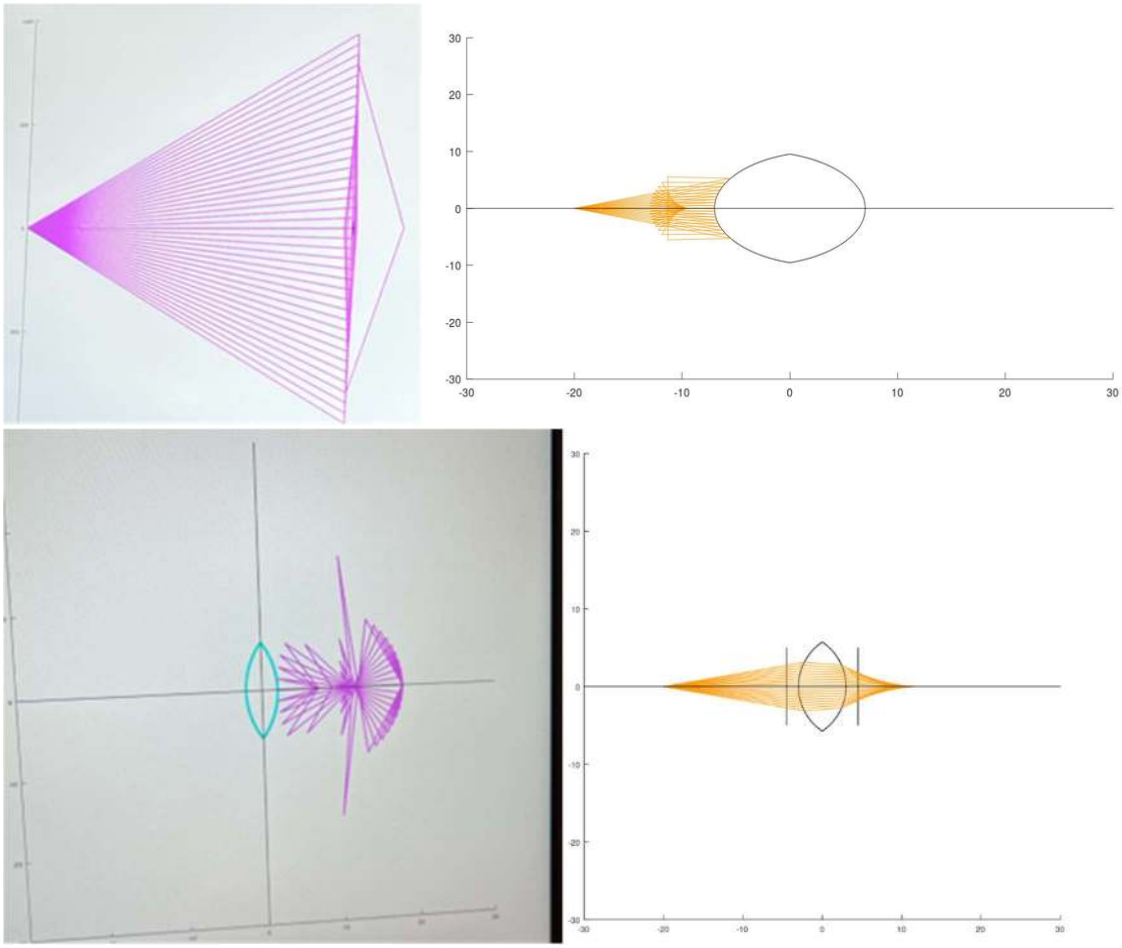
4.7.1. Chromatische Aberration

Das weiße Licht spaltet sich auf, weil die Farben unterschiedliche Wellenlänge haben und somit unterschiedlich stark gebrochen werden (Brechungsindex ist wellenlängenabhängig, daher ist auch die Brennweite wellenlängenabhängig).

4.7.2. Sphärische Aberration

Selbst wenn die Strahlen vom Unendlichen weg parallel auf die Linse treffen, werden Sie wie wir in den Simulationen gesehen haben, nicht immer alle durch den Brennpunkt verlaufen. Sondern abhängig von der Höhe des einfallenden Strahles unterschiedlich stark gebrochen, sodass es nicht möglich ist gleichzeitig alle Punkte eines Objektes gleich scharf abzubilden.

4.8. Fehler im Programm



Approximation

Close enough

Hannah Grabner

Amelie Lerch

Olivia Maria Ringhofer

Eileen Schmieger

Jakob Tegshee

Stefan Reiterer

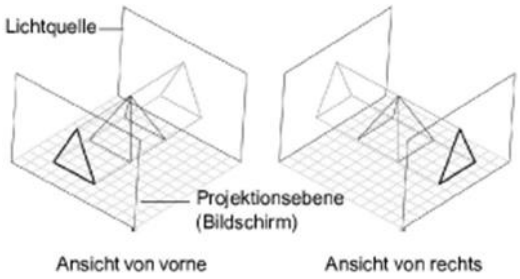
Inhalt

1. Approximation

Definition:

Die Approximation ist eine (An-)Näherung an den realen Wert einer beliebigen (schwer begreifbaren) Zahl in Form einer anderen Menge von Objekten, die begreifbar sind. Die Approximation verändert sich je nach Kontext des Problems.

Der Sinn besteht darin, die Realität möglichst nah und vereinfacht darzustellen und dabei den Fehler der Annäherung möglichst klein zu halten. Beispielsweise, indem man die kürzeste Distanz vom Punkt P (das Objekt mit Vektor (x | y | z)) zur Ebene E findet, wenn man ein Objekt auf eine Ebene projizieren will.



Diese kürzeste Distanz kann man finden, indem man Q als alle Punkte der Ebene definiert und die Distanz zwischen P und Q minimiert. Der Vektor PQ muss orthogonal zur Ebene stehen, weil der kürzeste Weg ein Lot ist. Das kann man unter anderem ausrechnen, indem man das Skalarprodukt des Vektors PQ mit sich selbst berechnet.

Vektor für Q

Die Formel für die Ebene hilft uns, damit wir den Vektor für Q aufstellen können.

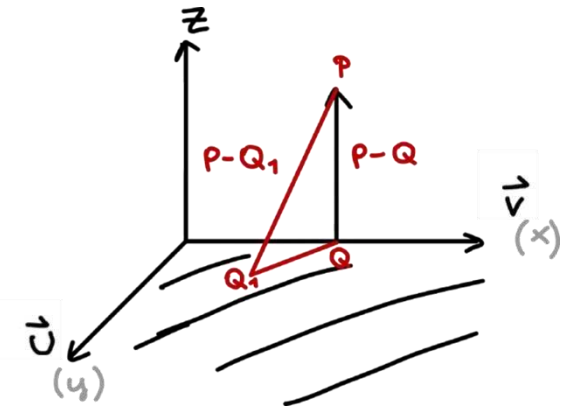
Dazu arbeiten wir mit Vektoren v (1|0|0) und u (|0|1|0) (entsprechen zwei der Basisvektoren im dreidimensionalen Raum):

$$E = \{tv + su | s, t \in R\} \quad \Rightarrow \quad \begin{pmatrix} s \\ t \\ 0 \end{pmatrix}$$

Um den Fehler minimal zu halten, sollen alle Terme des aufgestellten Vektors $(s|t|0)$ null sein, allerdings bleibt die Variable der z-Achse wegen der null im Vektor immer konstant und somit ist der Fehler der Approximation immer mindestens der Wert der z-Variable.

Beweis der Minimalität mit Pythagoras

Geht man davon aus, dass der Vektor PQ normal auf die Ebene E steht, kann man ein Dreieck aufstellen, das die Punkte P, Q und Q_1 verbindet. Q befindet sich in diesem Fall lotrecht unter P und Q_1 beschreibt alle Punkte der Ebene E. Somit bilden die Strecken P-Q und Q_1-Q einen rechten Winkel. Fügt man das in die Formel des pythagoräischen Satzes ein, findet man heraus, dass $Q_1 = Q$. Das beweist, dass die kürzeste Strecke zwischen zwei Punkten lotrecht auf die Ebene verlaufen muss.



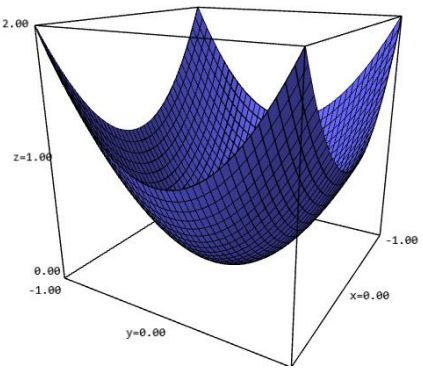
Spezialfall zur Berechnung des Tiefpunkts

Eine weitere Methode ist es, mithilfe der zweiten Ableitung einen Tiefpunkt der Funktion $d(s)$ festzustellen. Wenn $(P - Q)^2$, die Formel zum Berechnen des Abstands von P zu Q, binomisch aufgelöst wird, bekommt man $(P)^2 - 2(P - Q) + (Q)^2$. In diesem Spezialfall wird angenommen, dass der Wert Variable für die y-Achse für P null ist. Hierbei muss dann t null sein, weil t^2 durch das Quadrieren immer größer als null ist und die Variablen aber möglichst klein sein sollen. Das heißt, die Funktion d kann am kleinsten sein, wenn $t=0$. Dadurch kann diese Funktion $d(s)$ aufgestellt werden, die einmal abgeleitet wird,

um den Wert der Variable s beim Minimum der gebildeten quadratischen Funktion herauszufinden.

2. Partielle Ableitung nach einer Variable

Bei der partiellen Ableitung nach einer Variable wird bei einer Funktion mit zwei Termen, zunächst eine der Variablen null gesetzt und nach der anderen gelöst. Dieses Vorhaben wird dann reversiert, um die andere Variable ausrechnen zu können. Das bedeutet für eine Funktion $f(x, y)$, dass sie aufgebrochen wird auf eine $g(x)$ in der das ursprüngliche y festgesetzt wird und nach x gelöst wird und eine $h(y)$ in der das ursprüngliche x festgesetzt wird und nach y gelöst wird. Meistens gibt es mehrere Lösungen für beide und die Schnittstelle der beiden muss ermittelt werden, wobei die Punkte dann die richtigen x und y sind.



Die Ableitung einer Funktion hilft, weil sie eine andere Möglichkeit bietet, zu approximieren. Hierbei werden Gradienten verwendet, weil durch das Nullsetzen die Extremstelle und somit das Optimum der Funktion gefunden werden kann. In diesem Fall muss ∇(Nabla)$f = 0$ sein und der Gradient orthogonal auf f.

Partielle Ableitung nach einer Variablen:

$$\frac{\partial}{\partial x_i} f(\mathbf{a}) = \lim_{h \to 0} \frac{f(a_1, \ldots, a_{i-1}, a_i + h, a_{i+1} \ \ldots, a_n) \ - f(a_1, \ldots, a_i, \ldots, a_n)}{h}$$
$$= \lim_{h \to 0} \frac{f(\mathbf{a} + h\mathbf{e_i}) - f(\mathbf{a})}{h}.$$

Linear approximieren

Weil die Funktion kompliziert ist, approximieren wir sie zunächst als eine lineare Funktion *f* mit Variablen a, b linear: a*x+b. Dazu erstellen wir einen Code, um n nicht manuell berechnen zu müssen.

Wir erkennen, dass der Nenner von *f* die Standardabweichung ist:

```
g(a) = f(a,y_q - a*x_q)
```

```
[38]: pretty_print(solve(derivative(g,a)==0,a)[0])
```

$$a = \frac{(x_1 - x_q)y_1 + (x_2 - x_q)y_2 + (x_3 - x_q)y_3 + (x_4 - x_q)y_4 + (x_5 - x_q)y_5 - (x_1 + x_2 + x_3 + x_4 + x_5 - 5\,x_q)y_q}{x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2\,(x_1 + x_2 + x_3 + x_4 + x_5)x_q + 5\,x_q^2}$$

```
[43]: pretty_print(sum((xs[l]-x_q)**2 for l in range(n)).expand().simplify())
```

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2\,x_1x_q - 2\,x_2x_q - 2\,x_3x_q - 2\,x_4x_q - 2\,x_5x_q + 5\,x_q^2$$

Von da vereinfachen wir die Funktion noch mehr, weil wir den Nenner umschreiben können:

```
pretty_print(sum((xs[l]-x_q)*ys[l] for l in range (n))/(Var_x**2/n))
```

$$\frac{5\,((x_1 - x_q)y_1 + (x_2 - x_q)y_2 + (x_3 - x_q)y_3 + (x_4 - x_q)y_4 + (x_5 - x_q)y_5)}{Var_x^2}$$

Bei einer linearen Approximation wird an der Fehlerminimierung gearbeitet, indem nach a, b abgeleitet wird und das im Code gelöst und subsituiert wird. Die Fehlerquelle ist in diesem Fall nicht null, weil die originale Funktion nur annähernd linear ist.

Quadratisch approximieren

Bei einer Approximation mit einer quadratischen Funktion müssen x, y vorher substituiert werden. Danach muss man a, b, c (die Koeffizienten der Funktion) ableiten und herausheben, bevor das Ganze geplottet werden kann. Dadurch, dass die originale Funktion in diesem Beispiel bekannt war, konnte eine Approximation mit einer Fehlerquelle von null entstehen, indem man die gleiche Funktion darüberlegt.

Fehlerquelle

Es gibt viele Gründe, warum eine Fehlerquelle nicht gleich null ist. Mess- und Rundungsfehler beispielsweise schaffen automatisch ein Rauschen in das Endprodukt. Es steht also fest, dass die Approximation besser ist, wenn die Annäherungsfunktion das gleiche Polynom hat wie die Originalfunktion. Die Fehlerquelle wird nämlich kleiner, weil die Werte angepasst werden. Der Code muss deshalb für allgemeine Polynomfunktionen definiert werden, damit er breiter funktioniert. Dabei muss aufgepasst werden, dass kein sogenanntes „Overfitting" passiert, wenn man die genaue Lösung findet, da es gleich viele Koeffizienten wie Variablen gibt, weil dadurch der geplottete Graph zackig wird und somit suboptimal. In dem erstellten Code kann der Graph verändert werden, indem man die Variablen n, m (der Grad der Funktion) und Sigma (die Standardabweichung) verändert.

Um eine Formel zu finden, mit der man den Fehler der Approximation berechnen kann, muss zunächst angenommen werden, dass die Funktion *f* ungefähr der Funktion der Approximation entspricht. Man kann dann annehmen, dass im Worst-Case-Szenario bei einer Funktion bei zwei verschiedenen x Werten verschiedene Fehler hat.

$f_1(x)$ = Approximation

1. $f(x) = f_1(x) + E \Rightarrow f_1(x) = f(x) - E$
2. $f(x+h) = f_1(x+h) - E \Rightarrow f_1(x+h) = f(x+h) + E$

Wenn man von diesen zwei Gleichungen den Differenzenquotienten nimmt und herunterkürzt, bleibt immer $\frac{2*\varepsilon}{h}$ als Fehler über. Dabei sind h und ε indirekt proportional, was bedeutet, dass je näher h sich an 0 annähert, desto größer ist der Fehler und er bleibt immer mindestens ε und somit größer als 0.

3. Funktionsräume

Vektorraum

Ein Vektorraum muss abgeschlossen sein bezüglich der Addition und Multiplikation (8 Axiome). Es handelt sich dabei um eine Ansammlung von Vektoren, nach einer beliebigen Vorschrift, die immer gilt. Beispielsweise ist im $\mathrm{R3}$ der größtmögliche Vektorraum der $\mathrm{R3}$ selbst und eine Ebene wäre nur ein Untervektorraum. Außerdem wird jeder Vektorraum aus seinen Basisvektoren aufgespannt. Diese Basisvektoren sind linear voneinander unabhängig und es kann mit ihnen eben jeder beliebige Vektor erschafft werden, aber sie selbst können nicht durch andere Vektoren dargestellt werden.

Innerer Produktraum

Bei einem inneren Produktraum handelt es sich um die verschiedenen Skalarprodukte in einem Vektorraum. Das innere Produkt verallgemeinert das Skalarprodukt aus dem euklidischen Raum. Dabei gelten feste Axiome, allerdings kann das innere Produkt verschieden definiert sein, abhängig von der Struktur des Raumes (zum Beispiel Funktionen, Matrizen, komplexe Zahlen oder andere Räume). Das Skalarprodukt von zwei Funktionen zum Beispiel kann durch ein Integral definiert werden. Bei Matrizen muss man eine elementweise Multiplikation durchführen und dann die Summe der Produkte bilden.

Polynomraum

Der Polynomraum ist der Vektorraum der Polynome mit Koeffizienten aus dem Körper K (rationale, reelle oder komplexe Zahlen). Er ist abgeschlossen bezüglich der Addition und der Multiplikation mit, hat einen Nullvektor und die Funktion kann mithilfe der Koeffizienten als Vektor dargestellt werden, womit der Bezug zum Vektorraum bewiesen ist. Außerdem hat $\mathrm{P^n}$ die Dimension $\mathrm{n+1}$, was dem $\mathrm{R^{n+1}}$ entspricht. Ein Beispiel dafür wäre eine Polynomfunktion 2. Grades, die drei Koeffizienten hat und somit im $\mathrm{R3}$ besteht.

4. Gruppenarbeiten

Gruppe A: Least-Square/Kleinste Fehlerquadrate

Bei dieser Aufgabe beschäftigte sich die Gruppe A mit dem Gradienten und dem Gradientenverfahren. Mithilfe der Least-Square-Methode, also der Methode der kleinsten Fehlerquadrate, kann man die bestmögliche Annäherung an eine Menge von Datenpunkten berechnen, indem eine Funktion (z. B. eine Gerade) so angepasst wird, dass die Summe der quadrierten Abweichungen zwischen den tatsächlichen Datenpunkten und den vorhergesagten Werten minimiert wird. Diese Methode wird in Linearer Regression (z. B. Trendanalyse), Datenanpassung (z. B. in der Physik und Ingenieurwissenschaft) und im Maschinellen Lernen (Optimierung von Kostenfunktionen) wird angewendet.

Die Least-Square-Methode besteht aus folgenden Arbeitsschritten:

1. Datenpunkte erfassen: Man hat eine Menge von Messwerten (x_i, y_i).
2. Modell wählen: Häufig wird eine lineare Funktion $y=mx+b$ verwendet, aber auch nichtlineare Modelle sind möglich. Wir haben uns mit Polynomfunktionen und Exponentialfunktionen beschäftigt.

   Bei einer linearen Funktion $a(x)$ sieht die Fehlerfunktion $F(a_0,\ldots,a_n)$ zur Berechnung des Minimums (also die bestmögliche Annäherung an die tatsächlichen Werte) folgendermaßen aus und kann auch mit Sigma dargestellt werden.

$$a(x) = a_0 f_0(x) + a_1 f_1(x) + \cdots + a_n f_n(x)$$

$$F(a_0, \ldots, a_n) = \frac{1}{2}\left\{(y_1 - a(x_1))^2 + (y_2 - a(x_2))^2 + \cdots + (y_n - a(x_n))^2\right\} \quad \to \min$$

$$F(a_0, \ldots, a_n) = \frac{1}{2}\sum_{i=1}^{n}(y_i - a(x_i))^2$$

   Bei einer nicht-linearen Funktion wie der Exponentialfunktion wird die entsprechende Formel in die Fehlerfunktion eingefügt.

$$a(x) = a_0 e^{-a_1 x} \qquad = a_0 \exp(x a_1)$$

$$\rightarrow \min F(a_0, a_1)?$$

$$F(a_0, a_1) = \frac{1}{2} \sum_{i=1}^{n} \left(y_i - a_0 e^{-a_1 x_i}\right)^2$$

3. Fehler bestimmen: Die Differenz zwischen den vorhergesagten Werten $f(x_i)$ und den tatsächlichen Werten y_i wird berechnet.
4. Fehlersumme quadrieren: $S = \sum (y_i - f(x_i))^2$ um positive und negative Abweichungen gleich zu behandeln.
5. Minimierung: Ableitungen von S nach den Modellparametern (z. B. m,b) werden null gesetzt, um die optimale Lösung zu berechnen.

Der Gradient und das Gradientenverfahren:

Der Begriff *Gradient* hat verschiedene Bedeutungen in Mathematik, Physik und Ingenieurwissenschaften, aber das Grundkonzept bleibt gleich: Der Gradient gibt die *Richtung und die Stärke der größten Änderung einer Funktion* an.

Mathematisch gesehen ist der Gradient für eine Funktion $f(x_1,x_2,...x_n)$ mit mehreren Variablen ein Vektor, der aus den partiellen Ableitungen von f nach jeder Variablen besteht:

$$\operatorname{grad}(f) = \nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$$

Der negative Gradient von f entspricht der Richtung des steilsten Abstiegs einer Funktion, hat also den minimalsten Abstand zu den ursprünglichen Werten.

$$f(x+td) < f(x) \quad /-f$$
$$f(x+td) - f(x) < 0 \quad /:t$$
$$\frac{f(x+td)-f(x)}{t} < 0$$
$$\lim_{t\to 0} \frac{f(x+td)-f(x)}{t} = \left\langle \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}, d \right\rangle = \langle \nabla f, d \rangle \leq 0$$
$$d \overset{?}{\rightarrow} d = -\nabla f$$
$$\langle \nabla f, -\nabla f \rangle = -\langle \nabla f, \nabla f \rangle = -\underbrace{\|\nabla f\|^2}_{\geq 0} \leq 0$$
$$\Rightarrow d = -\nabla f$$

Um das Minimierungsverfahren anzuwenden haben wir einen Algorithmus erstellt, der ein Abbruchkriterium von $\varepsilon > 1/100$ hat, damit der Wert bestimmt werden kann, wenn das t nah genug an der 0 (Minimum) ist. Wenn für das gewählte $t = 1$ dieses Kriterium nicht stimmt, muss das t minimiert werden, bis die Aussage zutrifft.

Algorithmus

t → immer minimieren
d → Gradienten berechnen

Abbruchkriterium: $\|\nabla f\|^2 < \varepsilon^2$ $\quad \varepsilon > 0 = 0{,}001$

$d_k = -\nabla f(x_k)$

$t_k \quad t = 1$

$f(x+td) < f(x)$? nein: $t = \frac{t}{2} = \frac{1}{2}$ ja: ✓

$f(x+td) < f(x)$?

Anschließend haben wir uns angesehen, wie man einen Gradienten händisch berechnet:

geg.: $v_0 = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$

$f(x,y) = \|(x - x_0, y - y_0)\|^2 = (x-x_0)^2 + (y-y_0)^2$

$\min_{x,y} f(x,y) = 0 \quad$ und $x = x_0,\ y = y_0$

$\nabla f(x,y) = \begin{bmatrix} \frac{\partial f}{\partial x}(x,y) \\ \frac{\partial f}{\partial y}(x,y) \end{bmatrix} = \begin{bmatrix} 2(x-x_0) \\ 2(y-y_0) \end{bmatrix}$

$\nabla f(x,y) = 0$

Im folgenden Beispiel wird mit dem *Gradientenverfahren* gerechnet:

Gegeben sind v_0 und der Startvektor x_0 sowie die $f(x,y) = (x - x_0)^2 + (y - y_0)^2$. Es wird ein Abbruchskriterium festgelegt, sodass die Berechnung gestoppt wird, sobald der Gradient der Funktion in diesem Fall $f(x_k) < 1/100$. Dann setzt man $k = 0$ und berechnet den negativen Gradienten von f, da das Minimum gesucht ist. Für die Schrittweite t wurde 1 gewählt und mithilfe dieser die Bewegung entlang der Richtung für den Gradientenabstieg ausgedrückt. Die neue Funktion $f(x + t * d) = f(-1, -4)$ wird dafür angewendet, um das minimale t zu finden. Da 1 für t nicht zutrifft, wird die Zahl auf ½ halbiert. Dieselben Schritte wie vorher werden angewandt und es stellt sich heraus, dass das dieses t korrekt ist.

Beispiel: Gradientenverfahren

geg.: $v_0 = \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$ Start $\vec{x}_0 = \begin{pmatrix} 5 \\ 10 \end{pmatrix}$

$$f(x,y) = (x - x_0)^2 + (y - y_0)^2$$

→ Kriterium für STOP: $\| \nabla f(x_k) \| < \frac{1}{100}$

$\vec{x}_0 = \begin{pmatrix} 5 \\ 10 \end{pmatrix}$ $k = 0$ $f(x_0) = 3^2 + 7^2$ → $f(x,y) = (5-2)^2 + (10-3)^2$

$$\nabla f = \left\| \begin{bmatrix} 2(x-x_0) \\ 2(y-y_0) \end{bmatrix} \right\|^2 = \left\| \begin{bmatrix} 2x - 2x_0 \\ 2y - 2y_0 \end{bmatrix} \right\|^2 = \left\| \begin{bmatrix} 10-4 \\ 20-6 \end{bmatrix} \right\|^2 = \left\| \begin{bmatrix} 6 \\ 14 \end{bmatrix} \right\|^2 = \sqrt{6^2+14^2} > \frac{1}{100}$$

Abstand

$2 \cdot \sqrt{6^2 + 14^2} > 1$
→ nicht STOP

$k = 0$

$d = -\nabla f(x_k) = -\nabla f(x_0) = -\begin{bmatrix} 6 \\ 14 \end{bmatrix}$

$t_k = 1$ $f(x + td)$

t... Schrittweite
d... $-\nabla f$

$f\left((5,10) - t \cdot (6,14)\right) = f((-1,-4)) = (-1-2)^2 + (-4-3)^2 = 9 + 49 = f(x_0)$
→ neuen Vektor berechnen

58

$t = \frac{1}{2}$ $f(x_k + td) < f(x_k)$

$f((5,10) - \frac{1}{2} \cdot \binom{6}{14}) = f((5,10) - (3,7)) = f(2,3) = 2^2 + 3^2 = 25 < 58$ ✓

$x_1 = x_0 + \frac{1}{2} d = (5,10) - \frac{1}{2}(3,7) = (2,3)$

$x_1 = (2,3)$

$f(x_1) = 25$ x_1 / v_0

$-\nabla f(x_1) = \begin{pmatrix} 2(2-2) \\ 2(3-3) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ $\| \nabla f \| < \frac{1}{100}$ stop $x_1 = ja\ (2,3)$

Beweis von min

Nach der Berechnung des Gradienten mit der Hand implementierten wir diese auch auf SageMath wie dieser Code zeigt:

Gradient berechnen & Algorithmus erstellen

[17]:
```
d
```

[17]:
```
array([-0.0, -0.0], dtype=object)
```

[18]:
```
def calc_d(gradf,z):
    d = - np.array(gradf(*z))
    return d
```

[19]:
```
def check_kandidat(f,gradf,z,t):
    d = calc_d(gradf,z)
    kandidat = z + t*d
    return float(f(*kandidat)) < float(f(*z)), kandidat
```

[20]:
```
check, kandi = check_kandidat(f,gradf,z0,0.5)
```

[21]:
```
check
```

[21]:
```
True
```

[22]:
```
kandi
```

[22]:
```
array([2.0, 3.0], dtype=object)
```

[23]:
```
def abstiegsrichtungsverfahren(f,gradf,z0,eps=0.01,maxiter=1000):
    # maxiter -> Abbruch, wenn nach 1000 Durchgängen nichts gefunden
    z = np.array(z0)
    for k in range(maxiter):
        d = calc_d(gradf,z)
        if np.linalg.norm(d) < eps:
            break
            # wenn Länge des Vektors kleiner als eps (Gradient), dann Abbruch
        t = 1
        check, kandi = check_kandidat(f,gradf,z,t)
        for l in range(maxiter):
            if check is True:
                break
            t = t*0.5
            check, kandi = check_kandidat(f,gradf,z,t)
        z = kandi

    return z
```

[24]:
```
abstiegsrichtungsverfahren(f,gradf,z0)
```

[24]:
```
array([2.0, 3.0], dtype=object)
```

[ ]:

[16]:
```
d = -np.array(f.gradient()(*z1))
```

Least Square

Nun wandten wir in SageMath das Gradientenverfahren an einer Exponentialfunktion an.

1 Least Square

[25]:
```
import numpy as np
import matplotlib.pyplot as plt

n = 10
valsx = np.linspace(0, 5, n)
# intervall, darin aufgeteilt n Elemente im gleichen Abstand
a0_true = 2.0
a1_true = 1.0
# gesetzte Werte als Beispiel
valsy = a0_true * np.exp(-a1_true * valsx)
# valsy > Exponentialfunktion angeben mit valsx
```

[26]:
```
var('a0 a1 x y t')
fit_function = a0 * exp(-a1 * t)
error_func(a0,a1) = sum((valsy[k] - fit_function(t=valsx[k]))**2 for k in␣
 ↪range(n))
# alle Fehler möglichst klein
error_grad(x,y) = error_func.gradient()(a0=x,a1=y)
```

[27]:
```
error_func.gradient()(a0=0,a1=0)
```

[27]:
```
(-9.347960812578487, 0)
```

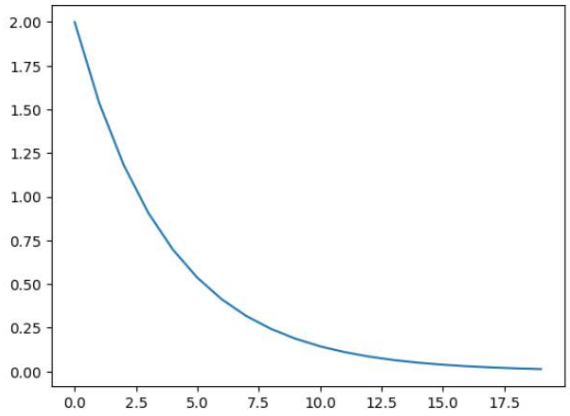
[28]:
```
error_func(0,0)
```

[28]:
```
5.962877766549741
```

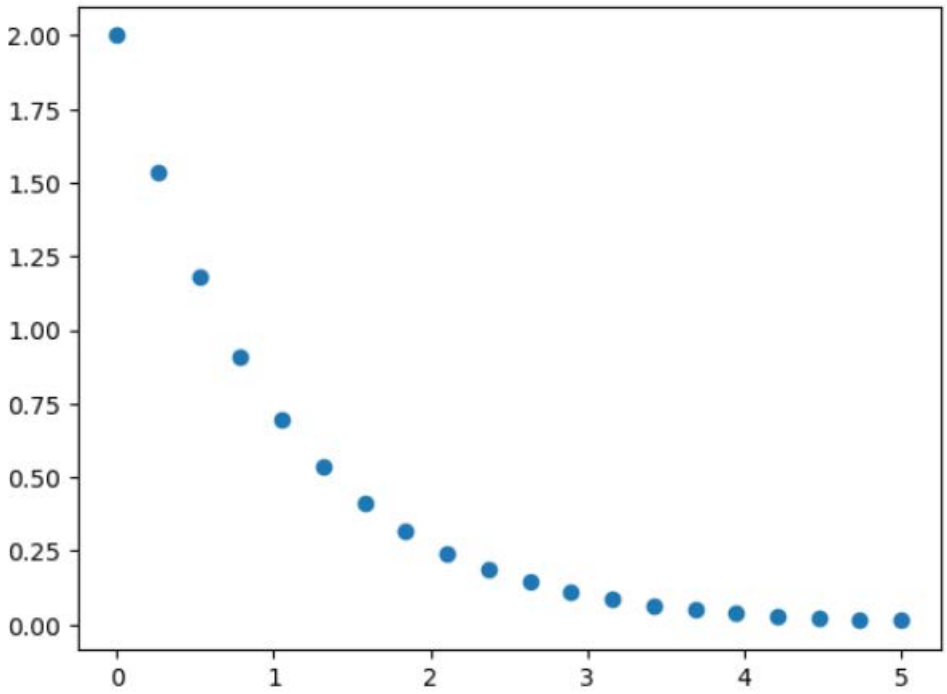
[29]:
```
#z0 = (a0_true,a1_true)
z0 = (0,0)

abstiegsrichtungsverfahren(error_func,error_grad,z0)
```

[29]:
```
array([2.001627450127883, 1.000204266804604], dtype=object)
```

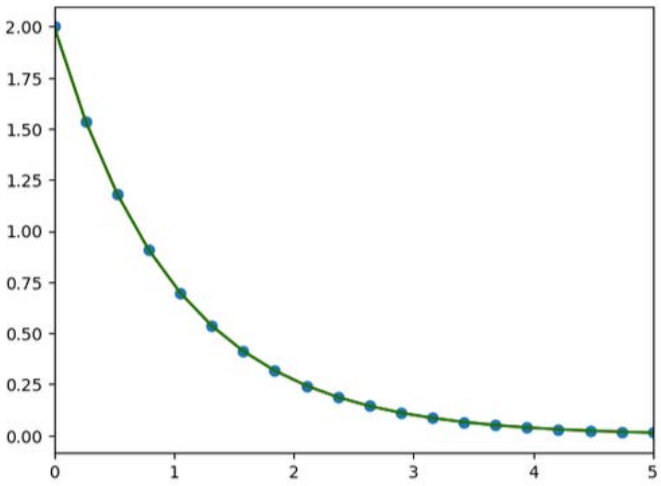
```
[2]: plt.plot(valsy)
     plt.show()
```



So sieht die normale Exponentialfunktion aus. Das folgende Bild zeigt nur die Punkte, welche von der ”Ursprungs-Exponentialfunktion” stammen und von einer zweiten Funktion approximiert werden sollen.



[46]:
```
valsy2 = [a0_true * np.exp(-a1_true * x) for x in valsx]
approx_y = [r1[0] * np.exp(-1*r1[1] * x) for x in valsx]

plt.xlim(0,5)
plt.scatter(valsx,valsy)
plt.plot(valsx,valsy2,color='red')
plt.plot(valsx,approx_y,color='green')

#richtige Funktion, richtige Punkte und approximation
plt.savefig(f'test_{μ}.png')
plt.show()
```



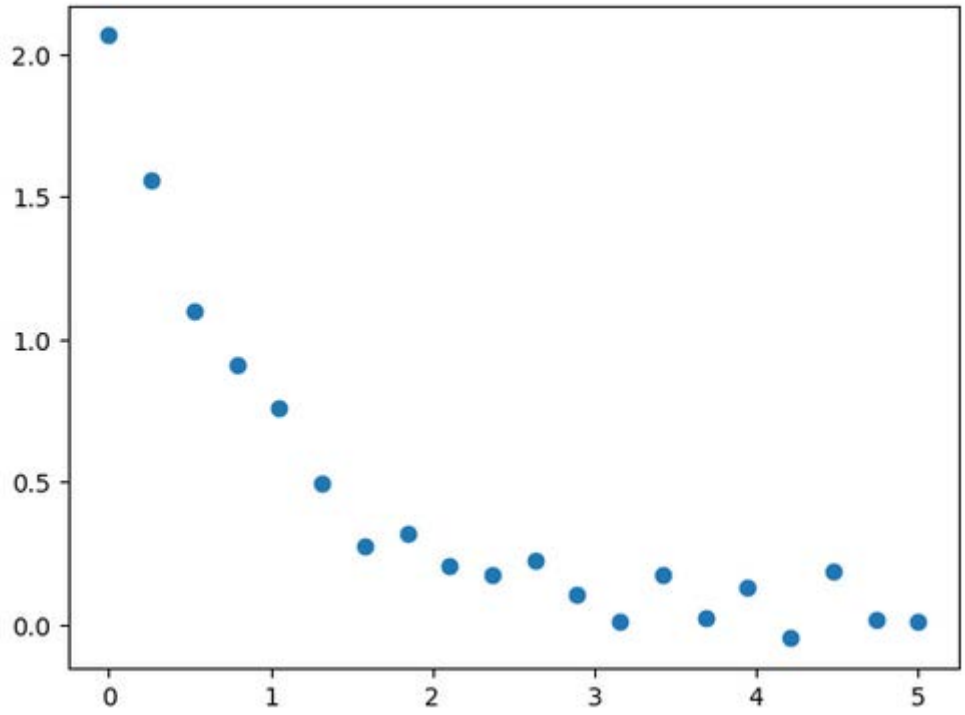
Zum Schluss kann man auch das Endprodukt des Codes graphisch darstellen und so auch überprüfen. Die rote Funktion ist die „Ursprungs-Exponentialfunktion“. Die grüne Funktion ist hingegen die Approximation aller Punkte. Da man nur die grüne Funktion sieht, kann angenommen werden, dass sie der „Ursprungs-Exponentialfunktion“ sehr ähnelt bzw. sie identisch sind.

Im nächsten Schritt wollten wir feststellen, ob dieser Code auch so gut funktioniert, wenn die Punkte, die von der ursprünglichen Exponentialfunktion abstammen, verrauscht werden. Dafür ändert sich beim Code nur folgendes:

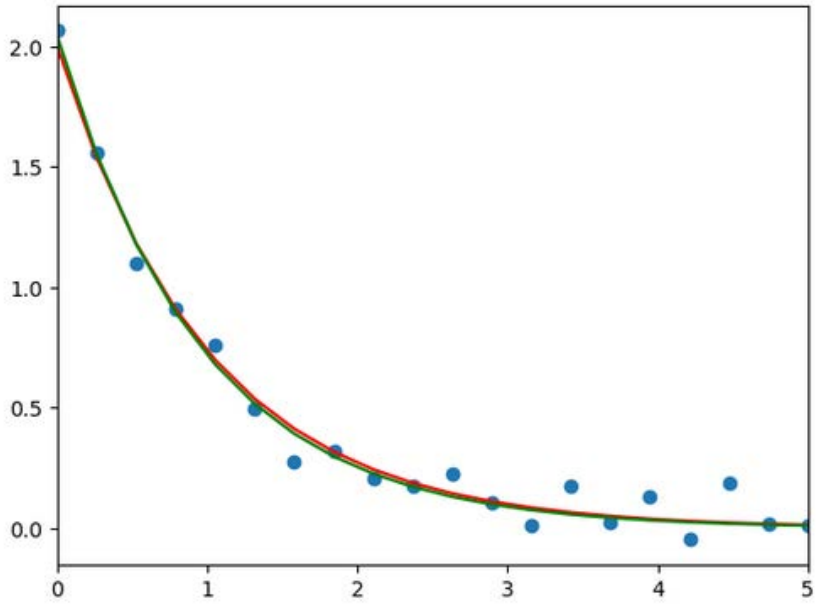
[48]:
```
import numpy as np
import matplotlib.pyplot as plt

n = 20 #Anzahl der Punkte
valsx = np.linspace(0, 5, n)
# n > wie viele punkte, linspace(intervall in dem ich n punkte will) > Punkte im gleichen Abstand
# intervall, darin aufgeteilt n Elemente im gleichen Abstand
a0_true = 2.0
a1_true = 1.0
# gesetzte Werte als Beispiel
μ = 0.1
valsy = a0_true * np.exp(-a1_true * valsx) + np.random.normal(0,μ,len(valsx))
# valsy -> exponentialfunktion die Wahrheit
# exp angeben mit valsx
```

Bei valsy muss eine random Normalverteilung hinzugefügt werden, damit sich die y-Werte der Punkte verändern.

```
[57]: <function show at 0x7f6e2decda80>
```



Die Punkte sehen dann so aus:



Plottet man die Funktionen wird deutlich, dass diese sehr ähnlich sind. Die Approximation ist somit sehr gut gelungen.

Polynomfunktion approximiert Exponentialfunktion

In weiterer Folge probierten wir im nächsten Programmcode zu den verrauschtten Punkten einer Exponentialfunktion eine geeignete Polynomfunktion zu finden.

Zuerst muss dafür wieder das Abstiegsrichtungsverfahren definiert werden:

[1]:
```
#Ziel: punkte von exponentialfunktion mit polynomfunktion darstellen
```

[2]:
```
var('x y')
import numpy as np
import matplotlib.pyplot as plt
```

[3]:
```
x0 = 2
y0 = 3
f(x,y) = (x-x0)**2 + (y-y0)**2
```

[4]:
```
g = [5,10]
g = np.array([5,10])
g*2
gradf = [f.diff(v) for v in (x, y)]
print(gradf)
```

```
[(x, y) |--> 2*x - 4, (x, y) |--> 2*y - 6]
```

[5]:
```
x = 2
y = 3
print(gradf)
eps = 1/100
gradf(x,y) = f.gradient()(x,y)
```

```
[(x, y) |--> 2*x - 4, (x, y) |--> 2*y - 6]
```

[6]:
```
z0 = np.array([5,10])
#array multiple values in one variable
f(*z0)
```

[6]:
```
58
```

[7]:
```
d=-np.array(f.gradient()(*z0))
```

[8]:
```
float(np.linalg.norm(d))<eps
t = 1
kandidat = z0 + t*d
float(f(*kandidat)) < float(f(*z0))
t = 0.5
```

```
kandidat = z0 + t*d
float(f(*kandidat)) < float(f(*z0))
```

[8]:
```
True
```

[9]:
```
z1 = kandidat
d=-np.array(f.gradient()(*z1))
```

[10]:
```
def calc_d(gradf,z):
    d = - np.array(gradf(*z))
    return d
```

[11]:
```
def check_kandidat(f,gradf,z,t):
    d = calc_d(gradf,z)
    kandidat = z + t*d
    return float(f(*kandidat)) < float(f(*z)), kandidat
```

[12]:
```
def abstiegsrichtungsverfahren(f,gradf,z0,eps=0.01,maxiter=1000):
    z = np.array(z0)
    for k in range(maxiter):
        d = calc_d(gradf,z)
        if np.linalg.norm(d) < eps:
            break
        t = 1
        check, kandi = check_kandidat(f,gradf,z,t)
        for l in range(maxiter):
            if check is True:
                break
            t = t*0.5
            check, kandi = check_kandidat(f,gradf,z,t)
        z = kandi

    return z
```

[13]:
```
abstiegsrichtungsverfahren(f,gradf,z0)
```

[13]:
```
array([2.0, 3.0], dtype=object)
```

Als nächstes wird die Exponentialfunktion aufgestellt. Dafür werden zuerst die Anzahl der Punkte n definiert und die „echten“ Parameter der Exponentialfunktion.

[14]:
```
n = 20 #Anzahl der Punkte
valsx = np.linspace(0, 5, n)

a0_true = 2.0
a1_true = 1.0

μ = 0.5
valsy = a0_true * np.exp(-a1_true * valsx) + np.random.normal(0,μ,len(valsx))
```

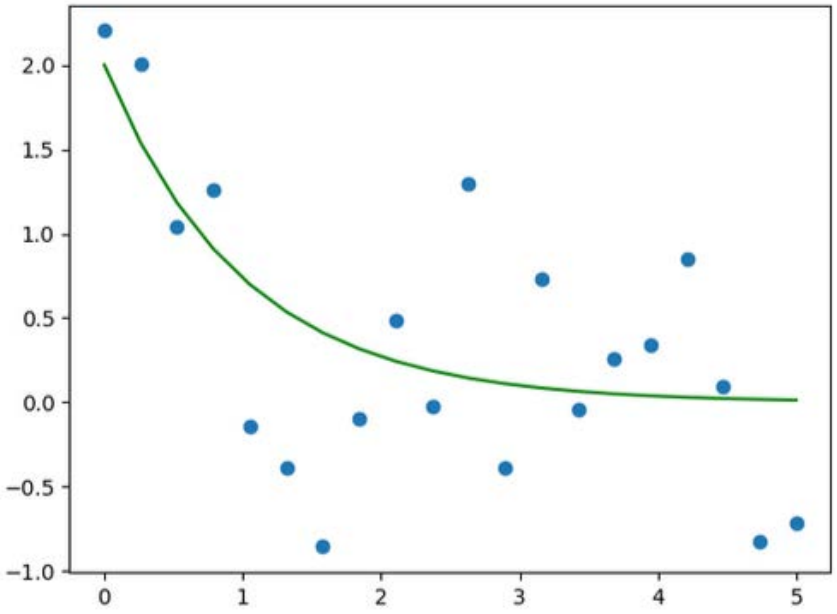
[ ]:
```

```

[15]:
```
valsy
```

[15]:
```
array([ 1.17340301,  1.64853143,  0.99538214,  0.41735942,  0.55339517,
       -0.16913473,  0.06613144, -0.19339003, -0.3684224 ,  0.19178111,
        0.48102495, -1.40401779,  0.39693848,  0.18572617, -0.07501945,
        0.51880538, -0.25041621,  0.93976693,  0.45415484, -0.07397914])
```

[20]:
```
from matplotlib import pyplot as plt

plt.scatter(valsx,valsy)

valsy2 = [a0_true * np.exp(-a1_true * x) for x in valsx]


plt.plot(valsx,valsy2,color='green')

plt.show
```

[20]: <function show at 0x7f760dfb2160>




```
xs = [None]*n
ys = [None]*n
#liste gpfuscht

for k in range(1,n+1):
    xs[k-1] = var('x_'+str(k))
    ys[k-1] = var('y_'+str(k))

subsi = [xs[l]==valsx[l] for l in range(n)] + [ys[l]==valsy[l] for l in range(n)]

m = 10 #Grad der approx
cs = [None]*(m+1) #Liste wird erstellt
for k in range(m+1):
    cs[k] = var('c_'+str(k))

def p(x):
    return sum(cs[k]*x**k for k in range(m+1))

f_m = sum((ys[l] - p(xs[l]))**2 for l in range(n))/2
```

[18]:
```
gleichungen = [derivative(f_m,cs[k])==0 for k in range(m+1)] #ableitung
gleichungen = [g.subs(*subsi) for g in gleichungen] #gleichungen für 0 setzen
result = solve(gleichungen,cs)[0]
result = [ex.rhs() for ex in result]
```

[19]:
```
gleichungen = [derivative(f_m,cs[k])==0 for k in range(m+1)]
gleichungen = [g.subs(*subsi) for g in gleichungen]
solve(gleichungen,cs)
```

[20]:
```
approx(x) = sum(result[k]*x**k for k in range(m+1))
```

[21]:
```
err = max([abs(valsy[k]-approx(valsx[k])) for k in range(n)])/max(valsy)
err
```

[21]:
```
0.7547258002651328
```

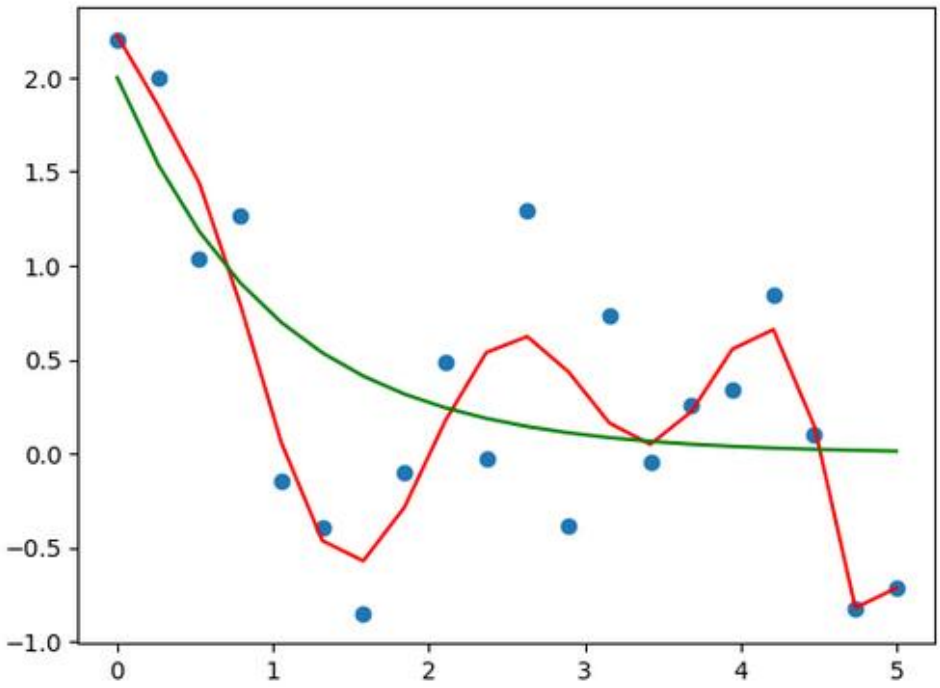
[22]:
```
sum(cs[k]*x**k for k in range(m+1))
```

[22]:
```
c_10*x^10 + c_9*x^9 + c_8*x^8 + c_7*x^7 + c_6*x^6 + c_5*x^5 + c_4*x^4 + c_3*x^3
+ c_2*x^2 + c_1*x + c_0
```

[23]:
```
var('x')
k = 1
sum(cs[k]*x**k for k in range(m+1))
```

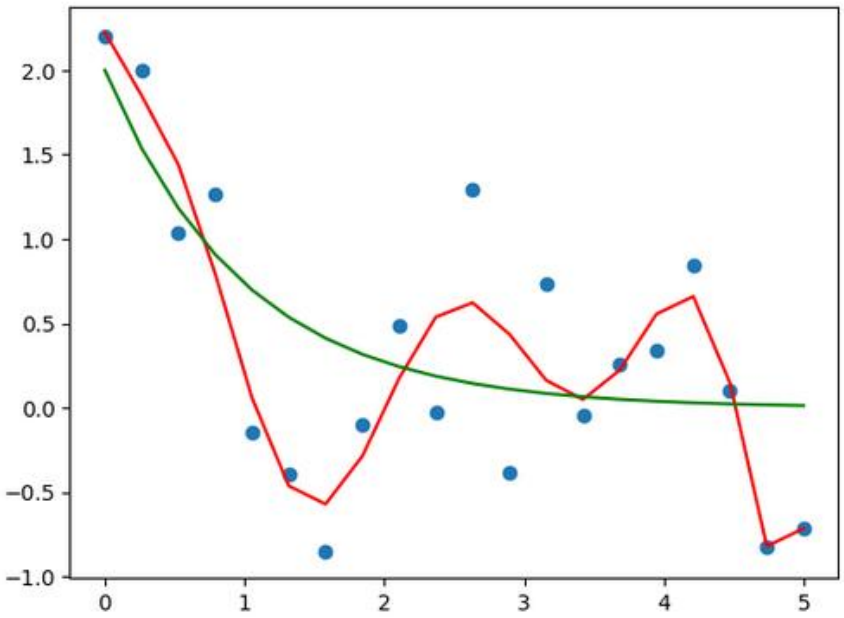
[23]:
```
c_10*x^10 + c_9*x^9 + c_8*x^8 + c_7*x^7 + c_6*x^6 + c_5*x^5 + c_4*x^4 + c_3*x^3
+ c_2*x^2 + c_1*x + c_0
```

[24]:
```
from matplotlib import pyplot as plt

plt.scatter(valsx,valsy)

valsy2 = [a0_true * np.exp(-a1_true * x) for x in valsx]
approx_y = [approx(x) for x in valsx]

plt.plot(valsx,approx_y,color='red')
plt.plot(valsx,valsy2,color='green')

plt.show
```



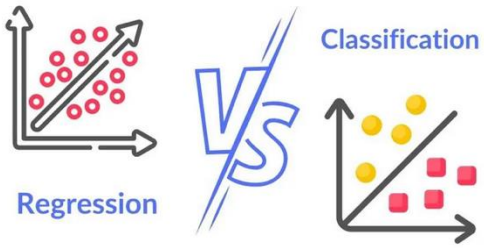
Man sieht im oberen Graphen, dass die Approx-Funktion mit dem Grad 10 sich der ursprünglichen Funktion nicht so gut annähert.

Ändert man nun den Grad der Funktion verbessert sich die Annäherung:



Machine learning

In weiterer Folge haben wir uns auch mit Machine Learning beschäftigt. Dabei haben wir versucht, dass eine Maschine die gegebenen Daten kategorisieren kann. Dies nennt man auch *data classification* bzw. Datenklassifikation. Datenklassifikation ist der Prozess, bei dem Daten basierend auf bestimmten Kriterien in Kategorien eingeteilt werden.



Wir klassifizieren in diesem Beispiel das Auto mpg Dataset, in dem Autos mit sieben Parametern und einem mpg-Wert gegeben sind. Daraus versuchen wir ein lineares Modell zu finden, dass aus den Parametern des Autos versucht zu erraten, ob der mpg-Wert größer bzw. kleiner als 23 ist, wobei kleiner als 23 als gut und größer als 23 als schlecht bewertet wird.

[1276]:
```
Y = (df[:,0] < 23).astype(int)
Y[Y==0]=-1
```

[1277]:
```
Y = (df[:,0] < 23).astype(int)
Y[Y==0]=-1
#Y = (df[:,0]-23)/(max(np.abs(df[:,0]-23)))
Y = Y
```

[1278]:
```
X = df[:,1:]
```

[1279]:
```
Y
```

[1279]:
```
array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1,  1,  1,
        1, -1, -1, -1, -1, -1, -1,  1,  1,  1,  1,  1, -1, -1, -1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1,  1, -1, -1, -1, -1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1,  1,  1, -1, -1,  1,
        1,  1, -1,  1,  1,  1,  1,  1, -1, -1, -1, -1,  1,  1,  1,  1,  1,
        1,  1,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1,  1, -1, -1,
       -1, -1,  1, -1,  1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1,  1,  1,
        1,  1,  1,  1, -1,  1, -1, -1, -1, -1,  1,  1,  1,  1, -1, -1, -1,
       -1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1,  1, -1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1, -1, -1, -1, -1,  1, -1, -1, -1,
        1,  1,  1,  1, -1, -1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
       -1,  1, -1,  1,  1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1,
       -1])
```

[1280]:
```
X
```

[1280]:
```
array([[  8. , 307. , 130. , ...,  12. ,  70. ,   1. ],
       [  8. , 350. , 165. , ...,  11.5,  70. ,   1. ],
       [  8. , 318. , 150. , ...,  11. ,  70. ,   1. ],
       ...,
       [  4. , 135. ,  84. , ...,  11.6,  82. ,   1. ],
       [  4. , 120. ,  79. , ...,  18.6,  82. ,   1. ],
       [  4. , 119. ,  82. , ...,  19.4,  82. ,   1. ]])
```

Gesucht: Gewichte w sodass $< w, x > \approx Y(x)$

[1281]:
```
M,N = X.shape
ws = [None]*N
for k in range(N):
    ws[k] = var(f'w_{k}')
```

Skaliere X da sonst zu groß

[1282]:
```
scale = np.array([max(np.abs(X[:,k])*2) for k in range(N)])
scale
```

[1282]:
```
array([1.600e+01, 9.100e+02, 4.600e+02, 1.028e+04, 4.960e+01, 1.640e+02,
       6.000e+00])
```

[1283]:
```
np.max(X/scale)
```

[1283]:
```
0.5
```

Gesucht: Gewichte w sodass $< w, x > \approx Y(x)$

[1281]:
```
M,N = X.shape
ws = [None]*N
for k in range(N):
    ws[k] = var(f'w_{k}')
```

Skaliere X da sonst zu groß

[1282]:
```
scale = np.array([max(np.abs(X[:,k])*2) for k in range(N)])
scale
```

[1282]:
```
array([1.600e+01, 9.100e+02, 4.600e+02, 1.028e+04, 4.960e+01, 1.640e+02,
       6.000e+00])
```

[1283]:
```
np.max(X/scale)
```

[1283]:
```
0.5
```

[1284]:
```
M = 200 # Veringere Daten
Xs = (X/scale).round(6)[:M,:]
#Xs = Xs[23:25]
#Y=Y[23:25]
#M=2
```

[1296]:
```
alpha = 0.5
logit = sum(ln(1.0+exp(-Y[k]*sum(ws[l]*Xs[k,l] for l in range(N)))) for k in range(M)) + alpha*sum(ws[l]**2 for l in range(N))
```

[1297]:
```
pretty_print(logit)
```

[1300]:
```
def func2(XX):
    subsi = [ws[k]==float(XX[k]) for k in range(N)]
    return float(logit.subs(*subsi))
```

[1300]:
```
def func2(XX):
    subsi = [ws[k]==float(XX[k]) for k in range(N)]
    return float(logit.subs(*subsi))
```

[1301]:
```
XX = Xs[0,:]
subsi = [ws[k]==float(XX[k]) for k in range(N)]
subsi
func(Xs[0,:])
```

[1301]:
```
9.339635793769002
```

[1302]:
```
logit.gradient().subs(subsi)
```

[1302]:
```
(-5.408377144451686, -6.938747634973443, -3.779283821811212, -4.357367776399187, 3.263659343168375, 2.155482
136242652, 7.76945551061239)
```

[1303]:
```
def gradf2(XX):
    subsi = [ws[k]==float(XX[k]) for k in range(N)]
    result = logit.gradient().subs(subsi)
    result = np.array([float(x) for x in result])
    return result
```

[1304]:
```
logit(w_0=1,w_1=1,w_2=1,w_3=1,w_4=1,w_5=5.0,w_6=1.0)
```

[1304]:
```
269.156151828794
```

[1305]:
```
logit.subs(*subsi)
```

[1305]:
```
120.323825711833
```

[1309]:
```
w0 = np.ones(N) #np.array((1,0))
sol = abstiegsverfahren(func2,gradf2,w0,scale=0.9,maxiter=20000,eps=1e-1)
```

[1310]:
```
success = 0
fail = 0
for k in range(M):
    err = np.dot(sol,Xs[k,:])/abs(np.dot(sol,Xs[k,:]))-Y[k]/abs(Y[k])
    if abs(err)<1e-1:
        success += 1
    else:
        fail += 1
    print(k,':',np.dot(sol,Xs[k,:])/abs(np.dot(sol,Xs[k,:]))-Y[k]/abs(Y[k]))
print('Success: ',success, 'Fails: ',fail)
```

```
Success:  177 Fails:  23
```

[1230]:
```
np.dot(sol,Xs[14,:])
```

[1230]:
```
-3.174556510011245
```

[ ]:
```
Y[14]
```

[ ]:
```
Y[Y==0]=-1; Y
```

[ ]:
```

```

[1054]:
```
for k in range(M):
    print(np.dot(sol,Xs[k,:]))
```

```
-0.005942499996881859
-0.03522122863697777
```

Bei 200 Datensätzen kann das trainierte Modell 177 als gut und 23 als schlecht richtig bewerten. Wenn ein besseres Ergebnis erzielt werden möchte, muss ein anderes Modell gewählt werden.

Gruppe B: Projektion

Aus einer WAV-Datei in die Welt der Mathematik

Die Aufgabe der Gruppe B bestand darin, eine WAV-Datei in seine einzelnen Frequenzen aufzuspalten und in ein Format, das einer MP3-Datei gleicht, zu konvertieren. Dies wird durch Herausfiltern von Frequenzen, die der Mensch nicht hören kann, vollbracht. Eine WAV-Datei kann auch als Funktionsgleichung dargestellt werden und in seine einzelnen Polynome aufgeteilt werden und jeder dieser Polynome hat einen Koeffizienten, der die Frequenz über die Zeitspanne hinweg beschreibt. Mit der Hilfe von der Funktionsgleichung könnte auch integriert werden, so wie es die Formel verlangt, aber ohne ihr muss die Trapezregel angewendet werden, um die ursprüngliche Funktionsgleichung herauszufinden, damit im Anschluss integriert werden kann. Um die Funktionsgleichung zu ermitteln, werden die einzelnen y-Werte, die die Tonspur besitzt, ermittelt und in eine eigens hergeleitete Gleichung eingesetzt, die sich die Eigenschaften

von Vektoren zunutze macht. Mit den Koeffizienten, die ausgerechnet wurden, kann man die Audiodatei von unerwünschten Daten bereinigen und fast ohne bedeutsamen Datenverlust wieder abspielen.

Projektion von WAV zu MP3

Trapezregel:

$$\int_x^{x_{c+1}} f(x)dx \approx h\frac{f(x_c)+f(x_c+1)}{2}$$

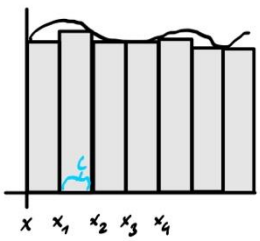
$$h = \frac{b-a}{n}$$

$x_0 = a + 0.h \ \ldots$

$x_1 = a \cdot 1 \cdot h$

$\rightarrow f(x) = ax$

$$x_n = a + n \cdot h \quad \leftarrow \text{eingesetzt } h = \frac{b-a}{n}$$
$$= a + \frac{b-a}{n} \cdot n =$$
$$= a + b - a = b$$



Trapezflächenformel:
$$\frac{(a+c)\cdot b}{2}$$

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \int_{x_{n-1}}^{x_n} f(x)dx$$

$$= h[\frac{f(x)+f(x_1)}{2} + \frac{f(x_1)+f(x_2)}{2} + \ldots + \frac{f(x_{n-2})+f(x_{n-1})}{2} + \frac{f(x_{n-1})+fx_n}{2}$$

hebt sich auf

$$= h \cdot [\frac{f(x)}{2} + f(x_1) + f(x_2) + \ldots + f(x_{n-1}) + \frac{f(x_n)}{2}]$$

Bsp

$$\frac{1}{2}\int_a^b f(x) - a(x) = \frac{h}{2}[\frac{f(x_0)-a(x_0)}{2} + \ldots + \frac{f(x_n)-a(x_n)}{2}]$$

$$F(a_0,\ldots,a_n) = ||f-a|^2 =< f - (a_0f_0 + a_1f_1 + \ldots + a_nf_n)(f_0a_0 + a_1f_1 + \ldots + a_nf_n) >$$

$$= ||f||^2 - 2a_0 < f \cdot f_0 > -2a_1(f \cdot f_1) \ldots - 2a_n < f \cdot f_0 > + \underbrace{a_0^2 < f_0f_0 >}_{1} + \underbrace{a_na_0^2 < f_0f_1 >}_{0} + \underbrace{2a_2a_0 < f_0f_2 >}_{0} + \ldots + \underbrace{a_n^2 < f_nf_n >}_{1}$$

↳ Skalarmultiplikation identisch = 1, orthogonal = 0

$$\frac{\partial F}{\partial Q_1}(a_1,\ldots,a_n) =< f \cdot f_1 > +a_1 \cdot 1 \equiv 0 \Rightarrow a_1 = (f \cdot f_1)$$

$$\frac{\partial F}{\partial Q_i}(a_1,\ldots,a_n) = -2 < f \cdot f_i > +2a_i = 0 \Rightarrow a_i < f_1f_i >$$

↳ F ... eine Funktion
Q ⊥ F → eine Funktion die normal auf F steht

Speziallfall: $f = c_0f_0 + c_1f_1 + \ldots + c_nf_n$

$$< f \cdot f_i >= c_0 < f_0 \cdot f_i > + \ldots + c_i < f_i \cdot f_i > + \ldots + c_n, f_i >= c_i$$

$$f \sim (c_0,\ldots,c_n) \qquad < f \cdot f_i >= \frac{2}{\pi}\int_0^{\pi} f(x)\cos(ix)dx$$

$$f(x)\sum_{n=0}^{\infty} c_if_i(x) \qquad \approx \frac{2h}{\pi}[\frac{f(x_0)\cos(ix)}{2} + f(x_1)\cos(ix_1) + \ldots + \frac{f(x_n)\cos(ix_n)}{2}]$$

```
def koeffizient(Y,i):
    Z = np.cos(i*X)*Y
    Z[0] /= 2
    Z[-1] /= 2
    return sum(Z)
```

Der gegebene Code führt eine Diskrete Kosinustransformation (DCT) auf eine WAV-Datei durch, filtert die Koeffizienten und rekonstruiert das Signal anschließend, um eine komprimierte Version der Audiodatei zu speichern. Hier folgt eine detaillierte Erklärung der einzelnen Abschnitte.

Importierte Bibliotheken

Der Code verwendet folgende Bibliotheken:

- `os.path`: Um Dateipfade zu verwalten.
- `scipy.io.wavfile`: Zum Einlesen und Schreiben von WAV-Dateien.
- `scipy.fftpack.dct` & `scipy.fftpack.idct`: Für die Diskrete Kosinustransformation (DCT) und deren Inverse.
- `numpy`: Für numerische Berechnungen.
- `matplotlib.pyplot`: Zum Plotten von Daten.

Globale Variablen

```
start = 0
end = 50000
```

Diese Variablen bestimmen den Bereich der Koeffizienten, die verwendet bzw. gefiltert werden.

Funktionen

Funktion Koeffizienten(Y)

```
def koeffizienten(Y):
    c = scipy.fftpack.dct(Y,type=2,norm='ortho')
    c /= (0.5*len(Y))**0.5
    return c
```

- Berechnet die DCT des Signals Y.
- Normiert die Koeffizienten durch `(0.5 * len(Y))**0.5`, was eine skalierte Version der Transformation ergibt.

Funktion signal(c, start, end)

```
def signal(c,start=start,end=end):
    c *= (0.5*len(c))**0.5
    c_inv = scipy.fftpack.idct(c,type=2,norm='ortho')
    return c_inv
```

- Führt die inverse DCT (IDCT) aus.
- Multipliziert die Koeffizienten zurück mit der Skalierungsfaktor, um das ursprüngliche Signal wiederherzustellen.

Filterfunktionen

filter1(x):

```
def filter1(x):
    return -1*np.tanh(x-30000)*0.5+0.5
```

- Eine Hyperbelfunktion `tanh`, die als Weichfilter dient.
- Werte um 30.000 werden gefiltert, indem hohe Frequenzen abgeschwächt werden.

filtern(c, start, end):

```
def filtern(c,start=start,end=end):
    d = c.copy()
    for x in range(len(c)):
        if np.abs(d[x]) > 1e-6:
            d[x] = d[x]*filter1(x)
        else:
            d[x] = 0
    d = d[start:end]
    return d
```

- Wendet den `filter1(x)` auf die DCT-Koeffizienten `c` an.
- Setzt sehr kleine Werte auf 0, um Kompression zu erreichen.

fill_up(c, total_frequencies, start, end):

```
def fill_up(c,total_frequencies,start=start,end=end):
    head = np.zeros(start)
    tail = np.zeros(total_frequencies-end)
```

```
    full_c = np.array(head.tolist()+c.tolist()+tail.tolist())
    return full_c
```

- Füllt das gefilterte Signal mit Nullen am Anfang und Ende auf, um die Originalgröße zu erhalten.

Hauptprogramm

```
if __name__ == '__main__':
```

- **Liest die WAV-Datei ein**:

```
samplerate, data = wavfile.read(wav_fname)
```

- **Skaliert die Audiodaten auf den Bereich [-1,1]**:

```
int16_info = np.iinfo(np.int16)
channel0 = data[:,0] / int16_info.max
channel1 = data[:,1] / int16_info.max
```

- **Durchläuft die beiden Audiokanäle und verarbeitet sie**:

```
for i in range(2):
    Y = data[:,i]/int16_info.max
    c = koeffizienten(Y)
    c_cut = filtern(c, start=start, end=end)
    Ycompressed = signal(fill_up(filtern(c, start=start, end=end),
len(Y), start=start, end=end))
```

    - Berechnet die DCT.
    - Filtert die Koeffizienten.
    - Rekonstruiert das komprimierte Signal mit der inversen DCT.
- **Speichert die komprimierten Koeffizienten**:

```
np.savetxt(f'uncompressed_dct.csv',              np.array(coeffs).T)
np.savetxt(f'compressed_dct.csv', np.array(coeffs_comp).T)
```

- **Speichert das komprimierte Audiosignal als WAV-Datei**:

```
wavfile.write('compressed_fast.wav', samplerate, data_c)
```

- **Zeichnet die Original- und komprimierten Daten auf**:

```
plt.plot(np.arange(len(coeffs_comp[0])), np.array(coeffs_comp[0]).T)
```

Zusammengefasst lässt sich sagen, dass der Code eine WAV-Datei nimmt, ihre DCT berechnet und sie filtert, das Signal rekonstruiert und es in einer neuen WAV-Datei speichert. Er visualisiert auch die Koeffizienten und das komprimierte Signal.

Approximation mit der Finiten-Elemente-Methode

Gegeben ist eine Saite, welche auf einem Intervall von 0 bis π eingespannt ist. Diese wird angezupft und in Schwingung versetzt. Diese Schwinung wollen wir modellieren:

Ist die Eigenfunktion der gegebenen Schwinung nicht bekannt, gibt es eine Möglichkeit, diese herauszufinden: Ziel ist es, mithilfe einer anderen Funktion $\varphi(x)$, die die Randbedingungen erfüllt, sich an die Eigenfunktion der Schwinung "heranzutasten". Diese andere Funktion muss in kleine Teilelemente (Finite-Elemente) zerlegbar sein, da so Berechnungen einfacher werden.

In diesem Fall habe ich eine Hütchenfunktion mit Hütchen der Höhe 1 gewählt. Berechnungen sind mit dieser Funktion realtiv einfach,

In [1]:
```
import scipy
import numpy as np
import scipy.integrate as scipy_integrate
import matplotlib.pyplot as plt
```

Bestimmung der Hütchenfunktion:

Das Intervall $[0, \pi]$ wird in n Teilintervalle aufgeteilt, mit einer Länge von h. Jeder dieser Punkte wird in der Liste X gespeichert. Je größer die Anzahl an Teilintervallen, desto genauer ist die Approximation an die Eigenfunktion.
Anmerkung: $(n+1)$ kommt daher, dass ein Abschnitt von n Intvervallen, $n+1$ Punkte enthält, die diese begrenzen.

In [2]:
```
def x_values(k,n):
    a = 0
    b = np.pi
    h = (b-a)/n
    return a + k*h
```

In [3]:
```
n = 10
X = []
for k in range(n+1):
    X.append(x_values(k,n))
```

Die Hütchenfunktion $\varphi(x)$ ist definiert durch:

$$\phi_k(x) = \begin{cases} \frac{x-x_{k-1}}{x_k-x_{k-1}}, & x \in [x_{k-1}, x_k) \\ \frac{x_{k+1}-x}{x_{k+1}-x_k}, & x \in [x_k, x_{k+1}) \\ 0, & \text{sonst} \end{cases}$$

wobei x_k die Einträge der Liste X sind. Zu beachten ist aber, dass x_1 erst der zweite Eintrag ist, da die beiden Randpunkte x_0 (= 0) und x_n (= π) kein "eigenes" Hütchen besitzen.

In [4]:
```
def phi(k,t):
    if X[k-1] <= t < X[k]:
        return (t-X[k-1])/(X[k]-X[k-1])
    elif X[k] <= t < X[k+1]:
        return (X[k+1]-t)/(X[k+1]-X[k])
    else:
        return 0
```

In [5]:
```
def phi_strich(k,t):
    if X[k-1] <= t < X[k]:
        return 1/(X[k]-X[k-1])
    elif X[k] <= t < X[k+1]:
        return ((-1))/(X[k+1]-X[k])
    else:
        return 0
```

Bestimmung der Massen- und Steifigkeitsmatrix und der Berechnung der Eigenwerte bzw. -vektoren:

Um die Eigenfunktion bestimmen zu können, müssen wir die Eigenwerte λ_n wissen. Diese ergeben sich aus der Lösung des verallgemeinerten Eigenwertproblems

$$C \cdot x = \lambda \cdot M \cdot x.$$

Die Massenmatrix M ergibt sich aus dem ursprünglichen Ziel, den Projektionsfehler so gering wie möglich zu halten. Um diesen zu minimieren, setzt man die erste Ableitung Null (Extremwertberechnung). Multiplizert man die zu transformierende Funktion mit der Massenmatrix, erhält man die Projektion mit der kleinstmöglichen Fehlerquote. Alternativ kann sie auch als die Projketion der Einheitsmatrix betrachtet werden. Die Einträge der Massenmatrix bestehen deshalb jeweils aus dem Skalarprodukt von der Funktion $\varphi(x)$. Der Eintrag der *l-ten* Spalte und der *k-ten* Zeile ist definiert durch

$$\langle \varphi(l), \varphi(k) \rangle.$$

Um den Fehler, der durch die Annäherung mit der FE-Methode entsteht, abzuschätzen, verwendet man die *Suboptimale Fehlereinschätzung der Finiten Elemente*. Diese besagt, dass der FEM-Fehler immer kleiner ist als eine Konstante mal dem Projektionsfehler. Die Größe der Konstante hängt von der Art des Projektionsoperators und auch von den Materialeigenschaften (z.B die Steifigkeit) ab. Je kleiner die Konstante, desto besser ist die Approximation mit der FE-Methode.
Die Steifigkeitsmatrix C bekommt man durch "freches Einsetzen" in das Teilproblem $A \cdot u = f$ oder kann auch als die Projektion des Differentialoperators A interpretiert werden. Die Berechnung der Steifigkeitsmatrix ist analog zu der der Massenmatrix, jedoch verwendet man die 1. Ableitung der Funktion $\varphi(x)$ nach dem Ort:

$$\langle \frac{\partial}{\partial x}\varphi(l), \frac{\partial}{\partial x}\varphi(k) \rangle.$$

Beide Matrizen sind sogenannte Bandmatrizen, das bedeutet, dass sie nur entlang ihrer Hauptdiagonale und entlang einer bestimmten Anzahl, in diesem Fall zwei, Nebendiagonalen Einträge haben - alle anderen Einträge sind Null. Dieses Phänomen ist eine direkte Folge der Eigenschaften der Basisfunktion $\varphi(x)$, da diese für jedes x_k außerhalb des *k-ten* Hütchens verschwindet.

Das L^2-Skalarprodukt zweier Funktionen ist definiert durch:

$$\langle f(x), g(x) \rangle_{L^2} = \int_a^b f(x)g(x)dx$$

In [6]:
```
def assemble(phi,n):
    M = np.zeros((n-2,n-2)) #(n-2)x(n-2) matrix mit Einträgen 0 wird erst
ellt
    for l in range(1,n-1): #zeilen -> 1 und n-1 weil erster und letzter e
intrag nicht betrachtet werden
        for k in range(1,n-1): #spalten
            def integrand(t):
                return phi(l,t)*phi(k,t)

            M[(l-1),(k-1)] = scipy_integrate.quad(integrand,0,np.pi,limit
=len(X)+1,points=X)[0] #skalarprodukt

    return M
```

In [7]:
```
def scalarproduct(f,phi,n):
    fs = np.zeros(n-2)
    for k in range(1,n-1):
        def integrand2(t):
            return f(t)*phi(k,t)

        fs[k-1] = scipy_integrate.quad(integrand2,0,np.pi,limit=len
(X)+1,points=X)[0] #skalarprodukt
    return fs
```

Massen- und Steifigkeitsmatrix:

In [8]:
```
M = assemble(phi,n)
C = assemble(phi_strich,n)
```

In [10]:
```
print(M.round(2))
```

```
[[0.21 0.05 0.   0.   0.   0.   0.   0.  ]
 [0.05 0.21 0.05 0.   0.   0.   0.   0.  ]
 [0.   0.05 0.21 0.05 0.   0.   0.   0.  ]
 [0.   0.   0.05 0.21 0.05 0.   0.   0.  ]
 [0.   0.   0.   0.05 0.21 0.05 0.   0.  ]
 [0.   0.   0.   0.   0.05 0.21 0.05 0.  ]
 [0.   0.   0.   0.   0.   0.05 0.21 0.05]
 [0.   0.   0.   0.   0.   0.   0.05 0.21]]
```

In [11]:
```
print(C.round(2))
```

```
[[ 6.37 -3.18  0.    0.    0.    0.    0.    0.  ]
 [-3.18  6.37 -3.18  0.    0.    0.    0.    0.  ]
 [ 0.   -3.18  6.37 -3.18  0.    0.    0.    0.  ]
 [ 0.    0.   -3.18  6.37 -3.18  0.    0.    0.  ]
 [ 0.    0.    0.   -3.18  6.37 -3.18  0.    0.  ]
 [ 0.    0.    0.    0.   -3.18  6.37 -3.18  0.  ]
 [ 0.    0.    0.    0.    0.   -3.18  6.37 -3.18]
 [ 0.    0.    0.    0.    0.    0.   -3.18  6.37]]
```

Eigenwerte und -vektoren:

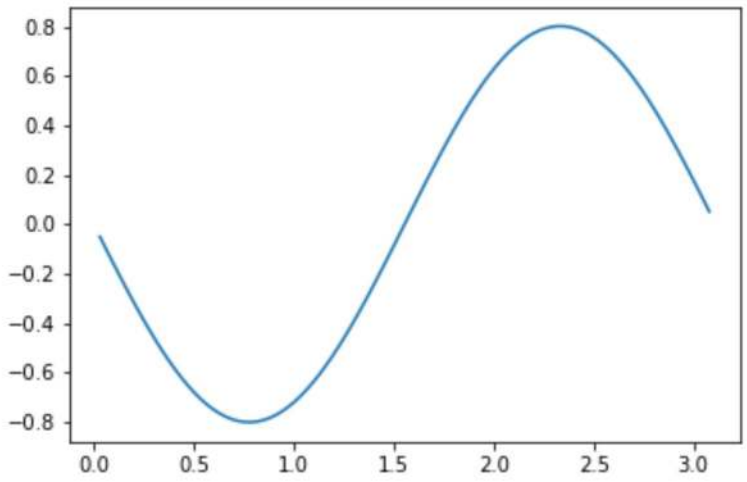
In [72]:
```
l,Q = scipy.linalg.eigh(C,b=M) #allg. Eigenwertproblem: A*x = lambda*B*x
(A=C, B=M)
```

In [73]:
```
print(l)
```

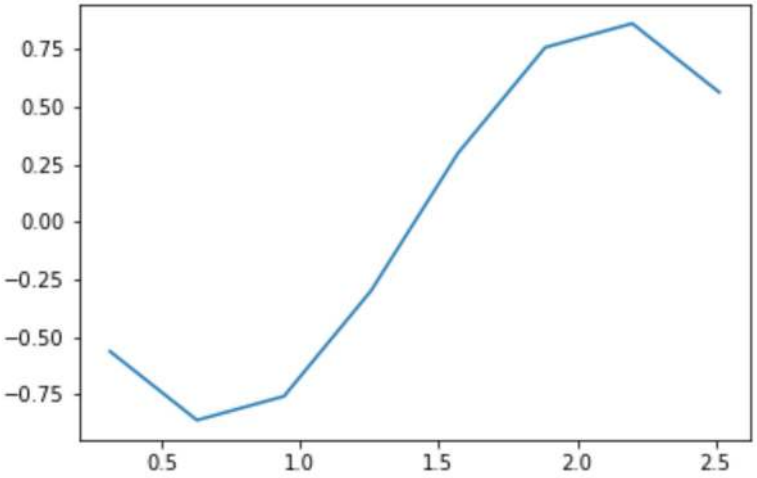
```
[  1.24715387   5.14192474  12.15854204  23.11145261  39.06654383
  60.79271019  87.00688401 111.2122519 ]
```

Annäherung an die Eigenfunktion -> man erkennt die Sinusfunktion. Je kleiner n, desto eckiger ist der Graph.

In [77]:
```
#plt.plot(X[1:-2],Q[:,1])
```



n = 100
x-Achse:x_k, y-Achse: Eigenvektoren



n = 10
x-Achse:x_k y-Achse: Eigenvektoren

Da nun die Eigenfunktion, also das grundsätzliche Schwingugsverhalten des Systems, bekannt ist, wollen wir auch noch das Verhalten der Amplituden der einzelnen Eigenmoden in Abhängigkeit von der Frequenz bestimmen. Zuerst zerlegt man die Anregungsschwingung in einzelne, einfachere Schwingungen (Fourier-Analyse). Im nächsten Schritt wird eine Lösung für u in der Gleichung

$$M \cdot u'' + C \cdot u = f$$

gesucht. Da es sich hierbei um eine Differentialgleichung, die sowohl vom Ort als auch von der Zeit abhängt, und das Lösen einer solchen relativ kompliziert ist, diskretisiert man sie nach dem Ort (die zeitliche Komponente fällt weg). Durch Umformen kommt man auf

$$(-\omega^2 \cdot I + A) \cdot \hat{y(\omega)} = \hat{f(\omega)}.$$

Eine Projektion in den Hütchenraum ergibt

$$(-\omega^2 \cdot M + C) \cdot \hat{y(\omega)} = \hat{f(\omega)}.$$

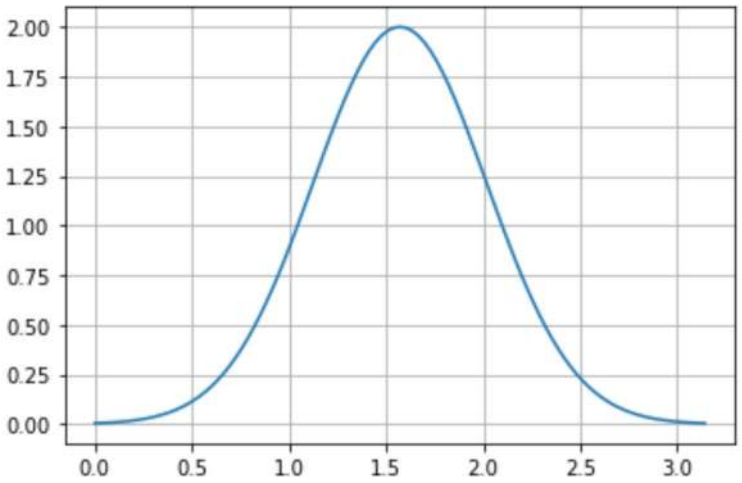
Eine Basistransformation in die Eigenvektorbasis macht die nächsten Rechenschritte deutlich einfacher: Die rechte Seite wird mit der Massenmatrix und ihrer Inversen (ergeben I) erweitert, um einen *Koeffizientenvergleich* von $M \cdot v_k$ durchzuführen. Durch weiteres Umformen und Einsetzen kommt man auf

$$\hat{u} = \frac{\hat{f}_k}{\omega_k^2 - \omega^2}$$

wobei $\hat{u}$ das Verhalten der Amplitude der jeweiligen Eigenmoden beschreibt.

In [19]:
```
f(x) = 2*exp(-((x-np.pi/2)**2)/0.4)
F = np.array([f(x) for x in X])
```

In [22]:
```
plt.plot(X,F)
plt.grid(True)
plt.show()
```



Graph der Anregungsfunktion f

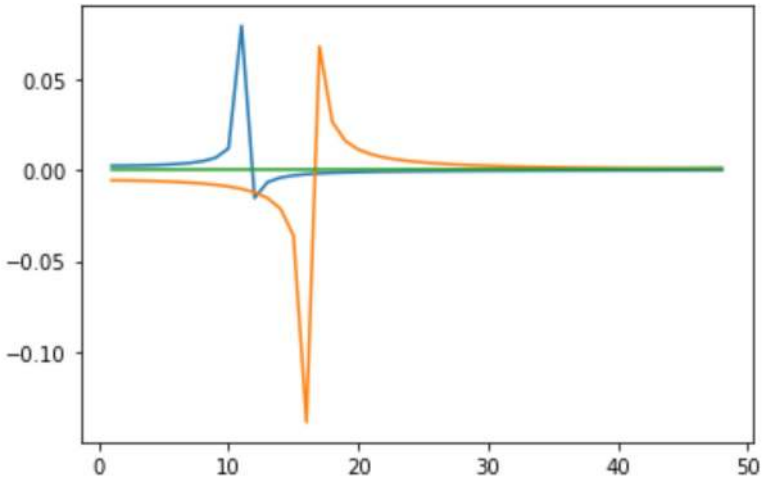
In [20]:
```
fs = scalarproduct(f,phi,n)
```

In [21]:
```
M_inv = scipy.linalg.inv(M)
Mf = M_inv.dot(fs)
```

In [22]:
```
f_dach = Mf.dot(Q.T)
```

In [23]:
```
U_dach = np.zeros(n-2, dtype=object)
Omega = np.arange(1,(n-1)//2)
for k in range(1,n-1):
    ell = l[k-1]
    U_dach[k-1] = f_dach[k-1]/(ell-Omega**2)
```

In [68]:
```
U_dach[0]
plt.plot(Omega,U_dach[10])
plt.plot(Omega,U_dach[15])
plt.plot(Omega,U_dach[50])
plt.show()
```



Der Graph zeigt das Verhalten der Amplituden ($\hat{u}$) in Abhängigkeit von der Frequenz ω. Für jede Eigenmode gibt es einen einzelnen Graphen. Exemplarisch sind hier die Amplituden der 10., 15. und 50. Eigenmoden dargestellt.

Durch Rekombination und Rückeinsetzen der berechneten Werte kann das Verhalten der Amplituden nun auch abhängig vom Ort (x_k) gezeigt werden.

Spektralmethode

In [2]:
```
import scipy
import numpy as np
```

In [3]:
```
def project_vector(f,phi,C=2/np.pi):
    F = [numerical_integral(f(x)*phi(k,x),0,pi)[0]*C for k in range(N)]
    return F
```

In [4]:
```
def funktion_ableitung(f):
    var("t")
    df = derivative(f(t),t)
    return df
```

In [14]:
```
N = 10
var("k")
phi(k,x) = sin(k*x)
```

In [15]:
```
f(x) = x*(np.pi-x)
g(x) = cos(x)
```

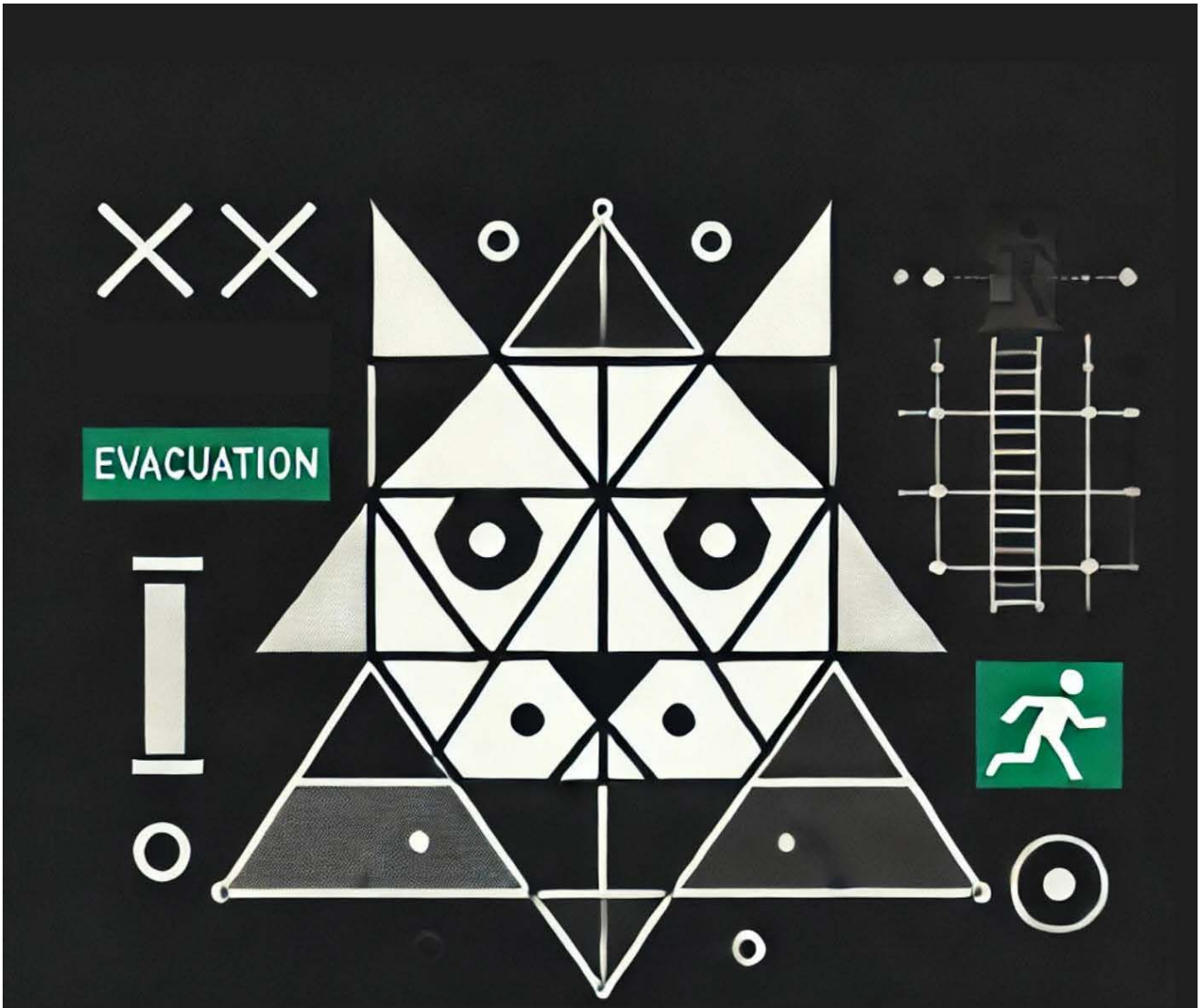
In [17]:
```
F1 = project_vector(f,phi)
F2 = project_vector(g,phi)
print(F1)
```

```
[0.0, 2.5464790894703255, 2.450326347649432e-16, 0.09431404035075273,
-1.4342313076640334e-16, 0.020371832715763045, -3.5564409411292e-18, 0.0
07424137287085315, -1.5068041602523254e-16, 0.0034931126055844064]
```

In [18]:
```
df1 = funktion_ableitung(f)
df2 = funktion_ableitung(g)
```

In [20]:
```
DF1 = project_vector(df1,phi)
print(DF1)
```

```
[0.0, -1.4028276694997357e-16, 1.9999999999999998, -2.735666126146123e-1
6, 1.0000000000000004, -8.409176553967707e-17, 0.6666666666666666, 1.202
2067085727367e-16, 0.4999999999999999, -2.3928661187938287e-16]

/opt/sagemath-9.3/local/lib/python3.7/site-packages/sage/repl/ipython_ke
rnel/__main__.py:2: DeprecationWarning: Substitution using function-call
syntax and unnamed arguments is deprecated and will be removed from a fu
ture release of Sage; you can use named arguments instead, like EXPR(x
=..., y=...)
See http://trac.sagemath.org/5930 for details.
  from sage.repl.ipython_kernel.kernel import SageKernel
```



ANTHROPOMETRIE

DER MENSCH IM MODELL

Teilnehmer: Valerie Hermann, Tina Matschinegg, Clara Muri, Navid Pavel, Klara Ranz, Julia Reuter, Moritz Röhrer, David Toplak

Gruppenleiter: Michael Fischer

Anthropometrie

1.) Methoden zur Berechnung des Idealgewichts

BMI-Body mass index

Broca-Normalgewicht

FFMI-Fettfreie Masse Index

BRI-Body roundness index

WHR - Waist Hip Ratio

WtHR - Waist to Height Ratio

2.) Modellierung zur Berechnung des Idealgewichts für Katzen

Polynomfunktion 3. Grades

Lineare Funktion

Quadratische Funktion

Exponentialfunktion

Interpretation

- ➔ Polynomfunktion 3. Grades
- ➔ Lineare Funktion
- ➔ Quadratische Funktion
- ➔ Exponentialfunktion

3.) Unser Modell: PCTC: Waist to Hip Ratio für Katzen

Optimales Gewicht:

Untergewichtige Katze

Übergewichtige Katze

Rechenprogramm

4.) Paper lesen, Daten analysieren

5.) Models for pedestrian dynamics

6.) Natural discretization of pedestrian movement in continuous space

7.) Positive und negative Aspekte unseres Arbeitsweges

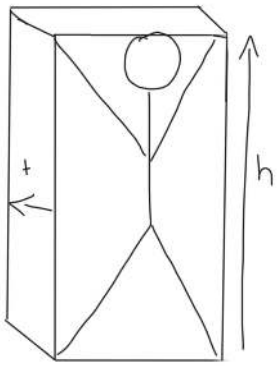
8.) Skilift-Experiment

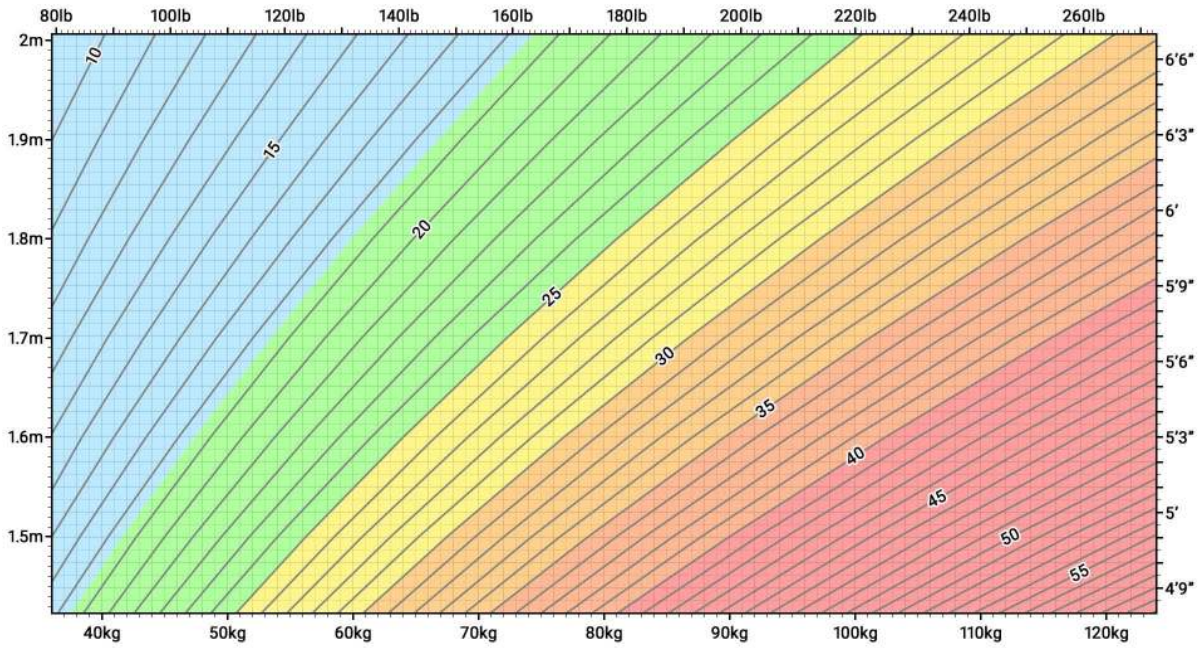
Methoden zur Berechnung des Idealgewichtes

BMI - Body mass Index

$$BMI = \frac{m(kg)}{h(m)^2}$$

- Anhaltepunkt
- kein Geschlecht, Alter etc.
- Körper wird als Würfel dargestellt -> Querschnittsfläche wird rausdividiert





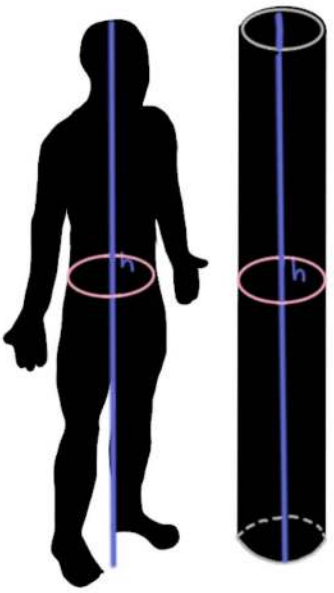
Unsere Ergebnisse:

	BMI
Moritz	19.8882205624224
Klara	21.456894631321
Valerie	19.5854290504496
Clara	24.4628099173554
David	17.2972972972973
Julia	19.9956452315285
Navid	20.1558720773985
Michael	22.8910680529301
Patrick	25.5307712980382
Tina	21.8464218319059

Broca Normalgewicht

$$Normalgewicht = (h - 100) * 0.9(M) \, v \, 0.85(F)$$

	Broka Idealgewicht	Abweichung Idealgewicht abs.	Abweichung Idealgewich rel.
Moritz	76.05	-8.35	-10.979618671926
Klara	57.375	2.825	4.923747276688
Valerie	52.7	-1.3	-2.466793168880
Clara	55.25	11.35	20.542986425339
David	76.5	-17.3	-22.614379084967
Julia	56.1	-1	-1.782531194295
Navid	74.7	-7.2	-9.638554216867
Michael	75.6	1.9	2.513227513227
Patrick	74.7	10.8	14.457831325301
Tina	56.1	4.1	7.308377896613



- einfach
- Geschlecht wird berücksichtigt
- alt
- linear
- Körper als Zylinder

FFMI - Fettfreie Masse Index

FFM = Fettfreie Masse KFA = Körperfettanteil

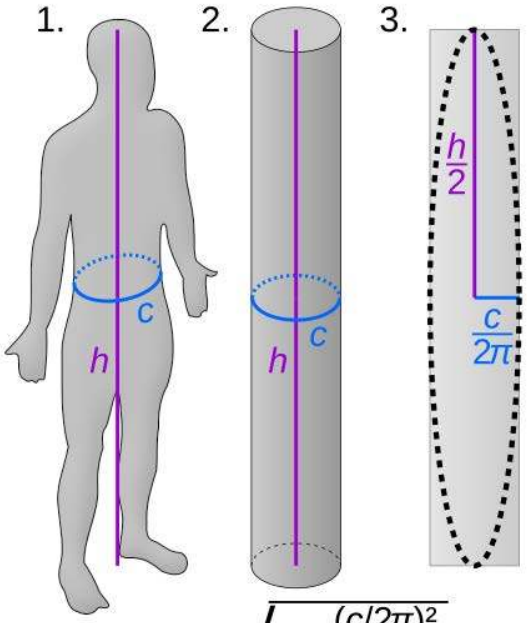
$$FFM = m(kg) * (1 - \frac{kFA}{100})$$

$$FFMI = \frac{FFM}{h(m)^2} + 6.3 * (1.8 - h(m))$$

- wird für Bodybuilding Wettkämpfe in Deutschland verwendet(Zulassungskriterium)
- unhandlich (schwierig den Körperfettanteil zu messen)

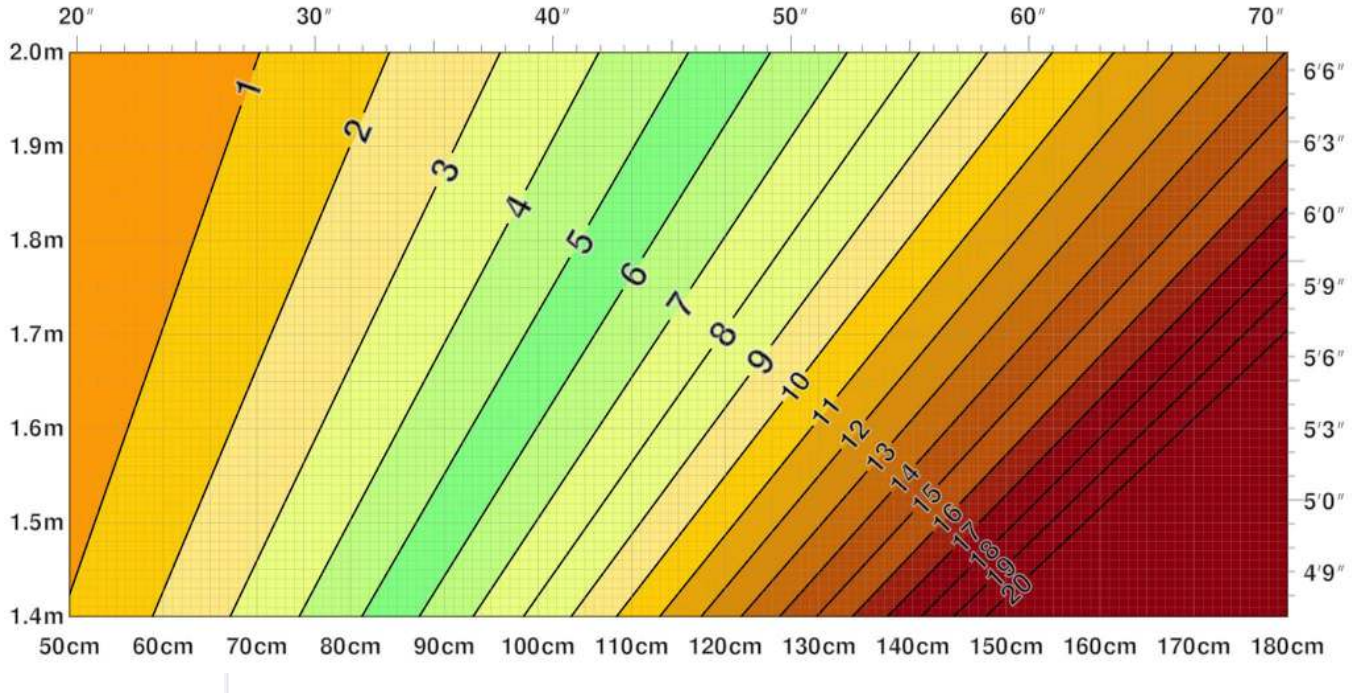
BRI - Body Roundness Index

$$BRI_1 = 364.2 - 365.5 * \sqrt{1 - (\frac{waist}{\pi * h})^2}$$



$$e = \sqrt{1 - \frac{(c/2\pi)^2}{(h/2)^2}}$$

$$BRI = 364.2 - 365.5e$$



	Body Roundness Index 1
Moritz	1.591766827332
Klara	1.7566318727204
Valerie	1.5940225946537
Clara	2.344568580568
David	1.8227961965872
Julia	2.1011422172983
Navid	1.7243268105799
Michael	2.5741616959749
Patrick	3.6259356764026
Tina	2.50312008124

- nicht für Jugendliche geeignet (wir sind alle extrem untergewichtig, Vermutung: auf Amerika ausgelegt?!)
- besser geeignet für exaktere Nuancen bei Übergewichtigen

-> andere Formel

$$BRI_2 = 364.2 - 365.5 * \sqrt{1 - \left(\frac{waist}{2*h}\right)^2}$$

- dadurch entsteht kein Ellipsoid

	Body Roundness Index 2
Moritz	6.1250942897809
Klara	6.5369504790176
Valerie	6.1307275820395
Clara	8.0080446526366
David	6.7023190117458
Julia	7.3985142617868
Navid	6.4562252748358
Michael	8.5835179334507
Patrick	11.2270667222086
Tina	8.4053922919191

- angepasst an uns -> wir sind jetzt im Normalgewicht
- Geometrisch macht es aber keinen Sinn
- Formel von ChatGPT

WHR - Waist Hip Ratio

$$WHR = \frac{waist}{hip}$$

	WHR
Moritz	0.8131868131868
Klara	0.8414634146341
Valerie	0.8227848101266
Clara	0.8505747126437
David	0.8850574712644
Julia	0.8470588235294
Navid	0.8620689655172
Michael	0.8673469387755
Patrick	0.959595959596

- < 0.85 für Frauen -> Normal
- < 0.9 für Männer -> Normal

- Die Mädchen sind eher knapp an der Obergrenze
- linear

WtHR - Waist to Height Ratio

$$WtHR = \frac{waist}{h}$$

	WtHR
Moritz	0.4010840108401
Klara	0.4119402985075
Valerie	0.4012345679012
Clara	0.4484848484848
David	0.4162162162162
Julia	0.433734939759
Navid	0.4098360655738
Michael	0.4619565217391
Patrick	0.5191256830601
Tina	0.4578313253012

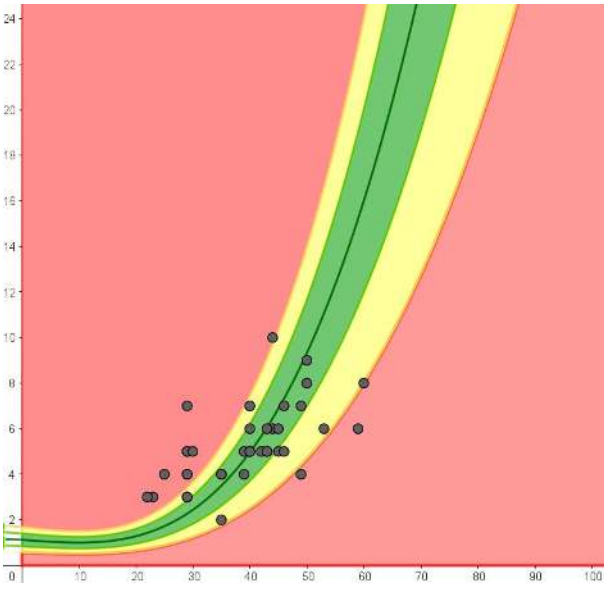
-

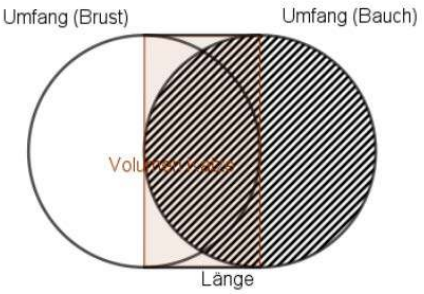
- Normal = 0.4 - 0.49
- linear
- passt für uns gut

Typ	Positiv	Negativ
BMI – Body Mass Index	Einfach Anhaltspunkt	Kein Gendern
Broca-Index	Einfach	Linear (unnatürlich)
FFMI- Fettfreier Masse Index	Gut für Sportler/innen	Umständlich, kompliziert
BRI- Body Roundness Faktor	exakte Nuancen für Fettleibige bestimmen	Nicht für jedes Alter, Körper geeignet
WHR- Waist Hip Ratio	Einfach Geschlechtsunterscheidung	ungenau
WtHR – Waist to Hight Ratio	Einfach	Linear (ungenau)

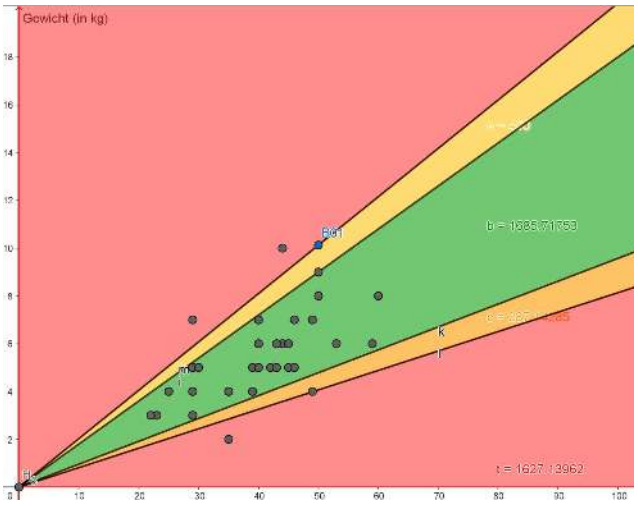
Modellierung zur Berechnung des Idealgewichts für Katzen

Polynomfunktion 3. Grades

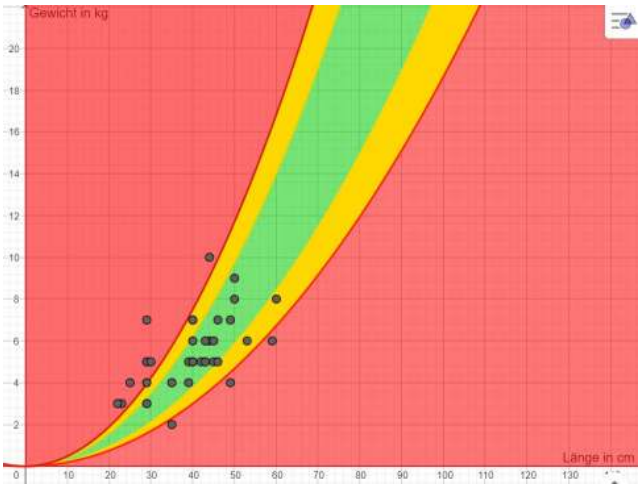




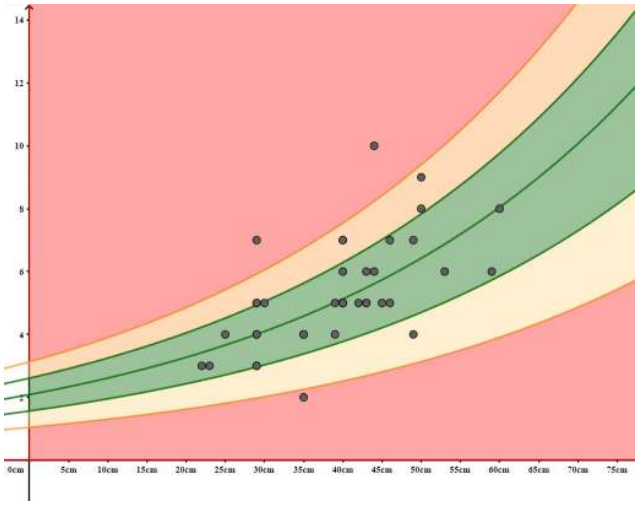
Lineare Funktion



Quadratische Funktion



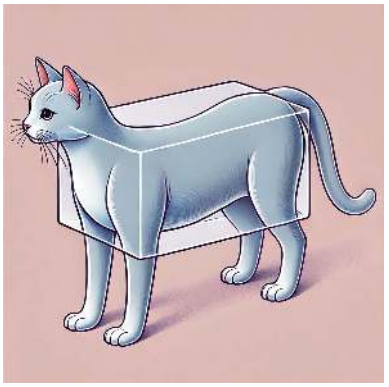
Exponentialfunktion



Interpretation

Kubische Funktion (Polynomfunktion 3. Grades)

$f(x)=ax^3+ bx^2+cx+d$

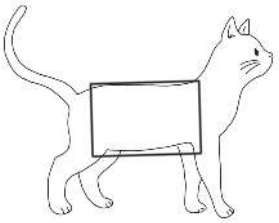


- Das Modell wächst in alle drei Dimensionen
- Für kleine Werte (Katzen) trifft dieses Modell zu, jedoch nicht für große Werte (Tiger)

Lineare Funktion

$f(x)= k*x+d$

- Wenn die Katze in die Länge wächst, steigt das Gewicht linear
- Daraus folgt je länger die Katze ist, desto schwerer darf sie sein
- Das Modell ist einfach und verständlich aufgebaut und ist bei kleinen Werten von Nutzen, jedoch ist die Spannbreite bei großen Längen zu hoch
- Außerdem ist es so, dass die Katze nicht linear wächst



Quadratische Funktion

$f(x)=ax^2+bx+c$

- Anfangs trifft das Modell für kleine Werte zu, doch die zugelassenen Idealgewichte für große Körper sind zu groß, was nicht der Realität entspricht

Exponentialfunktion

N(x)=N0*a^x

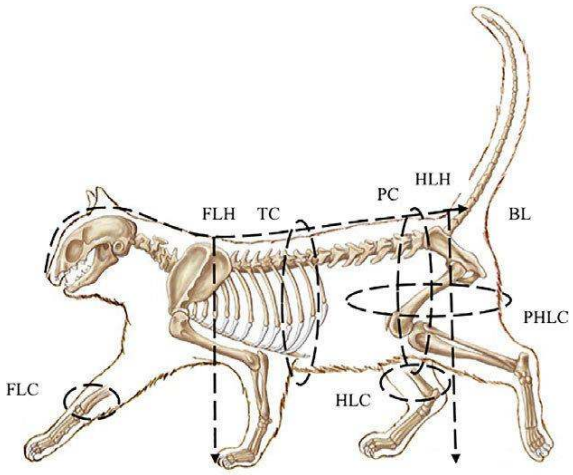
- Die Katze wächst in die Länge und das Gewicht steigt prozentual
- 5:11 Regel, das heißt:

- Katze wächst um 5 cm, das Gewicht nimmt um 11,5% zu
-Katze wächst um 10 cm, das Gewicht nimmt um 24,3% zu

- Das Modell ist am besten, da es dem natürlichen Wachstum der Katze am nächsten kommt
- Ist für Babykatzen nicht geeignet

Unser Modell: PCTC

Waist to Hip Ratio für Katzen



Unser PCTC Modell basiert auf der **Waist-to-Hip Ratio (WHR)** für Menschen und dem **BCS Score** für Katzen. Durch das ins Verhältnis setzen ist es dimensionslos und wird berechnet als:

$$PCTC_{Score} = \frac{Pelvic\ circumference}{Thoracic\ circumference}$$

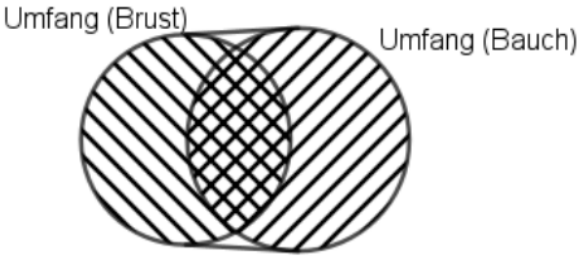
wobei:

- *Pelvic Circumference* der Umfang des Bauches der Katze (in cm) ist, und
- *Thoracic Circumference* Umfang der Brust der Katze (in cm) ist (Hinter den Vorderbeinen messen).

<u>Optimales Gewicht:</u>

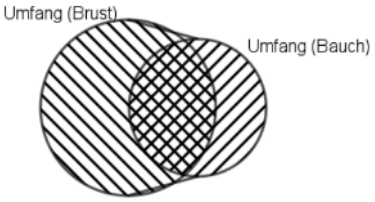
$$\frac{Pelvic\ circumference}{Thoracic\ circumference} = 1 \ldots => Optimal\ weight$$

Je näher sich die PC der TC annähert, desto eher wird der Quotient 1.



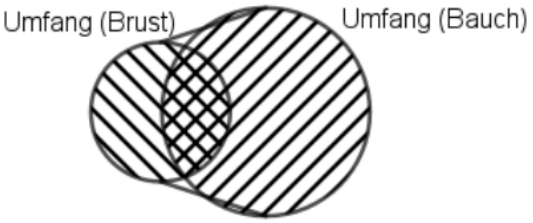
Untergewichtige Katze:

$$\frac{Pelvic\ circumference}{Thoracic\ circumference} < 1 \ldots => \ Underweight$$

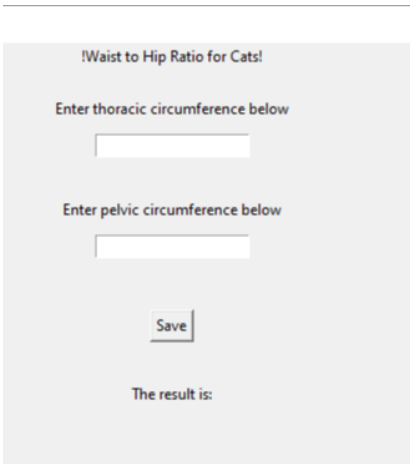


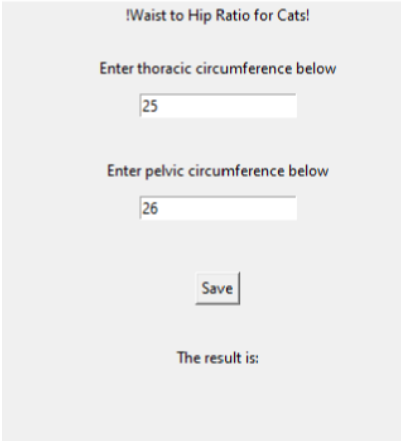
Übergewichtige Katze:

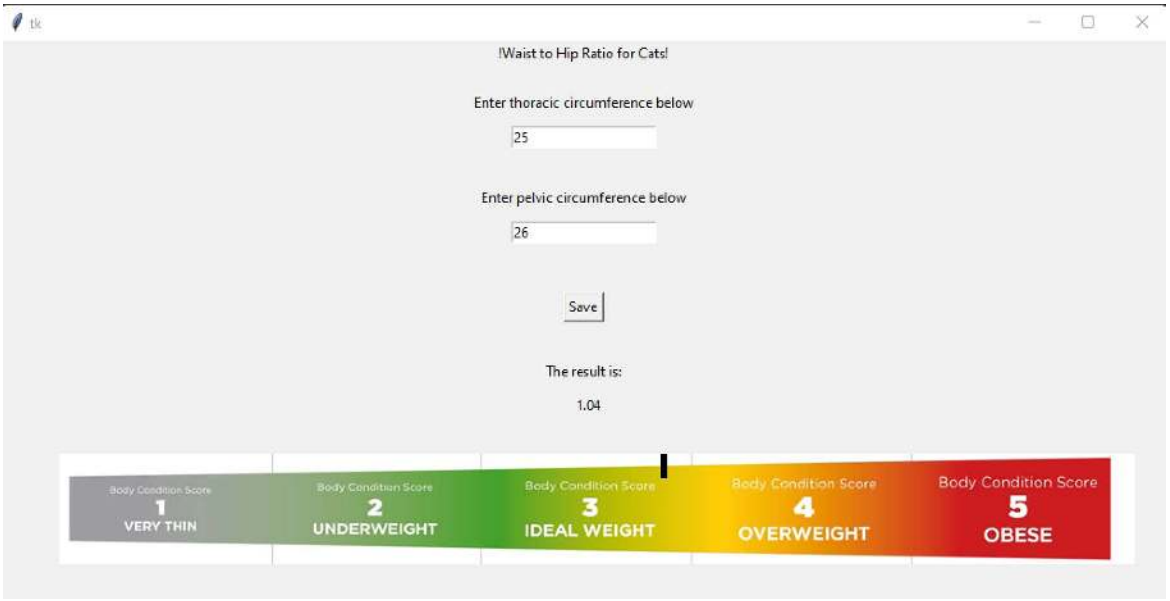
$$\frac{Pelvic\ circumference}{Thoracic\ circumference} > 1 \ldots => \ Overweight$$



Hierfür haben wir für den alltäglichen Benutzer auch ein kleines Rechenprogramm geschrieben:



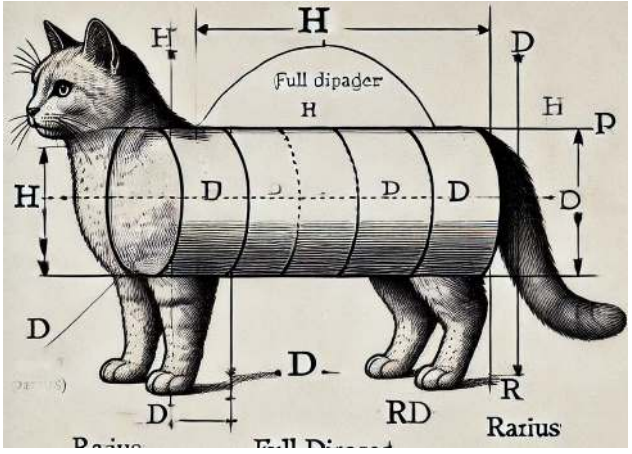


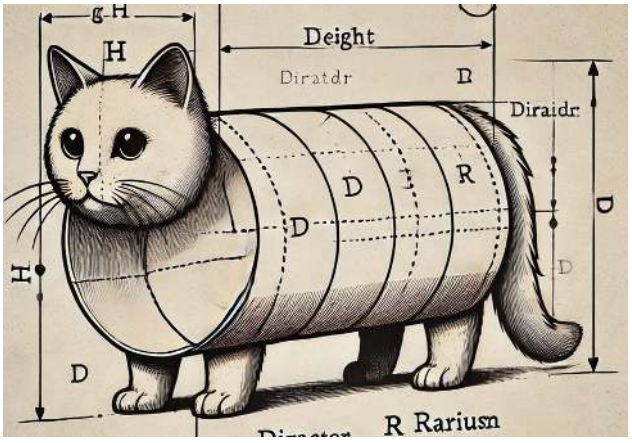


In diesem Fall ist die Katze leicht übergewichtig.

Paper lesen, Daten analysieren

Wir hatten große Schwierigkeiten, originale Daten und Messungen von Katzen zu finden. Ein Paper aus 1936 kam uns dann doch zu Hilfe.





American Journal of Anatomy

Article

Weights and linear measurements of the adult cat

Homer B. Latimer

First published: March 1936 | https://doi.org/10.1002/aja.1000580204 | Citations: 24

Each cat was killed by chloroform, weighed and then the linear measurements taken. A flexible steel tape, graduated

ischial tuberosity. Next the skin was removed and then the head and the extremities were separated from the body and

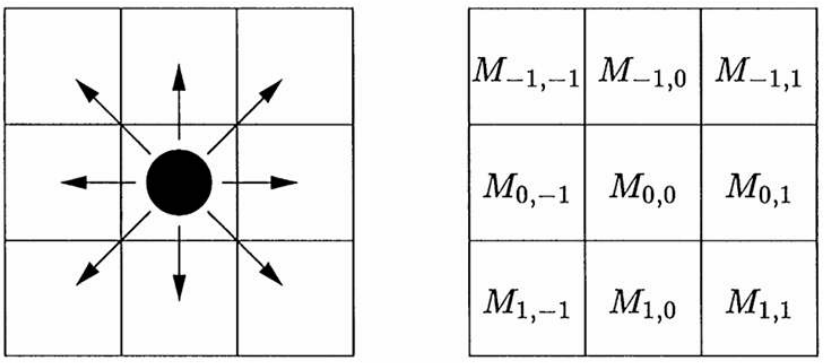


Fig. 1. A particle, its possible transitions and the associated matrix of preference $M = (M_{ij})$.

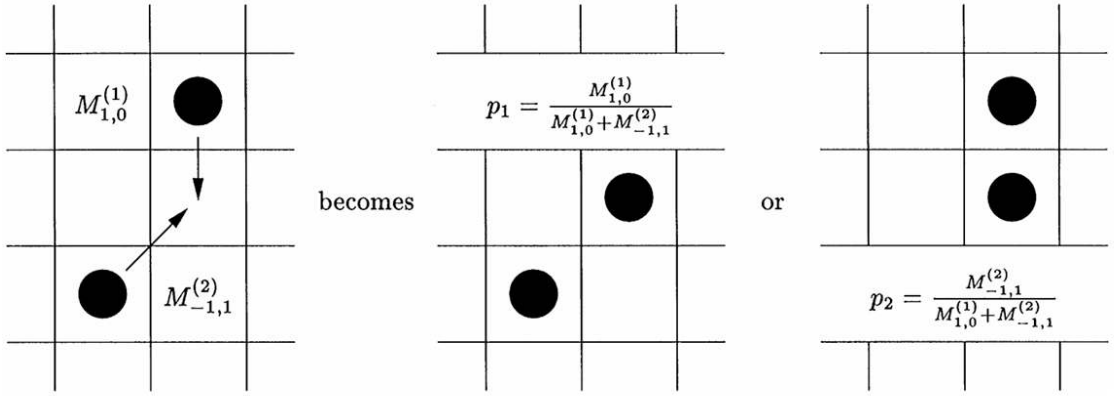


Fig. 2. Solving conflicts according to the relative probabilities for the case of two particles with matrices of preference $M^{(1)}$ and $M^{(2)}$.

Das oben dargestellte Modell beruht auf dem Konzept des zellulären Automaten. Der Mensch wird hier als Kreis in einem Raster, das aus Quadraten besteht, dargestellt und kann sich in 8 Richtungen - die benachbarten Quadrate - bewegen. Das dargestellte Beispiel modelliert eine Situation, bei der sich zwei Menschen auf das gleiche Feld bewegen möchten. In diesem Fall entscheiden relative Wahrscheinlichkeiten über den Ausgang der Situation.

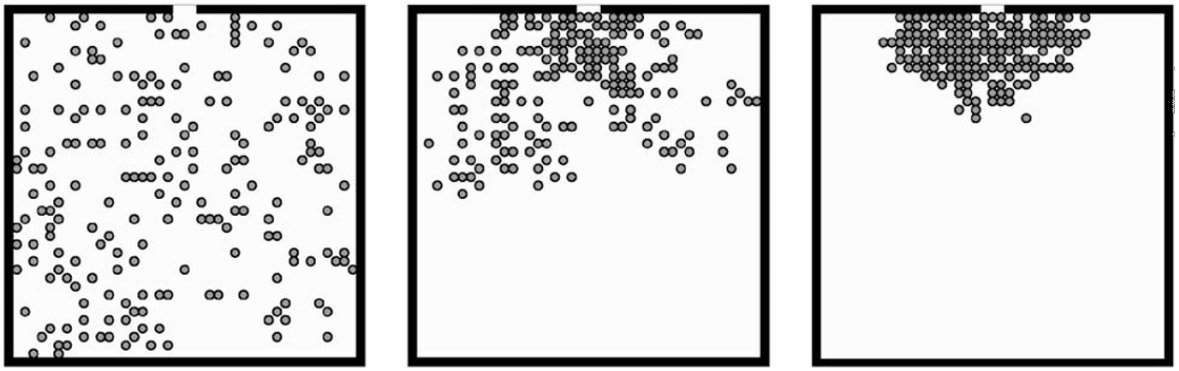


Fig. 3. People leaving a room with one door only. Displayed are three typical stages of the dynamics.

Models for pedestrian dynamics

Mithilfe dieses Papers konnten wir die Grundlagen der verschiedenen Modelle in Evakuierungs Szenarien besser verstehen

Bridging the gap: From cellular automata to differential equation models for pedestrian dynamics

Felix Dietrich [a b], Gerta Köster [a], Michael Seitz [a b], Isabella von Sivers [a b]

https://doi.org/10.1016/j.jocs.2014.06.005

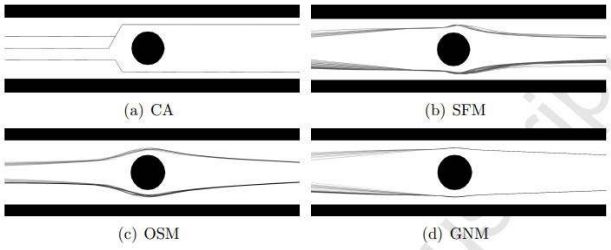
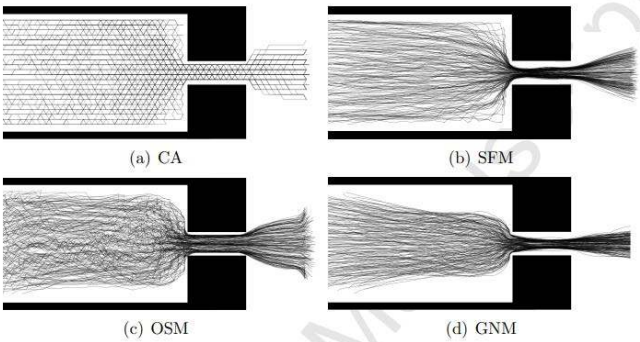
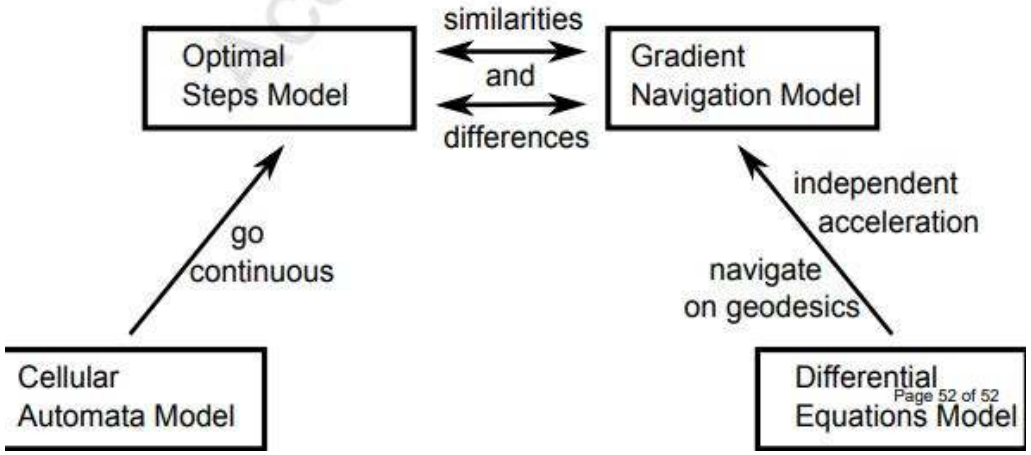


Figure 6: Trajectories for pedestrians passing from left to right. Pedestrians walk around a pillar; the opening between the wall and the pillar is exactly the torso diameter.

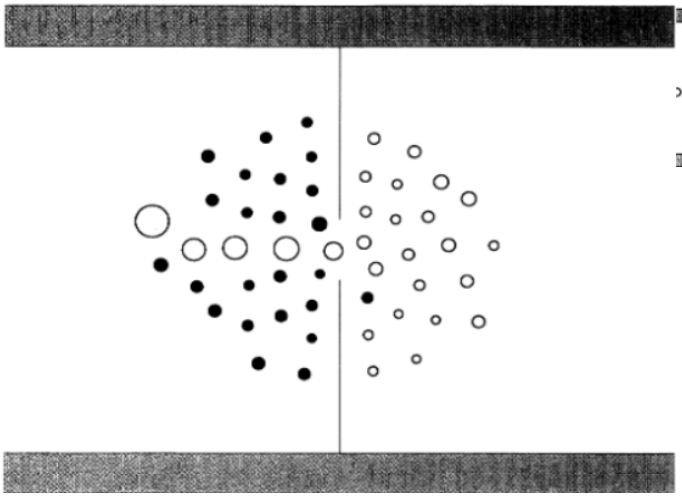
the overall behavior of the OSM and that of the GNM are very similar – forming a bridge between the CA and the SFM.

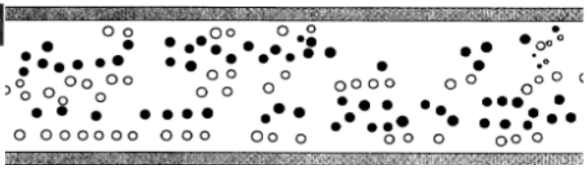


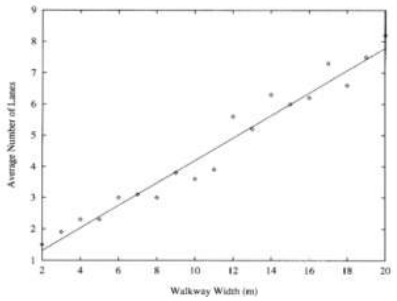


Social force model for pedestrian dynamics

Dirk Helbing and Péter Molnár
II. Institute of Theoretical Physics, University of Stuttgart, 70550 Stuttgart, Germany
(Received 14 April 1994; revised manuscript received 5 January 1995)







Der Mensch wird anders als bei anderen Modellen als Kreis dargestellt, der auch den Personal Space darstellt.
Zusätzlich gibt es abstoßende und anziehende Effekte, die die Menschen in gewisse Richtungen wegdrücken (wenn man zu nahe an wen anders, oder ein Objekt kommt) oder anziehen (ein Ziel zB). Zusätzlich wird der Mensch mit einem Sichtfeld ausgestattet, der anziehende Effekt wirkt beispielsweise stärker, wenn er in Blickrichtung des Menschen liegt. Im linken Bild erkennt man, dass die Kreise unterschiedliche Größen haben, das liegt an der normalverteilten Geschwindigkeit der Person und inwieweit sie diese Geschwindigkeit auch tatsächlich erreichen kann. Langsamere Menschen sind kleinere Kreise als Schnelle.

Grafik Links: Oszillation der Fußgänger: Öffnung, wo von jeweils beiden Seiten Menschen hindurchgehen wollen. Es zeigt sich, dass eine Art Oszillation auftritt, wo in einem Zeitabschnitt immer jeweils eine Seite durchgeht, durch zu hohen Andrang stoppt diese Bewegung dann, und die anderen können durchgehen.

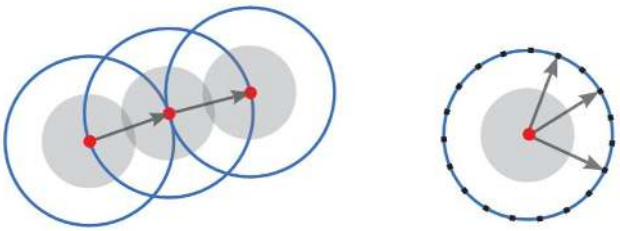
Grafik Rechts(oben): Linienbildung der Fußgänger: Bei einem Gangabschnitt, wo von beiden Richtungen Menschen den Gang passieren wollen. Empirisch getestet

findet dann eine Linienbildung statt, was in diesem mathematischen Modell auch passiert.

Grafik Rechts(unten): Hier sieht man ein interessantes Resultat zu dem dieses Modell gekommen ist. Auf der x-Achse ist die Breite des Ganges aufgetragen, auf der y-Achse ist die durchschnittliche Anzahl an gebildeten Linien abgebildet. Tatsächlich ist die Gangbreite zur durchschnittlichen Linienbildung direkt proportional.

Natural discretization of pedestrian movement in continuous space

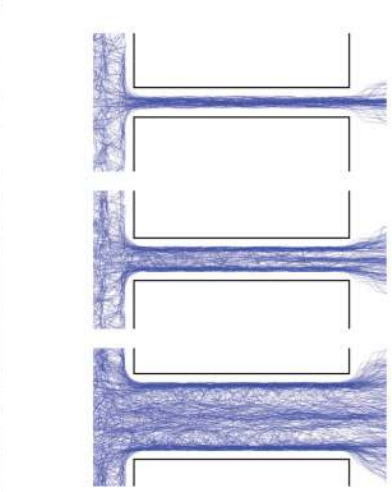
1. Der Oberkörper des Fußgängers wird durch den ausgefüllten inneren Kreis dargestellt. Die nächste Position muss auf dem Schrittkreis um den Fußgänger liegen.



2. Diskretisierung des durch Punkte gekennzeichneten Kreises.

Die drei Kreise stellen drei mögliche Auswahlmöglichkeiten für den nächsten Schritt dar.

Drei gleich lange Gänge



1. 0,7 m breit, Die Menschen werden als Linien dargestellt. Sie haben keinen Platz nebeneinander zu gehen ,weshalb sie eine Reihe bilden.
2. 1m breit , Die Menschen bilden an den Rändern zwei deutlich erkennbare Reihen und am Ende entsteht noch eine dritte in der Mitte.
3. 2 m breit , Erneut bilden die Menschen zwei deutliche Reihen an den Rändern und 2-3 und undeutliche in der Mitte.

Positive und negative Aspekte unseres Arbeitsweges

- Positiv:
- Werte ausrechnen
- Interpretation der Ergebnisse
- Entdeckung neuer Anhaltspunkte
- Teamwork
- Brainstorming
- Abmessungen
- Umsetzung der Snackwünsche

- Negativ:
- Finden echter/nutzbarer Daten
- Konstruktion und Verstehen mancher Modelle
- ChatGPT teilweise hilfreich, teilweise Blödsinn
- Zurechtfindung in Geogebra
- verschiedene Versionen von Geogebra
- Punkte und Tabellen
- Manchmal auf falsche Fährten gelenkt

Skilift Simulator

Experiment

DG1_Simulator

Anfangs sind die Leute einzeln hingegangen und es war flüssig, doch am Ende hat es sich vor den Hürden gestaut (unabhängig von dem zu frühen Sturz), es war eine leichte Traube zu erkennen

DG2_Simulator

Es ging etwas langsamer, die Partner waren von Anfang an beisammen, die Leute haben sich von Anfang an besser geordnet,

DG3_Simulator

Insgesamt hat der Durchgang länger gedauert, durch die Störfaktoren (Julia und Clara) staute es sich schon am Anfang und die Personen sind links und rechts ausgewichen, auch vor den Hürden gab es einen leichten Stau

DG4_Simulator

Durch die Personal Spaces hat der Durchgang länger gedauert, vor den Hürden war der bisher längste Stau,

DG5_Simulator

Der Stau vor den Hürden war wieder sehr lang, da die Partner wieder beisammen waren, haben sie keinen großen Unterschied gemacht

DG6_Simulator

Mit Personal Space und Störfaktoren war der Stau vor den Hürden deutlich größer, vor den Hürden mussten die Leute sogar stehen bleiben

DG7_Simulator

Mit Personal Space, Partner und Störfaktoren war der Stau am längsten, vor allem wegen Clara und Julia, Leute warteten und schauten zu anstatt, dass sie weitergehen

	1. Reihe	2. Reihe	3. Reihe	Zeit**	Stau**
1. DG*	10	8	8	17	8
2. DG*	8	8	10	15	10
3. DG*	9	9	8	15	8
4. DG*	9	8	9	16	10
5.DG	7	9	10	16	13
6. DG	10	7	9	20	14
7. DG	6	9	11	22	20

**in Sekunden

*DG=Durchgang

Versuchsaufbau

Um die Menschenmassen an einem Skilift zu simulieren, haben wir einen ,,Parcour'' im Athletik Raum auf der Laufstrecke aufgebaut.

Aufbau des Parcours:
Links und rechts haben wir Abgrenzungen zu den Wänden aufgestellt, um den Spielraum kleiner zu halten. Danach haben wir mithilfe der Hürden drei Schranken gebaut und dahinter mithilfe von Schaumwürfeln einen ,,Auffang Pool" für die Menschen.

Aus drei unterschiedlichen Perspektiven haben wir die Durchführung des Experiments aufgenommen, um das Material anschließend auswerten zu können.

Durchführung:
Um bestmögliche Ergebnisse zu erzielen, haben wir viele Teilnehmer eingeladen, bei unserem Experiment mitzumachen. Die 26 Personen (darunter nicht Patrick Frühmann) mussten die Schranken passieren und in jedem der sieben Durchgänge verschiedene Aufgaben ausführen.

Durchgänge:

1. Den Parcour ohne Aufgaben passieren
2. Den Parcour, wenn vorhanden mit Partner, passieren (2x3, 3x2)
3. Den Parcour mit unbekannten Störfaktoren passieren (drängeln, trödeln, hinfallen)
4. Den Parcour mit Personal Space (Reifen/Ball) passieren
5. Den Parcour mit den früheren Partnern und PS passieren
6. Den Parcour mit PS und den früheren Störfaktoren passieren
7. Den Parcour mit PS, Störfaktoren und Partnern passieren => Apocalypse

Unsere Rollen:

- Moritz = Drängler
- Clara & Julia = Trödler

- Michael = Hinfaller
- Valerie = normaler Geher
- Tina = Kommandantin
- Klara & Navid & David = Kamerafrau/mann

Forschungsfragen:

- Wo staut es?
- Wieso staut es? (Partner, Störfaktoren, PS)
- Wann staut es?
- Welche Einflüsse haben PS/Skiausrüstung

Erwartungshorizont:

- Vor den Hürden bildet sich eine Traube
- Partner haben größeren Einfluss als Störfaktoren
- Skitraube ist größer und löst größeres Gedränge aus
- Strichanordnung
- Apocalypse = Apocalypse

Quellen:

https://journals.sagepub.com/doi/full/10.1177/1098612X16649525

https://journals.sagepub.com/doi/10.1016/j.jfms.2009.07.008?icid=int.sj-full-text.similar-articles.1

https://journals.sagepub.com/doi/full/10.1177/1098612X16649525

https://www.mdpi.com/2076-2615/11/8/2246

https://avmajournals.avma.org/view/journals/javma/240/5/javma.240.5.570.xml

https://www.sciencedirect.com/science/article/pii/S0167587719304131?casa_token=Kxkxt9psgDcAAAAA:JAVgFX7v93Q47vz0j-WccVOuAje_ebZSGcM4pP3zzPfQYE16rqcQs24LPCii0H3RjxopfPp77A

https://journals.sagepub.com/doi/full/10.1177/1098612X221117348

https://www.tandfonline.com/doi/full/10.2147/VMRR.S40869#d1e134

https://onlinelibrary.wiley.com/doi/full/10.1111/j.1939-165X.2010.00227.x?casa_token=s8Df6BMPcZ4AAAAA%3AYU6auG_ysTwfEN3SEaEZcqJdHgiPUouEDInQF9TPu7gHVA6QRK7GMHDhEyW320oLE-cUhaUUGncfdw

https://www.sciencedirect.com/science/article/pii/S0167587700001471?casa_token=RHti0gYsNIcAAAAA:kvIsFpUX2jFxSpmjsGN2g8ktfpb9TI3dtk7y0q3bR21Gvlstz6niCezOl-XyPTtKqEaMx7ewDQ
https://images.app.goo.gl/e1TNUSMxJCpyKTpY8
https://www.kaggle.com/datasets/warcoder/cat-breeds-details
https://onlinelibrary.wiley.com/doi/pdf/10.1002/aja.1000580204
https://kindnessspreading.com/cat-breeds/cat-breeds-body-data-female/
https://www.kaggle.com/datasets/ukveteran/weight-data-for-domestic-cats/code
https://www.sciencedirect.com/science/article/abs/pii/S0378437101001418
https://www.researchgate.net/publication/1947096_Social_Force_Model_for_Pedestrian_Dynamics
https://www.sciencedirect.com/science/article/abs/pii/S1877750314000738
https://journals.aps.org/pre/abstract/10.1103/PhysRevE.86.046108
https://hal.science/tel-04318740/
https://www.asim.uni-wuppertal.de/fileadmin/bauing/asim/Thesen/Masterthesis_Ben_Hein.pdf
https://utheses.univie.ac.at/detail/53686

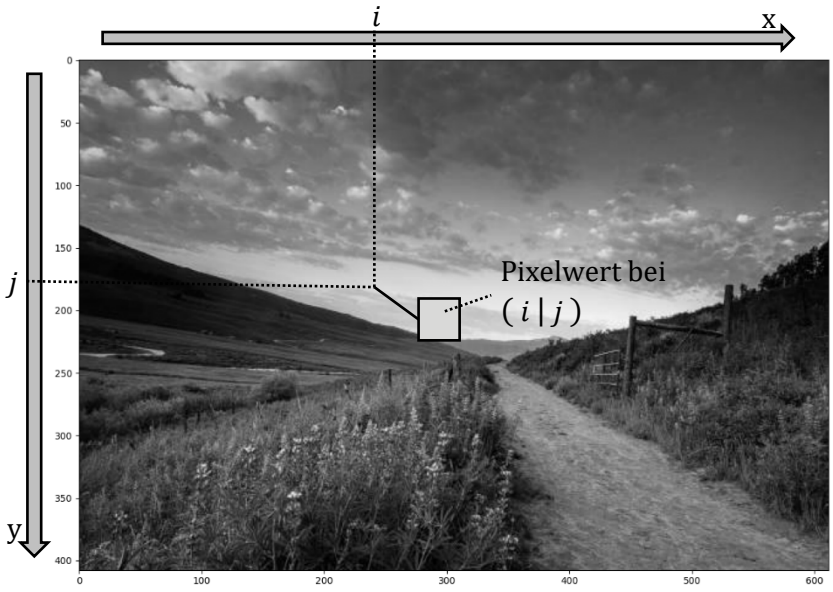
Modellierungswoche 2025 – Dokumentation zum Thema Image Stitching

1. Einleitung

Wie funktionieren Bilder? Diese zunächst sehr trivial klingende Frage sowie deren Transformationen und deren Kombination hat ein Team der Modellierungswoche 2025, bestehend aus Xandi, Helene, Moritz, Simon, Luisa, Benni und Maximilian, für eine gesamte Woche beschäftigt. Im Folgenden befindet sich eine Dokumentation der Errungenschaften dieser Arbeitsgruppe aus Sicht der Teilnehmerinnen und Teilnehmer.

2. Grundlagen

Am ersten Tag, dem 08.02.2025, an dem zunächst die Gruppeneinteilungen und ein Kennenlernspiel stattgefunden haben, haben wir uns sofort der Frage gewidmet, wie ein Bild im mathematischen Sinne beschrieben werden kann. Dabei hat sich herausgestellt, dass ein Bild als rasterförmiges Zahlenschema dargestellt werden kann. Einzelne Abschnitte dieses Rasters werden Pixel genannt und durch zwei Indizes (meist i und j), die jeweils bei dem Wert Null starten, gekennzeichnet.



Jeder Pixel kann einen Wert zwischen 0 und 255 ($= 2^8 - 1$) annehmen, wobei 0 für „schwarz“ bzw. „kein Licht“ und 255 für „weiß“ bzw. „maximales Licht“ steht. Der Maximalwert für die Lichtintensität in einem Punkt (in diesem Fall 255) wird auch als Bildtiefe bezeichnet. Er gibt an, wie viele verschiedene Grauabstufungen das Bild maximal speichern kann.

3. Verschieben und Rotieren von Bildern

Am zweiten Tag haben wir uns dem Transformieren von Bildern gewidmet. Dabei haben wir uns angesehen, wie Bilder verschoben und gedreht werden können. Wie das

mathematisch gesehen funktioniert und mögliche Implementierungen in der Programmiersprache Python aussehen könnten, wird im Folgenden näher beschrieben.

3.1 Bilder verschieben

Begonnen haben wir mit dem Verschieben von Bildern. Dabei gibt es ein Originalbild („*image_original*“) und ein um x in x-Richtung und y in y-Richtung verschobenes Bild („*image_translated*“). Das neue, verschobene Bild wird aus dem Originalbild errechnet, indem jedem seiner Bildpunkte (an der Position i, j) jeweils „versetzte“ Pixelwerte aus dem Originalbild nach folgendem Prinzip zugeordnet werden:

$$Verschobener\ Bildwert(i,j) = Originalbildwert(i-x, j-y)$$

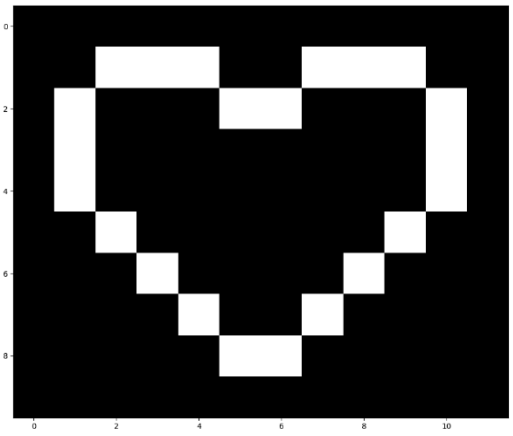
Wichtig hierbei ist zu beachten, dass die Indizes „$i-x$“ und „ $j-y$“ manchmal außerhalb des originalen Bildbereiches liegen können. Da Bildwerte an diesen Stellen nicht definiert sein können, würde die Zuordnung dieser (nicht vorhandenen) Daten zu einer Fehlermeldung führen. Daher werden betreffende Pixel des verschobenen Bildes mit der Farbe Schwarz (Wert Null) gefüllt.

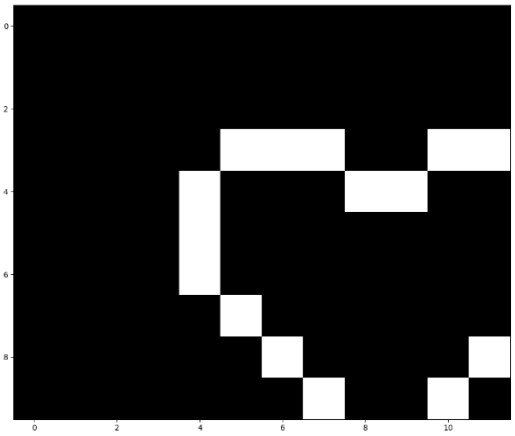
Hier befindet sich eine mögliche Implementierung in der Programmiersprache Python:

```python

# Import der benötigten Module numpy und matplotlib:
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import colormaps

# Definition eines Bildes, welches ein weißes Herz auf schwarzem Hintergrund zeigt:
image_original= [[0 for i in range(12)],
        [0,0,1,1,1,0,0,1,1,1,0,0],
        [0,1,0,0,0,1,1,0,0,0,1,0],
        [0,1,0,0,0,0,0,0,0,0,1,0],
        [0,1,0,0,0,0,0,0,0,0,1,0],
        [0,0,1,0,0,0,0,0,0,1,0,0],
        [0,0,0,1,0,0,0,0,1,0,0,0],
        [0,0,0,0,1,0,0,1,0,0,0,0],
        [0,0,0,0,0,1,1,0,0,0,0,0],
        [0 for i in range(12)]]

# Anzeigen des Bildes:
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.imshow(image_original, cmap=colormaps["gray"])
plt.show()

# Definition einer schwarzen Bildfläche:
image_translated = np.zeros((10,12))

# Eingabe für den Versatz in x- und y-Richtung:
x=int(input("Please enter x-change here >>> "))
y=int(input("Please enter y-change here >>> "))

```

```python

    # Berechnung des neuen, verschobenen Bildes:
    for i in range(10):
        for j in range(12):
            if i < x:
                image_translated[i, j] = 0
            elif j < y:
                image_translated[i, j] = 0
            elif i-x < 0:
                image_translated[i, j] = 0
            elif j-y < 0:
                image_translated[i, j] = 0
            else:
                image_translated[i, j] = image_original[i - x][ j - y]


    # Anzeigen des verschobenen Bildes:
    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ax.imshow(image_translated, cmap=colormaps["gray"])
    plt.show()

```

Als Ausgabe zu diesem Code erhält man bei einer Eingabe von $x = 2$ und $y = 3$ zeitversetzt folgende Bilder:





3.2 Bilder um den Koordinatenursprung drehen

Das Drehen eines Bildes um einen bestimmten Winkel mit dem Koordinatenursprung als Drehzentrum ist etwas komplizierter als das Verschieben in eine festgelegte Richtung. Dabei wird ein Verfahren der linearen Algebra, eine sogenannte Koordinatentransformation, angewandt. Eine lineare Koordinatentransformation erfolgt allgemein gesehen, indem ein ursprünglicher Punkt (Vektor) mit einer Matrix multipliziert wird. Das Ergebnis ist ein neuer, transformierter Punkt. Wichtig zu beachten sind hierbei die Eigenschaften linearer Koordinatentransformationen, dass mehrere Punkte, die im originalen Koordinatensystem auf einer Geraden gelegen sind, auch in transformierter „Form“ auf einer Gerade liegen müssen und dass sich die Koordinaten des Koordinatenursprungs niemals verändern dürfen.

Bei der Drehung eines Bildes um einen bestimmten Winkel α um den Koordinatenursprung können die „Verschiebungen“ der einzelnen Bildpunkte mithilfe folgender Matrix durchgeführt werden:

$$P_{transformiert} = \begin{pmatrix} \cos\alpha & -sin\alpha \\ sin\alpha & cos\alpha \end{pmatrix} \cdot P_{original}$$

Zur Herleitung dieser Formel können folgende Überlegungen in Betracht gezogen werden:

(1) Als erstes ist es, damit es später einfacher wird, nötig, die Komponenten von $P_{original}$ und $P_{transformiert}$ genauer zu definieren:

$$P_{original} = \begin{pmatrix} o_x \\ o_y \end{pmatrix} \qquad P_{transformiert} = \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

(2) $P_{original}$ kann nun in seine „Basisvektoren" zerlegt und neu angeschrieben werden:

$$P_{original} = \begin{pmatrix} o_x \\ o_y \end{pmatrix} = o_x \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + o_y \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

(3) Diese Basisvektoren können, im Vergleich zu dem originalen Vektor relativ einfach um den Koordinatenursprung gedreht werden. Die transformierten Basisvektoren sehen folgendermaßen aus und können durch Überlegungen am Einheitskreis unterstützt werden:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \xrightarrow[um\ \alpha\ drehen]{} \begin{pmatrix} cos\alpha \\ sin\alpha \end{pmatrix} \qquad \begin{pmatrix} 0 \\ 1 \end{pmatrix} \xrightarrow[um\ \alpha\ drehen]{} \begin{pmatrix} -sin\alpha \\ cos\alpha \end{pmatrix}$$

(4) Setzt man diese Vektoren in $P_{transformiert}$ ein, so erhält man:

$$P_{transformiert} = \begin{pmatrix} t_x \\ t_y \end{pmatrix} = o_x \cdot \begin{pmatrix} cos\alpha \\ sin\alpha \end{pmatrix} + o_y \cdot \begin{pmatrix} -sin\alpha \\ cos\alpha \end{pmatrix} = \begin{pmatrix} o_x \cdot cos\alpha - o_y \cdot sin\alpha \\ o_x \cdot sin\alpha - o_y \cdot cos\alpha \end{pmatrix}$$

(5) Nach ein paar weiteren algebraischen Umformungsschritten ergibt sich die Schreibweise der Transformation mittels einer Matrix, die wir zu beweisen versucht haben.

$$P_{transformiert} = \begin{pmatrix} o_x \cdot cos\alpha - o_y \cdot sin\alpha \\ o_x \cdot sin\alpha + o_y \cdot cos\alpha \end{pmatrix}$$

$$P_{transformiert} = \begin{pmatrix} cos\alpha & -sin\alpha \\ sin\alpha & cos\alpha \end{pmatrix} \cdot \begin{pmatrix} o_x \\ o_y \end{pmatrix}$$

$$P_{transformiert} = \begin{pmatrix} cos\alpha & -sin\alpha \\ sin\alpha & cos\alpha \end{pmatrix} \cdot P_{original}$$

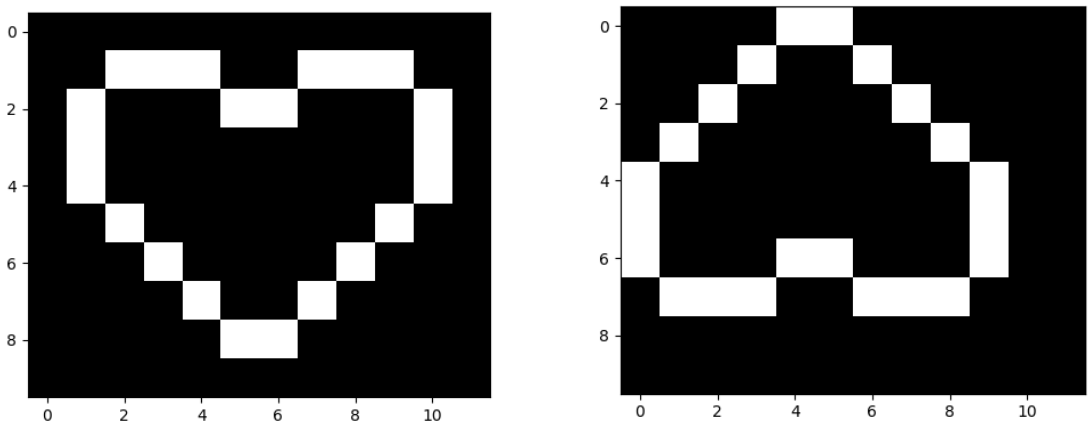
3.3 Bilder um einen beliebigen Punkt drehen

Das Drehen eines Bildes um einen beliebigen Punkt ist eine Kombination der bisher angeführten Techniken. Kurz gesagt wird dabei zunächst das originale Bild verschoben, sodass der „Drehpunkt“ im Koordinatenursprung liegt. Anschließend wird das Bild mit zuvor erwähnter Methode um einen beliebigen Winkel gedreht. Zum Schluss wird das Bild wieder, da keine Verschiebung in Richtung des Koordinatenursprunges gewünscht ist, zurückverschoben.

Eine mögliche Implementierung dieser Drehtechnik sieht folgendermaßen aus:

```python
# Import der benötigten Module:
from matplotlib import pyplot as plt
from matplotlib import colormaps

# Definition der Winkelfunktionen Sinus und Cosinus mittels einer Taylor-Reihe:
def sin(x):  6 usages
    return x-x**3/6+x**5/120-x**7/5040+x**9/362880

def cos(x):  6 usages
    return 1-x**2/2+x**4/24-x**6/720+x**8/40320

# Definition eines Bildes, welches ein weißes Herz auf schwarzem Hintergrund abbildet:
image_original= [[0 for i in range(12)],
        [0,0,1,1,1,0,0,1,1,1,0,0],
        [0,1,0,0,0,1,1,0,0,0,1,0],
        [0,1,0,0,0,0,0,0,0,0,1,0],
        [0,1,0,0,0,0,0,0,0,0,1,0],
        [0,0,1,0,0,0,0,0,0,1,0,0],
        [0,0,0,1,0,0,0,0,1,0,0,0],
        [0,0,0,0,1,0,0,1,0,0,0,0],
        [0,0,0,0,0,1,1,0,0,0,0,0],
        [0 for i in range(12)]]

# Anzeigen des Originalbilds:
fig = plt.figure()
ax1 = fig.add_subplot(2,1,1)
ax1.imshow(image_original, cmap=colormaps["gray"])

# Nutzereingabe für den Drehpunkt und den Drehwinkel:
cx=int(input("Please enter x-coordinate of center here >>> "))
cy=int(input("Please enter y-coordinate of center here >>> "))
angle=float(input("Please enter rotation angle here >>> "))

# Definition des leeren Output-Bildes:
image_rotated=[[0,0,0,0,0,0,0,0,0,0,0,0] for i in range(10)]

# Berechnung des rotierten Bildes:
for i in range(10):
    for j in range(12):
        if round(cos(angle)*(i-cx)+sin(angle)*(j-cy)+cx) < 0:
            image_rotated[i][j] = 0
        elif round(cos(angle)*(i-cx)+sin(angle)*(j-cy)+cx) > 9:
            image_rotated[i][j] = 0
        elif round(-sin(angle)*(i-cx)+cos(angle)*(j-cy)+cy) < 0:
            image_rotated[i][j] = 0
        elif round(-sin(angle)*(i-cx)+cos(angle)*(j-cy)+cy) > 11:
            image_rotated[i][j] = 0
        else:
            image_rotated[i][j]=image_original[round(cos(angle)*(i-cx)+sin(angle)*(j-cy)+cx)][round(-sin(angle)*(i-cx)+cos(angle)*(j-cy)+cy)]

# Anzeigen des rotierten Bildes:
ax2 = fig.add_subplot(2,1,2)
ax2.imshow(image_rotated, cmap=colormaps["gray"])
plt.show()
```

Bei der Ausführung dieses Codes und der Eingabe der Werte 4, 5 und 3.1415926… ($\approx \pi$, also eine Drehung um 180°) werden diese Bilder ausgegeben:



4. Homografie-Matrix

Am dritten Tag haben wir uns mit einer Generalisierungsmethode für Transformationsmatritzen beschäftigt – der Homographie-Matrix. Außerdem haben wir mittels eines Python-Moduls namens OpenCV markante Stellen in Bildern (Key Points) gesucht, diese mit anderen Bildern verglichen und Zusammenhänge hergestellt.

4.1 Homografie-Matrix

Eine Homografiematrix ($\boldsymbol{H}$) ist eine 3x3-Matrix, welche Informationen über die gewünschte Drehung, Verschiebung und Verzerrung enthält. x und y sind dabei die Koordinaten eines Bildpunktes im Originalbild, x' und y' die zugehörigen Koordinaten im transformierten Bild.

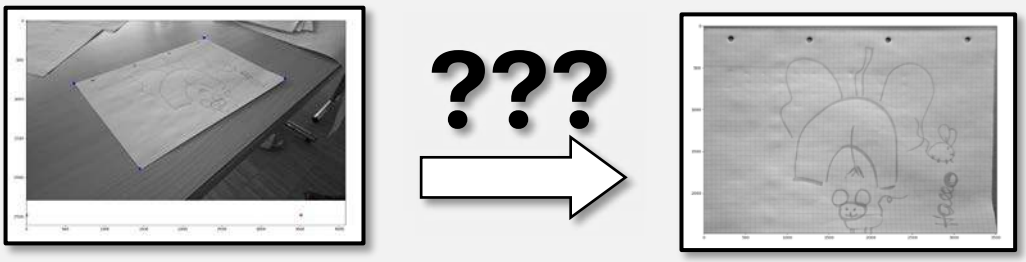
$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} \boldsymbol{cos\alpha} & \boldsymbol{-sin\alpha} & \boldsymbol{t_x} \\ \boldsymbol{sin\alpha} & \boldsymbol{cos\alpha} & \boldsymbol{t_y} \\ \boldsymbol{p} & \boldsymbol{q} & \boldsymbol{1} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad mit\ x' = \frac{a}{c}\ und\ y' = \frac{b}{c}$$

Die Homografie-Matrix funktioniert, indem sie jedem Punkt des zu transformierenden Bild anfangs eine zusätzliche Koordinate ($z = 1$) verleiht und eine 3x3-Matrix auf diesen neuen dreidimensionalen Vektor anwendet (lineare Transformation in 3 Dimensionen). Anschließend wird der entstandene Punkt so umgerechnet, sodass er wieder nur zwei Dimensionen hat bzw. auf der Ebene $z = 0$ liegt. Durch diesen Zwischenschritt in einer „höheren Dimension“ hat eine Transformation mit einer Homografie-Matrix 8 statt nur 4 veränderbare Parameter (Degrees of Freedom) und ist daher wesentlich flexibler.

Die Bereiche mit den Sinus- und Cosinuswerten von α bewirken genau wie bei der „Drehungsmatrix“ eine Drehung um den Koordinatenursprung um den Winkel α. Die beiden Parameter t_x und t_y geben den Versatz in x- bzw. y-Richtung an, p und q sind für eine perspektivische Verzerrung zuständig.

4.2 Ermittlung einer Homografie-Matrix für eine beliebige Transformation vier verschiedener Punkte

Nehmen wir an, dass ein schräges Bild von einem rechteckigen Blatt Papier aufgenommen worden ist. Dieses soll nun so transformiert werden, sodass das transformierte Bild aussieht, als wäre es genau von oben abfotografiert worden. Wie könnte man dieses Problem lösen?



Zunächst wird festgelegt, dass das Originalbild mithilfe einer Homografie-Matrix transformiert werden soll. Diese soll, wie das für ein solches Objekt üblich ist, 8 Einträge besitzen. Welche Zahlenwerte diesen Einträgen zugewiesen werden, ist zu Beginn jedoch gänzlich unbekannt. Sobald man alle Einträge in der Homografie-Matrix kennt kann man einfach das Bild transformieren. Jetzt stellt sich eine neue Frage, wie man diese Werte berechnen kann; Und das geht mithilfe eines Gleichungssystems!

Da die Homografie-Matrix 8 unbekannte Einträge besitzt, benötigt man zur Bestimmung der Matrix ein Gleichungssystem mit 8 Gleichungen. Diese liefern die 4 jeweils transformierten Punkte, welche aus je 2 Komponenten bestehen.

$$4\ Punkte \cdot 2\ \frac{Informationen}{Punkt} = 8\ Informationen \rightarrow 8\ Gleichungen$$

Es lassen sich folgende Gleichungen aufstellen:

$$x_i^{'} = \frac{x_i h_{0,0} + y_i h_{0,1} + h_{0,2}}{x_i h_{2,0} + y_i h_{2,1} + 1} \rightarrow x_i^{'} = x_i h_{0,0} + y_i h_{0,1} + h_{0,2} - x^{'} x h_{2,0} - x^{'} y h_{2,1}$$

$$y_i^{'} = \frac{x_i h_{1,0} + y_i h_{1,1} + h_{1,2}}{x_i h_{2,0} + y_i h_{2,1} + 1} \rightarrow y_i^{'} = x_i h_{1,0} + y_i h_{1,1} + h_{1,2} - y^{'} x h_{2,0} - y^{'} y h_{2,1}$$

$$mit\ i\ \in \{1,\ 2,\ 3,\ 4\},\ P_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}, P^{'}{}_i = \begin{pmatrix} x^{'}{}_i \\ y^{'}{}_i \end{pmatrix} und\ H = \begin{pmatrix} h_{0,0} & h_{0,1} & h_{0,2} \\ h_{1,0} & h_{1,1} & h_{1,2} \\ h_{2,0} & h_{2,1} & 1 \end{pmatrix}$$

Weil es etwas umständlich wäre, dieses aus 8 Gleichungen bestehendes Gleichungessystem händisch zu lösen, bietet es sich an, das Gleichungssystem in Matrixschreibweise anzuschreiben und mittels eines Python-LinAlg-Lösebefehls zu lösen:

$$\begin{pmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & x_1x_1' & y_1x_1' \\ 0 & 0 & 0 & x_1 & y_1 & 1 & x_1y_1' & y_1y_1' \\ x_2 & y_2 & 1 & 0 & 0 & 0 & x_2x_2' & y_2x_2' \\ 0 & 0 & 0 & x_2 & y_2 & 1 & x_2y_2' & y_2y_2' \\ x_3 & y_3 & 1 & 0 & 0 & 0 & x_3x_3' & y_3x_3' \\ 0 & 0 & 0 & x_3 & y_3 & 1 & x_3y_3' & y_3y_3' \\ x_4 & y_4 & 1 & 0 & 0 & 0 & x_4x_4' & y_4x_4' \\ 0 & 0 & 0 & x_4 & y_4 & 1 & x_4y_4' & y_4y_4' \end{pmatrix} \cdot \begin{pmatrix} h_{0,0} \\ h_{0,1} \\ h_{0,2} \\ h_{1,0} \\ h_{1,1} \\ h_{1,2} \\ h_{2,0} \\ h_{2,1} \end{pmatrix} = \begin{pmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{pmatrix}$$

Die Lösung dieses Gleichungssystems (also der Vektor mit den ganzen „*h*“s) enthält alle Elemente der Homografie-Matrix (*H*). Nach dem Lösen des Gleichungssystems ist nun die gesamte Homografie-Matrix bekannt, wodurch die ursprünglich geforderte Transformation des Bildes ermöglicht wird und durchgeführt werden kann. Folgender Code führt genau diese Theorie aus:

```python
# Importieren der nötigen Module:
import cv2
from matplotlib import pyplot as plt
from matplotlib import colormaps
import numpy as np

# Laden des Bildes:
file_path="C:/Users/Maximilian/PycharmProjects/Modellierungswoche_2025/code/adobe_scan_fake/20250210_124916.jpg"
image = cv2.imread(file_path,cv2.IMREAD_GRAYSCALE)

# Felstlegung der Koordinaten der Eckpunkte vor und nach der Transformation:
corners_old = np.array([[1448,1885],[608,799],[2270,219],[3306,736]])       # x, y
corners_new = np.array([[0,2480],[0,0],[3508,0],[3508,2480]])               # x', y'

# Anzeigen des Originalbildes mit gekennzeichneten Punkten:
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.imshow(image, cmap=colormaps["gray"])
for x,y in corners_old:
    plt.plot( *args: x,y,"bo")
for x,y in corners_new:
    plt.plot( *args: x,y,"ro")

# Erstellung einer leeren Homografie-Matrix:
H = np.zeros((3,3))

# Definition des am Gleichungssystem-Lösen beteiligten Vektors:
results = corners_new.flatten()

# Definition der am Gleichungssystem-Lösen beteiligten Matrix:
matrix = np.zeros((8,8))

```

```python

for i in range(4):
    tr = 2*i        # top row
    br = 2*i+1      # bottom row
    matrix[tr, 0] = corners_old[i][0]
    matrix[tr, 1] = corners_old[i][1]
    matrix[tr, 2] = 1
    matrix[tr, 3] = 0
    matrix[tr, 4] = 0
    matrix[tr, 5] = 0
    matrix[tr, 6] = -corners_new[i][0] * corners_old[i][0]
    matrix[tr, 7] = -corners_new[i][0] * corners_old[i][1]

    matrix[br, 0] = 0
    matrix[br, 1] = 0
    matrix[br, 2] = 0
    matrix[br, 3] = corners_old[i][0]
    matrix[br, 4] = corners_old[i][1]
    matrix[br, 5] = 1
    matrix[br, 6] = -corners_new[i][1] * corners_old[i][0]
    matrix[br, 7] = -corners_new[i][1] * corners_old[i][1]

# Lösen des linearen Gleichungssystem mittels eines Lösungsnefehls:
solution = np.linalg.solve(matrix,results)

# Zuweisung der Lösungen zu der Homografie-Matrix:
H[0,0]=solution[0]
H[0,1]=solution[1]
H[0,2]=solution[2]
H[1,0]=solution[3]
H[1,1]=solution[4]
H[1,2]=solution[5]
H[2,0]=solution[6]
H[2,1]=solution[7]
H[2,2]=1

# Transformieren des Bildes:
image_transformed = cv2.warpPerspective(image,H, dsize: (3508,2480))

# Ausgabe des transformierten Bildes in einem neuen Fenster:
fig2 =plt.figure()
ax2 = fig2.add_subplot(1,1,1)
ax2.imshow(image_transformed, cmap=colormaps["gray"])
plt.show()

```

5. Keypoint Detection

Die Keypoint-Detection ist eine beliebte Möglichkeit, Bilder nachträglich zusammenzusetzen. Keypoints sind bestimmte Features oder Punkte mit signifikanten Eigenschaften in Bildern, denen Koordinaten zugeordnet werden. Sind diese Punkte in beiden Bildern vorhanden, können die Bilder zusammengefügt werden. Diese Eigenschaften können Ecken, Kanten oder Punkte mit auffälligen Binärcodes an Farbe sein.

Um dies auszuführen, gibt es auf OpenCV, einem Python-Modul, mehrere verfügbare Keypoint-Extraktionsmethoden mit unterschiedlichen Vor- und Nachteilen. Eine der beliebtesten Methoden ist “SIFT” (Scale-Invariant Feature Transformation). Sie sucht nach lokalen Extrema in einer 3x3x3-Umgebung und arbeitet mit der “Difference of Gaussian”.

Eine weitere Option ist “SURF” (Speeded-Up Robust Feature), es ist jedoch patentiert und nicht auf OpenCV verfügbar.

“ORB” (Oriented FAST and BRIEF) ist eine Alternative zu SIFT, die ebenfalls auf OpenCV verfügbar ist. FAST steht heirbei für “Features from Accelerated Segment Test” und arbeitet mit Farbhelligkeiten. BRIEF, also “Binary Robust Independent Elemtary Features” wird als Keypoint-Descriptor verwendet.

ORB ist in der Lage, die meisten KeyPoints zu finden, ist schneller in der Berechnung und am robustesten für Beleuchtungs- und Rotationsänderung. In einigen Situationen sind jedoch trotzdem andere Keypoint-Extraktionsmethoden vorteilhaft, da alle unterschiedlich arbeiten und somit zu anderen Ergebnissen führen.

6. Image Stitching

Unter Image Stitching versteht man das perspektivische Zusammenfügen von zwei oder mehr Bildern, die, ausgehend von der selben Position in unterschiedliche Richtungen aufgenommen worden sind. Um Bilder erfolgreich stitchen zu können, sind einige Schritte, wie das Finden von gemeinsamen Merkmalen, die Ermittlung von Homografie-Matrizen und das Transformieren von Bildern notwendig. Unsere Erfahrungen im Umgang mit diesen und weiteren Techniken werden im Folgenden beschrieben.

6.1 Image Stitching von 2 Bildern

Image Stitching ist eine Methode, die durch Code Bilder zusammenfügt. Die Funktionen des Image Stitching sind ähnlich zu Standard Panorama Fotos. Um erfolgreich Bilder zu stitchen benötigt man ein Programm, das uns Code schreiben lässt. In diesem Fall haben wir Python benützt, welches wir mit Python Charm ausgeführt haben. In dem Programm selbst werden noch einige Ad-Ons benützt, die mit den mathematischen Gleichungen helfen. Um einen Stitch zu starten, sucht die Software erst die gewünschten Fotos heraus. Diesen Vorgang wiederholt man für jedes Bild, das man stitchen möchte. Wenn dieser Vorgang abgeschlossen ist, berechnet das Programm als erstes durch neuen Code die Größe der vorhandenen Bilder. Wenn diese Berechnungen fertig sind, fährt der Code fort und beginnt die Berechnung der sogenannten Keypoints. Hierfür benutzt man Keypointextraktionsmethoden wie ORB, SIFT oder SURF (mehr dazu im Teil Keypoints). Nach dem Ausrechnen der Keypoints werden die Bilder anhand eines so genannten “BruteForce-Matcher” aneinandergereiht und die jeweilig gleichen Keypoints werden gematched (BF-Matcher = BruteForce-Matcher, von uns auch BoyFriendmatcher gennant 😉). Zum Matchen werden von den berechneten Keypointpaaren zwischen den 2 Bildern durch eine neue Zeile Code die von der Distanz kürzesten benützt, um einen Informationsüberschuss und Ungenauigkeiten zu vermeiden.

Auch haben wir mit einem Learning Unified Imagination Stitching Algorythm (kurz. LUISA) herumgespielt, die uns das alles in Goodnotes zusammengeschoben hat und eine sehr ansehnliche Version erstellt hat.

Danach werden die Keypoint-Arrays mithilfe einer Homographie-Matrix zu einem Bild zurechtgeschoben. Das Programm transformiert mit Hilfe einer Translations-Matrix die Bilder, die sich in einer BorderBox befinden, in den positiven Bereich des Sichtspektrums des Fensters. Ohne diesen Schritt würden sich alle Werte im negativen Bereich des

Koordinatensystems befinden. Zuletzt verwendet man einen Command, um sich die fertigen Fotos anzeigen mithilfe von “Matplotlip” anzeigen zu lassen. Um diesen Vorgang mit 3 Bildern zu wiederholen, benötigt man am Ende des Codes einen Command, welcher besagt, dass das gleiche mit den anderen zwei Variablen durchgeführt werden soll und im Nachhinein die beiden gestitchen Bilder zu einem finalen Foto gestitched werden.

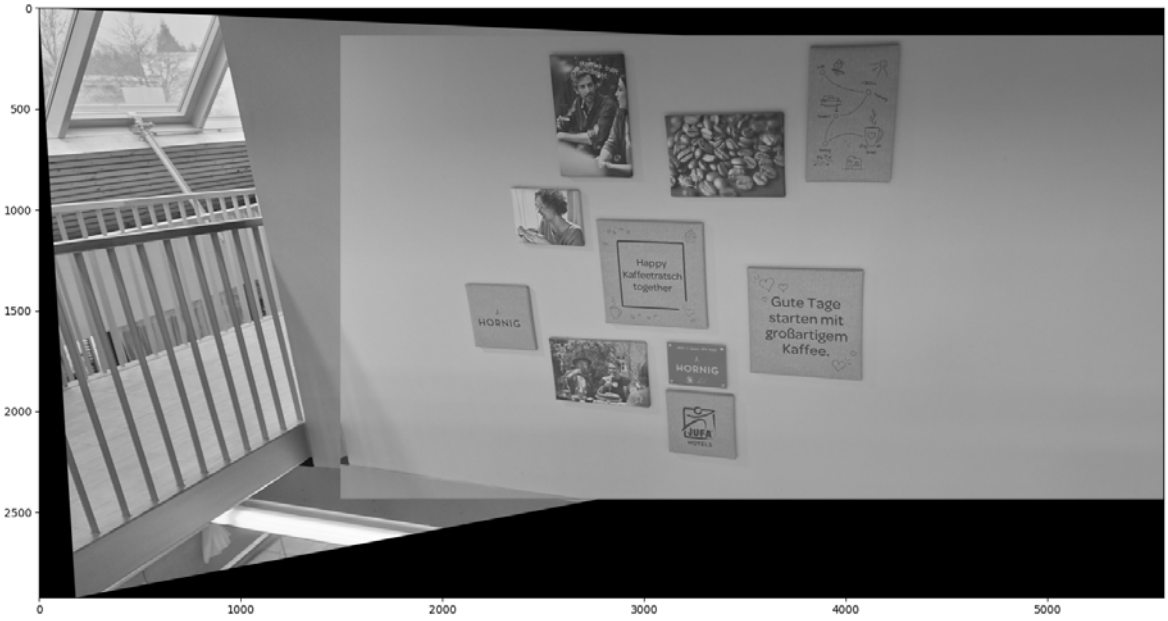
6.2 Image Stitching von 3 oder mehr Bildern

Das Image-Stitching von drei oder mehr Bildern ist sehr ähnlich zu dem von zwei Bildern. Zuerst lädt man die benötigten Python-Module. Danach stellt man eine Funktion auf, welche den ganzen Code einerseits in sich verstecken kann, um den Code strukturierter und organisierter wirken zu lassen. Andererseits sorgt sie dafür, dass der Code nur einmal geschrieben werden muss, weil man die Bilder so nur in die Definition einfügen muss. Danach definiert ein Befehl die Höhe und Breite des Bildes. Um die Features zu ermitteln, verwendet man eine Keypoint-Extraktionsmethode, wie zum Beispiel SIFT oder ORB. Die gefundenen Features sind bestimmte Punkte in Bildern, die besondere Merkmale aufweisen, wie beispielsweise Kanten, Sprünge in den Grauwerten oder Ecken. Wenn diese gefunden wurden, werden sie mit Koordinaten versehen und sind danach Keypoints. Danach wird der BF-Matcher (Brute-Force-Matcher) benötigt. Dieser vernetzt und vergleicht die gefundenen Keypoints der verschiedenen Bilder miteinander und verbindet jene, die zusammenpassen. Im Anschluss werden diese sortiert und ein festgelegter Anteil wird verwendet. Dann werden mit der Homografie-Matrix die Eckpunkte in einer Liste definiert. Danach werden die Bilder einander angepasst und die Eckpunkte miteinander verkettet. Das zusammengestitchte Bild wird in das Sichtfeld geschoben und die Translationsmatrix passt die Größe des finalen Bildes an. Im Anschluss werden die Höhe und Breite des Endbildes bestimmt und die Informationen zu einem Bild zusammengefügt. Durch den „return“-Befehl gibt die Funktion gewünschte Werte aus. Danach werden die verwendeten Fotos eingefügt, wobei man beliebig viele Fotos benutzen kann, weil der Code eine Schleife verwendet, die den Vorgang beliebig oft wiederholen kann. Dann wird bestimmt, welche Bilder in der Funktion verwendet werden sollen und was ausgegeben werden soll. Zum Schluss wird die Anzahl der Fenster und deren Inhalt festgelegt.

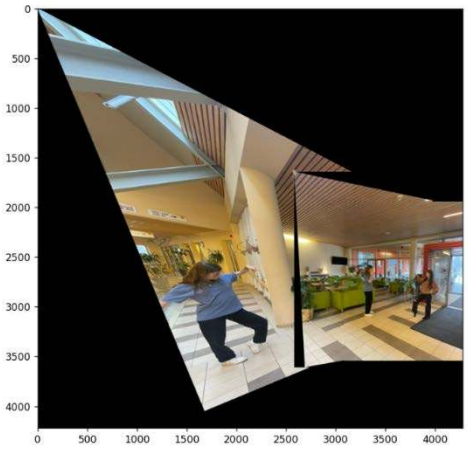
6.3 Bild-Masken

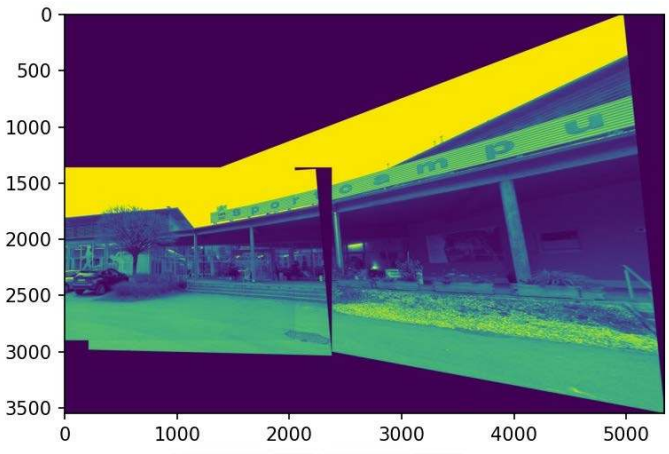
Um die transformierten Bilder letztendlich miteinander zu kombinieren, gibt es einige Möglichkeiten. Der simpelste Ansatz wäre jedes Bild und dessen Werte auf die leere Leinwand zu addieren. Dies sieht folgendermaßen aus:

Bereiche mit der größten Überlappung stechen hervor und man kann mehr oder weniger klar erkennen, aus wie vielen Bildern sich das finale Bild zusammensetzt. Dazu sind Übergänge noch klar erkennbar. Diese Methode ist aufgrunddessen auch sehr gut um unser Programm zu testen, eignet sich jedoch weniger um schöne Ergebnisse zu erzielen. Dazu haben wir zuerst das Ausgangsbild genommen und dann leere Pixel auf der Leinwand mit den Pixeln des transformierten Bildes ausgefüllt. So haben wir sozusagen unser Ausgangsbild auf das transformierte Bild gelegt. Für das Stiching von 2 Bildern liefert diese Methode auch relativ gute Ergebnisse:

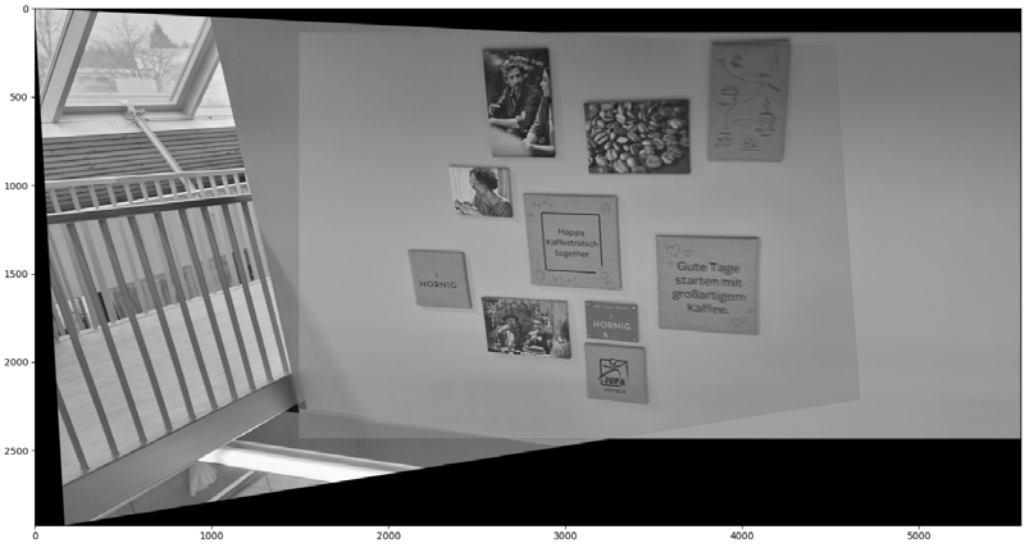


Probleme kommen hier zustande wenn wir versuchen noch ein drittes Bild an unsere bereits zusammengesetzten Bilder anzufügen, denn wenn unser Ausgangsbild selbst schwarze Flächen enthält, brauchen wir zusätzlich Methoden, um diese zu entfernen bzw. zu ignorieren, ansonsten ergeben sich je nach Anordnung solche Bilder:

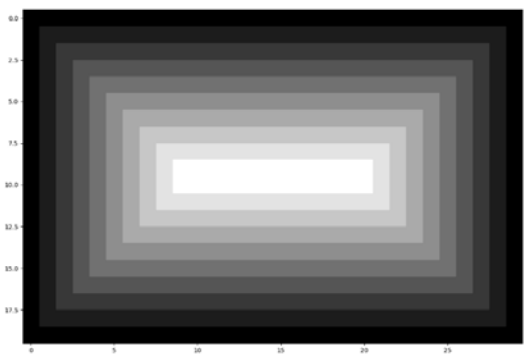




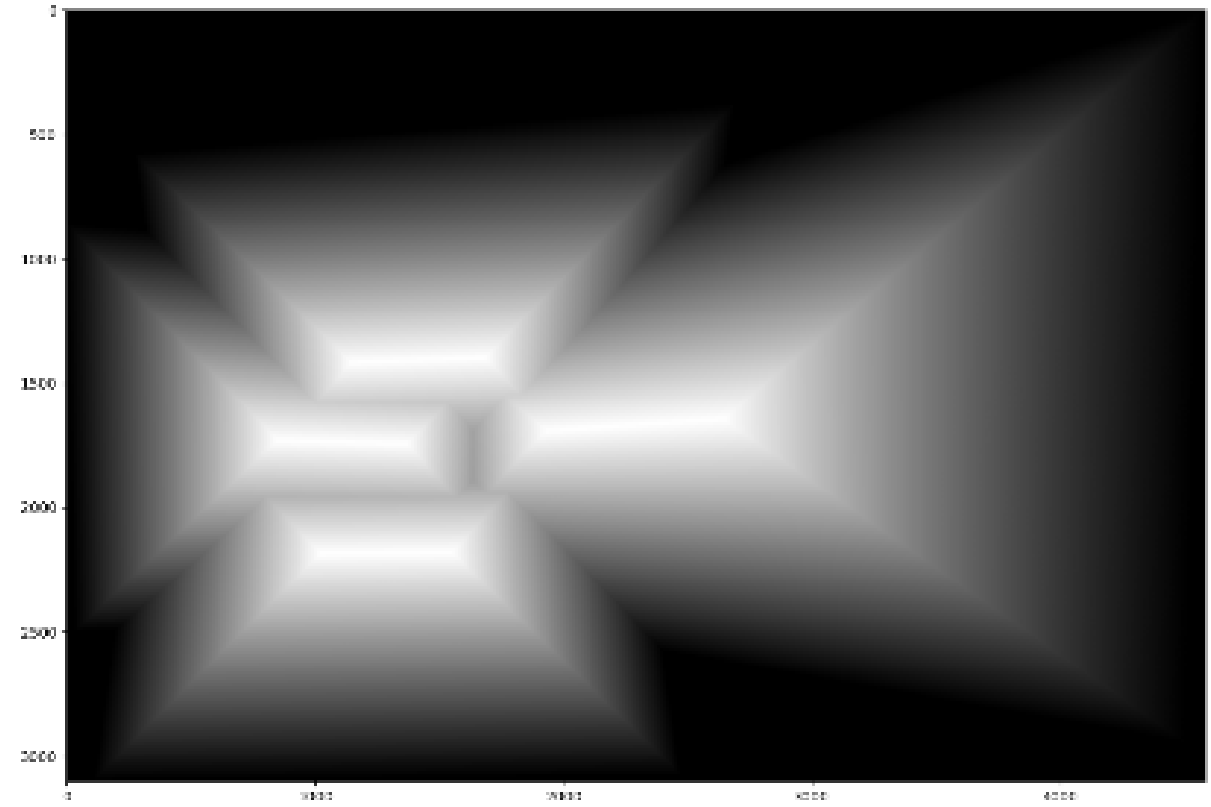
Außerdem sind hier Übergänge zwischen Bildern noch relativ deutlich erkennbar. Aufgrund von Vignetteneffekten der Kamera oder Belichtungsunterschieden bei verschiedenen Kameraausrichtungen sind diese auch schwer zu vermeiden, den Durchschnitt der sich überlappenden Pixelwerte zu bilden macht sie auch nicht unerkenntlicher, liefert aber Bilder wie folgendes, die auch relativ praktisch sind um Ungenauigkeiten zu erkennen:



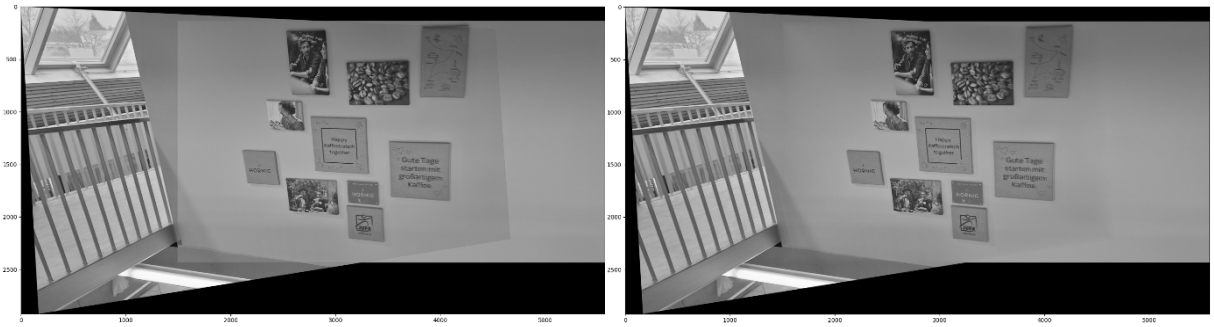
Unser Lösungsansatz war es Bilder so zusammenzufügen, dass Distanz vom Zentrum des eigenen Ausgangsbildes bestimmt, wie sehr der Pixelwert eines Bildes Einfluss auf das Finale Bild nimmt. Das bewerkstelligen wir mithilfe von Gewichtsmasken, die wir, wie im Bild rechts dargestellt, generieren.



Hier ordnen wir quasi Pixeln aufsteigende Werte zu je näher sie sich dem Zentrum des Bildes befinden und dividieren diese dann durch den Maximalwert, um normiert Werte zwischen 0 und 1 zu erhalten. Vor der Anwendung der Homografie-Matrix auf unsere Bilder erstellen wir zu jedem Bild eine zugehörige Maske mit den selben Dimensionen, die wir ebenfalls transformieren. Einige solche übereinandergelegt sehen wiefolgt aus:



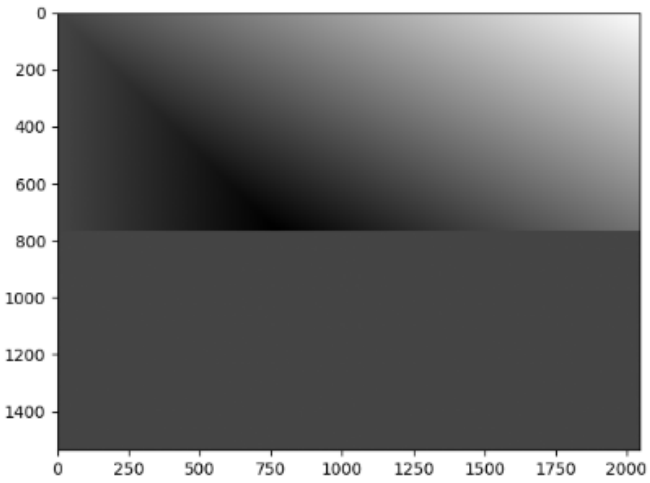
Nun bestimmen wir den finalen Pixelwert indem wir Einzelwerte mit ihren zugehörigen Gewichten multiplizieren und durch die Summe der Gewichte dividieren. Hier ein Vergleich zwischen durchschnittlichen Pixelwert und unserem Blendingfilter:

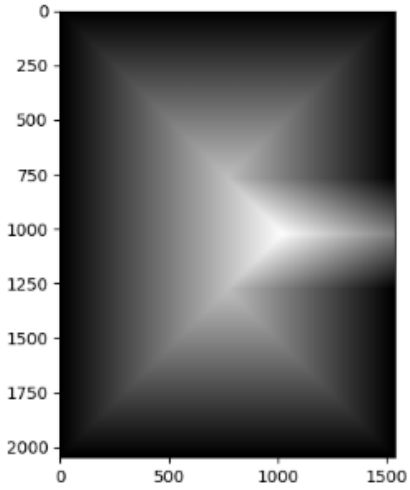


Um weitere Bilder an unser bereits gestichtes Bild anzufügen können wir auch alle verwendeten Masken anhand ihrer Maximalwerte zusammenlegen und gemeinsam mit dem Endbild zurückgeben. Sollte nun die Summe aller Maskengewichte 0 ergeben haben wir eine leere Stelle erkannt und lassen den Pixel leer um nicht durch 0 zu dividieren. So ergeben sich auch keine Überlagerungsartefakte mit schwarzen Rändern.

7. Lustige Fehlkreationen

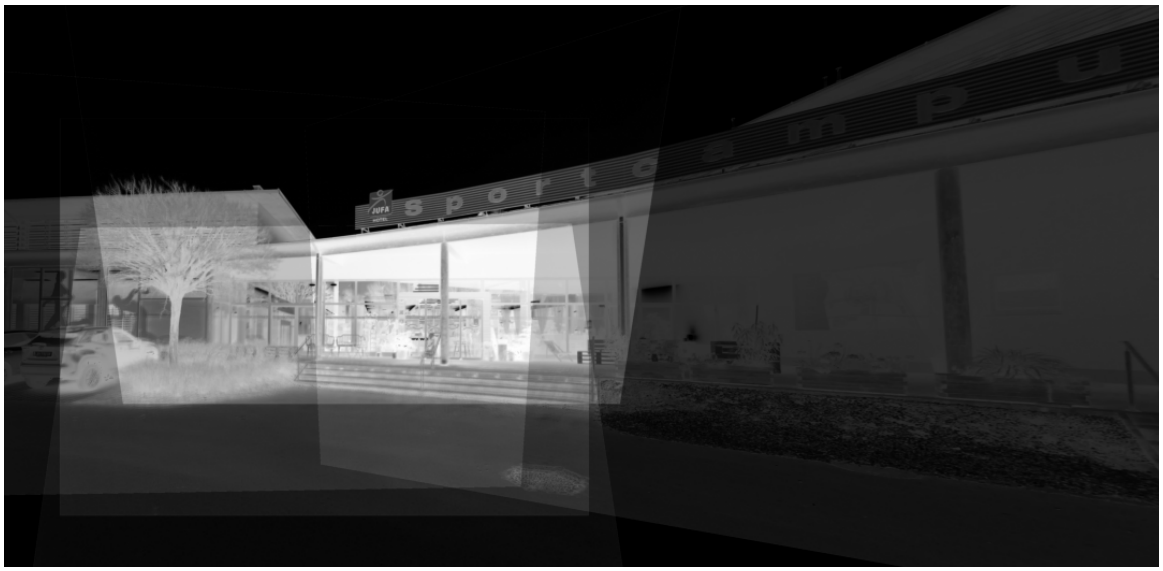
Im Anschluss befinden sich einige Bilder, die dank ihrer fehlerhaften Erstellung interessante Formen oder kuriose Verzerrungen besitzen. Die ersten Bilder stellen einige Fehlversuche dar, die bei der anfänglichen Generierung von Blending-Filtern entstanden sind:





Ein weiteres besonderes Kunstwerk ist bei der Zusammenfügung von 5 Bilden aus dem Treppenhaus im Jufa-Hotel entstanden entstanden:

Auch von außen bekommt das Jufa-Hotel mit manchen Codes einen interessanten Anstrich:



Fazit

Als Fazit lässt sich für uns festhalten, dass die Modellierungswoche 2025 ein großartiges Ereignis war. Vom Coden in der Programmiersprache Python über Keypoint-Detection bis zur linearen Algebra haben wir alle sehr viel gelernt. Es war allerdings nicht nur von spannender Mathematik und teilweise funktionierenden Codes, sondern auch von lustigen gemeinsamen Spieleabenden und und dem Zusammentreffen unterschiedlicher Musikgeschmäcker geprägt. Uns hat die Modellierungswoche 2025 sehr gut gefallen und wir freuen uns schon auf sehr auf das nächste Mal!

„THAT is

EXCITEMENT!”

(Mag. DDr. Patrick-Michel Frühmann, Leibnitz, 2025)