Seminararbeit

Julia zur Verarbeitung und Visualisierung von Daten

Autor: Theodor Schwarzrock

Tutorium: 12/7

Lehrer: Herr Möller

Seminarkurs Informatik

Friedrich-Schiller-Gymnasium

18.10.2024

**Inhaltsverzeichnis**

[1 Einleitung 1](#_Toc179822323)

[2 Die Programmiersprache Julia 2](#_Toc179822324)

[2.1 Entstehung und Geschichte 2](#_Toc179822325)

[2.2 Anwendungen 3](#_Toc179822326)

[2.3 Konkurrenten 3](#_Toc179822327)

[3 Vergleich mit Python 4](#_Toc179822328)

[3.1 Syntax und Semantik 4](#_Toc179822329)

[*3.1.1 Dynamische Typisierung Julias 4*](#_Toc179822330)

[*3.1.2 Variablen 4*](#_Toc179822331)

[*3.1.3 Booleans und Vergleichsoperationen 5*](#_Toc179822332)

[*3.1.4 Konditionen 5*](#_Toc179822333)

[*3.1.5 Schleifen 5*](#_Toc179822334)

[*3.1.6 Funktionen 6*](#_Toc179822335)

[*3.1.7 Auswertung der Syntax und Semantik 7*](#_Toc179822336)

[3.2 Leistung 7](#_Toc179822337)

[3.3 Community und Unterstützung 7](#_Toc179822338)

[3.4 Lernkurve und Lesbarkeit 7](#_Toc179822339)

[3.5 Auswertung 7](#_Toc179822340)

[4 Praktische Anwendung von Julia 7](#_Toc179822341)

[4.1 Erläuterung des praktischen Anteils 7](#_Toc179822342)

[*4.1.1 Begriff: Platonische Körper 7*](#_Toc179822343)

[*4.1.2 Ziel des Programms 8*](#_Toc179822344)

[*4.1.3 Thematischer Bezug 8*](#_Toc179822345)

[4.2 Umsetzung 8](#_Toc179822346)

[5 Fazit 8](#_Toc179822347)

[6 Literaturverzeichnis 8](#_Toc179822348)

[7 Anhang 9](#_Toc179822349)

# 1 Einleitung

*„Visualization gives you answers to the questions you didn’t know you had.”[[1]](#footnote-1)  
- Ben Schneiderman*

Unser heutiges Zeitalter ist geprägt von Informationen und Daten. In fast jeder alltäglichen Situation, welche sich mit Technologie werden unzählbar viele Daten ausgetauscht, ausgewertet und wiederum ausgetauscht. Diese Einsen und Nullen sind jedoch nicht für den Menschen verständlich, selbst wenn sie in lesbare Rohdaten umgewandelt werden. Um eine Intuition zu diesen Datenmengen zu erschaffen, müssen sie anschaulich gemacht werden. Darum kümmert sich die *Data-Science*.

Es wurden schon etliche Programmiersprachen entwickelt, die sich dieser Aufgabe widmen. Unter anderem *Python*, *C++*, *MATLAB*, *R* oder *FOTRAN*. Die genannten Sprachen sind Giganten in der Data-Science. Vor allem Python etablierte sich im Laufe der Zeit als die Beliebteste der vielen Optionen. Vor etwa 12 Jahren wurde ein neuer Konkurrent veröffentlicht. Die Programmiersprache *Julia*. Sie wurde erschaffen, um die besten Eigenschaften ihrer Vorgänger ein einer Sprache zu vereinen.

Das Ziel dieser Seminararbeit ist es einen umfassenden Vergleich zwischen Julia und Python zu erschaffen, wodurch schließlich auf Julias Relevanz und Daseinsberechtigung geschlossen wird. In dem Vergleich steht die Data-Science und darin vor allem die Visualisierung von Daten im Vorschein. So werden die beiden Programmiersprachen nicht nur auf Syntax, Semantik und weitere technische Details untersucht, sondern auch anhand eines praktischen Teils veranschaulicht. Der Fokus liegt währenddessen jedoch durchgehend auf Julia. In dieser Seminararbeit soll Python *nicht* bis auf das letzte Detail analysiert werden.

Die Relevanz der Forschung liegt im Innovationspotenzial von Julia. Für Entwickler und Datenwissenschaftler, welche stets nach effizienteren Werkzeugen suchen, ist ein Vergleich der vielen Möglichkeiten von großer Bedeutung. So gibt es zudem praktische Anwendungsbereiche dieser Untersuchung.

# 2 Die Programmiersprache Julia

## 2.1 Entstehung und Geschichte

Jeff Bezanson, Alan Edelman, Viral B. Shah und Stefan Karpinski hatten in den 2000ern ihr Informatikstudium beendet und verfolgten Karrieren in der Informatik. Sie waren in Sprachen wie *MATLAB*, *Python*, *Ruby*, *R* und *C* erfahren und sich deren jeweiligen Stärken und Schwächen bewusst. Jedoch waren die studierten Informatiker nicht zufrieden. Die verschiedenen Programmiersprachen deckten zwar gegenseitig ihre Schwächen ab, aber es gab nie eine Sprache, welche alle Vorteile mit möglichst wenig Nachteilen gleich vereinte. Die Gruppe tat sich somit mit dem Ziel zusammen, eine Sprache zu kreieren, die Leistung, Simplizität, Effizienz und Geschwindigkeit bietet.[[2]](#footnote-2)

Im Jahr 2009 begann schließlich die Entwicklung von Julia. Der Name hat keine Bedeutung. Bezanson wollte lediglich einen kurzen, prägnanten Namen. „Julia“ gefiel dem Team schlicht.[[3]](#footnote-3) Den Entwicklern war es wichtig, dass Julia Open Source ist und unter einer liberalen Lizenz veröffentlicht wird. Auf Benutzer-Ebene sollte Julia einfach zu erlernen sein, aber auch erfahrene Programmierer zufriedenstellen. Zudem war Interaktivität trotz Kompilierung ein sehr wichtiger Anhaltspunkt. Auf technischer Ebene sollte Julia so schnell sein wie *C*, aber dynamisch wie *Ruby*, eine mathematische Notation besitzen, ähnlich wie *MATLAB*, für die allgemeine Programmierung nutzbar sein, wie *Python* aber gleichzeitig auch für das wissenschaftliche Programmieren geeignet sein.[[4]](#footnote-4)

Das Team strebte außerdem grundlegende Funktionen an. Einfache Skalarschleifen sollen in Maschinencode kompiliert werden, um Effizienz zu steigern. Zudem soll eine Skalierung von Einzelprozessor-Berechnungen auf verteile Berechnungen über viele Maschinen hinweg möglich sein. So werden die Rechenleistung und Kapazität erheblich gesteigert. Um die gewollte Nutzerfreundlichkeit und Dynamik zu erreichen wurde großer Wert auf Typenflexibilität gelegt. Sie werden also nur dann spezifiziert, wenn es wirklich notwendig ist. „We are greedy: we want more.“ [[5]](#footnote-5) schrieb das Team passend in einem Artikel über die Schaffung von Julia.

Am 14. Februar 2012 wurde die erste Version von Julia schließlich offiziell veröffentlicht. Daraufhin durchging die Programmiersprache mehrere pre-1.0 Versionen, bis sechs Jahre später die Veröffentlichung von Julia 1.0 folgte. Damit war die erste stabile Version öffentlich.[[6]](#footnote-6)

## 2.2 Anwendungen

Der grundlegende Aufbau Julias verspricht ihr von Natur aus einen Platz in der Data-Science. Vor allem für Big-Data-Anwendungen und maschinelles Lernen eignet sich Julia aufgrund ihrer hohen Geschwindigkeit. Ähnlich werden ihre Fertigkeiten auch im Finanzwesen angewendet, um Märkte zu modellieren, Risiken zu bewerten oder komplexe Finanzinstrumente zu berechnen. Dabei wird wieder auf die Geschwindigkeit, aber besonders auch auf die Genauigkeit Julias Wert gelegt.

Durch ihre effiziente Ausführung mathematischer Operationen, ist Julia auch in der naturwissenschaftlichen Programmierung vielfältig angewandt. Da Julia im numerischen Rechnen exzelliert[[7]](#footnote-7), greifen Wissenschaftler zu ihr, um umfangreiche Simulationen zu berechnen und auch gleich zu visualisieren.

So strecken sich die hier möglichen Anwendungsbereiche von der Physik, über die Biowissenschaften bis hin zur künstlichen Intelligenz und wissenschaftlichem maschinellen Lernen, wo Julia ihre aktuelle und momentan relevanteste Applikation findet. Neue Untersuchungen zeigen, dass die Sprache exzellent darin ist, gewöhnliche Differentialgleichungen zu lösen, was für maschinelles Lernen und für die vorher erwähnten Simulationen ein großer Vorteil ist.[[8]](#footnote-8)

Mittlerweile wird Julia von den größten Unternehmen der Welt für KI-Forschung bis hin zum Finanzwesen verwendet. Dazu gehören *Amazon*, *Meta*, *Google*, *Nvidia*, *Microsoft*, *AMD*, *J.P. Morgan* oder auch *BlackRock*.[[9]](#footnote-9)

Beispiele für echt-welt Anwendungen sind zum einem die Kategorisierung von Millionen von astronomischen Objekten mit einem Supercomputer der NASA, wobei die Nutzung von Julia eine tausendfache Effizienzsteigerung ermöglichte. Zum anderen verwendet eine US-amerikanische Militärbehörde Julia, um ein System zur Vermeidung von Kollisionen im Luftverkehr zu entwickeln. Vorher wurden dafür eine generelle Struktur in *MATLAB* erschaffen, welche für Geschwindigkeit in *C++* implementiert wurde. Nun wird beides durch Julia realisiert.[[10]](#footnote-10)

## 2.3 Konkurrenten

Die Konkurrenz Julias besteht aus erklärlichen Gründen vor allem aus den Sprachen, an denen sich ihre Entwickler orientierten und welche auch in den gleichen Gebieten wie Julia verwendet werden. Also Programmiersprachen wie *C*, *Ruby* und *MATLAB* aber auch *R*, *Wolfram* oder *Fortran*. Besonders jedoch kristallisierte sich Python als Hauptkonkurrent Julias heraus.

Python ist abgesehen von der Webentwicklungssprache *JavaScript* inklusive *HTML* und *CSS* die weltweit meistgenutzte Programmiersprache der Welt. Fast die Hälfte aller Software-Entwickler der Welt haben mit Python gearbeitet.[[11]](#footnote-11) Somit dominiert Python verständlicherweise auch die Anwendungsbereiche Julias. Als aufkommender Neuling wird Julia oft mit der am etabliertesten Sprache, also Python verglichen.

# 3 Vergleich mit Python

## 3.1 Syntax und Semantik

### 3.1.1 Dynamische Typisierung Julias

Eine der Besonderheiten Julias ist, dass sie *Just-in-time-Kompilierung* (*JIT-Kompilierung*) nutzt, sodass nicht das gesamte Programm vorher kompiliert werden muss. Stattdessen verarbeitet Julia mit Hilfe des *Low-Level Virtual Machine Compilers* (*LLVM*) Funktionen oder Codeblöcke kurz vor der Ausführung in optimierten Maschinencode.[[12]](#footnote-12) Gleichzeitig werden die nötigen Variablentypen durch ein *Type Inference System* geschätzt, falls diese nicht vorher spezifiziert wurden. So hat Julia die Möglichkeit, dem Code speziell auf die Hardware des Nutzers und den konkreten Kontext zu optimieren und erhebliche Perfomancegewinne zu realisieren. Dadurch werden eine einfachere und schnellere Entwicklung und Flexibilität beim Programmieren geboten.[[13]](#footnote-13)

Python nutzt zwar grundlegend anderer Kompilierungssysteme, ist allerdings wie Julia auch eine *dynamisch-typisierte* Programmiersprache. Der Unterschied liegt hier also in der Art, wie die beiden Sprachen den Code ausführen und die dabei erbrachte Perfomance. Während Python einen interpretierten Ansatz verfolgt, der Zeile für Zeile analysiert, führt Julia eine gezielte Optimierung durch.[[14]](#footnote-14)

### 3.1.2 Variablen

Wie gerade angedeutet ist es in Julia möglich, den Datentyp einer Variable zu spezifizieren. Hierbei Julia vertritt viele Datentypen, die wichtigsten, unter anderem auch für die Data-Science, sind jedoch *Int64* für ganze Zahlen, *Float64* für reelle Zahlen, *Bool* für Booleans oder *String* für Strings genutzt werden. Die Zahl im Suffix gibt die Anzahl der Bits an, die zur Speicherung des jeweiligen Datentyps verwendet werden. Dies bietet dem Nutzer Optionen Leistungs- und Speicherverwaltung. Die Deklaration dieser Variablen erfolgt, wie in den meisten Programmiersprachen auch, indem man ihr einen Namen gibt und ihr mit dem Operator „=“ einen Wert zuweist. (Abb. 1) Möchte der Nutzer nun den Variablentyp selbst festlegen, kann er den Variablennamen mit „::Datentyp“ erweitern.[[15]](#footnote-15) (Abb. 2)

Python behandelt die Deklarierung von Variablen fast identisch wie Julia.[[16]](#footnote-16) Die beide Sprachen unterscheiden sich jedoch darin, dass es für den Nutzer in Python nicht möglich ist, einen Variablentyp selbstständig zu bestimmen. Julia ist also Python in diesem Aspekt einen Schritt voraus, da die Programmiersprache es ermöglicht, die Speicher- und Leistungsverluste des Komforts der *dynamischen Typisierung* durch eine optionale statische Variablentypisierung aufzuheben.[[17]](#footnote-17) Zudem bietet Julia dem Nutzer auch durch die Möglichkeit, die Speicherkapazität spezifischer Variablen zu bestimmen, umso mehr Kontrolle über die Interna seines Programms.

### 3.1.3 Booleans und Vergleichsoperationen

Die drei Vergleichsoperatoren für Booleans in Julia sind „!“ für *Nicht*, „&&“ für *Und* und „||“ für *Oder*. (Abb. 3) Um die Gleichheit zweier Werte zu vergleichen, wird „==“ verwendet. Das dementsprechende Invers kann durch „!=“ oder „≠“ geprüft werden. Andere Relationen wie *Größer* oder *Kleiner* werden jeweils mit „>“ und „<“ durchgeführt. Mit „>=“ und „≥“ sowie „<=“ und „≤“ sind akzeptierte Operatoren, um die Gleichheit mit einzuschließen.[[18]](#footnote-18)

Im Vergleich zu Python gibt es hier kaum signifikante Unterschiede. Die drei Vergleichsoperatoren werden in Python auf Englisch ausgeschrieben, sodass sie in derselben Reihenfolge „not“, „and“ und „or“ sind. Zudem hat Julia die Besonderheit, Unicode-Symbole zu akzeptieren. Dieses Feature bringt keine funktionellen Vorteile, kann aber durchaus die Leserlichkeit des Codes steigern.[[19]](#footnote-19)

### 3.1.4 Konditionen

Diese Vergleichsoperationen können in bedingten Anweisungen verwendet werden. In Julia gibt es die Schlüsselwörter „if“, „else“ und „elseif“, die diese indizieren. Nach diesen Schlüsselwörtern wird die Bedingung eingesetzt, wonach in der auszuführende Codeblock ab der nächsten Zeile eingerückt beginnt. Schließlich ist am Ende ein „end“ nötig, um das Ende der bedingten Anweisungen zu setzen.[[20]](#footnote-20) (Abb. 4)

Die Unterschiede zu Python findet man hier ebenfalls nur in den Formalien. So benötigt Python einen Doppelpunkt nach der Bedingung und kein „end“-Schlüsselwort. Außerdem nennt sich das Schlüsselwort zur Überprüfung einer weiteren Bedingung in Python „elif“.[[21]](#footnote-21)

### 3.1.5 Schleifen

Die zwei Schleifentypen, die *for-Schleife* und die *while-Schleife* sind natürlich auch in Julia vertreten. Der Aufbau ist gleich, wie bei den bedingten Anweisungen, mit den jeweiligen Schlüsselwörtern „for … in …“ und „while“. Wieder muss man ein „end“ and das Ende der Schleife setzten. Da Zahlenbereiche im Bezug auf *for-Schleifen* unverzichtbar sind, ist es von Vorteil die Erstellung dieser in Julia zu erläutern. Um einen Zahlenbereich zu erstellen verwendet man den Operator „a:b“, wobei *a* die Startzahl ist und *b* die Endzahl. Die Schrittgröße *c* lässt sich ebenfalls einfach mit „a:c:b“ festlegen.[[22]](#footnote-22) (Abb. 5)

Wieder wird in Python auf einen Doppelpunkt am Ende der ersten Zeile verlangt und auf das Schlüsselwort „end“ verzichtet. Ansonsten liegen hier zwischen Julia und Python dieselbe Syntax vor. Einen Zahlenbereich erschafft man in Python jedoch mit der *range()-Funktion*.[[23]](#footnote-23) Diese hat dieselben Funktionalitäten wie Julias Konterpart. Man könnte hier für mehr Platzeffizienz und Leserlichkeit zugunsten Julias argumentieren, jedoch ist dies letztlich subjektiv zu bewerten.

### 3.1.6 Funktionen

Funktionen sind ein zentrales Mittel, um Code zu strukturieren und wiederverwendbar zu machen. Eine grundlegende Funktion kann in Julia mittels des Schlüsselwortes „function“ und dem darauffolgenden Funktionsnamen mit den benötigten Argumenten in Klammern folgend eingeleitet werden. Daraufhin folgt eingerückt der Funktionsinhalt, wobei der Funktion durch „return“ ein Rückgabewert zugewiesen werden kann. Zuletzt wird wieder ein „end“ benötigt, um den Codeblock abzuschließen. Es ist auch möglich, Funktionen in einer Zeile zu deklarieren. Hierzu wird der Funktionsname mit Hilfe eines Gleichheitszeichens und den darauffolgenden Code definiert.[[24]](#footnote-24) (Abb. 6) Eine wertvoller Attribut Julias ist *Mutiple* *Dispatch*. *Mutiple* *Dispatch* ermöglicht es, verschiedene Implementationen einer Funktion zu schreiben, die abhängig von den Datentypen der gegebenen Elemente Unterschiedliches durchführen. Julia wählt dann beim Ausführen des Codes die Implementation der Funktion, die am besten zu den Datentypen passt.[[25]](#footnote-25)

Wieder ist in Python ein Doppelpunkt nach der Deklaration einer Funktion nötig, sowie kein „end“ benötigt wird. Zudem wird eine Funktion durch das Schlüsselwort „def“ anstatt von „function“ eingeleitet. Auch eine einzeilige Funktion ist möglich. Dafür wird der Funktionsinhalt schlicht nach dem Doppelpunkt etwas weitergerückt eingesetzt.[[26]](#footnote-26) Mit *Mutiple* *Dispatch* besitzt Julia ein mächtiges Feature, welches Python nicht besitzt. Python dahingegen arbeitet mit *Single* *Dispatch*, was deutlich weniger Flexibilität bietet. Diese Flexibilität stammt aus der Vernetzung Julias mit dem *generischen Programmieren*, die größtenteils durch *Mutiple* *Dispatch* entsteht. So ist es dem Nutzer möglich, seinen Code auf eine Vielzahl von Datentypen oder Datenstrukturen anwendbar zu gestalten. Dadurch wird er wiederwendbarer und das Arbeiten mit mehreren unterschiedlichen Bibliotheken wird stark vereinfacht.

### 3.1.7 Auswertung der Syntax und Semantik

Ausgenommen von kleinen Formalitäten sind die Syntax von Julia und Python sehr ähnlich. Vor allem im Vergleich zu anderen Sprachen, wie *C* oder *Fortran*, an denen sich Julia ebenfalls orientierte, welche eine deutlich andere Syntax aufweisen. Es macht sich also bemerkbar, dass die Entwickler von Julia sich stark von Pythons Syntax inspirieren lassen haben.

Auf semantischer Ebene unterscheiden sich die beiden Programmiersprachen wie erwähnt in der Art der Kompilierung, wobei Julia *JIT-Kompilierung* über die *LLVM* anwendet und Python im Gegensatz dazu einen *Interpreter* verwendet. Sie ähneln sich in dem Aspekt der *dynamischen* *Typisierung*, auch wenn diese unterschiedlich umgesetzt wird. Jedoch ist es dem Nutzer in Julia möglich, Variablen weitaus detaillierter anzupassen. Ebenfalls arbeitet Julia mit *Mutiple Dispatch*, während Python dahingegen mit *Single Dispatch* arbeitet.

Trotzt der verwandet Syntax, schafft es Julia durch einige Features dem Nutzer mehr Komfort, Flexibilität und Leistung in mehreren Aspekten zu fordern.

## 3.2 Leistung

Im bestehenden Vergleich wurde häufig erwähnt, dass Julia Perfomancegewinne gegenüber Python realisieren kann. Sei es durch *JIT-Kompilierung* oder andere Methoden. Nun stellt sich die Frage zu welchem Ausmaß Julia schneller ist als Python. Dazu werden Benchmarks, welche denselben optimierten Code in beiden Sprachen durchgeführt haben,ausgewertet.

Um den Umgang mit großen Zahlen zu untersuchen, bietet sich das Berechnen von Pi-Nachkommazahlen an. Dies ist nicht allzu rechen-lastig, hantiert jedoch mit sehr großen Zahlen. Ein ähnlicher Typ von Kalkulationen findet man beim Trainieren eines maschinellen Lernmodells.[[27]](#footnote-27) Hierbei schneidet Julia je nach Benchmark über einen Faktor von 5,2[[28]](#footnote-28) über etwa 10[[29]](#footnote-29) bis hin zu 80[[30]](#footnote-30) mal schneller ab.

Die Berechnung des Mandelbrot-Sets verlangt hohe Präzision und einen Umgang mit komplexen Zahlen.[[31]](#footnote-31) Bei den dazu durchgeführten Benchmarks ist Julia von 57-mal[[32]](#footnote-32) bis etwa 100-mal[[33]](#footnote-33) schneller als Python.

Im Bereich der wissenschaftlichen Simulationen, besonders in der Physik, sind Matrizenmultiplikationen nicht wegzudenken, um beispielsweise Felder zu modellieren.[[34]](#footnote-34) Zudem operieren viele Modelle im maschinellen Lernen mit Matrizen, insbesondere mit Matrizenmultiplikation.[[35]](#footnote-35)

## 3.3 Community und Unterstützung

…

## 3.4 Lernkurve und Lesbarkeit

…

## 3.5 Auswertung

…

# 4 Praktische Anwendung von Julia

## 4.1 Erläuterung des praktischen Anteils

### 4.1.1 Begriff: Platonische Körper

Etwa 400 v. Chr. untersuchte der griechische Mathematiker Platon symmetrische und regelmäßige Körper. Dabei stieß er zunächst auf vier Körper, welche einzigartige symmetrische Eigenschaften besaßen. Dem damaligen Weltbild entsprechend assoziierte Platon diese vier besonderen Körper mit den Elementen Feuer, Luft, Erde und Wasser. Später entdeckte er noch einen fünften Körper, der dieselben Eigenschaften mit den anderen Vier teilt. Jene Körper wurden nach ihm benannt und sind heutzutage als die fünf Platonischen Körper bekannt.[[36]](#footnote-36)

Die Wissenschaft hat nun präzise Definitionen für die Platonischen Körper: „Ein Polygon heißt regulär, wenn seine Ecken ein regelmäßiges n-Eck bilden. Ein Polyeder heißt regulär, wenn seine Flächen kongruente regelmäßige n-Ecke sind und jede Ecke zu genau m Polygonflächen gehört.“[[37]](#footnote-37)

Hierbei sind die Platonischen Körper die exakt fünf möglichen reguläre Polyeder. Sie heißen *Tetraeder*, *Hexaeder*, *Oktaeder*, *Dodekaeder* und *Ikosaeder*.

### 4.1.2 Ziel des Programms

Die Platonischen Körper haben einige besondere Eigenschaften. Eine davon ist, dass, bis auf den Tetraeder, jeder dieser Körper dual zu einen der anderen Körper ist. Der Tetraeder ist dual zu sich selbst.

Zwei Platonische Körper sind dual zueinander, wenn die Mittelpunkte der Seiten des einen Körpers die Eckpunkte des anderen Körpers bestimmen. (Abb. 1) Somit sind die Körper zueinander dual, die jeweils die gleiche Anzahl von Eckpunkten zu Seiten besitzen. Die drei verschiedenen Dualitäten sind daher Tetraeder zu sich selbst, Hexaeder und Oktaeder und Dodekaeder und Ikosaeder.[[38]](#footnote-38)

Nun ist es das Ziel des praktischen Anteils dieser Seminararbeit, diese Dualitäten der Platonischen Körper in einer Animation zu visualisieren. Diese Animation soll mit Hilfe von Julia erstellt werden.

### 4.1.3 Thematischer Bezug

Die Relevanz des praktischen Anteils in Bezug auf die Forschungsfrage macht sich auf mehreren Ebenen bemerkbar. Zum einem wird die Datenverarbeitung und mathematische Fähigkeit Julias benötigt, um die Mittelpunkte der Seiten zu berechnen, welche die neuen Eckpunkte sind. Zum anderen bietet der Animationsaspekt des Programms die Möglichkeit, die Visualisierungskompetenz von Julia zu untersuchen. Zuletzt kann auch auf die Effizienz und Geschwindigkeit und Nutzer- bzw. Lernfreundlichkeit Julias eingegangen werden.

## 4.2 Umsetzung

…

# 5 Fazit

…

# 6 Literaturverzeichnis

…

# 7 Anhang

…

1. Jones, B.: Avoiding Data Pitfalls, 2019, S. 173 [↑](#footnote-ref-1)
2. 1 vgl. Bezanson, J., Karpinski, S., Shah, V., & Edelman, A.: Why We Created Julia, 2012 [↑](#footnote-ref-2)
3. 2 vgl. <https://en.wikipedia.org/wiki/Julia_(programming_language)> (letzter Zugriff: 11.10.2024)

   3 vgl. Bezanson, J., Karpinski, S., Shah, V., & Edelman, A.: Why We Created Julia, 2012

   4 Bezanson, J., Karpinski, S., Shah, V., & Edelman, A.: Why We Created Julia, 2012

   5 vgl. <https://en.wikipedia.org/wiki/Julia_(programming_language)> (letzter Zugriff: 11.10.2024)

   6 vgl. <https://en.wikipedia.org/wiki/Julia_(programming_language)> (letzter Zugriff: 11.10.2024) [↑](#footnote-ref-3)
4. [↑](#footnote-ref-4)
5. [↑](#footnote-ref-5)
6. [↑](#footnote-ref-6)
7. vgl. Bezanson, J., Karpinski, S., Shah, V., & Edelman, A.: Julia: A Fresh Approach to Numerical Computing, 2017, S. 96-97 [↑](#footnote-ref-7)
8. vgl. <https://youtube.com/watch?v=FihLyzdjN_8&si=ID9S6E1QNt24mRhR> (letzter Zugriff: 11.10.2024) [↑](#footnote-ref-8)
9. vgl. <https://en.wikibooks.org/wiki/Introducing_Julia/Jobs#cite_note-5> (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-9)
10. vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 17-18 [↑](#footnote-ref-10)
11. vgl. <https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/> (letzter Zugriff: 11.10.2024) [↑](#footnote-ref-11)
12. vgl. <https://testsubjector.github.io/blog/2020/03/26/The-Julia-Compilation-Process> (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-12)
13. vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 20-21 [↑](#footnote-ref-13)
14. vgl. <https://www.geeksforgeeks.org/internal-working-of-python/> (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-14)
15. vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 21-24 [↑](#footnote-ref-15)
16. vgl. Downey, A.: Think Python, 2015, S. 9-10 [↑](#footnote-ref-16)
17. vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 21 [↑](#footnote-ref-17)
18. vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 24-26 [↑](#footnote-ref-18)
19. vgl. Downey, A.: Think Python, 2015, S. 40 [↑](#footnote-ref-19)
20. vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 31 [↑](#footnote-ref-20)
21. vgl. Downey, A.: Think Python, 2015, S. 41-42 [↑](#footnote-ref-21)
22. vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 32 [↑](#footnote-ref-22)
23. vgl. <https://www.learnpython.org/en/Loops> (letzter Zugriff: 13.10.2024) [↑](#footnote-ref-23)
24. vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 26-28 [↑](#footnote-ref-24)
25. vgl. <https://docs.julialang.org/en/v1/manual/methods/> (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-25)
26. vgl. Downey, A.: Think Python, 2015, S. 19-20 [↑](#footnote-ref-26)
27. vgl. <https://juliacomputing.com/assets/pdf/JuliaVsPython.pdf> (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-27)
28. vgl. <https://juliacomputing.com/assets/pdf/JuliaVsPython.pdf> (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-28)
29. vgl. https://julialang.org/benchmarks/ (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-29)
30. vgl. <https://kamemori.com/research/fortran/speed_montecarlo_lcgs.html> (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-30)
31. vgl. https://de.wikipedia.org/wiki/Mandelbrot-Menge (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-31)
32. <https://juliacomputing.com/assets/pdf/JuliaVsPython.pdf> (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-32)
33. vgl. https://julialang.org/benchmarks/ (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-33)
34. vgl. <https://www.youtube.com/watch?v=Ujvy2-o1I9c> (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-34)
35. vgl. https://medium.com/@sruthy.sn91/matrix-operations-and-applications-in-machine-learning-1d6281ac38ab (letzter Zugriff: 14.10.2024) [↑](#footnote-ref-35)
36. vgl. Ulbrich, D.: Die Platonischen Körper, o. J., S. 1 [↑](#footnote-ref-36)
37. Ulbrich, D.: Die Platonischen Körper, o. J., S. 2 [↑](#footnote-ref-37)
38. vgl. Ulbrich, D.: Die Platonischen Körper, o. J., S. 2-3 [↑](#footnote-ref-38)