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

„Visualization gives you answers to the questions you didn't know you had."[1]

- Ben Schneiderman

Unser heutiges Zeitalter ist geprägt von Informationen und Daten. In fast jeder alltäglichen Situation, welche sich mit Technologie beschäftigt, werden unzählbar viele Daten ausgetauscht, ausgewertet und wiederum ausgetauscht. Dieser Maschinencode ist 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, also *visualisiert* werden. Diese Verarbeitung und Auswertung von Daten beschreibt die *Data-Science*.[2]

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.[3] (Abb.1)

Das Ziel dieser Seminararbeit ist es einen umfassenden Vergleich zwischen Julia und Python zu erschaffen, wodurch schließlich auf Julias Relevanz und Daseinsberechtigung geschlossen wird. Die präzise Forschungsfrage lautet: Inwiefern ist Julia im Vergleich zu Python für die Data-Science, insbesondere in der Datenvisualisierung, relevant und geeignet?

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.

[1] Jones, B.: Avoiding Data Pitfalls, 2019, S. 173
[2] vgl. https://de.wikipedia.org/wiki/Data_Science (letzter Zugriff: 15.10.2024)
[3] vgl. Klok, H., Nazarathy, Y.: Statistics with Julia, 2020, S. 1-2

Die Relevanz dieser 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.[4]

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.[5] 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.[6]

---

[1] vgl. Bezanson, J., Karpinski, S., Shah, V., & Edelman, A.: Why We Created Julia, 2012
[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)

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.“ [7] 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.[8]

2.2 Anwendungen

Der grundlegende Aufbau von Julia verschafft 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[9], 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,

[9] vgl. Bezanson, J., Karpinski, S., Shah, V., & Edelman, A.: Julia: A Fresh Approach to Numerical Computing, 2017, S. 96-97

gewöhnliche Differentialgleichungen zu lösen, was für maschinelles Lernen und für die vorher erwähnten Simulationen ein großer Vorteil ist.[10]

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*.[11]

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.[12]

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.[13] Somit dominiert Python verständlicherweise auch die Anwendungsbereiche Julias. Als aufkommender Neuling wird Julia oft mit der am etabliertesten Sprache, also Python verglichen.

[10] vgl. https://youtube.com/watch?v=FihLyzdjN_8&si=ID9S6E1QNt24mRhR (letzter Zugriff: 11.10.2024)
[11] vgl. https://en.wikibooks.org/wiki/Introducing_Julia/Jobs#cite_note-5 (letzter Zugriff: 14.10.2024)
[12] vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 17-18
[13] vgl. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ (letzter Zugriff: 11.10.2024)

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.[14] 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.[15]

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.[16]

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.

[14] vgl. https://testsubjector.github.io/blog/2020/03/26/The-Julia-Compilation-Process (letzter Zugriff: 14.10.2024)
[15] vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 20-21
[16] vgl. https://www.geeksforgeeks.org/internal-working-of-python/ (letzter Zugriff: 14.10.2024)

(Abb. 2) Möchte der Nutzer nun den Variablentyp selbst festlegen, kann er den Variablennamen mit „::Datentyp“ erweitern.[17] (Abb. 3)

Python behandelt die Deklarierung von Variablen fast identisch wie Julia.[18] 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.[19] 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*. 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.[20] (Abb. 4)

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.[21]

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

[17] vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 21-24
[18] vgl. Downey, A.: Think Python, 2015, S. 9-10
[19] vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 21
[20] vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 24-26
[21] vgl. Downey, A.: Think Python, 2015, S. 40

auszuführende Codeblock ab der nächsten Zeile eingerückt beginnt. Schließlich ist am Ende ein „end“ nötig, um das Ende der bedingten Anweisungen zu setzen.[22] (Abb. 5)

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“.[23]

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.[24] (Abb. 6)

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*.[25] 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

[22] vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 31
[23] vgl. Downey, A.: Think Python, 2015, S. 41-42
[24] vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 32
[25] vgl. https://www.learnpython.org/en/Loops (letzter Zugriff: 13.10.2024)

abzuschließen. Es ist auch möglich, Funktionen in einer Zeile zu deklarieren. Hierzu wird der Funktionsname mit Hilfe eines Gleichheitszeichens und den darauffolgenden Code definiert.[26] (Abb. 7) 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.[27]

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. [28] 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

[26] vgl. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 26-28
[27] vgl. https://docs.julialang.org/en/v1/manual/methods/ (letzter Zugriff: 14.10.2024)
[28] vgl. Downey, A.: Think Python, 2015, S. 19-20

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.[29] Hierbei schneidet Julia je nach Benchmark über einen Faktor von 5,2[30] über etwa 10[31] bis hin zu 80[32] mal schneller ab.

Die Berechnung des Mandelbrot-Sets verlangt hohe Präzision und einen Umgang mit komplexen Zahlen.[33] Bei den dazu durchgeführten Benchmarks ist Julia von 57-mal[34] bis etwa 100-mal[35] schneller als Python.

Im Bereich der wissenschaftlichen Simulationen, besonders in der Physik, sind Matrizenmultiplikationen nicht wegzudenken, um beispielsweise Felder zu modellieren.[36] Zudem operieren viele Modelle im maschinellen Lernen mit Matrizen, insbesondere mit Matrizenmultiplikation. [37] Nun beginnen die unterschiedlichen

[29] vgl. https://juliacomputing.com/assets/pdf/JuliaVsPython.pdf (letzter Zugriff: 14.10.2024)
[30] vgl. https://juliacomputing.com/assets/pdf/JuliaVsPython.pdf (letzter Zugriff: 14.10.2024)
[31] vgl. https://julialang.org/benchmarks/ (letzter Zugriff: 14.10.2024)
[32] vgl. https://kamemori.com/research/fortran/speed_montecarlo_lcgs.html (letzter Zugriff: 14.10.2024)
[33] vgl. https://de.wikipedia.org/wiki/Mandelbrot-Menge (letzter Zugriff: 14.10.2024)
[34] https://juliacomputing.com/assets/pdf/JuliaVsPython.pdf (letzter Zugriff: 14.10.2024)
[35] vgl. https://julialang.org/benchmarks/ (letzter Zugriff: 14.10.2024)
[36] vgl. https://www.youtube.com/watch?v=Ujvy2-o1I9c (letzter Zugriff: 14.10.2024)
[37] vgl. https://medium.com/@sruthy.sn91/matrix-operations-and-applications-in-machine-learning-1d6281ac38ab (letzter Zugriff: 14.10.2024)

Benchmarks aufgrund der unterschiedlichen genutzten Algorithmen, Disparitäten aufzuzeigen. Dennoch ist Julia in allen Fällen schneller. In isolierten Fällen sogar bis zu 580-mal[38] schneller.

Die bisher beleuchteten Themengebiete sind bereits für die Data-Science relevant. Jedoch gibt es auch Benchmarks, die sich auf Data-Science-nische Operationen fokussieren. Zum Beispiel das Lesen von *CSV-Dateien*. *CSV* ist ein sehr weit verbreitetes Format zum Speichern von Daten in tabellarischer Form. Diese zu Lesen und dadurch Daten zu extrahieren ist also essenziell für jegliche Programme in der Data-Science. Um dies durchzuführen, werden aus ersichtlichen Gründen Bibliotheken benötigt. Dennoch zeigt das Benchmark deutlich die Stärken von Julia auf, da Julia etwa 10 bis 20-mal[39] schneller CSV-Dateien liest, als Python und auch R.

Es gibt noch viele weitere Benchmarks, die Julia in den unterschiedlichsten Ebenen mit anderen Programmiersprachen vergleicht. Verallgemeinert lässt sich sagen, dass Julia immer eine der, wenn nicht die schnellste untersuchte Programmiersprache ist.[40] Die hier angesprochenen Benchmarks decken ein paar der, für die Data-Science relevanten, Funktionen einer Programmiersprache, sodass sich auch im Bezug auf die Forschungsfrage ein Bild auf die Leistung von Julia und Python geschaffen werden kann.

3.3 Community und Unterstützung

Im letzten Abschnitt wurden Bibliotheken erwähnt. Diese sind ein Aspekt, der Unterstützung, die eine Programmiersprache durch ihre Community erhalten kann. Das Julia-Ökosystem besitzt über 10.000 von Nutzern der Sprache erstellte Bibliotheken.[41] Zudem gibt es acht lokale Communities, welche online, sowie offline Meetings veranstalten. Eine davon ist auch in Berlin.[42] Im Jahr 2014 fand die erste jährliche *JuliaCon*, eine hochkarätige Konferenz für alle Julia-Interessierten, statt. Dort halten Experten Vorträge, werden Bibliotheken vorgestellt oder Workshops

[38] https://juliacomputing.com/assets/pdf/JuliaVsPython.pdf (letzter Zugriff: 14.10.2024)
[39] vgl. https://www.juliabloggers.com/csv-reader-benchmarks-julia-reads-csvs-10-20x-faster-than-python-and-r/ (letzter Zugriff: 14.10.2024)
[40] siehe z.B. https://julialang.org/benchmarks/
[41] vgl. https://julialang.org/packages/ (letzter Zugriff: 14.10.2024)
[42] vgl. https://julialang.org/packages/ (letzter Zugriff: 14.10.2024)

durchgeführt.[43] Wie fast jede Programmiersprache hat auch Julia ein aktives Online-Forum, das als zentrale Diskussionsplattform rund um Julia dient.[44]

Nun fängt Python an zu scheinen. Etwa 45% aller Entwickler sind Teil einer Python-Community. [45] Zudem wurde es seit der Veröffentlichung in den 90ern von geschätzten 8.2 Millionen verschiedenen Nutzern verwendet.[46] Diese überwältigend große Community lässt Julias schlicht im Schatten stehen. Python hat ebenfalls seit 2003 seine eigene jährliche Konferenz, die *PyCon*[47] und ein Online-Forum[48]. Beides in einem weitaus größeren Ausmaß als Julia. Schließlich sind stand 2024 im *Python Package Index* (PyPI) mehr als 450.000 verschiedene Pakete registriert, was grob als die Anzahl der Python-Bibliotheken gezählt werden kann.[49]

Der Fakt, dass es kaum Programmiersprachen gibt, die mit Python in diesem Aspekt mithalten können, macht es Julia in diesem Vergleich nicht leichter. Julia ist zwar noch eine junge Programmiersprache und hat noch viel Potenzial zu wachsen, dennoch ist ihre Community der, von Python, um Magnituden unterlegen. In naher Zukunft wird sich daran auch nichts ändern.

3.4 Lernkurve und Nutzerfreundlichkeit

Wie schon etabliert, sind sich Python und Julia in ihrer Syntax sehr ähnlich. Da das Verhalten der Lernkurve, vor allem für Beginner, stark von der Syntax einer Sprache abhängt, sind die beiden Programmiersprachen in diesem Aspekt vergleichbar. Wenn sie sich jedoch syntaktisch unterscheiden, scheint Julia einen intuitiveren Ansatz zu verfolgen. Beispielweise kann Julias *Paketmanager* im *REPL* aufgerufen werden, wobei Python externe Paketmanager wie *pip* oder *conda* verwendet.[50]

---

[43] siehe: https://juliacon.org/2024/ (letzter Zugriff: 14.10.2024)
[44] siehe: https://discourse.julialang.org (letzter Zugriff: 14.10.2024)
[45] vgl. https://www.statista.com/statistics/1241923/worldwide-software-developer-programming-language-communities/ (letzter Zugriff: 14.10.2024)
[46] vgl. https://flatironschool.com/blog/python-popularity-the-rise-of-a-global-programming-language/#:~:text=A%20general-purpose%20programming%20language,an%20estimated%208.2%20million%20users. (letzter Zugriff: 14.10.2024)
[47] siehe: https://pycon.org (letzter Zugriff: 14.10.2024)
[48] siehe: https://www.python.org/community/ (letzter Zugriff: 14.10.2024)
[49] siehe: https://pypi.org (letzter Zugriff: 14.10.2024)
[50] vgl. https://codesolid.com/julia-vs-python-now-for-something-completely-different/ (letzter Zugriff: 15.10.2024)

Python wird allgemein als eine der am einfachsten zu lernenden Programmiersprachen angesehen. Allein die weit verbreitete Nutzung unter erfahrenden, aber auch neu beginnenden Software-Entwicklern[51] zeigt dies auf. Obwohl Julia auf sehr fortgeschrittene Programmierung optimiert und ausgelegt ist, schafft es die Sprache durch ihre einfache Syntax, die sich stark an der von Python orientiert, dennoch eine steile Lernkurve zu bieten.

3.5 Auswertung

Die Ergebnisse dieses Vergleichs zeigen klar, dass die Entwickler von Julia in Python eine große Inspiration sahen. Durch die Anlehnung an Pythons Syntax konnte dessen Nutzerfreundlichkeit größtenteils übernommen werden. Erinnern wir uns an das Versprechen der Erschaffer Julias, eine Programmiersprache zu erschaffen, die einfach ist wie Python und gleichzeitig so schnell wie *C*.[52] Den ersten Teil dieses Versprechens haben sie eingehalten. Auch der zweite Teil wurde, zumindest im Vergleich zu Python, definitiv umgesetzt. Die Betrachtung der für die Data-Science relevanten Aspekte einer Programmiersprache hat ergeben, dass Julia in jedem dieser Aspekte weitaus mehr Leistung bietet. Dies schafft Julia durch ihre Kompilierungsmethode, welche grundlegend anders zu Pythons ist. Einen klaren Vorteil sieht Python jedoch in dessen Community und die daher entstehenden Bibliotheken. Dadurch gibt es heutzutage kaum Grenzen dazu, was man mit der Sprache erstellen kann.

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

[51] vgl. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ (letzter Zugriff: 15.10.2024)
[52] vgl. Bezanson, J., Karpinski, S., Shah, V., & Edelman, A.: Why We Created Julia, 2012

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.[53]

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.“[54]

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. 8) 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.[55]

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

[53] vgl. Ulbrich, D.: Die Platonischen Körper, o. J., S. 1
[54] Ulbrich, D.: Die Platonischen Körper, o. J., S. 2
[55] vgl. Ulbrich, D.: Die Platonischen Körper, o. J., S. 2-3

Julia zu untersuchen. Zuletzt kann auch auf die Effizienz und Geschwindigkeit und Nutzerfreundlichkeit und Lernkurve Julias eingegangen werden.

4.2 Umsetzung

Es gibt einige Entwicklungsumgebungen für Julia, wie *Juno*, *Jupyter Notebooks*, *Pluto.jl* oder *JuliaBox*.[56] Letztendlich wurde jedoch *Visual Studio Code* (VS Code) für die Implementierung des praktischen Anteils gewählt, da es eine spezielle Julia-Erweiterung gibt, die viele hilfreiche Features mit sich bringt. Dazu gehören Syntaxhervorhebung, Auto-Vervollständigung, ein Debugger und ein integrierter *Julia Read-eval-print loop* (REPL), welcher interaktive Programmierung ermöglicht. Vor allem aber bietet die Erweiterung eine interne Plot-Unterstützung, was bedeutet, dass erstellte Plots direkt in einem weiteren Fenster in *VS Code* gezeigt und gespeichert werden. Dies hilft enorm bei Fehlerbehebung und beschleunigt den Programmierungsprozess im Allgemeinen.[57]

Um in Julia Grafiken zu erstellen, werden Bibliotheken benötigt. Zur Wahl stehen hier die beiden bekanntesten Pakete, die dies ermöglichen sind *Plots.jl* und *Makie*. Beide besitzen die, für das Programm, benötigten Funktionen, also die Fähigkeit, Plots zu erstellen und diese anschließend zu animieren. Da Plots.jl weitaus einsteigerfreundlich ist, [58] wurde es schließlich gewählt. Ebenfalls sollen die Bibliotheken nicht genutzt werden, um das Programm auf nur einige Zeilen zu kürzen. Dies ist nicht der Sinn des praktischen Anteils, im Bezug auf die Forschungsfrage. Vor allem *Makie* enthält sehr erweiterte Funktionen und Features, die das Programm stark vereinfachen könnten.

Da die nötigen Berechnungen im praktischen Anteil simpel sind, wäre es zwecklos auf die Geschwindigkeit dessen einzugehen. Da *Plots.jl* lediglich zum Darstellen der Berechneten Punkte verwendet wurde, lässt sich auf die Benutzererfahrung mit Julias Standardbibliothek eingehen. Besitzt man Vorkenntnisse über Python, dann lässt sich Julias Syntax schnell erfassen. Zudem vereinfacht die dynamische Typisierung den Programmierungsprozess stark und bietet Flexibilität.

[56] vgl. https://devm.io/programming/top-5-ides-julia-148054 (letzter Zugriff: 15.10.2024)
[57] vgl. https://code.visualstudio.com/docs/languages/julia (letzter Zugriff: 15.10.2024)
[58] vgl. https://discourse.julialang.org/t/comparison-of-plotting-packages/99860 (letzter Zugriff: 15.10.2024)

Letztlich lassen sich in dem verfassten Programm Julias Prommierparadigmen erkennen. Julia ist eine multi-paradigmatische Sprache[59], was sich beispielsweise anhand der Anwendung von Funktionen bemerkbar macht, welche prozedurale Programmierung ermöglichen. Ebenfalls kommen beim Berechnen der Mittelpunkte der Seiten Zustandsänderungen einiger Variablen vor. Demnach ist auch imperative Programmierung vertreten. Auf *Mutiple Dispatch* wurde verzichtet, da die Datentypen einheitlich sind. Die Struktur des Codes erinnert stark an objektorientierte Programmierung, obwohl sie nicht explizit in dem Programm zu finden ist. Es lassen sich also mehrere von Julias Paradigmen vorfinden, was beweist, dass sie eine multi-paradigmatische Sprache ist.

5 Fazit

In dieser Seminararbeit wurde die Daseinsberechtigung der Programmiersprache Julia in der Data-Science untersucht. Dies wurde anhand eines systematischen Vergleichs mit dem aktuellen Spitzenreiter in der Data-Science, Python, und der Auswertung eines Programms, welches in Julia geschrieben wurde, durchgeführt.

In Bezug auf die Forschungsfrage belegen die Ergebnisse, dass Julia durchaus eine wertvolle Programmiersprache für Datenwissenschaftler darstellt. Sie kombiniert die Einfachheit von Python mit einer beeindruckenden Geschwindigkeit, vergleichbar mit Sprachen wie *C* oder *Fortran*. Abgesehen von persönlichen Präferenzen sind die unzähligen und vielseitigen Bibliotheken Pythons einer der weniger technischen Gründe, Python als vorteilhaftere Option der beiden in der Data-Science zu betrachten. Obwohl Julia noch eine junge Programmiersprache ist, hat sich schon eine starke Community um sie etabliert, die aktiv an der Verbesserungen und Features arbeitet. Noch wird Julia eher als Nischensprache in den Bereichen der wissenschaftlichen Kalkulationen und des maschinellen Lernens angesehen. Doch sollte es ihr gelingen, in den Mainstream durchzubrechen, hat sie jede Grundlage dazu, eine weit verbreitete Programmiersprache zu werden. Daraufhin würden

[59] vgl. https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages (letzter Zugriff: 15.10.2024)

naturgemäß ein großer Anschub von Paketen-Entwicklern folgen, die Julia erweitern zu versuchen.

Da im systematischen Vergleich Julia weitaus detaillierter betrachtet wurde, als Python ist es durchaus möglich, dass Vorteile von Python, die in technischen Details liegen, nicht angeleuchtet wurden, und so eine verfälschte Sichtweise erschaffen wurde. Die Gewichtung dieser Fehler wäre zu hinterfragen.

Die Data-Science wird besonders in Anbetracht der kürzlichen Entwicklungen künstlicher Intelligenz, aber auch in weiteren Anwendungsbereichen noch eine lange Zeit sehr relevant sein. Die Grundlage jeder KI-Entwicklung sind Daten. Ohne eine große Menge von qualitativ hochwertigen Daten ist es unmöglich, leistungsfähige KI-Modelle zu trainieren. Data-Science bleibt also relevant. Damit offenbart sich auch die Relevanz dieser Untersuchung und schließlich auch die Relevanz Julias.

6 Literaturverzeichnis

Buchquellen:

1. Downey, A.: Think Python, Green Tea Press, 2015
2. Jones, B.: Avoiding Data Pitfalls, Wiley-VCH, 2019
3. Klok, H., Nazarathy, Y.: Statistics with Julia, Springer, 2020
4. Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, k. V., 2021

Fachartikel:

1. Bezanson, J., Karpinski, S., Shah, V., & Edelman, A.: Julia: A Fresh Approach to Numerical Computing, in: SIAM Review 59,1, 2017
2. Ulbrich, D.: Die Platonischen Körper, in: Mathematische Sammlung, o. J.

Internetquellen:

1. Bezanson, J., Karpinski, S., Shah, V., & Edelman, A.: Why We Created Julia https://julialang.org/blog/2012/02/why-we-created-julia/
2. Elizabeth, J.: Top 5 IDEs for Julia https://devm.io/programming/top-5-ides-julia-148054
3. Julia Computing, Inc.: Benchmarking Julia Against Python https://juliacomputing.com/assets/pdf/JuliaVsPython.pdf
4. Julia Computing, Inc.: CSV Reader Benchmarks https://www.juliabloggers.com/csv-reader-benchmarks-julia-reads-csvs-10-20x-faster-than-python-and-r/
5. Lockwood, John: Is Julia Easy to Learn for Python Developers? https://codesolid.com/julia-vs-python-now-for-something-completely-different/
6. Nath, S.: Matrix Operations and Applications in Machine Learning https://medium.com/@sruthy.sn91/matrix-operations-and-applications-in-machine-learning-1d6281ac38ab
7. Parth G.: The Math Found Everywhere in Physics: Matrices https://www.youtube.com/watch?v=Ujvy2-o1I9c
8. Rackauckas, C.: The Use and Practice of Scientific Machine Learning https://www.youtube.com/watch?v=FihLyzdjN_8
9. Vailshery, L. S.: Most widely utilized programming languages among developers worldwide 2024 https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/
10. Vailshery, L. S.: Programming language community sizes worldwide 2023

https://www.statista.com/statistics/1241923/worldwide-software-developer-programming-language-communities/
11. Van Deusen, Anna: The Rise of A Global Programming Language https://flatironschool.com/blog/python-popularity-the-rise-of-a-global-programming-language/#:~:text=A%20general-purpose%20programming%20language,an%20estimated%208.2%20million%20users.
12. o. A.: Data Science https://de.wikipedia.org/wiki/Data_Science
13. o. A.: Julia (programming language) https://en.wikipedia.org/wiki/Julia_(programming_language)
14. o. A.: Manedelbrot-Menge https://de.wikipedia.org/wiki/Mandelbrot-Menge
15. o. A.: Comparison of multi-paradigm programming languages https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages
16. o. A.: Introducing Julia/Jobs https://en.wikibooks.org/wiki/Introducing_Julia/Jobs#
17. o. A.: The Julia Compilation Process https://testsubjector.github.io/blog/2020/03/26/The-Julia-Compilation-Process
18. o. A: Interne Funktionsweise von Python https://www.geeksforgeeks.org/internal-working-of-python/
19. o. A.: Loops https://www.learnpython.org/en/Loops
20. o. A.: Methods https://docs.julialang.org/en/v1/manual/methods/
21. Julia Computing, Inc.: Benchmarking Julia Against Python https://juliacomputing.com/assets/pdf/JuliaVsPython.pdf
22. o. A.: Julia Micro-Benchmarks https://julialang.org/benchmarks/
23. o. A.: Speed Comparison Fortran vs Python vs Julia (Monte Carlo Method) https://kamemori.com/research/fortran/speed_montecarlo_lcgs.html
24. o. A.: Julia in Visual Studio Code https://code.visualstudio.com/docs/languages/julia
25. o. A.: Comparison of plotting packages https://discourse.julialang.org/t/comparison-of-plotting-packages/99860

7 Anhang

Abb. 1: (Storopoli, J., Huijzer, R., Alonso, L.: Julia Data Science, 2021, S. 11)

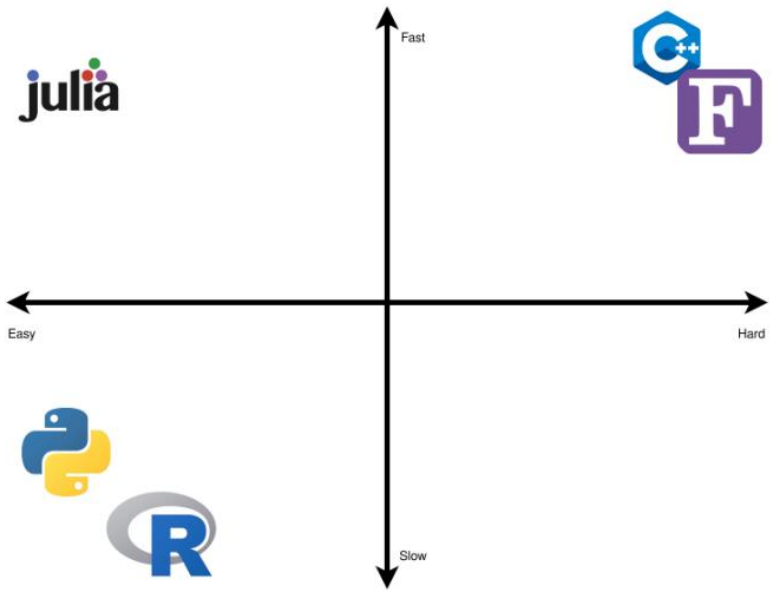


Figure 2.1: Scientific Computing Language Comparisons: logos for FORTRAN, C++, Python, R and Julia.

Abb. 2:

```
Zahl1 = 5 | 5
Zahl2 = 4.4 | 4.4
String1 = "Beispiel" | "Beispiel"
```

Abb. 3:

```
Zahl1::Int64 = 5
Zahl2::Float64 = 4.4
String1::String = "Beispiel"

typeof(Zahl1) | Int64
typeof(Zahl2) | Float64
typeof(String1) | String
```

Abb. 4:

```
(false != true) && (3 ≥ 2) || 4 ≠ 4 | true
```

Abb. 5:

```
if false
    "Beispiel"
elseif true
    "nicht Beispiel"
end | "nicht Beispiel"
```

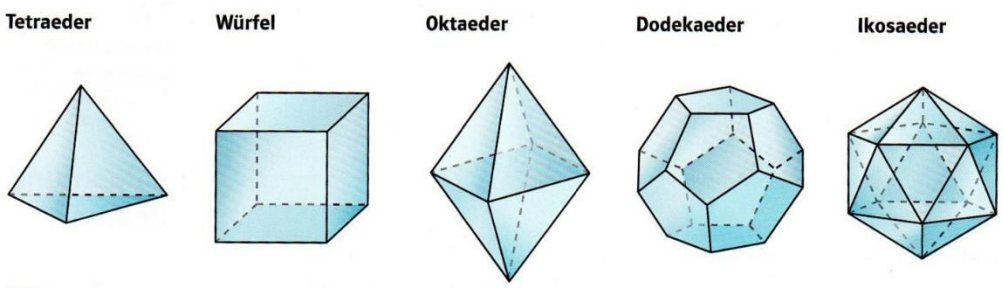
Abb. 6:

```
s = 0 | 0
for i in 1:0.5:3
    global s += i
end | ✓

s | 10.0
```

Abb. 7:

```
function Beispiel(Zahl1, Zahl2)
    return Zahl1 + Zahl2
end | Beispiel (generic function with 1 method)

Beispiel(5, 6) | 11
```

Abb. 8: (https://s3.eu-central-1.amazonaws.com/schuelerhilfe-bibliothek-dev/img-72e88ad5-c02e-455a-89a1-5b5a07c79e00.jpg)



Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Eichwalde, 16.10.2024_____
Ort, Datum

Theodor Schwarzrock_____
Unterschrift