Julia als Werkzeug für Visualisierungen in der Fraktalen Geometrie

Schöne, Roman
Hochschule Mannheim
Fakultät für Informatik
Paul-Wittsack-Str. 10, 68163 Mannheim

***Zusammenfassung*—In dieser Arbeit wird die Programmiersprache Julia bzgl. der Eignung für Visualisierungen innerhalb der fraktalen Geometrie untersucht. Betrachtet werden die Kriterien Performanz, Nachhaltigkeit, Parallelisierbarkeit, Verfügbarkeit von Softwarepaketen und Entwicklungsumgebungen. Das Ergebnis zeigt, dass Julia einen Kompromiss als Skriptsprache mit der Performanz bildet. Der Einsatz von Julia in einem ausgeweiteten Rahmen, lässt sich mithilfe der gegebenen Parallelisierung beschleunigen und nachhaltig gestalten. Das Julia-Ecosystem bietet aktuell die ausreichenden Mittel um Fraktale Kurven, Fraktale mittels Rastergrafiken darzustellen oder diese in 3-dimensionaler Form betrachten zu können. Die Erstellung eigener Visualisierungen gestaltet sich durch eine eingeschränkte Wahl von Entwicklungsumgebungen und einer hohen Kompilierzeit meist als langwieriger Prozess.**

Inhaltsverzeichnis

1. Einleitung

Schon im Jahr 1890 beschäftigte sich der Mathematiker Peano mit einer Kurve, die anschließend nach Ihm benannt wurde, die durch jeden Punkt einer 2-dimensionalen Oberfläche durchläuft [1]. Gefolgt 1904 mit Helge von Koch, der sich mit der nach ihm benannten Kochschen Schneeflocke auseinandersetzte. Die Entdeckungen aus der damaligen Zeit wurden von Mathematiker aus dem 19. Jahrhundert als „Monster“ bezeichnet, da diese seltsame Eigenschaften aufweisen [23] Weitere Grundlagenarbeit der fraktalen Geometrie beruht auf Werken des Mathematikers Gaston Julia und seinem Konkurrenten Pierre Fatou anfangs des 20. Jahrhunderts. Julia und Fatou erforschten das Verhalten der Iterationen von Funktionen der Form $f_c(z) = z^2 + c$ mit $z, c \in \mathbb{C}$ [25]. Diese Errungenschaften griff der Mathematiker Benoît Mandelbrot um die 1970er Jahre in seinem Buch *The Fractal Geometry of Nature* wieder auf und verlieh dem Bereich der fraktalen Geometrie an wachsender Popularität. In seinem Werk bedient sich Mandelbrot an einer Vielzahl von Visualisierungen für die betrachteten Fraktale. [23] Zu Ehren Ihrer Forschungsarbeiten wurden Objekte aus der fraktalen Geometrie nach Ihnen benannt. Die Mandelbrot-Menge $\mathbb{M}$ ist die Menge der komplexen Zahlen $c \in \mathbb{C}$, deren Konvergenz-Verhalten für die Iteration $f_c(z_i) = z_{i+1} = z_i^2 + c$ unter Wahl von $z_0 = 0$ beschränkt ist. Da sich die komplexen Zahlen als Punkte einer auffassen lassen, lässt sich die Mandelbrot-Menge in Form einer Rastergrafik visualisieren. Mithilfe der Darstellung lassen sich neue Vermutungen über die Eigenschaften der Mandelbrot-Menge aufstellen, welche ohne Visualisierung schwer zu erkennen sind. Die Begründer Gaston und Julia besaßen zur damaligen Zeit noch keine Computer, mit denen Sie ihre Forschungsobjekte darstellen konnten. [25] Mit dem Lauf der Zeit wächst die Rechenleistung der verfügbaren Computer stetig an und erschließt neue Möglichkeiten innerhalb der Forschung. Nach *Moore's Law* [18] verdoppelt sich die Anzahl der Komponenten die auf einen Chip passt jedes Jahr. Innerhalb der 70er Jahre wurde die Mandelbrotmenge mithilfe von ASCII-Art visualisiert [26]. Aktuell steht uns die Rechenleistung zur Verfügung Einblicke in die Welt der Fraktale mittels hochauflösender Rastergrafik zu gelangen. Die resultierenden Bilder wecken aufgrund der Ästhetik auch das Interesse vieler Nicht-Mathematiker sich mit dem Themengebiet zu befassen [23]. In der Informatik, die einen Spagat zwischen Formalwissenschaft und Ingenieurswissenschaft bildet, stellt sich die Herausforderung die Vielzahl an Objekten aus dem Zoo der fraktalen Geometrie auf effiziente und anschauliche Weise zu visualisieren.

2. Hintergrund

Für das Angehen von technischen Problemen werden ist populärste Ansatz zwei Sprachen zu verwenden. Eine Sprache mit leichter Syntax auf hoher Ebene wird in Kombination mit einer Sprache auf niedriger Abstraktionsebene mit hoher Performanz verwendet. Ein bekanntes Beispiel ist die Python-Bibliothek *NumPy*, die in C geschrieben ist [3]. Ein anderer Ansatz zur Lösung dieses Problems war es vorerst Algorithmen innerhalb von Skriptsprachen zu schreiben und diese später in hardwarenahe Sprachen zu übertragen. Dieses Vorgehen ist mit einem hohen zeitlichen Aufwand und einer hohen Fehleranfälligkeit während des Übertragungsprozess verbunden [21]. Julia wird als eine Lösung für diese Problematik, dem *Zwei Sprachen Problem* angesehen. Ein Team aus unabhängigen Entwicklern entschied sich 2009 den Startschuss für die Entwicklung von Julia zu setzen. Bis zur ersten Veröffentlichung von Julia verliefen 3

weitere Jahre . Im August 2018 wurde die Veröffentlichung von Julia 1.0.0 bekanntgegeben [14]. Julia ist eine dynamische Programmiersprache in welcher Variablen zwingend definiert werden müssen, bevor diese zum Einsatz kommen [4]. Eine Vielzahl an Elementen aus imperativen, funktionalen und objektorientieren Programmiersprachen lassen sich in Julia wiederfinden. Julia wurde für wissenschaftliches Berechnen konzeptioniert, aber ermöglicht auch allgemeine Programmierung. Die Inspiration von Julia liegt in den Sprachen Lisp, Perl, Lua und Ruby [2]. Wichtige Merkmale die Julia von anderen dynamischen Programmiersprachen abgrenzt sind nach Julia-Handbuch:

- Eine sehr schlanke Standard-Bibliothek, die alle grundlegenden numerischen Operationen und Typen bereitstellt
- Ein breit aufgestellte Sprache um Objekte zu konstruieren und deren Typen zu beschreiben
- Die Möglichkeit, dass Funktionen eine große Breite an unterschiedlichen Parametertypen entgegennehmen
- Eine gute Performanz, die derer statisch-kompilierter Sprachen wie beispielsweise C nahekommt

[2]. Die Einsatzmöglichkeiten beschränken sich nicht auf einen konkreten Bereich. Julia stellt einen Werkzeug zur Lösung von Problemen innerhalb der Informatik, Mathematik, Ingenieurswissenschaft, Medizin und Wirtschaft dar [4]. Von höherem Interesse ist für uns der Einsatz von Julia innerhalb Mathematik, konkreter als Werkzeug für Visualisierungen im Teilgebiet der fraktalen Geometrie.

3. Kriterien

Im folgenden Abschnitt betrachten wir die ausgewählten Kriterien bezüglich der Programmiersprache Julia und stellen jeweils die Relation der einzelnen Kriterien zur Fraktalen Geometrie her. Diese lassen sich unterteilen in generelle Eigenschaften der Programmiersprache Julia und welche, die speziell für Visualisierung innerhalb fraktale Geometrie von hoher Signifikanz sind. Wir evaluieren im Laufe der nächsten Abschnitte, weshalb die ausgewählten Kriterien für unsere Fragestellung relevant sind.

- Performanz
- Parallelisierbarkeit
- Verfügbarkeit Softwarepakete
- Entwicklungsumgebungen
- Nachhaltigkeit

Am Ende jedes Abschnitts beurteilen wir, wie gut das jeweilig genannte Kriterium durch Julia umgesetzt wurde.

3.1. Performanz

Für die Gestaltung interaktiver Software und der Erstellung von Bildern bzw. von Animationen bezüglich Fraktalen wird ein Anspruch auf eine schnelle Verarbeitung der Eingaben gesetzt. Besonders für Programme mit denen in Echtzeit verschiedene Formen von Fraktalen erkundet werden können ist eine gute Performanz von hoher Relevanz. Da viele Programmiersprachen heutzutage alle notwendigen Funktionalität zur Lösungen unterschiedlichen Problemtypen bereitstellen, spielt die Effizienz eine höhere Rolle innerhalb der Auswahl der Sprache, als im Vergleich zur Umsetzbarkeit [14]
In folgendem Abschnitt werden Performanz-Tests in den Programmiersprachen Julia 1.10.2, Python 3.12.2 und Java Open-JDK 19.0.2 durchgeführt. Die verwendete Central Processing Unit (CPU) ist ein AMD Ryzen 5 3600 unter Windows 10 mit 16GB verfügbaren Arbeitsspeicher. Der zu testende Code wird aus Sicht eines Einsteigers der jeweiligen Programmiersprache geschrieben. Der Code macht keinen Gebrauch von explizit verwendeter Parallelisierung. Außerdem vermeiden wir Optimierung einzugehen die von genaueren Wissen technischer Details, der jeweiligen Programmiersprache ausgehen. Die Zeiten werden mithilfe des Julia Package `BenchmarkTools.jl`, des Python Moduls `timeit` und in Java per Aufruf der Methode `System.nanoTime()` erfasst. Jeder parametrisierte Aufruf eine Funktion wird 10-Mal durchgeführt. Aus den 10 Aufrufen wählen wir das Minimum aus. Dieses Vorgehen liefert eine Untergrenze der Ausführungszeit des Codes. Die Kompilierzeit der jeweiligen Sprache wird in unseren Tests nicht miteinbezogen.

Performanz-Test. Die Elemente der Mandelbrot-Menge $\mathbb{M}$ können nicht genau bestimmt werden. Es ist dennoch möglich Näherungsbilder der Mandelbrot-Menge $\mathbb{M}$ zu skizzieren [26]. Wir überprüfen die Performanz von Julia indem wir die beanspruchte Zeit für die Erstellung eine solchen Näherungsbildes betrachten. Komplexe Zahlen sind im Kontext dieses Tests festgelegt als Datenstruktur bestehende aus zwei 64-Bit Fließkommazahlen nach Spezifikation IEEE-754. Die Mandelbrot-Menge ist nach Definition Teilmenge der komplexen Zahlen. Es lässt sich zeigen, dass alle Elemente der Mandelbrotmenge vollständig in der Kreisscheibe um 0 mit dem Radius 2 liegen [25]. Diese Eigenschaft machen wir uns in unserem Näherungsbild zu nutze. Demnach verwenden wir für die Visualisierung der Mandelbrot-Menge als Rastergrafik einen quadratischen Bildausschnitt mit Umkreismittelpunkt bei $0 + 0i$ und Seitenlänge 4. Um eine $n \times n$ Rastergrafik zu erhalten zerlegen wir unseren Ausschnitt in $n \times n$ Quadrate mit Seitenlänge $\frac{4}{n}$. Aus jedem Rasterquadrat wählen wir eine komplexe Zahl z_{ij} als Repräsentanten für dieses Quadrat aus. Wir bestimmen für jedes z_{ij} ob es nach einer festgelegten Anzahl an Iterationen ins Unendliche divergiert.
Wir nehmen an, dass alle z_{ij} Teil der Mandelbrot-Menge sind, wenn unsere maximale Iteration erreicht wurde und somit die Kreisscheibe nicht verlassen wurde. Diese Methodik fasst sich in Form des *Escape Time Algorithm* 1 zusammen. [5].

Algorithm 1 Escape Time Algorithmus

```
function EscapeTime(z, c, t_max)
    t ← 0
    while |z| < 2 and t < t_max do
        z ← z² + c
        t ← t + 1
    end while
    return t
end function
```

Der *Escape Time Algorithm* 1 muss für ein quadratisches $n \times n$ Raster n^2-Mal durchlaufen werden. Für die Anzahl der gesamten Ausführungen der `while`-Schleife des *Escape Time Algorithm* kann die obere Schranke $n^2 \cdot t_{max}$ bestimmt werden. Die obere Schranke wird erreicht wenn alle z_{ij} Teil der Mandelbrotmenge sind. Abbildung 3.1 stellt die benötigte Zeit zur Berechnung des Rasters unter maximaler Iteration $t_{max} = 10$ dar. Auf der Y-Achse lässt sich die Zeit in Sekunden logarithmisch ablesen. Auf der X-Achse liegen die betrachteten Rastergrößen n von 1 bis 100.

Python, Julia und Java weisen ein ähnliches logarithmisches Wachstumsverhalten auf, sind jedoch auf der Y-Achse versetzt. Die Berechnungszeit für Julia und Java liegt für kleinere Werte für n, relativ nah beieinander. Python hingegen benötigt für kleine Rastergrößen schon wesentlich länger. Unter den gemessenen Sprachen ist Java die schnellste, darauffolgend Julia und das Schlusslicht bildet Python.

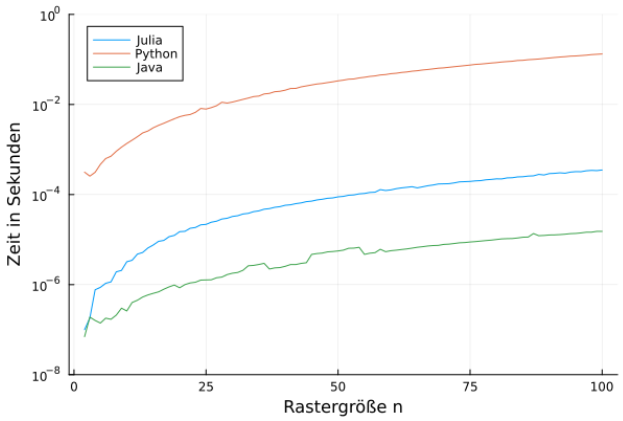

Abbildung 1. Zeitaufwand in Abhängigkeit der Rastergröße

3.2. Nachhaltigkeit

Im letzten Jahrhundert verhielt sich der Energieverbrauch zwischen unterschiedlichen Programmiersprachen nahezu analog zueinander. Innerhalb des 20. Jahrhunderts legen Computerhersteller und Softwareentwickler einen größeren Wert auf eine effizientere Energienutzung [19]. Neben dem Energieaufwand für Berechnungen wird weitere Energie für Kühlung, Anzeige, Lagerung und Kommunikation verwendet [9]. Bedeutet die überdurchschnittliche Performanz, dass Julia auch energieeffizienter ist? Der Energieverbrauch wird durch die Gleichung:

$$\text{Energie} = \text{Zeit} \times \text{Leistung}$$

beschrieben [19]. Visualisierungen innerhalb der Fraktalen Geometrie können einen hohen Rechenaufwand in Anspruch nehmen. Beispielsweise spielt die numerische Präzision Berechnung eines kleinen „Sichtfensters“ in die Mandelbrot-Menge einen limitierenden Faktor. Desto kleiner das Sichtfenster ist, desto mehr Zeit wird in Anspruch genommen dies zu Berechnen [13]. Durch Verwendung von Zahlen mit arbiträrer Präzision kann diese Rechendauer, die einer 64-Bit Fließkommazahl übersteigen. Ein weiterer Anwendungsfalls ist die Wahl eines hohen t_{max} innerhalb des *Escape Time Algorithmus* 1 welches ein genaueres Bild der Mandelbrotmenge liefert, aber auf der Gegenseite eine höhere Rechenzeit benötigt. Für die Darstellungen von Kurven, wie beispielsweise der Koch-Kurve n-ter Iteration wächst der Rechenaufwand mit wachsenden n mit an. Es lässt sich schließen dass in genauere und präzisere Berechnung ein höherer Energieaufwand einfließt. Ein Vergleich über die Energieeffizienz von Programmiersprachen liefert, dass kompilierte Sprachen im Durchschnitt schneller sind als Sprachen mit Virtueller Maschine, gefolgt von interpretierten Sprachen [19]. Dieser Vergleich wurde anfangs ohne Julia durchgeführt. Im Anschluss wurde Julia in der Version 1.3.1 hinzugefügt und in einem Vergleich mit 9 weiterer Sprachen evaluiert. Julia liegt in der neuen Durchführung auf Platz 2/10. [20]

3.3. Parallelisierbarkeit

Die Berechnung eines Bildes der Mandelbrot-Menge und der Julia-Menge lässt sich auf eine einfache Weise parallelisieren. Der Rechenaufwand für das große Bild kann in mehrere Teilprobleme, die Berechnung der jeweiligen Pixel, aufgeteilt werden. Dies ist von hohem Nutzen, da die Berechnungskosten für einzelne Pixel mit hohem Rechenaufwand verbunden ist [12]. Julia unterstützt die vier folgenden Wege zur Parallelisierung von Programmen [2]:

1) **Koroutinen** Julia bietet die Option die Ausführung von Abläufen zu Pausieren und an späterer Stelle wieder neu zu starten oder zu synchronisieren. Diese Berechnungslast kann auf unterschiedliche Threads verteilt werden.
2) **Multithreading** Aufgaben können auf unterschiedliche Threads oder CPU-Kerne aufgeteilt werden. Die unterschiedlichen Threads teilen sich denselben Speicher.
3) **Verteiltes Berechnen** Mithilfe des verteilten Berechnens können mehrere Julia-Prozesse, mit jeweils eigenem Speicher, laufen. Diese müssen nicht zwanghaft auf demselben Computer ausgeführt werden. Die Standard-Bibliothek `Distributed` bietet die Möglichkeit die Berechnungen von Funktionen auszulagern.
4) **Berechnen auf der GPU** Statt die Berechnungen auf der CPU stattfinden zu lassen, können Berechnungen auf die Grapics Processing Unit (GPU) ausgelagert werden.

Für allgemeine Probleme ist der leichteste Weg Multithreading zur Beschleunigung der Berechnungen einzusetzen [2]. Die Bereitstellung der Bibliothek für Verteiltes Berechnen reduziert zusätzlichen Aufwand seitens des Entwicklers sich um die Koordination der einzelnen Systeme zu kümmern. Eine Skalierung der Rechenleistung für große Projekte durch Julia von Haus aus gewährleistet.

3.4. Softwarepakete

In folgendem Abschnitt betrachten wir ob die durch das Julia Ecosystem bereitgestellte Ressourcen ausreichen um die Vielfalt unterschiedlicher Fraktale darzustellen. Dazu gehen wir nach einem *Top-Down*-Prinzip vor. Erstmalig wird das Julia Ecosystem auf einer allgemeinen Ebene evaluiert. Im weiteren Verlauf werden einzelne Softwarepakete im Rahmen als Hilfsmittels für die fraktale Geometrie bewertet. Wir überprüfen ob eine Auswahl von Fraktalen mithilfe Julia visualisiert werden können und wie gut die dafür existierenden Strategien umgesetzt sind. Wir unterteilen dabei die Fraktale in folgende Mengen, die nicht zwingend disjunkt zueinander sind.

- Fraktale, die Teilmenge der komplexen Zahlenebene sind. Beispiele dafür sind die Mandelbrot-Menge, sowie die Julia-Menge [25].
- Fraktale, die sich mithilfe eines Lindenmayer-Systems darstellen lassen, beispielsweise die Peano-Kurve oder die Kochsche Schneeflocke [1]
- 3-dimensionale Fraktale, wie zum Beispiel den Menger-Schwamm oder eine 3-dimensionale Form des Sierpinski-Dreiecks [24]

Julias Ecosystem. Neben den Standardbibliotheken, die mit der Installation einer Programmiersprache mitgeliefert werden, steht Nutzern die Möglichkeit offen eigene Softwarepakete zu entwickeln. Diese Pakete kapseln viele Funktionalitäten im Fokus eines Spezifischen Anwendungsbereichs bspw. Statistik. Um die Pakete der Öffentlichkeit zur Verfügung zu stellen, können diese in Paketverzeichnisse im Internet hochgeladen werden. Diese Möglichkeiten erweitern das traditionelle Verständnis von Softwarentwicklung im Sinner einer kollaborativen Zusammenarbeit von Entwicklern unabhängig ihres Standorts. [11].Tabelle 1 zeigt eine Auswahl von Verzeichnissen für Sprachen aus dem Bereich des wissenschaftlichen Berechnens basierend auf [22] [16] [8] [15] [6].

Nach dem PopularitY of Programming Language (PYPL) Index wird Python als populärste Programmiersprache evaluiert. Julia hingegen belegt hingegen Platz 24 [6]. Zwischen beiden Enden liegen die Programmiersprache R und MATLAB. Da MATLAB keinen klassischen Paketmanager besitzt wird hier zum Vergleich die Anzahl der von Nutzer bereit gestellten Programme auf dem MATLAB FileExchange verwendet. Aufallend ist das MATLAB trotz einer proprietären Lizenz eine höhere

Tabelle 1. PAKETANZAHL NACH SPRACHE UND PAKETVERZEICHNIS

Sprache	Paketverzeichnis	PYPL Rank	Anzahl Pakete
Python3	PyPi	1	553.784
R	CRAN	6	21.046
MATLAB	FileExchange	14	11.693
Julia	JuliaRegistries	24	11.066

Anzahl an Paketen bzw. Programmen bereitstellt. Diese hohe Popularität erklärt weshalb die Anzahl zwischen Paketen auf Python Package Index (PyPi), dem Python Package Index, und *JuliaRegistry* sich dem Verhältnis 50:1 nähert. Aufgrund des relativ jungen Alters, mit Veröffentlichung von Julia im Jahr 2012 [4] und der niedrigeren Popularität, besitzt Julia im Vergleich zu den restlichen ausgewählten Programmiersprachen die geringste Anzahl an Paketen.

Schildkröten und Kurven. Eine häufig verwendete Strategie zur Darstellung von Fraktalen Kurven sind die von Seymour Papert im Jahr 1980 entwickelten Turtle-Grafiken. Dabei bewegt sich eine unsichtbare „Schildkröte“ über eine Leinwand und hinterlässt eine farbige Spur. Dabei ist die Schildkröte auf eine Anzahl an Aktionen beschränkt, die aus der Perspektive der Schildkröte geschehen [1]. Diese können beispielsweise sein:

- F Die Schildkröte bewegt sich einen Schritt nach vorne und hinterlässt dabei eine farbige Spur
- \+ Die Schildkröte dreht sich um den Winkel α nach links
- \- Die Schildkröte dreht sich um den Winkel α nach rechts

Alleine mit einer kleinen Auswahl an Aktionen können komplexe Fraktale erschaffen werden. in Julia wird das Zeichnen von Turtle Grafiken durch das Paket `Luxor.jl` abgedeckt. Neben unserem Beispiel an 3 Aktionen stellt `Luxor.jl` 17 weitere möglichen Aktionen zur Bewegung der Schildkröte bereit [7].

Mithilfe der Regeln der Turtle-Grafiken lassen sich Lindenmayer Systeme, kurz L-System, darstellen. Ein L-System liefert eine Liste an Ersetzungsregeln die alle gleichzeitig auf eine Zeichenkette angewendet werden. [17]. Diese Ersetzungsregeln können beispielsweise auf Zeichenketten bzw. die Aktionen unserer Schildkröte, bestehend aus F, +, - angewendet werden. Der übrigbleibende Aufwand besteht nur noch darin das L-System in die Aktionen einer Turtle-Grafik zu übersetzen. Das Softwarepaket `Lindenmayer.jl` abstrahiert uns diesen Arbeitsaufwand weg.

Fraktale in $\mathbb{C}$. Die Mandelbrot-Menge, sowie die Julia-Menge liegen innerhalb der komplexen Zahlen. Mithilfe der Strategie die Mandelbrot-Menge in Form eines Rasters unter Nutzung des *Escape Time Algorithmus* 1 erhalten wir ein Raster bzw. eine Matrix mit den Fluchtzeiten der jeweiligen Eingabewerte. Für vielseitige Visualisierungen ist eine einheitliche Schnittstelle für Farbmodelle und Rastergrafiken vonnöten. Diese geforderte Funktionalität wird durch das Julia-Paket `JuliaImages.jl` realisiert. Das Paket `Makie.jl` liefert eine Darstellungsmöglichkeit für 2-dimensionale Rastergrafiken in Form interaktiver Anwendungen mithilfe Benutzereingaben wie bspw. Textboxen, Knöpfe, Slider [10].

3-dimensionale Fraktale. Neben der Möglichkeit 2-dimensionale Rastergrafiken zu erzeugen bietet `Makie.jl` die Option innerhalb eines 3-dimensionalen Raumes Objekte darzustellen. Mithilfe eines Voxel-Systems können fraktale Objekte mithilfe Würfeln angenähert werden. Neben Würfeln bieten Polygone und Linien weitere Grundbausteine für die Erstellung komplexer Objekte. [10]. Weitere Möglichkeiten zur Erstellung von 3-dimensionalen Objekten sind das angeben eines Volumens oder das Laden von `.obj`-Dateien. Die Konstruktion von größeren Objekten erstellt sich aufgrund der zusätzlichen Dimension als schwieriges Unterfangen dar [10].

3.5. Entwicklungsumgebungen

Die Julia Programmiersprache ist auf macOS, Windows und Linux frei erhältlich Mit der Veröffentlichung einer neuen Programmiersprache müssen auch Entwicklungsumgebungen geschaffen werden. Eine Entwicklungsumgebung bietet dem Programmierer nützliche Werkzeuge um Code effizient zu erstellen und zu verändern. Der Typ der Entwicklungsumgebung bzw. Integrated Devolopment Environment (IDE) lässt sich in 3 unterschiedliche Typen kategorisieren:

- eigenständiger Texteditor, die für das Schreiben einer spezifischen Programmiersprache oder einer engen Auswahl an zusammenhängender Programmiersprachen konzipierte wurde. Ein Beispiel für einen solchen Editor ist RStudio für R.
- Erweiterung eines schon existierenden Texteditor, die konzipiert um eine große Breite an Sprachen abzudecken. Ein Beispiel für einen solchen Texteditor ist Visual Studio Code, welches viele Sprachen durch Hinzufügen von Erweiterungen unterstützt.
- Notebooks, die eine Ansammlung an Zellen mit ausführbaren Code oder Notizen meist in Form von Markdown bieten. Die Entwicklung von Code in Notebooks geschieht in einer iterativen und interaktiven Form. Diese kommen meist innerhalb des Data-Science Bereichs zum Einsatz [27]. In der Praxis unterstützen Notebooks meist mehrere Sprachen oder es ist möglich diese per Erweiterung hinzuzufügen.

Wir klassifizieren eine Auswahl an Entwicklungsumgebungen für Julia. Für jede Entwicklungsumgebung fassen wir auf ob deren Entwicklung noch aktiv verfolgt wird. Die Daten erhalten wir durch die Repositories der jeweiligen Projekte auf Github. Die Arbeit an einer IDE klassifizieren wir als gestoppt oder pausiert, wenn seit Anfang des Jahres 2023 kein neuer Release erschienen ist.

Tabelle 2. EINORDNUNG ENTWICKLUNGSUMGEBUNGEN

IDE	Kategorie	Entwicklung
Julia (VS Code)	Erweiterung	Ja
Juno (Atom)	Erweiterung	Nein
Julia (IntelliJ)	Erweiterung	Nein
Julia-Studio	Eigenständig	Nein
Pluto	Notebook	Ja
IJulia (Jupyter)	Notebook	Ja

Aus Tabelle 2 ist zu erkennen, dass die Unterstützung von Julia in Form von Notebooks weiterhin gewährleistet ist. Eine eigenständige Entwicklungsumgebung an der aktiv gearbeitet wird ist in unserer Auswahl nicht zu finden. Aus der Kategorie der *Erweiterungen* wird alleinig Julia für Visual Studio Code unterstützt. Zu bemerken ist das die Programmiersprache Julia, sowie auch alle der genannten Entwicklungsumgebungen IDE aus Tabelle 2 einer proprietären Lizenz unterworfen ist [2]. Dies sorgt für eine hohe Zugänglichkeit.

4. Ergebnis

Im Gesamtbild liegen die Stärken der Programmiersprache Julia in der überdurchschnittlichen Geschwindigkeit als Skriptsprache. Die Kombination mit einer guten Energieeffizienz trägt dazu bei, dass Julia als grüne Programmiersprache angesehen werden kann. Für sehr große numerische Berechnungen über mehrere Computer eignet sich Julia aufgrund der gegebenen Unterstützung durch Bibliotheken. Im Allgemeinen erkennen wir dass Julia eine wesentlich geringere Anzahl an Softwarepaketen aufgrund der kleineren Community aufweist. Für Funktionalitäten die noch in keinem vorherig veröffentlichten Julia-Paket untergebracht wurde muss selbst Hand angelegt werden

oder auf eine andere Sprache umgestiegen werden. Das Starten eines größeren Entwicklungsprojektes wird aufgrund der Verfügbarkeit in Visual Studio Code stattfinden müssen, da restlichen Entwicklungsumgebung an denen aktuell entwickelt wird Notebooks sind. Notebooks eignen sich sehr gut für die spontane Darstellung und für ein „Proof of Concept“ von Ideen. Die allgemeine Entwicklung wird durch recht hohe Kompilierzeiten beeinflusst. Der Entwicklungsstart verzögert sich mit zunehmender Größe an Abhängigkeiten, da diese im Vorhinein erst kompiliert werden müssen. Die Suchen von Fehlern im Debug-Prozess kann situationsbedingt einen hohen zeitlichen Anspruch in Kauf nehmen. Die Auswahl an Softwarepaketen decken die betrachteten Anwendungsfälle der fraktalen Geometrie (Fraktale Kurven, Mandelbrot-Menge, Julia-Menge und 3-dimensionale Fraktale) ab. Julia als Skriptsprache liefert Schnittstellen die viele technische Details abstrahieren und Programmierer sich während des Arbeitsprozesses auf die wesentliche Entwicklung fokussieren können. Julia wird seines Versprechen ein Werkzeug für numerisches Berechnen zur Erstellung von Visualisierungen im Bereich der fraktalen Geometrie gerecht.

5. Einschränkungen

Aufgrund der eingeschränkten Zeit, die zur Erstellung dieses Artikels verfügbar ist konnte nur ein Performanz Test für die 3 Programmiersprachen Julia, Python und Java durchgeführt werden. Die Aufnahme weiterer Programmiersprachen wie die restlichen im Artikel aufgezählten Sprachen R, MATLAB und C liefert ein ausführlicheres Bild bezüglich der Vergleichbarkeit der Programmiersprachen untereinander. Die Messungen der Performanz-Test wurden auf einem fest gewählten Computer durchgeführt. Eine Durchführung auf einem weiteren Computer liefert eine zweite Menge an Messwerten die Aussagen über die Vergleichbarkeit der Messergebnisse bezüglich der verwendeten Hardware liefert. Für die Performanz der jeweiligen Sprachen erhalten wir ein zeitliches Abbild, da neuere Versionen die Performanz optimalerweise anheben oder verschlechtern können. Die Untersuchung der ausgewählten Entwicklungsumgebungen für Julia liefert auch ein temporäres Bild, da jederzeit die Entwicklung an einer Entwicklungsumgebung wieder neu gestartet werden kann oder andererseits auch zum Stillstand kommen kann. Unter anderem liefert die Auswahl der Entwicklungsumgebungen nur eine eingeschränkte Sicht auf die Realität. Die untersuchte Auswahl wurde aufgrund der Popularität der Entwicklungsumgebung getroffen, die wir keiner genauen Messung unterzogen haben. Entwicklern steht auch jederzeit die Möglichkeit offen, eine hier nicht aufgezählte Entwicklungsumgebung zu wählen. Die Einschränkungen bezüglich einer getroffenen Auswahl ist auf die ausgewählten Softwarepakete übertragbar. Es existieren Pakete innerhalb des Julia-Paketverzeichnisses, die nicht untersucht wurden, aber auch Möglichkeiten zur Visualisierung von Fraktalen liefern.

6. Ausblick

Abkürzungen

PyPi Python Package Index
PYPL PopularitY of Programming Language
CRAN Comprehensive R Archive Network
IDE Integrated Devolopment Environment
CPU Central Processing Unit
GPU Grapics Processing Unit

Literatur

[1] M. Alfonseca und A. Ortega, „Representation of fractal curves by means of L systems“, in *Proceedings of the Conference on Designing the Future*, Lancaster United Kingdom: ACM, Juni 1996, S. 13–21. DOI: 10.1145/253341.253348. (besucht am 15. 06. 2024) (siehe S. 1, 3, 4).

[2] J. Bezanson, S. Karpinski, V. Shah und A. Edelman, „Julia Language Documentation“, (siehe S. 2–4).

[3] J. W. Bezanson, „Abstraction in Technical Computing“, Diss., MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2015 (siehe S. 1).

[4] T. A. Cabutto, S. P. Heeney, S. V. Ault, G. Mao und J. Wang, „An Overview of the Julia Programming Language“, in *Proceedings of the 2018 International Conference on Computing and Big Data*, Charleston SC USA: ACM, Sep. 2018, S. 87–91. DOI: 10.1145/3277104.3277119. (besucht am 09. 06. 2024) (siehe S. 2, 4).

[5] Z. W. Cai und A. D. K. T. Lam, „A Study on Mandelbrot Sets to Generate Visual Aesthetic Fractal Patterns“, *Applied Mechanics and Materials*, Jg. 311, S. 111–116, Feb. 2013. DOI: 10.4028/www.scientific.net/AMM.311.111. (besucht am 15. 06. 2024) (siehe S. 2).

[6] P. Carbonelle, *PYPL (Popularity of Programming Language) Index*, 2023. (besucht am 12. 06. 2024) (siehe S. 3).

[7] cormullion, *Luxor.Jl Dokumentation* (siehe S. 4).

[8] *CRAN Contributed Packages*, Juni 2024. (besucht am 12. 06. 2024) (siehe S. 3).

[9] D. D'Agostino, I. Merelli, M. Aldinucci und D. Cesini, „Hardware and Software Solutions for Energy-Efficient Computing in Scientific Programming“, *Scientific Programming*, Jg. 2021, Nr. 1, S. 5 514 284, 2021. DOI: 10.1155/2021/5514284. (besucht am 16. 06. 2024) (siehe S. 3).

[10] S. Danisch und J. Krumbiegel, „Makie.jl: Flexible high-performance data visualization for Julia“, *Journal of Open Source Software*, Jg. 6, Nr. 65, S. 3349, Sep. 2021. DOI: 10.21105/joss.03349. (besucht am 09. 06. 2024) (siehe S. 4).

[11] A. Decan, T. Mens und M. Claes, „On the topology of package dependency networks: A comparison of three programming language ecosystems“, in *Proccedings of the 10th European Conference on Software Architecture Workshops*, Copenhagen Denmark: ACM, Nov. 2016, S. 1–4. DOI: 10.1145/2993412.3003382. (besucht am 13. 06. 2024) (siehe S. 3).

[12] V. Drakopoulos, N. Mimikou und T. Theoharis, „An Overview of Parallel Visualisation Methods for Mandelbrot and Julia Sets“, *Computers & Graphics*, Jg. 27, Nr. 4, S. 635–646, 2003. DOI: 10.1016/S0097-8493(03)00106-7 (siehe S. 3).

[13] C. Heiland-Allen, „Patterns in Deep Mandelbrot Zooms“, in *Algorithmic Pattern Salon*, Then Try This, 2023 (siehe S. 3).

[14] T. Januszek und M. Pleszczyński, „Comparative Analysis of the Efficiency of Julia Language against the Other Classic Programming Languages“, *Silesian Journal of Pure and Applied Mathematics*, Jg. 8, 2018 (siehe S. 2).

[15] *Julia Registries*, Apr. 2024 (siehe S. 3).

[16] *MATLAB Fileexchange*, Juni 2024. (besucht am 12. 06. 2024) (siehe S. 3).

[17] A. McAndrew, „Lindenmayer systems, fractals, and their mathematics“, (siehe S. 4).

[18] E. Mollick, „Establishing Moore's Law“, *IEEE Annals of the History of Computing*, Nr. 28, 2006. (besucht am 16. 06. 2024) (siehe S. 1).

[19] R. Pereira et al., „Energy efficiency across programming languages: How do energy, time, and memory relate?“,

in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, Vancouver BC Canada: ACM, Okt. 2017, S. 256–267. DOI: 10.1145/3136014.3136031. (besucht am 16. 06. 2024) (siehe S. 3).

[20] R. Pereira et al., *Original work in SLE'17*, https://sites.google.com/view/energy-efficiency-languages/home. (besucht am 16. 06. 2024) (siehe S. 3).

[21] J. M. Perkel, „Julia: Come for the syntax, stay for the speed", *Nature*, Jg. 572, Nr. 7767, S. 141–142, Aug. 2019. DOI: 10.1038/d41586-019-02310-3. (besucht am 14. 06. 2024) (siehe S. 1).

[22] *PyPi (Python Package Index)*, Juni 2024. (besucht am 17. 04. 2024) (siehe S. 3).

[23] G. Smith, „Fractal Geometry: History and Theory", Apr. 2011 (siehe S. 1).

[24] W. Sternemann und G. Canisianum, „Platonische Fraktale im Unterricht", (siehe S. 3).

[25] V. Walter, „Fraktale: Die geometrischen Elemente der Natur", Diplomarbeit, Karl-Franzens-Universität Graz, Graz, 2018 (siehe S. 1–3).

[26] E. Weitz, *Konkrete Mathematik (Nicht Nur) Für Informatiker*, 2. Aufl. 2021 (siehe S. 1, 2).

[27] Y. Wu, J. M. Hellerstein und A. Satyanarayan, „B2: Bridging Code and Interactive Visualization in Computational Notebooks", in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, Virtual Event USA: ACM, Okt. 2020, S. 152–165. DOI: 10 . 1145 / 3379337 . 3415851. (besucht am 16. 06. 2024) (siehe S. 4).

7. Appendix

Code-Schnipsel

Die Code-Schnipsel, sowie die erhaltenen Testdaten der Performanz-Tests lassen sich unter Gitea der Hochschule Mannheim finden.

Paketverzeichnisse

Erhalt der Daten am 08.07.2024.

1) **PyPi:** https://pypi.org/
2) **CRAN:** https://cran.r-project.org/web/packages/available_packages_by_name.html
3) **MATLAB FileExchange:** https://de.mathworks.com/matlabcentral/fileexchange/?s_tid=gn_mlc_fx_files
4) **JuliaRegistries:** https://github.com/JuliaRegistries/General/blob/master/Registry.toml

Entwicklungsumgebungen

Erhalt der Daten am 08.07.2024.

1) **Julia (VS Code):** https://github.com/julia-vscode/julia-vscode
2) **Juno (Atom):** https://github.com/JunoLab/Atom.jl
3) **Julia (IntelliJ):** https://github.com/JuliaEditorSupport/julia-intellij
4) **Julia-Studio:** https://github.com/forio/julia-studio
5) **Pluto:** https://github.com/fonsp/Pluto.jl
6) **IJulia (Jupyter):** https://github.com/JuliaLang/IJulia.jl