Julia als Werkzeug für Visualisierungen in der Fraktalen Geometrie

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

***Zusammenfassung*—Diese Arbeit untersucht, ob sich die Programmiersprache Julia zu Erstellung von Visualisierungen innerhalb der Fraktalen Geometrie eignet. Betrachtet werden die Kriterien Performanz, Nachhaltigkeit, Parallelisierbarkeit, Verfügbarkeit von Softwarepaketen und Entwicklungsumgebungen. Das Ergebnis zeigt, dass Julia seinem Ruf als performante Skriptsprache gerecht wird. Der Einsatz von Julia in einem ausgeweiteten Rahmen lässt sich parallelisieren und somit nachhaltig gestalten. Das Julia-Ecosystem bietet ausreichende Mittel um Fraktale Kurven auf einfache Weise darzustellen. Zusätzliche Softwarepakete ermöglichen die Visualisierung in 2- oder 3-dimensionaler Form. Die eingeschränkte Auswahl an Entwicklungsumgebungen und die teilweise hohe Kompilierzeit stellen ein Hindernis in der Erstellung eigener komplexer Visualisierungen dar.**

Inhaltsverzeichnis

1. Einleitung

Im Jahr 1891 beschäftigte sich der Mathematiker David Hilbert mit einer besonderen Kurve, die alle Punkte eines Quadrates mit Seitenlänge 1 durchläuft [21]. Helge von Koch entdeckte 1904 die nach ihm benannte „Kochsche Kurve“ bei der Suche nach einer nicht differenzierbaren Kurve. Diese Entdeckungen wurden von Mathematiker des 19. Jahrhunderts als „Monster“ bezeichnet, da diese seltsame Eigenschaften aufweisen [19]. Diese besonderen Objekte ordnet man im zeitlichen Verlauf der fraktalen Geometrie, einem relativ jungen Teilgebiet der Mathematik zu. Eine feste eindeutige Definition für ein Fraktal konnte sich nicht etablieren [21]. Eine Eigenschaft, die dennoch besonders häufig bei der Arbeit mit Fraktalen auftritt, ist der Begriff der *Selbstähnlichkeit*. Lässt sich ein Objekt aus kleineren Kopien seiner selbst zusammenstellen, so wird dies als *selbstähnlich* bezeichnet [21].
Ein einführendes Beispiel zur Selbstähnlichkeit ist die *Kochsche Kurve*. Diese lässt sich aus vier Kopien mit Verkleinerungsfaktor $\frac{1}{3}$ zusammenlegen [21]. Man lege dafür zwei Kopien an den linken und rechten Rand. Die beiden übrig bleibenden Kopien werden nach oben spitz aufeinander zulaufend zwischen die beiden äußeren Kopien gelegt.

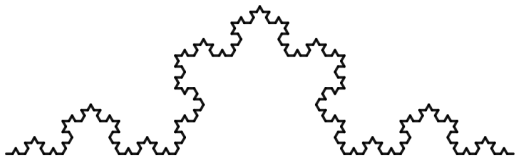

Abbildung 1. Kochsche Kurve

Weitere Grundlagenarbeit der fraktalen Geometrie beruht auf Werken des Mathematikers Gaston Julia und seinem Konkurrenten Pierre Fatou zu Beginn des 20. Jahrhunderts. Julia und Fatou erforschten das Verhalten der Iterationen von Funktionen der Form $f_c(z) = z^2 + c$ mit $z, c \in \mathbb{C}$ [21]. Diese Ergebnisse griff der Mathematiker Benoît Mandelbrot in den 1970er Jahren in seinem Buch *The Fractal Geometry of Nature* wieder auf und verlieh dem Bereich der Fraktalen Geometrie wachsende Popularität. In seinem Werk bedient sich Mandelbrot einer Vielzahl von Visualisierungen für die betrachteten Fraktale [19]. 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 Konvergenzverhalten 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 der Gaußschen Zahlenebene auffassen lassen, kann man die Mandelbrot-Menge in Form einer Rastergrafik visualisieren.

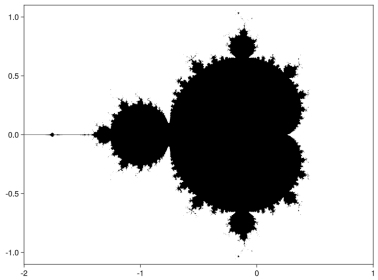

Abbildung 2. Mandelbrot-Menge

Färben wir die Elemente der komplexen Zahlenebene, die innerhalb der Mandelbrot-Menge $\mathbb{M}$ liegen schwarz und die

außerhalb liegenden weiß, erhalten wir Abbildung 2. Mithilfe der bildlichen 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 veranschaulichen konnten [21]. Im Laufe der Zeit wächst die Rechenleistung der verfügbaren Computer stetig und erschließt neue Möglichkeiten innerhalb der Forschung. Nach *Moore's Law* [15] verdoppelt sich die Anzahl der Komponenten, die auf einen Chip passen, jedes Jahr. Innerhalb der 1970er Jahre wurde die Mandelbrotmenge mithilfe von ASCII-Art visualisiert [22]. Die momentane Rechenleistung ermöglicht mittels hochauflösender Rastergrafiken tiefere Einblicke in die Welt der Fraktale. Die resultierenden Bilder wecken aufgrund der Ästhetik auch das Interesse vieler Nicht-Mathematiker sich mit dem Themengebiet zu befassen [19]. 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. Um sich die Rechenleistung der Computer zunutze zu machen, dient eine Programmiersprache als Schnittstelle zwischen Computer und Mensch. Wir betrachten genauer, ob sich die Programmiersprache Julia zur Erstellung von Visualisierungen im Rahmen der Fraktalen Geometrie eignet.

2. Hintergrund

Für das Lösen technischer Probleme ist ein meist genutzter Ansatz zwei Programmiersprachen 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 weiterer Ansatz zur Lösung dieses Problems ist, dass vorerst Algorithmen innerhalb von Skriptsprachen geschrieben werden und diese später in hardware-nahe Sprachen übertragen werden. Dieses Vorgehen ist mit einem hohen zeitlichen Aufwand und einer hohen Fehleranfälligkeit während des Übertragungsprozesses verbunden [18]. Julia wird als eine mögliche Lösung für dieses „Zwei Sprachen Problem" angesehen. Ein Team aus unabhängigen Entwicklern startete 2009 die Entwicklung von Julia. Bis zur ersten Veröffentlichung von Julia vergingen drei weitere Jahre. Im August 2018 wurde die Veröffentlichung von Julia 1.0 bekanntgegeben [13]. Julia ist eine dynamische Programmiersprache, in der Variablen zwingend definiert werden müssen, bevor diese zum Einsatz kommen [4]. Eine Vielzahl an Konzepten aus imperativen, funktionalen und objektorientierten Programmiersprachen lassen sich in Julia wiederfinden. Julia wurde für wissenschaftliche Berechnungen konzipiert, ermöglicht aber auch allgemeine Programmierung [2]. Wichtige Merkmale, die Julia von anderen dynamischen Programmiersprachen abgrenzt, sind nach Julia-Handbuch [2]:

- 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

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 der Mathematik, im Besonderen als Werkzeug für die Fraktale Geometrie.

3. Kriterien

Im folgenden Abschnitt betrachten wir ausgewählte 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 weitere, die speziell für Visualisierung innerhalb der Fraktalen Geometrie von hoher Signifikanz sind. Wir evaluieren im Laufe der nächsten Abschnitte, die Relevanz der ausgewählten Kriterien für unsere Fragestellung.

- Performanz
- Nachhaltigkeit
- Parallelisierbarkeit
- Verfügbarkeit von Softwarepaketen
- Entwicklungsumgebungen

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

3.1. Performanz

Für die Gestaltung interaktiver Software und die Erstellung von Bildern bzw. von Animationen bezüglich Fraktalen wird ein Anspruch auf eine schnelle Verarbeitung von Eingaben gesetzt. Besonders für Programme, mit denen in Echtzeit verschiedene Formen von Fraktalen erkundet werden können, ist eine gute Performanz von hoher Relevanz. Viele Programmiersprachen stellen heutzutage alle notwendigen Funktionalitäten zur Lösung unterschiedlicher Problemtypen bereit. Demnach spielt die Effizienz eine höhere Rolle innerhalb der Auswahl der Sprache, als die Umsetzbarkeit [13]
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 16 GB verfügbarem 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 Optimierungen einzugehen, die von genauerem 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 einer Funktion wird zehn Mal durchgeführt. Aus den zehn 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. Während der Arbeit mit Julia fällt auf, dass Kompilierzeiten in der Länge stark variieren können.

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 [22]. 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 bestehend aus zwei 64-Bit Fließkommazahlen nach IEEE Spezifikation IEEE-754. Die Mandelbrot-Menge ist nach Definition Teilmenge der komplexen Zahlen. Es lässt sich zeigen, dass alle Elemente der Mandelbrot-Menge vollständig in der Kreisscheibe um $0 + 0i$ mit Radius 2 liegen [21]. Diese Eigenschaft machen wir uns in unserem Näherungsbild zunutze. Demnach verwenden wir für die Visualisierung der Mandelbrot-Menge als Rastergrafik einen quadratischen Bildausschnitt mit Mittelpunkt bei $0 + 0i$ und einer Seitenlänge von 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 wird. Somit wird die Kreisscheibe nicht verlassen. 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 ← z^2 + 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 Mandelbrot-Menge sind. Abbildung 3 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.

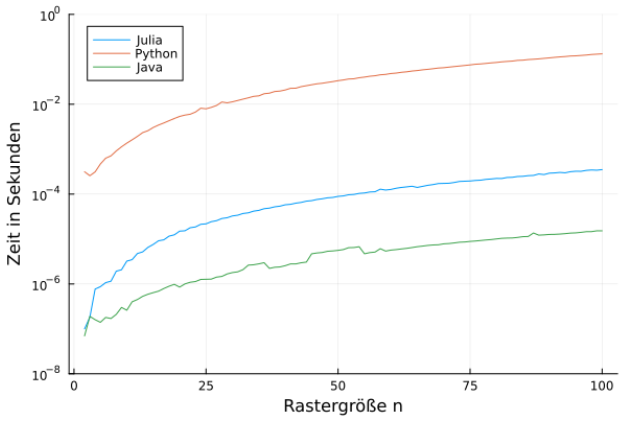

Abbildung 3. Zeitaufwand in Abhängigkeit der Rastergröße

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 wesentlich länger. Unter den gemessenen Sprachen ist Java die Schnellste, darauf folgt Julia. Das Schlusslicht bildet Python.

3.2. Nachhaltigkeit

Im letzten Jahrhundert war der Energieverbrauch unterschiedlicher Programmiersprachen nahezu ähnlich. Innerhalb des 20. Jahrhunderts legen Computerhersteller und Softwareentwickler einen größeren Wert auf eine effizientere Energienutzung [16]. Neben dem Energieaufwand für Berechnungen wird weitere Energie für Kühlung, Anzeige, Lagerung und Kommunikation von Hardware verwendet [8]. Bedeutet die überdurchschnittliche Performanz, dass Julia auch energieeffizienter ist? Der Energieverbrauch wird durch die Gleichung:

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

beschrieben [16]. Visualisierungen innerhalb der Fraktalen Geometrie können einen hohen Rechenaufwand in Anspruch nehmen. Beispielsweise spielt die numerische Präzision in der Berechnung eines kleinen „Sichtfensters“ in der Mandelbrot-Menge einen limitierenden Faktor. Je kleiner das Sichtfenster, desto mehr Zeit ist für die Berechnung notwendig [12]. Durch Verwendung von Zahlen mit arbiträrer Präzision kann diese Rechendauer, die einer 64-Bit Fließkommazahl übersteigen. Ein weiterer Anwendungsfall ist die Wahl eines hohen t_{max} innerhalb des *Escape Time Algorithmus* 1, welche ein genaueres Bild der Mandelbrot-Menge liefert. Daraus resultiert eine längere Rechenzeit. Für die Darstellung von Kurven, wie beispielsweise der Koch-Kurve n-ter Iteration, wächst der Rechenaufwand mit größerem n an. Daraus folgt, dass präzisere Berechnungen einen höheren Energieaufwand benötigen. Pereira et al. vergleichen die Energieeffizienz von Programmiersprachen und stellen fest, dass kompilierte Sprachen im Durchschnitt schneller sind als Sprachen mit virtueller Maschine, gefolgt von interpretierten Sprachen [16]. In dieser Veröffentlichung fehlt allerdings Julia. In einer späteren Ergänzung zur Arbeit ist Julia in der Version 1.3.1 zu finden und wird mit neun weiteren Programmiersprachen verglichen. Julia schneidet im neuen Vergleich mit Platz zwei von zehn ab. [17]

3.3. Parallelisierbarkeit

Die Berechnung einer grafischen Darstellung der Mandelbrot-Menge und einer Julia-Menge lässt sich auf simple Weise parallelisieren. Der Rechenaufwand für eine Rastergrafik kann in mehrere unabhängige Teilprobleme, die Berechnung der jeweiligen Pixel, aufgeteilt werden. Dies ist von hohem Nutzen, da die Berechnungskosten einzelner Pixel mit hohem Rechenaufwand verbunden ist [11]. 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. Diese greifen auf denselben Speicher zu.
3) **Verteiltes Berechnen**: Mithilfe des verteilten Berechnens können mehrere Julia-Prozesse mit jeweils eigenem Speicher durchgeführt werden. 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 die einfachste Option, 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 wird durch Julia von Haus aus gewährleistet.

3.4. Softwarepakete

Im folgenden 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. Zunächst wird das Julia Ecosystem auf allgemeiner Ebene evaluiert. Im weiteren Verlauf werden einzelne Softwarepakete als Hilfsmittel für die Fraktale Geometrie bewertet. Wir überprüfen ob, eine feste Auswahl von Fraktalen mithilfe von Julia visualisiert werden kann. Des weiteren betrachten wir die Umsetzungsstrategien der Softwarepakete. Wir unterteilen dabei die Fraktale in folgende Mengen, die nicht zwingend disjunkt zueinander sind:

- Fraktale, als Teilmenge der komplexen Zahlenebene. Beispiele dafür sind die Mandelbrot-Menge, sowie die Julia-Menge [21].
- Fraktale, die sich mithilfe eines Lindenmayer-Systems darstellen lassen, beispielsweise die Hilbert-Kurve oder die Kochsche Schneeflocke [1]
- Fraktale innerhalb des 3-dimensionalen Raums, wie beispielsweise der Menger-Schwamm oder die Sierpinski-Pyramide [20]

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 beispielsweise der Statistik. Um die Pakete der Öffentlichkeit zur Verfügung zu stellen, können diese in Paketverzeichnisse im Internet hochgeladen werden. Diese Möglichkeit erweitert das traditionelle Verständnis von Softwareentwicklung im Sinne einer Zusammenarbeit von Entwicklern unabhängig ihres Standorts [10].Tabelle 1 zeigt eine Auswahl von Verzeichnissen für Sprachen aus dem Bereich des wissenschaftlichen Berechnens, basierend auf [6].

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

Nach dem PopularitY of Programming Language (PYPL) Index wird Python als populärste Programmiersprache evaluiert. Julia hingegen belegt Platz 24 [6]. Zwischen Python und Julia liegen die Programmiersprachen R und MATLAB. Da MATLAB keinen klassischen Paketmanager besitzt, wird hier zum Vergleich die Anzahl der von Nutzern bereit gestellten Programme auf dem MATLAB FileExchange verwendet. Aufallend ist, dass MATLAB trotz einer proprietären Lizenz eine höhere Anzahl an Paketen bzw. Programmen als Julia bereitstellt. Die hohe Popularität von Python erklärt, weshalb die Anzahl zwischen Paketen auf dem Python Package Index (PyPi), dem Python Package Index, und *JuliaRegistry* sich dem Verhältnis 50:1 nähert. Aufgrund des relativ jungen Alters von Julia (veröffentlicht 2012) [4] und ihrer geringeren Popularität, besitzt Julia im Vergleich zu den übrigen 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. Der Schildkröte stehen nur eine beschränkte Anzahl an Aktionen zur Verfügung. Die Handlungsmöglichkeiten erfolgen aus der Perspektive der Schildkröte. [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

Mit einer kleinen Auswahl an Aktionen können bereits fraktale Kurven gezeichnet werden. In Julia wird das Zeichnen von Turtle- Grafiken durch das Paket `Luxor.jl` abgedeckt. Neben unserem Beispiel von drei Aktionen stellt `Luxor.jl` insgesamt 17 Handlungsmöglichkeiten zur Bewegung der Schildkröte bereit [7].

Mithilfe der Regeln der Turtle-Grafiken lassen sich Lindenmayer Systeme, kurz L-Systeme, darstellen. Ein L-System liefert eine Liste an Ersetzungsregeln, die alle gleichzeitig auf eine Zeichenkette angewendet werden [14]. Diese Ersetzungsregeln können beispielsweise auf Zeichenketten bzw. die Aktionen unserer Schildkröte, bestehend aus F, +, - angewendet werden. Das n-fache Anwenden der festgelegten Ersetzungsregeln liefert uns eine Näherung der n-ten Iteration des Fraktals. Alle hier gezeigten Kurven werden in vierter Iteration dargestellt. Der verbleibende Aufwand besteht nur noch darin, das L-System in die Aktionen einer Turtle-Grafik zu übersetzen. Das Softwarepaket `Lindenmayer.jl` abstrahiert diesen Arbeitsaufwand. Verwenden wir drei Kopien der *Kochschen Kurve* und legen diese zu einem Dreieck zusammen, so erhalten wir die *Kochsche Schneeflocke* [21]. Auch die erstmalig erwähnte Hilbert-Kurve können wir visualisieren. Beide Grafiken werden mithilfe von `Lindenmayer.jl` erstellt.

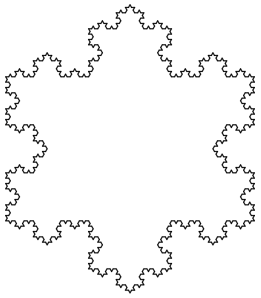

Abbildung 4. Koch Schneeflocke

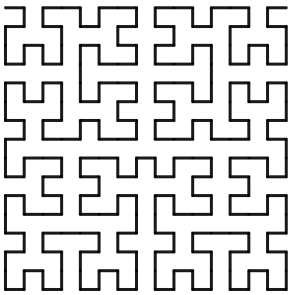

Abbildung 5. Hilbert Kurve

Fraktale in $\mathbb{C}$. Die Mandelbrot-Menge und Julia-Mengen liegen innerhalb der komplexen Zahlen. Mithilfe der Strategie die Mandelbrot-Menge in Form eines Rasters unter Nutzung des *Escape Time Algorithmus* 1 darzustellen, 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*: `Images.jl` realisiert. Das Paket `Makie.jl` liefert eine Darstellungsmöglichkeit für 2-dimensionale Rastergrafiken in Form interaktiver Anwendungen mithilfe von Benutzereingaben wie beispielsweise Textboxen, Knöpfe, Slider [9]. Die kolorierten Visualisierungen der Mandelbrot-Menge und einer konkreten Julia-Menge wurden mithilfe von `Makie.jl` erstellt:

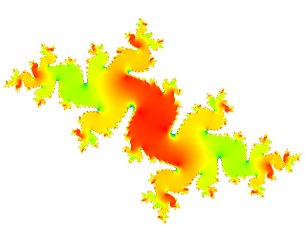

Abbildung 6. Julia-Menge

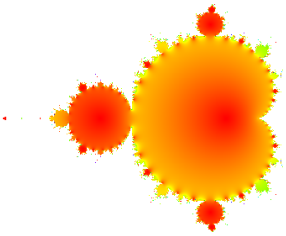

Abbildung 7. Mandelbrot-Menge

Fraktale im 3-dimensionalen Raum. Neben der Möglichkeit 2-dimensionale Rastergrafiken zu erzeugen, bietet `Makie.jl` die Option, innerhalb eines 3-dimensionalen Raumes Objekte darzustellen. Unter Verwendung eines Voxel-Systems können fraktale Objekte mit Würfeln angenähert werden. Neben Würfeln bieten Polygone und Linien weitere Grundbausteine für die Erstellung komplexer Objekte [9]. Zusätzliche Möglichkeiten zur Erstellung von 3-dimensionalen Objekten sind das Angeben eines Volumens oder das Laden von `.obj`-Dateien. Die Konstruktion größerer Objekten stellt sich aufgrund der zusätzlichen dritten Dimension als schwierigeres Unterfangen dar [9].

3.5. Entwicklungsumgebungen

Die Julia Programmiersprache ist auf macOS, Windows und Linux frei erhältlich [2]. 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. Entwicklungsumgebungen (Integrated Devolopment Environment (IDE)) lassen sich in drei unterschiedliche Typen kategorisieren:

- Eigenständiger Texteditor, der für das Schreiben einer spezifischen Programmiersprache oder einer engen Auswahl an zusammenhängenden Programmiersprachen konzipierte wurde. Ein Beispiel dafür ist RStudio für R oder IntelliJ für Java.
- Erweiterung eines schon existierenden Texteditors, der konzipiert wurde, um eine große Breite an Sprachen abzudecken. Visual Studio Code ist ein Beispiel für solch einen Texteditor.
- Notebooks, die eine Ansammlung an Zellen mit ausführbarem 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 [23]. In der Praxis unterstützen Notebooks meist mehrere Sprachen oder sie können per Erweiterung hinzugefügt werden.

Wir klassifizieren eine Auswahl an Entwicklungsumgebungen für Julia. Für jede Entwicklungsumgebung betrachten wir, ob deren Entwicklung noch aktiv erfolgt. 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

Tabelle 2 zeigt, dass die Unterstützung von Julia innerhalb von Notebooks weiterhin gewährleistet ist. Das bekannteste Beispiel ist das *Jupyter* Notebook, dessen Name sich aus den Programmiersprachen Julia, Python und R zusammensetzt [18]. 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, dass die Programmiersprache Julia, sowie auch alle anderen der genannten Entwicklungsumgebungen aus Tabelle 2, keiner proprietären Lizenz unterworfen sind. Dies resultiert in einer hohen Zugänglichkeit für Nutzer.

4. Ergebnis

Im Gesamtbild liegen die Stärken der Programmiersprache Julia in ihrer überdurchschnittlichen Geschwindigkeit als Skriptsprache. Die Kombination mit einer guten Energieeffizienz trägt dazu bei, dass Julia als „grüne“ Programmiersprache angesehen werden kann. Für umfangreich 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 einer kleineren Community aufweist. Für Funktionalitäten, die noch in keinem vorherig veröffentlichten Julia-Paket untergebracht wird, 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. Alle anderen Entwicklungsumgebungen, an denen aktiv entwickelt wird, sind Notebooks. Notebooks eignen sich sehr gut für eine 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 Vorfeld erst kompiliert werden müssen. Das Suchen von Fehlern im Debug-Prozess kann situationsbedingt einen hohen zeitlichen Aufwand in Anspruch nehmen. Die Auswahl an Softwarepaketen deckt die betrachteten Anwendungsfälle der Fraktalen Geometrie (fraktale Kurven, Mandelbrot-Menge, Julia-Menge und Fraktale im 3-dimensionalen Raum) ab. Julia als Skriptsprache liefert Schnittstellen, die viele technische Details abstrahieren. Programmierer können sich während des Arbeitsprozesses auf die wesentliche Entwicklung fokussieren. Julia wird seines Versprechens, ein Werkzeug für numerische Berechnungen, zur Erstellung von Visualisierungen im Bereich der fraktalen Geometrie, gerecht.

Aufgrund der eingeschränkten Zeit, die zur Erstellung dieses Artikels verfügbar war konnte nur ein Performanz Test für die drei Programmiersprachen Julia, Python und Java durchgeführt werden. Die Aufnahme weiterer Programmiersprachen, wie die übrigen im Artikel aufgezählten Sprachen R, MATLAB und C, liefert ein ausführlicheres Bild bezüglich der Vergleichbarkeit der Programmiersprachen untereinander. Die Messung des Performanz-Tests wurde auf einem fest gewählten Computer durchgeführt. Eine Durchführung auf einem weiteren Rechner 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 verringern können. Die Untersuchung der ausgewählten Entwicklungsumgebungen für Julia liefert auch ein temporäres Bild, da jederzeit die Entwicklung an einer Entwicklungsumgebung erneut gestartet werden kann oder im Gegensatz zum Stillstand kommen kann. Unter anderem liefert die Auswahl der Entwicklungsumgebungen 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 ebenfalls Möglichkeiten zur Visualisierung von Fraktalen liefern.

5. Ausblick

Um die Popularität der Programmiersprache Julia zu steigern muss Nutzern gezeigt werden, wie sie die beste Performanz

mit Julia erreichen können. Dieses Ziel kann mittels einer stetig guten bleibenden Dokumentation erreicht werden [4]. Eine größere Nutzerbasis sorgt zudem für eine Weiterentwicklung bzw. Neuentwicklung von Werkzeugen und Paketen für das Julia Ecosystem. Eine größere Auswahl an Werkzeugen gestaltet den Einstieg in Julia zunehmend attraktiver.

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. 4).

[2] J. Bezanson, S. Karpinski, V. Shah und A. Edelman, „Julia Language Documentation“, (siehe S. 2, 3, 5).

[3] J. W. Bezanson, „Abstraction in Technical Computing“, Diss., MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2015 (siehe S. 2).

[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, 6).

[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. 3).

[6] P. Carbonelle, *PYPL (Popularity of Programming Language) Index*, 2023. (besucht am 12. 07. 2024) (siehe S. 4).

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

[8] 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).

[9] 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, 5).

[10] 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. 4).

[11] 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).

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

[13] 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).

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

[15] E. Mollick, „Establishing Moore's Law“, *IEEE Annals of the History of Computing*, Nr. 28, 2006. (besucht am 16. 06. 2024) (siehe S. 2).

[16] 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).

[17] 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).

[18] 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. 2, 5).

[19] G. Smith, „Fractal Geometry: History and Theory“, Apr. 2011 (siehe S. 1, 2).

[20] W. Sternemann und G. Canisianum, „Platonische Fraktale im Unterricht“, (siehe S. 4).

[21] V. Walter, „Fraktale: Die geometrischen Elemente der Natur“, Diplomarbeit, Karl-Franzens-Universität Graz, Graz, 2018 (siehe S. 1, 2, 4).

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

[23] 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. 5).

Anhang

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