FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachochschule Aachen, Campus Jülich

Fachbereich 09 - Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik

Seminararbeit

Numerische Methoden für gewöhnliche Differentialgleichungen in Julia

Autor:
Jonas Böcker, 3515471

Betreuer:
Prof. Dr. Martin Reißel
Prof. Dr. Matthias Grajewski

Aachen, 18. Januar 2024

Zusammenfassung

Im Rahmen dieser Seminararbeit wurde sich näher mit den Möglichkeiten auseinander gesetzt gewöhnliche Differenzialgleichungen in der Programmiersprache Julia automatisch lösen zu lassen. Dies stand vor allem stets im Vergleich mit Paketen der Sprache Python, welche Lösealgorithmen für die selbe Art von Problemen zur Verfügung stellen.
Dafür wurde sich zunächst die Programmiersprache Julia genauer untersucht und die Ziele der Sprachentwickler im Hinblick auf die Umsetzung der Sprache betrachtet. Es wurde in dieser Hinsicht herausgefunden, dass Julia direkt entwickelt wurde, um Programmiersprachen mit einem Interpreter anstatt eines Compilers, wie Python oder R in der täglichen Arbeit von Wisschenschaftlern zu ersetzen. Dies soll vor allem durch die Implementierung einer vergleichbar komfortablen Syntax und Nutzbarkeit erreicht werden, wobei der dadurch geschriebene Code allerdings durch die Nutzung eines Just-In-Time Compilers eine, im Vergleich zu interpretierten Sprachen wie Python, deutlich schnellere Laufzeit bieten kann. Aufgrund dessen hat sich Julia bereits bei vielen Experten der Domäne als bevorzugte Programmiersprache durchgesetzt und ein weit gespanntes Netzwerk an gratis herunter ladbaren Packages ist entstanden, welche die Entwicklung in jeglichen Bereichen der Wissenschaft zu Erleichtern versuchen.
Anschließend werden mit dem Paket DifferentialEquations.jl das erste Paket zum Lösen von Differenzialgleichungen in Julia betrachtet. Dieses bietet eine übersichtliche Syntax beim Definieren und Lösen von Problemen und lieferte in verschiedenen Tests numerische Lösungen zu gewöhnlichen Differenzialgleichungen, welche bei einer Laufzeit von wenigen Nanosekunden mit dem bloßen Auge nicht von der exakten Lösung zu unterscheiden sind.
In einem nächsten Schritt wurde ebenfalls mit SciPy das meistgenutzte Paket zum Lösen von Differenzialgleichungen in Python untersucht. Es stellte sich heraus, dass unter der Nutzung einer vergeleichbaren Syntax zum Erstellen der Gleichungen in etwa die zehnfache Zeit gebraucht wird, um vergleichbar gute Ergebnisse zu erhalten wie unter Nutzung des DifferentialEquations-Pakets in Julia.
In einem letzten Schritt wurde noch die Möglichkeit betrachtet unter Nutzung der GPU die Laufzeit des Julia-Codes zu Beschleunigen. Hier stellte sich heraus, dass es bei einer geringen Menge an Problemen möglich ist, die Laufzeit der Lösealgorithmen erneut um einen Faktor von zehn zu verringern.

Inhaltsverzeichnis

1 Eigenhändigkeitserklärung **4**

2 Einführung in Julia **5**
2.1 Erste Schritte mit Julia 5
2.2 Typsystem 5
2.3 Ökosystem 9
2.4 Wissenschaftliche Anwendungsbereiche 10

3 Differenzialgleichungen in Julia **12**
3.1 DifferentialEquations.jl 12
3.2 Definieren eines Problems 13
3.3 Lösen der Differenzialgleichung 14
3.4 Beispiele 15

4 Vergleich mit Solvern des Python-Universums **21**
4.1 Definieren des Problems 21
4.2 Lösen der Differenzialgleichung 22
4.3 Beispiele 23
4.4 Vergleich zur Julia-Sprache 27

5 Laufzeitbeschleunigung durch Nutzung der GPU **28**

6 Fazit **30**

7 Quellenverzeichnis **31**

1 Eigenhändigkeitserklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema "Numerische Methoden für gewöhnliche Differentialgleichungen in Julia“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war. Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Autor:	Jonas Böcker
Matrikelnummer:	3515471
Datum:	18. Januar 2024
Ort:	Aachen

Unterschrift

2 Einführung in Julia

In der Welt des wissenschaftlichen Programmierens ist Performance oft von höchster Wichtigkeit, weswegen effiziente und hochperformante Sprachen wie C, C++ oder Fortran für diese Anwendungsfälle die idealen Tools sein sollten. Komplexe Berechnungen und realistische Simulationen erfordern häufig große Berechnungsgeschwindigkeit und hohe Rechenleistung, was diese Sprachen standardmäßig anbieten. Allerdings zeigt die Realität, dass Domänenexperten doch häufiger komfortablere Alternativen wie Python oder R für ihre alltägliche Arbeit nutzen. Diese sind meist zwar in der Laufzeit der Programme deutlich langsamer, da es sich bei diesen zumeist um interpretierte Sprachen handelt, als kompilierte Sprachen, allerdings liefern sie durch ihr dynamisches Typsystem und einfache Syntax für eine tägliche Nutzung mehr Vorteile liefern. Die Erfinder und Entwickler der Programmiersprache Julia versuchen diesen Konflikt der Interessen zwischen zeitkritischen Anwendungen und komfortabler Nutzung zu überbrücken und eine dynamische und flexible Programmiererfahrung zu bieten, welche allerdings, unter anderem durch Nutzung eines Just-In-Time-Compilers anstatt eines Interpreters, trotzdem vergleichbare Geschwindigkeiten wie C erreichen kann.

2.1 Erste Schritte mit Julia

Um die Entwicklung mit Julia beginnen zu können, muss zunächst die Sprache selbst von der offiziellen Website der Sprache[1] heruntergeladen und anschließend installiert werden. Diese ist zum Zeitpunkt des Schreibens für Windows, macOS und verschiedene Linux Versionen verfügbar. Nach der Installation kann entweder interaktiv im eigenen Julia-Terminal programmiert werden, oder in Julia-Skriptdateien, welche die Dateiendung `.jl` besitzen. Diese Skripte können anschließend direkt innerhalb einer IDE ausgeführt werden, oder im Julia-Terminal durch Angabe des absoluten Pfades des Files eingebunden werden, wodurch auf alle im Skript erstellten Variablen und Methoden zugegriffen werden kann. Auf der offiziellen Website von Julia werden unter anderem Visual Studio Code, Jupyter und Vim als mögliche Entwicklungsumgebungen vorgestellt, welche die Entwicklung und Codeausführung erleichtern. Im Rahmen dieser Arbeit wurde Julia-Code hauptsächlich in Jupyter Notebooks[2] in der IDE Visual Studio Code[3] entwickelt, wobei zusätzlich eine IDE-Extension[4] heruntergeladen werden muss, um die Entwicklung von Julia-Code zu ermöglichen. In Jupyter Notebooks können ebenfalls problemlos Julia Programmzeilen eingefügt werden, sodass im Folgenden unterstützende Codebeispiele erstellt werden konnten.

2.2 Typsystem

Eines der Kernfunktionen Julia ist das dynamische Typsystem und die damit scheinbar typenlose Entwicklung von Programmcode. Im Gegensatz zu vielen anderen kom-

[1] https://julialang.org/
[2] https://jupyter.org/
[3] https://code.visualstudio.com/
[4] https://marketplace.visualstudio.com/items?itemName=julialang.language-julia

pilierten Sprachen, wie beispielsweise der oben genannten Sprachen C und C++, implementiert Julia ein ähnlich dynamisches Typsystem für Variablen und Argumenten von Funktionen wie Python, MatLab und R. Dabei wird der Typ einer Variablen erst zur Laufzeit berechnet und kann durch den Programmierer nach Belieben verändert werden. Dadurch können viele Programme auch ganz ohne die explizite Angabe von Datentypen auskommen. Intern besitzt Julia jedoch trotzdem ein festes Typsystem, welches jeder Variable stets einen festen Typen zuordnet, welcher, wie in unten gezeigtem Codebeispiel demonstriert, vom Programmierer abgefragt werden kann. Genauso funktioniert es auch bei Funktionen, wobei die Typen von Übergabeparametern und Rückgabewerten nicht eindeutig festgelegt werden müssen und somit eine Funktion mit beliebig getypten Variablen aufgerufen werden kann und anschließend auch Werte von beliebigen Datentypen zurückgeben kann.

```julia
# Der Variable x wird zunächst der Integer-Wert 42 zugewiesen.
x = 42

# Anschließend wird dieser ausgelesen. Außerdem wird überprüft welchen Typen
# die Variable x besitzt
println("Variable x is assigned to value \"$x\" of type $(typeof(x))\n")

# Im zweiten Schritt wird der Variable x den Wert "Hello World" zugewiesen
x = "Hello World"

# Erneut wird hier x ausgegeben und der Typ der Variablen ünerprüft.
println("Variable x is assigned to value \"$x\" of type $(typeof(x))\n")

# Eine Methode namens f wird erstellt, welche ein Argument von beliebigem
# Typen mit dem Namen x übergeben bekommt.
# Es wird der Wert und der Typ von x ausgegeben und falls es sich bei x um
# eine Zahl handelt und diese größer als 5 ist, wird der String-Wert "Hi"
# von der Methode zurück gebeben.
# Ansonsten wird der Boolean false zurück gegeben.
function f(x)
    println("Argument x is \"$x\" of type $(typeof(x))")
    if (x > 5)
        return "Hi"
    else
        return false
    end
end

# Die Metode f wird zweimal aufgerufen, einmal mit dem Integer 4 und einmal
# mit der Floating-Point Zahl 10.5. Nach beidem Aufrufen wird der Rückgabewert
# der Methode abgespeichert und von diesem der Wert und dessen Typ ausgegeben.
x = f(4)
println("Function f returns \"$x\" of type $(typeof(x))\n")

x = f(10.5)
println("Function f returns \"$x\" of type $(typeof(x))")
```

```
Variable x is assigned to value "42" of type Int64

Variable x is assigned to value "Hello World" of type String

Argument x is "4" of type Int64
Function f returns "false" of type Bool

Argument x is "10.5" of type Float64
Function f returns "Hi" of type String
```

Abbildung 2.1: Typsystem von Julia

Programmierer können allerdings den Typ von Variablen und Funktionsparametern auch explizit festlegen, um zum Beispiel den Aufruf einer Funktion nur mit bestimmten Datentypen zu erlauben. Ebenso können Funktion überladen werden, welche denselben Namen allerdings verschiedene Parametertypen besitzen. Bei jedem Funktionsaufruf wird stets die spezifischste Methode gesucht und anschließend mit den gegebenen Werten ausgeführt. Dabei werden bei Funktionen mit mehreren Parametern stets der Datentyp aller Argumente beachtet, um die genaueste Funktion zu finden, welche dann aufgerufen wird. Sollte es dazu kommen, dass zwar anhand der Argumenttypen mehrere passende Funktionen existieren, allerdings die Wahl der spezifischsten Funktion nicht eindeutig ist, wird eine Exception geworden. Dies kann durch Definition weiterer Funktionen gelöst werden, welche für diese exakten Parametertypen eine neue spezifischste Funktion implementiert.

```julia
# 1.
function f(x, y)
     println("Hello World!")
end

# 2.
function f(x::Int64, y)
    println("Hello Sun!")
end

# 3.
function f(x, y::Float64)
    println("Hello Moon!")
end

# Aufruf der Funktion f mit einem Parameter vom Typen String und einem vom Typen
# Boolean ruft die allgemeinste Version 1 der Methode auf, wodurch "Hello World"
# ausgegeben wird.
f(1.23, "Bar")

# Bei diesem Aufruf wird die genauere Funktion 2 gefunden, welche den Datentypen
# des ersten Arguments x auf einen Integer festlegt. Dadurch wird "Hello Sun"
# ausgegeben.
f(42, false)

# Dieser Aufruf von f führt zu einer Exception, da die Funktionen 2 und 3 gleich
# spezifisch sind und damit nicht entschieden werden kann, welche Methode
# aufgerufen werden sollte.
# Dies geschieht obwohl alle angegebenen Funktionen passende Signatur für die
# gegebenen Argumente besitzen, allerdings keine Spezifischste festlegt werden
# kann.
# Durch Erstellen einer weiteren Methode f mit den Parametern
# (x::Int64, y::Float64) kann dieses Problem gelöst werden und da nun diese als
# eindeutig Spezifischste gefunden wird.
try
    f(42, 3.14)
catch e
    showerror(stdout, e)
end
```

```
Hello World!
Hello Sun!
MethodError: f(::Int64, ::Float64) is ambiguous.

Candidates:
  f(x, y::Float64)
  f(x::Int64, y)

Possible fix, define
  f(::Int64, ::Float64)
```

Abbildung 2.2: Methodendispatching in Julia

Im obigen Codebeispiel wird die selbe Methode dreimal geschrieben und mit verschiedenen Argumenttypen überladen. In der ersten Implementierung ist es möglich Methode mit zwei beliebigen Argumenten aufzurufen, in der Zweiten wird der Typ des ersten Arguments auf einen Integerwert festgelegt und in der dritten Implementierung ist der Typ des ersten Arguments zwar wieder frei wählbar, allerdings wird der Typ der

zweiten Argumentes auf den einer Float-Variable festgelegt. Bei einem Aufruf der Methode im weiteren Programmcode werden zunächst die Typen der Übergabeparameter überprüft, um die passende Methode zum Aufruf zu finden. Im ersten Beispiel dafür in der obigen Abbildung sind diese im ersten Argument eine Float-Variable und im zweiten Argument eine String-Variable. Da keine Methode geschrieben wurde, die passendere Argumente festlegt als die allgemeinste Form der Funktion, wird diese ausgeführt. Ähnliches passiert beim zweiten Funktionsaufruf mit der Übergabe eines Integer-Wertes und eines Boolean-Wertes. In diesem Fall jedoch wird die zweite Implementierung der Methode mit genaueren, passenden Argumenttypen gefunden, weswegen nun stattdessen diese ausgeführt wird. Bei dem dritten Aufruf der Methode kommt es jedoch zu einem Konflikt. Bei der Übergabe einer Integer-Variable und einer Float-Variable gibt es in der zweiten und in der dritten Implementierung der Funktion zwei Methode, welche genauer festgelegte Argumenttypen besitzen. Allerdings lässt sich hier keine genaueste Methode festlegen, da sowohl die zweite als auch die dritte Implementierung der Funktion genauere Argumenttypen festlegen, welche auf den Aufruf passen. Dies lässt sich zum Beispiel durch die Definition einer weiteren Überladung der Methode lösen, welche den Typen der Argumente weiter einschränkt.

2.3 Ökosystem

Julia hat sich seit der Veröffentlichung in 2012 als Standardsprache in vielen Bereichen der Softwareentwicklung durchgesetzt. Dadurch hat sich ein ausgedehntes Netz an frei verfügbaren Bibliotheken entwickelt, um verschiedenste Arbeiten so einfach wie möglich zu machen. Besonders nennenswert sind zum Beispiel verschiedene Pakete zur Datenvisualisierung und Zeichnung von Graphen. In der Vergangenheit mussten Entwickler dieser Packages meist eine Balance finden zwischen Features, Geschwindigkeit, einfacher Nutzung und Schönheit der erzeugten Diagramme. Julia-Bibliotheken wie `Plots.jl`, `Makie.jl`, oder `Gadfly.jl` bieten Interfaces an, um mit vergleichsweise wenig Code komplexe Grafiken und Animationen zu erstellen.

```julia
using Plots

# Hier ist ein sehr einfaches Beispiel der Nutzung der Plots Library in Julia Code.
# Zunächst wird eine Liste x erstellt mit den Zahlen von 0 bis 10 mit einer
# Schrittweite von 0.01. Anschließend wird die Sinus- bzw. Cosinus-Funktion auf jedes
# Element dieser Liste angewendet und in einer weiteren Liste abgespeichert.
x = range(0, 10, step=0.01)
y1 = sin.(x)
y2 = cos.(x)

# Mit der plot-Funktion können nun die Wertepaare gezeichnet werden. Über weitere
# Übergabeparameter der Funktion kann der Grafik noch eine Überschrift gegeben werden,
# sowie einer Legende zu den einzelnen Graphen.
plot(x, [y1, y2], title="Trigonometrische Funktionen", label=["sin(x)" "cos(x)"], linewidth=2)
```

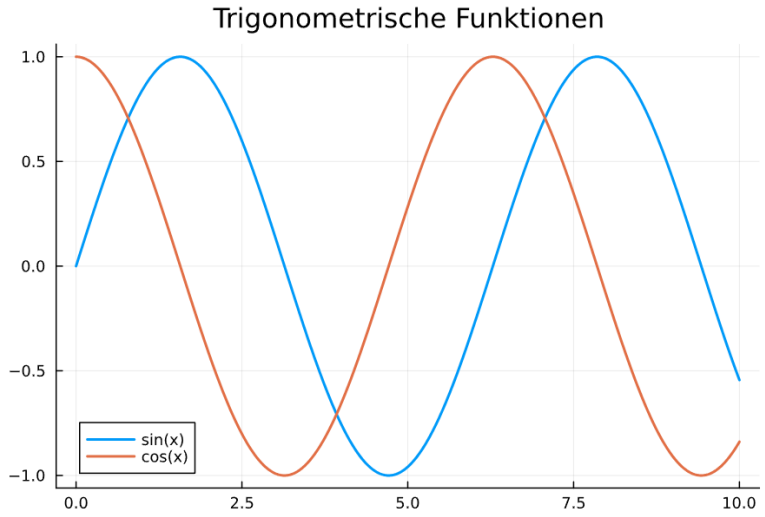

Abbildung 2.3: Erstellung einfacher Graphen unter Nutzung von `Plots.jl`

Auch im Bereich DataScience können Julia Pakete von großer Hilfe sein. Das `CSV.jl-Package` ist ein multithreaded CSV-Parser zum schnellen Einlesen von Datensätzen, über `DataFrames.jl` kann effizient mit Datensätzen umgegangen werden, um gewöhnliche Operationen durchzuführen, das Queryverse bietet Funktionalitäten zur Datenvisualisierung und UI-Tools, um Datenmengen übersichtlich anzeigen zu können. Eine Verbindung zu einem beliebigen Datenbanksystem kann ebenfalls einfach durch die JDBC- oder ODBC-Driver erstellt werden. Machine Learning kann durch die Pakete `MLJ.jl` für unter anderem lineare Modelle und Entscheidungsbäume genutzt werden, `Flux.jl` und Knet.jl bieten Funktionalitäten, um Deep-Learning Algorithmen zu schreiben und `AlphaZero.jl` liefert Schnittstellen mit Googles AlphaZero Netzwerken. Julia ist explizit dafür geschrieben eine Performance zu bieten und in parallelen Programmabläufen gut funktionieren zu können. Aufgrund dessen sind Features wie Multithreading, Berechnungen auf der GPU und verteilte Berechnungen in die Sprache mit eingebaut und Projekte wie `Celeste.jl` sind dafür gebaut eine maximale Leistung zu erbringen und erreichte bereits Geschwindigkeiten von 1.5 PetaFLOP/s (FLoating Point OPerations per second) unter Nutzung von 65.000 CPU-Kernen. Packages wie `DistributedArrays.jl` und `Dagger.jl` bieten Frameworks für Parallelisierungen und mit `Elemental.jl` oder `TSVD.jl` können Berechnungen der linearen Algebra in aufgeteilten Berechnungen durchgeführt werden.

2.4 Wissenschaftliche Anwendungsbereiche

Julia wurde von Grund auf entwickelt, um die erste Wahl der Programmiersprache in jeglichen numerischen und wissenschaftlichen Anwendungsbereichen zu sein und Experten der Domäne bei ihrer Arbeit zu helfen. Aufgrund dessen gibt es ein großes Angebot an Paketen, welche bei mathematischen Anwendungen eingebunden werden

können. Diese umfassen unter anderem `DifferentialEquations.jl` mit Implementierungen von modernen Systemen zum Lösen von Differenzialgleichungen, Iterative Löser von Linearen Systeme wie `IterativeSolvers.jl` und Fast Fourier Transformationen in `AbstractFFTs.jl`. Julia ist allerdings auch in weit mehr wissenschaftlichen Bereichen nützlich wie in der Biologie, Operations Research, Bildverarbeitung, Quantencomputing, quantitative Ökonomie, Astronomie und Ökologie. Wobei jeweils große Netzwerke und Systeme aus Paketen und Libraries vorhanden sind, welche stetig weiter entwickelt werden.

3 Differenzialgleichungen in Julia

Mit dem Fokus der Programmiersprache Julia auf die Entwicklung von wissenschaftlichen und mathematischen Applikationen, ist ebenfalls ein großes Angebot an Bibliotheken vorhanden, um das Lösen von Differenzialgleichungen einfach und effizient zu gestalten. In einer Differenzialgleichung wird eine beliebige gesuchte Funktion $y(t)$ mit ihrer eigenen Ableitung $y'(t)$ in Beziehung gesetzt. Eine gewöhnliche Differenzialgleichung erster Ordnung hat zum Beispiel die Form

$$y\prime(t) = y(t) \cdot 10t$$

welche die Lösung

$$y(t) = c \cdot e^{5t^2}$$

besitzt. Zunächst ist die Lösungsfunktion noch von den beiden Variablen `t` und c abhängig, wodurch keine eindeutige Lösung gebildet werden kann. Allerdings kann durch Angabe eines beliebigen bekannten Wertes der Funktion diese Variable durch einen festen Wert ersetzt werden, um eine eindeutige Lösung für die Differenzialgleichung zu erhalten. Dies wird der Anfangswert der Funktion bezeichnet. Ist zum Beispiel $y(0) = 2$ bekannt, so kann die spezifische Lösung für dieses Problem gefunden werden mit der Funktionsgleichung

$$y(t) = 2 \cdot e^{5t^2}$$

Differenzialgleichungen finden in vielen Anwendungsbereichen der Natur und Technik Nutzen, wie zum Beispiel, der Schwingung und Bewegung von mechanischen Bauteilen, der Berechnung des Belastungsverhaltens verschiedener Materialien aber auch der Bewegungen von Planeten und Sternen oder dem Verhalten von Strömungen. Deswegen ist das schnelle aber auch möglichst genaue Finden von Lösungen von von hoher Bedeutung. Diese Gleichungen sind jedoch häufig durch das Vorkommen höherer Ableitungen, partieller Ableitungen von Funktionen mit mehreren Parametern oder der Nutzung mehrerer Gleichungen für ein differentielles Gleichungssystem beliebig komplex, sodass für ein gegebenes System nicht immer durch mathematische Methoden eine exakte Lösung gefunden werden kann. Aufgrund dessen werden oft numerische Lösungsverfahren gewählt, um eine ausreichend gute Lösung zu finden und es zu ermöglichen, Gleichungen dieser Form computergestützt zu lösen.

3.1 DifferentialEquations.jl

Um in der Julia-Programmiersprache Differenzialgleichungen zu lösen, kann das `DifferentialEquations.jl`-Package genutzt werden, welches gratis zum Herunterladen verfügbar ist. Es bietet Methoden Differenzialgleichungen vieler verschiedener Form zu lösen und ist selbst in Julia programmiert, kann aber ebenfalls in Programmen, welche in Julia, Python oder R geschrieben wurden, verwendet werden. Es implementiert hochoptimierte klassische Algorithmen aber auch moderne Lösungsansätze aus neuester Forschung, welche vergleichbare oder sogar schnellere Laufzeiten wie ähnliche

Programme in C oder Fortran bieten können. Es implementiert intern Wrapperklassen für Lösungsalgorithmen in diesen Sprachen, sodass leicht zwischen verschiedenen Lösern gewechselt werden kann. Auf diese Weise soll garantiert werden, dass für jedes gegebene Problem der beste Lösungsalgorithmus genutzt wird.

3.2 Definieren eines Problems

Die allgemeine Form einer gewöhnlichen Differenzialgleichung ist

$$u^n(t) = f(u(t), u\prime(t), u\prime\prime(t), ..., u^{(n-1)}(t))$$

mit Anfangswerten

$$u(t_0) = u_{0,0}, u\prime(t_0) = u_{0,1}, ..., u^{(n-1)}(t_0) = u_{0,n-1}$$

falls diese vorhanden sind. Es kann vereinfachend allerdings von einer Differenzialgleichung erster Ordnung ausgegangen werden. Dies bedeutet, dass die höchste Ableitung der zu findenden Funktion, welche in der Differenzialgleichungen verwendet wurde, von erster Ordnung ist. Differenzialgleichungen höherer Ordnung können durch Umformung auf Funktionen höherer Dimensionen zu Gleichungen erster Ordnung umgeformt werden, sodass stets nur Gleichungen dieser Ordnung zu betrachten sind. Zum Lösen einer beliebigen Differenzialgleichung mithilfe des DifferentialEquations-Pakets muss zunächst die Gleichung durch das ODEProblem-Interface definiert werden.

```julia
using DifferentialEquations
using Plots

# Erstellen der Methode f, welche die DifferentialGleichung modelliert.
f(u, p, t) = u * 10 * t

# Erstellen des ODEProblem-Objektes durch Übergabe der zu lösenden
# Funktion, dem Anfangswert sowie der Zeitspanne, über welche die
# Gleichung gelöst werden soll.
equation = ODEProblem(f, 2, (0.0, 1.0))
```

Abbildung 3.1: Erstellen einer Differentialgleichung mit `DifferentialEquations.jl`

Zuerst wird eine Methode f definiert, welche die eigentliche Differenzialgleichung nach $y\prime(t)$ aufgelöst darstellt. Die drei Übergabeparameter `u`, `p` und `t` sind dabei von der API von DifferentialEquations vorgegeben, wobei u die Funktion $y(t)$ ist, welche in der Gleichung vorkommt. `t` ist die Variable der Funktion nach welcher integriert werden soll und `p` stellt weitere Parameter dar, von welcher die Funktion abhängig sein könnte. In diesem Beispiel werden keine zusätzlichen Parameter für die Funktion verwendet und p somit ungenutzt gelassen. Anschließend wird über das Interface ODEProblem eine neue gewöhnliche Differenzialgleichung (engl. Ordinary Differential Equation, ODE) per Übergabe der Funktion `f`, des Anfangswertes 2, nach welcher die exakte Lösung der ODE aufgelöst werden soll, und einer Spanne von 0 bis 1, die für den Lösungsalgorithmus die Berechnungsgrenzen (Zeitspanne) festlegt, erstellt. Die wird in der Variable `equation` abgespeichert, welche im folgenden Schritt gelöst werden soll. Falls in der Differenzialgleichung zusätzliche Parameter verwendet wurden, welche die Funktion `f` im Übergabeparameter `p` bekommt, müssen diese ebenfalls bei der Erstellung des `equation`-Objektes nach den Berechnungsgrenzen mit übergeben werden.

3.3 Lösen der Differenzialgleichung

Im folgenden Schritt wird die soeben erstellte Differenzialgleichung gelöst. Dafür wird die Methode `solve` des DifferentialEquations Packages aufgerufen mit Übergabe des soeben erstellten ODEProblem-Objekts. Als zweiter Parameter kann zusätzlich noch ein Lösealgorithmus angegeben werden, mit welchem das Ergebnis berechnet werden soll. Wird wie in diesem Beispiel kein Algorithmus bei Aufruf der Methode übergeben, wird durch das DifferentialEquations-Paket automatisch die übergebene Funktion analysiert und ein passender Lösungsalgorithmus ausgesucht.

```
# Lösen der DGL über die solve-Funktion und dem ODEProblem-Objekt
solution = solve(equation)

# Exakte Lösung der DGL, um einen Vergleich der Genauigkeit zu erstellen
trueSolution(x) = 2*exp(5 * x^2)

x = range(0.0, 1.0, step=0.01)
y = trueSolution.(x)

# Zeichnen der beiden Funktionen in ein Diagramm zum graphischen Vergleich
plot(solution, linewidth = 4, label = "Lösungsfunktion der DGL")
plot!(x, y, linewidth = 3, ls=:dash, label = "Exakte Lösung")
```

Abbildung 3.2: Lösen einer Differenzialgleichung mit `DifferentialEquations.jl`

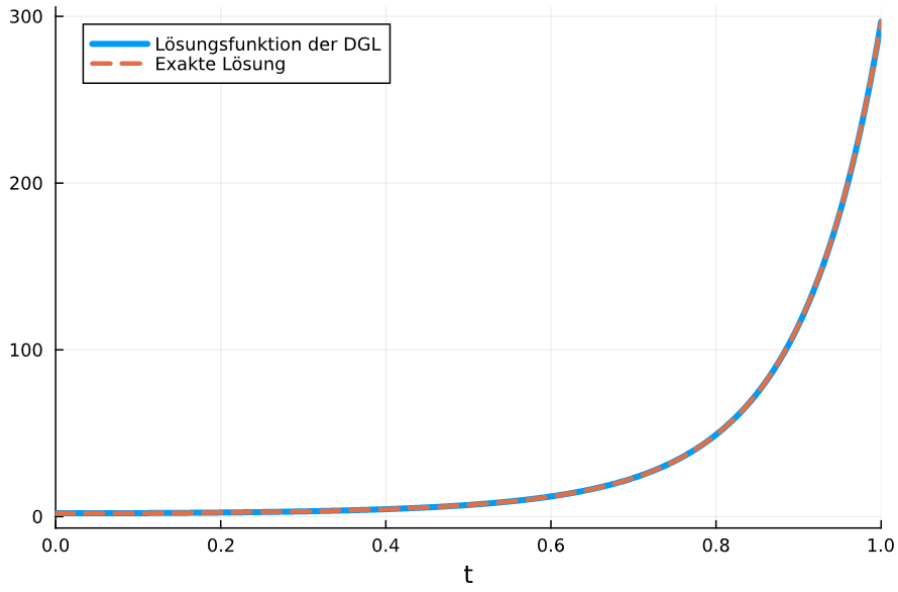

Abbildung 3.3: Ausgabe der Lösung des Algorithmus im Vergleich zur von Hand gelösten exakten Funktion

Das Ergebnis der `solve`-Methode ist ein eigenes Solution-Objekt, welches unter anderem eine Liste an Punkten enthält, welche die `y`-Werte an den gegebenen Werten von `t` in diesem Intervall annähern. Standardmäßig wird die Schrittweite während des Algorithmus dynamisch bestimmt. Diese lässt sich aber manuell anpassen, um eine schnellere Lösung zu erhalten auf Kosten einer geringeren Genauigkeit. In diesem Code-Beispiel zeichnet die blaue Linie das Ergebnis des Lösevorgangs der solve-Methode und die orange Linie die exakte Lösung des Differenzialgleichung. Wie anhand der Graphen zu erkennen ist, passen die beiden Graphen im angegebenen Zeitintervall genau aufeinander und die numerische Lösung dieser ODE gibt eine sehr genaue Approximation des Graphen zurück.

3.4 Beispiele

Im Folgenden werden eine Auswahl von ODEs erstellt und mithilfe der automatischen Lösungsmethode gelöst, um die Genauigkeit der vom Package implementierten Algorithmen zu überprüfen. Ebenso wird die durchschnittliche Rechenzeit ermittelt, indem dieselbe Gleichung 100000-Mal gelöst und die für jeden Vorgang benötigte Zeit gemessen wird. Diese wird im nächsten Kapitel mit äquivalenten Methoden des Python-Universums verglichen, um die Geschwindigkeit des Julia-Verfahrens zum Lösen von Differenzialgleichungen zu vergleichen.

```
function solveODEAndPlot(f, s, u0, t)

    # Definieren des Problems über das ODEProblem-Interface
    # per Übergabe der gegebenen Funktionsparameter
    problem = ODEProblem(f, u0,(t[1], t[2]))

    solution = solve(problem)

    # Die Gleichung wird 100000-mal durch die solve-Methdoe
    # gelöst, um eine Durchschnittlichszeit zu berechnen
    sum = 0.0
    for i in range(0, 100000)
        startTime = time()
        solution = solve(problem)
        endTime = time()
        sum += endTime-startTime
    end

    # Ausgabe der Durchschnittlichszeit, wobei das Teilen der
    # Gesamtzeit durch die Anzahl der Iterationen und das
    # Ändern der Einheit von Sekunden zu Mikrosekunden
    # verrechnet werden kann
    println("Durchschnittliche Berechnungszeit pro DGL:")
    result = sum * 100
    println("$result Mikrosekunden")

    # Berechnung von Wertepaaren der exakten Lösung
    x = range(t[1], t[2], step=0.01)
    y = s.(x)

    # Ausgabe der Ergebnisse als Diagramme
    plot(solution, linewidth = 4, label = "Lösungsfunktion der DGL")
    plot!(x, y, linewidth = 3, ls=:dash, label = "Exakte Lösung")
end
```

Abbildung 3.4: Erstellung einer Hilfsmethode, welche eine beliebige Differenzialgleichung löst und das Ergebnis in einem Diagramm zeichnet

Diese Hilfsmethode ermöglicht es eine beliebige übergebene Differenzialgleichung lösen zu lassen, sowie sie mit der exakten Lösung zu vergleichen. Zunächst wird hier erneut das ODEProblem-Objekt erstellt, welches ebenfalls bei Initialisierung den übergebenen Anfangswert sowie die übergebene Zeitspanne erhält. Anschließend wird die Gleichung durch die `solve`-Methode 100000-mal gelöst. Dabei werden vor und nach der Berechnung die aktuelle Zeit von der `time`-Methode abgefragt und die Differenz abgespeichert, welche über alle Iterationen hinweg summiert wird. Anschließend wird die durchschnittliche Zeit berechnet, für eine bessere Lesbarkeit von Sekunden zu Mikrosekunden umgerechnet und ausgegeben, um später einen Vergleich mit Python vornehmen zu können. Anschließend wird die Lösung von der `solve`-Methode und die eingegebene exakte Lösung gezeichnet, um einen grafischen Vergleich machen zu können.
Im Folgenden werden einige Beispiele für gewöhnliche Differenzialgleichungen erstellt, wobei zunächst immer der Code zum Erstellen und Lösen der Gleichungen unter Nutzung oben erstellter Hilfsfunktion gezeigt wird und anschließend ein Diagramm angefügt ist, welcher die numerische Lösung und die von Hand berechnete exakte Lösung

gezeichnet ist, um diese miteinander vergleichen zu können.

```
f(u, p, t) = - (t / u)
s(t) = sqrt(16.0-t^2)
t = [0, 4]

solveODEAndPlot(f, s, 4, t)
```

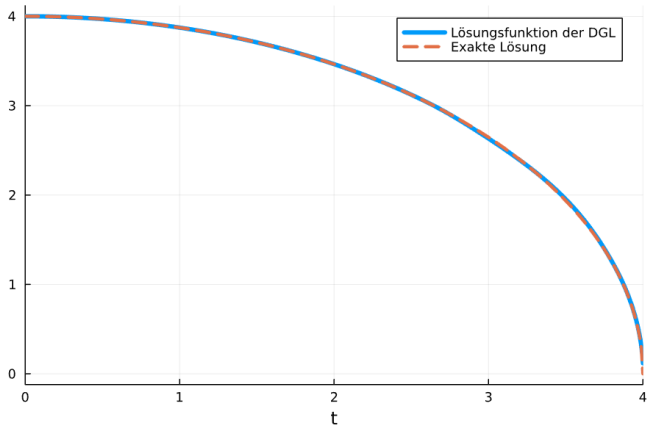

Abbildung 3.5: $y\prime = -\frac{t}{y}$
Durchschnittliche Berechnungszeit: 5.69000244140625 Mikrosekunden

```
f(u, p, t) = (2 * t) / exp(u)
s(t) = log(1 + t^2)
t = [0, 10]

solveODEAndPlot(f, s, 0, t)
```

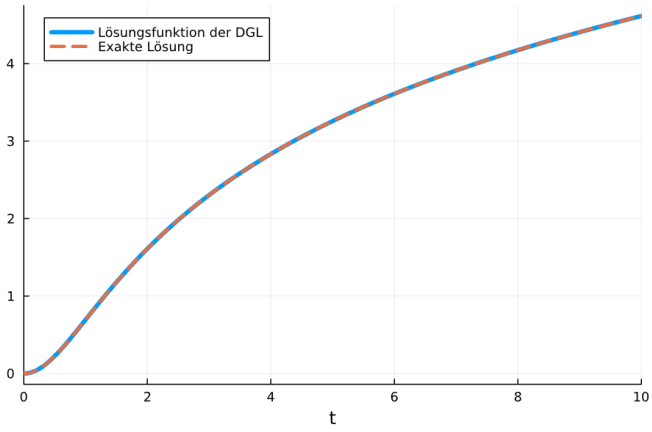

Abbildung 3.6: $y\prime = \frac{2 \cdot t}{e^y}$
Durchschnittliche Berechnungszeit: 6.719999313354492 Mikrosekunden

```
f(u, p, t) = (2 * t * u) / (t^2 + 1)
s(t) = t^2 + 1
t = [0, 10]

solveODEAndPlot(f, s, 1.0, t)
```

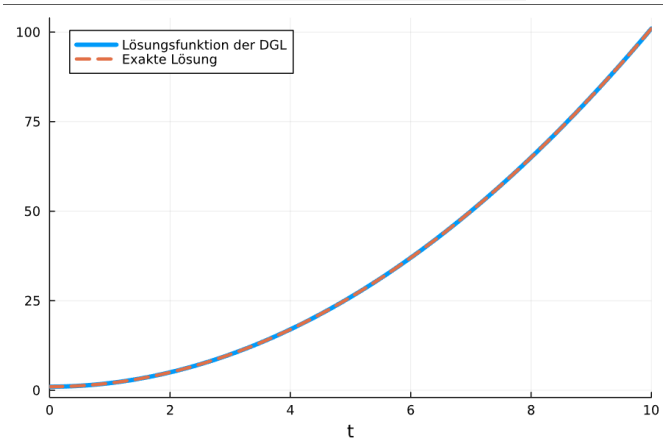

Abbildung 3.7: $y\prime = \frac{2 \cdot t \cdot y}{t^2+1}$
Durchschnittliche Berechnungszeit: 5.9599995613098145 Mikrosekunden

```
f(u, p, t) = 1 / (t+u) - 1
s(t) = sqrt(-0.187 + t^2 + 2 * (t - t^2 /2 )) - t
t = [0.1, 10]

solveODEAndPlot(f, s, 0.0, t)
```

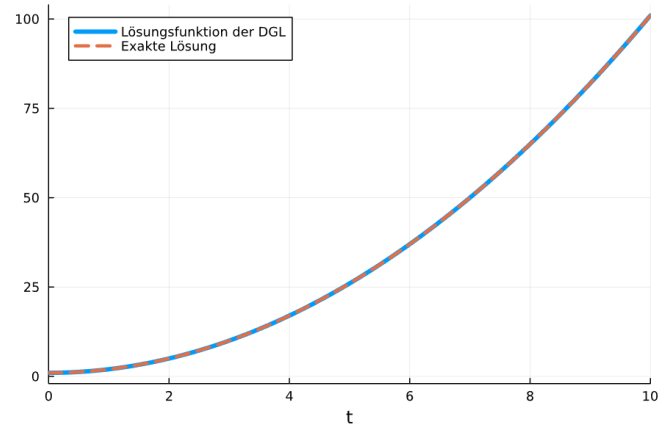

Abbildung 3.8: $y\prime = \frac{1}{t+y} - 1$
Durchschnittliche Berechnungszeit: 6.909999847412109 Mikrosekunden

```
f(u, p, t) = -5*u + sin(t*pi)
s(x) = -0.9 * exp(-5*x) + (5 * sin(pi * x)) / (25 + pi^2) - (pi * cos(pi * x)) / (25 + pi^2)
t = [0, 20]

solveODEAndPlot(f, s, -1.0, t)
```

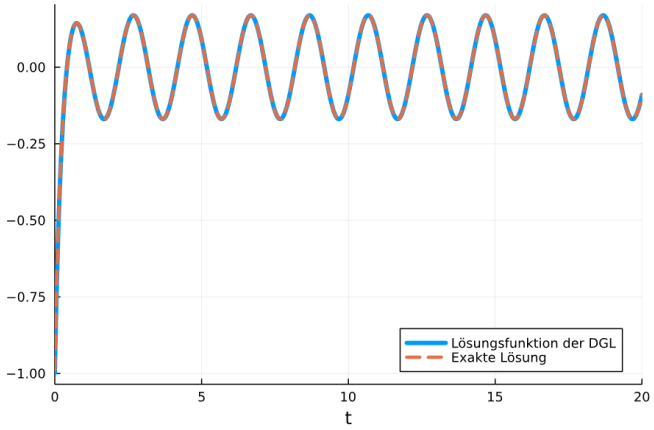

Abbildung 3.9: $y\prime = -5 \cdot u + sin(\pi t)$
Durchschnittliche Berechnungszeit: 19.28999423980713 Mikrosekunden

Anhand der obenliegenden Beispiele ist sehr gut zu erkennen, dass das `DifferentialEquations`-Paket sehr exakte Lösungen für die eingegebenen Probleme liefert, welche im grafischen Vergleich nicht von den exakten, eingegebenen Lösungen abweichen. Auf dem Rechner, auf dem diese Arbeit verfasst wurde, liegt die durchschnittliche Laufzeit bei 30 bis 180 Mikrosekunden abhängig von der Komplexität der Gleichung sowie der Größe des Intervalls.

4 Vergleich mit Solvern des Python-Universums

Da Julia direkt dafür entwickelt wurde dynamische Sprachen wie Python und R in der alltäglichen Arbeit von Mathematikern zu ersetzen, wird nun die Nutzbarkeit und Geschwindigkeit zu dem meist genutzten Solver-Paket des Python-Universums verglichen. Dieses Paket ist SciPy, welche eine Erweiterung des bekannten NumPy-Paketes ist und wichtige Formeln und Algorithmen in vielen Bereichen der Wissenschaft vorimplementiert und zur Nutzung verfügbar macht. Beispiele dafür sind Fourier Transformationen, Bildbearbeitung und Statistiktools. In diesen Anwendungsfällen wird das `scipy.integrate`-Unterpaket genutzt, welches Algorithmen zur Integration und Lösen von Differenzialgleichungen beinhaltet. Wie im DifferentialEquations-Packet der Julia-Programmiersprache wurde es sich zum Ziel gesetzt eine möglichst hohe Performance zu bieten, dabei aber so einfach nutzbar wie möglich zu sein. Um diese Versprechen zu halten werden zur Lösung der Probleme optimierte Implementierungen in kompilierten Sprachen wie Fortran oder C genutzt und im Python-Code aufgerufen, um die schnelle Performance kompilierten Bytecodes zu nutzen.
Dies steht in Kontrast zu anderen Lösepaketen wie SymPy, welches intern komplett in Python geschrieben ist und aufgrund des langsamen Interpreters deutlich längere Laufzeiten besitzt.

4.1 Definieren des Problems

Ähnlich wie im DifferentialEquations-Paket in Julia kann hier ebenfalls die Differenzialgleichung direkt als Funktion aufgeschrieben werden. Der Parameter `y` ist dabei die gesuchte Funktion und `t` die Variable, nach welcher die Differenzialgleichung integriert werden soll. Weitere Variablen, von denen die Gleichung abhängig sein kann, können ebenfalls als ein dritter Parameter der Methode in einer Liste angegeben sein, können im Gegensatz zum Julia-Äquivalent aber auch wie hier weg gelassen werden. Die Differenzialgleichung

$$y\prime(t) = y(t) \cdot 10t$$

kann nun in der Methode `equation` geschrieben werden. Zum Vergleich des Verfahrens mit der exakten Lösung wird ebenfalls die Methode `trueSolution` definiert, welche die exakte Lösung der DGL mit dem Anfangswert

$$y(0) = 2$$

implementiert.

```python
from scipy.integrate import *
import numpy as np
import matplotlib.pyplot as plt
import math
import time

# Definition der Differentialgleichung als Methode
def equation(y, t):
    return y * 10 * t

# Implementierung der exakten Lösung zum Vergleich
def trueSolution(x):
    return 2 * math.exp(5 * x**2)
```

Abbildung 4.1: Erstellen einer Differentialgleichung mit `scipy.integrate`

4.2 Lösen der Differenzialgleichung

Zum Lösen der Differenzialgleichung wird im nächsten Schritt die Methode `odeint` des SciPy-Packages aufgerufen. Im Gegensatz zu Implementierung der `solve`-Methode des DifferentialEquation-Paketes muss die Gleichung nicht zuerst in ein Objekt der Klasse `ODEProblem` umgewandelt werden, sondern kann der Methode direkt übergeben werden. Ebenfalls muss ein Anfangswert und eine Liste an Punkten übergeben werden, an welchen die Gleichung angenähert werden soll. Vergleichbar zum Lösungsobjekt des Julia-Paketes wird eine Liste an y-Werten zurückgegeben, welche die Funktion an den übergebenen Werten der Liste `t` annähert. Diese werden im Anschluss in einem Graphen gezeichnet, um einen Vergleich mit der exakten Lösung machen zu können.

```python
y0 = 2
t = np.linspace(0, 1, 101)

# Aufruf des Lösealgorithmus per Übergabe der Funktion
# sowie des Anfangswertes und einer Liste an x-Werten
solution = odeint(equation, y0, t)

# Berechnung der exakten y-Werte durch die exakte Lösung
yExact = [trueSolution(x) for x in t]

# Ausgabe der Ergebnisse als Graphen
plt.plot(t, solution[:, 0], label="Lösungsfunktion der DGL", linewidth = 3)
plt.plot(t, yExact, label="Exakte Lösung", linestyle='dashed', linewidth = 2)
```

Abbildung 4.2: Losen einer Differenzialgleichung mit `scipy.integrate`

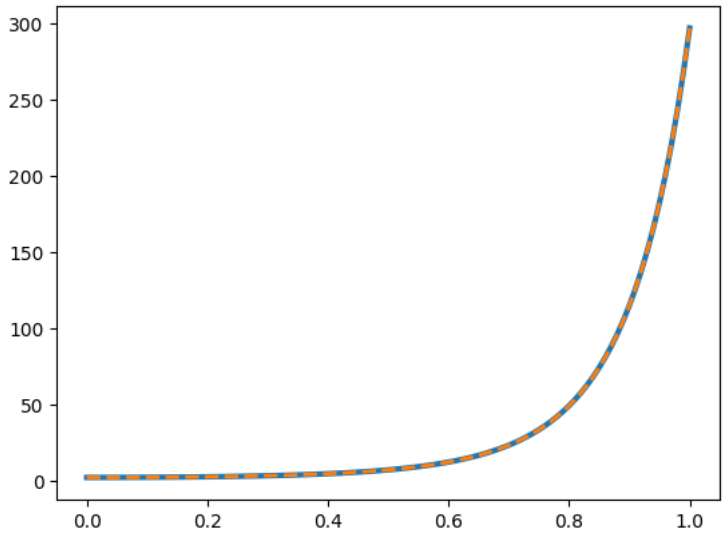

Abbildung 4.3: Ausgabe der Losung des Algorithmus im Vergleich zur von Hand gelösten exakten Funktion

4.3 Beispiele

Genauso wie im Test des Julia-Packages ist auch hier eine Auswahl an ODEs implementiert, welche mit SciPy gelöst werden sollen, um die Genauigkeit der Methoden zu vergleichen. Die unten stehende Hilfsmethode löst die übergebene Differenzialgleichung unter Nutzung der `odeinz`-Funktion von `scipy.integrate` und zeichnet den Graphen der berechneten Lösung sowie der ebenfalls übergebenen exakten Gleichung. Ebenso wie in der Berechnung der durchschnittlichen Berechnungszeit von `DifferentialEquation.jl` werden auch hier 10000 Iterationen durchgeführt, um die durchschnittlich benötigte Laufzeit zum Lösen der Gleichung ermitteln zu können.

```python
def solveODEAndPlot(equation, exactSolution, y0, t):

    # Erstes Lösen der eingegebenen Gleichung mit der odeint-Methode
    # und per Übergabe des Anfangswerts und der Zeitspanne
    solution = odeint(equation, y0, t)

    # Die Gleichung wird 1000-mal gelöst, um eine Durchschnittszeit
    # berechnen zu können. Bei jeder Iteration werden vor und nach
    # der Iteration die Zeitpunkte gemessen und anschließend die
    # Differernz berechnet. Die summierte Differenz über alle
    # Iterationen wird für die Durchschnittszeit abgespeichert.
    sum = 0.0
    for i in range(0, 10000):
        startTime = time.time()
        solution = odeint(equation, y0, t)
        endTime = time.time()
        sum += endTime - startTime

    # Die durchschnittliche Laufzeit wird berechnet und auf
    # Mikrosekunden umgerechnet. Das Ergebnis dieser Rechnung ist die
    # durchschnittliche Laufzeit in Mikrosekunden, welche anschließend
    # ausgegeben wird.
    print("Durchschnittliche Berechnungszeit pro DGL:")
    print(f"{sum * 100} Mikrosekunden")

    # Mithilfe der übergebenen exakten Lösung werden für diese die
    # y-Werte berechnet. Dafür wird erneut die übergebene Liste an
    # x-Werten genutzt und die exakte Gleichung auf diese angewendet.
    yExact = [exactSolution(x) for x in t]

    # Mit der MatPlot-Library werden die Ergebnisse ausgegeben, um einen
    # graphischen Vergleich machen zu können.
    plt.plot(t, solution[:, 0], label="Lösungsfunktion der DGL", linewidth = 3)
    plt.plot(t, yExact, label="Exakte Lösung", linestyle='dashed', linewidth = 2)
```

Abbildung 4.4: Erstellung einer Hilfsmethode, welche eine beliebige Differenzialgleichung löst und das Ergebnis in einem Diagramm zeichnet

```python
def equation(y, t):
    return - t / y

def s(x):
    return math.sqrt(16.0-x**2)

y0 = 4.0
t = np.linspace(0, 4, 401)
solveODEAndPlot(equation, s, y0, t)
```

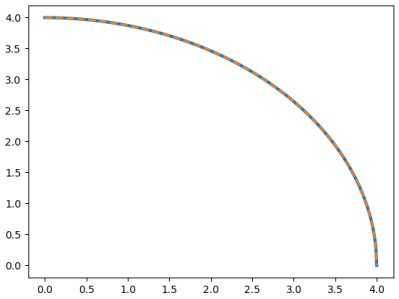

Abbildung 4.5: $y\prime = -\frac{t}{y}$

Durchschnittliche Berechnungszeit pro DGL:
1317.1658515930176 Mikrosekunden

```
def equation(y, t):
    return 2*t / math.exp(y)

def s(x):
    return math.log(1 + x**2)

y0 = 0.0
t = np.linspace(0, 10, 1001)
solveODEAndPlot(equation, s, y0, t)
```

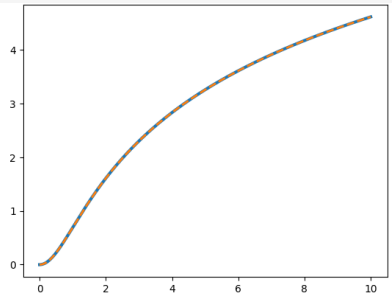

Abbildung 4.6: $y\prime = \frac{2 \cdot t}{e^y}$

Durchschnittliche Berechnungszeit pro DGL:
154.54483032226562 Mikrosekunden

```
def equation(y, t):
    return 2*t*y / (t**2 + 1)

def s(x):
    return x**2 +1

y0 = 1.0
t = np.linspace(0, 10, 1001)
solveODEAndPlot(equation, s, 1.0, t)
```

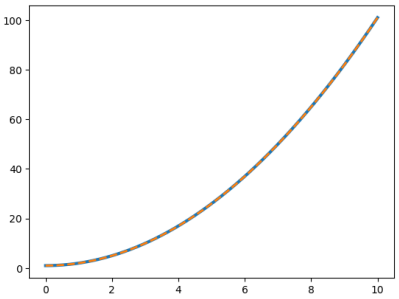

Abbildung 4.5: $y\prime = \frac{2 \cdot t \cdot y}{t^2+1}$
Durchschnittliche Berechnungszeit pro DGL:
36.00144386291504 Mikrosekunden

```
def equation (y, t):
    return 1 / (t + y) - 1

def s(x):
    return math.sqrt(-0.19 + x**2 + 2 * (x - x**2 /2 )) - x

y0 = 0.0
t = np.linspace(0.1, 10, 9991)
solveODEAndPlot(equation, s, 0, t)
```

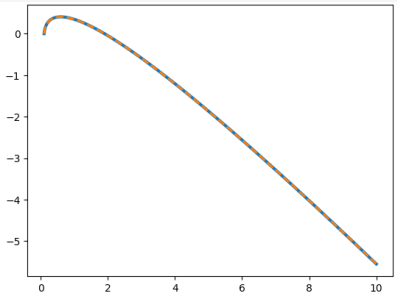

Abbildung 4.5: $y\prime = \frac{1}{t+y} - 1$
Durchschnittliche Berechnungszeit pro DGL:
837.7182483673096 Mikrosekunden

```
def equation (y, t):
    return -5 * y + math.sin(t * math.pi)

def s(x):
    return -0.9 * math.exp(-5*x) + (5 * math.sin(math.pi * x)) / (25 + math.pi**2) - (math.pi * math.cos(math.pi * x)) / (25 + math.pi**2)

t = np.linspace(0, 20, 2001)

solveODEAndPlot(equation, s, -1.0, t)
```

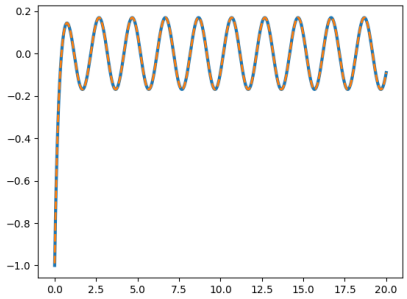

Abbildung 4.6: $y\prime = -5 \cdot y + sin(t \cdot \pi)$
Durchschnittliche Berechnungszeit pro DGL:
1202.7056217193604 Mikrosekunden

4.4 Vergleich zur Julia-Sprache

Wie anhand der Graphen der obigen Beispiele zu erkennen ist, liefert das Python-Package ebenfalls sehr genaue Ergebnisse als Lösung der Differenzialgleichung, welche sehr gut vergleichbar sind mit den Lösungen der Lösungsmethoden des `DifferentialEquations.jl` Packetes. Ebenfalls ist eine sehr einfach Syntax definiert um Probleme erstellen und lösen lassen zu können. lösen zu lassen. Allerdings ist an der Rechenzeit ein großer Unterschied an den beiden Programmiersprachen zu erkennen. Während es auf dem zum Schreiben dieser Arbeit genutzten Rechner in Julia problemlos möglich ist 100000 Iterationen in meist weniger als einer Sekunde durchzuführen, werden in Python bei 1000 Iterationen derselben Gleichung bereits vergleichbare Zeiten erreicht. Dies ist auch bei der durchschnittlichen Rechenzeit pro Lösungsvorgang zu erkennen, welche zwischen 30 und 1500 Mikrosekunden und damit durch den Interpreter pro Iteration 10- bis 1000-mal so lange dauern wie im kompilierten Julia-Ansatz.
Im Hinblick auf das Lösen von ODEs ist es also sehr gut möglich die Programmiersprache Python durch Julia zu ersetzen, da es mit einer vergleichbar leicht zu nutzenden Syntax und Nutzungsweise eine merkbar schnellere Laufzeit bietet. Im Vergleich der Laufzeiten des Python-Codes bemerkt man zudem, dass die Laufzeiten bei verschiedenen Gleichungen stark voneinander abweichen im Vergleich zu den Lösealgorithmen in der Julia-Sprache, welche zwar trotzdem abhängig vom eingegebenen Problem verschiedene Laufzeiten besitzen, diese sich allerdings nur um wenige Mikrosekunden voneinander unterscheiden.

5 Laufzeitbeschleunigung durch Nutzung der GPU

Um bei der Berechnung von sehr großen Differenzialgleichungen die Laufzeit des Codes noch weiter zu beschleunigen, kann es in manchen Fällen nützlich sein, den Großteil der Berechnungen auf die GPU des Computers zu verlegen. Diese sind speziell dafür gebaut mit großer Effizienz mathematische Rechnungen und Operationen mit Floating Point Zahlen durchführen zu können. Damit sind sie das ideal, um bei numerischen Algorithmus viele Berechnungen auszuführen.
Vor allem zwei Fälle von Differenzialgleichungen der Form

$$u\prime(t) = f(u(t))$$

sind bei der Umlegung der Berechnungen auf die GPU geeignet: einerseits Probleme, bei welchen die Gleichung `f` sehr groß, aber strukturiert ist, sodass die Auswertungen von `f` während der Ausführung des Algorithmus von der GPU übernommen werden kann. Dieser Fall liegt zum Beispiel vor bei der Gleichung

$$u\prime(t) = A \cdot u$$

wobei `A` eine beliebig große Matrix ist, welche an den Vektor der Funktion `u` multipliziert wird. Diese Matrix-Vektor-Multiplikation kann sehr stark von der GPU optimiert werden, wodurch der Code beschleunigt werden kann. Allerdings können auch kleinere Funktionen von `f` von der Geschwindigkeit der GPU profitieren, wenn diese zum Beispiel für sehr viele verschiedene Anfangswerte `u0` berechnet werden müssen. In diesem Fall kann die Berechnung der verschiedenen Gleichungen parallelisiert auf der GPU stattfinden.

```
using OrdinaryDiffEq, CUDA, LinearAlgebra

u01 = rand(10000)
A1 = randn(10000, 10000)
f1(du, u, p, t) = mul!(du, A1, u)

equation1 = ODEProblem(f1, u01, (0.0f0, 1.0f0))

u02 = cu(rand(10000))
A2 = cu(randn(10000, 10000))
f2(du, u, p, t) = mul!(du, A2, u)

equation2 = ODEProblem(f2, u02, (0.0f0, 1.0f0))
```

Abbildung 5.1: Erstellen von zwei gleichen Differenzialglechungen auf CPU und GPU

In diesem Beispiel wird der erste der obigen Fälle verdeutlicht und in zwei verschiedenen Differenzialgleichungen dasselbe Problem einmal für eine Berechnung auf der CPU und im unteren Teil für eine Berechnung auf der GPU vorbereitet. Es handelt sich jeweils um eine Matrix der Dimension 10.000 x 10.000, welche in der Gleichung mit der Funktion `y` multipliziert werden. Zusätzlich werden Anfangswerte `u0` erstellt, welche 10.000 Element große Vektoren sind. Die Matrix und der Vektor werden in

diesem Beispiel mit zufälligen Zahlen gefüllt mit der Methode `cu`, werden diese Elemente jedoch nicht mehr auf der CPU, beziehungsweise im RAM abgelegt, sondern werden im Grafikspeicher der GPU abgelegt. Wird nun während des Löseprozesses versucht die Funktion an einer Position auszuwerten, wird eine Anfrage an die GPU erstellt, welche dann die nötigen Berechnungen ausführt und das Ergebnis zurückgibt. Es muss zusätzlich das CUDA-Paket importiert werden und die Anfangs- sowie Endwerte der Berechnung in Float32-Literalen angegeben werden. Erstere ist eine von NVIDIA entwickelte Plattform, welche die Entwicklung von General-Purpose Code wie dem Berechnen einer Differenzialgleichung auf der GPU erlaubt. Die Nutzung von Float32-Variablen anstatt der genaueren Float64- oder sogar Float128-Datentypen ist ausschließlich durch Geschwindigkeitsoptimierungen der GPU-Architektur zu erklären. Da Float32-Variablen lediglich die Hälfte des Speicherplatzes der äquivalenten Float-64 Zahl einnehmen, sind diese in diesem Anwendungsfalll bevorzugt. Ohne besonders große Verluste in der Genauigkeit der Ergebnisse zu haben, führt diese Wahl der Datentypen zu deutlich schnelleren Laufzeiten, wobei meist auch nur ein Bruchteil des sonst benötigten Speicherplatzes benötigt wird.

```
startTime = time()
solve(equation1, Tsit5())
endTime = time()

println("Die Berechnung auf der CPU benötigte $(endTime-startTime) Sekunden.")

startTime = time()
solve(equation2, Tsit5())
endTime = time()

println("Die Berechnung auf der GPU benötigte $(endTime-startTime) Sekunden.")
```

Abbildung 5.2: Laufzeitenvergleich von CPU und GPU
Die Berechnung auf der CPU benötigte 10.659000158309937 Sekunden.
Die Berechnung auf der GPU benötigte 0.9179999828338623 Sekunden.

Wie in diesem Beispiel gesehen werden kann, können mit der Nutzung der GPU deutlich höherer Geschwindigkeiten erreicht werden, als nur unter Nutzung der CPU. Auf dem Computer, auf dem diese Arbeit geschrieben wird und welcher mit einer NVIDIA GeForce 3070 Grafikkarte besitzt, benötigt die Berechnung der Differenzialgleichung 9 - 12 Sekunden, während unter Nutzung der Rechenleistung der GPU lediglich eine Zeit von etwa 0.9-1.5 Sekunden anhängig von den zufälligen Werten der Matrizen zum Finden einer Lösung benötigt. Dabei ist darauf zu achten, dass beim ersten Nutzen der `solve`-Methode mit Parametern, welche im Grafikspeicher liegen, noch GPU-Code Konstruktion geschehen muss, sodass dieser erste Vorgang meist vergleichbar lange dauert, wie das Lösen mit den Methoden der CPU. Ab dem zweiten Vorgang werden die oben genannten Geschwindigkeiten jedoch erreicht.

6 Fazit

Wie in dieser Arbeit analysiert wurde, ist es sehr gut möglich den Python in der Nutzung des Lösens von Differenzialgleichungen zu ersetzen. Julia bietet viele Features, welche die Nutzung der Sprache sehr komfortabel machen und so einen Umstieg zu Julia von einer anderen Programmiersprache wie Python oder R so reibungslos gestalten wie möglich. Durch den Just-In-Time Compiler der Sprache können sehr hohe Geschwindigkeiten erreicht werden, welche bei außreichend guter Optimierung mit Sprachen wie Fortran und C mithalten kann.
Anhand der Codebeispiele ist ebenfalls gut zu erkennen, dass sowohl die Performance als auch die Qualität der Ergebnisse ähnlich bzw. sogar besser sind als solche der verglichenen Python-Varianten und die Syntax zum Definieren und Lösen von beliebigen Problemen sehr einfach gestaltet ist. Python Pakete liegen in Genauigkeit und Einfachheit der Nutzung nicht hinterher, bieten allerdings durch das Fehlen eines Compilers eine deutlich langsamere Laufzeit als Julia und hochperformante Pakete in diesen Sprachen müssen Nutzen von schnelleren Sprachen vor Fortran, C oder auch Julia machen, um eine vergleichbar lange Laufzeit erreichen zu können.
Durch die Nutzung der GPU kann der eigene Code ebenfalls mit den Kosten etwas komplizierterer Nutzung um ein Vielfaches beschleunigt werden, was jedoch eine spezielle Art von Problemen und eine rechenstarke Grafikkarte vorraussetzt.

7 Quellenverzeichnis

- https://julialang.org zuletzt besucht am 04.01.2024
- https://docs.sciml.ai/DiffEqDocs/stable/ zuletzt besucht am 06.01.2024
- https://scipy.org/ zuletzt besucht am 07.01.2024
- https://www.sympy.org/en/index.html zuletzt besucht am 07.01.2024
- https://docs.sciml.ai/DiffEqGPU/stable/ zuletzt besucht am 07.01.2024