Seminararbeit: Softwareentwicklung in der Wissenschaft

Eine Einführung

von

Sandra Schröder
geboren am
12.05.1989

Seminar: "Softwareentwicklung in der Wissenschaft"
im Studiengang Informatik
Arbeitsbereich Wissenschaftliches Rechnen
Universität Hamburg

Vortragsdatum: 9. Dezember 2010

Betreuerin: Petra Nerge

Inhaltsverzeichnis

1 Einleitung

Computer spielen seit ihrer Erfindung eine tragende Rolle in Wissenschaft und Technik. Angefangen hat dies mit der von Charles Babbage entwickelten *"Analytical Machine"*, mit der als erste Rechenmaschine aufwendige Rechenaufgaben ausgeführt wurden. Konrad Zuse baute schließlich den ersten Computer. Mit der raschen Entwicklung des Computers ergaben sich immer mehr Möglichkeiten zur Verarbeitung von sehr großen Datenmengen, sowie nichtlinearen Wechselwirkungen und Gleichungen. In nahezu jedem Bereich findet der Computer seinen Einsatz und er ist aus unserem Alltag als unterstützendes oder auch unterhaltendes Medium kaum noch wegzudenken. Besonders für Naturwissenschaften wie Physik, Biologie, Klimaforschung oder Chemie ist der Computer ein hilfreiches Werkzeug, um komplizierte Berechnungen oder virtuelle Experimente durchzuführen.
Die Informatik spielt in den Naturwissenschaften häufig die Rolle einer Basistechnologie und wird als sogenannte strukturelle Wissenschaft angesehen. Mit ihr lassen sich optimal große Mengen von Informationen strukturieren, verarbeiten und aufbereiten. Jedoch führt der Einsatz der Informatik zu Modellbildungen in den Naturwissenschaften, sodass sich neue Fragestellungen und Erkenntnisse ergeben. Fragestellung bilden sich durch Beobachtung, Nachforschungen und Überlegungen, die der Wissenschaftler versucht in einem Modell zu erfassen und sich zu erklären. Er beschreibt das Modell, indem er sich den Konzepten der Mathematik bedient, die ihm helfen, seine Erkenntnisse effektiv zu strukturieren und logisch zu begründen. Nachdem sich der Wissenschaftler ein mathematisches Modell in Form von Gleichungen geschaffen hat, möchte er diese nun lösen, da er an der exakten Lösung interessiert ist. Für die Lösung gibt es kontinuierliche Verfahren. Die Gleichungen werden, wenn sie nicht zu kompliziert sind, von Hand gelöst. Meistens ist dies jedoch nicht der Fall und der Wissenschaftler benötigt die Unterstützung des Rechners, da dieser natürlich schneller rechnen kann als der Wissenschaftler selbst. An dieser Stelle kommt die Informatik ins Spiel. Die in mathematischen Gleichungen beschriebenen Beobachtungen sollen nun effektiv erfasst und mit Hilfe eines diskreten Algorithmus gelöst werden. Hier tritt nun eine Herausforderung auf, mit der Wissenschaftler, Mathematiker und Informatiker sich auseinandersetzen müssen, wenn Kontinuierliches auf etwas Diskretem abgebildet werden soll. Kontinuierlich bedeutet hierbei die Modellierung der wissenschaftlichen Fragestellung in mathematischen Gleichungen, die auf der nicht endlichen und überabzählbaren Menge der reellen Zahlen erfolgt. Die Modellierung soll auf dem Rechner abgebildet werden, der aufgrund seines endlichen Speichers nur auf einer diskreten Zahlenmenge rechnet und nicht alle reellen Zahlen erfassen kann. Bei dieser Diskretisierung entstehen Fehler, denen sich Wissenschaftler, Mathematiker und Informatiker bewusst werden müssen. Abbildung 1.1 stellt genau diese Ausgangssituation dar. Die naturwissenschaftliche Fragestellung und die mathematische Modellierung fallen in den kontinuierlichen Kontext, während das Entwickeln von numerischen Verfahren bzw. Algorithmen sowie das Berechnen am Rechner in den diskreten Kontext gehören. Diese Seminararbeit beschäftigt sich mit dieser Situation. Zum einen wird die Rechnerarithme-

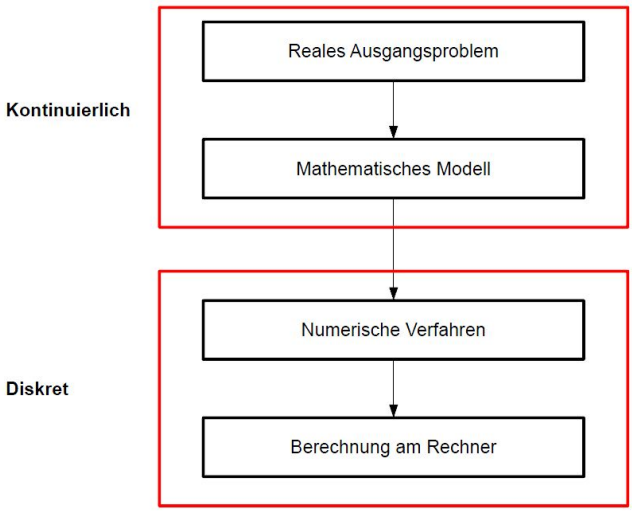

Abbildung 1.1: Abtastung

tik eingeführt, um zu zeigen, wie reelle Zahlen im Rechner optimal dargestellt werden können. Zudem zeigt das Kapitel Numerik, wie die Fehler analysiert und minimiert werden können, indem drei wichtige Fehlerbewertungsmechanismen vorgestellt werden.
Natürlich sollen auch die Konzepte der Softwareentwicklung gezeigt werden von der Planung der Software bis zu ihrem Test und ihrer Leistungsbewertung. Dabei soll auch darauf eingegangen werden, wie die Kommunikation zwischen Wissenschaftler und Informatiker stattfindet und welche Probleme dabei auftreten können. Die Leistung und Effizienz einer Software wird in einem separaten Kapitel betrachtet, indem Begriffe wie Speicherhierarchie und Lokalität erläutert werden.

2 Wissenschaftliche Softwareentwicklung

Die Entwicklung wissenschaftlicher Software unterscheidet sich in gewissen Zügen von der Entwicklung von Software außerhalb wissenschaftlicher Anwendungen. Sei es das generelle Vorgehensmodell und die Kommunikation zwischen Wissenschaftler und Softwareentwickler, die Anforderungen an die Software oder auch die Benutzung wissenschaftsspezifischer Programmierwerkzeuge. In diesem Kapitel werden die eben genannten Sachverhalte erläutert.

2.1 Was ist wissenschaftliche Softwareentwicklung?

Wissenschaftliche Software soll den Anwender dabei unterstützen, sein entwickeltes Modell zu verstehen und über Vorgänge in seinem Modell Erkenntnisse zu gewinnen. Dabei bleibt es nun dem Wissenschaftler überlassen zu entscheiden, ob er seine Software alleine entwickelt oder in Zusammenarbeit mit einem Softwareentwickler, der normalerweise keine Kenntnisse über den Kontext, mit dem sich der Wissenschaftler beschäftigt, hat. Diese Zusammenarbeit ist leider nicht leicht zu realisieren, da beide Seiten bei der Entwicklung der Software unterschiedliche Ziele verfolgen. Während der Softwareentwickler die Anforderungen und Spezifikationen des Modells benötigt, möchte der Wissenschaftler gerade diese Anforderungen und Spezifikationen mittels der zu entwickelnden Software herausfinden. Softwareentwickler und Wissenschaftler finden somit keinen gemeinsamen Nenner und das Erstellen und Entwickeln der Software erweist sich als besonders schwierig. Dieses Dilemma wird überspitzt in der Abbildung 2.1 dargestellt (Segal und Morris [1]). Während

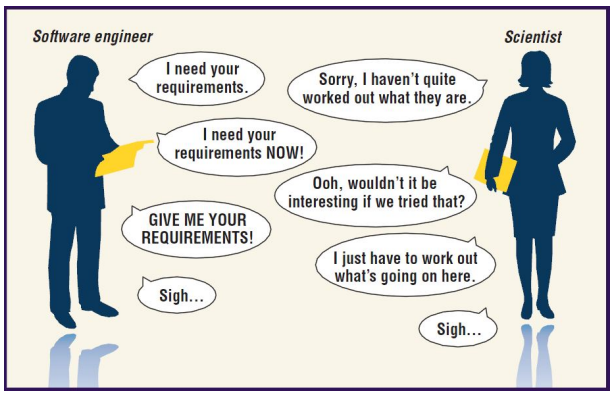

Abbildung 2.1: Zusammenarbeit zwischen Wissenschaft und Softwareentwicklung [1]

in nicht-wissenschaftlichen Kontexten, wie zum Beispiel einer Hotelbuchung, einem Softwareentwickler und dem Anwender unmittelbar klar sein kann, welche Anforderungen

an die Software benötigt werden, ist es für einen Programmierer, der sich normalerweise nicht mit wissenschaftlichen Problemen beschäftigt, schwer, sich in die Materie des Wissenschaftlers einzuarbeiten und diese nachzuvollziehen.
Wenn der Wissenschaftler eine Zusammenarbeit mit einem Softwareentwickler ablehnt, entwickelt er eigenständig seine Software für das Modell. Sein Vorgehensmodell fällt allerdings etwas abstrakter aus als die Vorgehensmodelle, die man als Informatiker aus der Softwareentwicklung und Softwaretechnik kennt. Dieses Vorgehensmodell ist in Abbildung 2.2 zu sehen (Segal und Morris [1]). Ein Softwareentwickler würde wahrscheinlich

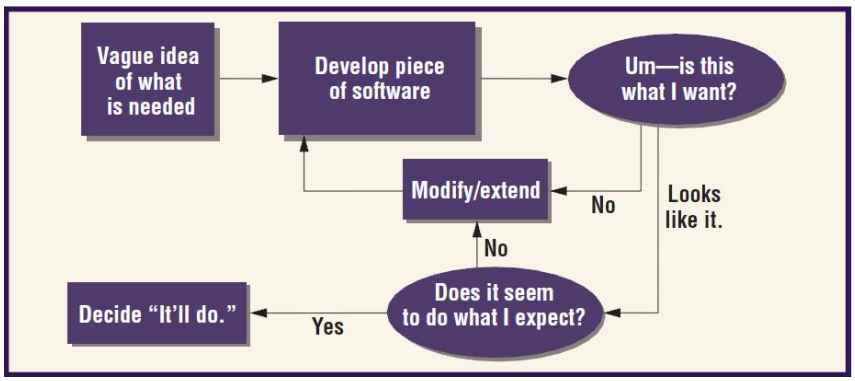

Abbildung 2.2: Wissenschaftliches Vorgehensmodell bei der Softwareentwicklung [1]

nicht nach solch einem Vorgehensmodell programmieren, da ihm andere gelehrt wurden. Tatsächlich wird aber das Vorgehensmodell aus Abbildung 2.2 von Wissenschaftlern praktiziert.

2.2 Anforderungen an wissenschaftliche Software

An jede Software werden Anforderungen gestellt. Besonders wichtig für wissenschaftliche Software sind die Zuverlässigkeit, Portabilität sowie die Effizienz. Diese drei signifikanten Anforderungen werden nun in diesem Abschnitt vorgestellt und genauer erläutert.

2.2.1 Zuverlässigkeit

Unter Zuverlässigkeit versteht man den Grad der Wahrscheinlichkeit, mit dem das Programm seine Funktion erfüllt. Dabei kann man die Zuverlässigkeit in drei weitere Anforderungen gliedern: Korrektheit, Robustheit und Genauigkeit.

Korrektheit

Wenn aus Eingabedaten richtige Ausgabedaten erzeugt werden, dann ist das Programm korrekt. Dies schließt die richtige Abarbeitung von Fehlereingaben (z.B. Tippfehlern) oder Ähnliches mit ein. In wissenschaftlichen Programmen ist dies jedoch nur schwer nachzuprüfen. Man nehme sich als Beispiel ein Programm, das ein lineares Gleichungssystem mittels eines bestimmten Algorithmus (wie das Gaußsche Eliminationsverfahren) löst. Durch einen formalen Korrektheitsbeweis kann nachgewiesen werden, dass dieses Verfahren nur

richtige Resultate liefert. Bei ungünstigen Eingaben kann es passieren, dass Ergebnisse berechnet werden, die nicht in einer einzigen Dezimalstelle mit dem exakten Ergebnis übereinstimmen. Obwohl das Programm korrekt gerechnet hat, weichen die Ergebnisse ab. Ursachen dafür sind die Eigenheiten der Maschinenarithmetik (siehe Abschnitt zur Rechnerarithmetik in Kapitel 4), sowie der Übergang von kontinuierlichen mathematischen Modellen zu diskreten für den Computer bearbeitbaren Modellen (siehe Einleitung in Kapitel 1). Durch diese Einschränkung entstehen zwangsweise Abweichungen in den Ergebnissen, denen man sich bewusst sein muss, wenn man mit Hilfe des Rechners mathematische Probleme löst.

Robustheit

Robustheit ist der Grad eines Programmes, mit dem es Fehler erkennt, für den Benutzer verständlich reagiert, aber trotzdem seine Funktionsfähigkeit wahrt. Es ist maximale Robustheit erreicht, wenn es keine Möglichkeit zur Eingabe gibt, die das Programm zur Fehlerreaktion oder zum Absturz bringt. Robustheit kann man, mit einem kleinen Vorblick auf die Rechnerarithmetik (siehe Kapitel 4), anhand des sogenannten *Overflows* und *Underflows* erklären. Da der Rechner nur endlichen Speicher besitzt, ist natürlich auch die Menge der auf dem Rechner darstellbaren Zahlen endlich. Somit gibt es für diese Menge eine untere Grenze und eine obere Grenze, also die kleinste und größte darstellbare Zahl. Wird die untere Grenze unterschritten, spricht man von einem *Underflow* und das Ergebnis wird auf Null abgerundet. Wird die obere Grenze überschritten, bezeichnet man dies als *Overflow*. Durch die Überschreitung des Wertebereichs kann es passieren, dass das Ergebnis ein anderes Vorzeichen hat, obwohl die Operation eigentlich keinen Vorzeichenwechsel bewirkt (zum Beispiel Addition zweier positiver, sehr großer Zahlen ergibt auf einmal eine negative Zahl). Ein robustes Programm zeichnet sich nun dadurch aus, dass es nach einem Overflow nicht abbricht, sondern den Benutzer darüber informiert. Darüber hinaus hat ein robustes Programm kaum Underflows. Bei einen Underflow lässt sich nämlich schwer feststellen, ob ein Ergebnis nun genau Null war oder fast Null, was bei einigen wissenschaftlichen Anwendungen zum Problem werden kann, wenn die Berechnung auf sehr kleine Zahlen angewiesen ist. Auch die Division durch Null kann zu einem Problem werden.

Genauigkeit

Genauigkeit ist der Grad der Übereinstimmung zwischen angezeigtem und exaktem Wert. Der maximale Genauigkeitsgrad ist genau dann erreicht, wenn das Resultat des Programms *"mit der durch Rundung auf die Maschinenzahlen exakten Lösung genau übereinstimmt"* [2]. Anstatt den maximalen Genauigkeitsgrad anzustreben, genügt es, die Korrektheit bei einer bestimmten Anzahl von Nachkommazahlen sicherzustellen.

2.2.2 Portabilität

Das Mooresche Gesetz besagt, dass die Transistordichte auf Prozessoren und anderen Hardwarekomponenten exponentiell wächst und zu hoher Rechenleistung führt. Dies bringt auch mit sich, dass Hardware schnelllebig und innerhalb kürzester Zeit veraltet. Wenn

Hardware - zum Beispiel wegen Reparatur - ausgetauscht wird, ist sie neuer als die Hardware, auf der ein bestimmtes Programm geschrieben wurde. Softwareentwicklung ist teuer und zeitaufwendig, deshalb soll die Software auch auf neue Hardware integrierbar sein, ohne dass man sie umschreiben oder im schlimmsten Falle neu schreiben muss. Man sagt, die Software soll *portabel* sein. Dabei ist die Software unabhängig vom spezifischen Computer - und/oder Prozessortypen.

2.2.3 Effizienz

Effizienz ist eines der Hauptqualitätsmerkmale in technisch-naturwissenschaftlichen Programmen. Dabei unterscheidet man im Wesentlichen zwischen Laufzeiteffizienz und Speichereffizienz. Laufzeiteffizienz bezieht sich auf die zur Problemlösung benötigte Zeit. Das Programm soll für die Bearbeitung der Aufgabe möglichst wenig Zeit aufwenden. Dies kann von verschiedenen Faktoren abhängen, wie zum Beispiel von der Leistung der Hardware, der gewählten Programmiersprache und von parallelen Prozessen, die die Rechenzeit beeinflussen. Die Speichereffizienz bezieht sich auf die effiziente Nutzung des Speichers. In Kapitel 3 wird im Detail auf die effiziente Speicherausnutzung eingegangen, unter anderem mit Hilfe der Speicherhierarchie.

2.2.4 Effizienz und Zuverlässigkeit

So wichtig Effizienz auch ist, erhält die Zuverlässigkeit eine höhere Priorität und zwar aus folgenden Gründen: Unzuverlässige effiziente Software ist wertlos, da ein Programm, das das Problem zwar innerhalb kürzester Zeit löst, jedoch falsche Ergebnisse liefert, für die wissenschaftliche Anwendung vollkommen nutzlos ist. Aufgrund von unzuverlässigen Programmteilen kann der Entwicklungsaufwand des Programms stark erhöht werden, um diese erst einmal aufzuspüren und dann zu korrigieren. Dies kostet nicht nur Zeit, sondern auch Geld. Effizienz ist natürlich auch sehr wichtig, ist aber leichter zu realisieren als Zuverlässigkeit, sodass beim Programmieren an erster Stelle die Zuverlässigkeit sichergestellt werden muss.

2.3 Software Design

Zur erfolgreichen Entwicklung von Software gehört ein strukturierter Prozess, der die Planung der Aufgabe, die Implementierung des Modells, das Testen des Programms, sowie die Betrachtung seiner Leistung bezüglich Speicherausnutzung und die Dokumentation während des Entwicklungsprozesses mit einbezieht. Diese Schritte erfolgen in keinem Fall sequentiell, sondern können parallel und mehrmals ausgeführt werden. Die einzelnen Schritte werden hier aufgeführt und erläutert.

2.3.1 Planung

In einem großen Software-Projekt ist es meist von Vorteil, die grobe Struktur des Produkts aufzuzeichnen und Programmierer zu organisieren, die arbeitsteilig die Komponenten der Software schreiben. Mit einem “Einfach-Drauf-Los”-Vorgehen kommt man in einem großen

Softwareprojekt meist nicht weit. Die Planung umfasst die Spezifikation der Anforderungen wie zum Beispiel die Anforderung an den Speicherverbrauch, die Performanz und die Ein -und Ausgabe der Software. Zudem kann man sich überlegen, ob es Bibliotheken gibt, die man für sein Programm benutzen kann. Die Verwendung von Bibliotheken beschleunigt die Entwicklung des Projekts und die Programmierer können sich auf die wesentlichen Dinge im Projekt konzentrieren. Dabei muss man jedoch über das Interface der Bibliothek Bescheid wissen, um die Kommunikation zwischen ihr und der zu entwickelnden Software sicherzustellen. Zu der Planung gehört natürlich auch die Festlegung einer oder auch mehrerer Programmiersprachen. Welche Programmiersprache gewählt werden soll beziehungsweise muss, hängt von dem Kontext, in dem man sie nutzen will, aber auch von der Erfahrung mit einer Programmiersprache, ab. Grundsätzlich gibt es drei typische Kategorien von Programmiersprachen: High-Level-Sprachen, Skriptsprachen und symbolische Sprachen. Unter High-Level-Sprachen fallen Sprachen wie C, C++ und Fortran. Fortran ist ausschließlich für numerische Berechnung geeignet, C für System -und Anwendungsprogrammierung und C++ ist eine Mehrzwecksprache, die Programmierparadigmen, wie die objektorientiere, prozedurale und generische Programmierung, unterstützt. Meist lohnt es sich auch, Programme zu parallelisieren, damit sie in deutlich geringerer Zeit ausgeführt werden können. Dafür gibt es das *"Message Passing Interface"*, welches nur für High-Level-Sprachen existiert. Skript-Sprachen sind Perl, Python oder auch Shell und finden ihren Einsatz bei der Analyse und Verarbeitung von großen Datenmengen, aber auch bei der Jobsteuerung. Symbolische Sprachen wie MatLab oder Mathematica sind zur Lösung von mathematischen Problemen und zur graphischen Darstellung von Funktionen und Ergebnissen gedacht. Natürlich können bei Bedarf mehrere Programmiersprachen kombiniert werden. Dabei muss man natürlich wissen, wie die Programmiersprachen miteinander kommunizieren können (Beispiel: Wie wird eine Fortran - Routine aus einem C - Programm aufgerufen?).
Ein wichtiger Aspekt, der bei der Planung mit beachtet werden soll, ist die Portabilität der Software. Wie in Abschnitt 2.2.2 erwähnt, ist Hardware schnelllebig, sodass sichergestellt werden muss, dass die zu entwickelnde Software auch auf neueren Hardware - Komponenten ausgeführt werden kann.

Bibliotheken

Bibliotheken sind eine Sammlung von Funktionen und Datentypen in Hilfsmodulen zusammengefasst, die aber keine eigenständigen lauffähigen Einheiten sind, sondern in das eigene Programm mit eingebunden werden können. Es sollen ein paar wichtige Bibliotheken genannt werden, die auch hilfreich für die wissenschaftliche Softwareentwicklung sind:

- **Builtin C/C++ Bibliothek**
  - *stdio.h*
  - *math.h*
  - *stdlib.h*
  - *time.h*
- **GNU Scientific Library**

- **Numerical Recipes**

Die *GNU Scientific Library* enthält viele Funktionen für die Arithmetik mit komplexen Zahlen, Vektoren und Matrizen oder auch für die Signalverarbeitung (zum Beispiel die schnelle Fouriertransformation). *Numerical Recipes* enthält eine große Sammlung von Routinen für viele numerische Berechnungen.

2.3.2 Implementierung

Die Implementierung beschäftigt sich mit dem tatsächlichen Schreiben des Programmes. Dabei einigen sich die Programmierer auf einen vordefinierten Stil, zum Beispiel die Benennung der Variablen. Wird abweichend davon eine andere Konvention benutzt, muss die Bedeutung der Variable im Quellcode dokumentiert werden, sodass ein anderer Programmierer, der ebenfalls an dieser Stelle arbeitet, sofort um die Bedeutung Bescheid weiß. Besonders durchgesetzt hat sich die modulare Programmierung. Dies bedeutet, dass Codeteile, die unabhängig voneinander funktionieren können, in unteilbare Einheiten eingeordnet werden. Modularität verbessert die Übersicht des Programms. Zudem können Module einfacher in andere Programme geladen werden als einzelne Codeteile.

2.3.3 Testen und Debugging

Wenn ein Programm von einem Compiler übersetzt wird, wird es in den meisten Fällen nicht sofort funktionieren. Dies bedeutet nun für den Programmierer, dass er Fehler, seien es syntakische oder semantische, in seinem Programm suchen muss. Testen ist ein wichtiger Bestandteil der Softwareentwicklung und findet parallel zu allen Stufen des Entwicklungsprozesses statt. In der modernen Softwareentwicklung wird sogar schon getestet, bevor es überhaupt ein Stück lauffähige Software gibt. Es gibt verschiedene Teststufen - und methoden, die zum Beispiel in einem sogenannten *V - Modell* zusammengefasst sind [10].
Um sein Programm zu inspizieren, kann sich der Programmierer sogenannten Debugging - Werkzeugen bedienen, mit denen er den Programmablauf steuern kann, indem er zum Beispiel Haltepunkte (englisch: Breakpoints) setzt. Solche Werkzeuge helfen ihm auch, den Speicher zu untersuchen und ungültige Speicherzugriffe zu finden. Die bekanntesten Debugger sind der *GNU Debugger* und *Valgrind*. *Valgrind* besitzt unter anderem ein nützliches Tool mit dem Namen *memcheck*, welches herausfinden kann, ob auf nicht initialisierten Speicher lesend oder schreibend zugegriffen wird oder ob über Speichergrenzen hinaus geschrieben wird.

2.3.4 Leistungsanalyse

Hat man nun eine lauffähige Version seines Programmes, kann man sich über das Laufzeitverhalten des Programms Gedanken machen. Dafür gibt es sogenannte Profiling - Werkzeuge, die dem Programmierer bei der Analyse des Programms helfen, um bestimmte Problembereiche zu finden, die die Ursache für ineffiziente Programme sein können. Leistungssteigerungen kann man zum Beispiel bei der Geschwindigkeit und der Speicherausnutzung erreichen. Um die Geschwindigkeit zu ermitteln, werden die Aufrufe von Funktionen gezählt und ihre Ausführungszeit gemessen. Aufgrund ihrer Laufzeit kann man nun

entscheiden, ob sich eine Optimierung lohnt. Wird eine Funktion sehr oft aufgerufen, kann eine Optimierung sehr sinnvoll sein, aber auch wenn eine Funktion nur selten aufgerufen, dafür aber lange ausgeführt wird. Auch zu gunsten der Speicherausnutzung kann optimiert werden, indem der Verbrauch von Arbeitsspeicher untersucht und gegebenenfalls angepasst wird (zur Speicherausnutzung siehe auch Kapitel 3). Die Leistung kann auch in Bezug auf die Fliesskommaoperationen, die ein Rechner während der Programmausführung in einer Sekunde schafft, bewertet werden (*Floating Point Operations Per Seconds*).

2.3.5 Dokumentation

In einem gut organisierten Softwareentwicklungsprojekt wird stets das Vorgehen, der Programmcode sowie die Testvorgänge kommentiert und dokumentiert. Bei der Entwicklungsdokumentation wird die Aufgabenstellung sowie die Lösung der Aufgabe festgehalten. Dazu gehören auch theoretische Grundlagen, die Randbedingungen, die Vorschriften und auch die Anforderungen an das Programm. Auch die Beschreibung eines Ablaufplans gehört dazu, sowie das Vorgehensmodell, an das man sich beim Programmieren halten will. Auch die Testprotokolle werden dokumentiert. Diese Arten von Dokumentationen sind für die Entwickler gedacht.
Die Programmdokumentation ist für den Benutzer gedacht. Sie beschreibt das Softwareprodukt, seinen Aufbau und seine Verwendung. Auch der Einsatzbereich der Software wird in der Dokumentation klargestellt, sowie die gültigen Eingabedaten und die passenden Ausgabedaten. Auch mögliche Fehlermeldungen gehören in eine Dokumentation, sowie die dazugehörigen Maßnahmen zu deren Behandlung.

3 Hardware

Wie bereits in Kapitel 2 erwähnt, ist Effizienz eine, nach der Zuverlässigkeit, der wichtigsten Anforderungen an wissenschaftliche Software. Besonders interessant ist hierbei die Speichereffizienz, die sich mit der optimalen Speicherausnutzung beschäftigt. In diesem Kapitel wird erläutert, wie mit Hilfe der Speicherhierarchie der Speicher am besten genutzt werden kann im besonderen Hinblick auf die Lokalität von Programmen.

3.1 Effizienz auf Hardware-Ebene

Die Kenntnis über das Speicherverhalten des Rechners kann überaus nützlich sein, wenn man speichereffizient programmieren will. Die Effizienz wird nicht nur an der Anzahl der Schritte gemessen, die ein Programm zur Ausführung braucht, sondern auch, wie lange es braucht, um diese Rechenschritte auszuführen. Die Zeit für den Zugriff auf den Speicher kann schwanken (von Nanosekunden bis Millisekunden) und damit auch die Ausführungszeit des Programms. Diese Schwankung ist dadurch begründet, dass es keinen Speicher gibt, die groß und schnell zugleich sind. Deshalb wird der Speicher in Computern als Speicherhierarchie realisiert, wie sie in Abbildung 3.1 zu sehen ist. Kleinere Speicher wie Register und Cache sind näher am Prozessor und größere Speicher sind weiter vom Prozessor entfernt, wie der Hauptspeicher und der Plattenspeicher (Abbildung 3.1: Virtueller Speicher). Je näher der Speicher am Prozessor ist, umso schneller erfolgt der Zugriff auf ihn. Wird auf Daten im Speicher zugegriffen, wird erst in der Ebene gesucht, die dem Prozessor am nächsten ist (der niedrigsten Ebene). Sind die Daten dort nicht aufzufinden, wird in der nächsten Ebene gesucht. Programme besitzen die sogenannte Lokalitätsei-

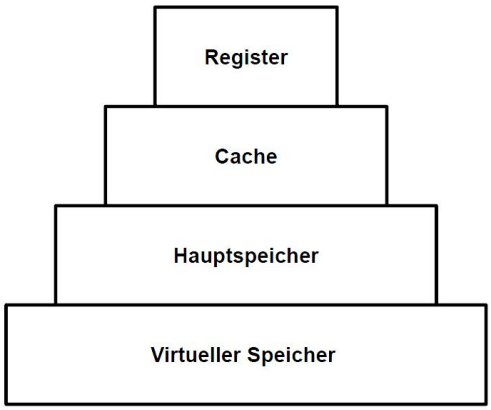

Abbildung 3.1: Speicherhierarchie

genschaft. Dies bedeutet, dass sie nur auf einen geringen Teil der gesamten Datenmenge zugreifen [3]. Bekannt ist dabei auch die 90/10 - Faustregel [11]:

Programme führen 90% ihrer Instruktionen in 10% des Programmcodes aus

Gründe dafür sind unter anderem Ausführungen innerer Schleifen, sowie rekursiver Aufrufe oder auch Code für Fehlerbehandlungen, der für die Sicherstellung der Korrektheit des Programmes zwar sehr wichtig ist, aber in nur seltenen Fällen aufgerufen wird. Die Lokalitätseigenschaft wird unterteilt in zeitliche und räumliche Lokalität. Zeitliche Lokalität bedeutet, dass auf Daten, auf die zugegriffen wurde, in einer gewissen Zeitspanne mit hoher Wahrscheinlichkeit noch einmal zugegriffen wird. Räumliche Lokalität bezieht sich hierbei auf einen Ort auf den zugegriffen wurde. Mit hoher Wahrscheinlichkeit wird in der Nähe dieses Ortes innerhalb einer gewissen Zeit noch einmal zugegriffen. Mit dem Wissen, dass die Programme Lokalität besitzen, lässt sich nun effizient die Speicherhierarchie ausnutzen und die mittlere Speicherzugriffszeit verringern, indem Daten, die häufig gebraucht werden, in den kleinen Speicher verlegt werden, sodass sie sich nah am Prozessor befinden und schnell aufgerufen werden können.

4 Rechnerarithmetik

Reelle Zahlen sind die Grundlage für wissenschaftliche Berechnungen. Wie bereits in der Einleitung gezeigt, beschäftigt sich die Wissenschaft mit kontinuierlichen Zusammenhängen, die im Rechner nicht exakt übernommen werden können, sondern, aufgrund des endlichen Speichers, diskretisiert werden müssen. Da in der Menge der reellen Zahlen zwischen zwei Zahlen unendlich viele reelle Zahlen liegen, stellt sich nun die Frage, wie im Rechner der unendliche Zahlenbereich optimal abgedeckt werden kann. Die Lösung ist die sogenannte Gleitkommazahl, die in diesem Kapitel Schritt für Schritt eingeführt werden soll, indem zunächst die b-adische Entwicklung vorgestellt und erläutert wird. Anschließend wird die Rundung betrachtet, indem sie formal definiert und analysiert wird, welche Fehler entstehen, wenn gerundet wird.

4.1 Maschinenzahlen und b-adische Entwicklung

Um eine darstellbare Zahlenmenge für den Rechner zu finden, legt man zunächst fest, dass diese Menge endlich sein soll (siehe Definition 4.1.1).

Definition 4.1.1. *Die endliche Menge* $\mathbb{M} \subset \mathbb{R}$ *sei die Menge der Maschinenzahlen.*

Wie diese Maschinenzahlen nun dargestellt werden, geht in Definition 4.1.1 noch nicht hervor. Mit Hilfe der b - adischen Entwicklung soll diese Darstellung nun entwickelt werden.

Die b-adische Entwicklung

Bei der b-adischen Darstellung gibt es eine Grundmenge $\mathbb{B}$ mit den verfügbaren Symbolen zur Darstellung jeder beliebig großen Zahl: $\mathbb{B} = \{0, 1, \ldots, b-1\}$. Die Grundmenge wird auch Basis genannt. Jede der b Ziffern ist bijektiv (eineindeutig) eine der Zahlen von 0 bis $(b-1)$ zugeordnet. Die b-adische Entwicklung wird auch als Stellenwertsystem bezeichnet, da der Wert einer Ziffer durch ihre Position in der Zahl bestimmt wird. Es ist dabei wichtig, zwischen den Begriffen *Ziffer* und *Zahl* zu unterscheiden. Die Zahl bezeichnet den Gesamtwert, der sich durch die Anordnung der Ziffern ergibt. Eine Ziffer ist ein Symbol aus dem Zeichenvorrat $\mathbb{B}$. Je nachdem wie die Ziffern angeordnet sind, ergibt sich ein anderer Wert für die Zahl.

Beispiel 4.1.1. *Man betrachte als einfaches Beispiel das Dezimalsystem mit der Grundmenge* $\mathbb{B} = \{0, \ldots, 9\}$. *Die Ziffern* 2, 3 *und* 4 *lassen sich auf verschiedene Weise anordnen:* 234, 324, 432 *usw. Man sieht, je nachdem, wie die Ziffern angeordnet werden, hat die Zahl einen kleinen oder großen Gesamtwert (denn* $234 < 324 < 432$*).*

Mit der Grundmenge $\mathbb{B}$ können nun die natürlichen, ganzen und rationalen Zahlen mittels der b-adischen Entwicklung dargestellt werden, wie es in den Definitionen 4.1.2, 4.1.3 und 4.1.4 gezeigt ist.

Definition 4.1.2 (Natürliche Zahlen)**.** *Eine natürliche Zahl lässt sich mittels der b-adischen Entwicklung durch* $S_n = \sum_{i=0}^{n} a_i b^i = a_0 + a_1 b^1 + \ldots + a_n b^n$ *darstellen, wobei* $a_i \in \mathbb{B}$ *und* $b = |\mathbb{B}|$.

Natürliche Zahlen werden als Summe des Produkts von Ziffer aus der Grundmenge $\mathbb{B}$ und der Kardinalität b der Menge $\mathbb{B}$ dargestellt. Beim Dualsystem sind beispielweise $a_i \in \{0, 1\}$ und $b = |\mathbb{B}| = |\{0, 1\}| = 2$. Das i gibt dabei die Position (die Stelle) der Ziffer in der Zahl an. In der b-adischen Entwicklung werden die natürlichen Zahlen als Folge $a_n a_{n-1} \ldots a_2 a_1 a_0$ dargestellt und sind der Summe S_n zugeordnet. a_n ist dabei die höchstwertige Ziffer und a_0 die Ziffer mit dem kleinsten Wert in der endlichen Folge.

Definition 4.1.3 (Ganze Zahlen)**.** *Eine ganze Zahl lässt sich mittels der b-adischen Entwicklung durch* $S_n = \pm \sum_{i=0}^{n} a_i b^i = \pm(a_0 + a_1 b^1 + \ldots + a_n b^n)$ *darstellen, wobei* $a_i \in \mathbb{B}$ *und* $b = |\mathbb{B}|$.

Die Menge der ganzen Zahlen zeichnet sich dadurch aus, dass sie zu den positiven (natürlichen) Zahlen,noch negative enthält. Die Definition der natürlichen Zahlen muss dementsprechend um ein negatives Vorzeichen erweitert werden.

Definition 4.1.4 (Rationale Zahlen)**.** *Eine rationale Zahl lässt sich mittels der b-adischen Entwicklung durch* $S_n = \pm \sum_{i=0}^{n} a_i b^i \pm \sum_{i=-\infty}^{-1} a_i b^i$ *darstellen, wobei* $a_i \in \mathbb{B}$ *und* $b = |\mathbb{B}|$.

Die Menge der rationalen Zahlen besitzt, neben den vorzeichenbehafteten (ganzen) Zahlen, noch einen gebrochenen Teil, den Nachkommateil. Die Werte hinter dem Komma werden mit b^{-i} multipliziert, wobei i die Position nach dem Komma angibt. Das Komma ist zudem ein fester Bestandteil der Zahl. Dies wird im nachfolgenden Abschnitt (4.1 Gleitkommazahlen) weiter erläutert.

Darstellung der reellen Zahlen - Gleitkommazahlen

Nachdem nun eine Grundlage für die Darstellung von natürlichen, ganzen und rationalen Zahlen im vorherigen Abschnitt geschaffen wurde, stellt sich die Frage, welche Darstellung eigentlich für die reellen Zahlen am besten geeignet ist. Wie man bereits bei rationalen Zahlen erkennen konnte, haben diese einen Nachkommateil, der jedoch (wenn man Periodizität hinter dem Komma ausschließt) endlich ist. Die Menge der reellen Zahlen kann jedoch unendlich viele Nachkommastellen besitzen. Zudem kommt noch hinzu, dass die Stelle, an der sich das Komma in der Zahl befindet, mitgespeichert werden muss, da sie ein fester Bestandteil der reellen Zahl darstellt. Im Rechner wird das sogenannte Dualsystem benutzt, das die Ziffern 0 und 1 besitzt, um Zahlen darzustellen. Für das Komma steht also keine Ziffer mehr zur Verfügung. Man muss sich einer Schreibweise bedienen, die die Position des Kommas implizit angibt. Dabei kann man sich an der wissenschaftlichen Darstellung von Zahlen orientieren, wie sie zum Beispiel in der Mathematik oder Physik benutzt wird. Diese Schreibweise wird Exponentialschreibweise genannt und hat folgende Form:

$$Z = Vorzeichen \cdot Mantisse \cdot Basis^{Exponent}$$

Die Basis ist die zuvor beschriebene Grundmenge $|\mathbb{B}|$, die die Ziffern zur Zahldarstellung enthält. Die Mantisse ist die darzustellende Zahl für die mittels des Exponenten ihre Größenordnung festgelegt wird, d.h. die Position des Kommas.

Beispiel 4.1.2. *Mittels der Exponentialschreibweise kann man die Zahl* 123 *im Dezimalsystem in die Darstellungen:*

- $123 \cdot 10^0$
- $12.3 \cdot 10^1$
- $1.23 \cdot 10^2$
- ...

konvertieren. Die drei Darstellungen sind äquvalent und haben denselben Wert.

Eindeutigkeit

Wie man an Beispiel 4.1.2 erkennen kann, gibt es mehrere Möglichkeiten zur Darstellungen für eine Zahl in der Exponentialschreibweise und sie ist somit nicht eindeutig. Eine eindeutige Darstellung für die Maschinenzahlen ist im Rechner jedoch notwendig. Deshalb muss *Eindeutigkeit* hergestellt werden, indem die Zahl normiert wird.

Definition 4.1.5 (Eindeutigkeit)**.** *Eine Zahl in der Darstellung*

$$Z = Vorzeichen \cdot Mantisse \cdot Basis^{Exponent}$$

ist normiert, wenn gilt:

$$1 \leq Mantisse < Basis$$

Da im Rechner die Zahlen im Dualsystem dargestellt werden, bedeutet dies, dass, aufgrund der Normierung aus Definition 4.1.5, vor dem Komma immer eine 1 steht. Die Basis ist im Dualsystem $b = 2$, also muss die Mantisse echt kleiner sein als 2, jedoch größer oder gleich 1. Somit muss die Vorkommastelle nicht explizit mitgespeichert werden, da sie immer gleich 1 ist. Es werden dann nur die Ziffern nach dem Komma gespeichert. Durch diese Einsparung können die Mantissenbits optimal ausgenutzt werden.

Der IEEE-Standard

Die reellen Zahlen, die im Rechner abgespeichert werden, werden als numerisch reelle Zahlen bezeichnet, die besser als Gleitkommazahl bekannt sind. Der *IEEE-Standard 754* [4] legt Standarddarstellungen für die Gleitkommazahlen sowie für die mathematischen Operationen und Rundung (Abschnitt 4.2) fest. Der Standard ist in den meisten (aber nicht allen) Programmiersprachen und Rechnersystemen implementiert. Er definiert unter anderem zwei grundlegende Arten für die Genauigkeiten der Zahlen: Die einfache und die

	Vorzeichen	Mantisse	Exponent	Wert des Exponenten	Biaswert
Single	1 Bit	23 Bit	8 Bit	$-126 \leq e \leq 127$	127
Double	1 Bit	52 Bit	11 Bit	$-1022 \leq e \leq 1023$	1023

Tabelle 4.1: IEEE 754 - Standard [4]

doppelte Genauigkeit. Es gibt noch zwei weitere Arten - die erweitert einfache und erweitert doppelte Genauigkeit. Diese sollen hier nicht besprochen werden, da die einfache und doppelte Genauigkeiten in dieser Seminararbeit als Beispiel genügen sollen. Für genauere Informationen zu den anderen Typen der Genauigkeiten siehe [4]. Die einfache Genauigkeit reserviert 32 Bit und die doppelte Genauigkeit 64 Bit für die Speicherung der Zahl. Die Aufteilung der Bits für Mantisse, Exponent und Vorzeichen ist in Tabelle 4.1 aufgeführt. Die Anzahl der Exponentenbits legt den Wertebereich der darstellbaren Zahl, die Anzahl der Mantissenbits legt die Genauigkeit fest, wobei 2^e die größte darstellbare Zahl und 2^{e-1} die kleinste darstellbare Zahl mit Exponenten e definiert. Der Exponent wird in der sogenannten Bias-/Exzessdarstellung gespeichert. Je nach Genauigkeit (einfach oder doppelt) wird ein sogenannter konstanter Biaswert zu dem Exponenten hinzuaddiert, was das Abspeichern erleichtert und einen negativen Exponenten verhindert. Die Mantisse ist wie in Definition 4.1.5 normiert, sodass das erste (und einzige) Vorkommabit nicht mitgespeichert werden muss. Dieses Bit wird als *Hidden Bit* bezeichnet.

Schlussbetrachtung

Um abschließend eine Darstellung für reelle Zahlen im Rechner festzulegen, werden die Eigenschaften der Gleitkommazahl im Satz 4.1.1 zusammengefasst.

Satz 4.1.1 (Gleitkommazahl)**.** *Eine reelle Zahl x wird im Computer als Gleitkommazahl $float(x) \in \mathbb{M}$ dargestellt in der Form:*

$$float(x) = (-1)^s \cdot (0.a_1a_2 \dots a_n) \cdot 2^e, a_1 \neq 0, s \in \{0, 1\}$$

Die Menge der Maschinenzahlen bzw. der Gleitkommazahlen kann somit als 4 - Tupel beschrieben werden:

$$\mathbb{M}(2, n, MIN, MAX)$$

wobei n die Nachkommastellen der Mantisse, MIN/MAX die minimal/maximal darstellbare Zahl und e den Exponenten bezeichnen. Dabei ist $MIN = 2^{e-1}$ und $MAX = 2^e$.

4.2 Rundung

Wenn eine Zahl abgespeichert werden soll, die nicht in der Menge der Maschinenzahlen liegt, muss gerundet werden. Dies kann beispielsweise bei einer arithmetischen Operation passieren, bei der das Ergebnis eine höhere Genauigkeit benötigt als der *IEEE-754-Standard* bieten kann. Damit das Ergebnis jedoch abgespeichert werden kann, muss so gerundet werden, dass die Zahl in das vorgegebene Zahlenformat passt (z.B. 32 Bit, einfache Genauigkeit).

4.2.1 gibt eine allgemeine Definition der Rundung [5]. Es müssen dabei zwei Bedingungen bei der Abbildung von reellen Zahlen auf Maschinenzahlen erfüllt sein [5]:

1. *Idempotenz*: Die Rundung einer Maschinenzahl ist wieder eine Maschinenzahl (siehe 4.1).
2. *Monotonie*: Wenn eine reelle Zahl r kleiner oder gleich einer anderen reellen Zahl s ist, so muss auch die Rundung von r kleiner oder gleich der Rundung von s sein (siehe 4.2).

Definition 4.2.1 (Rundung)**.** *Rundung ist eine Abbildung* $rd : \mathbb{R} \to \mathbb{M}$ *wenn die folgenden Bedingungen*

$$rd(m) = m, \forall m \in \mathbb{M} \tag{4.1}$$

und

$$r \leq s \Rightarrow rd(r) \leq rd(s), \forall r, s \in \mathbb{R} \tag{4.2}$$

erfüllt sind.

Bekannte Rundungsarten sind das Abschneiden von (überschüssigen) Nachkommastellen und das mathematische Runden. Beim Abschneiden von Nachkommastellen werden rechts von einer Ziffer, die erhalten bleiben soll, alle anderen Ziffern "abgeschnitten", sodass die Zahl wieder in das vorgeschriebene Zahlenformat passt. Das Problem dabei ist, dass durch das Abschneiden sehr schnell Genauigkeit verloren geht und Rundungsfehler (siehe auch Abschnitt 4.2.1) sehr groß werden können. Das mathematische Runden wird im *IEEE-754-Standard* für Gleitkommazahlen genutzt. Bei diesem Verfahren wird vorgesehen, dass bei Zahlen der Form $x.5$ (Dezimalsystem) oder $x.1$ (Dualsystem) zur nächsten geraden oder ungeraden Zahl gerundet wird.

4.2.1 Rundungsfehler

Meist lässt sich das Runden nicht verhindern. Wird gerundet, muss darauf geachtet werden, dass der Rundungsfehler möglichst gering bleibt. Ein hoher Rundungsfehler bedeutet auch einen hohen Informationsverlust. Wird zudem mit diesen Rundungsfehlern weitergerechnet, so summieren sich die Fehler und das Endergebnis kann komplett verfälscht werden. Mit Hilfe einer sogenannten Fehleranalyse lassen sich solche Rundungsfehler untersuchen. Sie lassen sich zwar leider aufgrund der Diskretisierung der kontinuierlichen Modelle nicht verhindern, man kann sie jedoch minimieren.

Fehleranalyse

Bei der Fehleranalyse betrachtet man zwei Fehlertypen [5]:

- den absoluten Fehler: $|x - rd(x)|$
- den relativen Fehler: $\frac{|x-rd(x)|}{|x|}$

wobei $x \in \mathbb{R}$ und $rd(x) \in \mathbb{M}$. Dies führt zu folgenden Satz:

Satz 4.2.1. *Gegeben sei* $\mathbb{M}(b, n, MIN, MAX)$ *und* $x \in \mathbb{R}$ *mit* $MIN \leq |x| \leq MAX$. *So gilt:*

$$\frac{|x - rd(x)|}{x} \leq \frac{1}{2} b^{1-n} =: eps \tag{4.3}$$

und

$$rd(x) = x(1 + \delta) \tag{4.4}$$

mit $|\delta| \leq eps$

eps ist dabei der maximale Fehler, der beim Runden auftreten kann. Dies soll nun in Beweis 4.2.1 [5] für eine beliebige Basis $b \geq 2$ und einer beliebig großen Mantisse n gezeigt werden.

Beweis 4.2.1. *Es soll stets zur nächstliegenden Maschinenzahl gerundet werden. Daraus ergibt sich, dass sich* x *und* $rd(x) = a_0.a_1 \dots a_{n-1} \cdot b^e$ *maximal um eine halbe Einheit in der letzten Mantissenstelle unterscheiden können. Also gilt für den absoluten Fehler:*

$$|x - rd(x)| \leq \tfrac{1}{2} \cdot b^{-(n-1)} \cdot b^e$$

Aufgrund der Normierung aus Definition 4.1.5 folgt allerdings $a_0 \geq 1$ *und somit*

$$|x| \geq b^e$$

Also:

$$\begin{aligned} |x - rd(x)| &\leq \frac{1}{2} \cdot b^{-(n-1)} \cdot b^e \\ &\leq \frac{1}{2} \cdot b^{-(n-1)} \cdot |x| \end{aligned}$$

Durch Division mit $|x| \geq MIN > 0$ *erhält man nun 4.3.*

Die Beziehung 4.4 lässt sich nun leicht beweisen.

Beweis 4.2.2. *Sei nun* $\delta := \frac{rd(x)-x}{x}$ *mit* $|\delta| \leq eps$. *Dann folgt mittels einfacher Umformung:*

$$\begin{aligned} \delta &= \frac{rd(x) - x}{x} \\ \delta x &= rd(x) - x \\ \delta x + x &= rd(x) \\ x(1 + \delta) &= rd(x) \end{aligned}$$

Es wurde gezeigt, wie groß der Fehler beim Runden tatsächlich werden kann. Abschließend sei noch erwähnt, dass die Größe *eps*, also der maximale relative Fehler, auch als Maschinengenauigkeit bezeichnet wird und den relativen Abstand zwischen zwei Gleitkommazahlen darstellt.

5 Numerik

Bevor kontinuierliche mathematische Gleichungen auf dem Rechner gelöst werden können, müssen diskrete Verfahren entwickelt werden, die die Gleichungen lösen, ohne auf der Menge der reellen Zahlen, sondern auf der Menge der Maschinenzahlen zu rechnen. Die Numerik beschäftigt sich genau mit dieser Situation, indem sie Algorithmen für mathematische kontinuierliche Probleme konstruiert und analysiert.

Beispiel 5.0.1. *Ein Beispiel für ein numerisches Verfahren ist das Simpson-Verfahren zur Lösung eines Integrals. Eine Funktion $f(x)$ wird dabei näherungsweise integriert, indem sie durch eine Parabel genähert wird. Die Parabel wird als sogenanntes Interpolationspolynom genutzt und das Integral der Funktion $f(x)$ wird dann durch das Integral der Parabel angenähert [6].*

Sind solche Verfahren entwickelt, werden sie analysiert und gegebenenfalls optimiert. Bei der Diskretisierung treten natürlich Fehler auf. Die Aufgabe der Numerik ist es, diese Fehler zu erkennen und zu minimieren.

5.1 Fehlerbewertungsmechanismen

In der Numerik werden unter anderem drei wichtige Fehlerbewertungsmechanismen angewandt:

- Kondition eines Problems
- Stabilität eines Algorithmus
- Konsistenz eines Algorithmus

Wodurch sich diese Mechanismen auszeichnen und voneinander unterscheiden wird in den folgenden Abschnitten erläutert.

5.1.1 Kondition eines Problems

Die Kondition bezeichnet eine Eigenschaft des Problems und bezieht sich nicht auf einen konkreten Algorithmus, der das Problem lösen soll. Dabei wird mathematisch untersucht, inwiefern sich gestörte Eingabedaten auf das Problem unabhängig vom gewählten Algorithmus auswirken. Die Kondition ist also die Empfindlichkeit der Lösung. Man betrachtet das Problem dargestellt als Funktion $f : D \to W$ mit einem $x \in D$, wobei x eine Eingabe frei von Störungen bezeichnet. Sind die Eingabedaten verfälscht, bezeichnet man dies als $\tilde{x}$. Somit verfälscht sich auch $f(x)$ und ist dann $f(\tilde{x})$. Nun wird der sogenannte Datenfehlereffekt untersucht, das bedeutet der Fehler $|f(x) - f(\tilde{x})|$. Dies ist die Abweichung des Fehlers von dem ungestörten Ergebnis. Ist diese Abweichung groß, ist das Problem schlecht konditioniert, sonst hat das Problem eine gute Kondition und gestörte Eingabedaten verstärken den Fehler nur minimal in der Lösung.

5.1.2 Stabilität eines Algorithmus

Die Stabilität ist eine Eigenschaft eines ausgewählten Algorithmus. Sie beschreibt, wie unempfindlich der Algorithmus gegenüber von Störungen in den Daten ist. Ein gutes Beispiel dafür ist der Rundungsfehler. Ist ein Algorithmus stabil, haben Rundungsfehler in der Rechnung kaum einen Einfluss auf das Endergebnis. Man untersucht den Fehler zwischen dem kontinuierlichen Verfahren und dem angenäherten Verfahren unter gestörten Eingabedaten: $|f(\tilde{x}) - \tilde{f}(\tilde{x})|$. Je kleiner diese Abweichung ist, umso stabiler ist der Algorithmus.

5.1.3 Kondition und Stabilität

Alle drei Eigenschaften bzw. Mechanismen hängen stark miteinander zusammen. Besonders gut lässt sich mathematisch die Beziehung zwischen Kondition und Stabilität zeigen.

Beweis 5.1.1. *Es soll der Zusammenhang zwischen der Kondition eines Problems und der Stabilität eines Algorithmus gezeigt werden. $f(x)$ bezeichnet hierbei das mathematische Verfahren unter der Eingabe x und $\tilde{f}(\tilde{x})$ der an das mathematische Verfahren angenäherte Algorithmus unter der gestörten Eingabe $\tilde{x}$. Es soll nun der Fehler*

$$|f(x) - \tilde{f}(\tilde{x})| \tag{5.1}$$

abgeschätzt werden. Umschreiben von 5.1 durch Addieren von $0 = f(\tilde{x}) - f(\tilde{x})$ und Anwendung der Dreiecksungleichung:

$$|f(x) - \tilde{f}(\tilde{x})| = |f(x) - f(\tilde{x}) + f(\tilde{x}) - \tilde{f}(\tilde{x})| \leq |f(x) - f(\tilde{x})| + |f(\tilde{x}) - \tilde{f}(\tilde{x})| \tag{5.2}$$

Aus Abschnitt 5.1.1 ist bekannt, dass $|f(x) - f(\tilde{x})|$ die Kondition ist und aus Abschnitt 5.1.2 weiß man, $|f(\tilde{x}) - \tilde{f}(\tilde{x})|$ ist die Kondition eines Algorithmus.

Der Beweis stammt aus [7].

5.1.4 Konsistenz eines Algorithmus

Mit der Konsistenz wird untersucht, ob der Algorithmus tatsächlich das Problem löst, welches er lösen soll oder ein anderes. Dies bedeutet also, wie *genau* berechnet ein Algorithmus ein mathematisches Problem. Man hat ein mathematisches kontinuierliches Problem mit dessen Verfahren $f(x)$ zur Lösung gegeben. Zudem hat man einen numerischen Algorithmus, der das mathematische Problem lösen kann: $f(\tilde{x})$. Zu dem Algorithmus wird eine beliebige, aber feste Schrittweite gewählt. Ideal wäre natürlich eine unendlich kleine Schrittweite, da der Fehler dann gegen Null geht. Durch die Eigenheiten des Rechners (endlicher Speicher) ist dies natürlich nicht möglich und man muss die Schrittweite so wählen, dass sie klein genug ist, damit der Fehler bei der "Abtastung" möglichst gering ist und der Rechner diese Schrittweite realisieren kann. Ist der Fehler also gering, ist der Algorithmus konsistent und bearbeitet auch das Problem, das er lösen soll. Ein typisches Verfahren, das solch einer Untersuchung unterzogen wird, ist das Eulersche Polygonzugverfahren [8].

6 Zusammenfassung und Fazit

Wie man nun abschließend feststellen kann, ist die wissenschaftliche Softwareentwicklung mit einigen Herausforderungen behaftet. Die Abbildung von kontinuierlichen auf diskrete Modelle, die vom Computer bearbeitet werden können, bringt immer Fehler mit sich, derer man sich bei der wissenschaftlichen Softwareentwicklung bewusst werden muss. Auch die Kommunikation zwischen Wissenschaftler und Informatiker ist nicht einfach, da Vorgehensmodelle fehlen. Viele Wissenschaftler programmieren daher ihre eigene Software ohne dabei jedoch nach einem gewissen Konzept vorzugehen, wie es der Informatiker eher machen würde. Deshalb fehlt die wichtige Dokumentation des Programms und macht es für andere Anwender für die Benutzung und für die Weiterentwicklung der Software oft unzugänglich. Informatiker empfinden den Kontext, mit dem sich der Wissenschaftler beschäftigt häufig als zu "komplex" und können die Wünsche und Anforderungen des Wissenschaftler an sein Modell nicht umsetzen. Meist liegt es einfach nur daran, dass der Informatiker sich nicht genug damit beschäftigt oder auch sich nicht damit beschäftigen will. Es gibt daher schon Ansätze, um Wissenschaftlern in programmiertechnischen Fähigkeiten zu schulen. Zum Beispiel vermitteln Bücher wie [9] wichtige Programmiertechniken, um (als Wissenschaftler) im guten Stil seine eigene wissenschaftliche Software zu schreiben. Neuerdings gibt es Studiengänge, die wissenschaftliche Praxis und Softwareentwicklung miteinander verbinden. Diese Studiengänge sind als "Computing in Science" (kurz: CiS) bekannt und werden bereits an der Universität Hamburg angeboten. Der Student lernt also Konzepte der Wissenschaft (zum Beispiel in Physik, Chemie, Biologie) und gleichzeitig wichtige Konzepte der Programmierung. Jedoch gibt es noch keine geeignete Lösung beziehungsweise ein geeignetes Vorgehensmodell, dass die Kommunikation zwischen Naturwissenschaftler und Informatiker verbessert oder besser gesagt überhaupt ermöglicht.
Diese Gegebenheiten sind mir ebenfalls aufgefallen. Ich studiere zwar "reine" Informatik, bin aber auch sehr in den Naturwissenschaften, besonders in der Physik, interessiert. Dadurch habe ich Kontakt zu beiden Seiten und erfahre auch jeweils die verschiedenen Sichtweisen der unterschiedlichen Parteien. Als Informatikerin kann ich sagen, dass es natürlich "abschreckend" ist, wenn man sich mit der Physik beschäftigt und auf unverständliche Formeln stößt. Auf der einen Seite kann ich mich als Neuling in der Physik sofern dazu äußern, dass es weniger "abschreckend" ist, wenn man sich mehr mit einem Thema beschäftigt. So ging es mir auch bei der Bearbeitung meines Seminarvortrags und dieser Seminararbeit. Während mir viele Konzepte aus dem "Software Design" (siehe Kapitel 2) schon aus Vorlesungen bekannt waren und für mich als selbstverständlich galten, brauchte ich mehr Zeit, um Themen wie Numerik und Fehlertheorie zu verstehen, da diese für mich sehr neu waren. Hätte ich nun als Physikerin oder Mathematikerin diese Arbeit geschrieben, wäre es mir bestimmt anders ergangen. Trotzalledem hat dieses Seminar viel Spaß gemacht und ich weiß nun, wie schwer es tatsächlich ist, zwei Disziplinen mit unterschiedlichen Sichtweisen zusammenzubringen.
Abschließend möchte ich noch meiner Seminar-Betreuerin Petra Nerge danken, die mir bei der Struktur meines Seminarvortrags sehr geholfen und mir immer die passenden Denkanstöße gegeben hat.

Literaturverzeichnis

[1] Judith Segal, Chris Morris
Developing Scientific Software
www.computer.org/csdl

[2] http://www.dorn.org/uni/sls/kap05/e01.htm#GENAUIGKEIT

[3] http://de.wikipedia.org/wiki/Lokalit%C3%A4tseigenschaft

[4] http://de.wikipedia.org/wiki/IEEE_754

[5] Thomas Huckle, Stefan Schneider
Numerische Methoden
Springer, 2. Auflage

[6] http://de.wikipedia.org/wiki/Simpsonregel

[7] http://de.wikipedia.org/wiki/Stabilit%C3%A4t_(Numerik)

[8] http://numerik.uni-hd.de/~richter/WS10/numerik/3-einschrittmethoden.pdf

[9] Suley Oliveira, David Stewart (2006)
Writing Scientific Software - A Guide To Good Style
Cambridge University Press

[10] http://www.torsten-horn.de/techdocs/sw-dev-process.htm#V-Modell

[11] http://www.uni-marburg.de/fb12/verteilte_systeme/lehre/ss07/vl/ti2/folien/k06