Komplexitätstheorie und Praxis der Integration Lipschitz-stetiger Funktionen in der exakten reellen Arithmetik

Complexity Theory and Practice of Integrating Lipschitz-continuous functions in Exact Real Arithmetic

Bachelor-Thesis von Holger Thies

September 2011

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Mathematik
Arbeitsgruppe Logik

Komplexitätstheorie und Praxis der Integration Lipschitz-stetiger Funktionen in der exakten reellen Arithmetik
Complexity Theory and Practice of Integrating Lipschitz-continuous functions in Exact Real Arithmetic

Vorgelegte Bachelor-Thesis von Holger Thies

1. Gutachten: Prof. Dr. Martin Ziegler
2. Gutachten: PD Dr. habil. Ulrike Brandt

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 18. September 2011

(Holger Thies)

Inhaltsverzeichnis

1 Zusammenfassung

Die Berechenbarkeitstheorie behandelt die algorithmische Lösbarkeit von Problemen. Dabei ist das Turingmaschinen-Modell heute weitestgehend als realistisches Berechnungsmodell akzeptiert. Die Komplexitätstheorie befasst sich mit der Komplexität berechenbarer Probleme. Dabei werden diese in Komplexitätsklassen eingeteilt. Die (für diese Arbeit) wichtigsten Klassen sind:

- P : Die Klasse der in polynomieller Zeit von einer deterministischen Turingmaschine entscheidbaren Probleme.
- NP : Die Klasse der in polynomieller Zeit von einer nichtdeterministischen Turingmaschine entscheidbaren Probleme.
- PSPACE : Die Klasse der in polynomiellem Platz von einer deterministischen Turingmaschine entscheidbaren Probleme.
- FP : Die Klasse der Funktionen, die von einer deterministischen Turingmaschine in polynomieller Zeit berechnet werden können.
- #P : Die Klasse der Funktionen, die die Anzahl der akzeptierenden Berechnungen einer nicht-deterministischen polynomialzeitbeschränkten Turingmaschine liefert.

Es gilt $P \subseteq NP \subseteq PSPACE$ und $FP \subseteq \#P$. Von keiner dieser Inklusionen ist die Echtheit bekannt. Man geht aber davon aus, dass alle Inklusionen echt sind. Die Frage ob $P = NP$ ist eines der wichtigsten ungelösten Probleme der theoretischen Informatik. Ob $FP = \#P$ ist ein noch stärkeres Problem, denn aus $FP = \#P$ folgt $P = NP$.
In der klassichen Berechenbarkeits- und Komplexitätstheorie wird die Berechenbarkeit und Komplexität von diskretwertigen Problemen (wie Probleme über den natürlichen Zahlen, endliche Graphen, etc.) untersucht. Computer werden jedoch vielfach dazu eingesetzt, numerische Probleme zu lösen. Dabei handelt es sich meistens um Probleme über den reellen Zahlen. Die berechenbare Analysis beschäftigt sich mit solchen Problemen und bildet somit die theoretische Grundlage der numerischen Analysis. Es gibt verschiedene Ansätze, den Berechenbarkeitsbegriff auf die reellen Zahlen zu erweitern. Im Gegensatz zum diskretwertigen Fall gibt es dabei bisher kein allgemein akzeptiertes Berechnungsmodell, sondern verschiedene nicht äquivalente Modelle.
Die so genannte Type-2 Theory of Effectivity (TTE) ist ein solches Modell für das Rechnen mit reellen Zahlen. Dabei wird das Turing-Maschinen Modell erweitert, um Berechnungen mit unendlichen Folgen (z.B unendlichen Binärstrings) zu ermöglichen. Jede reelle Zahl x lässt sich als unendliche Folge darstellen, z.B. durch eine Folge rationaler Approximationen $(a_n)_{n\in\mathbb{N}}$ mit $|a_n - x| \leq 2^{-n}$. TTE soll ein realistisches Berechnungsmodell sein, d.h. es sollen genau die Funktionen berechenbar sein, die durch Computer berechnet werden können.
Aufbauend auf diesem Modell wurde das C++-Framework iRRAM entwickelt. Die iRRAM stellt Klassen und Funktionen zum fehlerfreien Rechnen mit reellen Zahlen zur Verfügung. Der wichtigste Datentyp dabei ist REAL, der für eine reelle Zahl steht. So kann mit reellwertigen Variablen gerechnet werden, ohne dass sich der Programmierer Gedanken um Rundungsfehler oder Ähnliches machen muss. Intern wird dabei jede Berechnung approximativ mit endlicher Genauigkeit durchgeführt. Die Genauigkeit wird jedoch in mehreren Iterationen erhöht, so dass eine beliebig exakte Rechnung durchgeführt werden kann.
Ein wichtiges Problem aus der Numerik ist die Integration, also die Berechnung des Wertes $\int_a^b f(x)\,dx$ für $a, b \in \mathbb{R}$. Numerische Integration, auch Quadratur genannt, findet besonders dann Anwendung, wenn die analytische Berechnung des Integrals nicht möglich oder sehr schwierig ist. In der Vergangenheit wurden eine Vielzahl von Algorithmen zur Lösung dieses Problems entwickelt. 1984 konnte Harvey Friedman ein wichtiges Resultat zur Komplexität der Integration beweisen: in [Fri84] zeigt er, dass die Integration polynomialzeitberechenbarer Funktionen im allgemeinen so schwierig ist, wie die Komplexitätsklasse $\#P$.
Ziel dieser Arbeit war es, ein eindimensionales Integrationsverfahren für berechenbare, Lipschitz-stetige Funktionen in der iRRAM zu implementieren. Dazu wurden verschiedene numerische Integrationsverfahren betrachtet. Eine wichtige Anforderung an das Verfahren war es, dass es möglich ist, das Integral bis auf einen angegeben Fehler beliebig exakt zu berechnen. Deshalb kamen nur Verfahren in Frage, bei denen sich eine obere Schranke an den Fehler bestimmen lässt. Solche Verfahren erfordern allerdings, wenn sie effizient sein sollen, meist sehr viel Information über die Funktion, wie beispielsweise Schranken an sehr hohe Ableitungen. Zwei solcher Verfahren erwiesen sich als geeignet: Die Gauß-Quadratur und die Newton-Cotes-Formeln. Beide Verfahren eignen sich hauptsächlich für Funktionen, die sehr oft differenzierbar sind. Die Newton-Cotes-Formeln werden in der numerischen Praxis kaum verwendet, da sie anfällig für Rundungsfehler sind. Der Vorteil der iRRAM ist jedoch, dass Zwischenergebnisse exakt verarbeitet werden können und somit keine Probleme durch Rundungsfehler auftreten. Als Integrationsverfahren wurden deshalb die Newton-Cotes-Regeln gewählt.

2 Einführung in die Berechenbarkeits- und Komplexitätstheorie

In der Berechenbaren Analysis tauchen an vielen Stellen Konzepte der klassischen Berechenbarkeits- und Komplexitätstheorie wieder auf. Im Folgenden sollen daher die wichtigsten Resultate vorgestellt werden.

2.1 Turingmaschinen

Zur Formalisierung des Berechenbarkeitsbegriffs betrachtet man so genannte Turing-Maschinen. Die Turing-Maschine wurde 1936 von Alan M. Turing als allgemeines Berechnungsmodell erdacht, das stark genug sein soll, jedes mögliche algorithmische Berechnungsverfahren zu realisieren. Eine Turing-Maschine ist ein einfaches Modell für einen Rechner. Sie besteht aus einem beidseitig unendlichen Band, das in Felder eingeteilt ist. Jedes Feld enthält genau ein Symbol des so genannten Bandalphabets Γ, einer endlichen Menge von Symbolen. Die Maschine verfügt über einen Schreib-Lesekopf, der sich immer über genau einem Feld befindet. Zu jedem Zeitpunkt befindet sich die Turingmaschine in einem von endlich vielen Zuständen. In einem Berechnungsschritt kann die Maschine das sich unter dem Schreib-Lesekopf befindende Symbol durch ein eventuell anderes aus dem Alphabet ersetzen und den Kopf um maximal eine Position nach links oder rechts bewegen. Formal lässt sich eine Turingmaschine folgendermaßen definieren:

Definition 2.1. Eine Turing-Maschine ist ein 4-Tupel $(Q, \Sigma, \Gamma, \delta)$.
Dabei beschreibt

$$\begin{aligned} \Sigma &: \text{das endliches Eingabealphabet} \\ \Gamma &: \text{das endliches Bandalphabet (es gilt } \Sigma \subseteq \Gamma) \\ Q &: \text{die endliche Zustandsmenge} \\ \delta &: \text{Die Übergangsfunktion } \delta : Qx\Gamma \mapsto Qx\Gamma x\{L, R, N\} \end{aligned}$$

Weiter entählt die Zustandsmenge Q die folgenden ausgezeichneten Zustände:

$$\begin{aligned} q_0 \in Q &: \text{der Anfangszustand} \\ q_f \in Q &: \text{der Endzustand} \end{aligned}$$

Das Eingabealphabet Σ enthält alle Zeichen, aus denen die Eingabe für die Maschine bestehen kann.
Γ entält zusätzlich noch Zeichen, die von der Maschine auf das Band geschrieben werden können, die aber nicht in der Eingabe vorkommen dürfen.
Meist enthält Γ außer dem Eingabealphabet Σ nur ein weiteres Zeichen $\square$, das für ein leeres Feld steht.
Die Übergangsfunktion δ beschreibt, wie die Maschine verfahren soll, wenn sie im Zustand q ist und sich unter ihrem Schreib-Lese-Kopf das Zeichen x befindet. $\delta(q, x) = (q', x', D)$ mit $D \in \{L, R, N\}$ beschreibt dabei, dass die Maschine im nächsten Schritt x mit x' überschreibt, in den Zustand q' übergeht und den Kopf ein Feld nach links (L), ein Feld nach rechts (R) oder nicht (N) bewegt.

In der klassischen Berechenbarkeitstheorie betrachtet man partielle Funktionen $f : \Sigma^* \to \Sigma^*$ über einem endlichen Alphabet Σ.

Definition 2.2. Sei Σ ein endliches Alphabet. Eine Turingmaschine berechnet eine Funktion $f : \Sigma^* \to \Sigma^*$, wenn die Maschine gestartet mit der Bandinschrift $x \in \Sigma^*$ nach endlich vielen Schritten im Zustand q_f auf dem ersten Zeichen von $f(x) \in \Sigma^*$ stehen bleibt.

Weiter ist folgende Definition nützlich:

Definition 2.3. Sei Σ ein endliches Alphabet. Eine Menge $M \subseteq \Sigma^*$ heißt entscheidbar, wenn die charakteristische Funktion

$$\chi_m : \Sigma^* \to \{0, 1\}, \chi_m(w) = \begin{cases} 1, & \text{falls } w \in M, \\ 0, & \text{falls } w \notin M \end{cases}$$

berechenbar ist.

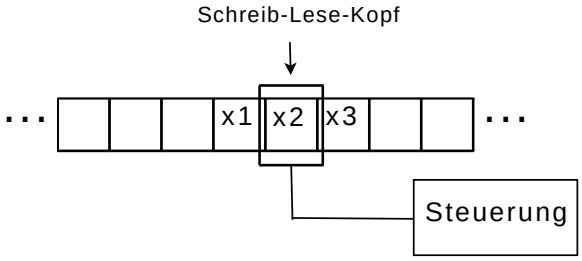

Abbildung 2.1: Schematische Darstellung einer Turingmaschine

Die Aussage, dass jede intuitiv berechenbare Funktion mithilfe von Turingmaschinen berechnet werden kann, ist als Church-Turing-These bekannt. Die These ist nicht beweisbar, da ein Beweis schon eine Formalisierung des intuitiven Berechenbarkeitsbegriffs benötigen würde, der ja durch die Turingmaschine erst gegeben ist. Man geht aber allgemein davon aus, dass die These wahr ist. In der Literatur finden sich eine Vielzahl von unterschiedlichen Definitionen, die aber von der Berechnungsstärke alle äquivalent sind. Eine wichtige Abwandlung der gewöhnlichen Turing-Maschine sind Turing-Maschinen mit mehr als einem Band.

Definition 2.4. Sei $k \in \mathbb{N}$. Eine k-Band-Turingmaschine ist gegeben durch ein 4-Tupel $M = (Q, \Sigma, \Gamma, \delta)$ wobei Q, Σ und Γ genau wie bei der (1-Band-)Turingmaschine definiert sind und $\delta : Q \times \Gamma^k \mapsto Q \times \Gamma^k \times \{L, R, N\}^k$.

Im Gegensatz zu der zuvor definierten Turingmaschine hat eine k-Band-Turingmaschine also k-Bänder mit k Schreib-Lese-Köpfen, die sich unabhängig voneinander bewegen können.
Da die Menge aller Turingmaschinen über einem festen Alphabet Σ abzählbar ist, die Menge der Funktionen $\Sigma^* \to \Sigma^*$ jedoch überabzählbar, folgt sofort, dass es auch nicht berechenbare Funktionen geben muss. Das wichtigste Beispiel für eine konkrete nicht berechenbare Funktion ist das Halteproblem. Beim Halteproblem geht es darum, für eine Turingmaschine M und ein Wort $w \in \Sigma^*$ zu entscheiden, ob die Maschine M bei Eingabe w jemals anhält, oder unendlich lange weiterläuft. Es lässt sich zeigen, dass es keine Turingmaschine gibt, die dieses Problem für beliebige Turingmaschinen M und Wörter w entscheidet.

2.2 Reduktionen

Angenommen von einem Problem P soll gezeigt werden, dass es unentscheidbar ist. Weiß man bereits von einem anderen Problem P', dass dieses unentscheidbar ist (wie z.B. das Halteproblem), kann man dies benutzen um die Unentscheidbarkeit von P zu beweisen. Dafür genügt es zu zeigen, dass P' mithilfe von P entschieden werden kann. Man reduziert also die Lösung von P auf P'. Formal:

Definition 2.5. Seien $P', P \subseteq \Sigma^*$. P heißt auf P' reduzierbar (geschrieben $P \leq P'$), wenn es eine totale und berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt

$$x \in P \Leftrightarrow f(x) \in P'.$$

Satz 2.1. Gilt $P \leq P'$ und P ist unentscheidbar, so ist auch P' unentscheidbar.

2.3 Komplexitätstheorie

Die Komplexitätstheorie beschäftigt sich mit dem Ressourcenbedarf (z.B. Rechenzeit) berechenbarer Probleme. Dazu definiert man zunächst Zeit- und Platzkomplexität für Turingmaschinen.

Definition 2.6. Für eine Turing-Maschine M beschreibt

$time_M(w)$: Die Anzahl an Kopfbewegungen, die die Maschine bei Eingabe w ausführt, bevor sie anhält.
$space_M(w)$: Die Anzahl an Bandzellen, die die Maschine bei Eingabe w besucht, bevor sie anhält.

Die Zeit- und Platzkomplexität einer Turing-Maschine M ist nun definiert als

$$\begin{aligned} t_M(n) &= max\{time_M(w) \mid |w| = n\} \\ s_M(n) &= max\{space_M(w) \mid |w| = n\} \end{aligned}$$

Für die Platzkomplexität ist es sinnvoll, den Platz, der für die Ein- und Ausgabe benötigt wird, nicht zu betrachten. Dazu verwendet man spezielle 3-Band-Turingmaschinen, bei denen es ein Eingabe-, ein Ausgabe- und ein Arbeitsband gibt. Von dem Eingabeband darf nur gelesen und nicht darauf geschrieben werden und auf das Ausgabeband darf nur geschrieben und nicht davon gelesen werden. Für die Platzkomplexität werden dann nur die benötigten Felder auf dem Arbeitsband betrachtet.

Damit lässt sich nun die Zeit- und Platzkomplexität von Problemen definieren:

Definition 2.7. Sei $f : \mathbb{N} \to \mathbb{N}$. Dann ist

$$\begin{aligned} TIME(f(n)) &= \{A \subseteq \Sigma^* \mid \text{es gibt eine Turingmaschine M mit } t_M(n) \in \mathcal{O}(f(n)), \text{ die A entscheidet.}\} \\ SPACE(f(n)) &= \{A \subseteq \Sigma^* \mid \text{es gibt eine Turingmaschine M mit } s_M(n) \in \mathcal{O}(f(n)), \text{ die A entscheidet.}\} \end{aligned}$$

Neben dem oben eingeführten Turingmaschinen-Modell spielt ein weiteres Modell eine wichtige Rolle in der Komplexitätstheorie, die nicht-deterministischen Turingmaschinen. Der Unterschied zwischen deterministischen und nichtdeterministischen Turingmaschinen ist, dass es für eine nichtdeterministische Maschine in jedem Schritt mehrere mögliche nächste Schritte geben kann. Die Kopfbewegung und das Symbol, das an die aktuelle Position geschrieben wird, ist also nicht durch das aktuelle Zeichen und den aktuellen Zustand eindeutig bestimmt. Formal äußert sich dieser Unterschied darin, dass statt einer Übergangsfunktion $\delta : Q \times \Gamma \to Q$ eine Übergangsrelation $\Delta \subseteq Q \times \Gamma \times Q$ definiert wird. Die Berechnung einer nichtdeterministischen Maschine bei einer Eingabe w ist ein Konfigurations-Baum mit der Startkonfiguration als Wurzel. Jeder Pfad in dem Konfigurationsbaum beschreibt einen möglichen Berechnungspfad. Eine Menge $A \subseteq \Sigma^*$ wird von einer nichtdeterministischen Turingmaschine akzeptiert, wenn es für jede Eingabe $w \in A$ mindestens einen Berechnungspfad gibt, der in einem akzeptierenden Zustand endet. Zeit- und Platzkomplexität für nichtdeterministische Maschinen lassen sich folgendermaßen definieren:

Definition 2.8. Sei M eine nichtdeterministische Turingmaschine und $w \in \Sigma^*$ ein Wort. Dann ist

$$\begin{aligned} time_M(w) &: \text{Die maximale Länge jeder Berechnung bei Eingabe } w \\ space_M(w) &: \text{Die maximale Anzahl an benutzten Feldern jeder Berechnung bei Eingabe } w \end{aligned}$$

und für $n \in \mathbb{N}$

$$\begin{aligned} t_M(n) &= max\{time_M(w) \mid |w| = n\} \\ s_M(n) &= max\{space_M(w) \mid |w| = n\} \end{aligned}$$

Analog zu der Definition bei deterministischen Maschinen erhält man:

Definition 2.9. Sei $f : \mathbb{N} \to \mathbb{N}$ eine Funktion. Dann ist

$$\begin{aligned} NTIME(f(n)) &= \{A \subseteq \Sigma^* \mid \text{es gibt eine nichtdeterministische Turingmaschine M mit } t_M(n) \in \mathcal{O}(f(n)), \text{ die A akzeptiert.}\} \\ NSPACE(f(n)) &= \{A \subseteq \Sigma^* \mid \text{es gibt eine nichtdeterministische Turingmaschine M mit } s_M(n) \in \mathcal{O}(f(n)), \text{ die A akzeptiert.}\} \end{aligned}$$

Mithilfe der obigen Definitionen lassen sich entscheidbare Probleme in verschiedene Komplexitätsklassen einordnen. Die wichtigsten Komplexitätsklassen für Entscheidungsprobleme sind:

$$\begin{aligned} L &:= SPACE(log(n)) \\ NL &:= NSPACE(log(n)) \\ P &:= \bigcup_{k \in \mathbb{N}} TIME(n^k) \\ NP &:= \bigcup_{k \in \mathbb{N}} NTIME(n^k) \\ PSPACE &:= \bigcup_{k \in \mathbb{N}} SPACE(n^k) \\ NPSPACE &:= \bigcup_{k \in \mathbb{N}} NSPACE(n^k) \end{aligned}$$

Nach dem Satz von Savitch ist $PSPACE = NPSPACE$.
Weiter gelten die Inklusionen

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE$$

Man vermutet, dass alle Inklusionen echt sind, es ist jedoch lediglich bekannt, dass $L \neq PSPACE$. Eine der wichtigsten ungelösten Probleme der theoretischen Informatik ist die Frage ob $P = NP$.

2.4 Vollständigkeit

Da die Echtheit der obigen Inklusionen nicht bekannt ist, ist die Einordnung von Problemen in Komplexitätsklassen nicht ausreichend für eine Klassifizierung von Problemen nach ihrer "Schwierigkeit". Stattdessen ist es nötig, die Probleme miteinander in Beziehung zu setzen. Dies geschieht wieder mithilfe von Reduktionen:

Definition 2.10. Eine Menge A ist in polynomieller Zeit auf eine Menge B reduzierbar (geschrieben $A \leq_P B$), wenn es eine in polynomieller Zeit berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt, so dass für alle $w \in \Sigma^*$ gilt

$$w \in A \Leftrightarrow f(w) \in B$$

Definition 2.11. Eine Menge A heißt $\leq_P$-schwer für die Komplexitätsklasse $\mathscr{C}$, wenn für jedes $B \in \mathscr{C}$ gilt $B \leq_P A$. Eine Menge A heißt $\leq_P$-vollständig für die Klasse $\mathscr{C}$, wenn $A \in \mathscr{C}$ und $A \leq_P$-schwer für $\mathscr{C}$ ist.

Stephen Cook konnte 1971 zeigen, dass das Erfüllbarkeitsproblem für die Aussagenlogik SAT $\leq_P$-Vollständig für NP ist. Bei SAT geht es darum, für eine gegebene aussagenlogische Formel Φ zu entscheiden, ob diese erfüllbar ist. Seitdem wurden viele weitere Probleme als NP-vollständig charakterisiert. Kann für eines dieser Probleme nachgewiesen werden, dass es in P liegt, folgt $P = NP$. Ebenso gibt es vollständige Probleme für die Klasse PSPACE. Die polynomielle Reduktion eignet sich allerdings nur, um Probleme in Komplexitätsklassen höher als P zu untersuchen. Für Probleme in niedrigeren Komplexitätsklassen betrachtet man stattdessen $\leq_L$-Reduktionen, bei denen die Reduktionsfunktion in logarithmischen Platz berechnet werden kann.

2.5 Orakel-Turingmaschinen

Eine Orakel-Turingmaschine ist eine Turing-Maschine mit einem zusätzlichem Band, dem so genannten Anfrage-Band, und zwei zusätzlichen Zustanden, dem Frage- und Antwortzustand. Bezeichne M^Φ die mit dem Orakel Φ ausgestattete Turingmaschine M. M^Φ verhält sich wie eine gewöhnliche Turingmaschine, solange sie nicht im Anfragezustand ist. Geht die Maschine in den Anfragezustand, wird das Orakel mit dem Text, der sich auf dem Anfrageband befindet als Eingabe ausgeführt. Dieses berechnet Φ und ersetzt den Text auf dem Anfrageband durch das Ergebnis der Rechnung. Danach geht die Maschine in den Antwortzustand über. Die Anfrage des Orakels nimmt lediglich einen Rechenschritt in Anspruch.

2.6 Funktions- und Zählprobleme

In den vorherigen Abschnitten wurde die Komplexität von Entscheidungsproblemen untersucht. Betrachtet man stattdessen Funktionsprobleme, erhält man eine duale Hierarchie von Komplexitätsklassen. Die beiden wichtigsten Klassen sind die Klasse FP, der in polynomieller Zeit von einer deterministischen Turingmaschine berechenbaren Funktionen und die Klasse $FPSPACE$, der in polynomiellem Platz von einer deterministischen Turingmaschine berechenbaren Funktionen. Eine spezielle Klasse von Funktionsproblemen ist die Klasse $\#P$. Eine Funktion $f : \Sigma^* \to \mathbb{N}$ ist in $\#P$, wenn es eine polynomial-zeitbeschränkte nichtdeterministische Turingmaschine M gibt, so dass für alle $w \in \Sigma^*$ die Anzahl der akzeptierenden Rechnungen von M bei Eingabe w gleich $\Phi(w)$ ist. Eine Funktion aus $\#P$ berechnet also die Anzahl der Lösungen für ein Entscheidungsproblem aus NP. Beispielsweise beschreibt $\#SAT$ das Problem, zu einer gegebenen aussagenlogischen Formel Φ die Anzahl der erfüllenden Belegungen zu bestimmen. Ein $\#P$-Problem ist mindestens so schwer lösbar wie das zugehörige NP-Problem, denn kann man die Anzahl der Lösungen zählen, kann man natürlich auch einfach bestimmen, ob es eine Lösung gibt.
Für die oben genannten Klassen gilt

$$FP \subseteq \#P \subseteq FPSPACE.$$

Auch hier ist nicht bekannt, welche der Inklusionen echt ist. Die Frage ob $\#P = FP$ ist sogar in gewissem Sinne stärker, als die Frage ob $P = NP$, denn es gilt

Satz 2.2. Ist $\#P = FP$, so ist $P = NP$.

Beweis. Ist $\#P = FP$, so ist insbesondere $\#SAT \in FP$, d.h. für eine gegebene aussagenlogische Formel kann die Anzahl der erfüllenden Belegungen in polynomieller Zeit berechnet werden. Dann kann auch in polynomieller Zeit berechnet werden, ob es so eine Belegung gibt, also ist $SAT \in P$ und damit $P = NP$. □

3 Berechenbare Analysis

3.1 Berechenbarkeit reeller Zahlen

Im vorigen Kapitel wurde die Berechenbarkeit und Komplexität von Funktionen $\Sigma^* \to \Sigma^*$ behandelt. Zur Untersuchung anderer Mengen und Strukturen (z.B. endliche Graphen, aussagenlogische Formeln) kodiert man diese zunächst durch (endliche) Wörter aus Σ^*. Dazu ordnet man jedem Element der Menge einen eindeutigen Code oder Namen aus Σ^* zu. Dieses Vorgehen ist jedoch nur für abzählbare Mengen möglich, ist also bei der Erweiterung der Definitionen auf reelle Zahlen nicht hilfreich. Stattdessen lässt sich eine reelle Zahl jedoch darstellen als eine unendliche Folge, z.B. durch ihre Binär- oder Dezimaldarstellung. Man kann einer reellen Zahl also einen unendlich langen Namen zuordnen. Es gibt verschiedene Darstellungsmöglichkeiten von reellen Zahlen als unendliche Folgen, aus denen sich mehr oder weniger sinnvolle Definitionen für die Berechenbarkeit von reellen Zahlen ergeben. Eine sinnvolle Definition erhält man beispielsweise durch eines der folgenden äquivalenten Kriterien:

- Es gibt eine Turingmaschine, die ohne Eingabe und ohne jemals zu stoppen x in Binärdarstellung ausgibt.
- Es gibt eine berechenbare Folge rationaler Zahlen, die schnell gegen x konvergiert, d.h. $\forall i \in \mathbb{N}\ |x - q_i| < 2^{-i}$.
- Es gibt eine berechenbare Folge von rationalen schrumpfenden Intervallen, d.h. es existieren berechenbare rationale Folgen a_n und b_n mit $a_0 < a_1 < \ldots < x < \ldots < b_n < \ldots < b_0$.
- $\{q \in \mathbb{Q} \mid q < x\}$ und $\{q \in \mathbb{Q} \mid q > x\}$ sind semi-entscheidbare Teilmengen der rationalen Zahlen.
- Es gibt ein $z \in \mathbb{Z}$ und eine entscheidbare Menge $A \subseteq \mathbb{N}$, so dass $x = z + x_A$ mit $x_A := \sum_{i \in A} 2^{-i-1}$

Im Folgenden heißt eine reelle Zahl berechenbar, wenn sie diese Kriterien erfüllt.

Beispiel 3.1. $\sqrt{2}$ ist berechenbar. Um dies zu zeigen, betrachtet man die Funktion $f : \mathbb{N} \to \mathbb{N}$, $f(n) := min\{k \in \mathbb{N} \mid k^2 < 2n^2 \leq (k+1)^2\}$. f ist berechenbar. Damit sind die Endpunkte der Intervalle $J_0 := [1;2]$ und $J_n := [\frac{f(n)}{n}, \frac{f(n)+2}{n}]$ berechenbare. Es gilt $\sqrt{2} \in J_n$ für alle n und $J_0 \subset J_1 \subset J_2 \subset \ldots$, also ist $\sqrt{2}$ nach obiger Definition berechenbar.

Die Menge der berechenbaren reellen Zahlen ist abzählbar. Damit sind fast alle reellen Zahlen nicht berechenbar. Eine nicht berechenbare reelle Zahl lässt sich leicht unter Ausnutzung der Erkenntnisse aus der diskreten Berechenbarkeitstheorie konstruieren:

Beispiel 3.2. Sei $M \subseteq \mathbb{N}$ eine nicht entscheidbare Teilmenge der natürlichen Zahlen. Dann ist die reelle Zahl

$$x := \sum_{n \in M} 2^{-n}$$

nicht berechenbar.

3.2 Berechenbarkeit reeller Funktionen

Die so genannte Type-2 Theory of Effectivity (TTE) ist eine Erweiterung der klassischen Berechenbarkeits- und Komplexitätstheorie auf Berechnungen mit reellen Zahlen. In TTE betrachtet man spezielle Turingmaschinen, genannt Typ-2 Turingmaschinen, die unendliche Folgen verarbeiten können:

Definition 3.1. Sei Σ ein endliches Alphabet. Σ^* bezeichne die endlichen und Σ^ω die unendlichen Folgen über Σ. Eine Typ-2 Turingmaschine über dem Alphabet Σ ist eine Mehrband-Turingmaschine mit k Eingabebändern, einer endlichen Zahl an Arbeitsbändern und einem Ausgabeband. Zusätzlich gibt es für die Eingabebänder eine Typ-Spezifizierung $(Y_1, \ldots, Y_k, Y_0)$, $Y_i \in \{\Sigma^*, \Sigma^\omega\}$. Von dem Eingabeband darf nur gelesen, auf dem Ausgabeband nur geschrieben werden. Zusätzlich darf der Kopf auf diesen Bändern nicht nach links bewegt werden.

Mithilfe von Typ-2 Maschinen lässt sich der Berechenbarkeitsbegriff auf Funktionen $f : Y_1 \times \ldots \times Y_k \to Y_0$ erweitern:

Definition 3.2. Eine Typ-2 Maschine wird mit Eingabe $(y_1, \ldots, y_k) \in Y_1 \times \ldots \times Y_k$ gestartet, wenn sich für jedes Eingabeband i die Folge $y_i \in Y_i$ ein Feld rechts vom Lesekopf befindet und sich in allen anderen Feldern das Leersymbol $\square$ befindet.
Für $Y_0 = \Sigma^*$ gilt $f_m(y_1, \ldots, y_k) := y_0$ für ein $y_0 \in \Sigma^*$, genau dann wenn M bei Eingabe $(y_1, \ldots, y_k)$ anhält und sich y_0 auf dem Ausgabeband befindet.
Für $Y_0 = \Sigma^\omega$ gilt $f_M(y_1, \ldots, y_k) := y_o$ für ein $y_i \in \Sigma^\omega$, genau dann wenn M bei Eingabe $(y_1, \ldots, y_k)$ niemals anhält und y_0 auf das Ausgabeband schreibt.
Eine Funktion $f : Y_1 \times \ldots \times Y_k \to Y_0$ heißt berechenbar, wenn es eine Typ-2 Maschine M mit $f_M = f$ gibt.

Beispiel 3.3. Die Multiplikation $f : \mathbb{R}^2 \to \mathbb{R}$ ist berechenbar. Eine 2-Band Typ-2 Maschine, die bei Eingabe von zwei Folgen von Intervallen $(I_0, I_1, \ldots)$ und $(J_0, J_1, \ldots)$ ein Intervall $(I_0 \cdot J_0, I_1 \cdot J_1, \ldots)$ berechnet, muss lediglich die rationalen Endpunkte multiplizieren.

Die obige Definition führt zu folgender Verallgemeinerung der Church-Turing-These:
Eine Funktion $f : Y_1 \times \ldots \times Y_k \to Y_0$ kann von einem physikalischen Gerät berechnet werden, genau dann wenn sie von einer Typ-2 Maschine berechnet werden kann.
Allerdings ist diese im Gegensatz zur ursprünglichen Church-Turing-These nicht allgemein akzeptiert, da es bisher keine einheitliche Theorie in der berechenbaren Analysis gibt.

Die Berechenbarkeit von Funktionen $\mathbb{R} \to \mathbb{R}$ lässt sich auch mit Orakel-Turingmaschinen definieren. Eine gewöhnliche Turingmaschine kann in endlicher Zeit jedoch nur endlich viele Symbole lesen und schreiben. Es ist also nicht möglich, den Wert $f(x)$ für eine Funktion $f : \mathbb{R} \to \mathbb{R}$ exakt zu bestimmen. Man begnügt sich deshalb damit, eine beliebig genaue Approximation für $f(x)$ finden zu können. Auch ist es nicht möglich, die reelle Zahl x exakt einzulesen. Allerdings kann eine Maschine beliebig genaue Approximationen für x einlesen. Dies führt zu folgender Definition:

Definition 3.3. Eine Funktion $f : \mathbb{R} \to \mathbb{R}$ heißt berechenbar, wenn es eine Orakel-Turingmaschine gibt, die bei Eingabe $k \in \mathbb{N}$ für alle $x \in \mathbb{R}$ eine rationale Zahl q mit $|q - f(x)| \leq 2^{-k}$ ausgibt. Dazu kann sie das Orakel nach beliebig genauen rationalen Approximationen für x befragen, d.h. das Orakel liefert für jedes $i \in \mathbb{N}$ eine rationale Zahl d mit $|d - x| \leq 2^{-i}$.

Eine wichtige Eigenschaft berechenbarer reeller Funktionen ist, dass sie berechenbare reelle Zahlen auf berechenbare reelle Zahlen abbilden. Außerdem sind sie abgeschlossen unter Komposition. Allerdings zeigt der folgende Satz, dass sehr viele wichtige Funktionen nicht berechenbar sind:

Satz 3.1. Jede berechenbare reelle Funktion ist stetig.

Beweis. siehe [Wei00] $\square$

Damit sind schon sehr einfache Funktionen wie die Signumfunktion oder Treppenfunktionen nicht berechenbar.

3.3 Komplexität reeller Zahlen und Funktionen

In der diskreten Komplexitätstheorie wird die Zeitkomplexität gemessen als Funktion der Eingabelänge. Allerdings ist bei reellwertigen Problemen die Eingabe eine reelle Zahl, also unendlich lang. Um Komplexität reeller Zahlen und Funktionen zu definieren, wird das Orakel-Turingmaschinen-Modell verwendet. Statt Typ-2 Maschinen betrachtet man also gewöhnliche Turing-Maschinen, die beliebig genaue Approximationen von reellen Zahlen berechnen. Die Komplexität einer solchen approximativen Berechnung hängt maßgeblich von der geforderten Ausgabegenauigkeit ab. Damit erhält man folgende Definition der Komplexität von reellen Zahlen in Abhängigkeit von der Ausgabegenauigkeit:

Definition 3.4. Eine reelle Zahl x ist berechenbar in Zeit $O(t)$, wenn es eine Konstante c und eine Turing-Maschine gibt, die für jedes $n \in \mathbb{N}$ (gegeben in Binärdarstellung) in $c \cdot t(n) + c$ Schritten die Binärdarstellung einer dyadisch rationalen Zahl d mit $|x - d| \leq 2^{-n}$ berechnet.

Beispiel 3.4. π ist in polynomieller Zeit berechenbar. Zur Berechnung kann die Bailey-Borwein-Plouffe-Formel

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

verwendet werden. Der Fehler bei Berechnung der ersten n Terme der Summe ist kleiner gleich 16^{-n}, also genügt es diese zu berechnen. Die Summanden können in polynomieller Zeit berechnet werden, da Division, Multiplikation und Addition von Ganzzahlen in polynomieller Zeit berechnet werden können.

Für die Komplexität reeller Funktionen werden nur Funktionen, deren Definitionsbereich ein kompaktes Intervall ist, betrachtet. Ohne diese Einschränkung lässt sich die Komplexität nicht sinnvoll als Funktion der Ausgabegenauigkeit angeben, da für beliebig große Eingaben auch beliebig viel Zeit nötig sein kann, um die Stellen vor dem Binärpunkt einzulesen.

Definition 3.5. Sei $f : K \to \mathbb{R}$ mit $K \subset \mathbb{R}$ kompakt. f ist berechenbar in Zeit $O(t)$, wenn es eine Konstante c und eine Orakel-Turingmaschine gibt, die f nach Definition 3.3 berechnet und nach maximal $c \cdot t(n) + c$ Schritten anhält.

Beispiel 3.5. Die Funktion $f : [0,1] \to \mathbb{R}$, $f(x) = e^x$ ist in polynomieller Zeit berechenbar. Dazu betrachtet man die Reihenentwicklung

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

Um eine Approximation mit Fehler kleiner 2^{-n} zu berechnen, genügen die ersten n Terme der Reihe. Dafür müssen $\mathcal{O}(n)$ Additionen und $\mathcal{O}(n^2)$ Multiplikationen durchgeführt werden. Das Ergebnis hat die gewünschte Genauigkeit, wenn x mit Fehler kleiner 2^{-3n} approximiert wird. Die Operationen müssen also auf dyadisch rationalen Zahlen der Länge $\mathcal{O}(n)$ durchgeführt werden. Damit ist die Komplexität insgesamt $\mathcal{O}(n^4)$.

3.4 Berechenbarkeit und Komplexität der Integration

Satz 3.2. Sei $f : [0,1] \to \mathbb{R}$ eine berechenbare Funktion, dann ist die Funktion $g : [0,1] \to \mathbb{R}$, $g(x) = \int_0^x f(t)\,dt$ für alle $x \in [0,1]$ berechenbar.

In [Ko91] wird die Aussage sogar für die größere Klasse der beschränkten und rekursiv approximierbaren Funktionen gezeigt.
Bei Komplexitätsuntersuchungen von reellwertigen Problemen trifft man erstaunlicherweise häufig wieder die Komplexitätsklassen der diskreten Komplexitätstheorie an.
Für die Integration gilt insbesondere:

Satz 3.3. Die folgenden beiden Aussagen sind äquivalent

1. $FP = \#P$
2. Für jede polynomial-zeitberechenbare Funktion $f : [0,1] \to \mathbb{R}$ ist die Integralfunktion $g : [0,1] \to \mathbb{R}$, $g(x) := \int_0^x f(t)\,dt$ in polynomieller Zeit berechenbar für alle $x \in [0,1]$

Beweis. Siehe [Fri84] □

Also kann sogar schon die Integration von polynomialzeitberechenbaren Funktionen schwierig sein. Für analytische Funktionen gilt jedoch nach [Ko91]

Satz 3.4. Sei $f : [0,1] \to \mathbb{R}$ polynomialzeitberechenbar und analytisch. Dann ist die Funktion $g : [0,1] \to \mathbb{R}, g(x) := \int_0^x f(t)\,dt$ polynomialzeitberechenbar.

4 Die iRRAM

Die iRRAM (interactive Real-RAM) ist ein von Norbert Müller[1] entwickeltes C++ Framework für exakte reelle Arithmetik. Die iRRAM basiert auf dem Konzept der Real Ram ([BH98]). iRRAM Programme werden in gewöhnlichem C++ geschrieben, das durch das Framework um den Datentyp REAL erweitert wird. Dieser ermöglicht das fehlerfreie Rechnen mit reellen Zahlen.
Im Folgenden wird kurz auf das zugrunde liegende Modell eingegangen und die wichtigsten Funktionen der iRRAM vorgestellt. Eine ausführliche Beschreibung findet sich in [MÖ1].

4.1 Darstellung reeller Zahlen

Die iRRAM soll die Implementation einer imperativen Programmiersprache für reelle Zahlen sein. Aufgrund der Überabzählbarkeit der reellen Zahlen kann natürlich kein Modell implementiert werden, das direkt mit reellen Zahlen rechnet. Stattdessen wird die Berechnung unter Verwendung von abzählbaren Strukturen simuliert. Dazu muss eine geeignete Repräsentation für die reellwertigen Teile eines Programms verwendet werden. Die iRRAM benutzt Intervallarithmetik, d.h. eine reelle Zahl x wird dargestellt durch eine Folge von Intervallen

$$J = ((l_0, r_o), (l_1, r_1), (l_2, r_2), \ldots)$$

mit $\lim l_i = \sup l_i = \inf r_i = x$ und $l_i, r_i \in \mathbb{Q}$. x wird also als eine Folge von schrumpfenden Intervallen mit rationalen Endpunkten dargestellt, die x enthalten.
In der Implementierung dieses Modells wird eine reelle Zahl dargestellt als ein einziges Intervall $I = (d - e, d + e)$ mit $x \in I$. Dieses kann im Laufe der Programmausführung kleiner werden, wenn eine höhere Genauigkeit erforderlich ist. Eine reelle Zahl wird also dargestellt durch ein Paar (d, e). d wird als multiple precision number (also beliebig exakte Gleitkommazahlen) gespeichert. Die Fehlerschranke e besteht aus zwei Variablen p und z vom Typ `long` mit $e = z \cdot 2^p$.

4.2 Genauigkeitssteuerung

Für jede Operation $c = a \circ b$ auf Variablen a, b vom Typ REAL muss eine geeignete Intervallrepräsentation für das Ergebnis $c = (d_c - e_c, d_c + e_c)$ berechnet werden. Die Genauigkeit, mit der die multiple precision number d_c berechnet wird, hängt dabei im Wesentlichen von den Fehlern in a und b ab. Sind diese allerdings sehr klein, ist es nicht immer wünschenswert, die Genauigkeit von d_c allein davon abhängig zu machen. So könnten einfache Operationen wie die Division dazu führen, dass das Ergebnis sehr viel exakter berechnet wird, als benötigt und deshalb Rechnungen sehr lange dauern. Um dies zu verhindern, gibt es in der iRRAM eine Genauigkeitsschranke $\bar{p}$. d_c wird maximal mit Genauigkeit $2^{\bar{p}}$ berechnet. Wird eine höhere Genauigkeit benötigt (z.B. weil sich der Approximationsfehler so sehr verstärkt hat, dass Ergebnisse nicht mehr die gewünschte Fehlerschranke erfüllen können), muss diese Genauigkeitsschranke angepasst werden. Die iRRAM startet dann die komplette Berechnung neu mit einer kleineren Genauigkeitsschranke $\bar{p}$. Rechnungen werden also nicht in einem Schritt ausgeführt, sondern die Genauigkeit iterativ verbessert, bis das Ergebnis ausreichend exakt ausgegeben werden kann. Es gibt somit eine unendliche Folge von schrumpfenden Genauigkeitsschranken $\bar{p}_1 > \bar{p}_2 > \bar{p}_3 \ldots$.
Diese lassen sich berechnen durch $\bar{p}_i = \bar{p}_0 + g \cdot \frac{f^i - 1}{f - 1}$. Die voreingestellten Werte für diese Parameter sind $p_0 = g = -50$ und $f = 1.25$.
Ist bekannt, dass ein Programm eine sehr hohe Genauigkeit erfordern wird, ist es sinnvoll, diese Parameter anders zu wählen, um unnötige Iterationen zu vermeiden. Dazu können beim Start eines Programms folgende Parameter angegeben werden:

- `--prec_init=d`: Setzt die Anfangsgenauigkeit p_0 auf d.
- `--prec_start=n`: Setzt die Genauigkeisstufe, mit der das Programm gestartet wird, auf n.
- `--prec_inc=n`: Setzt g auf n.
- `--prec_factor=x`: Setzt den Faktor f auf x.
- `--prec_skip=n`: Setzt den Wert, wie viele Schritte bei einer neuen Iteration übersprungen werden auf n.

Durch die Nutzung dieser Parameter kann die Programmausführung erheblich beschleunigt werden. Die Startgenauigkeit sollte allerdings auch nicht zu hoch gewählt werden, da sehr genaue Rechnungen natürlich auch mehr Rechenzeit benötigen.

[1] http://www.uni-trier.de/index.php?id=3571

4.3 Mehrwertigkeit in der iRRAM

Vergleicht man in der iRRAM zwei reelle Zahlen mittels $<$, versucht diese die Intervalle für beide solange zu verkleinern, bis diese sich nicht mehr überlappen. Sind beide Zahlen gleich, ist das nie der Fall, so dass das Programm niemals terminieren kann. Wie im ersten Kapitel beschrieben, müssen berechenbare Funktionen stetig sein. Dadurch können sehr viele Funktionen nicht direkt in die iRRAM übernommen werden. Um dennoch solche Funktionen verwenden zu können, gibt es mehrwertige Funktionen. Eine mehrwertige Funktion F ist definiert als eine Funktion $F : \mathbb{R} \mapsto 2^{\mathbb{R}}$, die für $x \in \mathbb{R}$ eine Menge von möglichen Ergebnissen liefert. Die Funktion F für ein $x \in \mathbb{R}$ berechnen heißt nun, dass das Ergebnis der Rechnung ein Element der Menge $F(x)$ ist. Die intuitive Bedeutung ist, dass eine mehrwertige Funktion mehrere zulässige Ergebnisse hat und die Maschine eines davon auswählen kann. In der iRRAM werden solche Funktionen genau wie gewöhnliche Funktionen behandelt, mit dem Unterschied, dass für den selben Eingabewert x an verschiedenen Stellen im Programm unterschiedliche Werte für $F(x)$ möglich sind. Die iRRAM wählt bei der Berechnung einen möglichen Wert der Menge aus. Für den Benutzer ist diese Wahl nicht sichtbar, die Berechnung erscheint also nicht-deterministisch. So wäre es aber auch möglich, dass eine Berechnung mit höherer Genauigkeit neu gestartet wird und einen völlig anderen Berechnungspfad durchläuft. Somit wäre das Verhalten in unterschiedlichen Iteration völlig verschieden, wodurch der Programmfluss gestört werden kann. Um dies zu verhindern, werden in der iRRAM die Ergebnisse mehrwertiger Funktionsauswertungen im so genannten multi-value cache zwischengespeichert und bei einer neuen Iteration wieder ausgelesen. Da dies speicheraufwendig ist, sollten mehrwertige Funktionen nicht zu häufig verwendet werden.
Zur Konstruktion mehrwertiger Operationen gibt es in der iRRAM den Datentyp LAZY_BOOLEAN, der neben den beiden üblichen Booleschen Werten 0 und 1 einen dritten Wert $\perp$ annehmen kann. $\perp$ hat dabei die intuitive Bedeutung, dass die momentane Genauigkeit nicht ausreicht, um die Aussage zu überprüfen. Bei der Umwandlung von LAZY_BOOLEAN in `bool`, veranlasst $\perp$ die iRRAM zur Reiteration. Die Berechnung wird also solange mit höherer Genauigkeit neu gestartet, bis der Ausdruck zu 0 oder 1 ausgewertet kann (oder läuft endlos, falls dies nie der Fall ist). Die logischen Operatoren werden somit folgendermaßen auf LAZY_BOOLEAN erweitert:

$a \mid\mid b$	0	1	$\perp$
0	0	1	$\perp$
1	1	1	1
$\perp$	$\perp$	1	$\perp$

a && b	0	1	$\perp$
0	0	0	0
1	0	1	$\perp$
$\perp$	0	$\perp$	$\perp$

$!a$	
0	1
1	0
$\perp$	$\perp$

Weiter gibt es die Funktion

```
int choose(const LAZY_BOOLEAN& x1,
           const LAZY_BOOLEAN& x2,
           const LAZY_BOOLEAN& x3,
           const LAZY_BOOLEAN& x4,
           const LAZY_BOOLEAN& x5,
           const LAZY_BOOLEAN& x6),
```

die bis zu 6 LAZY_BOOLEANs als Eingabeparameter erhält. Wird einer dieser Eingabeparameter zu 1 ausgewertet, liefert sie den Index dieses Parameters (also eine Zahl zwischen 1 und 6). Werden alle Eingabeparameter zu 0 ausgewertet, liefert sie 0. Die Genauigkeit wird so lange erhöht, bis einer dieser Fälle eintritt. Wählt man die Eingaben also so, dass bei genügend hoher Genauigkeit mindestens eine der Bedingungen zu wahr ausgewertet wird, erhält man immer ein Ergebnis. Damit können sehr einfach mehrwertige Funktionen definiert werden.
So kann beispielsweise die Funktion

$$equals(x, y, k) = \begin{cases} \text{true}, & \text{falls } |x - y| \leq 2^{k-2} \\ \text{true oder false}, & \text{falls } 2^k \geq |x - y| > 2^{k-2} \\ \text{false}, & \text{falls } |x - y| > 2^k, \end{cases}$$

die die Gleichheit von zwei Eingabeparametern x und y bis auf eine vorgegebene Fehlerschranke bestimmt, folgendermaßen implementiert werden:

Listing 4.1: Beispiel für eine mehrwertige Funktion

```
bool equals(REAL x, REAL y, long k){
  return (choose(abs(x-y) < power(2, -k), abs(x-y) > power(2, -k-2)) == 1);
}
```

Die beiden Bedingungen sind so gewählt, dass bei ausreichend genauer Berechnung von $|x-y|$ sicher eine Bedingung zu 1 ausgewertet werden kann. Ist $|x-y|$ so, dass beide Fälle eintreten, kommt es darauf an, wie die Intervallrepräsentation der Zahl $|x-y|$ aussieht und welcher Fall zuerst entschieden werden kann. Da der Benutzer für gewöhnlich die interne Darstellung der Zahl nicht sieht, scheint zufällig eines der beiden Ergebnisse ausgewählt zu werden. Im Hintergrund ist die Berechnung aber natürlich deterministisch.
Mehrwertige Funktionen werden auch zur Umwandlung von reellen Zahlen in andere Datentypen benötigt. So gibt es die mehrwertige Funktion `DYADIC approx(const REAL& x, const long p)`, die zu einer gegebenen Zahl x vom Typ `REAL` eine dyadische Approximation d mit $|x-d| \leq 2^{-p}$ liefert.

4.4 Klassen und Funktionen der iRRAM

Im folgenden werden die wichtigsten Klassen und Funktionen der iRRAM vorgestellt. Dabei werden hauptsächlich die für die Implementierung des Integrals benutzten Funktionalitäten vorgestellt. Die Aufzählung ist also keineswegs vollständig, sondern bietet nur einen groben Überblick über die Möglichkeiten der iRRAM.

- **Rechnen mit diskretwertigen Datentypen**: der Datentyp `DYADIC` ermöglicht das Rechnen mit beliebig exakten Gleitkommazahlen (multiple precision arithmetic). Die üblichen arithmetischen Operatoren können verwendet werden. Es gibt außerdem den Datentyp `INTEGER` für das Rechnen mit beliebig großen Ganzzahlen und den Datentyp `RATIONAL` zum Rechnen mit Brüchen.

- **Rechnen mit reellen Zahlen**: Der Datentyp für das exakte Rechnen mit reellen Zahlen ist `REAL`. Für diesen stehen verschiedene Konstruktoren zur Verfügung, die aus Variablen vom Typ `int`, `long`, `char*`, `double`, `DYADIC` oder `REAL` ein Objekt vom Typ `REAL` erstellen. Die üblichen arithmetischen Operationen $+, \times, -, \div$ stehen für `REAL` zur Verfügung und können, da die Operatoren `+`, `-`, `*`, `/` überladen sind, auf natürliche Weise verwendet werden. Außerdem gibt es Funktionen wie `abs(x)`, `power(x,n)`, `root(x,n)`, `modulo(x,y)`, `maximum(x,y)`, `minimum(x,y)`, `exp(x)`, `log(x)` sowie trigonometrische und Hyperbelfunktionen (und deren Inverse). Weiter sind Vergleiche mittels den Operatoren $<, <=, >, >=$ möglich, führen jedoch bei Gleichheit zu einer Endlosschleife. Die Konstanten π, e und $ln(2)$ sind als Funktionen `pi()`, `euler()`, `ln2()` implementiert.

- **Rechnen mit Matrizen**: Neben dem Datentyp `REAL` für das exakte Rechnen mit reellen Zahlen, stellt die iRRAM den Datentyp `REALMATRIX` für das exakte Rechnen mit Matrizen mit reellen Einträgen zur Verfügung. Eine `REALMATRIX` wird mittels des Konstruktors

  ```
  REALMATRIX::REALMATRIX(unsigned int rows,unsigned int columns)
  ```

  erstellt. Auf einen Eintrag i, j einer Matrix m kann mittels `m(i,j)` zugegriffen werden. Die Operatoren $+, -, *$ sind überladen und ermöglichen die üblichen Matrixoperationen Addition, Subtraktion und Multiplikation, sowie Multiplikation und Division mit Skalaren durch `m*x` und `m/x`.
  Außerdem ist es möglich, ein lineares Gleichungssystem $M \cdot x = b$ durch die Operation

  ```
  REALMATRIX solve
       (REALMATRIX& lside,
        REALMATRIX& rside,
        int               use_pivot)
  ```

  mittels des Gauß-Algorithmus zu lösen, wobei der letzte Parameter angibt, ob Spalten-Pivotisierung benutzt werden soll. Für `solve(M,b,1)` kann auch die Abkürzung `b/M` verwendet werden.
  Speziell für dünnbesetzte Matrizen gibt es zusätzlich den Datentyp `SPARSEREALMATRIX`, der die gleichen Funktionen zur Verfügung stellt, aber für solche Matrizen optimiert ist.

5 Numerische Quadratur

In der Numerischen Mathematik wurden verschiedene Verfahren entwickelt, um Integrale von reellwertigen Funktionen zu approximieren. Diese Verfahren bezeichnet man als numerische Quadraturverfahren. Eine n-Punkt Quadraturformel ist eine Summe von gewichteten Funktionswerten

$$Q_n(f) = \sum_{i=0}^{n} w_i f(x_i)$$

mit von der Funktion unabhängigen Stützstellen $x_0, \dots x_n$ und Gewichten $w_0, \dots, w_n$.

5.1 Riemann-Summen

Ein einfaches Quadraturverfahren erhält man schon direkt aus der Definition des Integrals durch Riemann-Summen:
Sei $f : [a,b] \to \mathbb{R}$. Für eine Unterteilung $a = x_1 < x_2 < \ldots < x_{n+1} = b$ des Intervalls $[a,b]$ und für eine beliebige Wahl von Punkten $\xi_i \in [x_i, x_{i+1}]$, $i = 1, \dots, n$ heißt die Summe

$$R_n := \sum_{i=1}^{n} (x_{i+1} - x_i) f(\xi_i)$$

Riemann-Summe für die Funktion f. f heißt Riemann-Integrierbar auf $[a,b]$, wenn es eine Zahl $V \in \mathbb{R}$ gibt, so dass jede Riemann-Summe R für f gegen V konvergiert, wenn die Feinheit der Unterteilung $\mu = max\{x_2 - x_1, \dots, x_{n+1} - x_n\}$ gegen 0 geht. Man schreibt dann $\int_a^b f(x)\,dx = V$.
Das wohl naheliegenste Verfahren zur numerischen Integration ist somit die Konstruktion einfacher Riemann-Summen. Man erhält beispielsweise eine konvergente Folge von Riemann-Summen durch Teilen des Integrationsintervalls in n Teilintervalle gleicher Länge

$$x_i := a + ih,\ i = 1, \dots, n, \text{ mit } h := \frac{b-a}{n}$$

und Wahl der Zwischenstellen $\xi_i = x_i$. Damit erhält man die Quadraturformeln

$$R_n(f) := \sum_{i=1}^{n} (x_i - x_{i-1}) f(x_i) = h \sum_{i=0}^{n-1} f(x_i)$$

$R_n(f)$ ist eine Riemann-Summe für f, also folgt unmittelbar, dass für jede Riemann-integrierbare Funktion gilt

$$\lim_{n\to\infty} R_n(f) = \int_a^b f(x)\,dx.$$

Neben der Konvergenz spielt aber auch die Effizienz der Formel eine wichtige Rolle. Der entscheidende Faktor ist dabei die Anzahl der Funktionsauswertungen, die benötigt werden, um eine bestimmte vorgegebene Genauigkeit zu erreichen. Dazu ist es nötig, den absoluten Integrationsfehler zu kennen.

Satz 5.1. Für stetige Funktionen $f \in C[a,b]$ gilt

$$|R_n(f) - \int_a^b f(x)\,dx| \le (b-a) \cdot W$$

mit

$$W := max\{|f(x_1) - f(x_2)| \,|\, x_1, x_2 \in [a,b], |x_1 - x_2| \le \frac{b-a}{n}$$

Beweis. Siehe [Kü94] □

Man sieht, dass das Verfahren nicht besonders effizient ist. Beispielsweise benötigt man schon mehr als 5000 Funktionsauswertungen, um das Integral $\int_0^1 x\,dx$ mit einem Fehler kleiner 10^{-4} zu berechnen.

5.2 Quadratur durch Interpolation

Ein Standardverfahren zur Approximation von Integralen besteht darin, die zu integrierende Funktion durch ein Polynom festen Grades anzunähern und dieses zu integrieren. Um das Integral $\int_a^b f(x)\,dx$ zu approximieren, werden dazu $n+1$ Punkte $x_0,\ldots,x_n \in [a,b]$ gewählt und die Funktion durch das Lagrange-Polynom n-ter Ordnung

$$L(x) = \sum_{k=0}^{n} L_k(x_k) f(x_k)$$

,

$$L_k(x) = \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{x - x_j}{x_i - x_j}$$

interpoliert.
Sei $R(x)$ der Fehlerterm der Interpolation, dann gilt also $f(x) = L(x) + R(x)$. Daraus ergibt sich die Formel

$$\int_a^b f(x)\,dx = \int_a^b L(x) + R(x)\,dx = (b-a)\sum_{k=0}^{n} f(x_k) w_k + \int_a^b R(x)\,dx$$

mit den Gewichten $w_k = \frac{1}{b-a}\int_a^b L_k(x)\,dx$.
Die Gewichte ergeben sich dabei jeweils durch Integration von Polynomen, können also einfach durch die Berechnung einer Stammfunktion gefunden werden.
Ein weiterer Weg zur Bestimmung der Stützstellen, ist zu fordern, dass die Quadraturformel auf $[a,b]$ Polynome bis zu einem gewissen Grad $M \geq n$ exakt integrieren soll.
Durch Einsetzen der Monome $1, x, x^2, \ldots, x^m$ erhält man das Gleichungssystem

$$\sum_{k=0}^{n} w_k x_k^m = \frac{b^{m+1} - a^{m+1}}{m+1}, \quad m = 0 \ldots M. \tag{5.1}$$

Dieses ist linear in den w_k und nichtlinear in den x_k. Sind die Stützstellen vorgegeben, können die zugehörigen Gewichte also durch Lösen eines linearen Gleichungssystems berechnet werden.

5.3 Newton-Cotes-Formeln

Die einfachste Form dieser Integrationsmethode sind die (abgeschlossenen) Newton-Cotes-Formeln. Dabei wählt man die Stützstellen im Intervall $[a,b]$ äquidistant als $x_i = a + h\cdot i,\ i = 0 \ldots n$ mit $h = \frac{b-a}{n}$. Die Gewichte hängen dann nicht vom Integrationsintervall ab, denn es gilt

$$\begin{aligned} w_k &= \frac{1}{b-a}\int_a^b \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{x - x_j}{x_k - x_j}\,dx \\ &= \frac{1}{b-a}\int_a^b \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{x - (a+hj)}{h\cdot(k-j)}\,dx \\ &= \frac{1}{n}\int_0^n \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{s - j}{k - j}\,ds \end{aligned}$$

Wobei der letzte Schritt mit der Substitution $s = \frac{x-a}{h}$ folgt.
Aus dieser Darstellung wird insbesondere die nützliche Eigenschaft ersichtlich, dass alle w_k rationale Zahlen sind. Für die Implementierung des Verfahrens können die Stützstellen also einmal vorberechnet und gespeichert werden. Zur Berechnung der Stützstellen ergibt sich aus (5.1) das lineare Gleichungssystem

$$\begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 0 & 1 & 2 & \ldots & n \\ 0 & 1 & 4 & \ldots & n^2 \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ 0 & 1 & 2^n & \ldots & n^n \end{pmatrix} \begin{pmatrix} w_1' \\ w_2' \\ w_3' \\ \vdots \\ w_n' \end{pmatrix} = \begin{pmatrix} n \\ \frac{n^2}{2} \\ \frac{n^3}{3} \\ \vdots \\ \frac{n^{n+1}}{n+1} \end{pmatrix} \tag{5.2}$$

$$\begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{pmatrix} = \frac{1}{n} \begin{pmatrix} w_1' \\ w_2' \\ w_3' \\ \vdots \\ w_n' \end{pmatrix}$$

Für die Matrix auf der linken Seite des Gleichungssystems gilt nach [Kü94] $det(A) \neq 0$. Die Stützstellen lassen sich also eindeutig bestimmen.
Der Interpolationsfehler berechnet sich zu

$$|R(x)| = |f(x) - P_n(x)| \leq \frac{|f^{(n+1)}(\xi)|}{(n+1)!} \prod_{i=0}^{n} |(x - x_i)| \text{ für ein } \xi \in [a, b]$$

Damit erhält man für den Quadraturfehler die grobe Abschätzung

$$E(f, a, b) = \frac{1}{(n+1)!} \int_a^b f^{(n+1)}(\xi) \prod_{i=0}^{n} |(x - x_i)| \, dx \leq \frac{h^{n+2}}{(n+1)!} \|f^{(n+1)}\|_\infty$$

Beispiel 5.1. Als Spezialfall der Newton-Cotes Formel erhält man als Formel vom Grad 1 die bekannte Trapez-Regel

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}(f(a) + f(b)) =: Q_1(f).$$

Für den Approximationsfehler der Trapezregel gilt $E = Q_1(f) - \int_a^b f(x) = \frac{(b-a)^3}{12} f''(\xi)$ für ein $\xi \in [a, b]$.

Da die Newton-Cotes Formeln exakt für das Polynom $f = 1$ sind, folgt für Formeln, bei denen alle Gewichte w_i positiv sind

$$\sum_{i=0}^{n} |w_i| = \sum_{i=0}^{n} w_i = 1.$$

Bei allen Newton-Cotes Formeln vom Grad $D \geq 10$ treten jedoch auch negative Gewichte auf. In diesem Fall gilt die obige Gleichung nicht. Man kann sogar zeigen, dass

$$\sum_{i=0}^{n} |w_i| \to \infty \text{ für } n \to \infty.$$

Wird die Berechnung der Quadraturformel wie üblich approximativ durchgeführt, verstärken sich Rundungsfehler bei den Funktionsauswertungen. In der Numerik werden daher nur Newton-Cotes niedriger Ordnung eingesetzt. Man kann außerdem zeigen, dass es analytische Funktionen gibt, für die Newton-Cotes Formeln nicht gegen das Integral der Funktion konvergieren. Allein durch die Erhöhung der Ordnung, erhält man also nicht unbedingt bessere Approximationen.

5.4 Zusammengesetzte Newton-Cotes-Formeln

Statt die zu integrierende Funktion im gesamten Integrationsgebiet durch ein einziges Polynom von möglicherweise sehr hohem Grad anzunähern, kann die Funktion auch stückweise in Teilintervallen durch Polynome niedrigeren Grads approximiert werden. Dazu teilt man das Integrationsintervall $[a, b]$ in k Teilintervalle gleicher Länge und approximiert auf jedem dieser Teilintervalle $[a_j, b_j]$, $j = 1, \ldots, k$ das Integral $\int_{a_j}^{b_j} f(x)\ dx$ durch anwenden einer n-Punkte Quadraturformel. Man erhält dann

$$\int_a^b f(x)\,dx = \sum_{j=1}^{k} \int_{a_j}^{b_j} f(x)\,dx \approx \sum_{j=1}^{k} Q_n(f)[a_j, b_j].$$

Also die Quadraturformel

$$Q_{n,k}(f) = \frac{b-a}{nk} \sum_{j=1}^{k} \sum_{i=0}^{n} w_i f(x_{i,j}) = \frac{b-a}{nk} \sum_{i=0}^{n} w_i \sum_{j=1}^{k} f(x_{i,j}) \tag{5.3}$$

mit

$$x_{i,j} = a + j \cdot \frac{b-a}{k} + i \cdot \frac{b-a}{nk},\ i = 1, \ldots, n\ j = 1, \ldots n$$

Im Gegensatz zu den einfachen Quadraturformeln gilt für zusammengesetzte Formeln

Satz 5.2. Sei Q_n eine n-Punkte Quadraturformel, dann gilt für jede in $[a,b]$ beschränkte Riemann-integrierbare Funktion

$$\lim_{k\to\infty} Q_{n,k}(f) = \int_a^b f(x)\,dx$$

Beweis. Siehe [Kü94] □

Zusammengesetzte Formeln konvergieren somit immer gegen das Integral der Funktion.
Ist Q_n eine n-Punkt Quadraturformel mit Fehler

$$Q_n(f) - \int_a^b f(x)\,dx = C\cdot(b-a)^{p+1} f^{(p)}(\xi) \text{ für ein } \xi \in (a,b),$$

dann erhält man durch Aufsummieren der Fehler in den Subintervallen den Fehler der zusammengesetzten Regel als

$$E_{Q_{n,k}} = \sum_{j=1}^{k} C\left(\frac{b-a}{k}\right)^{p+1} f^{(p)}(\xi_j) = C\left(\frac{b-a}{k}\right)^{p+1} \sum_{j=1}^{k} f^{(p)}(\xi_j) \text{ mit } \xi_j \in [a_j, b_j]$$

Setzt man den Fehler der Newton-Cotes-Formeln ein, bekommt man folgende Fehlerabschätzung für die zusammengesetzen Newton-Cotes-Formeln:

$$\begin{aligned} E_{Q_{n,k}} &\leq \sum_{j=1}^{k} \frac{1}{(n+1)!}\left(\frac{b_j-a_j}{n}\right)^{n+2} \|f^{(n+1)}\|_\infty \\ &= \sum_{j=1}^{k} \frac{1}{(n+1)!}\left(\frac{b-a}{nk}\right)^{n+2} \|f^{(n+1)}\|_\infty \\ &= \frac{k}{(n+1)!}\left(\frac{b-a}{nk}\right)^{n+2} \|f^{(n+1)}\|_\infty \end{aligned}$$

Beispiel 5.2. Die zusammengesetzte Trapezregel berechnet sich durch die Formel

$$T_k[a,b] = \frac{h}{2}(f(a) + 2f(a+h) + \ldots + 2f(a+(k-1)h) + f(b))$$

mit $h = \frac{b-a}{k}$. Für den Fehler erhält man eine genauere Abschätzung als die Obige durch

$$|T_k[a,b] - \int_a^b f(x)\,dx| \leq \frac{h^2}{12}\|f''\|_\infty$$

5.5 Weitere Quadraturformeln

Aufgrund der starken Auswirkung von Rundungsfehlern bei Newton-Cotes-Formeln höherer Ordnung, werden in der numerischen Praxis fast ausschließlich zusammengesetzte Formeln niedriger Ordnung verwendet. Diese sind aber nicht sonderlich effizient, sodass für die Berechnung von Integralen mit hoher Genauigkeit andere Verfahren verwendet werden müssen.

5.5.1 Das Romberg-Verfahren

Beim Verfahren von Romberg werden Trapezformeln verschiedener Schrittweite so miteinander kombiniert, dass sich die Fehler möglichst ausgleichen. Man nutzt dabei aus, dass für $f \in C^{2k+1}[a,b]$ der Integrationsfehler bei der Trapezregel genauer charakterisiert werden kann. Mithilfe der Euler-Maclaurin-Summen-Formel erhält man für die zusammengesetze Trapezregel mit n Stützstellen die Darstellung

$$T_n(f) = \int_a^b f(x)\,dx + C_2h^2 + c_4h^4 + \ldots + C_{2k}h^{2k} + h^{2k+1}P_{k+1}$$

Die C_k hängen nur von der Funktion und nicht von der Anzahl der Stützstellen ab. Somit gilt

$$T_{2n} = \int_a^b f(x)\,dx + \frac{1}{4}C_2h^2 + \frac{1}{16}c_4h^4 + \ldots + C_{2k}h^{2k}$$

und damit

$$\frac{4T_{2n}(f) - T_n(f)}{3} = \int_a^b f(x)\,dx + \frac{2^{-2}-1}{3}C_4h^4 + \frac{2^{-4}-1}{3}C_6h^6 + \ldots = \int_a^b f(x)\,dx + O(h^4)$$

Setzt man dies durch

$$\begin{aligned} T_n^0(f) &:= T_n(f) \\ T_n^k &:= \frac{4^kT_{2n}^{k-1} - T_n^{k-1}}{4^k - 1} \end{aligned}$$

rekursiv fort, erhält man Formeln mit immer besserem asymptotischen Fehler. Es ist allerdings schwierig, aus dieser Darstellung eine exakte Fehlerschranke zu bestimmen, weshalb sich das Verfahren nur dann eignet, wenn die Richtigkeit der Ausgabegenauigkeit nicht garantiert werden muss.

5.5.2 Gauß-Quadratur

Bei den bisher gesehenen Formeln wurden jeweils die Stützstellen vorgegeben und die Gewichte so gewählt, dass die Genauigkeit möglichst hoch ist. Um Polynome von möglichst hohem Grad durch eine Quadraturformel

$$Q_n(f) = \sum_{i=0}^{n} w_i f(x_i)$$

exakt zu integrieren, ist es jedoch notwendig die $n+1$ Stützstellen und $n+1$ Koeffizienten entsprechend zu wählen. Durch Formeln vom obigen Typ können maximal Polynome vom Grad $2n+1$ exakt integriert werden. Bei der Gauß-Quadratur werden die Stützstellen und Parameter so gewählt, dass diese maximale Genauigkeit erreicht wird. Die Gewichte und Stützstellen wählt man dabei meist für das Integrationsintervall $[-1,1]$. Für die Berechnung eines Integrals über einem anderen Intervall $[a,b]$ muss dies zunächst auf ein Integral über $[-1,1]$ zurückgeführt werden. Dies erreicht man durch die Substitution

$$\int_a^b f(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} f(\frac{b-a}{2}x + \frac{a+b}{2})\,dx$$

Als Stützstelle x_i wählt man die i-te Nullstelle des Legendre-Polynoms

$$L_n(x) = \frac{d^n}{dx^n}(x-a)^n(x-b)^n.$$

Die Gewichte lassen sich dann zu

$$w_i = \frac{2}{(1-x_i^2)L_n'(x_i)^2}$$

berechnen.
Der Approximationsfehler bei der Integration ist

$$|Q_n(f) - \int_a^b f(x)\,dx| = \frac{((m+1)!)^4}{((2m+2)!)^3(2m+3)}h^{2m+3}|f^{(2m+2)}(\xi)|. \tag{5.4}$$

Es lässt sich zeigen, dass alle Gewichte der Gauß-Formeln positiv sind, so dass die Formeln weniger anfällig für Rundungsfehler bei der Funktionsauswertung sind als die Newton-Cotes-Formeln. Außerdem sind die Gauß-Formeln G_n Riemann-Summen, so dass

$$\lim_{n\to\infty} G_n(f) = \int_a^b f(x)\,dx$$

für jede Riemann-Integrierbare Funktion f. Im Gegensatz zu den Newton-Cotes-Formeln treten aber auch irrationale Koeffizienten und Gewichte auf, so dass es nicht möglich ist, diese vorzuberechnen. Außerdem ist die Berechnung der Stützstellen nicht durch Lösen eines linearen Gleichungssystems möglich. Eine Implementierung der Gauß-Quadratur ist deshalb deutlich schwieriger, als die Implementierung der Newton-Cotes-Regeln.

5.5.3 Adaptive Verfahren

Bei den zusammengesetzten Verfahren wird das Integrationsintervall geteilt und die gleiche Quadraturformel auf alle Teilintervalle angewandt. Allerdings benötigt man im Allgemeinen nicht in jedem Teilintervall die gleiche Anzahl Funktionsauswertungen, um die Funktion mit einer bestimmten Genauigkeit zu approximieren. Wendet man also überall die gleiche Regel an, muss man die Funktion möglicherweise sehr viel häufiger auswerten als notwendig, um eine gewünschte Fehlerschranke zu unterschreiten. Bei adaptiven Verfahren passt man die Wahl der Stützstellen an die zu integrierende Funktion an. Man wertet die Funktion also dort häufiger aus, wo eine höhere Genauigkeit nötig ist. Damit verringert man insgesamt die Anzahl der Funktionsauswertungen, die nötig sind, um eine bestimmte Fehlerschranke zu erreichen. Aus einer einfachen Quadraturformel erhält man beispielsweise ein adaptives Verfahren, indem man zunächst den Fehler berechnet, der bei einmaliger Ausführung des Verfahrens auftritt. Ist dieser klein genug, führt man das Quadraturverfahren aus, andernfalls teilt man das Integrationsintervall in der Mitte und wendet den Algorithmus rekursiv auf beide Teilintervalle an. Für adaptive Verfahren sind allerdings mehr Informationen über die Funktion nötig. So muss der Fehler für jedes Teilintervall bekannt sein, um zu berechnen, wie viele Auswertungen in diesem nötig sind. Man benötigt also beispielsweise lokale Fehlerschranken für die Ableitungen.

5.6 Fehlerschätzung

Die bisher betrachteten Fehlerterme haben den Nachteil, dass Informationen über die Ableitungen der zu integrierenden Funktion benötigt werden. Da diese oft nicht bekannt und aufwendig zu bestimmen sind, begnügt man sich in der numerischen Praxis meist damit, den Fehler abzuschätzen. Beispielsweise kann man ein Verfahren mit unterschiedlicher Schrittweite ausführen und aus der Differenz der Ergebnisse eine Fehlerapproximation berechnen. Da man dadurch aber nur Schätzungen des Fehlers und keine wirklichen oberen Schranken bekommt, kann nicht garantiert werden, dass das Ergebnis die gewünschte Genauigkeit hat. Verfahren, die Integrale mit einer garantierten Fehlerschranke berechnen sollen, benötigen zwingend weitere Informationen über die Funktion, wie z.B. Schranken an bestimmte Ableitungen. Man kann auch davon ausgehen, dass für effizientere Verfahren mehr Information benötigt wird. Beispielsweise benötigt die Trapezregel nur eine Schranke an die zweite Ableitung, für die Gauß-Quadratur mit 100 Stützstellen wird allerdings schon die 200. Ableitung benötigt.

6 Integration in der iRRAM

Die meisten der oben vorgestellten Verfahren können direkt in der iRRAM implementiert werden. Das Rechnen in exakter Arithmetik ist natürlich vor allem von Vorteil, wenn es darum geht, eine bestimmte Genauigkeit für das Ergebnis zu fordern. Es macht also nur Sinn, Verfahren zu verwenden, bei denen eine obere Schranke für den Approximationsfehler berechnet werden kann. Von den oben vorgestellten Verfahren eignen sich somit die Newton-Cotes-Formeln und die Gauß-Quadratur. Die Newton-Cotes-Formeln sind dabei besonders interessant, da in der iRRAM Rundungsfehler nicht auftreten können und somit im Gegensatz zum Rechnen mit Gleitkommazahlen auch Regeln höherer Ordnung verwendet werden können. Außerdem lassen sich diese sehr einfach implementieren, da fast alle notwendigen Methoden schon in der iRRAM vorhanden sind.

6.1 Newton-Cotes-Formeln

Für diese Arbeit wurde eine Funktion zur Berechnung der zusammengesetzten Newton-Cotes-Formeln beliebiger Ordnung implementiert.

6.1.1 Berechnung der Stützstellen

Die Berechnung der Stützstellen der Newton-Cotes-Formeln zu vorgegebener Ordnung n lässt sich in der iRRAM direkt durch Ausnutzung des Gleichungssystems 5.3 berechnen.
Die Funktion zur Berechnung der Stützstellen sieht folgendermaßen aus:

Listing 6.1: Funktion zur Berechnung der Stützstellen

```
REALMATRIX getWeights(int n){
  REALMATRIX l(n+1,n+1);
  REALMATRIX r(n+1,1);
  for(int i=0; i<=n; i++){
    r(i,0) = power((REAL)n,i+1)/(REAL)(i+1);
    for(int k=0; k<=n; k++){
      l(i,k) = power((REAL)k, i);
    }
  }
  REALMATRIX s = solve(l, r,1);
  return s;
}
```

Der linke und rechte Teil des Gleichungssystems wird also jeweils direkt in eine REALMATRIX gespeichert und das Gleichungssystem mithilfe der Funktion `solve` gelöst. Rückgabewert der Funktion ist ein Vektor (als $n \times 1$-Matrix) mit den Stützstellen. Die folgende Tabelle enthält die Ausführungsdauer der Funktion für verschiedene Parameter n. Dabei wurde für die Anfangsgenauigkeit `--prec_start=55` für $n >= 200$ gewählt und für alle anderen 45.

n	t
10	0.02
50	1.12
100	8.18
150	41.38
200	147.36
300	440.92
400	913.78
500	2820
700	57000

Da es sich bei den Stützstellen um rationale Zahlen handelt, ist es nicht notwendig, die zeitaufwendige Berechnung jedes Mal zu wiederholen. Stattdessen werden die Stützstellen in eine Datei gespeichert und beim Programmstart wieder daraus gelesen. Für diese Arbeit wurde dies beispielhaft für die Stützstellen der 500-Punkte Regel durchgeführt. Um

die berechneten Stützstellen als rationale Zahlen zu speichern, müssen zunächst Zähler und Nenner gefunden werden. Durch Umformung von 5.1 erhält man

$$w_k = \frac{1}{nk!(n-k)!n!} \underbrace{(-1)^{n-k} n! \int_0^n \Pi_{j=0}^n (s-j) ds}_{\in \mathbb{Z}}$$

Hat man also w_k bereits berechnet erhält man die (ungekürzte) Bruchdarstellung

$$w_k = \frac{nk!(n-k)!n! w_k}{nk!(n-k)!n!}.$$

Damit erhält man zum Speichern und Auslesen der Stützstellen die Funktionen

Listing 6.2: Funktionen zum Speichern und Laden der Stützstellen

```
void saveWeightsToFile(int n){
  INTEGER facn = IntFactorial(n+1);
  REALMATRIX w;
  std::stringstream filename;
  filename << "weights"<<n<<".txt";
  orstream file(filename.str(), std::_S_out);
  // compute weights
  w = getWeights(n);
  for(int k=0; k<=n; k++){
    // compute denominator and numerator of the rational number w(k)
    INTEGER denom = facn*IntFactorial(k)*IntFactorial(n-k);
    INTEGER num = w(k,0)*(REAL)denom;
    // save as rational to simplify the fraction
    RATIONAL wk(num, denom);
    string nums = swrite(numerator(wk), 10000);
    string denoms = swrite(denominator(wk), 10000);
    // save fraction as string
    string wks = nums+"/"+denoms;
    // remove white-spaces
    size_t pos = wks.find(" ");
    while(pos != string::npos){
      wks.replace(pos, 1, "");
      pos = wks.find(" ");
    }
    // save to file
    file << wks << "\n";
  }
}

REALMATRIX getWeightsFromFile(int n){
  FILE* file;
  std::stringstream filename;
  filename << "weights"<<n<<".txt";
  file = fopen(filename.str().c_str(), "r");
  REALMATRIX w(n+1,1);
  char s[10000];
  // read weights from file and save to REALMATRIX
  for(int i=0; i<=n; i++){
    fgets(s, 10000, file);
    char* s2 = strtok(s, "/");
    REAL nom = s2;
    s2 = strtok(NULL, "/");
    REAL denom = s2;
    w(i,0) = nom/denom;
  }
  fclose(file);
  return w;
}
```

Beim Speichern werden dabei zunächst zwei ganze Zahlen a, b mit $w_k = \frac{a}{b}$ berechnet. Diese werden dann in eine Variable vom Typ Rational gespeichert, da dadurch ein gekürzter Bruch entsteht. Danach kann die Bruchdarstellung in eine Datei

in der Form Zähler/Nenner gespeichert werden. Dieses Verfahren ist natürlich nicht besonders effizient[1], da es jedoch in der Regel nur einmal ausgeführt wird, ist das kein wirkliches Problem.

6.1.2 Implementierung

Aus der Fehlerabschätzung der Newton-Cotes-Formeln lässt sich bei fester Ordnung n die Anzahl der nötigen Unterteilungen k berechnen, um das Integral mit einem Fehler kleiner als eine gewünschte Schranke e zu approximieren:

$$k = \frac{(b-a)^{n+2} \cdot \|f^{(n+1)}\|_\infty \cdot}{(n+1)! \cdot e}$$

Damit lässt sich 5.3 direkt implementieren:

Listing 6.3: Implementierung des Newton-Cotes-Algorithmus

```
REAL newtonCotesComp(REAL (*f)(const REAL& x), REAL prec,
    REAL maxfn, REAL a, REAL b, int n){
  // compute number of abscissas needed to get small enough error
  REAL v1 = power(b-a, n+2)*maxfn/((REAL)IntFactorial(n+1)*prec);
  INTEGER m = INTEGER(root(v1, n+1)+1);

  REAL h = (b-a)/(REAL)m;
  REAL h2 = h/(REAL)n;

  REAL res = 0;
  for(int i=0; i<=n; i++){
    REAL sumf = 0;
    for(int j=0; j<m; j++){
      REAL t = a + (REAL)j*h+(REAL)i*h2;
      sumf += (REAL)f(t);
    }
    res += sumf*(REAL)w(i,0);
  }
  res *= h2;
  return res;

}
```

Die Funktion bekommt als Eingabeparameter die zu integrierende Funktion, die gewünschte Ausgabegenauigkeit, eine Schranke an die $(n+1)$.-te Ableitung der zu integrierenden Funktion, den Start- und Endpunkt des Integrationsintervalls und die Ordnung, der Regel die zur Integration verwendet werden soll.

6.2 Gauß-Quadratur

Zum Vergleich wurde noch das Gauß-Quadraturverfahren der Ordnung 5 implementiert. Die Stützstellen wurden von [Wik11] übernommen und fest vorgegeben.

Listing 6.4: Implementierung des Gauß-Algorithmus

```
REAL GaussLegendre(REAL (*f)(const REAL& x), REAL prec, REAL maxf10, REAL a, REAL b){
  int order=5;
  // comoute error
  REAL err = maxf10*power(b-a, 2*order+1)*power(factorial(order), 4)/
        ((REAL)(2*order+1)*power(factorial(2*order),3));
    // check if error small enough
   bool reached = (bool)(choose(err < prec, err > prec/10) == 1);
  if(!reached){
    return GaussLegendre(f, prec/2, maxf10, a, (a+b)/2) +
        GaussLegendre(f, prec/2, maxf10, (a+b)/2, b);
  }
  // weights and abscissas
  REAL w[5];
```

[1] Die Speicherung der Stützstellen für $n = 500$ dauerte ca. 8 Stunden.

```
    REAL x[5];
    w[0] = (REAL)128/(REAL)225;
    w[1] = (322+13*sqrt((REAL)70))/(REAL)900;
    w[2] = (322+13*sqrt((REAL)70))/(REAL)900;
    w[3] = (322-13*sqrt((REAL)70))/(REAL)900;
    w[4] = (322-13*sqrt((REAL)70))/(REAL)900;
    x[0] = 0;
    x[1] = (1/(REAL)3)*sqrt((REAL)5-(REAL)2*sqrt((REAL)10/(REAL)7));
    x[2] = (-1)*(1/(REAL)3)*sqrt((REAL)5-(REAL)2*sqrt((REAL)10/(REAL)7));
    x[3] = (1/(REAL)3)*sqrt((REAL)5+(REAL)2*sqrt((REAL)10/(REAL)7));
    x[4] = (-1)*(1/(REAL)3)*sqrt((REAL)5+(REAL)2*sqrt((REAL)10/(REAL)7));

    REAL result = 0;
    for(int i=0; i<order; i++){
      result += w[i]*(f(((b-a)/(REAL)2)*x[i]+(a+b)/(REAL)2));
    }
    return (b-a)*result/2;
  }
```

Die Eingabeparameter sind die zu integrierende Funktion, eine Fehlerschranke, eine obere Schranke für die 10.-te Ableitung und die Grenzen des Integrationsintervalls. Zunächst wird mit der Formel 5.4 eine Schranke für den Quadraturfehler bei einmaliger Ausführung des Verfahrens berechnet. Ist der Fehler größer als die geforderte Genauigkeit, wird das Integrationsintervall in der Mitte geteilt und die Regel auf jedes Teilintervall mit halbierter Fehlerschranke angewandt. Da dazu ein Vergleich von zwei REALs notwendig ist, wird der choose-Operator verwendet.

6.3 Beispiele

6.3.1 Durchführung

Die implementierten Quadraturverfahren wurden mit 7 Beispielfunktionen getestet. Die folgenden Integrale wurden berechnet:

- $f_0 := \int_0^\pi sin(x)\,dx = 2$
- $f_1 := \int_0^\pi sin(10x)\,dx = 0$
- $f_2 := \int_0^\pi sin(1000x)\,dx = 0$
- $f_3 := \int_0^1 sin(sin(sin(x)))\,dx$
- $f_4 := \int_0^1 \frac{1}{1+x^2}\,dx = \frac{\pi}{4}$
- $f_5 := \int_0^1 e^{-x^2}\,dx$
- $f_6 := \int_0^1 e^x sin(x)\,dx$

Bei dem Integrand in f_2 handelt es sich um eine hochoszillierende Funktion. Die Integrale solcher Funktionen spielen eine große Rolle in der Praxis und sind ein aktuelles Forschungsgebiet in der Numerik. Solche Funktionen gelten im allgemeinen als schwierig zu integrieren. Die oben vorgestellten Standardquadraturverfahren sind im allgemeinen nicht dazu geeignet, solche Funktionen exakt und effizient zu integrieren.

6.3.2 Auswertung

Jede der obigen Funktion wurde mit verschiedenen Regeln und verschiedener geforderter Genauigkeit berechnet und die Rechenzeit gemessen [2]. Die Ergebnisse der Messung sind in der folgenden Tabelle angegeben. Neben der verwendeten Regel und der geforderten Genauigkeit, ist auch angegeben welche Schranke für die benötigte Ableitung dem Algorithmus gegeben wurde. Protokolliert wurde die Anzahl der Intervallteilungen k und die Laufzeit (bei den Newton-Cotes-Formeln ohne Berechnung der Stützstellen). Zu Beachten ist, dass die precision dem Algorithmus als Mindestgenauigkeit übergeben wurde. Die tatsächliche Genauigkeit liegt also meist (besonders in den Fällen mit $k = 1$ ist) deutlich darüber.

[2] Alle Rechnungen wurden auf einem Acer Aspire 7720ZG Laptop mit 1.6 Ghz Pentium dual-core Prozessor und 4 GB RAM durchgeführt.

Funktion	Regel	precision	$max f^{(n)}$	k	Zeit
f_0	GL5	10	1	7	0.01
f_0	GL5	20	1	94	0.26
f_0	GL5	50	1	98302	12.91
f_0	GL5	100	1	98302	12.91
f_0	NC5	10	1	60	0.14
f_0	NC20	10	1	2	0.02
f_0	NC20	100	1	22093	66.39
f_0	NC50	10	1	1	0.04
f_0	NC50	100	1	16	0.66
f_0	NC50	200	1	1352	31.49
f_0	NC100	10	1	1	0.11
f_0	NC100	100	1	2	0.31
f_0	NC100	200	1	9	8.09
f_0	NC200	10	1	1	1.53
f_0	NC200	100	1	1	1.93
f_0	NC200	200	1	1	1.25
f_0	NC200	500	1	14	151.28
f_0	NC500	10	1	1	6.16
f_0	NC500	100	1	1	4.35
f_0	NC500	200	1	1	5.65
f_0	NC500	500	1	1	6.24
f_0	NC500	1000	1	3	69.71
f_0	NC500	2000	1	167	817.02
f_1	GL5	10	10^{10}	94	0.03
f_1	GL5	20	10^{10}	766	0.13
f_1	NC5	10	10^{6}	590	1.45
f_1	NC20	10	10^{21}	12	0.13
f_1	NC20	100	10^{21}	220921	>1000
f_1	NC50	10	10^{51}	4	0.28
f_1	NC50	100	10^{51}	149	3.15
f_1	NC50	200	10^{51}	13510	>1000
f_1	NC100	10	10^{101}	2	0.1
f_1	NC100	100	10^{101}	9	0.37
f_1	NC100	200	10^{101}	80	3.61
f_1	NC200	10	10^{201}	1	1.08
f_1	NC200	100	10^{201}	2	6.70
f_1	NC200	200	10^{201}	5	8.77
f_1	NC200	500	10^{201}	130	135.08
f_1	NC500	10	10^{501}	1	4.28
f_1	NC500	100	10^{501}	1	6.00
f_1	NC500	200	10^{501}	1	3.61
f_1	NC500	500	10^{501}	3	46.86
f_1	NC500	1000	10^{501}	18	78.7
f_1	NC500	2000	10^{501}	-	>1000
f_2	GL5	10	10^{30}	6124	1.24
f_2	GL5	20	10^{30}	98302	19.28
f_2	NC5	10	10^{18}	58947	>1000
f_2	NC20	10	10^{63}	3426	1.35
f_2	NC20	100	10^{63}	22093	>1000
f_2	NC50	10	10^{153}	255	2.09
f_2	NC50	100	10^{153}	14786	308.59
f_2	NC50	200	10^{153}	1	-
f_2	NC100	10	10^{303}	105	6.01
f_2	NC100	100	10^{303}	811	35.04

f_2	NC100	200	10^{303}	7914	332.78
f_2	NC200	10	10^{603}	48	96.37
f_2	NC200	100	10^{603}	133	80.57
f_2	NC200	200	10^{603}	416	314.35
f_2	NC200	500	10^{603}	12900	-
f_2	NC500	10	10^{1503}	19	187.6
f_2	NC500	100	10^{1503}	28	315.82
f_2	NC500	200	10^{1503}	43	409.36
f_2	NC500	500	10^{1503}	170	443.62
f_2	NC500	1000	10^{1503}	1680	>1000
f_2	NC700	1000	10^{2103}	325	593.29
f_3	GL5	10	10^{20}	190	0.06
f_3	GL5	20	10^{20}	3070	1.48
f_3	NC5	10	10^{3}	50	0.08
f_3	NC50	10	10^{40}	1	0.1
f_3	NC50	100	10^{40}	29	3.44
f_3	NC50	200	10^{40}	2560	241.15
f_3	NC100	10	10^{100}	15	0.68
f_3	NC100	100	10^{100}	3	5.99
f_3	NC100	200	10^{100}	25	20.03
f_3	NC500	10	10^{800}	1	8.32
f_3	NC500	100	10^{800}	1	9.92
f_3	NC500	200	10^{800}	2	19.29
f_3	NC500	500	10^{800}	3	106.2
f_3	NC500	1000	10^{800}	22	376.92
f_4	GL5	10	10^{15}	94	0.01
f_4	GL5	20	10^{15}	766	0.08
f_4	NC5	10	720	47	0.02
f_4	NC20	10	10^{20}	4	0.09
f_4	NC20	20	10^{20}	10	0.02
f_4	NC50	10	10^{70}	3	2.03
f_4	NC50	20	10^{70}	4	1.16
f_4	NC50	100	10^{70}	110	1.69
f_4	NC50	200	10^{70}	2560	241.15
f_4	NC100	10	10^{200}	4	10.29
f_4	NC100	20	10^{200}	5	15.22
f_4	NC100	100	10^{200}	25	8.79
f_4	NC100	200	10^{200}	239	11.58
f_4	NC500	10	10^{1000}	2	1.35
f_4	NC500	100	10^{1000}	2	0.48
f_4	NC500	200	10^{1000}	2	19.29
f_4	NC500	500	10^{1000}	6	72.72
f_4	NC500	1000	10^{1000}	54	87.26
f_5	GL5	10	25000	4	0.01
f_5	GL5	20	25000	46	0.03
f_5	GL5	100	25000	46	0.03
f_5	NC5	10	150000	107	0.08
f_5	NC5	20	150000	10564	5.52
f_5	NC20	10	10^{13}	2	0.01
f_5	NC20	20	10^{13}	5	0.01
f_5	NC20	100	10^{13}	27700	116.16
f_5	NC50	10	10^{41}	2	0.04
f_5	NC50	20	10^{41}	2	0.04
f_5	NC50	100	10^{41}	30	1.17

f_5	NC50	200	10^{41}	2678	139
f_5	NC100	10	10^{100}	1	0.23
f_5	NC100	20	10^{100}	1	0.23
f_5	NC100	100	10^{100}	3	2.86
f_5	NC100	200	10^{100}	25	12.4
f_5	NC500	10	10^{800}	1	2.79
f_5	NC500	100	10^{800}	1	3.46
f_5	NC500	200	10^{800}	2	6.98
f_5	NC500	500	10^{800}	3	40.72
f_5	NC500	1000	10^{800}	22	98.68
f_6	GL5	10	2^{12}	4	0.01
f_6	GL5	20	2^{12}	40	0.03
f_6	NC5	10	2^{8}	23	0.04
f_6	NC5	20	2^{8}	1019	0.83
f_6	NC20	10	2^{23}	1	0.01
f_6	NC20	20	2^{23}	2	0.01
f_6	NC20	100	2^{23}	7600	55.25
f_6	NC50	10	2^{53}	1	0.02
f_6	NC50	20	2^{53}	1	0.04
f_6	NC50	100	2^{53}	6	0.4
f_6	NC50	200	2^{53}	451	38.21
f_6	NC100	10	2^{103}	1	0.36
f_6	NC100	20	2^{103}	1	0.37
f_6	NC100	100	2^{103}	1	0.39
f_6	NC100	200	2^{103}	4	6.41
f_6	NC500	10	2^{503}	1	4
f_6	NC500	100	2^{503}	1	4.24
f_6	NC500	200	2^{503}	1	4.08
f_6	NC500	500	2^{503}	1	4.13
f_6	NC500	1000	2^{503}	2	8.12
f_6	NC500	2000	2^{503}	54	770.52

Anhand der Beispiele sieht man, dass sich nur Formeln hoher Ordnung dazu eignen, Integrale auf sehr viele Stellen exakt zu berechnen. Sind dagegen nur wenige Stellen erforderlich, kann auch eine Regel niedrigerer Ordnung effizienter sein. Dies hängt natürlich einmal damit zusammen, dass bei zu hoher Ordnung das Ergebnis schon bei einmaliger Ausführung exakter bestimmt wird als nötig. Außerdem müssen die Zwischenberechnungen bei höherer Ordnung mit einer höheren Genauigkeit durchgeführt werden, so dass diese mehr Zeit benötigen. Die Effizienz des implementierten Verfahrens hängt maßgeblich von der Schranke an die Ableitung ab. Daher eignet es sich besonders für die Integration von Funktionen, bei denen auch noch sehr hohe Ableitungen im Integrationsintervall beschränkt sind. Auch wenn die Schranken an die Ableitungen mit höheren Ableitungen sehr schnell größer werden, ist das Verfahren eher ungeeignet. Wie erwartet benötigt deshalb die Berechnung von f_2 sehr viel Zeit. In der Tabelle sieht man, dass die Berechnung von 1000 Dezimalstellen nur mit 700 Stützstellen in unter 1000 Sekunden möglich ist. Die Newton-Cotes-Regeln eignen sich (genau wie die Gauß-Quadratur) nicht, um hochoszillierende Funktionen exakt zu integrieren. Für solche Funktionen gibt es jedoch spezielle Verfahren, mit denen eine effiziente Berechnung möglich ist. Ein Überblick findet sich z.B. in [Olv08].

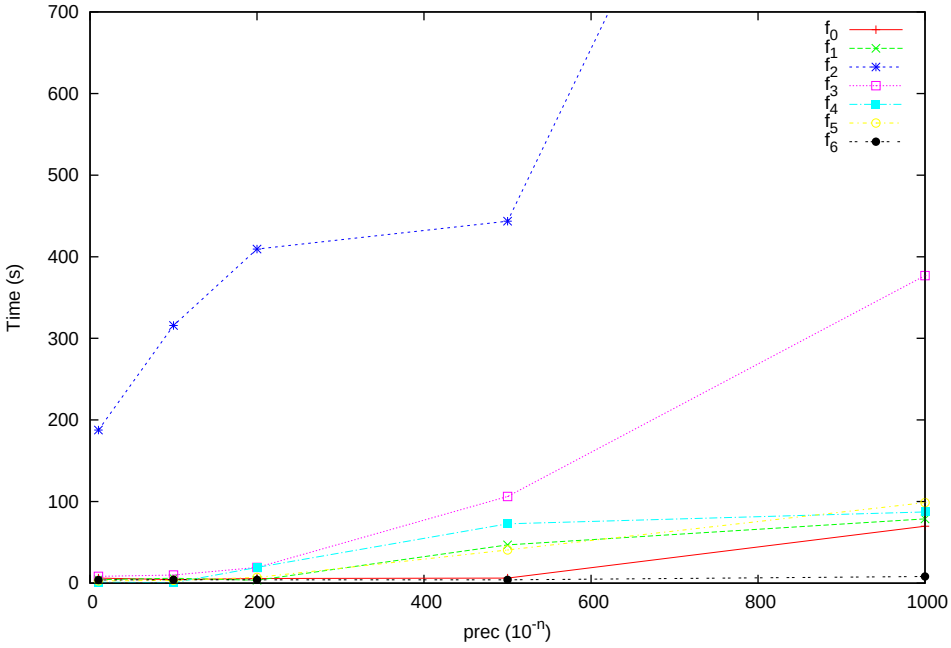

Abbildung 6.1: Die benötigte Rechenzeit in Abhängigkeit von der gegebenen Fehlerschranke bei der 500-Punkte-Regel

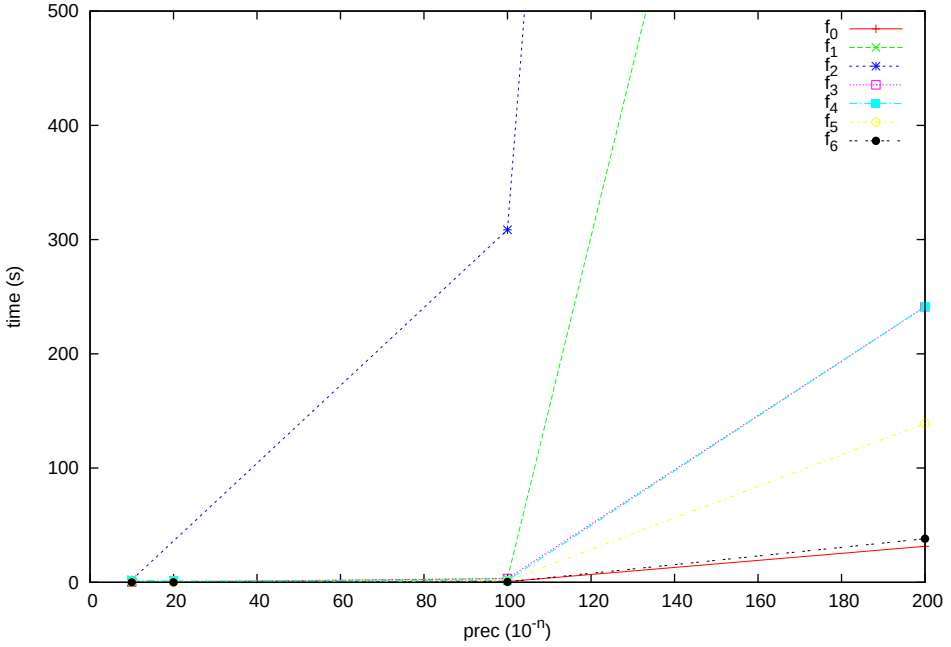

Abbildung 6.2: Die benötigte Rechenzeit in Abhängigkeit von der gegebenen Fehlerschranke bei der 50-Punkte-Regel

7 Fazit und Ausblick

Die Implementierung des Newton-Cotes-Verfahrens zeigt, dass Integration in der Praxis effizient möglich ist, sofern die zu integrierende Funktion ausreichend oft differenzierbar ist und die Ableitungen nicht zu stark wachsen. Dies deckt sich mit dem theoretischen Resultat von Ko und Friedman, dass Integration von analytischen und polynomialzeitberechenbaren Funktionen effizient möglich ist. Trotzdem bleibt die Integration insgesamt natürlich ein schwieriges Problem. Ist eine Funktion nur wenige Male differenzierbar, so kann auch nur eine Regel niedriger Ordnung angewendet werden, die im allgemeinen nicht sehr effizient ist. In der numerischen Praxis werden die Newton-Cotes-Formeln kaum genutzt, da sie sich bei begrenzter Rechengenauigkeit nicht für exakte Berechnungen eignen. Bei Regeln höherer Ordnung verstärken sich Rundungsfehler sehr stark, so dass das Ergebnis verfälscht wird. In der iRRAM treten jedoch keine Rundungsfehler auf, so dass die effizienten Formeln höherer Ordnung problemlos genutzt werden können. In der iRRAM können also Verfahren sinnvoll eingesetzt werden, die bisher aufgrund von begrenzter Rechengenauigkeit nicht verwendet werden.

Für zukünftige Verbesserungen und Weiterentwicklungen bieten sich folgende Themen an:

Verbesserung der Fehlerschranken

Bei der Auswertung einiger Beispielfunktionen zeigt sich, dass die bei der Newton-Cotes-Regel verwendete Approximation des Fehlers sehr viel höher ist als der wirkliche Fehler. Eine Verbesserung dieser Approximation würde das Verfahren vermutlich deutlich effizienter machen. Außerdem benötigt die Berechnung des Fehlers (bzw. der Anzahl der benötigten Intervallteilungen) bei hoher Genauigkeit sehr viel Rechenzeit. Dies macht sich vor allem bei Regeln höherer Ordnung bemerkbar. Eine Vereinfachung der Fehlerapproximation könnte zu geringeren Rechenzeiten führen.

Vergleich mit anderen Verfahren

In dieser Arbeit wurden nur die Newton-Cotes-Regeln beliebiger Ordnung implementiert. Interessant wäre ein Vergleich mit der Gauß-Quadratur. Hier stellt sich die Frage, ob die Gauß-Quadratur, obwohl bei jeder Rechnung die Stützstellen berechnet werden müssen, immernoch effizienter ist als die Newton-Cotes-Formeln mit vorberechneten Stützstellen. Auch könnten Verfahren implementiert werden, die für spezielle Klassen von Funktionen, wie beispielsweise hochoszillierende Funktionen, besser geeignet sind.

automatische Wahl der Regel

Der hier implementierte Algorithmus zur Berechnung der Newton-Cotes-Formeln erhält die Ordnung der zu verwendeten Regel als Eingabeparameter. Man könnte auch einen Algorithmus implementieren, der nicht nur die Anzahl der Unterteilungen, sondern auch die Ordnung der Regel automatisch wählt. Allerdings erfordert dies, dass dem Algorithmus noch mehr Information über die Funktion gegeben werden muss, da man nun Schranken für alle Ableitungen angeben muss.

Mehrdimensionale Integration

Neben den Quadraturverfahren wurden in der Numerik auch viele Verfahren zur mehrdimensionalen numerischen Integration (manchmal auch Kubatur genannt). Auch solche Verfahren könnten in der iRRAM implementiert werden.

Anhang

A Quelltext

Listing 7.1: Kompletter Quelltext

```
#include "iRRAM.h"
#include "stdio.h"
#include <sstream>

using namespace iRRAM;


// function used for testing
REAL fun1(const REAL& x){
    return sin(10*x);
}

// function used for testing
REAL fun2(const REAL& x){
    return sin(1000*x);
}

// function used for testing
REAL fun3(const REAL& x){
    return sin(sin(sin(x)));
}

// function used for testing
REAL fun4(const REAL& x){
  return 1/(1+x*x);
}


// function used for testing
REAL fun5(const REAL& x){
  return exp(-x*x);
}

// function used for testing
REAL fun6(const REAL& x){
  return exp(x)*sin(x);
}


REALMATRIX getWeights(int n){
  REALMATRIX l(n+1,n+1);
  REALMATRIX r(n+1,1);
  for(int i=0; i<=n; i++){
    r(i,0) = power((REAL)n,i+1)/(REAL)(i+1);
    for(int k=0; k<=n; k++){
      l(i,k) = power((REAL)k, i);
    }
  }
  REALMATRIX s = solve(l, r,1);
  return s;
}


REAL factorial(int n){
  REAL r = 1;
  for(int i = 2; i<=n; i++)
    r *= i;
  return r;
}
```

```
INTEGER IntFactorial(int n){
  INTEGER r = 1;
  for(int i = 2; i<=n; i++)
    r *= i;
  return r;
}

void saveWeightsToFile(int n){
  INTEGER facn = IntFactorial(n+1);
  REALMATRIX w;
  std::stringstream filename;
  filename << "weights"<<n<<".txt";
  orstream file(filename.str(), std::_S_out);
  // compute weights
  w = getWeights(n);
  for(int k=0; k<=n; k++){
    // compute denominator and numerator of the rational number w(k)
    INTEGER denom = facn*IntFactorial(k)*IntFactorial(n-k);
    INTEGER num = w(k,0)*(REAL)denom;
    // save as rational to simplify the fraction
    RATIONAL wk(num, denom);
    string nums = swrite(numerator(wk), 10000);
    string denoms = swrite(denominator(wk), 10000);
    // save fraction as string
    string wks = nums+"/"+denoms;
    // remove white-spaces
    size_t pos = wks.find(" ");
    while(pos != string::npos){
      wks.replace(pos, 1, "");
      pos = wks.find(" ");
    }
    // save to file
    file << wks << "\n";
  }
}

REALMATRIX getWeightsFromFile(int n){
  FILE* file;
  std::stringstream filename;
  filename << "weights"<<n<<".txt";
  file = fopen(filename.str().c_str(), "r");
  REALMATRIX w(n+1,1);
  char s[10000];
  // read weights from file and save to REALMATRIX
  for(int i=0; i<=n; i++){
    fgets(s, 10000, file);
    char* s2 = strtok(s, "/");
    REAL nom = s2;
    s2 = strtok(NULL, "/");
    REAL denom = s2;
    w(i,0) = nom/denom;
  }
  fclose(file);
  return w;
}


REALMATRIX w;

// GaussLegendre Algorithm of order 5
REAL GaussLegendre(REAL (*f)(const REAL& x), REAL prec, REAL maxf10, REAL a, REAL b){
  int order=5;
  // comoute error
  REAL err = maxf10*power(b-a, 2*order+1)*power(factorial(order), 4)/
        ((REAL)(2*order+1)*power(factorial(2*order),3));
    // check if error small enough
   bool reached = (bool)(choose(err < prec, err > prec/10) == 1);
```

```
    if(!reached){
      return GaussLegendre(f, prec/2, maxf10, a, (a+b)/2) +
          GaussLegendre(f, prec/2, maxf10, (a+b)/2, b);
    }
    // weights and abscissas
    REAL w[5];
    REAL x[5];
    w[0] = (REAL)128/(REAL)225;
    w[1] = (322+13*sqrt((REAL)70))/(REAL)900;
    w[2] = (322+13*sqrt((REAL)70))/(REAL)900;
    w[3] = (322-13*sqrt((REAL)70))/(REAL)900;
    w[4] = (322-13*sqrt((REAL)70))/(REAL)900;
    x[0] = 0;
    x[1] = (1/(REAL)3)*sqrt((REAL)5-(REAL)2*sqrt((REAL)10/(REAL)7));
    x[2] = (-1)*(1/(REAL)3)*sqrt((REAL)5-(REAL)2*sqrt((REAL)10/(REAL)7));
    x[3] = (1/(REAL)3)*sqrt((REAL)5+(REAL)2*sqrt((REAL)10/(REAL)7));
    x[4] = (-1)*(1/(REAL)3)*sqrt((REAL)5+(REAL)2*sqrt((REAL)10/(REAL)7));

    REAL result = 0;
    for(int i=0; i<order; i++){
      result += w[i]*(f(((b-a)/(REAL)2)*x[i]+(a+b)/(REAL)2));
    }
    return (b-a)*result/2;
}

// composite Newton-Cotes Algorithm
REAL newtonCotesComp(REAL (*f)(const REAL& x), REAL prec,
    REAL maxfn, REAL a, REAL b, int n){
  // compute number of abscissas needed to get small enough error
  REAL v1 = power(b-a, n+2)*maxfn/((REAL)IntFactorial(n+1)*prec);
  INTEGER m = INTEGER(root(v1, n+1)+1);

  REAL h = (b-a)/(REAL)m;
  REAL h2 = h/(REAL)n;

  REAL res = 0;
  for(int i=0; i<=n; i++){
    REAL sumf = 0;
    for(int j=0; j<m; j++){
      REAL t = a + (REAL)j*h+(REAL)i*h2;
      sumf += (REAL)f(t);
    }
    res += sumf*(REAL)w(i,0);
  }
  res *= h2;
  return res;

}

void compute() {
  double start_time = clock();
  double end_time = clock();
  double beginning_time = start_time;
  //saveWeightsToFile(700);
  //w = getWeightsFromFile(700);

  int dec[7] = {10, 20, 100, 200, 500, 1000, 2000};
  int rule[9] = {5, 10, 20,40, 50, 100, 200, 250, 500};
  typedef REAL (*REALFUNCTION) (const REAL &x);
  REALFUNCTION functions[] = {sin, fun1, fun2, fun3, fun4, fun5, fun6};

    int i, j,f, e, a, b;
  cout << "select rule: 5 [0], 10 [1], 20 [2], 40 [3], "
     << "50 [4], 100 [5], 200 [6], 250 [7], 500 [8]" << std::endl;
  cin >> i;
  cout << "select number of decimals: 10 [0], 20 [1], 100 [2], "
     << "200 [3], 500 [4], 1000 [5], 2000 [6] " << std::endl;
  cin >> j;
  cout << "select function [0 - 6]" << std::endl;
  cin >> f;
```

```
    cout << "integrate from:" << std::endl;
201 cin >> a;
    cout << "to:" << std::endl;
203 cin >> b;
    cout << "give an upper bound e, so that the " << (i+1)
205    << ". derivative of f in ["<<a<<","<<b<<"] is <= 10^e"
       << std::endl;
207 cin >> e;
    if(rule[i] == 500)
209   w = getWeightsFromFile(500);
    else
211   w = getWeights(rule[i]);

213 REAL prec = power(10, -1*dec[j]);
    cout << "Computing " << dec[j] << " digits of int f" << f
215    <<" dx from " << a << " to " << b << " with rule "
       << rule[i] << std::endl;
217 start_time = clock();
    REAL x = newtonCotesComp(functions[f], prec,power(10, e), a,b, rule[i]);
219 cout << setRwidth(dec[j]+10) << x << std::endl;
    end_time = clock();
221 cout << "Time for computing " << dec[j] << " digits of int f" << f
       <<" dx from " << a << " to " << b << " with rule "
223    << rule[i] << ":"
       << (end_time - start_time )/CLOCKS_PER_SEC
225    << std::endl;
  }
```

Literaturverzeichnis

[BH98] Vasco Brattka and Peter Hertling. Feasible real random access machines. *Journal of Complexity*, 14(4):490 – 526, 1998.

[BHW08] Vasco Brattka, Peter Hertling, and Klaus Weihrauch. A Tutorial on Computable Analysis. In S. Barry Cooper, Benedikt Löwe, and Andrea Sorbi, editors, *New Computational Paradigms*, pages 425–491. Springer, 2008.

[EMNW10] Gisela Engeln-Müllges, Klaus Niederdrenk, and Reinhard Wodicka. *Numerik-Algorithmen: Verfahren, Beispiele, Anwendungen (Xpert.press) (German Edition)*. Springer, 10. überab. u. erw. aufl. edition, 9 2010.

[Fri84] H Friedman. The computational complexity of maximization and integration. *Advances in Mathematics*, 53(1):80–98, 1984.

[Ko91] Ker-I Ko. *Complexity theory of real functions*. Birkhauser Boston Inc., Cambridge, MA, USA, 1991.

[Kü94] A. R. Krommer and C. W. überhuber. *Numerical integration on advanced computer systems*. Springer, Berlin, 1994.

[MÖ1] Norbert Th. Müller. The irram: Exact arithmetic in c++. In *Selected Papers from the 4th International Workshop on Computability and Complexity in Analysis*, CCA '00, pages 222–252, London, UK, 2001. Springer-Verlag.

[Olv08] S. Olver. *Numerical approximation of highly oscillatory integrals*. University of Cambridge, 2008.

[Sch08] U. Schöning. *Theoretische Informatik- kurz gefasst*. Spektrum Hochschultaschenbücher. Spektrum Akademischer Verlag, 2008.

[Wei00] Klaus Weihrauch. *Computable Analysis : An Introduction (Texts in Theoretical Computer Science. An EATCS Series)*. Springer, November 2000.

[Wik11] Wikipedia. Gaussian quadrature — wikipedia, the free encyclopedia, 2011. [Online; accessed 30-July-2011].

[Yu07] F. Yu. *On the complexity of real and complex functions defined on the two-dimensional plane*. State University of New York at Stony Brook, 2007.