Ein effektiver Zwischenwertsatz als Fallstudie für exakte reelle Arithmetik
关于实数精确计算的一个可行中值定理

Jingwen Zhang

15. Juli 2011

Zusammenfassung

In dieser Bachelorarbeit wurde das Trisektionsverfahren unter Zuhilfenahme der C++–Bibliothek iRRAM implementiert, um Nullstellen von stetigen, berechenbaren, reellen Funktionen beliebig genau zu approximieren. Die Arbeit geht auf die theoretischen Grundlagen zu Berechenbarkeit von reellen Zahlen, Folgen und Funktionen ein. Die Berechenbarkeit einer Nullstelle für eine beliebige berechenbare stetige, reelle Funktion mit dem Trisektionsverfahren wird bewiesen und die Implementierung des Algorithmus mit iRRAM ausführlich erklärt. Dabei werden auch die wesentlichen Funktionen von iRRAM kurz erläutert.

Inhaltsverzeichnis

1 Einleitung

Im Rahmen dieser Arbeit wurde ein Programm erstellt, welches mit dem Trisektionsverfahren Nullstellen von berechenbaren reellen Funktionen beliebig genau berechnen kann. Dazu verwendet das Programm die C++-Bibliothek iRRAM, welche von Prof. Norbert Müller an der Universität Trier geschrieben wurde. iRRAM simuliert eine Real-RAM Maschine, wobei diese iterativ simuliert wird, so daß eine beliebige Ausgabegenauigkeit erreicht werden kann.

Die Arbeit ist in drei Teile gegliedert, zunächst wird als theoretische Grundlage die Berechenbarkeit von reellen Zahlen, Folgen und Funktionen untersucht und bewiesen, daß das Trisektionsverfahren verwendet werden kann, um die Nullstellen von reellen Funktionen (unter bestimmten Umständen) zu berechnen.

Danach werden die C++–Bibliothek iRRAM und die wesentlichen Funktionen, welche zur Implementierung des Trisektionsverfahren benötigt wurden, kurz vorgestellt und erklärt. Im dritten Teil werden dann die wichtigsten Teile der Implementierung des Programms erläutert.

Die Dokumentation ist auch auf Chinesisch (unter dem Titel „关于实数精确计算的一个可行中值定理“) verfügbar.

2 Grundlagen

Das vorliegende Programm kann approximativ mit unbeschränkter Genauigkeit rechnen und basiert auf den Erkenntnissen der Berechenbaren Analysis.

In der Analysis werden reelle Zahlen, Funktionen und darauf aufbauende Strukturen untersucht, während in der Berechenbaren Analysis untersucht wird, wann und wie man diese „reellen“ Strukturen mit digitalen Computern berechnen kann.

Der Begriff der Berechenbarkeit hängt dabei eng mit dem der Turingmaschine zusammen, welche im folgenden Kapitel eingeführt wird.

2.1 Turingmaschine

„Die Turingmaschine ist ein von dem britischen Mathematiker Alan Turing im Jahre 1936 entwickeltes Modell, um eine Klasse von berechenbaren Funktionen zu bilden. Sie gehört zu den grundlegenden Konzepten der Theoretischen Informatik.“ [6]

Sie ist ein idealisiertes Modell eines sehr einfachen Computers und dient dazu, Algorithmen zur Berechnung von Funktionen formal zu definieren.

Gemäß der „Church-Turing-These“ entspricht die Klasse der Turing-berechenbaren Funktionen genau der Klasse der „intuitiv berechenbaren“ Funktionen. Diese These ist nicht beweisbar, da der Begriff der „intuitiv berechenbaren Funktionen“ formal nicht faßbar ist. Um die Nicht-Berechenbarkeit von Funktionen zu beweisen, wird jedoch oft angenommen, daß die These wahr ist. Weder Turing noch andere fanden jemals ein Rechenmodell welches mächtiger ist als das der Turingmaschine.

2.2 Berechenbarkeit reeller Zahlen

Das grundlegende Hilfsmittel zur Berechnung reeller Zahlen, Folgen, Potenzreihen und Funktionen sind die Cauchy-Folgen[1]. Ausgehend von der Berechenbarkeit rationaler Folgen, kann die Berechenbarkeit reeller Zahlen und reeller Folgen leicht definiert werden [5]:

Definition 2.2.1. *Eine reelle Zahl x heißt berechenbar, falls eine berechenbare rationale Folge (ε_m) exisitiert, und*

$$|x - q_m| \leq \varepsilon_m \xrightarrow{m\to\infty} 0$$

gilt, für eine berechenbare rationale Folge (q_m), $\forall m \in \mathbb{N}$.

Äquivalent dazu gilt folgende Definition:

Definition 2.2.2. *Eine reelle Zahl x heißt berechenbar, falls*

$$|x - q_n| \leq 2^{-n}$$

gilt, für eine berechenbare rationale Folge (q_n), $\forall n \in \mathbb{N}$.

2.3 Berechenbarkeit reeller Folgen

Nun kann die Berechenbarkeit von reellen Zahlen bewiesen werden. Da Computer nur über begrenzten Speicher verfügen, können diese reellen Zahlen zwar berechnet, aber nicht gespeichert werden. Daher werden die reelle Zahlen angenähert, wodurch eine Folge reeller Zahlen entsteht. Damit stellt sich die Frage nach der Berechenbarkeit von Folgen von reellen Zahlen:

Definition 2.3.1. *Eine reelle Folge (x_m) heißt berechenbar, falls*

$$|x_m - q_{m,n}| \leq 2^{-n}$$

gilt, für eine berechenbare rationale Doppelfolge $(q_{m,n})$, $\forall m, n \in \mathbb{N}$.

Um eine reelle Zahl anzunähern, muß für eine solche Folge gelten:

Definition 2.3.2. *Eine berechenbare reelle Folge (x_m) heißt effektiv konvergent, falls eine berechenbare rationale Folge (ε_m) existiert, und gilt:*

$$|x_m - x| \leq \varepsilon_m \xrightarrow{m\to\infty} 0$$

Dies führt uns zu folgendem Satz:

Satz 2.3.3. *Es sei (x_m) eine berechenbare reelle Folge und konvergiert effektiv. Dann ist $x := \lim_{m\to\infty} x_m$ berechenbar.*

[1] Diese werden auch als „Fundamentalfolgen“ bezeichnet.

Beweis. (x_m) ist eine berechenbare reelle Folge,

$$|x_m - q_{m,n}| \leq 2^{-n}, \text{ für eine berechenbare rationale Folge } (q_{m,n})$$

(x_m) konvergiert effektiv, es existiert eine berechenbare rationale Folge (ε_m),

$$|x_m - x| \leq \varepsilon_m \xrightarrow{m\to\infty} 0$$

Nun setzen wir $q_m := q_{m,m} \subseteq \mathbb{Q}$ berechenbar

$$|x - q_m| = |x - x_m + x_m - q_m| \leq \underbrace{|x - x_m|}_{\varepsilon_m \xrightarrow{m\to\infty} 0} + \underbrace{|x_m - q_{m,m}|}_{2^{-m} \xrightarrow{m\to\infty} 0} \xrightarrow{m\to\infty} 0$$

Damit ist x berechenbar nach der Definition 2.2.1. □

Daraus kann man ableiten, daß effektive Konvergenz ein hinreichendes Kriterium für die Berechenbarkeit des Grenzwerts einer berechenbaren Folge ist.

2.4 Berechenbarkeit von reellen Funktionen

Ausgehend von der Berechenbarkeit von reellen Folgen kann nun die Berechenbarkeit von reellen Funktionen definiert werden. Zunächst betrachten wir die Potenzreihe und beweisen daß sie berechenbar ist:

Satz 2.4.1. *Es sei $P(x) := \sum_{n=0}^{\infty} a_n x^n$ eine Potenzreihe mit einer berechenbaren reellen Koeffizientenfolge a_n, und konvergiere effektiv im Inneren ihres Konvergenzradius*

$$R := \frac{1}{\lim_{n\to\infty} \sup \sqrt[n]{|a_n|}}$$

Dann ist $P(x)$ berechenbar, wenn x eine berechenbare reelle Zahl ist.

Zwischen zwei reellen Zahlen a und b mit $a < b$ liegt immer wenigstens eine rationale Zahl r mit $a < r < b$. Damit liegen zwischen zwei reellen Zahlen auch unendlich viele rationale Zahlen. Man sagt auch, die rationalen Zahlen liegen „dicht“ in den reellen Zahlen.

Beweis.

$$x, R \in \mathbb{R}, |x| < R \Rightarrow \exists q, r \in \mathbb{Q},\ |x| < q < r < R$$

Cauchy-Hadamard: $\exists M \in \mathbb{N}, |a_n| < Mr^{-n} \forall n \in \mathbb{N}$

$$\underbrace{\left|\sum_{k=0}^{\infty} a_k x^k\right.}_{P(x)=:x} - \underbrace{\left.\sum_{k=0}^{m} a_k x^k\right|}_{x_m} = \left|\sum_{k=m+1}^{\infty} a_k x^k\right| \leq \sum_{k=m+1}^{\infty} |a_k||x|^k$$

$$\overset{\text{Cauchy-Hadamard}}{<} \sum_{k=m+1}^{\infty} Mr^{-k}|x|^k = M \sum_{k=m+1}^{\infty} \left(\frac{|x|}{r}\right)^k = M \left(\frac{|x|}{r}\right)^{m+1} \underbrace{\sum_{k=0}^{\infty} \left(\frac{|x|}{r}\right)^k}_{\text{geometrische Reihe}}$$

$$= M \left(\frac{|x|}{r}\right)^{m+1} \frac{1}{1-\frac{|x|}{r}} \overset{|x|<q<r}{\leq} \underbrace{M \left(\frac{q}{r}\right)^{m+1} \frac{1}{1-\frac{q}{r}}}_{\in \mathbb{Q}} =: \varepsilon_m \xrightarrow{m\to\infty} 0 \text{ berechenbar und}$$

x_m ist berechenbar (aus dem Vortrag von Herrn Neeb im *Proseminar Berechenbare Analysis* [4])

$\Rightarrow x_m$ konvergiert effektiv nach der Definition 2.3.2.
Damit ist $P(x)$ berechenbar nach dem Satz 2.3.3.

□

Nun ist die Berechenbarkeit folgender Funktionen einfach nachzuvollziehen:

2.4.1 Exponentialfunktion

$$e^x = \sum_{n=0}^{\infty} \frac{1}{n!} x^n$$

Die Koeffizientenfolge $a_n = \frac{1}{n!} \in \mathbb{Q}$ ist berechenbar und der Konvergenzradius $R = \infty$. Damit ist die *Exponentialfunktion* e^x nach Satz 2.4.1 $\forall x \in \mathbb{R}$ berechenbar.

2.4.2 Trigonometrische Funktionen

Betrachten wir nun die *Sinusfunktion*:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

Die Koeffizientenfolge

$$a_n = \begin{cases} -\frac{1}{m!}, & m = 2n+1 \\ 0, & m = 2n \end{cases}$$

ist berechenbar und der Konvergenzradius $R = \infty$. Nach Satz 2.4.1 ist $\sin x$ ebenfalls $\forall x \in \mathbb{R}$ berechenbar.

Analog zur Sinusfunktion ist auch die *Cosinusfunktion*

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

berechenbar.

2.4.3 Logarithmusfunktion

Die *Logarithmusfunktion* ist ebenfalls berechenbar:

$$\ln(1+x) = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{x^n}{n}$$

Wobei die Koeffizientenfolge

$$a_n = -\frac{(-1)^n}{n} \in \mathbb{Q}, \forall n > 0$$

berechenbar ist, mit folgendem Konvergenzradius:

$$R = \frac{1}{\lim_{n\to\infty} \sup \sqrt[n]{\frac{1}{n}}} = 1$$

$$r = 1$$

$$|x| < 1$$

$$I := (0, 2)$$

Nach Satz 2.4.1 ist auch die *Logarithmusfunktion* $\ln(1+x)$, $\forall |x| < 1$ berechenbar.

Für diese Funktionen wurde nun beispielhaft die Berechenbarkeit bewiesen, es sind jedoch fast alle analytischen Funktionen der Mathematik berechenbar, es gibt also Rechenverfahren, welche nach endlich vielen Schritten den Funktionswert ermitteln. [1]

2.5 Zwischenwertsatz

Bis jetzt wurde gezeigt, wann für eine berechenbare reelle Zahl x die Funktion $x \to f(x)$ berechenbar ist. Nun soll untersucht werden, unter welchen Bedingungen für eine berechenbare reelle Funktion $f(x)$ reelle Nullstelle x berechenbar ist, eine Zahlt x also, für die gilt $f(x) = 0$.

Die Existenz einer solchen reellen Zahl wird durch den Zwischenwertsatz garantiert:

Satz 2.5.1. *Es sei $f : [a, b] \to \mathbb{R}$ eine stetige reelle Funktion, die auf einem Intervall definiert ist. Dann existiert zu jedem $f(a) \leq u \leq f(b)$ bzw. $f(b) \leq u \leq f(a)$ ein $x \in [a, b]$ mit $f(x) = u$.*

Es sei nun $f : [0, 1] \to \mathbb{R}$ eine stetige reelle Funktion, die auf dem Intervall [0,1] definiert ist. Haben f(0) und f(1) verschiedene Vorzeichen, dann garantiert der Zwischenwertsatz die Existenz von mindestens einer Nullstelle $x \in [0, 1]$ mit $f(x) = 0$. [1, Seite 458] Dieser Sonderfall wurde zuerst von Bernard Bolzano im Jahre 1817 bewiesen und ist als Nullstellensatz von Bolzano bekannt. [7]

2.6 Trisektionsverfahren

Nun wird bewiesen, daß eine Nullstelle $x \in [0,1]$ mit $f(x) = 0$ näherungsweise aus gegebener, berechenbarer, stetiger Funktion $f : [0,1] \to \mathbb{R}$ mit $f(0) * f(1) < 0$ berechnet werden kann, unter der Einschränkung, daß f in $[0,1]$ genau eine Nullstelle hat.

Satz 2.6.1. *Es sei $f : [0,1] \to \mathbb{R}$ mit $f(0) * f(1) < 0$ eine berechenbare stetige reelle Funktion, die genau eine Nullstelle hat. Dann ist die Nullstelle berechenbar.*

Beweis. [1, Seite 459] Beim klassischen Bisektionsverfahren wird der Funktionswert mit 0 verglichen. Aber dieser Vergleich ist nicht sicher in endlicher Zeit möglich. Deswegen benutzen wir das Trisektionsverfahren, welches eine stabilere Variante des Bisektionsverfahrens ist. Angefangen mit $a_0 := 0$ und $d_0 := 1$, angenommen $a_i, d_i \in [0,1]$ mit $d_i - a_i = (2/3)^i$ und $f(a_i) * f(d_i) < 0$ (Das ist richtig, für $i = 0$).Wir berechnen gleichzeitig den Wert von $f(a_i) * f(a_i + \frac{2}{3}(d_i - a_i))$ und $f(a_i + \frac{1}{3}(d_i - a_i)) * f(d_i)$. Wenn ein Wert kleiner als 0 ist, rechnen wir eine Iteration weiter. Das Trisektionsverfahren garantiert dabei, daß mindestens ein Wert von beiden kleiner als 0 ist. Wenn der erste Wert kleiner als 0 ist, dann gilt $a_{i+1} := a_i$, $d_{i+1} := a_i + \frac{2}{3}(d_i - a_i)$. Ist der zweite Wert kleiner als 0, dann gilt $a_{i+1} := a_i + \frac{1}{3}(d_i - a_i)$, $d_{i+1} := d_i$. So entsteht eine Folge von Intervallen $[a_i, d_i]$ mit der Länge $(2/3)^i$, welche gegen die Nullstelle von f konvergiert. Damit ist die Nullstelle nach dem Satz 2.3.3 auch berechenbar. □

Für Funktionen mit mehreren Nullstellen ist das Trisektionsverfahren nicht direkt anwendbar, stattdessen sind kompliziertere Verfahren notwendig.

3 iRRAM (iterative Real-RAM)

iRRAM ist eine C++ Bibliothek mit deren Hilfe fehlerfreie Berechnungen mit reellen Zahlen durchgeführt werden können. Die Bibliothek basiert auf der Idee einer Real-RAM, also einer Turingmaschine, welche mit reellen Zahlen rechnen kann (statt mit diskreten Zahlen, wie die normale Turingmaschine). iRRAM simuliert eine solche Real-RAM Maschine, wobei die Genauigkeit des Ergebnisses iterativ erhöht werden kann, um eine beliebig bestimmbare Ausgabegenauigkeit zu erreichen: „iterative Real-RAM“ [3]

In diesem Kapitel wird eine kurze Einführung in die Grundstruktur von iRRAM gegeben und werden die wichtigsten Funktionen der Bibliothek vorgestellt, welche zur Implementierung des Zwischenwertsatzes verwendet wurden.

3.1 Einführung in iRRAM

Um iRRAM zu verwenden, genügt es, für alle Zahlen, die reell sind, die Klasse `REAL` zu verwenden. Listing 1 zeigt eine Beispielimplementation aus der iRRAM-Dokumentation, den Algorithmus von Heron, welcher $\sqrt{x}$ näherungsweise berechnet:

```
1  // Funktion, welche die Wurzel zur Basis 2 approximiert
2  REAL wurzel_approx(long p, REAL x)
3  {
4   // Definiere zwei reelle Zahlen, a und b
5   REAL a = 1, b = x;
6   // Iteriere den Algorithmus, bis das Ergebnis der vorgegebenen Genauigkeit entspricht
7   do {a = (a + b) / 2; b = x / a; } while( !bound(a - b, p) );
8   // Liefere das Ergebnis zurueck
9   return a;
10 }
11
12 // Der Grenzwert der reellen Folge entspricht der "reellen" Wurzel zur Basis 2
13 REAL wurzel(const REAL& x) {return limit(wurzel_approx,x);}
```

Listing 1: iRRAM-Program am Beispiel des Heron-Verfahrens, um Wurzeln zu berechnen

In Zeile 5 werden die beiden Variable a und b initialisiert und man kann sehen, dass sie Instanzen der Klasse `REAL` sind. Die Klasse `REAL` wird in Kapitel 3.2 kurz vorgestellt. Zeile 7 implementiert den Algorithmus, wobei die Funktion `bound` von iRRAM verwendet wird, um die Genauigkeit der Annäherung zu prüfen. Ist sie nicht genau genug, wird ein weiterer Iterationsschritt durchgeführt. Die Funktion `bound` wird in Kapitel 3.3 vorgestellt.

Mit der Funktion `limit` wird dann in Zeile 13 der Grenzwert der reellen Folge, welche durch die Approximationsfunktion `wurzel_approx` vorgegeben wird, als reelle Zahl berechnet. Die Ausgabe von `wurzel` ist dann $\sqrt{x}$ als reelle Zahl.

`limit` ist definiert als `REAL limit (REAL a(long, const REAL&), const REAL& x);` und erwartet zwei Parameter: Eine reelle Folge, beschrieben durch eine Funktion mit zwei Parametern `a(long, const REAL&)` und eine reelle Zahl x für die der Grenzwert berechnet werden soll. Die Ausgabe ist eine reelle Zahl, welche dem Grenzwert der Folge entspricht.

In Listing 1 kann man sehen, daß es ausreicht, die Klasse `REAL` zu instanziieren, um reelle Zahlen mit beliebiger Genauigkeit zu berechnen. iRRAM stellt sicher, daß die Zahlen in der geforderten Genauigkeit approximiert werden, ohne daß der Programmierer sich explizit darum kümmern muß.

3.2 Die Klasse REAL

Die Klasse `REAL` ist der Kern des iRRAM-Paketes. Es gibt verschiedene Konstruktoren, wobei für die vorgestellte Implementierung folgende verwendet wurden:

```
1 // Gibt die relle Zahl 0 zurueck
2 REAL ();
3 // Gibt die reelle Zahl i zurueck
4 REAL (const int i);
5 // Gibt die reelle Zahl y zurueck
6 REAL (const REAL& y);
```

Listing 2: Konstruktoren der Klasse REAL

Da die Klasse `REAL` die allgemeinen arithmetischen Operatoren, wie $+, -, *, /$ überlädt, muß man in den meisten Fällen beim Definieren der Funktionen für iRRAM diesbezüglich nichts beachten.

iRRAM stellt sicher, daß die Operationen in der geforderten Genauigkeit durchgeführt werden.

3.3 Mehrwertige Funktionen

Da alle Funktionen, die iRRAM berechnen kann, stetig sein müssen [3, Seite 15], aber viele Funktionen der klassischen Arithmetik nicht stetig sind, können diese nicht direkt in iRRAM umgesetzt werden. Dazu müssen die Funktionen in stetige Intervalle aufgeteilt werden, wie beispielsweise die Division, bei der die Division durch 0 ausgenommen werden muß.

Bei der Implementierung der Beispielfunktion `wurzel_approx` (Listing 1) und bei der Implementierung der Trisektionsmethode wurde die Funktion $bound(x, k)$ als Abbruchkriterium für die Approximation verwendet. Die Funktion hat in iRRAM folgende Signatur `bool bound(const REAL& x, const long k)` und prüft, ob die reelle Zahl x „klein genug“ ist und ist definiert als:

$$\texttt{bound(x,k)} \;=\; \begin{cases} T, & |x| < 2^k \\ \left.\begin{matrix} T \\ F \end{matrix}\right\}, & 2^{k-2} \le |x| \le 2^k \\ F, & |x| > 2^{k-2} \end{cases}$$

In Abbildung 1 kann man die Ausgabewerte der Funktion sehen. Sie gibt als Rückgabewerte die booleschen Werte T oder F zurück, wobei man sehen kann, daß die Funktion für $2^{k-2} \le |x_0| \le 2^k$ beide Werte annimmt, also eine „mehrwertige Funktion“ ist. Ob die Funktion in diesem Bereich T oder F liefert, scheint nicht deterministisch zu sein, da die Wahl des Ausgabewertes in der Funktion gekapselt ist und in jedem Iterationsschritt unvorhersehbar erfolgt.

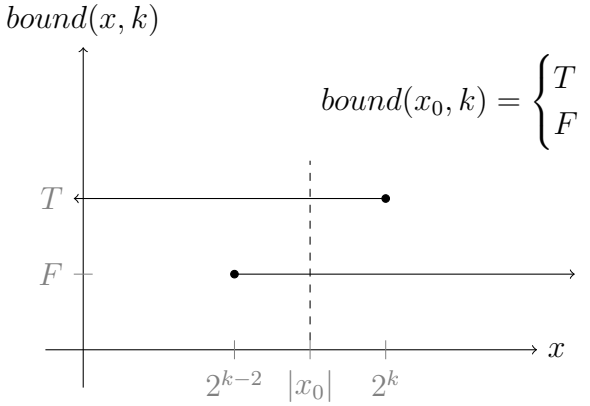

Abbildung 1: Die mehrwertige Funktion $bound(x, k)$

Da $bound(x, k)$ eine mehrwertige Funktion ist, ist auch die Funktion `wurzel_approx` eine mehrwertige Funktion:

- Wenn $|a-b| > 2^p$ ist, wird auf jeden Fall weiter iteriert (weil `!bound(a-b,p)==TRUE`)
- Sobald $|a-b| < 2^{p-2}$ wird auf keinen Fall mehr weiter iteriert (weil `!bound(a-b,p)==FALSE`)

- Für $|a-b| \in [2^{p-2}, 2^p]$ kann die Funktion `wurzel_approx` mehrere Werte annehmen, da die Iterationsschleife (wegen `!bound(a-b,p)==TRUE` oder `FALSE`) in unterschiedlicher Tiefe abbrechen kann

3.4 Der Datentyp LAZY_BOOLEAN

Da in iRRAM mit Intervallannäherungen an reelle Zahlen gearbeitet wird, kann es zum Beispiel beim Größenvergleich zwischen zwei reellen Zahlen zu unentscheidbaren Situationen kommen, wenn die beiden Intervalle sich überlagern, wie in Abbildung 2 skizziert.

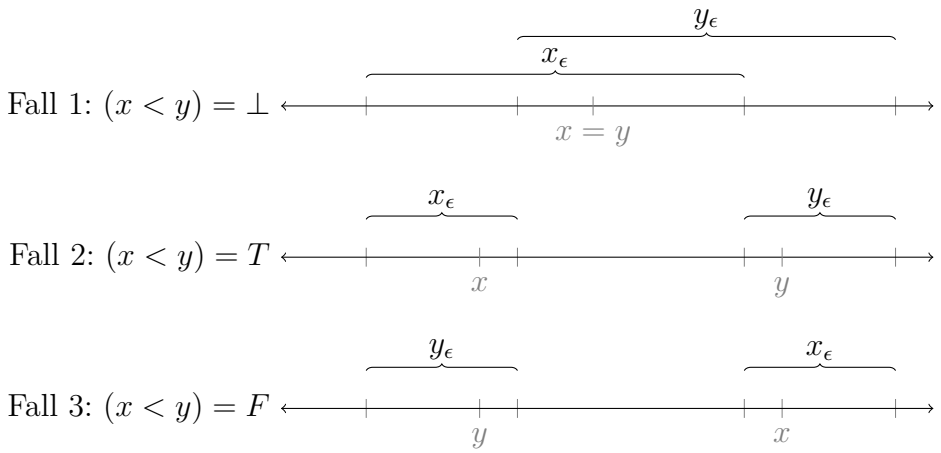

Abbildung 2: Veranschaulichung der verschiedenen Ausgabewerte der Funktion $x < y$, wobei $x, y \in \mathbb{R}$ und x_ϵ, y_ϵ die von iRRAM verwendeten Aprroximationsintervalle darstellen

Solange die Intervalle x_ϵ und y_ϵ disjunkt sind (Fälle 2 und 3 in Abbildung 2), ist der Wert von $x < y$ als T oder F definiert. Für Fall 1 sind die beiden Intervalle nicht disjunkt und daher kann $x = y$ gelten, da x und y reelle Zahlen sind. In diesem Fall ist der Funktionswert $\bot$. Der Versuch, zu entscheiden, welche Zahl größer ist, würde in einer Endlosschleife resultieren.

`x||y`	T	F	$\bot$
T	T	T	T
F	T	F	$\bot$
$\bot$	T	$\bot$	$\bot$

`x&&y`	T	F	$\bot$
T	T	F	$\bot$
F	F	F	F
$\bot$	$\bot$	F	$\bot$

`!x`	
T	F
F	T
$\bot$	$\bot$

Abbildung 3: Die Wahrheitstabellen der Operatoren &&, || und ! des Datentyps *LAZY_BOOLEAN*

Da dabei nicht klar ist, ob und wann eine entsprechende Entscheidung treffbar ist, bietet iRRAM die Möglichkeit mit dem Wert $\bot$ weiterzurechnen. Dazu wird der Datentyp

`LAZY_BOOLEAN` verwendet, welcher die drei Werte T, F und $\perp$ annehmen kann und der die logischen Operatoren `&&`, `||` und `!` erweitert, so daß diese ebenfalls mit dem undefinierten Wahrheitswert umgehen können. Die Wahrheitstabellen dieser Operatoren sind in Abbildung 3 aufgelistet.

Als Beispiel, unter welchen Bedingungen Funktionen den undefinierten Wahrheitswert ausgeben, zeigt Abbildung 4 die Funktionen $<$ und $bool(x)$ mit den Wahrheitswerten für die verschiedenen Fälle.

$$\texttt{(x<y)} = \begin{cases} T, & x < y \\ F, & x > y \\ \perp, & x = y \end{cases}$$

x	bool(x)
T	T
F	F
$\perp$	undefined

Abbildung 4: Wahrheitswerte der Funktionen $<$ und $bool(x)$

Es gibt zwei Möglichkeiten, mit dem Wert eines `LAZY_BOOLEAN` weiterzuarbeiten: Er kann in `bool` umgewandelt werden, falls der Wert T oder F ist, wobei für den Wert $\perp$ die Umwandlung, in einer Endlosschleife resultiert oder es kann die Funktion `choose` verwendet werden, welche im nächsten Kapitel vorgestellt wird. [2, Vgl.]

3.5 Die Funktion choose

Um den Datentyp `LAZY_BOOLEAN` auszuwerten, ohne in Gefahr zu laufen, in eine Endlosschleife zu geraten, steht die Funktion `choose` zur Verfügung:

```
1 int choose ( const LAZY_BOOLEAN& x1= false,
2              const LAZY_BOOLEAN& x2= false,
3              const LAZY_BOOLEAN& x3= false,
4              ... ) ;
```

Listing 3: Signatur der Funktion choose

Es können bis zu 6 Parameter $x1, x2, x3...$ angegeben werden, wobei die Funktion den Index eines Wertes zurückliefert, welcher T ist. Dazu muß mindestens einer der Werte T sein, wobei der Index bei 1 beginnt und die anderen Werte T, F oder auch $\perp$ sein dürfen (ohne dass `choose` in eine Endlosschleife gerät). Falls mehrere Werte T sind, wird der Index eines der Werte zurückgeliefert, welcher T ist. Es ist allerdings vorher nicht klar, welcher Wert (mit T) ausgewählt wird, `choose` ist also eine mehrwertige Funktion.

Sind alle Werte F, liefert die Funktion 0 zurück.

Falls mindestens ein Argument undefiniert ($\perp$) ist und die übrigen F sind, resultiert der Aufruf von `choose` in einer Endlosschleife. (Dieser Fall sollte daher nach Möglichkeit durch geschickte Auswahl der Parameter verhindert werden.)

4 Nullstellensuche mit der Trisektionsmethode in iRRAM

Das Programm kann für Funktionen $f : [a, b] \to \mathbb{R}$ mit $f(a) * f(b) < 0$ die Nullstelle finden und mit beliebiger Genauigkeit nach Satz 2.6.1 berechnen. Wenn die Anzahl der Nullstellen in dem Intervall bekannt ist, kann das Programm diese Nullstellen ebenfalls finden und beliebig genau berechnen.

4.1 Beschreibung des Programms

Die Funktion `null_points(FUNCTION<REAL,REAL> f, int n, REAL l, REAL r)` bekommt als ersten Parameter ein Funktions-Objekt f, als zweiten Parameter n die Anzahl der Nullstellen und die beiden Intervallgrenzen l und r. Die Funktion teilt das Intervall $[l, r]$ in n Intervalle, wobei jedes dieser Intervalle einen Vorzeichenwechsel für f (und damit genau eine Nullstelle) beinhaltet.

```
214   // vzw: used to count the intervals with different sign of function values at boundary
215   int vzw=0;
216   if (vl!=vr) vzw++;
217
218   // number of null points which are not yet found
219   int numNullPoints = n;
220   while (vzw<n)
221   {
222     intvl z=q.front();
223     q.pop();
224
225     // compute m
226     REAL m = Find_Non_null_point(f, numNullPoints, z.l, z.r);
227     bool vm=(f(m)>0);
228     q.push(intvl(z.l,m,z.vl,vm));
229     q.push(intvl(m,z.r,vm,z.vr));
230     // count the change of signs ...
231     if (z.vl == z.vr && z.vl != vm)
232     {
233       vzw+=2;
234       numNullPoints -= 2;
235     }
236   }
```

Docs/nullstellensuche.cc

In Variable `vzw` (Zeile 215) wird die Anzahl der gefundenen Vorzeichenwechsel mitgezählt und in Schleife (Zeile 220) als Abbruchkriterium verwendet.

Bei jedem Durchlauf wird das oberste Intervall z mit den Grenzen $z.l$ und $z.r$ aus der Warteschlange `q` entnommen. Dann wird mit der Funktion `Find_Non_null_point` ein Punkt m in diesem Intervall gesucht, der sicher keine Nullstelle ist ($f(m)! = 0$). Nun kann das Intervall in m geteilt werden und es ergeben sich die beiden neuen Intervalle $[l, m]$ und $[m, r]$. Diese beiden neuen Intervalle werden nun in die FIFO-Warteschlange eingefügt und weiter unterteilt. Bei jeder dieser Unterteilungen wird geprüft (Zeile 231), ob ein Vorzeichenwechsel vorliegt und gegebenenfalls die Anzahl der Vorzeichenwechsel `vzw` erhöht.

Sobald eine passende Unterteilung in Intervalle gefunden wurde, also jedes Intervall sicher genau eine Nullstelle enthält, wird auf den Intervallen die Funktion `trisection` aufgerufen, um die Nullstelle jeweils genau zu berechnen (Zeile 245). Die Funktion `trisection` verwendet, analog zu dem Beispiel in Listing 1 `limit` um den reellen Grenzwert der Folge, die durch die Funktion `trisection_approx` vorgegeben wird, zu berechnen.

4.2 Die Funktion Find_Non_null_point

Diese Funktion findet eine Zahl m, welche in dem gegebenen Intervall $[left, right]$ keine Nullstelle der gegebenen Funktion $iFunc$ ist. Die Funktion erwartet als Parameter die Funktion $iFunc$, die Anzahl der Nullstellen der Funktion in dem Intervall $iNumberOfNullPoints$ und die Intervallgrenzen $left$ und $right$. Die Funktion wählt in dem gegebenen Intervall $[left, right]$ gleichmäßig verteilt die Punkte $m_1 ... m_{iNumberOfNullPoints+1}$ aus und berechnet die zugehörigen absoluten Funktionswerte $|f(m_1)|...|f(m_{iNumberOfNullPoints+1})|$ und speichert diese (in dem Vektor `func_values`).

```
167 // Find an arbitrary x in (left, right), whose y value is not 0.
168 // Function values at n+1 positions (x) are calculated. At least one of them are not null.
169 // The absolute values of n+1 function values are saved in an array. Use function choose
        to check if they are
170 // greater than zero (not undefined). To do this, loop the array in the reverse order, use
         the k-th element
171 // and the 0 to k-1 elements, which are or-ed, as the two parameters of the function
        choose.
172 REAL Find_Non_null_point(FUNC iFunc, int iNumberOfNullPoints, REAL left, REAL right)
173 {
174   REAL diff = right - left;
175   std::vector<REAL> func_values;
176   std::vector<REAL> x_values;
177   for(int i=1; i<=iNumberOfNullPoints+1; ++i)
178   {
179     REAL x = left + i * diff / (iNumberOfNullPoints + 2);
180     REAL abs_value = abs(iFunc(x));
181     func_values.push_back(abs_value);
182     x_values.push_back(x);
183   }
184   std::vector<LAZY_BOOLEAN> tests(func_values.size());
185   for(unsigned int i=0; i < func_values.size(); ++i)
186   {
187     tests[i] = func_values[i] > 0;
188   }
189
190   std::vector<LAZY_BOOLEAN> new_tests(func_values.size());
191   new_tests[0] = tests[0];
192   for(unsigned int i=1; i<new_tests.size(); ++i)
193     new_tests[i] = new_tests[i-1] || tests[i];
194   for(unsigned int i=tests.size()-1; i>0; --i)
195   {
196    if (choose(new_tests[i-1], tests[i])==2)
197    return x_values[i];
198   }
199   return x_values[0];
200 }
```

Docs/nullstellensuche.cc

Dann berechnet die Funktion für jeden dieser Funktionswerte, ob dieser größer als 0 ist, wobei diese Ergebnisse als `LAZY_BOOLEAN` ebenfalls in einem Vektor gespeichert werden (in `tests`). Da die Funktion `choose` nur aus 6 Parametern wählen kann, werden in den Zeilen 192 - 198 die Werte so ausgewertet, dass das Programm nicht in eine Endlosschleife geraten kann (was passieren würde, wenn in einer Teilauswahl kein Wert mit T verfügbar wäre). Da ein Punkt mehr ausgewählt wurde, als die Funktion in dem Intervall Nullpunkte hat, ist sicher, dass mindestens einer der zuvor berechneten Funktionswerte keine Nullstelle ist, also gilt: $\exists i.\ |f(m_i)| > 0 = T$.

4.3 Die Funktion trisection_approx

Die Funktion erwartet als Parameter zwei geschachtelte „pairs“, wobei das erste die gewünschte Präzision p und die Funktion $func$ enthält und die zweite die beiden Intervallgrenzen a und d (Zeilen 111 - 114).

```
107 // in order to use the iRRAM limit operator for FUNCTION<int,REAL>
108 // we formulate with a "trisection_parameter" as argument
109 REAL trisection_approx(const trisection_parameter& p_f_l_r)
110 {
111   int  p           = p_f_l_r.first.first;
112   FUNCTION<REAL,REAL> func = p_f_l_r.first.second;
113   REAL a           = p_f_l_r.second.first;
114   REAL d           = p_f_l_r.second.second;
115
116   REAL result = d;
117   while( !bound(d-a,p-1) )
118   {
119     REAL new_a = calc_a(a, d);
120     REAL new_d = calc_d(a, d);
121
122     REAL f_left_left = func(a);
123     REAL f_left_right = func(new_d);
124     REAL f_right_left = func(new_a);
125     REAL f_right_right = func(d);
126
127     REAL f_left = f_left_left * f_left_right;
128     REAL f_right = f_right_left * f_right_right;
129
130     int test=choose(f_left<0,f_right<0);
131     if(test==1) // f_left<0: [a, new_d] is the next interval to be concerned
132       {
133         d = new_d;
134         result = d;
135       }
136     else        // f_right<0: [new_a, d] is the next interval to be concerned
137       {
138         a = new_a;
139         result = a;
140       }
141   }
142   return result;
143 }
```

Docs/nullstellensuche.cc

In Zeile 117 wird geprüft, ob die Genauigkeit ausreichend ist, oder weiter iteriert werden soll, wobei jeder Iterationsschritt wie in dem Beweis von Satz 2.6.1 beschrieben abläuft.

Da in dem gegebenen Intervall immer ein Vorzeichenwechsel erfolgt, ist beim Aufruf der Funktion `choose` in Zeile 130 auf jeden Fall einer der beiden Werte T. Daher wird die Funktion `choose` an dieser Stelle nie in eine Endlosschleife geraten.

Sobald die geforderte Genauigkeit erreicht wurde, liefert die Funktion die entsprechend angenäherte Nullstelle zurück.

Die Schwierigkeit bei der Programmierung dieser Funktion lag darin, die von iRRAM gegebene Beschränkung auf einen Funktionsparameter durch die Verwendung von `pair` zu umgehen.

4.4 Ergebnisse

Das Programm wurde mit 12 verschiedenen mathematischen Funktionen getestet. Die Funktionen sind im Programmcode mit `Test 1` bis `Test 12` kommentiert. Wenn man das Programm startet, kann man durch Eingabe einer Zahl von 1 bis 12 bestimmen, welche der Testfunktionen berechnet werden soll. Danach fordert das Programm zur Eingabe der gewünschten Präzision auf, welche in Dezimalstellen eingegeben wird. Danach startet das Programm mit der Berechnung der gewünschten Funktion in der vorgegebenen Präzision.

Die Funktion `Test 3` erfüllt nicht die notwendige Bedingung $f(a) * f(b) < 0$ und kann daher nicht durch den Algorithmus berechnet werden. Desweiteren stellte sich heraus, dass das Programm für die Funktion `Test 5` kein Ergebnis berechnen konnte, da diese Funktion zu flach ist. Der Speicherbedarf für die Approximation rund um die Nullstelle steigt exorbitant und machte diese auf unseren Computern unmöglich.

In Tabelle 1 sind die Laufzeiten für die Berechnung der Nullstellen in Abhängigkeit von der Genauigkeit aufgelistet. Man kann sehen, daß ab Funktion `Test 10` eine Berechnung der Nullstellen nicht mehr unter 5 Minuten, bzw. ohne Speicherplatzmangel möglich war.

Bei der Analyse wurde herausgefunden, daß die Funktion `Find_Non_null_point` sehr langsam ist, besonders wenn eine Funktion viele Nullstellen hat. In der ursprünglichen Implementierung werden die Punkte auf der rechten Seite des Intervalls bevorzugt (die Schleife fängt von hinten an). Dies bedeutet, daß die Intervalle nicht gleichmäßig verteilt sind. Am rechten Rand sind die Intervalle dichter als am linken. Dies mag für wenige spezielle Funktionen von Vorteil sein, ist aber für die meisten Funktionen ineffizient.

Die Idee war nun, den Algorithmus so zu modifizieren, daß die Punkte in der Mitte zuerst bewertet werden. Eine relativ gleichmäßige Unterteilung der Intervalle macht es wahrscheinlicher, daß die n richtigen Intervalle gefunden werden, was zu einer kürzeren Laufzeit führen sollte.

Der Algorithmus zur Unterteilung der Intervalle wurde wie folgt abgeändert:

```
177   // begin new algorithm
178   std::queue<intvl> q;
179   intvl first(left, right, true, true);
180   q.push(first);
181   REAL l = left;
182   REAL r = right;
183   while(x_values.size() < iNumberOfNullPoints+2)
184   {
185     intvl item = q.front();
186     q.pop();
```

```
187     // the interval is always cut in half
188     REAL x = (item.l + item.r) / 2;
189     REAL y = abs(iFunc(x));
190     func_values.push_back(y);
191     x_values.push_back(x);
192
193     q.push(intvl(item.l, x, true, true));
194     q.push(intvl(x, item.r, true, true));
195   }
196   // end new algorithm
```

Docs/nullstellensuche–final.cc

In Zeile 188 kann man erkennen, daß die Intervalle nun jeweils in der Mitte geteilt werden. Mit dieser Änderung konnte die Performance für Funktionen mit vielen Nullstellen entscheidend verbessert werden, wie aus Tabelle 2 ersichtlich ist. Die Laufzeiten der Funktionen `Test 1` bis `Test 9` haben sich nicht oder nicht wesentlich geändert. Die Laufzeiten der Funktionen `Test 10` bis `Test 12` haben sich jedoch sehr stark verbessert und führten teilweise (für die Funktionen `Test 11` und `Test 12`) dazu, daß die Funktionen überhaupt erst auf dem Testrechner berechenbar waren.

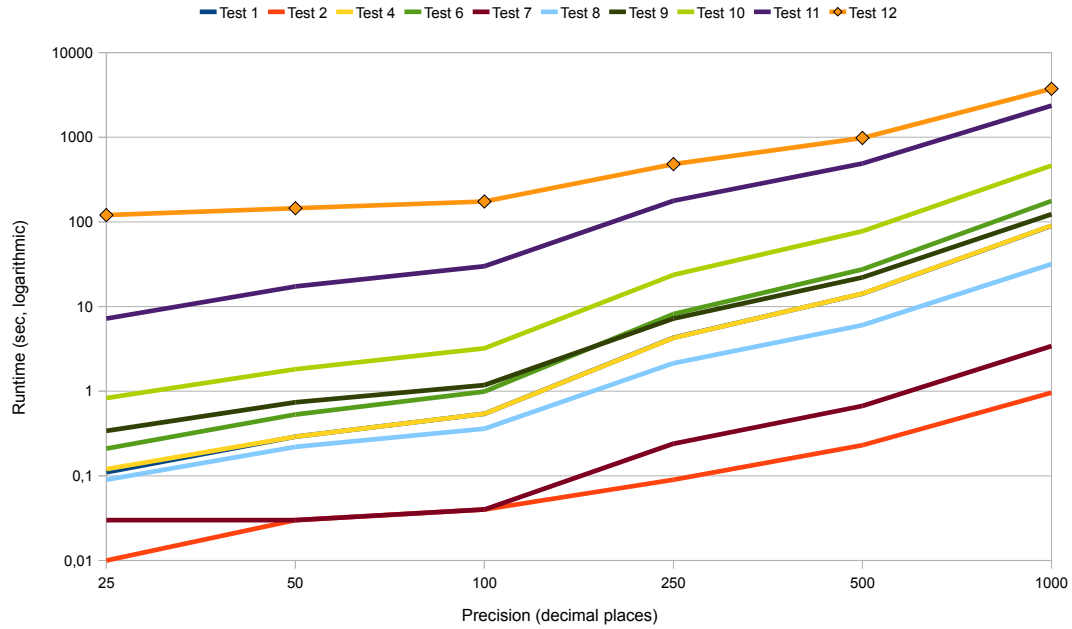

Abbildung 5: Laufzeiten des optimierten Algorithmus

Folgender Punkt könnten zu einer Performancesteigerung führen, wurde aber nicht innerhalb dieser Arbeit implementiert:

- Die Funktion `choose` liefert den Index eines Parameters zurück, der den Wert T annimmt. Wenn aber mehrere Parameter den Wert T haben, kann der Index irgendeines Parameters zurückgegeben werden. Das ist der Grund, warum ein komplizierter Algorithmus verwendet werden muß. Wenn aber die Funktion `choose` immer den ersten

Parameterindex zurückgäbe, der T ist, könnte die Performanz nochmals verbessert werden.

Literatur

[1] V. Brattka, P. Hertling, and K. Weihrauch. A tutorial on computable analysis. *New Computational Paradigms*, pages 425–491, 2008.

[2] N. Mueller. *Exact Real Arithmetic.*

[3] N. Mueller. The irram: Exact arithmetic in c++. *Computability and Complexity in Analysis*, pages 222–252, 2001.

[4] M. Neeb. Beispiele berechenbarer reeller zahlen und abgeschlossenheitseigenschaften, 2010. [Online; Stand 11. Mar 2010].

[5] K. Weihrauch. *Computable analysis: an introduction.* Springer Verlag, 2000.

[6] Wikipedia. Turingmaschine — wikipedia, die freie enzyklopaedie, 2011. [Online; Stand 11. Juni 2011].

[7] Wikipedia. Zwischenwertsatz — wikipedia, die freie enzyklopaedie, 2011. [Online; Stand 14. Juni 2011].

A Tabellen der gemessenen Laufzeiten

Testfunktion	**Nullstellen**	**Genauigkeit**					
		25	50	100	250	500	1000
TF 1	1	0.11	0.29	0.53	4.25	14.1	89.82
TF 2	1	0.01	0.03	0.04	0.09	0.23	0.96
TF 4	1	0.12	0.29	0.54	4.26	14.25	90.32
TF 6	2	0.21	0.53	0.99	8.16	27.51	176.85
TF 7	2	0.03	0.03	0.04	0.24	0.67	3.41
TF 8	5	0.09	0.22	0.36	2.14	6.04	31.73
TF 9	10	0.68	1.1	1.62	8.29	24.33	127.62
TF 10	20	306.95	–	–	–	–	–
TF 11	40	–	–	–	–	–	–
TF 12	40	–	–	–	–	–	–

Tabelle 1: Laufzeiten des nicht optimierten Algorithmus, (–) bedeutet, dass das Ergebnis nicht unter 600s, bzw ohne Speicherplatzmangel berechenbar war

Testfunktion	**Nullstellen**	**Genauigkeit**					
		25	50	100	250	500	1000
TF 1	1	0.11	0.29	0.53	4.25	14.1	89.82
TF 2	1	0.01	0.03	0.04	0.09	0.23	0.96
TF 4	1	0.12	0.29	0.54	4.26	14.25	90.32
TF 6	2	0.21	0.53	0.99	8.16	27.51	176.85
TF 7	2	0.03	0.03	0.04	0.24	0.67	3.41
TF 8	5	0.09	0.22	0.36	2.14	6.04	31.73
TF 9	10	0.34	0.74	1.18	7.21	22.09	123.12
TF 10	20	0.83	1.82	3.21	23.63	77.67	463.07
TF 11	40	7.21	17.28	29.95	177.44	490.07	2364.05
TF 12	40	120.32	145.08	174.11	481.21	980.46	3738.51

Tabelle 2: Tabelle mit den Laufzeiten des optimierten Algorithmus

B Programmcode

```
#include "iRRAM.h"
#include <queue>
#include <vector>
#include <utility>
#include <time.h>
using namespace iRRAM;

//*******************  Test functions  ***********************************/
// Test 1: cos3x has exact one null point in interval [0, 1], and x = pi/6.
REAL cos3x(const REAL& iX)
{
  return cos(3 * iX);
}
// Test 2: line has one null point at 1/3.
REAL line(const REAL& iX)
{
  return REAL(3) * iX - REAL(1);
}
// Test 3: sin3x has one null point on boundary. Error message is shown.
REAL sin3x(const REAL& iX)
{
  return sin(3 * iX);
}
// Test 4: flat is a ver flat function and has the same null point as cons3x.
REAL flat(const REAL& iX)
{
  return cos(3 * iX)/1000000000;
}
// Test 5: exp_flat is an exponential (!) sloping flat function with the null point at
    1/3.
REAL exp_flat(const REAL& iX)
{
  return (iX-REAL(1)/3)*exp(-1/(iX-REAL(1)/3)/(iX-REAL(1)/3));
}
// Test 6: sinus has 2 null points: 0 and pi
REAL sinx(const REAL& iX)
{
  return sin(iX);
}
// Test 7: parable has 2 null points: -1 and 1
REAL parable(const REAL& iX)
{
  return iX* iX-1;
}
// Test 8: x5 has 5 null points: 1/3,1/6,1/7,1/9,1/11
REAL x5(const REAL& iX)
{
  return (iX-REAL(1)/3)*(iX-REAL(1)/6)*(iX-REAL(1)/7)*(iX-REAL(1)/9)*(iX-REAL(1)/11);
}
// Test 9: x10 has 10 null points: 1/3,1/6,1/7,1/9,1/11,6,7,8,9,10
REAL x10(const REAL& iX)
{
  return (iX-REAL(1)/3)*(iX-REAL(1)/6)*(iX-REAL(1)/7)*(iX-REAL(1)/9)*(iX-REAL(1)/11)*(iX
      -6)*(iX-7)*(iX-8)*(iX-9)*(iX-10);
}
// Test 10: x20 has 20 null points: -10,-9,-8,...,-2,-1,1,2,...,8,9,10
REAL x20(const REAL& iX)
{
  REAL Z=1;
  for (int i=-20;i<= -1;i++) Z=Z*(iX - REAL(i) );
  for (int i=  1;i<= 20;i++) Z=Z*(iX - REAL(i) );
  return Z;
```

```
61 }
62 // Test 11: x40 has 40 null points: -1, -1/2, -1/3, ..., -1/20, 1/20, 1/19, ..., 1
63 REAL x40(const REAL& iX)
64 {
65   REAL Z=1;
66   for (int i=-20;i<= -1;i++) Z=Z*(iX - 1/REAL(i) );
67   for (int i=  1;i<= 20;i++) Z=Z*(iX - 1/REAL(i) );
68   return Z;
69 }
70 // Test 12: x40a has 40 null points: -1, -1/2^3, -1/3^3, ..., -1/20^3, 1/20^3, 1/19^3,
       ...1/2^3, 1
71 REAL x40a(const REAL& iX)
72 {
73   REAL Z=1;
74   for (int i=-20;i<= -1;i++) Z=Z*(iX - 1/REAL(i)/REAL(i)/REAL(i));
75   for (int i=  1;i<= 20;i++) Z=Z*(iX - 1/REAL(i)/REAL(i)/REAL(i) );
76   return Z;
77 }
78 /*****************************************************************/
79
80 // new left boundary is calculated.
81 REAL calc_a(REAL old_a, REAL old_d)
82 {
83   return old_a + (old_d - old_a) / 3;
84 }
85
86 // new right boundary is calculated.
87 REAL calc_d(REAL old_a, REAL old_d)
88 {
89   return old_a + 2 * (old_d - old_a) / 3;
90 }
91
92 // we need four parameters to do the trisection:
93 // 1) the intended precision
94 // 2) the function under consideration
95 // 3) a left border
96 // 4) a right border
97
98 // first we define a corresponding type.
99 // the standard libaries only allow for pairs, so we have to nest the pairing
100 typedef FUNCTION<REAL,REAL> FUNC;
101
102 typedef std::pair< std::pair<int, FUNCTION<REAL,REAL> >, std::pair<REAL,REAL> >
       trisection_parameter;
103
104 // the following routine returns an approximation to one of the roots
105 // of the function under consideration
106
107 // in order to use the iRRAM limit operator for FUNCTION<int,REAL>
108 // we formulate with a "trisection_parameter" as argument
109 REAL trisection_approx(const trisection_parameter& p_f_l_r)
110 {
111   int  p           = p_f_l_r.first.first;
112   FUNCTION<REAL,REAL> func = p_f_l_r.first.second;
113   REAL a           = p_f_l_r.second.first;
114   REAL d           = p_f_l_r.second.second;
115
116   REAL result = d;
117   while( !bound(d-a,p-1) )
118   {
119     REAL new_a = calc_a(a, d);
120     REAL new_d = calc_d(a, d);
121
122     REAL f_left_left = func(a);
123     REAL f_left_right = func(new_d);
```

```
124     REAL f_right_left = func(new_a);
125     REAL f_right_right = func(d);
126
127     REAL f_left = f_left_left * f_left_right;
128     REAL f_right = f_right_left * f_right_right;
129
130     int test=choose(f_left<0,f_right<0);
131     if(test==1) // f_left<0: [a, new_d] is the next interval to be concerned
132       {
133         d = new_d;
134         result = d;
135       }
136     else        // f_right<0: [new_a, d] is the next interval to be concerned
137       {
138         a = new_a;
139         result = a;
140       }
141   }
142   return result;
143 }
144
145 // now we add the limit process and do the necessary conversions between the types
146 REAL trisection(FUNCTION<REAL,REAL> f, const REAL& left,  const REAL& right){
147   FUNCTION<int,REAL> fp=bind_second(
148         bind_second(
149           FUNCTION<trisection_parameter,REAL>(trisection_approx),
150           std::pair<REAL,REAL>(left,right)),
151         f);
152   return limit(fp);
153 }
154
155 // Interval structure to save the boundaries and signs of function values on the
      boundaries
156 struct intvl
157 {
158   REAL l;
159   REAL r;
160   bool vl;
161   bool vr;
162
163   intvl(REAL il, REAL ir, bool ivl, bool ivr) : l(il), r(ir), vl(ivl), vr(ivr)
164   {}
165 } ;
166
167 // Find an arbitrary x in (left, right), whose y value is not 0.
168 // Function values at n+1 positions (x) are calculated. At least one of them are not null.
169 // The absolute values of n+1 function values are saved in an array. Use function choose
      to check if they are
170 // greater than zero (not undefined). To do this, loop the array in the reverse order, use
       the k-th element
171 // and the 0 to k-1 elements, which are or-ed, as the two parameters of the function
      choose.
172 REAL Find_Non_null_point(FUNC iFunc, int iNumberOfNullPoints, REAL left, REAL right)
173 {
174   REAL diff = right - left;
175   std::vector<REAL> func_values;
176   std::vector<REAL> x_values;
177   for(int i=1; i<=iNumberOfNullPoints+1; ++i)
178   {
179     REAL x = left + i * diff / (iNumberOfNullPoints + 2);
180     REAL abs_value = abs(iFunc(x));
181     func_values.push_back(abs_value);
182     x_values.push_back(x);
183   }
184   std::vector<LAZY_BOOLEAN> tests(func_values.size());
```

```
185   for(unsigned int i=0; i < func_values.size(); ++i)
186   {
187     tests[i] = func_values[i] > 0;
188   }
189
190   std::vector<LAZY_BOOLEAN> new_tests(func_values.size());
191   new_tests[0] = tests[0];
192   for(unsigned int i=1; i<new_tests.size(); ++i)
193     new_tests[i] = new_tests[i-1] || tests[i];
194   for(unsigned int i=tests.size()-1; i>0; --i)
195   {
196    if (choose(new_tests[i-1], tests[i])==2)
197    return x_values[i];
198   }
199   return x_values[0];
200 }
201
202 // Find the n null points of the function in the interval (l, r).
203 // We search n sub intervals, which have only one null point.
204 // Then we use the algorithm 'trisection' to calculate the x value.
205 void null_points(FUNCTION<REAL,REAL> f, int n, REAL l, REAL r, int prec)
206 {
207   // use queue to store the interval information
208   using std::queue;
209   queue<intvl> q;
210   bool vl=f(l)>0,vr=f(r)>0;
211   // first item in queue: the initial interval
212   q.push(intvl(l,r,vl,vr));
213
214   // vzw: used to count the intervals with different sign of function values at boundary
215   int vzw=0;
216   if (vl!=vr) vzw++;
217
218   // number of null points which are not yet found
219   int numNullPoints = n;
220   while (vzw<n)
221   {
222     intvl z=q.front();
223     q.pop();
224
225     // compute m
226     REAL m = Find_Non_null_point(f, numNullPoints, z.l, z.r);
227     bool vm=(f(m)>0);
228     q.push(intvl(z.l,m,z.vl,vm));
229     q.push(intvl(m,z.r,vm,z.vr));
230     // count the change of signs ...
231     if (z.vl == z.vr && z.vl != vm)
232     {
233       vzw+=2;
234       numNullPoints -= 2;
235     }
236   }
237   // use trisection to calculate null points
238   int deci_places = prec;
239   while(q.empty()==false)
240   {
241     intvl item = q.front();
242     q.pop();
243     if(item.vl == item.vr)
244       continue;
245     REAL result = trisection( f, item.l, item.r);
246     cout << "result: ";
247     cout << setRwidth(deci_places+8);
248     cout << result << "\n" ;
249   }
```

```
250 }
251
252 void compute(){
253   int test, prec;
254   clock_t start, end;
255   cout <<
256   "Function selection: 1#cos(3x); 2#line; 3#sin(3x); 4#flach; 5#exp-flach;\n"
257   <<  " 6#sin(x); 7#parable(x); 8#x5; 9#x10; 10#x20; 11#x40; 12#x40a\n";
258   cin >> test;
259   cout << "Praezision: \n";
260   cin >> prec;
261   FUNCTION<REAL,REAL> f;
262   unsigned int numOfNullPoints;
263   REAL left_boundary, right_boundary;
264   switch (test) {
265       case  1: f= FUNCTION<REAL,REAL>(cos3x)    ; numOfNullPoints=1;  left_boundary=0;
              right_boundary=1;   break;
266       case  2: f= FUNCTION<REAL,REAL>(line)     ; numOfNullPoints=1;  left_boundary=0;
              right_boundary=1;   break;
267       case  3: f= FUNCTION<REAL,REAL>(sin3x)    ; numOfNullPoints=1;  left_boundary=0;
              right_boundary=1;   break;
268       case  4: f= FUNCTION<REAL,REAL>(flat)     ; numOfNullPoints=1;  left_boundary=0;
              right_boundary=1;   break;
269       case  5: f= FUNCTION<REAL,REAL>(exp_flat) ; numOfNullPoints=1;  left_boundary=0;
              right_boundary=1;   break;
270       case  6: f= FUNCTION<REAL,REAL>(sin)      ; numOfNullPoints=2;  left_boundary=-1;
              right_boundary=4;   break;
271       case  7: f= FUNCTION<REAL,REAL>(parable)  ; numOfNullPoints=2;  left_boundary=-3;
              right_boundary=5;   break;
272       case  8: f= FUNCTION<REAL,REAL>(x5)       ; numOfNullPoints=5;  left_boundary=-0.9;
              right_boundary=5.1; break;
273       case  9: f= FUNCTION<REAL,REAL>(x10)      ; numOfNullPoints=10; left_boundary=-2;
              right_boundary=50;  break;
274       case 10: f= FUNCTION<REAL,REAL>(x20)      ; numOfNullPoints=20; left_boundary=-20;
              right_boundary=20;  break;
275       case 11: f= FUNCTION<REAL,REAL>(x40)      ; numOfNullPoints=40; left_boundary=-2;
              right_boundary=2;   break;
276       case 12: f= FUNCTION<REAL,REAL>(x40a)     ; numOfNullPoints=40; left_boundary=-2;
              right_boundary=2;   break;
277       default: cout << "Wrong input!\n"; return;
278     }
279     start = clock();
280     null_points(f, numOfNullPoints, left_boundary, right_boundary, prec);
281     end = clock();
282     cout << "Zeit (s):" <<(float)(end - start)/(float)(CLOCKS_PER_SEC) << "\n\n\n";
283 }
```

Docs/nullstellensuche.cc