Implementierung nichtlinearer Taylormodelle

Forschungspraktikum

Universität Trier
FB IV - Informatikwissenschaften
Professur Arithmetische Algorithmen

Betreuer: apl. Prof. Dr. Norbert Müller

Vorgelegt am xx.xx.xxxx von:

Benedikt Lüken-Winkels
Baltzstraße 6
54296 Trier
s4beluek@uni-trier.de
Matr.-Nr. 1138844

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Das Berechnen reeler Zahlen und Funktionen am Computer bringt einige Probleme mit sich, da diese nicht ohne Weiteres durch eine endliche Menge von Informationen dargestellt werden können (TTE). Mit der Verwendung von Intervallalgorithmen kann der Rechenfehler, beziehungsweise die Ungenauigkeit, welche sich durch die Approximation der Zahlen ergibt, berücksichtigt werden. Das Ziel dabei ist, den Fehler möglichst klein zu halten und dabei so wenig Geschwindigkeit, wie möglich bei der Berechnung zu verlieren. Mit Taylormodellen werden multivariate reelle Funktionen $f :\subseteq \mathbb{R}^k \to \mathbb{R}$ durch Polynome mit Intervallkoeffizienen zur lokalen Fehlerabschätzungen und Fehlersymbolen für Abhängigkeiten zwischen den einzelnen Monomen, dargestellt. Diese Herangehensweise soll das Problem der Überschätzung (des 'blow-ups') der Funktionswerte duch die Fehlerintervalle angehen, was man leicht an einem Beispiel verdeutlichen kann:

Seien $x_0 \in [1, 2], x_1 \in [2, 3]$. Es existiert also die Information, in welchen Wertebereichen x_0 und x_1 jeweils liegen, beziehungsweise wie ungenau die einzelnen Werte bekannt sind. Werden diese nun ohne Berücksichtigung der Abhängigkeiten verrechnet wie hier,

$$\begin{aligned}(x_0 + x_1) - x_0 \to &[1 + 2, 2 + 3] - [1, 2] \\ = &[1, 4]\end{aligned}$$

so entsteht beim Zusammenrechnen die Überschätzung $[1, 4]$, während das tatsächliche Ergebnis $[2, 3]$ einen deutlich kleineren Bereich abdeckt. Zwar ist auch die Überschätzung korrekt, jedoch könnte das Ergebnis genauer errechnet werden. Bei iterierenden Funktionen, wie

$$x_{n+1} = c \cdot x_n \cdot (1 - x_n)$$

welche in dieser Forschungsarbeit häufiger betrachtet wird. Für verschiedene Werte von c und x_0 weist die Funktion sowohl konvergierendes, als auch chaotisches Verhalten auf.

In dieser Forschungsarbeit wird in x_0 ein gewisser Fehler kodiert, um beispielsweise Messungenauigkeiten zu simulieren. Dieser Fehler ergibt in jeder Iteration eine wachsendende Überschätzung, da x_0 nicht als genauer Wert vorliegt. Unkontrolliert wächst dieses Wrapping sehr schnell, sodass nach einigen Iterationen keine genauen Aussagen mehr über ein Ergebnis getroffen werden können. Durch verschiedene Techniken wird daher versucht, die Intervallbreite zu beschränken und klein zu halten, indem neue Variablen (Fehlersymbole) eingeführt werden, welche den Fehler und dessen Ursprung repräsentieren. Je länger die Abhängigkeitsinformationen über diese Fehler bei einer wachsenden Anzahl von Iterationen beibehalten, also die Intervalle nicht wieder eingesetzt werden, desto größer wird das berechnete Polynom

und dessen Grad. Es ergibt sich ein Trade-off zwischen der Menge an behaltener Information und Wartbarkeit des zu berechnenden Polynoms.

Eine Herangehensweise an dieses Problem ist die Verwendung von Taylormodellen ([**?**], [**?**]), welche im Fokus dieser Forschungsarbeit stehen. Die Implementierung im Rahmen dieses Forschungsprojektes geschah in der Programmiersprache *C++*, wobei für die Koeffizienten (und die Intervalle der Fehlersymbole) zum Einen die Rationalen Zahlen `mpq_class` der `GMP`-Bibliothek ([**?**]) und zum Anderen die reellen Zahlen der `iRRAM`-Bibliothek ([**?**]) verwendet wurden. In beiden Fällen wurde das Programm `Tangentspace`[1], einem Programm zum Erstellen linearer Schranken für nichtlinearer Funktionen an einem gegebenen Punkt, derart erweitert, als dass die bei den Berechungen entstehenden Polynome Grade > 1 erreichen können. Um ein Produkt zweier Taylormodelle (zweier Polynome) linear zu halten, wird beim paarweisen Multiplizieren der Monome jeweils ein Fehlersymbol durch sein Intervall ersetzt, wodurch sich das Intervall des Koeffizienten und damit die Ungenauigkeit der Berechnung vergrößert. Wird dies nicht getan, bleiben die Fehlerintervalle zunächst klein, jedoch nicht das Polynom, welches in Grad und Anzahl der Monome wächst. Mit dem im Rahmen dieses Forschungsprojektes entstandenen Programm `hotm` (High Order Taylor Models) soll das bestmögliche Maß zwischen der Reduzierung des Grades, beziehungsweise der Vereinfachung der Polynome und dem Erhalt der Abhängigkeitsinformationen gefunden werden, um dann an verschiedenen Stellen verwendet zu werden.

1.1 Anwendung

Taylormodelle höherer Ordung könnten in `Tangentspace` dazu verwenden werden, um diese in CDCL-artigen (Conflict Driven Clause Learning) SMT Programmen einzupflegen, wo derzeit lineare Taylormodelle verwendet werden. Außerdem können sie als Zahlentyp zur Berechnung reeller Funktionen, wie zum Beispiel Iterationen der Hénon-Abbildung dienen.

Betrachtet man die Fehlerintervalle, beziehungsweise deren Breite als Rauschen, können Taylormodelle dazu genutzt werden, physikalische Vorgänge, die eine gewisse Unsicherheit der Werte haben, darzustellen. Hierzu kann der Fehler durch die Definition des Startwertes als Taylormodell ausgedrückt werden. Um eine sich ändernde Umgebung darzustellen könnte im Falle der Anwendung bei der Logistic Map durch die Konstante in jeder Iteration ein neues, leicht verändertes Fehlersymbol eingeführt werden; oder zu jeder Iteration wird ein bestimmes Fehlerintervall addiert.

1.2 Gliederung

In diesem Forschungsbericht wird der Aufbau und die Funktionsweise der einzelnen Bestandteile der Implementierung der `hotm` neben der Hintergrund für dieselben beschrieben.

[1] http://informatik.uni-trier.de/~mueller/Research/Tangentspace/

2. Taylormodelle

Ein Taylormodell, in [**?**] eingeführt und in [**?**] erweitert, ist ein Polynom $T = \sum_n c_n \lambda^n$, mit k Variablen. λ_i ist ein Fehlersymbol aus $\lambda = (\lambda_1, ..., \lambda_k)$, dem Vektor der Fehlersymbole, wodurch funktionale Abhängigkeiten zwischen Monomen und anderen Taylormodellen hergestellt werden können. Die Koeffizienten des Polynoms sind Intervalle $c_i = [\tilde{c}_i \pm e_i] \subseteq \mathbb{R}$ mit denen der aktuelle Rechenfehler ('approximation error') abgeschätzt wird. Ist ein Wert genau bekann, also ohne Ungenauigkeit oder Fehler, kann dieser als Punktintervall mit $c_i = [\tilde{c}_i \pm 0]$ definiert werden, also mit einem Zentrum ohne Radius.

2.1 Darstellung

Ein Taylormodell besteht in der Implementierung aus dem Kernintervall `kernel`, einer Liste von sortierten von Monomen und einer Liste von Supportintervallen, den Intervallen, der Fehlersymbole. Die Liste der Supportintervalle ist für jedes Taylormodell dieselbe.

2.2 Spezielle Operationen auf Taylormodellen

Quadratisches Sweeping

Um die Länge der Polynome und deren Grad in den Taylormodellen zu kontrollieren können einzelne Fehlersymbole in einem Monom durch ihr Intervall ersetzt werden. So erhöht sich zwar die Breite des Koeffizienten, allerdings wird auch das Verrechnen mit einem anderen, nun gleichen Monom ermöglicht. Dieses *Sweeping* bietet eine natürliche Möglichkeit, den durch die Fehlersymbole eingeführten Fehler klein zu halten, indem versucht wird, bei *quadratischen* Terme nicht ein Fehlersymbol nach dem anderen zu Sweepen, sondern deren Quadrate einzusetzen:

Gegeben sei ein Monom eines Polynoms $... + c \cdot \lambda^3 + ...$ mit $\lambda \in [-1.5, 1.5]$. Nun sieht man, dass zwar λ sowohl positiv, als auch negativ sein kann, jedoch das Quadrat nicht. Daher kann man statt

$$\begin{aligned} &... + c \cdot \lambda^3 + ... \\ \rightarrow &... + c \cdot [-1.5, 1.5] \cdot \lambda^2 + ... \\ \rightarrow &... + c \cdot [-1.5, 1.5] \cdot [-1.5, 1.5] \cdot \lambda + ... \\ \rightarrow &... + c \cdot [-2.25, 2.25] \cdot \lambda + ... \end{aligned}$$

das Quadrat einsetzen

$$\begin{aligned} &... + c \cdot \lambda^3 + ... \\ \rightarrow &... + c \cdot [0, 2.25] \cdot \lambda + ... \end{aligned}$$

und somit das Wachstum des Rechenfehlers verlangsamen.

Polishing

Rechenoperationen auf Intervallen sorgen dafür, dass sich diese Vergrößern. Mit gut gewählten Sweeping-Stategien kann dieser Effekt zwar verlansamt, aber nicht aufgehalten werden. Wie in Kapitel **??** zu sehen ist, kann ab einer bestimmten Intervallbreite keine Aussage mehr über das Ergebnis getroffen werden. Mit *Polishing* [**?**] kann ein zu großes Intervall eines Koeffizienten in zwei Monome mit jeweils einem Punktintervall und einem neuen Fehlersymbol aufgeteilt werden. Das verlängert zwar das Polynom, verhindert jedoch ein zu großes Wachstum der Intervallbreite. Eine Implementierung in `hotm` ist derzeit nicht vorhanden, aber angedacht.

2.3 Sweeping-Strategien

In einer linearen Implementierung der Taylormodelle und deren Polynome, muss Sweeping in jeder Multiplikation geschehen, da sich sonst der Grad des Polynome erhöhte. Folgt man der Idee des *quadratischen Sweepens*, so ergibt sich bei der Multiplikation zweier Polynome und dementsprechend der paarweisen Multiplikation der Monome:

$$(c_1 \cdot \lambda_i) \cdot (c_2 \cdot \lambda_j) = \begin{cases} c_1 c_2 s_j \lambda_i & \text{wenn } i \neq j \text{ und } s_i > s_j \\ c_1 c_2 s_i \lambda_j & \text{wenn } i \neq j \text{ und } s_i \leq s_j \\ c_1 c_2 s_i^2 & \text{wenn } i = j \end{cases} \qquad \lambda_i \in s_i, \lambda_j \in s_j$$

Im dritten Fall, also bei $i = j$, wird das Ergebnis der Multiplikation zum Kernintervall addiert. Das Problem bei dieser Herangehensweise ist, dass ein Fehler, der ins Kernintervall gesweept wurde, nicht mehr mit einem Fehlersymbol (λ) assoziiert werden kann und damit keinerlei Abhängigkeitsinformationen mehr enthält. Zwar tritt dieser Fall nicht so häufig auf, wie die beiden anderen, allerdings bedeutet das auch, dass nicht so häufig quadratisch gesweept wird. Eine nicht-lineare Implementierung erlaubt ein Wachsen der Grade der Polynome und andere Sweeping-Strategien. In `hotm` geschieht das Sweeping eines Taylormodells t per Funktionsaufruf unter Angabe der zu verwendenden Sweeping-Strategie und dem Grad, auf den t reduziert werden soll. Wird beispielsweise als Grad „1“ mitgegeben, ist die Ausgabe höchstens linear, es werden also keine Monome auf den Grad 0 reduziert. Um ein Taylomodell zu evaluieren und auf ein einziges Intervall zu reduzieren, muss der Zielgrad „0“ sein.

Da sich der Grad eines Polynoms nach dem Grad des größten Monoms richtet, muss jedes Monom auf den Zielgrad reduziert werden. Dafür wird über die Fehlersymbole der einzelnen Monome iteriert und eine der Strategien angewandt:

square_only

Das Sweeping wird auf quadratisches Sweeping beschränkt. Für jedes Fehlersymbol mit einem Exponenten > 1 wird versucht, einen möglichst großen, geraden Wert von demselben abzuziehen.
Beispiel sweep_to(1):

$$\begin{aligned} ... + c \cdot \lambda_1^2 \lambda_2^5 \lambda_3^1 + ... \quad & (\lambda_i \in s_i) \\ ... + c s_1^2 \cdot \lambda_2^5 \lambda_3^1 + ... \quad & \lambda_1 \text{ wird quadriert und somit quasi weggekürzt} \\ ... + c s_1^2 s_2^{2^2} \cdot \lambda_2^1 \lambda_3^1 + ... \quad & \lambda_2 \text{ kann nur 2-mal quad. gesweept werden} \end{aligned}$$

Unter Verwendung von `square_only` ist es möglich, dass das Ergebnis, wie im Beispiel, nicht den angegebenen Zielgrad erreicht.

Algorithm 1: Sweeping eines (nicht-linearen) Taylormodells

input: Taylormodell t, Grad deg, Strategie $strat$

```
1  Polynomial p ← [ ];
2  foreach Monomial m ∈ t.polynomial do
3  |   rest ← degree(m) − deg;
4  |   if rest < 1 then
5  |   |   p.insert(m);
6  |   else
7  |   |   foreach ErrorSymbol es ∈ m do
8  |   |   |   if rest < threshold then
9  |   |   |   |   break;
10 |   |   |   Sweep according to Strategy;
11 |   |   |   rest ← rest − amount swept;
12 |   |   end
13 |   |   p.insert(m);
14 end
15 return t
```

square_first / simple

Hier wird zunächst `square_only` auf das Monom angewandt, um danach die restlichen Grade ohne quadratisches Sweeping zu reduzieren. Allerdings hat diese Methode keinerlei Vorteile gegenüber `simple`, da der Effekt des Quadrierens von einem nicht-quadratischen Sweep wieder aufgehoben wird.

3. Implementierung der Polynome

Die Implementierung der einzelnen Bestandteile ist objektorientiert. Dieses Kapitel bietet einen Überblick über die Struktur des Programmes und den wichtigsten zu Verfügung stehenden Funktionen.

3.1 Darstellung

Intervalle

Intervalle enthalten Informationen über deren Zentrum `center` und Radius `radius`, beziehungsweise deren Breite, welche, je nach gewähltem Zahlentypen, reelle oder rationale Zahlen sind. Zudem können Intervalle ein- oder beidseitig unendlich sein, weshalb wiederum eine Abfrage nach `IntType`[1], also dem Intervalltypen möglich ist. Existiert eine Schranke, also ist das Intervall endlich oder einseitig unendlich, kann diese abgefragt werden.

Beim Verrechnen von Intervallen sind folgende Regeln zu beachten:

$$[x_1, x_2] + [y_1, y_2] = [x_1 + y_1, x_2 + y_2]$$

$$[x_1, x_2] - [y_1, y_2] = [x_1 - y_2, x_2 - y_1]$$

$$[x_1, x_2] \cdot [y_1, y_2] = [min(x_1y_1, x_1y_2, x_2y_1, x_2y_2), max(x_1y_1, x_1y_2, x_2y_1, x_2y_2)]$$

$$[x_1, x_2]/[y_1, y_2] = [min(x_1/y_1, x_1/y_2, x_2/y_1, x_2/y_2), max(x_1/y_1, x_1/y_2, x_2/y_1, x_2/y_2)]$$

Hier wird der Effekt deutlich, dass Intervalle nach jeder Operation die Tendenz haben, zu wachsen. Wird bespielsweise ein Intevall von sich selbst subtrahiert, wird es größer, statt sich wegzukürzen.

Fehlersymbole

Ein Fehlersymbol λ besteht aus seinem Exponenten, einem `Pointer` auf das Intervall in dem sein Wert liegt, welches sich während der Berechnung verändern kann und dem Index des Fehlersymbols.

Monome

Momome werden durch einen Intervallkoeffizienten und eine Liste von nach ihrem Index aufsteigend geordneten Fehlersymbolen dargestellt. Die Implementierung der Liste der Fehlersymbole als `std::vector`, ermöglicht die Verwendung der *C++ Standard Template Library* für Funktionen, wie zum Beispiel die Berechnung des Grades eines Monoms.

[1] FINITE, LOWER_BOUND, UPPER_BOUND, ENTIRE

Polynome

Ein Polynom setzt sich aus einem Kernintervall (`kernel`), also dem Monom ohne Fehlersymbole und einer Liste ungleicher Monome zusammen. Ungleich bedeutet in diesem Fall, dass sich die Zusammensetzung oder der Exponent einzelner Fehlersymbole zwischen den Monomen unterscheidet.

3.2 Addition

Wird wie in [**?**] eine Ordnung $\succ$ auf den Monomen definiert, können die Polynome sortiert werden.

- $\lambda_1 \succ \lambda_2 \succ ... \succ \lambda_k$
- $\lambda_n^i \lambda_m^j \succ \lambda_n^{i'} \lambda_m^{j'}$ für $n > m$, falls

  $i > i'$, oder
  $i = i'$ und $j > j'$

Diese Ordnung ist partiell, da gleiche Monome, beziehungsweise Monome mit gleichen Fehlersymbolen nicht vergleichbar sind. Tritt dieser Fall während der Addition zweier Polynome auf, bedeutet das, dass die Koeffizienten beider Monome addiert werden und das so entstandene Monom in die Polynom-Summe aufgenommen wird. Haben die verglichenen Monome die selbe Anzahl an Variablen, entscheidet die Größe des Exponenten des Fehlersymbols mit dem kleinsten Index. Algorithmus 2 implementiert diese Ordnung und hat den Vorteil, dass nicht immer alle Fehlersymbole aus beiden Listen betrachtet werden müssen. Sobald sich die beiden Monome, genauer deren Fehlersymbole, an der ersten Stelle unterscheiden, terminiert die Schleife. Die Vorbedingung hierfür ist, dass die Fehlersymbole in den Listen aufsteigend sortiert sind.

In vorhergegangen Implementierungen der Fehlersymbole waren diese noch keine Objekte. Stattdessen hatte jedes Monom einen `vector` mit einem Eintrag pro Fehlersymbol. Der Index des Fehlersymbol entsprach dem Index innerhalb des Vectors, der Wert im `vector` dem Exponenten, was die Datenstruktur vereinfachte, aber erforderte, dass alle Monome dieselbe Anzahl an Fehlersymbolen hatten. Da sich diese allerdings ändern kann, wenn durch ein `Polish` ein neues λ hinzugefügt oder eines der λs durch `Sweeping` kein Vorkommen mehr hat und entfernt werden kann, wurde eine andere Herangehensweise verwendet.

Algorithm 2: max_errorsymbols

input: Zwei Listen von Fehlersymbolen ess_1, ess_2

```
1 size ← max{length(ess_1), length(ess_2)};
2 i ← 0 ;
3 while i < size do
4 |   if i ≥ length(ess_1) or ess_1[i] ≻ ess_2[i] then
5 |   |   return right
6 |   else if i ≥ length(ess_2) or ess_1[i] ≺ ess_2[i] then
7 |   |   return left
8 end
9 return equal
```

Werden nun sortierte Polynome p_1, p_2 addiert, geschieht lediglich ein Zusammenführen (merge) zweier geordneter Listen, was im schlimmsten Fall $\#p_1 + \#p_2$ Vergleiche der Monome benötigt[2].

Algorithm 3: Addition zweier Polynome

input: Polynome p_1, p_2

```
1  p ← (), i ← 0, j ← 0 ;
2  while i < #p1 and j < #p2 do
3  |   max ← max_errorsymbols(p1[i], p1[j]) ;
4  |   if max = left then
5  |   |   p.append(p1[i]);
6  |   |   i ← i + 1;
7  |   else if max = right then
8  |   |   p.append(p2[j]);
9  |   |   j ← j + 1;
10 |   else
11 |   |   // max = equal
12 |   |   p.append(p1[i] + p1[j]);
13 |   |   p1.remove(i);
14 |   |   i ← i + 1; j ← j + 1;
15 end
16 Append the rest to p;
17 return p
```

Algorithmus 3 erzeugt aus zwei nach der oben definierten Ordnung $\prec$ sortierten Polynomen wiederum ein sortiertes Polynom. Endet die While-Schleife in Zeile 2, so ist im Regelfall eines der Polynome noch nicht gänzlich betrachtet, was bedeutet, dass alle übrigen Monome in diesem Polynom (im Hinblick auf $\prec$) kleiner sind, als die des anderen Summanden und daher dem Ergebnispolynom angehangen werden können, ohne sie weiter zu betrachten.

3.3 Multiplikation

In einer linearen Implementierung der Taylormodelle wird bei der Multiplikation immer eines der Fehlersymbole gesweept, sodass die Länge eines Polynoms nicht über die Dimension hinausgeht und somit der Schwerpunkt bei der Intervallarithmetik liegt. Lässt man allerdings höhere Ordnungen zu, werden auch die Polynome länger und ein Weiteres Problem tritt auf. Um die oben (3.2) definierte Ordnung $\prec$ auf dem Polynom $p = p_1 \cdot p_2$ zu erhalten, müssen die $n \cdot m$ Monome ($n := \#p_1, m := \#p_2$), die bei der Multiplikation entstehen sortiert werden. Werden diese sequentiell zu p hinzugefügt bedeutet das 2 Vergleiche, dann 3, dann 4, und so weiter, bis hin zu nm Vergleichen:

$$2 + 3 + 4 + 5 + ... + nm = \frac{nm \cdot (nm + 1)}{2} - 1$$

So ergibt sich für die naive Multiplikation eine quadratische Laufzeit in O-Notation von $O(nm + (nm)^2)$

[2] $\#p$ liefert die Anzahl an Monomen des Polynoms p

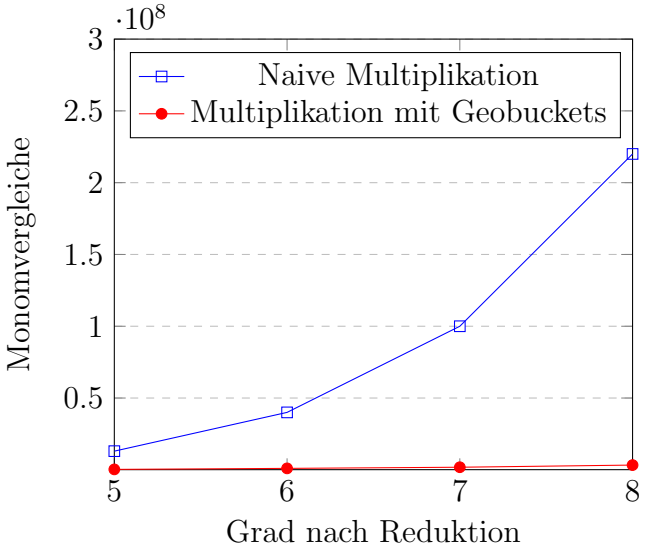

Abbildung 3.1: Naive Multiplikation gegen Multiplikation mit Geobuckets

Für effizientere Polynommultiplikation existieren verschiedene Algorithmen. Viele der schnellen bekannten Algorithmen basieren auf der Schnellen-Fourier-Transformation, wobei die Polynome in Stützvektoren und wieder zurück umgewandelt werden. Da es sich bei den Koeffizienten den Monome um Intervalle mit möglicherweise unendlichen Schranken handelt, ist das Rückführen eines Stützvektors nicht immer ohne Weiteres möglich. (TODO Nachgucken FFT Intervallartihmetik)

Ein Weiterer Ansatz besteht darin, die Anzahl der benötigten Monomvergleiche zu reduzieren, wenn die Monome einsortiert werden müssen. Betrachtet man die Summe dreier Polynome $p_1 + p_2 + p_3$ mit $\#p_1 >> \#p_2 = \#p_3 = 1$, benötigt das Aufsummieren von links nach rechts $2\#p_1 + 1$ Vergleiche, da mit dem oben verwendeten Additionsverfahren zweifach in p_1 eingefügt wird. Summiert man jedoch von rechts nach links so werden lediglich $\#p_1 + 2$ Vergleiche benötigt. Nach dieser Idee kann die Anzahl der Monomvergleiche reduziert werden, indem zunächst Polynome gleicher Länge miteinander addiert werden. Mit Geobuckets, von Yan ([**?**]) eingeführt und Monagan und Pearce ([**?**]) für die Multiplikation angepasst, werden die Zwischenergebnisse der Polynommultiplikation in geometrisch wachsenden 'Buckets' gespeichert. Hat ein Bucket seine Kapazität erreicht, wird das Polynom zum nächst größeren Bucket hinzuaddiert. Der Unterschied zum Originalalgorithmus besteht darin, dass die Größe der hinzukommenden Polynome bereits kennt und die Bucketkapazität dementsprechend anpassen kann. Algorithmus **??** zeigt die in `hotm` implementierte Version der Geobucketmultiplikation. Durch den Teile und Herrsche Ansatz, kann die Multiplikation mit Geobuckets in $O(nm \log nm)$ durchgeführt werden.

Sei $x_0 = 0.5 + 1\lambda_1 + 1\lambda_2 + 1\lambda_3$ und einem Startfehler von 2^{-21}, also $\lambda_i \in [0 \pm 2^{-20}]$ für $i \in \{1, 2, 3\}$. Berechnet man nun $x_{i+1} = 3.75 \cdot x_i \cdot (1 - x_i)$ zeigen sich deutliche Unterschiede bei den benötigten Monomvergleichen, beziehungsweise dem Aufruf der Funktion `max_monomial` (Algorithmus 2).

Algorithm 4: Polynommultipliaktion mit Geobuckets

input: Polynome p_1, p_2

```
// Initialize
n ← #p_1, m ← #p_2 ;
p ← [ ] ;
Allocate buckets with space for polynomials: {2m, 4m, ..., 2^⌈log_2(n)⌉−1 m};
i ← 0;
// Main Loop
while i < n do
    p ← p_1[i] · p_2;
    if i < (n − 1) then
        p ← p + p_1[i + 1] · p_2;
    i ← i + 2;
    j ← 0;
    while buckets[j].not_empty() do
        p ← p + buckets[j] ;
        buckets[j] ← [ ];
        j ← j + 1;
    end
    if i < n then
        buckets[j] ← p;
        p ← [ ] ;
end
// Merge each bucket into the final polynomial
foreach bucket ∈ buckets do
    p ← p + bucket;
end
return p
```

Naive Multiplikation		
Grad nach Reduktion	CPU-Zeit	#Vergleiche
5	2.375 s	$\approx 1.3 \cdot 10^7$
6	5.281 s	$\approx 4 \cdot 10^7$
7	11.281 s	$\approx 1 \cdot 10^8$
8	22.406 s s	$\approx 2.2 \cdot 10^8$
Multiplikation mit Geobuckets		
Grad nach Reduktion	CPU-Zeit	#Vergleiche
5	1.875 s	$\approx 3 \cdot 10^5$
6	3.859 s	$\approx 1 \cdot 10^6$
7	6.406 s	$\approx 1.8 \cdot 10^6$
8	11.14 s	$\approx 3.3 \cdot 10^6$

Tabelle 3.1: Berechnung von x_{10} mit Reduzierung des Grades nach jeder Iteration. Gemessen mit dem iRRAM-internen Debugmodus und Code-Coverage.

4. Durchführung

4.1 Verwendung der iRRAM-REALs

Die Software-Bibliothek `iRRAM` [**?**] basiert auf Intervallen als Zahlentyp, um diese mit einer beliebigen Genauigkeit darstellen zu können. Zunächst wird mit Double-Präzision (64-Bit Zahlen) gerechnet, welche verwendet werden, bis das Ergebnis für die angefragte Präzision nicht mehr genau ausgegeben werden kann, beziehungsweise bis zu einer bestimmten Anzahl an Bits. Ist dies der Fall, geschieht eine Iteration mit einer erhöhten Genauigkeit, also längeren Zahlen, welche dann mit Hilfe von `MPFR` dargestellt werden. Für eine solche Iteration werden gerade so viele Zwischenergebnisse während der Berechnung gespeichert, dass eine Wiederholung der Schritte mit höherer Präzision möglich ist. Da nicht die gesamte Berechnung in jeder Iteration wiederholt wird, müssen während der Laufzeit Sichtbarkeit von Variablen und Zwischenergebnisse für den Nutzer genau kontrolliert und unter Umständen beschränkt werden, da sonst unerwartetes Verhalten und Exceptions entstehen können, indem die zugegriffenen Werte gegebenfall nicht in der aktuellen Iteration existieren.

Einige der für rationale Zahlen zur Verfügung stehenden Funktionen, wie der Vergleich zweier Zahlen, sind mit reellen Zahlen nicht ohne Weiteres Möglich. Dies gilt insbesondere für den Test auf Gleichheit und die Vorzeichenfunktion *sign*. Bei all diesen Funktionen handelt es sich im Reellen (und bei der `iRRAM`) um mehrwertige Funktionen, da sich das jeweilige Ergebnis mit veränderter Präzsion in der Darstellung der Zahlen ändern kann. Dieses Problem wird in `hotm` durch den `SignType` adressiert. Zusätzlich zu den Werten 'positiv' (=`POS`) und 'negativ' (=`NEG`), kann die Vorzeichenfunktion `sign` den Wert 'ambivalent' (=`AMBI`) ausgeben, wenn nicht entscheidbar ist, wo genau die reelle Zahl um die Null liegt. Hierfür erhält die Vorzeichenfunktion einen Parameter, der den Bereich er Unsicherheit definiert. Ist der Ausgabewert 'ambivalent', so wird in den Funktionen, welche die Vorzeichenfunktion aufrufen, der schlechteste Fall im Hinblick auf die Genauigkeit des Ergebnisses angenommen.

Eine weitere Besonderheit ergibt sich aus dem Aufbau der Zahlen, denn es entstehen zwei Intervall-'Ebenen': Zum Einen, die Darstellung des Koeffizienten als Intervall aus Mitte und Radius. Zum Anderen die Darstellung von Mitte und Radius, als `iRRAM-REAL`, also auch wiederum jeweils als Intervall mit Wert und Fehler, wie in Grafik **??** zu sehen ist. Im Vergleich zu den `mpq-RATIONALS` erhöht sich hier zwar die Komplexität deutlich, allerdings lassen sich Ungenauigkeiten sehr genau steuern, indem zum Beispiel der Rechenfehler des Mittelpunkts eines Koeffizienten auf den Radius 'verlagert' wird. So vergrößert sich zwar der Radius des Koeffizienten, welcher dadurch ungenauer wird, jedoch verkleinert sich der Rechenfehler auf der Zahlenebene der `REAL`s. Ein ähnlicher Effekt sollte sich durch Rundung auch mit den `mpq-RATIONALS` erreichen lassen.

Das Verlagern der Ungenauigkeit (e in der Grafik **??**) auf den Wert des Radius' (v in der Grafik), *Micro-Housekeeping* genannt, verringert die Intervallbreite und erzeugt Punktintervalle auf der Zahlenebene, allerdings werden die Intervalle auf der Intervallebene breiter. Hier greift dann wiederum der Polish-Mechanismus, der auf Polynomebene Monome hinzufügt, um aus den zu groß gewordenen Intervallen wiederum Punktintervalle zu machen, *Macro-Housekeeping* genannt. So wird der Rechenfehler von der untersten Ebene bis zur Polynomebene propagiert. Diese Housekeeping-Funktionen werden durch Parameter gesteuert, die bestimmen, ab wann ein Intervall zu breit ist und die jeweilige Prozedur angewandt werden soll.

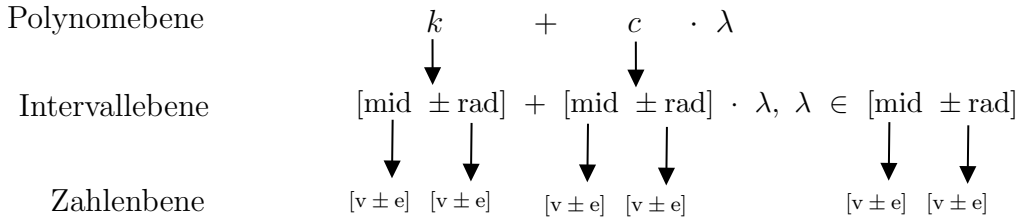

Abbildung 4.1: Ebenen der Polynomdarstellung mit REALs (informell)

4.2 Genauigkeitsmodell

Die `iRRAM` bietet die Möglichkeit, zwischen *absoluter* und *relativer* Genauigkeit zu unterscheiden. Absolute Genauigkeit bedeutet, dass beim Verrechnen zweier `REAL`s ein global verwendeter Wert für die Genauigkeit zugrunde liegt und die Zahlen dementsprechen skaliert werden. Bei relativer Genauigkeit hängt die verwendete Genauigkeit von der tatsächlichen Größe der jeweiligen Zahlen ab. Diese kann zum Beispiel durch punktuelle Anwendung von Micro-Housekeeping stark variieren, weshalb die Verwendung von relativer Genauigkeit in der Praxis besser funktioniert.

Für der Vergleich wurde x_{1000} mit $x_n = c \cdot x_{n-1} \cdot (1 - x_{n-1})$ mit der folgenden Konfiguration berechnet:

1. Nur quadratisches Sweeping zur 0,
2. Entfernen aller Fehlersymbole, bis auf die 3 zuletzt hinzugefügten,
3. Micro-Housekeeping (Cleaning) ab einem Fehler χ auf Zahlenebene $> 10^{-100}$,
4. Macro-Housekeeping (Polish) ab einer Intervallbreite $\psi > 10^{-50}$,
5. $x_0 = 0,5$ und
6. Ausgabe des Ergebnisses auf 20 Stellen genau.

Zu Beginn einer jeden Iteration werden zunächst die höhergradigen Monome gesweept (1.), dann die älteren Fehlersymbole (2.). Im nächsten Schritt wird der Fehler und die Intervalle bereinigt (3.) und zuletzt das Polynom poliert (4.), bevor erneut multipliziert wird.

Für $c = 3.25$ konvergiert die Funktion gegen zwei Werte und lässt sich sehr genau bestimmen.

x_{1000} mit $c = 3.25$		
mit Housekeeping	relative Genauigkeit	absolute Genauigkeit
Anzahl Polishs	142	6961
Präzision (Bits)	`double`	136
CPU-Zeit	0.8s	4.43s

Wie in der Tabelle zu sehen ist, benötigt die Berechnung mit relativer Genauigkeit erheblich weniger Zeit und Bits um ein Ergebnis mit der geforderten Auflösung zu erhalten. Wird auf die Housekeeping-Methoden verzichtet, zeigt sich dieser massive Unterschied jedoch nicht:

x_{1000} mit $c = 3.25$		
ohne Housekeeping	relative Genauigkeit	absolute Genauigkeit
Anzahl Polishs	-	-
Präzision (Bits)	7440	7440
CPU-Zeit	0.120s	0.118s

Die Ergenisse werden ohne Housekeeping erheblich schneller berechnet, allerdings auf Kosten der benötigten Bits für die Zahlendarstellung. Da hier kein Polieren geschieht, wird auch kein Taylormodell als solches initialisiert, das heißt, es werden keine Fehlersymbole eingesetzt und rein auf der Ebene der reellen Zahlen der `iRRAM` gerechnet.

Für $c = 3.75$, einem Wert, bei dem die Funktion chaotisches Verhalten aufweist und zwischen vielen Werten hin- und herspringt, ist erkennbar, dass ein Rechnen mit diesen Housekeeping-Parametern und absoluter Genauigkeit nicht praktikabel ist, da die Berechnung bereits nach 60 Iterationen eine Laufzeit von über 15 Sekunden hat, welche scheinbar exponentiell steigt. Für relative Genauigkeit ist die Berechnung zwar zeitaufwändig, liefert jedoch ein (korrektes) Ergebnis und benötigt weniger Bits, als eine Berechnung ohne Housekeeping. Die Parameter mussten hier auf $\psi = 10^{-140}$ und $\chi = 10^{-150}$ angepasst werden, damit die Grenzwerte erreicht werden.

x_{1000} mit $c = 3.75$		
mit Housekeeping	relative Genauigkeit	absolute Genauigkeit
Anzahl Polishs	64	?
Präzision (Bits)	5894	?
CPU-Zeit	8.2s	>60s

Wie oben zeigt sich zwischen relativer und absoluter Genauigkeit bei der Verwendung ohne Housekeeping-Methoden kein nennenswerter Unterschied. Zudem scheint bei der hier verwendeten Konfiguration lediglich ein Berechnungsoverhead zu entstehen, welcher keine Auswirkungen auf die Qualität des Ergebnisses hat.

x_{1000} mit $c = 3.75$		
ohne Housekeeping	relative Genauigkeit	absolute Genauigkeit
Anzahl Polishs	-	-
Präzision (Bits)	7440	7440
CPU-Zeit	0.12s	0.12s

Insgesamt lässt sich beobachten, dass die Berechnung der Interationen mit relativer Genauigkeit in den untersuchten Fällen bessere Ergebnisse liefert, weshalb diese im Folgenden zugrunde liegt.

4.3 Housekeeping

Die Werte der Parameter für die Housekeeping-Methoden hängen unmittelbar voneinander ab: Je größer der Wert des Cleaning χ, also as Umlagern des Rechenfehlers eines Koeffizienten auf den Mittelpunkt des Radius', desto schneller wird der Grenzwert des Polishs ψ, also des Einführens eines neuen Fehlersymbols erreicht.

Für $c = 3.25$ kann x_{1000} mit $\psi = 10^{-50}$ in `double`-Präzision berechnet werden. Je mehr sich ψ und χ annähern, desto öfter überschreitet der Radius eines Koeffizienten den Grenzwert ψ und wird poliert, wie in Grafik 4.2 zu sehen ist.

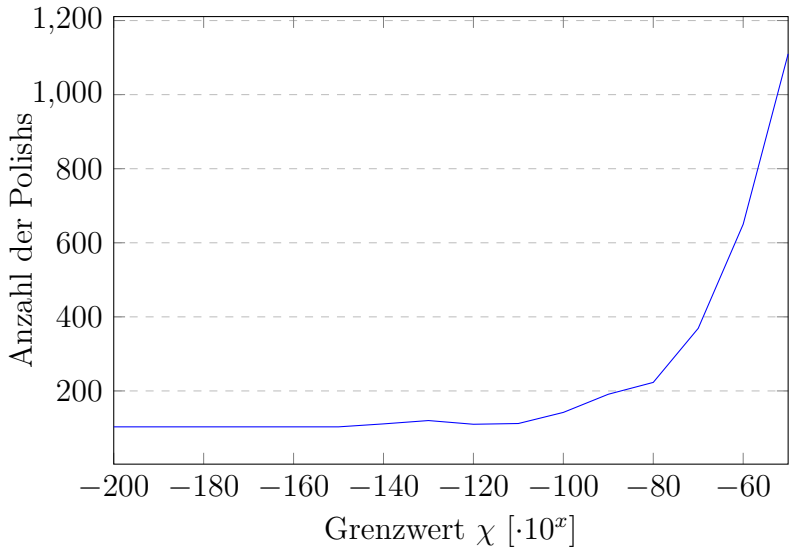

Abbildung 4.2: Anzahl der Polishs für x_{1000} bei $\psi = 10^{-50}$ und wachsendem χ

Ab einem $\chi < 10^{-70}$ und $\psi < 0.1$ und mit $\chi < 10^{-50}$ mit $\psi < 10^{-50}$ mit `double`-Präzision genau berechenbar. In Tabelle 4.1 sind dies die Werte mit 50 Bit Genauigkeit.

Für $c = 3.75$ ist $\chi = 10^{-150}$ und $\psi = 10^{-140}$ eine gute Konfiguration mit 5894 benötigten Bits für x_{1000}. Dieser Wert wird bei keiner der getesteten Konfigurationen unterschritten, was mit Hilfe des Lyapunov Exponenten für diese spezielle Iteration erklärt werden kann. Der Lyapunov Exponent gibt eine Annäherung an die untere Schranke der pro Iteration durch Rundungsfehler verlorenen Bits an, die für $c = 3.25$ niedriger ist, als für $c = 3.75$, welhalb eine Berechnung mit `double`-Präzision möglich ist [**?**].

$\cdot 10^{-x}$ ψ χ	0	10	20	30	40	50	60	70	80	90	100
0	7440 0	7440 0	7440 0	7440 0	7440 0	7440 0	7440 0	7440 0	7440 0	7440 0	7440 0
10	7440 0	9372 0	2876 172	2876 124	2876 113	2876 103	2876 101	2876 107	2876 124	2876 130	2876 142
20	7440 0	242 1277	50 2780	748 565	748 555	375 1594	375 1880	748 913	1008 672	375 2790	1008 793
30	7440 0	242 1231	2876 178	2876 103	2876 78	2876 74	2876 55	2876 51	2876 49	2876 51	2876 55
40	7440 0	50 1256	2876 176	2876 103	748 830	2876 59	2876 51	2876 48	2876 45	2876 47	1008 3249
50	7440 0	541 1092	2876 157	2876 100	136 626	50 1110	50 2058	50 3061	50 3104	50 3252	50 4307
60	7440 0	50 1330	2876 197	2876 102	50 360	50 650	50 793	50 1478	50 2831	50 3006	50 3046
70	7440 0	50 1209	50 305	50 182	50 244	50 369	50 585	50 730	50 1498	50 2198	50 3052
80	7440 0	50 1265	50 283	50 209	50 193	50 223	50 346	50 618	50 619	50 2099	50 2540
90	7440 0	50 1279	50 285	50 179	50 133	50 191	50 229	50 365	50 651	50 851	50 1760
100	7440 0	50 1181	50 332	50 184	50 134	50 142	50 199	50 236	50 391	50 571	50 709

Tabelle 4.1: Kombinationsmöglichkeiten der Grenzwerte für Housekeeping-Methoden. Die Zelle besteht aus den benötigten Bits für die Ausgabe und der Anzahl der Polishs für die Berechnung von x_{1000} für $c = 3.25$, wie in Kapitel 4.2 beschrieben

4.4 Messungenauigkeiten

Die feingranularen Konfigurationsmöglichkeiten ermögleichen eine manuelle Eingabe eines Rechenfehlers zu Beginn der Berechnung, wodurch bespielsweise das Rechnen mit ungenauen Werten simuliert werden kann. Für $x_n = x_{n-1} \cdot c \cdot (1 - x_{n-1})$ und $x_0 = 0.5$ kann x_0 als Taylormodell mit Intervallkoeffizienteneinen solchen Rechenfehler an verschiedenen Stellen enthalten:

$$x_0 = [0 \pm 0] + [1 \pm 0] \cdot \lambda \qquad \lambda \in [0.5 \pm \varepsilon] \tag{4.1}$$
$$x_0 = [0.5 \pm 0] + [1 \pm 0] \cdot \lambda \qquad \lambda \in [0 \pm \varepsilon] \tag{4.2}$$
$$x_0 = [0.5 \pm 0] + [\varepsilon \pm 0] \cdot \lambda \qquad \lambda \in [0 \pm 1] \tag{4.3}$$
$$x_0 = [0 \pm 0] + [0.5 \pm \varepsilon] \cdot \lambda \qquad \lambda \in [1 \pm 0] \tag{4.4}$$
$$x_0 = [0.5 \pm 0] + [0 \pm \varepsilon] \cdot \lambda \qquad \lambda \in [1 \pm 0] \tag{4.5}$$
$$x_0 = [0.5 \pm \varepsilon] \tag{4.6}$$

Bei einem Fehler $\varepsilon = 10^{-200}$ und $c = 3.25$ kann bei **??** und **??** bereits nach wenigen Iterationen keine Aussage mehr über das Ergebnis getroffen werden, da die Intervalle zu stark wachsen, beziehungsweise zu breit werden. Mit den restlichen Definitionen kann die Berechnung auch bei höheren Iterationszahlen ($n = 1000, n = 10000$) mit `double`-Präzision ausgeführt werden.

Für die Iteration mit $c = 3.75$, also einem Wert mit größerem Informationsverlust pro Iteration verhalten sich die Berechnungen etwas anders. Mit derselben manuell auf 743 Bit fixierten Genauigkeit entwickelt sich die Intervallbreite des Ergebnisses stark abhängig von der Darstellung von x_0, wie in Grafik **??** zu sehen ist.

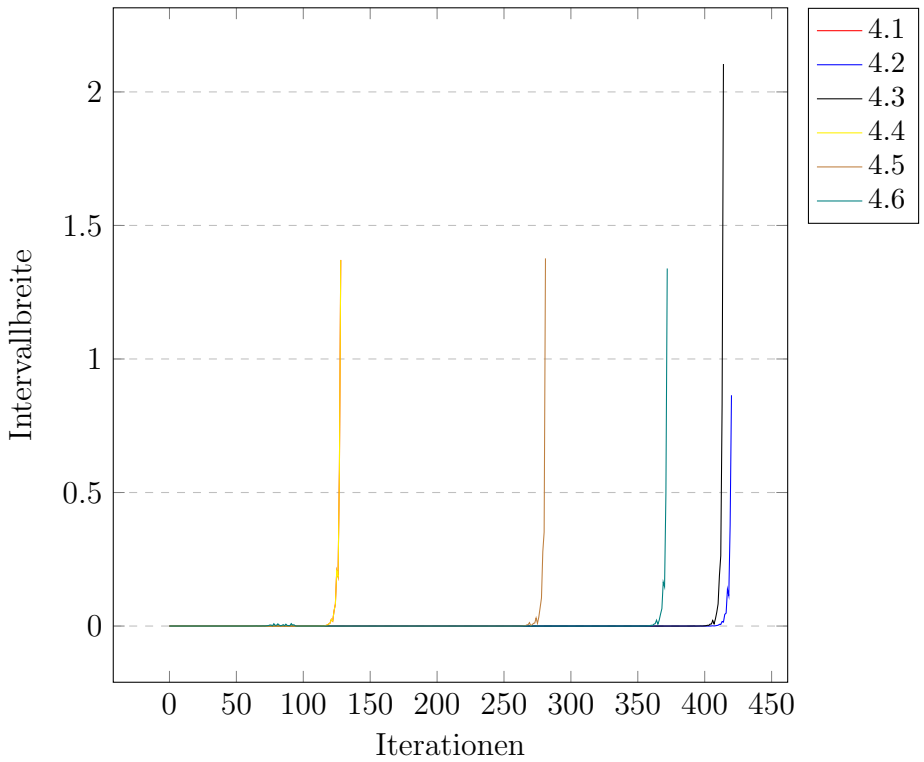

Abbildung 4.3: Taylormodelle mit verschiedenen Definitionen für x_0 mit $c = 3.75$

Die Intervallbreite wächst ab einer bestimmten Iterationszahl exponentiell an und lässt keine genaue Ausgabe des Ergebnisses für die aktuelle Präzision mehr zu. Für das Taylormodell **??** wächst das Intervall hierbei am spätesten, die Ergebnisse bleiben also länger aussagekräftig. Wie auch bei $c = 3.25$ sind die Taylormodelle **??** und **??**, also solche mit 0-Kernel am schlechtesten geeignet.

Als Genauigkeit wurde 743 Bit gewählt, da sich hier große Unterschiede zwischen den Taylormodell-Definitionen gezeigt haben. Bei einem kleineren Wert zeigten sich diese zwar auch, jedoch war dann keine so hohe Iterationszahl möglich.

5. Evaluation

Mit verschiedenen Sweepingstrategien, Zahlentypen, Definitionsmöglichkeiten der Taylormodelle, initialer Ungenauigkeit, Genauigkeitsmodell (absolute oder relative Genauigkeit) und weiteren Einstellungsmöglichkeiten, ergibt sich ein sehr großer Suchraum zum Finden der optimalen Konfiguration für nichtlineare Taylormodelle. Allerdings ermöglichen sich dadurch sehr feingranulare Vergleiche, um herauszufinden, ob eine Implementierung nichtlinearer Taylormodelle in der Praxis schneller oder genauer als eine lineare sein kann. In diesem Kapitel werden Ergebnisse einige dieser Optionen betrachtet und versucht, die beobachteten Effekte zu erklären.

6. Diskussion und Ausblick

6.1 TODO

- 'Runden' für rationale Zahlen auf dyadische Zahlen: Abschneiden einer Dezimalzahl, wo die Genauigkeit zu groß wird und dann Einführen eines Taylormodells, dass den 'vorderen Teil der Zahl' als Kernel und die nun fehlende Genauigkeit als im Lambda umschließt.
- Finden oder Entwickeln von Heuristiken, die entscheiden, welche Fehlersymbole am besten gesweept werden sollten, wenn der Grad eines Monoms reduziert werden soll. Bei einer Reihe von Tests, wo bestimmte Konfigurationen zum Sweepen verwendet wurden, gab es Anzeichen darauf, dass die Qualität des Ergebnisses der Evaluation/Linearisierung von der Reihenfolge abhängt mit der die einzelnen Fehlersymbole entfernt werden.
- Anzahl der zu behaltenen Fehlersymbole pro Iteration beobachten.
- Metawissen über die Intervalle verwenden: Punktintervalle gesondert behandeln und den Radius nicht als iRRAM-Real mit Wert 0.

Zum aktuellen Zeitpunkt beschränkt sich diese Forschungsarbeit auf die Anwendung der Taylormodelle bei Iterationen der logistischen Abbildung, da es sich hierbei um ein gut verstandenes und viel untersuchtes Problem handelt. Diese Beschränkung birgt jedoch die Gefahr des 'Overfittings' der Implementierung auf dieses spezielle Problem, weshalb im Weiteren Verlauf des Forschungsprojekt auch andere Probleme betrachtet werden, wie zum Beispiel die Henon-Abbildung.

Literaturverzeichnis

[] F. Brauße, M. V. Korovina und N. T. Müller. Using Taylor Models in Exact Real Arithmetic. In I. S. Kotsireas, S. M. Rump und C. K. Yap (Hrsg.), *Mathematical Aspects of Computer and Information Sciences - 6th International Conference, MACIS 2015, Berlin, Germany, November 11-13, 2015, Revised Selected Papers*, Band 9582 der *Lecture Notes in Computer Science*. Springer, 2015, S. 474–488.

[] T. Granlund und the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.2.0. Auflage, 2020. http://gmplib.org/.

[] K. Makino und M. Berz. Higher order verified inclusions of multidimensional systems by taylor models. *Nonlinear Analysis: Theory, Mehtods and Applications* 47(5), 2001, S. 3514–3503.

[] M. B. Monagan und R. Pearce. Polynomial Division Using Dynamic Arrays, Heaps, and Packed Exponent Vectors. In V. G. Ganzha, E. W. Mayr und E. V. Vorozhtsov (Hrsg.), *Computer Algebra in Scientific Computing, 10th International Workshop, CASC 2007, Bonn, Germany, September 16-20, 2007, Proceedings*, Band 4770 der *Lecture Notes in Computer Science*. Springer, 2007, S. 295–315.

[] N. T. Müller. The iRRAM: Exact Arithmetic in C++. In J. Blanck, V. Brattka und P. Hertling (Hrsg.), *Computability and Complexity in Analysis, 4th International Workshop, CCA 2000, Swansea, UK, September 17-19, 2000, Selected Papers*, Band 2064 der *Lecture Notes in Computer Science*. Springer, 2000, S. 222–252.

[] N. Müller. Enhancing imperative exact real arithmetic with functions and logic. 2009.

[] C. Spandl. Computational Complexity of Iterated Maps on the Interval (Extended Abstract). In X. Zheng und N. Zhong (Hrsg.), *Proceedings Seventh International Conference on Computability and Complexity in Analysis, CCA 2010, Zhenjiang, China, 21-25th June 2010*, Band 24 der *EPTCS*, 2010, S. 139–150.

[] K. Weihrauch. *Computable Analysis - An Introduction*. Texts in Theoretical Computer Science. An EATCS Series. Springer. 2000.

[] T. Yan. The Geobucket Data Structure for Polynomials. *J. Symb. Comput.* 25(3), 1998, S. 285–293.