Fakultät für Elektrotechnik, Informatik und Mathematik
Fachgruppe Algorithmen und Komplexität
Fürstenallee 11
33102 Paderborn

Diplomarbeit

Zur Praktikabilität schneller Polynomarithmetik

Sven Köhler
21. Juni 2007

Betreut durch:
Dr. Martin Ziegler

Zusammenfassung

Die Anwendungen schneller Polynomarithmetik, d.h. Algorithmen mit subquadratischer Laufzeit, liegen z.B. in der Kryptographie aber auch in der Physik (N-Teilchen Simulation). Diese Diplomarbeit widmet sich der Untersuchung verschiedener Varianten der schnellen Polynommultiplikation und der darauf aufbauenden Algorithmen zur Division mit Rest und schnellen Mehrfachauswertung. Sie untersucht die Praktikabilität der Algorithmen unter verschiedenen Aspekten: Laufzeit (Break-Even-Points, Hybridisierung) und numerischer Stabilität, experimentell wie grundsätzlich (reelle Berechenbarkeit im Sinne der Rekursiven Analysis).

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit untersucht die Praktikabilität schneller Polynomarithmetik. Polynomarithmetik ist ein Überbegriff für alle arithmetischen Operationen auf Polynomen. Dazu gehören Addition, Subtraktion, Multiplikation, Division mit Rest, aber auch die Auswertung von Polynomen.

Aus der Schule sind einfache Verfahren für das Rechnen mit Polynomen bekannt. Die Addition und Subtraktion zweier Polynome sowie die Auswertung eines Polynoms kann in Linearzeit berechnen werden. Die Verfahren für Multiplikation und Division benötigen allerdings quadratische Laufzeit.

Verfahren zur Multiplikation und Division mit subquadratischer Laufzeit sind bekannt [GG03]. Sie werden schon vielfach eingesetzt – z.B. in der Krytographie, wo Polynome über endlichen Körpern betrachtet werden.
Eine weitere Anwendung ist die N-Teilchen Simulation – eine Problemstellung aus der Physik. Die Berechnungen finden hier typischerweise im Reellen statt. Neben der Geschwindigkeit ist in diesem Fall auch die Genauigkeit der Ergebnisse ein entscheidenes Kriterium für die Güte eines Algorithmus.

In Kapitel 2 werden zunächst die bekannten Algorithmen für schnelle Polynommultiplikation vorgestellt: ein Verfahren mit Laufzeit $\mathcal{O}(n^{1.585})$ und dann ein FFT-basiertes Verfahren mit Laufzeit $\mathcal{O}(n \log(n))$. Das Laufzeitverhalten der eigenen C++-Implementierungen wird dann genau analysiert. Die Algorithmen werden untereinander verglichen und zusätzlich der hoch optimierten FFT-Bibliothek FFTW [FJ05] gegenüber gestellt.
Der zweite Teil des Kapitels stellt die bekannten Verfahren zur schnellen Division und der schnellen Mehrfachauswertung vor.

Kapitel 3 gibt einen Überblick, in wie fern die schnellen Verfahren Verbreitung gefunden haben. Anschließend wird in Kapitel 4 die Stabilität der Verfahren – insbesondere der schnellen Mehrfachauswertung – für reelle Eingaben untersucht. Es werden essentielle Probleme aufgezeigt. Mit der Bibliothek iRRAM für „exakte reelle Arithmetik in C++“ wird dann versucht, die Probleme zu kompensieren.

Abschließend wird in Kapitel 5 die Berechenbarkeit der Division mit Rest im Sinne der rekursiven Analysis erörtert.

2 Schnelle Polynomarithmetik

Dieses Kapitel stellt verschiedene bekannte schnelle Algorithmen zum Rechnen mit Polynomen vor [GG03].

Zunächst werden Algorithmen zur schnellen Multiplikation betrachtet. Anschließend wird die praktische Laufzeit von C++-Implementierungen dieser Algorithmen ermittelt. Es werden Optimierungen aufgezeigt, die die Laufzeit einiger Algorithmen verbessern konnten. Außerdem wird die Frage beantwort, welche der Multiplikationsalgorithmen sich für welche Eingabegrößen bezüglich ihrer Laufzeit am besten eignen.

Danach werden weitere Algorithmen zur schnellen Division mit Rest und zur schnellen Mehrfachauswertung eines Polynoms vorgestellt. Die schnelle Division baut dabei auf der effizienten Multiplikation auf, und die schnelle Mehrfachauswertung baut wiederrum auf der schnellen Division sowie der schnellen Multiplikation auf. Abbildung 2.1 veranschaulicht diesen Zusammenhang.

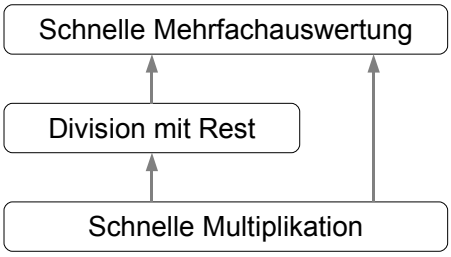

Abbildung 2.1: Beziehungen zwischen den Algorithmen

Neben der einfachen Darstellung eines Polynoms $f(x) := a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} \in \mathbb{R}[x]$ vom Grad kleiner $n \in \mathbb{N}$ wird im Folgenden auch die Darstellung als Vektor $(a_0, a_1, \ldots, a_{n-1}) \in \mathbb{R}^n$ benutzt. Die Polynome vom Grad kleiner n bildet dabei den n-dimensionalen Vektorraum mit der Basis $\{1, x, x^2, \ldots, x^{n-1}\}$, welcher isomoprh zu $\mathbb{R}^n$ ist.

Eine besondere Regelung gilt für das Nullpolynom $f(x) := 0$. Diesem wird aus Konsistenzgründen der Grad $-\infty$ zugeordnet. Es kann auch durch den leeren Vektor $() \in \mathbb{R}^0$ dargestellt werden.

2.1 Multiplikation

Betrachten wir die zwei Polynome $f := f_0 + f_1 x + \ldots + f_{n-1} x^{n-1} \in \mathbb{R}[x]$ und $g := g_0 + g_1 x + \ldots + g_{n-1} x^{n-1} \in \mathbb{R}[x]$, mit $n \in \mathbb{N}$. Das Produkt der beiden Polynome kann durch einfaches Ausmultiplizieren und Ausklammern berechnen werden:

$$\begin{aligned} r &= f \cdot g \\ &= (f_0 + f_1 x + \ldots + f_{n-1} x^{n-1})(g_0 + g_1 x + \ldots + g_{n-1} x^{n-1}) \\ &= f_0 g_0 + (f_0 g_1 + f_1 g_0) x + \ldots + (f_{n-2} g_{n-1} + f_{n-1} g_{n-2}) x^{2n-3} + f_{n-1} g_{n-1} x^{2n-2} \end{aligned}$$

Die Potenzen der Variablen x im Ergebnispolynom r haben dabei immer höchstens den Exponent $2n-2$. Man stellt also fest, dass das Produkt der zwei Polynome f und g vom Grad kleiner n im schlimmsten Fall $2n-1$ Koeffzienten ungleich 0 besitzt.

Die Schulmethode, d.h. Ausmultiplizieren und Ausklammern, mündet dann direkt in einer konkreten Formel für den k-ten Koeffizienten des Polynoms $r = f \cdot g$, welche auch als Cauchy-Produktformel bekannt ist:

$$r_k = \sum_{\substack{i+j=k \\ 0 \le i,j < n}} a_i b_j \qquad \text{für } 0 \le k < 2n-1$$

Satz 2.1. *Die naive Polynommultiplikation (Schulmethode, Cauchy-Produktformel) berechnet das Produkt zweier Polynome f und g vom Grad kleiner n in Laufzeit $\mathcal{O}(n^2)$.*

Beweis. Bei der Anwendung der Schulmethode bzw. beim Auswerten der Cauchy-Produktformel wird jeder Koeffizient von f mit jedem Koeffizienten von g multipliziert und die n^2 Ergebnisse werden dann zu $n+m-1$ Werten aufsummiert. Die obigen Aussagen über die Laufzeit folgen sofort daraus. □

Eine wesentliche Verbesserung der Laufzeit erreicht der folgende Algorithmus.

2.1.1 Karatsuba

Der Karatsuba-Algorithmus verfolgt einen divide-and-conquer Ansatz. Er zerteilt die zu multiplizierenden Polynome f und g jeweils in zwei Teile, die dann geschickt dazu benutzt werden, dass Ergebnis der Multiplikation $f \cdot g$ schneller zu berechnen.

Betrachten wir wieder die beiden Polynome $f := f_0 + f_1 x + \ldots + f_{n-1} x^{n-1} \in \mathbb{R}[x]$ und $g := g_0 + g_1 x + \ldots + g_{n-1} x^{n-1} \in \mathbb{R}[x]$. Die Polynome f und g werden dann jeweils in zwei Hälften zerteilt:

$$\begin{aligned} f &= F_0 + F_1 x^m \\ g &= G_0 + G_1 x^m \end{aligned}$$

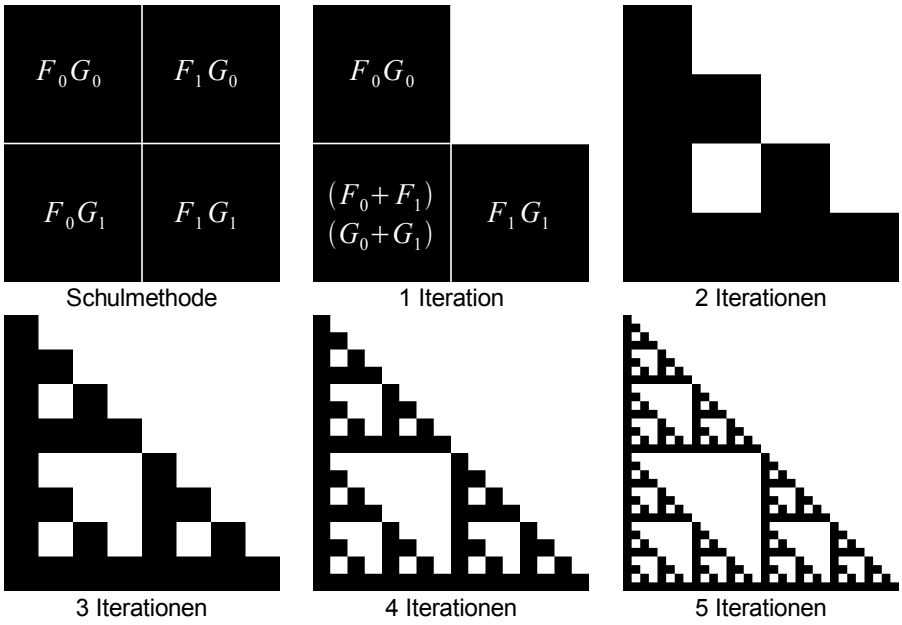

Abbildung 2.2: [GG03, FIGURE 8.2] Veranschaulichung der Laufzeit des Karatsuba Algorithmus

Dabei wird $m = \lfloor \frac{n}{2} \rfloor$ gewählt, um die Polynome genau in der Mitte zuzerteilen. Das Polynom F_0 vom Grad kleiner $\lfloor \frac{n}{2} \rfloor$ enthält dann die untere Hälfte der Koeffizienten von f, und das Polynom F_1 vom Grad kleiner $\lceil \frac{n}{2} \rceil$ enthält dann die obere Hälfte der Koeffizienten. Mit den Polynomen G_0 und G_1 verhält es sich entsprechend. Durch die Zerteilung ergibt sich nun:

$$\begin{aligned} f \cdot g &= (F_0 + F_1 x^m)(G_0 + G_1 x^m) \\ &= F_0G_0 + (F_0G_1 + F_1G_0)x^m + F_1G_1x^{2m} \end{aligned}$$

Der eigentliche Trick besteht nun darin, den Term $F_0G_1+F_1G_0$ mit nur einer anstatt zwei zusätzlichen Multiplikationen zu berechnen. Dies gelingt durch die folgende Umformung:

$$\begin{aligned} f \cdot g &= F_0G_0 + (F_0G_1 + F_1G_0)x^m + F_1G_1x^{2m} \\ &= F_0G_0 + (F_0G_1 + F_1G_0 + F_0G_0 + F_1G_1 - F_0G_0 - F_1G_1)x^m + F_1G_1x^{2m} \\ &= F_0G_0 + ((F_0 + F_1)(G_0 + G_1) - F_0G_0 - F_1G_1)x^m + F_1G_1x^{2m} \end{aligned}$$

In der obigen Formel sind nun insgesamt fünf Produkte enthalten. Die Produkte F_0G_0 und F_1G_1 werden allerdings doppelt verwendet und müssen daher nur einmal berechnet werden. Um die obige Formel auszuwerten, müssen also nur noch drei verschiedene Multiplikationen ausgeführt werden, nämlich:

$$F_0G_0 \qquad F_1G_1 \qquad (F_0 + F_1)(G_0 + G_1)$$

Diese drei Multiplikationen werden wiederum rekursiv mit dem Karatsuba-Algorithmus ausgerechnet. Im Laufe der Rekursion wird dabei der Grad der zu multiplizierenden Polynome immer kleiner, bis sich das Problem auf die Multiplikation zweier Körperelemente reduziert. Abbildung 2.2 illustriert, was durch die Einsparung einer der vier Multiplikationen und der rekursiven Anwendung dieser Idee passiert.

Die Multiplikationen mit x^m und x^{2m} fallen für die Laufzeit nicht ins Gewicht, da sie in der Implementierung lediglich als Veschiebeoperationen bzw. Manipulation von Arrayindizes auftreten. Abbildung 2.3 zeigt, wie die als Array dargestellten Zwischenergebnisse zum Endergebnis zusammengerechnet werden.

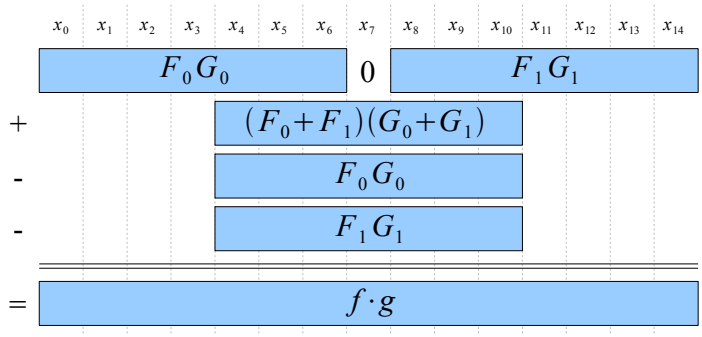

Abbildung 2.3: Berechnung des Ergebnisses des ersten Karatsuba-Rekursionsschritts mit $n = 8$ und $m = 4$

Satz 2.2. *Die Karatsuba-Multiplikation berechnet das Produkt zweier Polynome f und g vom Grad kleiner n in Laufzeit $\mathcal{O}\left(n^{1.585}\right)$*

Beweis. In einem Rekursionsschritt ruft sich der Algorithmus drei mal rekursiv auf, um Polynome der Länge $\frac{n}{2}$ miteinander zu multiplizieren. Der zusätzliche Aufwand, um die Eingabe zu zerlegen und die Teilergebnisse zum Endergebnis zusammenzusetzen besteht im Wesentlichen aus drei Polynomadditionen (siehe Abbildung 2.3). Die Längen der zu addierenden Polynome sind dabei linear in n. Für die Laufzeit $T(n)$ ergibt sich:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}\left(n\right)$$

Unter Anwendunge des Master-Theorems folgt daraus:

$$T(n) \in \mathcal{O}\left(n^{\log_2 3}\right)$$

□

2.1.2 FFT-Multiplikation

Die FFT-basierte Polynommultiplikation kann die asymptotische Laufzeit gegenüber dem Kararsuba-Algorithmus abermals verbessern. Die Polynome werden mit Hilfe der „**F**ast **F**ourier **T**ransform“ (kurz FFT) in eine andere Darstellung überführt in welcher sie effizient miteinander multipliziert werden können. Das Ergebnis wird dann mit Hilfe der inversen FFT zurücktransformiert.
In der digitalen Signalverarbeitung ist dieses Prinzip ebenfalls verbreitet und wird z.B. benutzt, um große digitale Filter effizient anwenden zu können.

Die FFT bildet also den Kern dieses Multiplikationsalgorithmus. Sie ist ein effizienter Algorithmus zur Berechnung der diskreten Fourier Transformation (kurz DFT). Die diskrete Fourier Transformierte $\hat{f} = (\hat{f}_0, \hat{f}_1, \ldots, \hat{f}_{n-1})$ eines Vektors $f = (f_0, f_1, \ldots, f_{n-1})$ ist dabei wie folgt definiert:

$$\begin{aligned}
\hat{f}_k &= \sum_{j=0}^{n-1} e^{-2\pi\imath\frac{jk}{n}} f_j \\
&= \sum_{j=0}^{n-1} \left(\left(e^{\frac{-2\pi\imath}{n}} \right)^k \right)^j f_j \\
&= \sum_{j=0}^{n-1} \left(\omega^k \right)^j f_j \qquad \text{mit } \omega := e^{\frac{-2\pi\imath}{n}}
\end{aligned}$$

Der Vektor f repräsentiert dabei das entsprechende Polynom $f = f_0 + f_1 x + \ldots + f_{n-1} x^{n-1}$ und $\omega := e^{\frac{-2\pi\imath}{n}}$ ist eine primitive n-te Einheitswurzel. Man stellt fest, dass die obige Definition der DFT eine mehrfache Auswertung des Polynoms f darstellt, nämlich:

$$\hat{f}_k = f(\omega^k)$$

Mit anderen Worten: Ist die Eingabe der DFT/FFT ein n-dimensionaler Koeffizientenvektor eines Polynoms, dann ist die Ausgabe ein Vektor mit n Stützwerten. Die Stützstellen sind die Potenzen ω^k mit $k = 0, 1, \ldots, n-1$. Diese sind wie in Abbildung 2.4 regelmäßig (startend bei 1, im Uhrzeigersinn) auf dem Einheitskreis in der komplexen Ebene verteilt.

Die inverse DFT/FFT hingegen nimmt eine Interpolation vor. Das heisst: Ist die Eingabe der iDFT/iFFT ein n-dimensionaler Vektor mit n Stützwerten an den entsprechenden Stützstellen ω^k, so ist die Ausgabe eine n-dimensionaler Koeffizientenvektor eines Polynoms vom Grad kleiner n dessen Graph alle Stütztpunkte enthält.
Dieses Interpolationsproblem ist dabei weder über- noch unterbestimmt. Es gibt genau ein Polynom vom Grad kleiner n, welches durch alle n Stützpunkte geht.

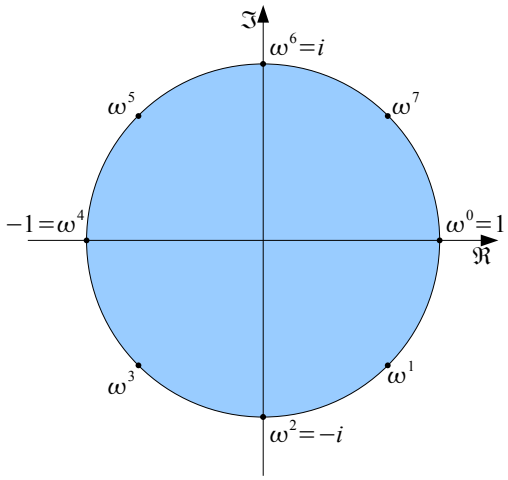

Abbildung 2.4: Einheitskreis mit den Potenzen von ω für $n = 8$

Wir betrachten die Polynome $f = f_0 + f_1 x + \ldots + f_{n-1} x^{n-1} \in \mathbb{C}[x]$ und $g = g_0 + g_1 x + \ldots + g_{n-1} x^{n-1} \in \mathbb{C}[x]$. Mit Hilfe der FFT können die Polynome wie folgt miteinander multipliziert werden:

- Berechne jeweils $N \geq 2n - 1$ Stützwerte:
  $\hat{f} := \text{FFT}\left((f_0, \ldots, f_{n-1}, 0, \ldots, 0)\right)$
  $\hat{g} := \text{FFT}\left((g_0, \ldots, g_{n-1}, 0, \ldots, 0)\right)$
- Multipliziere komponentenweise:
  $\hat{r} := \hat{f} * \hat{g}$
- Bestimme Koeffizienten des Produktpolynoms:
  $r := \text{iFFT}\left(\hat{r}\right)$

Im ersten Schritt werden die Koeffizienten von f und g mit der FFT transformiert. Die Eingabe der FFT wird dabei mit Nullen zu einem N-dimensionalen Vektor aufgefüllt. Man erhält dann $N \geq 2n - 1$ viele Stützwerte. N wird dabei so nah an $2n - 1$ gewählt, wie es die zugrundeliegende FFT-Implementierung erlaubt. Viele Implementierungen erfordern einen Sprung zur nächsten Zweierpotenz.
Da für das Produktpolynom $r = f \cdot g$ die Gleichung

$$r(x) = f(x)g(x) \qquad \forall x \in \mathbb{C}$$

gelten muss, erhält man nun im zweiten Schritt durch komponentenweise Multiplikation $N \geq 2n - 1$ Stütztwerte des Polynoms r. Mit weniger als $2n - 1$ Stützwerten kann das Polynom r im Allgemeinen nicht ausreichend beschrieben werden, da es im schlimmsten Fall vom Grad $2n - 2$ ist.

Die Koeffizienten des gesuchten Polynoms r erhalten wir nun durch Rücktransformation mit der inversen FFT.

Für den Fall, dass zwei reelle Polynome miteinander multipliziert werden sollen, werden diese von der verwendeten Implemtierung in eine komplexe Eingabe konvertiert. Die Ausgabe der Multiplikation im Komplexen wird dann durch abschneiden der Imaginärteile zurück ins Reelle konvertiert. Es wurde keine spezielle Optimierung der FFT für reelle Eingaben vorgenommen.

Satz 2.3. *Die FFT-Multiplikation berechnet das Produkt zweier reeller oder komplexer Polynome f und g vom Grad kleiner n in Laufzeit $\mathcal{O}(n\log(n))$.*

Beweis. Für einen n-dimensionalen Eingabevektor benötigen die FFT und iFFT jeweils $\mathcal{O}(n\log(n))$ Additionen und Multiplikation im Körper $\mathbb{C}$. [GG03, Theorem 8.18]
Die FFT wird zweimal und die iFFT wird einmal für N-dimensionale Eingabevektoren aufgerufen. Dabei gilt $N \geq 2n-1$. Weiterhin wird im schlimmsten Fall zur nächsten Zweierpotenz aufgerundet. Daraus folgt $N < 4n-2$. Die Aufrufe der FFT verursachen also Aufwand $\mathcal{O}((4n-2)\log(4n-2)) \subseteq \mathcal{O}(n\log(n))$. Die punktweise Multiplikation der Stützwerte schlägt lediglich mit linearem Aufwand zu buche. □

2.1.2.1 Cooley Tukey FFT

Wie oben bereits erläutert, entspricht das Berechnen einer diskreten Fourier Transformierten $\hat{f} = \mathrm{DFT}(f)$ dem Auswerten eines Polynoms f vom Grad kleiner n an den Stellen ω^k, mit $k = 0, \ldots, n-1$ und $\omega := e^{\frac{-2\pi\imath}{n}}$. Nimmt man diese n-fache Auswertung auf naive Art und Weise vor, benötigt sie Laufzeit $\mathcal{O}(n^2)$.
Unter dem Begriff FFT werden nun verschiedene Algorithmen zusammengefasst, die dieses Problem in subquadratischer Laufzeit – typischerweise $\mathcal{O}(n\log(n))$ – lösen. Ein heute weit verbreiteter Ansatz für einen FFT-Algorithmus wurde 1995 von James Cooley und John W. Tukey veröffentlicht [CT65]. Es handelt sich dabei um einen divide-and-conquer Ansatz.

Betrachten wir das Polynom $f = f_0 + f_1 x + \ldots + f_{n-1}x^{n-1} \in \mathbb{C}[x]$. Die klassische Cooley Tukey FFT arbeitet nun folgendermaßen [GG03, Algorithm 8.14]:

- Berechne Polynom $r_0(x) := \sum_{j=0}^{n/2-1} (f_j + f_{j+n/2})x^j$
- Berechne Polynom $r_1(x) := \sum_{j=0}^{n/2-1} (f_j - f_{j+n/2})\omega^j x^j$:
- Berechne durch Rekursion: $r_0(1), r_1(1), r_0(\omega^2), r_1(\omega^2), \ldots, r_0(\omega^{n-2}), r_1(\omega^{n-2})$

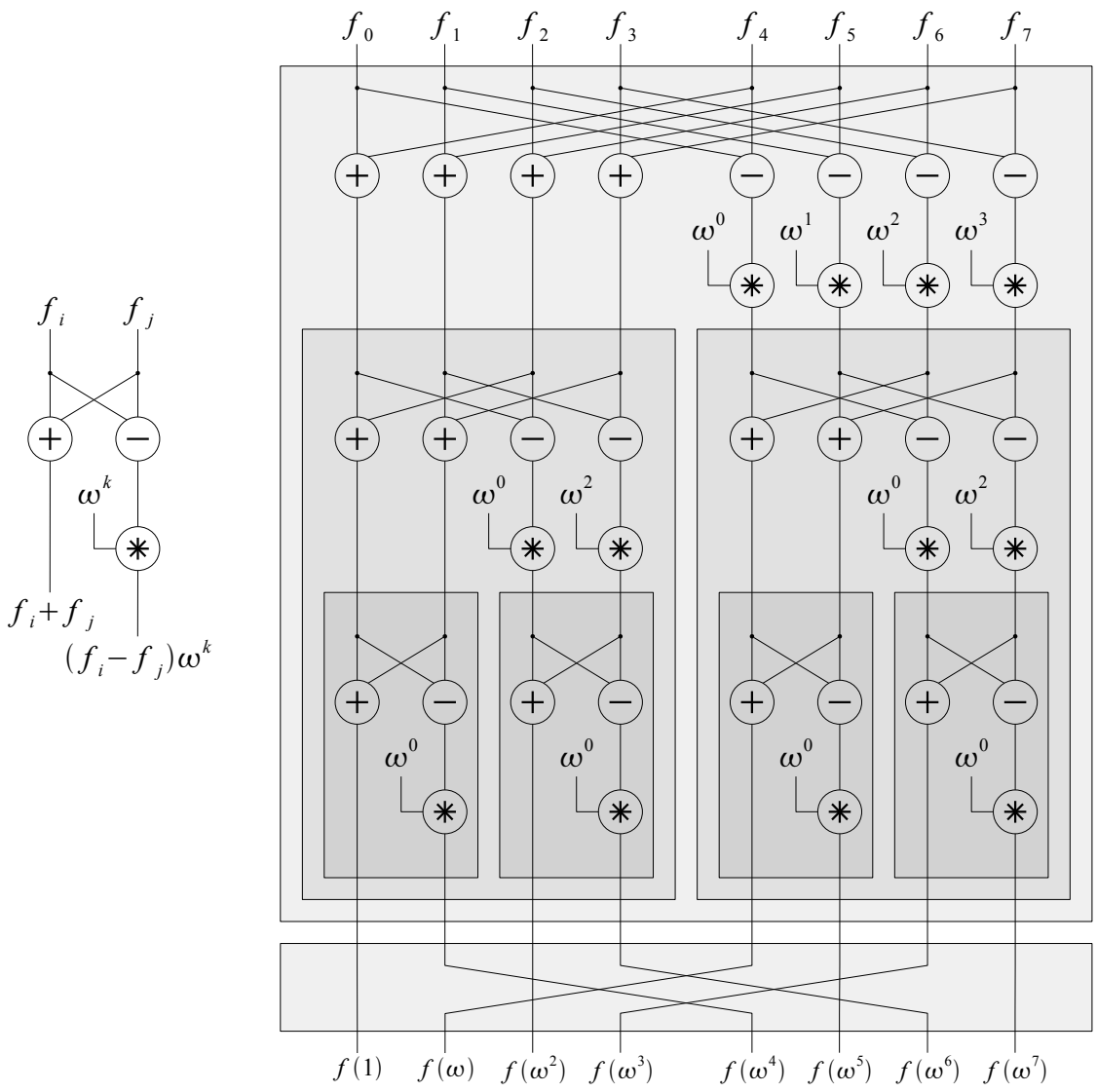

Abbildung 2.5: [GG03, FIGURE 8.4] Einzelne Butterfly-Operation und komplettes Schaltnetz einer Cooley Tukey FFT für $n = 8$

Die Polynome r_0 und r_1 sind vom Grad kleiner n. Diese werden dann stellvertretend für f durch Rekursion an den Potenzen von ω^2 ausgewertet. Dies entspricht einer FFT mit Eingabelänge $\frac{n}{2}$. Durch die Rekursion wird das Problem also in immer kleinere Teilprobleme zerlegt, bis es sich auf eine FFT mit Eingablänge 1 reduziert (entspricht der Identität).
Die Korrektheit folgt sofort aus den folgenden Betrachtungen.
Sei $k \in \mathbb{N}$, dann gilt:

$$\begin{aligned}
r_0(\omega^{2k}) &= \sum_{j=0}^{n/2-1} (f_j + f_{j+n/2})\omega^{2kj} \\
&= \sum_{j=0}^{n/2-1} f_j\omega^{2kj} + \sum_{j=0}^{n/2-1} f_{j+n/2}\omega^{2kj}\omega^{kn} \\
&= \sum_{j=0}^{n/2-1} f_j\omega^{2kj} + \sum_{j=n/2}^{n-1} f_j\omega^{2k(j-n/2)}\omega^{kn} \\
&= \sum_{j=0}^{n-1} f_j\omega^{2kj} = f(\omega^{2k})
\end{aligned}$$

$$\begin{aligned}
r_1(\omega^{2k}) &= \sum_{j=0}^{n/2-1} (f_j - f_{j+n/2})\omega^{j}\omega^{2kj} = \sum_{j=0}^{n/2-1} (f_j - f_{j+n/2})\omega^{(2k+1)j} \\
&= \sum_{j=0}^{n/2-1} f_j\omega^{(2k+1)j} + \sum_{j=0}^{n/2-1} f_{j+n/2}\omega^{(2k+1)j}\omega^{kn}\omega^{n/2} \\
&= \sum_{j=0}^{n/2-1} f_j\omega^{(2k+1)j} + \sum_{j=n/2}^{n-1} f_j\omega^{(2k+1)(j-n/2)}\omega^{kn}\omega^{n/2} \\
&= \sum_{j=0}^{n-1} f_j\omega^{(2k+1)j} = f(\omega^{2k+1})
\end{aligned}$$

Es wurde $\omega^{kn} = 1$ und $\omega^{n/2} = -1$ benutzt. Beides folgt direkt aus der Definition von ω (siehe auch Abbildung 2.4).

Der obige Algorithmus lässt sich nun zu einem Schaltnetz ausrollen, wie es Abbildung 2.5 zeigt. Es besteht hauptsächlich aus Butterfly-Operationen. In dem Schaltnetz lässt sich ein Muster erkennen:
Zuerst wird *ein* großer Block aus $\frac{n}{2}$ Butterfly-Operationen auf die Eingabe angewendet, dann *zwei* Blöcke mit je $\frac{n}{4}$ Butterfly-Operationen, dann *vier* Blöcke mit je $\frac{n}{4}$ Butterfly-Operationen, usw. usf., bis hin zu $\frac{n}{2}$-vielen Blöcken mit je einer Butterfly-Operation.

Der erste große Block aus Butterfly-Operationen entspricht der Zerlegung von f in die Polynome r_0 und r_1. Deren Auswertung findet dann durch die rekursive Zerlegung statt.

Auf dem Rückweg der Rekursion werden die Ergebnisse allerdings in jedem Schritt permutiert. So entsprechen die Werte $r_0(1), r_1(1), r_0(\omega^2), r_1(\omega^2), \ldots$ nur in dieser Reihenfolge den Werten $f(1), f(\omega), f(\omega^2), f(\omega^3), \ldots$. Die Werte von r_0 und r_1 müssen also verzahnt werden. Alle durch die Rekursion entstehenden Verzahnungen summieren sich auf zu einer symmetrischen Permutation – dem letzten Schritt im FFT-Schaltnetz in Abbildung 2.5. Bei dieser Permutaion wird ein Element an i-ter Stelle (bei 0 beginnend) mit dem Element j vertauscht, wobei die Zahl j durch Umkehren der Reihenfolghe der $\log(n)$-vielen Bits von i entsteht. Also wird das Element $i = 6 = (110)_2$ mit dem Element $j = 3 = (011)_2$ vertauscht. Ein weiteres Beispiel wäre $i = 1 = (001)_2$ und $j = 4 = (100)_2$.

Wie oben beschrieben kann der Algorithmus durch Anwenden von Butterfly-Blöcken und anschließender Permutaion ohne Rekursion implementiert werden. Desweiteren kann er auch „in-place“ (ohne zusätzlichen Speicher) durchgeführt werden. Auch die abschließende Permutation kann „in-place“ durchgeführt werden, da es sich um eine symmetrische Permutation handelt.

Für die Berechnung der inversen FFT wird eine spezielle Eigenschaft der DFT ausgenutzt [CB93, S. 14]. Für einen Vektor $f \in \mathbb{R}^n$ gilt $f' = \frac{1}{n} DFT(DFT(f))$, wobei der Vektor f' eine Permutation des Vektors f darstellt. Ein Algorithmus für die inverse FFT berechnet also zuerst den Vektor $f' = \frac{1}{n} FFT(\hat{f})$ um diesen dann anschließend zu permutieren.

2.2 Zur Laufzeit der Multiplikation

Es wurden nun drei Varianten der Polynommultiplikation vorgestellt:

- Schulmethode: $\mathcal{O}(n^2)$
- Karatsuba: $\mathcal{O}(n^{1.585})$
- FFT: $\mathcal{O}(n \log(n))$

Die Erwartung war, dass die Schulmethode für kleine Eingaben schneller als Karatsuba und FFT ist, da sich Karatsuba und FFT die asymptotisch schnellere Laufzeit durch große Konstanten bei der $\mathcal{O}$-Notation erkaufen müssen. Weiterhin wurde erwartet, dass der Karatsuba erst die Schulmethode einholt und dann selbst von der FFT eingeholt wird. Diese drei Ansätze würden sich dann kombinieren lassen, indem je nach Eingabegröße das schnellste Verfahren benutzt würde.

Bei den ersten Messungen stellte sich allerdings schnell heraus, dass der Karatsuba erst für relativ große Eingabelängen schneller als die Schulmethode wurde. Zudem hatte die

FFT basierte Methode sowohl Karatsuba als auch die Schulmethode schon für kleinere Eingabegrößen eingeholt. Dies führte zu diesen Fragen:

- Kann der Karatsuba optimiert werden, so dass er mit der FFT konkurieren kann?
- Für welche Eingabegrößen ist welches der drei Verfahren am besten geeignet?

2.2.1 Hybridisierung

In Abbildung 2.6 sieht man die Laufzeitentwicklung der drei vorgestellen Verfahren. Der Karatsuba Algorithmus holt die Schulmethode erst jenseits der Eingabelänge $n = 16384 = 2^{14}$. Beide benötigen für das Multiplizieren von zwei Polynomen vom Grad kleiner 20480 ca. $1,15$ Sekunden. Der relative Abstand der beiden Laufzeitkurven ist ca. zwischen 32 und 64 am größten. Dort benötigt der Karatsuba ca. die siebenfache Laufzeit der Schulmethode.

Der Karatsuba wurde nun so geändert, dass er für Eingaben, deren Länge unterhalb eines gewissen Schwellwertes liegt, nicht mehr sich selbst rekursiv aufruft, sondern die viel schnellere Schulmethode benutzt. Die dadurch entstandene Verbesserung der Laufzeit wurde durch die Rekursion nach oben weitergegeben werden und wirkte sich auch erheblich auf größere Eingaben aus.

In Abbildung 2.7 sind die Laufzeiten verschiedener Karatsuba-Hybriden für Eingabelängen von 2^{12} bis 2^{14} dargestellt. Sie unterschieden sich nur durch den gewählten Schwellwert. Erst für Eingabelängen unter diesem Schwellwert wird die Schulmethode eingesetzt.

Die Hybriden mit Schwellwerten zwischen 32 und 64 sind am schnellsten. Kleinere oder größere Schwellwerte verschlechtern die Laufzeit wieder. Der Schwellwert 64 wurde deshalb in der Implementierung als Standardwert eingesetzt.

Die Laufzeit des Karatsuba-Algorithmus hat sich durch die Hybridisierung entscheidend verbessert. Die Abbildung 2.8 stellt den Karatsuba nochmal den Hybriden mit Schwellwert 32 und 64 gegenüber. Der normale Karatsuba benötigt für Eingabelänger $n = 16384$ fast 600 Millisekunden während die Hybrid-Varianten mit nur 100 Millisekunden auskommen.

2.2.2 FFTW - eine hoch optimierte FFT

Für diese Arbeit wurde eine einfache Cooley Tukey FFT implementiert. Diese wurde bereits in Abschnitt 2.1.2.1 vorgestellt. In diesem Abschnitt wird diese relativ unoptimierte Implementierung mit der hoch optimierten OpenSource-Bibliothek FFTW [FJ05] verglichen. Diese Bibliothek bietet eine Vielzahl von Funktionen, darunter:

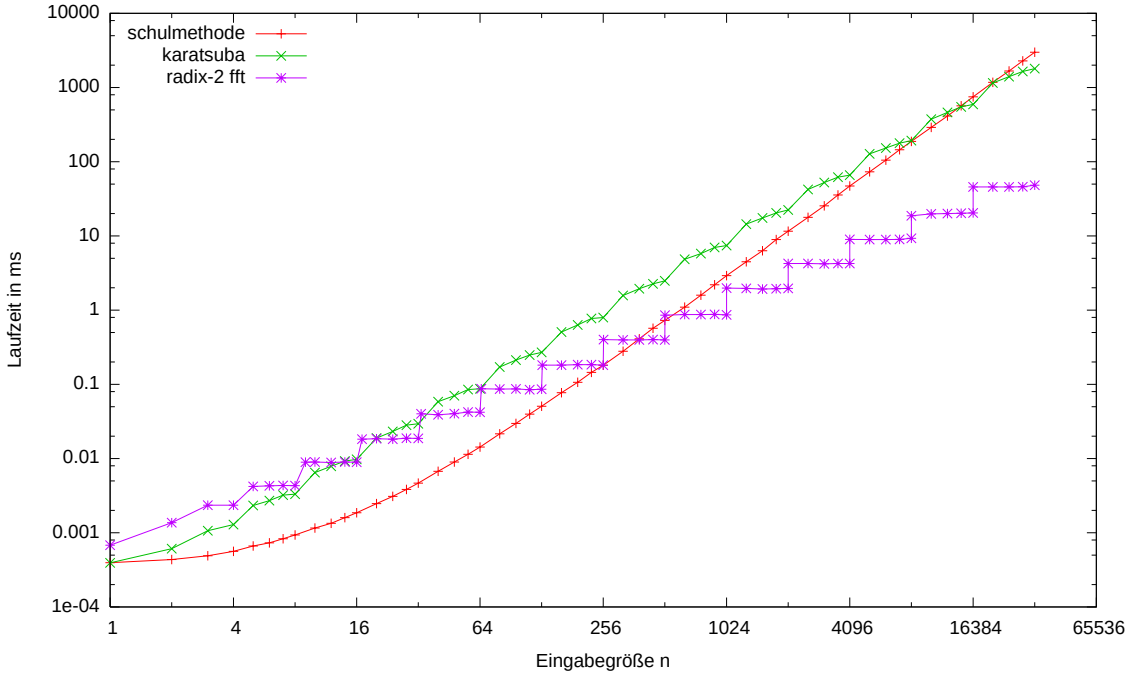

Abbildung 2.6: Laufzeit der unoptimierten Algorithmen im Überblick

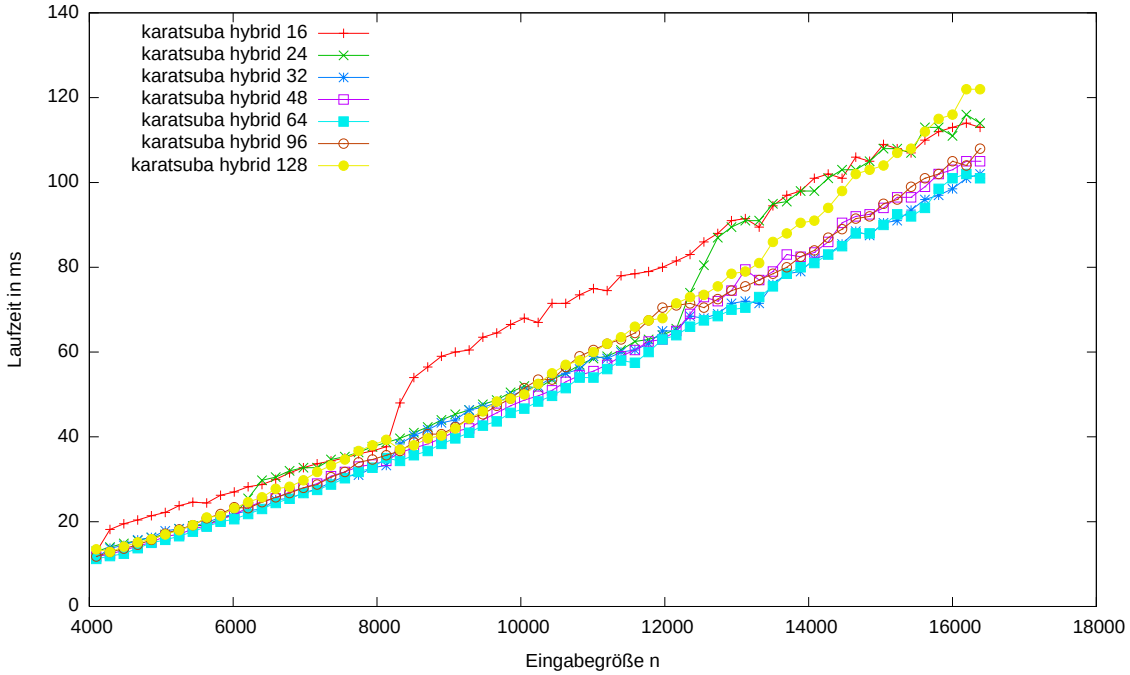

Abbildung 2.7: Verschiedene Karatsuba-Hybriden mit unterschiedlichen Schwellwerten

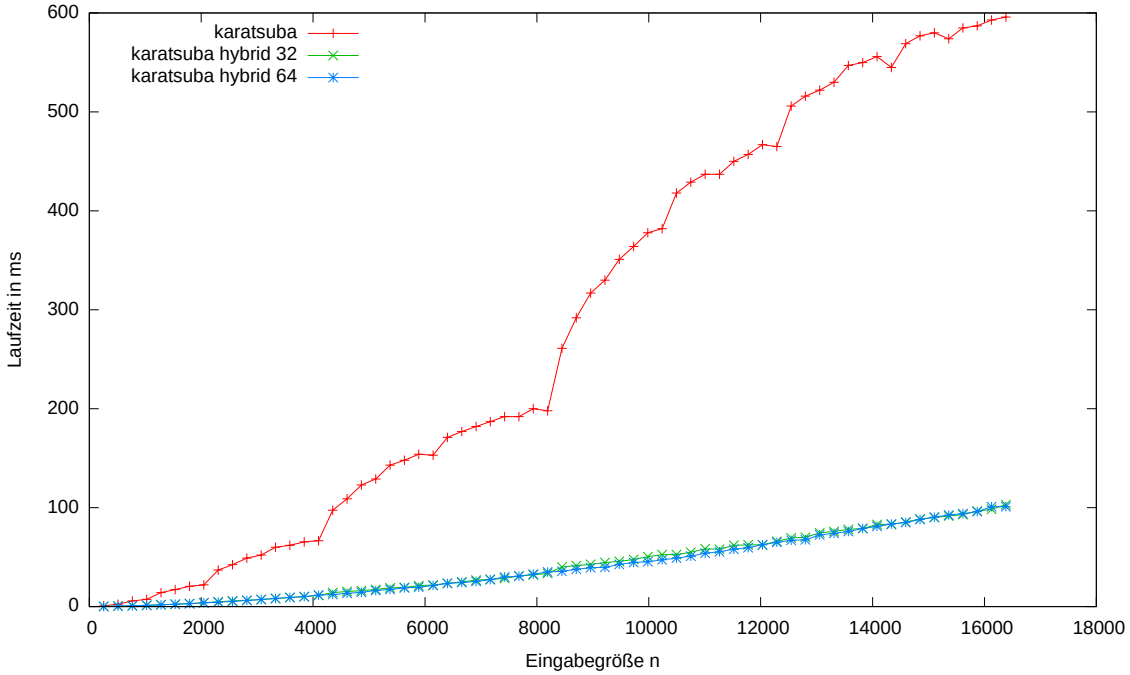

Abbildung 2.8: Gegenüberstellung des normalen Karatsuba und Karatsuba-Hybriden mit Schwellwert 32 und 64

- Unterstützung für drei Datentypen:
  float (32-bit), double (64-bit), long double (80-bit)
- Optimierung für spezielle Prozessorerweiterung:
  SSE/SSE2 (Intel), 3dNow (AMD) und Altivec (PowerPC)
- FFT-Transformationen für rein reelle Eingabedaten
- Beliebige Eingabelängen

Obwohl die FFTW beliebige Eingabelängen unterstützt, wurde bei Transformationen weiterhin auf die nächste Zweierpotenz aufgerundet. Der Grund dafür liegt in der Architektur der FFTW Bibliothek. Für jede Eingabelänge muss zunächst ein sogenannter Plan generiert werden. In einem Plan wird gespeichert, welche Algorithmen und Strategien miteinander kombiniert werden, um eine Transformation für die entsprechende Eingabelänge umzusetzen. Diese Pläne können beliebig oft wiederverwendet werden. Deren Generierung ist allerdings aufwendig und daher sollten die Pläne vorher berechnet oder zumindest zwecks Wiederverwendung zwischengespeichert werden.

In der Implementierung, die den folgenden Messungen zu Grunde liegt, wurden die Pläne vor den eigentlichen Messungen generiert. Durch die Beschränkung auf Zweierpotenzen

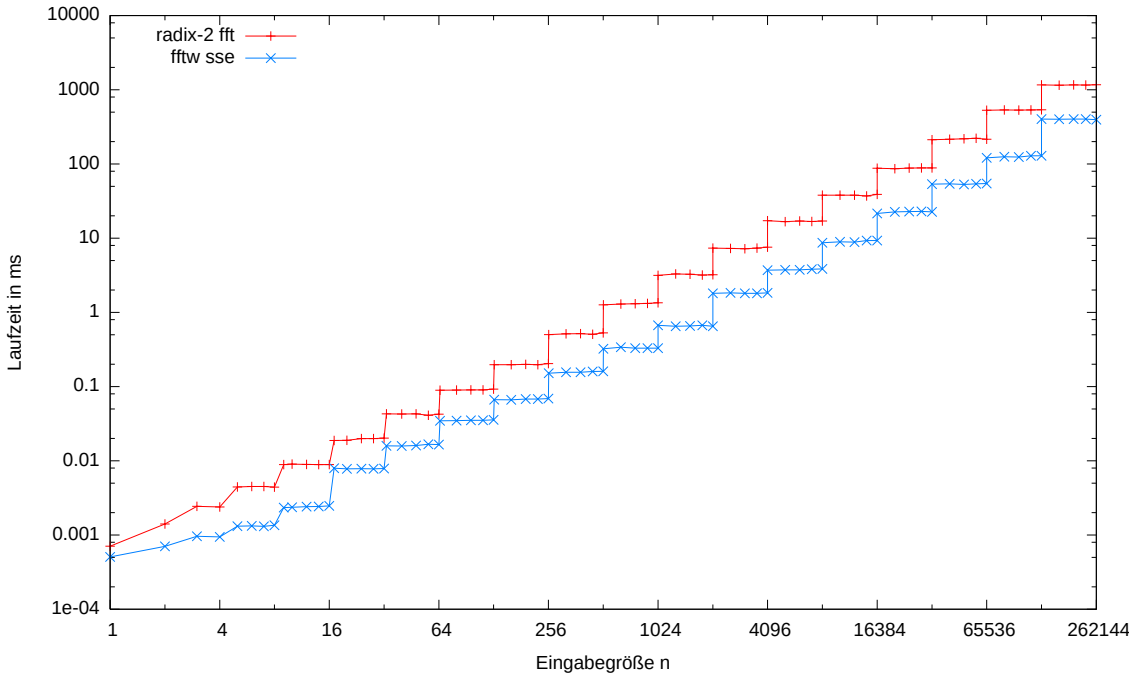

Abbildung 2.9: Komplexe Polynommultiplikation, Cooley Tukey FFT und hoch optimierte FFTW

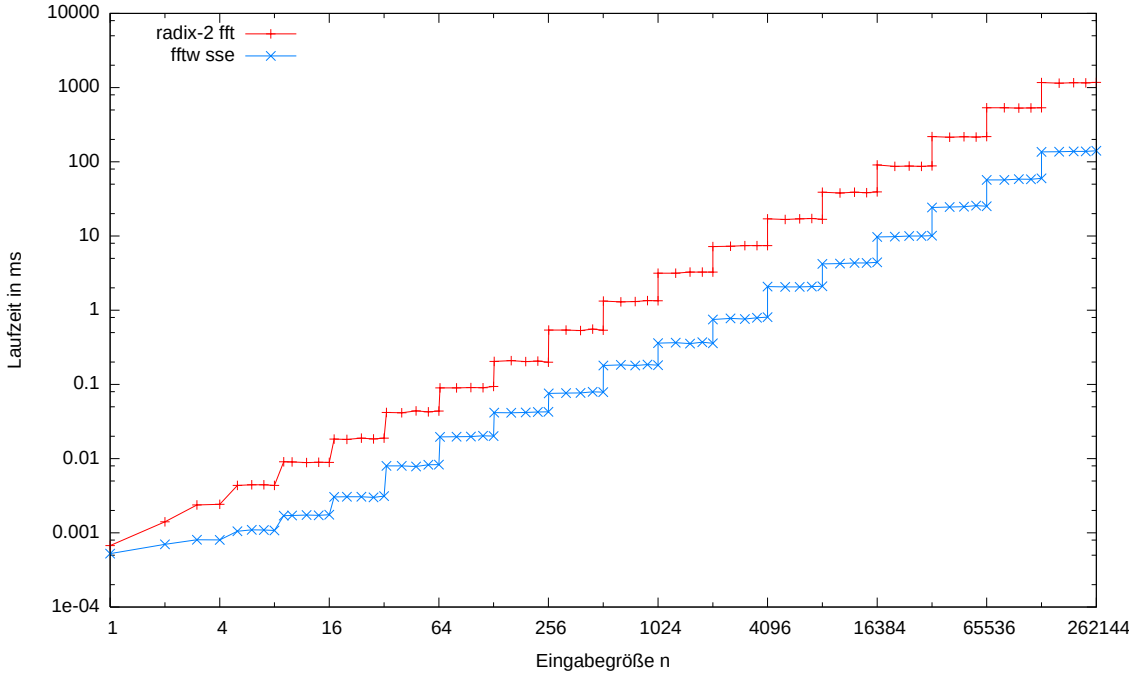

Abbildung 2.10: Reelle Polynommultiplikation, Cooley Tukey FFT und hoch optimierte FFTW

mussten für eine erwartete maximale Eingabelänge n nur $\mathcal{O}(\log(n))$-viele Pläne im Speicher gehalten werden.

Abbildung 2.9 zeigt die Ergebnisse für das Multiplizieren von Polynomen mit komplexen Koeffizienten. Durch das Aufrunden zur nächsten Zweierpotenz ergibt sich der typische Treppenverlauf. Im Vergleich mit der einfachen Cooley Tukey FFT benötigte die FFTW Bibliothek lediglich ein Viertel der Laufzeit.

Die FFTW beherrscht, wie oben erwähnt, auch das Transformieren reeller Eingaben. Für reelle Eingaben lässt sich zeigen, dass die Ausgabe eine bestimmte Symmetrie besitzt. Für die diskrete Fourier Transformierte $\hat{f} = DFT(f)$ eines reellen Vektors $f \in \mathbb{R}^n$ gilt:

$$\begin{aligned} \hat{f}_0 &= \overline{\hat{f}_0} \\ \hat{f}_k &= \overline{\hat{f}_{n-k}} \qquad k = 1 \ldots n-1 \end{aligned}$$

Die Elemente $\hat{f}_0, \ldots, \hat{f}_{n/2}$ bilden also den nicht-redundanten Teil der Ausgabe. Die anderen Elemente entsprechen lediglich dem komplex-konjugierten von $\hat{f}_{n/2-1}, \ldots, \hat{f}_1$. Die FFTW kann diese Redundanz ausnutzen. Man bekommt als Ausgabe einer Transformation von n reellen Werten nur noch $\frac{n}{2} + 1$ viele komplexe Sützwerte. Diese reichen aber aus, um ein reelles Polynom eindeutig zu repräsentieren. Durch diese Optimierung reduziert sich der Aufwand für das Transformieren von n reellen Koeffizienten auf die Hälfte.

Abbildung 2.10 zeigt nun diese speziell auf reelle Eingaben optimierte Version der FFT-basierten Multiplikation. Diese wird mit einer Version verglichen, die die Koeffizienten der reellen Polynome zuerst ins Komplexe konvertiert, um dann die einfache Cooley Tukey FFT benutzen zu können. Es zeigt sich, dass die FFTW hier durch diese spezielle Optimierung nur noch ein achtel (statt ein Viertel im Komplexen) der Laufzeit benötigt.

2.2.3 BreakEven-Points

Der Karatsuba wurde nun hybridisiert. Wie wir sehen werden, wurde er dadurch wieder konkurrenzfähig zur Schulmethode und auch zur FFT-basierten Multiplikation. Nun bleibt die anfangs schon gestellte Frage, für welche Eingabegrößen welches der drei Verfahren zum Einsatz kommen sollte. In diesem Abschnitt werden nun anhand der gemachten Laufzeitmessungen die BreakEven-Punkte dargestellt. Das sind die Eingabegrößen, für die ein Verfahren vom anderen eingeholt wird.

In Abbildung 2.11 ist der Karatsuba direkt im Vergleich zur Schulmethode zu sehen. Unterhalb des Schwellwerts 64 delegiert der Karatsuba Hybrid die Aufrufe direkt an die Schulmethode. Daher sind die Laufzeiten natürlich zueinander identisch. Positiv hervorzuheben ist allerdings das Verhalten des Karatsuba Hybriden jenseits des Schwellwerts.

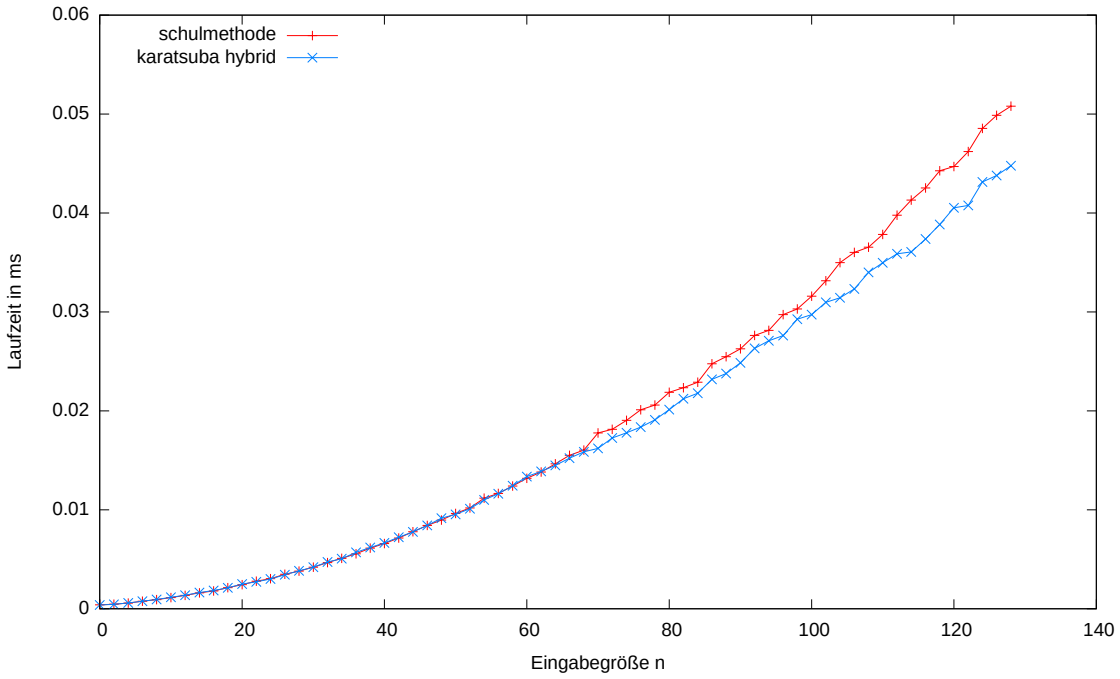

Abbildung 2.11: Gegenüberstellung Karatsuba Hybrid Schwellwert 32 mit Schulmethode

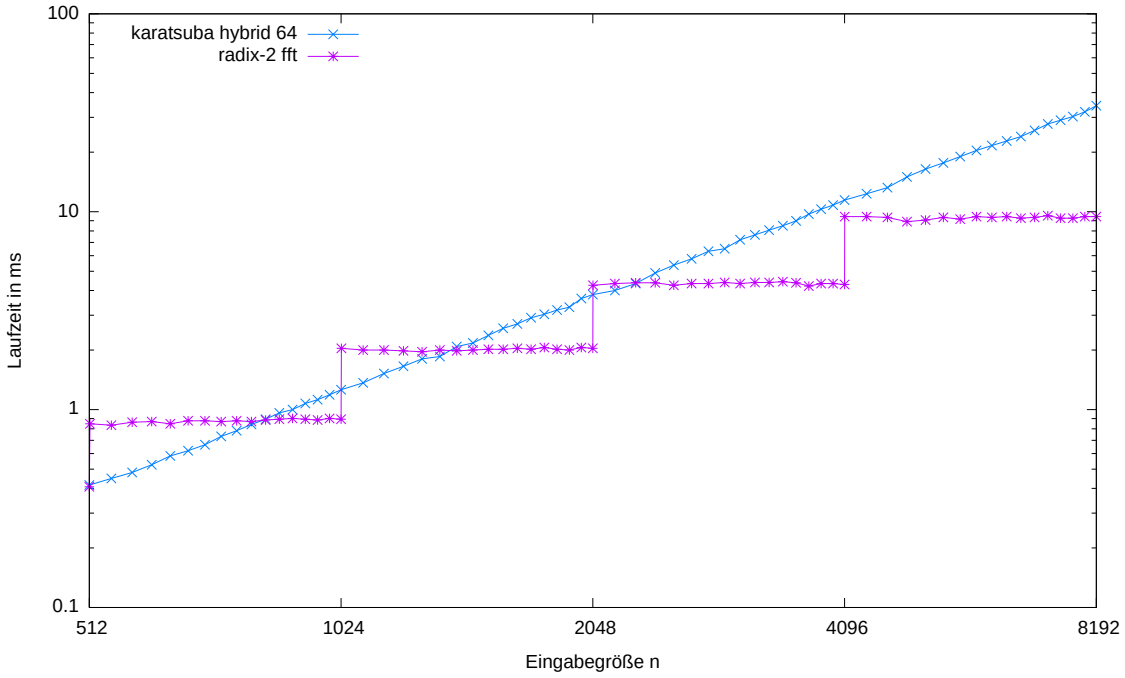

Abbildung 2.12: Gegenüberstellung Karatsuba Hybrid Schwellwert 32 mit normaler Cooley Tukey FFT

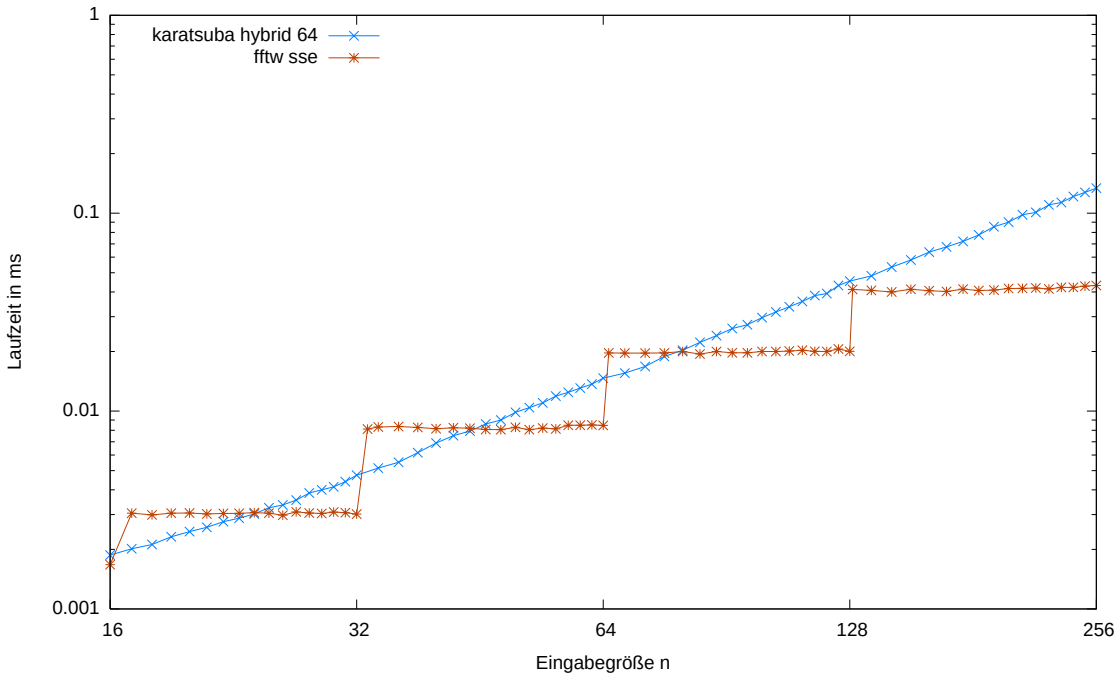

Abbildung 2.13: Gegenüberstellung Karatsuba Hybrid Schwellwert 32 mit hoch optimierter FFT

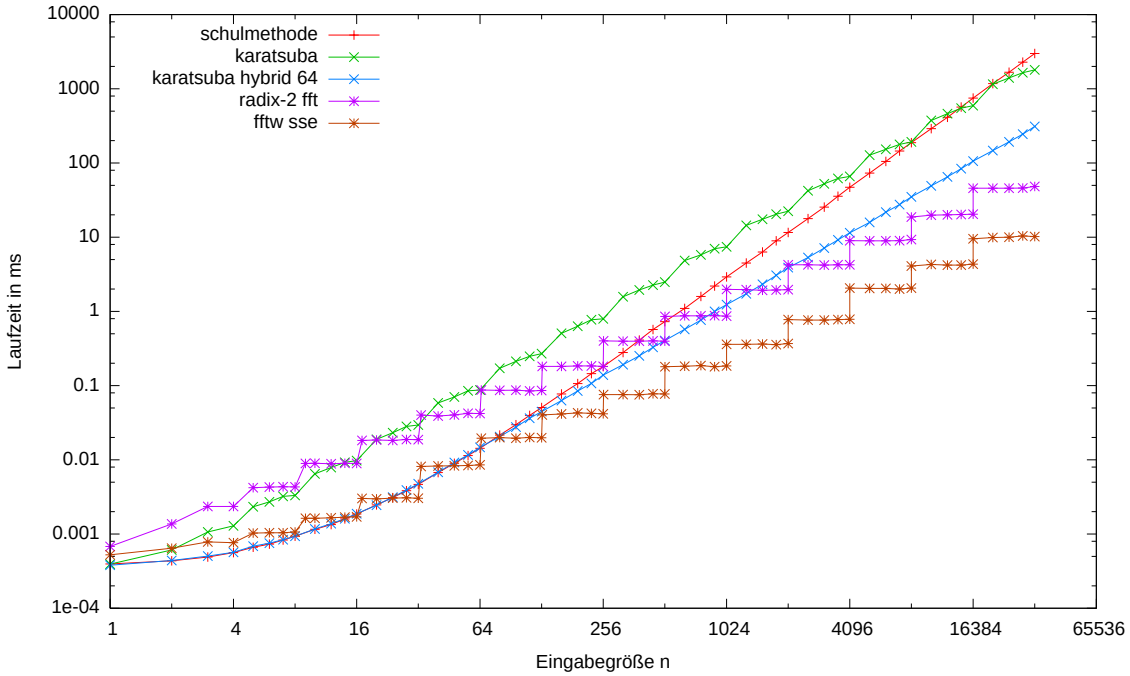

Abbildung 2.14: Laufzeit aller Multiplikationsalgorithmen im Überblick

Er setzt sich von der Schulmethode ab und bleibt konsequent schneller. Er zeigt dabei das charakteristische Laufzeitverhalten des normalen Karatsuba Algorithmus, wie man schön in Abbildung 2.14 sehen kann. Dort verlaufen die Kurven des normalen Karatsuba und des Hybriden quasi parallel. Der Karatsuba Hybrid kann also als Ersatz für die Schulmethode benutzt werden. Die folgenden Betrachtungen beschränken sich deshalb auf Vergleiche des Karatsuba Hybriden mit der FFT-basierten Methode.

Der Vergleich von Karatsuba und FFT-basierten Algorithmen gestaltet sich allerdings schwierig. Während der Karatsuba Hybrid eine relativ „glatte“ Laufzeitkurve aufweist, entsteht bei FFT-basierten Algorithmen eine Treppenkurve. Dies liegt daran, dass die vorliegenden Implementierungen sich auf FFT-Transformationen beschränken, deren Eingabelänge einer Zweierpotenz entsprechen. Die Sprünge ergeben sich dabei immer zwischen den Eingabelängen 2^n und 2^n+1. Bei den Laufzeitmessungen der FFT-basierten Multiplikation wurden die Eingabelängen $2^n + 1$ daher als zusätzliche Messstellen herangezogen.

In Abbildung 2.12 wird nun der Karatsuba Hybrid der einfachen Cooley-Tukey-FFT aus Abschnitt 2.1.2.1 gegenüber gestellt. In dem dargestellten Bereichen schneidet sich die Laufzeitkurve des Karatsuba Hybriden mehrere Male mit der der Kurve der FFT. Bei Eingabelänge 512 sind beide Algorithmen quasi gleich schnell. Die Kurve der FFT macht dann allerdings einen Sprung. Erst für die Eingabelängen 896 bis 1024 ist die FFT wieder schneller. Bei 1025 erfolgt wieder ein Sprung und erst bei ca. $n = 1408$ kann die FFT den Karatsuba Hybriden wieder einholen. Bei Eingabelänge 2049 erfolgt dann der letzte Sprung, durch den die FFT wieder ein wenig langsamer als der Karatsuba Hybrid wird. Ab Eingabelänge 2304 ist die FFT-basierte Methode dann endgültig schneller als der Karatsuba Hybrid.

Die hoch optimierte FFT übertrifft die Erwartungen. Wie man in Abbildung 2.13 sehen kann, ist sie bereits für $n = 16$ schneller als der Karatsuba Hybrid und damit auch schneller als die Schulmethode. Das typische Treppenverhalten ergibt dann, dass sie für Eingabelängen $n \in [17, 24]$, $n \in [33, 44]$ und $n \in [65, 80]$ langsamer ist, als der Karatsuba Hybrid. Für alle größeren Eingabelängen $n > 80$ ist sie dann durchgehend schneller.
Weitere Optimierungen sind allerdings möglich. In Abschnitt 2.2.2 wurde bereits beschrieben, dass die FFTW Bibliothek beliebige Eingabelängen beherrscht. Es muss aber immer erst ein Plan berechnet werden. Nimmt man also konstant viel zusätzlichen Speicherbedarf in Kauf, so könnte man FFTW-Pläne für $n = 16, \ldots, 80$ vorberechnen. Die Treppencharakteristik würde dann für diesen Bereich entfallen. Die FFTW-basierte Multiplikation würde die Schulmethode dann schon ab $n = 16$ oder spätestens ab $n = 32$ ablösen.

Einen Teil ihres Geschwindigkeitsvorteils gewinnt die FFTW durch hoch optimierte, in Assembler geschriebene Routinen und wenn verfügbar auch spezielle Prozessorweiterungen. Es ist eventuell möglich, diese Optimierungen auch auf den Karatsuba Algorithmus oder sogar auf die Schulmtehode zu übertragen.

Zusammenfassend kann man sagen, dass es sich lohnt, den Karatsuba Algorithmus zu hybridisieren. Der Hybrid kombiniert die niedrige Laufzeit der Schulmethode für kleine Eingaben mit gutem asymptotischen Verhalten. Der Karatsuba-Ansatz lohnt sich also erst für Eingabelängen jenseits des optimalen Hybrid-Schwellwerts (in diesem Fall 64). Eine einfach Cooley-Tukey-FFT lohnt sich dagegen erst für $n \in [1408, 2048]$ und dann für alle $n \geq 2304$.
Die hoch optimierte FFT macht den Karatsuba fast überflüssig. Dazu kommt, dass die FFT häufig Anwendung findet (z.B. auch in der Signalverarbeitung) und somit die Wahrscheinlichkeit recht hoch ist, dass man speziell auf die Hardware optimierte FFT-Bibliotheken nutzen kann.

2.3 Division mit Rest

Neben einer schnellen Multiplikation benötigen wir im nachfolgenden Abschnitt auch eine schnelle Polynomdivision. Der dort erklärte Algorithmus zur schnelle Mehrfachauswertung eines Polynoms käme ohne diese nicht auf subquadratische Laufzeit. Mit der aus der Schule bekannten Polynomdivision steht uns aber bislang nur ein langsame Variante mit quadratischer Laufzeit zur Verfügung. Im Folgenden wird nun ein Algorithmus zur schnellen Division beschrieben. Er basiert im wesentlichen auf einer Newton-Iteration.

Betrachten wir die Polynome $f(x) := f_0 + f_1 x + \ldots + f_{n-1}x^{n-1} \in \mathbb{R}[x]$ und $g(x) := g_0 + g_1 x + \ldots + g_{m-1}x^{m-1} \in \mathbb{R}[x]$, mit $n, m \in \mathbb{N}$. Zusätzlich soll $g_{m-1} \neq 0$ sowie $m \leq n$ gelten. Das Polynom f ist also vom Grad kleiner n und das Polynom g hat exakt den Grad $m-1$. Desweiteren sei $q = f \operatorname{div} g$ das Quotientenpolynom vom Grad kleiner $n-m+1$ und das Prolynom $r = f \operatorname{rem} g$ vom Grad kleiner $m-1$ der Rest der Division. Dann gilt die folgende Gleichung:

$$f(x) = q(x)g(x) + r(x)$$

In dieser Gleichung wird nun x durch $\frac{1}{x}$ substituiert. Eine zusätzliche Multiplikation mit x^{n-1} ergibt dann:

$$x^{n-1} f\left(\frac{1}{x}\right) = x^{n-m} q\left(\frac{1}{x}\right) \cdot x^{m-1} g\left(\frac{1}{x}\right) + x^{n-m+1}\left(x^{m-2} r\left(\frac{1}{x}\right)\right) \tag{2.1}$$

Um den obigen Ausdruck zu vereinfachen, wird die Umkehrung („reversal“) eines Polynoms a wie folgt definiert:

$$\mathrm{rev}_k(a) = x^{k-1} a\left(\frac{1}{x}\right) \qquad \text{für } k \in \mathbb{N}$$

Wenn das Polynom a vom Grad kleiner k ist, gilt außerdem:

$$\mathrm{rev}_k(a) = a_{k-1} + a_{k-2}x + a_{k-3}x^2 + \ldots + a_0 x^{k-1}$$

Es wird also quasi die Reihenfolge der k Koeffizienten des Polynoms a umgedreht.

Das Polynom f ist nach Voraussetzung vom Grad kleiner n, und das Polynom g ist vom Grad kleiner m. Auch von den Polynomen q und r ist bekannt, dass q vom Grad kleiner $n-m+1$ sowie r vom Grad kleiner $m-1$ sein muss. Gleichung 2.1 lässt sich daher wie folgt schreiben:

$$\mathrm{rev}_n(f) = \mathrm{rev}_{n-m+1}(q) \cdot \mathrm{rev}_m(g) + x^{n-m+1}\,\mathrm{rev}_{m-1}(r)$$

Daraus folgt:

$$\begin{aligned} & \mathrm{rev}_n(f) \equiv \mathrm{rev}_{n-m+1}(q) \cdot \mathrm{rev}_m(g) \mod x^{n-m+1} \\ \Rightarrow \quad & \mathrm{rev}_{n-m+1}(q) \equiv \mathrm{rev}_n(f) \cdot \mathrm{rev}_m(g)^{-1} \mod x^{n-m+1} \end{aligned}$$

Das gesuchte Polynom q lässt sich durch erneue Umkehrung seiner Koeffizienten berechnen. Es gilt $q = \mathrm{rev}_{n-m+1}(\mathrm{rev}_{n-m+1}(q))$.

Das Polynom $q = f \operatorname{div} g$ kann also im Wesentlichen mit einer Invertierung und einer Multiplikation modulo x^{n-m+1} bestimmt werden. Durch die Voraussetzung $g_{m-1} \neq 0$ ist sichergestellt, dass das Polynom $\mathrm{rev}_m(g)$ modulo x^{n-m+1} invertierbar ist.
Anhand der Gleichung $r = f - q \cdot g$ lässt mit einer weiteren Multiplikation nun auch das Polynom $r = f \operatorname{rem} g$ bestimmen.

Das Problem der Invertierung wird durch Newton-Iteration gelöst. Dazu muss die Invertierung zuerst in ein Nullstellen-Problem umgeformt werden. Zu einem Polynom $f \in \mathbb{R}[x]$ und einem $l \in \mathbb{N}$ soll ein Polynom g gefunden werden, so dass gilt:

$$f \cdot g \equiv 1 \mod x^l$$

Die Gleichung kann zu folgendem Nullstellenproblem umgeformt werden:

$$\varphi(g) \equiv 0 \mod x^l \qquad \text{mit } \varphi(g) := \frac{1}{g} - f$$

Die Newton-Iteration erzeugt eine Folge (a_n) von Polynomen. Als Startwert wird das konstante Polynom $a_0 := 1/f(0)$ gewählt. Die weiteren Folgenglieder ergeben sich dann durch den folgenden Newton-Schritt:

$$\begin{aligned} a_{i+1} &:= a_i - \frac{\varphi(a_i)}{\varphi'(a_i)} \\ &= a_i - \frac{1/a_i - f}{-1/a_i^2} \\ &= 2a_i - f a_i^2 \end{aligned}$$

Der folgende Algorithmus führt die obigen Betrachtungen zusammen und berechnet das Inverse des Polynoms f modulo x^l, $l \in \mathbb{N}$:

- Setze $r := \lceil \log(l) \rceil$ und $a_0 := \dfrac{1}{f(0)}$
- Für $i = 1, 2, \ldots, r$ berechne:
  $a_i := (2a_{i-1} - f a_{i-1}^2) \operatorname{rem} x^{2^i}$

Es gilt dann $f a_r \equiv 1 \mod x^l$. In [GG03, Theorem 9.2] wird außerdem gezeigt, dass sich mit dem Newton-Schritt die Anzahl der korrekten Koeffizienten verdoppelt. Es gilt also stets:

$$f \cdot a_i \equiv 1 \mod x^{2^i} \qquad \forall i \in \mathbb{N}$$

Daraus folgt sofort die Korrektheit des obigen Algorithmus.
Die Operation $\operatorname{rem} x^{2^i}$ kann durch einfaches „Abschneiden“ von Koeffizienten implementiert werden. Diese Operation bedarf keines Algorithmus zur Polynomdivision.

Satz 2.4. *Sei $M(n)$ die Laufzeit der Multiplikation zweier Polynom vom Grad kleiner n. Dann berechnet der obige Algorithmus das Inverse eines Polynoms vom Grad kleiner n in Laufzeit $\mathcal{O}(M(n))$.*

Beweis. Für die Laufzeit der Polynommultiplikation gilt $M(n) \in \Omega(n)$. Daraus folgt, dass eine Konstante $c \in \mathbb{N}$ existiert, so dass für alle $n > c$ gilt:

$$2M(n) \leq M(2n)$$

Der Algorithmus führt nun bei jeder Iteration drei Polynommultiplikation und eine Subtraktion durch. Die Polynome sind dabei vom Grad immer kleiner als 2^i. Insgesamt werden $r = \lceil \log(l) \rceil$ Iterationen durchgeführt. Es ergibt sich also folgende Laufzeit:

$$\begin{aligned}\sum_{i=1}^{r}(3M(2^i)+2^i) &\leq 3M(2^r)\sum_{i=1}^{r}2^{i-r}+\sum_{i=1}^{r}2^i \\ &\leq 6M(2^r)+2^{r+1} \\ &\leq 6M(2n)+4n \\ &\in \mathcal{O}\left(M(n)\right)\end{aligned}$$

□

Der folgende Algorithmus, berechnet die Polynome $q = f \operatorname{div} g$ und $r = f \operatorname{rem} g$. Er besteht, wie am Anfang dieses Abschnitts schon beschrieben wurde, im Wesentlichen nur noch aus der Invertierung und zwei Multiplikationen. Das Polynom f sei vom Grad kleiner $n \in \mathbb{N}$ und das Polynom g habe den Grad $m-1$, $m \in \mathbb{N}$.

- Falls $n < m$
  - Setze $q := 0$ und $r = f$
- Sonst:
  - Berechne $g^* := \mathrm{rev}_m(g)^{-1}$ modulo x^{n-m+1}
  - Berechne $q^* := \mathrm{rev}_n(f) \cdot g^* \ \operatorname{rem} x^{n-m+1}$
  - Berechne $q := \mathrm{rev}_{n-m+1}(g^*)$
  - Berechne $r := f - q \cdot g$

Satz 2.5. *Sei $M(n)$ die Laufzeit der Multiplikation zweier Polynom vom Grad kleiner n. Dann berechnet der obige Algorithmus die Division mit Rest zweier Polynome f und g vom Grad kleiner n in Laufzeit $\mathcal{O}\left(M(n)\right)$.*

Beweis. Für den Fall $n < m$ ist nur linearer Aufwand nötig. Das Polynom q muss 0 gesetzt werden, und das Polynom f muss in das Polynom r kopiert werden.

Für den anderen Fall $m \leq n$ sind alle auftretenden Polynome vom Grad kleiner n. Es werden mehrere Umkehrungen durchgeführt. Deren Laufzeit ist also linear in n. Weiterhin werden eine Invertierung und zwei Multiplikation durchgeführt. Mit $M(n) \in \Omega\left(n\right)$ und Satz 2.4 ergibt sich dann die Laufzeit:

$$2M(n) + \mathcal{O}\left(M(n)\right) + \mathcal{O}\left(n\right) \in \mathcal{O}\left(M(n)\right)$$

□

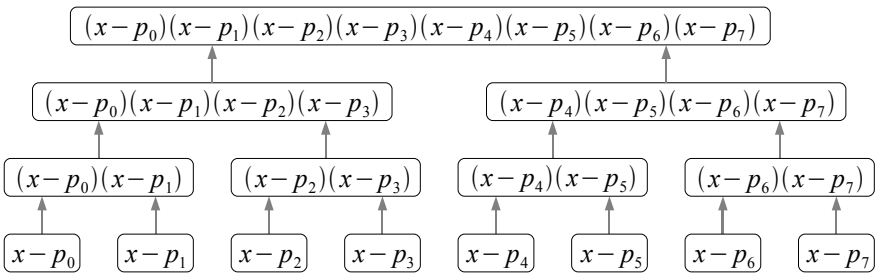

Abbildung 2.15: Ein Produktbaum für $n = 8$

2.4 Mehrfachauswertung

Mit Hilfe der schnellen Polynommultiplikation und -division kann man nun einen divide-and-conquer Algorithmus entwerfen, der ein Polynom vom Grad kleiner n in subquadratischer Laufzeit an n Stellen gleichzeitig auswertet. [GG03, Algorithm 10.7]

Wir betrachten das Polynom $f = f_0 + f_1 x + \ldots + f_{n-1}x^{n-1}$. Dieses soll nun an den n Stellen $p_0, \ldots, p_{n-1}$ ausgewertet werden. Es sollen also die Werte $f(p_0), \ldots, f(p_{n-1})$ berechnet werden.

Um dieses Problem zu lösen, wird zuerst ein Produktbaum aufgebaut. Dieser Produktbaum ist ein vollständiger binärer Baum der Tiefe $t = \lceil \log(n) \rceil$. Ein solcher Baum besitzt genau 2^t-viele Blätter. Zuerst werden nun n Polynome der Form $x - p_i$ (für $i = 0, \ldots, n-1$) „gleichmäßig“ auf die 2^t Blätter verteilt. Die restlichen $2^t - n$ Blätter werden mit dem konstanten Polynom 1 gefüllt. Der Rest des Baumes wird dann von den Blättern zur Wurzel hin aufgebaut, indem jedem Knoten das Produkt seiner Kinder zugewiesen wird. Entsprechende Bäume werden in Abbildung 2.15 und Abbildung 2.16 gezeigt. Das Polynom in der Wurzel wird allerdings für die Mehrfachauswertung nicht benötigt.

Beobachtung 2.6. *Gegeben zwei Polynome f und $g \neq 0$. Die Polynome q und r seien die Ergebnisse der Polynomdivision f/g so dass $f = gq + r$ gilt. Sei weiterhin x_0 eine Nullstelle von g. Dann gilt:*

$$\begin{aligned} & g(x_0) = 0 \\ \Rightarrow \quad & f(x_0) = g(x_0)q(x_0) + r(x_0) \\ & \qquad = 0q(x_0) + r(x_0) \\ & \qquad = r(x_0) \end{aligned}$$

Mit Hilfe von Beobachtung 2.6 lässt sich nun erklären, wie das Problem der Auswertung eines Polynoms f an n Stellen in kleinere Probleme zerschlagen werden kann. Und zwar

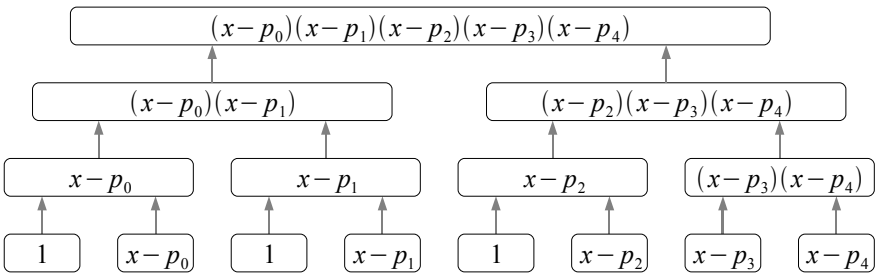

Abbildung 2.16: Ein Produktbaum für $n = 5$

wird das Polynom f durch geeignete Polynome dividiert, so dass kleinere Polynome entstehen, die anstelle von f ausgewertet werden können.

Dieses Vorgehen bildet den zweiten Teil des Algorithmus zur schnellen Mehrfachauswertung:

- Wähle $m := \left\lfloor \frac{n}{2} \right\rfloor$
- Setze $g_0 := \prod_{i=0}^{m-1}(x - p_i)$ und $g_1 := \prod_{i=m}^{n-1}(x - p_i)$
- Berechne $r_0 = f \operatorname{rem} g_0$ und $r_1 = f \operatorname{rem} g_1$
  ($a \operatorname{rem} b$ bezeichnet den Rest der Division a/b)
- Berechne $r_0(p_0), \ldots, r_0(p_{m-1})$ und $r_1(p_m), \ldots, r_0(p_{n-1})$

Die beiden Polynome g_0 und g_1 müssen dabei nicht extra berechnet werden. Sie können dem entsprechend aufgebauten Produktbaum entnommen werden und sind genau so konstruiert, dass ihre Nullstellen den Auswertungspunkten $p_0, \ldots, p_{n-1}$ entsprechen. Nach Beobachtung 2.6 kann man daher die Polynome r_0 und r_1 stellvertretend für f auswerten.
Desweiteren beträgt der Grad der Polynome g_0 und g_1 genau $m = \left\lfloor \frac{n}{2} \right\rfloor$ und $n - m = \left\lceil \frac{n}{2} \right\rceil$. Daraus folgt, dass die durch die Divisionen entstehenden Restpolynome r_0 und r_1 vom Grad her kleiner m bzw. kleiner $n - m$ sind.

Die Polynome r_0 und r_1 werden dann rekursiv mit Hilfe des obigen Algorithmus ausgewertet. Durch die Rekursion werden die Polynome wieder und wieder zerlegt, bis sich das Problem auf die Auswertung eines konstanten Polynoms reduziert.

Satz 2.7. *Sei $M(n)$ die Laufzeit der Multiplikation zweier Polynome vom Grad kleiner n. Dann wertet die schnelle Mehrfachauswertung ein Polynom vom Grad kleiner n an n-vielen Stellen in Laufzeit $\mathcal{O}(\log(n)M(n))$ aus.*

Beweis. Für die Laufzeit der Polynommultiplikation gilt $M(n) \in \Omega(n)$. Daraus folgt, dass ein $c \in \mathbb{N}$ existiert, so dass für alle $n > c$ gilt:

$$2M(n) \leq M(2n)$$

Beim Aufbau des Produktbaums der Tiefe $t = \lceil \log(n) \rceil$ werden nun für jede Ebene $i < t$ des Baumes 2^i-viele Multiplikationen von jeweils zwei Polynomen vom Grad kleiner 2^{t-i} durchgeführt. Dies ergibt dann folgende Laufzeit:

$$\begin{aligned}\sum_{i=0}^{t-1} 2^i M(2^{t-i}) &\leq \sum_{i=0}^{t-1} M(2^t) \\ &= t \cdot M(2^t) \\ &\leq (\log(n) + 1) M(2n) \\ &\in \mathcal{O}(\log(n) M(n))\end{aligned}$$

Für die Laufzeit $D(n)$ der schnellen Polynomdivision gilt wieder $D(n) \in \Omega(n)$, und somit existiert ebenfalls eine Konstante c', so dass für alle $n > c'$ gilt:

$$2D(n) \leq D(2n)$$

Nach Satz 2.5 gilt außerdem:

$$D(n) \in \mathcal{O}(M(n))$$

Im zweiten Teil des Algorithmus werden nun pro Ebene $i > 0$ des Baumes 2^i Polynomdivisionen von Polynomen vom Grad kleiner 2^{t-i+1} durchgeführt. Das ergibt folgende Laufzeit:

$$\begin{aligned}\sum_{i=1}^{t} 2^i D(2^{t-i+1}) &\leq \sum_{i=1}^{t} D(2^{t+1}) \\ &= t \cdot D(2^{t+1}) \\ &\in \mathcal{O}(\log(n) M(n))\end{aligned}$$

□

3 Übersicht über Programmbibliotheken

Bevor die hier vorgestellten Algorithmen implementiert wurden, stellte sich die Frage, in wie weit diese Verfahren bereits Verbreitung gefunden habe. Zu diesem Zweck wurden mehrere Bibliotheken untersucht.

Im Folgenden wird ein Überblick über die untersuchten Bibliotheken und ihren Fähigkeiten in Bezug auf reelle Polynomarithmetik gegeben.

3.1 GMP

http://gmplib.org/

Die „**G**NU **M**ultiple **P**recision Arithmetic Library“ (kurz GMP) stellt Arithmetik für beliebig große Ganzzahl- und Fließkommazahlen zur Verfügung. Die Entwickler von GMP legen dabei großen Wert auf Geschwindigkeit. Daher benutzt GMP neben der Schulmethode auch die auf Binärzahlen übertragenen Varianten der bereits vorgestellten Algorithmen Karatsuba und FFT.
Zusätzlich wird ein „Toom 3-Way“ genannter Ansatz mit Laufzeit $\mathcal{O}\left(n^{1.465}\right)$ verwendet. Dieser Ansatz ist eng mit dem Karatsuba-Algorithmus verwandt. Die zu multiplizierenden Binärzahlen werden in drei statt zwei Teile geteilt, um die Multiplikation dann geschickt durchzuführen. Der Algorithmus stellt wie auch der Karatsuba einen Spezialfall des Toom-Cook-Algorithmus dar.

Die Bibliothek bietet aber keine Datenstrukturen oder Algorithmen für Polynomarithmetik.

3.2 NTL

http://shoup.net/ntl/

Die NTL ist eine Bibliothek von Victor Shoup. Sie bietet unter anderem Arithmetik für Fließkomma- und Ganzzahlen beliebiger Größe und weiterer Ringe wie $\mathbb{Z}_2, \mathbb{Z}_p$. Die Implementierung der NTL kann dabei optional auf die schnellen Routinen der Bibliothek

GMP zurückgreifen.
Es können auch eindimensionale Polynome gebildet werden. Die Polynomarithmetik wird vom Author explizit gelobt, da die hier vorgestellen asymptotisch schnelle Karatsuba und FFT-basierte Algorithmen verwendet werden.

Reelle Polynomarithmetik wird allerdings nicht unterstützt.

3.3 CLN

http://www.ginac.de/CLN/

In der „**C**lass **L**ibrary for **N**umbers“ (kurz CLN) stehen Ganzzahlen und Fliesskommazahlen beliebiger Größe sowie rationale und komplexe Zahlen zur Verfügung. Die Bibliothek kann dabei ebenfalls auf die schnellen Algorithmen der GMP zugreifen. Desweiteren können mit all diesen Datentypen eindimensionale Polynome gebildet werden.

Die Polynomarithmetik greift aber meines Wissen nach nicht auf die asymptotisch schnellen Algorithmen wie Karatsuba oder FFT zurück.

3.4 LiDIA

http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/

Die Bibliothek LiDIA stellt ebenfalls Arithmetik für beliebig große Ganz- und Fliesskommazahlen sowie rationale und komplexe Zahlen zur Verfügung. LiDIA kann dabei auf GMP, CLN oder eigene Routinen zurückgreifen.
Eindimensionale Polynome werden ebenfalls unterstützt. Die Polynomarithmetik greift dabei über endlichen Körpern zu schnelleren Algorithmen wie Karatsuba und FFT.

Bei Polynomen über den Fliesskomma- oder Ganzzahlen wird allerdings die langsame Schulmethode angewendet.

3.5 GiNaC

http://www.ginac.de/

GiNaC ist eine C++-Bibliothek, die viele Fähigkeiten moderner Computer-Algebra-Systeme mitbringt. Es kann also unter anderem symbolisch grechnet werden. GiNaC baut zu großen Teilen auf der oben erwähnten Bibliothek CLN auf. Diese wurde vom selben Author entwickelt.

Auch GiNaC stellt keine schnelle Polynomarithmetik zur Verfügung.

3.6 MuPAD

http://www.mupad.de/

MuPAD ist ein Computer-Algebra-System. Es wird überwiegend symbolisch gerechnet. Zusätzlich stehen Ganz- und Fliesskommazahlen beliebiger Größe zur Verfügung. Welche Algorithmen für die Ganz- und Fliesskommazahlen verwendet werden, ist nicht bekannt. Für Polynome stellt MuPAD einen separaten Datentyp zur Verfügung. Die Polynomarithmetik wurde optimiert, in dem sie im Kern implementiert wurde. Asymptotisch schnelle Algorithmen werden aber nicht verwendet.

3.7 Maple

http://www.maplesoft.com/

Maple ist ein weiteres Computer-Algebra-System. Es stehen ebenfalls Ganz- und Fliesskommazahlen beliebiger Größe zur Verfügung. Für die Multiplikation großer Ganzzahlen wird ab Version 5 der Karatsuba Algorithmus verwendet. Seit Version 7 wird ein Algorithmus mit gleicher Komplexität auch für die Division eingesetzt. Und seit Version 9 wird die schnelle GMP Bibliothek für Ganzzahlen benutzt.

Schnelle Polynomarithmetik wird allerdings nicht zur Verfügung gestellt.

3.8 MATLAB

http://www.mathworks.de/

Das eher numerisch orientierte System MATLAB basiert überwiegend auf Fliesskommaarithmetik mit herkömmlichen 64-Bit Fliesskommazahlen. Polynome werden durch Vektoren repräsentiert.

Die Polynommultiplikation wird hier explizit durch eine FFT-basierte Lösung realisiert.

3.9 Zusammenfassung

Es gibt viele Bibliotheken, die die asymptotisch schnellen Algorithmen zumindest für die Flieskomma- und Ganzzahlen einsetzen. Dazu wird sehr oft auf die GMP Bibliothek zurückgegriffen. Außer MATLAB bietet keine andere Bibliothek die asymptotisch schnellen Verfahren für reelle Polynome an.

Asymptotisch schnelle Polynomarithmetik über $\mathbb{R}$ oder $\mathbb{C}$ scheint nicht verbreitet zu sein. Dies scheint grundsätzliche Gründe zu haben, die im folgenden Kapitel untersucht werden.

4 Numerische Stabilität

Dieses Kapitel widmet sich der Praktikabilität schneller Polynomarithmetik unter dem Gesichtspunkt der numerischen Stabilität. Dazu soll zunächst die in Abschnitt 2.4 beschriebene schnelle Mehrfachauswertung herangezogen werden. Anhand des Spezialfalls der Polynompotenzierung werden anschließend Probleme im Kern der schnellen Mehrfachauswertung illustriert. Mit der iRRAM wird dann abschließend eine Bibliothek betrachtet, die „exakte reelle Arithmetik in C++“ erlaubt und damit über numerische Probleme hinweghelfen kann.

Die Tests der schnellen Mehrfachauswertung haben gezeigt, dass der Algorithmus extrem unpraktikabel ist. Während für kleine n das Ergebnis exakt dem erwarteten entspricht, wächst der Fehler für große n rasant. Abbildung 4.1 sowie Abbildung 4.2 zeigen die auftretenden relativen Fehler $\left(\frac{\Delta y}{y}\right)$ in Abhängigkeit zu $n \in \mathbb{N}$. Die Fehler-Achse ist logarithmisch gewählt. Zu jedem n wurde das Polynom $f_n(x) := \sum_{i=0}^{n-1} x^i$ mit dem Algorithmus aus Abschnitt 2.4 n-mal an der Stelle $x = 1$ und der Stelle $x = 0.125$ ausgewertet. Für das Polynom f_n gilt:

$$f_n(x) = \begin{cases} n & \text{für } x = 1 \\ \frac{x^n - 1}{x - 1} & \text{für } x \neq 1 \end{cases}$$

Die Werte des Polynoms an den Stellen $x = 1$ und $x = 0.125$ verhalten sich also sehr gutartig. Insbesondere $f_n(0.125)$ konvergiert für wachsendes n schnell gegen $\frac{8}{7}$. Die Diagramme geben daher auch indirekt Aufschluss über den absoluten Fehler. Die Division durch das jeweilige erwartete Ergebnis (n bzw. $\frac{8}{7}$) bei der Berechnung des relativen Fehlers fällt bei dieser Fehlergröße kaum ins Gewicht.

Die Diagramme zeigen nun fünf verschiedene Konstellationen:

- Horner-Schema
- Schulmethode
- Karatsuba
- Karatsuba Hybrid, Schwellwert 64
- Cooley Tukey FFT

Das Horner-Schema steht hier für eine simple Einzelauswertung des Polynoms f_n. Der Algorithmus aus Abschnitt 2.4 wurde hier noch nicht benutzt.

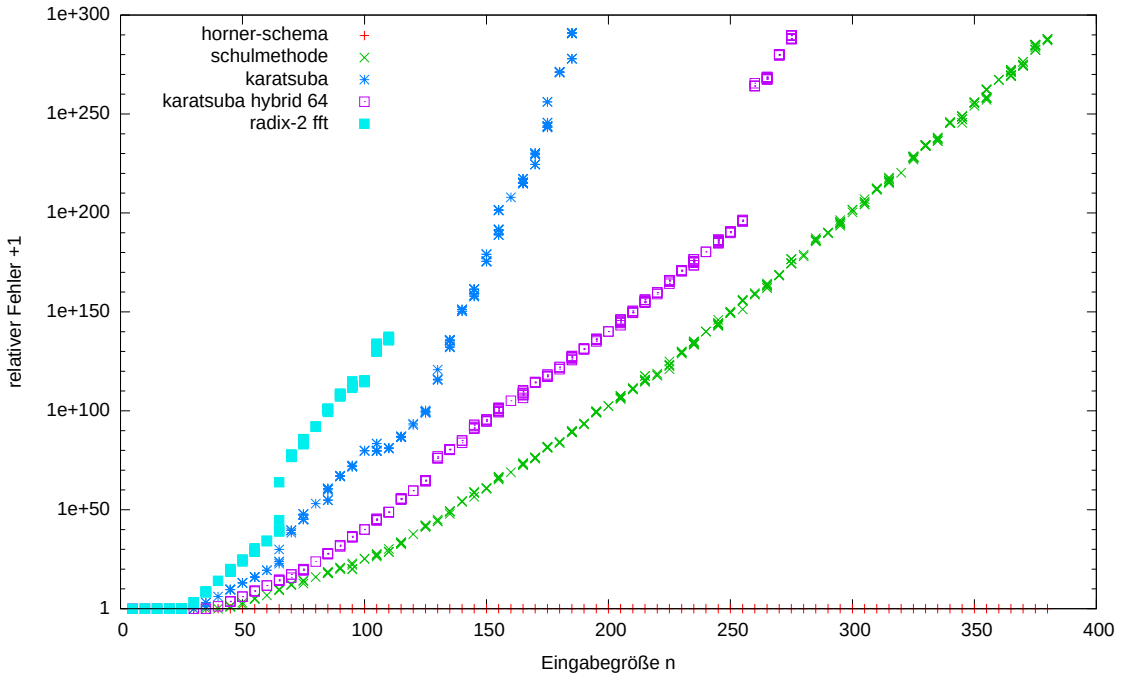

Abbildung 4.1: Relativer Fehler beim Auswerten von $f_n(x) = \sum_{i=0}^{n-1} x^i$ an $x = 1$

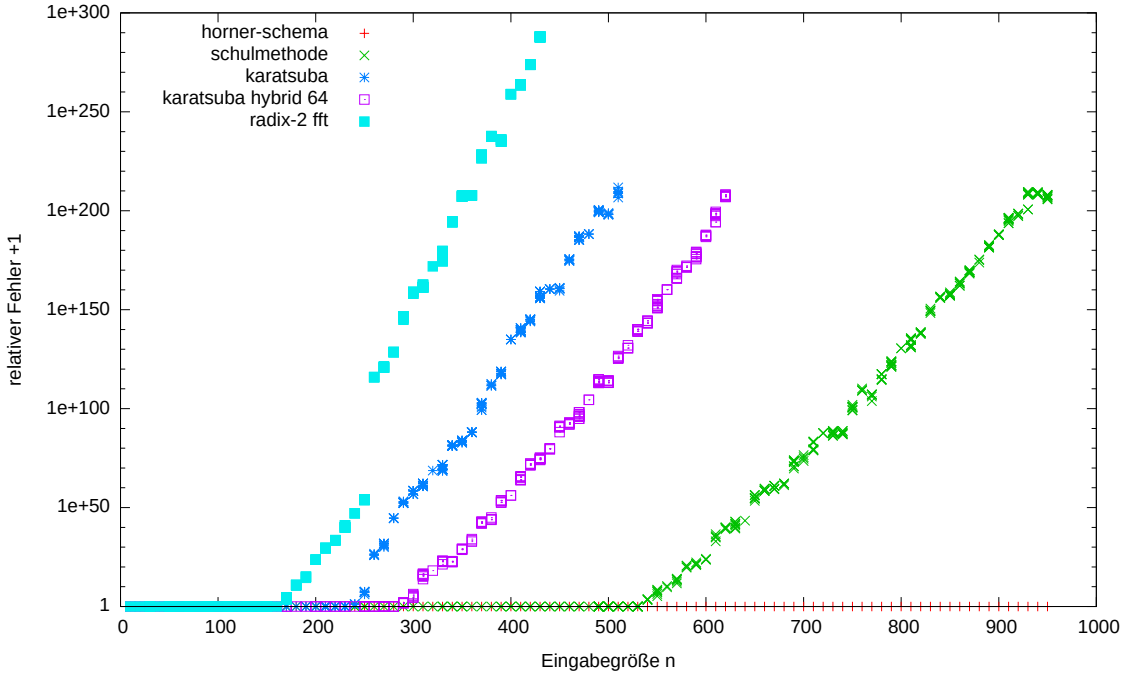

Abbildung 4.2: Relativer Fehler beim Auswerten von $f_n(x) = \sum_{i=0}^{n-1} x^i$ an $x = 0.125$

Unter „Schulmethode“ ist nun zu verstehen, dass der in Abschnitt 2.4 beschriebene Algorithmus zusammen mit einer Polynommultiplikation und einer Polynomdivision betrieben wird, wie sie aus der Schule bekannt ist.
In den anderen Fällen arbeitet die schnelle Mehrfachauswertung mit den entsprechenden Multiplikationsalgorithmen (Karatsuba, Karatsuba Hybrid, FFT) und der darauf über Newton-Iteration aufgesetzten Division, wie sie in Abschnitt 2.3 beschrieben ist.

Abbildung 4.1 zeigt also die Fehler bei der schnellen n-maligen Auswertung an der Stelle $x = 1$. Das Diagramm zeigt einen stark wachsenden Fehler. Während die Schulmethode noch für $n = 40$ das korrekte Ergebnis liefert, liefert sie für $n = 45$ schon das 16-fache vom korrekten Ergebnis ($f_n(1) = n$). Die Schulmethode wartet zwar erwartungsgemäß mit den kleinsten Fehlern auf, allerdings ist der Fehler wie bei allen anderen Verfahren viel zu gross. Der praktische Einsatz ist damit quasi ausgeschlossen.
Die Karatsuba-basierte Lösung liefert nur bis ca. $n = 30$ brauchbare Ergebnisse – der Karatsuba-Hybrid immerhin bis $n = 35$ – der FFT-basierte Algorithmus hingegen nur bis etwa $n = 25$.

Ein Faktor, der zu schlechter Stabilität der FFT führen kann, ist die Berechnung der Einheitswurzeln. In [PST02] wurden mehrere Methoden darauf getestet, welche Auswirkungen sie auf die Stabilität der FFT haben. Bei der FFT-Implementierung, die den Tests zu Grunde liegt, wurde aber bereits die Methode mit dem besten Resultat gewählt: Berechnung durch jeweils einen sin / cos-Aufruf pro Einheitswurzel.

In den Diagrammen ist auch zu beobachten, dass sich der Logarithmus des Fehlers in etwa linear zur Eingabegröße n verhält. Dies bedeutet, dass der Fehler exponentiell in n wächst.

Überraschend ist nicht nur der stark wachsende Fehler, sondern auch die Tatsache, dass die Ergebnisse der Auswertung für größere n schnell zwischen `+Infinity`, `-Infinity` oder `NaN` (Not a Number) variieren. Dies sind spezielle Werte in der Fliesskomma-Arithmetik, welche durch Ausnahmen wie Überläufe oder Division durch 0 entstehen. Die in Abbildung 4.1 und Abbildung 4.2 dargestellten Messreihen laufen dabei genau bis zum ersten Auftreten einer solchen Ausnahme.

Schon die Messreihen für $x = 1$ zeigen, dass die Hybridisierung des Karatsuba anscheinend auch Vorteile für die Stabilität des Algorithmus mit sich bringt. Dies mag daran liegen, dass durch Einsatz der Schulmethode am Rekursionsboden des Karatsuba Hybriden weniger Fehler durch die Rekursion nach oben gereicht werden. Dies bestätigt sich auch in der zweiten Messung für $x = 0.125$.

In Abbildung 4.2 sehen wir nun die Fehler für schnelle n-malige Auswertung an der Stelle $x = 0.125$. Es ist anzumerken, dass $f_n(0.125)$ für $n \to \infty$ gegen $\frac{8}{7}$ konvergiert.

Das erwartete Ergebnis ist also keineswegs bösartig. Die Schulmethode liefert diesmal bis $n = 530$ brauchbare Ergebnisse. Jenseits dieser Grenze liegen die Ergebnisse um ein hundertfaches daneben.
Die Karatsuba-basierten Ergebnisse sind bis etwa $n = 230$ brauchbar, der Karatsuba Hybrid schafft es bis etwa $n = 280$ und die FFT liefert breits ab $n = 160$ keine brauchbaren Ergebnisse mehr.

Neben der schnellen Mehrfachauswertung an n-mal derselben Stelle x, wurden auch schnelle Mehrfachauswertungen an n zufällig gleichverteilten oder auch äquidistante Stellen getestet. Diese wurden jeweils aus den Intervallen $[0, 1]$ und $[-1, 1]$ gewählt. Es ergab sich jedesmal dasselbe Bild: die Fehler treten schon für vergleichsweise kleine n auf und wachsen dann rasant.

Ab welcher Eingabegröße n und in welcher Geschwindigkeit die Fehler wachsen, schien bei den Tests weniger vom auszuwertenden Polynom f abzuhängen.
Vielmehr war es so, dass dies von den Stellen abhing, an denen das Polynom f ausgewertet werden sollte. Diesen Zusammenhang erläutert der folgende Abschnitt.

4.1 Polynompotenzierung

Die schnelle Mehrfachauswertung, wie sie in Abschnitt 2.4 beschrieben ist, berechnet zuerst einen Produktbaum. Die eigentliche Auswertung eines Polynoms findet dann durch die Division mit den Polynomen aus diesem Baum statt. Bei den Tests wurde allerdings ein starkes Wachstum der Koeffizienten dieser Polynome beobachtet. Dieses Wachstum wird als Grund für die oben beschriebenen Probleme angesehen. Die Gründe für dieses Wachstum sollen im folgenden erläutert werden. Dazu wird zunächst erneut der Aufbau des Produktbaums betrachtet um dann auf den Spezialfall Polynompotenzierung überzugehen.

Zur Auswertung eines Polynoms an n Stellen $p_0, \ldots, p_{n-1}$ wird der Produktbaum als vollständiger binärer Bäume mit Tiefe $t = \lceil \log n \rceil$ aufgebaut. Auf dessen 2^t Blättern werden n Polynome der Form $x - p_i$ verteilt. Die anderen $2^t - n$ Blätter sind mit dem konstanten Polynom 1 gefüllt. Alle anderen Knoten enthalten jeweils das Produkt ihrer Kindknoten. (Siehe auch Abbildung 2.15 und Abbildung 2.16)

Setzt man nun $p_0 = p_1 = \ldots = p_{n-1} = p$ für eine Konstante $p \in \mathbb{R}$, so enthält der Produktbaum lauter Potenzen des Polynoms $x - p$. Die Wurzel enthält das Produkt aller Blätter – also das Polynom $(x - p)^n$. Bei einer „gleichmäßigen“ Verteilung, wie sie in Abschnitt 2.4 gewählt wurde, enthält das linke Kind der Wurzel dann das Polynom $(x - p)^{\lfloor \frac{n}{2} \rfloor}$, und das rechte Kind enthält das Polynom $(x - p)^{\lceil \frac{n}{2} \rceil}$.

Der Spezialfall $p_i = p$ reduziert die Berechnung des Produktbaums also auf die Berechnungen der Potenzen des Polynoms $x - p$ (Polynompotenzierung). Die Koeffizienten dieser Potenzen lassen sich wie folgt beschreiben:

Beobachtung 4.1. *Die Koeffizienten des Polynoms $f(x) := (x-p)^n$ mit $n \in \mathbb{N}, p \in \mathbb{R}$ lassen sich wie folgt berechnen:*

$$f_k = \binom{n}{k}(-p)^{n-k} \qquad \text{für } k = 0, 1, \ldots, n$$

Die obige Formel folgt sofort aus dem binomischen Lehrsatz

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^k b^{n-k}$$

durch Einsetzen von $a := x$ und $b := -p$.

Mit der Formel aus Beobachtung 4.1 lassen sich nun die Koeffizienten der Polynompotenz $f(x) = (x-p)^n$ genauer untersuchen. Dazu soll allerdings nun zunächst der Binomialkoeffizient mit Hilfe der Stirling-Formel abgeschätzt werden. Dann werden zwei konkrete Koeffizienten genauer untersucht.

Die Stirling-Formel bietet folgende Abschätzung für die Fakultät:

$$n! = \sqrt{2\pi n}\,\frac{n^n}{e^n}\lambda_n \qquad \text{mit} \quad 1 < e^{\frac{1}{12n+1}} < \lambda_n < e^{\frac{1}{12n}} < 1.087 \qquad \forall n \in \mathbb{N}$$

Die Abschätzung wird nun auf den Binomialkoeffizienten übertragen:

$$\begin{aligned}
\binom{n}{k} &= \frac{n!}{k!(n-k)!} \\
&= \frac{\sqrt{2\pi n}\ \frac{n^n}{e^n}\lambda_n}{\sqrt{2\pi k}\ \frac{k^k}{e^k}\lambda_k \quad \sqrt{2\pi(n-k)}\ \frac{(n-k)^{n-k}}{e^{n-k}}\lambda_{n-k}} \\
&= \frac{n^n}{\sqrt{2\pi\frac{k(n-k)}{n}}\ k^k(n-k)^{n-k}} \cdot \frac{\lambda_n}{\lambda_k\lambda_{n-k}} \qquad \forall k = 1, \ldots, n-1 \quad \forall n \in \mathbb{N}
\end{aligned}$$

Der zusätzliche Faktor $\frac{\lambda_n}{\lambda_k\lambda_{n-k}}$ wird nun mit den zuvor bei der Stirling-Formel angegebenen festen Grenzen nach unten und nach oben abgeschätzt. Daraus ergibt sich:

$$0.8463 < \frac{\lambda_n}{\lambda_k\lambda_{n-k}} < 1.087$$

Insbesondere folgt aus der Abschätzung auch eine untere Schranke, indem man den Faktor als 0.8463 annimmt. Und da sich der Fehler in einem konstanten Bereich bewegt,

wird er im Folgenden nicht weiter angegeben, da er auf das asymptotische Wachstum keinen Einfluss hat.

Mit der obigen Abschätzung des Binomialkoeffizienten lässt sich der Betrag des mittleren Koeffzienten $f_{n/2}$ (für $2n \in \mathbb{N}$) wie folgt abschätzen:

$$\begin{aligned}
\left\|f_{n/2}\right\| &= \left\| \binom{n}{n/2} (-p)^{n/2} \right\| \\
&= \binom{n}{n/2} \|p\|^{n/2} \\
&\approx \frac{n^n}{\sqrt{2\pi\frac{n}{4}}\ (\frac{n}{2})^{\frac{n}{2}} (\frac{n}{2})^{\frac{n}{2}}} \|p\|^{n/2} \\
&= \frac{n^n}{\sqrt{\frac{1}{2}\pi n}\ (\frac{n}{2})^n} \|p\|^{n/2} \\
&= \frac{\|4p\|^{n/2}}{\sqrt{\frac{1}{2}\pi n}}
\end{aligned}$$

Und für den Koeffizienten $f_{3n/4}$ (mit $4n \in \mathbb{N}$) gilt die folgende Abschätzung:

$$\begin{aligned}
\left\|f_{3n/4}\right\| &= \binom{n}{3n/4} \|p\|^{n/4} \\
&\approx \frac{n^n}{\sqrt{2\pi\frac{3n}{16}}\ (\frac{3n}{4})^{\frac{3n}{4}} (\frac{n}{4})^{\frac{n}{4}}} \|p\|^{n/4} \\
&= \frac{\left\|\frac{256}{27}p\right\|^{n/4}}{\sqrt{\frac{3}{8}\pi n}}
\end{aligned}$$

Folgerung 4.2. *Für den mittleren Koeffizienten $f_{n/2}$ und den Koeefizienten $f_{3n/4}$ der n-ten Polynompotenz $f(x) = (x-p)^n$ gilt:*

$$\begin{aligned}
\left\|f_{n/2}\right\| &\in \Theta\left(\frac{\|4p\|^{n/2}}{\sqrt{n}}\right) \\
\left\|f_{3n/4}\right\| &\in \Theta\left(\frac{\left\|\frac{256}{27}p\right\|^{n/4}}{\sqrt{n}}\right)
\end{aligned}$$

Für $2n \in \mathbb{N}$ bzw. $4n \in \mathbb{N}$.

Der Betrag des mittleren Koeffizienten $f_{n/2}$ wächst also für den Fall $\|p\| > \frac{1}{4}$ exponentiell in n, der Betrag des Koeffizienten $f_{3n/4}$ sogar schon für den Fall $\|p\| > \frac{27}{256} \approx 0.1055$.

Das betragsmäßige Maximum der Koeffizienten des Polynoms verschiebt sich abhängig von p. Für $p = 1$ ergibt sich genau eine Binomialverteilung mit dem Maximum bei $f_{n/2}$. Für $\|p\| > 1$ verschiebt sich der Index des Koeffizienten mit der maximalen Größe in Richtung 0 und für $\|p\| < 1$ verschiebt er sich in Richtung n. Insbesondere kann exponentielles Wachstum für beliebig kleine $p > 0$ nachgewiesen werden.

Das exponentielle Wachstum, welches Folgerung 4.2 feststellt, ist wohl die Hauptursache für die hohen Fehler und die frühen Überläufe, die am Anfang des Kapitels geschildert werden. Als Beispiel seien hier die Koeffizienten $g'_{n/2}$ und $g''_{7n/8}$ der Polynome $g' = (x-1)^n$ und $g'' = (x - 0.125)^n$ für einige n gegeben:

$$\begin{aligned}
((x-1)^{200})_{100} &\approx 9.054851466 \cdot 10^{58} \\
((x-1)^{250})_{125} &\approx 9.120836693 \cdot 10^{73} \\
((x-1)^{300})_{150} &\approx 9.375970277 \cdot 10^{88} \\
((x-0.125)^{608})_{532} &\approx 1.37295533 \cdot 10^{29} \\
((x-0.125)^{704})_{616} &\approx 1.508511199 \cdot 10^{34} \\
((x-0.125)^{800})_{700} &\approx 1.674559372 \cdot 10^{39}
\end{aligned}$$

Die Werte stehen in keinem Verhältnis zu den Ergebnissen der Auswertungen, wie sie am Anfang dieses Kapitels getestet werden. So sind die exakten Ergebnisse $f_{300}(1) = 300$ sowie $f_{800}(0.125) \approx 1.143$ der Auswertungen des Polynoms $f_n(x) := \sum_{i=0}^{n-1} x^i$ gegenüber den den obigen Werten verschwindend klein.

Allein die Speicherung von Zahlen dieser Größenordnung in gewöhnlichen 64-Bit langen Fliesskommazahlen verursacht durch Rundungsfehler im schlimmsten Fall absolute Fehler, die ebenfalls exponentiell in n wachsen. Zusätzlich wird Auslöschung gefördert.
Dies stimmt mit den Beobachtungen vom Anfang des Kapitels überein, dass die Fehler in Abbildung 4.1 und Abbildung 4.2 exponentiell in n zu wachsen scheinen.

4.2 iRRAM

Im vorhergehenden Abschnitt werden die numerischen Schwierigkeiten mit den vorliegenden Methoden und Algorithmen beleuchtet. Anstelle dem dort zu Grunde liegenden Fliesskomma-Modells, wird die Polynompotenzierung nun auf Basis eines anderen Rechenmodells untersucht. Dieses Rechenmodell wird von der C++-Bibliothek iRRAM [Mül00] implementiert. Sie erlaubt das Rechnen mit speziellen Repräsentationen von reellen Zahlen.

Dafür stellt die Bibliothek iRRAM den Datentyp `REAL` zur Verfügung. Durch Operatorüberladung kann mit diesem Datentyp in ähnlicher Weise wie mit herkömmlichen

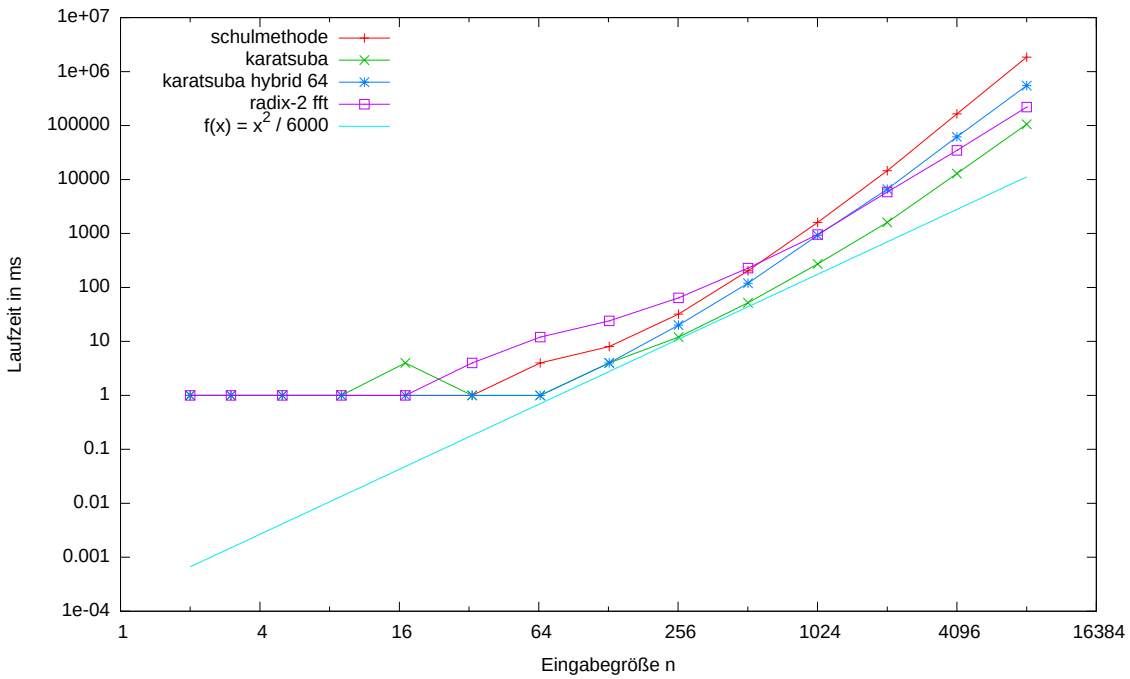

Abbildung 4.3: Quadrierung des Polynoms $(x - 0.5)^n$ mit absoluter Genauigkeit

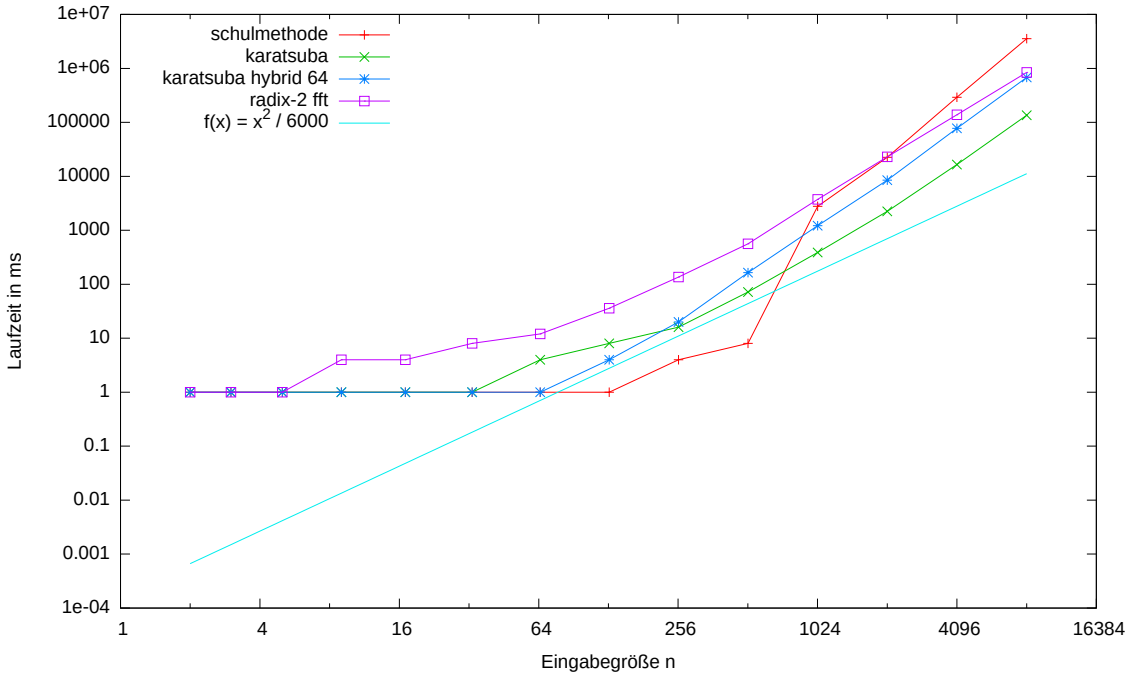

Abbildung 4.4: Quadrierung des Polynoms $(x - 0.5)^n$ mit relativer Genauigkeit

Datentypen gerechnet werden. Der Datentyp repräsentiert eine reelle Zahl $a \in \mathbb{R}$ durch das Tupel (d_a, ε_a), so dass gilt:

$$\|a - d_a\| \leq \varepsilon_a$$

Es gilt also immer $a \in [d_a - \varepsilon_a, d_a + \varepsilon_a]$. Die Zahl d_a stellt also eine Approximation von a mit Fehlerschranke ε_a dar.

Die Approximation d_a wird als Fliesskommazahl gespeichert. Man hat die Wahl zwischen drei verschiedenen Modulen, die von der iRRAM für die Verwaltung dieser Fliesskommazahl benutzt werden. Den folgenden Messungen liegt dabei das Modul zu Grunde, welches die GMP-Bibliothek benutzt (siehe Abschnitt 3.1). Die Größe der Fliesskommazahlen ist dabei nicht auf die üblichen 32, 64 oder 80-Bit beschränkt, sondern wird von der iRRAM gewählt. Die Wahl hängt von mehreren Faktoren ab. Genaueres wird in [Mül00, Kapitel 6] erläutert.

Für die Speicherung der Fehlerschranken geht die iRRAM einen Kompromiss ein. Einerseits könnte man den Fehler *sehr* genau (z.B. wieder als Fliesskommazahl beliebiger Größe) speichern oder ihn grob durch Zweierpotenzen der Form $2^p, p \in \mathbb{Z}$ abschätzen. Die iRRAM geht hier nun einen Mittelweg und benutzt eine Speicherung der Form $z \cdot 2^p$ mit $z \in \mathbb{N}$ und $p \in \mathbb{Z}$. Die Zahlen z und p werden dabei jeweils als etwa 30-bittige Ganzzahlen gespeichert (C++-Datentyp `int`). Dieses Verfahren vermeidet grobe Abschätzungen, braucht nur wenig Platz und ist daher ein sehr guter Kompromiss.

Bei jeder arithmetischen Operation $\circ \in \{+, -, *, /, \ldots\}$ mit den Eingaben a und b (jeweils dargestellt durch die Tupel (d_a, ε_a) und (d_b, ε_b) wird nun ein Tupel (d_c, ε_c) berechnet, welches das Ergebnis $c = a \circ b$ darstellt, so dass gilt:

$$\|c - d_c\| < \varepsilon_c$$

Es kann auch durchaus vorkommen, dass die iRRAM bewusst größere Werte für ε_c wählt, als sie eigentlich müsste. Dadurch kann die iRRAM die Speicherung einer zu langen Mantisse der Zahl d_c vermeiden.

An die iRRAM kann nun ein Zeiger auf eine C++-Funktion übergeben werden, welche den neuen Datentyp `REAL` benutzt. Die iRRAM führt diese Funktion dann in spezieller Weise aus. Typischerweise bekommt die Funktion einige Eingabeparameter – z.B. einen Exponent n als 32-Bit Ganzahl und den Wert 0.1 als String – welche dann bei Bedarf in den `REAL` Datentyp konvertiert werden um dann damit zu rechnen. Die Konvertierung des Strings `"0.1"` ist dabei ein Fall, bei dem die iRRAM die Genauigkeit der Konvertierung steuern und bewusst Fehler in Kauf nehmen muss, da die Zahl 0.1 im Binärsystem eine periodische Darstellung hat.

Während mit den `REAL`-Variablen gerechnet wird, kann es vorkommen, dass eine Zahl nicht in ausreichender Genauigkeit vorliegt. Dies ist z.B. bei einer Division $b = 1/a$

der Fall, wenn 0 in dem die Zahl a darstellenden Intervall $[d_a - \varepsilon_a, d_a + \varepsilon_a]$ enthalten ist. In diesem Fall kann keine sinnvolle Darstellung der Zahl b berechnet werden. Die iRRAM bricht dann ab, und führt den Algorithmus von neuem und mit gesteigerter Genauigkeit aus. Eine berechenbare Funktion vorausgesetzt, garantiert die Rekursive Analysis (siehe Kapitel 5), dass die Darstellung der Zahl a bei sukzessiver Steigerung der Genauigkeit konvergiert. Wenn zusätzlich $a \neq 0$ gilt, dann ist sichergestellt, dass der Intervall irgendwann klein genug wird und die 0 nicht mehr enthält.

Desweiteren können Zahlen vom Datentyp `REAL` auch wieder in herkömmliche Fliesskommazahlen zurückkonvertiert werden. Dafür bietet die iRRAM die Möglichkeit, Approximationen zu ermitteln, die eine vorgegebene Fehlerschranke einhalten. Die Fehlerschranke wird in der Form $2^p, p \in \mathbb{Z}$ angegeben. Liegt der Wert nicht in der entsprechenden Genauigkeit vor, wird der Algorithmus mit gesteigerter Genauigkeit neu gestartet.
Neben absoluten Fehlerschranken gibt es auch die Möglichkeit, Approximation mit relativer Fehlerschranke abzufragen.

Für die Tests wurden nun Approximationen der Polynompotenzen $(x - 0.5)^n$ für $n = 2, 4, 8, 16, \ldots$ berechnet. Die iRRAM wurde dann mit jeder dieser Polynompotenzen aufgerufen, um diese mit Hilfe der Multiplikationsalgorithmen zu quadrieren. Die Laufzeit des Quadrierens wurde dann gemessen. Die Abbildung 4.3 zeigt nun die Laufzeit für den Fall, dass die Ergebnisse mit der absoluten Fehlerschranke 2^{-30} angefragt wurde. In Abbildung 4.4 wurde eine relative Fehlerschranke 2^{-30} benutzt - es wurden also quasi die oberen 30 Bit der Koeffizienten abgefragt.
Es sei angemerkt, dass jeweils immer nur der letzte Durchgang gemessen wurde, wenn der Algorithmus von der iRRAM mehrfach ausgeführt werde musste.

Beiden Diagrammen wurde zum Vergleich auch eine quadratische Funktion hinzugefügt. Und sofern dies aus den Diagrammen erkennbar ist, zeigen die effizienten Verfahren auf der iRRAM keine subquadratische Laufzeit mehr. Der Grund ist, dass die iRRAM mit wesentlich längeren Fliesskommazahlen arbeiten muss, als die üblichen 64 Bit. Es kommt hinzu, dass die Größe der Zahlen von n abhängt und keineswegs konstante Kosten für Addition und Multiplikation zweier Zahlen angenommen werden dürfen.

Allerdings benutzt die iRRAM, solange es ihr möglich ist, die ganz normalen Fliesskommazahlen, wie sie von der CPU direkt unterstützt werden. Diese Optimierung ist wahrscheinlich für die Sprünge in den Diagrammen verantwortlich zu machen.

Leider war es nicht möglich, die Tests für größere Werte durchzuführen um eindeutigere Trends in den Laufzeitkurven feststellen zu können. Die Laufzeit des Testsprogramms, welches die hier beschriebenen Messungen durchführt, betrug schon für diese relativ kleinen Eingaben mehrere Tage.

5 Berechenbarkeit der Division mit Rest

In der Numerik werden Algorithmen anhand ihrer Stabilität und Komplexität verglichen und entworfen. Das Ziel ist die Approximation von Funktionen mit der üblichen Fliesskomma-Arithmetik und möglichst kleinen oberen Fehlerschranken. Die verwendeten Fehler- und Komplexitätsmodelle sind dabei auf die heutigen Rechner abgestimmt. Die Rekursive Analysis stellt dem ein Rechenmodell gegenüber, in dem reelle Werte durch unendliche Folgen von rationalen Approximationen dargestellt werden. Trotzdem steht im Mittelpunkt immer noch eine herkömmliche Turing Maschine. Daher kann dieses Modell auch praktisch umgesetzt werden. Mit der iRRAM, die bereits vorgestellt wurde, existiert schon eine praktische Umsetzung dieser Ideen.

Im Jahr 1936 stellte Alan Turing eine Definition für die Berechenbarkeit von reellen Zahlen vor [Tur36]. Da diese Definition einige Probleme mit sich brachte, veröffentliche er 1937 eine neue korrigierte Definition [Tur37]. Diese fusst auf der Darstellung einer reellen Zahl $a \in \mathbb{R}$ durch zwei rationale Folgen (a_n) und (b_n), die folgende Bedingungen erfüllen:

$$a_i \leq a_{i+1} < b_{i+1} \leq b_i \quad \wedge \quad a_i \leq a \leq b_i \quad \wedge \quad \|a_i - b_i\| < 2^{-i} \qquad \forall i \in \mathbb{N}$$

Die Darstellung lässt sich auch so auffassen, dass die reelle Zahl a durch eine Folge von Intervallen $[a_n, b_n] \ni a$ approximiert wird. Die Intervallgrenzen, d.h. die Folgen (a_n) und (b_n), konvergieren dabei exponentiell schnell von unten bzw. von oben gegen a.

Eine ganz ähnliche Darstellung wurde 1957 von Grzegorczyk benutzt [Grz57]. Eine reelle Zahl $x \in \mathbb{R}$ wird ebenfalls durch eine Folge von Intervallen $[x_1^{(n)}, x_2^{(n)}]$ mit rationalen Grenzen repräsentiert. Allerdings müssen die Intervalle lediglich die folgende Bedingung erfüllen:

$$\sup_{i \in \mathbb{N}} x_1^{(i)} = \inf_{i \in \mathbb{N}} x_2^{(i)} = x$$

Jede Folge von Intervallen, die den Bedingungen von Turing genügt, erfüllt diese Bedingung. Umgekehrt ist das nicht der Fall. Grzegorczyk stellt weder Anfordungen an die Monotonie der beiden Folgen $x_1^{(n)}$ und $x_2^{(n)}$ (vgl. $a_i \leq a_{i+1}$ bzw. $b_{i+1} \leq b_i$ bei Turing) noch wird eine Aussage über die Geschwindigkeit der Konvergenz gemacht (vgl.

$\|a_i - b_i\| < 2^{-i}$). Auch die Ungleichheit der Intervallgrenzen (vgl. $a_i < b_i$) wird nicht gefordert.
Dennoch folgt aus Grzegorczyks Bedingung, dass die Folgen $x_1^{(n)}, x_2^{(n)}$ jeweils monotone, exponentiell-schnell konvergierende Teilfolgen $x_1'^{(n)}, x_2'^{(n)}$ besitzen müssen, die die Bedingung $\left\|x_1'^{(i)} - x_2'^{(i)}\right\| < 2^{-i-1}$ erfüllen. Durch die Vorschrift $x_2''^{(i)} := x_2'^{(i)} + 2^{-i-1}$ erhalten wir dann die Folge $x_2''^{(n)}$ welche zusammen mit der Folge $x_1'^{(n)}$ die Zahl x korrekt im Sinne von Turing darstellt.
Daraus folgt nun sofort die Äquivalenz der beiden Zahlendarstellungen. Zusätzlich stellt man fest, dass die Konvertierung zwischen den Darstellungen – inbesondere das Herausfiltern der monotonen exponentiell konvergierenden Teilfolgen – von einer Turing Maschine geleistet werden kann. Die beiden Darstellungen sind also sogar uniform äquivalent.

Eine weitere uniform äquivalente Darstellung kodiert eine Zahl $a \in \mathbb{R}$ durch eine rationale Folge $(a_1', \varepsilon_1, a_2', \varepsilon_2, \ldots)$. Für diese Folge soll gelten:

$$\|a - a_i'\| \leq \varepsilon_i \to 0$$

Das Tupel (a_i', ε_i) besteht also aus einer Approximation a_i' der Zahl a mit bekannter Fehlerschranke ε_i. Und während die Folge (ε_n) der Fehlerschranken gegen 0 geht, konvergiert die Folge (a_n') der Approximationen gegen die Zahl a. Solch eine Folge von Approximationen und Fehlerschranken lässt sich problemlos aus einer Darstellung im Sinne von Turing gewinnen. Man setzt einfach $a_i' := \frac{a_i + b_i}{2}$ und $\varepsilon_i := 2^{-i-1}$. Auf der anderen Seite sind die Intervalle der Form $[a_i' - \varepsilon_i, a_i' + \varepsilon_i]$ eine korrekte Darstellung der Zahl a im Sinne von Grzegorczyk.

Bei einer anderen Variante der obigen Darstellung wird die Fehlerschranke implizit vorausgesetzt, anstatt sie explizit anzugeben. Die Zahl $a \in \mathbb{R}$ wird nun einfach durch eine rationale Folge (a_n'') dargestellt, wobei die folgende Bedingung gelten muss:

$$\|a - a_i''\| < 2^{-i} \qquad \forall i \in \mathbb{N}$$

Es wird also zugesichert, dass die Folge (a_n'') exponentiell schnell gegen a konvergiert. Die Äquivalenz zur Darstellung nach Turing und zur Darstellung nach Grzegorczyk kann ganz analog zur Darstellung mit expliziter Fehlerschranke gezeigt werden. Aufgrund ihrer Einfachheit wird letztere Darstellung in dieser Arbeit bevorzugt. Sie mündet inbesondere in folgendem Berechenbarkeitsbegriff:

Definition 5.1. *Eine reelle Zahl $x \in \mathbb{R}$ heisst* berechenbar*, wenn es eine Turing Maschine gibt, die eine rationale Folge (x_n) mit $x_i \in \mathbb{Q}$ und $\|x - x_i\| < 2^{-i}$ berechnet.*

Eine reelle Funktion $f : A \to \mathbb{R}$ heisst berechenbar*, wenn es eine Turing Maschine gibt, die bei Eingabe einer beliebigen rationalen Folge (x_n) mit $\|x - x_i\| < 2^{-i}$ und $x \in A$ eine rationale Folge (y_i) mit $\|y - y_i\| < 2^{-i}$ und $y = f(x)$ berechnet.*

Die Unberechenbarkeit einer Funktion an der Stelle x, wird gemäß dieser Definition zu einem sehr starken Hindernis: Obwohl die Turing Maschine Zugang zu beliebig genauen Approximationen der Eingabe x hat, kann der Wert $f(x)$ nicht beliebig genau berechnet werden. Diese Grenze ist von so prinzipieller Natur, dass sie sich auch direkt auf die Numerik auswirkt.

Definition 5.1 gilt auch in intuitiver Weise für n-dimensionale Vektoren aus $\mathbb{R}^n$. Aus der Definition der p-Normen folgt nämlich sofort, dass die Fehlerschranken nicht nur für den Vektor insgesamt gelten, sondern auch für die einzelnen Vektorkomponenten.

Später benötigen wir allerdings auch Darstellungen von Vektoren aus $\mathbb{R}^* = \bigcup_{i \in \mathbb{N}} \mathbb{R}^i$. Für die Approximation dieser beliebig-dimensionalen Vektoren müssen allerdings zusätzliche Vereinbarungen getroffen werden. Einerseits sind Kodierungen denkbar, in denen die Dimension des Vektors explizit angegeben wird. Allerdings werden die Fälle, in denen die Dimension der Vektoren bekannt ist, separat betrachtet werden. Daher wurde für diese Arbeit die folgende Kodierung gewählt, in der der Vektor $x \in \mathbb{R}^*$ durch eine Folge endlich dimensionaler Vektoren $x^{(n)}$ approximiert wird. Für diese Folge soll gelten:

$$x^{(i)} \in \mathbb{Q}^i \qquad \forall i \in \mathbb{N}$$

Für die Komponenten der einzelnen Folgenglieder soll gelten:

$$\left\| x_j - x_j^{(i)} \right\| \leq 2^{-i} \qquad \forall j \in \{0, \ldots, i-1\} \quad \forall i \in \mathbb{N}$$

Die Folge $x^{(n)}$ besteht also aus Vektoren wachsender Dimension. Deren Komponenten dürfen nicht um mehr als die Fehlerschranke 2^{-i} von den Komponenten des zu repräsentierenden Vektors x abweichen, als es die Fehlerschranke 2^{-i} erlaubt. (Es wird $x_j = 0$ für alle $j \geq \dim(x)$ angenommen) Die Dimension des Vektors x ist in der Eingabe nicht explizit enthalten.

Beispiel 5.2. *Die folgenden Funktionen sind berechenbar:* [Wei00, Theorem 4.3.2]

- Addition: $(x, y) \mapsto x + y$
- Subtraktion: $(x, y) \mapsto x - y$
- Multiplikation: $(x, y) \mapsto x \cdot y$
- Division: $(x, y) \mapsto x/y \quad$ für $y \neq 0$
- Die Verkettung $f \circ g$ von berechenbaren Funktionen f, g [Wei00, Theorem 3.1.6]

Jeweils mit $x, y \in \mathbb{R}$.

Mit Beispiel 5.2 steht uns nun ein Grundrepertoire an berechenbaren Funktionen zur Verfügung. Aus Definition 5.1 folgt desweiteren sofort, dass auch alle rationalen Zahlen $q \in \mathbb{Q}$ und auch die zugehörigen konstanten Funktionen $f_q(x) := q$ berechenbar sind. Über Verkettung lassen sich aus den einzelnen Funktionen beliebig komplizierte Ausdrücke bilden.
Berechenbarkeit ist hier im wesentlichen als Existenzaussage zu verstehen: Es *gibt* eine Turingmaschine, die die Summe/das Produkt/usw. zweier Zahlen ausgibt. Es *gibt* eine Turingmaschine, die der Verkettung $f \circ g$ entspricht.
Insbesondere die Aussage zur Verkettung kann weitergeführt werden. Die Verkettung zweier beliebiger berechenbarer Funktionten f und g ist nämlich uniform berechenbar. Das heisst, dass es eine Turingmaschine gibt, die bei Eingabe der Kodierung der zu f und g gehörigen Turingmaschinen M_f und M_g die Kodierung der zu $f \circ g$ gehörigen Turingmaschine $M_{f \circ g}$ ausgibt. Im Wesentlichen wird hier eine „Hintereinanderschaltung“ der beiden Turingmaschinen M_f und M_g berechnet, in der die Ausgabe von M_g zur Eingabe von M_f wird.
Unter anderem wird dadurch deutlich, dass eine Turingmaschine *zur Laufzeit* die Kodierung von Turingmaschinen für beliebige Verkettungen berechnen und anschließend auf einer universellen Turingmaschine ausführen kann. Turingmaschinen für Addition/Mutliplikation/usw. sind bekannt.

Beispiel 5.3. *Die folgenden Funktionen sind* nicht *berechenbar:*

- Vorzeichenfunktion: $x \mapsto \begin{cases} -1 & \text{für } x < 0 \\ 0 & \text{für } x = 0 \\ 1 & \text{für } x > 0 \end{cases}$
- Test auf Gleichheit: $x \mapsto \begin{cases} 1 & \text{für } x = 0 \\ 0 & \text{für } x \neq 0 \end{cases}$
- Jede unstetige Funktion
- Die Haltefunktion: $x \mapsto 2^{-H} = \sum_{i \in H} 2^{-i}$

Jeweils mit $x \in \mathbb{R}$

In der Tat ist bereits bekannt, dass jede berechenbare Funktion notwendigerweise stetig ist [Wei00, Theorem 4.3.1]. Daraus folgt sofort, dass jede unstetige Funktion unberechenbar sein muss. Desweiteren nennt Beispiel 5.3 zwei konkrete unstetige Funktionen: die Vorzeichenfunktion und den Test auf Gleichheit. Aus diesen Unberechenbarkeitsaussagen folgt jedoch auch die *Unentscheidbarkeit* dieser Probleme. Dies ist wichtig, da somit die Möglichkeit von exakten Tests auf Gleichheit oder die Ermittlung des Vorzeichens reeller Zahlen nicht für bedingte Sprünge oder Verzweigungen im Programm benutzt werden kann. Dies ist eine eklatante Einschränkung gegenüber dem normalen Programmieren.
Die praktische Implementierung iRRAM unterstützt allerdings „unscharfe“ Tests und Vergleiche. Diese erlauben es wieder, gewisse Zusagen über reelle Zahlen machen zu können. (Siehe Beschreibung von `bound` und `positive` in [Mül00])

Die Zahl $2^{-H} \in \mathbb{R}$ ist die sogenannte Haltezahl. Für jede Gödelnummer $i \in \mathbb{N}$ eines haltenden Programms, ist das entspreche Bit 2^i in der Binärdarstellung Zahl 2^{-H} auf 1 gesetzt. Alle anderen Bits der Zahl sind 0. Die konstante Funktion, die die Haltezahl zurückgibt, ist natürlich stetig. Sie ist ein Beispiel einer nicht-berechenbaren stetigen Funktion. Es gilt also *nicht*, dass jede stetige Funktion berechenbar ist.

Mit den geschaffenen Grundlagen lässt sich der folgende Satz beweisen:

Satz 5.4. *Folgende Abbildungen sind berechenbar:*

- $\text{mult}_n : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^{2n-1}, (f, g) \mapsto f \cdot g$
- $\text{mult}_* : \mathbb{R}^* \times \mathbb{R}^* \to \mathbb{R}^*, (f, g) \mapsto f \cdot g$

Wobei $n \in \mathbb{N}$.

Beweis. In Abschnitt 2.1 wurde bereits gezeigt, dass das Ergebnis der Abbildung mult_n maximal $2n-1$-dimensional ist. Dort wurde auch bereits das Cauchy-Produkt für diesen endlichen Fall vorgestellt. Daraus ergibt sich für jede Komponente des Ergebnisses eine

konkrete Rechenvorschrift (Summe von Produkten), die eine endliche Verkettung von berechenbaren Funktionen darstellt. Damit ist das Ergebnis ebenfalls berechenbar.

Für die Abbildung mult_* wird das allgemeine Cauchy-Produkt benötigt. Die Multiplikation $r = f \cdot g$ mit $f, g, r \in \mathbb{R}^*$ ist dabei wie folgt definiert:

$$r_k = \sum_{\substack{i+j=k \\ i,j \geq 0}} f_i g_j = \sum_{i=0}^{k} f_i g_{k-i} \qquad \text{für } k = 0, 1, 2, \ldots$$

Damit hängt die $k-te$ Komponente von r von den ersten $k+1$ Komponenten von jeweils f und g ab. Der Vektor r muss nun durch die Folge der Vektoren $r^{(i)} \in \mathbb{R}^i$ kodiert werden. Der i-te Vektor der Folge hat dabei i Komponenten, deren Werte sich durch die obige Cauchy-Formel ergeben.
Eine Turingmaschine, die die Abbildung mult_* berechnet, baut nun zur Laufzeit in Abhängigkeit von i die Verkettungen für die i Komponenten des auszugebenen Vektors auf, um diese anschließend auszuwerten. Das dies möglich ist, wird durch die uniforme Berechenbarkeit der Verkettungen (Summe von Produkten) garantiert. □

Für die weiteren Betrachtungen – insbesondere die Untersuchungen der Division mit Rest – benötigen wir nun die folgende Definition:

Definition 5.5. *Sei $n \in N$ und $\mathbb{R}^n$ der Raum der Polynome vom Grad kleiner n. Dann ist*

$$\mathbb{R}^{=n} := \mathbb{R}^{n-1} \times (\mathbb{R} \setminus 0) \subset \mathbb{R}^n$$

der Raum der Polynome vom Grad genau $n-1$.

Für Polynome aus $\mathbb{R}^n$ ist lediglich die obere Grenze n des Grades bekannt. Und bei Polynomen aus $\mathbb{R}^*$ ist nicht mal eine solche obere Grenze bekannt. Welcher der Koeffizienten ungleich 0 ist, kann die Turingmaschine aufgrund der Unentscheidbarkeit der Tests auf Gleichheit nicht feststellen. Den Grad des Polynoms als bekannt vorauszusetzen, ist daher eine sinnvolle Einschränkung des Definitionsbereichs.

Satz 5.6. *Folgende Abbildungen sind* nicht *berechenbar:*

- $\text{rem}_{n,m} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^{m-1}, (f, g) \mapsto f \operatorname{rem} g$
- $\text{rem}_{*,=m} : \mathbb{R}^* \times \mathbb{R}^{=m} \to \mathbb{R}^{m-1}, (f, g) \mapsto f \operatorname{rem} g$

Wobei $n, m \in \mathbb{N}$, $m \geq 2$ und $g \neq 0$.

Beweis. Wir nehmen an, es gäbe eine Turing Maschine M, die $\text{rem}_{n,m}(f, g)$, (n, m beliebig aber fest) berechnet. OBdA wird ein Beispiel mit $n = 2$ und $m = 2$ gewählt. Das

Beispiel lässt sich auf alle Fälle mit n beliebig und $m \geq 2$ übertragen.
Die Turing Maschine wird nun mit den beiden Folgen $(f_n) \to (1,0)$ und $(g_n) \to (1,0)$ als Eingabe gestartet. Wir wählen außerdem $f_i = (1,0)$ und $g_i = (1, 2^{-i})$. Nach endlicher Zeit und dem Lesen der jeweils N ersten Folgenglieder hat die Turing Maschine M die ersten drei Glieder einer Folge (r_n) ausgegeben, die gegen das Ergebnis $f \operatorname{rem} g = 0$ geht. Es gilt also $\|0 - r_3\| < 2^{-3}$. Daraus folgt $r_3 \in [-\frac{1}{8}, +\frac{1}{8}]$
Die Turing Maschine M wird ein zweites mal gestartet. Diesmal mit den Folgen (f_n) und (g'_n) als Eingabe. Wir wählen $(g'_n) = (g_0, \ldots, g_N, (1, 2^{-N}), (1, 2^{-N}), \ldots) \to (1, 2^{-N})$. Aufgrund des Determinismus von M liefert die Turing Maschine als Ausgabe wieder die drei Glieder der Folge (r_n). Die Ausgabe kann nicht wieder zurückgenommen werden. Der Rest der Division f/g' ist diesmal allerdings 1. Die Fehlerschranke $\|1 - r_3\| < 2^{-3}$ ist nicht erfüllt. Dies ist ein Widerspruch zur Annahme.

Es wird nun angenommen, dass die Turing Maschine M die Abbildung $\operatorname{rem}_{*,=m}(f,g)$ berechnet. Die Turing Maschine wird mit den beiden Folgen $(f_n) \to \vec{0} \in \mathbb{R}^*$ und $(g_n) \to (1,1) \in \mathbb{R}^2$ als Eingabe gestartet. Nach endlicher Zeit und dem Lesen der jeweils N ersten Folgenglieder hat die Turing Maschine M dann die ersten drei Glieder einer Folge (r_n) ausgegeben. Diese Folge muss exponentiell schnell gegen das Ergebnis $0 \in \mathbb{R}$ konvergieren. Es gilt also $\|0 - r_3\| < 2^{-3}$.
Die Turing Maschine wird ein zweites mal gestartet. Diesmal bekommt sie die Folgen $(f'_n) \to (0, \ldots, 0, 1) \in \mathbb{R}^{N+1}$ und (g_n) als Eingabe. Die ersten N Glieder von (f'_n) sind dabei identisch zu denen von (f_n). Aufgrund des Determinismus wird die Turingmaschine wieder die drei ersten Glieder der Folge (r_n) ausgeben. Das Ergebnis ist diesmal allerdings $x^N \operatorname{rem}(x+1) = (-1)^N$. □

Folgerung 5.7. *Folgende Abbildungen sind ebenfalls* nicht *berechenbar:*

- $\operatorname{div}_{n,m} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n, (f,g) \mapsto f \operatorname{div} g$
- $\operatorname{div}_{*,=m} : \mathbb{R}^* \times \mathbb{R}^{=m} \to \mathbb{R}^*, (f,g) \mapsto f \operatorname{div} g$

Wobei $n, m \in \mathbb{N}$, $m \geq 2$ *und* $g \neq 0$.

Desweiteren sind auch alle Abbildungen mit erweitertem Definitionsbereich nicht berechenbar. Dazu zählen $\operatorname{rem}_{*,m}, \operatorname{div}_{*,m}, \operatorname{rem}_{*,*}, \operatorname{div}_{*,*}, \ldots$

Beweis. Die Berechenbarkeit der Division stünde im Widerspruch zu Satz 5.6, denn es gilt:

$$f \operatorname{rem} g = f - (f \operatorname{div} g) \cdot g$$

Und somit folgt aus der Berechenbarkeit der Division auch die Berechenbarkeit des Rests. □

Im ersten Teil des Beweises von Satz 5.6 wird zuerst ausgenutzt, das der Grad des Divisors g nicht bekannt ist. Dies führt dazu, dass man die Turingmaschine „austricksen“ kann, weil sie nicht unterscheiden kann, ob es sich bei der Eingabe um das Polynom $g = 1 + 0x$ oder um das Polynom $g = 1 + \varepsilon x$ handelt.
Und im zweiten Teil wird ausgenutzt, dass keine obere Grenze für den Grad des Polynoms f bekannt ist. Die Turingmaschine kann dadurch nicht wissen, ob nicht noch irgendwann jenseits der von ihr bereits gelesenen Eingaben ein von 0 verschiedener Koeffizient auftaucht, welcher das Ergebnis beeinflusst.

Die Definitionsbereiche lassen sich nun soweit einschränken, dass beide Probleme nicht mehr auftauchen. Und in der Tat stellt der folgender Satz die Berechenbarkeit fest:

Satz 5.8. *Die folgenden Abbildungen sind berechenbar:*

- $\mathrm{div}_{n,=m} : \mathbb{R}^n \times \mathbb{R}^{=m} \to \mathbb{R}^{n-m+1}, (f, g) \mapsto f \operatorname{div} g$
- $\mathrm{rem}_{n,=m} : \mathbb{R}^n \times \mathbb{R}^{=m} \to \mathbb{R}^{m-1}, (f, g) \mapsto f \operatorname{rem} g$

Wobei $n, m \in \mathbb{N}$ *und* $g \neq 0$.

Beweis. Die Berechenbarkeit der Polynomdivision $\mathrm{div}_{n,=m}(f, g)$ kann konstruktiv gezeigt werden. Durch den Definitionsbereich ist bereits gegeben, dass das Divisorpolynom genau den Grad $m - 1$ hat und damit gilt: $g_{m-1} \neq 0$. Für jeden Koeffizienten des Quotientenpolynoms $f \operatorname{div} g$ lässt sich jetzt eine konkrete Rechenforschrift finden (z.B. anhand der Polynomdivision, wie sie aus der Schule bekannt ist). Für jeden Koeffizienten erhält diese dann eine endliche Verkettung von Additionen, Multiplikationen und Divisionen. Geteilt wird bei der Schulmethode immer nur durch den führenden Koeffizienten g_{m-1} welcher ungleich Null ist. Es ist also sicher, dass alle Divisionen und damit alle Koeffizienten berechenbar sind.

Die Berechenbarkeit des Rests der Division ergibt sich automatisch, da er durch eine zusätzliche Multiplikation und Subtraktion berechnet werden kann:

$$f \operatorname{rem} g = f - (f \operatorname{div} g) \cdot g$$

□

Zusammenfassend lässt sich also sagen, dass die Dision mit Rest nur unter sehr restriktiven Bedingungen berechenbar ist. Der Grad des Divisors g muss exakt bekannt sein und für den Grad des Dividenden f muss eine obere Schranke bekannt sein. Ist dies nicht der Fall, kann die Division und ihr Rest nicht berechnet werden.

In diesem Zusammenhang ist auch wichtig, was über den Grad der Ergebnisse $f \operatorname{div} g$ und $f \operatorname{rem} g$ ausgesagt werden kann – inwiefern die Ergebnisse also als Eingabe für eine erneute Division verwendet werden können. Es wird angenommen, dass gilt:

$$\operatorname{grad}(f) < n \quad \wedge \quad \operatorname{grad}(g) = m - 1 \quad \wedge \quad g \neq 0 \qquad \text{mit } n, m \in \mathbb{N}$$

Weiterhin wird angenommen, dass $m \leq n$ gilt, da sonst für alle f das triviale Ergebnis $f \operatorname{div} g = 0$ entsteht. Da allerdings n nur eine obere Grenze darstellt, kann es immer noch passieren, dass die Situation $\operatorname{grad}(f) < \operatorname{grad}(g) = m - 1$ eintritt. Für diesen Fall gilt dann:

$$\begin{array}{rcll} f \operatorname{div} g & = & 0 & \Rightarrow \quad \operatorname{grad}(f \operatorname{div} g) < n - m + 1 \\ f \operatorname{rem} g & = & f & \Rightarrow \quad \operatorname{grad}(f \operatorname{rem} g) < m - 1 \end{array}$$

Für den anderen Fall $\operatorname{grad}(f) \geq \operatorname{grad}(g)$ gilt:

$$\begin{array}{rcll} \operatorname{grad}(f \operatorname{div} g) & = & \operatorname{grad}(f) - \operatorname{grad}(g) & < \quad n - m + 1 \\ \operatorname{grad}(f \operatorname{rem} g) & < & \operatorname{grad}(g) & = \quad m - 1 \end{array}$$

Sowohl für den Grad des Polynoms $f \operatorname{div} g$ als auch für den Grad des Polynoms $f \operatorname{rem} g$ lässt sich lediglich eine obere Grenze angeben. Der Grad ist nicht exakt bekannt.
Nur unter der zusätzlichen Vorraussetzung $\operatorname{grad}(f) = n - 1$ kann gefolgert werden:

$$\operatorname{grad}(f \operatorname{div} g) = n - m$$

6 Zusammenfassung und Ausblick

Neben der Schulmethode (Cauchy-Produkt) wurden zwei weitere Algorithmen zur Polynommultiplikation in C++ implementiert: Karatsuba und FFT-basierte Multiplikation. Der normale Karatsuba Algorithmus überraschte mit hoher Laufzeit, zeigte aber das erwartete asymptotische Laufzeitverhalten.Die Laufzeitprobleme konnten durch Hybridisierung behoben werden. Der entstandene Karatsuba-Hybrid kombiniert die geringe Laufzeit der Schulmethode für kleine Eingaben mit dem asymptotischen Verhalten des Karatsuba-Algorithmus.
Schulmethode und Karatsuba wurden dann mit der selbst implementierten komplexen Cooley-Tukey-FFT verglichen. Es zeigte sich, dass sich diese FFT für reelle Polynome erst ab Grad 2048 lohnt. Mit spezieller Optimierung für reelle Polynome wäre es möglich, diesen Schwellwert auf ca. 1024 zu drücken.
Zusätzlich wurde auch eine Polynommultiplikation basierend auf der hoch-optimierten OpenSource Bibliothek FFTW getestet. Durch entsprechenden Optimierungsaufwand kann eine auf dieser FFT-Bibliothek basierende Polynommultiplikation schon etwa ab einem Polynomgrad von 16 eingesetzt werden. Dabei ist diese Bibliothek aber auf spezielle Prozessorerweiterungen moderner Intel-Pozessoren angewiesen und ist auf die normalen 32/64/80-Bit Fliesskommadatentypen beschränkt.
Desweiteren wurde ein Algorithmus zur schnellen Mehrfachauswertung implementiert. Dieser arbeitet mit einem Divisionsalgorithmus, der mit Hilfe einer Newton-Iteration auf die obigen Multiplikationsalgorithmen aufgesetzt wurde.

In Kapitel 3 wurde ein Überblick über die Verbreitung der vorgestellten Verfahren gegeben. Einige Bibliotheken (unter anderem die NTL) benutzen die asymptotisch schnellen Verfahren ausschließlich für Polynome über endlichen Körpern. Einzig MATLAB benutzt die FFT zur reellen Polynommultikation. In Kapitel 4 wurde von den Verfahren aber gezeigt, dass sie für Anwendungen wie der schnellen Mehrfachauswertung und den Spezialfall der Polynompotenzierung im Reellen nicht geeignet sind. Verantwortlich hierfür ist das extreme Wachsum der Koefizienten bei einer Potenzierung. Insbesondere die FFT bringt hier die größten absoluten komponentenweisen Fehler hervor. Die iRRAM konnte die numerischen Probleme kompensieren. Allerdings nur mit extremen Laufzeiteinbußen.

Kapitel 5 stellt abschließend die Berechenbarkeit der Multiplikation fest. Die Berechenbarkeit der Division ist nur unter gewissen Vorbedingungen gegeben. So muss eine obere

Schranke des Dividenden bekannt sein und der Grad des Divisors muss exakt bekannt sein.

Es stellen sich folgende weiterführenden Fragen:

- Gibt es einen Algorithmus, der die Koeffizienten der Polynompotenzen $(x-p)^n$ schnell und mit hoher Genauigkeit berechnet?
- Gilt dies auch für das Polynom $\prod_{i=0}^{n-1}(x-p_i)$?
- Lässt sich der Algorithmus zur Polynompotenzierung besser auf die iRRAM optimieren?
- Lässt sich ein Verfahren zur schnellen Mehrfachauswertung finden, welches das Problem der Polynompotenzierung vermeidet?

Literaturverzeichnis

[CB93] Clausen, Michael ; Baum, Ulrich: *Fast Fourier Transforms*. Mannheim : Bibliographisches Institut & F.A. Brockhaus AG, 1993

[CT65] Cooley, James W. ; Tukey, John W.: An Algorithm for the Machine Calculation of Complex Fourier Series. In: *Mathematics of Computation* 19 (1965), April, Nr. 90, S. 297–301. – ISSN 0025–5718

[FJ05] Frigo, Matteo ; Johnson, Steven G.: The Design and Implementation of FFTW3. In: *Proceedings of the IEEE* 93 (2005), Nr. 2, S. 216–231. – special issue on "Program Generation, Optimization, and Platform Adaptation"

[GG03] Gathen, Joachim von zur ; Gerhard, Jürgen: *Modern Computer Algebra*. New York, NY, USA : Cambridge University Press, 2003. – ISBN 0–521–82646–2

[Grz57] Grzegorczyk, Andrzej: On the definitions of computable real continuous functions. In: *Fundamenta Mathematicae* 44 (1957), S. 61–71

[Mül00] Müller, Norbert: The iRRAM: Exact Arithmetic in C++. In: *4th International Workshop on Constructivity and Complexity in Analysis*, Springer, 2000, 222–252

[PST02] Potts, Daniel ; Steidl, Gabriele ; Tasche, Manfred: Numerical stability of fast trigonometric transforms – a worst case study. In: *Concrete Appl. Math.* 1 (2002), S. 1–36

[Tur36] Turing, Alan M.: On Computable Numbers, with an application to the Entscheidungsproblem. In: *Procedings of the London Mathematical Society* 42 (1936), Nr. 2, S. 230–265

[Tur37] Turing, Alan M.: On Computable Numbers, with an application to the Entscheidungsproblem, A Correction. In: *Procedings of the London Mathematical Society* 43 (1937), Nr. 2, S. 544–546

[Wei00] Weihrauch, Klaus: *Computable Analysis*. Berlin : Springer, 2000. – ISBN 3–540–66817–9

Danksagung

Bei Dr. Martin Ziegler möchte ich mich besonders für die ausgezeichnete Betreuung und die aufschlussreichen Diskussionen bedanken. Meiner Mutter sowie Matthias und Jochen danke ich für die Unterstützung beim Korrekturlesen.

Versicherung

Ich versichere hiermit, dass ich diese Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Diese Arbeit wurde in gleicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Paderborn, den 21. Juni 2007,

Sven Köhler