

LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

**Grundlagenpraktikum: Rechnerarchitektur**

Gruppe 213 – Abgabe zu Aufgabe A326

Sommersemester 2023

Noah Schlenker

Leon Baptist Kniffki

Christian Krinitzin

## 1 Einleitung

Verwendung von Wurzel 2:

- Das verhältnis der beiden seitenlängen eines blattes im din-a-format beträgt  $1 / \sqrt{2}$  mit rundung auf ganze millimeter. Dadurch ist sichergestellt, dass bei halbierung des blattes entlang der längeren seite wieder ein blatt im din-a-format (mit um eins erhöhter nummerierung) entsteht.
- Die wurzel aus 2 ist das frequenzverhältnis zweier töne in der musik bei gleichschwebender stimmung, die einen tritonus, also eine halbe oktave bilden.
- In der elektrotechnik enthält die beziehung zwischen scheinwert und effektivwert von sinusförmiger wechselfeldspannung ebenfalls die konstante  $\sqrt{2}$ .

Geschichte über Näherung der Wurzel:

- Die alten Inder schätzen  $\sqrt{2} \approx \frac{577}{408} = 1,414215686\dots$ . Stimmt auf 5 Nachkommastellen. Abweichung beträgt nur +0,0001502 Prozent.
- Babylonier aus 1800 v. Chr.:  $\frac{30547}{21600} = 1,414212962\dots$ . Abweichung von -0,0000424 Prozent.

Möglichkeit,  $\sqrt{2}$  mit einer unendlichen Präzision zu berechnen. Vorwegnahme: Bignums, Newton-Raphson-Division, Karazuba, Matrixexponentiation.  
(0,75 Seiten)

## 2 Lösungsansatz

### 2.1 Big-Num

Der Anspruch der Arbeit liegt darin,  $\sqrt{2}$  beliebig genau zu berechnen, herkömmliche Variablen mit festgelegter Größe können damit nicht verwendet werden. Hiermit wird die Datenstruktur *Big-Num* eingeführt. Sie ermöglicht die Darstellung von Zahlen mit beliebiger Größe und Genauigkeit, braucht in C aber eine eigene Implementierung, die im Folgenden erläutert wird.

Die Definition der Datenstruktur sieht wie folgt aus:

---

```

1 struct bignum {
2     uint32_t *digits;
3     size_t size;
4     size_t fracSize;
5 };

```

Zahlen werden in einem Array von 32-Bit großen *digits* gespeichert. Die Reihenfolge der DoubleWords ist Little-Endian. *Size* gibt die Größe des Arrays an, *fracSize* bestimmt, wie viele Nachkommastellen in dieser Zahl vorhanden sind, dazu mehr in 2.5.

Nun werden verschiedene Arithmetische Operationen ausgeführt.

### Addition und Subtraktion

Bei der Addition und der Subtraktion werden die einzelnen Elemente aufeinander addiert bzw. subtrahiert. Entsteht ein Overflow, wird dieser auf die nächsten Blöcke übertragen.

Seien  $a$  und  $b$  zwei Big-Nums, unterteilt in ihre Blöcke von 0 bis  $m$ . Falls die *digits* eines Big-Nums kleiner sind als die des anderen, werden die entsprechenden Blöcke mit Nullen aufgefüllt. Für die Addition und Subtraktion gilt:

$$c = \sum_{i=0}^m (2^{32i} (a_i + b_i)) \quad \text{und} \quad (1)$$

$$c = \sum_{i=0}^m (2^{32i} (a_i - b_i)). \quad (2)$$

Hiermit sind beide Operationen trivial mit einer Laufzeit von  $\mathcal{O}(n)$  implementiert.

### Multiplikation

Die Multiplikation orientiert sich an der Schulmethode des schriftlichen Multiplizierens. Es wird jeweils ein Block des einen Faktors mit jedem Block des anderen Faktors multipliziert, die Produkte werden aufsummiert. Dies wird mit jedem Block aus dem ersten Faktoren wiederholt. Mathematisch lässt sich dies mit den eingangs definierten Big-Nums  $a$  und  $b$  folgendermaßen darstellen:

$$c = \sum_{i=0}^m (2^{(32i)} b_i \sum_{j=0}^m a_j). \quad (3)$$

Damit funktioniert auch die triviale Multiplikation mit einer Laufzeit von  $\mathcal{O}(n^2)$ . Im Folgenden wird ein Algorithmus zur schnelleren Berechnung des Produkts erläutert.

## 2.2 Karazuba-Multiplikation

Um zwei Zahlen miteinander zu multiplizieren läuft man bei der russischen Bauernmultiplikation des Multiplikators einmal den Multiplikanten ab, wenn man diesen auf das Zwischenergebnis addiert. Für zwei Zahlen der Längen  $n, m \in \mathbb{N}$  hat die Bauernmultiplikation also Laufzeit von  $\mathcal{O}(n * m)$  und im häufigen Fall von  $n = m$   $\mathcal{O}(n^2)$ . Die Multiplikation spielt auf der untersten Ebene für die Matrixmultiplikation und somit auch für die Berechnung von  $\sqrt{2}$  eine wichtige Rolle.

Dank des Karazuba-Algorithmus kann die Laufzeit einer Multiplikation auf  $\mathcal{O}(n^{1.59})$  verringert werden. Dafür werden die zu multiplizierenden Zahlen in die Form  $a_0 + a_1 * 2^m$  gebracht, wobei  $a_0$  und  $a_1$  maximal die Größe  $\lceil \frac{n}{2} \rceil$  haben. Durch folgende Umformung kann  $a * b$  mit nur noch drei  $\lceil \frac{n}{2} \rceil$  großen Multiplikationen und 6 zu vernachlässigenden Additionen/Subtraktionen und einigen shifts ermittelt werden:

$$\begin{aligned} a * b &= (a_0 + a_1 * 2^m) * (b_0 + b_1 * 2^m) \\ &= a_0 b_0 + a_0 a_1 * 2^m + a_1 b_0 * 2^m + a_1 b_1 * 2^m * 2^m \\ &= a_0 b_0 + 2^m (a_0 a_1 + a_1 b_0) + a_1 b_1 * 2^{2m} \\ &= a_0 b_0 + 2^m (a_0 a_1 + a_1 b_0 + a_0 b_0 + a_1 b_1 - a_0 b_0 - a_1 b_1) + a_1 b_1 * 2^{2m} \\ &= a_0 b_0 + 2^m ((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1) + a_1 b_1 * 2^{2m} \end{aligned}$$

## 2.3 Matrixmultiplikation

Die Matrixmultiplikation kann wie nach Definition implementiert werden. Um zwei  $\mathbb{N}^{2 \times 2}$  Matrizen miteinander zu multiplizieren, werden acht Multiplikation und vier Additionen benötigt:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{pmatrix}$$

Da die Laufzeit der Multiplikation definitiv aufwändiger ist als die der Addition, ist das Ziel, die Multiplikationen zu minimieren.

Für allgemeine  $\mathbb{N}^{2 \times 2}$  Matrizen wäre dies bereits die minimale Form, allerdings befindet sich die Matrix  $\begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}$  in der Form  $\begin{pmatrix} x_{n-1} & x_n \\ x_n & x_{n+1} \end{pmatrix}$  und behalte diese für Multiplikationen mit sich selbst bei. Diese Eigenschaft macht es nicht nur möglich, die Anzahl der gespeicherten Werte von 4 auf drei zu reduzieren, sondern auch die Multiplikationen auf vier zu halbieren:

$$\begin{pmatrix} x_{n-1} & x_n \\ x_n & x_{n+1} \end{pmatrix} * \begin{pmatrix} y_{n-1} & y_n \\ y_n & y_{n+1} \end{pmatrix} = \begin{pmatrix} x_{n-1} * y_{n-1} + x_n * y_n & x_{n-1} * y_n + x_n * y_{n+1} \\ x_n * y_{n-1} + x_{n+1} * y_n & x_n * y_n + x_{n+1} * y_{n+1} \end{pmatrix}$$

Multipliziert zwei mal die gleiche Matrix oder Matrizen, die Potenzen der selben Basis sind, gilt  $x_{n-1} * y_n + x_n * y_{n+1} = x_n * y_{n-1} + x_{n+1} * y_n$ . Durch das Einsparen der doppelten Berechnung von  $x_n$  und das Wiederverwenden von  $x_n * y_n$  sind nur noch fünf

Multiplikationen nötig. Im Fall von  $\begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}$  kann  $x_n$  auch als eine rekursiv definierte Folge mit  $x_n = 2x_{n-1} + x_{n-2}$  und  $x_0 = 0, x_1 = 1$  interpretiert werden:

$$\begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix} * \begin{pmatrix} x_{n-1} & x_n \\ x_n & x_{n+1} \end{pmatrix} = \begin{pmatrix} x_n & x_{n+1} \\ x_{n-1} + 2x_n = x_{n+1} & x_n + 2x_{n+1} = x_{n+2} \end{pmatrix}$$

Es müssen demnach lediglich  $x_{n-1}$  und  $x_n$  mit vier Multiplikationen berechnet werden.  $x_{n+1}$  erhält man durch einen shift und eine Addition.

## 2.4 Schnelle Exponentiation

Die schnelle Exponentiation nutzt Assoziativität und Potenzgesetze, um die Zahl der Multiplikationen bei der Exponentiation von  $\mathcal{O}(n)$  auf  $\mathcal{O}(\log n)$  zu verringern. Naiv kann eine Potenz  $a^n$  mit  $n \in \mathbb{N}$  nach der Schulmethode mit  $\prod_1^n a$  berechnet werden. Dafür benötigt man allerdings  $n - 1$  Multiplikationen, was bei großen Werten für  $n$  zu einer langen Berechnung ausartet.

Um dieses Problem effizienter zu lösen, werfen wir erst einmal einen Blick auf die Potenzgesetze für assoziative Operatoren. Denn sowohl eine Multiplikation, als auch eine Addition im Exponenten kann aufgeteilt werden:

$$a^{n+m} = \overbrace{a * \dots * a}^{n+m} = \overbrace{(a * \dots * a)}^n * \overbrace{(a * \dots * a)}^m = a^n * a^m \quad (4)$$

$$a^{n*m} = \overbrace{a * \dots * a}^{n*m} = \underbrace{\overbrace{(a * \dots * a)}^n * \dots * \overbrace{(a * \dots * a)}^n}_m = (a^n)^m \quad (5)$$

Wenn man also  $a^n$  und  $a^m$  effizienter als mit  $n + m$  Multiplikationen berechnen kann, kann man auch  $a^{n+m}$  mit 4 effizient berechnen.

Bei der schnellen Exponentiation berechnet man durch wiederholtes Quadrieren alle  $a^{(2^k)}$  mit  $2^k \leq n$ . Denn nach 5 gilt:

$$\left(a^{(2^k)}\right)^2 = a^{(2^k * 2)} = a^{(2^{k+1})}.$$

Um  $a^n$  mit  $n = 2^k$  zu berechnen, sind damit nur noch  $k = \log_2 n$  Multiplikationen notwendig.

Potenzen mit der Form  $a^{(2^k)}$  können also effizient berechnet werden, um nun auch Potenzen mit  $n \in \mathbb{N}$  berechnen zu können, nutzt man 4. Jede Zahl  $n \in \mathbb{N}$  kann durch Addition von Zweierpotenzen dargestellt werden (Binärsystem).

Sei  $n$  in Binärdarstellung  $b_0 * 2^0 + b_1 * 2^1 + b_2 * 2^2 \dots$ , so erhält man  $a^n$  mit:

$$a^n = a^{b_0 * 2^0 + b_1 * 2^1 + b_2 * 2^2 \dots} = a^{b_0 * 2^0} * a^{b_1 * 2^1} * a^{b_2 * 2^2} \dots$$

Da  $b_i$  nur die Werte 0 und 1 annehmen kann, ist es am Ende eine boolsche Entscheidung, ob der aktuelle Wert von  $a^{(2^k)}$  auf das Zwischenergebnis aufmultipliziert wird, oder

nicht.

Außerdem gilt:

$$a^n * a^m = a^m * a^n \quad (6)$$

Auch wenn diese Gleichung auf den ersten Blick nach einem Kommutativgesetz aussieht, gilt sie aufgrund der Assoziativität, da nur die Klammerung geändert wird:

$$\overbrace{(a * \dots * a)}^n * \overbrace{(a * \dots * a)}^m = \overbrace{(a * \dots * a)}^m * \overbrace{(a * \dots * a)}^n$$

Ein Beispiel:

$$7^3 * 7^4 = \overbrace{(7 * 7 * 7)}^3 * \overbrace{(7 * 7 * 7 * 7)}^4 = \overbrace{(7 * 7 * 7 * 7)}^4 * \overbrace{(7 * 7 * 7)}^3 = 7^4 * 7^3$$

Demnach macht es keinen Unterschied, ob zuerst  $a^{(2^k)}$  mit dem kleinsten oder dem größten  $k$  aufmultipliziert wird.

Da  $(\mathbb{N}^{2 \times 2}, *)$  eine Gruppe und damit assoziativ ist, kann die schnelle Exponentiation auch für das Lösen von  $a \in \mathbb{N}^{2 \times 2}$  genutzt werden.

## 2.5 Matrizen

anwendung der schnellen exponentiation auf matrizen, beispiel dazu (0.75 Seiten)

## 2.6 Newton-Raphson-Division

Mithilfe der bereitgestellten Matrix werden nun die Werte ausgerechnet, die bei der Division an  $\sqrt{2}$  konvergieren. Nun beschäftigen wir uns mit der Darstellung von Kommazahlen und einem Algorithmus zur Berechnung eines Quotienten in dieser Darstellung. Wir kennen zwei verschiedene Formen der Darstellung von Kommazahlen: Fließkommazahlen nach *IEEE-754* und Fixpunktzahlen. Der Vorteil von Fließkommazahlen ist ihr großer Wertebereich, der dafür mit schwankender Genauigkeit einherkommt, wie man in Abbildung 1 erkennen kann.

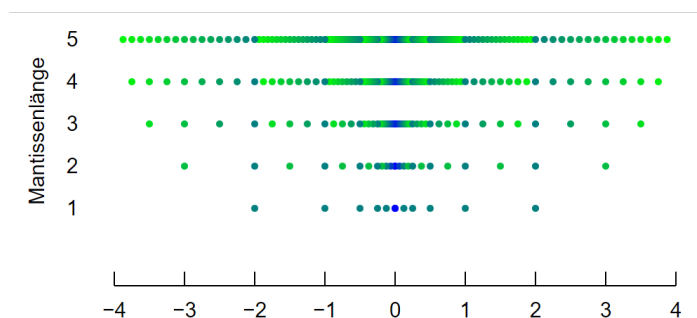


Abbildung 1: Exakt darstellbare Fließkommazahlen mit verschiedenen Mantissen (Entnommen aus [?])

Fixpunktzahlen haben einen kleineren Wertebereich bei gleichem Speicherverbrauch, besitzen dafür eine schnellere und deutlich einfachere Arithmetik und haben eine gleichbleibende Genauigkeit im gesamten Wertebereich - zu sehen in Abbildung 2.

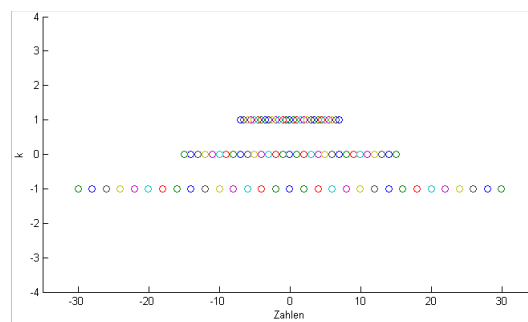


Abbildung 2: Exakt darstellbare Fixpunktzahlen mit k Nachkommastellen (Entnommen aus [1])

Ein weiterer Vorteil von Fixpunktzahlen liegt darin, dass man unendlich viele Nachkommastellen einfach darstellen kann, in dem man beispielsweise die implementierten Big-Nums aus 2.1 verwendet. Aus den genannten Gründen lässt sich schließen, dass

sich die Nutzung von Fixpunktzahlen viel besser eignet, um  $\sqrt{2}$  mit beliebig vielen Nachkommastellen darzustellen.

Die Newton-Raphson-Division ist ein Algorithmus zur Berechnung der Division. Er nutzt das Newton-Verfahren, das dafür da ist, Nullstellen von Funktionen zu approximieren und wendet dies so an, um Quotienten zu berechnen. Das Verfahren besteht aus 3 Schritten, die nun erläutert werden [2]. Wir wollen den Quotienten aus  $a$  und  $b$  berechnen, mit  $a, b \in \mathbb{N}$ .

Zunächst brauchen wir eine erste Näherung. Wir bitshiften den Quotienten so lange nach rechts, bis  $0,5 \leq b < 1,0$ . Die erste Näherung erhalten wir mit:

$$x_0 = \frac{48}{17} - b \frac{32}{17}. \quad (7)$$

Als Nächstes berechnen wir iterativ immer bessere Näherungen:

$$x_{i+1} = x_i(2 - bx_i). \quad (8)$$

Mit jeder Iteration erhält das Ergebnis doppelt so viele Nachkommastellen wie vorher, es wird auch doppelt so genau. Um die Anzahl der benötigten Iterationen zu erhalten, betrachten wir folgende Formel, wobei  $k$  für die erwartete Anzahl an Nachkommastellen im Ergebnis steht:

$$\text{Iterationen} = \lfloor \log_2 k \rfloor. \quad (9)$$

Um nun das Endergebnis zu erhalten, multiplizieren das Zwischenergebnis mit  $a$  und erhalten somit unser Ergebnis der Division. Es kann passieren, dass durch die ständige Verdoppelung der Nachkommastellen, die Zahl zu viele davon besitzt als notwendig. Diese Ziffern werden einfach abgeschnitten.

Als illustratives Beispiel berechnen wir  $5/12$  mit sieben Nachkommastellen. Die Anzahl der Iterationen beträgt  $\lfloor \log_2 7 \rfloor = 2$ . Nach wiederholten bitshiften der Zahl 12 erhalten wir  $(0,1100)_2 = (0,75)_{10}$ . Wir berechnen die Zwischenschritte:

$$\begin{aligned} x_0 &= (10,1101.001)_2 - (0,1100)_2 \cdot (1,111)_2 = (1,0110.101)_2; \\ x_1 &= (1,0110.101)_2 \cdot (2_{10} - (0,1100)_2 \cdot (1,0110.101)_2) = (1,0101.0100.0001.0101.00)_2; \\ x_2 &= (1,0101.0100.0001.0101.00)_2 \cdot (2_{10} - (0,1100)_2 \cdot (1,0101.0100.0001.0101.00)_2) \\ &= (1,0101.0101.0101.0100.0010.1000.1011.0101.0100.0000)_2. \end{aligned}$$

Multipliziert man  $x_2$  mit dem geshifteten  $a = (0,0101)_2$  und shiftet dieses so weit, so dass die Zahl nur noch sieben Nachkommastellen besitzt, erhält man das Ergebnis  $(0,0110.101)_2 = (0,4141)_{10}$ , welches sich an der dritten dezimalen Nachkommastelle vom eigentlichen Ergebnis  $0,41\bar{6}$  abweicht.

Wir sind in der Lage, Divisionen durchzuführen und effizient Werte auszurechnen, die das Ergebnis approximieren.

(1 Seite gedacht, jetzt 2)

### 3 Genauigkeit

Wahrscheinlich Genauigkeit, da es die Aufgabe ist,  $\sqrt{2}$  beliebig genau darzustellen.

Umfangreiche Erklärung darüber, wie die Matrix Elemente an  $\sqrt{2}$  konvergiert und Newton-Raphson and die Division. Erklärung, wie die Kombination aus Bignum und Fixkommazahlen unendliche Genauigkeit ermöglicht, auf Kosten von Laufzeit, die im nächsten Kapitel beleuchtet wird.

(1,5 - 2 Seiten?)

### 4 Performanzanalyse

Newton Raphson Laufzeit erklären, mit Graphiken demonstrieren, das selbe mit Exponentiation.

Vergleichsimpementierungen ansetzen (SIMD, nicht SIMD? / Karazuba, normale Multiplikation / Bitshifts?), Graphisch laufzeiten vergleichen, tatsächliche Performanz erklären und schlussfolgern.

(2 - 3 Seiten)

### 5 Zusammenfassung und Ausblick

Ziel erreicht, unendliche Präzision ist gegeben. Nutzer können, abhängig von ihren Anforderungen oder "Computerspezifikationen", die Wurzel von 2 mit diesem Programm berechnen.

Ausblick: SIMD in AVX, mithilfe von 256 Bit kann man Multiplikationen noch schneller machen. Division lässt sich wahrscheinlich nicht optimieren, da SIMD nicht verwendet werden kann, und immer eine gewisse Anzahl an Iterationen gebraucht wird.

(0,75 - 1 Seite)

(Insgesamt 6 - 7,75 Seiten, 2 - 4 Seiten fehlen!)

### Literatur

[1] <https://de.wikipedia.org/wiki/Datei:Fixpointnumbers.png>, Zugriff am ??..??..????

[2] Pawan Kumar Pandey, Dilip Singh, and Rajeevan Chandel. Fixed-point divider using newton raphson division algorithm. In Vijay Nath and J. K. Mandal, editors, *Proceeding of Fifth International Conference on Microelectronics, Computing and Communication Systems*, pages 225–234, Singapore, 2021. Springer Singapore.