MaximilianKolbeSchule
Berufliche Oberschule Neumarkt i.d.OPf.
Staatliche Fachoberschule und Berufsoberschule
Menschen | Bildung | Zukunft

Seminararbeit 2022/23

Thema:

Der Fast-Inverse-Square-Root-Algorithmus mit Bezug zur 3D-Computergrafik

Name: Paul Schulz

Klasse: 13FT1

Betreuende Lehrkraft: OStR Stefan Baier
Dipl. Math. Johannes Ghiroga

Maximilian-Kolbe-Schule

Kerschensteinerstraße 7

92318 Neumarkt i. d. OPf.

1 Vorwort

Im Jahr 2002 und 2003 begannen in verschiedenen Internet-Foren Diskussionen über ein bestimmtes Stück Programmcode. Es handelte sich um einen Ausschnitt aus dem Quelltext des 1999 veröffentlichten Computerspiels *Quake III Arena*, einem Ego-Shooter-Spiel (McEniry 2007: S. 1).

```
float Q_rsqrt( float number ) {
	long i;
	float x2, y;
	const float threehalfs = 1.5F;
	x2 = number * 0.5F;
	y  = number;
	i  = * ( long * ) &y;            // evil floating point bit level hacking
	i  = 0x5f3759df - ( i >> 1 );    // what the fuck?
	y  = * ( float * ) &i;
	y  = y * ( threehalfs - ( x2 * y * y ) );     // 1st iteration

	return y;
}
```

Abbildung 1-1 zeigt den Quellcode einer Funktion aus dem Spiel *Quake III Arena* in leicht abgewandelter Form. Der Ausschnitt ist in der Programmiersprache C geschrieben.

Dem Namen der Funktion zufolge, berechnet der Code den Kehrwert der Wurzel einer positiven Zahl. Jedoch erscheinen die Berechnungen selbst für Entwickler mit großer Erfahrung in der Programmiersprache C nicht nachvollziehbar. Als besonders fragwürdig erscheint hierbei der Ursprung der hexadezimalen Konstante $5F3759DF$. Genauso ungewöhnlich ist die Interpretation einer Fließkommazahl als natürliche Zahl. Die Kommentare der Entwickler im Quellcode tragen ebenfalls wenig zum Verständnis bei.

Was zunächst willkürlich erschien, entpuppte sich als ein genialer, schneller und gut durchdachter Algorithmus, welcher dem Computerspiel eine bessere Performance verleihen sollte. In den folgenden Jahren erlangte das Verfahren unter dem englischen Namen *Fast-Inverse-Square-Root-Algorithmus* über das Internet große Popularität (McEniry 2007: S. 1).

Als gelernter Fachinformatiker für Anwendungsentwicklung erfuhr ich erstmals von der Existenz dieses Algorithmus durch einen Arbeitskollegen. Jedoch fehlten mir zu diesem Zeitpunkt die mathematischen Kenntnisse zum Verständnis des Verfahrens. Während des Seminars erinnerte ich mich durch einen Hinweis von Herrn Baier an den Algorithmus. Damit stand das Thema meiner Arbeit fest.

Diese Seminararbeit hat zum Ziel den Algorithmus mathematisch herzuleiten und verständlich zu erklären. Außerdem wird seine Bedeutung für die 3D-Computergrafik erläutert und damit ein Blick auf den Ursprung des Algorithmus geworfen.

2 Hinführung zum Thema

Diese Seminararbeit behandelt den *Fast-Inverse-Square-Root-Algorithmus*. Diese Bezeichnung wird von jetzt an mit FISR-Algorithmus abgekürzt. Das Verfahren dient zur Berechnung der reziproken Quadratwurzel einer positiven rationalen Zahl.

$$f(x) = \frac{1}{\sqrt{x}};\ D_f = \mathbb{Q}^+ \tag{2-1}$$

Der Algorithmus hat insbesondere eine schnelle und effiziente Berechnung zum Ziel, weswegen lediglich eine hinreichend genaue Annäherung (Approximation) an das tatsächliche Ergebnis berechnet wird (McEniry 2007: S. 1). Mit einem erhöhten Rechenaufwand wäre jedoch auch eine wesentlich genauere Berechnung möglich.

Dem Wort Algorithmus wird folgende Definition zugrunde gelegt:
„Algorithmen sind Vorschriften für die Lösung eines Problems, welche die Handlungen und ihre Abfolge […] beschreiben." (Heinisch et al. 2010: S. 4)

Algorithmen stellen damit eine Schritt-für-Schritt-Anleitung zur Lösung eines (z.B. mathematischen) Problems dar. Beispiele für bekannte mathematische Algorithmen sind: Der Euklidische Algorithmus oder die Polynomdivision. Aber auch alltägliche Dinge, wie Kochrezepte oder Montage-Anleitungen können als Algorithmen verstanden werden (Heinisch et al. 2010: S. 4).

Da die Herleitung und Erklärung des Algorithmus vor allem theoretischer Natur sind, wurde eigens für diese Seminararbeit eine Demo-Website programmiert. Auf dieser kann der Algorithmus praktisch ausprobiert und Schritt-für-Schritt nachvollzogen werden. Die Web-Anwendung kann unter der folgenden URL aufgerufen werden: https://fisr.schulz-paul.de

2.1 Aufbau der Arbeit

Der FISR-Algorithmus wurde ursprünglich für die 3D-Computergrafik entworfen und eingesetzt. Kapitel 3 erläutert daher den konkreten Bezug zur Computergrafik und damit die Notwendigkeit eines solchen Algorithmus.

Die Kapitel 4, 5 und 6 vermitteln wichtige mathematische Grundlagen, die zum weiteren Verständnis notwendig sind.

Schließlich beschäftigen sich die Kapitel 7 und 8 ausführlich mit dem Algorithmus selbst. Er wird dabei mathematisch hergeleitet und bewiesen. Da es sich bei der Berechnung um eine Approximation handelt, wird außerdem auf die Geschwindigkeit und Genauigkeit eingegangen.

Alle wichtigen Abbildungen, Definitionen, Formeln und Sätze sind mit der Kapitelnummer und einer innerhalb des Kapitels fortlaufenden Nummer beschriftet. Im Verlauf der Ausführungen wird auf vorangegangene Aussagen mithilfe eben dieser Beschriftungen verwiesen.

3 Bezug zur 3D-Computergrafik

Die Aufgabe der Computergrafik ist es, aus einer abstrakten Objektbeschreibung ein Bild zu generieren. Die Objektbeschreibung umfasst dabei üblicherweise Informationen über Form, Position oder Farbgebung der abgebildeten Objekte. Aber auch die Position von Lichtquellen, Reflexionseigenschaften oder Transparenz von Objekten sind Teil dieser Beschreibung (Nischwitz et al. 2019: S. 6f.).

In der 3D-Computergrafik werden dabei Objekte zunächst im 3-dimensionalen Raum positioniert. Da die Ausgabegeräte allerdings üblicherweise flach sind, muss das Bild noch in ein 2-dimensionales Bild umgewandelt werden. Dabei kommen verschiedenen mathematische Verfahren und Konzepte zum Einsatz (Nischwitz et al. 2019: S. 205f.).

Die Position von Objekten im Raum wird dabei üblicherweise durch Vektoren beschrieben. Insbesondere bei der Berechnung von Beleuchtung und Reflexionseffekten kommen dabei normalisierte Vektoren zum Einsatz.

3.1 Normalisierung von Vektoren

Unter Normalisierung eines Vektors versteht man die Skalierung seiner Länge auf 1 LE unter Beibehaltung seiner Richtung (Aryeh 2020: TC. 00:01:32 – 00:01:40).

Gegeben sei der 3-dimensionale Vektor $\vec{v} \neq \vec{0}$:

$$\vec{v} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix};\ v1, v2, v3 \in \mathbb{R} \tag{3-1}$$

Für den Betrag (die Länge) desselben Vektors gilt demnach:

$$|\vec{v}| = \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{3-2}$$

Werden alle Vektorkoordinaten von $\vec{v}$ mit einem Faktor $x \in \mathbb{R}_0^+$ multipliziert, so gilt für den Betrag des neuen Vektors:

$$|x \cdot \vec{v}| = \sqrt{(x \cdot v_1)^2 + (x \cdot v_2)^2 + (x \cdot v_3)^2} = \sqrt{x^2 \cdot (v_1^2 + v_2^2 + v_3^2)} = x \cdot |\vec{v}| \tag{3-3}$$

Somit gilt für den normalisierten Vektor $\overrightarrow{v_0}$ mit $|\overrightarrow{v_0}| = 1$:

$$\overrightarrow{v_0} = \frac{1}{|\vec{v}|} \cdot \vec{v} = \frac{1}{\sqrt{v_1^2 + v_2^2 + v_3^2}} \cdot \vec{v} \tag{3-4}$$

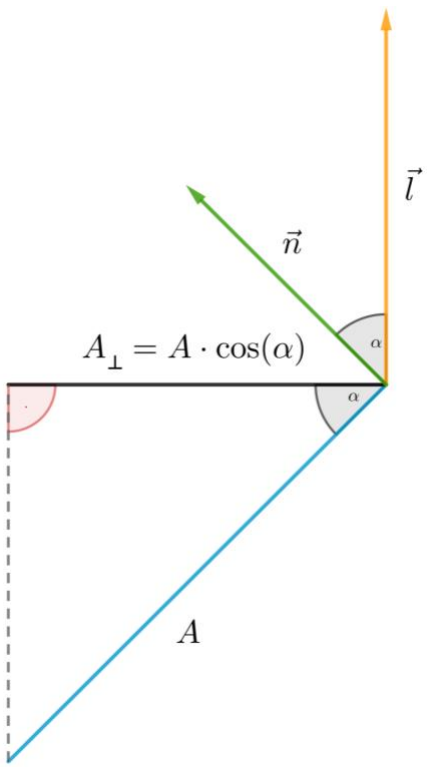
Der Skalierungsfaktor des Vektors $\vec{v}$ entspricht dabei der Funktion f aus (2-1).

3.2 Beispiel Lichtintensität

Die Beleuchtung ist in der 3D-Computergrafik dafür verantwortlich, dass Objekte auf dem 2-dimensionalen Bildschirm für das menschliche Auge 3-dimensional erscheinen. Durch Reflexions- und Schattierungseffekte wird dem Beobachter die Tiefe des Bildes vorgetäuscht (Nischwitz et al. 2019: S. 317).

In folgendem Beispiel soll die Verwendung normalisierter Vektoren bei der Beleuchtungs-Rechnung demonstriert werden. Es wird die Berechnung der von einer Lichtquelle auf ein Oberflächenstück einfallende Lichtintensität beschrieben. Abbildung 3-1 zeigt den geometrischen Aufbau als Grundlage für die Berechnung.

Abbildung 3-1 zeigt den Einfall eines Lichtstrahls $\vec{l}$ auf eine ebene Oberfläche mit dem Flächeninhalt A. Der Lichtvektor zeigt dabei zur Lichtquelle hin. $\vec{n}$ ist der Normalenvektor auf die Oberfläche. Die Skizze dient als Grundlage der Berechnung.



Es gelten die folgenden Festlegungen:

$$\vec{n}, \vec{l} \in \mathbb{R}^3;\ A \in \mathbb{R}^+;\ \alpha \in [0°; 90°] \tag{3-5}$$

Der Vektor $\vec{n}$ steht senkrecht auf der Oberfläche.

Gegeben seien die Vektoren $\vec{n}$ und $\vec{l}$ und der Flächeninhalt A, nicht aber der Winkel α. Für die Berechnung der Lichtintensität $I \in \mathbb{R}_0^+$ ist nur die Projektion des Oberflächenstücks auf eine zum Lichtvektor $\vec{l}$ senkrechte Ebene relevant (Nischwitz et al. 2019: S. 333). Dabei besteht folgender Zusammenhang:

$$I \sim A_\perp \tag{3-6}$$

Der Abbildung 3-1 kann entnommen werden, dass sich die projizierte Fläche $A_\perp$ wie folgt berechnen lässt:

$$A_\perp = A \cdot \cos(\alpha) \tag{3-7}$$

Bekanntlich kann der Winkel zwischen zwei Vektoren mithilfe des Skalarprodukts ermittelt werden.

$$\vec{n} \circ \vec{l} = |\vec{a}| \cdot \left|\vec{l}\right| \cdot \cos(\alpha) \tag{3-8}$$

Sind die beiden Vektoren normalisiert, und haben daher die Länge 1, gilt folgender Zusammenhang (Nischwitz et al. 2019: S. 333):

$$\begin{gathered} \vec{n} \circ \vec{l} = \cos(\alpha) \\ \Rightarrow A_\perp = A \cdot \left(\vec{n} \circ \vec{l}\right) \end{gathered} \tag{3-9}$$

Auf diese Weise lässt sich sehr einfach, allein mit Multiplikation, die Lichtintensität berechnen.

3.3 Notwendigkeit des Algorithmus

Das weite Feld der Computer-Grafik kann in die Echtzeit- bzw. Interaktive- und Nichtechtzeit-Computer-Grafik eingeteilt werden. Der Unterschied liegt dabei in der benötigten Zeit für die Bildberechnung. Für das Exportieren eines Bildes aus einer CAD-Anwendung ist beispielsweise die Dauer der Bildgenerierung nicht weiter relevant. Bei interaktiven Anwendungen allerdings (z.B. einem Flugsimulator) ist es essenziell, dass zwischen einer Benutzereingabe und einer Reaktion der Anwendung nicht zu viel Zeit vergeht (Nischwitz et al. 2019: S. 25f.). Üblicherweise spricht man ab einer Bildwiederholrate von 1 Hz von interaktiver Computer-Grafik. Von einem flüssigen Bild kann man aber erst ab ca. 30 Hz sprechen. Das bedeutet das pro Sekunde 30 Bilder berechnet werden müssen (Nischwitz et al. 2019: S. 26f.).

Berechnungen wie die aus Kapitel 3.2 müssen dabei für jeden Vertex (Eckpunkt eines Objekts) durchgeführt werden. Bei einer entsprechend hohen Anzahl an Vertices müssen dafür Vektoren

millionenfach pro Sekunde normalisiert werden. Diese Art der Berechnungen stellen einen Flaschenhals in der Computergrafik dar (Aryeh 2020: TC. 00:02:24 – 00:03:00).

Im Vergleich zur Addition und Multiplikation sind Operationen wie Division und Wurzelziehen auch für Computersysteme vergleichsweise zeitaufwändig (Noe 2019). Berechnungen wie die der Funktion f aus (2-1) in sehr großer Anzahl durchzuführen, erfordert deshalb einen effizienten Algorithmus. Der FISR-Algorithmus stellt ein solches ausgeklügeltes Verfahren dar, das speziell für die Computergrafik entworfen und eingesetzt wurde (Aryeh 2020: TC. 00:03:00 – 00:03:20).

4 Darstellung von Zahlen in Computersystemen

Computersysteme stellen Zahlen intern nicht in dem am meisten verwendeten Stellenwertsystem, dem Dezimalsystem, sondern im Binärsystem (auch Dualsystem genannt) dar (Hesse 2020: S. 35). Ein Stellenwertsystem definiert dabei die Anzahl und Art der verwendeten Ziffern. Das Dezimalsystem kennt die zehn Ziffern 0 – 9, das Binärsystem nur die beiden Ziffern 0 und 1 (Goll/Dausmann 2014: S. 701).

Die Anzahl der Ziffern eines Stellenwertsystems wird als Basis $b \in \mathbb{N}$ bezeichnet. Der Ziffernvorrat (Menge aller möglichen Ziffern) wird V genannt. Für das Binärsystem gilt $b = 2$ und $V = \{0; 1\}$.

4.1 Natürliche Zahlen

Jede Zahl besteht aus einer oder mehreren Stellen. In Computersystemen werden diese Bits genannt. Für die Anzahl dieser Stellen gilt: $m \in \mathbb{N}$. Eine Stelle enthält dabei genau eine Ziffer aus dem Ziffernvorrat V. Die Stellen einer Zahl unterscheiden sich dabei in ihrer Wertigkeit s_n. Für die Stelle $n \in \mathbb{N}_0$ gilt (Goll/Dausmann 2014: S. 142):

$$s_n = 2^n \tag{4-1}$$

Um den Wert w_n einer konkreten Stelle in einer Zahl zu berechnen wird der Wert der Ziffer a_n mit der Wertigkeit der Stelle s_n multipliziert (Goll/Dausmann 2014: S. 142).

$$w_n = a_n \cdot s_n = a_n \cdot 2^n \tag{4-2}$$

Schlussendlich ergibt sich der Wert z einer natürlichen Zahl aus der Addition der Werte aller ihrer Stellen (Goll/Dausmann 2014: S. 142).

$$z = \sum_{i=0}^{m-1} w_i = \sum_{i=0}^{m-1} a_i \cdot 2^i \tag{4-3}$$

4.2 Fließkommazahlen

Der FISR-Algorithmus setzt die Darstellung von rationalen Zahlen als Fließkommazahlen nach dem Standard IEEE 754 voraus (Lomont 2003: S. 2). Deshalb wird im Folgenden der Aufbau einer solchen Zahl beschrieben.

4.2.1 Abgrenzung zu Festkommazahlen

Zur Darstellung rationaler Zahlen in Computersystemen existieren verschiedene Konzepte. Die wichtigsten und bekanntesten Konzepte sind die der Fließkommazahlen (auch Gleitkommazahlen) und der Festkommazahlen. Beide Konzepte speichern dabei den Anteil der Zahl vor dem Komma getrennt von dem Anteil nach dem Komma.

Alle Festkommazahlen desselben Typs haben eine fest definierte Anzahl an Vor- und Nachkommastellen. Das Komma befindet sich an einer festen Stelle (Böhme 2013: S. 13).

Fließkommazahlen dagegen speichern zusätzlich, an welcher Stelle sich das Komma befindet. Mathematisch führt das zur sogenannten Exponentialschreibweise (auch wissenschaftliche Schreibweise) (Böhme 2013: S. 16). Beispielsweise wird der Betrag der Lichtgeschwindigkeit c in Dezimalschreibweise in der Physik oft als Gleitkommazahl angegeben.

$$c = 2{,}99792458 \cdot 10^8 \frac{m}{s} \tag{4-4}$$

$2{,}99792458$ ist hierbei die Mantisse, 10 die Basis des Stellenwertsystems und 8 der Exponent. Außerdem liegt hier die Mantisse in ihrer normalisierten Form vor. Eine Mantisse $m \in \mathbb{Q}_0^+$ ist in einem Stellenwertsystem mit der Basis b normalisiert, wenn gilt (Wikipedia (Hg.) 2022a):

$$1 \leq m < b \tag{4-5}$$

4.2.2 Binäre Fließkommazahlen

Eine ähnliche Darstellung lässt sich auch für das Binärsystem umsetzen. Eine Fließkommazahl besteht dabei aus einer Mantisse $m \in \mathbb{Q}_0^+$ und einem Exponenten $e \in \mathbb{Z}$. Der Wert der rationalen Zahl x berechnet sich daher wie folgt (Moroz et al. 2016: S. 2):

$$x = m \cdot 2^e \tag{4-6}$$

In Computersystemen steht zur Speicherung von Mantisse und Exponent jeweils nur eine begrenzte Anzahl an Stellen (Bits) zur Verfügung. Ein Standard legt deshalb fest, welche Anzahl an Bits jeder Teil hat und wie diese zu interpretieren sind. Abbildung 4-1 zeigt den Aufbau einer 32-Bit Fließkommazahl nach IEEE 754 (Goll/Dausmann 2014: S. 144).

S	E	M
1	8	23

$$S \in \{0;1\};\ E \in \{x | x \in \mathbb{N}_0 \wedge x < 2^8\};\ M \in \{x | x \in \mathbb{N}_0 \wedge x < 2^{23}\}$$

Abbildung 4-1 zeigt den Aufbau einer 32-Bit Fließkommazahl nach IEEE 754.
Zeile 1 gibt den Namen des Bestandteils, Zeile 2 die Anzahl der Bits für diesen Bestandteil an. Jeder Bestandteil ist dabei als natürliche binäre Zahl gespeichert.
S: Sign-Bit (Vorzeichen-Bit)
E: Exponent
M: Mantisse

Das Sign-Bit gibt das Vorzeichen der Fließkommazahl an. Der FISR-Algorithmus operiert ausschließlich auf positiven Zahlen (Moroz et al. 2016: S. 3). Im Folgenden wird daher stets $S = 0$ angenommen und das Sign-Bit deshalb nicht in die Berechnungen mit einbezogen.

Der Standard IEEE 754 kennt zwar auch nicht normalisierte Mantissen, der FISR-Algorithmus setzt jedoch eine normalisierte Mantisse voraus (Aryeh 2020: TC. 00:09:07 – 00:09:40). Auf die Beschreibung der denormalisierten Gleitkommazahlen wird deshalb verzichtet. Für eine normalisierte Mantisse m gilt nach (4-5) im Binärsystem:

$$1 \leq m < 2 \tag{4-7}$$

Die erste Stelle der Mantisse im Binärsystem ist damit immer 1. Daher muss diese nicht mitgespeichert werden (Goll/Dausmann 2014: S. 146). m berechnet sich daher aus M wie folgt:

$$m = 1 + \frac{M}{2^{23}} \tag{4-8}$$

Um auch negative Exponenten e und damit sehr kleine Fließkommazahlen darstellen zu können, wird der Exponent E um ein sog. Bias verkleinert (Goll/Dausmann 2014: S. 145).

$$e = E - 127 \tag{4-9}$$

Für den Wert einer positiven normalisierten 32-Bit Fließkommazahl x nach IEEE 754 gilt somit insgesamt:

$$x = \left(1 + \frac{M}{2^{23}}\right) \cdot 2^{E-127} \tag{4-10}$$

Das Bitmuster einer Fließkommazahl nach Abbildung 4-1 kann auch als natürliche Zahl z mit 32 binären Stellen interpretiert werden (Moroz et al. 2016: S. 3). Diese Uminterpretation erscheint zwar auf den ersten Blick nicht sinnvoll, jedoch macht sich der FISR-Algorithmus genau diese zu Nutze.

$$z = E \cdot 2^{23} + M \tag{4-11}$$

Das erläuterte Konzept lässt sich genauso auf 64-Bit Gleitkommazahlen übertragen, die ebenfalls im Standard IEEE 754 beschrieben werden (Lomont 2003: S. 2). Die Konstanten für die Berechnung können dem Standard entnommen werden. Aus Gründen der Übersichtlichkeit wird jedoch in allen folgenden Darlegungen auf 64-Bit Gleitkommazahlen verzichtet.

4.2.3 Menge der Fließkommazahlen

Die Menge der Fließkommazahlen stellt eine Teilmenge der rationalen Zahlen dar. Im Folgenden wird daher die Menge F aller positiven normalisierten 32-Bit Fließkommazahlen nach IEEE 754 definiert, auf die in den folgenden Kapiteln immer wieder Bezug genommen wird. Der Exponent E darf dabei weder seinen minimalen Wert 0 noch seinen maximalen Wert $2^8 - 1$ annehmen, da es sich in diesen Fällen nicht um eine normalisierte Zahl handelt (Goll/Dausmann 2014: S. 145).

$$\begin{gathered} F = \left\{\left(1 + \frac{M}{2^{23}}\right) \cdot 2^{E-127} | M \in A; E \in B\right\} \\ A = \{x | x \in \mathbb{N}_0 \wedge x < 2^{23}\} \\ B = \{x | x \in \mathbb{N} \wedge x < 2^8 - 1\} \end{gathered} \tag{4-12}$$

5 Approximation der binären Logarithmus-Funktion

Der Geschwindigkeitsvorteil des FISR-Algorithmus liegt insbesondere in der Genauigkeit der ersten Näherung an den Funktionswert $f(x)$ aus (2-1) (Moroz et al. 2016: S. 1f.). Dabei wird eine Linearisierung einer binären Logarithmusfunktion verwendet.

5.1 Linearisierung

Gegeben seien das Intervall I, die Funktion g und die Funktionenscharen h_μ und d_μ mit $\mu \in \mathbb{R}$ (Aryeh 2020: TC. 00:10:50 – 00:11:10).

$$I = [0; 1] \tag{5-1}$$

$$g(x) = \log_2(x + 1) \tag{5-2}$$

$$h_\mu(x) = x + \mu \tag{5-3}$$

$$d_\mu(x) = g(x) - h_\mu(x) = \log_2(x + 1) - x - \mu = \frac{\ln(x + 1)}{\ln(2)} - x - \mu \tag{5-4}$$

$$D_g = D_{h_\mu} = D_{d_\mu} = I \tag{5-5}$$

Satz 5-1: Für $\mu \in [0; 2\mu_0]$ gilt die folgende Approximation:

$$g(x) \approx h_\mu(x), \forall\, x \in I \Leftrightarrow d_\mu(x) \approx 0, \forall\, x \in I$$

Für

$$\mu = \mu_0 = \frac{1}{2} - \frac{1 + \ln(\ln(2))}{2 \cdot \ln(2)}$$

ist die maximale Abweichung von $d_\mu(x)$ zu 0 minimal.

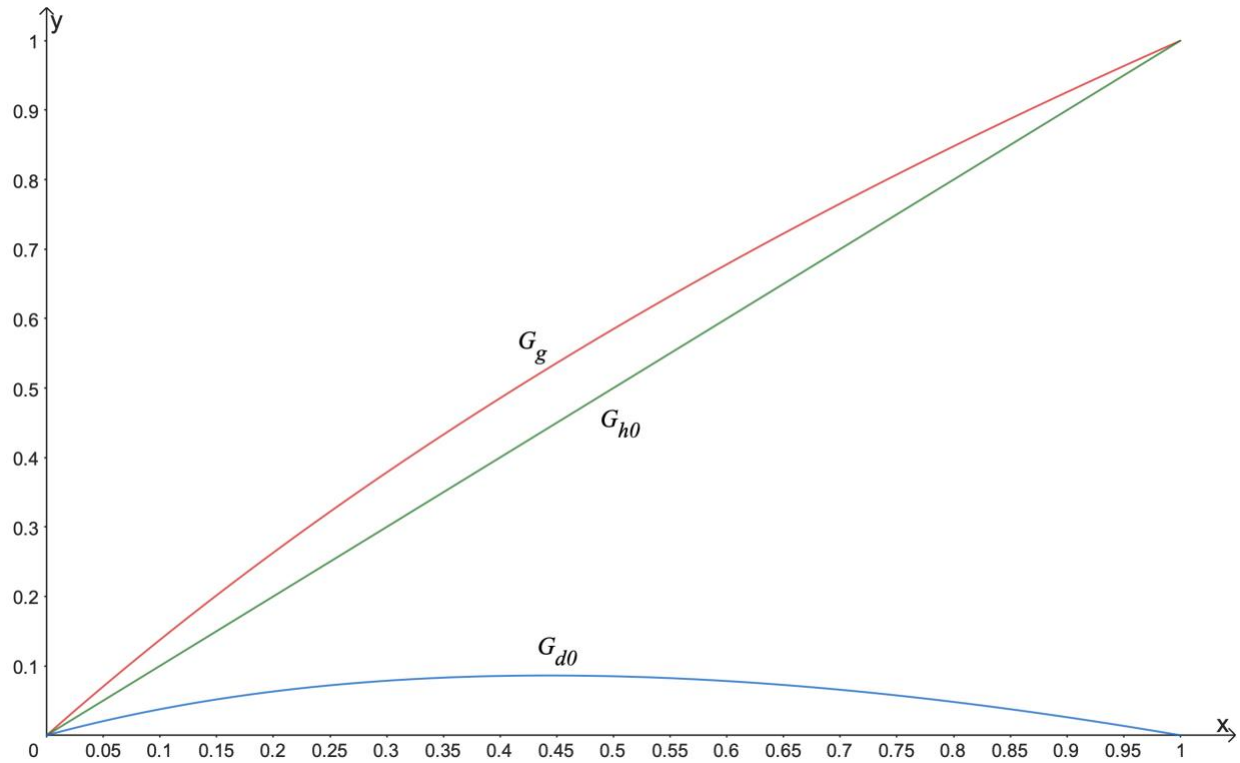


Abbildung 5-1 zeigt die Graphen G_g, G_{h_0} und G_{d_0} der Funktionen g, h_0 und d_0 für $\mu = 0$.
G_{d_0} hat Nullstellen für $x = 0 \vee x = 1$. Dazwischen erreicht der Graph seinen absoluten Hochpunkt.
Für $\mu > 0$ wird der Graph der Funktion h_0 nach oben, und damit der Graph der Funktion d_0 nach unten verschoben.

Beweis:
Es gilt zunächst $\mu = 0$. Das geübte Auge erkennt schnell, dass die Ränder der Definitionsmenge von d_0 die beiden Nullstellen der Funktion sind (Aryeh 2020: TC. 00:11:00 – 00:11:05).

$$d_0(x) = 0 \Leftrightarrow \log_2(x+1) - x = 0 \Leftrightarrow x = 0 \vee x = 1$$
$$x_{min,1} = 0;\ x_{min,2} = 1 \tag{5-6}$$

Die beiden Nullstellen sind die absoluten Tiefpunkte des Graphen von G_{d_0}. Zwischen den beiden Nullstellen erreicht die Funktion für $x_{max} \in I$ ihren Maximalwert. Die Extremstellen können allgemein durch das Monotonieverhalten der Funktion d_μ nachgewiesen werden.

$$d'_\mu(x) = \frac{\frac{1}{x+1}}{\ln(2)} - 1 = \frac{1}{\ln(2)\cdot(x+1)} - 1$$
$$D_{d'_\mu} =]0;1[\tag{5-7}$$

$$d'_\mu(x_{max}) = 0 \Leftrightarrow \frac{1}{\ln(2)\cdot(x_{max}+1)} - 1 = 0 \Leftrightarrow \ln(2)\cdot(x_{max}+1) = 1$$
$$\Leftrightarrow \ln(2)\cdot x_{max} + \ln(2) = 1 \tag{5-8}$$
$$\Leftrightarrow x_{max} = \frac{1}{\ln(2)} - 1$$

Da es sich bei der gefundenen Nullstelle x_{max} der Ableitung um eine einfache Nullstelle handelt, wechselt die Funktion d'_μ an der Nullstelle ihr Vorzeichen. Für das Monotonieverhalten gilt daher:

$$d_\mu\ ist\ s.m.\ zunehmend\ in\ ]0; x_{max}]$$
$$d_\mu\ ist\ s.m.\ abnehmend\ in\ [x_{max}; 1[\tag{5-9}$$

Für $\mu > 0$ wird der Graph der Funktion d_0 lediglich nach unten verschoben. Damit die maximale Abweichung von $d_\mu(x)$ zu 0 minimal wird, muss daher folgende Bedingung erfüllt sein:

$$-d_\mu(x_{min,1}) = d_\mu(x_{max})$$
$$\Leftrightarrow -\left(\frac{\ln(0+1)}{\ln(2)} - 0 - \mu\right) = \frac{\ln\left(\frac{1}{\ln(2)} - 1 + 1\right)}{\ln(2)} - \frac{1}{\ln(2)} + 1 - \mu \tag{5-10}$$
$$\Leftrightarrow \mu = \frac{\ln(1) - \ln(\ln(2)) - 1}{\ln(2)} + 1 - \mu$$

$$\Leftrightarrow 2\mu = 1 - \frac{1 + \ln(\ln(2))}{\ln(2)}$$

$$\Rightarrow \mu = \mu_0 = \frac{1}{2} - \frac{1 + \ln(\ln(2))}{2 \cdot \ln(2)} \approx 0{,}043$$

Als minimierte maximale Abweichung $\hat{d}$ der Funktion d_{μ_0} zu 0 ergibt sich somit:

$$\begin{aligned} \hat{d} = |d_{\mu_0}(0)| &= |\log_2(0+1) - 0 - \left[\frac{1}{2} - \frac{1 + \ln(\ln(2))}{2 \cdot \ln(2)}\right]| = \frac{1}{2} - \frac{1 + \ln(\ln(2))}{2 \cdot \ln(2)} \\ &= \mu_0 \approx 0{,}043 \end{aligned} \tag{5-11}$$

Diese maximale Abweichung ist hinreichend genau für die Approximation aus Satz 5-1.

■

5.2 Der binäre Logarithmus einer Fließkommazahl

Mithilfe der gefundenen Approximation aus Satz 5-1 lässt sich der binäre Logarithmus einer Fließkommazahl annähernd berechnen (Willberger 2019).

Satz 5-2: Sei $x \in F$ eine Fließkommazahl nach (4-12) und z die Interpretation von x als natürliche Zahl nach (4-11), so gilt nach Anwendung von Satz 5-1 die folgende Approximation:

$$\log_2(x) \approx \frac{1}{2^{23}} \cdot z + \mu - 127$$

Beweis:
Zunächst wird die Zahl x in die Funktion des binären Logarithmus eingesetzt und der entstehende Term vereinfacht. Dabei wird der Term zur Berechnung einer Fließkommazahl aus (4-10) verwendet (Aryeh 2020: TC. 00:10:30 – 00:10:40).

$$\log_2(x) = \log_2\left[\left(1 + \frac{M}{2^{23}}\right) \cdot 2^{E-127}\right] = \log_2\left(1 + \frac{M}{2^{23}}\right) + E - 127 \tag{5-12}$$

Der übriggebliebene Logarithmus-Term entspricht der Funktion g aus (5-2). Auch die Definitionsmenge der Funktion g wird eingehalten, da nach (4-7) und (4-8) stets gilt:

$$0 \leq \frac{M}{2^{23}} < 1 \tag{5-13}$$

Daher lässt sich nun auch die Approximation aus Satz 5-1 für den Logarithmus-Term einsetzen (Willberger 2019).

$$\log_2(x) \approx \frac{M}{2^{23}} + \mu + E - 127$$

$$\Leftrightarrow \log_2(x) \approx \frac{1}{2^{23}} \cdot (E \cdot 2^{23} + M) + \mu - 127 \tag{5-14}$$

Der ausgeklammerte Term entspricht der natürlichen Zahl z. Der Logarithmus von x lässt sich daher direkt mit z approximieren.

$$\log_2(x) \approx \frac{1}{2^{23}} \cdot z + \mu - 127 \tag{5-15}$$

■

6 Das Newton-Verfahren

Das Newton-Verfahren ist ein iterativer Algorithmus zur Bestimmung von Nullstellen stetig differenzierbarer Funktionen. Ausgehend von einer Stelle x_0, die sich in der Nähe einer Nullstelle der Funktion befindet, wird in jedem Schritt (Iteration) des Algorithmus eine Annäherung x_n an die Nullstelle berechnet. Das Ergebnis des vorherigen Schritts wird dabei für den nächsten Schritt verwendet. Die Nullstelle wird deshalb nicht exakt, sondern nur näherungsweise bestimmt (Approximation) (Hesse 2020: S. 53f.).

6.1 Grafische Herleitung

Sei $f: \mathbb{R} \mapsto \mathbb{R}$ eine stetig differenzierbare Funktion mit ihrem Graphen G_f und sei $z \in \mathbb{R}$ eine Nullstelle der Funktion f.

Das Newton-Verfahren nähert sich einer Nullstelle durch Linearisierung der Funktion f an der Stelle x_0 an. Linearisierung bedeutet, dass die Funktion f durch eine lineare Funktion g um die Stelle x_0 angenähert wird. Die Linearisierung entspricht der Tangente an den Graphen G_f an der Stelle x_0. Für die linearisierte Funktion gilt daher (Hesse 2020: S. 53):

$$g(x) = f(x_0) + f'(x_0) \cdot (x - x_0)$$

$$D_g = \mathbb{R} \tag{6-1}$$

Aufgrund der Annäherung des Graphen G_g der Funktion g an den Graphen G_f, liegt auch die Nullstelle von G_g in der Nähe von z. Durch Berechnung der Nullstelle x_1 von g wird damit eine Annäherung an die Nullstelle von f berechnet (Hesse 2020: S. 53).

$$g(x_1) = 0 \Leftrightarrow f(x_0) + f'(x_0) \cdot (x_1 - x_0) = 0 \Leftrightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{6-2}$$

Wurde x_0 richtig gewählt, liegt x_1 nun näher an z. Der vorherige Schritt kann nun mit x_1 als Ausgangswert wiederholt werden, wodurch eine höhere Genauigkeit des berechneten Wertes erreicht wird. Der Wert der vorherigen Iteration wird jeweils für die nächste Iteration verwendet. Allgemein gilt damit für die jeweils nächste Iteration (Hesse 2020: S. 54):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)};\ n \in \mathbb{N}_0 \tag{6-3}$$

Im günstigsten Fall konvergiert das Verfahren so gegen z, was bedeutet, dass sich die Werte von x_n asymptotisch der Nullstelle annähern. Allerdings gibt es auch Fälle, in denen das Verfahren gegen eine andere Nullstelle von f konvergiert oder zwischen zwei Nullstellen pendelt (Oszillation) (Wikipedia (Hg.) 2022b). Für den FISR-Algorithmus sind diese Fälle allerdings nicht von Bedeutung, weswegen keine weiteren Erläuterungen dazu erfolgen.

6.2 Berechnung von Termen

Mithilfe des Newton-Verfahrens können Terme berechnet werden, die sich als Nullstelle einer stetig differenzierbaren Funktion darstellen lassen. Für den FISR-Algorithmus von Bedeutung sind die Nullstellen der folgenden Funktionenschar (Aryeh 2020: TC. 00:19:10 – 00:19:20):

$$\begin{gathered} p_a(x) = \frac{1}{x^2} - a \\ D_{p_a} = \mathbb{R} \setminus \{0\};\ a \in \mathbb{R}^+ \end{gathered} \tag{6-4}$$

Die Nullstellen der Funktionenschar in Abhängigkeit des Parameters a lassen sich leicht darstellen.

$$p_a(x) = 0 \Leftrightarrow \frac{1}{x^2} - a = 0 \Leftrightarrow x = \pm\frac{1}{\sqrt{a}} \tag{6-5}$$

Ist bereits eine Näherung für den Term $\frac{1}{\sqrt{a}}$ bekannt, lässt sich mit wenigen Iterationen des Newton-Verfahrens ein wesentlich genauerer Wert berechnen.

7 Der Algorithmus

Dieses Kapitel hat zum Zweck, den FISR-Algorithmus auf Grundlage der Erkenntnisse aus den vorangegangenen Kapiteln herzuleiten, zu erklären und zu beweisen. Gegeben sei dafür zunächst die Funktion f aus (2-1) mit der Definitionsmenge $D_f = F$ aus (4-12). Ziel des Algorithmus ist die Berechnung einer möglichst genauen Approximation $y \in F$ an $f(x)$ für ein gegebenes x der Definitionsmenge.

Der Ablauf des Algorithmus kann in zwei Teilbereiche aufgeteilt werden (Lomont 2003: S. 3):

1. Berechnen einer ersten Annäherung $y_0 \in F$ an $f(x)$
2. Berechnen weiterer Annäherungen an $f(x)$ durch Iterationen des Newton-Verfahrens. Dabei dient y_0 als Startwert für die erste Iteration.

7.1 Berechnen der ersten Annäherung

Es werden folgende Festlegungen getroffen: $x \in F$ und $y_0 \in F$ sind Fließkommazahlen nach (4-12). Ihre Mantissen und Exponenten heißen M_x und E_x bzw. M_{y0} und E_{y0}. Die Zahlen z_x und z_{y0} sind die Interpretationen von x und y_0 als natürliche Zahlen nach (4-11).

Satz 7-1: Ist die Approximation aus Satz 5-2 anwendbar, so gilt:

$$z_{y0} = c - \frac{1}{2} \cdot z_x$$

$$\Leftrightarrow E_{y0} \cdot 2^{23} + M_{y0} = c - \frac{1}{2} \cdot (E_x \cdot 2^{23} + M_x)$$

$$c = 3 \cdot 2^{22} \cdot (127 - \mu)$$

Für die Interpretation des Bitmusters von z_{y0} als Fließkommazahl y_0 gilt:

$$y_0 \approx f(x)$$

Beweis:
Es wird zunächst von der zu beweisenden Gleichung ausgegangen.

$$y_0 \approx f(x) \Leftrightarrow y_0 \approx \frac{1}{\sqrt{x}} \Leftrightarrow y_0 \approx x^{-\frac{1}{2}} \tag{7-1}$$

Anschließend wird der binäre Logarithmus auf beide Seiten angewandt und die Gleichung vereinfacht (Aryeh 2020: TC. 00:15:50 – 00:16:10).

$$\log_2(y_0) \approx \log_2\left(x^{-\frac{1}{2}}\right) \Leftrightarrow \log_2(y_0) \approx -\frac{1}{2} \cdot \log_2(x) \tag{7-2}$$

Da auf beiden Seiten der Gleichung der binäre Logarithmus einer Fließkommazahl stehen bleibt, kann die Approximation aus Satz 5-2 eingesetzt werden (Willberger 2019). Die entstehende Gleichung dient zur Berechnung von z_{y0}. Daher wird nun das Gleichheitszeichen verwendet.

$$\frac{1}{2^{23}} \cdot z_{y0} + \mu - 127 = -\frac{1}{2} \cdot \left(\frac{1}{2^{23}} \cdot z_x + \mu - 127\right) \tag{7-3}$$

Die Gleichung kann nun nach z_{y0} umgestellt werden (Willberger 2019).

$$\frac{1}{2^{23}} \cdot z_{y0} = -\frac{1}{2} \cdot \left(\frac{1}{2^{23}} \cdot z_x + \mu - 127\right) - \mu + 127$$

$$\Leftrightarrow z_{y0} = -2^{22} \cdot \left(\frac{1}{2^{23}} \cdot z_x + \mu - 127\right) - \mu \cdot 2^{23} + 127 \cdot 2^{23} \tag{7-4}$$

$$\Leftrightarrow z_{y0} = -\frac{1}{2} \cdot z_x - \mu \cdot 2^{22} + 127 \cdot 2^{22} - \mu \cdot 2^{23} + 127 \cdot 2^{23}$$

Die konstanten Terme auf der rechten Seite der Gleichung können als Konstante c ausgelagert werden (Aryeh 2020: TC. 00:17:00 – 00:17:20).

$$z_{y0} = c - \frac{1}{2} \cdot z_x$$

$$c = -\mu \cdot 2^{22} + 127 \cdot 2^{22} - \mu \cdot 2^{23} + 127 \cdot 2^{23} \tag{7-5}$$

$$= 127 \cdot (2^{22} + 2^{23}) - \mu \cdot (2^{22} + 2^{23}) = (2^{22} + 2^{23}) \cdot (127 - \mu)$$

$$= 2^{22} \cdot (1 + 2) \cdot (127 - \mu) = 3 \cdot 2^{22} \cdot (127 - \mu)$$

■

Für die Berechnung von y_0 mithilfe des gezeigten Verfahrens ist es wichtig zu verstehen, dass durch die Gleichung aus Satz 7-1 nicht y_0 direkt, sondern lediglich die natürliche Zahl z_{y0} berechnet wird.

Um im Dezimalsystem die Komponenten von y_0 (E_{y0} und M_{y0}) zu erhalten, müsste die Gleichung zuerst nach diesen umgestellt und diese einzeln berechnet werden. Der Standard IEEE 754 stellt jedoch für das Binärsystem sicher, dass die Bitfolge von z_{y0} genau der Bitfolge von y_0 entspricht. Dies wurde bereits in (4-11) gezeigt. Für Computersysteme entsteht so kein zusätzlicher Rechenaufwand, da die ohnehin berechnete natürliche Zahl z_{y0} einfach als Fließkommazahl interpretiert werden kann.

7.2 Anwendung des Newton-Verfahrens

Wie bereits in Kapitel 6.2 gezeigt, lassen sich die Werte der Funktion f als Nullstellen der Funktionenschar p_a aus (6-4) darstellen. Da die Variable x bereits als Argument für den Algorithmus als Ganzes verwendet wird, wird die Funktion p_x wie folgt definiert (Aryeh 2020: TC. 00:19:10 – 00:19:20):

$$p_x(y) = \frac{1}{y^2} - x$$
$$D_{p_x} = \mathbb{R} \setminus \{0\};\ x \in F \tag{7-6}$$

Für die Ableitung von p_x gilt damit:

$$p_x'(y) = -\frac{2y}{y^4} = -\frac{2}{y^3}$$
$$D_{p_x'} = \mathbb{R} \setminus \{0\} \tag{7-7}$$

Für eine Newton-Iteration gilt nach (6-3):

$$y_{n+1} = y_n - \frac{p_x(y_n)}{p_x'(y_n)};\ n \in \mathbb{N}_0$$

$$\Leftrightarrow y_{n+1} = y_n - \frac{\frac{1}{y_n^2} - x}{-\frac{2}{y_n^3}} = y_n + \frac{\left(\frac{1}{y_n^2} - x\right) \cdot y_n^3}{2} = y_n + \frac{y_n - x \cdot y_n^3}{2} = \frac{3}{2} y_n - \frac{x \cdot y_n^3}{2} \tag{7-8}$$

$$= \frac{1}{2} y_n \cdot (3 - x \cdot y_n^2)$$

Als Startwert für die erste Iteration wird die Approximation y_0 aus Kapitel 7.1 verwendet. Danach können weitere Iterationen durchgeführt werden, um ein genaueres Ergebnis zu erhalten.

8 Effizienz der Berechnungen

Wie bereits in Kapitel 3.3 beschrieben, soll der FISR-Algorithmus einen Geschwindigkeitsvorteil gegenüber der herkömmlichen Berechnung bieten. Dieser Vorteil des Algorithmus kommt allerdings auf Kosten der Genauigkeit zustande (McEniry 2007: S. 1). Im folgenden Kapitel wird daher die Effizienz des Algorithmus diskutiert.

8.1 Geschwindigkeit

Zur Zeit der Entwicklung des Algorithmus war er ca. vier Mal so schnell wie eine Standardimplementierung der Berechnung in vielen Programmiersprachen (Lomont 2003: S. 1). Die hohe Performance des Verfahrens kommt vor allem dadurch zustande, dass ein Computersystem für die Berechnung keinerlei Division durchführen muss (Aryeh 2020: TC. 00:02:40 – 00:03:00).

Dass Divisionen langsamer sind als Multiplikationen, liegt vor allem daran, dass Divisionen nur schwer parallelisiert werden können. Für die Multiplikation gelten sowohl das Assoziativ- als auch das Kommutativgesetz. Diese Gesetze gelten nicht für die Division. So kann die Multiplikation zweier Zahlen in viele kleinere Multiplikationen und Additionen zerlegt werden, die parallel berechnet werden können. Bei der Division ist das nicht ohne weiteres möglich (Noe 2019).

In der Formel für eine Newton-Iteration aus (7-8) kommt zwar mit $^1/_2$ ein Bruch vor. Da es sich allerdings um einen konstanten Faktor handelt, kann dieser problemlos durch eine Multiplikation mit $0{,}5$ ersetzt werden.

Der Bruch $^1/_2$ kommt auch in der Formel zur Berechnung von z_{y0} aus Satz 7-1 vor. Dabei geht es allerdings um die Berechnung einer natürlichen Zahl, so dass sich hier anders beholfen werden kann. Wird bei einer Dezimalzahl das Komma um eine Stelle verschoben, kommt dies einer Multiplikation mit bzw. Division durch 10 gleich. Im Binärsystem führt eine Verschiebung des Kommas zu einer Multiplikation mit bzw. Division durch 2. Mit sogenannten Bitshifts kann somit sehr einfach durch 2 geteilt werden (McEniry 2007: S. 3).

8.2 Genauigkeit

Die Genauigkeit des Verfahrens hängt stark von der Konstante μ aus Satz 5-1 und damit auch von der Konstante c aus Satz 7-1 ab. Obwohl die Konstante aus (5-10) die beste Annäherung an die Logarithmus-Funktion darstellt, ist sie nicht die beste Konstante für den Algorithmus. Das hat den Grund, dass die Annäherung nachfolgende Schritte (z.B. die Newton-Iterationen) nicht berücksichtigt (Wikipedia (Hg.) 2023).

Im Laufe der Zeit wurden deshalb andere Konstanten c von diversen Mathematikern vorgeschlagen. Aufgrund der Größe der Zahlen, werden diese Konstanten üblicherweise im Hexadezimal-Format (Stellenwertsystem mit Basis 16) angegeben. Die folgende Tabelle zeigt mögliche Konstanten und den dadurch entstehenden Fehler. Die Fehlerwerte wurden im Rahmen dieser Seminararbeit selbst ermittelt. Die Konstanten stammen allerdings aus anderen mathematischen Arbeiten. Ihr Ursprung ist in der letzten Spalte angegeben.

c	μ	y_0		y_1		Quelle
		Maximaler Fehler (%)	Durchschn. Fehler (%)	Maximaler Fehler (%)	Durchschn. Fehler (%)	
$5F3759DF_{16}$	0,0450466	3,438	2,327	0,175	0,095	(id Software (Hg.) 2012)
$5F37BCB6_{16}$	0,0430357	3,638	2,444	0,201	0,105	μ_0 aus Satz 5-1
$5F375A86_{16}$	0,0450333	3,437	2,328	0,175	0,095	(Moroz et al. 2016: S. 10)

Abbildung 8-1 zeigt verschiedene Konstanten c und den dadurch entstehenden Fehler über alle positiven normalisierten 32-Bit Fließkommazahlen nach IEEE 754. Die Werte sind gerundet.
y_0 ist die Annäherung nach Satz 7-1.
y_1 ist die Verbesserung durch eine Newton-Iteration.

Man erkennt, dass bereits nach der ersten Newton-Iteration der maximale relative Fehler bei ca. $0{,}2\ \%$ liegt. Für viele Anwendungen in der Computer-Grafik ist das schon ausreichend genau (Willberger 2019).

8.3 Heutige Relevanz

Obwohl der Geschwindigkeitsvorteil zur Zeit der Entwicklung des FISR-Algorithmus groß war, spielt der Algorithmus heute eine eher untergeordnete Rolle. Dies liegt vor allem daran, dass viele Hardware-Hersteller Algorithmen für die Berechnung der reziproken Quadratwurzel in ihre Produkte integriert haben. Die Berechnungen werden dabei von der Hardware selbst unterstützt (McEniry 2007: S. 1). Bei dem FISR-Algorithmus handelt es sich um eine reine Software-Implementierung.

Mit dem Assembler-Befehl `rsqrtss` steht ein solcher Algorithmus heute auf den weit verbreiteten x86-Prozessoren zur Verfügung. Auch dieser Befehl berechnet lediglich eine Approximation an den Wert der reziproken Quadratwurzel. Jedoch schlägt dieser den FISR-Algorithmus in Sachen Geschwindigkeit und Genauigkeit (Elan 2009). Auch für viele ARM-Prozessoren, die vor allem in kleineren Systemen Verwendung finden, steht heute ein ähnlicher Assembler-Befehl zur Verfügung (Arm Limited (Hg.) 2021).

Für Systeme jedoch, die über keine zusätzliche Hardware-Beschleunigung verfügen, kann der Einsatz des Algorithmus auch heute noch sinnvoll sein.

9 Schlussbemerkung

Diese Seminararbeit kratzt lediglich an der Oberfläche der mathematischen Forschungsmöglichkeiten, die der FISR-Algorithmus bietet. Das Verfahren verfügt über weiteres Optimierungspotential. So wurde beispielsweise ein angepasstes Newton-Verfahren eigens für diesen Algorithmus entwickelt. Auch mit der Herleitung verschiedener Konstanten c haben diverse Autoren mathematische Werke gefüllt.

Wenngleich das Verfahren heutzutage in der Computergrafik eine kleinere Rolle spielt, so bleibt es doch ein mathematisch faszinierender Algorithmus. Besonders für Informatikerinnen und Informatiker bietet der Algorithmus die Gelegenheit, über den eigenen Tellerrand hinauszuschauen und in der Mathematik – neben bewährten Konzepten – neue, bessere Lösungen für alte Probleme zu finden.

An dieser Stelle sei noch einmal an die Demo-Website unter https://fisr.schulz-paul.de erinnert. Nach der Lektüre dieser Arbeit stellt sie eine gute Gelegenheit dar, die theoretisch gewonnenen Erkenntnisse praktisch nachzuvollziehen.

10 Abbildungsverzeichnis

11 Literaturverzeichnis

Arm Limited (Hg.), 2021: FRSQRTE*,* https://developer.arm.com/documentation/ddi0596/2021-12/SIMD-FP-Instructions/FRSQRTE--Floating-point-Reciprocal-Square-Root-Estimate-, abgerufen am 15.01.2023

Aryeh, Nemean, 2020: Fast Inverse Square Root — A Quake III Algorithm, https://www.youtube.com/watch?v=p8u_k2LIZyo, abgerufen am 15.01.2023

Böhme, Hans-Joachim, 2013: Grundlagen der Informatik I, https://www2.htw-dresden.de/~boehme/Hans_Wings2013/GI_Ueb_WS2013.pdf, abgerufen am 15.01.2023

Elan, Ruskin, 2009: Timing square root, https://web.archive.org/web/20210208132927/http://assemblyrequired.crashworks.org/timing-square-root/, abgerufen am 15.01.2023

Goll, Joachim/Dausmann, Manfred, 2014: C als erste Programmiersprache, Mit den Konzepten von C11, Wiesbaden

Heinisch, Cornelia/Müller-Hofmann, Frank/Goll, Joachim, 2010: Java als erste Programmiersprache, Vom Einsteiger zum Profi, Wiesbaden

Hesse, Kerstin, 2020: Modellieren und Anwendungen: Numerische Analysis, https://math.uni-paderborn.de/fileadmin/mathematik/Kerstin-Hesse/Numerische_Analysis_Skript_aktualisiert_Nov-2020.pdf, abgerufen am 15.01.2023

id Software (Hg.), 2012: q_match.c, https://github.com/id-Software/Quake-III-Arena/blob/master/code/game/q_math.c#L552, abgerufen am 15.01.2023

Lomont, Chris, 2003: Fast inverse square root, http://www.lomont.org/papers/2003/InvSqrt.pdf, abgerufen am 25.09.2022

McEniry, Charles, 2007: The mathematics behind the fast inverese square root function code*.,* https://web.archive.org/web/20150511044204/http://www.daxia.com/bibis/upload/406Fast_Inverse_Square_Root.pdf, abgerufen am 14.01.2023

Moroz, Leonid/Walczyk, Cezary/Hrynchyshyn, Andriy et al., 2016: Fast calculation of inverse square root with the use of magic constant – analytical approach, https://arxiv.org/pdf/1603.04483.pdf, abgerufen am 22.09.2022

Nischwitz, Alfred/Fischer, Max/Haberäcker, Peter et al., 2019: Computergrafik, Band I des Standardwerks Computergrafik und Bildverarbeitung [E-Book], Wiesbaden

Noe, Marcel, 2019: Why does division of floating point numbers take a lot more time than multiplication of floats in computers?, https://www.quora.com/Why-does-division-of-floating-point-numbers-take-a-lot-more-time-than-multiplication-of-floats-in-computers, abgerufen am 15.01.2023

Wikipedia (Hg.), 2022a: Gleitkommazahl, https://de.wikipedia.org/wiki/Gleitkommazahl, abgerufen am 15.01.2023

Wikipedia (Hg.), 2022b: Newtonverfahren, https://de.wikipedia.org/wiki/Newtonverfahren, abgerufen am 15.01.2023

Wikipedia (Hg.), 2023: Fast inverse square root, https://en.wikipedia.org/wiki/Fast_inverse_square_root, abgerufen am 15.01.2023

Willberger, Thomas, 2019: Wie funktioniert der Fast-Inverse Square Root Algorithmus?, https://de.quora.com/Wie-funktioniert-der-Fast-Inverse-Square-Root-Algorithmus, abgerufen am 15.01.2023