Rechnen

Warum selber rechnen – es gibt doch Computer?

Teilnehmer:

Christian Draeger	Andreas-Oberschule
Dennis Menge	Heinrich-Hertz-Oberschule
Patrick Mikut	Immanuel-Kant-Oberschule
Andre Rozek	Herder-Oberschule
Patrick Klitzke	Andreas-Oberschule
Christopher Schmidt	Immanuel-Kant-Oberschule
Robert Schlämann	Immanuel-Kant-Oberschule

Gruppenleiter:

Falk Ebert	Technische Universität Berlin, Mitglied im DFG-Forschungszentrum MATHEON „Mathematik für Schlüsseltechnologien“

1 Einleitung

Wenn man einmal die Schüler eines Mathematikkurses beobachtet, dann stellt man schnell fest, dass sie ihre Taschenrechner öfter benutzen als ihren Kopf. Wozu man sich auch die Mühe machen, einen Wert wie $(2^{50}+1)-2^{50}$ selbst auszurechnen, wenn der Taschenrechner schon griffbereit liegt?
Warum man seinem Taschenrechner nicht immer trauen kann, haben wir in diesem einwöchigen Kurs erfahren. Dazu muss man ersteinmal wissen, wie ein Rechner überhaupt rechnet, weshalb wir uns zuerst mit der Maschinenarithmetik befasst haben. Weiterhin haben wir die möglichen Rechenoperationen betrachtet, denn für einen Computer sind Addition und Multiplikation mit relativ geringem Aufwand möglich. Wie man aber die Division und die Exponentialfunktion oder auch trigonometrische Funktionen nur auf diese beiden Operationen zurückführen kann, wollen wir in unserem Bericht zeigen. Um dies zu ermöglichen, benutzten wir Approximationen mit dem Newton-Verfahren und mit Taylor-Polynomen. Neben der Möglichkeit, Werte selbst zu errechnen, haben wir gezeigt, wie es durch Interpolation von Tabellen möglich ist, alle Werte exakt und effizient zu bestimmen.

2 Maschinenarithmetik

Es gibt verschiedene Zahlensysteme. Das gängiste für uns ist das Dezimalsystem. In diesem System stehen uns die Ziffern 0 bis 9 zur Verfügung. Einem Computer bzw. einem prozessorgesteuerten Gerät stehen aber nur zwei Zustände zur Verfügung, nämlich an und aus. Diese Zustände werden üblicherweise durch die Ziffern 0 (für aus) und 1 (für an) dargestellt. Das Zahlensystem, das nur aus den Ziffern 0 und 1 besteht wird als Dual- oder auch Binärsystem bezeichnet.

2.1 Darstellung von Dualzahlen

Da die Darstellung und Umrechnung von Ganzzahlen im Dualsystem allgemein bekannt ist, möchten wir in unserem Bericht nur auf die Darstellung von Fließkommazahlen eingehen. Eine typische normierte Fließkommazahl wird folgendermaßen beschrieben:

$$\pm 0,1 m_2 m_3 .. m_{53} \cdot 2^{e_{10} e_9 .. e_0 - 1023}$$

Hierbei handelt es sich um ein 64bit-System. Das erste Bit wird für das Vorzeichen der Dualzahl verwendet: Eine 0 bedeutet, dass die Zahl positiv ist, eine 1 dagegen, dass sie negativ ist. Die Mantisse der Dualzahl beginnt mit 0,1. Dies ist eine Festlegung, um die Eindeutigkeit der Zahlen zu garantieren. Danach folgen 52 Ziffern der Mantisse, welche im Bereich von $0,5$ bis $1-2^{53}$ liegt. Desweiteren

stehen für den Exponenten 11 Bit zur Verfügung. Von der sich daraus ergebene Zahl wird dann 1023 subtrahiert. Der Exponent ist dann eine Ganzzahl, die sich im Bereich von 1024 bis -1023 befindet.
Desweiteren gibt es folgende Festlegungen:

$$\begin{aligned} m_2..m_{53} &= 0 \wedge E = 0 \rightarrow 0 && (1)\\ m_2..m_{53} &= 0 \wedge E = 2047 \rightarrow \pm\infty && (2)\\ \text{mindestens ein } m_i &\neq 0 \wedge E = 2047 \rightarrow NaN(NotaNumber) && (3) \end{aligned}$$

E beizeichnet hierbei die durch e_0 bis e_{10} gebilete Zahl und NaN eine Zahl, die gar keine ist, wie z.B. $\frac{0}{0}$.
Im weiteren Verlauf dieses Kapitels, wird das 8Bit-System verwendet, wo die Mantisse 5 Stellen und der Exponent 2 Stellen besitzt. Damit lassen sich beispielsweise positive Zahlen im Bereich von $0,10000 \cdot 2^{-11} = 0,0625$ bis $0,11111 \cdot 2^{+11} = 7,75$ darstellen.

2.2 Maschinengenauigkeit

Jeder Computer macht beim Rechnen Fehler. Mathematisch bezeichnet man diesen Fehler wie folgt:

$$eps = \frac{b}{2} \cdot b^{-\mu},$$

wobei μ die Anzahl der Stellen der Mantisse und b die Basis des Zahlensystems angibt. Speziell für das Dualsystem gilt:

$$eps = \frac{2}{2} \cdot 2^{-5} \;=\; \frac{1}{32}$$

Im 8 Bit System ist die Maschinengenauigkeit also etwas mehr als eine Dezimalstelle. Allgemein gilt Folgendes:

1. $rd(x) = x(1 + \varepsilon_1)$
2. $x \otimes y = (x \times y)(1 + \varepsilon_2), \quad \times \in \{+, -, \cdot, \div\}$
3. $|\varepsilon_{1/2}| \leq eps$

Dies heißt, dass es sowohl zu Fehlern bei der Darstellung von Dezimalzahlen in dualen Zahlen kommt, als auch bei jeder möglichen Kombination von mehreren Dualzahlen. Diese relativen Fehler können maximal so groß sein, wie die Maschinengenauigkeit.

Beispielumwandlung Nun wollen wir die Maschinengenauigkeit anhand der Umwandlung von π von der dezimalen Zahl in die duale Zahl durch das 8Bit-System demonstrieren. Durch die Maschiengenauigkeit von $\frac{1}{32}$ ergibt sich, dass π zwischen $\frac{31}{32}\pi \approx 3,04$ und $\frac{33}{32}\pi \approx 3,24$ liegt.
Zuerst berechnen wir den Exponenten. Es wird bald ersichtlich, dass π zwischen $0,5 \cdot 2^2 = 2$ und $0,5 \cdot 2^3 = 4$ liegt . Demnach ist der Exponent 2. Man kann dies auch durch den Logarithmus berechnen:

$$\begin{aligned} 0.5 \cdot 2^x &= \pi \\ \log(0.5 \cdot 2^x) &= \log(\pi) \\ \log(0.5) + x \cdot \log(2) &= \log(\pi) \\ x &= \frac{\log(\pi) - \log 0.5}{\log(2)} \\ x &\approx = 2.65 \end{aligned}$$

Hierbei wird abgerundet, da die eigentliche Mantisse größer als 0,5 sein kann.
Nun muss nur noch die Mantisse berechnet werden. Dazu teilen wir die Zahl durch $2^{Exponent}$. Wir erhalten rund 0,7854. Nun wird diese Zahl mit zwei multipliziert. Wenn Sie größer als eins ist, ist die nächste Stelle der Mantisse [1] eins und eins wird abgezogen, anderenfalls null. Dies wird nun solange gemacht, bis alle Mantissenstellen gefüllt sind:

$$\begin{aligned} \pi : 2^2 &\approx 0,7854 \\ 0,7854 \cdot 2 &= 0,5708 + 1 \\ 0,5708 \cdot 2 &= 0,1416 + 1 \\ 0,1416 \cdot 2 &= 0,2832 + 0 \\ 0,2832 \cdot 2 &= 0,5664 + 0 \\ 0,5664 \cdot 2 &= 0,1328 + 1 \\ 0,1328 \cdot 2 &= 0,2656 + 0 \\ 0,11001|0 & \end{aligned}$$

Da das carry-Bit null ist, muss nicht gerundet und auch nicht normalisiert werden. Wir erhalten nun $0,11001 \cdot 2^{10} = 0,7854 \cdot 2^2 = 3.125$. Wir erhalten also 3.125 im 8Bit System für π.

2.3 Addition mit Dualzahlen

Vorgehensweise Um zwei Dualzahlen miteinander addieren zu können, müssen sie den gleichen Exponenten besitzen. Der kleinere wird an den größeren ange-

[1] In unserem Fall die erste

glichen, indem die Mantisse der kleineren Zahl entsprechend oft durch 2 dividiert (also das Komma nach links verschoben) wird. Dieser Vorgang wird 'Shiften' genannt und ist auch in die andere Richtung möglich. Addiert man nun die Mantissen, muss die Dualzahl eventuell normalisiert werden, denn durch Überträge kann diese von der üblichen Darstellungsweise abweichen (z.B. $1, m_2 m_3..$).Dies geschieht ebenfalls durch eine additive Veränderung des Exponenten bei entsprechender Multiplikation bzw. Division der Mantisse mit 2. Anschließend wird die Mantisse gerundet, was mit Hilfe des sogenannten 'carry-Bits' geschieht.
Das carry-Bit wird nur kurzzeitig gespeichert und gibt die 6. Stelle der Mantisse an. Beträgt das carry-Bit 0, so wird abgerundet, anderenfalls aufgerundet. Die Aufrundung kann dazu führen, dass die Mantisse gleich 1 wird und ein weiteres mal normalisiert werden muss.

Beispielrechnung Nehmen wir nun ein einfaches Beispiel: Gegeben seien die zwei Zahlen 3 und 1,4375. Wenn wir diese in Binärzahlen umwandeln, ergibt sich $0,11000 \cdot 2^{10}$ und $0,10111 \cdot 2^{01}$. Die kleinere Zahl wird nun nach rechts verschoben: $0,010111 \cdot 2^{10}$. Nun addieren wir diese:

$$\begin{array}{rl} 0,11000 & 2^{10} \\ \underline{+0,010111} & 2^{10} \\ 1,00011|1 & 2^{10} \end{array}$$

Da nun die Mantisse größer als eins ist, wird sie normaliesiert. Wir verschieben die Mantisse um eins nach rechts[2]und erhöhen den Exponenten um eins: $0,1000111 \cdot 2^{11}$. Da das carry-Bit[3] 1 ist, wird aufgerundet. Daraus ergibt sich : $0,10010 \cdot 2^{11}$. Da die Mantisse nicht gleich eins ist, muss nicht nocheinmal normalisiert werden. Wir erhalten $0,10010 \cdot 2^{11} = 4,5$. Der richtige Wert beträgt 4,4375. Der Fehler entsteht aufgrund der Maschinengenauigkeit.

2.4 Multiplikation von Dualzahlen

Vorgehensweise Die Dualzahlen werden multipliziert, indem die Exponenten miteinander addiert und die Mantissen miteinander multipliziert werden. Wie bei der Addition wird die Dualzahl nun mit einer geeigneten Shiftoperation normalisiert, anschließend gerundet und ggf. ein weiteres mal normalisiert.

[2]Division durch zwei
[3]6. Stelle

Beispielrechnung Auch an einer dieser Stelle ist eine Beispielsrechnung für die Anschalichkeit und dem besseren Verständnis sehr hilfreich.
Aus Gründen der Einfachheit nehmen wir wieder die gleichen Zahlen wie auch beim Additionsbeispiel. Wir multiplizieren 3 mit 1,4375 im 8Bit-System:

$$\begin{aligned} 0,10111 \cdot 0,11000 &= 0,10111 \cdot 0,1 + 0,10111 \cdot 0,01 \\ &= 0,10001|01 \end{aligned}$$

Jetzt muss gerundet werden: Dabei wird die sechste und siebte Stelle der Mantisse abgeschnitten. Es wird nicht aufgerundet, da das carry-Bit null ist. Anschließend müssen die Exponenten noch addiert werden. Es ergibt sich dann $0,10001 \cdot 2^{10+01} = 0,10001 \cdot 2^{11} = 0.53125 \cdot 2^3 = 4.25$. Dieser stimmt gerundet mit der ersten Stelle des tatsächlichen Wertes von 4,3125 überein.

3 Approximation von Funktionswerten

Wir werden im Folgenden einige Techniken zum näherungsweisen Auswerten von analytischen Funktionen entwickeln. Die Standard-Methode dazu ist die Taylorentwicklung. Andere spezielle Funktionswerte - insbesondere die Werte von Umkehrfunktionen - lassen sich mit Hilfe von Nullstellensuch bestimmen. Zu diesem Zweck wird das Newton-Verfahren vorgestellt.

3.1 Taylor-Entwicklung

Wir betrachten eine genügend oft differenzierbare Funktion f in der Nähe des Punktes x_0. Es gilt dann, wenn man genügend nahe an x_0 bleibt, dass $f(x) \approx f(x_0)$. Dabei setzen wir $p(x) = f(x_0)$. Dies ist eine konstante Funktion. Eine bessere Näherung an f in der Nähe von x_0 ist zu erwarten, wenn man statt einer konstanten eine lineare Funktion p_1 zulässt. Dabei fordern wir zusätzlich zu $p_1(x_0) = f(x_0)$, dass $p_1'(x_0) = f'(x_0)$. Diese Gerade ist die Tangente an die Funktion f im Punkt x_0. Eine Gleichung für p_1 ergibt sich mit der Grenzwertbetrachtung

$$\lim_{x \to x_0} \frac{f(x) - p_0(x)}{x - x_0}. \tag{4}$$

Zähler und Nenner dieses Ausdrucks konvergieren gegen 0, da $p_0(x) = f(x_0)$. Mit dem Satz von *L'Hospital* erhält man

$$\lim_{x \to x_0} \frac{f(x) - p_0(x)}{x - x_0} = \lim_{x \to x_0} \frac{f'(x) - 0}{1 - 0} = f'(x_0).$$

Ausmultiplizieren liefert

$$\lim_{x \to x_0} f(x) = \lim_{x \to x_0} f(x_0) + f'(x_0)(x - x_0) \stackrel{\text{def}}{=} \lim_{x \to x_0} p_1(x).$$

Diese Gerade stimmt in x_0 mit f sowohl im Wert als auch in der ersten Ableitung überein. Wir versuchen jetzt eine quadratische Parabel zu finden, die mit f in Wert, erster und zweiter Ableitung übereinstimmt. Dazu betrachten wir die Differenz $f(x) - p_1(x)$. An der Stelle x_0 is diese Differenz null und auch die erste Ableitung. $f(x) - p_1(x)$ hat also einen kritischen Punkt an x_0 und verhält sich in einer kleinen Umgebung wie $\alpha(x-x_0)^2$. Um die Zahl α zu bestimmen, betrachten wir

$$\lim_{x \to x_0} \frac{f(x) - p_1(x)}{(x - x_0)^2}.$$

Die mehrfache Anwendung des Satzes von *L'Hospital* erhält man

$$\begin{aligned} \lim_{x \to x_0} \frac{f(x) - p_1(x)}{(x - x_0)^2} &= \lim_{x \to x_0} \frac{f(x) - (f(x_0) + f'(x_0)(x - x_0))}{(x - x_0)^2} \\ &= \lim_{x \to x_0} \frac{f'(x) - (f'(x_0))}{2(x - x_0)} \\ &= \lim_{x \to x_0} \frac{f''(x)}{2} = \frac{f''(x_0)}{2}. \end{aligned} \tag{5}$$

Ausmultiplizieren liefert

$$\lim_{x \to x_0} f(x) = \lim_{x \to x_0} f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2 \stackrel{\text{def}}{=} \lim_{x \to x_0} p_2(x).$$

Mit der analogen Vorgehensweise wie in (4) und (5) erhält man für beliebige n

$$p_n(x) = \frac{f(x_0)}{0!} + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \ldots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n. \tag{6}$$

Diese Näherung an f in der Nähe von x_0 nennt sich *Taylor-Polynom.*
Die Genauigkeit der Approximation von f durch p_n zeigen wir im Folgenden. Es gilt

$$(f(x) - p_n(x))^{(n+1)} = f^{(n+1)}(x) - p_n^{(n+1)} = 0$$

weil p_n ein Polynom n-ten Grades ist und die $(n+1)$te Ableitung gleich null ist. Weiterhin gilt

$$(f(x) - p_n(x))^{(n)} = \int_{x_0}^{x} (f(x) - p_n(x))^{(n+1)} = (f^{(n)}(x) - f^{(n)}(x_0)) - (p_n^{(n)}(x) - p_n^{(n)}(x_0)).$$

Mit dem Mittelwertsatz der Differentialrechnung ergibt sich

$$(f(x) - p_n(x))^{(n)} = f^{(n+1)}(\xi)(x - x_0) - \underbrace{p_n^{(n+1)}(\xi)}_{} = 0 \tag{7}$$

für ein $\xi \in [x, x_0]$. Die n malige Integration von (7) liefert den Ausdruck

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}.$$

Beispiel
Wir nehmen $f(x) = e^x$ und $x_0 = 0$. Dann ergibt sich mit (6)

$$p_n(x) = 1 + x + \frac{x}{2} + \frac{x}{6} + \ldots + \frac{x^n}{n!}. \tag{8}$$

Und es gilt

$$e^x - p_n(x) = e^\xi \frac{x^{n+1}}{(n+1)!} \leq \max(1, e^\xi) \frac{x^{n+1}}{(n+1)!}.$$

Um e^x mit $x = M \cdot 2^E$ zu bestimmen, kann man wiefolgt vorgehen. Es gilt $|M| \in [\frac{1}{2}, 1)$ und $E \in \mathbb{Z}$.

$$e^x = (e^M)^{2^E}. \tag{9}$$

Es genügt also, die Exponentialfunktion für Werte in $[-1, 1]$ auswerten zu können. Dies kann mit Hilfe der Taylorpolynome p_n geschehen. Dabei ist nach (7) der Fehler maximal $\frac{e}{(n+1)!}$. Für 8 Stellen Genauigkeit benötigt man schon p_{12}. Die Nutzung der Potenzgesetze spart aber Rechenzeit.

$$e^M = (e^{M/256})^{256} = (((((((e^{M/256})^2)^2)^2)^2)^2)^2)^2)^2. \tag{10}$$

- Bestimme $\frac{M}{2^8}$.
- Bestimme $p_4(M)$ mit p_4 aus (8).
- Quadriere $p_4(M)$ $|E| + 8$ mal.
- Wenn E negativ ist, bestimme das Reziproke des letzten Ergebnisses.
- Passe das Ergebnis an die Rechnerarithmetik an.

Dabei sind ein 8-Stellen Shift sowie 4 Additionen und 4 Multiplikationen für die Auswertung von p_4 nötig. Hinzu kommen weitere $|E| + 8$ Multiplikationen und eine potentielle Division. Da selbst in einer 64bit Arithmetik $e^{2^{10}} = Inf$ ist, ist auch die Anzahl an Multiplikationen auf 22 beschränkt.
Für höhere Genauigkeiten kann das benutzte p_n noch weiter erhöht werden.

3.2 Newton-Verfahren

Das Finden von Nullstellen von Funktionen ist nicht immer einfach. Aus diesem Grund gibt es Nährungsverfahren, wie z.B. das Newton-Verfahren.
Das Newton-Verfahren funktioniert am besten, wenn als Startpunkt ein Punkt gewählt wird, der möglichst nah an der gesuchten Nullstelle liegt. Wir versuchen in diesem Verfahren, die Nullstelle x^* durch mehrere Iterationen anzunähern.

a	3
M	$0,75$
E	2
$M/256$	$1,4142136$
$p_4(M/256)$	$1,00293398$
$(p_4(M/256))^2 56$	$2,11700002$
$((p_4(M/256))^2 56)^4$	$20,0855369$
M	$0,627673039$
E	5
e^3	$20,0855369$

Tabelle 1: Schritte bei der Bestimmung von e^3

Voraussetzungen dafür, dass das Newton-Verfahren überhaupt funktioniert, sind die Folgenden:

- eine *stetig differenzierbare* Funktion, die mindestens eine Nullstelle besitzt
- die erste Ableitung sollte nicht 0 werden an der Nullstelle

Es gibt verschiedene Herleitungen für dieses Verfahren. Im Folgenden stellen wir eine von diesen vor. Diese basiert auf der zuvor vorgestellten Taylor-Entwicklung. Sei $f(x)$ eine stetig differenzierbare Funktion. Weiterhin sei $f(x^*) = 0$, al die Funktion f besitzt eine Nullstelle bei x^*. Dann sei $x_0 \approx x^*$. Jetzt betrachten wir das p_1-Polynom der Taylorentwicklung, also das Polynom bis zum Glied mit der 1. Ableitung.

$$f(x) \approx f(x_0) + f'(x_0) \cdot x - x_0 = p_1(x)$$

Jetzt bestimmen wir die Nullstelle x_1 dieses Polynoms.

$$f(x_0) + f'(x_0) \cdot (x_1 - x_0) = 0$$

Nun stellt man die Gleichung einfach nur noch nach x_1 um.

$$\begin{aligned} f'(x_0) \cdot (x_1 - x_0) &= -f(x_0) \\ x_1 - x_0 &= -\frac{f(x_0)}{f(x_0)} \\ x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} \end{aligned}$$

Diese Vorgehensweise kann beliebig oft wiederholt werden um immer bessere Näherungen für x^* zu erhalten und es ergibt sich dann folgende Iterationsvorschrift:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{11}$$

3.3 Konvergenz des Newton-Verfahren

Wir betrachten die Iterationsfunktion des Newton-Verfahrens

$$\varphi(x) = x - \frac{f(x)}{f'(x)}.$$

Es gilt für die Nullstelle x^* dass $\varphi(x^*) = x^*$. und weiterhin

$$\begin{aligned} \varphi'(x) &= 1 - \frac{(f'(x))^2 - f(x) \cdot f''(x)}{(f'(x))^2} \\ &= 1 - \frac{(f'(x))^2}{(f'(x))^2} + \frac{f(x) \cdot f''(x)}{(f'(x))^2} \\ &= \frac{f(x) \cdot f''(x)}{(f'(x))^2}. \end{aligned} \tag{12}$$

Damit gilt

$$\begin{aligned} x^* - x_{n+1} &= x^* - \varphi(x_n) \\ &= \varphi(x^*) - \varphi(x_n) \\ &= \varphi(x^*) - (\varphi(x^*) + \varphi'(x^*)(x_n - x^*) + \frac{\varphi(\xi)}{2}(x_n - x^*)^2. \end{aligned} \tag{13}$$

Dabei ist $\varphi(x_n)$ bis zum Polynom p_1 der Taylorentwicklung um x^* entwicklelt und das Fehlerglied addiert worden. Mit (12) folgt $\varphi'(x^*) = 0$ da $f(x^*) = 0$ und $f'(x^*) \neq 0$. Für (12) folgt damit

$$x_{n+1} - x^* = \frac{\varphi''(\xi)}{2}(x_n - x^*)^2$$

für ein $\xi \in [x^*, x_n]$. Da der Abstand zur Nullstelle im $(n+1)$ten Schritt vom Quadrat des Abstands des $n$ten Schrittes abhängt, spricht man auch von *quadratischer Konvergenz*. Wenn $(x_n - x^*)$ genügend klein ist, dann ist $(x_{n+1} - x^*)$ noch sehr viel kleiner. Anschaulich kann man sagen, dass sich die Anzahl korrekter Stellen der Iterierten x_n in jedem Schritt etwa verdoppelt.

3.4 Division mit Hilfe des Newton-Verfahrens

Die Division mit Hilfe des Newton-Verfahrens beruht auf dem Prinzip, dass wir uns eine Funktionsschar suchen die als Nullstellen die Reziprokenwerte der Zahlen a hat, durch die wir dividieren möchten. Eine spezielle Wahl dieser Funktionen sorgt dafür, dass das Newton-Verfahren ohne Divisionen anwendbar ist.

Zur Bestimmung des Reziproken einer Zahl a kann man wie folgt vorgehen. Zur Vereinfachung gehen wir von $a > 0$ aus. Das Vorzeichen von a und a^{-1} ist identisch. Wir machen die folgenden Vorbetrachtungen

- $a = M \cdot 2^E$,
- $\frac{1}{a} = \frac{1}{M} \cdot 2^{-E}$,
- $f(x) = \frac{1}{x} - M$ hat $\frac{1}{M}$ als Nullstelle,
- $x_{n+1} = x_n - \frac{\frac{1}{x_n} - M}{-\frac{1}{x_n^2}} = x_n(2 - M \cdot x_n)$.

Man muss also nur das Reziproke von Zahlen $M \in [0,5; 1)$ bestimmen können.

$$\begin{aligned} a &= M \cdot 2^E \\ x_{n+1} &= x_n - \frac{\frac{1}{x_n} - M}{-\frac{1}{x_n^2}} = x_n(2 - M \cdot x_n) \end{aligned}$$

Diese Gleichung dient uns zur Reziprokenberechnung. Da das Verfahren schneller konvergiert, wenn man Startwerte wählt, die in der Nähe der gesuchten Nullstelle liegen, benötigen wir ein Verfahren, das uns geeignete Startwerte (x_0) liefert, also solche, die bereits nahe an $\frac{1}{M}$ liegen. Eine einfache Möglichkeit ist eine Aproximation von $1/M$ durch eine lineare Funktion der Form $g(x) = mx + n$.

Die Funktionsgleichung kann man durch folgendes Gleichunggssystem gewinnen:

$$\begin{aligned} f(x) &= \frac{1}{x} \\ f(\frac{1}{2}) - g(\frac{1}{2}) &= \delta \\ f(1) - g(1) &= \delta \\ f(\xi) - g(\xi) &= -\delta \\ (f(\xi) - g(\xi))' &= 0 \end{aligned}$$

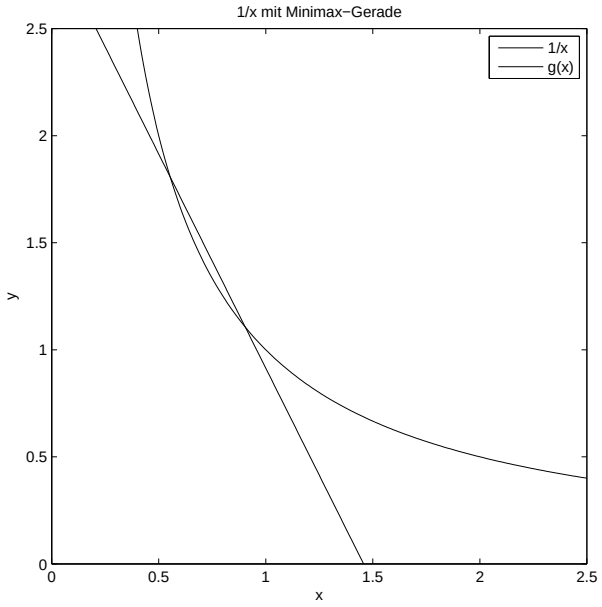


Abbildung 1: Minimax-Gerade

Dabei fordern wir, dass der maximale Abstand zwischen f und g an den Rändern und an einer Zwischenstelle angenommen wird und insgesamt minimal ist. Die Lösung dieses Gleichungsystems ist die sogenannte *Minimax-Gerade*

$$g(x) \quad = \quad -2 \cdot x + \frac{3}{2} + \sqrt{2}.$$

Mit diesem Startwert reichen für 8 Stellen Genauigkeit etwa 3 Newton-Schritte.

- Bestimme $x_0 = g(M) = -2 \cdot M + (\frac{3}{2} + \sqrt{2})$.
- Bestimme $x_3 \approx \frac{1}{M}$.
- Rechenbedarf: 7 Multiplikationen, 4 Additionen

Da $1/M \in (1; 2]$ liegt müssen wir nun noch die Zahl an den Mantissenbereich und den Exponenten anpassen.
Dieses Verfahren zur Division (nach Multiplikation mit dem entsprechenden Zähler) ist tatsächlich schneller als die Vorgehensweise wie beim schriftlichen Dividieren.

$$\frac{1}{a} \quad \approx \quad \frac{x_3}{2} \cdot 2^{-E+1}$$

a	3
M	$0,75$
E	2
x_0	$1,4142136$
x_1	$1,3284271$
x_2	$1,3333153$
x_3	$1,3333333$
M	$0,6666667$
E	-1
$1/a$	$0,3333333$

Tabelle 2: Schritte bei der Bestimmung von 1/3

Anwendungen

Weitere Anwendungsmöglichkeiten sind zum Beispiel das Logarithmieren. Es gilt

- $a = M \cdot 2^E$,
- $ln(a) = ln(M) + ln(2) \cdot E$,
- $e^x - M$ hat $ln(M)$ als Nullstelle.
- $x_{n+1} = x_n - \frac{e^{x_n} - M}{e^{x_n}} = x_n - 1 + M \cdot e^{-x_n}$ (Newton)

Es muss also nur $ln(M)$ für $M \in [\frac{1}{2}, 1)$ durch einige Newtoniterationen ausgewertet werden. Dafür sind nur Additionen und Multiplikationen notwendig. Zusätzlich muss $ln(2)$ als Konstante zur Verfügung stehen. Die Minimax-Gerade für den Startwert ist $g(x) = 1,38629436x - 1,35646431$.

4 Der CORDIC-Algorithmus

Der CORDIC[4] Algorithmus ist ein effektives Verfahren, um den Cosinus und den Sinus und damit den Tangens eines Winkels nährungsweise zu bestimmen. Der Algorithmus basiert auf folgender Iteration:

[4]Die Abkürzung steht für *Co*ordinate *R*otation *Di*gital *C*omputer und wurde von Jack E. Volder Ende der 50er Jahre entwickelt.

a	3
M	$0,75$
E	2
x_0	$-0,316743539$
x_1	$-0,287255667$
x_2	$-0,287681982$
x_3	$-0,287682073$
$ln(2) \cdot 2$	$1,38629436$
$x_3 + ln(2) \cdot 2$	$1,09861229$
M	$0,549306144$
E	1
$ln(3)$	$1,09861229$

Tabelle 3: Schritte bei der Bestimmung von $ln3$

$$\begin{cases} x_{n+1} & = x_n - d_n \cdot y_n \cdot 2^{-n} \\ y_{n+1} & = y_n + d_n \cdot x_n \cdot 2^{-n} \\ z_{n+1} & = z_n - d_n \cdot \arctan 2^{-n} \\ d_n & = sign(z_n) \end{cases} \tag{14}$$

Alle möglichen Werte von $\arctan 2^{-n}$ für die verschiedenen n werden vorgespeichert, wobei üblicherweise n etwas größer als die Mantissenlänge gewählt wird. Wenn $|z_0|$ kleiner oder gleich

$$\sum_{k=0}^{\infty} \arctan 2^{-k} = 1.743286047... \tag{15}$$

ist, dann ergibt sich

$$lim_{n\to\infty} \begin{pmatrix} x_n \\ y_n \\ z_n \end{pmatrix} = K \times \begin{pmatrix} x_0 \cdot \cos z_0 - y_0 \cdot \sin z_0 \\ x_0 \cdot \sin z_0 + y_0 \cdot \cos z_0 \\ 0 \end{pmatrix},$$

wobei der Skalierungsfaktor K gleich $\Pi_{n=0}^{\infty} \sqrt{1 + 2^{-2n}} = 1.64676025...$ ist. Um nun den Cosinus und Sinus eines Winkels zu bestimmen, benötigt man noch die Startwerte für x, y und z. Die Zahl für die Cosinus bzw. Sinus bestimmt werden

sollen, ist θ. Dabei kann θ maximal so groß wie der bei (15) berechnete Wert sein.

$$\begin{aligned} x_0 &= \frac{1}{K} = 0,60725... \\ y_0 &= 0 \\ z_0 &= \theta \end{aligned}$$

Das Grundprinzip ist, dass man versucht durch Drehung eines Einheitsvektors um verschiedene immer kleiner werdende Winkel $\pm \arctan(2^{-n})$ einzukreisen. Ersetzt man die Drehung um $\pm \arctan(2^{-n})$ durch eine Drehstreckung mit dem zusätzlichen Faktor $\sqrt{1+2^{-2n}}$, dann sind diese Drehstreckungen durch den Algorithmus (14) realisiert. Zum Ausgleich der Streckung wird mit dem auf $1/K$ verkürzten Vektor begonnen. Der finale Vektor ist der um θ gedrehte Einheitsvektor, von dessen Komponenten $\cos\theta$ und $\sin\theta$ abgelesen werden können.
Die Vorteile bei diesem Verfahren sind zum einen, dass es sowohl Cosinus als auch Sinus gleichzeitig berechnet und dass man zum Berechnen nur addieren und Mantissen shiften können muss.

5 Oder doch lieber Tabellen?

5.1 Grundlagen

Neben der Möglichkeit die Werte einer Funktion auszurechenen, ist es auch möglich, Tabellen zur Wertebestimmung heranzuziehen. Dabei werden bestimmte Stellen – die sogenannten Stützstellen – und die dazugehörigen Werte gespeichert. Dieses Verfahren wurde bevor es Taschenrechner gab zur Bestimmung von beispielsweise Sinuswerten herangezogen. Damals ergab sich meist eine Genauigkeit von ca. 4 Stellen, weil die tabellarisierten Werte nicht genauer angegeben waren. Um die Werte zwischen zwei Stützstellen zu bestimmen, wurde einfach eine Gerade durch diese beiden Punkte gelegt und deren Werte als Näherungen angenommen. Dies ist jedoch bei einer geforderten Maschinengenauigkeit von mindestens 16 Stellen zu ungenau. Und besser als eine solche Gerade wäre ein Polynom $p(x)$ $(n-1)$-ten Grades, wobei n die Anzahl der Stützstellen ist.

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

Dieses Polynom muss an allen Stützstellen die in der Tabelle abgespeicherten Werte annehmen, $p(x_i) = y_i$. Aber noch ist unklar ob dieses Polynom überhaupt existiert.

Um die Existenz dieses Polynoms zu beweisen, definieren wir zuerst ein sogenanntes *Lagrange-Polynom* $(n-1)$-Grades, für das gilt:

$$L_i(x_j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \tag{16}$$

Durch Multiplikation der Linearfaktoren ergibt sich:

$$\begin{aligned} l_i &= (x - x_1)(x - x_2) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n) \\ l_i(x_i) &= (x_i - x_1)(x_i - x_2) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n) \end{aligned}$$

Dadurch ist in l_i die Bedingung $l_i(x_j) = 0$ für $i \neq j$ erfüllt. Durch Teilung von l_i durch $l_i(x_i)$ ist auch die zweite Bedingung $l_i(x)/l_i(x_i) = 1$ für $i = j$ erfüllt und $L_i(x)$ ist nun definiert mit

$$L_i(x) = \frac{l_i}{l_i(x_i)}.$$

Daraus ergibt sich für p

$$p = a_1 L_1 + a_2 L_2 + \ldots + a_n L_n.$$

Und für die Stützstellen x_i

$$p(x_i) = a_1 L_1(x_i) + a_2 L_2(x_i) + \ldots + a_n L_n(x_i).$$

Dabei ist jedoch wegen Bedingung (16) jeder Summand außer $a_i L_i(x_i)$ gleich Null, denn L_i und a_i beziehen sich nur bei der Stützstelle x_i auf das gleiche i und damit ist der Faktor L_i in jedem anderem Summanten Null. Folglich ist $p(x_n) = a_n = y_n$. Durch Koeffizientenvergleich an allen Stützstellen ergibt sich

$$p(x) = y_1 L_1(x) + y_2 L_2(x) + \ldots + y_n L_n(x).$$

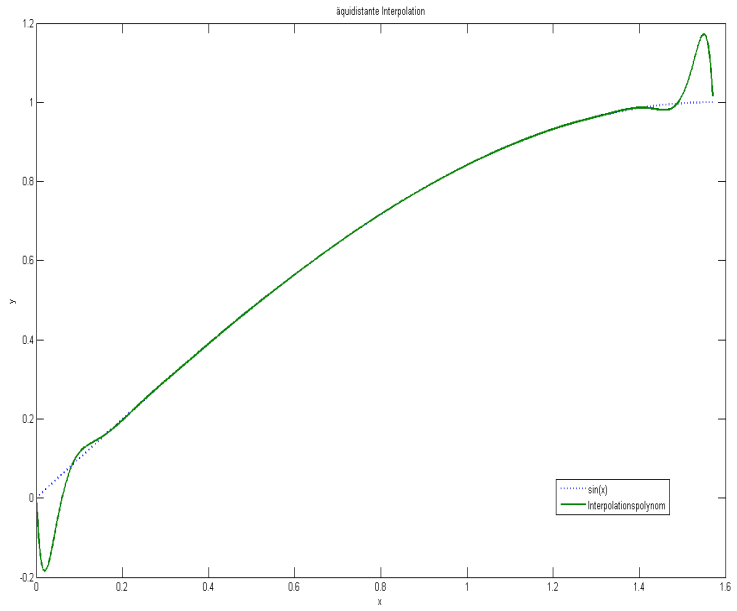
Dieses Polynom existiert und erfüllt alle vorher an $p(x)$ gestellten Anforderungen. Auch wenn die Existenz dieses Polynoms nun bewiesen ist, ist noch nicht klar, dass dieses eindeutig ist, dies wie folgt zeigen.
Nehmen wir einmal an, es gibt neben dem Polynom $p(x)$ das Polynom $q(x)$,

welches auch an allen Stützstellen x_i durch die zugehörigen Funktionswerte y_i läuft. $q(x)$ ist wie $p(x)$ ein Polynom $(n-1)$-ten Grades.
Wären $p(x)$ und $q(x)$ unterschiedlich, dann ließe sich die Differenz r durch $r = p - q$ beschreiben. Da dies an allen Stellen von p und q gilt, folgt daraus auch

$$r(x_i) \quad = \quad p(x_i) - q(x_i). \tag{17}$$

$r(x)$ ist ein Polynom $(n-1)$-ten Grades, da es die Differenz von zwei Polynomen $(n-1)$-ten Grades ist. Aus (17) geht hervor, dass $r(x_i)$ immer gleich Null ist, da p und q an den Stellen x_i gleich y_i sein müssen. x_i steht für die Stützstellen, deren Anzahl n ist (siehe oben). Damit hat r n Nullstellen. Es gibt nur ein Polynom, dessen Grad kleiner ist als die Anzahl seiner Nullstellen und das ist die Nullfunktion. Deshalb gilt $r(x) = 0$ woraus folgt $p(x) = q(x)$ und bewiesen wäre, dass nur ein Polynom $p(x)$ vom Grad $(n-1)$ existiert, welches durch alle Stützstellen verläuft.

Durch Interpolation mit dem Polynom lassen sich die Werte zwischen den tabellarisierten Stützstellen approximieren. Um eine sehr hohe Genauigkeit zu erhalten, müsste nun eigentlich nur die Anzahl an Stützstellen entsprechend hoch gewählt werden. Wenn alle Stützpunkte gleich verteilt werden ergibt sich jedoch eine sehr große Abweichung an den Rändern des in der Tabelle angegebenen Intervalls.
Dieses Bild verdeutlicht die Abweichung bei einer Annäherung an die Sinuskurve (gestrichelt) mit einem Polynom bei 20 gleich verteilten Stützstellen:



5.2 Tschebyschev-Interpolation

Um diese Abweichung zu beheben ist es sinnvoll die Verteilung der Stützstellen zu verändern, indem man mehr Stützstellen an den Rändern wählt. Um diese Verteilung zu optimieren wurde das sogenannte *Tschebyschev-Verfahren* entwickelt.

Bei diesem Verfahren wird nach einem Polynom n-ten Grades gesucht, das in dem Intervall $[-1;1]$ den maximalen Betrag 1 hat sowie alle Maxima und Minima bei ± 1. Dabei steht n für die Anzahl der späteren Stützstellen. Das Polynom, welches diese Bedingungen erfüllt heißt *Tschebyschev-Polynom* T_n:

$$\begin{aligned} n = 0 : T_0 &= 1 \\ n = 1 : T_1 &= x \\ n = 2 : T_2 &= 2x^2 - 1 \\ &\vdots \\ n = k : T_k &= cos(k \cdot arccos(x)) \end{aligned}$$

Um die Nullstellen zu bestimmen wird T_n gleich Null gesetzt, da der Cosinus seine Nullstellen bei $\frac{\pi}{2} + (\pi)i$ für alle $i \in \mathbb{N}$ hat, ergibt sich:

$$\begin{aligned} k(arccos(x)) &= \frac{\pi}{2} + (\pi)i \\ arccos(x) &= \frac{\pi}{k}(\frac{1}{2} + i) \\ x &= cos(\frac{\pi}{k}(\frac{1}{2} + i)) \end{aligned}$$

Der Cosinus nimmt auf dem Intervall $[0;\pi]$ die Werte aus dem Intervall $[-1;1]$ an. Daraus ergibt sich die Ungleichung $0 \leq \frac{\pi}{k}(\frac{1}{2}+i) \leq \pi$, das heißt i nimmt alle Werte zwischen $i = 0$ und $i = k-1$ an. Dies bedeutet auch, dass es ingesamt k Nullstellen gibt welche als Stützstellen dienen. Die Polynominterpolation einer beliebigen Funktion an diesen Stützstellen weist eine bedeutend höhere Genauigkeit auf als mit gleichverteilten Stützstellen. Eine Vergleichsrechnung zu den gleichverteilten Stützstellen zeigte nur noch Abweichungen in der Größenordnung der Maschinengenauigkeit und die Graphen der Funktion und der Interpolierenden waren deckungsgleich.

5.3 Das Horner-Schema

Wie bereits gesehen, basieren viele Näherungsverfahren auf Polynomauswertungen (Taylorpolynome, Interpolationspolynome). Dabei kommt der Effizienz und Rechengeschwindigkeit eine immer größere Bedeutung zu. Dies hat zur Folge, dass versucht wird, die Algorithmen zu optimieren.

Das Horner-Schema ist eine spezielle Anordnung der Faktoren eines Polynoms

$$p(x) = a_0 + xa_1 + x^2a_2 + x^3a_3... + x^na_n$$

Bei Betrachtung ergibt sich, dass hier n Additionen benötigt werden. Hinzu kommen noch zur Auswertung der x^k-Potenzen jeweils $(k-1)$ Multiplikationen. Durch den steigenden Exponenten addiert mit den Multiplikationen der x-Potenzen und der dazugehörigen Koeffizienten a_k ergeben sich insgesamt $\frac{n(n+1)}{2}$ Multiplikationen.

Wird das x ausgeklammert ergibt sich folgedes Polynom:

$$p(x) = a_0 + x(a_1 + xa_2 + x^2a_3... + x^{n-1}a_n)$$

Nun wird x im entstandenen Faktor wieder ausgeklammert so dass sich folgendes Polynom ergibt:

$$p(x) = a_0 + x(a_1 + x(a_2 + xa_3... + x^{n-2}a_n))$$

Dieses Prinzip lässt sich n-mal wiederholen, sodass sich folgendes Polynom ergibt:

$$q(x) = a_0 + x(a_1 + x(a_2 + x(a_3... + xa_n))...))$$

Dieses Polynom $q(x)$, das durch äquivalente Umformung aus dem ursprünglichen Polynom $p(x)$ hervorgegangen ist wird jetzt noch auf Effizienz untersucht. Genaues Zählen liefert, dass n Additionen und nur noch n Multiplikationen nötig sind.

6 Fazit

Wozu nun selber rechnen? Mit unserem Projekt sollte verdeutlicht werden, wie ein Computer rechnet und vor allem sollte gezeigt werden, wo die Grenzen unserer Rechner liegen. Wenn man diese Grenzen kennt, weiß man auch, ab wann man selbst rechnen muss. Auch wenn die Maschinengenauigkeit bei einem 64bit-Zahlensystem sehr klein erscheint, kann in längeren Rechnungen mit vielen Iterationen so oft aufsummiert werden, dass das Ergebnis entweder verfälscht oder gänzlich ungenau wird. Wenn man jedoch weiß, dass das Ergebnis eines Rechners

nicht immer exakt sein kann, lohnt es sich längere Rechnungen schon anfangs so einfach wie möglich zu halten, damit die vermutete Ungenauigkeit so gering wie möglich wird und besser berücksichtigt werden kann.