Warum Computer doch nicht so präzise rechen

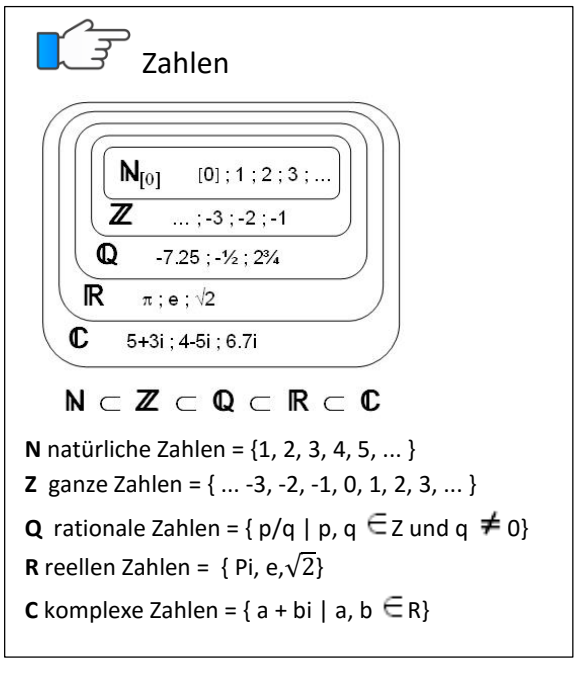
Thomas Staub

Gleitkommazahlen:

Dieses Dokument ist ein Zusammenschnitt mehrerer Beiträge und Berichte aus dem Internet zum Thema Gleitkommazahlen. Die Quellen sind am Ende des Dokuments erwähnt.

Inhalt

Rechnen mit dem Computer



Der heutige Beitrag bleibt in der Rubrik "Wie funktionieren Computer" und soll ein wenig mit dem (Vor-) Urteil aufräumen, dass Computer besser als Menschen rechnen. Dass sie schneller sind, ist unbestreitbar – kein Rechenkünstler der Welt wird mehrere Millionen Berechnungen in der Sekunde schaffen. Aber wie sieht es mit der Genauigkeit aus, insbesondere im Bereich der reellen Zahlen?

Eins vorweg: Computer können in der Theorie natürlich so beliebig genau rechnen, wie sie wollen (und wie es ihr Speicher zulässt – die Zahl Pi wird vermutlich in keinen Computer der Welt passen). Allerdings erfordert das zusätzlichen Aufwand, der mit steigender Genauigkeit immer größer wird und damit natürlich immer mehr Zeit verbraucht. In modernen Computern kommt daher ein standardisiertes Verfahren zur Berechnung zur Anwendung, welches auf einer festen ***Bitkettenlänge*** beruht. Stehen für eine Zahl etwa 32 Bit zur Verfügung, so können damit insgesamt 2^{32}, also 4294967296 (rund 4 Milliarden) Zahlen dargestellt werden. Bei 64 Bit sind es dann schon gewaltige 18446744073709551616 (mehr als 18 Trillionen) Zahlen. Hält man sich hier nur im Bereich der natürlichen oder ganzen Zahlen auf, gibt es kein Problem, da jede der Zahlen im Wertebereich eindeutig dargestellt werden kann. Nach der 1 kommt nun einmal die 2, dazwischen ist nichts. Auch das Rechnen ist relativ einfach.

Bisher habe ich allerdings verschwiegen, wie am Computer all das dargestellt wird, was ausserhalb des Wertebereichs der ganzen Zahlen liegt; konkret: die reellen Zahlen. Wer sich nicht mehr ganz erinnert: reelle Zahlen umfassen praktisch alles, was wir so im Alltagsgebrauch an Zahlen benutzen, angefangen von den natürlichen und ganzen Zahlen über die rationalen Zahlen (all die Zahlen, die als Verhältnis zweier ganzer Zahlen darstellbar sind) bis hin zu den irrationalen Zahlen (wozu auch Pi und e zählen). Da die reellen Zahlen in der modernen Mathematik unabdingbar sind und ein Großteil der Probleme, die mit Computern gelöst werden, auf diesem Zahlenbereich basieren, müssen sie natürlich auch effektiv im Rechner dargestellt werden können. Da wir bei der Darstellung auf Bitketten angewiesen sind, die nur aus 0en und 1en bestehen können, fällt die in der Mathematik übliche Darstellung der Zahlen mit einem Trennzeichen weg, es ist also ein anderes Verfahren notwendig. Bevor es jetzt gleich in die Details geht, noch eine Anmerkung: im deutschsprachigen Raum ist das Komma als (Dezimal-)Trennzeichen in reellen Zahlen üblich – im englischsprachigen Raum dagegen der Punkt. Da in der Informatik üblicherweise ein Punkt als Trennzeichen benutzt wird, halte ich mich auch hier daran; die Zahl “1.5” ist also für deutsche Augen zu lesen als “Eins Komma Fünf” – dennoch werde ich von einem “Komma” *sprechen* und nur bei der Darstellung den Punkt verwenden.

Festkommazahlen

Das einfachste Verfahren zur Darstellung von reellen Zahlen durch Bitketten ist, ein Komma an einer bestimmten Stelle der Bitkette anzunehmen, ohne es aber hinschreiben zu müssen. Da das (gedachte) Komma hier an einer festen Stelle ist, spricht man auch von Festkommazahlen; die Bits vor der gedachten Kommastelle codieren hierbei entsprechend den Vorkommaanteil der Zahl, die Bits hinter der gedachten Kommastelle entsprechend den Nachkommaanteil. Die Umrechnung einer Bitkette in eine Festkommazahl ist relativ einfach: dazu wird die Bitkette zuerst als ganze Zahl interpretiert und dann durch den Wert 2^f geteilt, wobei f die Anzahl der Bits hinter der gedachten Kommaposition bezeichnet.

Ein Beispiel: angenommen, wir haben Bitketten der Länge 8, deren gedachte Kommaposition genau in der Mitte liegt; f wäre somit 4, 2^4 also 16. Haben wir nun die Bitkette *10101010* (mit Komma: *1010.1010*) so stellt die ganze Bittkette in der Dezimaldarstellung die Zahl 170 dar. Geteilt durch 16 ergibt sich für die Bitkette *10101010* also die Zahl 10.625

Wertigkeit	8	4	2	1		1/2	1/4	1/8	1/16
Bittkette	1	0	1	0	,	1	0	1	0
	8	+	2			0.5	+	0.125	
Dezimalwert	10					625			

Noch ein Beispiel: angenommen, wir haben Bitketten der Länge 4, deren gedachte Kommaposition genau in der Mitte liegt; f wäre somit 2, 2^2 also 4. Haben wir nun die Bitkette *1010* (mit Komma: *10.10*) so stellt sie die in der Dezimaldarstellung die Zahl 10

dar (nämlich 1*8 + 0*4 + 1*2 + 0*1); geteilt durch 4 ergibt sich für die Bitkette *1010* also die Zahl 2.5

Der große Vorteil von Festkommazahlen ist, dass mit ihnen relativ einfach gerechnet werden kann. Der Nachteil dieser Darstellung ist allerdings offensichtlich: durch die feste Kommaposition ist der darstellbare Wertebereich sehr eingeschränkt. Aber Achtung: hier ist natürlich nicht die *Anzahl* der darstellbaren Zahlen gemeint – die beträgt weiterhin 2^{32}; vielmehr ist die Ausdehnung des Wertebereiches beschränkt, da für den ganzzahligen Anteil weniger Bits zur Verfügung stehen; die *größte* darstellbare Zahl ist also beschränkt.

Gleitkommazahlen Theorie

Um dem Problem mit dem Beschränkten Wertebereich entgegen zu treten, wurde die Darstellung in Form sogenannter Gleitkommazahlen entwickelt. Wie es der Name schon andeutet, ist hier die Position des Kommas nicht festgelegt, sondern wird in der Zahl selber codiert; durch dieses Vorgehen kann ein weitaus größerer Zahlenbereich abgedeckt werden (was zu anderen Problem führt, aber dazu gleich mehr).

Mathematisch gesehen entspricht die Gleitkommadarstellung einer normalisierten Exponentialschreibweise. Wer sich nicht mehr erinnert: eine Zahl in Exponentialschreibweise lässt sich als $m * b^e$ darstellen, wobei *m* die sogenannte ***Mantisse*** (die eigentliche Zahl), *b* die **Basis** (im Dezimalsystem die 10) und *e* der **Exponent** ist. Üblicherweise wird die Schreibweise zur Darstellung sehr großer oder sehr kleiner Zahlen verwendet, etwa Die Lichtgeschwindigkeit im Vakuum beträgt c = 299‘792‘458 m/s => 299‘792,458 · 10^3 m/s => 0,299792458 · 10^9 m/s => 2,99792458 · 10^8 m/s. Nur die Mantisse der letzten Darstellung ist auf den Wertebereich [1, 10) normalisiert. Von einer normalisierten Exponentialschreibweise spricht man, wenn sich *vor* dem Komma in der Mantisse lediglich eine einzelne Ziffer *ungleich 0* befindet.

Im Computer lohnt sich hier nun natürlich, als Basis der Darstellung die 2 zu wählen. Durch die Forderung, dass die Zahl immer normalisiert sein soll, ergibt sich *immer* die folgende Darstellung (wobei *p* den Nachkommateil der Zahl beschreibt): $1.p * 2^e$. Die 1 vor dem Komma ergibt sich automatisch durch die erzwungene Normalisierung: vor dem Komma darf nur eine Ziffer ungleich 0 stehen; da im binären System nur 1 und 0 zur Verfügung stehen, ist die Wahl hier eindeutig.

1010.1010 => $1.0101010 * 2^3$

Jetzt muss diese Zahl nur noch in einer Bitkette codiert werden. Da liegt es nun natürlich nahe, in einem Teil der Bitkette den Exponenten (3) und im zweiten Teil der

Bitkette die Mantisse (0101010) zu hinterlegen. Der IEEE-Standard 754 für Gleitkommazahlen *einfacher Genauigkeit* (das heißt unter Ausnutzung von 32 Bits) sieht etwa vor, dass für den Exponenten 8 Bit und für die Mantisse 23 der 32 Bit zur Verfügung stehen. Die führende 1 der Mantisse muss übrigens nicht mitgespeichert werden, da sie ja definitiv immer vorhanden ist; es reicht also, sich auf den Nachkommateil zu beschränken. Das eine fehlende Bit (23+8=31) ist übrigens für das Vorzeichen reserviert, damit auch negative reelle Zahlen dargestellt werden können, wobei in der Regel eine 0 für positive Zahlen und eine 1 für negative Zahlen verwendet wird. Die Bits werden in der Reihenfolge Vorzeichen (s) – Exponent (e) – Mantisse (m) abgespeichert, so dass sich für eine 32-Bit-Zahl das folgende Muster ergibt:

s eeeeeeee mmmmmmmmmmmmmmmmmmmmmmm

Bleibt nur noch ein Problem: der Exponent kann positiv sein wie bei der Zahl $2{,}99792458 \cdot 10^8$ oder aber negativ wie bei der Zahl $2.5 * 10^{-3}$ => 0.0025 (beide Zahlen der Einfachheit halber im Dezimalsystem). Bei der Speicherung des Exponenten muss also noch ein Vorzeichen untergebracht werden. Eine Möglichkeit wäre natürlich die bereits bekannte Darstellung im Zweierkomplement, welche allerdings aus gleich noch näher erläuterten Gründen nicht benutzt wird.

Zweierkomplement

Das Zweierkomplement ist eine Möglichkeit, negative Zahlen im Dualsystem darzustellen. Dabei werden keine zusätzlichen Symbole wie + und − benötigt.

Positive Zahlen werden in der Zweierkomplementdarstellung mit einer führenden 0 (Vorzeichenbit) versehen und ansonsten nicht verändert.

Negative Zahlen werden wie folgt aus einer positiven Zahl kodiert: Sämtliche binären Stellen werden negiert und zu dem Ergebnis der Wert 1 addiert.

Stattdessen wird der Exponent mit einem sogenannten ***Bias*** gespeichert. Der Bias ist ein vereinbarter Wert, welcher bei der Konvertierung einer Zahl in die Bitkette auf den Exponenten aufaddiert wird. Gemäss IEEE ist dies 127 bei Single und 1023 bei Double.
So wird für einen Exponenten *e = 3* also der Wert 130 (127+3), für einen Exponenten *e = -3* (127-3) der Wert 124 abgespeichert.

Das Prinzip der Biased-Darstellung für negative Zahlen:

Binär	Exponent	Wert	Berechnung Bias+Exp
00000000	-127	0	127-127
00000001	-126	1	127-126
00000010	-125	2	127-125
:	:		
01111101	-2	125	127-2
01111110	-1	126	127-1
01111111	0	127	127-0
10000000	1	128	127+1
10000001	2	129	127+2
10000010	3	130	127+3
:	:		
11111101	126	253	127+126
11111110	127	254	127+127
11111111	128	255	127+128

Die Verwendung des Bias hat gegenüber dem Zweierkomplement einen entscheidenden Vorteil, da sie den Vergleich zweier Zahlen erheblich vereinfacht – in dieser Art der Darstellung reicht es, die Bitketten zweier zu vergleichender Zahlen einfach *lexikografisch*, das heißt Bit für Bit von links nach rechts (unter Ausschluss des Vorzeichenbits), miteinander zu vergleichen.

Der Nachteil der Biased-Darstellung gegenüber der Zweierkomplement-Darstellung besteht darin, dass nach einer Addition zweier Biased-Exponenten der Bias subtrahiert werden muss, um das richtige Ergebnis zu erhalten.

Gleitkommazahlen Beispiel

Damit können wir nun endlich ein kleines Beispiel anschauen. Nehmen wir dazu einmal als darzustellenden Wert die 23.625. Das Vorzeichenbit ist schnell gelöst, das ist nämlich 0 (da wir es mit einer positiven Zahl zu tun haben). Aber wie bringen wir den Rest in eine normalisierte Darstellung? Dazu wandelt man die Zahl zuerst in eine Festkommazahl um, und zwar Vor- und Nachkommateil getrennt voneinander.

Die 23 lässt sich ziemlich einfach durch wiederholtes Dividieren durch 2 mit Rest umwandeln; folgende Tabelle zeigt die Rechnung, wobei der errechnete Rest den Vorkommateil in Binärdarstellung (in umgekehrter Reihenfolge) angibt:

23/2= 11, Rest: 1
11/2= 5, Rest: 1
5/2 = 2, Rest: 1
2/2 = 1, Rest: 0
1/2 = 0, Rest: 1

Somit ist: $23_d=10111_b$

Für den Nachkommateil muss man sich ein klein wenig mehr Mühe geben, aber das ist auch nicht sehr schwierig. Statt durch 2 zu teilen, multiplizieren wir einfach mit 2 und schauen, ob das Ergebnis größer als 1 ist; wenn ja, steht an der entsprechenden Stelle (diesmal in der "normalen" Reihenfolge von links nach rechts) eine 1 und man fährt mit der um 1 reduzierten Zahl fort, ansonsten steht eine 0 und es geht direkt weiter. Die Rechnung selber ist aber weitaus anschaulicher als eine Erklärung, daher hier die Tabelle:

0.625*2= 1.25, Bit: 1
0.25*2 = 0.5, Bit: 0
0.5*2 = 1.0, Bit: 1
0.0*2 = 0.0, Bit: 0
…

Somit ist: $0.625_d=1010_b$

Insgesamt ergibt sich also die Festkommazahl 10111.10100000… als Repräsentation der 23.625; zur Kontrolle:

$10111_b = 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 16 + 4 + 2 + 1 = 23$

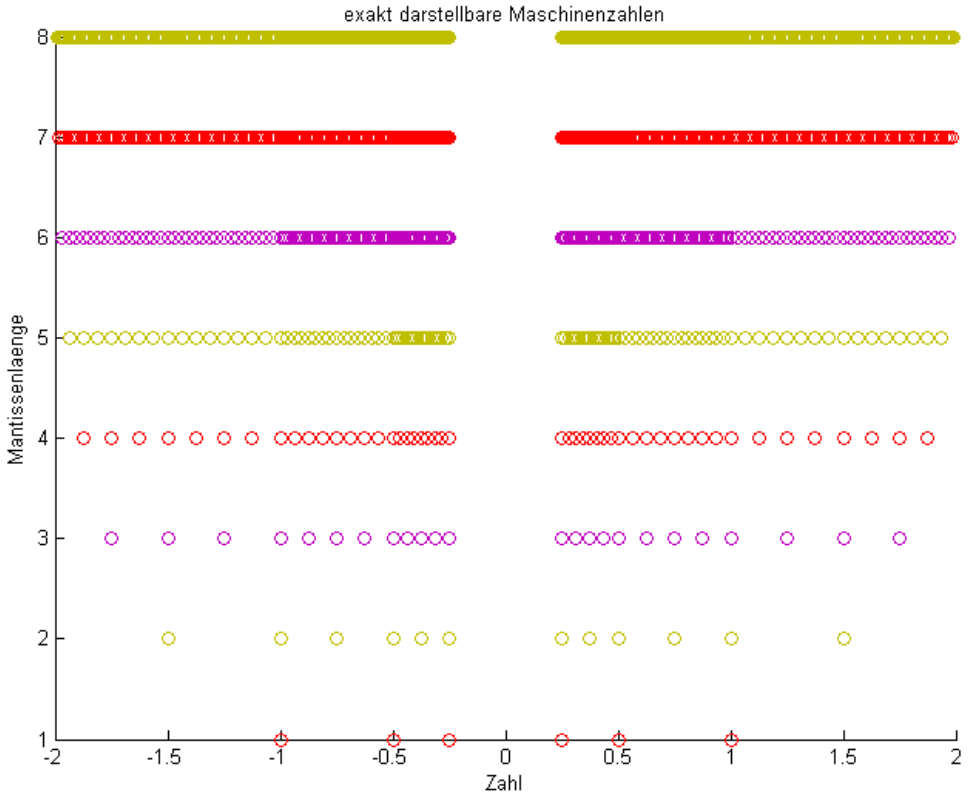
und

$101_{.b} = 1*2^{-1} + 0*2^{-2} + 1*2^{-3} = 0.5 + 0.125 = 0.625.$

Zur Normalisierung muss das Komma in dieser Zahl nun um 4 Stellen nach links verschoben werden; das entspricht einer Multiplikation mit 2^4, ganz so wie im Dezimalsystem. Für die normalisierte Darstellung erhalten wir also: $1.011110100000 * 2^4$. Die Mantisse ist also 10111101_b und der Exponent $4_d = 100_b$. Nun muss noch der Bias auf den Exponenten addiert werden; bei 32-Bit-Zahlen ist das in der Regel 127, also erhalten wir für den Exponenten $131_d = 10000011_b$. Setzen wir alle gesammelten Zahlen zusammen (wir erinnern uns: die führende 1 der Mantisse kann ignoriert werden), ergibt sich demzufolge:
$23.625_d = 0\ 10000011\ 01111010000000000000000_b$
So schwer ist es also gar nicht.

Vor- und Nachteile von Gleitkommazahlen

Gleitkommazahlen haben gegenüber Festkommazahlen den großen Vorteil, dass auch sehr große und sehr kleine Zahlen dargestellt werden können, indem der Exponent sehr groß beziehungsweise sehr klein wird. Ganz unproblematisch ist das allerdings nicht: je größer der Exponent ist, desto weniger Bits stehen zur Verfügung, um die niederwertigen Ziffern der Zahl darzustellen. Hat der Exponent zum Beispiel den Wert 23, so muss das (gedachte) Komma der gespeicherten Mantisse um 23 Stellen nach rechts geschoben werden – die dadurch entstehende Festkommazahl hat dementsprechend gar keine Nachkommastellen mehr. Je größer der Exponent wird, desto mehr niederwertige Stellen der Zahl gehen demzufolge verloren. Und genau hier liegt das eigentliche Problem der Gleitkommazahlen; mit steigender Größe der Zahlen (im positiven wie im negativen Bereich) werden auch die *Lücken* zwischen den darstellbaren Zahlen immer größer. Die Wikipedia bietet hierfür eine schöne Abbildung; gezeigt sind die darstellbaren Zahlen für verschiedene Mantissenlängen. Jeder Kreis markiert eine Zahl, die als Gleitkommazahl nach dem vorgestellten Schema dargestellt werden kann. Es ist leicht zu sehen, dass insbesondere bei kurzer Mantissenlänge die Abstände zwischen den Zahlen immer größer werden (bei größerer Mantissenlänge verschiebt sich das Problem einfach in höhere Wertebereiche).



Man kann sich leicht vorstellen, dass das zu Problemen verschiedener Art führen kann. Insbesondere beim Rechnen mit Gleitkommazahlen ergeben sich daraus bestimmte Dinge, die unbedingt beachtet werden müssen.

Rechnen mit Gleitkommazahlen

Wie wir gesehen haben lassen sich nicht sämtliche Zahlen in einem bestimmten Bereich darstellen (was bei den reellen Zahlen ohnehin mit endlichem Speicher nicht möglich wäre); schlimmer noch ist aber, dass die *Abstände* zwischen den darstellbaren Zahlen immer größer werden, je größer die Zahlen selber werden.
Tückisch wird das vor allem dann, wenn es ans Rechnen mit Gleitkommazahlen geht.

Bevor die Probleme beim Rechnen dargelegt werden können, muss natürlich erst einmal geklärt werden, wie mit Gleitkommazahlen überhaupt gerechnet wird. Glücklicherweise ist das nicht allzu umständlich und lässt sich relativ schnell erklären. Insbesondere Multiplikation und Division lassen sich (in der Theorie) sehr einfach umsetzen; bei der Multiplikation werden einfach die beiden Mantissen (welche ja Festkommazahlen darstellen) miteinander multipliziert und die beiden Exponenten addiert. Wollen wir zum Beispiel die beiden Dezimal-Zahlen $2.5*10^5$ und $6.1*10^3$ miteinander multiplizieren, so rechnen wir:

$2.5 * 6.1 = 15.25$

und

$5 + 3 = 8$

Damit erhalten wir als Ergebnis $15.25*10^8$, was in der Tat der gewünschten Zahl entspricht. Bei der Division kann man genauso vorgehen, nur dividiert man hier die Mantissen und subtrahiert die Exponenten. Im dualen Zahlensystem funktioniert die Rechnung genauso, kann also auch auf den Bitketten der bereits vorgestellten Gleitkommadarstellung im Computer durchgeführt werden – ich führe die weiteren Rechnung aber alle im Dezimalsystem aus, damit sie besser verständlich sind.
Für Addition und Subtraktion muss nur ein zusätzlicher Schritt durchgeführt werden: vor der eigentlichen Rechnung müssen die Exponenten angeglichen werden. Sollen also etwa die beiden oben genannten Zahlen $2.5*10^5$ und $6.1*10^3$ addiert werden, so muss die erste Zahl entweder als $250.0*10^3$ oder die zweite als $0.061*10^5$ umgeschrieben werden – üblicherweise transformiert man die kleinere Zahl hin zum größeren Exponenten. Anschließend reicht es, die beiden Mantissen zu addieren. Für obiges Beispiel ergibt sich damit die folgende Rechnung:

$$2.5*10^5 + 6.1*10^3 = 2.5*10^5 + 0.061*10^5 = 2.561*10^5$$

So weit, so einfach. Die Anpassung des Exponenten kann übrigens ziemlich einfach durchgeführt werden, indem das Komma der Mantisse um die benötigte Anzahl an Stellen nach links oder rechts geschoben wird (das geht im dualen wie im dezimalen System gleichermaßen).
Die Problematik entsteht dadurch, dass im Computer nur eine bestimmte Menge an Bits für eine Gleitkommazahl zur Verfügung stehen; insbesondere die beschränkte Mantissenlänge (für Gleitkommazahlen einfacher Genauigkeit sind das 23 Bit) erweist sich hier als der größte Fallstrick mit verschiedenen Auswirkungen.

Auslöschung

Eine dieser Auswirkungen ist die sogenannte *Auslöschung* (im Englischen *cancellation* genannt). Sie tritt auf, wenn bei der Verrechnung ähnlicher Zahlen niederwertige Informationen verloren gehen und sich dabei zwar nicht der *absolute Fehler*, dafür aber der *relative Fehler* erhöht. Als **absoluten Fehler** bezeichnet man die tatsächliche Abweichung einer (berechneten) Zahl vom eigentlich gewünschten Wert. Soll etwa das Ergebnis einer Rechnung den Wert 0.51 ergeben, der Computer errechnet aber stattdessen 0.50, so haben wir einen absoluten Fehler von 0.01. Berechnen wir 100.50, wollten aber 100.51, so ist der absolute Fehler ebenfalls 0.01. Der **relative Fehler** bezeichnet dementsprechend die relative Abweichung der beiden Werte (des berechneten und des eigentlich korrekten) voneinander. Beim Wertepaar 100.51/100.50 beträgt der relative Fehler (die relative Abweichung) knapp 0.01%, wohingegen es beim Wertepaar 0.51/0.50 bereits 2% Abweichung sind. Der Effekt tritt insbesondere bei der Subtraktion auf.

Ein Beispiel:

Angenommen, wir haben die Zahlen $1.2345*10^5$ und $1.2340*10^5$ und wollen sie voneinander subtrahieren. In der "gewöhnlichen" Mathematik ist das kein Problem und wir erhalten als Ergebnis:

$$\begin{array}{l} \quad 1.2345*10^5 \\ - \; 1.2340*10^5 \\ \hline = \; 0.0005*10^5 \end{array}$$

Rechnen wir nun aber am Computer, so ist ja die maximale Anzahl an speicherbaren Ziffern in einer Zahl beschränkt. Der Einfachheit halber, nehmen wir in unserem Beispiel nur 4 Bit anstelle von den üblichen 23 Bit die zum Speichern zur Verfügung stehen, so würden statt der originalen Zahlen gerundete verwendet werden. Wir hätten also die folgende Rechnung:

$$\begin{array}{l} \quad 1.235*10^5 \\ - \; 1.234*10^5 \\ \hline = \; 0.001*10^5 \end{array}$$

Man sieht, dass der absolute Fehler bei der ersten (gerundeten) Ausgangszahl und dem (demzufolge inkorrekt berechneten) Ergebnis jeweils gleich ist, nämlich $0.0005*10^5$. Der relative Fehler jedoch ändert sich dramatisch, von lächerlichen 0.04% (1.2345 zu 1.235) auf gewaltige 50% (0.0005 zu 0.001)! Man kann sich einfach vorstellen, dass derartige Abweichungen schnell zu vollkommen falschen Ergebnissen führen können, wenn nämlich mehrere Berechnungen nacheinander ausgeführt werden, wobei die (Zwischen-)Ergebnisse vorhergehender Rechnungen als Werte in die nächste Rechnung mit eingehen.

Absorption

Doch damit nicht genug: auch bei der Verrechnung von Zahlen sehr unterschiedlicher Größe kann es zu einem Fehler kommen, der sogenannten *Absorption*. Sie hat die gleiche Ursache wie die Auslöschung, nämlich die beschränkte Anzahl an Mantissenstellen. Zur Absorption kann es kommen, wenn der Exponent einer sehr kleinen Zahl an den einer sehr großen Zahl angepasst wird und dabei viele oder sämtliche signifikanten Mantissenziffern verloren gehen.

Ein Beispiel:
Nehmen wir die Zahlen $1.0*10^5$ und $1.5*10^2$, welche wir addieren wollen (wobei wir weiterhin 4 Ziffern (4 Bit) für die Mantisse speichern können). Für die Addition muss der Exponent der zweiten Zahl angepasst werden; man würde sie also folgendermaßen umschreiben:
$1.5*10^2 = 0.0015*10^5$

Da wir aber nur 4 Ziffern speichern können, ergibt sich stattdessen die Zahl $0.001*10^5$. Führen wir nun die Addition aus, ergibt sich statt der korrekten Rechnung

$$\begin{array}{l} \quad 1.0000*10^5 \\ +\ 0.0015*10^5 \\ \hline = 1.0015*10^5 \end{array}$$

die nicht korrekte Rechnung

$$\begin{array}{l} \quad 1.000*10^5 \\ +\ 0.001*10^5 \\ \hline = 1.001*10^5 \end{array}$$

Durch die Rundung haben wir also ein inkorrektes Ergebnis erhalten. Dies kann sogar so weit führen, dass bei hinreichend großem Unterschied in den Exponenten die zweite Zahl *überhaupt keinen* Beitrag mehr zur Rechnung hat, wenn nämlich sämtliche Signifikanten Ziffern bei der Anpassung des Exponenten verloren gehen.

Glücklicherweise tritt dieses Problem nur bei der Addition und Subtraktion auf, da bei Multiplikation und Division keine Anpassung des Exponenten und damit kein Verlust von Mantissenziffern stattfindet.

Zusätzlich können die Auswirkungen dieses Effekts durch geschicktes Rechnen vermindert werden. Angenommen, zu einer sehr großen Zahl sollen viele sehr kleine Zahlen addiert werden. Führt man die Rechnung in der üblichen Art und Weise durch (indem man die ganzen kleinen Zahlen auf die große addiert), erhält man am Ende unter Umständen ein falsches Ergebnis, da sämtliche Ziffern der kleinen Zahlen bei der Addition verloren gegangen sind. Addiert man dagegen *zuerst* die kleinen Zahlen miteinander und verrechnet dieses Ergebnis erst mit der großen Zahl, kann man das Glück haben, dass die Addition der kleinen Zahlen ein genügend großes Ergebnis ergibt, dass es bei der Addition mit der großen Zahl noch einen Einfluss auf die Rechnung hat.

Die beiden vorgestellten Effekte ziehen noch einen weiteren nach sich, nämlich, dass das aus der Mathematik bekannte Assoziativ- und das Distributivgesetz im Bereich der Gleitkommazahlen am Computer nicht mehr gelten. Wir erinnern uns: das Assoziativgesetz sagt aus, dass `(a+b)+c = a+(b+c)` gelten soll (auf deutsch, dass es egal ist, welche Bestandteile einer Summe zuerst miteinander addiert werden). Gerade eben haben wir aber gesehen, dass gerade das aber bei Gleitkommazahlen nicht gilt, da es durchaus relevant ist, ob man zuerst Zahlen gleicher Größenordnung oder zuerst Zahlen unterschiedlicher Größenordnung miteinander verrechnet. Das Distributivgesetz besagt, dass `a*(b+c) = (a*b)+(a*c)` gelten soll, was auf Grund der eben genannten Problematik bei Gleitkommazahlen leider auch nicht uneingeschränkt gilt.

All die genannten Probleme zeigen, dass das Rechnen mit Gleitkommazahlen am Computer nicht ganz einfach ist. Insbesondere in Bereichen mit vielen Berechnungen müssen sich die Programmierer einer Anwendung daher genaue Gedanken darüber machen, *wie* und in welcher Reihenfolge sie ihre Berechnungen durchführen. Allein die

Änderung der Berechnungsreihenfolge kann da schon zu erheblichen Verbesserungen führen; manchmal sind aber auch neue oder abgewandelte Formeln notwendig, um die gröbsten Probleme bei der Berechnung zu umgehen. In Umgebungen, wo allerhöchste Präzision gefragt ist, ist das natürlich keine Lösung; hier würde der Programmierer dann auf andere Methoden der Zahlendarstellung zurückgreifen (die so viele Bits für eine Zahl zur Verfügung stellen, wie benötigt werden – die aber auch nicht mehr einfach so von der CPU verarbeitet werden können).

Quellen:

http://www.meinstein.ch/schule/mathe/zahlenmengen.php

http://scienceblogs.de/von_bits_und_bytes/2011/06/11/wie-rechner-rechnen-teil-1/