Rechnerarithmetik

Ac 2013-2021

Ganzzahlen, Gleitkommazahlen, Maschinengenauigkeit

Im folgenden soll ein Überblick über die in Computersystemen bzw. Programmiersprachen verwendeten Zahlen inklusive ausgewählter Algorithmen (in neutraler Notation) gegeben werden. Des öfteren wird auch Bezug auf die verbreitete Programmiersprache Java (Version 7) genommen .

Dastellung von Zahlen:

Computer (bzw. Compiler) stellen Zahlen durch Bitfolgen im Dualsystem (Basis b = 2) dar.
Bitfolgen sind Folgen, die ausschließlich aus den Ziffern 0 und 1 bestehen, z.B. `10011010110001` .

Zum Vergleich: Das Dezimalsystem verwendet die Ziffern 0 … 9 .

Der Wert einer Zahl im Dualsystem wird dabei genauso ermittelt wie im Dezimalsystem.

Beispiel b = 10: $276_{10} = 2 \cdot 10^2 + 7 \cdot 10^1 + 6 \cdot 10^0 = 200+70+6$
Beispiel b = 2: $110101_2 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 32+16+0+4+0+1 = 53_{10}$

Anmerkungen:

- Die tiefgestellte Zahl verdeutlicht jeweils die Basis .
- Das ganz links stehende Bit ist dasjenige mit dem höchsten Wert .

Ganzzahlen (engl.: integers):

Mit einer Bitfolgenlänge von n kann man nach den Gesetzen der Kombinatorik 2^n Zahlen darstellen.

Bei natürlichen Zahlen $0 \; 1 \; \ldots \; 2^n - 1$ und bei
ganzen Zahlen $\mathbf{-2^{n-1} \; -2^{n-1}+1 \; \ldots \; -1 \; 0 \; 1 \; 2 \; \ldots \; 2^{n-1}-1}$.

Beispiel n=4 Bit : `0 1 2 … 14 15` (natürliche Z.)
`-8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7` (ganze Z.)

Beispiel n=64 Bit : `0 1 … 18446744073709551615` (natürliche Z.)
`-9223372036854775808 … 9223372036854775807` (ganze Z.)

Für die ganzen Zahlen verwendet man folgende Dualdarstellung (hier für n=4 dargestellt):

1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011
-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3
0100	0101	0110	0111								
4	5	6	7								

Man erkennt, dass jede negative Zahl im höchstwertigen Bit eine 1 hat, während bei positiven Zahlen dort eine 0 steht . Das erste Bit stellt also das Vorzeichen dar (1 negativ ; 0 positiv) .

Diese Darstellung nennt man auch **Zweierkomplementdarstellung** . Die Bezeichnung rührt daher, dass jede Addition einer Zahl zu ihrem bitweisen Komplement (aus 0 wird 1 und aus 1 wird 0) stets als Ergebnis die Zahl `111...1` hat. Dies entspricht dezimal der Zahl `-1` .

Beispiel: `1011 + 0100 = 1111` .

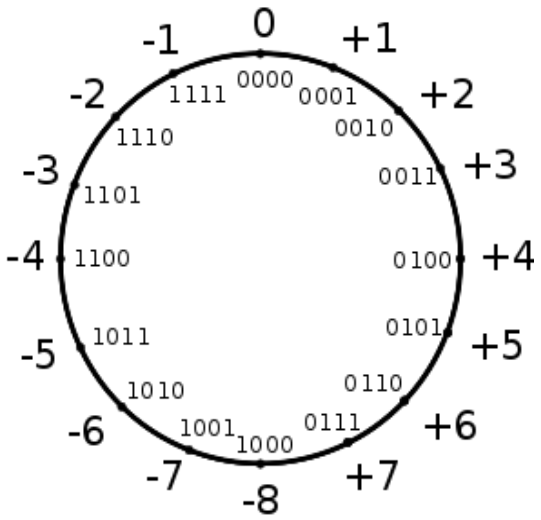
Regel für die Erzeugung einer negativen Zahl aus einer gegebenen positiven Zahl:
Bilde das Komplement der positiven Zahl und addiere anschließend die Zahl 1.
Achtung: Jede positive Zahl muss hierbei mit einer 0 beginnen (die 0 ggfs. einfügen) !!

Beispiel: $12_{10} = 01100_2$
Komplement+1: $10011 + 00001 = 10100 = 20_{10} - 32_{10} = -12_{10}$.

Die Zweierkomplementdarstellung kann man sich mithilfe eines Zahlenkreises veranschaulichen (siehe Abbildung).

Hier werden lediglich 4 Bits verwendet .
Damit können $2^4 = 16$ Zahlen dargestellt werden, -8 .. +7.

Man sieht u.a. , dass
- +8 bei 4 Bits aus dem Zahlenkreis herausfällt und durch -8 ersetzt wird.
- das höchstwertige Bit bei negativen Zahlen stets 1 ist, während bei allen positiven Zahlen dort eine 0 steht.



Wie sieht die Zweierkomplementdarstellung beim Datentyp **Integer** aus ?

Integer: 4 Bytes = 32 Bits ; Bereich: $-2^{31} .. 2^{31} - 1 =$ -2147483648 .. 2147483647

Addiert man zum höchsten positiven Wert 2147483647 die Zahl 1, so ergibt sich -2147483648 !

0
1
...
2147483646
2147483647
-2147483648
-2147483647
-2147483646
...
-2
1
0 (womit der Kreis wieder geschlossen ist)

Man kann sich folgende Zuordnungen vorstellen:
-2147483648 entspricht der positiven Zahl 2147483648 ;
-2147483647 entspricht der positiven Zahl 2147483649 ;
-2147483646 entspricht der positiven Zahl 2147483650 ; usw.

Allgemein gilt für Integers: Die Zahl -z (z > 0) entspricht der positiven Zahl $2^{32} - z$!

Zur Darstellung solcher positiven Zahlen eignet sich der vorzeichenlose Delphi-Datentyp **UInt32**.
Java kennt keine vorzeichenlosen Datentypen.

Für Int64 (in Java: long) ist die größte positive Zahl $2^{63} - 1 = 9223372036854775807 \approx 9 \cdot 10^{18}$.

Für Int64 bzw. long gilt: Die Zahl -z (z > 0) entspricht der positiven Zahl $2^{64} - z$!

Umrechnungmethoden / Algorithmen für Basisumwandlungen:

① Eine dezimale Ganzzahl z lässt sich folgendermaßen in ihre Dualdarstellung $b_{n-1}b_{n-2} \ldots b_1b_0$ umwandeln (**Basis 10 zu Basis 2**):

Algorithmus (10 ⟶ 2) :

```
i = 0
wiederhole
   b[i] = z mod 2
   z = z div 2
   i = i+1
bis z = 0
```

Die Stellen der Dualdarstellung (zunächst ohne Zweierkomplement) werden also von rechts nach links ermittelt !

Beispiel: z = 37 . Es wird fortwährend durch 2 dividiert und der Rest notiert:

```
37 / 2 = 18 Rest 1.
18 / 2 = 9 Rest 0.
9 / 2 = 4 Rest 1.
4 / 2 = 2 Rest 0.
2 / 2 = 1 Rest 0.
1 / 2 = 0 Rest 1.
```

Beim Ergebnis 0 endet der Algorithmus.
Die Reste (oberster Wert ganz rechts notiert) sind 100101.
Dies ist die Dualdarstellung von **37** .
Wählt man eine Bitlänge von n = 8, so ist die Dualdarstellung von 37 gleich 00100101 .

Bei negativen Zahlen muss noch das Zweierkomplement gebildet werden. Beispiel **-37** :
Die Dualdarstellung von +37 ist 00100101 (siehe oben) .

Das **Zweierkomplement** wird nun gebildet, indem alle Bits umgedreht werden (**Einerkomplement**) und anschließend 1 addiert wird:
00100101 → 11011010 (Einerkomplement)
b = 11011010 + 1 = **11011011** (8bit-Zweierkomplement von **-37**)

② Für die Umrechnung einer im Zweierkomplement vorliegenden n-bit-Dualzahl b (Bitfolge $b_{n-1}b_{n-2} \ldots b_1b_0$) in das Dezimalsystem kann man folgende Formel verwenden:

$$b_{n-1}b_{n-2}\ldots b_1b_0 = \left(\sum_{i=0}^{n-2} b_i \cdot 2^i\right) - b_{n-1} \cdot 2^{n-1}$$

Zweierkomplementzahl (**Basis 2 zu Basis 10**)

Algorithmus (2 ⟶ 10) :

```
z = b[n-2]
für i von n-3 ab bis 0 wiederhole
   z = 2·z + b[i]
z = z - b[n-1]·2^(n-1)
```

Ein Beispiel für eine **8bit**-Ganzzahl (n=8) :

1100	0101

11000101 = $1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 0 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 - 1 \cdot 2^7$ = 1+0+4+0+0+0+64-128
= 69-128 = **-59** dezimal

Ein Beispiel für eine **32bit**-Ganzzahl (n=32) :

1000	0000	0000	0000	0000	0000	0000	0000

Die Zahl ist $0 - 2^{31}$ = **-2147483648** dezimal

Ein Beispiel für eine **64bit**-Ganzzahl (n=64) :

0101	0001	1010	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111

Die Zahl ist 1+2+4+8+16+ ... $+2^{51}+0+2^{53}+0+2^{55}+2^{56}+0+0+0+2^{60}+0+2^{62}+0$
= **5886204712973238271** dezimal

Das Programmiersystem **Delphi** kennt verschiedene Typen von Ganzzahlen, u.a. auch solche ohne Vorzeichen (Tabelle):

Datentyp	Format	Bereich
Byte	8 Bit	0 … 2^8-1 = 0 … 255 (auch: **UInt8**)
Shortint	8 Bit	-2^7 … 2^7-1 = -128 … 127 (auch: **Int8**)
Word	16 Bit	0 … $2^{16}-1$ = 0 … 65535 (auch: **UInt16**)
Smallint	16 Bit	-2^{15} … $2^{15}-1$ = -32768 … 32767 (auch: **Int16**)
DWord	32 Bit	0 … $2^{32}-1$ = 0 … 4294967295 (auch: **UInt32**)
Integer	32 Bit	-2^{31} … $2^{31}-1$ = -2147483648 … 2147483647 (auch: **Longint Int32**)
QWord	64 Bit	0 … $2^{64}-1$ = 0 … 18446744073709551615 (auch: **UInt64**)
Int64	64 Bit	-2^{63} … $2^{63}-1$ = -9223372036854775808 … 9223372036854775807

Die Programmiersprache **Java** kennt verschiedene (vorzeichenbehaftete) Typen von Ganzzahlen (Tabelle):

Datentyp	Format	Bereich
byte	8 Bit	-2^7 … 2^7-1 = -128 … 127
short	16 Bit	-2^{15} … $2^{15}-1$ = -32768 … 32767
int	32 Bit	-2^{31} … $2^{31}-1$ = -2147483648 … 2147483647
long	64 Bit	-2^{63} … $2^{63}-1$ = -9223372036854775808 … 9223372036854775807

Zu beachten ist, dass bei Ganzzahloperationen ein **Überlauf** (englisch: overflow) auftreten kann, der bei Programmiersprachen (z.B. Java) nicht zu Fehlermeldungen führt !
Zum Beispiel liefert die Rechnung 2147483647+3 beim Datentyp int (Java) das Ergebnis -2147483646 und nicht wie erwartet +2147483650 .
Die Erklärung dafür ist in der Zweierkomplementdarstellung zu suchen (s.o.).

Demgegenüber sind **Operationen** mit Ganzzahlen (div usw.) empfindlich, wenn z.B. irgendeine Ganzzahl durch 0 geteilt wird. Java „wirft“ dann eine sog. „**ArithmeticException**“ .

Später wird zu sehen sein, dass Fließkommaoperationen diesbezüglich etwas anders reagieren.

<u>**Gleitkommazahlen**</u> <u>(engl.: floating point numbers):</u>

Eine Gleitkommazahl z hat die Darstellung $\mathbf{z = V \cdot M \cdot B^E}$
(V=Vorzeichen; M=Mantisse; B=Basis; E=Exponent)
Ein Beispiel ist $z = (-1) \cdot 0{,}342 \cdot 10^7$.

Wenn $\mathbf{1 \leq M < 10}$ gilt, dann nennt man die Gleitkommadarstellung **normalisiert**.
Damit erreicht man eine einheitliche Schreibweise. $0{,}342 \cdot 10^7$ wird dann geschrieben als $3{,}42 \cdot 10^6$.

Bei der Darstellung von Gleitkommazahlen hat sich ein Standard durchgesetzt, nämlich
IEEE 754 (IEEE = Institute of Electrical and Electronics Engineers) .

Zur Speicherung einer Gleitkommazahl verwendet der Computer laut IEEE 754 eine Folge von Dualzahlen (0 und 1) mit einer Länge von <u>32 Bits (einfache Genauigkeit)</u> bzw. <u>64 Bits (doppelte Genauigkeit)</u> .

Für <u>32bit-Zahlen</u> gilt:
Das erste Bit nimmt das Vorzeichen V der Zahl auf (0 = positiv; 1 = negativ). Es folgen 8 Bits für den Exponenten E der Basis 2 , und anschließend 23 Bits für den Nachkommateil M der Mantisse.

Für <u>64bit-Zahlen</u> gilt:
Das erste Bit nimmt das Vorzeichen V der Zahl auf (0 = positiv; 1 = negativ). Es folgen 11 Bits für den Exponenten E der Basis 2 , und anschließend 52 Bits für den Nachkommateil M der Mantisse.

VEEE	EEEE	EEEE	MMMM	MMMM	MMMM	MMMM	MMMM	MMMM	MMMM	MMMM	MMMM	MMMM	MMMM	MMMM	MMMM

Der **11-Bit-Exponent** E zur Basis 2 läuft von <u>E=0 bis E=2047</u> = 2^{11} -1 , somit ist E nichtnegativ !
Damit aber auch negative Exponenten berücksichtigt werden, interpretiert man <u>bei der Auswertung</u> die Bitfolge E so, dass man noch den sog. „**bias**“ (die Verschiebung) von E subtrahiert. Bei einem n-stelligen Exponenten wählt man $\mathbf{bias = 2^{n-1}-1}$. Für 64bit-Zahlen (n=11) gilt dann **bias=1023**.
Der Exponent E zur Basis 2 liegt somit (bei der Auswertung) im Bereich -1023 … +1024 .

Bei Gleitkommazahlen findet also (anders als bei Ganzzahlen) das „Zweierkomplement“ <u>keine</u> Anwendung !

<u>Achtung:</u> <u>E=2047</u> (dual: 11 … 1) wird verwendet für ±∞ (±Infinity) und NaN (Not a Number) .
Bei ±∞ ist außerdem M = 00 … 0 , bei NaN ist M ≠ 00 … 0 .
<u>E=0</u> (dual: 00 … 0) wird verwendet bei <u>fehlender führender 1</u> bei der Mantisse (s.u. (*)) .

Der **52-Bit-Nachkommateil** (Mantisse M) wird so interpretiert, als ob er die Form 1,M (= 1 + 0,M) hätte (Normalisierung; vgl. oben), wobei jedoch die führende 1 nicht notiert wird („hidden Bit“) .

Mit einer führenden 1 lässt sich aber die Zahl 0,0 nicht darstellen . Dieses Problem wird gelöst durch die Möglichkeit, E=0 als „**Denormalisierung**“ (es wird keine führende 1 angenommen) zu interpretieren (*) .
So braucht man nur die Mantisse M ebenfalls auf 0 zu setzen, und man hat die Zahl 0 codiert .

Für eine <u>**64 bit – Gleitkommazahl**</u> VEM berechnet man die zugehörige Dezimalzahl zdez so :

$$\mathbf{zdez} = (-1)^V \cdot 1.M \cdot 2^{E-1023} = (-1)^V \cdot (1+M \cdot 2^{-52}) \cdot 2^{E-1023}$$
$$= \mathbf{(-1)^V \cdot (M+2^{52}) \cdot 2^{E-1075}} \quad \text{(normalisiert)}$$

$$\mathbf{zdez} = (-1)^V \cdot 0.M \cdot 2^{1-1023} = (-1)^V \cdot M \cdot 2^{-52} \cdot 2^{-1022} = \mathbf{(-1)^V \cdot M \cdot 2^{-1074}} \quad \text{(denormalisiert)}$$

Für die Mantisse gilt: $\boxed{M = m_{51} m_{50} \ldots m_1 m_0 = \sum_{i=0}^{51} m_i \cdot 2^i}$

Beispiel:

0100	0001	1000	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111

Die Zahl ist positiv (V=0) . Der Exponent E = 1048 .

Die Mantisse: $M = 11...11 = \sum_{i=0}^{51} 1 \cdot 2^i = 4503599627370495$

Dann ist die Zahl zdez = $(-1)^0 \cdot (4503599627370495+2^{52}) \cdot 2^{1048-1075} =$
$9007199254740991 \cdot 2^{-27} = 6{,}7108863999999992 \cdot 10^7$.

Weitere Beispiele:

Die **Zahl 0** wird dargestellt (E=0 ; denormalisiert ; V=0 oder V=1 für „-0“ möglich)

0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

Die **größte** darstellbare 64bit-Gleitkommazahl (E=2046) ist

0111	1111	1110	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111

Wert = $(11 \ldots 1) \cdot 2^{2046-1075} = (2^{52}-1+2^{52}) \cdot 2^{971} = \mathbf{(2^{53}-1) \cdot 2^{971}} =$
$\mathbf{1{,}79769313486231570 \cdot 10^{308}}$

Die **kleinste** darstellbare (denormalisierte) 64bit-Gleitkommazahl (E=0) ist

0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0001

Wert = $(00 \ldots 01) \cdot 2^{-1074} = 1 \cdot \mathbf{2^{-1074}} = \mathbf{4{,}94065645841246544 \cdot 10^{-324}}$

Die Darstellung von **-∞** bzw. -Infinity (E=2047 ; M=0) ist

1111	1111	1111	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

Die Darstellung von +∞ bzw. +Infinity ist ist entsprechend, mit V=0 .

Die Darstellung von NaN (E=2047 ; M≠0) ist z.B.

1111	1111	1111	1000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

Es handelt sich hier um ein sog. **stilles NaN** , bei dem das höchstwertige Bit von M den Wert 1 hat . Bei stillen NaNs ist das Ergebnis nicht (mathematisch) definiert , z.B 0.0/0.0 , log(-1.0), √(-1.0) etc .

Es gibt auch NaNs, bei denen das höchstwertige Bit von M den Wert 0 hat (**anzeigende** NaNs). Mindestens 1 anderes Bit der Mantisse M muss aber den Wert 1 besitzen. Anzeigende NaNs unterstützen fortgeschrittene Eigenschaften wie gemischte numerische und symbolische Berechnungen oder andere grundlegende Gleitkommaarithmetik .

Interessant ist, dass Programmiersprachen (z.B. Java) bei mathematisch nicht definierten Fließkommaoperationen anders reagieren als bei entsprechenden Ganzzahloperationen. Bei Fließkommaoperationen wird in diesen Fällen keine „**ArithmeticException**“ geworfen, sondern es erscheint als Ergebnis z.B. Infinity oder NaN (vgl. Beispiele oben) .

Es ist noch zu klären, wie man zu einer gegebenen Dezimalzahl x die zugehörige Codierung VEM findet:

Beispiel: x = 37,1

Als erstes wird eine Dualdarstellung (Vorkomma und Nachkomma) von 37,1 erzeugt .
Der Vorkommateil 37 wird verarbeitet wie bereits bei Ganzzahlen gezeigt. Es gilt $\mathbf{37_{10} = 100101_2}$.
Beim Nachkommateil nk = 1 wird nach folgendem Algorithmus gerechnet:

```
mult = 10^m  // m ist die Länge von nk
wiederhole
   nk =  nk·2
   Schreib nk div mult
   nk = nk mod mult
bis nk = 0 oder  „Periode erkannt"
```

```
nk = 1      mult = 10
1·2 div 10 = 0      nk = 1·2 mod 10 = 2
2·2 div 10 = 0      nk = 2·2 mod 10 = 4
4·2 div 10 = 0      nk = 4·2 mod 10 = 8
8·2 div 10 = 1      nk = 8·2 mod 10 = 6
6·2 div 10 = 1      nk = 2·2 mod 10 = 2
2·2 div 10 = 0      nk = 2·2 mod 10 = 4
```

Man erkennt eine Periode (mit Vorperiode) ! Der duale Nachkommateil ist $0\overline{0011}$.
Die komplette Dualdarstellung für $37{,}1_{10}$ ist also $\mathbf{100101{,}0\overline{0011}_2}$.
Diese Dualdarstellung muss jetzt normiert werden, d.h. gesucht ist eine Form $1{,}M\cdot 2^E$.

Das Komma muss um 5 Stellen nach links verschoben werden. Das entspricht einer Multiplikation mit 2^5 .
Damit gilt: $100101{,}0\overline{0011} = 1{,}0010100\overline{0011}\cdot 2^5$. Also ist der Exponent $E = 5_{10}$.
Es muss aber noch der Biaswert von 1023 addiert werden, daher $E = 1028_{10} = 10000000100_2$.
Lässt man bei der Mantisse die führende 1 weg und ergänzt die Periode bis auf 52 Stellen, so ergibt sich als
Mantisse M = 0010100011001100110011001100110011001100110011001101 .
Anmerkung: Das letzte Bit entsteht durch **Aufrundung**, denn 1100(1) = 1101(0) = 1101 .

Für das Vorzeichen wird V = 0 gespeichert, denn 37,1 ist positiv .

Die dezimale 37,1 wird also im IEEE-Gleitkommaformat gespeichert als VEM =

0100	0000	0100	0010	1000	1100	1100	1100	1100	1100	1100	1100	1100	1100	1100	1101

Weitere Codierungsbeispiele:

Die Zahl $\pi \approx 3.1415926535897932384$ wird so codiert :

0100	0000	0000	1001	0010	0001	1111	1011	0101	0100	0100	0100	0010	1101	0001	1000

Wert = $(2570638124657944+2^{52})\cdot 2^{1024-1075} = 7074237752028440\cdot 2^{-51} =$
3,1415926535897931...
Nur die ersten 15 Nachkommastellen sind richtig. Das entspricht der Präzision von 64bit-Gleitkommazahlen.

Die Zahl **1.0** wird so codiert :

0011	1111	1111	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

Die Zahl **2.0** wird so codiert :

0100	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000

Abstand benachbarter Zahlen :

Ändert man das letzte Bit auf 1, so hat man die nächste hinter 2,0 darstellbare Gleitkommazahl erzeugt. Diese hat den Wert `2,0000000000000004440892098500626...` , unterscheidet sich also von 2,0 um den Wert `4,440892098500626E-16` .
In Java7 entspricht dies übrigens dem Aufruf von **Math.ulp(2.0)** .

Der **Abstand benachbarter Zahlen** ist bei großen Zahlen größer als bei kleinen Zahlen.

Beispiele (Java7):

```
Math.ulp(1.0)   = 2.220446049250313E-16
Math.ulp(10.0)  = 1.7763568394002505E-15
Math.ulp(100.0) = 1.4210854715202004E-14
```

Programmierung der Ausgabe auf dem Bildschirm (Java-Methoden):

```
public static String doubleToBinaryIEEE754(double x) {
     String sBits = Long.toBinaryString(Double.doubleToLongBits(x));
     // da führende Nullen weggelassen werden ...
     while (sBits.length() < 64)
          sBits = "0" + sBits;
     return sBits;
}

public static double binaryIEEE754ToDouble(String sBin) {
     double x = Double.longBitsToDouble(Long.valueOf(sBin, 2));
     return x;
}
```

Bemerkungen zur Genauigkeit bzw. Rundung bei Gleitkommazahlen:

1) Die Mantisse hat eine Bitlänge von 52 , dies entspricht
$[52 \cdot \log(2)+1] = 15$ bis 16 Dezimalen .

2) Gleitkommaarithmetik kann zu groben Rundungsfehlern führen.

Beispiel: Zu berechnen ist $64919121 \cdot 102558961 - 41869520{,}5 \cdot 159018721$

Exakte Rechnung mit Datentyp long: 6658037598793281-6658037598793280,5 = 0,5.

Rechnung mit Gleitkommazahlen wie z.B. dem Datentyp double:
Hier werden die Zwischenergebnisse in die normalisierte Form umgewandelt.
Dabei ist 6658037598793280,5 = 6,6580375987932805E15 .
Der Computer rundet jedoch diese Zahl nach unten ab zu 6.65803759879328E15 .
Somit wäre bei Verwendung von Gleitkommazahlen das Subtraktionsergebnis = 1,
und nicht 0,5 !

3) Gleitkommazahlen werden in der Regel **dual unexakt dargestellt**, d.h. das Bitmuster entspricht in den meisten Fällen nicht der dezimal vorgegebenen Zahl .

Beispiele:

① Die Dezimalzahl 0,25 soll ins Dualsystem übertragen werden:
$(0{,}25)_{10}=1/4=1/2^2 = (0{,}01)_2$.
Bei diesem Beispiel klappt die Umwandlung exakt .

② Bei der Dezimalzahl 0,1 ist das aber nicht der Fall, denn 1/10 lässt sich nicht in 2er-Potenzen zerlegen. Dual ergibt sich hier eine Periode (wie oben bereits per Algorithmus gezeigt !), denn $(0{,}1)_{10}=(0{,}0\overline{0011})_2$.
Der Computer speichert für die dezimale 0,1 folgendes Bitmuster:
0 01111111011 1001100110011001100110011001100110011001100110011010
V E=$(1019)_{10}$ Z=$(2702159776422298)_{10}$
Dezimal ist also die gespeicherte Zahl = 0,10000000000000000055511151... ≠ 0,1

Folgerungen für die Rechnung mit Gleitkommazahlen:

- Vermeide Addition von Zahlen sehr unterschiedlicher Größe, denn Stellen der kleineren Zahl können dabei verloren gehen.

- Vermeide Subtraktion von sehr dicht beieinander liegenden Zahlen, denn es besteht die Gefahr der Auslöschung (cancellation) .

- Vermeide die Division durch Zahlen, die sehr nahe bei Null liegen, denn es besteht die Gefahr des Überlaufs (overflow).

- Bei Summen ist die Reihenfolge der Summanden zu beachten. Kleinere Zahlen sollten zuerst verrechnet werden.

Maschinengenauigkeit:

Die Maschinengenauigkeit eines Computers ist die **kleinste positive Zahl ε** , für die $1 + \varepsilon > 1$ gilt.
Für alle Zahlen x zwischen 0 und ε gilt $1 + x = 1$.

Grund: 1 + x wird wegen der beschränkten Zahl an Dezimalen zu 1 gerundet !

Für **Java** (double) ermittelt man als Maschinengenauigkeit:
$\varepsilon = 2^{-52} \approx 2{,}22044604925031308 \cdot 10^{-16}$.

Für den **TI84** ermittelt man: Maschinengenauigkeit $\varepsilon = 2^{-39} \approx 1{,}8189894035458 \cdot 10^{-12}$.

Für **Calc** (Open Office) ermittelt man: Maschinengenauigkeit $\varepsilon = 2^{-47} \approx 7{,}105427357601 \cdot 10^{-15}$.

Für **Delphi7** (extended) ermittelt man: Maschinengenauigkeit $\varepsilon = 2^{-63} \approx 1{,}08420217248 \cdot 10^{-19}$.
Anmerkung: extended arbeitet bei der 32-bit-Version von Delphi mit 80bit – Speicher !
Achtung: 64-bit-Systeme von Delphi setzen unter Windows extended = double (64 bit).
Für double erhält man in Delphi7 (wie in Java) $\varepsilon = 2^{-52} \approx 2{,}22044604925031308 \cdot 10^{-16}$.

Vorsicht !

Rechnet man mit Zahlen in der Nähe der Maschinengenauigkeit, so können total falsche Ergebnisse entstehen.

Beispiel: Berechne $(1 + 2^{-39} - 1) \cdot 2^{39}$. Exaktes Ergebnis müsste 1 sein.
Der TI84 berechnet aber $0{,}989560465$.