Vorlesung: Numerik 1 für Ingenieure

Version 1.11.20

Michael Karow

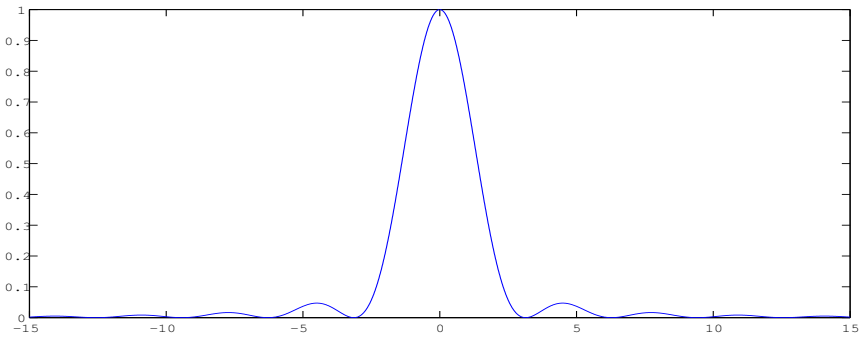
1. Vorlesung

Zahlendarstellung im Rechner

Zur Motivation drei Beispiele für fehlerhafte Berechnungen in MATLAB.

Beispiel 1: Wir betrachten die Funktion

$$f(x) = \frac{1 - \cos(2x)}{2\,x^2}$$



Aus dem Plot erkennt man, dass $f(10^{-9}) \approx 1$. MATLAB liefert aber $f(10^{-9}) = 0$.

Umformulierung: Es ist

$$f(x) = \frac{1 - \cos(2x)}{2\,x^2} = \frac{(\sin(x)^2 + \cos(x)^2) - (\cos(x)^2 - \sin(x)^2)}{2\,x^2} = \left(\frac{\sin(x)}{x}\right)^2. \qquad (*)$$

Rechnet man mit dem rechten Ausdruck in $(*)$, dann liefert MATLAB
die korrekte Näherung $f(10^{-9}) = 1$.

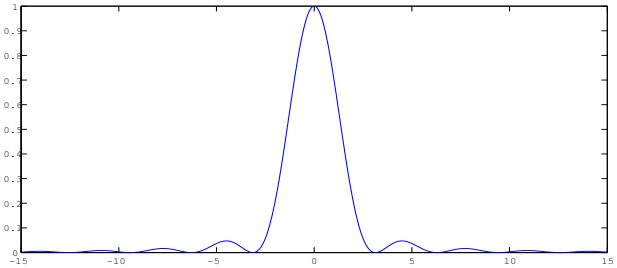
Erklärung des obigen Fehlers:

Im Rahmen der Rechengenauigkeit ist $\cos(2 * 10^{-9}) = \cos(0) = 1$. Daher

$$f(10^{-9}) = \frac{1 - \cos(2 * 10^{-9})}{4 * 10^{-18}} = \frac{0}{4 * 10^{-18}} = 0.$$

Einschub: MATLAB-Erzeugung des Plots

$$f(x) = \frac{1 - \cos(2x)}{2\,x^2}$$



Variante 1:

```
>> x=linspace(-10,10,200);
>> y=(1-cos(2*x))./(2*x.^2);
>> plot(x,y)
```

Erklärung: `x` ist ein Vektor mit 200 Einträgen zwischen -10 und 10. Das Semikolon verhindert, dass der Vektor auf den Bildschirm ausgegeben wird. `y` ist auch ein Vektor der Länge 200 mit Einträgen, die aus den Einträgen von `x` nach der angegebenen Formel ausgerechnet werden. Da es sich um Vektoroperationen handelt, muss ein Punkt vor der Division und der Potenzierung stehen. `plot(x,y)` erzeugt eine Zickzacklinie mit 200 Eckpunkten, die wie der Graph von `f` aussieht. In Variante 2 wird die Definition von `y` direkt in `plot` geschrieben.

Variante 2:

```
>> x=linspace(-10,10,200);
>> plot(x,(1-cos(2*x))./(2*x.^2))
```

In der 3. Variante wird eine (sogenannte anonyme) Funktion definiert.

Variante 3:

```
>> f=@(x)(1-cos(2*x))./(2*x.^2)
>> x=linspace(-10,10,200);
>> plot(x, f(x))
```

Beispiel 2:

Für jede reelle Zahl $k \in \mathbb{R}$ ist

$$(345 + 10^k) - 10^k = 345.$$

MATLAB rechnet

$$\begin{aligned}
(345 + 10^{15}) - 10^{15} &= 345 \\
(345 + 10^{16}) - 10^{16} &= 344 \\
(345 + 10^{17}) - 10^{17} &= 352 \\
(345 + 10^{18}) - 10^{18} &= 384 \\
(345 + 10^{19}) - 10^{19} &= 0
\end{aligned}$$

Erklärung:

Für große k wird die Zahl $345 + 10^k$ nicht exakt im Rechner abgespeichert (wegen zu großer Mantissenlänge).

Im Rahmen der Rechengenauigkeit ist z.B. $345 + 10^{19} = 10^{19}$, weil 345 im Vergleich zu 10^{19} sehr klein ist. Beachte aber, dass

$$345 + (10^{19} - 10^{19}) = 345,$$

denn bei dieser Klammerung wird die Differenz zuerst (und korrekt) ausgerechnet.

Beispiel 3:

MATLAB rechnet $1234 * (0.1 + 0.1 + 0.1 - 0.3)^{1/10} = 29.22\ldots$.

Der exakte Wert ist aber 0.

Erklärung:

MATLAB speichert Zahlen nicht im Dezimalsystem, sondern im Binärsystem
(Genaueres auf den folgenden Seiten)
und kann daher die Zahlen 0.1 und 0.3 intern nicht exakt darstellen.

Daher rechnet MATLAB

$$0.1 + 0.1 + 0.1 - 0.3 = 5.55 * 10^{-17}$$

Dieser kleine Rechenfehler wird extrem verstärkt, wenn man die 10te Wurzel zieht.

Fragen:

1. Wie speichert ein Rechner Zahlen ab? Wie rechnet er?

2. Wie verstärken sich Rechenfehler?

Erinnerung an das Dezimalsystem

Im Dezimalsystem wird jede Zahl durch eine endliche oder unendliche Folge der zehn Symbole (Ziffern) 0,1,2,3,4,5,6,7,8,9 dargestellt. Zusätzlich gibt es ein Trennzeichen (Komma im Deutschen bzw. Punkt im Englischen) zwischen dem ganzzahligem und dem nicht ganzzahligen Anteil. Die Position einer Ziffer und das Trennzeichen bestimmen die Gewichtung mit einer Zehnerpotenz.

$$
\begin{aligned}
273.534 &= 2*10^{2}+7*10^{1}+3*10^{0}+5*10^{-1}+3*10^{-2}+4*10^{-3}\\
1/3 &= 0.33333\ldots = 0.\overline{3} = 3*10^{-1}+3*10^{-2}+3*10^{-3}+\ldots\\
17/8 &= 2.125 = 2*10^{0}+1*10^{-1}+2*10^{-2}+5*10^{-3}\\
22/7 &= 3.142857142857\ldots = 3.\overline{142857}\\
\pi &= 3.14159265358\ldots \quad \text{(keine Regelmäßigkeit)}\\
1 &= 0.9999\ldots = 0.\overline{9}
\end{aligned}
$$

Problem: Ein Bruch (Quotient ganzer Zahlen) ist nur dann durch eine endliche Ziffernfolge darstellbar, wenn der Nenner durch Erweiterung zu einer Zehnerpotenz gemacht werden kann. Für jeden anderen Bruch wird die Ziffernfolge ab einer Stelle periodisch. Irrationale Zahlen haben eine nichtperiodische unendliche Ziffernfolge.

Oft ist es praktikabel, eine Zahl in sogenannter **Fließpunktform** (synonym: **Gleitpunktform**) darzustellen. Dabei wird eine Ziffernfolge (Mantisse) mit einer Zehnerpotenz multipliziert. Der Exponent bestimmt die Größenordnung der Zahl. Beispiel:

$$\text{Lichtgeschwindigkeit im Vakuum} = 2.99729458*10^{5}\ \frac{km}{s}$$

Allgemeine Fließpunktzahlen (floating point numbers)

An der Zahl Zehn als Basis für ein Stellenwertsystem ist nichts besonderes, abgesehen davon, dass der Mensch zehn Finger hat. Tatsächlich kann man jede ganze Zahl $b \geq 2$ als Basis nehmen. Grundlage dafür ist folgender Satz.

Satz: Sei $b \geq 2$ eine ganze Zahl und $x \in \mathbb{R}$, $x \geq 0$. Dann gibt es ein $k \in \mathbb{Z}$ und eine endliche oder unendliche Folge $z_0, z_1, z_2, z_3 \ldots \in \{0, 1, 2, \ldots, b-1\}$ mit $z_0 \neq 0$, so dass

$$\begin{aligned} x &= z_0 * b^k + z_1 * b^{k-1} + z_2 * b^{k-2} + z_3 * b^{k-3} \ldots \\ &= (z_0 + z_1 * b^{-1} + z_2 * b^{-2} + z_3 * b^{-3} \ldots) * b^k \end{aligned}$$

Die Folge z_j ist eindeutig, wenn man den Fall $z_j = b - 1$ für alle $j \geq j_0$ ausschließt.

Notation:

$$\begin{aligned} x &= (z_0 . z_1\, z_2\, z_3 \ldots)_b * b^k \qquad (*) \\ &= (z_0\, z_1 \ldots z_k \,.\, z_{k-1}\, z_{k-2} \ldots)_b \quad \text{falls } k > 0 \end{aligned}$$

Terminologie: b=Basis, k=Exponent, Symbole für die z_j = Ziffern,
Ziffernfolge=Mantisse.
$(*)$ heißt normalisierte Gleitpunktdarstellung, $z_0 \neq 0$.

Für das Rechnen mit Computern wichtige Basen:
$b = 2, 8, 16$ (Dual-, Oktal- und Hexadezimalsystem)

Es werden Ziffern (Symbole) für die Zahlen $0, 1, \ldots, b-1$ benötigt.

Ziffern im Hexadezimalsystem: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$

Ein anderes interessantes Zahlensystem ist das Duodezimalsystem (Zwölfersystem) mit Ziffern 0-9,X,E, siehe Wikipedia.

Beispiele für die Darstellung einer Zahl im Binär- und im Hexadezimalsystem

$$
\begin{aligned}
(27.625)_{10} &= \left(16+8+2+1+\frac{1}{2}+\frac{1}{8}\right)_{10} \\
&= 1\cdot 2^4+1\cdot 2^3+0\cdot 2^2+1\cdot 2^1+1\cdot 2^0+1\cdot 2^{-1}+0\cdot 2^{-2}+1\cdot 2^{-3} \\
&= (11011.101)_2 \\
&= \left(16+11+\frac{10}{16}\right)_{10} \\
&= (1b.a)_{16}
\end{aligned}
$$

$$
\begin{aligned}
(42)_{10} &= (32+8+2)_{10} = (101010)_2 \\
&= (2*16+10)_{10} = (2a)_{16}
\end{aligned}
$$

Konvertierung einer ganzen Dezimalzahl in ein anderes Zahlensystem

Beispiel: Binärsystem (synonym: Dualsystem). Ziffern: 0,1

Aufgabe: Die Dezimalzahl 27 soll im Binärsystem dargestellt werden.

Methode: wiederholte Teilung durch 2 mit Rest. Reste geben Ziffern.

Ansatz: $(27)_{10} = (\ldots z_2\, z_1\, z_0)_2$

$$
\begin{array}{lcll}
(27)_{10} = 2*(13)_{10}+1 & = & 2*(\ldots z_2\, z_1)_2 + z_0 & \Rightarrow \quad (13)_{10} = (\ldots z_2\, z_1)_2 \text{ und } z_0 = 1 \\
(13)_{10} = 2*(6)_{10}+1 & = & 2*(\ldots z_3\, z_2)_2 + z_1 & \Rightarrow \quad (6)_{10} = (\ldots z_3\, z_2)_2 \text{ und } z_1 = 1 \\
(6)_{10} = 2*(3)_{10}+0 & = & 2*(\ldots z_4\, z_3)_2 + z_2 & \Rightarrow \quad (3)_{10} = (\ldots z_4\, z_3)_2 \text{ und } z_2 = 0 \\
(3)_{10} = 2*(2)_{10}+1 & = & 2*(\ldots z_5\, z_4)_2 + z_3 & \Rightarrow \quad (2)_{10} = (\ldots z_5\, z_4)_2 \text{ und } z_3 = 1 \\
(2)_{10} = 2*(1)_{10}+0 & = & 2*(\ldots z_6\, z_5)_2 + z_4 & \Rightarrow \quad z_j = 0 \text{ für } j \geq 6,\ z_5 = 1 \text{ und } z_4 = 0
\end{array}
$$

Ergebnis: $(27)_{10} = (11011)_2$

Probe: $(11011)_2 = 1*1+1*2+0*4+1*8+1*16$

Konvertierung einer Dezimalzahl < 1 in ein anderes Zahlensystem

Beispiel: Binärsystem (synonym: Dualsystem). Ziffern: 0,1

Aufgabe: Die Dezimalzahl 0.7 soll im Binärsystem dargestellt werden.

Ansatz: $(0.7)_{10} = (0.z_1\, z_2\, z_3\, \ldots)_2$ [Rechenweg: beide Seiten mal 2 nehmen und Anteile vor und hinter dem Punkt vergleichen]

$$
\begin{aligned}
&\Rightarrow (1.4)_{10} = 2*(0.7)_{10} = (z_1.z_2\, z_3 \ldots)_2 \Rightarrow z_1 = 1 \text{ und } (0.4)_{10} = (0.z_2\, z_3 \ldots)_2 \\
&\Rightarrow (0.8)_{10} = 2*(0.4)_{10} = (z_2.z_3\, z_4 \ldots)_2 \Rightarrow z_2 = 0 \text{ und } (0.8)_{10} = (0.z_3\, z_4 \ldots)_2 \quad (1) \\
&\Rightarrow (1.6)_{10} = 2*(0.8)_{10} = (z_3.z_4\, z_5 \ldots)_2 \Rightarrow z_3 = 1 \text{ und } (0.6)_{10} = (0.z_4\, z_5 \ldots)_2 \\
&\Rightarrow (1.2)_{10} = 2*(0.6)_{10} = (z_4.z_5\, z_6 \ldots)_2 \Rightarrow z_4 = 1 \text{ und } (0.2)_{10} = (0.z_5\, z_6 \ldots)_2 \\
&\Rightarrow (0.4)_{10} = 2*(0.2)_{10} = (z_5.z_6\, z_7 \ldots)_2 \Rightarrow z_5 = 0 \text{ und } (0.4)_{10} = (0.z_6\, z_7 \ldots)_2 \\
&\Rightarrow (0.8)_{10} = 2*(0.4)_{10} = (z_6.z_7\, z_8 \ldots)_2 \Rightarrow z_6 = 0 \text{ und } (0.8)_{10} = (0.z_7\, z_8 \ldots)_2 \quad (2) \\
&\vdots
\end{aligned}
$$

Zeilen (1) und (2) sind identisch $\Rightarrow$ periodische Ziffernfolge.

Ergebnis: $7/10 = (0.7)_{10} = (\,0.1\overline{0110}\,)_2$

Weitere Beispiele:

$$
\begin{aligned}
1/4 &= (0.25)_{10} = (0.01)_2 \\
3/8 &= (0.375)_{10} = (0.011)_2 \\
10/16 &= (0.625)_{10} = (0.a)_{16} \\
1/7 &= (0.\overline{142857})_{10} = (0.1)_7
\end{aligned}
$$

Warum Binär- und Hexadezimalsystem?

Elementare Computerspeicher haben zwei Zustände, symbolisiert mit 0 und 1.
Die Entscheidung zwischen diesen Zuständen ist die

kleinste Informationseinheit = 1 Bit (binary digit)

Größere Informationseinheiten:

8 Bit = 1 Byte, 1 Kilobyte = Tausend Byte, 1 Megabyte = 1 Million Byte, ...

Die Darstellung einer Zahl oder eines Speicherinhalts mit 0 und 1 ist oft zu lang und unbequem. Daher werden jeweils 4 Binärziffern zu einer Hexadezimalziffer zusammengefasst. Beispiel:

$$\underbrace{1\,0\,0\,1}_{9}\,\underbrace{1\,1\,1\,1}_{f}\,\underbrace{1\,1\,0\,1}_{d}\,\underbrace{0\,0\,1\,0}_{2}$$

Es ist nützlich, die folgenden 2er-Potenzen zu kennen:

$$2^8 = 256, \qquad 2^9 = 512 \qquad 2^{10} = 1024, \qquad 2^{11} = 2048.$$

1 addieren im Binärsystem

Geometrische Summenformel: $1 + q + q^2 + \ldots + q^{n-1} = \dfrac{1 - q^n}{1 - q}, \quad q \in \mathbb{R},\ q \neq 1.$

Für $q = 2$ folgt:

$$\overbrace{(11\ldots111)_2}^{n\ 1en} \;=\; 1 + 2 + 4 + \ldots + 2^{n-1} = \frac{1 - 2^n}{1 - 2} = 2^n - 1 = (1\overbrace{00\ldots000}^{n\ 0en})_2 - 1.$$

Also

$$(11\ldots111)_2 + 1 \;=\; (100\ldots000)_2$$

Andere Zahlen erhöht man um 1, indem man die erste 0 von rechts durch 1 ersetzt und alle 1en dahinter durch 0. Beispiel:

$$(1010110111)_2 + 1 = (1010111000)_2.$$

Bemerkung: Im Dezimalsystem hat man analog $(9999)_{10} + 1 = (10000)_{10}$.
Dies folgt auch aus der geometrischen Summenformel.

Addition von Binärzahlen

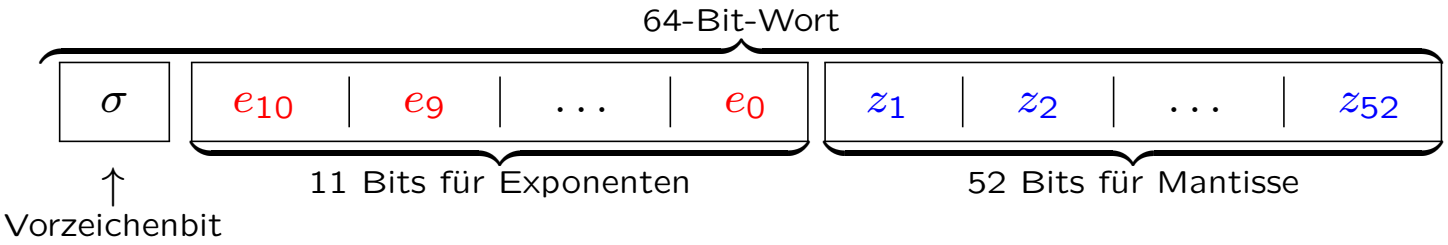
Die Addition von Binärzahlen geschieht nach den gleichen Regeln wie die aus der Grundschule bekannte schriftliche Addition von Dezimalzahlen: Man beginnt mit der letzten Stelle. Ziffern werden einzeln addiert. Wenn das Ergebnis sich nicht mehr durch eine Ziffer darstellen lässt, macht man einen Übertrag, der in die nächste Addition einbezogen wird.

Im folgenden Beispiel werden x=54 und y=39 als Binärzahlen addiert.

	1	1	0	1	1	0	$\leftarrow x$
	1	0	0	1	1	1	$\leftarrow y$
1	0	0	1	1	0		Übertrag
1	0	1	1	1	0	1	$\leftarrow x+y$

Beachte:
Die Regel zum Addieren ist im Dezimalsystem etwas komplizierter.
Man muss da kleine 1+1 beherrschen.

Matlab rechnet mit dem IEEE-Standard 754 (von 1985)
für doppeltes Grundformat (double precision)



Durch diesen Speicherinhalt wird die Zahl

$$\boxed{x = (-1)^{\sigma} * (1.\, z_1\, z_2\, \ldots z_{52})_2 * 2^{(e_{10}\, e_9 \ldots e_0)_2 - (1023)_{10}}}$$

dargestellt, sofern

$$(0, 0, \ldots 0) \neq (e_{10}\, e_9\, \ldots e_0) \neq (1, 1, \ldots, 1).$$

Wenn $(e_{10}\, e_9\, \ldots e_0) = (0, 0, \ldots 0)$, dann ist (subnormale Zahl)

$$\boxed{x = (-1)^{\sigma} * (0.z_1 z_2\, z_3\, \ldots z_{52})_2 * 2^{-(1022)_{10}}.}$$

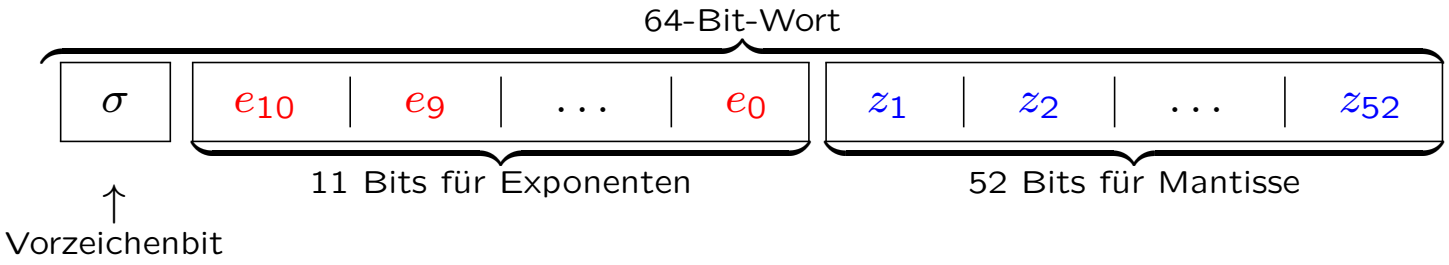
Wenn $(e_{10}\, e_9\, \ldots e_0) = (1, 1, \ldots, 1)$ und $z_1 = z_2 = \ldots = z_{52} = 0$, dann ist $x = \pm$`INF`.
Wenn $(e_{10}\, e_9\, \ldots e_0) = (1, 1, \ldots, 1)$ und $z_1 = 1$, $z_2 = \ldots = z_{52} = 0$, dann ist $x =$ `NAN`.

(`INF`='infinity' steht für Zahlen (einschließlich ∞), die größer als die größte darstellbare reelle Zahl (`realmax`) sind. Beispiel: $1/0 =$ `INF`. `NAN` steht für 'not a number'. `NAN` ist das Ergebnis von Rechnungen, die nicht sinnvoll sind. Beispiel: 0/0=`NAN`)

Bemerkung: IEEE = Institute of Electrical and Electronics Engineers

Die größte darstellbare Zahl `realmax`

Speicherinhalt x:



Durch diesen Speicherinhalt wird die (normale) Zahl

$$\boxed{x = (-1)^{\sigma} * (1.\, z_1\, z_2\, \ldots z_{52})_2 \; * \; 2^{(e_{10}\, e_9 \ldots e_0)_2 - (1023)_{10}}}$$

dargestellt, sofern

$$(0, 0, \ldots 0) \neq (e_{10}\, e_9\, \ldots e_0) \neq (1, 1, \ldots, 1).$$

Die größte so darstellbare Zahl ist ($z_1 = \ldots = z_{52} = 1$, $e_{10} = \ldots = e_1 = 1$, $e_0 = 0$),

$$\begin{aligned} x &= (1.11\ldots1)_2 * 2^{(11\ldots110)_2 - (1023)_{10}} \\ &= (2 - 2^{-52})_{10} * 2^{(2^{11} - 2 - 1023)_{10}} \\ &= (2 - 2^{-52})_{10} * 2^{(1023)_{10}} \\ &\approx 1.79769 * 10^{308} \end{aligned}$$

Diese Zahl wird in MATLAB als `realmax` bezeichnet.

Alle größeren Zahlen werden als `inf`=infinity= unendlich dargestellt.

Rechnungen mit `inf` und `nan`

`inf`=infinity=unendlich, `nan`=not a number

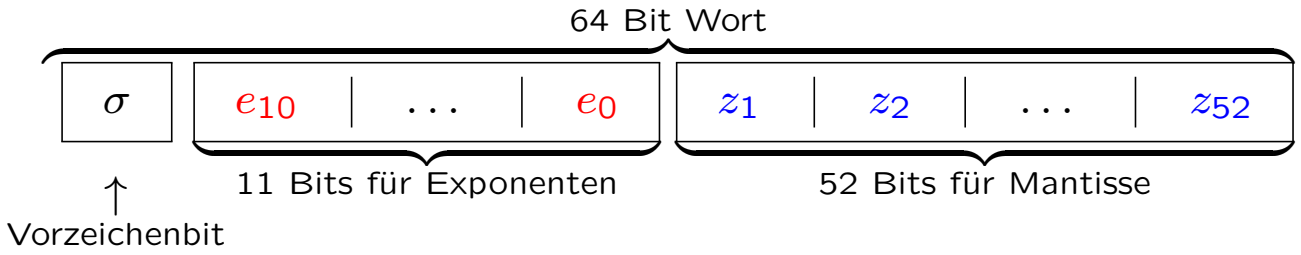
`nan` ist das Ergebnis von mathematisch nicht definierten Rechnungen.

```
1/0 = inf,   -1/0=-inf,   0/0=nan,

2+inf=inf,  2-inf=-inf,  inf+inf=inf,  inf-inf=nan

2*inf=inf,  (-2)*inf=-inf,  inf*inf=inf,  inf/inf=nan,

1/inf=0,  inf/inf=nan,


2^inf=inf^2=inf^inf=inf,  inf^0=1
```

Alle Rechnungen mit `nan` ergeben `nan`

Die kleinste positive normale Zahl `realmin`

Speicherinhalt x:



Durch diesen Speicherinhalt wird die Zahl

$$x = (-1)^{\sigma} * (1.\, z_1\, z_2\, \ldots\, z_{52})_2 * 2^{(e_{10}\, e_9 \ldots e_0)_2 - (1023)_{10}}$$

dargestellt, sofern

$$(0, 0, \ldots 0) \neq (e_{10}\, e_9\, \ldots e_0) \neq (1, 1, \ldots, 1)$$

Die kleinste so darstellbare positive Zahl erhält man für

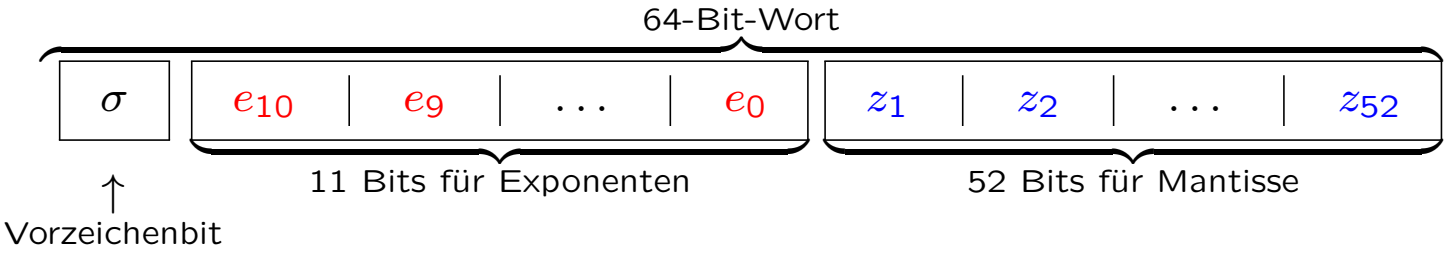
$$e_{10} = e_9 = \ldots = e_1 = 0, \qquad e_0 = 1 \qquad z_1 = z_2 = \ldots = z_{52} = 0.$$

Also:

$$x = 2^{-(1022)_{10}} =: \texttt{realmin} \approx 2.22 * 10^{-308}.$$

Die kleinste positive subnormale Zahl

Speicherinhalt x:



Wenn $(e_{10}\, e_9 \ldots e_0) = (0, 0, \ldots 0)$, dann ist

$$x = (-1)^{\sigma} * (0.z_1 z_2\, z_3 \ldots z_{52})_2 * 2^{-(1022)_{10}}.$$

Die kleinste so darstellbare positive Zahl erhält man für

$$z_1 = z_2 = \ldots = z_{51} = 0, \qquad z_{52} = 1.$$

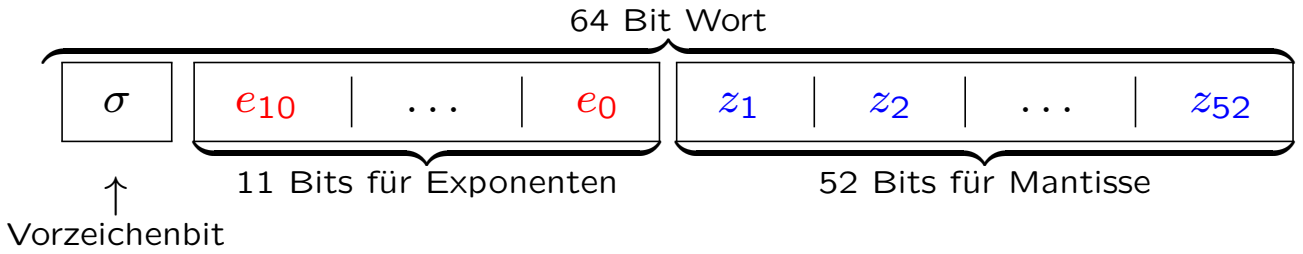
Also:

$$x = 2^{-(1074)_{10}} \approx 4.94 * 10^{-324}.$$

Die Maschinengenauigkeit eps

Definition: eps ist der Abstand der Zahl 1 zur nächst größeren Maschinenzahl.

Speicherinhalt x:



Durch diesen Speicherinhalt wird die Zahl

$$x = (-1)^{\sigma} * (1.\, z_1\, z_2\, \ldots\, z_{52})_2 * 2^{(e_{10}\, e_9 \ldots e_0)_2 - (1023)_{10}}$$

dargestellt, sofern

$$(0, 0, \ldots 0) \neq (e_{10}\, e_9\, \ldots e_0) \neq (1, 1, \ldots, 1)$$

Die kleinste Maschinenzahl x, die größer als 1 ist, erhält man für

$$e_{10} = 0, \qquad e_9 = e_8 = \ldots = e_0 = 1, \qquad z_1 = z_2 = \ldots = z_{51} = 0, \qquad z_{52} = 1.$$

Also:

$$x = 1 + 2^{-52} \qquad \Rightarrow \qquad \text{eps} = 2^{-52} \approx 2.22 * 10^{-16}.$$

Folgerung: Im 'double precision'-Format kann eine (nicht zu große und nicht zu kleine) reelle Zahl auf 15 Dezimalstellen genau abgespeichert werden.

Beispiel: ein Zehntel als 64-Bit-Zahl

Exakte Darstellung von $(1/10)_{10}$ als Binärzahl:

$$\left(\frac{1}{10}\right)_{10} = 0.0001\,\overline{1001} = \frac{1}{16}\left(1 + 9\sum_{k=1}^{\infty} 16^{-k}\right)$$

Bei Eingabe von $x =$`0.1` wird die (aufgerundete) Zahl

$$\tilde{x} = 0.0001\,\underbrace{(1001...1001)}_{\text{12 Blöcke}}\,101 = \frac{1}{16}\left(1 + 9\sum_{k=1}^{12} 16^{-k}\right) + 10 * 16^{-14}$$

abgespeichert.

Merke: Wenn ein Rechner nur Zahlen im Binärsystem speichert, dann kann 1/10 nicht exakt gespeichert werden. 1/10 ist dann keine Maschinenzahl.
Dies sieht man bei der Ein- und Ausgabe aber nicht, weil dafür die intern gespeicherte Binärzahl in eine Dezimalzahl umgerechnet wird.

Häufig gestellte Frage:

Obiges Problem würde nicht auftreten, wenn der Rechner im Dezimalsystem rechnen würde. Geht das nicht? **Antwort:** Man könnte das Rechnen im Dezimalsystem programmieren. Das Speichern von und Rechnen mit Dezimalzahlen kostet aber mehr Speicherplatz und Rechenzeit. Rechnungen im Dezimalsystem haben kompliziertere Regeln als im Binärsystem.

Bemerkungen zur Zahlenausgabe in MATLAB

MATLAB stellt verschiedene Ausgabeformate für den Speicheinhalt zur Verfügung. Voreingestellt ist die Ausgabe im `format short` mit 5 Ziffern für die Mantisse. Beispiel:

```
>> x=tan(1.57)

   x =

      1.2558e+03
```

Die ausgegebene Zahl bedeutet $1.2558 * 10^3$. Die genaue Anzeige von `x` im Dezimalsystem bekommt durch Umstellung auf `format long`. Beispiel:

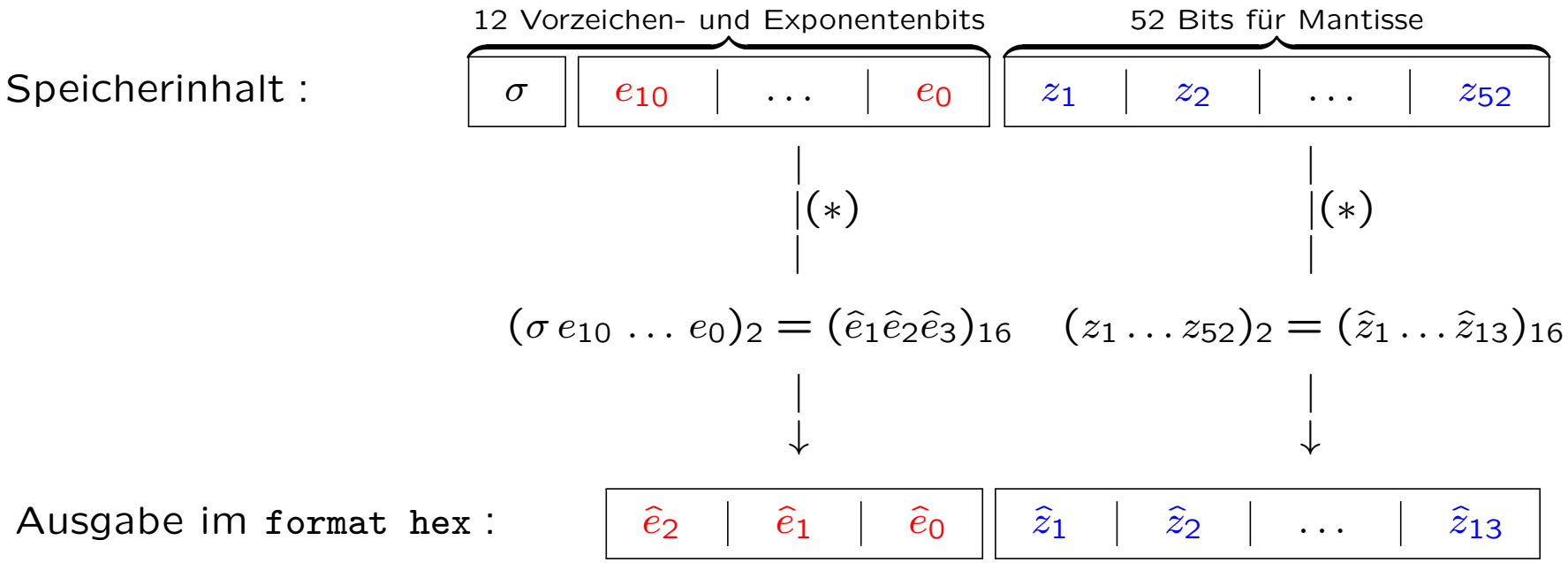
```
>> format long

>> x=tan(1.57)

   x=

     1.255765591500790e+03
```

Das Ausgabeformat `hex` wird auf den folgenden Seiten besprochen. Weitere Ausgabenformate werden in der MATLAB-Dokumentation erklärt. Dazu in MATLABs Command Window folgendes eingeben:

```
 >> doc format
```

Ausgabe des Speicherinhalts in MATLAB mit `format hex`

Den exakten abgespeicherten Wert (d.h. das 64-Bit-Wort) einer Variablen `x` bekommt man angezeigt, wenn man zunächst die Anweisung **`format hex`** und dann `x` eingibt. Die Ausgabe ist dann eine 16-stellige Hexadezimalzahl, die nach folgendem Schema berechnet wird. (Bemerkung: Nach Eingabe des Kommandos `format` oder `format long` werden wieder alle Zahlen als Dezimalzahlen angezeigt.)



(∗) Bits als Ziffern einer Binärzahl auffassen und diese als Hexadezimalzahl schreiben. Dies läuft darauf hinaus, jeweils 4 aufeinander folgende Bits als Ziffer $\hat{e}_k, \hat{z}_k \in \{0, 1, \ldots, 9, a, b, c, d, e, f\}$ darzustellen.

Beispiel zum `format hex`

Frage: Eine Zahl x wird im `format hex` als Ziffernfolge

$$x_{\text{hex}} = \texttt{c04a8000000000}$$

angezeigt. Welche Zahl ist das?

Antwort:

Übersetzung der ersten drei Ziffern:

$$\begin{aligned} \texttt{c04} &= 12 * 16^2 + 0 * 16 + 4 \\ &= (1100)_2 * 16^2 + (0000)_2 * 16 + (0100)_2. \end{aligned}$$

$\Rightarrow \sigma\, e_{10}\, e_9 \ldots e_0 = 11000000100$

$\Rightarrow \sigma = 1, \quad (e_{10} \ldots e_0)_2 = (10000000100)_2 = 2^{10} + 2^2 = (1028)_{10}.$

Übersetzung der restlichen Ziffern:

$a = (1010)_2$, $8 = (1000)_2$, $0 = (0000)_2 \Rightarrow$

$$a8000000000 = 10101000\underbrace{0 \ldots\ldots 0}_{44\text{Nullen}}$$

Insgesamt ergibt sich

$$\begin{aligned} x &= -1 * (1.10101000\,0 \ldots\ldots 0)_2 * 2^{(1028)_{10} - (1023)_{10}} \\ &= -1 * (1 + 2^{-1} + 2^{-3} + 2^{-5}) * 2^5 \\ &= (-53)_{10} \end{aligned}$$

Der Rechner macht bei der Ausführung der 4 Grundrechnungsarten in der Regel einen Fehler

Beispiel: Angenommen, es wird im Dezimalsystem mit Mantissenlänge 4 gerechnet.

Aufgabe für den Rechner: addiere 1 und $5.431 * 10^{-3} = 0.00543$.

Rechnung: $(1.000 + 0.005431) * 10^0 = 1.005 * 10^0$

Die letzten 3 Stellen sind verloren gegangen
(wegen Beschränkung der Mantissenlänge).

Subtraktion von ungefähr gleich großen Zahlen (Auslöschung)

Beispiel:

	Exakte Werte:	Werte im Rechner:
	$x_1 = 0.10024$	$\widetilde{x}_1 = 0.1002$
	$x_2 = 0.10011$	$\widetilde{x}_1 = 0.1001$
Differenz:	$x_1 - x_2 = 0.00013$	$\widetilde{x}_1 - \widetilde{x}_2 = 0.0001$
	$= 0.13 * 10^{-3}$	$= 0.1 * 10^{-3}$

Die Differenz $\widetilde{x}_1 - \widetilde{x}_2$ wird (in diesem Beispiel) exakt berechnet.
Trotzdem gibt es eine starke Vergrößerung des relativen Fehlers, denn

$$\left|\frac{\widetilde{x}_1 - x_1}{x_1}\right| \approx 4 * 10^{-4}, \qquad \left|\frac{\widetilde{x}_2 - x_2}{x_2}\right| \approx 1 * 10^{-4},$$

aber

$$\left|\frac{(\widetilde{x}_1 - \widetilde{x}_2) - (x_1 - x_2)}{x_1 - x_2}\right| \approx 2 * 10^{-1}$$

Man verliert 3 Stellen (3 Ziffern) Genauigkeit.
(x_i und $\widetilde{x}_i$ stimmen in 4 Ziffern überein, $x_1 - x_2$ und $\widetilde{x}_1 - \widetilde{x}_2$ stimmen nur noch in der ersten von 0 verschiedenen Ziffer überein)

Dieses Phänomen bezeichnet man als **Auslöschung**.

$\Rightarrow$ Subtraktion von nahezu gleich großen Zahlen vermeiden.