Methodische und Praktische Grundlagen der Informatik 4

Übungsblatt 2

mpgi4@cg.tu-berlin.de

WiSe 2014/2015

Gleitkommazahlen

Die Darstellung von reellen Zahlen im Computer birgt, auf Grund der endlichen Resourcen, welche dort nur vorhanden sind, große Herausforderungen. Die Dezimaldarstellung von Zahlen in der Form

$$x_1 \cdots x_m, x_{m+1} \cdots x_n = x_1, x_2 \cdots x_n \times 10^{m-1}, \tag{1a}$$

zum Beispiel

$$124,26749 = 1,2426749 \times 10^2, \tag{1b}$$

ist grundsätzlich für eine Abbildung im Computer geeignet, da es ausreicht, die Zahlenfolge $x_1 \cdots x_n$ und die Position des Kommas zu speichern. Durch die endliche Menge von Speicher, welche in Computern nur zur Verfügung steht, können irrationale Zahlen wie zum Beispiel π und rationale Zahlen mit einer "sehr langen" Dezimaldarstellung jedoch nicht exakt dargestellt werden. Für beliebige reelle Zahlen ist also nur eine approximative Darstellung im Computer möglich. Verschiedene Ansätze für eine solche Darstellung existieren. Bei jedem dieser Ansätze wird versucht, den Approximationsfehler sowohl für die Darstellung von Zahlen als auch für entstehende Folgefehler bei Rechnungen möglichst gering zu halten, bei möglichst geringem Aufwand für die Durchführung von Berechnungen.

Festkommazahlen Bei Festkommazahlen wird die Position des Kommas nicht verändert. Damit hat eine Zahl die Darstellung:

$$x_1, x_2 \cdots x_n. \tag{2}$$

Die feste Position des Kommas hat bei Operationen mit „sehr"' großem oder „sehr" kleinem Ergebnis allerdings Überläufe zur Folge, das heißt, man erhält Ergebnisse, welche mit der gewählten Kommaposition nicht mehr oder nur sehr ungenau darstellbar sind. Dies macht eine erneute Durchführung der Rechnung mit veränderter Kommaposition notwendig, was wiederrum erhöhten Rechenaufwand für arithmetische Operationen zur Folge hat.

Gleitkommazahlen Im Gegensatz zu Festkommazahlen wird bei Gleitkommazahlen, basierend auf der Magnitude der darzustellenden Zahl, die Position des Kommas verschoben. Dies führt zu einem wesentlich größeren Wertebereich und damit auch wesentlich flexiblere Einsatzmöglichkeiten.

Eine Gleitkommazahl besteht aus 3 Teilen:

Basis b: Die Basis bestimmt bezüglich welcher Basis die Zahlen dargestellt werden. Zum menschlichen Verständnis ist $b = 10$ am besten geeignet. Der Computer verwendet die Basis 2.

Mantisse m: Die Mantisse $m = (m_1, \cdots, m_n)$ enthält die Ziffern der darzustellenden Zahl mit $m_i \in \{0, \cdots, b-1\}$ und $m_1 > 0$. Die Bedingung $m_1 > 0$ stellt eine sinnvolle Wahl des Exponenten e sicher, da dieser dann exakt der Größenordnung der Zahl entspricht.

Exponent e: Der Exponent speichert die Position des Kommas und damit die Größenordnung der Zahl.

Mit diesen Teilen stellt man eine Gleitkommazahl wie folgt dar:

$$\pm \underbrace{m_0, m_1 \cdots m_n}_{\text{Mantisse}} \times \underbrace{b^e}_{\text{Basis}} \in \mathbb{G}(b, n, k)$$

wobei wir mit $\mathbb{G}(b,n,k)$ die Menge aller Gleitkommazahlen meinen, welche mit gewählten Parametern b,n,k darstellbar sind (wenn b,n, k sich nicht ändern, dann werden wir oft $\mathbb{G}$ schreiben ohne die Abhängigkeit von den Parameters des Formats explizit aufzuführen). Es sei angemerkt, dass natürlich gilt, dass $\mathbb{G}$ eine diskrete Teilmenge $\mathbb{G} \subset \mathbb{R}$ der reellen Zahlen $\mathbb{R}$ ist.

Bei der Darstellung einer Gleitkommazahl im Computer wird immer die Basis $b = 2$ verwendet und muss deshalb nicht explizit gespeichert werden.[1] Die Darstellung einer Gleitkommazahl im Computer ist damit (hier eine IEEE 64-Bit Gleitkommazahl):

$$\underbrace{\pm}_{\text{1-Bit Vorzeichen}} \qquad \underbrace{e_0\ e_1 \cdots e_{10}}_{\text{11-Bit Exponent}} \qquad \underbrace{m_0\ m_1 \cdots m_{51}}_{\text{52-Bit Mantisse}}$$

Der Fehler, welcher bei der Abbildung einer beliebigen reellen Zahl in ein Gleitkommazahl-Format $\mathbb{G} = \mathbb{G}(b,n,k)$ entsteht, hängt offensichtlich (oder zumindest intuitiv) von den gewählten Parametern b,n,k ab. Wir nehmen im Folgenden an, dass eine beliebige reelle Zahl immer auf die, entsprechend den üblichen arithmetischen Regeln (was wohl-definiert ist, da $\mathbb{G} \subset \mathbb{R}$), nächste Zahl $\hat{x} \in \mathbb{G}$ abgebildet wird, d.h.

$$G \ : \ \mathbb{R} \mapsto \mathbb{G} \ : \ x \mapsto \mathrm{rd}_{\mathbb{G}}(x). \tag{3}$$

Wie bei der Rundung üblich tritt der maximale Fehler, welcher durch $G : \mathbb{R} \to \mathbb{R}$ entstehen kann genau dann auf, wenn man in der Mitte zwischen zwei benachbarten Gleitkommazahlen ist, siehe Abbildung 1. Um den maximalen Rundungsfehler zu verstehen, müssen wir also den Abstand zweier Gleitkommazahlen

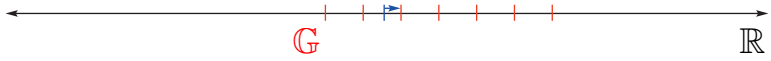


Abbildung 1: Beliebige reelle Zahlen (blau) werden auf die nächste Gleitkommazahl (rot) gerundet.

bestimmen. Dafür haben wir (wir nehmen hier an, dass kein Überlauf im letzten Bit auftritt; dass diese Annahme hier sinnvoll ist, wird in Kürze klar werden):

$$\begin{array}{rccccccc} & m_1 & , & m_2 & \cdots & m_n & \times & b^e \\ - & m_1 & , & m_2 & \cdots & (m_n+1) & \times & b^e \\ \hline & 0 & , & 0 & \cdots & 1 & \times & b^e \end{array} \tag{4}$$

wobei die Rechnung analog zur schriftlichen Subtraktion erfolgt. Um erneut eine gültige Gleitkommazahl zu erhalten, für welche gilt $m_1 > 0$, muss das Komma um $n-1$ Stellen verschoben werden. Damit erhalten wir für die Differenz

$$1 \quad , \quad 0 \quad \cdots \quad 0 \quad \times \quad b^{e-(n-1)}. \tag{5}$$

Um eine konkrete Vorstellung über den Abstand benachbarter Gleitkommazahlen zu erhalten, wählen wir $b = 10$ und $n = 1$, d.h. ein in der Praxis wenig geeignetes Gleikommazahl-Format mit einer Ziffer und Basis 10. Ein wenig überraschend hängt der Abstand auch vom Exponenten ab:[2]

e	Abstand
-2	0.01
-1	0.1
0	1.0
1	10.0
2	100.0

d.h. in $[1,10)$ haben unsere Gleitkommazahlen also einen Abstand von 1, in $[10,100)$ einen Abstand von 10 usw. Wir sehen also, dass es nicht *einen* Abstand zwischen benachbarten Gleitkommazahlen gibt, sondern dass dieser von der Magnitude der Zahlen abhängt. Dies ist in Abbildung 2 graphisch dargestellt. Daraus folgt, dass der maximale Rundungsfehler für das Runden $G : \mathbb{R} \to \mathbb{G}$ in die Gleitkommazahlen

[1] Diese Aussage bezieht sich auf die Darstellung, welche im Prozessor verwendet wird. In Software können beliebige Gleitkommaformate emuliert bzw. verwendet werden. In Python gibt es zum Beispiel das `Decimal` Modul, welches Gleitkommazahlen mit Basis darstellt, siehe `https://docs.python.org/2/library/decimal.html#module-decimal`.

[2] Es sei daran erinnert, dass, auf Grund der Bedingung $m_1 > 0$, der Exponent nicht frei wählbar ist, sondern eine inhärente Eigenschaft der Zahl ist

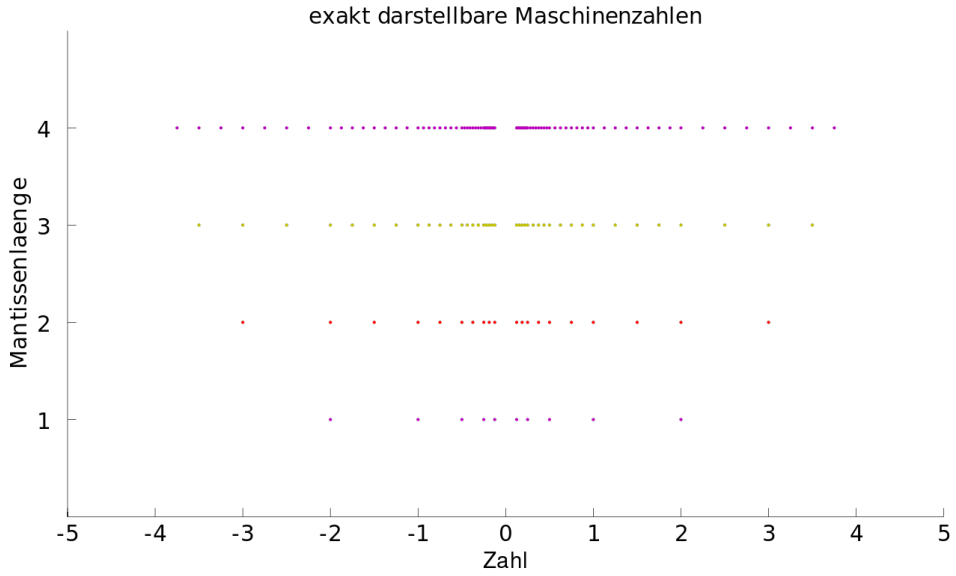


Abbildung 2: Abstände von Gleitkommazahlen mit Basis $b = 2$ (Quelle: `http://upload.wikimedia.org/wikipedia/de/0/0b/Gleitkommazahlen.svg`).

auch von der Größenordnung abhängt. Zum Beispiel haben wir für den maximalen Rundungsfehler:

$$G(2.5) = 3 \qquad |2.5 - G(2.5)| = 0.5$$
$$G(25) = 30 \qquad |25 - G(25)| = 5$$

Diese Abhängig des Abstandes von Gleitkommazahlen und des Rundungsfehlers bei der Abbildung in diese von der Magnitude der darzustellenden Zahl ist nicht zufällig, sondern stellt einen Kompromiss dar zwischen der Abdeckung eines großen Wertebereiches (für 64-bit Gleitkommazahlen hat man $\hat{x}_{min} \approx 2.2251 \times 10^{-308}$ und $\hat{x}_{max} = 1.7977 \times 10^{308}$) in $\mathbb{R}$ und einer fest und nicht von der Größe der Zahl abhängigen Anzahl von Bits zur Darstellung einer Zahl. Der Kompromiss basiert dabei auf der Beobachtung, dass in sehr vielen Anwendungen der tolerierbare Fehler von der Magnitude der Ergebnisse abhängt. Zum Beispiel, wenn die darzustellende Zahl $6.0 \in \mathbb{R}$ ist und man stellt diese als $10 \in \mathbb{G}$ dar, dann ist der Fehler für die allermeisten Anwendungen vollkommen inakzeptabel. Will man jedoch $1000006.0 \in \mathbb{R}$ darstellen und dies wird als $1000000 \in \mathbb{G}$ abgebildet, so ist dies in vielen Fällen verwendbar. Diese Abhängigkeit des Fehlers von der Größenordnung einer Zahl wird im relativen Fehler formalisiert, welchen wir im Folgenden betrachten.

Relativer vs. absoluter Fehler

Man unterscheidet bei Gleitkommazahlen zwischen *relativem Fehler* und *absolutem Fehler*. Der absolute Fehler ist der numerische Wert des tatsächlichen Fehlers, wohingegen der relative Fehler die Größe des Fehlers in Relation zum Ergebnis beschreibt. Somit misst der relative Fehler die Wichtigkeit des Fehlers im Bezug zum Ergebnis. Mit ihm kann die Frage beantwortet werden, ob eine Abweichung, egal ob sie sehr klein oder sehr groß ist, für das Ergebnis akzeptabel ist. Die Fehler sind wie folgt definiert:

$$\text{Absoluter Fehler: } E_a = |x_a - x|$$

$$\text{Relativer Fehler: } E_r = \frac{|x_a - x|}{|x_r|}$$

wobei x der korrekte (oder ursprüngliche) Wert ist und x_r die Approximation.

Um sowohl den absoluten als auch den relativen Fehler besser zu verstehen betrachten wir zwei Beispiele, in denen leicht falsche Lösung gegeben sind. Diese Abweichungen sollen als durch Rundung verursachte

Fehler verstanden werden.

$$\begin{aligned}
\text{Rechnung: } & 1+1? \\
\text{„Ergebnis": } & 3. \\
\text{Absoluter Fehler: } & E_a = |3-2| = 1 \\
\text{Relativer Fehler: } & E_r = \frac{|3-2|}{|2|} = 0.5 = 5 \cdot 10^{-1}
\end{aligned}$$

$$\begin{aligned}
\text{Rechnung: } & 10000+10000? \\
\text{„Ergebnis": } & 20001. \\
\text{Absoluter Fehler: } & E_a = |20001-20000| = 1 \\
\text{Relativer Fehler: } & E_r = \frac{|20001-20000|}{|20000|} = 0.00005 = 5 \cdot 10^{-5}
\end{aligned}$$

Wir sehen, dass der absolute Fehler konstant bleibt. Der relative Fehler ist in der zweiten Rechnung jedoch sehr viel kleiner als in der ersten, was der intuitiven Idee entspricht, dass die Wichtigkeit von Abweichungen von der Magnitude einer Zahl abhängen.

Genauigkeit von Gleitkommazahlen

Die maximale Genauigkeit einer Gleitkommazahldarstellung wird Maschinengenauigkeit ϵ genannt (obwohl sie heutzutage nicht mehr von der Maschine, d.h. der Hardware, abhängig ist, sondern lediglich von der für die Anzahl der zur Computerdarstellung verwendeten Bits). Zwei äquivalente Definition der Maschinengenauigkeit existieren:

a) Die Maschinengenauigkeit ist (zwei Mal) die kleinste Zahl δ, so dass $(1+\delta)$ nicht wieder auf 1 gerundet wird, d.h.

$$\epsilon = \frac{1}{2} \min_{\delta \in \mathbb{G}} G(1+\delta) > 1, \tag{6}$$

b) Der maximale *relative* Fehler, welche durch die Rundung einer reellen Zahl in das Gleitkommaformat ergibt,

$$\epsilon = \max_{x \in \mathbb{R}} \frac{|x - G(x)|}{|x|} \tag{7}$$

In Python erhält man die Maschinengenauigkeit mit:

```
Maschinengenauigkeit = np.finfo(beliebiger Datentyp).eps
#Zum Beispiel:
MaschinengenauigkeitFloat32 = np.finfo(np.float32).eps
```

Dabei sollte immer der Datentyp eines Objektes verwendet werden, so dass die Maschinengenauigkeit auch immer der erreichbaren Genauigkeit entspricht.

```
A = np.random.randn( 3, 3)
eps = np.finfo( A.dtype).eps
```

Neben der notwendigen Rundung, um eine beliebige reelle Zahl im Gleitkommaformat darstellen zu können treten bei der Rechnung mit Gleitkommazahlen weitere Fehler auf. Zum Beispiel gilt selbst für die elementaren arithmetischen Operationen, dass das Ergebnis im Allgemeinen keine Gleitkommazahl ist, d.h.

$$G(G(x) * G(y)) \neq G(x) * G(y) \neq G(x * y), \tag{8}$$

wobei "$*$" hier für eine beliebige elementare arithmetische Operation steht, siehe Abbildung 3. Für Addition, Multiplikation und Division liegt der Fehler in der Größenordnung Maschinengenauigkeit. Wie wir im Folgenden sehen, kann bei der Subtraktion die sogenannte Auslöschung auftreten und ein beliebig großer Fehler entstehen.

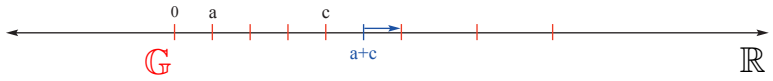


Abbildung 3: Notwendige Rundung bei der Addition, um für $a + c$ wieder eine gültige Gleitkommazahl zu erhalten.

Auslöschung

Beginnen wir mit einem Beispiel. Gegeben sein 3 Zahlen a,b,c die wie folgt dargestellt werden:

	$\mathbb{R}$	$\mathbb{G}$	rel. Fehler
a	1.22	1.22×10^0	0.0
b	3.34	3.34×10^0	0.0
c	2.28	2.28×10^0	0.0

In der 3-Ziffer Gleitkomma-Darstellung wollen wir $b^2 - 4\,a\,c$ berechnen, was aus der Lösung einer quadratischen Gleichung bestimmt werden muss. Wir nehmen an, dass zunächst b^2 und $4\,a\,c$ berechnet werden, und dann die Differenz gebildet wird. Für die Zwischenergebnisse erhalten wir:

	$\mathbb{R}$	$\mathbb{G}$	rel. Fehler
b^2	11.1556	1.12×10^1	0.00398
$4\,a\,c$	11.1264	1.11×10^1	0.00237

Für die Gleitkomma-Darstellung des obrigen Ergebnisses muss eine Rundung erfolgen, da nur drei Ziffern für die Mantisse zur Verfügung stehen. Der durch die Rundung entstehende relative Fehler ist jedoch klein. Um das Ergebnis zu erhalten, muss noch eine Subtraktion ausgeführt werden. Hierfür erhalten wir:

	$\mathbb{R}$	$\mathbb{G}$	rel. Fehler
$b^2 - 4\,a\,c$	0.0292	1.00×10^{-1}	2.42466

Trotz des sehr kleinen relativen Fehlers für b^2 und $4\,a\,c$ im Zwischenergebnis erhalten wir durch die Subtraktion ein Ergebnis, welches für die meisten Anwendungen unbrauchbar ist. Diese Potenzierung des relativen Fehlers bei der Subtraktion wird als "Auslöschung" (oder "catastrophic cancellation") bezeichnet.

Die Ursache für die Auslöschung liegt in den verschiedenen Größenordnungen von Minuend bzw. Subtrahend und Differenz. In der Gleitkomma-Darstellung ist dieser Unterschied erkennbar ist. Die Subtraktion löscht alle Ziffern am Anfang der Mantisse aus, welche für Minuend und Subtrahend gleich sind. Die Genauigkeit des Ergebnisses wird damit nur durch die Ziffern bestimmt, in welchen sich die Terme unterscheiden. Durch den Unterschied in der Größenordnung von Eingabe und Ausgabe der Subtraktion können sich jedoch nur wenige Ziffern am Ende unterscheiden (ansonsten wäre das Ergebnis nicht wesentlich kleiner als Minuend und Subtrahend). Im obigen Beispiel gehen uns durch die Subtraktion zwei Ziffern verloren, Minuend und Subtrahend stimmen in den ersten beiden überein, und anstatt drei Ziffern hat das Ergebnis effektiv nur noch eine, welche Informationen enthält (Die übrig gebliebenen Ziffern werden auch "significant digits" genannt).

Wir sehen also, dass Auslöschung immer dann auftritt, wenn zwei relativ zum Ergebnis große Zahlen subtrahiert werden, welche sich nur geringfügig unterscheiden, d.h. wenn die Differenz wesentlich kleiner ist, als die Zahlen selbst. Interessanter Weise tritt bei keiner anderen der verbleibenden grundlegenden arithmetischen Operationen, Addition, Multiplikation und Division, eine vergleichbare Potenzierung des relativen Fehlers auf.

Aufgabe 1: Maschinengenauigkeit

a) Schreiben Sie ein Python Programm, welches die Maschinengenauigkeit der von Python verwendeten Gleitkommazahlen ermittelt. Die Maschinengenauigkeit ist definiert als die kleinste positive Gleitkommazahl ε, so dass $1 + \varepsilon \neq 1$ ist.

b) Wiederholen Sie das Experiment unter Verwendung der `numpy` Datentypen `float32` und `float64`. Diese lassen sich mittels der Funktionen `numpy.float32(x)` und `numpy.float64(x)` erzeugen, z.B.:

```
import numpy as np
eps = np.float32(1)
```

Nutzen Sie die Funktion `type`, um zu überprüfen, dass alle Berechnungen mit dem jeweils gewünschten `numpy` Datentyp durchgeführt werden.

Lösung: a) $\varepsilon = 2^{-53}$

```
epsi = 1.0
i  = 0

print('  i  |       2^(-i)        |  1 + 2^(-i)  ')
print('  ----------------------------------------')

while 1.0 + epsi != 1.0:
    if i % 5 == 0:
        print('{0:4.0f} |  {1:16.8e}   | ungleich 1'.format(i, epsi))
    epsi = epsi / 2.0
    i = i + 1

print('{0:4.0f} |  {1:16.8e}   |  gleich 1\n'.format(i, epsi))
```

```
  i  |       2^(-i)        |  1 + 2^(-i)
  ----------------------------------------
   0 |    1.00000000e+00   | ungleich 1
   5 |    3.12500000e-02   | ungleich 1
  10 |    9.76562500e-04   | ungleich 1
  15 |    3.05175781e-05   | ungleich 1
  20 |    9.53674316e-07   | ungleich 1
  25 |    2.98023224e-08   | ungleich 1
  30 |    9.31322575e-10   | ungleich 1
  35 |    2.91038305e-11   | ungleich 1
  40 |    9.09494702e-13   | ungleich 1
  45 |    2.84217094e-14   | ungleich 1
  50 |    8.88178420e-16   | ungleich 1
  53 |    1.11022302e-16   |  gleich 1
```

b) $\varepsilon = 2^{-24} \approx 5.96 \cdot 10^{-8}$ und $\varepsilon = 2^{-53} \approx 1.11 \cdot 10^{-16}$

```
import numpy as np

epsi = np.float32(1.0)
i  = 0
print(type(epsi))

print('  i  |       2^(-i)        |  1 + 2^(-i)  ')
print('  ----------------------------------------')

while np.float32(1) + epsi != np.float32(1):
    if i % 5 == 0:
        print('{0:4.0f} |  {1:16.8e}   | ungleich 1'.format(i, epsi))
    epsi = epsi / np.float32(2)
    i    = i + 1

print('{0:4.0f} |  {1:16.8e}   |  gleich 1\n'.format(i, epsi))
```

```
<class 'numpy.float32'>
  i  |       2^(-i)        |  1 + 2^(-i)
  ----------------------------------------
   0 |    1.00000000e+00   | ungleich 1
   5 |    3.12500000e-02   | ungleich 1
  10 |    9.76562500e-04   | ungleich 1
  15 |    3.05175781e-05   | ungleich 1
  20 |    9.53674316e-07   | ungleich 1
  24 |    5.96046448e-08   |  gleich 1
```

Aufgabe 2: Kreisumfang nach Archimedes

Das wohl älteste bekannte Verfahren zur Bestimmung des Kreisumfanges geht auf Archimedes zurück (ca. 250 v.Chr) und approximiert den Kreisumfang mittels einbeschriebener regelmäßiger Polygone. Durch Verdopplung der Kanten wird die Approximation sukzessiv verbessert. In dieser Aufgabe wollen wir untersuchen, ob sich das Verfahren geeignet auf auf einem Computer implementieren lässt.

a) Wie lautet die Formel für den Umfang eines Kreises mit Radius r? Welchen Umfang hat der Einheitskreis?

b) Gegeben sei der Einheitskreis und ein einbeschriebenes Quadrat (siehe Abbildung 4(a)). Berechnen Sie die Seitenlänge und den Umfang des einbeschriebenen Quadrates.

c) Durch Verdopplung der Kanten erhält man aus dem Viereck (Abbildung 4(a)) ein Achteck (Abbildung 4(b)); dann ein Sechzehneck, usw. Leiten Sie eine Formel her, welche aus der Seitenlänge s_n des 2^n-Eck die Seitenlänge s_{n+1} des 2^{n+1}-Eck berechnet.

Hinweis: Überlegen Sie, wo sich in Abbildung 4(c) immer rechte Winkel befinden und nutzen Sie den Satz des Pythagoras.

d) Schreiben Sie ein Python Programm, welches unter Verwendung der hergeleiteten Formel den Kreisumfang möglichst genau approximiert. Geben Sie die jeweilige Approximation und den Fehler auf der Konsole für aufsteigende n aus. Erzeugen Sie außerdem unter Verwendung von `matplotlib` einen Graph des Fehlers in Abhängigkeit von n. Welche Genauigkeit lässt sich maximal erreichen?

e) Lässt sich Genauigkeit der Approximation noch verbessern?

Lösung: a) $U = 2\pi r$. Der Umfang des Einheitskreises ist also: 2π

b) Nach dem Satz von Pythagoras gilt:

$$s_2^2 = r^2 + r^2 = 1 + 1 = 2 \quad \Longrightarrow \quad s_2 = \sqrt{2} \tag{9}$$

Für den Umfang des Quadrates erhalten wir:

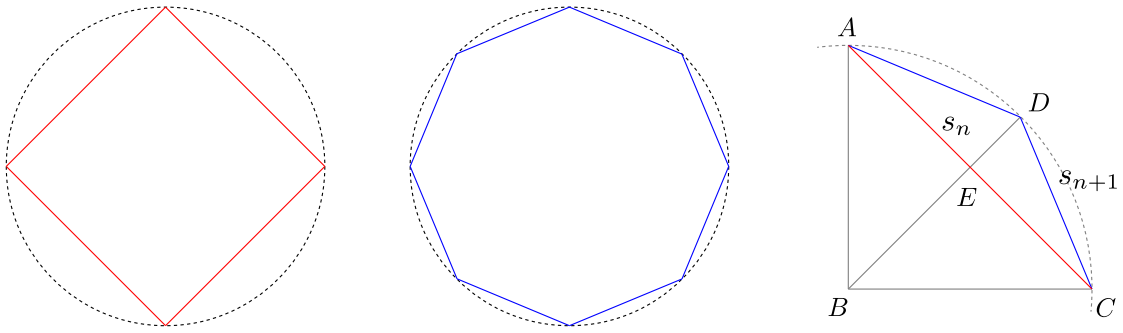
$$U = 2^2\sqrt{2} = 4\sqrt{2} \approx 5.656854 \tag{10}$$

(c) Da wir den Einheitskreis betrachten gilt: $\overline{BA} = \overline{BC} = \overline{BD} = r = 1$. Um s_{n+1} aus s_n zu erhalten, schauen wir uns Abbildung 4(c) an. Mit Hilfe des Satzes von Satz von Pythagoras folgt:

$$\left(\frac{s_n}{2}\right)^2 + \overline{BE}^2 = \overline{BA}^2 = 1 \tag{11}$$

Außerdem gilt:

$$\overline{ED} = \overline{BD} - \overline{BE} = 1 - \overline{BE} \tag{12}$$



(a) $n = 2$ (b) $n = 3$ (c) $n \to n + 1$

Abbildung 4: Approximation des Kreisumfanges mittels einbeschriebener 2^n-Ecke.

Erneute Anwendung des Satzes von Pythagoras ergibt dann:

$$s_{n+1}^2 = \overline{ED}^2 + \left(\frac{s_n}{2}\right)^2 = \left(1 - \overline{BE}\right)^2 + \left(\frac{s_n}{2}\right)^2 \tag{13a}$$

$$= \left(1 - \sqrt{1 - \left(\frac{s_n}{2}\right)^2}\right)^2 + \left(\frac{s_n}{2}\right)^2 \tag{13b}$$

$$= 1 - 2\sqrt{1 - \left(\frac{s_n}{2}\right)^2} + 1 - \left(\frac{s_n}{2}\right)^2 + \left(\frac{s_n}{2}\right)^2 \tag{13c}$$

$$= 2 - 2\sqrt{1 - \left(\frac{s_n}{2}\right)^2} = 2 - \sqrt{4 - s_n^2} \tag{13d}$$

Für s_{n+1} erhalten wir also:

$$s_{n+1} = \sqrt{2 - \sqrt{4 - s_n^2}} \tag{14}$$

d)

```
from math import sqrt, fabs, pi
import matplotlib.pyplot as plt

N = 30
err = []

sn = sqrt(2)
for n in range(2, N):
    pn = (2 ** n) * sn
    en = fabs(pn - 2.0 * pi)
    err.append(en)
    print('{0:2d}\t{1:1.20f}\t{2:1.20e}'.format(n, pn, en))
    sn = sqrt(2.0 - sqrt(4.0 - sn ** 2))

plt.figure(figsize=(6.0, 4.0))
plt.semilogy(range(2, N), err, 'rx')
plt.xlim(2, N - 1)
plt.ylim(1e-16, 10)
plt.show()
```

```
 2        5.65685424949238058190        6.26331057687205650097e-01
 3        6.12293491784143739665        1.60250389338148835350e-01
 4        6.24289030451610571504        4.02950026634805169579e-02
 5        6.27309698109188129678        1.00883260877049352189e-02
 6        6.28066231390947837809        2.52299327010785390257e-03
 7        6.28255450186551378522        6.30805314072446776663e-04
 8        6.28302760228829093592        1.57704891295296079079e-04
 9        6.28314588073576540950        3.94264438208224987648e-05
10        6.28317545055992177083        9.85661966446116366569e-06
11        6.28318284300927043518        2.46417031579682088704e-06
12        6.28318469122215361722        6.15957432614777644631e-07
13        6.28318515309000868996        1.54089577542038114188e-07
14        6.28318526692649648169        4.02530897503083906486e-08
15        6.28318530961517840439        2.43559217238953351625e-09
16        6.28318529064243058713        1.65371556448690171237e-08
17        6.28318521475143931809        9.24281469139032196836e-08
18        6.28318582187934548955        5.14699759257553068892e-07
19        6.28318825039038220126        2.94321079596926438171e-06
20        6.28319310740963921091        7.80023005297891813825e-06
21        6.28319310740963921091        7.80023005297891813825e-06
22        6.28334853004351501227        1.63222863928780270726e-04
23        6.28365936377840306193        4.74056598816829932730e-04
24        6.28490254498826761420        1.71723780868138220512e-03
25        6.28490254498826761420        1.71723780868138220512e-03
26        6.32455532033675904557        4.13700131571728135782e-02
27        6.32455532033675904557        4.13700131571728135782e-02
28        6.92820323027550877271        6.45017923095922540710e-01
29        8.00000000000000000000        1.71681469282041376800e+00
```

Für $n \leq 15$ verbessert sich die Approximation sukzessive. Für $n > 15$ vergrößert sich der Fehler wieder. Gemessen an der Maschinengenauigkeit von 2^{-53} ist dies aus numerischer Sicht ein unzureichendes Ergebnis.

Problematisch in Gleichung (14) ist die Subtraktion unter der ersten Wurzel. Für kleine s_n wird hier die Differenz von zwei fast gleich großen Zahlen gebildet. Um zu verstehen warum dies zu Problemen führt untersuchen wir im folgenden den relativen Fehler bei der Subtraktion. Es bezeichne $\mathrm{rd}\colon \mathbb{R} \to \mathbb{M}$ die Rundungsfunktion welche einer reellen Zahl eine Gleitkommazahl in der gewählten Darstellung zuordnet. Der relative Fehler, welcher bei der Rundung entsteht, lässt sich dann angeben als:

$$\bar{\varepsilon}_x = |\varepsilon_x| = \left|\frac{\mathrm{rd}(x) - x}{x}\right| \quad \text{bzw.} \quad \mathrm{rd}(x) = x(1+\varepsilon_x) \tag{15}$$

Für den relativen Fehler der Subtraktion zweier Gleitkommazahlen x und y ergibt sich dann:

$$\varepsilon_{x-y} = \frac{\mathrm{rd}\big(\mathrm{rd}(x) - \mathrm{rd}(y)\big) - (x-y)}{x-y} \tag{16a}$$

$$= \frac{\big(x(1+\varepsilon_x) - y(1+\varepsilon_y)\big)(1+\varepsilon_-) - (x-y)}{x-y} \tag{16b}$$

$$\approx \frac{x}{x-y}\varepsilon_x - \frac{y}{x-y}\varepsilon_y + \varepsilon_- \tag{16c}$$

Ist also $x \approx y$, dann sind $|\frac{x}{x-y}|, |\frac{y}{x-y}| > 1$ und es tritt eine Verstärkung des relativen Fehlers auf, welche als Auslöschung bezeichnet wird.

e) Um die Auslöschung zu vermeiden erweitern wir Gleichung (14) wie folgt:

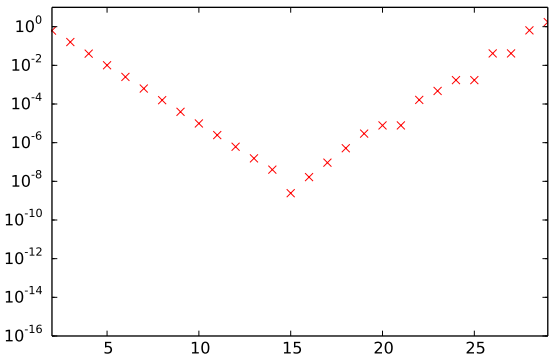
$$s_{n+1} = \sqrt{2 - \sqrt{4 - s_n^2}} \tag{17a}$$

$$= \sqrt{2 - \sqrt{4 - s_n^2}} \cdot \frac{\sqrt{2 + \sqrt{4 - s_n^2}}}{\sqrt{2 + \sqrt{4 - s_n^2}}} \tag{17b}$$

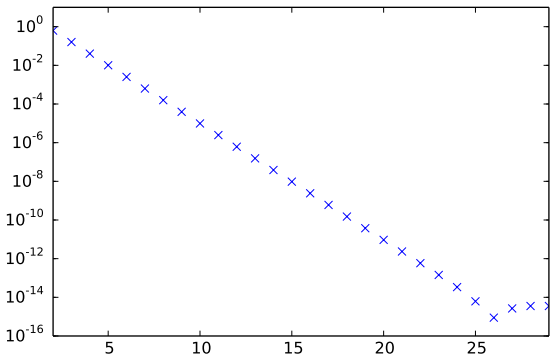
$$= \frac{\sqrt{2^2 - \sqrt{4 - s_n^2}^2}}{\sqrt{2 + \sqrt{4 - s_n^2}}} \tag{17c}$$

$$= \frac{|s_n|}{\sqrt{2 + \sqrt{4 - s_n^2}}} \tag{17d}$$

```python
from math import sqrt, fabs, pi
import matplotlib.pyplot as plt

N = 30
err = []

sn = sqrt(2)
for n in range(2, N):
    pn = 2 ** n * sn
    en = fabs(pn - 2.0 * pi)
    err.append(en)
    print('{0:2d}\t{1:1.20f}\t{2:1.20e}'.format(n, pn, en))
    sn = sn / sqrt(2 + sqrt(4 - sn ** 2))

plt.figure(figsize=(6.0, 4.0))
plt.semilogy(range(2, N), err, 'bx')
plt.xlim(2, N - 1)
plt.ylim(1e-16, 10)
plt.show()
```



(a)



(b)

2	5.65685424949238058190	6.26331057687205650097e-01
3	6.12293491784143650847	1.60250389338149723528e-01
4	6.24289030451610482686	4.02950026634814051363e-02
5	6.27309698109187863224	1.00883260877075997541e-02
6	6.28066231390950591162	2.52299327008032037156e-03
7	6.28255450186554575964	6.30805314040472353554e-04
8	6.28302760228860268654	1.57704890983545453764e-04
9	6.28314588073418356373	3.94264454026682642507e-05
10	6.28317545055432002954	9.85662526620245671438e-06
11	6.28318284302240037675	2.46415718585524246009e-06
12	6.28318469114023603339	6.16039350198605006881e-07
13	6.28318515316974579576	1.54009840436231115746e-07
14	6.28318526867712723316	3.85024589988347543112e-08
15	6.28318529755397214842	9.62561408357487380272e-09
16	6.28318530477318315519	2.40640307680450860062e-09
17	6.28318530657798657302	6.01599658978102524998e-10
18	6.28318530702918742747	1.50398804521501006093e-10
19	6.28318530714198786313	3.75983688627457013354e-11
20	6.28318530717018841614	9.39781585884702508338e-12
21	6.28318530717723877643	2.34745556326743098907e-12
22	6.28318530717900181060	5.84421400162682402879e-13
23	6.28318530717944234709	1.43884903991420287639e-13
24	6.28318530717955248122	3.37507799486047588289e-14
25	6.28318530717958001475	6.21724893790087662637e-15
26	6.28318530717958712017	8.88178419700125232339e-16
27	6.28318530717958889653	2.66453525910037569702e-15
28	6.28318530717958978471	3.55271367880050092936e-15
29	6.28318530717958978471	3.55271367880050092936e-15