3.1. **Rechnungen mit dem Taschenrechner.**

Aufgabe 3.1.1. Berechne die folgenden beiden Zahlen möglichst genau (als Kommazahl) mit deinem Taschenrechner, notiere die beiden Ergebnisse und entscheide, welche Zahl grösser ist:
(Du kannst den TI Nspire oder einen Handy-Taschenrechner oder ein Taschenrechner-Programm auf dem Computer oder die Python-Shell verwenden.)

$$\frac{1}{\sqrt{10^5+10^{-4}}-\sqrt{10^5}} \approx$$

$$\frac{\sqrt{10^5+10^{-4}}+\sqrt{10^5}}{10^{-4}} \approx$$

Aufgabe 3.1.2. Wie im folgenden Screenshot ersichtlich (jede Zeile per Return/Enter abschicken!): Definiere in der Python-Shell drei Variablen $x = 0.1$, $y = 0.2$ und $z = 0.3$ und frage den Computer, ob $x + y = z$ gilt!

```
Python 3.10.6 (n
Type "help", "co
ion.
>>> x = 0.1
>>> y = 0.2
>>> z = 0.3
>>> x + y == z
```

Danach lass Python $x + y$ und z ausgeben.

Kannst du die Antworten des Computers erklären?

Aufgabe 3.1.3. (a) Stimmen die folgenden Umformungen?

$$\begin{aligned}\frac{1}{\sqrt{a+b}-\sqrt{a}} &= \frac{1}{\sqrt{a+b}-\sqrt{a}}\cdot 1\\ &= \frac{1}{\sqrt{a+b}-\sqrt{a}}\cdot\frac{\sqrt{a+b}+\sqrt{a}}{\sqrt{a+b}+\sqrt{a}}\\ &= \frac{\sqrt{a+b}+\sqrt{a}}{\left(\sqrt{a+b}-\sqrt{a}\right)\cdot\left(\sqrt{a+b}+\sqrt{a}\right)}\\ &= \frac{\sqrt{a+b}+\sqrt{a}}{a+b-a}\\ &= \frac{\sqrt{a+b}+\sqrt{a}}{b}\end{aligned}$$

(b) Hoffentlich bist du nun (korrekterweise) davon überzeugt, dass

$$\frac{1}{\sqrt{a+b}-\sqrt{a}} = \frac{\sqrt{a+b}+\sqrt{a}}{b}$$

gilt. Setze nun $a = 10^5$ und $b = 10^{-4}$ ein und vergleiche dies mit deinen Rechen-Ergebnissen in Aufgabe 3.1.1.

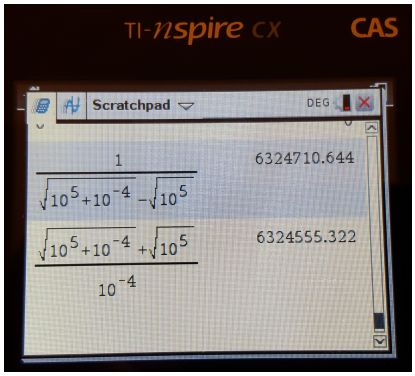
(c) Wieso liefert der Taschenrechner so unterschiedliche Ergebnisse?

. . . Erklärungen und Lösungen folgen auf der nächsten Seite . . .

3.2. **Lösung der Aufgaben 3.1.1 und 3.1.3.**

3.2.1. Die Umformungen in Aufgabe 3.1.3 sind alle korrekt und zeigen, dass die beiden Resultate in Aufgabe 3.1.1 gleich sein sollten. Jedoch zeigt der linke Screenshot, dass sich die beiden Ergebnisse des Taschenrechners TI Nspire um (die alles andere als kleine Zahl) 155.322 unterscheiden.

Der rechte Screenshot zeigt dieselbe Berechnung in der Python-Shell. Man sieht, dass Python (auf meinem Laptop) etwas genauer rechnet als der Taschenrechner, die Ergebnisse sich aber immer noch um 1.922... unteschieden.




```
Python 3.10.6 (main, Mar 10 2023, 10:5
Type "help", "copyright", "credits" or
>>> a = 10**5
>>> b = 10**(-4)
>>> ergebnis1 = 1/((a+b)**0.5-a**0.5)
>>> ergebnis2 = ((a+b)**0.5+a**0.5)/b
>>> ergebnis1
6324557.244121301
>>> ergebnis2
6324555.321917897
>>> ergebnis1-ergebnis2
1.9222034038975835
>>>
```

Der Taschenrechner bzw. Computer rechnet hier mit gerundeten Kommazahlen und nicht mit den exakten Werten[3]. Kleine Ungenauigkeiten können sich bei Rechnungen vergrössern (Fehlerfortpflanzung) und insgesamt zu sehr unterschiedlichen Ergebnissen führen (Stichwort Fehlerrechnung: Welchen Einfluss haben Ungenauigkeiten in den Ausgangsdaten auf die Ergebnisse der Rechnung).

3.2.2. Beachte, dass im Allgemeinen gilt:

- Die Summe/Differenz der Rundungen ist nicht die Rundung der Summe/Differenz.
- Das Produkt der Rundungen ist nicht die Rundung des Produkts.
- Der Quotient der Rundungen ist nicht die Rundung des Quotienten.
- Die Wurzel einer Rundung ist nicht die Rundung der Wurzel.

Slogan 3.2.3.

Beim Rechnen mit Kommazahlen am Computer können stets Fehler passieren!

Das gilt insbesondere auch für Rechnungen in Excel und anderen Standard-Programmen. Dasselbe gilt natürlich auch für den Taschenrechner.

3.3. **Lösung der Aufgabe 3.1.2.**

3.3.1. Der Screenshot der Python-Shell zeigt, dass in Python 0.1+0.2 nicht dasselbe ist wie 0.3.

```
Python 3.10.6 (main,
Type "help", "copyri
>>> x = 0.1
>>> y = 0.2
>>> z = 0.3
>>> x + y == z
False
>>> x + y
0.30000000000000004
>>> z
0.3
>>>
```

Das liegt daran, dass die drei beteiligten Zahlen **im Binärsystem gespeichert werden und dort unendlich viele Nachkommastellen haben**, der Computer aber nur mit endlich vielen Nachkommastellen rechnet (wird unten genauer erklärt)!

Solche Dinge können in **jedem Stellenwertsystem** passieren. Ein Beispiel im Dezimalsystem ist $\frac{1}{3}+\frac{1}{3}+\frac{1}{3}=1$. Wenn der Computer $\frac{1}{3}$ nur auf (beispielsweise) vier Nachkommastellen genau als 0.3333 speichert, so ist das Ergebnis 0.9999 und nicht 1.

Slogan 3.3.2.

Beim Vergleich von Kommazahlen machen Computer oft Fehler!

Insbesondere sollte man beim Vergleich mit Null aufpassen: Man sollte stets skeptisch sein, wenn ein Computer behauptet, dass eine (kleine) Zahl positiv, negativ oder Null ist.

[3]Beachte: Die auftauchenden Wurzeln haben als Kommazahlen unendlich viele Nachkommastellen und sind nicht-periodisch. Dies bedeutet im Wesentlichen (wenn man nicht symbolisch rechnet, was der TI Nspire auch kann: Enter statt Ctrl+Enter), dass der Computer runden muss, denn sein Speicher ist endlich.

Statt Kommazahlen am Computer direkt zu vergleichen ist es oft besser zu prüfen, ob ihre Differenz nahe bei Null liegt (= betragsmässig klein ist): Statt `x == y` verwende `-0.0000001 < x-y < 0.0000001`.

3.4. Speicherung von ganzen Zahlen.

- Zahlen werden binär gespeichert!
- Beispielsweise kann man mit einem Byte eine natürliche Zahl zwischen 0 und $255 = 2^8 - 1$ speichern.
- Mit zwei Byte kann man eine natürliche Zahl zwischen 0 und $65536 = 2^{16} - 1$ speichern.
- ganze Zahlen: Wenn man auch negative Zahlen speichern möchte, verwendet man in der Regel ein Bit, das angibt, ob die Zahl negativ ist (wenn dieses Bit 1 ist, ist die Zahl negativ, sonst positiv). So kann man mit einem Byte alle Zahlen von -128 bis 127 speichern.[4]
- Auf Grund des endlichen Speicherplatzes kann ein Computer nicht beliebige grosse ganze Zahlen genau speichern. Als extremes Beispiel denke man an eine Zahl mit so vielen Stellen, wie der Computerspeicher (oder das Universum) Atome hat.

3.5. Speicherung von Kommazahlen (und Fliesskommazahlen).

- Auch im Binärsystem gibt es Kommazahlen: Beispielsweise gilt

$$(10.101)_2 = 1 \cdot 2 + 0 \cdot 1 + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + 1 \cdot \frac{1}{8} = 1.625$$

- Achtung: Abbrechende Kommazahlen im Zehnersystem brechen im Binärsystem nicht notwendig ab (sind aber zumindest periodisch): Beispielsweise ist die Dezimalzahl $0.2 = \frac{1}{5}$ binär die Zahl $0.\overline{0011}$ (vgl. Aufgabe 3.1.2).
- „floating point numbers“ (= Fliesskommazahlen) werden zum Speichern von Kommazahlen (mit einer gewissen Genauigkeit) verwendet. Zum Beispiel speichert man mit 64 Bit (= 8 Byte) alle Zahlen, die sich (in einer Art wissenschaftlicher Schreibweise) wie folgt schreiben lassen:

$$\pm\, 1.(52 \text{ binäre Nachkommastellen}) \cdot 2^{11\text{-stellige positive oder negative Binärzahl}}$$

$$\text{Beispiel: } \pm\, 1.01001011101010110100110001010111000100101001010010 11 \cdot 2^{11000101101}$$

Der Exponent bei 2 gibt an, wo das Komma steht („where the comma floats to“, was den Namen *floating point number* erklärt).

Für Details siehe https://en.wikipedia.org/wiki/Double-precision_floating-point_format#IEEE_754_double-precision_binary_floating-point_format:_binary64

[4] Rechentechnisch am einfachsten ist die sogenannte „clock arithmetic“, vgl. https://en.wikipedia.org/wiki/Modular_arithmetic und https://de.wikipedia.org/wiki/Zweierkomplement.

4.0.1. Jeder Text ist in eine Folge von Nullen und Einsen (= eine Binärzahl) umzuwandeln, um ihn am Computer abspeichern zu können. Naheliegend ist die Idee, jedem Zeichen eine Zahl zuzuordnen und diese Zahl binär abzuspeichern.

4.1. **ASCII-Code.** Dabei hat sich der sogenannte ASCII-Code durchgesetzt (*American Standard Code for Information Interchange*):

- ⋮
- A hat ASCII-Code $(65)_{10} = (1000001)_2$ (als Dezimal- bzw. Binärzahl);
- B hat ASCII-Code $(66)_{10} = (1000010)_2$;
- C hat ASCII-Code $(67)_{10} = (1000011)_2$;
- ⋮
- Z hat ASCII-Code $(90)_{10} = (1011010)_2$;
- ⋮
- a hat ASCII-Code $(97)_{10} = (1100001)_2$;
- b hat ASCII-Code $(98)_{10} = (1100010)_2$;
- c hat ASCII-Code $(99)_{10} = (1100011)_2$;
- ⋮
- z hat ASCII-Code $(122)_{10} = (1111010)_2$;
- ⋮

Genauer besteht der ASCII-Code aus $2^7 = 128$ Zeichen. Es sinnvoll, ihn wie in der Tabelle rechts aufzuschreiben (denn dann stehen etwa „kleines“ und “grosses“ Alphabet nebeneinander).

Vom Zeichen zum ASCII-Code: Das Zeichen F findet sich in der Spalte 10 und der Zeile 00110; hintereinandergeschrieben ergibt sich der zugehörige ASCII-Code 10 00110 $= (1000110)_2 = (70)_{10}$.

Vom ASCII-Code zum Zeichen: Gegeben ist der ASCII-Code 43 (als Dezimalzahl). Wandle diese Dezimalzahl in eine **siebenstellige** (mit führenden Nullen) Binärzahl um: $(43)_{10} = (0101011)_2$. Ihre ersten beiden Stellen 01 liefern die Spalte und die hinteren fünf Stellen 01011 ihre Zeile in der Tabelle rechts: Das zugehörige Zeichen ist das Pluszeichen.

Steuerzeichen: Die Zeichen in der Zeile 00 sind sogenannte Steuerzeichen (control characters); beispielsweise steht LF für *line feed* (Zeilenvorschub; neue Zeile); CR steht für *carriage return* (Wagenrücklauf, etwa zum Steuern von Druckern). Auch DEL (ganz rechts unten) ist ein Steuerzeichen, es steht für *delete*.

Druckbare Zeichen: Die anderen 128 - 32 - 1 = 95 Zeichen sind druckbare Zeichen (Spc steht für *space* (Leerzeichen)).

ASCII ist 7-Bit-Zeichenkodierung: Der ASCII-Code ist eine 7-Bit-Zeichenkodierung, der $2^7 = 128$ Zeichen (davon 95 druckbar) codiert, die sogenannten ASCII-Zeichen. Da Byte (= 8 Bit) die übliche Speichereinheit ist, wird meist ein Byte zum Speichern eines ASCII-Zeichens verwendet.

Aufgabe 4.1.1.

(a) Welche Zeichenfolge codiert die folgende Folge von Bytes (jedes Byte steht per ASCII-Codierung für ein Zeichen; da der ASCII-Code nur 7 Bits verwendet, ist das erste Bit stets eine Null und du kannst es ignorieren).

01000001 01101100 01101100 01100101 01110011 00100000
01101001 01110011 01110100 00100000 01000010 01101001
01101110 01100001 01100101 01110010 01111010 01100001
01101000 01101100 00100001

(b) Welche Zeichenfolge codieren die folgenden Dezimalzahlen, wenn man sie mit ASCII entschlüsselt?

71 117 116 32 103 101 109 97 99 104 116 33

(c) (freiwillig) Wer mag, kann etwa in Visual Studio Code einen Hex-Editor als Extension installieren und sich die gerade entschlüsselten Texte damit anschauen.

00	01	10	11	
NUL	Spc	@	`	00000
SOH	!	A	a	00001
STX	"	B	b	00010
ETX	#	C	c	00011
EOT	$	D	d	00100
ENQ	%	E	e	00101
ACK	&	F	f	00110
BEL	'	G	g	00111
BS	(	H	h	01000
TAB	)	I	i	01001
LF	*	J	j	01010
VT	+	K	k	01011
FF	,	L	L	01100
CR	-	M	m	01101
SO	.	N	n	01110
SI	/	O	o	01111
DLE	0	P	p	10000
DC1	1	Q	q	10001
DC2	2	R	r	10010
DC3	3	S	s	10011
DC4	4	T	t	10100
NAK	5	U	u	10101
SYN	6	V	v	10110
ETB	7	W	w	10111
CAN	8	X	x	11000
EM	9	Y	y	11001
SUB	:	Z	z	11010
ESC	;	[	{	11011
FS	<	\	|	11100
GS	=	]	}	11101
RS	>	^	~	11110
US	?	_	DEL	11111

Abbildung 2. Four column ASCII table
Quelle: https://github.com/stevenlinx/Four-Column-ASCII

4.1.2. Vielleicht wurde bemerkt, dass im klassischen 7-Bit-ASCII nicht alle hier üblichen Zeichen vorkommen (etwa keine Umlaute ä, ö, ü, kein Euro-Zeichen). Diese wurde später in gewissen 8-Bit-Varianten von ASCII aufgenommen, vgl. Wikipedia:ISO/IEC 8859-15.

4.2. **Unicode.** Die 95 ASCII-Zeichen (eventuell ein wenig erweitert) reichen nicht aus, wenn man etwa in anderen Schriften (chinesisch, kyrillisch, arabisch, griechisch etc.) schreiben oder andere Zeichen (etwa Emojis) verwenden möchte.

Deswegen wurde eine neue Zeichencodierung kreiert, der sogenannte Unicode. Er besteht heutzutage (Unicode Version 15.0) aus fast 150'000 Zeichen.

Unicode ist wie ASCII eine Zuordnung zwischen gewissen Zeichen und Zahlen. Diese Zahlen schreibt man meist **hexadezimal** (also im Sechzehnersystem, Ziffern sind 0,1,2,3, ..., 9, A, B, C, D, E, F) mit `U+` als Präfix und nennt so etwas *Unicode code point*.

Zum Beispiel ist dem Eurozeichen € der Unicode code point U+20AC zugeordnet (Beispiel aus Wikipedia UTF-8; Examples):

Zeichen: € $\xleftarrow{\text{Unicode}}$ Unicode code point: U+20AC

Im folgenden Screenshot sind einige Unicode-Zeichen abgebildet, startend mit dem Unicode code point U+1F63C.

Bisher sind die $1'114'112 = 17 \cdot 16^4$ Codepunkte von U+0000 bis U+10FFFF „erlaubt", von denen aber wie oben erwähnt bisher nur ca. 150'000 belegt sind.

Aufgabe 4.2.1. (a) Für welche Zeichen stehen die folgenden Unicode code points?
Hinweis: Internet-Suche.

U+2764 U+0061 U+1f99c U+1fbf9 U+1f939 U+1f3bc U+4e23

(b) Unter Windows kann man teilweise Unicode-Zeichen direkt eingeben. Bei mir hat folgendes in Word und in Outlook funktioniert: Gib einen Unicode code point, etwa U+1f99c, ein und direkt danach Alt+c. Man kann auch U+ weglassen und nur 1f99c eingeben und danach Alt+c drücken.

Unter Linux klappt oft Ctrl+Shift+u gefolgt von dem Unicode code point.

(c) Achtung: Manche Unicode-Zeichen sehen sich sehr ähnlich!
Suche, wofür der code point U+0430 steht!
Ich vermute, dass Betrüger so etwas ausnutzen können, um Benutzer auf Fake-Webseiten zu locken. Dies ist ein Grund, weshalb man etwa beim Onlinebanking die Bankadresse per Tastatur eingeben sollte.

Aufgabe 4.2.2. Stöbere ein bisschen auf den folgenden Seiten, um einen Einblick zu bekommen, wie viele Unicode-Zeichen es gibt:

- https://www.vertex42.com/ExcelTips/unicode-symbols.html
- https://unicodelookup.com/
- https://unicode.org/emoji/charts/full-emoji-list.html
- https://home.unicode.org/

4.3. **Zusatzinfos.** Der Unicode code point wird im Computer meist platzsparend per UTF-8 (Unicode transformation format) codiert. Links:

- https://de.wikipedia.org/wiki/Unicode
- https://de.wikipedia.org/wiki/UTF-8

4.4. **Speicherplatzbedarf von Text.**

4.4.1 (Erinnerung: Bits and Bytes)**.**

- **Bit** = **b**inary digi**t** = Binärziffer, also 0 oder 1.
- **Byte** = Folge von 8 Bit = 8-stellige Binärzahl, z. B. 0100 1101 bzw. (im Kontext von Speichern, etwa Festplatten) die Möglichkeit, eine solche Zahl zu speichern.

Weil man an jeder der 8 Positionen zwei mögliche Ziffern hat, kann ein Byte $2^8 = 256$ verschiedene Werte annehmen, nämlich alle Binärzahlen von 00000000 bis 11111111.

4.4.2. Vermutlich ist bekannt, dass

- 1 Kilobyte, 1 kB, für $1'000 = 10^3$ Byte steht;
- 1 Megabyte, 1 MB, für $1'000'000 = 10^6$ Byte steht;
- 1 Gigabyte, 1 GB, für $1'000'000'000 = 10^9$ Byte steht;
- 1 Terabyte, 1 TB, für $1'000'000'000'000 = 10^{12}$ Byte steht.

Weitere Dezimal- und Binärpräfixe (Kibi-, Mebi-, Gibi-, etc.) findet man hier (beachte, dass $2^{10} = 1024$ nahe bei $1000 = 10^3$ liegt): https://de.wikipedia.org/wiki/Byte#Vergleichstabelle

Aufgabe 4.4.3. Auf eine Seite DIN-A4-Papier passen ca. 2000 Text-Zeichen (in normaler Grösse). Wenn man jedes Zeichen per ASCII durch ein Byte codiert, wieviel Speicherplatz benötigt man, um den Inhalt von 500 Seiten Text abzuspeichern?

Wie viele solche 500-seitigen Bücher (die nur aus Text bestehen) kann man auf einer handelsüblichen 400 Gigabyte-Festplatte abspeichern?

Lösung in dieser Fussnote[5]

4.4.4. Wenn eine Festplatte eine Speicherkapazität von 400 Gigabyte hat, kann man auf ihr $400 \cdot 10^9$ 8-stellige Binärzahlen abspeichern. Wie viel Information ist das?

Auf eine Seite DIN-A4-Papier passen ca. 2000 Zeichen, was 2000 Byte oder 2 Kilobyte entspricht, wenn man jedes Zeichen mit einem Byte (etwa per ASCII) abspeichert. Auf eine Festplatte von 400 Gigabyte passen also ca. $\frac{400 \cdot 10^9}{2 \cdot 10^3} = 2 \cdot 10^8$ Seiten Text. Drucken wir diesen Text doppelseitig aus und nehmen an, dass eine Seite 0,1 Millimeter dünn ist, so ergibt dies einen Stapel der Höhe $10^8 \cdot 0{,}1$ mm $= 10^4$ m $= 10$ km; der Papierstapel ist also ungefähr so hoch wie der Mount Everest. Zum Vergleich: Überschlagsmässig gerechnet ist der Stapel aller Bücher der British Library, der grössten Bibliothek der Welt, „nur“ 70 mal so hoch.

[5]Pro Seite benötigt man 2000 Byte = 2 Kilobyte. Für 500 Seiten benötigt man also $500 \cdot 2$ Kilobyte = 1000 Kilobyte = 1 Megabyte. Wegen 400 Gigabye = 400'000 Megabyte kann man 400'000 Bücher darauf abspeichern.