technische universität
dortmund

Statistical Methods of Data Analysis

Numerical Foundations

Prof. Dr. Dr. Wolfgang Rhode Dr. Maximilian Linhoff

2023

Overview

Motivation

Arithmetic Expressions

Data Representation on Computers

Rounding and Error Propagation

Stability and Condition

Overview

Motivation

Arithmetic Expressions

Data Representation on Computers

Rounding and Error Propagation

Stability and Condition

Current Research Areas in Physics

Many of the current research areas in physics need extensive computations to be answered:

- (Astro-)Particle physics
  - Very large amounts of data, both observed and simulated
  - Numerical fitting of models (e. g. using the Maximum Likelihood Method)
  - Machine Learning
- Particle Theory
  - Mainly numerical integration and symbolic calculations
- Solid-State Theory
  - Many-Particle-Simulations (→ Computational Physics)

Most questions that can be answered by pen and paper are already answered.

Limits of Computers

- Computations with infinite precision are not possible using finite time / space / energy
- Computations with limited precision often lead to counter-intuitive results or hard-to-find bugs.

⇒ Knowledge about how numbers and other data are represented in computers and the resulting consequences necessary

Overview

Motivation

Arithmetic Expressions

Data Representation on Computers

Rounding and Error Propagation

Stability and Condition

Some examples up-front

$\lvert \mathbf{v} \rvert = \sqrt{x^2 + y^2}$	Length of a 2d-vector
$E = E_0(1 + \varepsilon)^n$	Particle energy after stochastic accelartion
$\sigma^2 = \frac{1}{2} \sum_{i=1}^{N} (x_i - \bar{x})^2$	Variance
$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$	Quadratic formula
$\frac{1}{\sqrt{2\pi}} \int_{-x}^{x} e^{-t^2/2}\, dt$	Area under the standard normal distribution
$a = \arcsin(\sin\phi \sin\delta + \cos\phi \cos\delta \cos h)$	Altitude of a star

Definitions

Variables	$x_1, x_2, \dots, x_n \in \mathcal{R}$
Unary operators	$\mathcal{U} = \left\{+, -, !, \frac{\partial}{\partial x}, \dots\right\}$
Binary operators	$\mathcal{O} = \{+, -, \cdot, \div, \times, \dots\}$
Elementary functions	$\mathcal{F} = \left\{\sin(x), \cos(x), \tan(x), \exp(x), \ln(x), \sqrt{x}, \lvert x \rvert, \dots\right\}$

Definition

The set $\mathcal{A} = \mathcal{A}(x_1, x_2, \dots, x_n)$ of arithmetic expressions in $x_1, x_2, \dots, x_n$ is defined by:

i) $\mathcal{R} \subseteq \mathcal{A}$

ii) $x_l \in \mathcal{A}, \quad \text{for } l = 1, 2, \dots, n$

iii) $g \in \mathcal{A}, \quad \circ \in \mathcal{U} \Rightarrow (\circ g) \in \mathcal{A}$

iv) $g, h \in \mathcal{A}, \quad \circ \in \mathcal{O} \Rightarrow (g \circ h) \in \mathcal{A}$

v) $g \in \mathcal{A}, \quad \phi \in \mathcal{F} \Rightarrow \phi(g) \in \mathcal{A}$

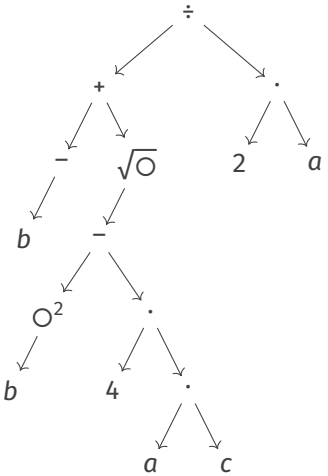
vi) $\mathcal{A}(x_1, x_2, \dots, x_n)$ is minimal over the sets $\mathcal{A}_i$, that fulfill i) – v)

Example: Quadratic Formula

Standard mathematical notation:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \in \mathcal{A}(a, b, c)$$

As a tree of operations:



Computation of Polynomials

- Given:

$$p(x) = a_0 x^n + a_1 x^{n-1} + \cdots + a_{n-1} x + a_n$$

- Naïve approach:
  1. Compute all $n - 1$ powers $x^k, k \in [2, \ldots, n]$
  2. Multiply results with the n coefficients a_i
  3. Compute the n additions
- Better alternative: Horner's Method

Horner's Method

- Recursive definition:

$$p(x) = p_n(x), \text{ with } \begin{cases} p_0 = a_0 \\ p_i(x) = p_{i-1} \cdot x + a_i \quad i = 1, 2, \dots, n \end{cases} \tag{1}$$

- Alternatively:

$$p(x) = (\cdots (a_0 x + a_1) \cdot x + \cdots) \cdot x + a_n \tag{2}$$

- Advantages:
  - Fewer operations
  - No powers, only multiplications and additions
  - Easy recursive definition of derivatives
- Disadvantages
  - No parallel evaluation possible

Horner's Method – First Order Derivative

$$p'(x) = p'_n(x), \text{ with } \begin{cases} p'_0 = 0 \\ p'_i(x) = p'_{i-1} \cdot x + p_{i-1} \quad i = 1, 2, \dots, n \end{cases} \tag{3}$$

Horner's Method – Example

$$p(x) = 4x^2 + 2x + 3 \tag{4}$$

Horner's Method:

$$p_0 = 4$$
$$p_1 = 4 \cdot x + 2$$
$$p_2 = (4 \cdot x + 2) \cdot x + 3$$

First-Order Derivative:

$$p'_0 = 0$$
$$p'_1 = (0 \cdot x) + 4$$
$$p'_2 = (4 \cdot x) + 4 \cdot x + 2$$
$$= 8 \cdot x + 2$$

Overview

Motivation

Arithmetic Expressions

Data Representation on Computers

Rounding and Error Propagation

Stability and Condition

- Computers can only store and exchange *bytes*
- 1 byte is the smallest addressable unit of storage
- On most modern computers, a byte comprises 8 *bits*, also known as *octet*
- An 8-Bit byte can take the values $\mathbf{0, 1, \ldots, 255}$
- Different data types assign specific *meaning* to 1 or more bytes
- In almost all cases, the data type has to be known to be able to interpret the bytes

Elementary Data Types

- Booleans (True / False)
- Integers in two general variants:
  - Signed ($\mathbb{Z}$)
  - Unsigned ($\mathbb{N}_0$)
- Floating Point Numbers

Booleans

- Elementary Data Type for binary logic
- In many programming languages equivalent or identical to integers
  - `True` = 1
  - `False` = 0
  - Mostly, also all other non-zero-integers are “true-ish”
- Logical operators: not / and / or / exclusive or (xor)
- Result of comparisons

Truth Tables for Binary Logical Operators

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

a	b	a or b
False	False	False
False	True	True
True	False	True
True	True	True

a	b	a xor b
False	False	False
False	True	True
True	False	True
True	True	False

Integers

- Different sizes, common today:
  8, 16, 32, 64 Bits
- Signed or unsigned
- Signed integers most often implemented using *two's complement*
- Many languages support larger sizes or even "arbitrary precision"
  - Python (`int`)
  - Java (`BigInteger`)
  - C++ (e. g. with `Boost.Multiprecision`)

4-Bit unsigned integer

Value	Bits	Value	Bits
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Integers

- Different sizes, common today:
  8, 16, 32, 64 Bits
- Signed or unsigned
- Signed integers most often implemented using *two's complement*
- Many languages support larger sizes or even "arbitrary precision"
  - Python (`int`)
  - Java (`BigInteger`)
  - C++ (e. g. with `Boost.Multiprecision`)

4-Bit signed integer (two's complement)

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

Integers

- Different sizes, common today:
  8, 16, 32, 64 Bits
- Signed or unsigned
- Signed integers most often implemented using *two's complement*
- Many languages support larger sizes or even "arbitrary precision"
  - Python (`int`)
  - Java (`BigInteger`)
  - C++ (e. g. with `Boost.Multiprecision`)

4-Bit signed integer (ones's complement)

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

Integer – Value Ranges

Unsigned $[0, 2^N - 1]$

Signed (two's complement) $[-2^{N-1}, 2^{N-1} - 1]$

Type	Minimum Value	Maximum Value
`int8`	−128	127
`uint16`	0	65 535
`int32`	−2 147 483 648	2 147 483 647
`int64`	−9 223 372 036 854 775 808	9 223 372 036 854 775 807

Consequences of limit value range: Over-/Underflow

Python with numpy, like most other languages, without warning:

```
>>> import numpy as np
>>> a = np.array([125, 126, 127], dtype=np.int8)
>>> a += 1
>>> a
array([126, 127, -128], dtype=int8)
```

Rust offers dedicate methods

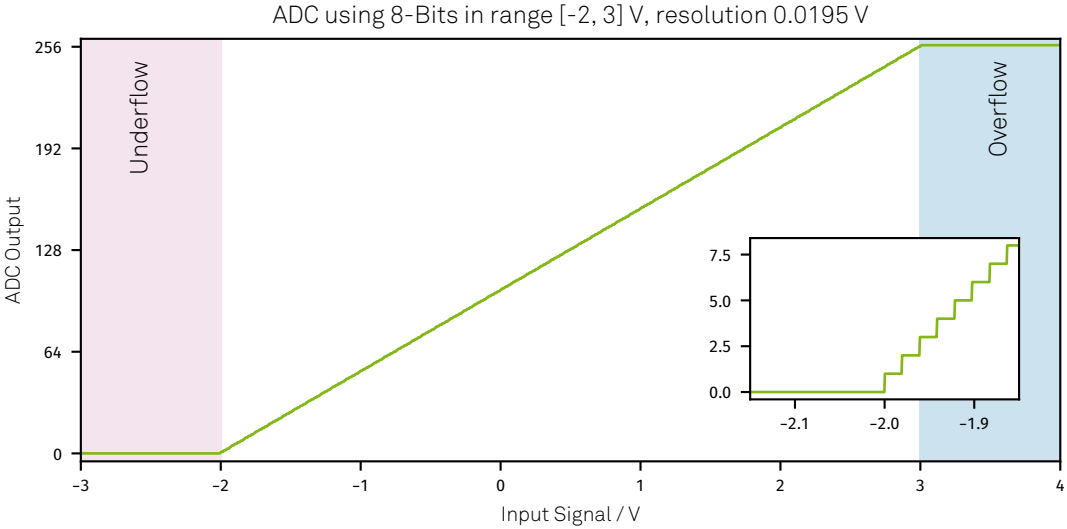
```
1 let x: i8 = 127;
2 let result = x.checked_add(1);
3 match result {
4     Some(val) => println!("Result {}", val),
5     None => println!("Overflow occured!"),
6 }
```

In C and C++, signed integer over-/underflow is Undefined Behavior

Y2038

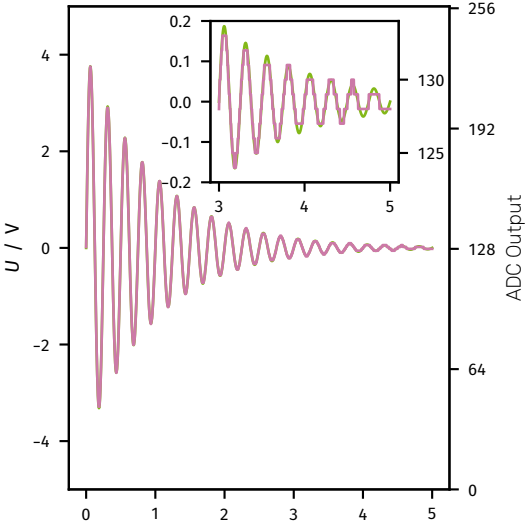
- Time is represented on many devices as “Unix-Time”:
  Integer seconds since `1970-01-01 00:00:00`
  - Many systems used a signed, 32-bit integer
  - `1970-01-01 00:00:00` + $(2^{31} - 1)$ s = `2038-01-19 03:14:07`
- Preparations for changing everything to 64-bit since many years
- Overflow with 64-bit happens in 292 billion years on a Sunday, December 4th
- Similar to the famous “millenium problem” (Y2K)
  - Many systems stored years only using two digits
  - Some panic before New Years 2000, but also massive effort ⇒ not much happened

Excursion: Analog-Digital-Converters



Excursion: Analog-Digital-Converters

- Transformation of analog signals (usually a voltage) into machine-readable data
- Dividing measurement range into 2^N_{bits} intervals
- Resolution for $N_{\text{bits}} = 8$: $1/2^8 = 1/256 \approx 0.39\,\%$



Floating Point Numbers

- Binary representation of a real number with limited precision
- IEEE 754 standard
- Represented as $x = (-1)^s \cdot m \cdot b^e$

Sign	1 Bit	s	
Mantissa	p Bits	$1 \leq m \leq 2$	encoded as $m = 1 + \frac{M}{2^p}$
Base		$b = 2$	
Exponent	r Bits	e	encoded as $e = E - B$

- This allows many different definitions, only two are defined in the standard and ubiquitous:

  32-Bit $r = 8, p = 23, B = 127$

  64-Bit $r = 11, p = 52, B = 1023$

Examples: 32-Bit

$$x = 1.0$$

- Bits: `0 0111 1111 0000 0000 0000 0000 0000 000`
- $s = 0$
- $E = 127 \Rightarrow e = 0$
- $M = 0 \Rightarrow m = 1$
- $x = (-1)^0 \cdot \left(1 + \frac{0}{2^{23}}\right) \cdot 2^{127-127} = 1 \cdot 1 \cdot 2^0 = 1.0$

Examples: 32-Bit

$$x = 2.0$$

- Bits: `0 1000 0000 0000 0000 0000 0000 0000 000`
- $s = 0$
- $E = 128 \Rightarrow e = 1$
- $M = 0 \Rightarrow m = 1$
- $x = (-1)^0 \cdot \left(1 + \frac{0}{2^{23}}\right) \cdot 2^{128-127} = 1 \cdot 1 \cdot 2^1 = 2.0$

Examples: 32-Bit

$$x = \pi$$

- Bits: `0 1000 0000 1001 0010 0001 1111 1011 011`
- $s = 0$
- $E = 128 \Rightarrow e = 1$
- $M = 4\,788\,187 \Rightarrow m = 1.570\,796\,370\,506\,286\,621\,093\,75$

$$\tilde{x} = (-1)^0 \cdot \left(1 + \frac{4\,788\,187}{2^{23}}\right) \cdot 2^{128-127}$$

$$= 3.141\,592\,741\,012\,573\,242\,187\,50$$

$$\pi = 3.141\,592\,653\,589\,793\,238\,462\,64\ldots$$

$$\Delta = 8.74 \times 10^{-8}$$

How to convert a number to binary float?

- Convert integer part to binary by dividing by two, keeping track of remainder, until reaching 0 $\Rightarrow$ remainders are binary representation
- Convert fractional part to binary by multiplying the decimal part by two, keeping track of the integer part until reaching 0, a repeating pattern, or the maximum number of digits $\Rightarrow$ integer part are binary representation
- Convert to scientific, base 2 notation with mantissa in $[1, 2)$ to get s, m, e
- Calculate M, E

Example: Convert x_{10} = 263.3 to IEEE 754

From `https://youtu.be/8afbTaA-gOQ`

Example: Convert x_{10} = 263.3 to IEEE 754

From `https://youtu.be/8afbTaA-gOQ`

$$
\begin{aligned}
263 \div 2 &= 131 + 1 \\
131 \div 2 &= 65 + 1 \\
65 \div 2 &= 32 + 1 \\
32 \div 2 &= 16 + 0 \\
16 \div 2 &= 8 + 0 \\
8 \div 2 &= 4 + 0 \\
4 \div 2 &= 2 + 0 \\
2 \div 2 &= 1 + 0 \\
1 \div 2 &= 0 + 1
\end{aligned}
$$

$\Rightarrow 263_{10} = 1\,0000\,0111_2$

Example: Convert $x_{10} = 263.3$ to IEEE 754

From `https://youtu.be/8afbTaA-gOQ`

$$
\begin{aligned}
263 \div 2 &= 131 + 1 \\
131 \div 2 &= 65 + 1 \\
65 \div 2 &= 32 + 1 \\
32 \div 2 &= 16 + 0 \\
16 \div 2 &= 8 + 0 \\
8 \div 2 &= 4 + 0 \\
4 \div 2 &= 2 + 0 \\
2 \div 2 &= 1 + 0 \\
1 \div 2 &= 0 + 1
\end{aligned}
$$

$$
\begin{aligned}
0.3 \cdot 2 &= 0.6 \Rightarrow 0 \\
\hline
0.6 \cdot 2 &= 1.2 \Rightarrow 1 \\
0.2 \cdot 2 &= 0.4 \Rightarrow 0 \\
0.4 \cdot 2 &= 0.8 \Rightarrow 0 \\
0.8 \cdot 2 &= 1.6 \Rightarrow 1 \\
\hline
0.6 \cdot 2 &= 1.2 \Rightarrow 1 \\
0.2 \cdot 2 &= 0.4 \Rightarrow 0
\end{aligned}
$$

$$\Rightarrow 0.3_{10} = 0.0\overline{1001}_2$$

$$\Rightarrow 263_{10} = 1\,0000\,0111_2$$

Example: Convert $x_{10} = 263.3$ to IEEE 754

From `https://youtu.be/8afbTaA-gOQ`

$$
\begin{aligned}
263 \div 2 &= 131 + 1 \\
131 \div 2 &= 65 + 1 \\
65 \div 2 &= 32 + 1 \\
32 \div 2 &= 16 + 0 \\
16 \div 2 &= 8 + 0 \\
8 \div 2 &= 4 + 0 \\
4 \div 2 &= 2 + 0 \\
2 \div 2 &= 1 + 0 \\
1 \div 2 &= 0 + 1
\end{aligned}
$$

$$
\begin{aligned}
0.3 \cdot 2 &= 0.6 \Rightarrow 0 \\
\hline
0.6 \cdot 2 &= 1.2 \Rightarrow 1 \\
0.2 \cdot 2 &= 0.4 \Rightarrow 0 \\
0.4 \cdot 2 &= 0.8 \Rightarrow 0 \\
0.8 \cdot 2 &= 1.6 \Rightarrow 1 \\
\hline
0.6 \cdot 2 &= 1.2 \Rightarrow 1 \\
0.2 \cdot 2 &= 0.4 \Rightarrow 0
\end{aligned}
$$

$$\Rightarrow 0.3_{10} = 0.0\overline{1001}_2$$

$$
\begin{aligned}
263.3_{10} &= 1\,0000\,0111.0\overline{1001} \\
&= 1.0000\,0111\,0\overline{1001} \times 2^8 \\
b = 8_{10} &\Rightarrow B = 135_{10} = 1\,000\,0111_2
\end{aligned}
$$

$$\Rightarrow 263_{10} = 1\,0000\,0111_2$$

Example: Convert $x_{10} = 263.3$ to IEEE 754

From `https://youtu.be/8afbTaA-gOQ`

$$
\begin{aligned}
263 \div 2 &= 131 + 1 \\
131 \div 2 &= 65 + 1 \\
65 \div 2 &= 32 + 1 \\
32 \div 2 &= 16 + 0 \\
16 \div 2 &= 8 + 0 \\
8 \div 2 &= 4 + 0 \\
4 \div 2 &= 2 + 0 \\
2 \div 2 &= 1 + 0 \\
1 \div 2 &= 0 + 1
\end{aligned}
$$

$$
\begin{aligned}
0.3 \cdot 2 &= 0.6 \Rightarrow 0 \\
\hline
0.6 \cdot 2 &= 1.2 \Rightarrow 1 \\
0.2 \cdot 2 &= 0.4 \Rightarrow 0 \\
0.4 \cdot 2 &= 0.8 \Rightarrow 0 \\
0.8 \cdot 2 &= 1.6 \Rightarrow 1 \\
\hline
0.6 \cdot 2 &= 1.2 \Rightarrow 1 \\
0.2 \cdot 2 &= 0.4 \Rightarrow 0
\end{aligned}
$$

$$\Rightarrow 0.3_{10} = 0.0\overline{1001}_2$$

$$
\begin{aligned}
263.3_{10} &= 1\,0000\,0111.0\overline{1001} \\
&= 1.0000\,0111\,0\overline{1001} \times 2^8 \\
b = 8_{10} &\Rightarrow B = 135_{10} = 1\,000\,0111_2
\end{aligned}
$$

$$\Rightarrow 263_{10} = 1\,0000\,0111_2$$

Result: `0 1000 0111 0000 0111 0100 1100 1100 110` = 263.299 987 79

Floating Point Numbers

- Many numbers are not exactly representable
- Precision: 7-8 significant digits (32-bit) or 15-16 significant digits (64-bit)
- Behavior often counter-intuitive, well-known example:

```
>>> (0.2 + 0.1) == 0.3
False
>>> repr(0.3), repr(0.1 + 0.2)
'0.3', '0.30000000000000004'
```

See `https://0.30000000000000004.com/`

⇒ In most circumstances, floats should not be compared using exact equality, instead:

```
>>> import numpy as np
>>> np.isclose(0.3, 0.1 + 0.2)
True
```

Floating Point Numbers – NaN, -Inf, Inf

- Not all possible byte patterns yield *normalized* numbers
- Invalid values

  NaN "Not a Number", for invalid, undefined or missing data: $0 \div 0, \sqrt{-1}$

  -Infinity, Infinity result of e. g. $1/0$, $-1/0$, 10^{3000}

- These values are stored by setting all exponent bits to 1 (32-Bit):

NaN	0	11111111	10000000000000000000000
Inf	0	11111111	00000000000000000000000
-Inf	1	11111111	00000000000000000000000

Floating Point Numbers – Rounding

- Not-representable numbers are rounded to the next representable number
- Example: 1.0 as 32-bit float, the two neighboring representable numbers are:

s	*E*	*M*	Decimal
0	0111 1011	1001 1001 1001 1001 1001 100	0.0999999940
0	0111 1011	1001 1001 1001 1001 1001 101	0.1000000015

Excursion: Fixed Point Numbers

- As an alternative to floating point numbers, there are libraries for real numbers with fixed precision
- In general, these are not suitable for scientific computing
  ⇒ learn to handle traps of floating point numbers
- Fixed point numbers are used e. g. in finance:
  - ⇒ calculations in € or $ to a precision of four decimal digits
  - ⇒ Exact representation of all numbers up-to four decimal digits
- Essentially, storing integers with a scale

```
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 4
>>> Decimal("0.1") + Decimal("0.2") == Decimal("0.3")
True
>>> Decimal(0.1) # careful with floats, rounding happens before
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

Excursion: Text

- Since everything on a computer is just bytes, so is text
- *Encodings* give bytes a meaning
- Historically, many, many different encodings
- Encodings have to be known and can only poorly be guessed

American Standard Code for Information Interchange (ASCII)

- 7-Bit ⇒ 128 codepoints
- 96 printable characters
- No accents, umlauts, ...

USASCII code chart

b_7 b_6 b_5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Row ↓ / Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	–	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

"Thinking that English only needs ASCII – even briefly – is naïve, said Zoë"

American Standard Code for Information Interchange (ASCII)

- 7-Bit ⇒ 128 codepoints
- 96 printable characters
- No accents, umlauts, ...

USASCII code chart

b_7 b_6 b_5 → Bits					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Column → Row ↓	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	–	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

"Thinking that English only needs ASCII – even briefly – is naïve, said Zoë"

Ok, but we have some space left in our byte

- Many encodings build on ASCII and define the upper 128 bits of a byte to have more characters
  - Most commonly Latin-1 or Windows cp1252
- Foreign languages completely redefine all 256 possible codepoints of one byte
- Many problems!
  - Some languages have more than 256 characters
  - What a about text with multiple languages?

Unicode

- Project to make every human-made text representable
- Extremely complex
- Current version: 14.0 from September 2021
- 144 697 characters from 159 writing systems, symbols, emoji, plus control codes
- Unicode defines code points with a semantic meaning, some examples:

Codepoint (Decimal)	Codepoint (Hex)	Character	Name
97	61	a	Latin Small Letter A
246	6F6	ö	Latin Small Letter O With Diaresis
8463	210F	ℏ	Planck Constant Over Two Pi
128169	1F4A9	💩	Pile of Poo
128567	1F637	😷	Face with Medical Mask

UTF-8

- Unicode is not an encoding. We still need a way to convert codepoints to and from bytes
- UTF-8 is an encoding for Unicode, using a variable number of bytes
- The first 7 Bit are compatible with ASCII ⇒ valid ASCII is the same in UTF-8!
- Currently reserved are 1 114 112 code points using up to 4 bytes
- Text files do not contain information about the encoding, it has to be known
- Using the wrong encoding to decode text results in “Mojibake”:

```
>>> 'Maximilian Nöthe'.encode('utf-8').decode('windows-1252')
'Maximilian NÃ¶the'
```

Overview

Motivation

Arithmetic Expressions

Data Representation on Computers

Rounding and Error Propagation

Stability and Condition

Rounding: Definitions

- A rounding is *correct*, iff[1] there is no other possible number between $x \in \mathbb{R}$ and the rounded number $\tilde{x}$.
- Optimal Rounding (as prescribed by IEEE 754):
  If two numbers have equal distance, the number with the lowest bit = 0 (even) is taken.
  Rounding towards even ensures that, statistically, rounding up happens with the same frequency as rounding down.
- Truncating: additional digits are removed
- Other possible algorithms:
  - Rounding towards ∞ (always round up)
  - Rounding towards $-\infty$ (always round down)
  - Rounding towards 0 (always reduce absolute value)

[1] iff = if and only if

Examples: Rounding to integers

Number	Optimal	Truncation	$\to 0$	$\to \infty$	$\to -\infty$
1.1	1	1	1	1	1
1.9	2	1	2	2	2
1.5	2	1	1	2	1
2.5	2	2	2	3	2
-1.9	-2	-1	-2	-2	-2
-1.5	-2	-1	-1	-1	-2
-2.5	-2	-2	-2	-2	-3
-3.5	-4	-3	-3	-3	-4

Rounding Errors

- Upper bound on the relative error:

$$\frac{|x - \tilde{x}|}{x} \leq \varepsilon = b^{1-p} \tag{5}$$

⇒ Rounding error is bounded by number ε, which depends on the data type

32-Bit $\varepsilon \approx 1.19 \times 10^{-7}$

64-Bit $\varepsilon \approx 2.22 \times 10^{-16}$

Python / numpy

```python
import numpy as np
np.finfo(np.float64).eps
```

C++

```cpp
#include <limits>
const double eps = std::numeric_limits<double>::epsilon();
```

Rounding errors for different Operations

- For the binary operators $\circ \in \{+, -, \cdot, \div\}$ and *correct* rounding:

$$\frac{|x \circ y - \overline{x \circ y}|}{|x \circ y|} \leq \varepsilon, \text{for } x \circ y \neq 0 \tag{6}$$

- For the computation of powers x^y:
  - If y is integer and small $\Rightarrow$ conversion to multiplication
  - Else: $x^y = \exp(y \cdot \ln x) \Rightarrow$ in general, relative error larger than above:

$$\frac{\Delta f}{f} = c \cdot \varepsilon, \text{with } c > 1$$

Approximations of Elementary Functions

- Most other elementary functions are evaluated using approximation methods
- Approach: $x \rightarrow$ Argument Reduction $\rightarrow$ Approximation $\rightarrow$ Inversion of Argument Reduction

Approximations of Elementary Functions

Argument Reduction Transformation to smaller argument range

Approximation using different approaches

- Continued fraction $a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + {}^{1}/_{a_n}}}}$
- Polynomial approximation
- Rational approximation (most common): $f(x) \approx x \cdot \frac{P(x)}{Q(x)}$
- Iterative approaches
- Power series

Jean-Michel Muller. “Elementary functions and approximate computing”. In: *Proceedings of the IEEE* 108.12 (2020), pp. 2136–2149. DOI: 10.1109/JPROC.2020.2991885

Example

- Square root: $f(x) = \sqrt{x}$ for $x = m \cdot b^e$
- Nowadays, implemented in hardware: ALU (Arithmetic Logical Unit)
- We will have a look at some methods for computing $\sqrt{x}$

1. Argument reduction

$$\sqrt{x} = \sqrt{x_0} \cdot b^S, \text{with} \begin{cases} x_0 = m, S = \frac{e}{2} & \text{for even } e \\ x_0 = \frac{m}{b}, S = \frac{e+1}{2} & \text{for odd } e \end{cases} \tag{7}$$

where $x_0 \in [1/b^2, 1]$

Example: Continued Fraction

- Find continued fraction with optimal coefficients for interval $[0.01, 1]$
- Ansatz:

$$w^*(x) = t_2 x + t_1 + \frac{t_0}{x + s0}$$

- Determine coefficients, so that

$$\sup_{x \in [0.01, 1.0]} \left| \sqrt{x} - w^*(x) \right| \stackrel{\text{def}}{=} \min$$

- One possible result (using $x \in$ `np.linspace(0, 1, 100000)`):

$$t_2 = 0.576749 \qquad t_1 = 0.475612$$
$$t_0 = -0.042936 \qquad s_0 = 0.111170$$

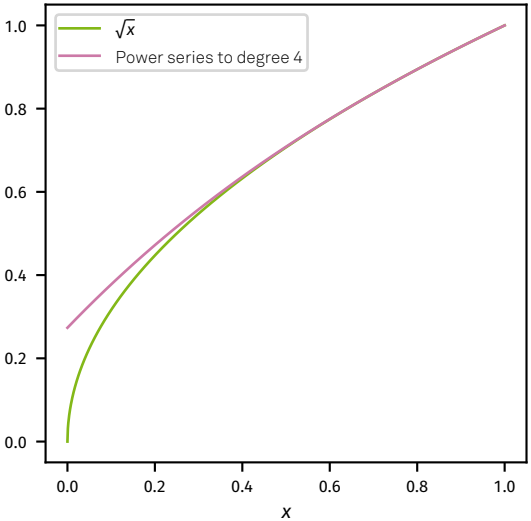
- Relative error: $\forall x_0 \in [0.01, 1] < 0.02$

Example: Power Series

- Powerseries, e. g. around $x = 1$ of:

$$\sqrt{1-z} = 1 - \frac{1}{2}z - \frac{1}{8}z^2 - \frac{1}{16}z^3 - \frac{5}{128}z^4 \ldots$$

- Has a radius of convergence of 1 $\Rightarrow$
- Decent convergion only around $x \approx 1$ ($z \approx 0$)
- Very slow convergence for small x
  $\Rightarrow$ not suitable for practical application



Example: Iterative Approach

- Bisection

$$w_0 > 0$$
$$\bar{w}_i = \frac{x}{w_i}$$
$$w_{i+1} = \frac{\bar{w}_i + w_i}{2}$$

- Abort when desired precision is reached: $|w_i| - \bar{w}_i < 10^{-n}$
- Example, $x = 0.01, n = 5$, converges with $\mathcal{O}(n^2)$:

i	1	2	3	4	5	6	7
w_i	0.505000	0.262401	0.150255	0.108404	0.100326	0.100001	0.100000

Rounding Errors for different Operations

- With the exception of roots and poles, the relative error is bounded:

$$\frac{f(x) - \tilde{f(x)}}{f(x)} = c_f \cdot \varepsilon, \text{ with } c > 1$$

- c_f depends on the chosen approximation

Example: Evaluation of Polynomials

- We investigate

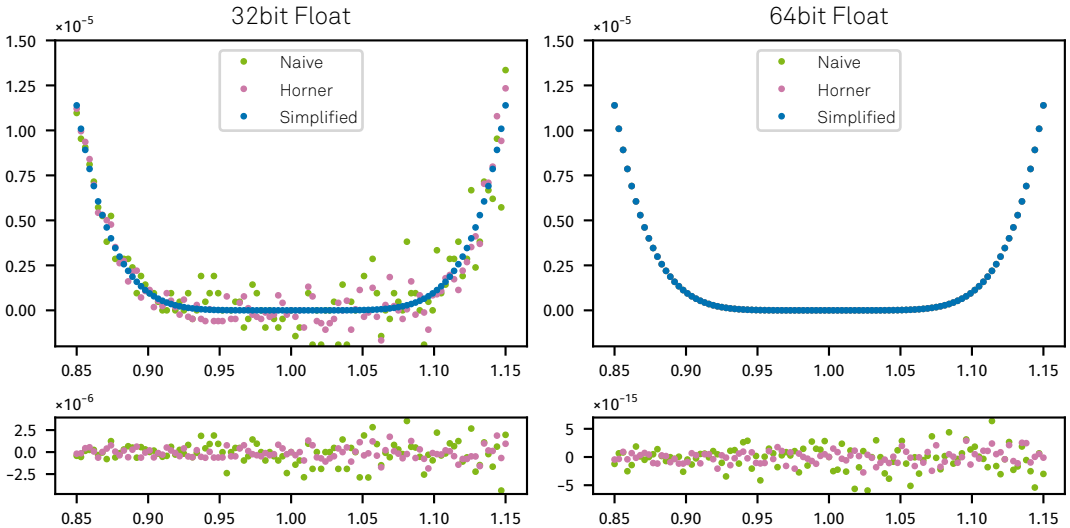
$$f(x) = (1 - x)^6 = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1 \tag{8}$$

- Evaluated naïvely: 5 powers, 5 multiplications, 6 additions
- Simplified: 1 power, 1 addition
- Horner's Method:

$$f(x) = \Big(\Big(\Big(\Big((x - 6) \cdot x + 15\Big) \cdot x - 20\Big) \cdot x + 15\Big) \cdot x - 6\Big) \cdot x + 1 \tag{9}$$

$\Rightarrow$ 6 additions, 5 multiplications

$f(x)$ with 32-Bit and 64-Bit precision

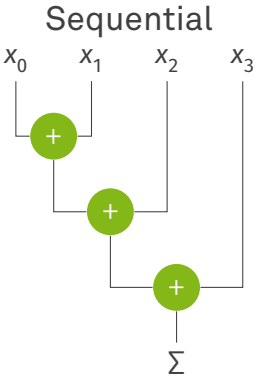


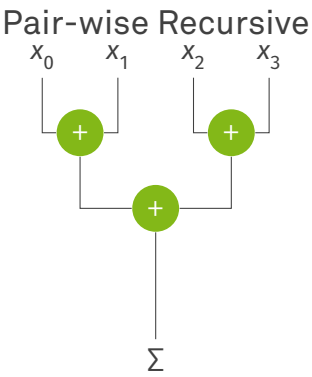
Performance with numpy

```
In [1]: import numpy as np
In [2]: x = np.linspace(0.85, 1.15, 1000)

In [3]: %timeit x**6 - 6 * x**5 + 15 * x**4 - 20 * x**3 + 15 * x**2 - 6 * x + 1
83.5 µs ± 1.38 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

In [4]: %timeit (((((x - 6) * x + 15) * x - 20) * x + 15) * x - 6) * x + 1
9.94 µs ± 39.2 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)

In [5]: %timeit (x - 1)**6
20 µs ± 168 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

In [6]: coefficients = np.array([1, -6, 15, -20, 15, -6, 1], dtype=np.float64)
In [7]: %timeit np.polyval(coefficients, x)
17.5 µs ± 50.3 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Example: Summation

- We can perform summation using different strategies
- This will affect the rounding errors





Example: Summation

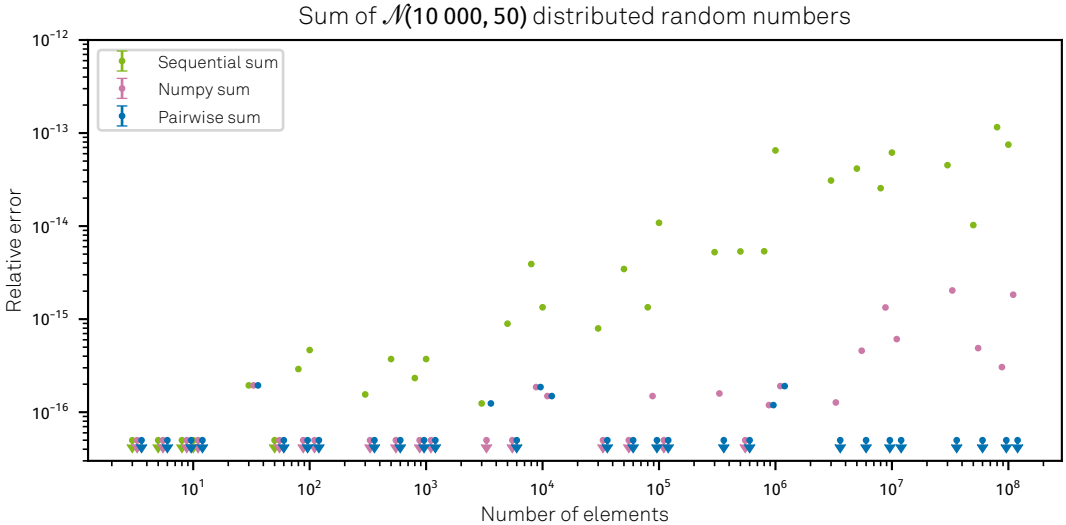
- Exact computation of the sum using IEEE 754 floats is possible, as

$$s = x + y$$
$$\Delta = y - (s - x), \text{ for } y > x$$

⇒ Calculate the sum while bookkeeping the errors

- `https://docs.python.org/3/library/math.html#math.fsum`

Example: Summation



Explanation

- Sum of two numbers of the same magnitude is less prone to cancellation
- Pair-wise recursive sum always sums numbers of the same magnitude if data has similar magnitudes
- This approach is very natural for GPUs or CPU vector registers (AVX)
- Bookkeeping of rounding errors is resource-intensive and mostly not needed
- Counter-example: `astropy.time.Time` for very precise timestamps uses two 64-Bit floats and keeps track of these rounding errors
  "so that the Time object maintains sub-nanosecond precision over times spanning the age of the universe."

⇒ Check computations (unit tests!) and adapt program structure if needed

Overview

Motivation

Arithmetic Expressions

Data Representation on Computers

Rounding and Error Propagation

Stability and Condition

Definitions

Stability Influence of rounding errors for *inexact* computation

Condition Propagation of initial uncertainties for *exact* computation

Motivation: Stability

- Look again at well-known function $f(x)$ with the two representations
  a) $f(x) = (1 - x)^6$
  b) $f(x) = 1 - 6x + 15x^2 - 20x^3 + 15x^4 - 6x^5 + x^6$

a) Two operations $\Rightarrow$ stable

b) Largish sequence of operations, including sums of numbers of different magnitude operations $\Rightarrow$ unstable

Numerical Stability Examples

$$f(x) = \left(x^3 + \frac{1}{3}\right) - \left(x^3 - \frac{1}{3}\right), \quad \Rightarrow f(x) = \frac{2}{3} \forall x$$

- Unstable for $x \to \infty$
- Round-off errors for differences of large numbers
- Using 64 bits:

x	$f(x)$
1	0.666 666 666 ...
10^3	0.666 666 746 ...
10^6	0.0

Numerical Stability Examples

$$f(x) = \frac{\left(3 + \frac{x^3}{3}\right) - \left(3 - \frac{x^3}{3}\right)}{x^3}, \quad \Rightarrow f(x) = \frac{2}{3} \forall x$$

- Unstable for $x \to 0$
- Round-off errors for differences of large numbers
- Using 64 bits:

x	$f(x)$
1	0.666 666 666 ...
10^{-3}	0.666 666 721 ...
10^{-6}	0.0

Numerical Stability Examples

$$f(x) = \frac{\sin^2(x)}{1 - \cos^2(x)} \quad \Rightarrow f(x) = 1 \forall x$$

- Unstable for $x \to 0$
- Division by small number from a difference of equal magnitude numbers

Numerical Stability Examples

$$f(x) = \frac{\sin^2(x)}{\sqrt{1 - \sin^2(x)}} \quad \Rightarrow f(x) = \tan(x)\,\forall x$$

- Unstable for $x \to 90°$

Numerical Stability Examples

Standard deviation σ with:

$$\sigma^2 = \langle x^2 \rangle - \langle x \rangle^2 = \frac{1}{n} \sum_{i=1}^{n} \left(x_i^2 \right) - \frac{1}{n^2} \left(\sum_{i=1}^{n} x_i \right)$$

For $x_i = x = \text{const.} \Rightarrow \sigma = 0$, but:

x	$\sigma, n = 10$	$\sigma, n = 20$
$^{100}/_{3}$	4.768×10^{-7}	NaN
$^{10000}/_{22}$	0.0	NaN

Conclusion: What to avoid?

- Sum of different-magnitude numbers
  - Extinction of smaller contributions
- Difference of equal-magnitude numbers
  - Truncation of leading digits
  - Increases relative errors
- Division by small numbers
  - Increases absolute error
- Multiplication of large numbers
  - Increases absolute error

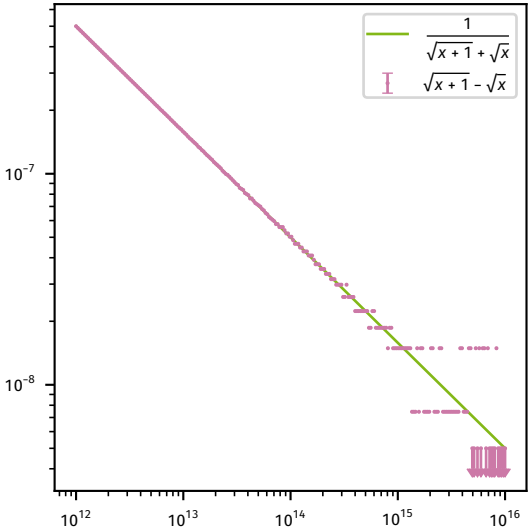
Improving Stability

- Transform differences by expanding fractions

$$\underbrace{\sqrt{x+1}-\sqrt{x}}_{\text{Unstable for } x \to \infty} = \frac{(x+1)-1}{\sqrt{x+1}+\sqrt{x}} = \underbrace{\frac{1}{\sqrt{x+1}+\sqrt{x}}}_{\text{Stable for } x \to \infty}$$

- Using trigonometric relations

$$\underbrace{1-\cos(x)}_{\text{Unstable for } x \to 0} = \underbrace{2\sin^2\left(\frac{x}{2}\right)}_{\text{Stable for } x \to 0}$$



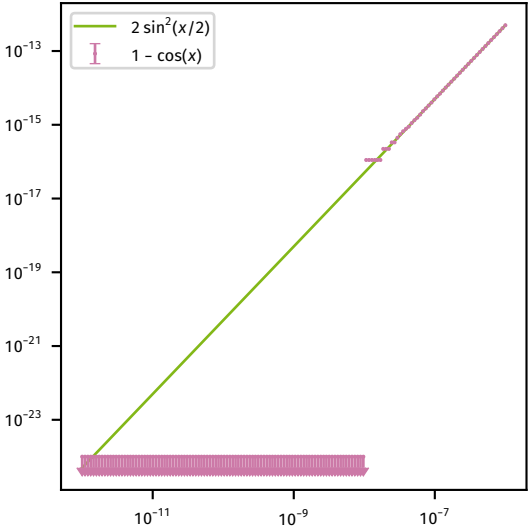
Improving Stability

- Transform differences by expanding fractions

$$\underbrace{\sqrt{x+1}-\sqrt{x}}_{\text{Unstable for } x \to \infty} = \frac{(x+1)-1}{\sqrt{x+1}+\sqrt{x}} = \underbrace{\frac{1}{\sqrt{x+1}+\sqrt{x}}}_{\text{Stable for } x \to \infty}$$

- Using trigonometric relations

$$\underbrace{1-\cos(x)}_{\text{Unstable for } x \to 0} = \underbrace{2\sin^2\left(\frac{x}{2}\right)}_{\text{Stable for } x \to 0}$$



Welford–Knuth-Algorithm

- Calculating variance as $\langle x^2 \rangle - \langle x \rangle^2$ not numerically feasible
- Naïve calculation of mean and variance as $\langle x \rangle$ and $\langle (x - \langle x \rangle)^2 \rangle$:
  - Requires going through all data twice
  - Is still numerically unstable for large numbers

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Welford–Knuth-Algorithm: Mean

$$
\begin{aligned}
\bar{x}_n - \bar{x}_{n-1} &= \frac{(n-1)\bar{x}_{n-1} + x_n}{n} - \bar{x}_{n-1} \\
&= \frac{x_n - \bar{x}_{n-1}}{n} = \frac{\delta_n}{n} \\
\Rightarrow \bar{x}_n &= \bar{x}_{n-1} + \frac{\delta_n}{n}, \quad \text{with } \delta_n = x_n - \bar{x}_{n-1}
\end{aligned}
$$

Welford–Knuth-Algorithm: Variance

$$
\begin{aligned}
t_n &= \sum_{i=1}^{n} (x_i - \bar{x}_n)^2 \\
\Rightarrow t_n - t_{n-1} &= \left(\sum_{i=1}^{n} (x_i - \bar{x}_n)^2 \right) - \left(\sum_{i=1}^{n-1} (x_i - \bar{x}_{n-1})^2 \right) \\
&= x_n^2 - n\bar{x}_n^2 + (n-1)\bar{x}_{n-1}^2 \\
&= (\delta_n + \bar{x}_{n-1})^2 - n\left(\bar{x}_{n-1} + \frac{\delta_n}{n}\right)^2 + (n-1)\bar{x}_{n-1}^2 \\
&= \delta_n \left(\delta_n + \frac{\delta_n}{n}\right) \\
&= \delta_n (x_n - \bar{x}_n)
\end{aligned}
$$

Welford–Knuth-Algorithm

$$\bar{x}_1 = x_1, \quad t_1 = 0$$
$$\delta_i = x_i - \bar{x}_{i-1}$$
$$\bar{x}_i = \bar{x}_{i-i} + \frac{\delta_i}{i}$$
$$t_i = t_{i-1} + \delta_i(x_i - \bar{x}_i)$$
$$\Rightarrow \sigma_i^2 = \frac{t_i}{i}$$

- Single iteration over samples to calculate both mean and variance
- Numerically stable
- Can be used *online* while gathering data

Solutions of quadratic equations

$$ax^2 + bx + c = 0$$

$$\Rightarrow x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad \text{Quadratic Formula}$$

Unstable for $b^2 \gg 4ac$ for the solutions where b and $\pm\sqrt{b^2 - 4ac}$ have the same sign

Solutions of quadratic equations

- We can transform to:

$$\Rightarrow x_{1,2} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}} \quad \text{Citardauq Formula}$$

- Which is is also unstable, but for the opposite case.
- Combining both stable cases yields:

$$q = -\frac{1}{2}\left(b - \operatorname{sgn}(b)\sqrt{b^2 - 4ac}\right)$$

$$\Rightarrow x_1 = \frac{q}{a}, \quad x_2 = \frac{c}{q}$$

Condition: Motivation

$$f(x) = \frac{1}{1-x}, \quad x = 0.999 \Rightarrow f(x) = 1000$$

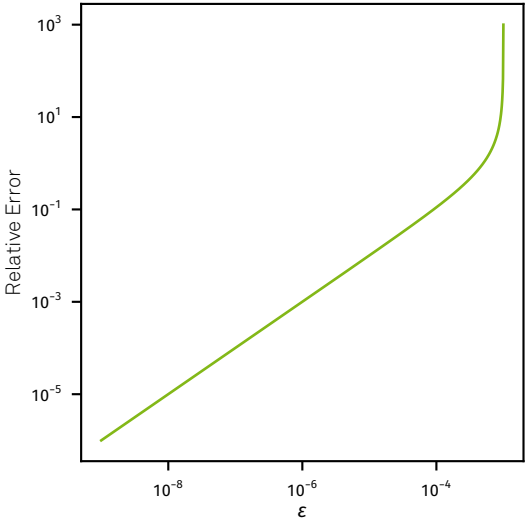
- Propagation of uncertainties: $\tilde{x} = 0.999 + \varepsilon$ with *small* ε:

$$f(\tilde{x}) = \frac{1000}{1-1000\varepsilon} = 1000\cdot(1+10^3\varepsilon+10^6\varepsilon^2+\cdots)$$

- Relative errors:

$$\frac{|x-\tilde{x}|}{x} = \frac{\varepsilon}{0.999}, \frac{|f(x)-f(\tilde{x})|}{f(x)} = 10^3\varepsilon+\mathcal{O}(\varepsilon^2)$$

$\Rightarrow$ Ill-conditioned, uncertainties are amplified



Condition Number

- Let's choose $\tilde{x}$ to be an approximation of x with small, relative error:

$$\varepsilon = \frac{\tilde{x} - x}{x} \Rightarrow \tilde{x} = x(1 + \varepsilon)$$

- Taylor series of $f(\tilde{x})$ around x:

$$f(\tilde{x}) = f(x + \varepsilon x) = f(x) + \varepsilon x f'(x) + \mathcal{O}(\varepsilon^2)$$
$$\Rightarrow f(\tilde{x}) - f(x) = \varepsilon x f'(x) + \mathcal{O}(\varepsilon^2)$$

- Relative error:

$$\left| \frac{f(x) - f(\tilde{x})}{f(x)} \right| = \left| x \frac{f'(x)}{f(x)} \right| \cdot |\varepsilon| + \mathcal{O}(\varepsilon^2) = K|\varepsilon| + \mathcal{O}(\varepsilon^2)$$

- Condition number:

$$K \stackrel{\text{def}}{=} \left| x \frac{f'(x)}{f(x)} \right|$$

Condition Number

- Discarding higher-order contributions yields:

$$\left|\frac{f(x) - f(\tilde{x})}{f(x)}\right| = K\left|\frac{x - \tilde{x}}{x}\right|$$

- This means:

$K < 1$	Error dampening
$K > 1$	Error amplification
$K \gg 1$	Problem is ill-conditioned

One-dimensional Condition Analysis

a) If at a position $f'(x^*) \neq 0, f(x) \to 0$ for $x \to x^* \neq 0$, then $K \to \infty$
$\Rightarrow$ f is ill-conditioned around simple roots where $x_0 \neq 0$

One-dimensional Condition Analysis

b) Let $f(x) = (x - x^*)^m \cdot g(x)$ with $g(x^*) \neq 0$ and $m \neq 0$
$\Rightarrow$ Root with multiplicity m at x^* if $m > 0$, pole with multiplicity $|m|$ if $m < 0$
With

$$f'(x) = m(x - x^*)^{m-1} \cdot g(x) + (x - x^*)^m \cdot g'(x)$$

$$\Rightarrow K = \left| x \frac{f'(x)}{f(x)} \right| = |x| \cdot \left| \frac{m}{x - x^*} + \frac{g'(x)}{g(x)} \right| = |m| \cdot \left| \frac{x}{x - x^*} \right| + \cdots$$

Thus, for $x \to x^*$:

$$K = \begin{cases} \infty, & \text{if } x^* \neq 0 \\ |m|, & \text{if } x^* = 0 \end{cases}$$

One-dimensional Condition Analysis

c) If $f'(x)$ has a pole at x^*, it is also ill-conditioned there
Example:

$$f(x) = 1 + \sqrt{x-1}$$

$$\Rightarrow K = \left| \frac{x}{2}\left(1 + \frac{1}{\sqrt{x-1}}\right)\right|$$

With $K \to \infty$ for $x \to 1$.

Relationship of Condition Number and Stability

- Correlation between stability and condition?
- Example:

$$f(x) = \sqrt{\frac{1}{x} - 1} - \sqrt{\frac{1}{x} + 1}, \quad \text{for } 0 < x < 1$$

- Stability:

  $x \to 0$ Extinction $\Rightarrow$ unstable

  $x \to 1$ Stable

- Condition Number:

$$f'(x) = \frac{-\frac{1}{x^2}}{2\sqrt{\frac{1}{x} - 1}} - \frac{-\frac{1}{x^2}}{2\sqrt{\frac{1}{x} + 1}} \Rightarrow K = \left| x \frac{f'(x)}{f(x)} \right| = \frac{1}{2\sqrt{1 - x^2}}$$

$x \to 0$ $K \to \frac{1}{2} \Rightarrow$ well-conditioned

$x \to 1$ $K \to \infty \Rightarrow$ ill-conditioned

Reminder: The Scientific Question

$$g(y) = \int A(x,y)\, f(x)\, \mathrm{d}x + b(y)$$

Inverse problems are usually ill-posed

$g(y)$ Distribution of observables

$b(y)$ Distribution of background

$A(x,y)$ Response function, describes the measurement process, transforms $x \to y$

$f(x)$ Desired quantity

Reminder: The Scientific Question

Are ill-conditioned problems exactly solvable?

No!

But: by making additional assumptions,
we can improve the condition of the problem,
so that an exact solution becomes possible.

But the solved problem is not the same...

What does that imply for the solution?