Numerical errors in computer simulations

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6BER03), 2024-2025

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Example 1

Start your spreadsheet program (Excel, ...)

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`
A3	`=(A2*10)-0.9`

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`
A3	`=(A2*10)-0.9`
A4	`=(A3*10)-0.9`

(repeat until A30)

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`
A3	`=(A2*10)-0.9`
A4	`=(A3*10)-0.9`

(repeat until A30)

What's happening?

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`
A3	`=(A2*10)-0.9`
A4	`=(A3*10)-0.9`

(repeat until A30)

What's happening?

Enter:

Cell	Value
A1	`2`

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`
A3	`=(A2*10)-0.9`
A4	`=(A3*10)-0.9`

(repeat until A30)

Enter:

Cell	Value
A1	`2`
A2	`=(A1*10)-18`
A3	`=(A2*10)-18`
A4	`=(A3*10)-18`

(repeat until A30)

What's happening?

Example 2

Start Python

Example 2

Start Python

Investigate the result of `np.sin(1e40 * np.pi)`

Example 2

Start Python

Investigate the result of `np.sin(1e40 * np.pi)`

Create a vector `v` containing the powers of 10, e.g., from 10^0 up to 10^{40}, and solve `np.sin(v * np.pi)`:

```python
import numpy as np
import matplotlib.pyplot as plt

v = np.logspace(0, 40, 41)
y = np.sin(v * np.pi)
plt.loglog(v, np.abs(y)) # Double log plot, values on y-axis must be positive.
plt.show()
```

Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

Errors in computer simulations

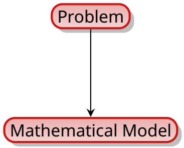
In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)

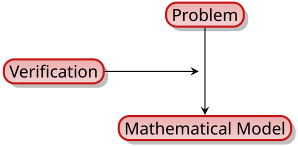
Verification and validation

Problem

Verification and validation



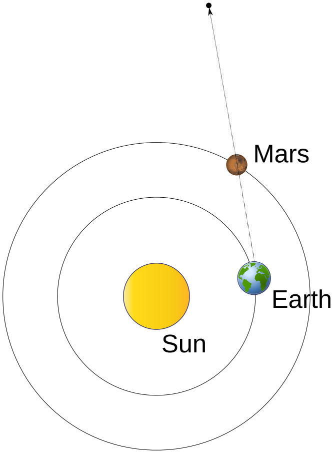
Verification and validation



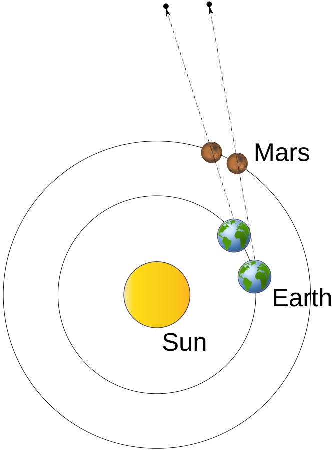
Verification

Verification is the process of mathematically and computationally assuring that the model computes the equations you intended to implement.

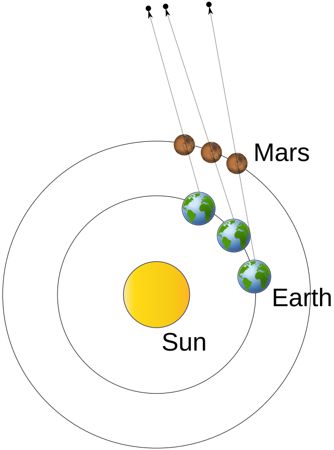
Verification of the physical model



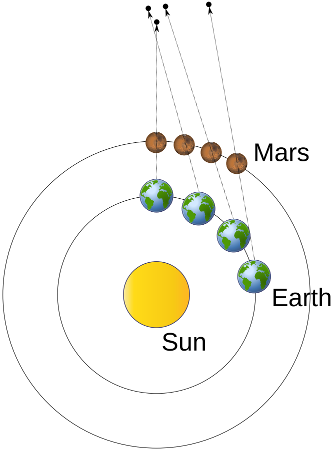
Verification of the physical model



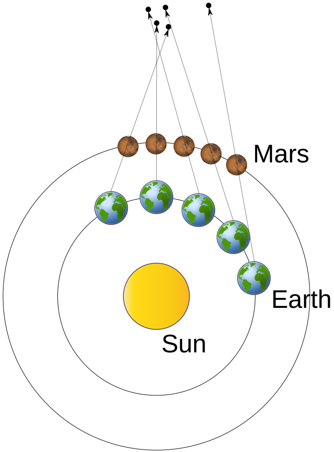
Verification of the physical model



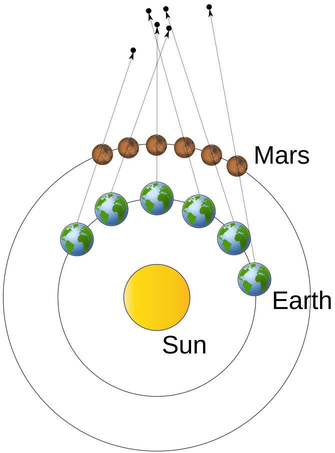
Verification of the physical model



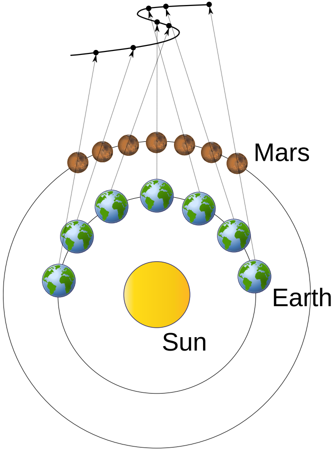
Verification of the physical model

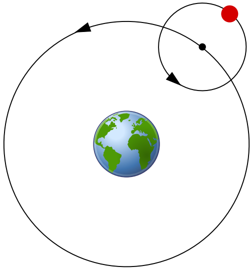


Verification of the physical model



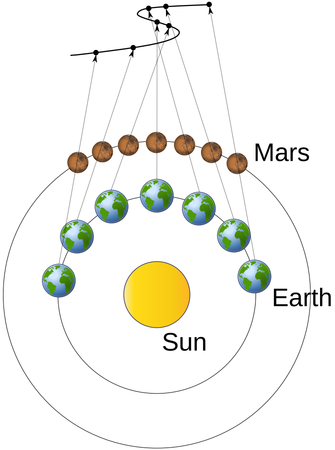
Verification of the physical model

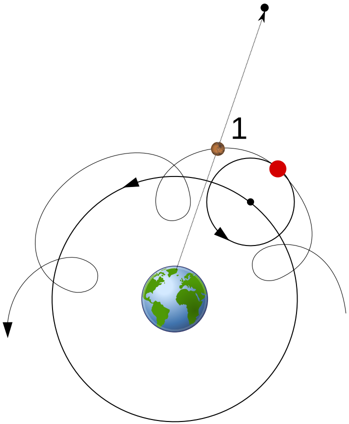




- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

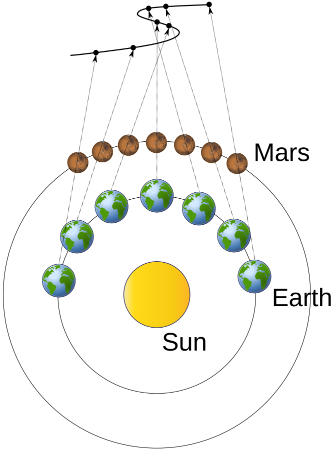
Verification of the physical model

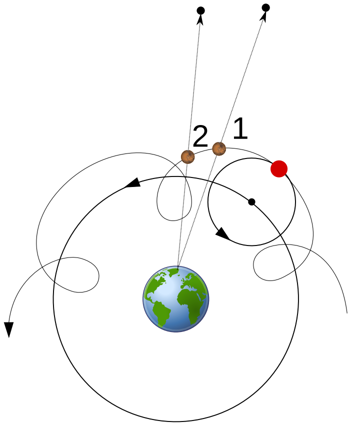




- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

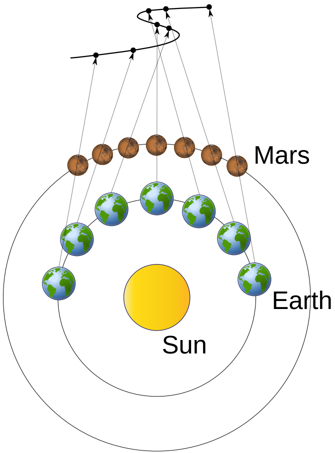
Verification of the physical model

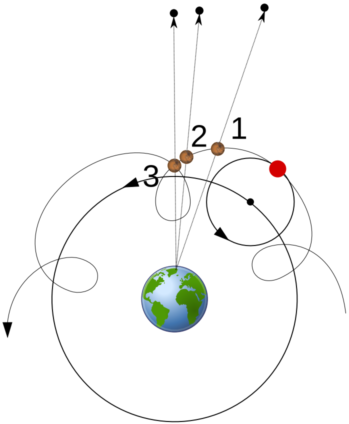




- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

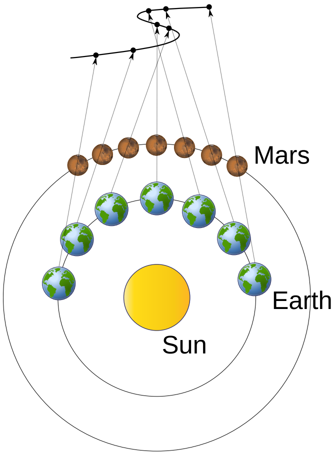
Verification of the physical model

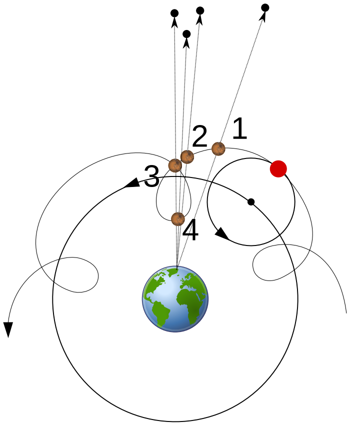




- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

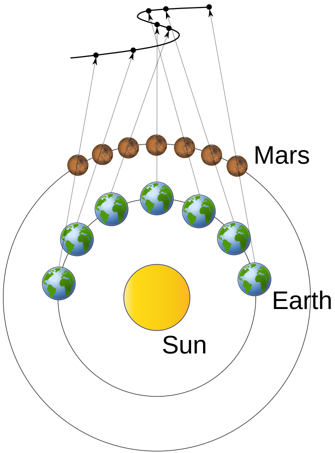
Verification of the physical model

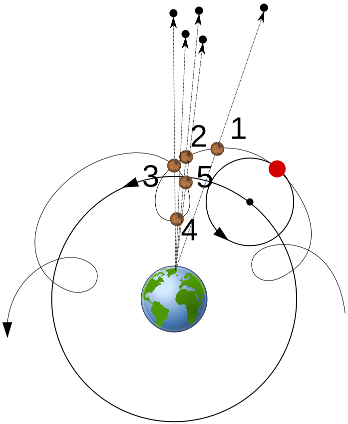




- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

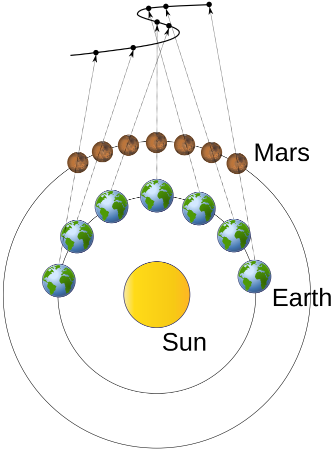
Verification of the physical model

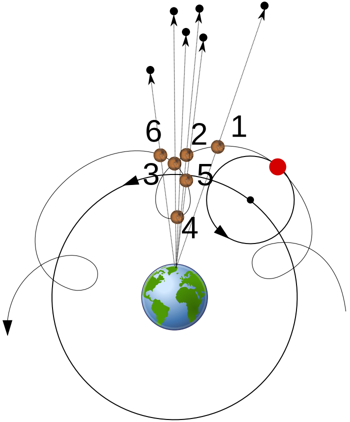




- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

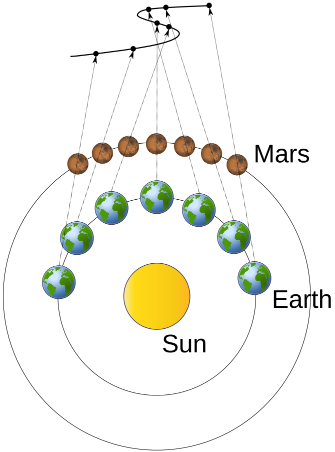
Verification of the physical model

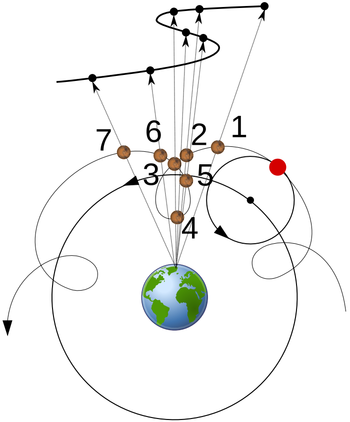




- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

Verification of the physical model





- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)
- Even though they were not 'right', Earth-centered models (Ptolemy) were still valid

Be aware of your uncertainties

Aleatory uncertainty

Uncertainty that arises due to inherent randomness of the system, features that are too complex to measure and take into account

Epistemic uncertainty

Uncertainty that arises due to lack of knowledge of the system, but could in principle be known

Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

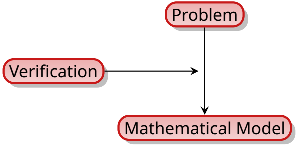
- Errors in the mathematical model (physics)

Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)

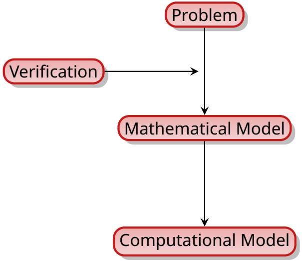
Verification and validation



Verification

Verification is the process of mathematically and computationally assuring that the model computes the equations you intended to implement.

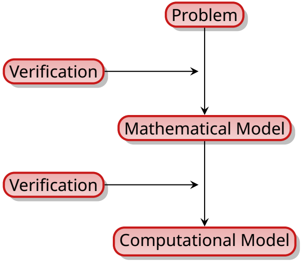
Verification and validation



Verification

Verification is the process of mathematically and computationally assuring that the model computes the equations you intended to implement.

Verification and validation



Verification

Verification is the process of mathematically and computationally assuring that the model computes the equations you intended to implement.

Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)

Errors in computer simulations

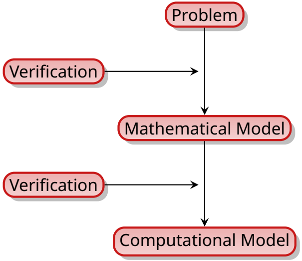
In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters

Verification and validation

Verification

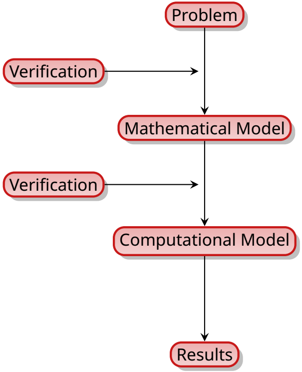
Verification is the process of mathematically and computationally assuring that the model computes the equations you intended to implement.



Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes the equations you intended to implement.



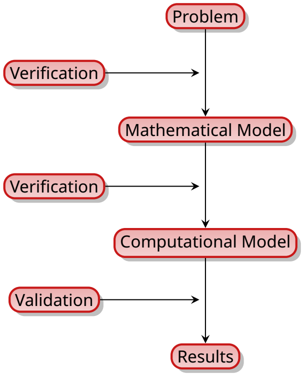
Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes the equations you intended to implement.

Validation

Validation is the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model



Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters

Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters
- Roundoff- and truncation errors

Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters
- Roundoff- and truncation errors
- Break errors

Errors in computer simulations

In this lecture I will outline different numerical errors that can appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters
- Roundoff- and truncation errors
- Break errors

Significant digits

A numerical result $\tilde{x}$ is an approximation of the real value x.

- Absolute error

$$\delta = |\tilde{x} - x|, x \neq 0$$

- Relative error

$$\frac{\delta}{\tilde{x}} = |\frac{\tilde{x} - x}{\tilde{x}}|$$

- Error margin

$$\tilde{x} - \delta \leq x \leq \tilde{x} + \delta$$

$$x = \tilde{x} \pm \delta$$

Significant digits

A numerical result $\tilde{x}$ is an approximation of the real value x.

- Absolute error

$$\delta = |\tilde{x} - x|, x \neq 0$$

- Relative error

$$\frac{\delta}{\tilde{x}} = |\frac{\tilde{x} - x}{\tilde{x}}|$$

- Error margin

$$\tilde{x} - \delta \leq x \leq \tilde{x} + \delta$$

$$x = \tilde{x} \pm \delta$$

Significant digits

A numerical result $\tilde{x}$ is an approximation of the real value x.

- Absolute error

$$\delta = |\tilde{x} - x|, x \neq 0$$

- Relative error

$$\frac{\delta}{\tilde{x}} = |\frac{\tilde{x} - x}{\tilde{x}}|$$

- Error margin

$$\tilde{x} - \delta \leq x \leq \tilde{x} + \delta$$

$$x = \tilde{x} \pm \delta$$

Significant digits

- $\tilde{x}$ has m significant digits if the absolute error in x is smaller or equal to 5 at the $(m+1)$-th position:

$$10^{q-1} \leq |\tilde{x}| \leq 10^{q}$$

$$|x - \tilde{x}| = 0.5 \times 10^{q-m}$$

- For example:

$$x = \frac{1}{3}, \tilde{x} = 0.333 \Rightarrow \delta = 0.00033333\ldots$$

3 significant digits

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Significant digits

- $\tilde{x}$ has m significant digits if the absolute error in x is smaller or equal to 5 at the $(m+1)$-th position:

$$10^{q-1} \leq |\tilde{x}| \leq 10^{q}$$

$$|x - \tilde{x}| = 0.5 \times 10^{q-m}$$

- For example:

$$x = \frac{1}{3}, \tilde{x} = 0.333 \Rightarrow \delta = 0.00033333\ldots$$

3 significant digits

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.
- In the decimal system, a digit c at position n has a value of $c \times 10^{n-1}$

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.
- In the decimal system, a digit c at position n has a value of $c \times 10^{n-1}$

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.
- In the decimal system, a digit c at position n has a value of $c \times 10^{n-1}$

4 5 2 1

Number of 1's: $1 \times 10^{(0)}$

Number of 10's: $2 \times 10^{(1)}$

Number of 100's: $5 \times 10^{(2)}$

Number of 1000's: $4 \times 10^{(3)}$

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.
- In the decimal system, a digit c at position n has a value of $c \times 10^{n-1}$

4 5 2 1

- 1 → Number of 1's: $1 \times 10^{(0)}$
- 2 → Number of 10's: $2 \times 10^{(1)}$
- 5 → Number of 100's: $5 \times 10^{(2)}$
- 4 → Number of 1000's: $4 \times 10^{(3)}$

$$(4521)_{10} = 4 \times 10^3 + 5 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$(4521)_{10} =$

$=($

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} +$$

$$=(1$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} +$

$=(10$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} +$$

$$= (100$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} +$$

$$= (1000$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \ldots$$

$$=(10001$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \ldots$$

$$\ldots 1 \times 2^{7} +$$

$$=(100011$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \dots$$

$$\dots 1 \times 2^{7} + 0 \times 2^{6} +$$

$$= (1000110$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1\times 2^{12} + 0\times 2^{11} + 0\times 2^{10} + 0\times 2^{9} + 1\times 2^{8} + \ldots$$
$$\ldots 1\times 2^{7} + 0\times 2^{6} + 1\times 2^{5} +$$

$$=(10001101$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \dots$$

$$\dots 1 \times 2^{7} + 0 \times 2^{6} + 1 \times 2^{5} + 0 \times 2^{4} + \dots$$

$$=(100011010$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$
\begin{aligned}
(4521)_{10} = & 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \dots \\
& \dots 1 \times 2^{7} + 0 \times 2^{6} + 1 \times 2^{5} + 0 \times 2^{4} + \dots \\
& \dots 1 \times 2^{3} + \\
& = (1000110101
\end{aligned}
$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$
\begin{aligned}
(4521)_{10} = {} & 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \ldots \\
& \ldots 1 \times 2^{7} + 0 \times 2^{6} + 1 \times 2^{5} + 0 \times 2^{4} + \ldots \\
& \ldots 1 \times 2^{3} + 0 \times 2^{2} + \\
& = (10001101010
\end{aligned}
$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$
\begin{aligned}
(4521)_{10} = {} & 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \ldots \\
& \ldots 1 \times 2^{7} + 0 \times 2^{6} + 1 \times 2^{5} + 0 \times 2^{4} + \ldots \\
& \ldots 1 \times 2^{3} + 0 \times 2^{2} + 0 \times 2^{1} + \\
& = (100011010100
\end{aligned}
$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$
\begin{aligned}
(4521)_{10} = &\, 1\times 2^{12} + 0\times 2^{11} + 0\times 2^{10} + 0\times 2^{9} + 1\times 2^{8} + \ldots \\
&\ldots 1\times 2^{7} + 0\times 2^{6} + 1\times 2^{5} + 0\times 2^{4} + \ldots \\
&\ldots 1\times 2^{3} + 0\times 2^{2} + 0\times 2^{1} + 1\times 2^{0} \\
&\quad =(1000110101001)_2
\end{aligned}
$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$
\begin{aligned}
(4521)_{10} = {} & 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \ldots \\
& \ldots 1 \times 2^{7} + 0 \times 2^{6} + 1 \times 2^{5} + 0 \times 2^{4} + \ldots \\
& \ldots 1 \times 2^{3} + 0 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0} \\
& = (1000110101001)_2
\end{aligned}
$$

- In general:

$$(c_m \ldots c_1 c_0)_q = c_0 q^0 + c_1 q^1 + \ldots + c_m q^m, c \in \{0, 1, 2, \ldots, q-1\}$$

Representation of numbers

- Numbers are stored in binary in the memory of a computer, in segments of a specific length (called a *word*).
- We distinguish multiple types of numbers:
  - Integers: $-301, -1, 0, 1, 96, 2293, \ldots$
  - Floating points: $-301.01, 0.01, 3.14159265, 14498.2$
- A binary integer representation looks like the following bit sequence:

$$z = \sigma\left(c_0 2^0 + c_1 2^1 + \ldots + c_{\lambda-1} 2^{\lambda-1}\right)$$

  σ is the sign of z (+ or -), and λ is the length of the word
- Endianness: the order of bits stored by a computer

Representation of numbers

- Numbers are stored in binary in the memory of a computer, in segments of a specific length (called a *word*).
- We distinguish multiple types of numbers:
  - Integers: $-301, -1, 0, 1, 96, 2293, \ldots$
  - Floating points: $-301.01, 0.01, 3.14159265, 14498.2$
- A binary integer representation looks like the following bit sequence:

$$z = \sigma\left(c_0 2^0 + c_1 2^1 + \ldots + c_{\lambda-1} 2^{\lambda-1}\right)$$

  σ is the sign of z (+ or -), and λ is the length of the word
- Endianness: the order of bits stored by a computer

Representation of numbers

- Numbers are stored in binary in the memory of a computer, in segments of a specific length (called a *word*).
- We distinguish multiple types of numbers:
  - Integers: $-301, -1, 0, 1, 96, 2293, \ldots$
  - Floating points: $-301.01, 0.01, 3.14159265, 14498.2$
- A binary integer representation looks like the following bit sequence:

$$z = \sigma\left(c_0 2^0 + c_1 2^1 + \ldots + c_{\lambda-1} 2^{\lambda-1}\right)$$

  σ is the sign of z (+ or -), and λ is the length of the word
- Endianness: the order of bits stored by a computer

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Representation of numbers

- Numbers are stored in binary in the memory of a computer, in segments of a specific length (called a *word*).
- We distinguish multiple types of numbers:
  - Integers: $-301, -1, 0, 1, 96, 2293, \ldots$
  - Floating points: $-301.01, 0.01, 3.14159265, 14498.2$
- A binary integer representation looks like the following bit sequence:

$$z = \sigma\left(c_0 2^0 + c_1 2^1 + \ldots + c_{\lambda-1} 2^{\lambda-1}\right)$$

  σ is the sign of z (+ or -), and λ is the length of the word
- Endianness: the order of bits stored by a computer

Excercise

- Convert the following decimal number to base-2: 214

Excercise

- Convert the following decimal number to base-2: 214

$$214_{10} = 11010110_2$$

Excercise

- Convert the following decimal number to base-2: 214

$$214_{10} = 11010110_2$$

- Excel:
  - Decimal: `=DEC2BIN(214)`
  - Octal: `=DEC2OCT(214)`
  - Hexadecimal: `=DEC2HEX(214)`

Excercise

- Convert the following decimal number to base-2: 214

$$214_{10} = 11010110_2$$

- Excel:
  - Decimal: `=DEC2BIN(214)`
  - Octal: `=DEC2OCT(214)`
  - Hexadecimal: `=DEC2HEX(214)`
- Python:
  - Decimal: `bin(214)`
  - Octal: `oct(214)`
  - Hexadecimal: `hex(214)`

Arithmetic operations with binary numbers

Addition:

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 0$ (carry one)

Arithmetic operations with binary numbers

Addition:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 0$ (carry one)

$$
\begin{array}{cccc}
 & 1 & 4 & 5 \\
+ & & 2 & 3 \\
\hline
 & 1 & 6 & 8
\end{array}
$$

Arithmetic operations with binary numbers

Addition:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 0$ (carry one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array}$$

$$\begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline \end{array}$$

Arithmetic operations with binary numbers

Addition:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 0$ (carry one)

$$\begin{array}{cccc} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array}$$

$$\begin{array}{ccccccccc} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Arithmetic operations with binary numbers

Addition:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 0$ (carry one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array}$$

$$\begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Subtraction:

$0 - 0 = 0$
$1 - 0 = 1$
$1 - 1 = 0$
$0 - 1 = 1$ (borrow one)

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Arithmetic operations with binary numbers

Addition:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 0$ (carry one)

$$\begin{array}{cccc} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array}$$

$$\begin{array}{ccccccccc} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Subtraction:

$0 - 0 = 0$
$1 - 0 = 1$
$1 - 1 = 0$
$0 - 1 = 1$ (borrow one)

$$\begin{array}{cccc} & 1 & 4 & 5 \\ - & & 2 & 3 \\ \hline & 1 & 2 & 2 \end{array}$$

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Arithmetic operations with binary numbers

Addition:

$0+0=0$
$0+1=1$
$1+0=1$
$1+1=0$ (carry one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array}$$

$$\begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Subtraction:

$0-0=0$
$1-0=1$
$1-1=0$
$0-1=1$ (borrow one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ - & & 2 & 3 \\ \hline & 1 & 2 & 2 \end{array}$$

$$\begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ - & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline \end{array}$$

Arithmetic operations with binary numbers

Addition:

$0+0=0$
$0+1=1$
$1+0=1$
$1+1=0$ (carry one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array}$$

$$\begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Subtraction:

$0-0=0$
$1-0=1$
$1-1=0$
$0-1=1$ (borrow one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ - & & 2 & 3 \\ \hline & 1 & 2 & 2 \end{array}$$

$$\begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ - & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{array}$$

- Multiplication and division are more expensive, and more elaborate

Exercise

Try the following commands in Python:

Command	Result
`np.iinfo(np.int32).min`	`-2147483648`

Exercise

Try the following commands in Python:

Command	Result
`np.iinfo(np.int32).min`	−2147483648
`np.iinfo(np.int32).max`	2147483647

Exercise

Try the following commands in Python:

Command	Result
`np.iinfo(np.int32).min`	`-2147483648`
`np.iinfo(np.int32).max`	`2147483647`
`i = np.int16(np.iinfo(np.int32).max)`	`i = 32767`

Exercise

Try the following commands in Python:

Command	Result
`np.iinfo(np.int32).min`	`-2147483648`
`np.iinfo(np.int32).max`	`2147483647`
`i = np.int16(np.iinfo(np.int32).max)`	`i = 32767`
`type(i)`	`<class 'numpy.int16'>`

Exercise

Try the following commands in Python:

Command	Result
`np.iinfo(np.int32).min`	`-2147483648`
`np.iinfo(np.int32).max`	`2147483647`
`i = np.int16(np.iinfo(np.int32).max)`	`i = 32767`
`type(i)`	`<class 'numpy.int16'>`
`i = i + 100`	`i = 32767`

Exercise

Try the following commands in Python:

Command	Result
`np.iinfo(np.int32).min`	`-2147483648`
`np.iinfo(np.int32).max`	`2147483647`
`i = np.int16(np.iinfo(np.int32).max)`	`i = 32767`
`type(i)`	`<class 'numpy.int16'>`
`i = i + 100`	`i = 32767`
`np.finfo(np.float64).max`	`1.7976931348623157e+308`

Exercise

Try the following commands in Python:

Command	Result
`np.iinfo(np.int32).min`	-2147483648
`np.iinfo(np.int32).max`	2147483647
`i = np.int16(np.iinfo(np.int32).max)`	i = 32767
`type(i)`	<class 'numpy.int16'>
`i = i + 100`	i = 32767
`np.finfo(np.float64).max`	1.7976931348623157e+308
`f = 0.1`	

Exercise

Try the following commands in Python:

Command	Result
`np.iinfo(np.int32).min`	`-2147483648`
`np.iinfo(np.int32).max`	`2147483647`
`i = np.int16(np.iinfo(np.int32).max)`	`i = 32767`
`type(i)`	`<class 'numpy.int16'>`
`i = i + 100`	`i = 32767`
`np.finfo(np.float64).max`	`1.7976931348623157e+308`
`f = 0.1`	
`type(f)`	`<class 'float'>`

Exercise

Try the following commands in Python:

Command	Result
np.iinfo(np.int32).min	−2147483648
np.iinfo(np.int32).max	2147483647
i = np.int16(np.iinfo(np.int32).max)	i = 32767
type(i)	<class ’numpy.int16’>
i = i + 100	i = 32767
np.finfo(np.float64).max	1.7976931348623157e+308
f = 0.1	
type(f)	<class ’float’>
print(":.16f".format(f))	0.1000000000000000

Exercise

Try the following commands in Python:

Command	Result
`np.iinfo(np.int32).min`	`-2147483648`
`np.iinfo(np.int32).max`	`2147483647`
`i = np.int16(np.iinfo(np.int32).max)`	`i = 32767`
`type(i)`	`<class 'numpy.int16'>`
`i = i + 100`	`i = 32767`
`np.finfo(np.float64).max`	`1.797693134862315 7e+308`
`f = 0.1`	
`type(f)`	`<class 'float'>`
`print(":.16f".format(f))`	`0.1000000000000000`
`print(":.20f".format(f))`	`0.10000000000000000555`

Representation of integer numbers

- In Python, integers are typically represented with at least 32 bits (though this can be larger depending on the Python implementation and system architecture). For a typical 64-bit Python installation, the integer type would have 64 bits.

Representation of integer numbers

- In Python, integers are typically represented with at least 32 bits (though this can be larger depending on the Python implementation and system architecture). For a typical 64-bit Python installation, the integer type would have 64 bits.
- The set of numbers that a 32-bit integer ‘int32‘ can represent in numpy is:

$$-2^{31} \leq z \leq 2^{31} - 1 \approx 2 \times 10^9$$

where z is a variable representing an `np.int32` type value.

Representation of integer numbers

- In Python, integers are typically represented with at least 32 bits (though this can be larger depending on the Python implementation and system architecture). For a typical 64-bit Python installation, the integer type would have 64 bits.
- The set of numbers that a 32-bit integer ‘int32‘ can represent in numpy is:

$$-2^{31} \leq z \leq 2^{31} - 1 \approx 2 \times 10^{9}$$

  where z is a variable representing an `np.int32` type value.
- If, during a calculation, an integer number in Python exceeds its maximum limit, Python automatically promotes it to a larger type if possible. In the case of numpy, an overflow error will be thrown if a number exceeds the maximum value representable by the ‘np.int32‘ type.

Representation of integer numbers

- In Python, integers are typically represented with at least 32 bits (though this can be larger depending on the Python implementation and system architecture). For a typical 64-bit Python installation, the integer type would have 64 bits.
- The set of numbers that a 32-bit integer 'int32' can represent in numpy is:

$$-2^{31} \leq z \leq 2^{31} - 1 \approx 2 \times 10^{9}$$

  where z is a variable representing an `np.int32` type value.
- If, during a calculation, an integer number in Python exceeds its maximum limit, Python automatically promotes it to a larger type if possible. In the case of numpy, an overflow error will be thrown if a number exceeds the maximum value representable by the 'np.int32' type.
- How can a computer identify an overflow? Usually through error reporting mechanisms within the language. Python's numpy library, for instance, will throw an OverflowError when an operation results in a number larger than what can be represented by the current data type.

Representation of real (floating point) numbers

- Formally, a real number is represented by the following bit sequence

$$x = \sigma\left(2^{-1} + c_2 2^{-2} + \ldots + c_m 2^{-m}\right) 2^{e-1023}$$

  Here, σ is the sign of x and e is an integer value.

- A floating point number hence contains sections that contain the sign, the exponent and the mantissa

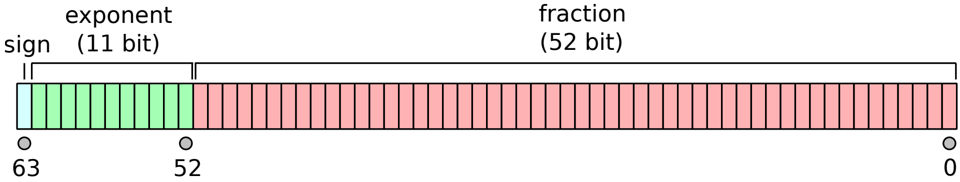


Image: Wikimedia Commons CC by-SA

Representation of real (floating point) numbers

- Example: $\lambda = 3$, $m = 2$, $x = \frac{2}{3}$

$$x = \pm\left(2^{-1} + c_2 2^{-2}\right)2^e$$

  - $c_0 \in \{0, 1\}$
  - $e = \pm a_0 2^0$
  - $a_0 \in \{0, 1\}$
- Truncation: $fl(x) = 2^{-1} = 0.5$
- Round off: $fl(x) = 2^{-1} + 2^{-2} = 0.75$

Today's outline

Trigonometric, Logarithmic, and Exponential computations

- Processors can do logic and arithmetic instructions
- Trigonometric, logarithmic and exponential calculations are “higher-level” functions: exp, sin, cos, tan, sec, arcsin, arccos, arctan, log, ln, ...
- Such functions can be performed using these “low level” instructions, for instance using a Taylor series:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Trigonometric, Logarithmic, and Exponential computations

- Processors can do logic and arithmetic instructions
- Trigonometric, logarithmic and exponential calculations are "higher-level" functions: exp, sin, cos, tan, sec, arcsin, arccos, arctan, log, ln, ...
- Such functions can be performed using these "low level" instructions, for instance using a Taylor series:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Trigonometric, Logarithmic, and Exponential computations

- Processors can do logic and arithmetic instructions
- Trigonometric, logarithmic and exponential calculations are "higher-level" functions: $\exp, \sin, \cos, \tan, \sec, \arcsin, \arccos, \arctan, \log, \ln, \ldots$
- Such functions can be performed using these "low level" instructions, for instance using a Taylor series:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

Trigonometric, Logarithmic, and Exponential computations

- These operations involve many multiplications and additions, and are therefore *expensive*
- Computations can only take finite time, for infinite series, calculations are interrupted at N

$$\sin(x) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^N}{(2N+1)!} x^{2N+1}$$

$$e^x = \sum_{n=0}^{N} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^N}{N!}$$

- This results in a *break error*

Trigonometric, Logarithmic, and Exponential computations

- These operations involve many multiplications and additions, and are therefore *expensive*
- Computations can only take finite time, for infinite series, calculations are interrupted at N

$$\sin(x) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^N}{(2N+1)!} x^{2N+1}$$

$$e^x = \sum_{n=0}^{N} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^N}{N!}$$

- This results in a *break error*

Trigonometric, Logarithmic, and Exponential computations

- These operations involve many multiplications and additions, and are therefore *expensive*
- Computations can only take finite time, for infinite series, calculations are interrupted at N

$$\sin(x) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^N}{(2N+1)!} x^{2N+1}$$

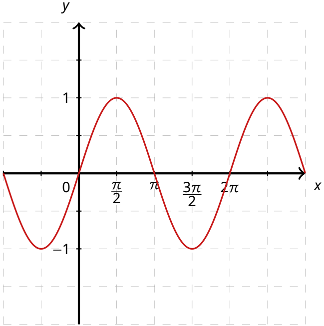
$$e^x = \sum_{n=0}^{N} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^N}{N!}$$

- This results in a *break error*

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Algorithm for sine-computation

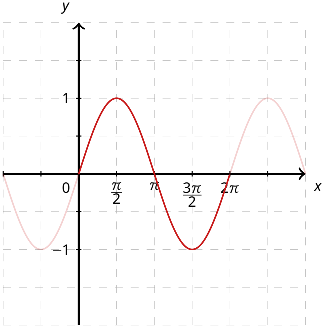
A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:



Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

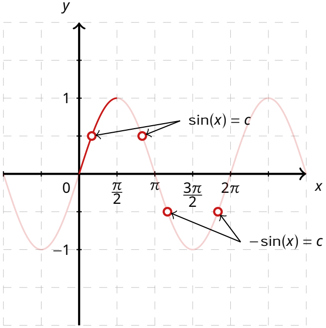
1. Use periodicity so that $0 \leq x \leq 2\pi$



Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry ($0 \leq x \leq \frac{\pi}{2}$)

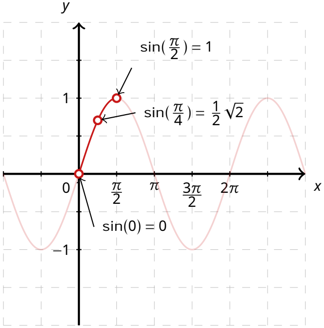


EINDHOVEN UNIVERSITY OF TECHNOLOGY

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry ($0 \leq x \leq \frac{\pi}{2}$)
3. Use lookup tables for known values

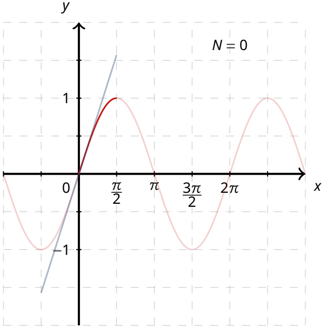


EINDHOVEN UNIVERSITY OF TECHNOLOGY

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

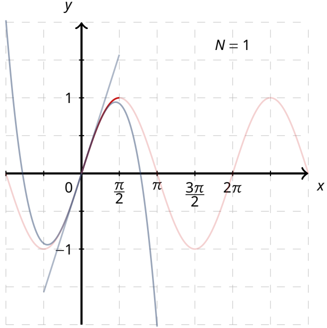
1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry ($0 \leq x \leq \frac{\pi}{2}$)
3. Use lookup tables for known values
4. Perform taylor expansion



Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

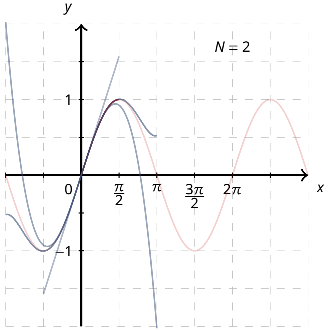
1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry $(0 \leq x \leq \frac{\pi}{2})$
3. Use lookup tables for known values
4. Perform taylor expansion



Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

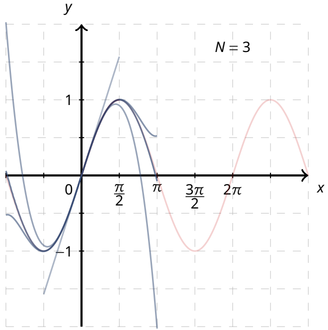
1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry ($0 \leq x \leq \frac{\pi}{2}$)
3. Use lookup tables for known values
4. Perform taylor expansion



Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry ($0 \leq x \leq \frac{\pi}{2}$)
3. Use lookup tables for known values
4. Perform taylor expansion



Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Loss of digits

- During operations such as $+$, $-$, $\times$, $\div$, an error can add up
- Consider the summation of x and y

$$\tilde{x} - \delta \leq x \leq \tilde{x} + \delta \quad \text{and} \quad \tilde{y} - \varepsilon \leq y \leq \tilde{y} + \varepsilon$$

$$(\tilde{x} + \tilde{y}) - (\delta + \varepsilon) \leq x + y \leq (\tilde{x} + \tilde{y}) + (\delta + \varepsilon)$$

Loss of digits: Example 1

$x = \pi, \tilde{x} = 3.1416$

$y = 22/7, \tilde{y} = 3.1429$

Loss of digits: Example 1

$$\left.\begin{array}{l} x = \pi, \tilde{x} = 3.1416 \\ y = 22/7, \tilde{y} = 3.1429 \end{array}\right\} \Rightarrow \left.\begin{array}{l} \delta = \tilde{x} - x = 7.35 \times 10^{-6} \\ \varepsilon = \tilde{y} - y = 4.29 \times 10^{-5} \end{array}\right\}$$

Loss of digits: Example 1

$$\left.\begin{array}{l} x=\pi, \tilde{x}=3.1416 \\ y=22/7, \tilde{y}=3.1429 \end{array}\right\} \Rightarrow \left.\begin{array}{l} \delta=\tilde{x}-x=7.35\times 10^{-6} \\ \varepsilon=\tilde{y}-y=4.29\times 10^{-5} \end{array}\right\}$$

$$x+y=\tilde{x}+\tilde{y}\pm(\delta+\varepsilon)\approx 6.2845-5.025\times 10^{-5}$$

$$x-y=\tilde{x}-\tilde{y}\pm(\delta+\varepsilon)\approx -0.0013+3.55\times 10^{-5}$$

- The absolute error is small ($\approx 10^{-5}$), but the relative error is much bigger (0.028).
- Adding up the errors results in a loss of significant digits!

Loss of digits: Example 2

- Calculate e^{-5}
  - Use the Taylor series
  - Calculate the first 26 terms ($N = 26$)
- Now repeat the calculation, but use for each calculation only 4 digits. What do you find?
- Without errors you would find: $e^{-5} = 0.006738$
- If you only use 4 digits in the calculations, you'll find 0.00998

Loss of digits: Example 2

- Calculate e^{-5}
  - Use the Taylor series
  - Calculate the first 26 terms ($N = 26$)
- Now repeat the calculation, but use for each calculation only 4 digits. What do you find? Use: `round(term, 4)`
- Without errors you would find: $e^{-5} = 0.006738$
- If you only use 4 digits in the calculations, you'll find 0.00998

Loss of digits: Example 2

- Calculate e^{-5}
  - Use the Taylor series
  - Calculate the first 26 terms ($N = 26$)
- Now repeat the calculation, but use for each calculation only 4 digits. What do you find? Use: `round(term, 4)`
- Without errors you would find: $e^{-5} = 0.006738$
- If you only use 4 digits in the calculations, you'll find 0.00998

Loss of digits: Example 2

- Calculate e^{-5}
  - Use the Taylor series
  - Calculate the first 26 terms ($N = 26$)
- Now repeat the calculation, but use for each calculation only 4 digits. What do you find? Use: `round(term, 4)`
- Without errors you would find: $e^{-5} = 0.006738$
- If you only use 4 digits in the calculations, you'll find 0.00998

Badly (ill) conditioned problems

We consider a system $F(x,y)$ that computes a solution from input data. The input data may have errors:

$$\xrightarrow{x} \boxed{F(x,y) = 0} \xrightarrow{y}$$ (Without errors)

Badly (ill) conditioned problems

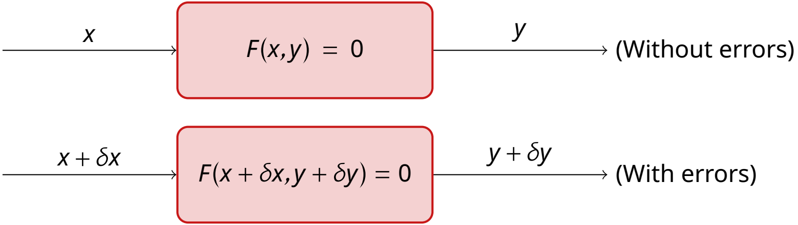
We consider a system $F(x,y)$ that computes a solution from input data. The input data may have errors:

$$x \longrightarrow \boxed{F(x,y) \;=\; 0} \xrightarrow{\quad y \quad} \text{(Without errors)}$$

$$x+\delta x \longrightarrow \boxed{F(x+\delta x, y+\delta y) = 0} \xrightarrow{\quad y+\delta y \quad} \text{(With errors)}$$

Badly (ill) conditioned problems

We consider a system $F(x,y)$ that computes a solution from input data. The input data may have errors:



$$y(x+\delta x) - y(x) \approx y'(x)\delta x$$

Propagated error on the basis of Taylor expansion

$$C = \max_{\delta x}\left(\left|\frac{\delta y/y}{\delta x/x}\right|\right)$$

Condition criterion, $C < 10$ error development small

Badly (ill) conditioned problems: Example

Solve the following linear system in Python using double and single precision:

$$A = \begin{bmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{bmatrix}, \quad x = \begin{bmatrix} 0.8642 \\ 0.1440 \end{bmatrix}$$

Badly (ill) conditioned problems: Example

Solve the following linear system in Python using double and single precision:

$$A = \begin{bmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{bmatrix}, \quad x = \begin{bmatrix} 0.8642 \\ 0.1440 \end{bmatrix}, \quad y = \begin{bmatrix} 2.0 \\ -2.0 \end{bmatrix}$$

Badly (ill) conditioned problems: Example

Solve the following linear system in Python using double and single precision:

$$A = \begin{bmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{bmatrix}, \quad x = \begin{bmatrix} 0.8642 \\ 0.1440 \end{bmatrix}, \quad y = \begin{bmatrix} 2.0 \\ -2.0 \end{bmatrix}$$

Double precision

```python
import numpy as np

# Double precision
A = np.array([[1.2969, 0.8648], [0.2161,
     0.1441]], dtype=np.float64)
x = np.array([0.8642, 0.1440], dtype=np.float64)
y = np.linalg.solve(A, x)
print("y =")
print(y)
```

Badly (ill) conditioned problems: Example

Solve the following linear system in Python using double and single precision:

$$A = \begin{bmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{bmatrix}, \quad x = \begin{bmatrix} 0.8642 \\ 0.1440 \end{bmatrix}, \quad y = \begin{bmatrix} 2.0 \\ -2.0 \end{bmatrix}$$

Double precision

```python
import numpy as np

# Double precision
A = np.array([[1.2969, 0.8648], [0.2161,
     0.1441]], dtype=np.float64)
x = np.array([0.8642, 0.1440], dtype=np.float64)
y = np.linalg.solve(A, x)
print("y =")
print(y)
```

Single precision

```python
import numpy as np

# Single precision
A = np.array([[1.2969, 0.8648], [0.2161,
     0.1441]], dtype=np.float32)
x = np.array([0.8642, 0.1440], dtype=np.float32)
y = np.linalg.solve(A, x)
print("y =")
print(y)
```

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Badly (ill) conditioned problems: Example

- Python does not automatically warn about bad condition number. You need to compute and check it manually using NumPy:

```
if np.linalg.cond(A)> threshold: raise("Ill conditioned error")
```

- The `cond` number is the condition number computed using NumPy.
- A small error in A (the matrix) results in a big error in the solution. This is called an ill-conditioned problem.

Badly (ill) conditioned problems: Example

- Python does not automatically warn about bad condition number. You need to compute and check it manually using NumPy:

```
if np.linalg.cond(A)> threshold: raise("Ill conditioned error")
```

- The `cond` number is the condition number computed using NumPy.
- A small error in A (the matrix) results in a big error in the solution. This is called an ill-conditioned problem.

Badly (ill) conditioned problems: Example

- Python does not automatically warn about bad condition number. You need to compute and check it manually using NumPy:

  ```
  if np.linalg.cond(A)> threshold: raise("Ill conditioned error")
  ```
- The `cond` number is the condition number computed using NumPy.
- A small error in *A* (the matrix) results in a big error in the solution. This is called an ill-conditioned problem.

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- **(Un)stable methods**
- Symbolic math
- Summary

(Un)stable methods

- The condition criterion does not tell you anything about the quality of a numerical solution method!
- It is very well possible that a certain solution method is more sensitive for one problem than another
- If the method propagates the error, we call it an *unstable method*. Let's look at an example.

The Golden mean

The Golden Mean is a well known ratio and one of the solutions of $\phi^2 = \phi + 1$, such that $\phi = \frac{1+\sqrt{5}}{2}$, and $\phi - 1 = \phi^{-1}$. Many relations to compute ϕ have been established (which pose often an interesting source for creating small test programs).

- Let's evaluate the following recurrent relationship:

$$y_{n+1} = y_{n-1} - y_n$$

$$y_0 = 1, \quad y_1 = 1 - \phi = \phi^{-1} = \frac{2}{1+\sqrt{5}}$$

- Alternatively, a closed form power law relation exists, which again computes y_n:

$$y_n = x^{-n}, \quad n = 0, 1, 2, \ldots, \quad x = \frac{1+\sqrt{5}}{2}$$

EINDHOVEN UNIVERSITY OF TECHNOLOGY

The Golden mean

The Golden Mean is a well known ratio and one of the solutions of $\phi^2 = \phi + 1$, such that $\phi = \frac{1+\sqrt{5}}{2}$, and $\phi - 1 = \phi^{-1}$. Many relations to compute ϕ have been established (which pose often an interesting source for creating small test programs).

- Let's evaluate the following recurrent relationship:

$$y_{n+1} = y_{n-1} - y_n$$

$$y_0 = 1, \quad y_1 = 1 - \phi = \phi^{-1} = \frac{2}{1+\sqrt{5}}$$

- Alternatively, a closed form power law relation exists, which again computes y_n:

$$y_n = x^{-n}, \quad n = 0, 1, 2, \ldots, \quad x = \frac{1+\sqrt{5}}{2}$$

The Golden mean

Recurrent version

```python
import numpy as np

def golden_mean_recurrent(Ntot):
    # Initialize the series with the given
        initial conditions
    y = np.zeros(Ntot)
    y[0] = 1
    y[1] = 2 / (1 + np.sqrt(5))

    # Perform the recurrence to fill in the rest
        of the series
    for n in range(2, Ntot):
        y[n] = y[n-1] - y[n-2]
    return y
```

The Golden mean

Recurrent version

```python
import numpy as np

def golden_mean_recurrent(Ntot):
    # Initialize the series with the given initial conditions
    y = np.zeros(Ntot)
    y[0] = 1
    y[1] = 2 / (1 + np.sqrt(5))

    # Perform the recurrence to fill in the rest of the series
    for n in range(2, Ntot):
        y[n] = y[n-1] - y[n-2]
    return y
```

Power law version

```python
import numpy as np

def golden_mean_powerlaw(Ntot):
    # Initialize the constant value
    x = (1 + np.sqrt(5)) / 2

    # Generate a range of values from 0 to Ntot and apply the power law
    y = x ** -np.arange(0, Ntot + 1)

    return y
```

The Golden mean

Recurrent version

```python
import numpy as np

def golden_mean_recurrent(Ntot):
    # Initialize the series with the given
        initial conditions
    y = np.zeros(Ntot)
    y[0] = 1
    y[1] = 2 / (1 + np.sqrt(5))

    # Perform the recurrence to fill in the rest
        of the series
    for n in range(2, Ntot):
        y[n] = y[n-1] - y[n-2]
    return y
```

Power law version

```python
import numpy as np

def golden_mean_powerlaw(Ntot):
    # Initialize the constant value
    x = (1 + np.sqrt(5)) / 2

    # Generate a range of values from 0 to Ntot
        and apply the power law
    y = x ** -np.arange(0, Ntot + 1)

    return y
```

- Compare the outcomes: `plot(goldenMeanPowerlaw(40)- goldenMeanRecurrent(40))`

The Golden mean

Recurrent version

```python
1  import numpy as np
2
3  def golden_mean_recurrent(Ntot):
4      # Initialize the series with the given initial conditions
5      y = np.zeros(Ntot)
6      y[0] = 1
7      y[1] = 2 / (1 + np.sqrt(5))
8
9      # Perform the recurrence to fill in the rest of the series
10     for n in range(2, Ntot):
11         y[n] = y[n-1] - y[n-2]
12     return y
```

Power law version

```python
1  import numpy as np
2
3  def golden_mean_powerlaw(Ntot):
4      # Initialize the constant value
5      x = (1 + np.sqrt(5)) / 2
6
7      # Generate a range of values from 0 to Ntot and apply the power law
8      y = x ** -np.arange(0, Ntot + 1)
9
10     return y
```

- Compare the outcomes: `plot(goldenMeanPowerlaw(40)- goldenMeanRecurrent(40))`
- See what happens if you use single precision (add `dtype=np.float32` to line 5 (`np.zeros`) and line 8 (`np.arange`)).

The Golden mean

n	Recurrent	Power law
0	1.0000	1.0000
1	0.6180	0.6180
2	0.3820	0.3820
3	0.2361	0.2361
...	...	...
37	$1.714 \cdot 10^{-08}$	$1.851 \cdot 10^{-08}$
38	$1.366 \cdot 10^{-08}$	$1.144 \cdot 10^{-08}$
39	$3.485 \cdot 10^{-08}$	$7.071 \cdot 10^{-09}$
40	$1.017 \cdot 10^{-08}$	$4.370 \cdot 10^{-09}$

- The recurrent approach enlarges errors from earlier calculations!

The Golden mean

n	Recurrent	Power law
0	1.0000	1.0000
1	0.6180	0.6180
2	0.3820	0.3820
3	0.2361	0.2361
...	...	...
37	$1.714 \cdot 10^{-08}$	$1.851 \cdot 10^{-08}$
38	$1.366 \cdot 10^{-08}$	$1.144 \cdot 10^{-08}$
39	$3.485 \cdot 10^{-08}$	$7.071 \cdot 10^{-09}$
40	$1.017 \cdot 10^{-08}$	$4.370 \cdot 10^{-09}$

- The recurrent approach enlarges errors from earlier calculations!

Example 1: Explanation

Recall example 1, where the errors blew up our computation of 0.1, whereas they did not for 2. Why did we see these results?

Example 1: Explanation

Recall example 1, where the errors blew up our computation of 0.1, whereas they did not for 2. Why did we see these results?

- The number 0.1 is not exactly represented in binary
  - A tiny error can accumulate up to catastrophic proportions!
- The number 2 does have an exact binary representation

Example 1: Explanation

Recall example 1, where the errors blew up our computation of 0.1, whereas they did not for 2. Why did we see these results?

- The number 0.1 is not exactly represented in binary
  - A tiny error can accumulate up to catastrophic proportions!
- The number 2 does have an exact binary representation

Example 2 (large sine series)

The `np.sin(1e40*np.pi)` result gives poor results because 1e40 has an error margin on the order of floating-point machine epsilon, which is roughly 1×10^{-16} in Python (double-precision floating-point format). In Python, as in many computing environments, the number of $2 \cdot \pi$ cycles is still much larger than $10^{40} \cdot 10^{-16}$. Also, π is not stored with an infinite number of digits, which further contributes to the imprecision.

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Example 3

Start your calculation program of choice (Excel, Python, ...)

Example 3

Start your calculation program of choice (Excel, Python, ...)

Calculate the result of y:

$$y = e^{\pi} - \pi$$

Example 3

Start your calculation program of choice (Excel, Python, ...)

Calculate the result of y:

$$y = e^{\pi} - \pi = 19.999099979$$

Example 3

Start your calculation program of choice (Excel, Python, ...)

Calculate the result of y:

$$y = e^{\pi} - \pi = 19.999099979 \neq 20$$

Example 3

Start your calculation program of choice (Excel, Python, ...)

Calculate the result of y:

$$y = e^{\pi} - \pi = 19.999099979 \neq 20$$

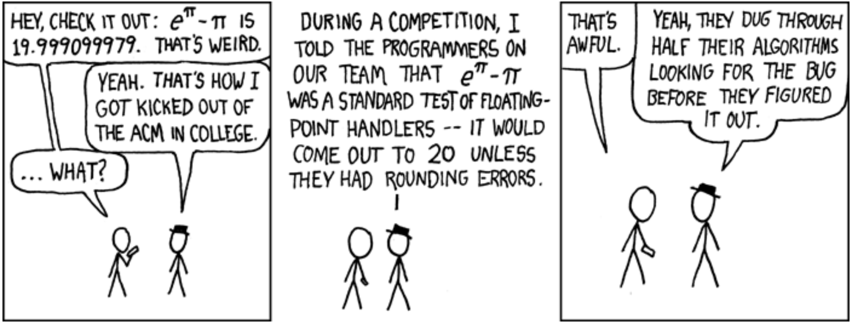


Image: xkcd

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Symbolic math packages

Definition

The use of computers to manipulate mathematical equations and expressions in symbolic form, as opposed to manipulating the numerical quantities represented by those symbols.

- Symbolic integration or differentiation, substitution of one expression into another
- Simplification of an expression, change of subject etc.
- Packages and toolboxes:

Symbolic math packages

Mathematica	Well known software package, license available via TU/e
Maple	Well known, license available via TU/e
Wolfram|Alpha	Web-based interface by Mathematica developer. Less powerful in mathematical respect, but more accessible and has a broad application range (unit conversion, semantic commands).
Sage	Open-source alternative to Maple, Mathematica, Magma, and MATLAB.
Matlab	Symbolic math toolbox
Python	SymPy

Symbolic math: simplify

$$f(x) = (x-1)(x+1)(x^2+1)+1$$

Symbolic math: simplify

$$f(x) = (x-1)(x+1)(x^2+1)+1$$

```python
from sympy import symbols, simplify

x = symbols('x') # Alt: from sympy.abc import x,y,z
f = (x - 1)*(x + 1)*(x**2 + 1) + 1
f_simplified = simplify(f)
print(f_simplified)
```

Symbolic math: simplify

$$f(x) = (x-1)(x+1)(x^2+1)+1$$

```python
from sympy import symbols, simplify

x = symbols('x') # Alt: from sympy.abc import x,y,z
f = (x - 1)*(x + 1)*(x**2 + 1) + 1
f_simplified = simplify(f)
print(f_simplified)
```

```
f_simplified = x**4
```

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

```python
from sympy import symbols, simplify, integrate, diff

x = symbols('x')
f = 1/(x**3+1)
my_f_int = integrate(f, x)
my_f_diff = diff(my_f_int, x)
my_f_diff_simplified = simplify(my_f_diff)
print(my_f_diff_simplified)
```

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

```python
from sympy import symbols, simplify, integrate, diff

x = symbols('x')
f = 1/(x**3+1)
my_f_int = integrate(f, x)
my_f_diff = diff(my_f_int, x)
my_f_diff_simplified = simplify(my_f_diff)
print(my_f_diff_simplified)
```

```
my_f_diff_simplified = 1/(x**3 + 1)
```

Symbolic math: exercises

Exercise 1

Simplify the following expression:

$$f(x) = \frac{2\tan x}{(1+\tan^2 x)}$$

Symbolic math: exercises

Exercise 1

Simplify the following expression:

$$f(x) = \frac{2\tan x}{(1+\tan^2 x)} = \sin 2x$$

```python
from sympy import symbols, trigsimp, tan

x = symbols('x')
expr = 2*tan(x)/(1 + tan(x)**2)
simplified_expr = trigsimp(expr)
```

Symbolic math: exercises

Exercise 2

Calculate the *value* of p:

$$p = \int_0^{10} \frac{e^x - e^{-x}}{\sinh x} dx$$

Symbolic math: exercises

Exercise 2

Calculate the *value* of p:

$$p = \int_0^{10} \frac{e^x - e^{-x}}{\sinh x} dx$$

```python
from sympy import exp, sinh, integrate, symbols

x = symbols("x")
f = ((exp(x)- exp(-x))/sinh(x)).simplify()
p = integrate(f, (x, 0, 10))
print(p)
```

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Symbolic math: exercises

Exercise 2

Calculate the *value* of p:

$$p = \int_0^{10} \frac{e^x - e^{-x}}{\sinh x} dx$$

```python
from sympy import exp, sinh, integrate, symbols

x = symbols("x")
f = ((exp(x)- exp(-x))/sinh(x)).simplify()
p = integrate(f, (x, 0, 10))
print(p)
```

p = 20

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Symbolic math: root finding

A root finding method searches for the values where a function reaches zero. We will cover the numerical methods later, here we show how to use root finding with symbolic math in Python.

Symbolic math function

$$f(x) = \frac{3}{x^2 + 3x} - 2$$

Symbolic math: root finding

A root finding method searches for the values where a function reaches zero. We will cover the numerical methods later, here we show how to use root finding with symbolic math in Python.

Symbolic math function

$$f(x) = \frac{3}{x^2 + 3x} - 2$$

```python
from sympy import solve, symbols

x = symbols("x")
f = 3 / (x**2 + 3*x) - 2
solutions = solve(f, x)
print(solutions)
```

Symbolic math: root finding

A root finding method searches for the values where a function reaches zero. We will cover the numerical methods later, here we show how to use root finding with symbolic math in Python.

Symbolic math function

$$f(x) = \frac{3}{x^2 + 3x} - 2$$

```python
from sympy import solve, symbols

x = symbols("x")
f = 3 / (x**2 + 3*x) - 2
solutions = solve(f, x)
print(solutions)
```

```
solutions = [-3/2 + sqrt(15)/2, -sqrt(15)/2 - 3/2]
```

Today's outline

Summary

- Numerical errors may arise due to truncation, roundoff and break errors, which may seriously affect the accuracy of your solution
- Errors may propagate and accumulate, leading to smaller accuracy
- Ill-conditioned problems and unstable methods have to be identified so that proper measures can be taken
- Symbolic math computations may be performed to solve certain equations algebraically, bypassing numerical errors, but this is not always possible.

Summary

- Numerical errors may arise due to truncation, roundoff and break errors, which may seriously affect the accuracy of your solution
- Errors may propagate and accumulate, leading to smaller accuracy
- Ill-conditioned problems and unstable methods have to be identified so that proper measures can be taken
- Symbolic math computations may be performed to solve certain equations algebraically, bypassing numerical errors, but this is not always possible.

Summary

- Numerical errors may arise due to truncation, roundoff and break errors, which may seriously affect the accuracy of your solution
- Errors may propagate and accumulate, leading to smaller accuracy
- Ill-conditioned problems and unstable methods have to be identified so that proper measures can be taken
- Symbolic math computations may be performed to solve certain equations algebraically, bypassing numerical errors, but this is not always possible.

Summary

- Numerical errors may arise due to truncation, roundoff and break errors, which may seriously affect the accuracy of your solution
- Errors may propagate and accumulate, leading to smaller accuracy
- Ill-conditioned problems and unstable methods have to be identified so that proper measures can be taken
- Symbolic math computations may be performed to solve certain equations algebraically, bypassing numerical errors, but this is not always possible.