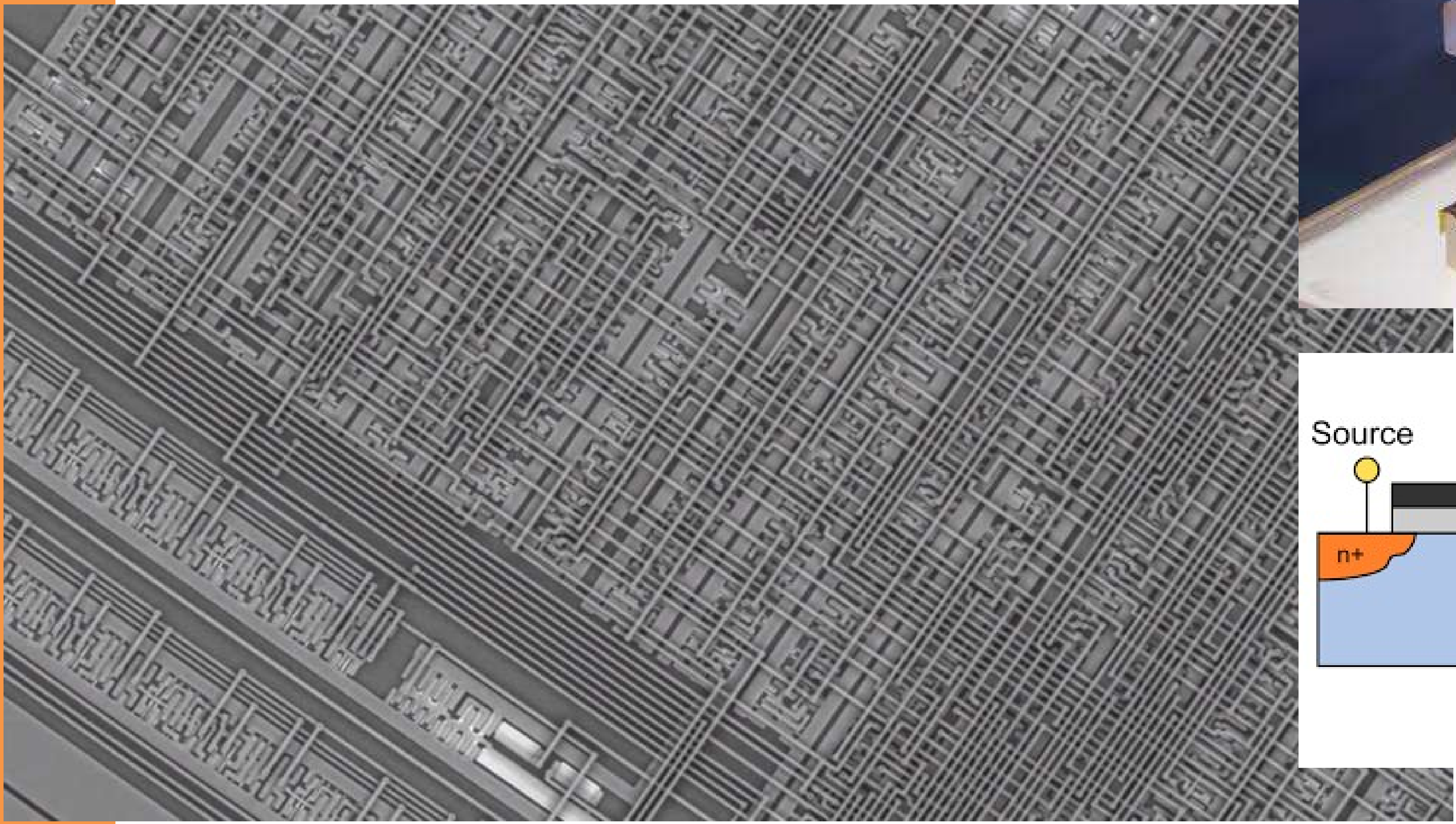
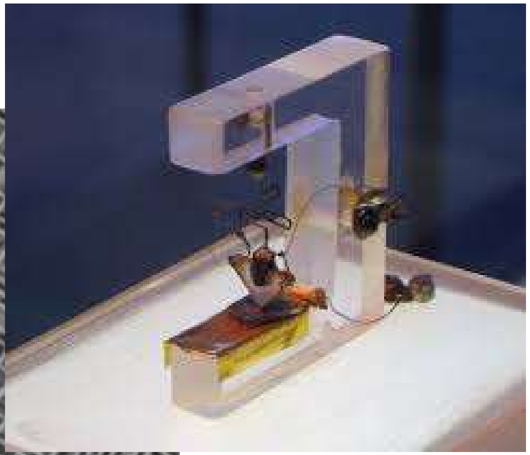
THE UNIVERSITY of TENNESSEE KNOXVILLE

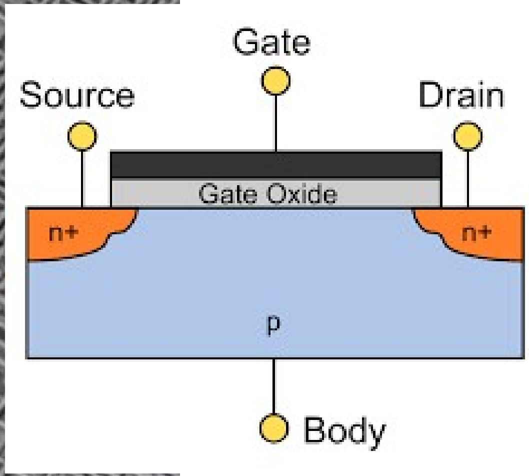
Lecture 03: Number Representations and Format

Sergei V. Kalinin

What's inside the computer?

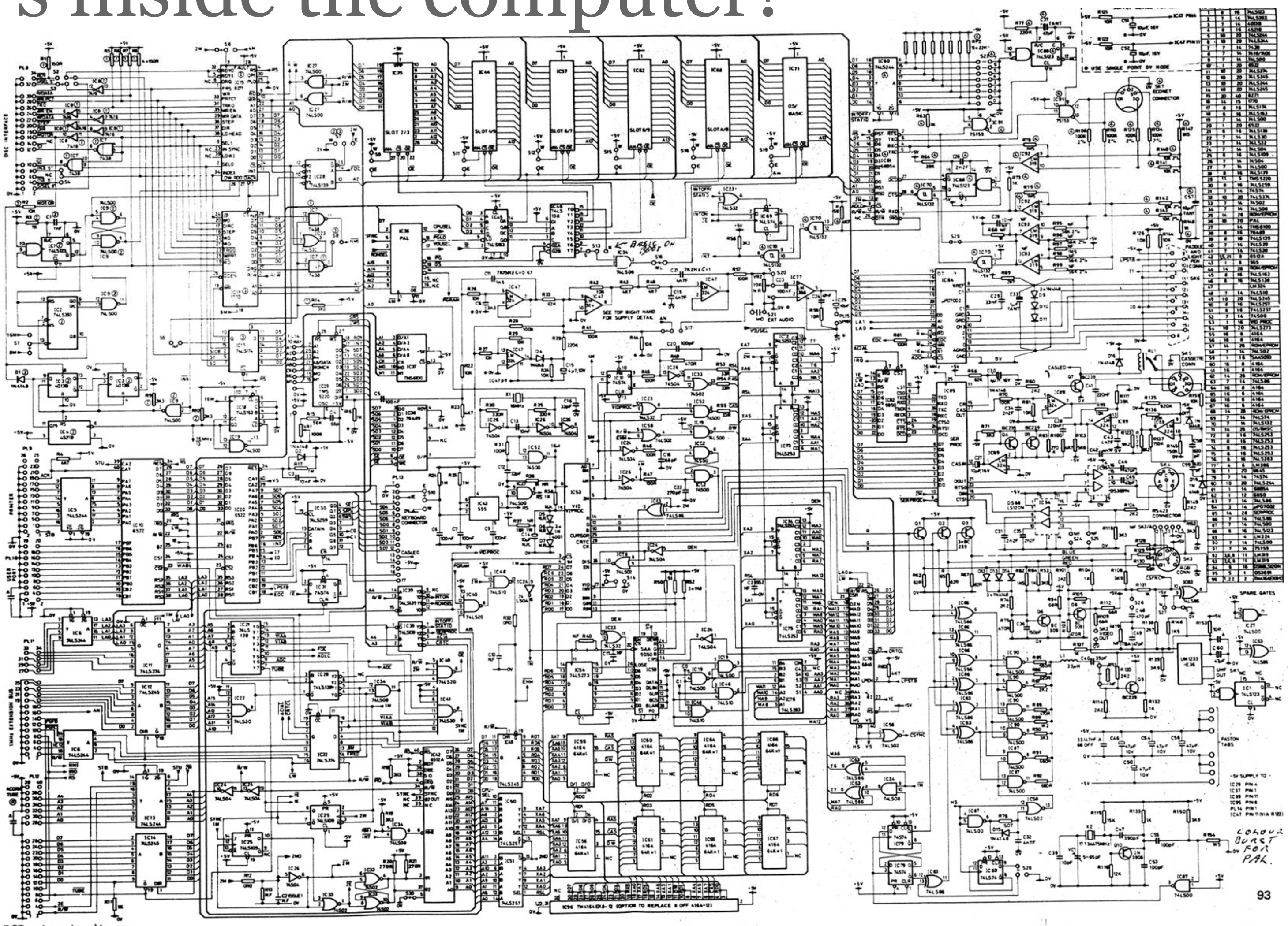






https://www.extremetech.com/extreme/191996-zoom-into-a-computer-chip-watch-this-video-to-fully-appreciate-just-how-magical-modern-microchips-are
https://www.britannica.com/technology/transistor/Innovation-at-Bell-Labs

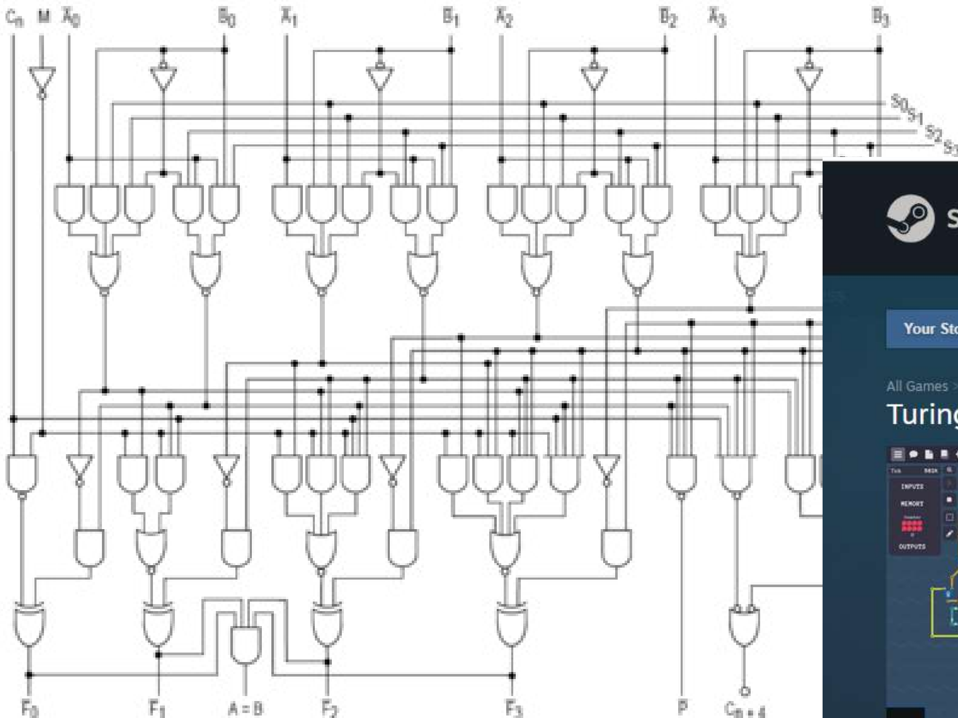
What’s inside the computer?

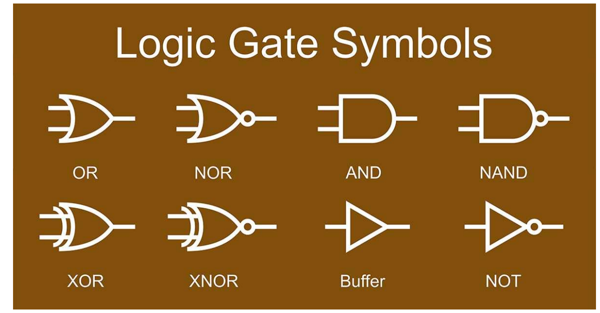


PCB circuit diagram

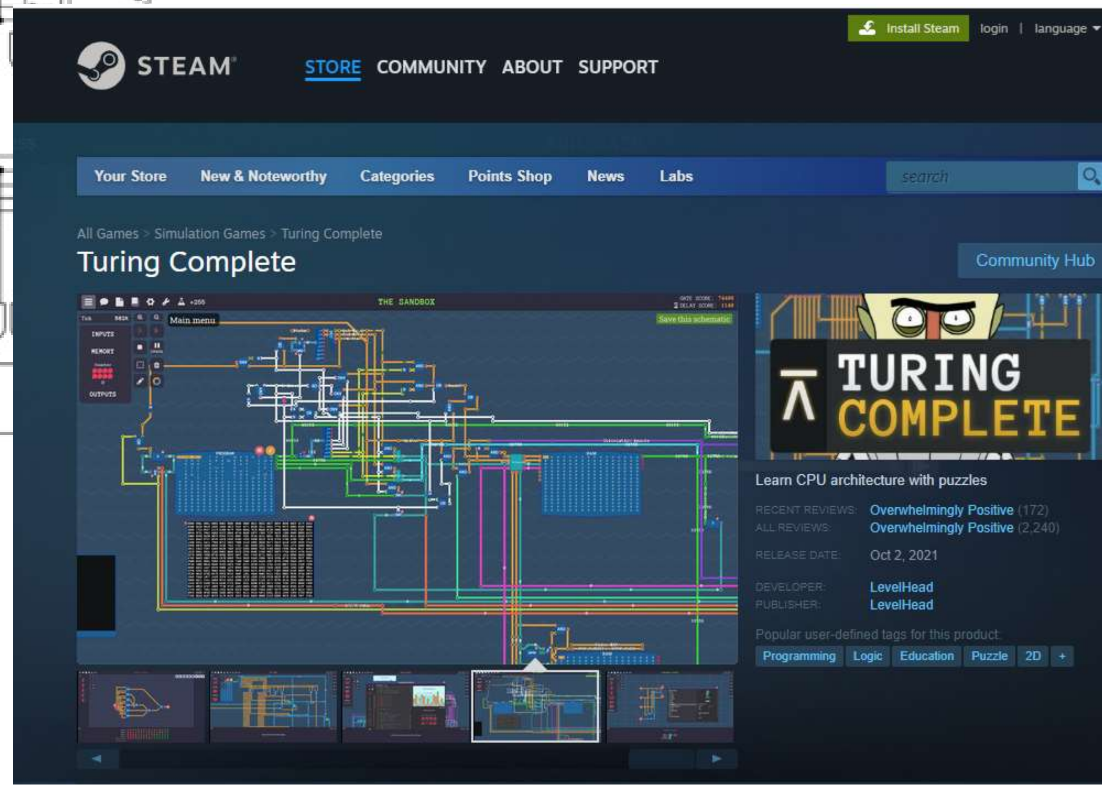
https://mdfs.net/Info/Comp/BBC/Circuits/BBC/bbcplus.gif

Binary logic!





https://en.wikipedia.org/wiki/74181



Numbers

Integer (int): Represents whole numbers, both positive and negative. Example: 5, -3, 42

Floating Point (float): Represents real numbers (numbers with a fractional part). Includes a decimal point. Example: 3.14, -0.001, 2.0

Complex Numbers (complex): Consists of a real and an imaginary part. The imaginary part is denoted by a 'j' or 'J'. Example: 3 + 4j, -1.5 + 2.5j

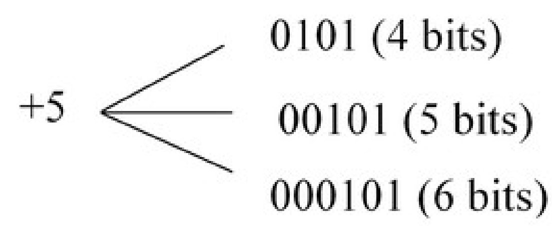
Binary: Represents numbers in base 2. Prefixed with 0b or 0B. Example: 0b1010 (equivalent to decimal 10)

Octal: Represents numbers in base 8. Prefixed with 0o or 0O (the letter 'o', not the number '0'). Example: 0o12 (equivalent to decimal 10)

Hexadecimal: Represents numbers in base 16. Prefixed with 0x or 0X. Uses digits from 0 to 9 and letters from A to F (or a to f). Example: 0xA (equivalent to decimal 10)

Integer numbers

Numbers on a computer are represented by bits



Most typical native formats:

- 32-bit integer, range –2,147,483,647 (-2^{31}) to +2,147,483,647 (2^{31})
- 64-bit integer, range ~ -10^{18} (-2^{63}) to $+10^{18}$ (2^{63})

Python supports natively larger numbers but calculations can become slow

Not all numbers can be fully represented!

A floating-point number in Python is composed of two parts: the mantissa (or significand) and the exponent, both of which are based on powers of two. The format is similar to scientific notation, where a number is represented as a * 2^b. Here, a is the mantissa, and b is the exponent.

Precision: Floating-point numbers are typically double precision (64-bit) following the IEEE 754 standard. This provides a significant degree of accuracy but can still lead to rounding errors in complex calculations.

Syntax: Floating-point numbers can be declared simply by including a decimal point. For example, 3.0, 4.2, -0.5. They can also be specified using scientific notation, e.g., 1.23e4 which is equivalent to 12300.0.

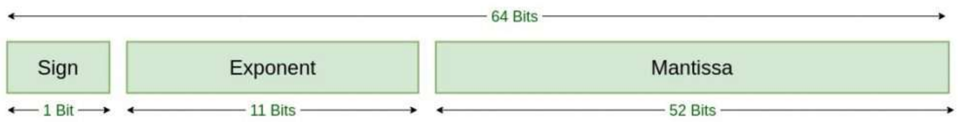
Limitations: Due to their binary nature, not all decimal fractions can be precisely represented. For instance, 0.1 in Python is an approximation, leading to potential precision errors in calculations.

Python uses a type of rounding to minimize this error, but it's important to be aware of it, especially in numerical computations.

Not all numbers can be fully represented!

Floating point numbers represented by bit sequences separated into:

- Sign S
- Exponent E
- Mantissa M (significant digits)



Double Precision
IEEE 754 Floating-Point Standard

$$x = S \times M \times 2^{E-e}$$

Main consequence: Floating-point numbers are not exact!

For example, with 52 bits one can store about 16 decimal digits

Range: from ~ -10^{308} to 10^{308} for a 64-bit float

There are workarounds

- **Creation of Rational Numbers:** You can create fractions from integers, floats, decimal numbers, or strings representing a fraction.

- **Arithmetic Operations:** The module supports basic arithmetic operations like addition, subtraction, multiplication, and division with fractions.

- **Maintaining Exactness:** Fractions are stored as two integers, representing the numerator and the denominator. This ensures exact arithmetic operations, unlike floating-point numbers where precision issues can arise.

- **Conversion and Simplification:** Fractions are automatically simplified. For example, fractions.Fraction(4, 6) will simplify to 2/3. You can also convert fractions to other numeric types like floats or decimals.

```
from fractions import Fraction

# Creating fractions
f1 = Fraction(3, 4)  # Fraction from two integers
f2 = Fraction('1/4')  # Fraction from a string
f3 = Fraction(0.5)    # Fraction from a float

# Arithmetic operations
sum_f = f1 + f2      # Adds 3/4 and 1/4
mul_f = f1 * f3      # Multiplies 3/4 and 1/2

print("Sum:", sum_f)  # Output: Sum: 1
print("Product:", mul_f)  # Output: Product: 3/8
```

Floating-point number representation

When you write

$$x = 1.$$

What it means

$$x = 1. + \varepsilon_M, \qquad \varepsilon_M \sim 10^{-16}$$ for a 64-bit float

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Example: Equality test

```python
x = 1.1 + 2.2

print("x = ",x)

if (x == 3.3):
    print("x == 3.3 is True")
else:
    print("x == 3.3 is False")
```

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Example: Equality test

```python
x = 1.1 + 2.2

print("x = ",x)

if (x == 3.3):
    print("x == 3.3 is True")
else:
    print("x == 3.3 is False")
```

```
x =  3.3000000000000003
x == 3.3 is False
```

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Example: Equality test

```python
x = 1.1 + 2.2

print("x = ",x)

if (x == 3.3):
    print("x == 3.3 is True")
else:
    print("x == 3.3 is False")
```

```
x =  3.3000000000000003
x == 3.3 is False
```

Instead, you can do

```python
print("x = ",x)

# The desired precision
eps = 1.e-12

# The comparison
if (abs(x-3.3) < eps):
    print("x == 3.3 to a precision of",eps,"is True")
else:
    print("x == 3.3 to a precision of",eps,"is False")
```

```
x =  3.3000000000000003
x == 3.3 to a precision of 1e-12 is True
```

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Error accumulation

$x = 1. + \varepsilon_M, \qquad \varepsilon_M \sim 10^{-16}$ unavoidable round-off error

Errors also accumulate through arithmetic operations,
e.g.

$$y = \sum_{i=1}^{N} x_i$$

- $\sigma_y \sim \sqrt{N}\epsilon_M$ if errors are independent
- $\sigma_y \sim N\epsilon_M$ if errors are correlated
- σ_y can be large in some other cases

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Example: Two large numbers with small difference

Let us have $x = 1$ and $y = 1 + \delta\sqrt{2}$

$$\delta^{-1}(y - x) = \sqrt{2} = 1.41421356237\,\ldots$$

Let us test this relation on a computer for a very small value of $\delta = 10^{-14}$

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Example: Two large numbers with small difference

Let us have $x = 1$ and $y = 1 + \delta\sqrt{2}$

$$\delta^{-1}(y - x) = \sqrt{2} = 1.41421356237\ldots$$

Let us test this relation on a computer for a very small value of $\delta = 10^{-14}$

```
from math import sqrt

delta = 1.e-14
x = 1.
y = 1. + delta * sqrt(2)
res = (1./delta)*(y-x)
print(delta,"* (y-x) = ",res)
print("The accurate value is sqrt(2) = ", sqrt(2))
print("The difference is ", res - sqrt(2))
```

```
1e-14 * (y-x) =  1.4210854715202004
The accurate value is sqrt(2) =  1.4142135623730951
The difference is  0.006871909147105226
```

From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Other examples (see the sample code)

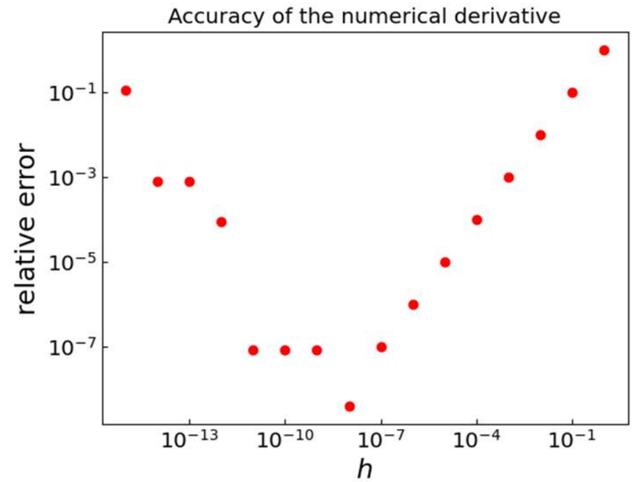
- Roots of a quadratic equation with $|ac| \ll b^2$ (cancellation of two large numbers)

$$ax^2 + bx + c = 0 \qquad\qquad x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

- Simple numerical derivative

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Sometimes a small *h* is too small



From the course by Volodymyr Vovchenko,
https://github.com/vlvovch/PHYS6350-ComputationalPhysics

Reference Materials

I will provide copies of lecture notes, presentations, and Colabs on GitHub and Canvas. There is no specific textbook for the course, and we will take material from a variety of sources including:

- Andrew Bird et al, Python Workshop, https://www.packtpub.com/product/the-python-workshop/9781839218859
- Oswaldo Martin, Bayesian Analysis with Python - Second Edition, https://subscription.packtpub.com/book/data/9781789341652/
- Alexander Molak, Causal Inference and Discovery in Python, https://subscription.packtpub.com/book/data/9781804612989/

Homework 1:

- Create new Colab, https://colab.google/
- Chapter 1-4 and 10, Python Workshop.

Homework, midterm, and finals format

- All homeworks, midterms, and finals will be in the Google Colab format
- Use the code for programming exercises and markdown fields for text responses
- Share in the "comment" or "editor" modes
- The Colabs should save all graph outputs
- The Colabs should be able to run from the beginning to end (e.g. if I restart the runtime and run all)
- Submit to

Homework 1:

- Create new Colab, https://colab.google/
- Chapter 1-4 and 10, Python Workshop.