Computational Physics (PHYS6350)

Lecture 2: Data Visualization, Machine Precision

- Data visualization (plotting with matplotlib as an example)
- Accuracy of integer and floating-point number representation

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics
Online textbook: https://vovchenko.net/computational-physics/

Data visualization

- Line plots
- Scatter plots
- Contour/density plots (2D data)

References: [Chapter 3](#) of *Computational Physics* by Mark Newman

Matplotlib documentation

Plotting the data

Computer programs produce numerical data

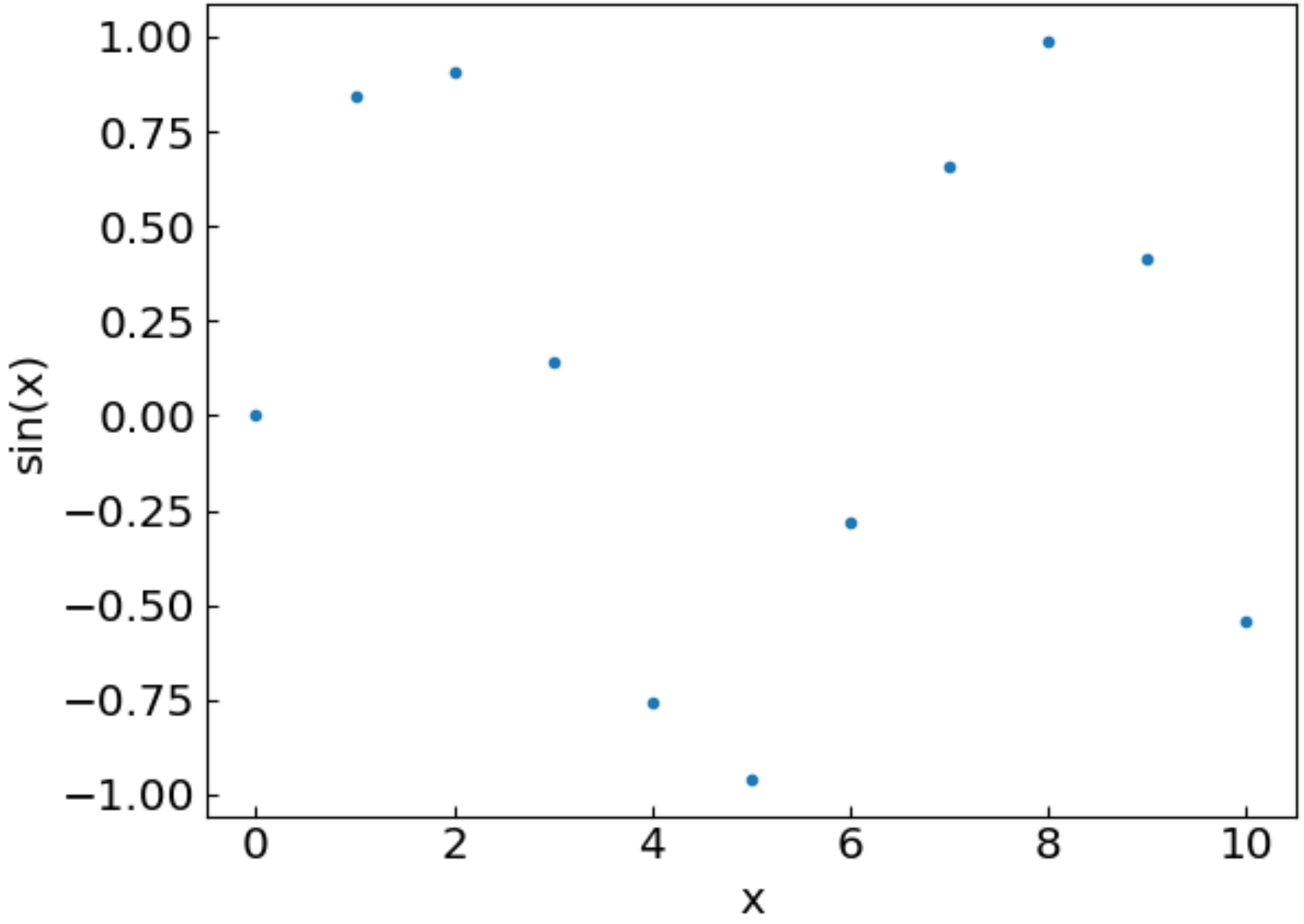
Numbers alone do not always make it easy to understand the behavior of the system and its properties

Consider a function $y = \sin(x)$

Let us calculate it for 10 equidistant points in the interval x = 0…10

x	sin(x)
0	0.
1	0.841471
2	0.9092974
3	0.14112
4	-0.7568025
5	-0.9589243
6	-0.2794155
7	0.6569866
8	0.9893582
9	0.4121185
10	-0.5440211

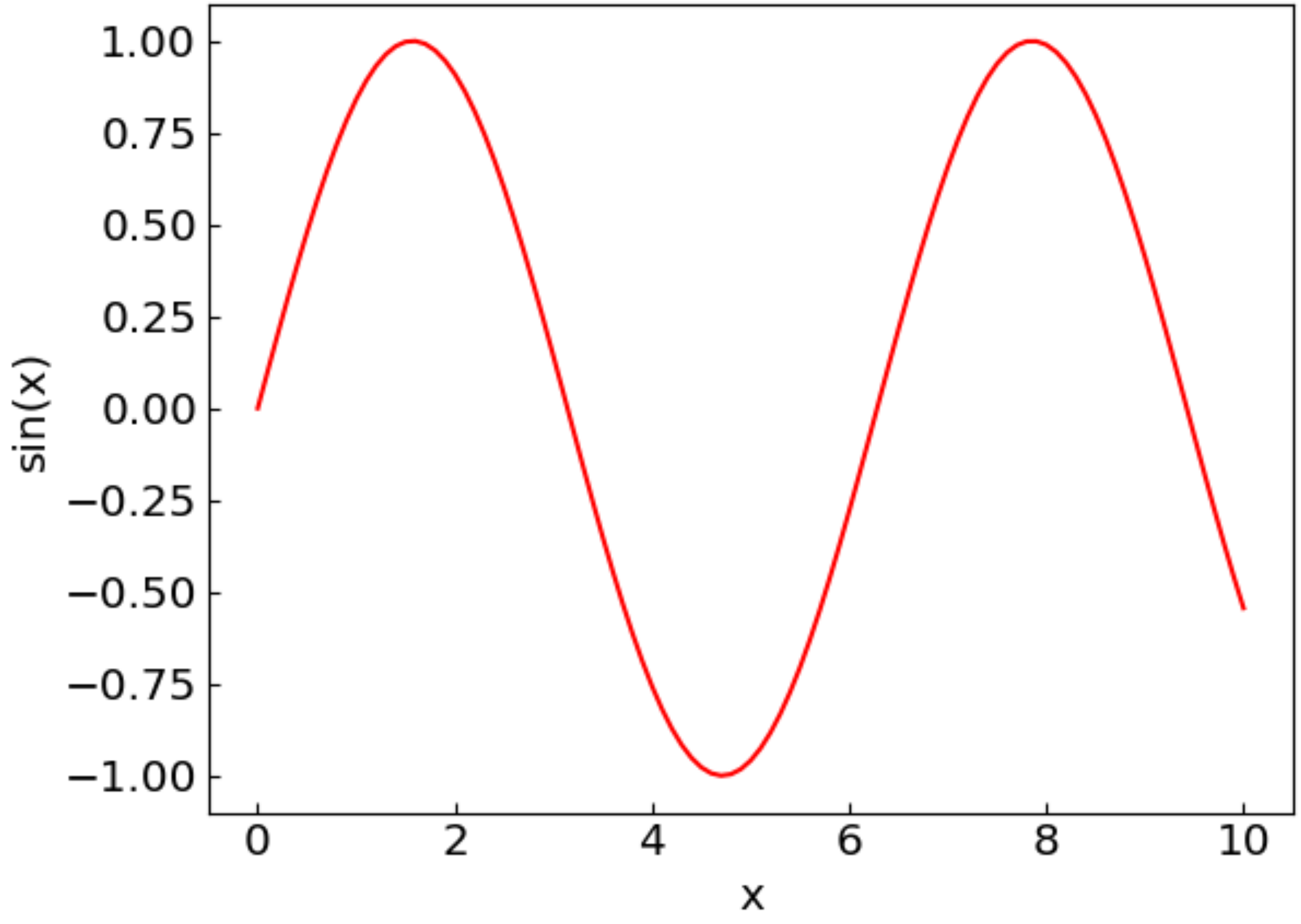
Putting it on a graph



x	sin(x)
0	0.
1	0.841471
2	0.9092974
3	0.14112
4	-0.7568025
5	-0.9589243
6	-0.2794155
7	0.6569866
8	0.9893582
9	0.4121185
10	-0.5440211

Let us add more points...

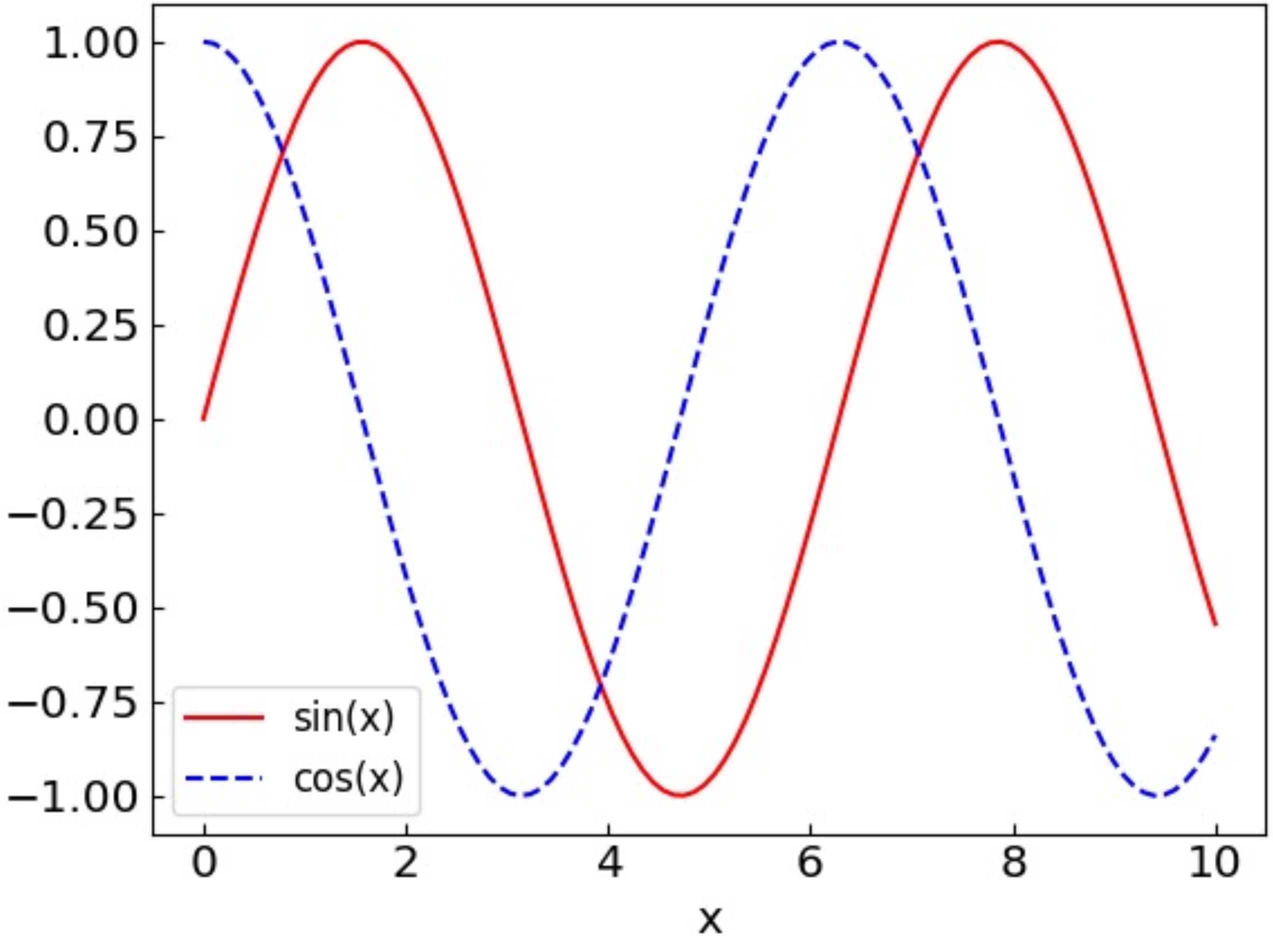
Putting it on a graph



x	sin(x)
0.	0.
0.1	0.09983342
0.2	0.1986693
0.3	0.2955202
0.4	0.3894183
0.5	0.4794255
0.6	0.5646425
0.7	0.6442177
0.8	0.7173561
0.9	0.7833269
1.	0.841471
	⋮
9.9	-0.4575359
10.	-0.5440211

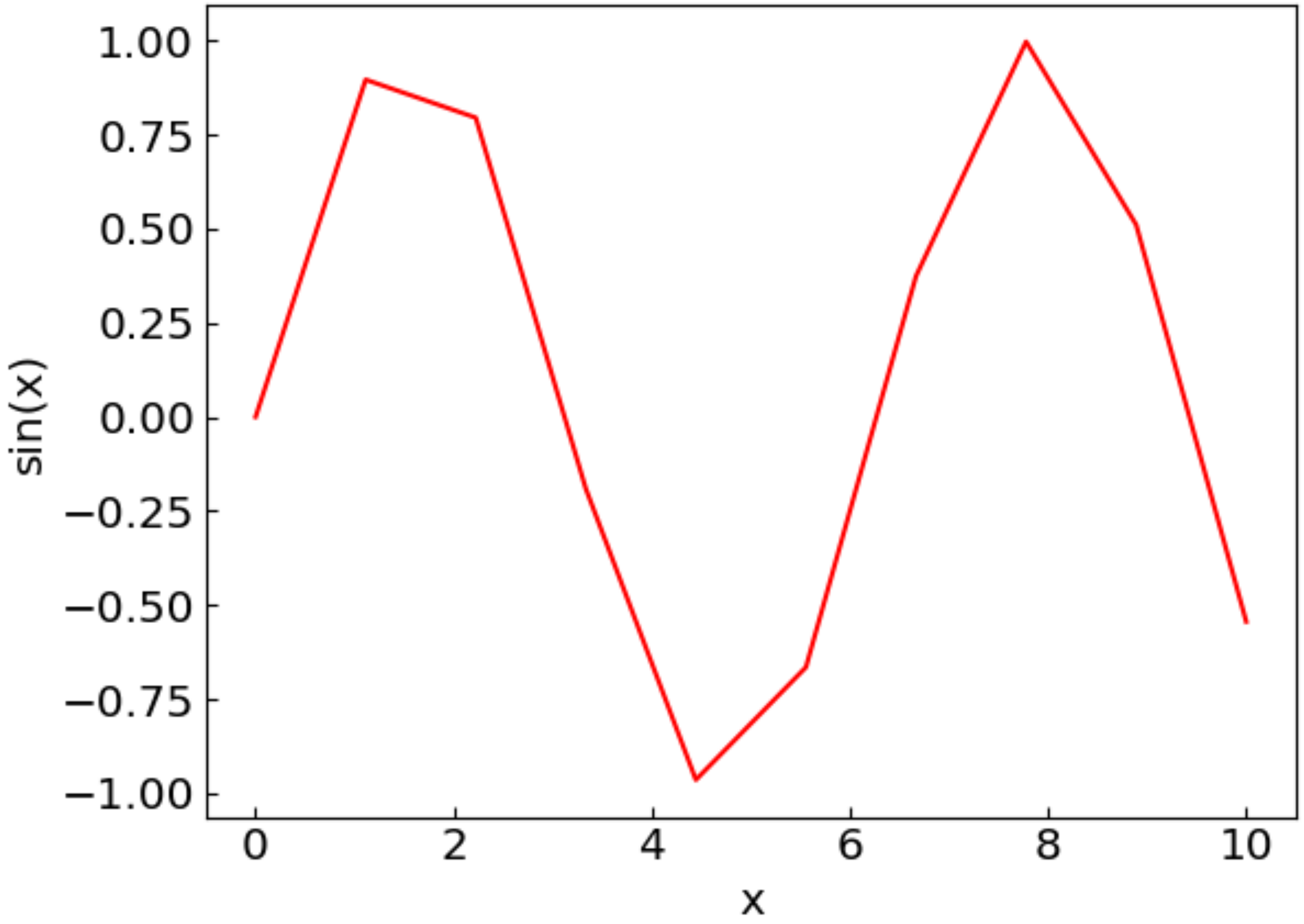
Now we have enough points to join them by a smooth line

Plot multiple lines to compare functions, profiles, etc.



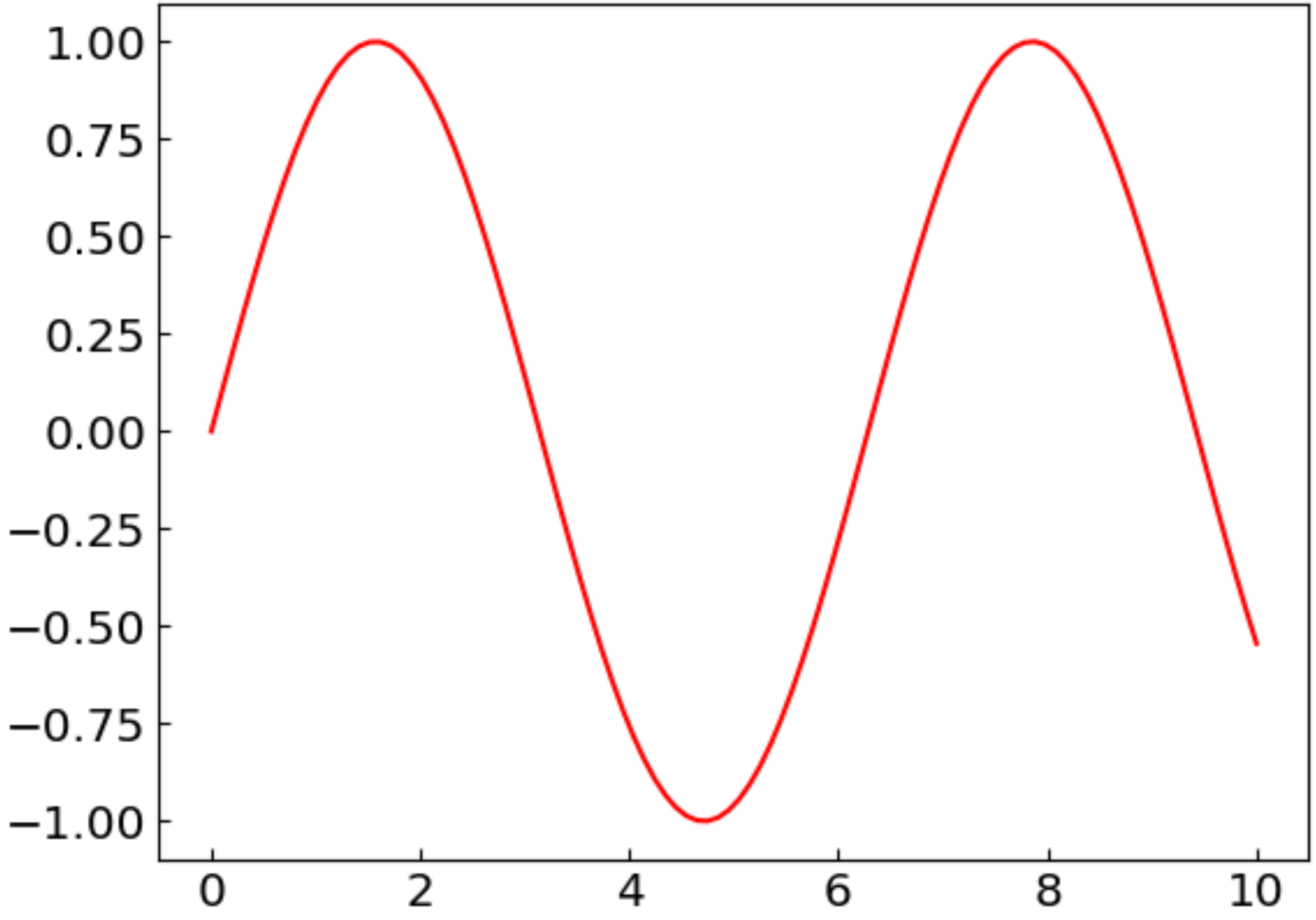
Things to avoid

Insufficient number of data points



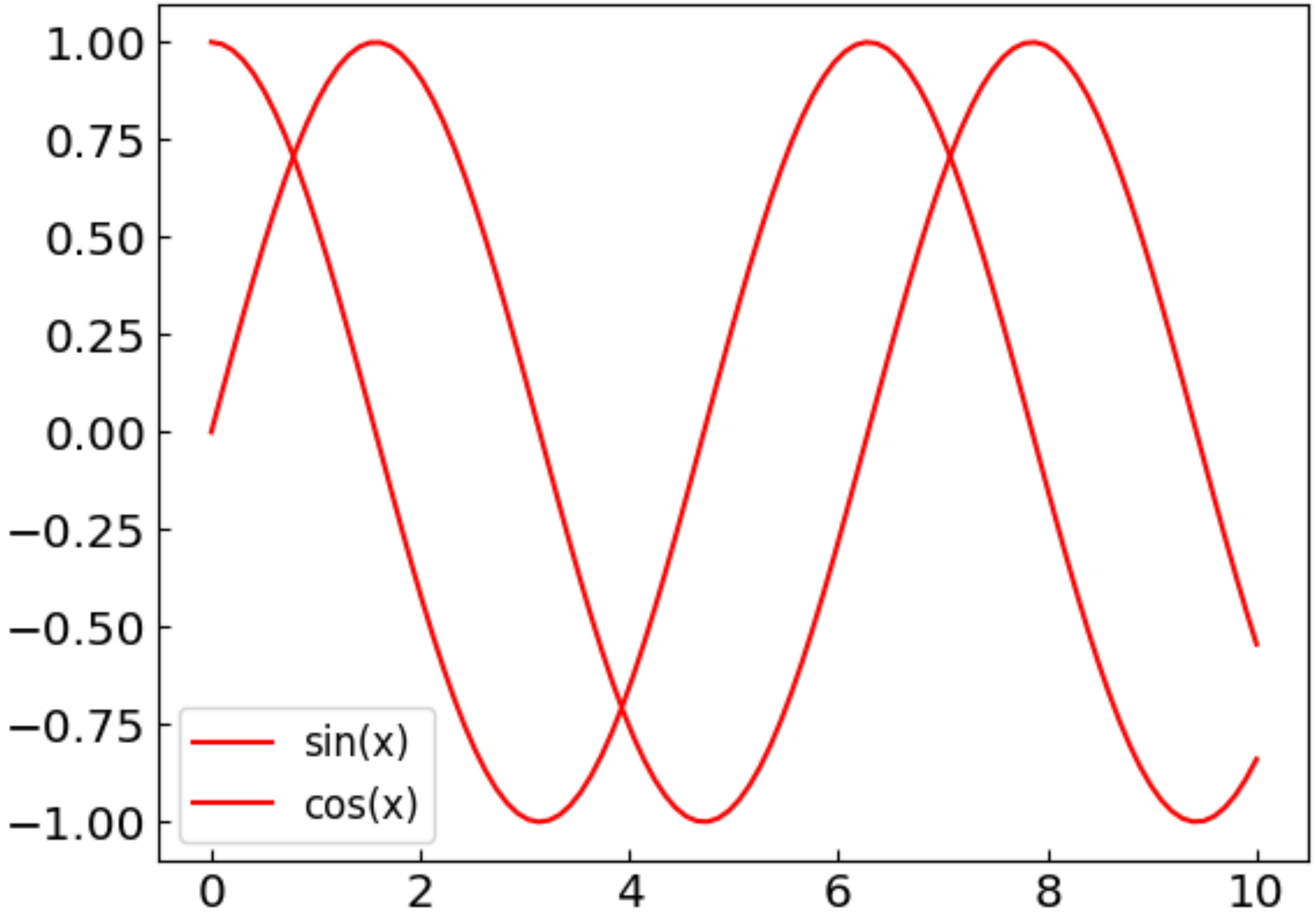
Things to avoid

Unlabeled axes



Things to avoid

Indistinguishable line styles

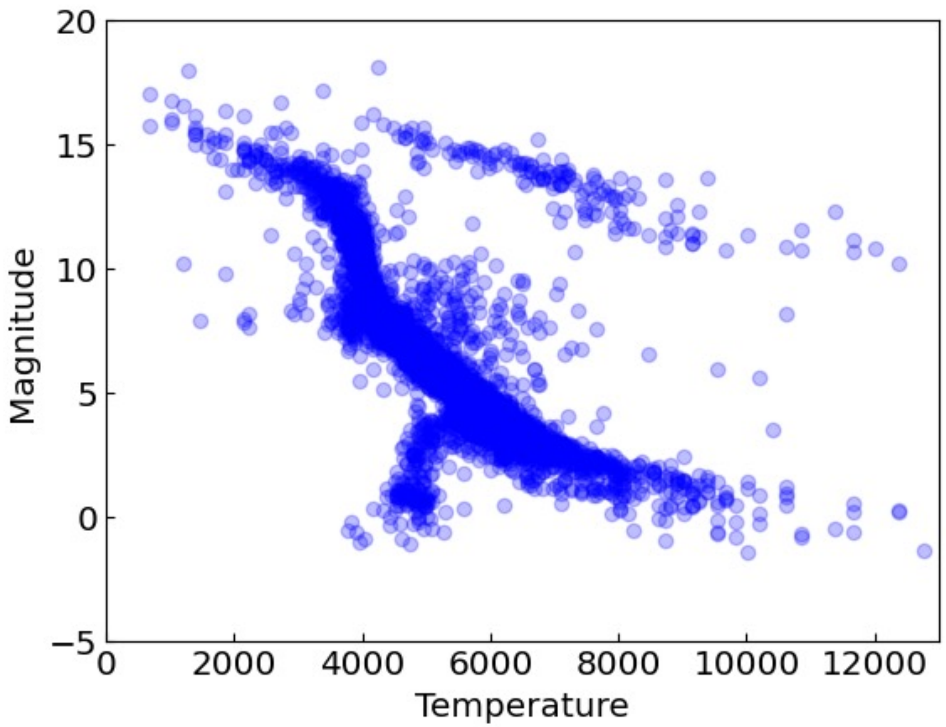


Scatter plots

Not all data points are suitable to be joined by lines

Consider the observations of star surface temperature (= x) and brightness (= y)

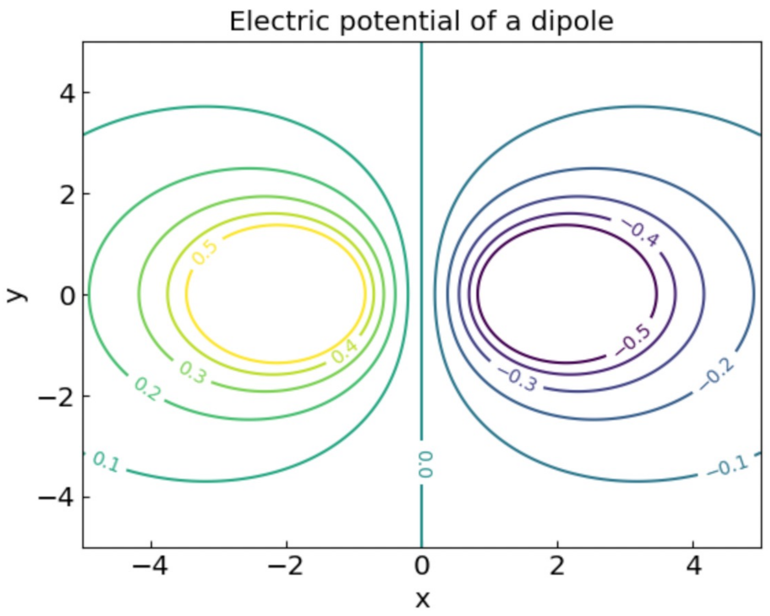
Use scatter plot to study correlation and structures between these features

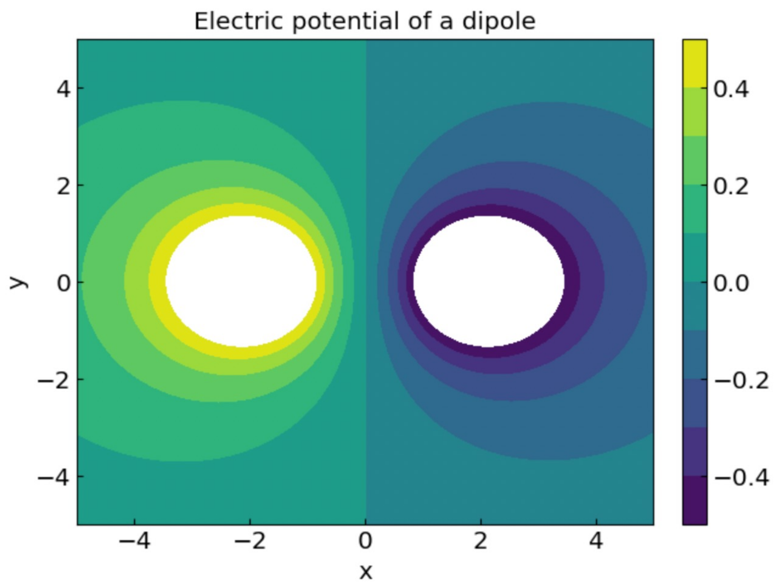


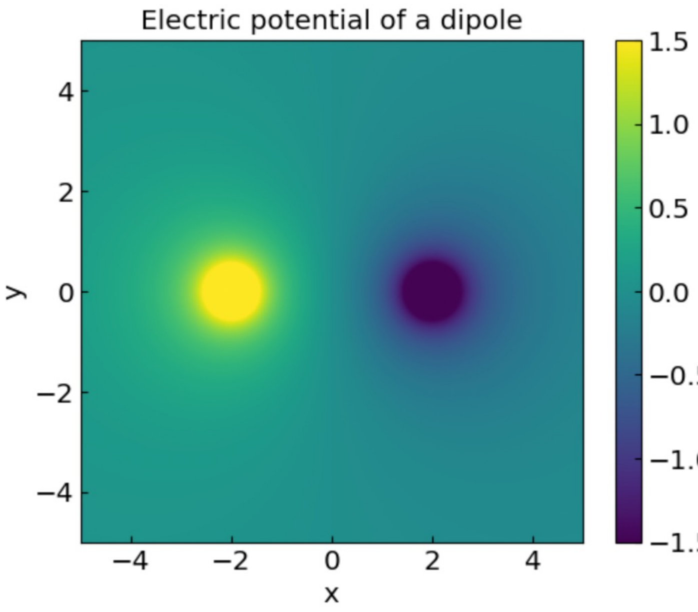

```
683.14508541 15.73
683.14508541 17.01
1012.83217289 15.86
1012.83217289 15.98
1012.83217289 16.73
1195.25068152 10.19
1195.25068152 16.56
1289.42232154 17.99
1384.98930374 15.0
1384.98930374 15.38
1384.98930374 15.39
1384.98930374 15.56
1384.98930374 15.64
1384.98930374 16.15
1481.51656803 7.86
⋮
```

Contour and density plots

For example fields, such as electric potential of a dipole







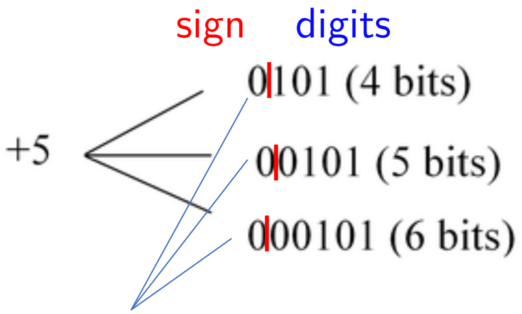
Errors and accuracy

References: [Chapter 4](#) of *Computational Physics* by Mark Newman

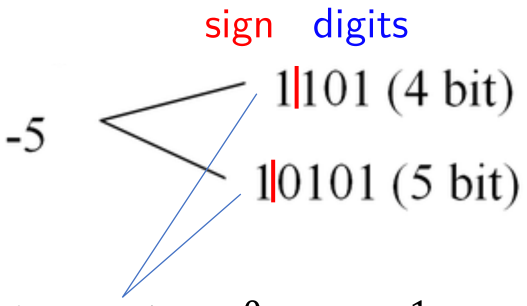
Chapter 1.1 of *Numerical Recipes Third Edition* by W.H. Press et al.

Integer representation

Numbers on a computer are represented by bits – the sequences of 0s and 1s



$$+5 = +(1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2)$$



$$-5 = -(1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2)$$

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 $\sim -10^{18}$ (-2^{63}) to $+10^{18}$ (2^{63})

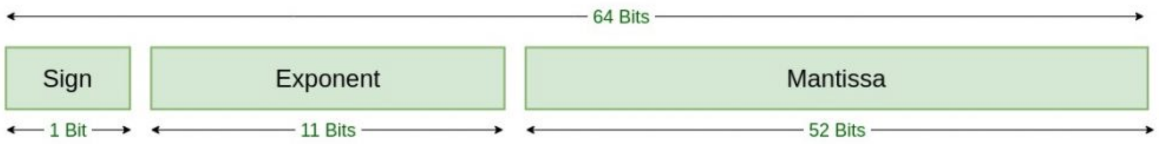
Python supports natively larger numbers but calculations can become slow

In C++ it is important to avoid under/over-flow

Floating-point number representation

Floating-point, or real, numbers are represented by a bit sequence as well, which are separated into:

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



Double Precision
IEEE 754 Floating-Point Standard

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

e.g. $-2195.67 = -2.19567 \times 10^{3}$

Main consequence: Floating-point numbers are **not exact!**

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

	32-bit float (single precision)	64-bit float (double precision)
Bits: (sign-exponent-mantissa)	1-8-23	1-11-52
Significant digits:	~7 decimal digits	~16 decimal digits
Range:	~ -10^{38} to 10^{38}	~ -10^{308} to 10^{308}

Floating-point number representation

When you write

$$x = 1.$$

What it means

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

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
```

You can do instead

```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
```

Error accumulation

$x = 1. + \varepsilon_M, \qquad \varepsilon_M \sim 10^{-16} \qquad$ 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
- In some cases σ_y can become “large” even in a single operation

Two large numbers with a small difference

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

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

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
```

Catastrophic loss of precision!

What happened?

significant digits these do not fit

y = 1.00000000000014 142135623730951 …

x = 1.00000000000000 000000000000000 …

Quadratic equation

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

Symbolically, the roots are:

$\sqrt{b^2 - 4ac}$ is very close to b

$$x_{1,2} = \frac{-b \pm \boxed{\sqrt{b^2 - 4ac}}}{2a}$$

Let us calculate the roots for $a = 10^{-4}, b = 10^{4}, c = 10^{-4}$

$|ac| << b^2$

```
a = 1.e-4
b = 1.e4
c = 1.e-4

x1 = (-b + sqrt(b*b - 4.*a*c)) / (2.*a)
x2 = (-b - sqrt(b*b - 4.*a*c)) / (2.*a)

print("x1 = ", x1)
print("x2 = ", x2)
```

```
x1 =  -9.094947017729282e-09
x2 =  -100000000.0
```

x_2 looks ok but x_1 seems off(?)

Quadratic equation

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

Standard form:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Alternative form:

$$x_{1,2} = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}$$

Using the alternative form

```
x1 = 2*c / (-b - sqrt(b*b-4.*a*c))
x2 = 2*c / (-b + sqrt(b*b-4.*a*c))

print("x1 = ", x1)
print("x2 = ", x2)
```

```
x1 =  -1e-08
x2 =  -109951162.7776
```

x_1 is fixed but now x_2 is off

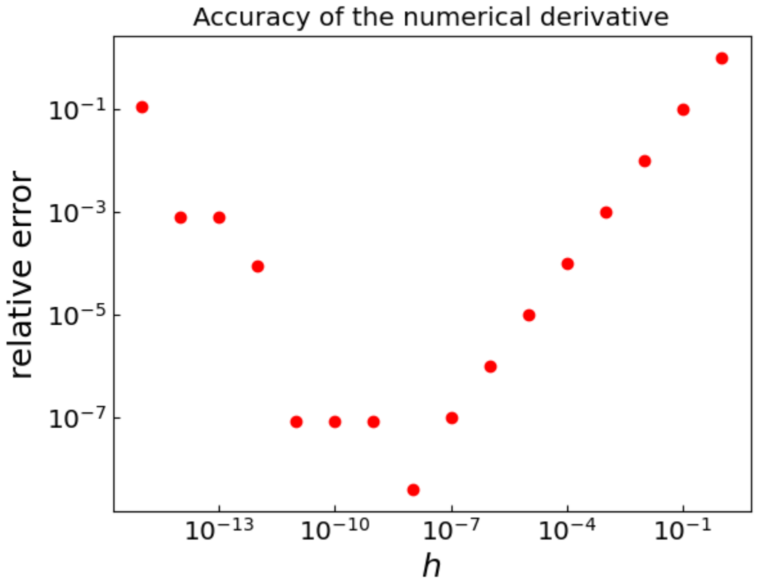
Solution: Make a judicious choice between standard and alternative form for each root separately, such that subtraction of two similar number is avoided

Other common situations

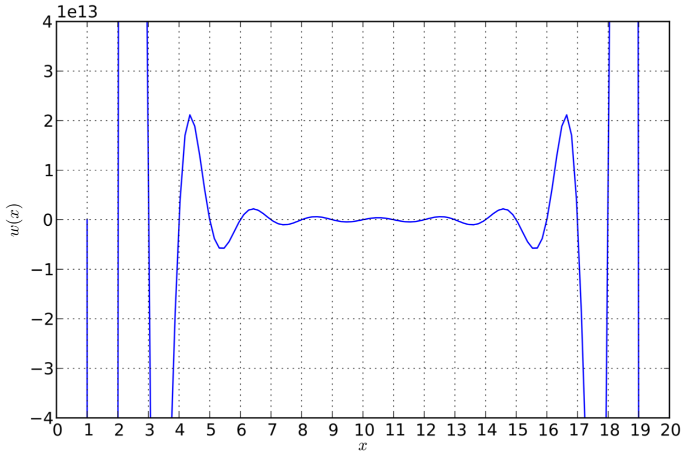
- Simple numerical derivative (see the sample code)

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

Sometimes a small h is too small



- Roots of high-degree polynomials



Advanced topic: **Kahan summation**

final project idea(?)