PHOT 110: Introduction to programming

Lecture 07: supporting materials

Michaël Barbier, Spring semester (2023-2024)

Visualizing functions or numerical data with line plots

Installing required Python packages

The Matplotlib and Numpy packages need to be installed

- Matplotlib: provides functions for plotting data
- Numpy: fast numerical functions (vectors, matrices, …)
- tornado: a web server currently used for showing plots

To use package functions in a script we `import` them:

```python
# Import statements
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
matplotlib.use("WebAgg")  # requires the "tornado" package
```

Currently there is a small issue with using Matplotlib on the computers in the computer lab. There is a missing library (tkinter) on Ubuntu, which is normally used by Matplotlib to show a window on the screen (used as "backend"). As a temporary solution we will use another available backend: "WebAgg".

(1) Install the "tornado" package (a web-server), and select the "WebAgg" backend option in your script, similar to above import example:

```python
import matplotlib
matplotlib.use("WebAgg")
```

(2) Further, to have all plots of the script within one browser tab we will use the command "plt.show()" only at the end, after defining all plots.

(3) To plot a figure, you can use the following:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(xs, ys)  # xs and ys are lists or arrays of coordinates
```

This is slightly different than the `plt.plot()` command we used before, and allows us to plot different plots within one browser-tab. This is more convenient when using the "WebAgg" backend.

Making a plot

Plotting a function is often done by approximating the function by piece-wise straight lines. For this we divide the interval that we want to plot in small pieces and generate (x, y)-coordinates. Suppose we want to plot the function $y = x^2$ over interval $x \in [A, B]$, then first we would create the x-coordinates:

$$x_i = A + i \cdot \frac{B - A}{N} = A + i \cdot \delta x \quad \text{with } i \in \{0, 1, .., N\}$$

with $\delta x = (B - A)/N$ the distance between the x-coordinates.

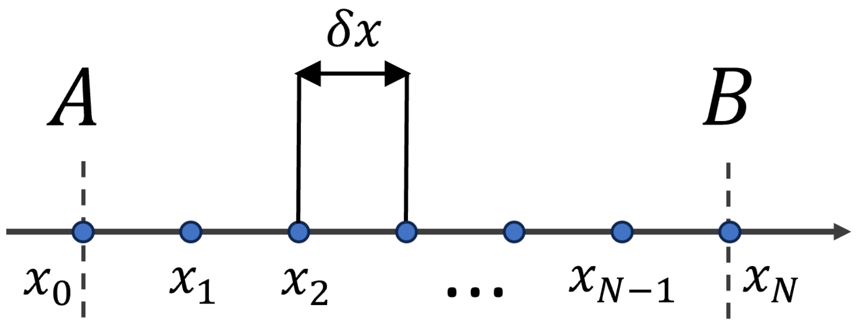
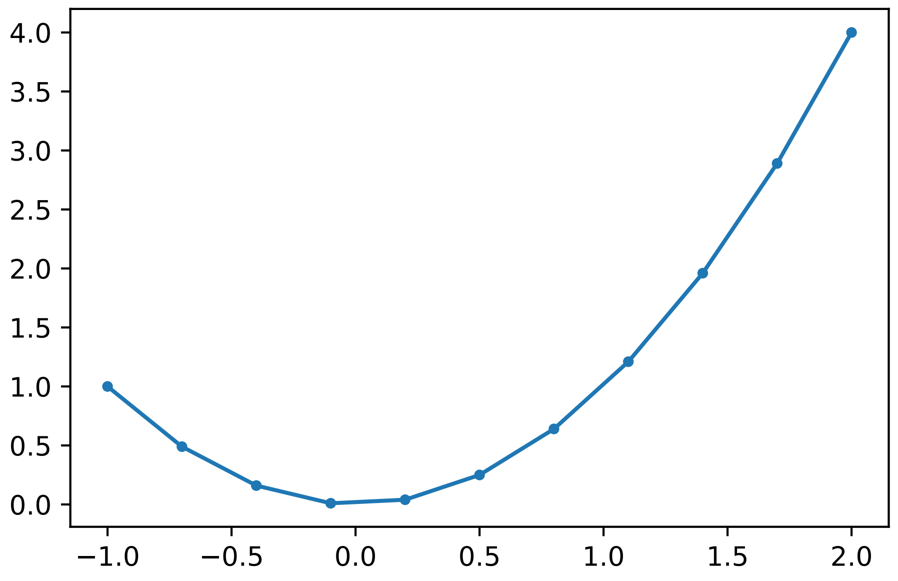


Figure 1: Interval divided in N pieces, i.e. a regularly spaced interval.

Afterwards we calculate the y-coordinates from the x-coordinates: $y_i = x_i^2$. Now that we have both coordinates we can plot the points with a line plot, which will connect the points with line pieces. The following code plots the curve $y = x^2$ within interval $[-1, 2]$

```python
# Import statements
import numpy as np
import matplotlib.pyplot as plt

# Generating the x-coordinates with a list comprehension
N = 10
A = -1; B = 2
xs = [ A + i*(B-A)/N for i in range(N+1)]

# Calculating the y coordinates from the x coordinates
ys = [ x**2 for x in xs]

# Creating a Matplotlib figure
fig, ax = plt.subplots()
# Plotting the coordinates
ax.plot(xs, ys, marker='.')
```



Within this example we used a low number of points, $N = 10$, which is too small to obtain a smooth realistic curve. Normally we plot curves with $N = 100$ or larger. I added dots on the curve (provided by the option `marker='.'` in the plot command) to indicate where the points are.

Different ways to generate the x-values

To create the $N+1$ coordinates x_i in interval $[A, B]$ there are various methods available. First I will list the methods not using the Numpy package:

(1) Using a `while` loop (not optimal when you know the number of divisions):

```
N = 10; A = -1; B = 2;
xs = []
i = 0
while i <= N:
  x = A + i*(B-A)/N
  xs.append(x)
  i = i + 1
```

(2) Using a `for` loop and a `range`:

```
N = 10; A = -1; B = 2;
xs = []
for i in range(N+1):
  x = A + i*(B-A)/N
  xs.append(x)
```

(3) Using list comprehension:

```
N = 10; A = -1; B = 2;
xs = [ A + i*(B-A)/N for i in range(N+1)]
```

Different ways to generate the x-values: with Numpy

Numpy is designed for numerical computations so it has built-in functions for the generation of coordinates within intervals (see also https://numpy.org/doc/stable/user/how-to-partition.html#how-to-partition). Numpy functions `arange()` and `linspace()` create regularly spaced coordinates within 1D intervals:

```
import numpy as np
# Numpy: suppress scientific notation for small numbers and use 3 digits
np.set_printoptions(precision=3, suppress=True)

N = 10; A = -1; B = 2; dx = 0.25
x1 = np.arange(A, B, dx)    # x1 = points in interval [A, B[ with step dx
print(f"x1 = {x1}")
```

```
x2 = np.linspace(A, B, N+1)  # x2 = N points in interval [A, B]
print(f"x2 = {x2}")
```

```
x1 = [-1.   -0.75 -0.5  -0.25  0.    0.25  0.5   0.75  1.    1.25  1.5   1.75]
x2 = [-1.  -0.7 -0.4 -0.1  0.2  0.5  0.8  1.1  1.4  1.7  2. ]
```

The created coordinates are in Numpy arrays (not lists). **Numpy arrays are much faster** to access than lists. But lists can change their size easily and contain mixed object types.

Numpy also provides operations and functions on Numpy arrays. We can use this to generate for example the y-coordinates:

```
# Numpy arrays provide element-wise operations
y = x1**2   #  y contains squared array elements
print(f"y = {y}")
z = 5 * x1 + 3   #  z contains elements of x multiplied by 5 and increased with 3
print(f"z = {z}")
```

```
y = [1.    0.562 0.25  0.062 0.    0.062 0.25  0.562 1.    1.562 2.25  3.062]
z = [-2.   -0.75  0.5   1.75  3.    4.25  5.5   6.75  8.    9.25 10.5  11.75]
```

When to use which method ?

It seems that Numpy is much easier for creating **regularly spaced points within an interval**, especially when using the values afterwards within a calculation, so why would we use anything else? There are some details to it though, I tried to summarize the PRO's and CON's in the following table:

type	PRO's	CON's	Use-case
while loop	versatile, only option when number of iterations is unknown	cumbersome	Unknown number of iterations
for loop	Lower memory footprint if used with `range()`, lazy evaluation	Cumbersome	large number of iterations, performing a direct action each iteration
list comprehension	Clear, easy	Less convenient than Numpy `arange()`	When using Numpy is not desired/possible

type	PRO's	CON's	Use-case
Numpy arange	Easiest method for fixed step, Numpy provides array operations and functions	End of interval is exclusive, no lazy evaluation, slower than Python built-in `Array`	Regularly spaced intervals (fixed step) for computational purposes
Numpy linspace	Easiest method for fixed number of points, Numpy provides array operations and functions	No lazy evaluation, slower than Python built-in `Array`	Regularly spaced interval (fixed number of points) for computational purposes

Basically, if you need to create a regularly spaced interval:

- Use a `while` loop when you don't know the number of iterations up front.
- Use a `for` loop for huge number of iterations which cannot fit into memory, so that the lazy evaluation of `range` can help you there. In this case you would not create the actual list, but take an action at each iteration (e.g. save the result to a file).
- Use list comprehension if you don't want to use Numpy (or it is not available)
- Use Numpy's `arange` or `linspace` for numerical computations and plotting.

Note that this comparison is specifically for regularly spaced intervals. When iterating over lists of objects, using a `for` loop or list comprehension is more appropriate.

Exercises:

EX 1: x^2 in $[-2, 2]$ interval using a `for` loop to generate coordinates.

EX 2: Plot the same curve but use list comprehension to generate the coordinates

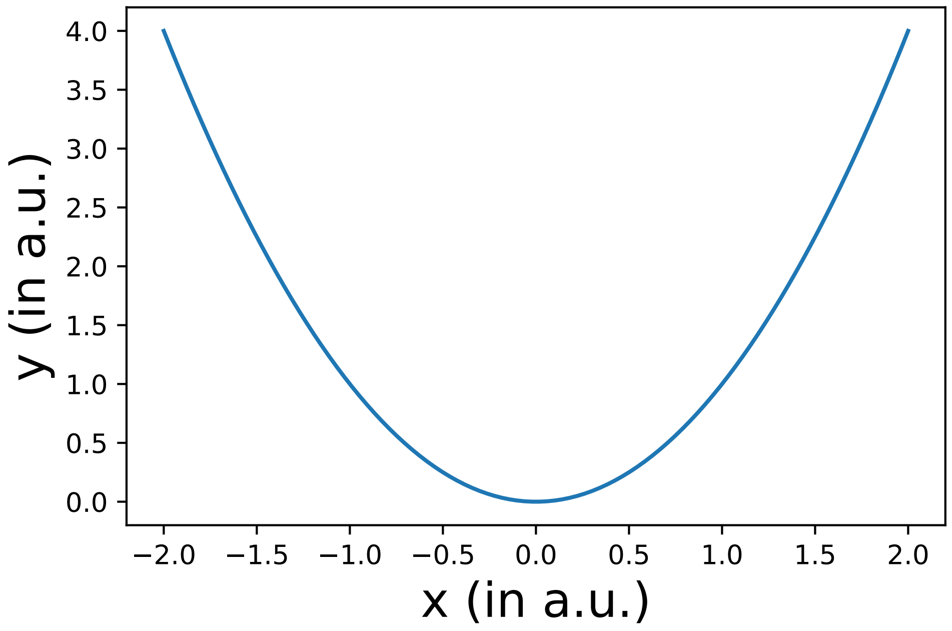
EX 3: Plot the same curve but use arrays and Numpy `np.linspace()`

EX 4: plot a cosine curve $[0, 2\pi]$ interval using arrays and Numpy `np.linspace()` and `np.cos()` functions

```python
# Syntax for loop
for i in range(start, stop, step):
  <statements> ...

# Plotting:
plt.plot(xs, ys)
```

```
ax = plt.gca()
ax.set_xlabel("x (in m)")
plt.show()

# List comprehension
<list> = [x**2 for x in <sequence>]

# Numpy array xs
xs = np.linspace(start, stop, n_el)
```

Output plot of exercises 1 - 3



Exercise 5: Lissajoux figures

Lissajoux figures: sinusoidal signals in x and y.

$$x = \sin(f_x t + \delta\phi)$$
$$y = \sin(f_y t)$$

- Traversed coordinates in time `t` form shapes
- Signal frequencies `fx` and `fy`

- Phase difference `dphi`

Lissajoux figures are typically encountered on an oscilloscope (see the figure below). In an oscilloscope electric signals are connected to the vertical and horizontal

```
# Use following parameters
fx = 3; fy = fx - 1; dphi = pi/2 # or dphi = pi/4

# Hint: Use np.linspace(), np.pi, np.sin()
#       take time in interval [0, 2*pi], e.g.:
#           t = np.linspace(0, 2*np.pi, 1000)
```

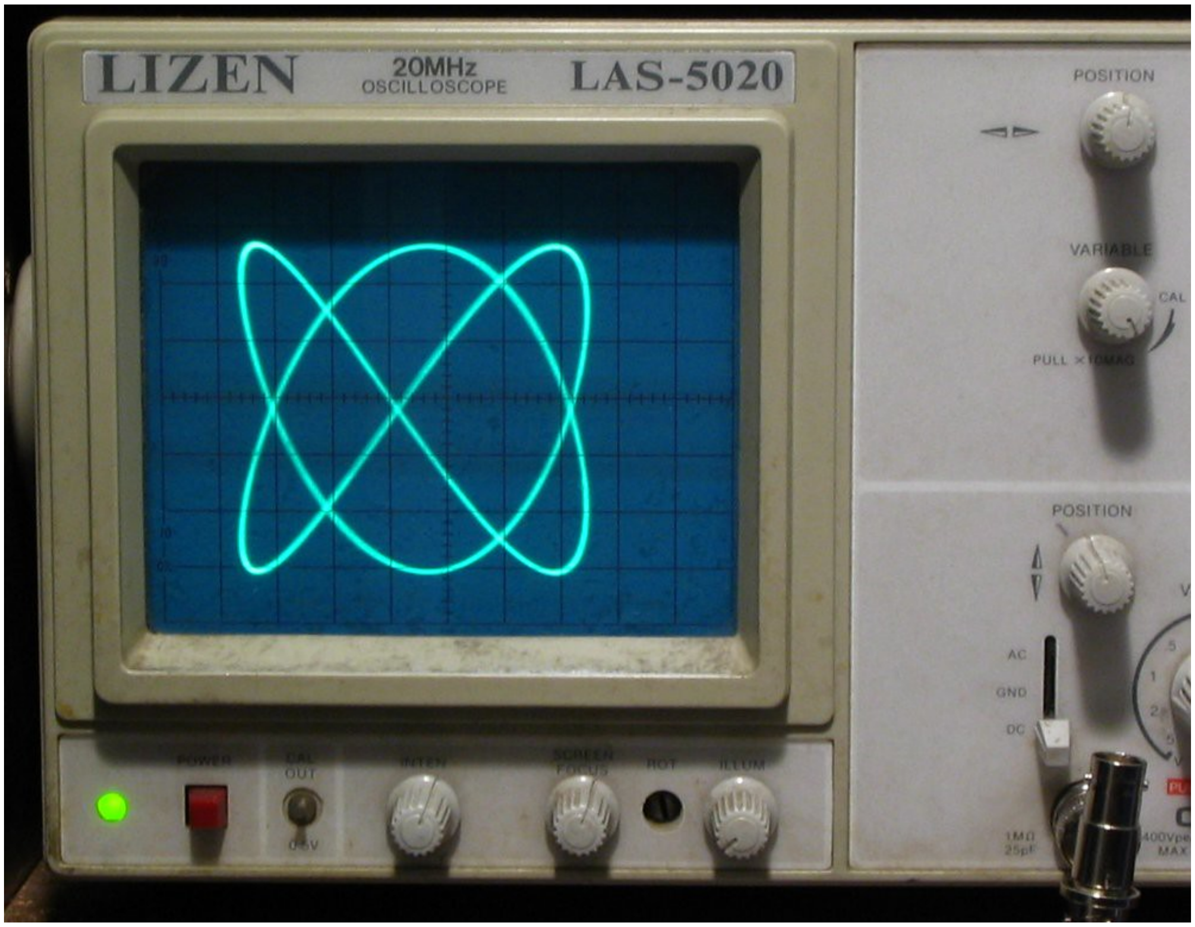
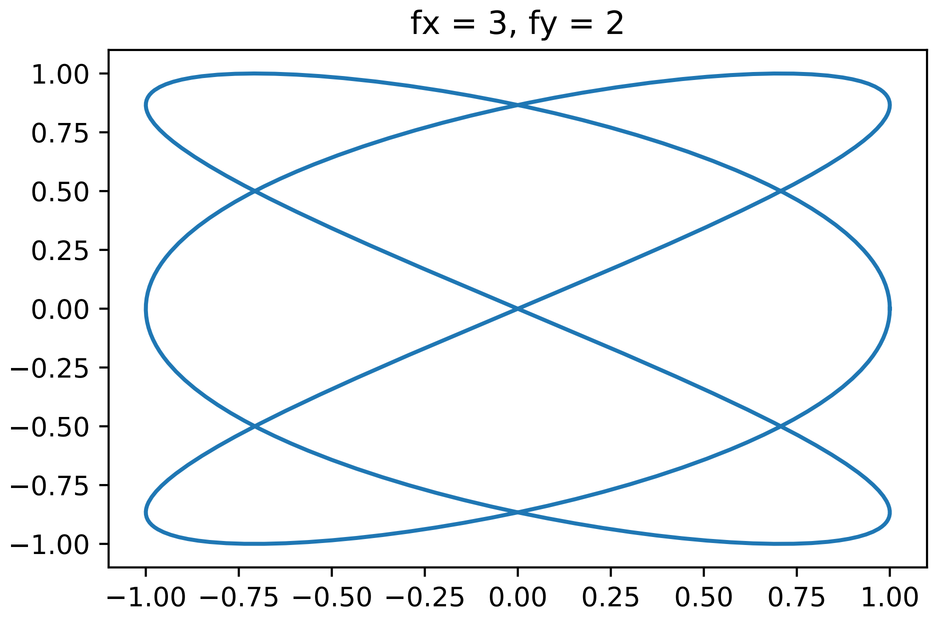


Figure 2: An oscilloscope allows to see the correlation between two electric signals, by plotting one signal horizontally and the second vertically.

Output plot example



Exercise 6: Compute an integral numerically

Numerically integrate the following integral

$$\int_0^2 \mathrm{dx}\,(x^3 - x/3)$$

(1) As a first approximation try to use the rectangular approximation (Riemann sum). Therefore, divide the interval $[0, 2]$ in N parts and approximate the area surrounding each point by a thin rectangle. The sum of all rectangle areas then approximates the total area under the curve (see the figure below), i.e. the integral. For a general interval $[A, B]$ and function $y = f(x)$ we can numerically calculate the integral using:

$$\int_A^B \mathrm{dx}\, f(x) \approx \sum_{i=1}^{N} f(x_i)\delta x$$

With $x_i = A + (i - 1/2) \times \delta x$, and $\delta x = (B - A)/N$. This sum is also called **Riemann sum**. In principle, we should take the limit $N \to \infty$ to obtain the analytic result or actual Riemann integral, but as an approximation we take N a finite (large number).

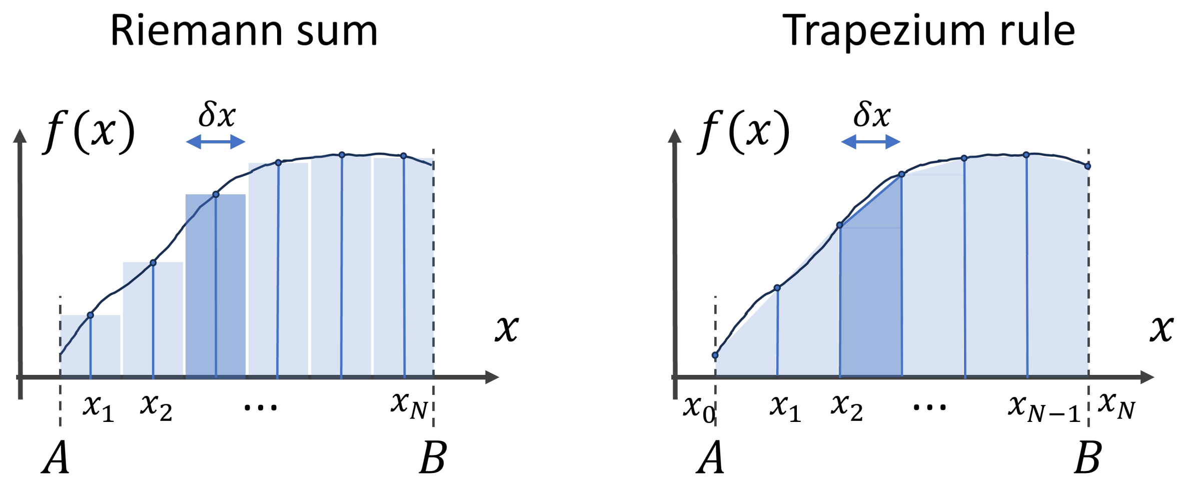


Figure 3: Numerical integration of a definite integral over interval $[A, B]$ interval. The **left** panel illustrates the Riemann sum, where the area under the curve is approximated by rectangles. On the **right** the Trapezium rule is illustrated where the area is approximated by trapeziums instead of rectangles, increasing the accuracy.

(2) Compare the result with the analytic result:

$$[x^4/4 - x^2/6\,]_0^2 = 10/3$$

(3) Then use the **trapezium rule**, see the figure above, in which instead of rectangles we use trapeziums, following the curve more accurately. The area of a trapezium with two vertical sides and one horizontal bottom side is given the width times the average length of the vertical sides.

$$\int_A^B \mathrm{d}x\, f(x) \approx \sum_{i=1}^{N} \frac{f(x_i) + f(x_{i-1})}{2} \delta x$$

With $x_i = A + i \times \delta x$, and $\delta x = (B - A)/N$. This sum should more accurately reflect the area under the curve (the integral) than the Riemann sum.

(4) This formula is used so often that Numpy has a function for it: `np.trapz(y, x)`. Compare your result with the result of this function.

Exercise 7: Compute derivatives numerically

The derivative of a function in a point can be numerically approximated by finite difference methods. The following formula gives us the **forward finite difference**:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} \approx \frac{f(x_{i+1}) - f(x_i)}{\delta x}$$

If we would take the limit of $\delta x \to 0$ then we would obtain the actual derivative. Here however we will approximate the derivative with a finite-sized δx. We we want to calculate the derivative over an interval we will as we did for the integral divide the interval in N pieces and set $\delta x = (B - A)/N$. Thereby we can plot the derivative over the whole interval $[A, B]$ (except of the last point at B, for which we can't compute the forward derivative). Similarly, the **backward finite difference** is defined as:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} \approx \frac{f(x_i) - f(x_{i-1})}{\delta x}$$

And the mean of the two becomes the **central difference**:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} \approx \frac{1}{2}\left(\frac{f(x_{i+1}) - f(x_i)}{\delta x} + \frac{f(x_i) - f(x_{i-1})}{\delta x}\right) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\delta x}$$

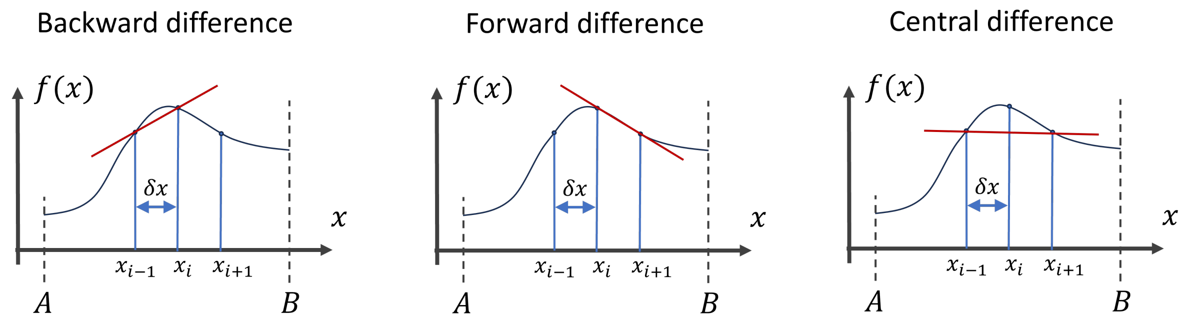


Figure 4: Numerical derivatives. The central difference allows the most accurate derivative and can be considered as the mean derivative of the forward and the backward difference approximation.

(1) Numerically compute and plot the forward derivative of $f(x) = \sin(5\pi x)/(1 + x^2)$ within interval $[-3, 3]$. Ignore the issue at the last point.

(2) Do the same for the central difference, do you find any difference for small values of N?