Technische Universität Berlin Winter term 2025/26
Institute of Mathematics
Prof. Dr. T. Breiten, Prof. Dr. S. May, Dr. Juliette Dubois

Numerical Mathematics II for Engineers
Tutorial 0

Topics : Reminder on Python and NumPy, Finite difference quotients, solving the 1D Poisson equation with finite differences.

Discussion in the tutorials of the week 20–26 October 2025

Disclaimer: This exercise sheet looks very lengthly, but the exercises are only a revision of things that should already be known; they should therefore not take too long to complete.

Exercise 0.1: Introduction to Numpy and Matplotlib

This exercise is an introduction to the NumPy package, inspired from the official quickstart guide[1]. NumPy is a package for the multidimensional array object `ndarray`. It allows fast operations on large arrays, and has become the basis for scientific computing in Python. Like classical Python `list`, NumPy `array` are tables of elements, but they have the following characteristics:

- arrays have a fixed, predetermined size,
- all elements of an array must have the same type,
- dimensions are called "axes".

NumPy enables **vectorization**: explicit looping is generally avoided in the code. Example: assume two arrays `u`, `v` of dimension $(N, 1)$ are given (i.e., essentially vectors). We want to know their sum. In traditional programming languages, one would do that by using loops

```
w = [0 for k in range(N)]
for k in range(N):
    w[k] = u[k] + v[k]
```

In a vectorized form, this would just be

```
w = u + v
```

which is faster. It also makes it easier to understand what the code is doing.

Vectorization also involves the usage of built-in functions when possible. As a second example, we compare the runtime to compute the scalar product of two vectors `u,v` of dimension N, either using loops or using vectorization and built-in NumPy functions.

```
## Using loops
for i in range(N):
    w[i] = u[i]*v[i]

## Using the built-in NumPy functions
w = np.dot(u,v)
```

[1] https://numpy.org/doc/stable/user/quickstart.html

N	Time with loop (s)	Time with NumPy (s)
10	3.34e-06	1.74e-05
1000	6.29e-05	5.48e-06
1e6	7.21e-02	7.16e-03
5e6	3.51e-01	1.82e-02
1e7	6.91e-01	3.75e-02

Table 1: Runtime for computing the scalar product of two vectors of size N

The runtimes obtained for increasing values of N are presented in Table 1. For small N, the times are comparables, but for large N, the NumPy implementation is at least ten times faster.

Built-in and vectorized operations are optimized and pre-compiled in C, making them very fast and adapted for scientific computing. **Whenever possible, use vectorized code!**

Now to the exercise: install numpy (preferably in a virtual environment), open an interactive console and do the following tasks[2]:

(a) Run `import numpy as np`.

(b) Define a,b,c as follows:

```
a = np.array([1, 0, 0])
b = np.array([[1., 0., 0.],[0., 1., 2.]])
c = np.array([[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]],
              [[12, 13, 14, 15], [16, 17, 18, 19],
                                   [20, 21, 22, 23]]])
```

For each array, check the following properties: `ndim`, `nshape`, `size`, `dtype`. Example: `a.ndim`

(c) Arrays can be created in various ways. In question 1.1(a), you converted lists into arrays, but the built-in functions are more suited. Try the following functions:

```
np.zeros((3, 4))
np.ones((2, 3, 4))
np.arange(10, 30, 5)
np.linspace(0, 2, 9)
```

Once an array is created, its shape can be changed. Try the following operations:

```
a = np.ones((3, 4))
a.flatten()
a.reshape(6, 2)
a.T
```

Note that those operations return a new array, they do not change `a`. Now, use `arange` and `reshape` to create the array `c` from Question 1.1(b) in one line of code.

(d) Most operations on arrays apply elementwise. Try the following manipulations:

```
a = np.array([20, 30, 40, 50])
b = np.arange(4)
c = a - b
```

[2]It is unfortunately not possible to provide support from our side for the installation of Python, NumPy and Matplotlib. Using tools like ChatGPT though this should hopefully be straightforward.

```
b**2
a < 35
a *= 3

## Classical functions are available in NumPy
10 * np.sin(a)
np.exp(B)
np.sqrt(B)
```

Finally, note the difference between matrix dot product and elementwise product:

```
A = np.array([[1, 1],
              [0, 1]])
B = np.array([[2, 0],
              [3, 4]])
A * B     # elementwise product
A @ B     # matrix product
A.dot(B)  # another matrix product
```

(e) Like Python lists, arrays can be indexed, sliced and iterated over. Indices for different axes are separated by a comma. Try the following operations:

```
def f(x, y):
    return 10 * x + y
b = np.fromfunction(f, (5, 4), dtype=int)
b[2, 3]
b[0:5, 1]  # each row in the second column of b
b[:, 1]    # equivalent to the previous example
b[1:3, :]  # each column in the second and third row of b
```

(f) To visualize our results, we will use the `matplotlib` package[3]. Install matplotlib (again, preferably in a virtual environment) and run the following code:

```
import matplotlib.pyplot as plt
x = np.linspace(0,2*np.pi)
plt.plot(x, np.sin(x), "x-")
plt.xlabel("$x$")             # LaTeX can be used for plotting
plt.ylabel("$\\sin(x)$")     # But be careful with the "\" command
plt.show()
```

There are many options to customize plots, you will see some examples in the templates of this homework.

For more information on NumPy and Matplotlib, look at their officiel documentation: https://numpy.org/doc/stable/index.html for NumPy and https://matplotlib.org/stable/ for Matplotlib.

Exercise 0.2: Revision on Landau notation

This exercise is a review on the Landau, or ”big O”, notation.

- The following polynomial is given:

$$f(x) = 2x^2 + 3x^3 + 8x^5.$$

Give the behavior of $f(x) = \mathcal{O}(x^?)$ for the limit of $x \to 0$ and for the limit of $x \to \infty$.

[3] https://matplotlib.org/

- The following polynomial is given:

$$f(x) = 2x^2 + 3x^3 + 8x^5.$$

Give the behavior of $f(x) = 2x^2 + \mathcal{O}(x^?)$ for the limit of $x \to 0$.

Exercise 0.3: Revision on convergence curves and tables

This exercise is a reminder on convergence orders, and on how to compute and display them. For the programming questions, the template `convergence_template.py` is provided. As a model problem, we will consider two different quadrature rules to integrate a function $f \in C^4([a,b])$ over the interval $[a,b] \subset \mathbb{R}$, i.e., to approximately compute

$$I = \int_a^b f(x)\,dx.$$

For the numerical integration, the interval $[a,b]$ is discretized into n subintervals $[x_j, x_{j+1}]$, with $x_j = a + jh$ and $h = (b-a)/n$. We will compare the following quadrature rules:

- the **composite Trapezoidal rule** (on each interval, the two end points are used)

$$\int_a^b f(x)\mathrm{d}x \approx T(h) = h\left(\frac{f(a)+f(b)}{2} + \sum_{j=1}^{n-1} f(x_j)\right),$$

- the **composite Simpson's rule** for an even number of subdivision n (on each interval, the two end points and the midpoint are used)

$$\int_a^b f(x)\mathrm{d}x \approx S(h) = \frac{h}{3}\left(f(a) + 4\sum_{j=1}^{n/2} f(x_{2i-1}) + 2\sum_{j=1}^{n/2-1} f(x_{2i}) + f(b)\right).$$

If the function f is sufficiently smooth (which we assume for this exercise), then there holds for the error $E(h)$:

- for the Trapezoidal rule: $E_T(h) = |I - T(h)| = \mathcal{O}(h^2) \quad$ for $h \to 0$;
- for the Simpson's rule $E_S(h) = |I - S(h)| = \mathcal{O}(h^4) \quad$ for $h \to 0$.

That means, if we are already in the asymptotic region, i.e., if h is fine enough, then if we reduce the interval length by a factor of 2, i.e., $h \to \frac{h}{2}$, we expect that

- for the Trapezoidal rule: the error is reduced by a factor $2^2 = 4$,
- for Simpson's rule: the error is reduced by a factor of $2^4 = 16$.

We integrate the function $f(x) = 2 + \cos(x)$ over the interval $[0, \pi/2]$.

(a) Compute the exact integral.

(b) Implement the Trapezoidal rule in the function `approximate_integral` in the file `convergence_template.py`.

(c) For the Simpson's rule, we use the `simpson` function from the `scipy.integrate` package. Compute the relative error between the approximated integral and the exact integral for $n \in \{2, 4, 8, 16, 32\}$ for both quadrature rules. Plot both errors in the same picture using `plot` from the `matplotlib.pyplot` package. Which rule is more accurate? Can you easily read the convergence order on the curve?

(d) To better distinguish between both curves, plot the relative errors in **logarithmic scaling** using `loglog` from the `matplotlib.pyplot` package. Also plot the curves $f(h) = h^2$ and $f(h) = h^4$.

You should get straight lines for the curves h^2 and h^4, explain why. Which convergence order do you observe for the Trapezoidal and for the Simpson's rule?

(e) Using `loglog` plots is very helpful for reading off convergence orders. Another very good alternative is to use error tables to compute the orders. Assuming that we have convergence of the form h^p and we want to determine p, we can use

$$p \approx \log_2 \left(\frac{E(2h)}{E(h)} \right). \tag{1}$$

A derivation of the formula can be found in the appendix A.1. In the code, compute the table of order from the errors. Since you computed the errors for five different h, you should get four values for the order, for each quadrature rule.

Exercise 0.4: Finite difference quotients

We first recall the definitions and results of the lecture. We consider an open interval $I \subset \mathbb{R}$ and a function $f \in C^4(\bar{I})$. For all $x \in I$ and $h \in \mathbb{R}$ such that $x + h \in I$, the forward difference quotient is defined by

$$D^+ f(x) = \frac{f(x+h) - f(x)}{h} \qquad \text{(FD)}.$$

In the lecture we have proved, using Taylor expansion, that there holds

$$D^+ f(x) = f'(x) + hC_1, \tag{2}$$

with

$$|C_1| \leq \frac{1}{2} \max_{\xi \in [x, x+h]} |f'(\xi)|. \tag{3}$$

The equations (2)-(3) imply that

$$\left| f'(x) - D^+ f(x) \right| = \mathcal{O}(h) \quad (h \to 0),$$

hence, we say that (FD) **converges with first order / is an approximation of first order** to $f'(x)$.

(a) For all $x \in I$ and $h \in \mathbb{R}$ such that $x + h \in I$ and $x - h \in I$, the central difference quotient is defined by

$$D_1^c f(x) = \frac{f(x+h) - f(x-h)}{2h} \qquad \text{(CD)}.$$

Prove the equation from the lecture

$$D_1^c f(x) = f'(x) + h^2 C_2, \tag{4}$$

with

$$|C_2| \leq \frac{1}{6} \max_{\xi \in [x-h, x+h]} |f''(\xi)|. \tag{5}$$

We say that (CD) **converges with second order / is an approximation of second order** to $f'(x)$.

(b) Implement the schemes (FD) and (CD) in the template `FD_template.py`. To check your implementation, apply the scheme (FD) to a linear function, and the scheme (CD) to a quadratic function: in each case, the derivative should be approximated to machine precision.

(c) We choose $f(x) = \exp(x)$ and $x = 1$, and want to test **numerically** the convergence orders of (FD) and (CD). Plot the relative error for (FD) and (CD) in a `loglog` plot (with h on the x-axis and the relative error on the y-axis) for $h = 10^{-k}$ with $k \in \{1, 2, 3, 4, 5\}$. Also plot the functions h and h^2 for comparison. Does the result meet your expectation?

(d) Now do the same for $h = 10^{-k}$ with $k = 1, 2, \ldots, 11, 12$. What do you observe now? Can you explain the phenomenom?

A Appendix

A.1 Computing the convergence order

Derivation of the formula (1): assume that the error has the form

$$E(h) = Ch^p + \mathcal{O}(h^{p+1})$$

for some constant C independent of h.

We investigate how the error changes when we divide the step length h by 2. Assuming that the constant C does not change, we get

$$E(2h) = C(2h)^p + \mathcal{O}((2h)^{p+1}) = 2^p Ch^p + \mathcal{O}(h^{p+1})$$

and then

$$\frac{E(2h)}{E(h)} = \frac{2^p Ch^p + \mathcal{O}(h^{p+1})}{Ch^p + \mathcal{O}(h^{p+1})} = \frac{2^p + \mathcal{O}(h)}{1 + \mathcal{O}(h)} = 2^p + \mathcal{O}(h).$$

Taking the logarithm on both sides, we get

$$p \approx \log_2 \left(\frac{E(2h)}{E(h)} \right).$$