CDS: Numerical Methods - Assignment week 1

Solutions to the exercise have to be handed in via Brightspace in form of one or several executable Python scripts (*.py) which run without any errors. The deadline for the submission is **Monday Feb. 10, 13:30**. Feel free to use the Science Gitlab repository to submit your solutions.

1 Rounding and Truncation Error Analysis

Euler's number e can be represented as the infinite series $e = \sum_{n=0}^{\infty} \frac{1}{n!}$. In order to evaluate it in Python we need to truncate the series. Furthermore, we learned that every number representation and floating-point operation introduces a finite error. Thus, let's analyze the truncated series

$$\tilde{e} = \sum_{n=0}^{N} \frac{1}{n!}$$

in more detail.

(a) Calculate $\tilde{e}$ with Python and plot the relative error $\delta = \left|\frac{\tilde{e}-e}{e}\right|$ as a function of N (use a log-scale for the y axis).

(b) Compare the relative errors of δ for different floating point precision as a function of N. To this end, we define each element of the series $e_n = \frac{1}{n!}$ and convert it to double-precision (64 bit) and single-precision (32 bit) floating points by using Numpy's functions `numpy.float64(e_n)` and `numpy.float32(e_n)`, respectively, *before* adding them up.

(c) Compare the relative errors of δ for different rounding accuracies as a function of N using Python's `round(e_n, d)` function to round each e_n element before adding them up. Plot δ vs. N for $d = 1, 2, 3, 4, 5$ (which is the number of digits Python's round() function returns) and add a corresponding legend.

Examples:

```python
import numpy as np
# using float32
a = 0.1234
b = np.float32(a)
# using round
c = round(a, 2)
```

2 Lagrange Polynomial Interpolation

Write your own Lagrange polynomial interpolation routine which calculates

$$P(x) = \sum_{k=0}^{n} f(x_k) L_{n,k}(x)$$

with $k = 0, 1, \ldots, n$. Start with a first function `myLagrange(xk, yk, x)` of the form

```python
def myLagrange(xk, yk, x):
  p = np.zeros(np.size(x), dtype=np.float64)
  ...
  return p
```

which internally calls another function `myLagrangePolynomials(xk, n, k, x)` generating the Lagrange interpolation polynomials

$$L_{n,k}(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i}$$

```python
def myLagrangePolynomials(xk, n, k, x):
  L = np.zeros(np.size(x), dtype=np.float64)
  ...
  return L
```

where xk and yk are arrays of the same size representing the $(x_k, y_k = f(x_k))$ pairs which we like to interpolate and x is a an array of x values.

(a) Use your Lagrange interpolation routine to construct the interpolating polynomial for the pairs $x_k = [2, 3, 4, 5, 6]$ and $y_k = [2, 5, 5, 5, 6]$ and plot it from $x = 2$ to $x = 6$ using x-step-sizes of 0.01.

(b) Make sure your result is identical to the one obtained from Scipy's Lagrange function `scipy.interpolate.lagrange()`. Plot both results in the same figure.

(c) Implement a simple unit test to test your routine using the `pytest` package (see `https://docs.pytest.org/en/latest/`, more details will follow in the computer course).

3 Runge's Phenomenom

Use your own (or Scipy's) Lagrange interpolation routine to interpolate the function

$$f(x) = \frac{1}{1 + 25x^2}$$

between $x = -1$ and $x = +1$

(a) using equidistant $x_i = \frac{2i}{n} - 1$ with $i \in \{0, 1, \ldots, n\}$.

(b) using Chebychev nodes $x_i = \cos\left(\frac{2i-1}{2n}\pi\right)$ with $i \in \{1, \ldots, n\}$.

(c) using n randomly chosen points x_i.

Plot the results and (shortly) discuss their differences.