CDS: Numerical Methods - Assignment week 2

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. 17, 13:30**. Feel free to use the Science Gitlab repository to submit your solutions.

1 Discrete and Fast Fourier Transforms (DFT and FFT)

In the following we will implement a DFT algorithm and, based on that, a FFT algorithm. Our aim is to experience the drastic improvement of computational time in the FFT case.

(a) Implement a Python function `DFT(yk)` which returns the Fourier transform defined by

$$\beta_j = \sum_{k=0}^{N-1} f(x_k) e^{-i\,j\,x_k} = \sum_{k=0}^{N-1} f(x_k) e^{-i\,j\,\frac{2\pi k}{N}},$$

with $x_k = \frac{2\pi k}{N}$, $k = 0, 1, \ldots, N-1$, and $j = 0, 1, \ldots, N-1$ by evaluating the full sum (Tip: Try to write the sum as matrix-vector product and use `numpy.dot()` to evaluate it). Here, `yk` represent the array corresponding to $y_k = f(x_k)$. Please note: This definition is slightly different to the one we introduced in the lecture. Here we follow the notation of Numpy and Scipy.

(b) Make sure your function `DFT(yk)` and Numpy's FFT function (`numpy.fft.fft(yk)`) return the same data by plotting $|\beta_j|$ vs. j for

$$y_k = f(x_k) = e^{20i\,x_k} + e^{40i\,x_k}$$

and

$$y_k = f(x_k) = e^{i\,5x_k^2}$$

using $N = 128$ with routines.

(c) Analyze the evaluation-time scaling of your `DFT(yk)` function with the help of the `timeit` module based on the following example:

```python
import timeit

tOut = timeit.repeat(stmt=lambda: DFT(yk), number=10, repeat=5)
tMean = np.mean(tOut)
```

This example evaluates `DFT(yk)` 5×10 times and returns 5 evaluation times which are saved to `tOut`. Afterwards we calculate the mean value of these 5 repetitions. Use this example to calculate and plot the evaluation time of your `DFT(yk)` function for $N = 2^2, 2^3, \ldots, 2^M$. Depending on your implementation you might be able to go up to $M = 10$. Be careful and increase M just step by step!

(d) A very simple FFT algorithm can be derived by the following separation of the sum from above:

$$\begin{aligned}
\beta_j = \sum_{k=0}^{N-1} f(x_k) e^{-i\,j\,\frac{2\pi k}{N}} &= \sum_{k=0}^{N/2-1} f(x_{2k}) e^{-i\,j\,\frac{2\pi 2k}{N}} + \sum_{k=0}^{N/2-1} f(x_{2k+1}) e^{-i\,j\,\frac{2\pi(2k+1)}{N}} \\
&= \sum_{k=0}^{N/2-1} f(x_{2k}) e^{-i\,j\,\frac{2\pi k}{N/2}} + \sum_{k=0}^{N/2-1} f(x_{2k+1}) e^{-i\,j\,\frac{2\pi k}{N/2}}\, e^{-i\,j\,\frac{2\pi}{N}} \\
&= \beta_j^{\text{even}} + \beta_j^{\text{odd}}\, e^{-i\,j\,\frac{2\pi}{N}},
\end{aligned}$$

where β_j^{even} is the Fourier transform based on only even k (or x_k) and β_j^{odd} the Fourier transform based on only odd k. In case $N = 2^M$ this even/odd separation can be done again and again in a recursive way. Use the following template to implement a `FFT(yk)` function on your own (using your `DFT(yk)` from above):

```
def FFT(yk):

N  = # ... get the length of yk

if (N%2 > 0):
  # ... display an error message

elif N <= 2:
  return # ... call DFT with all yk points

else:
  betaEven = # ... call FFT but using just even yk points
  betaOdd = # ... call FFT but using just odd yk points

  expTerms = np.exp(-1j * 2.0 * np.pi * np.arange(N) / N)

  # Remember: beta_j is periodic in j!
  betaEvenFull = np.concatenate([betaEven, betaEven])
  betaOddFull = np.concatenate([betaOdd,  betaOdd])

  return betaEvenFull + expTerms * betaOddFull
```

Make sure that you get the same results as before by comparing the results from `DFT(yk)` and `FFT(yk)` for both functions defined in (b).

(e) Analyze the evaluation-time scaling of your `FFT(yk)` function with the help of the `timeit` module and compare the scaling to the one of `DFT(yk)`.

2 Composite Numerical Integration: Trapezoid and Simpson Rules

In the following we will implement the composite trapezoid and Simpson rules to calculate definite integrals. These rules are defined by

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(a) + 2\sum_{j=1}^{n-1} f(x_j) + f(b)\right] \qquad \text{trapezoid} \qquad (1)$$

$$\approx \frac{h}{3}\left[f(a) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(b)\right] \qquad \text{Simpson} \qquad (2)$$

with $a = x_0 < x_1 < \cdots < x_{n-1} < x_n = b$ and $x_k = a + kh$ with $k = 0, \ldots, n$ and $h = (b-a)/n$ being the step size.

(a) Implement both integration schemes as Python functions `trapz(yk, dx)` and `simps(yk, dx)` where `yk` is an array of length $n+1$ representing $y_k = f(x_k)$ and `dx` being the step size h. Compare your results with Scipy's functions `scipy.integrate.trapz(yk, xk)` and `scipy.integrate.simps(yk, xk)` for a $f(x_k)$ of your choice.

(b) Implement at least one unit test (using `pytest`) for each of your integration functions.

(c) Study the accuracy of these integration routines by calculating the following integrals for a variety of step sizes h:

- $\int_0^1 x\,dx$
- $\int_0^1 x^2\,dx$
- $\int_0^1 x^{\frac{1}{2}}\,dx$

Plot the integration error, defined as the difference (not the absolute difference) between your numerical results and the exact results, as a function of h for both integration routines and all listed functions. Comment on the comparison between both integration routines. Does the sign of the error match your expectations? If so / If not: Why?