Image & Signal Processing - Group work 1

Group: **Honza Zavadil**, **Jakob Rieke**
Date: **24/09/2021**

Task 1

Task 1.1

To find the form of Fourier series of function $f(x)$ we first need to compute the coefficients. Let

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{+\infty} \left(a_n \sin\left(\frac{2\pi n x}{T}\right) + b_n \cos\left(\frac{2\pi n x}{T}\right) \right),$$

where

$$a_n = \frac{2}{T}\int_0^T f(x)\cos\left(\frac{2\pi n x}{T}\right) \mathrm{d}x$$
$$b_n = \frac{2}{T}\int_0^T f(x)\sin\left(\frac{2\pi n x}{T}\right) \mathrm{d}x.$$

Since $f(x)$ is odd, all the coefficients a_n are equal to zero. For b_n we receive

$$b_n = 2\int_{-\frac{1}{2}}^{\frac{1}{2}} f(x)\sin(2\pi n x)\,\mathrm{d}x = 2\left(\int_{-\frac{1}{2}}^{0} -\sin(2\pi n x)\,\mathrm{d}x + \int_0^{\frac{1}{2}} \sin(2\pi n x)\,\mathrm{d}x\right) = \frac{2}{\pi n}\Big(1-\cos(\pi n)\Big) = \frac{4}{\pi n}\sin^2\left(\frac{\pi n}{2}\right.$$

Then we receive

$$f(x) = \sum_{n=1}^{+\infty} b_n \sin\left(\frac{2\pi n x}{T}\right) = \sum_{n=1}^{+\infty} \frac{4}{\pi n}\sin^2\left(\frac{\pi n}{2}\right)\sin(2\pi n x),$$

since $\sin^2\left(\frac{\pi n}{2}\right) = \begin{cases} 0 & \text{if } n \text{ is even} \\ 1 & \text{if } n \text{ is odd} \end{cases}$, we can write

$$f(x) = \frac{4}{\pi}\sum_{n=0}^{+\infty} \frac{\sin(2\pi(2n+1)x)}{2n+1}.$$

Task 1.2

The code to solve this task can be found in algorithm one.

```
import numpy as np
import matplotlib.pyplot as plt


# N should be > 0 and an even number.
# Also phiN must have been calculated over t.
def uN(phiN, t, N):
    Nhalf = int(N / 2)
    result = 0

    for k in range(-Nhalf + 1, Nhalf):
        # Use the left rectangle method to calculate the integral
        integral = np.sum(phiN * np.e**(-2j * np.pi * k * t) * 1/N)
        ck = 1 / (1 + 4 * (np.pi * k) ** 2) * integral
        result += ck * np.e ** (2j * np.pi * k * t)
    return result


def u(x):
    return np.cos(8 * np.pi * x)


def phi(x):
    return (1 + 64 * np.pi ** 2) * np.cos(8 * np.pi * x)


if __name__ == '__main__':
    # Plot either the regular graphs or their approximation errors
    plotErrors = False

    # Calculate u with a sample of 1000 in [0, 1]
    if not plotErrors:
        domain = np.linspace(0, 1, 1000, endpoint=False)
        plt.plot(domain, u(domain), color='cornflowerblue',
                 linewidth=6, alpha=0.5, label="u(x)")

    # Calculate and show u_N for multiple N's (or its errors)
    for N in [10, 20, 30, 40]:
        domain = np.linspace(0, 1, N, endpoint=False)

        # Approximate u_N
        resultUN = np.real(uN(phi(domain), domain, N))

        # Show u_N
        if not plotErrors:
            plt.plot(domain, resultUN, label=f"N = {N}")
        # Calculate and plot the errors
        else:
            plt.plot(domain, np.abs(u(domain) - resultUN), label=f"N = {N}")

    # Add a legend and show the plot
    plt.legend(loc="center right")
    plt.show()
```

Algorithm 1 Code for exercise 1.2.

We plotted our results with n-terms $\in \{100, 1000\}$ and x sampled by $10,000$ values. The corresponding graphs can be seen in image one and two.

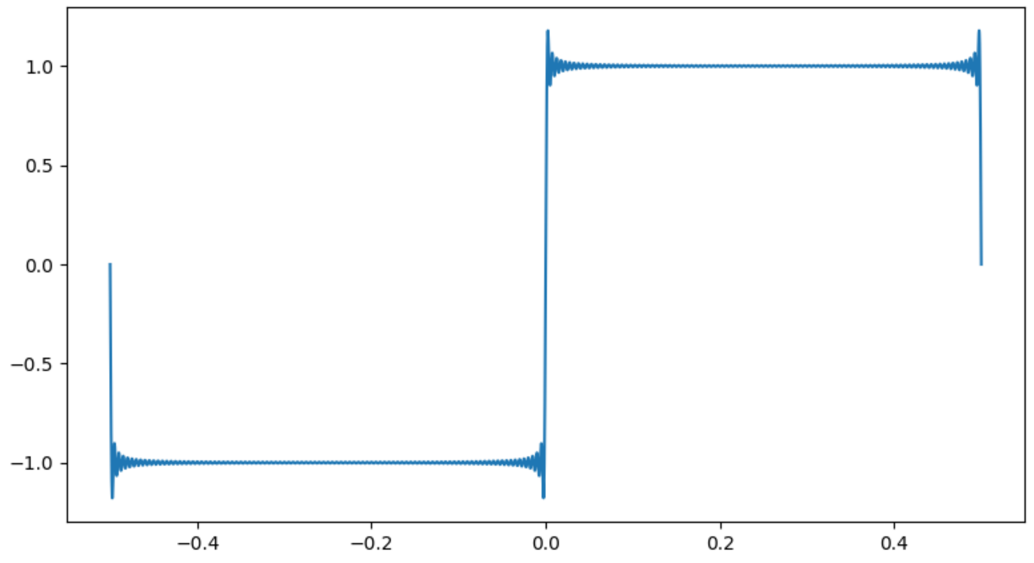

Image 1 Approximation of function $f(x)$ with n-terms = 100.

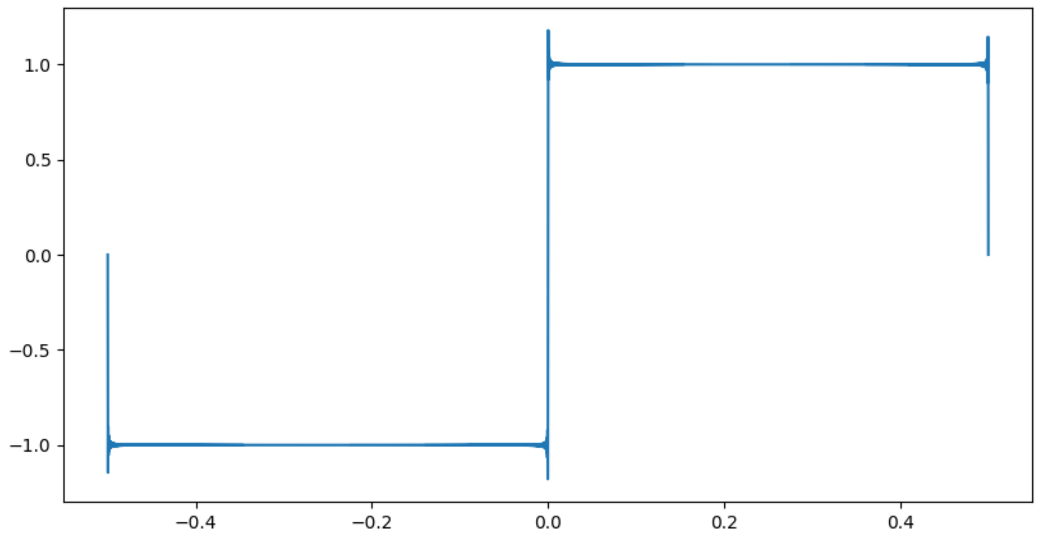

Image 2 Approximation of function $f(x)$ with n-terms = 1000.

Task 1.3

With higher number of terms in the sum, the frequency of the sinus function increases. If our sampling is not dense enough, we might end up having too few samples of the function to represent it. In the extreme case, the distance between two sampling points could be longer than the whole period of sinus function and we could end up only receiving positive values for example, which would render these last terms of the sum useless. In our implementation in case of $\text{n-terms} = 1000$ we get approx. 2.5 sampling points per period for the sinus with highest frequency, which does not represent the last function very well.

Task 1.4

Large oscillation close to the jump discontinuities (points $\{-\frac{1}{2}, 0, \frac{1}{2}\}$) can be observed. At these points the approximation overshoots the real function $f(x)$. Even by increasing n-terms, the overshoot is not reduced but only moves closer to the discontinuities. This phenomenon is in general called the Gibbs phenomenon.

Task 2

The following is our code enabling the computation of an approximation of the Fourier coefficients using FFT.

```
import numpy as np
import matplotlib.pyplot as plt


# Implementation of the function s(x) given in task one of the group work
def s(x):
    if (-1 / 2 <= x and x < 0):
        return -1
    elif (0 < x and x < 1 / 2):
        return 1
    else:
        return 0


# Checks if a number is even
def is_even(num):
    return not num & 0x1


if __name__ == '__main__':
    N = 100
    tn = np.linspace(-1/2, 1/2, 100)

    # Calculate stn
    stn = []
    for i in tn:
        stn.append(s(i))

    # Approximate the coefficients of the periodic function s with FFT
    approxCoeff = np.fft.fft(np.array(stn) / N)
    approxCoeff = np.absolute(approxCoeff)

    # Shift the coefficients
    split = np.split(approxCoeff, 2)
    approxCoeff = np.concatenate((split[1], split[0]))

    # Calculate the real coefficients
    domain = np.linspace(-N / 2, N / 2 - 1, N)
    realCoeff = np.absolute(4 / np.pi * 1 / (2j * (2 * domain + 1)))

    # Set real coefficients to zero
    for i, number in enumerate(domain):
        if is_even(int(number)):
            realCoeff[i] = 0

    # Plot the results
    plt.plot(domain, approxCoeff, domain, realCoeff)
    plt.show()
```

Algorithm 2 Code for exercise two.

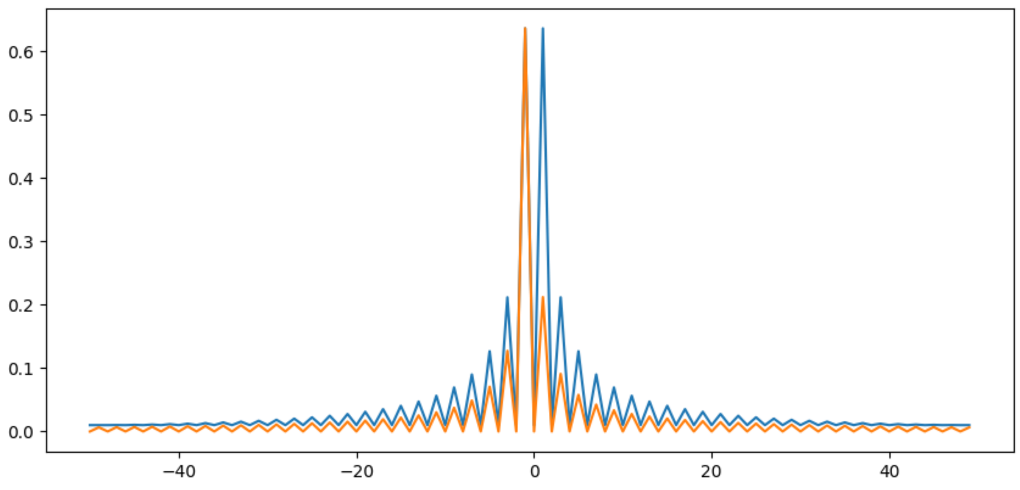

Image 3 The real coefficients are depicted in orange and the with FFT approximated ones in blue.

The result of algorithm one can be seen in image one. It shows a clear difference between the real and the approximated coefficients. It can be assumed that this is due to the integration with the left rectangle method of the function $s(x)$.

Task 3

Task 3.1

Since φ is periodic with period one, all the exponential functions in the definition of φ_N

$$\varphi_N(t) = \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} d_k e^{2i\pi kt}$$

are in the form of exponential basis for 1-periodic functions. With $N \to +\infty$ function φ_N converges to the Fourier series of function φ then and only then if the coefficients d_k are defined as

$$d_k = \int_0^1 \varphi(t) e^{-2i\pi kt} \mathrm{d}t.$$

Task 3.2

Let

$$u_N(t) = \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} c_k e^{2i\pi kt},$$

then

$$u_N''(t) = \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} -4\pi^2 k^2 c_k e^{2i\pi kt}.$$

After replacing u with u_N and φ with φ_N in equation (4), we receive

$$\sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} (1+4\pi^2k^2)c_k e^{2i\pi kt} = \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} d_k e^{2i\pi kt}.$$

Therefore, we receive the following relation between c_k and d_k

$$c_k = \frac{1}{(1+4\pi^2k^2)} d_k,$$
$$c_k = \frac{1}{(1+4\pi^2k^2)} \int_0^1 \varphi(t)e^{-2i\pi kt}\mathrm{d}t$$

Task 3.3

The exact solution for $u(x) - u''(x) = (1+64\pi^2)\ \cos(8\pi x)$ can been calculated using e. g. Symbolab or Wolfram Alpha and results in the following formula:

$$u(x) = c_1e^x + c_2e^{-x} + \cos(8\pi x)$$

For the implementation, which can be seen in algorithm 3, we defined $c_1 = c_2 = 0$.

```
import numpy as np
import matplotlib.pyplot as plt


# N should be > 0 and an even number.
# Also phiN must have been calculated over t.
def uN(phiN, t, N):
    Nhalf = int(N / 2)
    result = 0

    for k in range(-Nhalf + 1, Nhalf):
        # Use the left rectangle method to calculate the integral
        integral = np.sum(phiN * np.e**(-2j * np.pi * k * t) * 1/N)
        ck = 1 / (1 + 4 * (np.pi * k) ** 2) * integral
        result += ck * np.e ** (2j * np.pi * k * t)
    return result


def u(x):
    return np.cos(8 * np.pi * x)


def phi(x):
    return (1 + 64 * np.pi ** 2) * np.cos(8 * np.pi * x)


if __name__ == '__main__':
    # Plot u
    domain = np.linspace(0, 1, 1000, endpoint=False)
    plt.plot(domain, u(domain), color='cornflowerblue', linewidth=6, alpha=0.5)

    # Plot u_N
    for N in [10, 20, 30, 40]:
        domain = np.linspace(0, 1, N, endpoint=False)

        # Approximate u_N
        resultUN = np.real(uN(phi(domain), domain, N))
        plt.plot(domain, resultUN)

    plt.show()
```

Algorithm 3 Approximation of u by u_N.

This gives us the resulting graphs of u and u_{10}, u_{20}, u_{30} and u_{40} which can be seen in image four. As well as their difference graphs shown in image five. Interestingly all errors are nearly zero.

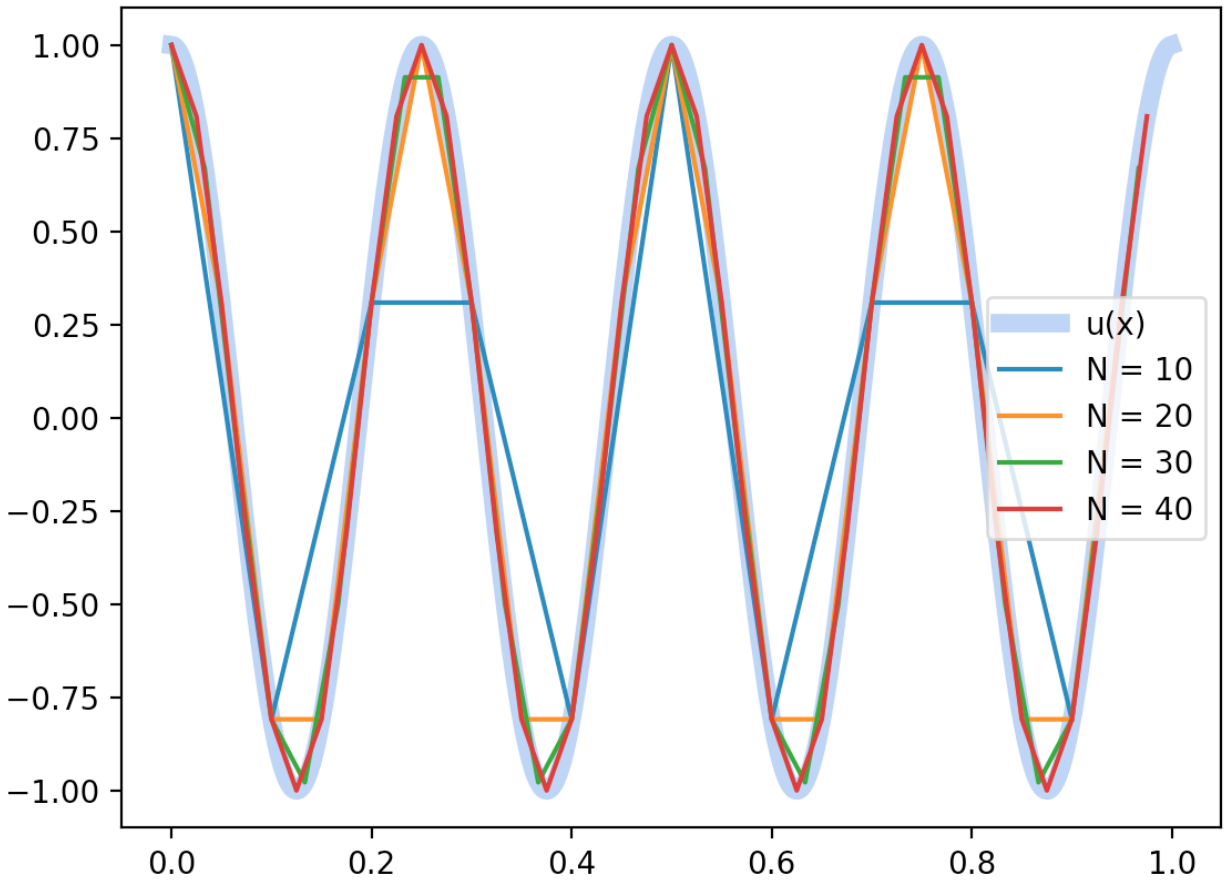

Image 4 Approximation of u_N for $N \in \{10, 20, 30, 40\}$ with the real values of u shown in thick light blue in the background.

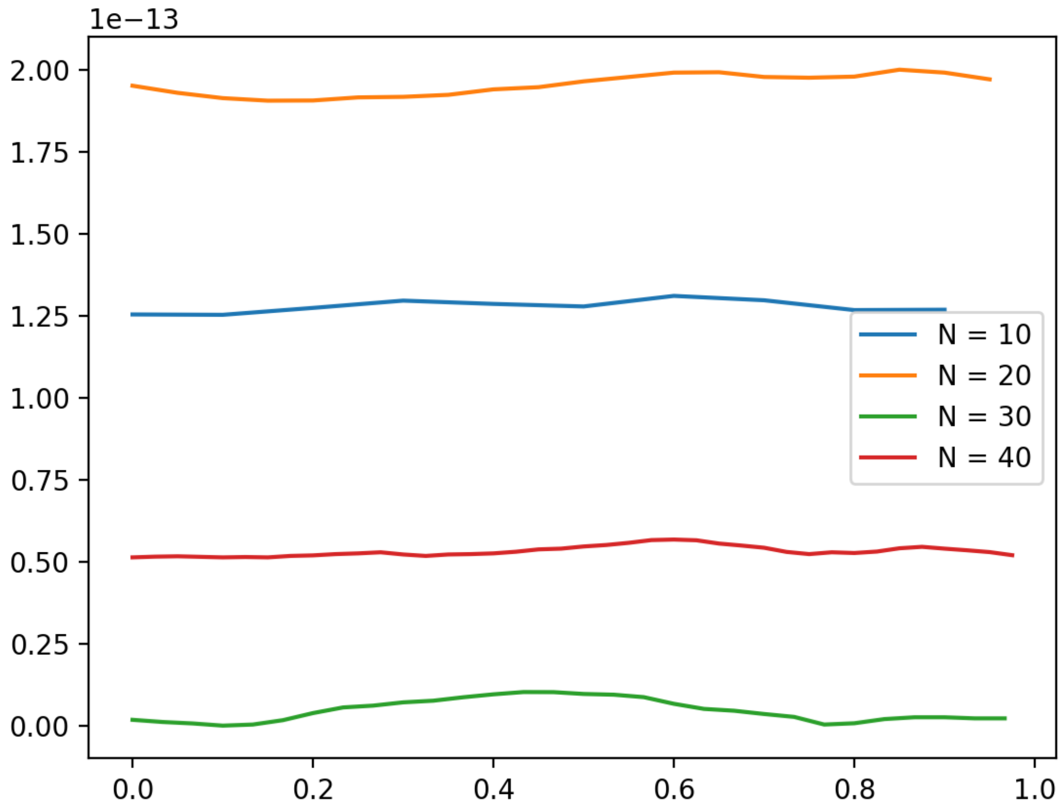

Image 5 Errors of the u_N compared to the real values of u.