Cem Daloğlu
21602161 / 25.04.2021

EEE 361 Homework-4 Report

In this homework report, denoising, random sampling and ADMM implementations will be discussed.

Sparse Signals and Denoising

Closed form solution for each $\hat{x_i}$ can be found via the following equation [1].

$$\hat{x_i} = y\left(1 - \frac{\lambda}{|y|}\right)_+ \quad eq.1$$

If $\left(1 - \frac{\lambda}{|y|}\right)$ is not greater than zero $\hat{x_i} = 0$. Therefore, when y is small compared to λ, $\hat{x_i}$ vector becomes more sparse since the equation becomes $\left(1 - \frac{\lambda}{|y|}\right) < 0$. When y is large compared to λ, $\hat{x_i}$ vector becomes less sparse or it can be said noisy.

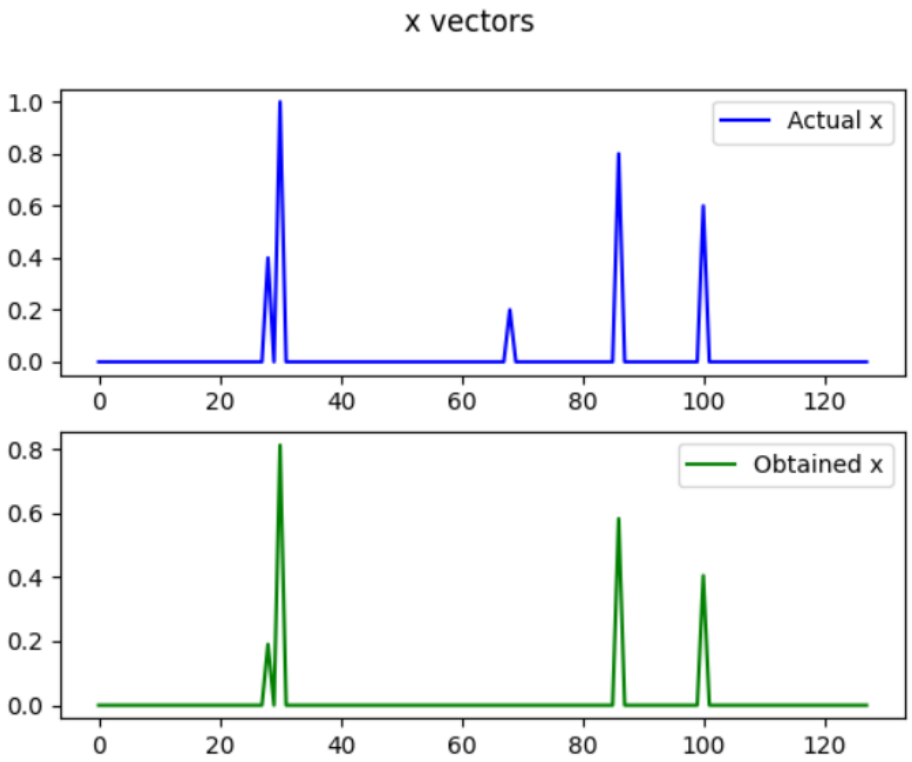

Figure 1: Obtained and actual x vectors for $\lambda = 0.1$

By looking at the Figure 1, it can be said that the actual signal is harshly denoised because denoise algorithm considered the x signal with magnitude 0.2 as a noise for $\lambda = 0.1$.

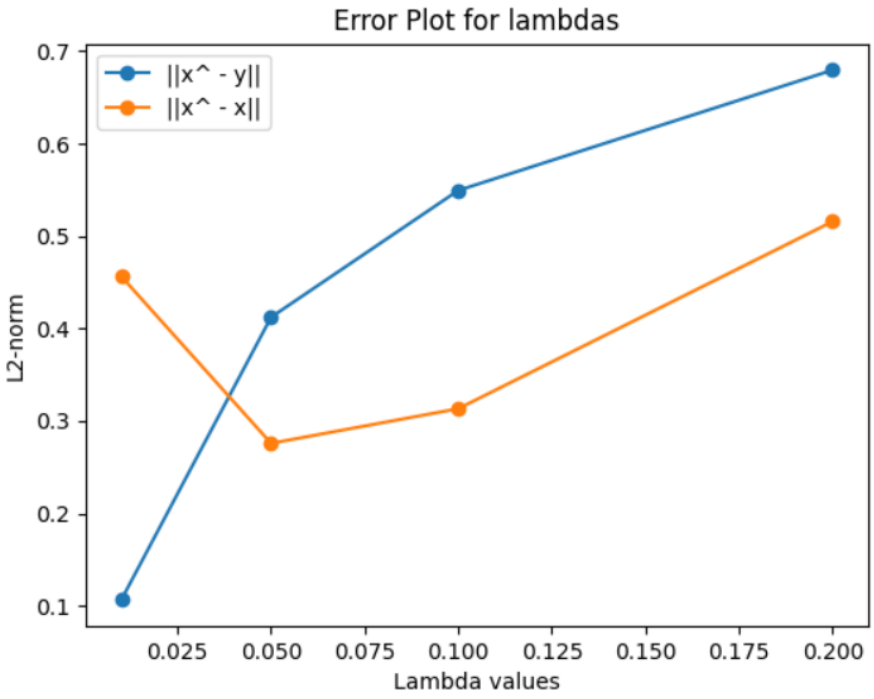

Figure 2: Obtained and actual x vectors for different λ values

As it can be seen in Figure 2 orange lines, for the smallest and the largest λ values the $\widehat{x_i}$ signal is away from converging to the actual x signal. However, for the middle λ values denoised and the actual signals are closer than the other λ values. In other words, when $\lambda = 0.01$, the denoising level is not enough to reconstruct the actual signal. The minimum value of the blue lines is obtained at the first λ value which means that the noise reduction is minimum. When $\lambda = 0.2$ signal is denoised too much and eliminated some parts of the original signal as well. Thus, when $\lambda = 0.05, \lambda = 0.1$ denoising is good enough to reconstruct the original signal.

Random Frequency Domain Sampling and Aliasing

From the Minimum Norm Least Square solution $\hat{x}$ becomes;

$$\hat{x} = Fu^{-1}y \;\; eq.2$$

When $y = fftc(x) = X$, $\hat{x}$ becomes undersampled inverse-dft of X which is equal to $xu/4$. The comparison of $xu/4$ and x_{mnls} can be seen in Figure 5. Since Fu contains $1/4$ of the all signal x_{mnls} becomes equal to $xu/4$ not xu.

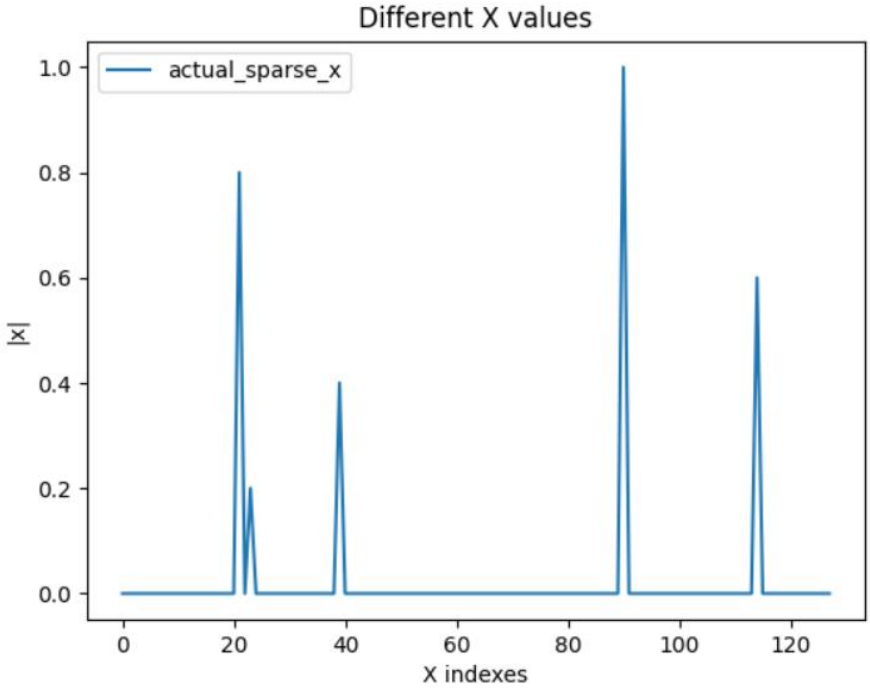

Figure 3: Actual x signal

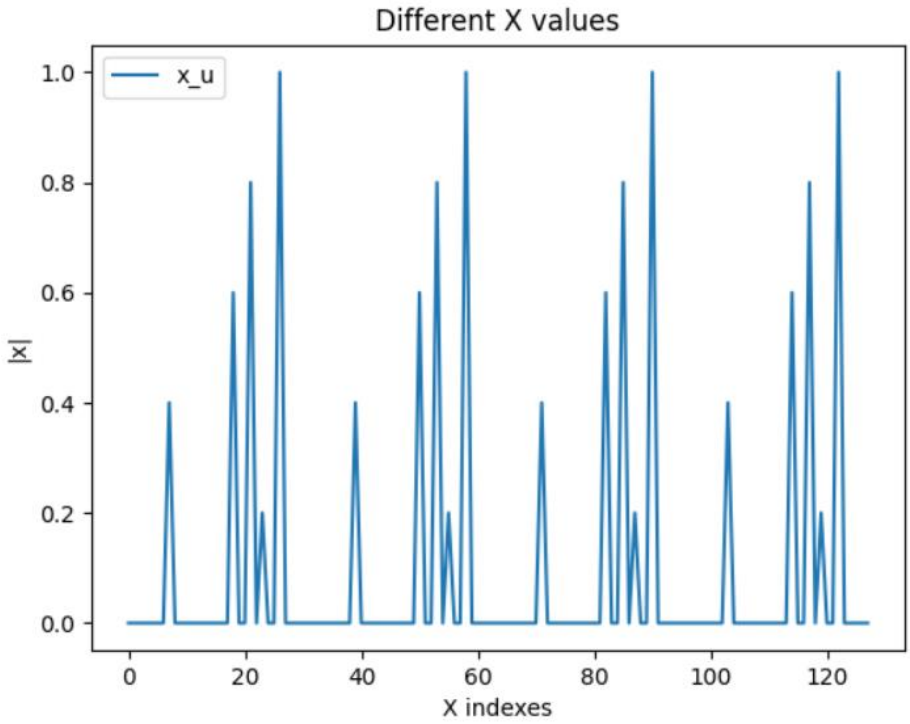

Figure 4: Undersampled x by equispaced samples (x_u)

As it can be seen in Figure 4, the actual x signal is obtained more than once, this is caused by undersampling. If 128 equispaced samples were taken we would get exactly the original signal. However, undersampling eliminates the signals which compensate the repeated signals and it causes aliasing.

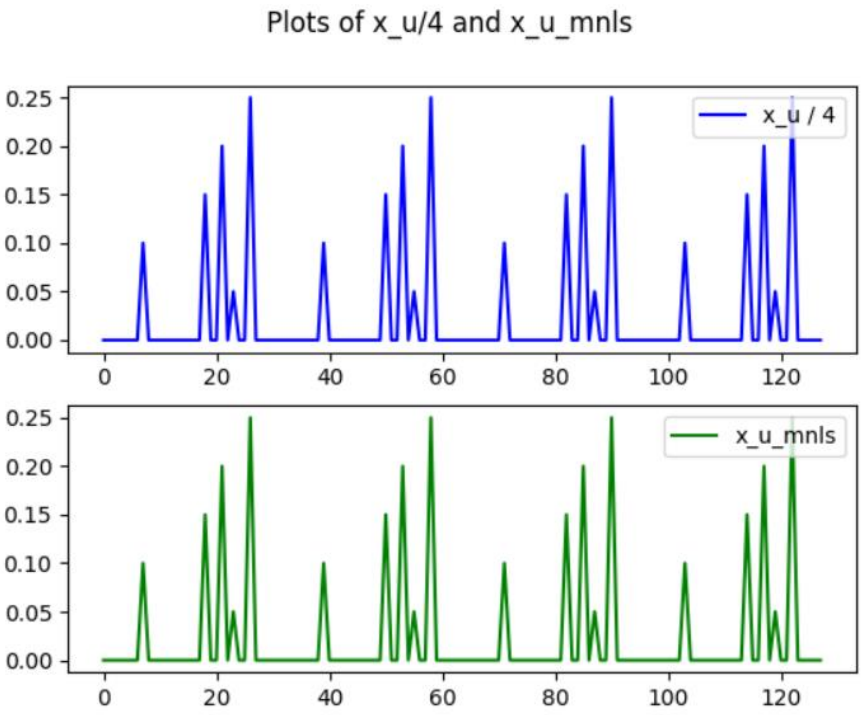

Figure 5: $x_u/4$ and x_{mnls} vectors

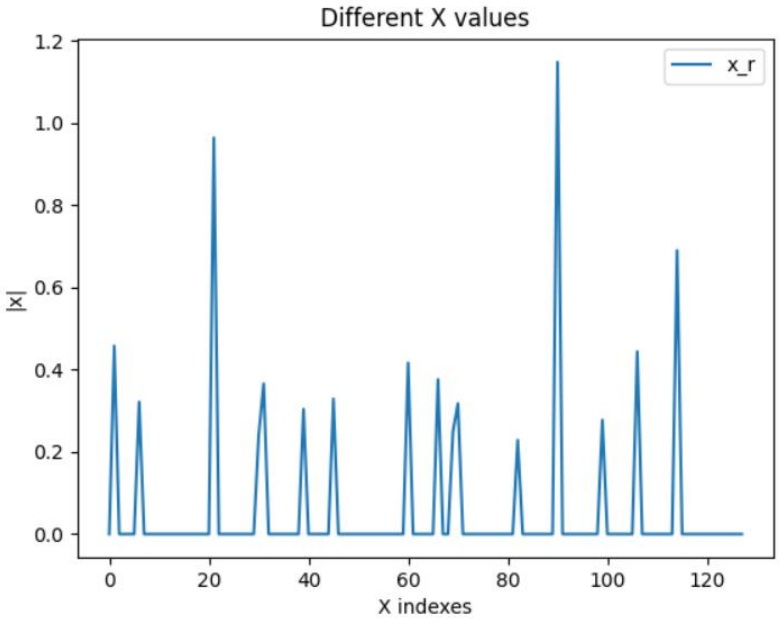

Figure 6: Undersampled x by random samples (x_r)

From Figure 6, it is difficult to compare the obtained x_r signal with the original signal because it is difficult to say which signal is caused by aliasing and which signal is obtained. Since samples are taken randomly, it is more difficult to see aliased signals than the obtained signal by equispaced samples. To obtain a better plot, the signals whose magnitudes are less than 0.2 are eliminated or dropped to zero. Thus, the x_r signal might be enough to recover from noise but to see this, a regularization component will be implemented in the next section.

Cem Daloğlu
21602161 / 25.04.2021

Reconstruction from Randomly Sampled Frequency Domain Data

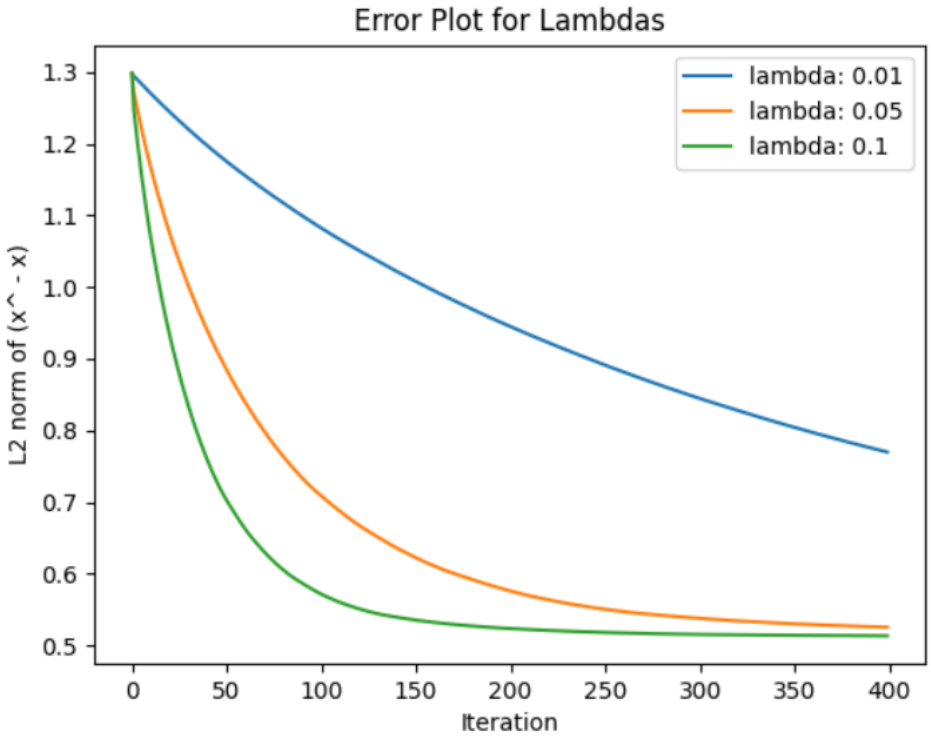

Figure 7: $\|\hat{x} - x\|_2^2$ plot for different λ

From Figure 7, it can be said that as the λ increases $\hat{x}$ converges to x faster. For $\lambda = 0.01$ 400 iterations are not enough to converge and for other λ values it converged to x almost with the same error rate. The results can be seen in Figure 8. In these examples ρ is taken as 10.

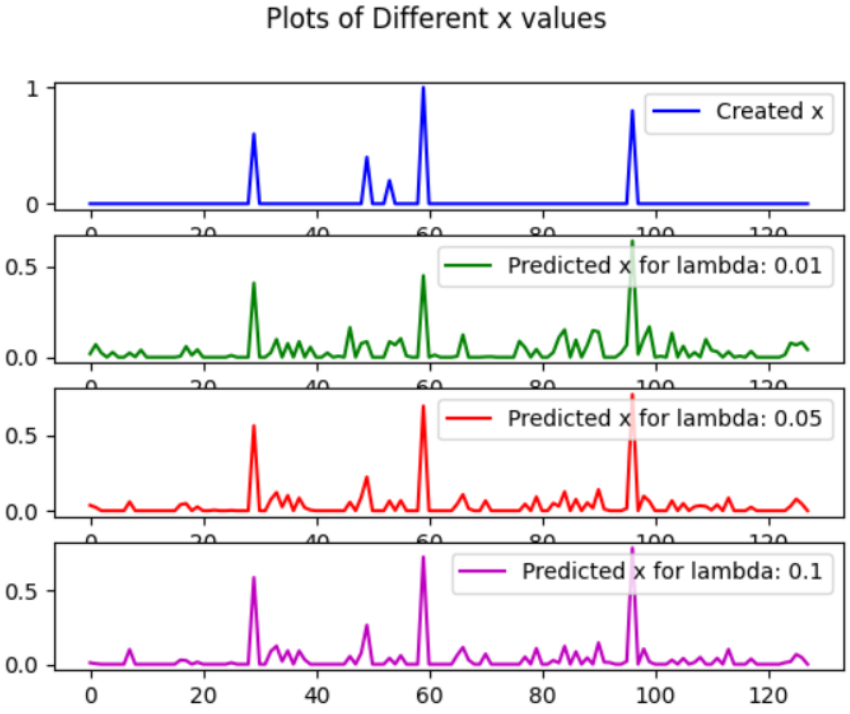

Figure 8: Actual and obtained x vector for different λ

By comparing the green lines and the purple lines in Figure 8, the effect of the λ can be seen. As λ increase denoising rate also increases as discussed in the first part of the report. For $\lambda = 0.01$ there is too much noise to make a close prediction for the signal with magnitude 0.2. For $\lambda = 0.05 \; and \; \lambda = 0.1$, except the signal with 0.2 magnitude other signals are predicted closely to the original signal. But it eliminates the signal with magnitude 0.2 so it could be said that %80 of the signals are predicted truly. To get a noiseless prediction λ value can be still increased.

To sum up, undersampling is applied to the DFT of the noisy data and to reconstruct the original signal Lasso formulation is implemented by using ADMM. Results showed that, 1-norm penalty term eliminated the aliasing parts that are obtained in the second part. However, denoising also eliminates the signal with magnitude 0.2.

References

[1] O. Arıkan, Lecture Notes 3, p.g. 49.

Appendix

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.linalg import dft


def part_1_1():
    lamda = [0.01, 0.05, 0.1, 0.2]
    results = np.zeros([8], dtype=float)
    x_i = np.zeros([1, 128], dtype=float)
    for l in range(len(lamda)):
        for i in range(128):
            if abs(y[0, i]) > lamda[l]:
                x_i[0, i] = y[0, i] * (1 - lamda[l] / abs(y[0, i]))
            else:
                x_i[0, i] = 0
        results[l] = np.linalg.norm(y[0] - x_i[0])
        results[l + 4] = np.linalg.norm(x_i[0] - vector_x[0])
    plt.plot([0.01, 0.05, 0.1, 0.2], results[:4], marker='o')
    plt.plot([0.01, 0.05, 0.1, 0.2], results[4:], marker='o')
    plt.legend(['||x^ - y||', '||x^ - x||'])
    plt.xlabel('Lambda values')
    plt.ylabel("L2-norm")
    plt.title('Error Plot for lambdas')
    plt.show()


def part_1_2():
    fftc_X = np.fft.fftshift(np.fft.fft(np.fft.fftshift(vector_x[0])))
    X_u = np.zeros([1, 128], dtype=complex)
    X_u[0, 0:128:4] = fftc_X[0:128:4]
    x_u = np.fft.fftshift(np.fft.ifft(np.fft.fftshift(X_u))) * 4

    Fc = dft(128)
    Fc = np.roll(Fc, 64)
    M = np.zeros((128, 128))
    for i in range(0, 128, 4):
        M[i][i] = 1
    Fu = M @ Fc
```

```
    x_u_mnls = np.dot(np.linalg.pinv(Fu), fftc_X.T)
    print(np.linalg.norm(x_u[0] / 4 - x_u_mnls))

    X_r = np.zeros([1, 128], dtype=complex)
    prm = np.random.permutation(128)
    X_r[0, prm[:32]] = fftc_X[prm[:32]]
    x_r = np.fft.fftshift(np.fft.ifft(np.fft.fftshift(X_r))) * 4

    for i in range(len(x_r[0])):
        if x_r[0, i] < 0.2:
            x_r[0, i] = 0

    plt.plot(vector_x[0])
    plt.legend(['actual_sparse_x'])
    plt.xlabel('X indexes')
    plt.ylabel("|x|")
    plt.title('Different X values')
    plt.show()
    plt.plot(abs(x_u[0]))
    plt.legend(['x_u'])
    plt.xlabel('X indexes')
    plt.ylabel("|x|")
    plt.title('Different X values')
    plt.show()
    plt.plot(abs(x_r[0]))
    plt.legend(['x_r'])
    plt.xlabel('X indexes')
    plt.ylabel("|x|")
    plt.title('Different X values')
    plt.show()

    fig, axs = plt.subplots(2)
    fig.suptitle('Plots of x_u/4 and x_u_mnls')
    axs[0].plot(abs(x_u[0] / 4), color='b', label='x_u / 4')
    axs[0].legend(loc="upper right")
    axs[1].plot(abs(x_u_mnls), color='g', label='x_u_mnls')
    axs[1].legend(loc="upper right")
    plt.show()


def ADMM(A, b, lamda, rho, iter):
    z = 0.001 * np.random.randn(128)
    u = 0.001 * np.random.randn(128)
    error = []
    for k in range(iter):
        x_hat = np.linalg.inv(A.conj().T @ A + rho * np.identity(128)) @
(np.dot(A.conj().T, b) + rho * (z - u))
        magnitude = np.maximum(abs(x_hat + u) - lamda / rho, 0)
        phase = np.angle(x_hat + u)
        z = np.exp(1j * phase) * magnitude
        u = u + x_hat - z
        error.append(np.linalg.norm(vector_x - abs(x_hat)))
    return x_hat, error


def part_1_3():
    Fc = dft(128)
    Fc = np.roll(Fc, 64)
    M = np.zeros((128, 128))
    prm = np.random.permutation(128)
    for i in range(32):
```

```
        M[prm[i]][prm[i]] = 1
    Fu = M @ Fc
    x_hat1, error1 = ADMM(Fu, Y, 0.01, 10, 40)
    x_hat2, error2 = ADMM(Fu, Y, 0.05, 10, 40)
    x_hat3, error3 = ADMM(Fu, Y, 0.1, 10, 40)

    plt.plot(error1)
    plt.plot(error2)
    plt.plot(error3)
    plt.legend(['lambda: 0.01', 'lambda: 0.05', 'lambda: 0.1'])
    plt.xlabel('Iteration')
    plt.ylabel("L2 norm of (x^ - x)")
    plt.title('Error Plot for Lambdas')
    plt.show()

    fig, axs = plt.subplots(4)
    fig.suptitle('Plots of Different x values')
    axs[0].plot(vector_x[0], color='b', label='Created x')
    axs[0].legend(loc="upper right")
    axs[1].plot(abs(x_hat1), color='g', label='Predicted x for lambda:
0.01')
    axs[1].legend(loc="upper right")
    axs[2].plot(abs(x_hat2), color='r', label='Predicted x for lambda:
0.05')
    axs[2].legend(loc="upper right")
    axs[3].plot(abs(x_hat3), color='m', label='Predicted x for lambda:
0.1')
    axs[3].legend(loc="upper right")
    plt.show()


vector_x = np.zeros([1, 128], dtype=float)
vector_x[0, 0] = 1 / 5
vector_x[0, 1] = 2 / 5
vector_x[0, 2] = 3 / 5
vector_x[0, 3] = 4 / 5
vector_x[0, 4] = 1
np.random.shuffle(vector_x[0, :])
sigma = 0.05
n = sigma * np.random.randn(128)
y = vector_x + n
Y = np.fft.fftshift(np.fft.fft(np.fft.fftshift(y[0])))

part_1_1()
part_1_2()
part_1_3()
```