CS 530 Fall 2025 Homework 2

Calvin Stahoviak – cstahoviak@unm.edu

November 22, 2025

Collaborators: N/A

By turning in this assignment, I agree to the UNM Student Code of Conduct and declare that all of this is my own work.

Problem 1

(a) Plot the real part, the imaginary part, and the length (or norm) of the following vectors in the Fourier Basis of $\mathbb{C}^{64}$:

$$F_0, F_1, F_{63}, F_2, F_{62}, F_8, F_{56}, F_{32}, \tag{1}$$

Note that the FFT function in Matlab counts the indices from $1, 2, \ldots, N$ instead of the usual CS programming convention $0, 1, 2, \ldots, N-1$. If you are using Matlab, remember to use the correct index.

Solution: The vectors were plotted in Figure 1. The Python code used to generate these plot is shown below. The main computation comes from the method `get_fourier_basis_vector`.

```
N = 64
indices = [0, 1, 63, 2, 62, 8, 56, 32]

def get_fourier_basis_vector(k, N):
    n = np.arange(N)
    return np.exp(-2j * np.pi * k * n / N)

fig, axes = plt.subplots(len(indices), 3, figsize=(8, 12))
fig.suptitle(r'Fourier Basis Vectors in $\mathbb{C}^{64}$',
             fontsize=18)

for row, k in enumerate(indices):
    F_k = get_fourier_basis_vector(k, N)
```

```
axes[row, 0].plot(np.real(F_k), 'b-', linewidth=1.5)
axes[row, 0].set_ylabel(f'$F_{{{k}}}$')
if row == 0:
    axes[row, 0].set_title('Real Part')

axes[row, 1].plot(np.imag(F_k), 'r-', linewidth=1.5)
if row == 0:
    axes[row, 1].set_title('Imaginary Part')

axes[row, 2].plot(np.abs(F_k), 'g-', linewidth=1.5)
if row == 0:
    axes[row, 2].set_title('Magnitude')
```

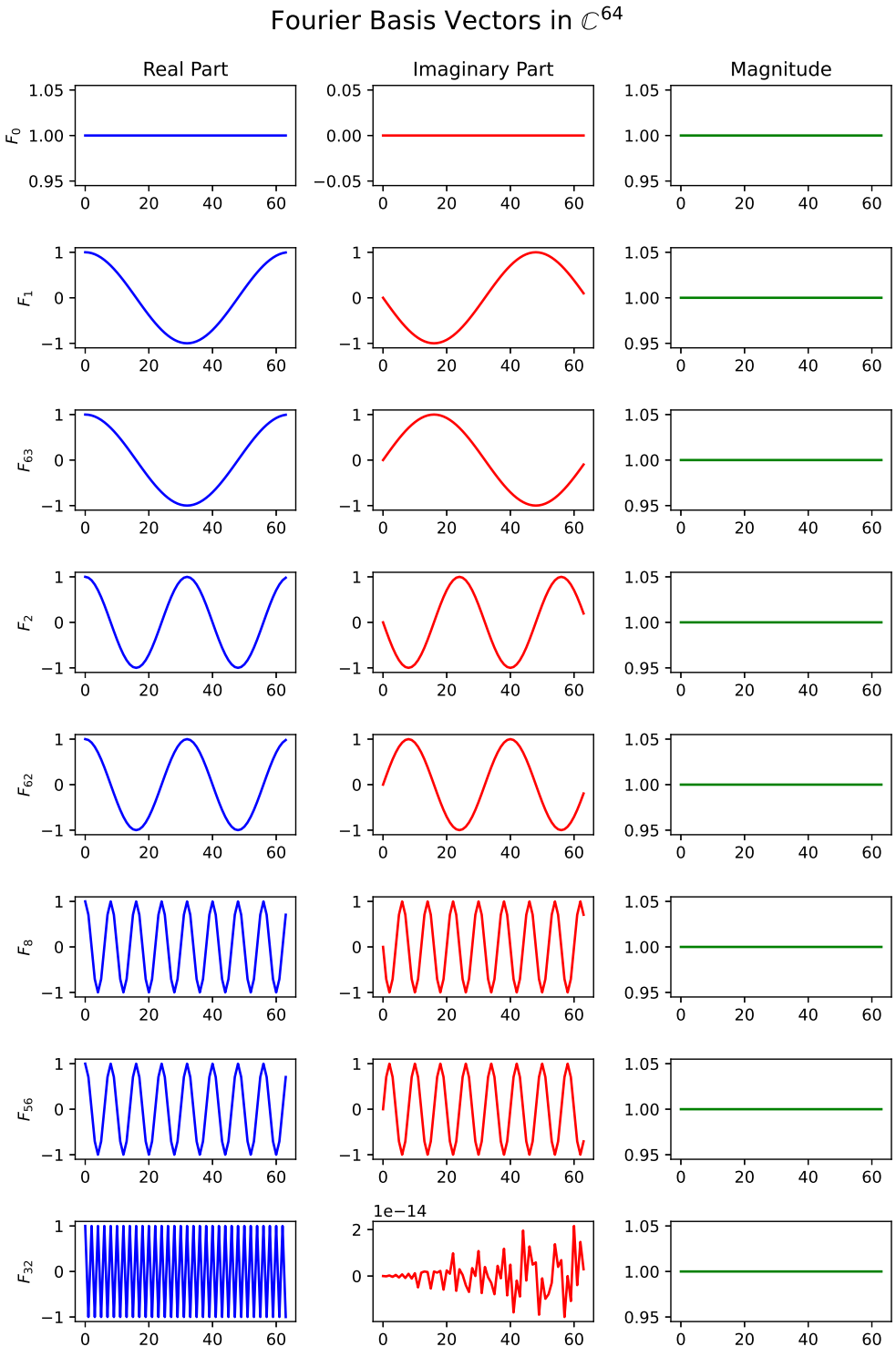

Figure 1: Fourier basis vectors in $\mathbb{C}^{64}$

Problem 2

(Practice 1 Dimensional FFT and DCT)

(a) Consider the following 512-dimension column vector , where:

$$z(n) = \begin{cases} 0 & 0 \leq n \leq 64, \quad 384 \leq n \leq 511 \\ 1 & 65 \leq n \leq 383 \end{cases} \tag{2}$$

Use Matlab or Python:

Calculate the Fourier Transform and Discrete Cosine Transform of z and plot the result. Is the Fourier Transform of z real? Explain why or why not.

Low Pass: For both the Fourier Transform and Discrete Cosine Transform, set the 128 entries of high frequency components to 0 and perform respective inverse transforms and plot the result.

High Pass: For both the Fourier Transform and Discrete Cosine Transform, set the 64 entries of low frequency components to 0 and perform respective inverse transforms and plot the result.

Compression: For both the Fourier Transform and the Discrete Cosine Transform, remove the 448 entries of the smallest norm or value and perform the respective inverse transforms and plot the result. Note that this is an 8:1 compression.

Solution: The following plots were made in Python. The Fourier Transform is computed using the Fast Fourier Transform. Which along with the Discrete Cosine Transform are imported from SciPy `fftpack`.

The original signal $z(n)$, the real component, imaginary component, and magnitude of the Fourier Transform $\hat{z}$, and Discrete Cosine Transform is show in Figure 2. After the Fourier Transform, $\hat{z}$ has a maximum imaginary component of 57.765968, indicating that $\hat{z}$ is not completely real. It is not real because because $z(n)$ is not an even/symmetric function. The function $z(n)$ has a high in the domain $[64, 383]$ which is asymmetric in the total domain $[0, N-1]$ where $N = 512$. For a real signal to have a real values Fourier Transform then it must be symmetric or even.

The low pass and high pass of both the Fourier Transform and Discrete Cosine Transform is seen in Figure 3. An 8:1 compression of both the Fourier Transform and Discrete Cosine Transform is shown in Figure 4.

The total code for generating these figures is too long (168 lines). The most critical parts are shown below:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft, ifft, dct, idct
```

```
# Create signal and compute transforms
N = 512
z = np.zeros(N)
z[65:384] = 1
z_fft = fft(z)
z_dct = dct(z, type=2, norm='ortho')

# Low Pass: Zero 128 high frequency components
z_fft_lowpass = z_fft.copy()
z_fft_lowpass[192:320] = 0
z_fft_lowpass_recon = np.real(ifft(z_fft_lowpass))

z_dct_lowpass = z_dct.copy()
z_dct_lowpass[-128:] = 0
z_dct_lowpass_recon = idct(z_dct_lowpass,
                           type=2,
                           norm='ortho')

# High Pass: Zero 64 low frequency components
z_fft_highpass = z_fft.copy()
z_fft_highpass[0:32] = z_fft_highpass[-32:] = 0
z_fft_highpass_recon = np.real(ifft(z_fft_highpass))

z_dct_highpass = z_dct.copy()
z_dct_highpass[0:64] = 0
z_dct_highpass_recon = idct(z_dct_highpass,
                            type=2,
                            norm='ortho')

# Compression: Keep only 64 largest magnitude components
z_fft_compressed = z_fft.copy()
z_fft_compressed[np.argsort(np.abs(z_fft))[:448]] = 0
z_fft_compressed_recon = np.real(ifft(z_fft_compressed))

z_dct_compressed = z_dct.copy()
z_dct_compressed[np.argsort(np.abs(z_dct))[:448]] = 0
z_dct_compressed_recon = idct(z_dct_compressed,
                              type=2,
                              norm='ortho')
```

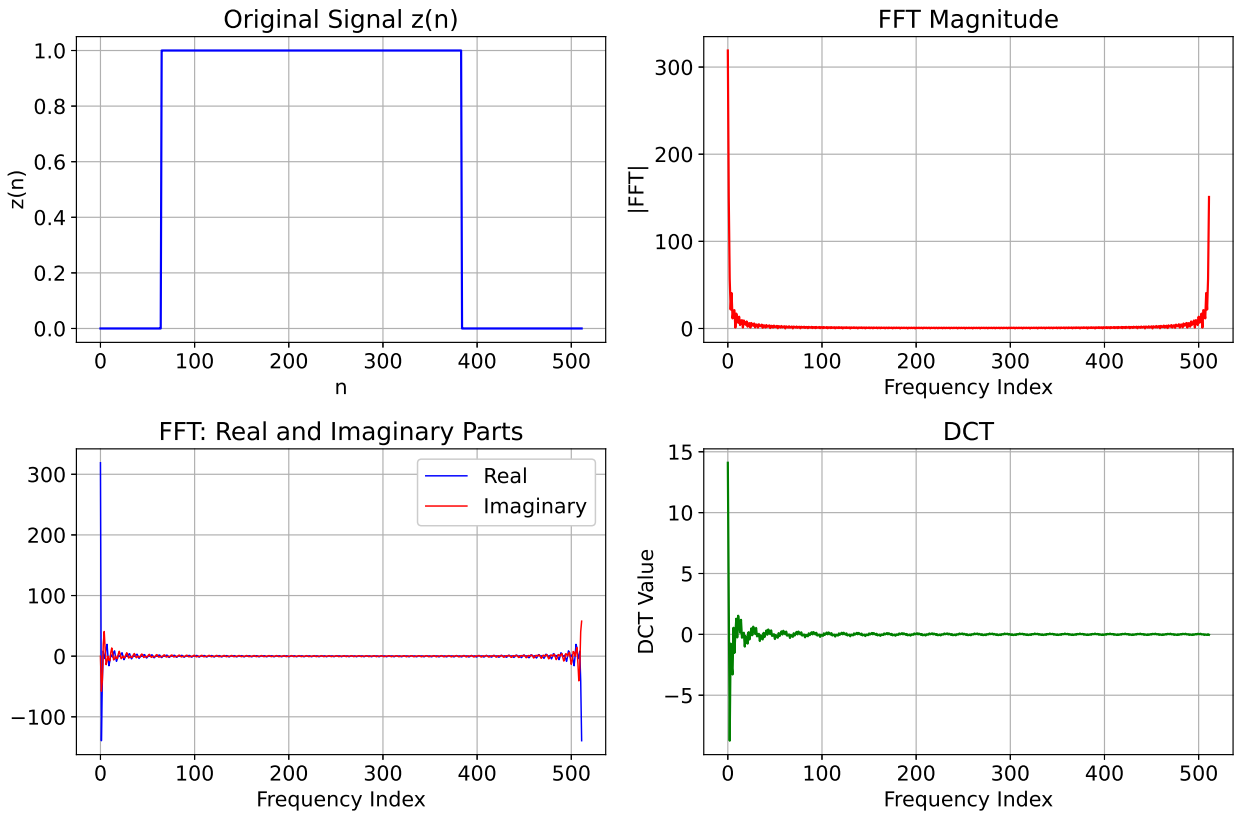

Figure 2: Original signal z(n) and its Fourier Transform (FFT) and Discrete Cosine Transform (DCT)

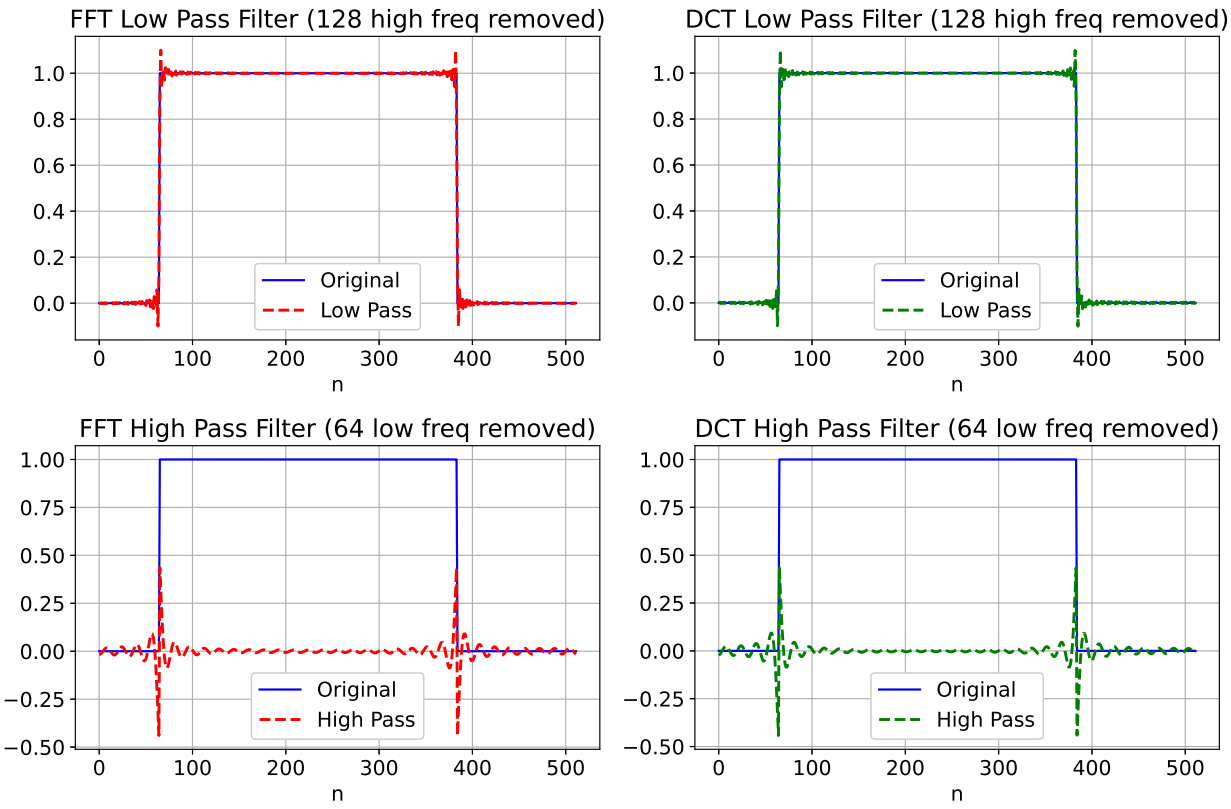

Figure 3: Low pass and high pass filtering results for FFT and DCT

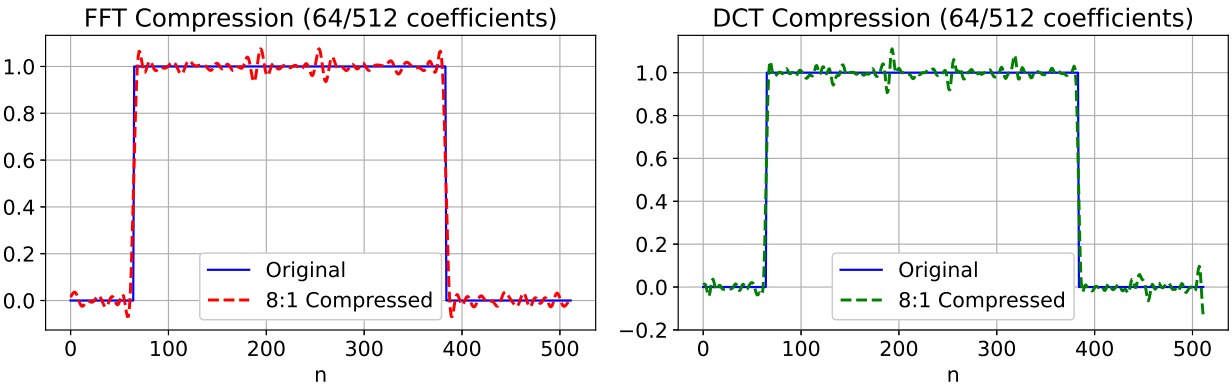

Figure 4: 8:1 compression results using FFT and DCT (keeping 64 out of 512 coefficients)

Problem 3

(Prove Parseval's Theorem)

(a) When **normalized** Fourier Transform is used, the Parseval's Theorem states that for $z, w \in \mathbb{C}^N$:

$$\langle z, w \rangle = \langle \hat{z}, \hat{w} \rangle \tag{3}$$

where $\hat{z}$ and $\hat{w}$ are the Fourier transforms of z and w respectively.

(Hint: this proof is very similar to the Plankerel's Theorem as presented in class.)

Solution: First let's establish the known formulas. The normalized Fourier Transform is:

$$\hat{z}(m) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} z(n) e^{-2\pi i \frac{mn}{N}} \tag{4}$$

And the inverse Fourier Transform is:

$$z(n) = \frac{1}{\sqrt{N}} \sum_{m=0}^{N-1} \hat{z}(m) e^{2\pi i \frac{mn}{N}} \tag{5}$$

When looking to prove $\langle z, w \rangle = \langle \hat{z}, \hat{w} \rangle$, we start with the left-hand side:

$$\langle z, w \rangle = \sum_{n=0}^{N-1} z(n) \overline{w(n)} \tag{6}$$

We can substitute $z(n)$ and $\overline{w(n)}$ with their inverse Fourier Transform from Equation (5):

$$\langle z, w \rangle = \sum_{n=0}^{N-1} \left(\frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \hat{z}(k) e^{2\pi i \frac{kn}{N}} \right) \left(\frac{1}{\sqrt{N}} \sum_{l=0}^{N-1} \overline{\hat{w}(l)} e^{-2\pi i l n/N} \right) \tag{7}$$

Pulling out the constant $1/\sqrt{N}$ and aligning summations results in:

$$\langle z, w \rangle = \frac{1}{N} \sum_{n=0}^{N-1} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \hat{z}(k) \overline{\hat{w}(l)} e^{2\pi i (k-l) n/N} \tag{8}$$

Next, rearrange summations sot that a conclusion can be made about k and l:

$$\langle z, w \rangle = \frac{1}{N} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \hat{z}(k) \overline{\hat{w}(l)} \left(\sum_{n=0}^{N-1} e^{2\pi i (k-l) n/N} \right) \tag{9}$$

Notice that latter summation evaluates to a piecewise function:

$$\sum_{n=0}^{N-1} e^{2\pi i (k-l) n/N} = \begin{cases} N & \text{if } k = l \\ 0 & \text{if } k \neq l \end{cases} \tag{10}$$

This is the orthogonality of Fourier basis vectors. When $k \neq l$, corresponding terms reduce to zero. In the case $k = l$, the latter summation simplifies to N. Using Equation (10), results in:

$$\langle z, w \rangle = \frac{1}{N} \sum_{k=0}^{N-1} \hat{z}(k)\overline{\hat{w}(k)} \cdot N = \sum_{k=0}^{N-1} \hat{z}(k)\overline{\hat{w}(k)} \tag{11}$$

Using the definition of inner product, Equation (6), the proof is complete:

$$\langle z, w \rangle = \langle \hat{z}, \hat{w} \rangle \tag{12}$$

Problem 4

(a) In the class, we discussed circular convolution:

$$(w * z)(m) = \sum_{n=0}^{N-1} w(m-n)z(n) \tag{13}$$

and the convolution theorem:

$$\hat{(w * z)}(m) = \hat{w}(m)\hat{z}(m) \tag{14}$$

The goal of this problem is to investigate the so-called "de-convolution" process. More specifically, assume that you are given the vectors z and x, can you find a vector w such that $w * z = x$? Develop a fast algorithm for calculating de-convolution.

Solution: Given the vectors z and x, we want to find a vector w such that:

$$w * z = x \tag{15}$$

Let's try using Equation (14) after a Fourier Transform for this algorithm.

1. Let $\hat{x}$ and $\hat{z}$ be the Fourier Transforms of x and z.
2. Apply the Fourier transformation to both Equation (15):

$$\hat{(w * z)} = \hat{x} \tag{16}$$

$$\hat{w} \cdot \hat{z} = \hat{x} \tag{17}$$

3. Solve for $\hat{w}$:

$$\hat{w}(k) = \frac{\hat{x}(k)}{\hat{z}(k)} \quad \text{for } k = 0, 1, \ldots, N-1 \tag{18}$$

4. Finally solve for w by taking the inverse Fourier Transform.

$$w(n) = \frac{1}{N} \sum_{k=0}^{N-1} \frac{\hat{x}(k)}{\hat{z}(k)} e^{2\pi i k n/N} \quad \text{for } n = 0, 1, \ldots, N-1 \tag{19}$$

Note that when $\hat{z}(k) = 0$ then de-convolution is undefined (zero division).

Problem 5

(a) Let w, z, and e_0 be N dimension complex vectors, and e_0 be defined as:

$$e_0 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{20}$$

We say that w is the convolution inverse of z if $w * z = e_0$.

Design an efficient algorithm such that given a vector z, determine if z has a convolution inverse, and if yes find the convolution inverse of z.

Solution: Given a vector z to start. Notice that this is a special case of de-convolution where $x = e_0$.

1. Let $\hat{e}_0$ and $\hat{z}$ be the Fourier Transforms of e_0 and z.
2. Since $e_0 = [1, 0, \ldots, 0]^T$, then $\hat{e}_0$ can be simplified:

$$\hat{e}_0(m) = \sum_{n=0}^{N-1} e_0(n) e^{-2\pi i \frac{mn}{N}} = 1 \tag{21}$$

3. Verify that $\hat{z}(m) \neq 0$ for all m.

$$w = \begin{cases} \text{Convolution inverse exists} & \text{if } \hat{z}(m) \neq 0 \text{ for all } m \in \{0, 1, \ldots, N-1\} \\ \text{undefined (does not exist)} & \text{otherwise} \end{cases} \tag{22}$$

   where w is the convolution inverse.
4. Apply the Fourier transformation to $w * z = e_0$:

$$\hat{(w * z)} = \hat{e}_0 \tag{23}$$

$$\hat{w} \cdot \hat{z} = \hat{e}_0 \tag{24}$$

5. Compute $\hat{w}$:

$$\hat{w}(m) = \frac{\hat{e}_0(m)}{\hat{z}(m)} = \frac{1}{\hat{z}(m)} \quad \text{for } m = 0, 1, \ldots, N-1 \tag{25}$$

6. Take the inverse Fourier Transform of $\hat{w}(m)$

$$w(n) = \frac{1}{N} \sum_{m=0}^{N-1} \hat{w}(m) e^{2\pi i mn/N} = \frac{1}{N} \sum_{m=0}^{N-1} \frac{1}{\hat{z}(m)} e^{2\pi i mn/N} \quad \text{for } n = 0, 1, \ldots, N-1 \tag{26}$$

Problem 6

In this problem, you will be exploring the frequency domain using both Fourier Transform and Discrete Cosine Transform.

Suppose you are given the following target distribution:

$$D^*(n) = \begin{cases} 0 & 0 \leq n \leq 223, \quad 288 \leq n \leq 511 \\ 1 & 224 \leq n \leq 287 \end{cases} \tag{27}$$

You may imagine that the given D^* is a radiosurgery dose prescription, which aims to deliver a high dose to the tumor and keeping the dose to everywhere else to 0.

Suppose that the Gamma Knife machine can deliver radiation "shots" of Gaussian dose distribution $N(\mu, 10)$. Notice that μ is the center of the distribution, while $\sigma = 10$ governs the shape of the distribution. You may imagine that a Gamma Knife Unit is capable of adjusting the μ to move the shots around. The figure below shows $N(256, 10)$.

The 2 plots shown here are obtained using the following Matlab code:

```
D = zeros (1, 512);
D (1, 224:287) = 1 ;
figure(1) ;
plot (D) ;
ylim([-0.1, 1.2] );

x = 0:1:511;
y = normpdf(x, 256, 10) ;
figure(2) ;
plot(y) ;
ylim([-0.001, 0.05] );
```

The Gamma Knife Radiosurgery planning problem is to find the "beam-on" times t_j for a set of shots such that:

$$\min \left\| \sum_{j=224}^{287} t_j N(j, 10) - D^* \right\|_2^2 \quad \text{s.t.} \quad t_j \geq 0 \text{ for } j = 224, 225, \ldots, 287 \tag{28}$$

In this optimization,

$$\sum_{j=224}^{287} t_j N(j, 10) \tag{29}$$

is the dose distribution created by the plan. The reason why $j = 224, 225, \ldots, 287$ is because it makes no sense to deliver any shots that is focused outside the tumor. The goal of the optimization is to minimize the difference

$$\left(\sum_{j=224}^{287} t_j N(j, 10) \right) - D^* \tag{30}$$

The non-negative constraints are due to that beam-on times (i.e., weighting of the shots) must be non-negative.

(a) Gain the basic understanding of the problem by solving the above non-negative least square optimization problem using Matlab or Python and plot:

$$\sum_{j=224}^{287} t_j N(j, 10) \quad \text{vs.} \quad D^* \tag{31}$$

(The matlab routine for solving non-negative least square is lsqnonneg.)

Solution: The goal is to minimize Equation (28). This is equivalent to the minimization:

$$\min_t ||At - D^*||_2^2 \quad \text{s.t.} \quad t \geq 0 \tag{32}$$

where A is a 512×64 matrix and t is a vector of beam times for each valid position. The matrix A represents the dosage of each beam across all positions

- Columns: $j = 64$ columns for all valid beam inputs (don't want to apply dosage outside of the tumor area).
- Rows: 512 rows show the dosage at each position n for a given beam at coll um j.
- Entry $A_{n,j}$: Dose at position n from a beam centered at position $(224 + j)$.

The dosage at position n can be calculated from A in the following way:

$$\text{dose}(n) = \sum_{j=0}^{63} A_{n,j} \cdot t_j = \sum_{j=224}^{287} t_j \, N(j, 10; n) \tag{33}$$

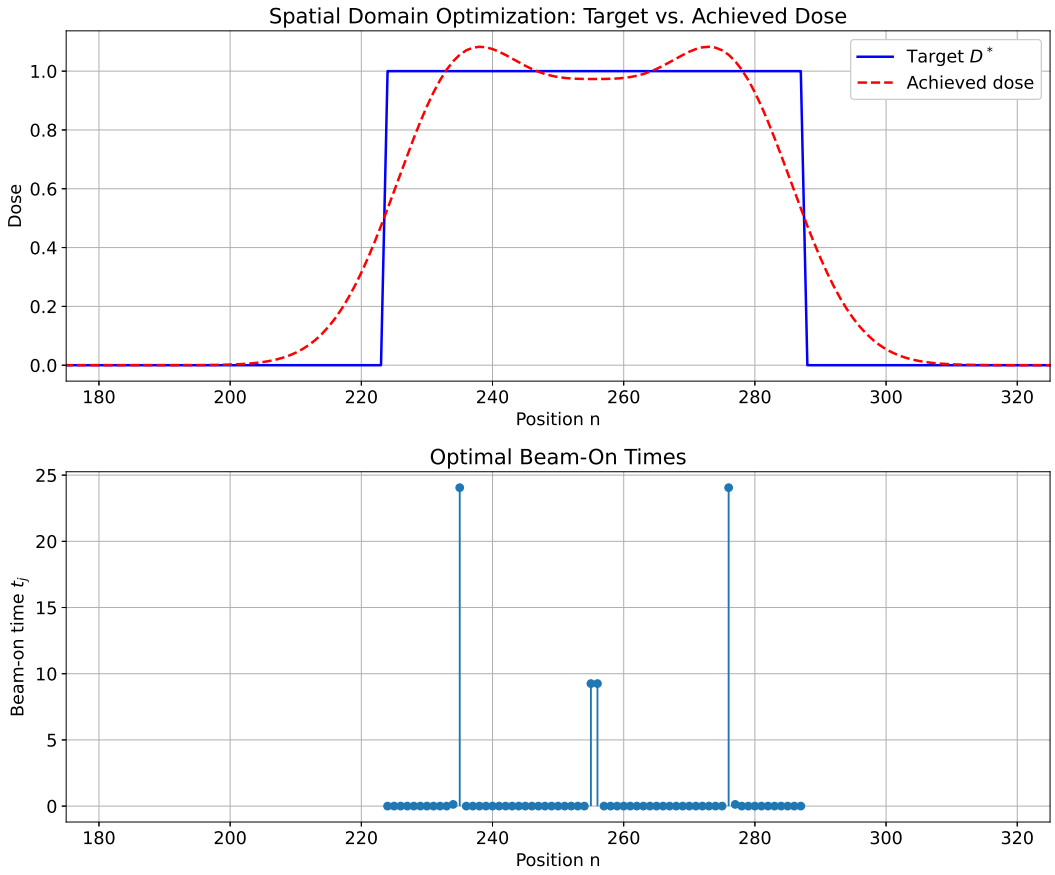

Figure 5: Spatial domain optimization results. (Top) Target dose distribution D^* versus achieved dose distribution from optimal beam-on times. (Bottom) Optimal beam-on times t_j for shots centered at positions $j = 224, 225, \ldots, 287$.

The result of the optimization, visualized in Figure 5, yielded 6 non-zero beams and had an L2 error of 1.790499.

The following Python code was used for calculations:

```
# Problem parameters
N = 512
tumor_start = 224
tumor_end = 287
sigma = 10

# Create target distribution D*
D_star = np.zeros(N)
D_star[tumor_start:tumor_end+1] = 1

# Create the matrix A
```

```
# 64 shots
n_shots = tumor_end - tumor_start + 1
A = np.zeros((N, n_shots))

for j in range(n_shots):
    # Center position: 224, 225, ..., 287
    mu = tumor_start + j
    # Evaluate Gaussian N(mu, sigma)
    A[:, j] = norm.pdf(np.arange(N), loc=mu, scale=sigma)

# Solve non-negative least squares:
# min ||A*t - D_star||^2, s.t. t >= 0
t_optimal, residual = nnls(A, D_star)

# Compute the resulting dose distribution
dose_distribution = A @ t_optimal
```

(b) Perform the above optimization in the frequency domain using both Fourier Transform and Discrete Cosine Transform and compare the performance of the results with those from the spatial domain as in (1).

Solution: The previous optimization is done in both Fourier Transform and Discrete Cosine Transform. These dose distributions and beam weights combined with the spatial domain analysis are shown in Figure 6. All three methods yeilded the same L2 error (1.790499) and beam weights.

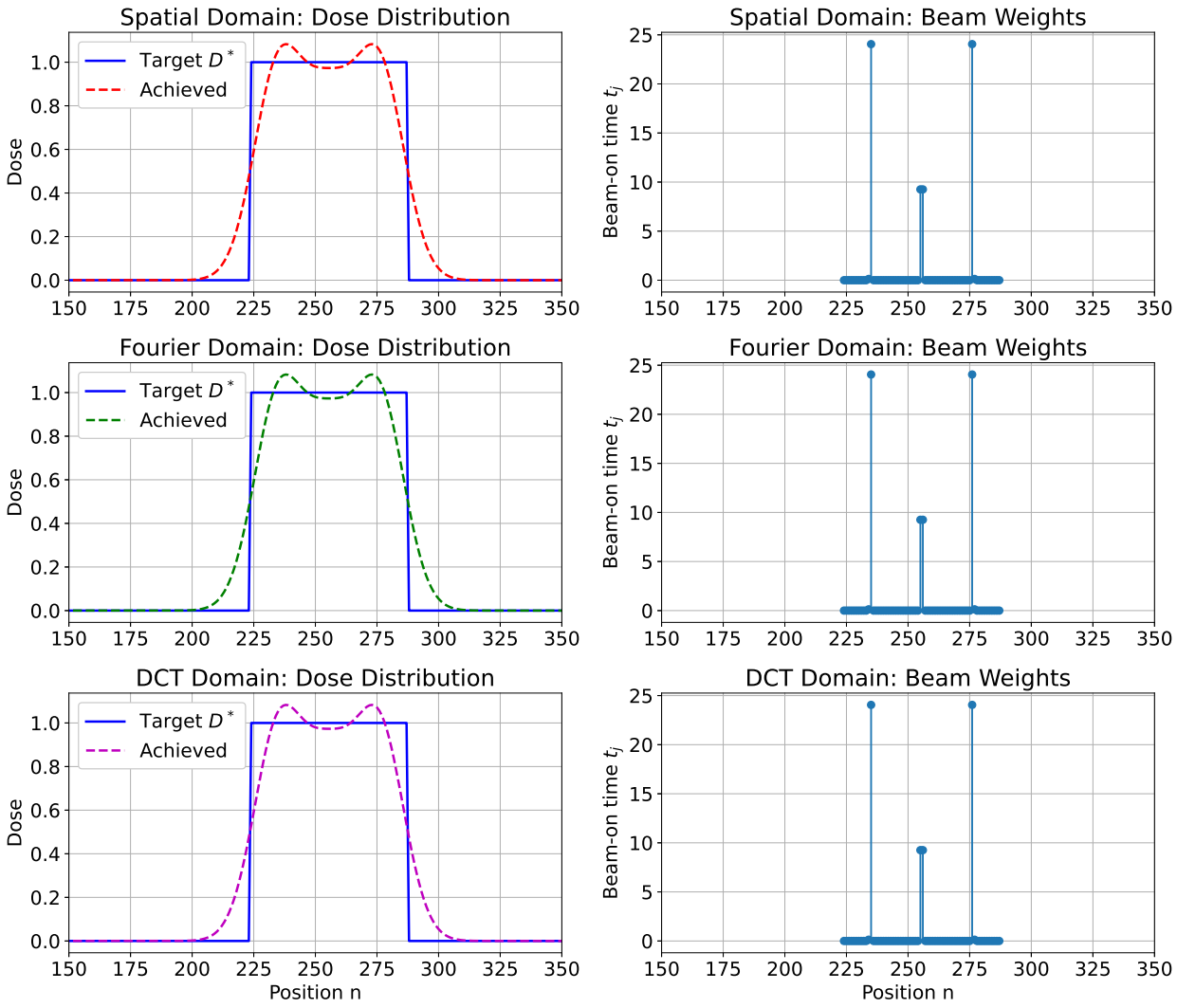

Figure 6: Comparison of optimization in spatial, Fourier, and DCT domains. Left column shows target dose distribution D^* versus achieved dose. Right column shows optimal beam-on times t_j for each method.

The following Python code was used for calculations:

```
# Fourier Domain
D_star_fft = fft(D_star)
A_fft = np.zeros((N, n_shots), dtype=complex)
for j in range(n_shots):
    gauss = norm.pdf(np.arange(N), tumor_start+j, sigma)
    A_fft[:, j] = fft(gauss)

# Stack real/imag for NNLS
```

```
A_stacked = np.vstack([A_fft.real, A_fft.imag])
D_stacked = np.hstack([D_star_fft.real,
                       D_star_fft.imag])
t_fft, _ = nnls(A_stacked, D_stacked)
dose_fft = np.real(ifft(A_fft @ t_fft))

# DCT Domain
D_star_dct = dct(D_star, type=2, norm='ortho')
A_dct = np.zeros((N, n_shots))
for j in range(n_shots):
    gauss = norm.pdf(np.arange(N), tumor_start+j, sigma)
    A_dct[:, j] = dct(gauss, type=2, norm='ortho')

t_dct, _ = nnls(A_dct, D_star_dct)
dose_dct = idct(A_dct @ t_dct, type=2, norm='ortho')
```

(c) One of issues of the above optimization is that the ideal dose distribution D^* has a perfectly sharp edge, which is impossible to obtain from Gaussian shots. To overcome this, one idea is to blur the sharp edges of the dose distribution D^* by removing its high frequency components that is not deliverable by a Gaussian distribution. Please implement this approach and see if it produces any improvement.

Solution: The original signal D^* can be filtered in a way that is easier to optimize for and more realistic for Gaussian shots to actually deliver. This is done by analyzing the frequency spectrum of a Gaussian shot to identify a natural cutoff frequency (where the Gaussian's energy drops below 1% of its maximum). Using this cutoff, we transform D^* into the frequency domain and remove all components that are beyond that cutoff. Transforming back into the spatial domain then gives a smooth filtered version of D^*.

After filtering $D*$, optimization usin Fourier Transformation had an L2 error of 1.011801, which is an improvement. For Discrete Cosine Transform the L2 error was 2.547098, more than without filtering.

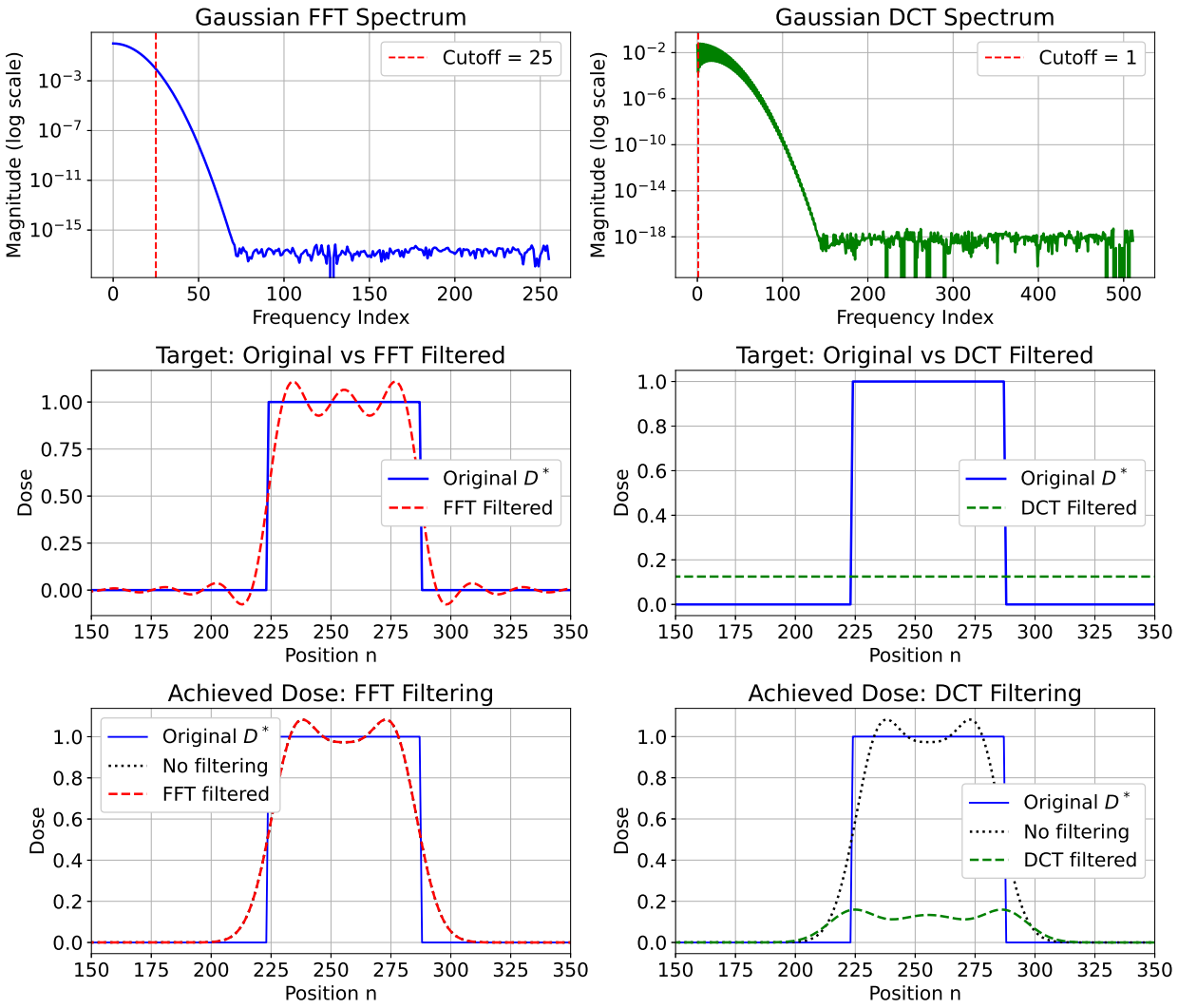

Figure 7: Problem 6(c): Pre-filtering approach. (Top row) Frequency spectrum of a representative Gaussian showing natural cutoff frequencies. (Middle row) Original target D^* versus filtered targets using FFT and DCT. (Bottom row) Comparison of achieved dose distributions with and without pre-filtering.

The following Python code was used for calculations:

```
# Analyze Gaussian frequency content
gauss = norm.pdf(np.arange(N), loc=256, scale=sigma)
gauss_fft = np.abs(fft(gauss))
gauss_dct = np.abs(dct(gauss, type=2, norm='ortho'))

# Find cutoff (1% of max energy)
fft_cutoff = np.where(gauss_fft
```

```
                        < 0.01*np.max(gauss_fft))[0][0]
dct_cutoff = np.where(gauss_dct
                        < 0.01*np.max(gauss_dct))[0][0]

# FFT Filtering
D_fft = fft(D_star)
D_fft[fft_cutoff:N-fft_cutoff+1] = 0
D_filtered_fft = np.real(ifft(D_fft))

# DCT Filtering
D_dct = dct(D_star, type=2, norm='ortho')
D_dct[dct_cutoff:] = 0
D_filtered_dct = idct(D_dct, type=2, norm='ortho')

# Optimize with filtered targets
t_fft, _ = nnls(A_spatial, D_filtered_fft)
t_dct, _ = nnls(A_spatial, D_filtered_dct)
```