ELEC4620 Assignment 1

Deren Teo

August 9, 2023

Question 1

The function for a rectangular pulse around $t = 0$, with amplitude A and width T, is:

$$h(t) = \begin{cases} A, & |t| < T/2 \\ 0, & |t| > T/2 \end{cases}$$

This is equivalently two step functions of equal magnitude and opposite sign at $t = \pm T/2$. Hence, the derivative of the rectangular pulse is composed of two impulses of equal magnitude and opposite sign, coinciding in time with the discontinuities in the pulse.

$$h'(t) = A\delta(t + \frac{T}{2}) - A\delta(t - \frac{T}{2})$$

Taking the Fourier transform of the derivative, which by definition is

$$\widehat{H'}(f) \coloneqq \int_{-\infty}^{\infty} h'(t)e^{-j2\pi ft}dt$$

we aim to evaluate the following expression, split into two integrals for simplicity.

$$\widehat{H'}(f) = A\int_{-\infty}^{\infty} \delta(t + \frac{T}{2})e^{-j2\pi ft}dt - A\int_{-\infty}^{\infty} \delta(t - \frac{T}{2})e^{-j2\pi ft}dt$$

By definition of the Dirac delta, $\delta(t - T)$, for arbitrary T:

$$\delta(t - T) = \begin{cases} \infty, & t = T \\ 0, & t \neq T \end{cases} \qquad \text{and} \qquad \int_{-\infty}^{\infty} \delta(t - T)dt = 1$$

Therefore, the Fourier transform of the derivative simplifies to

$$\widehat{H'}(f) = Ae^{-j2\pi f(-T/2)} - Ae^{-j2\pi f(T/2)} = A\left[e^{j\pi fT} - e^{-j\pi fT}\right]$$

Finally, we can integrate in the time domain by dividing by $j2\pi f$ in the frequency domain.

$$H(f) = \frac{A}{j2\pi f}\left[e^{j\pi fT} - e^{-j\pi fT}\right]$$

Some re-arranging and substitutions can be performed to neaten the result, if desired:

$$H(f) = \frac{A}{\pi f}\sin(\pi Tf) = AT\frac{\sin(\pi Tf)}{\pi Tf} = AT\text{sinc}(Tf)$$

Thus, we have derived the Fourier transform of a rectangular pulse.

We now repeat this procedure for a triangle function using a double derivative. The function for a triangular pulse around $t = 0$, with amplitude A and width T, is:

$$h(t) = \begin{cases} A(1 - 2|t|/T), & |t| \leq T/2 \\ 0, & |t| > T/2 \end{cases}$$

The first derivative produces a result composed of two rectangular pulses of equal magnitude and opposite sign, or equivalently three step functions.

$$h'(t) = \begin{cases} 2A/T, & -T/2 \leq t \leq 0 \\ -2A/T, & 0 \leq t \leq T/2 \\ 0, & |t| > T/2 \end{cases}$$

Hence, as before, the second derivative is composed of three impulses coinciding with the discontinuities in the first derivative.

$$h''(t) = \frac{2A}{T}\delta(t + \frac{T}{2}) - \frac{4A}{T}\delta(t) + \frac{2A}{T}\delta(t - \frac{T}{2})$$

The Fourier transform of the second derivative is therefore

$$\widehat{H''}(f) = \frac{2A}{T}\int_{-\infty}^{\infty}\delta(t + \frac{T}{2})e^{-j2\pi ft}dt - \frac{4A}{T}\int_{-\infty}^{\infty}\delta(t)e^{-j2\pi ft}dt + \frac{2A}{T}\int_{-\infty}^{\infty}\delta(t - \frac{T}{2})e^{-j2\pi ft}dt$$

Once again, using the definition of the Dirac delta, the Fourier transform simplifies to

$$\widehat{H''}(f) = \frac{2A}{T}\left[e^{-j2\pi f(-T/2)} - 2e^{-j2\pi f(0)} + e^{-j2\pi f(T/2)}\right]$$

and further to

$$\widehat{H''}(f) = \frac{2A}{T}\left[e^{j\pi fT} - 2 + e^{-j\pi fT}\right]$$

We can integrate twice in the time domain by dividing by $(j2\pi f)^2$ in the frequency domain.

$$H(f) = \frac{2A}{(j2\pi f)^2 T}\left[e^{j\pi fT} - 2 + e^{-j\pi fT}\right] = \frac{-A}{2\pi^2 f^2 T}\left[e^{j\pi fT} - 2 + e^{-j\pi fT}\right]$$

Finally, as with the rectangular pulse, we can re-arrange this result into a more familiar form:

$$H(f) = \frac{A}{\pi^2 f^2 T}(1 - \cos(\pi Tf))$$

Thus, we have derived the Fourier transform of a triangular pulse.

Having derived the Fourier transforms of the two functions, we are interested in comparing the rates at which their magnitudes decrease as frequency increases. We note the only term contributing to a change in magnitude with frequency is the $1/f$ term for the rectangular pulse, and the $1/f^2$ term for the triangular pulse.

Indeed, in general, if a function has discontinuities in the n^{th} derivative, the sidelobes of its Fourier transform will fall off as $1/f^{n+1}$. Intuitively, this is because the function must be derived $n + 1$ times to obtain a number of impulses which can be Fourier transformed without yielding any frequency-dependent coefficients. The transform of the derivative is then integrated $n + 1$ times by dividing by $(j2\pi f)^{n+1}$; hence, the term $1/f^{n+1}$ is produced.

Figure 1.1 presents the Fourier transforms of the rectangular and triangular pulses, enabling a visual comparison of the rates at which their sidelobes fall off.

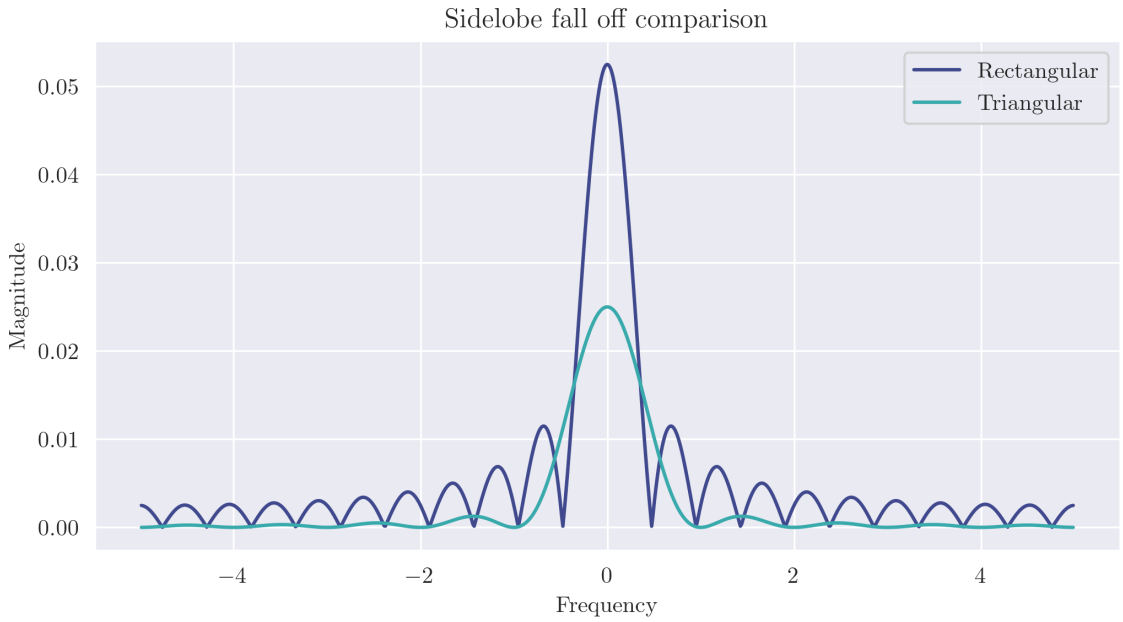

Figure 1.1: Fourier transform sidelobe fall off comparison: rectangular and triangular pulses.

Question 2

Denote the given polynomials in z by $X(z)$ and $Y(z)$, as follows:

$$X(z) = 1 + 2z^{-1} + 6z^{-2} + 11z^{-3} + 15z^{-4} + 12z^{-5}$$
$$Y(z) = 1 - 3z^{-1} - 3z^{-2} + 7z^{-3} - 7z^{-4} + 3z^{-5}$$

Their corresponding vectors are constructed from their respective coefficients:

$$v_X = [1, 2, 6, 11, 15, 12] \qquad \text{and} \qquad v_Y = [1, -3, -3, 7, -7, 3]$$

The result of multiplying $X(z)$ and $Y(z)$ can be obtained by convolving their respective vectors and interpreting the outcome as the coefficients of the polynomial product. That is,

$$v_X * v_Y = [1, -1, -3, -6, -29, -35, -40, 10, 12, -39, 36]$$

This can be calculated using the `convolve` function from the Python `scipy.signal` library:

```
signal.convolve(vx, vy, mode="full", method="direct")
```

which calculates the full discrete linear convolution, automatically zero-padding the vectors as necessary, using traditional convolution (i.e. multiplying and summing, as opposed to the FFT).

Hence, we can interpret the convolution result as the polynomial product of $X(z)$ and $Y(z)$ as

$$1 - z^{-1} - 3z^{-2} - 6z^{-3} - 29z^{-4} - 35z^{-5} - 40z^{-6} + 10z^{-7} + 12z^{-8} - 39z^{-9} + 36z^{-10}$$

Since convolution is equivalent to multiplication in the Fourier domain, we could equivalently Fourier transform both vectors, multiply in the Fourier domain, then perform an inverse Fourier transform to obtain the same vector of coefficients derived above.

When doing this in Python, we must manually zero-pad the vectors before performing the FFT.

```
vx = np.pad(vx, (0, len(vy) - 1))
vy = np.pad(vy, (0, len(vx) - 1))
```

Here, the Python `numpy` package is used to zero-pad both vectors on the right side only to the appropriate length. Then, `fft` and `ifft` from `scipy.fft` can be applied:

```
ifft(fft(vx) * fft(vy))
```

This calculates the following vector, which we can observe is identical to the vector determined through direct convolution:

```
[ 1.  -1.  -3.  -6.  -29.  -35.  -40.  10.  12.  -39.  36.]
```

Hence, the same polynomial product can be constructed from either convolving the vector representations of the polynomials, or multiplying the Fourier transforms of the vectors, then taking the inverse Fourier transform.

Question 3

Given the following numbers in base 10, we seek to apply similar methods to Question 2 to multiply the numbers, first using convolution, then using Fourier transform techniques.

$$x = 8755790 \qquad \text{and} \qquad y = 1367267$$

First, however, we can perform regular multiplication to determine the correct answer that we should get from the convolution and Fourier transform methods:

$$8755790 \times 1367267 = 11971502725930$$

Next, the numbers are constructed digit-wise into vectors:

$$v_x = [8, 7, 5, 5, 7, 9, 0] \qquad \text{and} \qquad v_y = [1, 3, 6, 7, 2, 6, 7]$$

As before, the result of multiplying x and y can be obtained by convolving their respective vectors. However, the "carry" step must then be performed to produce a number in base 10.

Once again, using `convolve` from `scipy.signal` with the same keyword arguments:

$$v_x * v_y = [8, 31, 74, 118, 117, 157, 212, 192, 142, 95, 103, 63, 0]$$

Now, however, unlike with the polynomial multiplication, we cannot expect to arrive at the correct value by simply stringing together all of these digits. Instead, starting from the right, each value must be taken modulo 10, and the remainder added to the value immediately to the left. This yields the following vector, which now can be concatenated into the result of $x \times y$:

$$[1, 1, 9, 7, 1, 5, 0, 2, 7, 2, 5, 9, 3, 0] \longrightarrow 11971502725930$$

Naturally, the same outcome can be achieved by Fourier transforming the vectors, multiplying in the Fourier domain, then inverse Fourier transforming the result. Again, in Python, the vectors must be zero-padded before performing the FFT.

```
vx = np.pad(vx, (0, len(vy) - 1))
vy = np.pad(vy, (0, len(vx) - 1))
```

Then, using the same Python function calls as from Question 2:

`ifft(fft(vx) * fft(vy)) =` [8. 31. 74. 118. 117. 157. 212. 192. 142. 95. 103. 63. 0.]

The result is identical to the vector produced by convolution. Therefore, applying the same "carrying" process will yield the same number result.

Hence, integer multiplication also can be accomplished by either convolving the vector representations of the numbers and converting to base 10, or multiplying the Fourier transforms of the vectors, then taking the inverse Fourier transform and converting to base 10.

Question 4

a) First, we individually transform the sequences using the `fft` function from `scipy.fft`:

$$\begin{aligned}\texttt{fft(x)} &= [34,\ -1.879 + j6.536,\ -5 + j7,\ -6.121 + j0.536,\\ &\quad 0,\ -6.121 - j0.536,\ -5 - j7,\ -1.879 - j6.536]\\ \texttt{fft(y)} &= [28,\ 2.243 + j4.243,\ -2 - j2,\ -6.243 + j4.243,\\ &\quad -8,\ -6.243 - j4.243,\ -2 + j2,\ 2.243 - j4.243]\end{aligned}$$

Now, we combine x and y element-wise into a single complex vector:

$$z = [1 + j \quad 2 + j5 \quad 4 + j3 \quad 4 + j \quad 5 + j3 \quad 3 + j5 \quad 7 + j3 \quad 8 + j7]$$

and Fourier transform this to obtain:

$$\begin{aligned}\texttt{fft(z)} = [&34 + j28,\ -6.121 + j8.778,\ -3 + j5,\ -10.364 - j5.707,\\ &-j8,\ -1.879 - j6.778,\ -7 - j9,\ 2.364 - j4.293]\end{aligned}$$

The Fourier transforms of x and y can be determined from the Fourier transform of z as

$$\begin{aligned}X &= \text{Ev}(\text{Re}(Z)) + j\text{Od}(\text{Im}(Z))\\ Y &= \text{Ev}(\text{Im}(Z)) - j\text{Od}(\text{Re}(Z))\end{aligned}$$

where Z is the Fourier transform of z. Since x is purely real and y purely imaginary, the Fourier transform of x has a purely even real component and purely odd imaginary component, and vice versa for the Fourier transform of y. Even and odd components are orthogonal; hence, X and Y can be independently reconstructed.

The even and odd components of a sequence $H(n)$ are determined as:

$$\text{Ev}(n) = \frac{H(n) + H(-n)}{2} \qquad\qquad \text{Od}(n) = \frac{H(n) - H(-n)}{2}$$

where $H(-n)$ is the vector H with all elements after the first in reversed order. Hence,

$$\begin{aligned}\text{Ev}(\text{Re}(Z)) &= [34,\ -1.879,\ -5,\ -6.121,\ 0,\ -6.121,\ -5,\ -1.879]\\ \text{Od}(\text{Im}(Z)) &= [0,\ 6.536,\ 7,\ 0.536,\ 0,\ -0.536,\ -7,\ -6.535]\\ \text{Ev}(\text{Im}(Z)) &= [28,\ 2.243,\ -2,\ -6.243,\ -8,\ -6.243,\ -2,\ 2.243]\\ \text{Od}(\text{Re}(Z)) &= [0,\ -4,\ 243,\ 2,\ -4,243,\ 0,\ 4.243,\ -2,\ 4.243]\end{aligned}$$

wherein from the second element of each vector onward, the even and odd symmetries can be observed. Finally, we can reconstruct the individual Fourier transforms of x and y:

$$\begin{aligned}X = [&34,\ -1.879 + j6.536,\ -5 + j7,\ -6.121 + j0.536,\\ &0,\ -6.121 - j0.536,\ -5 - j7,\ -1.879 - j6.536]\\ Y = [&28,\ 2.243 + j4.243,\ -2 - j2,\ -6.243 + j4.243,\\ &-8,\ -6.243 - j4.243,\ -2 + j2,\ 2.243 - j4.243]\end{aligned}$$

Comparing these vectors to those determined individually at the start, we can see they are identical. Therefore, we have shown that the double transform algorithm gives the same answer as directly transforming the sequences.

b) In the previous part, the double transform algorithm was applied to sequences of equal length. However, it is also applicable to sequences of unequal length by right-padding the shorter sequence with zero. For example,

$$x = [1\ 2\ 4\ 4\ 5\ 3\ 7\ 8] \qquad\qquad y = [1\ 5\ 3\ 1\ 3\ 5\ 3\ 0]$$

The individual Fourier transforms of x and y are:

$$\begin{aligned}
\texttt{fft(x)} = [&34,\ -1.879 + j6.536,\ -5 + j7,\ -6.121 + j0.536\\
&0,\ -6.121 - j0.536,\ -5 - j7,\ -1.879 - j6.536]\\
\texttt{fft(y)} = [&21,\ -2.707 - j0.707,\ -2 - j9,\ -1.293 - j0.707\\
&-1,\ -1.293 + j0.707,\ -2 + j9,\ -2.707 + j0.707]
\end{aligned}$$

As before, we combine x and y element-wise into a single complex vector, but this time we must right-pad y with zero such that the vectors are equal length.

$$z = [1 + j\ \ 2 + j5\ \ 4 + j3\ \ 4 + j\ \ 5 + j3\ \ 3 + j5\ \ 7 + j3\ \ 8 + j0]$$

The Fourier transform of z is

$$\begin{aligned}
\texttt{fft(z)} = [&34 + j21,\ -1.172 + j3.828,\ 4 + j5,\ -5.414 - j0.757,\\
&-j,\ -6.828 - j1.828,\ -14 - j9,\ -2.586 - j9.243]
\end{aligned}$$

Again, the Fourier transforms of x and y can individually be determined from Z using the even and odd components of the real and imaginary parts of Z, per part (a). Hence,

$$\begin{aligned}
\text{Ev}(\text{Re}(Z)) &= [34,\ -1.879,\ -5,\ -6.121,\ 0,\ -6.121,\ -5,\ -1.879]\\
\text{Od}(\text{Im}(Z)) &= [0,\ 6.536,\ 7,\ 0.536,\ 0,\ -0.536,\ -7,\ -6.536]\\
\text{Ev}(\text{Im}(Z)) &= [0,\ 0.707,\ 9,\ 0.707,\ 0,\ -0.707,\ -9,\ -0.707]\\
\text{Od}(\text{Re}(Z)) &= [21,\ -2.707,\ -2,\ -1.293,\ -1,\ -1.293,\ -2,\ -2.707]
\end{aligned}$$

Part part (a), we can reconstruct the individual Fourier transforms of x and y:

$$\begin{aligned}
X = [&34,\ -1.879 + j6.536,\ -5 + j7,\ -6.121 + j0.536\\
&0,\ -6.121 - j0.536,\ -5 - j7,\ -1.879 - j6.536]\\
Y = [&21,\ -2.707 - j0.707,\ -2 - j9,\ -1.293 - j0.707\\
&-1,\ -1.293 + j0.707,\ -2 + j9,\ -2.707 + j0.707]
\end{aligned}$$

Comparing these vectors to those determined individually at the start, we can see they are identical. We can additionally check that the shorter sequence can be recovered using the inverse Fourier transform:

```
ifft(Y) = [ 1.  5.  3.  1.  3.  5.  3.  -0.]
```

Hence, given we know how many zeros were padded onto the end of the shorter sequence, it can indeed be recovered. Therefore, the double transform algorithm can be applied even if the sequences differ in length.

Question 5

a) Given the polynomial

$$F(z) = 1 + 5z^{-1} + 3z^{-2} + 4z^{-3} + 4z^{-4} + 2z^{-5} + z^{-6}$$

the absence of a denominator indicates an all-zero model. The positions of the zeros in the complex plane are the roots of $F(z)$, which can be found using `numpy.polynomial`:

```
Polynomial([1, 5, 3, 4, 4, 2, 1]).roots()
```

This finds the following six roots:

$$z = -1.332,\ -0.628 \pm j1.565,\ -0.223,\ 0.405 \pm j1.011$$

Figure 5.1 plots these on the complex plane, with the unit circle for reference.

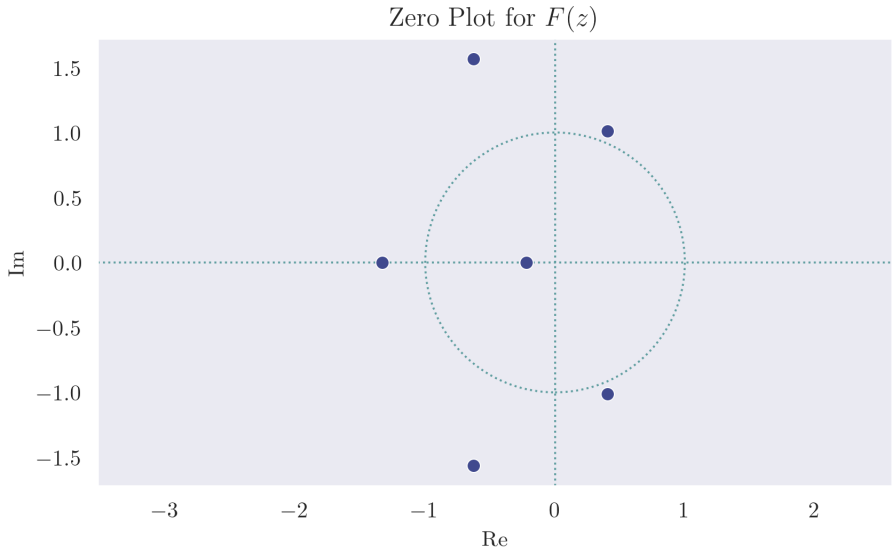

Figure 5.1: Zero plot for $F(z)$; no poles are present.

b) The sinusoidal steady-state response of the system can be modelled by evaluating the magnitude of $F(z)$ around the unit circle. This is effectively what is done by the DFT.

That is,

$$F(z)\big|_{z=e^{-j\frac{2\pi}{N}k},\ k=0,\ldots,N-1}$$

Figure 5.2 compares `scipy.fft` and a self-implemented DFT function for $N = 128$.

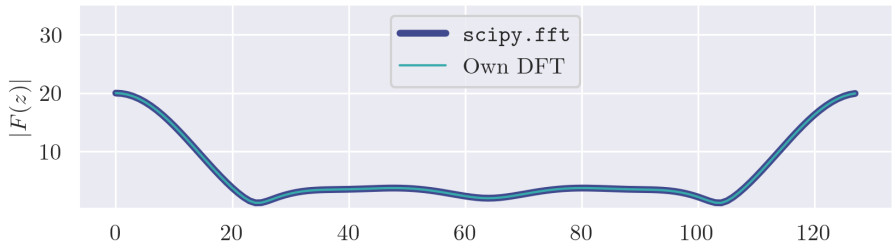

Figure 5.2: $|F(z)|$ evaluated at 128 points around the unit circle.

Appendix

A sidelobes.py

```
"""
Script associated with Q1.

Plots Fourier transform of rectangular and triangular pulse functions, enabling
comparison of rates at which sidelobes fall off.
"""

from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

from scipy.fft import fft, fftfreq

from config import A1_ROOT, SAVEFIG_CONFIG

### DSP FUNCTIONS ##################################################################

def rectangular_pulse(_t: np.array, A: float = 1, T: float = 1) -> np.array:
    """
    Constructs a rectangular pulse of amplitude 'A' and width 'T', centred about
    t=0 on '_t'. Returns array of y-values corresponding to '_t'.
    """
    return A * (np.abs(_t) <= T)

def triangular_pulse(_t: np.array, A: float = 1, T: float = 1) -> np.array:
    """
    Constructs a triangular pulse of maximum amplitude 'A' and width 'T',
    centred about t=0 on '_t'. Returns array of y-values corresponding to '_t'.
    """
    return A * (1 - np.abs(_t) / T) * (np.abs(_t) <= T)

### ENTRYPOINT #####################################################################

def main():

    N = 800      # Number of sample points
    T = 1e-1     # Sample spacing

    _t = np.linspace(-N*T/2, N*T/2, N, endpoint=False)
    _f = fftfreq(N, T)

    _y1 = rectangular_pulse(_t)
    _H1 = 2 / N * np.abs(fft(_y1))

    _y2 = triangular_pulse(_t)
    _H2 = 2 / N * np.abs(fft(_y2))

    fig, ax = plt.subplots(figsize=(8, 4))

    sns.lineplot(x=_f, y=_H1, ax=ax, label="Rectangular")
    sns.lineplot(x=_f, y=_H2, ax=ax, label="Triangular")

    ax.set_title("Sidelobe fall off comparison")
    ax.set_xlabel("Frequency")
    ax.set_ylabel("Magnitude")

```

```
59      fname = Path(A1_ROOT, "output", "q1_sidelobes.png")
60      fig.savefig(fname, **SAVEFIG_CONFIG)
61
62
63  if __name__ == "__main__":
64      main()
```

B multiplying.py

```
"""
Script associated with Q2 and 3.

Multiplies polynomials/numbers in vector representation using convolution and
Fourier transform methods.
"""

import numpy as np

from scipy.signal import convolve
from scipy.fft import fft, ifft

### UTILITY FUNCTIONS ###########################################################

def multiply_carry(z: np.array) -> np.array:
    """
    Perform the "carry" steps of the multiplication process. Starting from the
    right end of `z`, each digit is taken modulo 10 and the remainder is added
    to the value immediately to the left. Returns an array of single digits,
    possibly except the first value (though the answer will still be correct).
    """
    r = 0; ret = []
    for n in z[::-1]:
        n += r
        ret.append(n % 10)
        r = n // 10
    ret.append(r)
    return np.array(ret[::-1])

### DSP FUNCTIONS ###############################################################

def multiply_conv(x: np.array, y: np.array, carry: bool = False) -> np.array:
    """
    Convolve arrays `x` and `y` using direct convolution. If `carry` is True,
    the output array is modified to be equivalent to the vectorised integer
    product of vectorised integers `x` and `y`. Otherwise, the convolution
    result is returned "as-is".
    """
    z = convolve(x, y, mode="full", method="direct")
    if carry:
        z = multiply_carry(z)
    return z

def multiply_fft(x: np.array, y: np.array, carry: bool = False) -> np.array:
    """
    Convolve arrays `x` and `y` by performing the FFT on `x` and `y`,
    multiplying the results, then performing the inverse FFT. If `carry` is
    True, the output array is modified to be equivalent to the vectorised
    integer product of vectorised integers `x` and `y`. Otherwise, the
    convolution result is returned "as-is".
    """
    xpad = np.pad(x, (0, len(y) - 1))
    ypad = np.pad(y, (0, len(x) - 1))
    z = ifft(fft(xpad) * fft(ypad)).real
    if carry:
        z = multiply_carry(z)
    return z

### ENTRYPOINT ##################################################################

def main():
```

```
62
63      ### QUESTION 2 ###
64
65      vx = np.array([1, 2, 6, 11, 15, 12])
66      vy = np.array([1, -3, -3, 7, -7, 3])
67
68      print("Q2 by convolution:\n", multiply_conv(vx, vy))
69      print("Q2 by FFT and IFFT:\n", multiply_fft(vx, vy))
70
71      ### QUESTION 3 ###
72
73      vx = np.array([8, 7, 5, 5, 7, 9, 0])
74      vy = np.array([1, 3, 6, 7, 2, 6, 7])
75
76      print("Q3 by multiplication:", 8755790 * 1367267)
77      print("Q3 by convolution:", multiply_conv(vx, vy, carry=True))
78      print("Q3 by FFT and IFFT:", multiply_fft(vx, vy, carry=True))
79
80
81  if __name__ == "__main__":
82      main()
```

C double_transform.py

```
"""
Script associated with Q4.

Implementation of the double transform algorithm to Fourier transform two real
N-point sequences using one commplex N-point transform.
"""

import numpy as np

from scipy.fft import fft, ifft

### DSP FUNCTIONS ###################################################################

def ev(H: np.array) -> np.array:
    """
    Returns the even component of the given sequence 'H'.
    """
    H_minus = np.concatenate([H[:1], H[-1:0:-1]])
    return 0.5 * (H + H_minus)

def od(H: np.array) -> np.array:
    """
    Returns the odd component of the given sequence 'H'.
    """
    H_minus = np.concatenate([H[:1], H[-1:0:-1]])
    return 0.5 * (H - H_minus)

### ENTRYPOINT ######################################################################

def main():

    x = np.array([1, 2, 4, 4, 5, 3, 7, 8])
    # y = np.array([1, 5, 3, 1, 3, 5, 3, 7])  # PART A: comment out for the other
    y = np.array([1, 5, 3, 1, 3, 5, 3, 0])  # PART B: comment out for the other

    Z = fft(np.array([a+b*1j for a, b in zip(x, y)]))
    X = ev(np.real(Z)) + 1j * od(np.imag(Z))
    Y = ev(np.imag(Z)) - 1j * od(np.real(Z))

    print(f"{fft(x) = }")
    print(f"{fft(y) = }")

    print(f"{Z = }")

    print(f"{ev(np.real(Z)) = }")
    print(f"{od(np.imag(Z)) = }")
    print(f"{od(np.real(Z)) = }")
    print(f"{ev(np.imag(Z)) = }")

    print(f"{X = }")
    print(f"{Y = }")

    print(f"ifft(Y) = {np.round(ifft(Y).real, 3)}")


if __name__ == "__main__":
    main()
```

D polezero_dft.py

```
"""
Script associated with Q5.

Determines the roots of a certain polynomials and produces a pole-zero plot.
Evaluates the magnitude of the polynomial around the unit circle using the DFT.
"""

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

from pathlib import Path

from matplotlib.axes._axes import Axes
from matplotlib.lines import Line2D
from matplotlib.patches import Circle
from numpy.polynomial.polynomial import Polynomial, polyval
from scipy.fft import fft

from config import A1_ROOT, PLT_CONFIG, SAVEFIG_CONFIG

### DSP FUNCTIONS ##########################################################

def zdft(poly_coef: np.array, N: int) -> np.array:
    """
    Computes the 1D `n`-point discrete Fourier transform of some sequence from
    its Z transform, given by `poly`.
    """
    return np.array([polyval(np.exp(-1j*2*np.pi*k/N), poly_coef) for k in range(N)])

### VISUALISATION ##########################################################

def plot_poles_or_zeros(F: Polynomial, type: str, ax: Axes) -> Axes:
    """
    Plots the roots of the polynomial in the complex plane on the given axes.
    """
    roots = F.roots()

    marker = {"poles": "X", "zeros": "o"}[type]
    sns.scatterplot(x=np.real(roots), y=np.imag(roots), ax=ax, marker=marker)

    ax.set_xlabel("Re")
    ax.set_ylabel("Im")

    return ax

def axes_ratio_scale(ax: Axes, ratio: float, padto: str = None) -> Axes:
    """
    Sets axes aspect as equal and autoscales the axes. If the axes limits ratio
    does not match the given aspect ratio (i.e. the ratio height / width), the
    x- or y-axis is lengthened to the desired ratio. Returns the modified axes.
    """
    if padto and padto not in ("upper", "lower", "left", "right", "center"):
        raise ValueError("invalid 'padto' specified")
    padto = padto or "center"

    ax.set_aspect("equal")
    ax.autoscale()

    xlim, ylim = ax.get_xlim(), ax.get_ylim()
    xrng, yrng = xlim[1] - xlim[0], ylim[1] - ylim[0]
```

```
62      curr_ratio = yrng / xrng
63
64      if curr_ratio > ratio: # i.e. the current ratio is too tall and narrow
65          add_xlim = (yrng / ratio - xrng) * 0.5
66          if padto == "right":
67              new_xlim = (xlim[0], xlim[1] + 2 * add_xlim)
68          elif padto == "left":
69              new_xlim = (xlim[0] - 2 * add_xlim, xlim[1])
70          else:
71              new_xlim = (xlim[0] - add_xlim, xlim[1] + add_xlim)
72          ax.set_xlim(new_xlim)
73
74      if curr_ratio < ratio: # i.e. the current ratio is too short and wide
75          add_ylim = (xrng * ratio - yrng) * 0.5
76          if padto == "upper":
77              new_ylim = (ylim[0], ylim[1] + 2 * add_ylim)
78          elif padto == "lower":
79              new_ylim = (ylim[0] - 2 * add_ylim, ylim[1])
80          else:
81              new_ylim = (ylim[0] - add_ylim, ylim[1] + add_ylim)
82          ax.set_ylim(new_ylim)
83
84      return ax
85
86  def draw_unit_circle(ax: Axes) -> Axes:
87      """
88      Draws dotted axes and unit circle on the given axes, similar in style to
89      MATLAB's zplane function.
90      """
91      style_config = {"ls": "dotted", "lw": 0.9, "color": "cadetblue", "zorder": 0}
92
93      u_circ = Circle(xy=(0, 0), radius=1, fill=False, **style_config)
94      ax.add_patch(u_circ)
95
96      x_axis = Line2D(xdata=ax.get_xlim(), ydata=(0, 0), **style_config)
97      y_axis = Line2D(xdata=(0, 0), ydata=ax.get_ylim(), **style_config)
98      ax.add_line(x_axis)
99      ax.add_line(y_axis)
100     ax.set_aspect("equal")
101
102     return ax
103
104 ### SUBPARTS ########################################################################
105
106 def run_part_a(F: Polynomial) -> None:
107     """
108     Plots the roots of the polynomial with given coefficients on the complex
109     plane, with a unit circle underlay.
110     """
111     print(f"{F.roots() = }")
112
113     # Override default style to hide grid
114     sns.set_style("dark")
115
116     # Re-set the plot text customisation, which gets overriden by set_style
117     plt.rcParams.update(PLT_CONFIG)
118
119     fig, ax = plt.subplots()
120
121     ax = plot_poles_or_zeros(F, "zeros", ax)
122     ax = axes_ratio_scale(ax, ratio=9/16, padto="center")
123     ax = draw_unit_circle(ax)
124     ax.set_title("Zero Plot for $F(z)$")
```

```
125
126     fname = Path(A1_ROOT, "output", "q5a_polezero.png")
127     fig.savefig(fname, **SAVEFIG_CONFIG)
128
129 def run_part_b(poly: Polynomial) -> None:
130     """
131     Plots the magnitude of the polynomial with the given coefficients at 128
132     uniformly spaced points around the unit circle using the DFT.
133     """
134     y_fft = np.abs(fft(poly.coef, n=128))
135     y_dft = np.abs(zdft(poly.coef, N=128))
136
137     fig, ax = plt.subplots()
138
139     sns.lineplot(x=np.arange(128), y=y_fft, ax=ax, lw=3, label=r"$\texttt{scipy.fft}$")
140     sns.lineplot(x=np.arange(128), y=y_dft, ax=ax, lw=1, label=r"Own_DFT")
141
142     ax = axes_ratio_scale(ax, ratio=1/4, padto="upper")
143
144     ax.set_title("")
145     ax.set_xlabel("")
146     ax.set_ylabel("$|F(z)|$")
147
148     ax.legend(loc="upper_center")
149
150     fname = Path(A1_ROOT, "output", "q5b_dftsample.png")
151     fig.savefig(fname, **SAVEFIG_CONFIG)
152
153 ### ENTRYPOINT ###################################################################
154
155 def main():
156
157     poly = Polynomial([1, 5, 3, 4, 4, 2, 1])
158     # run_part_a(poly)
159     run_part_b(poly)
160
161
162 if __name__ == "__main__":
163     main()
```