ELEC4620 Assignment 1

Deren Teo

August 13, 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.

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

Figure 1.1 visualises the rectangular pulse and its first derivative.

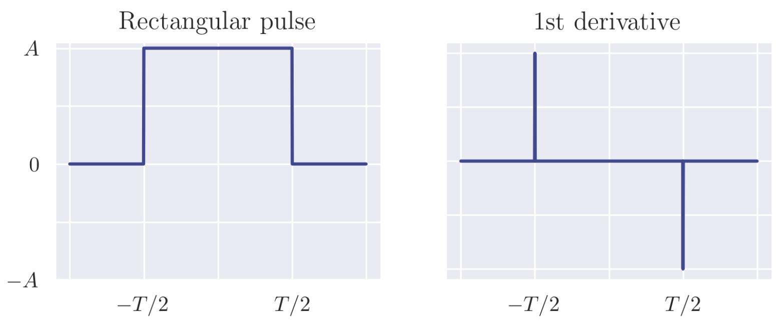

Figure 1.1: A rectangular pulse and its first derivative

Writing out the Fourier transform of the derivative, which by definition is

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

we get the following expression, which has been 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.

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

Figure 1.2 visualises the triangular pulse and its first and second derivatives.

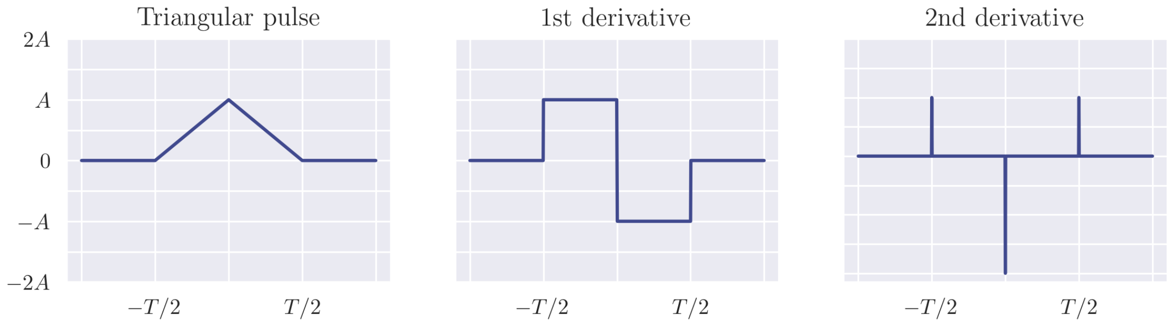

Figure 1.2: A triangular pulse and its first and second derivatives

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:

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

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

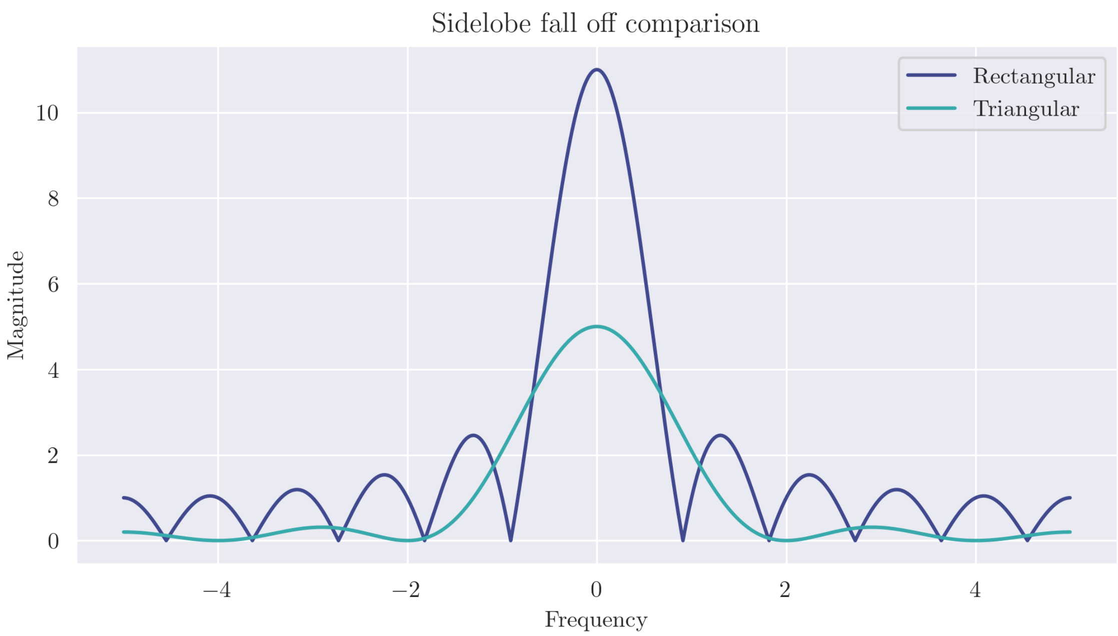

Figure 1.3: Fourier transform sidelobe fall off comparison: rectangular and triangular pulses.

The full python code for this question, and all following questions, can be found in the Appendix.

Question 2

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

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

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,

$$X(z)Y(z) = \sum_{i=0}^{M+N-1} (v_X * v_Y)_i z^{-i}$$

where M and N are the lengths of v_X and v_Y, respectively, and $(v_X * v_Y)_i$ denotes the i^{th} element of the vector produced by the convolution of v_X with v_Y. In terms of the latter, $v_X * v_Y$ can be calculated using the `convolve` function from the `scipy.signal` library:

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

This 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).

The result is:

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

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.

To demonstrate this in Python, we 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 `numpy` package is used to zero-pad both vectors to the right 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, as expected, is identical to the vector determined through direct convolution:

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

Hence, the coefficients of the product of two polynomials can be determined from either convolving vectors of their respective coefficients, or by Fourier transforming those vectors, multiplying them together, then taking the inverse Fourier transform of the result.

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.

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

Before that, however, we can perform regular multiplication to determine the correct answer:

$$8755790 \times 1367267 = 11971502725930$$

Having done that, we construct the numbers 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. This will become clearer after the convolution has been performed.

Using the same `signal.convolve` method from Question 2, the following result is determined:

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

Now, however, unlike with polynomial multiplication, we cannot expect to arrive at the correct product by simply stringing together all the 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 base-10 vector, which can now be concatenated into the product 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, in the same manner as Question 2:

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

As expected, the vector produced by this method is identical to that produced by direct convolution. Therefore, if we apply the same "carrying" process that we applied above, we will no doubt arrive at the same value for the product of $x \times y$.

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`:

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

This gives us the expected result of the double transform algorithm. Now, to proceed, we combine x and y element-wise into a single complex vector:

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

We can use the same `fft` function without any additional considerations to Fourier transform this complex vector. Doing so, we obtain:

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

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

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

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:

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

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

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

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:

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

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:

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

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

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

Using `scipy.fft`, the Fourier transform of z is

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

Akin to part (a), 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.

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

Finally, we can reconstruct the individual Fourier transforms of x and y:

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

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.]
```

Given we know how many zeros were padded onto the shorter sequence, it can indeed be recovered by truncating the extra length. Therefore, the double transform algorithm can be applied even if the sequences differ in length.

Question 5

a) We are given the following polynomial, and want to determine its poles and zeros.

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

Immediately, the absence of a denominator indicates an all-zero model. The coordinates of the zeros are the complex 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.

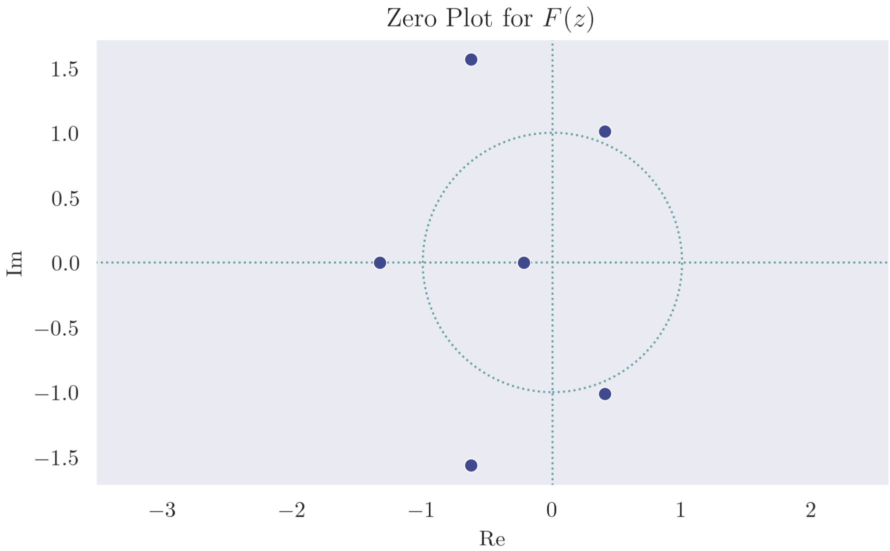

Figure 5.1: Zero plot for $F(z)$; no poles are present.

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

That is,

$$F(z)\Big|_{z=e^{-j2\pi k/N},\ k=0,\ldots,N-1}$$

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

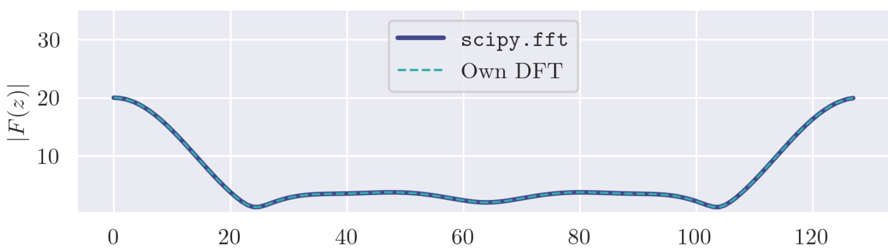

Figure 5.2: $|F(z)|$ evaluated at 128 points around the unit circle.

Question 6

Consider the following “continuous” 7 Hz sine wave and its representation in the Fourier domain.

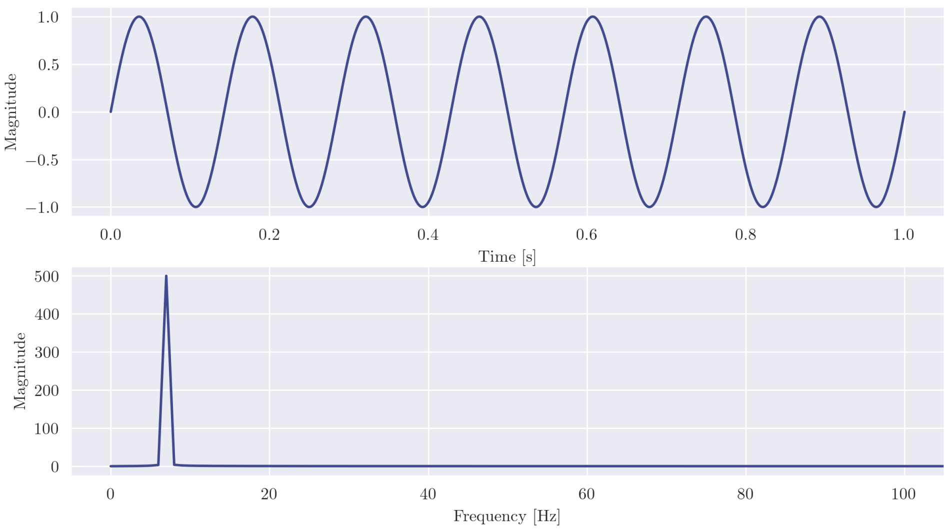

Figure 6.1: Time and frequency views of a 7 Hz sine wave (rendered at 1 kHz)

The discrete nature of computer graphics means the visualisation is not truly continuous, and is actually rendered in timesteps of 1 millisecond. However, for the purposes of this question, we treat it as continuous to enable comparisons with subsequent down- and up-sampled signals. The Fourier transform of the function demonstrates a single peak at 7 Hz, as one would expect.

We now sample the sine wave at 20 Hz, above the Nyquist frequency of 14 Hz. Therefore, in theory, it should be possible to perfectly reconstruct the original signal.

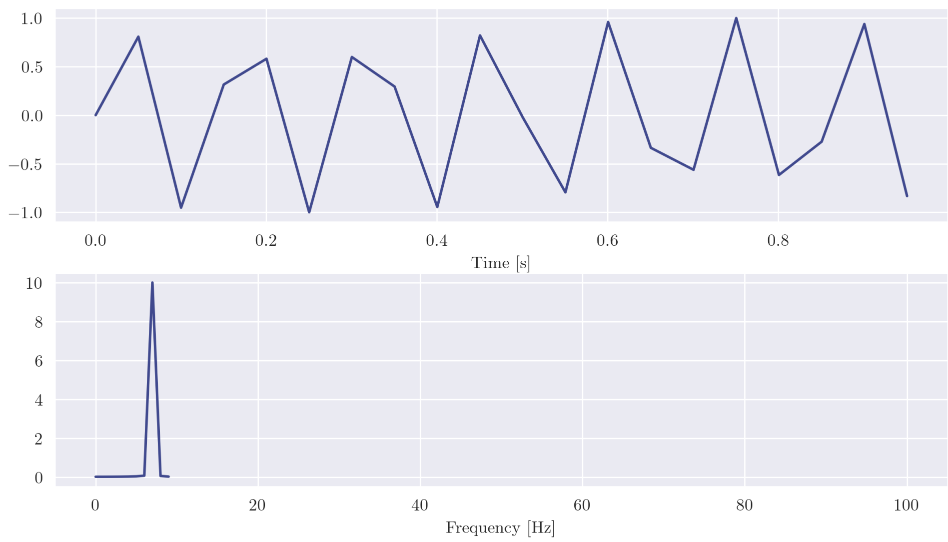

Figure 6.2: Time and frequency views of the 7 Hz sine wave sampled at 20 Hz

In the Fourier domain of Figure 6.2, we observe the same single peak at 7 Hz, indicating that the signal indeed has not aliased. Yet, linear interpolation clearly produces an inaccurate reconstruction of the original sine wave.

A more accurate reconstruction can be achieved using sinc interpolation, whereby the sampled signal is convolved with a sinc function in the time domain. Alternatively, this is equivalent to multiplying the Fourier transform of the sampled signal by a rectangular window.

As a first step, we up-sample the sampled signal from 20 Hz to 80 Hz by padding three zeros between each sampled value. This produces the following result:

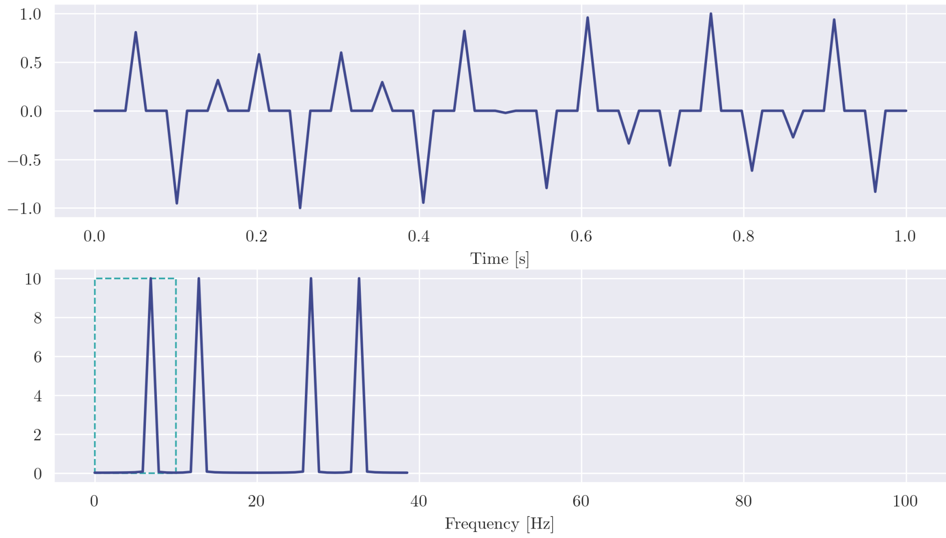

Figure 6.3: Time and frequency views of the intermediate upsampled signal

Evidently, upsamping by zero padding has introduced three new peaks in the Fourier domain, corresponding to the number of zeros padded between each value. To remove these peaks, we multiply the signal by a rectangular window to filter out all peaks except the desired 7 Hz peak (and its negative counterpart). The positive half of this window is indicated in Figure 6.3.

To restate, this is equivalent to convolution with a sinc function in the time domain, and produces the sinc-interpolated signal presented in Figure 6.4.

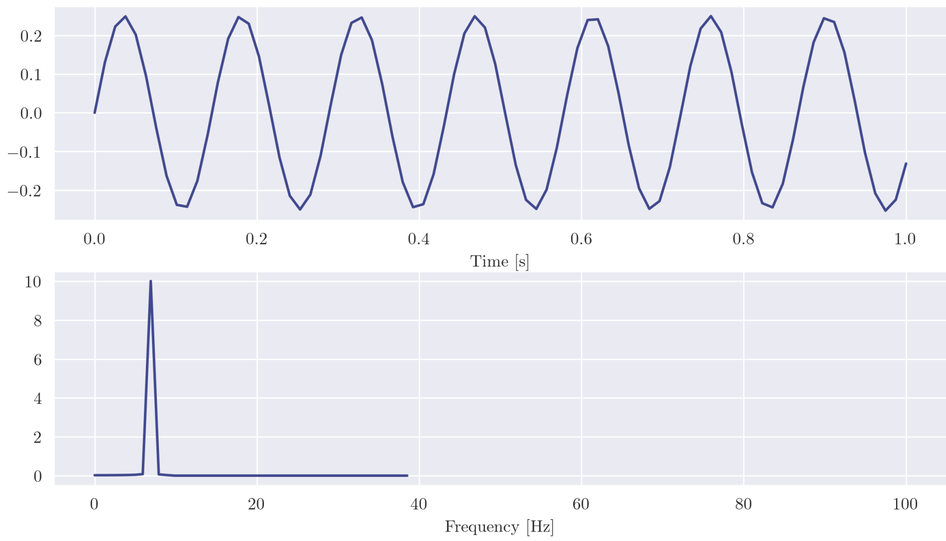

Figure 6.4: Time and frequency views of the 80 Hz sinc interpolated signal

Though still not a perfect reconstruction (which we could improve by padding with more zeros), it is much improved over the linear interpolation of Figure 6.2.

Question 7

As an alternative to sinc interpolation, a signal sampled above the Nyquist frequency can be interpolated by zero-padding its DFT, then inverse transforming back to the time domain. To compare this with sinc interpolation, we consider the same 7 Hz sine wave sampled at 20 Hz.

Consider the Fourier transforms of the sampled and sinc interpolated signals from Question 6.

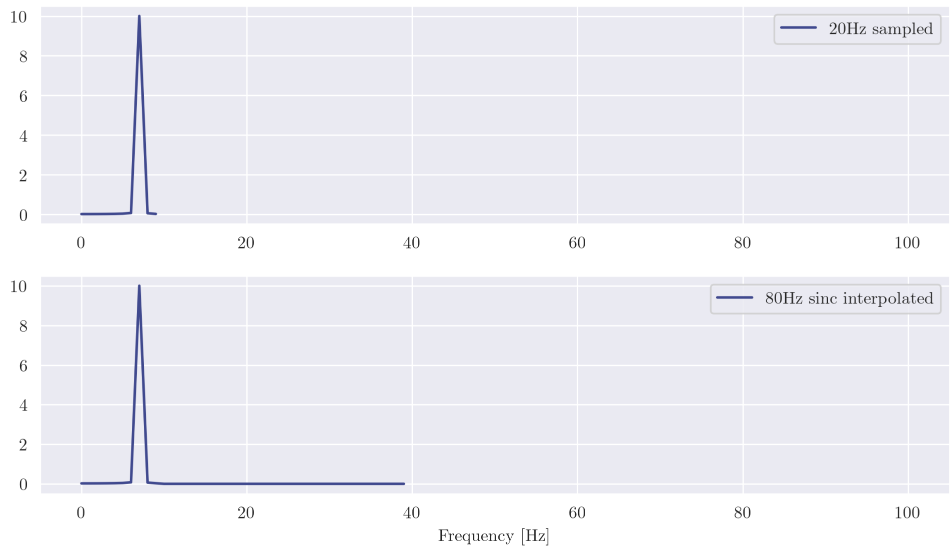

Figure 7.1: Frequency comparison: 20 Hz sampled signal and 80 Hz sinc interpolated signal

The Fourier transform of the 20 Hz sampled signal has a positive bandwidth of 10 Hz (and an equal negative bandwidth, the sum corresponding with the sampling frequency). The 80 Hz interpolated signal has a positive bandwidth of 40 Hz. The only difference between them is the length of the (approximately) zero magnitude tail. Therefore, it should be possible to recreate the interpolation by zero-padding the Fourier transform of the sampled signal to the desired length, in the both positive and negative frequencies. This produces Figure 7.2.

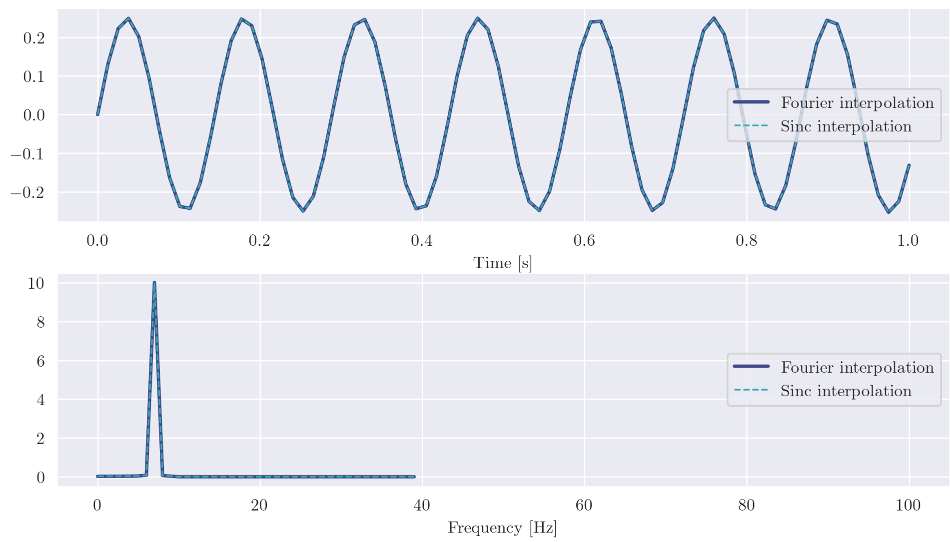

Figure 7.2: Time and frequency views of the 80 Hz Fourier interpolated signal

As expected, the result is visually indistinguishable from that of sinc interpolation.

Question 8

Question 9

Figure 9.1 visualises the ten homing pigeon departure headings and their average value.

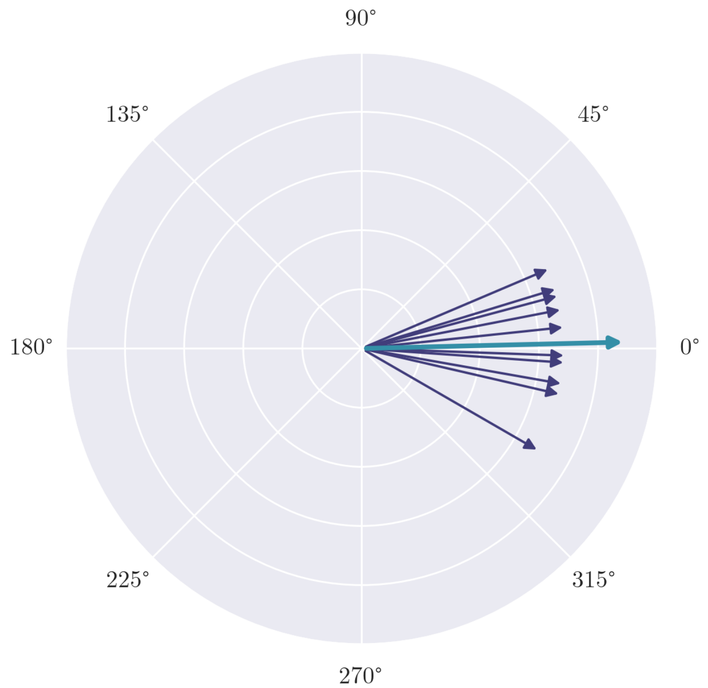

Figure 9.1: Homing pigeon departure headings and average heading

To calculate the average heading, one might consider naively summing and dividing the values.

$$\frac{11 + 15 + 350 + 330 + 23 + 347 + 17 + 356 + 6 + 358}{10} = 181.3°$$

In doing so, one quickly realises that the result is exactly in the opposite direction to the expected average heading, because the headings are distributed on either side of the 360° modulus angle.

A better method is to convert the headings into complex numbers using polar ($e^{j\theta}$) form, average them, and determine the argument of the result. For example, for the heading of 11°:

$$e^{j(11\times(\pi/180))} \approx 0.982 + j0.191$$

We can perform the conversion for each heading and average the real and imaginary parts separately to obtain the following average heading as a complex number: $0.964 + j0.023$.

Finally, we determine the argument of the complex number and convert from radians to degrees:

$$\arctan\left(\frac{0.023}{0.964}\right) \approx 0.024\text{rad} \approx 1.390°$$

Hence, we have determined an average heading matching our observation of Figure 9.1.

Appendix

A sidelobes.py

```python
"""
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 / 2)

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 - 2 * np.abs(_t) / T) * (np.abs(_t) <= T / 2)

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

def visualise_rectangular() -> None:
    """
    Produces a series of plots visualising the rectangular pulse and its first
    derivative.
    """
    t = np.linspace(-1, 1, 1001)

    fig, axs = plt.subplots(1, 2, figsize=(6, 2))

    # Rectangular pulse
    sns.lineplot(x=t, y=(np.abs(t)<=0.5), ax=axs[0])

    # 1st derivative
    sns.lineplot(x=t, y=(-np.sign(t)*(np.abs(t)==0.5)), ax=axs[1])

    axs[0].set_title("Rectangular pulse")
    axs[1].set_title("1st derivative")
    for i in range(2):
        axs[i].set_xticks(np.linspace(-1, 1, 5))
        axs[i].set_xticklabels(["", "$-T/2$", "", "$T/2$", ""])
        axs[i].set_yticks(np.linspace(-1, 1, 5))
        axs[i].set_yticklabels([])
    axs[0].set_yticklabels(["$-A$", "", 0, "", "$A$"])
```

```python

    fname = Path(A1_ROOT, "output", "q1_rectangular.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)

def visualise_triangular() -> None:
    """
    Produces a series of plots visualising the triangular pulse and its first
    and second derivatives.
    """
    t = np.linspace(-1, 1, 1001)

    fig, axs = plt.subplots(1, 3, figsize=(9, 2))

    # Triangular pulse
    sns.lineplot(x=t, y=((1-2*np.abs(t))*(np.abs(t)<=0.5)), ax=axs[0])

    # 1st derivative
    sns.lineplot(x=t, y=(-np.sign(t)*(np.abs(t)<=0.5)), ax=axs[1])

    # 2nd derivative
    ddy = np.zeros(t.shape); ddy[np.abs(t)==0.5] = 1; ddy[t==0] = -2
    sns.lineplot(x=t, y=ddy, ax=axs[2])

    axs[0].set_title("Triangular pulse")
    axs[1].set_title("1st derivative")
    axs[2].set_title("2nd derivative")
    for i in range(3):
        axs[i].set_xticks(np.linspace(-1, 1, 5))
        axs[i].set_xticklabels(["", "$-T/2$", "", "$T/2$", ""])
        axs[i].set_yticks(np.linspace(-2, 2, 9))
        axs[i].set_yticklabels([])
    axs[0].set_yticklabels(["$-2A$", "", "$-A$", "", 0, "", "$A$", "", "$2A$"])

    fname = Path(A1_ROOT, "output", "q1_triangular.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)

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

def main():

    t_min = -50     # Configuration of input time sequence
    t_max =  50
    N = 1001

    t = np.linspace(t_min, t_max, N)
    f = fftfreq(N, (t_max - t_min) / (N - 1))

    # (Optionally) visualise the rectangular/triangular pulses and derivatives
    visualise_rectangular()
    visualise_triangular()

    H_rect = np.abs(fft(rectangular_pulse(t)))
    H_tria = np.abs(fft(triangular_pulse(t)))

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

    sns.lineplot(x=f, y=H_rect, ax=ax, label="Rectangular")
    sns.lineplot(x=f, y=H_tria, ax=ax, label="Triangular")

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

```

```
    fname = Path(A1_ROOT, "output", "q1_sidelobes.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)


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

B multiplying.py

```python
"""
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():
```

```
    ### QUESTION 2 ###

    vx = np.array([1, 2, 6, 11, 15, 12])
    vy = np.array([1, -3, -3, 7, -7, 3])

    print("Q2 by convolution:\n", multiply_conv(vx, vy))
    print("Q2 by FFT and IFFT:\n", multiply_fft(vx, vy))

    ### QUESTION 3 ###

    vx = np.array([8, 7, 5, 5, 7, 9, 0])
    vy = np.array([1, 3, 6, 7, 2, 6, 7])

    print("Q3 by multiplication:", 8755790 * 1367267)
    print("Q3 by convolution:", multiply_conv(vx, vy, carry=True))
    print("Q3 by FFT and IFFT:", multiply_fft(vx, vy, carry=True))


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

C double_transform.py

```python
"""
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

```python
"""
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.
"""

from pathlib import Path

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

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, SNS_STYLE

### 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]
```

```python
    curr_ratio = yrng / xrng

    if curr_ratio > ratio: # i.e. the current ratio is too tall and narrow
        add_xlim = (yrng / ratio - xrng) * 0.5
        if padto == "right":
            new_xlim = (xlim[0], xlim[1] + 2 * add_xlim)
        elif padto == "left":
            new_xlim = (xlim[0] - 2 * add_xlim, xlim[1])
        else:
            new_xlim = (xlim[0] - add_xlim, xlim[1] + add_xlim)
        ax.set_xlim(new_xlim)

    if curr_ratio < ratio: # i.e. the current ratio is too short and wide
        add_ylim = (xrng * ratio - yrng) * 0.5
        if padto == "upper":
            new_ylim = (ylim[0], ylim[1] + 2 * add_ylim)
        elif padto == "lower":
            new_ylim = (ylim[0] - 2 * add_ylim, ylim[1])
        else:
            new_ylim = (ylim[0] - add_ylim, ylim[1] + add_ylim)
        ax.set_ylim(new_ylim)

    return ax

def draw_unit_circle(ax: Axes) -> Axes:
    """
    Draws dotted axes and unit circle on the given axes, similar in style to
    MATLAB's zplane function.
    """
    style_config = {"ls": "dotted", "lw": 0.9, "color": "cadetblue", "zorder": 0}

    u_circ = Circle(xy=(0, 0), radius=1, fill=False, **style_config)
    ax.add_patch(u_circ)

    x_axis = Line2D(xdata=ax.get_xlim(), ydata=(0, 0), **style_config)
    y_axis = Line2D(xdata=(0, 0), ydata=ax.get_ylim(), **style_config)
    ax.add_line(x_axis)
    ax.add_line(y_axis)
    ax.set_aspect("equal")

    return ax

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

def run_part_a(F: Polynomial) -> None:
    """
    Plots the roots of the polynomial with given coefficients on the complex
    plane, with a unit circle underlay.
    """
    print(f"{F.roots() = }")

    # Override default style to hide grid
    sns.set_style("dark")

    # Re-set the plot text customisation, which gets overriden by set_style
    plt.rcParams.update(PLT_CONFIG)

    fig, ax = plt.subplots()

    ax = plot_poles_or_zeros(F, "zeros", ax)
    ax = axes_ratio_scale(ax, ratio=9/16, padto="center")
    ax = draw_unit_circle(ax)
    ax.set_title("Zero Plot for $F(z)$")
```

```python

    fname = Path(A1_ROOT, "output", "q5a_polezero.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)

def run_part_b(poly: Polynomial) -> None:
    """
    Plots the magnitude of the polynomial with the given coefficients at 128
    uniformly spaced points around the unit circle using the DFT.
    """
    y_fft = np.abs(fft(poly.coef, n=128))
    y_dft = np.abs(zdft(poly.coef, N=128))

    # Re-set the default Seaborn style, which was changed by part (a)
    sns.set_style(SNS_STYLE)

    # Re-set the plot text customisation, which gets overridden by set_style
    plt.rcParams.update(PLT_CONFIG)

    fig, ax = plt.subplots()

    sns.lineplot(x=np.arange(128), y=y_fft, ax=ax, ls="-", lw=2,
        label=r"$\texttt{scipy.fft}$")
    sns.lineplot(x=np.arange(128), y=y_dft, ax=ax, ls="--", lw=1,
        label=r"Own DFT")

    ax = axes_ratio_scale(ax, ratio=1/4, padto="upper")

    ax.set_title("")
    ax.set_xlabel("")
    ax.set_ylabel("$|F(z)|$")

    ax.legend(loc="upper center")

    fname = Path(A1_ROOT, "output", "q5b_dftsample.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)

def main():

    poly = Polynomial([1, 5, 3, 4, 4, 2, 1])
    run_part_a(poly)
    run_part_b(poly)


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

E interpolation.py

```python
"""
Script associated with Q6 and 7.

Performs two methods of upsamping a sine wave: sinc interpolation and
zero-padding in the Fourier domain.
"""

from pathlib import Path

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

from matplotlib.axes import Axes
from matplotlib.figure import Figure
from matplotlib.patches import Rectangle
from scipy.fft import fft, fftfreq, ifft

from config import A1_ROOT, SAVEFIG_CONFIG

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

def sinc_interpolate(x: np.array, n: int, viz: bool = False) -> np.array:
    """
    Upsamples the given signal by the specified factor using sinc interpolation.
    """
    # Increases the sampling rate of x by inserting n-1 zeros between samples
    x_upsamp = np.concatenate([[p]+[0]*(n-1) for p in x])

    # (Optionally) Visualise the intermediate upsampled result
    if viz:
        fig, axs = time_fourier_plot(np.linspace(0, 1, len(x_upsamp)), x_upsamp)
        axs[1].set_xlim(-5, 105)
        axs[1].add_patch(Rectangle(
            (0, 0), 10, 10, color=sns.color_palette()[1], fill=False, ls="--"))

        fname = Path(A1_ROOT, "output", "q6_intermediate.png")
        fig.savefig(fname, **SAVEFIG_CONFIG)

    # Convolve with sinc in time domain by applying rect window in freq. domain
    H_upsamp = fft(x_upsamp)
    H_upsamp[10:-10] = 0
    x_upsamp = ifft(H_upsamp)

    return x_upsamp

def fourier_interpolate(x: np.array, n: int) -> np.array:
    """
    Upsamples the given signal by the specified factor by applying the Fourier
    transform, zero-padding, then inverse Fourier transforming.
    """
    H = fft(x); N = len(H)

    # Pad N*(n-1) zeros between positive and negative frequencies
    H_upsamp = np.concatenate([H[:N//2], np.zeros(len(x)*(n-1)), H[N//2:]])
    x_upsamp = ifft(H_upsamp)

    return x_upsamp

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

```

```python
def time_fourier_plot(t: np.array, x: np.array) -> tuple[Figure, list[Axes]]:
    """
    Plot the given signal and its discrete Fourier transform.
    """
    f = fftfreq(n=len(t), d=(t[1]-t[0]))[:len(t)//2]
    H = np.abs(fft(x))[:len(t)//2]

    fig, axs = plt.subplots(2, figsize=(8, 4.5))
    fig.tight_layout()

    sns.lineplot(x=t, y=x, ax=axs[0])
    sns.lineplot(x=f, y=H, ax=axs[1])

    axs[0].set_xlabel("Time [s]")
    axs[1].set_xlabel("Frequency [Hz]")

    return fig, axs

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

def run_question_6(x_samp: np.array):
    """
    Performs sinc interpolation to upsample the given signal to 80 Hz and plots
    the results in the time and Fourier domains.
    """
    # Upsample from 20 Hz to 80 Hz
    x_upsamp = sinc_interpolate(x_samp, 4, viz=True)
    t_upsamp = np.linspace(0, 1, 80)

    fig, axs = time_fourier_plot(t_upsamp, x_upsamp)
    axs[1].set_xlim(-5, 105)

    fname = Path(A1_ROOT, "output", "q6_upsampled.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)

def run_question_7(x_samp: np.array):
    """
    Performs zero-padding in the Fourier domain to upsample the given signal to
    80 Hz and plots the results in the the time and Fourier domains.
    """
    # Upsample from 20 Hz to 80 Hz
    x_upsamp = fourier_interpolate(x_samp, 4)
    t_upsamp = np.linspace(0, 1, 80)

    # Plot a before/after frequency domain comparison
    fig, axs = plt.subplots(2, figsize=(8, 4.5))
    fig.tight_layout()

    f_samp = fftfreq(n=20, d=1/20)[:len(x_samp)//2]
    H_samp = np.abs(fft(x_samp))[:len(x_samp)//2]

    N_upsamp = len(x_upsamp)
    f_upsamp = fftfreq(n=80, d=1/80)[:N_upsamp//2]
    H_upsamp = np.abs(fft(x_upsamp))[:N_upsamp//2]

    sns.lineplot(x=f_samp,   y=H_samp,   ax=axs[0], label="20Hz sampled")
    sns.lineplot(x=f_upsamp, y=H_upsamp, ax=axs[1], label="80Hz sinc interpolated")

    axs[0].set_xlim([-5, 105])
    axs[1].set_xlim([-5, 105])

    axs[1].set_xlabel("Frequency [Hz]")

```

```python
    fname = Path(A1_ROOT, "output", "q7_freqcompare.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)

    # Plot a comparison this method and sinc interpolation
    x_upsinc = sinc_interpolate(x_samp, 4)
    H_upsinc = np.abs(fft(x_upsinc))[:N_upsamp//2]

    fig, axs = plt.subplots(2, figsize=(8, 4.5))
    fig.tight_layout()

    sns.lineplot(x=t_upsamp, y=x_upsamp, ax=axs[0], ls="-",  lw=2,
        label="Fourier interpolation")
    sns.lineplot(x=t_upsamp, y=x_upsinc, ax=axs[0], ls="--", lw=1,
        label="Sinc interpolation")

    sns.lineplot(x=f_upsamp, y=H_upsamp, ax=axs[1], ls="-",  lw=2,
        label="Fourier interpolation")
    sns.lineplot(x=f_upsamp, y=H_upsinc, ax=axs[1], ls="--", lw=1,
        label="Sinc interpolation")

    axs[0].set_xlabel("Time [s]")
    axs[1].set_xlabel("Frequency [Hz]")

    axs[0].legend(loc="center right")
    axs[1].legend(loc="center right")

    axs[1].set_xlim(-5, 105)

    fname = Path(A1_ROOT, "output", "q7_upsampled.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)

def main():

    # "Continuous time" 7 Hz sine wave (actually sampled at 1 kHz)
    t = np.linspace(0, 1, 1000)
    x = np.sin(2 * np.pi * 7 * t)

    # Plot in the time and Fourier domains
    fig, axs = time_fourier_plot(t, x)
    axs[1].set_xlim(-5, 105)

    fname = Path(A1_ROOT, "output", "q6_sine7hz.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)

    # Sine wave sampled at 20 Hz
    t_samp = t[::1000//20]
    x_samp = x[::1000//20]

    # Plot in the time and Fourier domains
    fig, axs = time_fourier_plot(t_samp, x_samp)
    axs[1].set_xlim(-5, 105)

    fname = Path(A1_ROOT, "output", "q6_sampled.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)

    ### QUESTION 6 ###
    run_question_6(x_samp)

    ### QUESTION 7 ###
    run_question_7(x_samp)


if __name__ == "__main__":
```

```
    main()
```

F headings.py

```python
"""
Script associated with Q9.

Plots a number of headings on a polar plot and calculates the average heading.
"""

from pathlib import Path

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

from matplotlib.axes import Axes

from config import A1_ROOT, SAVEFIG_CONFIG, SNS_PALETTE

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

def average_heading(headings: list[float]) -> float:
    """
    Returns the average heading in degrees of the given list of headings.
    """
    return np.rad2deg(np.angle(np.average(np.exp(1j * np.deg2rad(headings)))))

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

def draw_arrow(x: float, y: float, dx: float, dy: float, ax: Axes,
        arrowprops: dict = None):
    """
    Draws an arrow on the given axes from (x, y) to (x+dx, y+dy).
    """
    arrowprops = arrowprops or {
        "arrowstyle": "-|>", "color": sns.color_palette(SNS_PALETTE)[1]}

    ax.annotate("", xy=(x + dx, y + dy), xytext=(x, y), arrowprops=arrowprops)

def visualise_headings(headings: list[float]) -> None:
    """
    Visualises the given headings in degrees on a polar plot.
    """
    fig = plt.figure(figsize=(8, 4.5))
    ax = fig.add_subplot(projection='polar')

    for phi in np.deg2rad(headings):
        draw_arrow(0, 0, phi, 0.7, ax)

    avg = average_heading(headings)
    print("Average_heading:", np.round(avg, decimals=3), "[deg]")

    draw_arrow(0, 0, np.deg2rad(avg), 0.9, ax, arrowprops={"arrowstyle": "-|>",
        "color": sns.color_palette(SNS_PALETTE)[3], "lw": 2})

    # Hide magnitude labels
    ax.set_yticklabels([])

    fname = Path(A1_ROOT, "output", "q9_headings.png")
    fig.savefig(fname, **SAVEFIG_CONFIG)

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

```

```
def main():

    headings = [11, 15, 350, 330, 23, 347, 17, 356, 6, 358] # in degrees
    visualise_headings(headings)


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