

ELEC4620 Assignment 1

Deren Teo

August 10, 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.

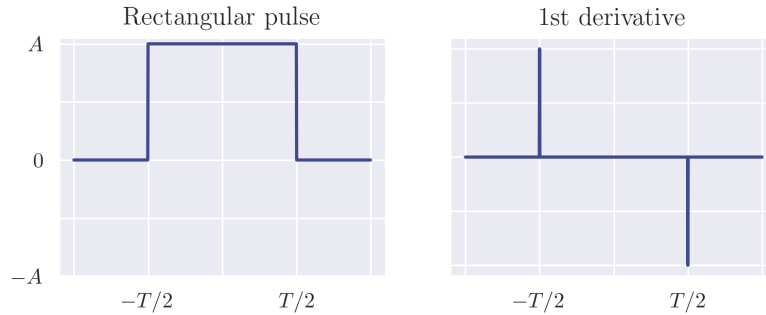


Figure 1.1: A rectangular pulse and its first derivative

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 :

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

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:

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

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.

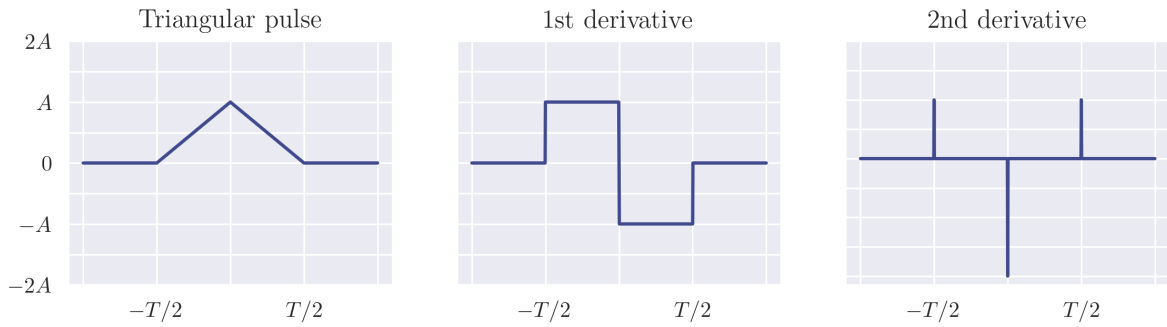


Figure 1.2: A triangular pulse and its first and second derivatives

The Fourier transform of the second derivative is therefore

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

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.

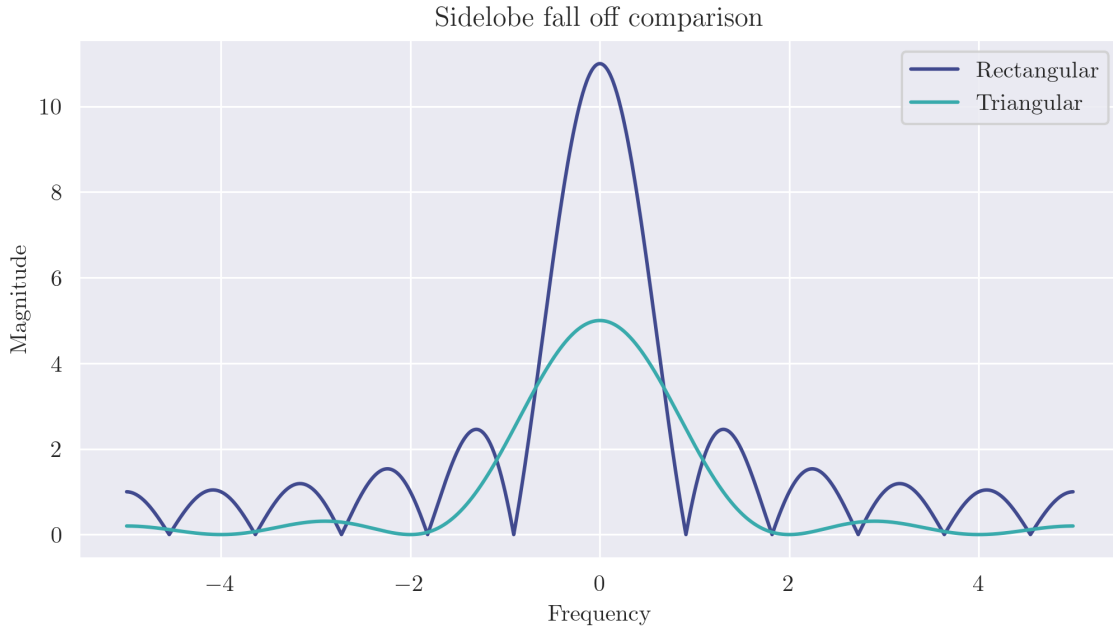


Figure 1.3: Fourier transform sidelobe fall off comparison: rectangular and triangular pulses.

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:

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

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:

$$[1. \quad -1. \quad -3. \quad -6. \quad -29. \quad -35. \quad -40. \quad 10. \quad 12. \quad -39. \quad 36.]$$

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`:

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

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:

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

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:

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

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 :

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

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,

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

The individual Fourier transforms of x and y are:

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

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

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

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 .

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

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:

$$\text{ifft}(Y) = [1. \ 5. \ 3. \ 1. \ 3. \ 5. \ 3. \ -0.]$$

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.

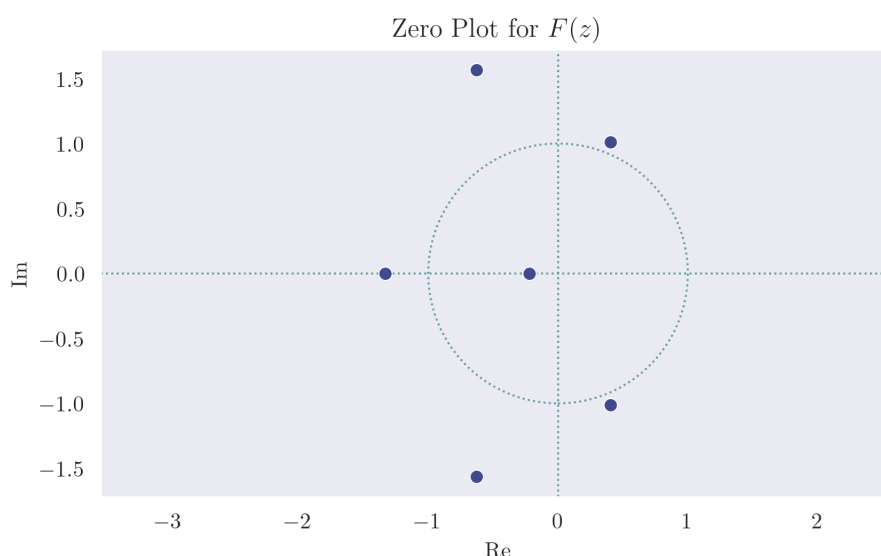


Figure 5.1: Zero plot for $F(z)$; no poles are present.

- 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,

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

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

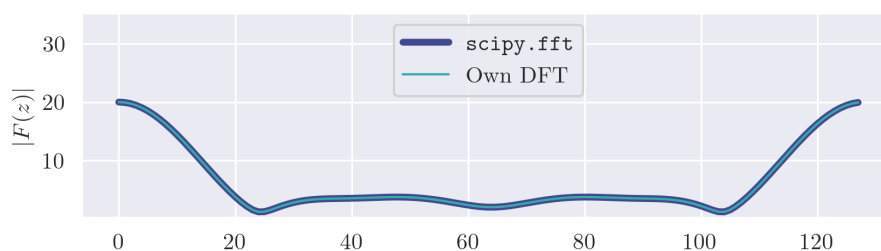


Figure 5.2: $|F(z)|$ evaluated at 128 points around the unit circle.

Question 6

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

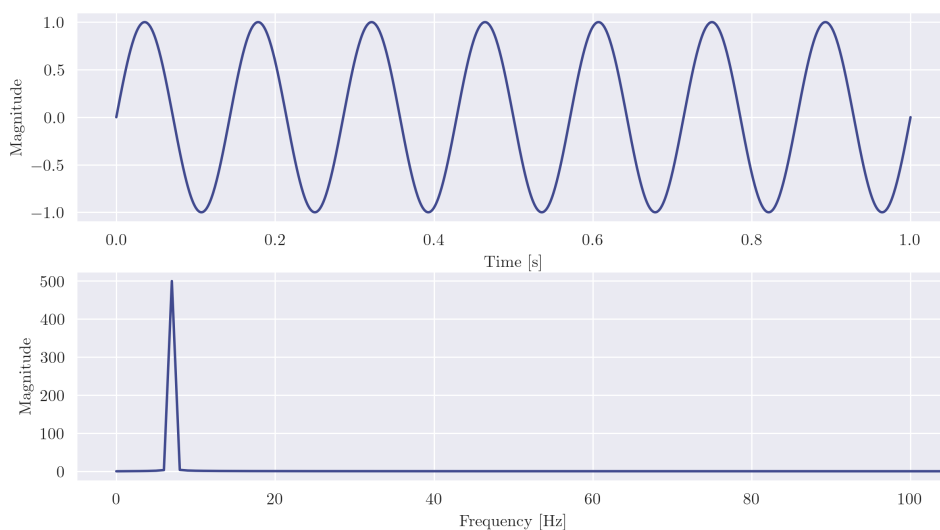


Figure 6.1: Time and frequency views of a 7 Hz sine wave (rendered at 1 kHz)

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.

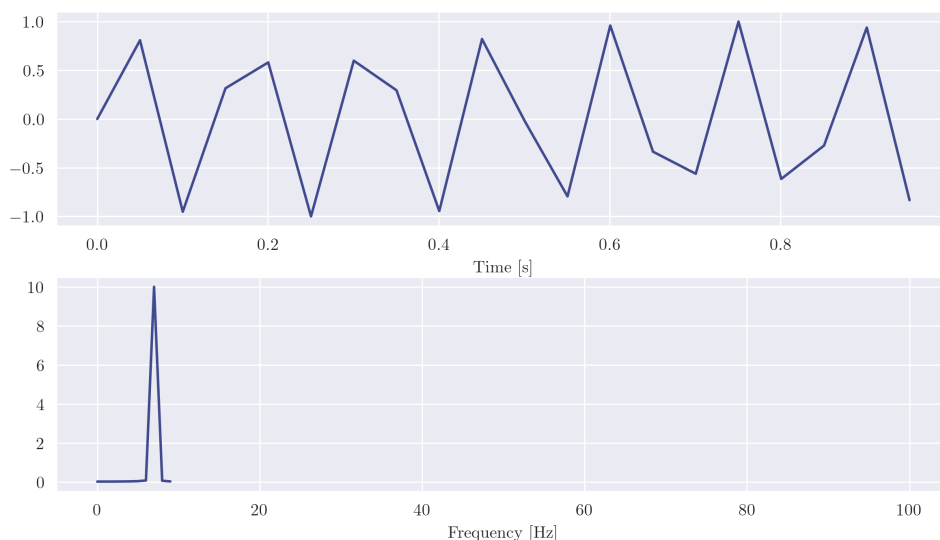


Figure 6.2: Time and frequency views of the 7 Hz sine wave sampled at 20 Hz

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:

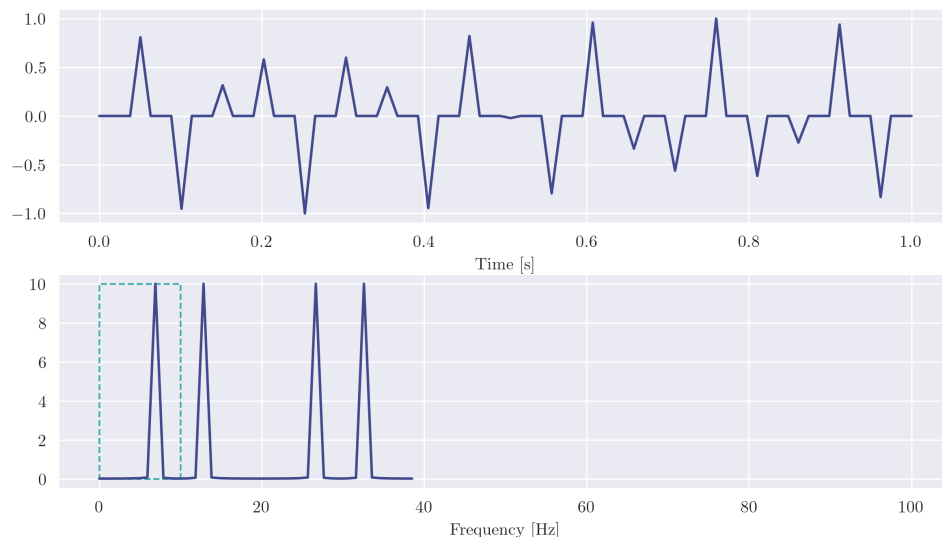


Figure 6.3: Time and frequency views of intermediate upsampled signal

Evidently, upsampling 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.

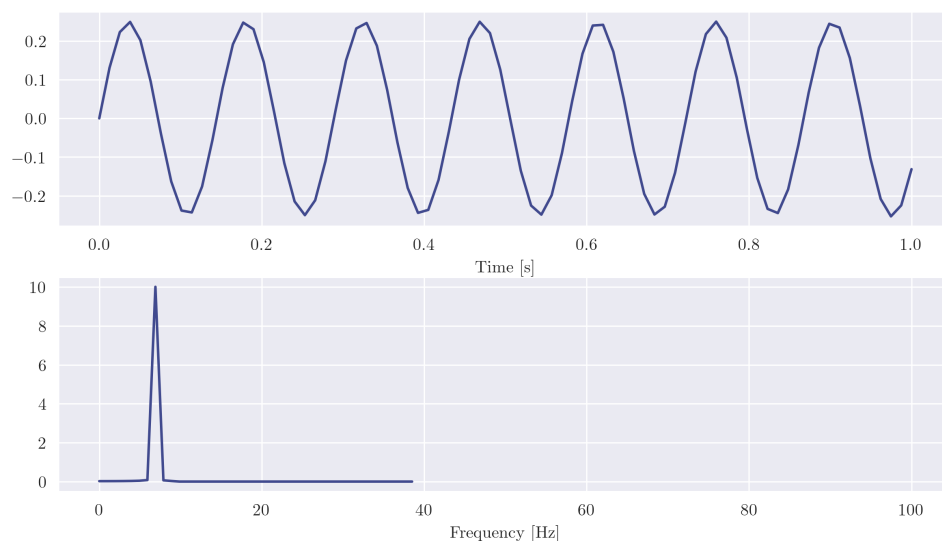


Figure 6.4: Time and frequency views of the 80 Hz upsampled signal

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

Question 8

Question 9

Appendix

A sidelobes.py

```
1  """
2  Script associated with Q1.
3
4  Plots Fourier transform of rectangular and triangular pulse functions, enabling
5  comparison of rates at which sidelobes fall off.
6  """
7
8  from pathlib import Path
9
10 import matplotlib.pyplot as plt
11 import numpy as np
12 import seaborn as sns
13
14 from scipy.fft import fft, fftfreq
15
16 from config import A1_ROOT, SAVEFIG_CONFIG
17
18 ### DSP FUNCTIONS #####
19
20 def rectangular_pulse(_t: np.array, A: float = 1, T: float = 1) -> np.array:
21     """
22     Constructs a rectangular pulse of amplitude 'A' and width 'T', centred about
23     t=0 on '_t'. Returns array of y-values corresponding to '_t'.
24     """
25     return A * (np.abs(_t) <= T / 2)
26
27 def triangular_pulse(_t: np.array, A: float = 1, T: float = 1) -> np.array:
28     """
29     Constructs a triangular pulse of maximum amplitude 'A' and width 'T',
30     centred about t=0 on '_t'. Returns array of y-values corresponding to '_t'.
31     """
32     return A * (1 - 2 * np.abs(_t) / T) * (np.abs(_t) <= T / 2)
33
34 ### VISUALISATION #####
35
36 def visualise_rectangular() -> None:
37     """
38     Produces a series of plots visualising the rectangular pulse and its first
39     derivative.
40     """
41     t = np.linspace(-1, 1, 1001)
42
43     fig, axs = plt.subplots(1, 2, figsize=(6, 2))
44
45     # Rectangular pulse
46     sns.lineplot(x=t, y=(np.abs(t) <= 0.5), ax=axs[0])
47
48     # 1st derivative
49     sns.lineplot(x=t, y=(-np.sign(t) * (np.abs(t) == 0.5)), ax=axs[1])
50
51     axs[0].set_title("Rectangular_pulse")
52     axs[1].set_title("1st_derivative")
53     for i in range(2):
54         axs[i].set_xticks(np.linspace(-1, 1, 5))
55         axs[i].set_xticklabels(["", "$-T/2$", "", "$T/2$", ""])
56         axs[i].set_yticks(np.linspace(-1, 1, 5))
57         axs[i].set_yticklabels([])
58     axs[0].set_yticklabels(["$-A$", "", 0, "", "$A$"])
```

```

59
60     fname = Path(A1ROOT, "output", "q1_rectangular.png")
61     fig.savefig(fname, **SAVEFIG_CONFIG)
62
63     def visualise_triangular() -> None:
64         """
65         Produces a series of plots visualising the triangular pulse and its first
66         and second derivatives.
67         """
68         t = np.linspace(-1, 1, 1001)
69
70         fig, axs = plt.subplots(1, 3, figsize=(9, 2))
71
72         # Triangular pulse
73         sns.lineplot(x=t, y=((1-2*np.abs(t))*(np.abs(t)<=0.5)), ax=axs[0])
74
75         # 1st derivative
76         sns.lineplot(x=t, y=(-np.sign(t)*(np.abs(t)<=0.5)), ax=axs[1])
77
78         # 2nd derivative
79         ddy = np.zeros(t.shape); ddy[np.abs(t)==0.5] = 1; ddy[t==0] = -2
80         sns.lineplot(x=t, y=ddy, ax=axs[2])
81
82         axs[0].set_title("Triangular_pulse")
83         axs[1].set_title("1st_derivative")
84         axs[2].set_title("2nd_derivative")
85         for i in range(3):
86             axs[i].set_xticks(np.linspace(-1, 1, 5))
87             axs[i].set_xticklabels(["", " $-T/2$ ", "", " $T/2$ ", ""])
88             axs[i].set_yticks(np.linspace(-2, 2, 9))
89             axs[i].set_yticklabels([])
90         axs[0].set_yticklabels([" $-2A$ ", "", " $-A$ ", "", 0, "", " $A$ ", "", " $2A$ "])
91
92         fname = Path(A1ROOT, "output", "q1_triangular.png")
93         fig.savefig(fname, **SAVEFIG_CONFIG)
94
95     ### ENTRYPOINT #####
96
97     def main():
98
99         t_min = -50      # Configuration of input time sequence
100        t_max = 50
101        N = 1001
102
103        t = np.linspace(t_min, t_max, N)
104        f = fftfreq(N, (t_max - t_min) / (N - 1))
105
106        # (Optionally) visualise the rectangular/triangular pulses and derivatives
107        visualise_rectangular()
108        visualise_triangular()
109
110        H_rect = np.abs(fft(rectangular_pulse(t)))
111        H_tri = np.abs(fft(triangular_pulse(t)))
112
113        fig, ax = plt.subplots(figsize=(8, 4))
114
115        sns.lineplot(x=f, y=H_rect, ax=ax, label="Rectangular")
116        sns.lineplot(x=f, y=H_tri, ax=ax, label="Triangular")
117
118        ax.set_title("Sidelobe_fall_off_comparison")
119        ax.set_xlabel("Frequency")
120        ax.set_ylabel("Magnitude")
121

```

```
122     fname = Path(A1_ROOT, "output", "q1_sidelobes.png")
123     fig.savefig(fname, **SAVEFIG_CONFIG)
124
125
126 if __name__ == "__main__":
127     main()
```

B multiplying.py

```
1  """
2  Script associated with Q2 and 3.
3
4  Multiplies polynomials/numbers in vector representation using convolution and
5  Fourier transform methods.
6  """
7
8  import numpy as np
9
10 from scipy.signal import convolve
11 from scipy.fft import fft, ifft
12
13 ### UTILITY FUNCTIONS #####
14
15 def multiply_carry(z: np.array) -> np.array:
16     """
17     Perform the "carry" steps of the multiplication process. Starting from the
18     right end of 'z', each digit is taken modulo 10 and the remainder is added
19     to the value immediately to the left. Returns an array of single digits,
20     possibly except the first value (though the answer will still be correct).
21     """
22     r = 0; ret = []
23     for n in z[::-1]:
24         n += r
25         ret.append(n % 10)
26         r = n // 10
27     ret.append(r)
28     return np.array(ret[::-1])
29
30 ### DSP FUNCTIONS #####
31
32 def multiply_conv(x: np.array, y: np.array, carry: bool = False) -> np.array:
33     """
34     Convolve arrays 'x' and 'y' using direct convolution. If 'carry' is True,
35     the output array is modified to be equivalent to the vectorised integer
36     product of vectorised integers 'x' and 'y'. Otherwise, the convolution
37     result is returned "as-is".
38     """
39     z = convolve(x, y, mode="full", method="direct")
40     if carry:
41         z = multiply_carry(z)
42     return z
43
44 def multiply_fft(x: np.array, y: np.array, carry: bool = False) -> np.array:
45     """
46     Convolve arrays 'x' and 'y' by performing the FFT on 'x' and 'y',
47     multiplying the results, then performing the inverse FFT. If 'carry' is
48     True, the output array is modified to be equivalent to the vectorised
49     integer product of vectorised integers 'x' and 'y'. Otherwise, the
50     convolution result is returned "as-is".
51     """
52     xpad = np.pad(x, (0, len(y) - 1))
53     ypad = np.pad(y, (0, len(x) - 1))
54     z = ifft(fft(xpad) * fft(ypad)).real
55     if carry:
56         z = multiply_carry(z)
57     return z
58
59 ### ENTRYPOINT #####
60
61 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

```
1  """
2  Script associated with Q4.
3
4  Implementation of the double transform algorithm to Fourier transform two real
5  N-point sequences using one complex N-point transform.
6  """
7
8  import numpy as np
9
10 from scipy.fft import fft, ifft
11
12 ### DSP FUNCTIONS #####
13
14 def ev(H: np.array) -> np.array:
15     """
16     Returns the even component of the given sequence 'H'.
17     """
18     H_minus = np.concatenate([H[:1], H[-1:0:-1]])
19     return 0.5 * (H + H_minus)
20
21 def od(H: np.array) -> np.array:
22     """
23     Returns the odd component of the given sequence 'H'.
24     """
25     H_minus = np.concatenate([H[:1], H[-1:0:-1]])
26     return 0.5 * (H - H_minus)
27
28 ### ENTRYPOINT #####
29
30 def main():
31
32     x = np.array([1, 2, 4, 4, 5, 3, 7, 8])
33     # y = np.array([1, 5, 3, 1, 3, 5, 3, 7]) # PART A: comment out for the other
34     y = np.array([1, 5, 3, 1, 3, 5, 3, 0]) # PART B: comment out for the other
35
36     Z = fft(np.array([a+b*1j for a, b in zip(x, y)]))
37     X = ev(np.real(Z)) + 1j * od(np.imag(Z))
38     Y = ev(np.imag(Z)) - 1j * od(np.real(Z))
39
40     print(f"{fft(x)}")
41     print(f"{fft(y)}")
42
43     print(f"{Z}")
44
45     print(f"{ev(np.real(Z))}")
46     print(f"{od(np.imag(Z))}")
47     print(f"{od(np.real(Z))}")
48     print(f"{ev(np.imag(Z))}")
49
50     print(f"{X}")
51     print(f"{Y}")
52
53     print(f"ifft(Y) == {np.round(ifft(Y).real, 3)}")
54
55
56 if __name__ == "__main__":
57     main()
```

D polezero_dft.py

```
1  """
2  Script associated with Q5.
3
4  Determines the roots of a certain polynomials and produces a pole-zero plot.
5  Evaluates the magnitude of the polynomial around the unit circle using the DFT.
6  """
7
8  from pathlib import Path
9
10 import matplotlib.pyplot as plt
11 import numpy as np
12 import seaborn as sns
13
14 from matplotlib.axes._axes import Axes
15 from matplotlib.lines import Line2D
16 from matplotlib.patches import Circle
17 from numpy.polynomial.polynomial import Polynomial, polyval
18 from scipy.fft import fft
19
20 from config import A1ROOT, PLT_CONFIG, SAVEFIG_CONFIG
21
22 ### DSP FUNCTIONS #####
23
24 def zdft(poly_coef: np.array, N: int) -> np.array:
25     """
26     Computes the 1D 'n'-point discrete Fourier transform of some sequence from
27     its Z transform, given by 'poly'.
28     """
29     return np.array([polyval(np.exp(-1j*2*np.pi*k/N), poly_coef) for k in range(N)])
30
31 ### VISUALISATION #####
32
33 def plot_poles_or_zeros(F: Polynomial, type: str, ax: Axes) -> Axes:
34     """
35     Plots the roots of the polynomial in the complex plane on the given axes.
36     """
37     roots = F.roots()
38
39     marker = {"poles": "X", "zeros": "o"}[type]
40     sns.scatterplot(x=np.real(roots), y=np.imag(roots), ax=ax, marker=marker)
41
42     ax.set_xlabel("Re")
43     ax.set_ylabel("Im")
44
45     return ax
46
47 def axes_ratio_scale(ax: Axes, ratio: float, padto: str = None) -> Axes:
48     """
49     Sets axes aspect as equal and autoscales the axes. If the axes limits ratio
50     does not match the given aspect ratio (i.e. the ratio height / width), the
51     x- or y-axis is lengthened to the desired ratio. Returns the modified axes.
52     """
53     if padto and padto not in ("upper", "lower", "left", "right", "center"):
54         raise ValueError("invalid 'padto' specified")
55     padto = padto or "center"
56
57     ax.set_aspect("equal")
58     ax.autoscale()
59
60     xlim, ylim = ax.get_xlim(), ax.get_ylim()
61     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    ### ENTRYPOINT #####
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()}\n")
112
113        # Override default style to hide grid
114        sns.set_style("dark")
115
116        # Re-set the plot text customisation, which gets overridden 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     def main():
154
155         poly = Polynomial([1, 5, 3, 4, 4, 2, 1])
156         # run_part_a(poly)
157         run_part_b(poly)
158
159
160     if __name__ == "__main__":
161         main()

```

E interpolation.py

```
1  """
2  Script associated with Q6 and 7.
3
4  Performs two methods of upsampling a sine wave: sinc interpolation and
5  zero-padding in the Fourier domain.
6  """
7
8  from pathlib import Path
9
10 import matplotlib.pyplot as plt
11 import numpy as np
12 import seaborn as sns
13
14 from matplotlib.axes import Axes
15 from matplotlib.figure import Figure
16 from matplotlib.patches import Rectangle
17 from scipy.fft import fft, fftfreq, ifft
18
19 from config import A1.ROOT, SAVEFIG.CONFIG
20
21 ### DSP FUNCTIONS #####
22
23 def sinc_interpolate(x: np.array, n: int, viz: bool = False) -> np.array:
24     """
25     Upsamples the given signal by the specified factor using sinc interpolation.
26     """
27     # Increases the sampling rate of x by inserting n-1 zeros between samples
28     x_upsamp = np.concatenate([[p]+[0]*(n-1) for p in x])
29
30     # (Optionally) Visualise the intermediate upsampled result
31     if viz:
32         fig, axs = time_fourier_plot(np.linspace(0, 1, len(x_upsamp)), x_upsamp)
33         axs[1].set_xlim(-5, 105)
34         axs[1].add_patch(Rectangle(
35             (0, 0), 10, 10, color=sns.color_palette()[1], fill=False, ls="—"))
36
37         fname = Path(A1.ROOT, "output", "q6_intermediate.png")
38         fig.savefig(fname, **SAVEFIG.CONFIG)
39
40     # Convolve with sinc in time domain by applying rect window in freq. domain
41     H_upsamp = fft(x_upsamp)
42     H_upsamp[10:-10] = 0
43     x_upsamp = ifft(H_upsamp)
44
45     return x_upsamp
46
47 ### VISUALISATION #####
48
49 def time_fourier_plot(t: np.array, x: np.array) -> tuple[Figure, list[Axes]]:
50     """
51     Plot the given signal and its discrete Fourier transform.
52     """
53     f = fftfreq(n=len(t), d=(t[1]-t[0]))[:len(t)//2]
54     H = np.abs(fft(x))[:len(t)//2]
55
56     fig, axs = plt.subplots(2, figsize=(8, 4.5))
57     fig.tight_layout()
58
59     sns.lineplot(x=t, y=x, ax=axs[0])
60     sns.lineplot(x=f, y=H, ax=axs[1])
61
```

```

62     axs[0].set_xlabel("Time [s]")
63     axs[1].set_xlabel("Frequency [Hz]")
64
65     return fig, axs
66
67     ### ENTRYPOINT #####
68
69     def run_question_6(x_samp: np.array):
70         """
71         Performs sinc interpolation to upsample the given signal to 80 Hz and plots
72         the results in the time and Fourier domains.
73         """
74         # Upsample from 20 Hz to 80 Hz
75         x_upsamp = sinc_interpolate(x_samp, 4, viz=True)
76         t_upsamp = np.linspace(0, 1, 80)
77
78         fig, axs = time_fourier_plot(t_upsamp, x_upsamp)
79         axs[1].set_xlim(-5, 105)
80
81         fname = Path(A1_ROOT, "output", "q6_upsampled.png")
82         fig.savefig(fname, **SAVEFIG_CONFIG)
83
84     def run_question_7():
85         """
86         Performs zero-padding in the Fourier domain to upsample the given signal to
87         80 Hz and plots the results in the the time and Fourier domains.
88         """
89
90     def main():
91
92         # "Continuous time" 7 Hz sine wave (actually sampled at 1 kHz)
93         t = np.linspace(0, 1, 1000)
94         x = np.sin(2 * np.pi * 7 * t)
95
96         # Plot in the time and Fourier domains
97         fig, axs = time_fourier_plot(t, x)
98         axs[1].set_xlim(-5, 105)
99
100        fname = Path(A1_ROOT, "output", "q6_sine7hz.png")
101        fig.savefig(fname, **SAVEFIG_CONFIG)
102
103        # Sine wave sampled at 20 Hz
104        t_samp = t[::1000//20]
105        x_samp = x[::1000//20]
106
107        # Plot in the time and Fourier domains
108        fig, axs = time_fourier_plot(t_samp, x_samp)
109        axs[1].set_xlim(-5, 105)
110
111        fname = Path(A1_ROOT, "output", "q6_sampled.png")
112        fig.savefig(fname, **SAVEFIG_CONFIG)
113
114        run_question_6(x_samp)
115
116
117     if __name__ == "__main__":
118         main()

```