ELEC4620 Assignment 1

Deren Teo

August 17, 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.
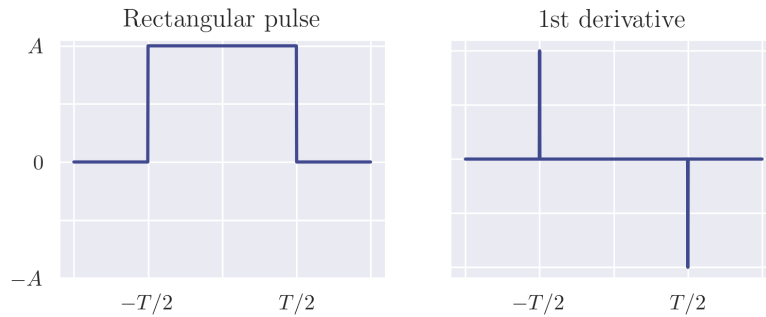

Figure 1.1: A rectangular pulse and its first derivative

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:

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

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.
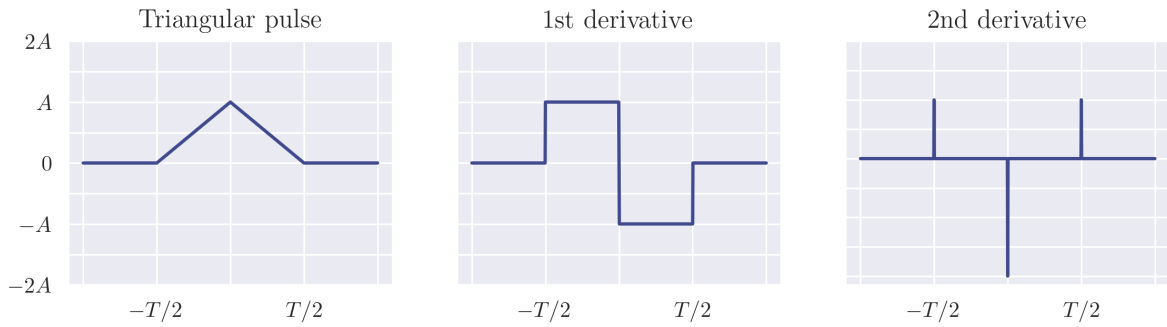

Figure 1.2: A triangular pulse and its first and second derivatives

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 nth 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.
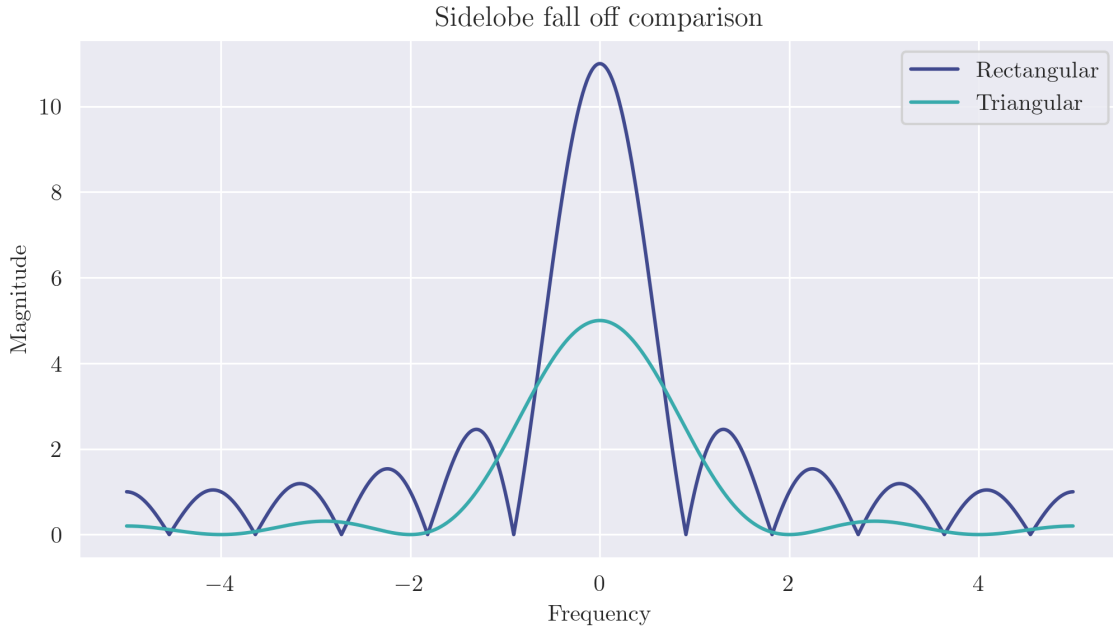

Figure 1.3: Fourier transform sidelobe fall off comparison: rectangular and triangular pulses.

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:

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

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,

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

where M and N are the lengths of v_X and v_Y, respectively, and $(v_X * v_Y)_i$ denotes the ith 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:

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

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`:

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

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:

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

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

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

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,

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

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:

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

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:

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

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

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

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.

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

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

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

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:

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

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:

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

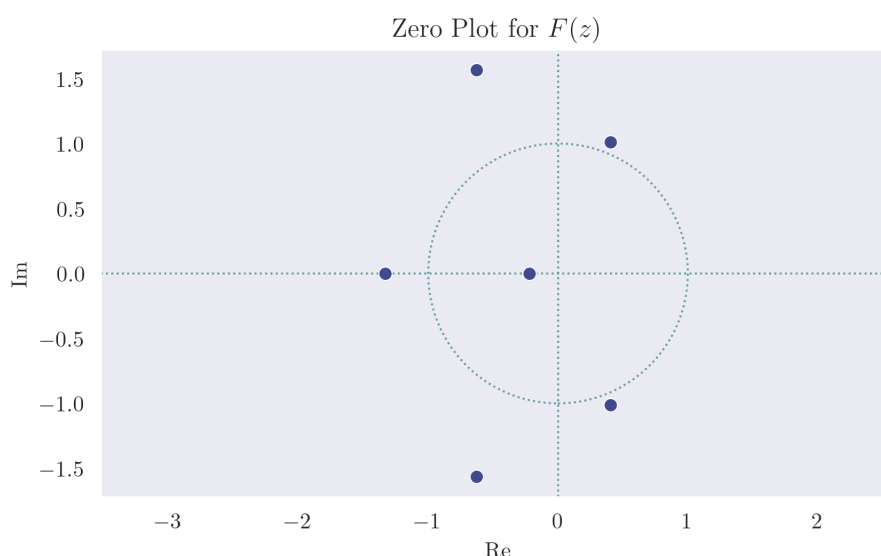Figure 5.1 plots these on the complex plane, with the unit circle for reference.

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,

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

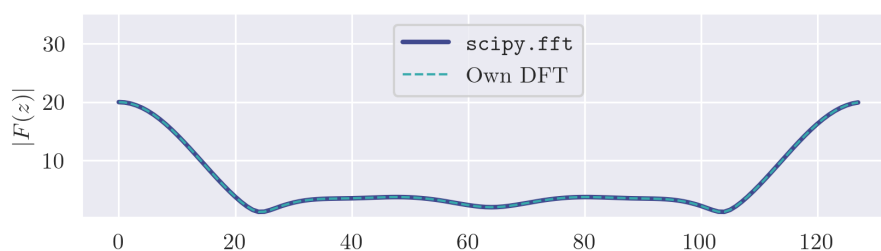Figure 5.2 compares `scipy.fft` and a self-implemented DFT function for $N = 128$.

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

8

Question 6

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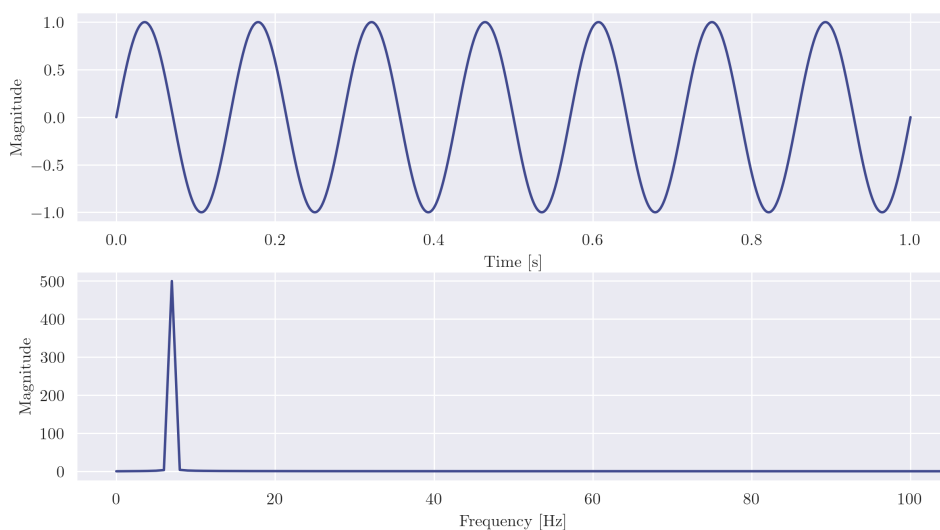

Figure 6.1: Time and frequency views of a 7 Hz sine wave (rendered at 1 kHz)

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.
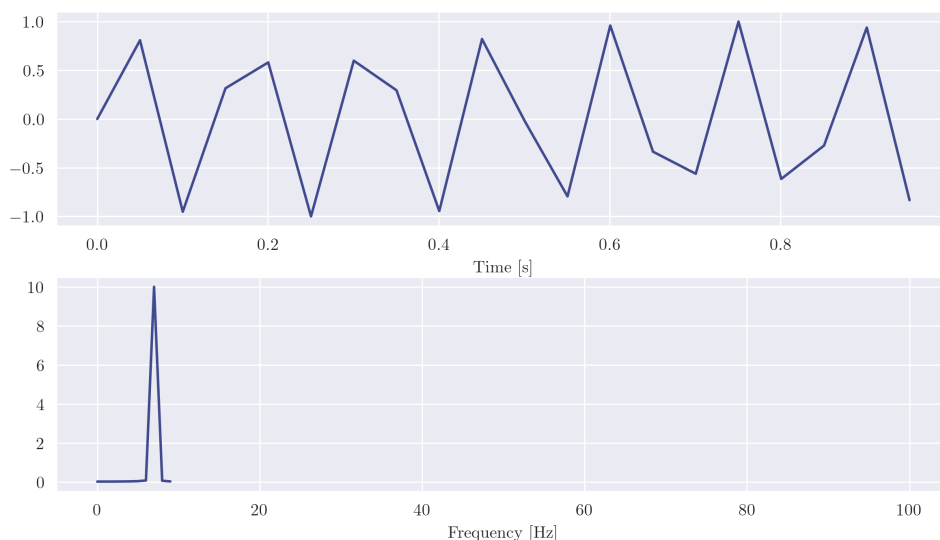

Figure 6.2: Time and frequency views of the 7 Hz sine wave sampled at 20 Hz

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.

9

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:
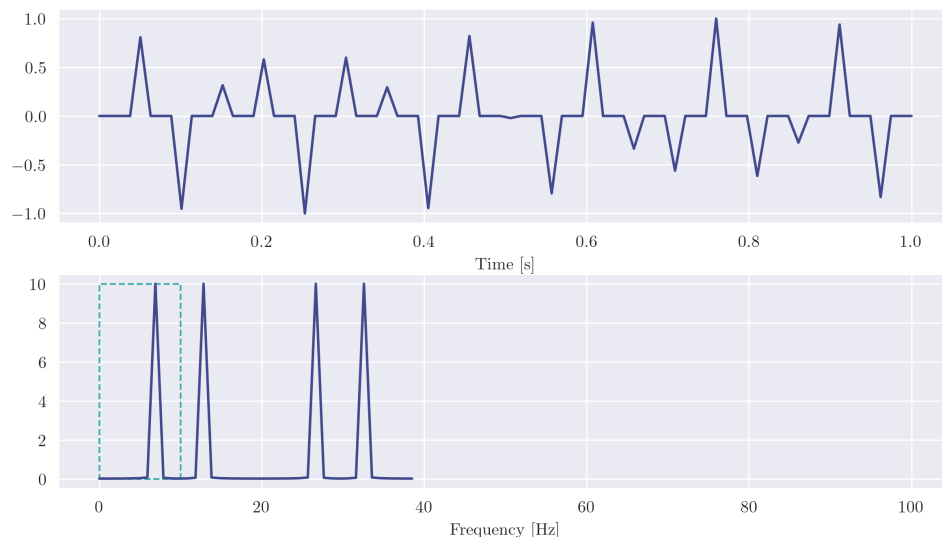

Figure 6.3: Time and frequency views of the intermediate upsampled signal

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.
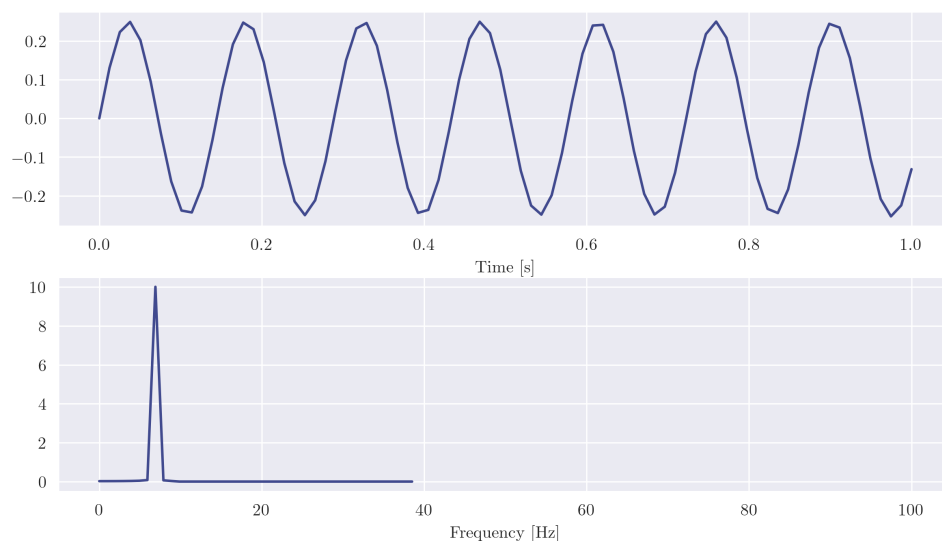

Figure 6.4: Time and frequency views of the 80 Hz sinc interpolated signal

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.
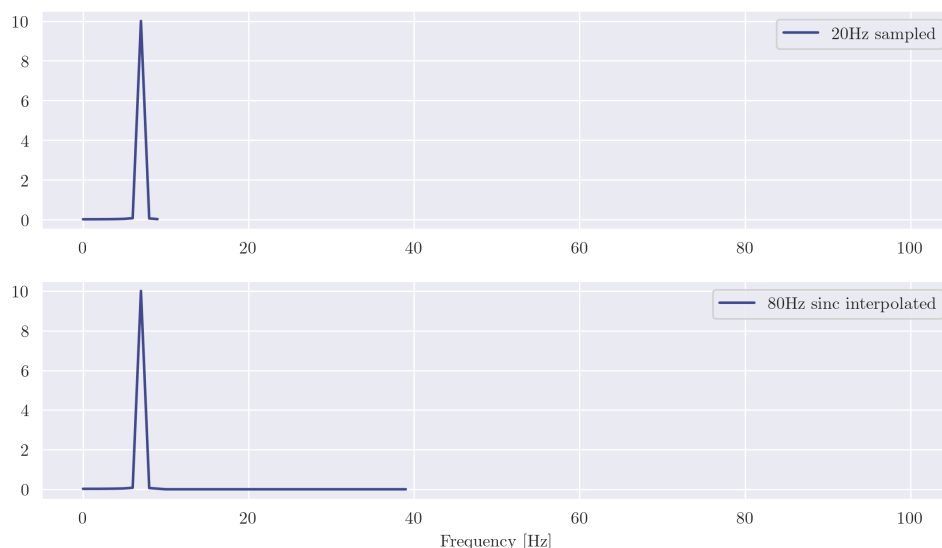

Figure 7.1: Frequency comparison: 20 Hz sampled signal and 80 Hz sinc interpolated signal

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.
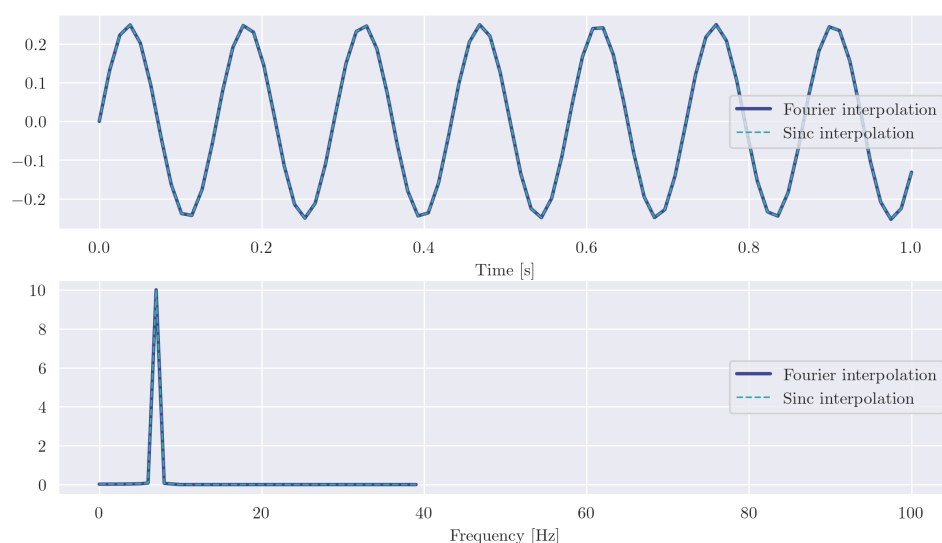

Figure 7.2: Time and frequency views of the 80 Hz Fourier interpolated signal

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

Question 8

This question aims to use the Fourier transform to determine the duration of a solar cycle using annual sunspot data from 1700 to 2018. There are several assumptions associated with this:

- the solar cycle is approximately sinusoidal;

- the sampling rate of the data is 1 year;

- the Nyquist effects are negligible in the data; and

- the data has a DC gain; i.e. a non-variable number of sunspots occuring each year.

Given these assumptions hold, the following analysis should find the average number of years in the solar cycle. We begin by observing the annual sunspot data, presented in Figure 8.1.
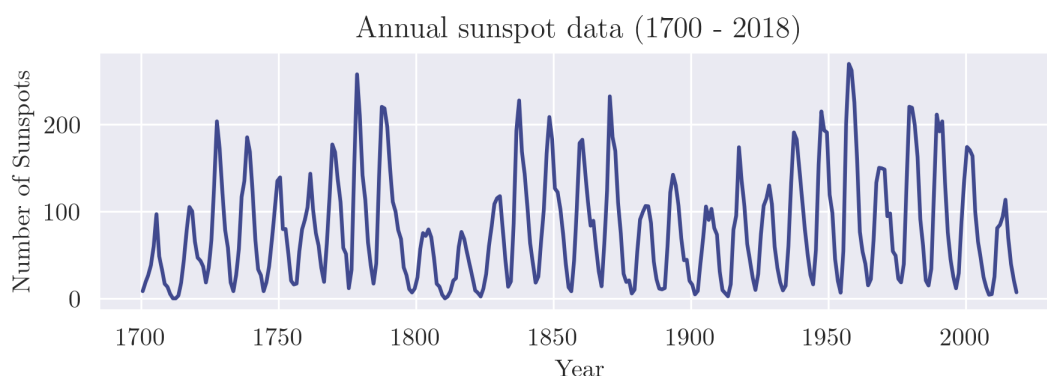
Figure 8.1: Annual sunspot data from 1700 to 2018

The data appears to demonstrate two superposed cyclic trends: one of approximately 10 year period, and another of approximately 80 year period. We will use the DFT to examine this.
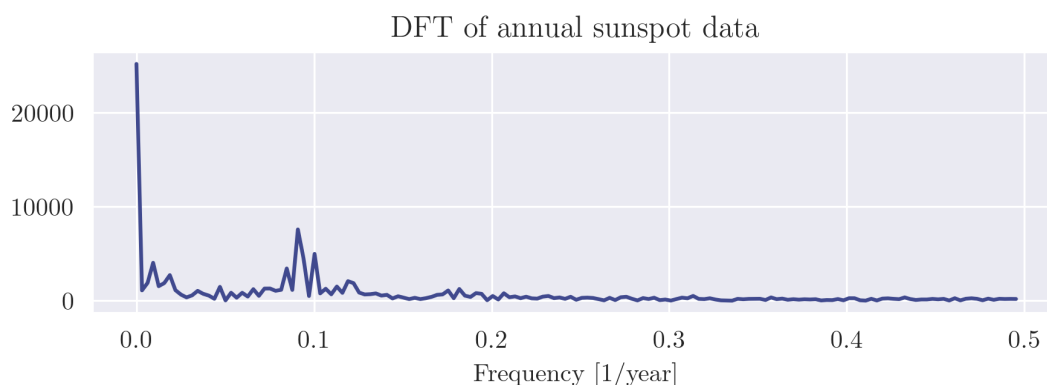
Figure 8.2: Fast Fourier transform applied to the annual sunspot data

Taking the DFT of the data, we find validation of the DC gain assumption, and not much else due to the relative magnitude of the DC gain. To address this, it is necessary to remove the DC gain by subtracting the mean of the data from the entire series. Doing so and repeating the DFT, we find the result of Figure 8.3.
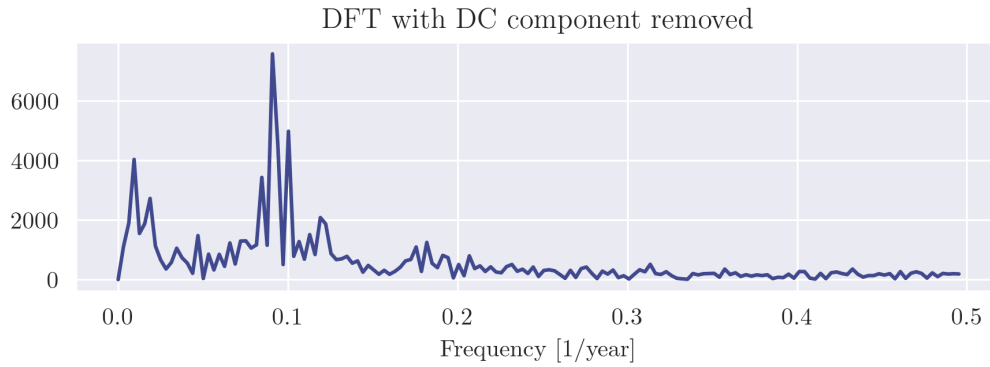
Figure 8.3: Fast Fourier transform applied after removing DC gain

Having removed the DC gain, the other features in the data become much more apparent. We now turn our attention to the possibility of spurious components in the DFT due to the truncation of the series before the first and after the last years in the data. To investigate this, we apply the rectangular, Hann and Blackman windows to the time series data and compare the resulting Fourier transforms.
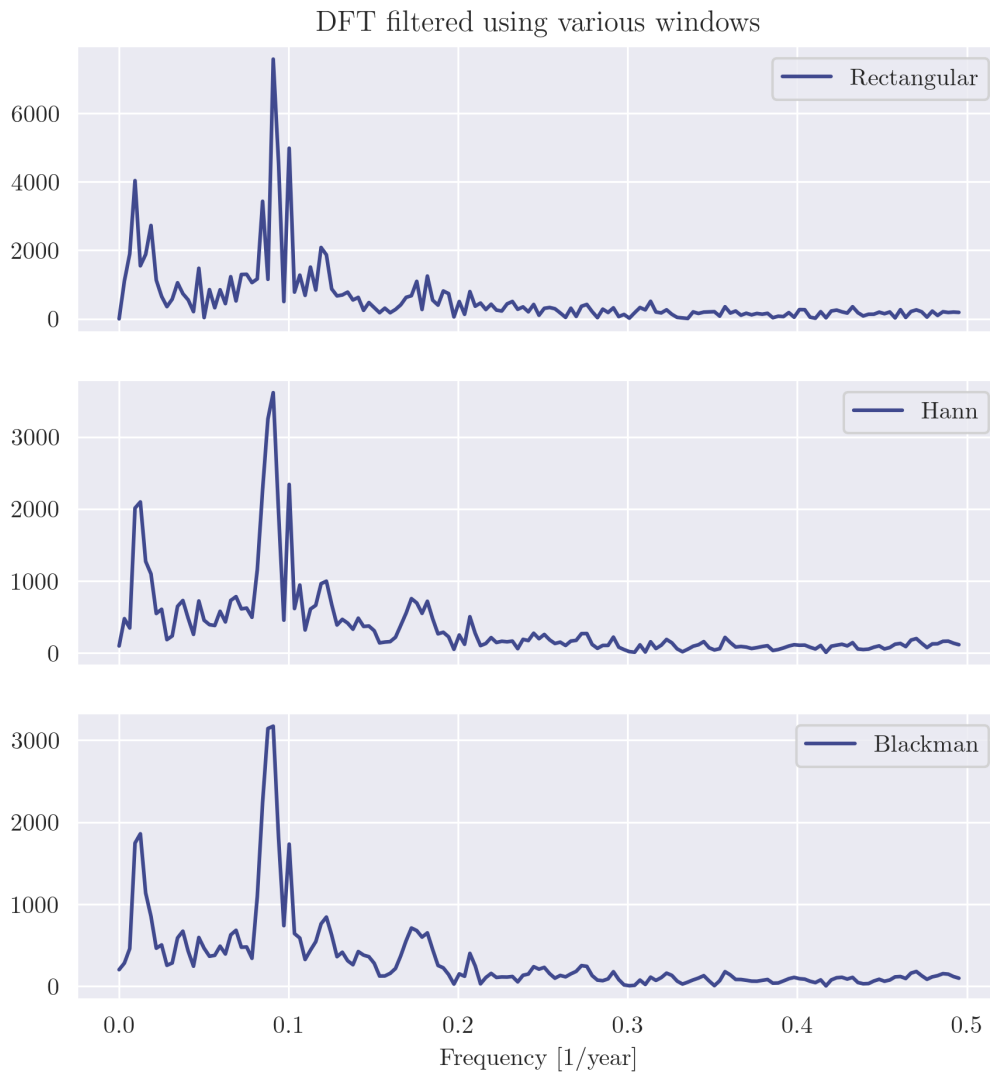
Figure 8.4: Effects of rectangular, Hann and Blackman windows

Addressing the windows in order, we first observe that applying the rectangular window has no visible effect. This is because the original series, truncated to the range 1700 to 2018, is effectively multiplied by a rectangular window around the same range. Hence, applying the rectangular window again is redundant. Meanwhile, the Hann and Blackman windows emphasise the peaks in the Fourier transform and suppress the other frequencies; the effect is more pronounced for the Blackman window than the Hann window. As we are after the maxima in the Fourier transform, we prefer the Blackman window over the other two.

To precisely locate the maxima, we will refer to the coarse and fine search algorithm described by Rife and Boorstyn[1]. However, whereas the Rife and Boorstyn algorithm finds a single global maximum, we are interested in identifying two peaks. Therefore, we will apply the algorithm twice separately on the frequency ranges $[0, 0.5]$ and $[0.5, 1.5]$.

The Rife and Boorstyn algorithm searches for a maximum of $|A(\omega)|$, where $A(\omega)$ is defined as

$$A(\omega) = \frac{1}{N} \sum_{n=0}^{N-1} Z_n \exp(-jn\omega T)$$

in which Z_n is the n-th observation of the discrete signal Z of total length N, and the frequency ω is defined between 0 and ω_s, the sampling frequency[1]. The sampling period T is the inverse of ω_s. The coarse search evaluates $|A(\omega)|$ at the set of frequencies $\{\omega_k\}$ defined by:

$$\omega_k = \frac{2\pi k}{MT}, \quad k = 0, 1, 2, \ldots, M-1$$

where M is $2N$, $4N$, or $8N$[1]. This is equivalent to evaluating the DFT of the set $\{\bar{Z}_n\}$[1]:

$$\bar{Z}_n = \begin{cases} (M/N)Z_n, & n = 0, 1, 2, \ldots, N-1 \\ 0, & n = N, N+1, \ldots, M \end{cases}$$

Hence, the coarse search can be performed by constructing the set $\{\bar{Z}_n\}$ and returning the discrete frequency with the largest amplitude in the DFT of $\{\bar{Z}_n\}$[1]. Applied to the sunspot data and using $M = 2N$, the coarse search finds the two peaks of interest at frequencies 0.089 and 0.011 per year, corresponding with periods of approximately 11.2 and 91.1 years.

The fine search performs a narrow uni-directional sweep of the region around the coarse result. Let us denote an arbitrary coarse search result as ω_l. The direction of the sweep depends on the gradient of $|A(\omega_l)|$; if the gradient of $|A(\omega_l)|$ is positive, the desired maximum is at a value of $\omega > \omega_l$, else if the gradient of $|A(\omega_l)|$ is negative, the desired maximum $\omega < \omega_l$[1].

As recommended by Rife[2], the fine search iterates in steps of size $\omega_s/(5N)$. The search terminates when the gradient of $|A(\omega)|$ crosses zero[2]. Finally, the last two points on either side of the zero are input to the secant root-finding method[3]. The secant method is considered to have converged once the difference between successive approximations is less than half the desired precision; we choose a precision of five decimal places, which will yield the required three decimal places in the reciprocals.

Applying the fine search to the coarse results from the sunspot data, the peaks are found at frequencies of approximately 0.01115 and 0.08947 per year, or equivalently, periods of approximately 11.176 and 89.664 years.

[1]D. C. Rife and R. R. Boorstyn, "Single-tone parameter estimation from discrete-time observations," *IEEE Trans. Inf. Theory*, vol. IT-20, no. 5, pp. 591-598, Sep., 1974.

[2]D. C. Rife, "Digital tone parameter estimation in the presence of Gaussian noise," Ph.D. disseration, Polytech. Inst. Brooklyn, Brooklyn, N.Y., Jun. 1973.

[3]Wikipedia. "Secant method." Wikipedia.org. https://en.wikipedia.org/wiki/Secant_method (accessed Aug. 14, 2023).

Figure 8.5 presents DFT of the Blackman-windowed signal with the identified frequencies.
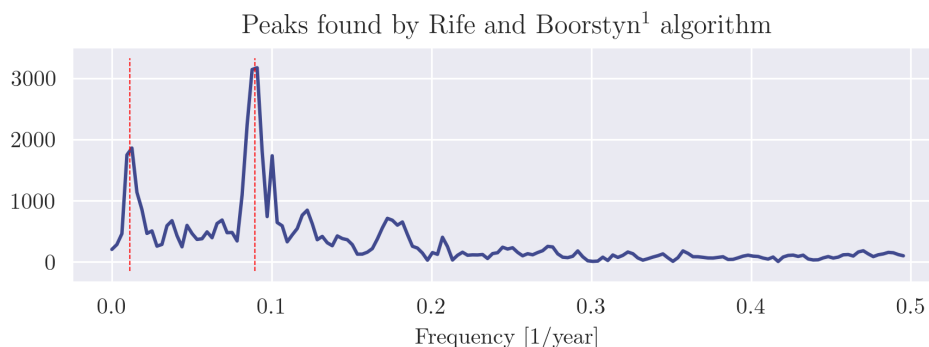
Figure 8.5: DFT of Blackman-windowed signal with identified frequencies in red

Notice also how the identified frequencies occur between points of the DFT, especially evident in the second peak. The Rife and Boorstyn[1] algorithm enables higher precision than simply returning the frequency which produces the largest response in the DFT.

The 11.176-year period in the sunspot data corroborates the general consensus of a solar cycle period of approximately 11 years[4]. However, observations of cycles as short as 9 years or as long as 14 years have been recorded[4], somewhat undermining the precision of the identified value. Similarly, the 89.664-year feature corroborates the period of the Gleissberg cycle, which describes an amplitude modulation of solar cycles with a period between 70 to 100 years[4].

The large variance in both sets of observations makes it more difficult to justify the three decimal point precision from a purely practical standpoint. Nevertheless, this investigation has proven to be a useful exercise in applying DSP methods to extract information from real data.

---

[4]Wikipedia. "Solar cycle." Wikipedia.org. https://en.wikipedia.org/wiki/Solar_cycle (accessed Aug. 14, 2023).

Question 9

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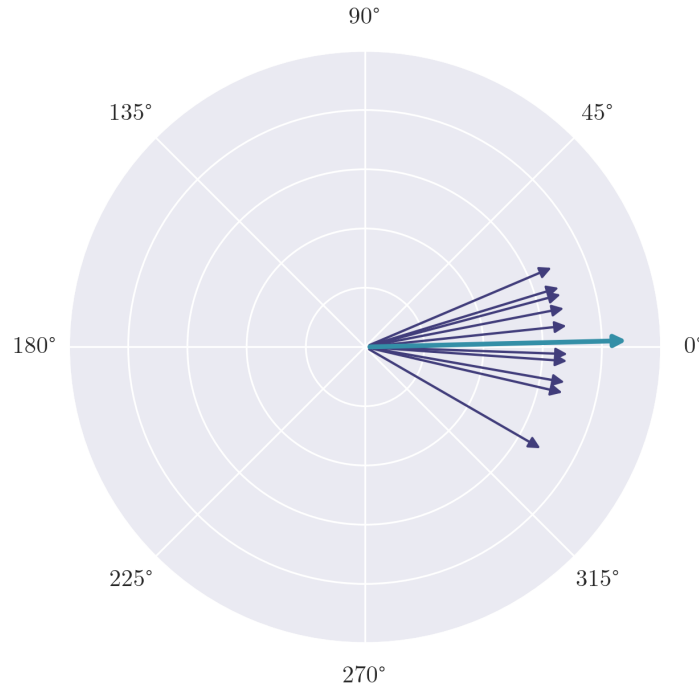
Figure 9.1: Homing pigeon departure headings and average heading

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 Code Listings

A.1 sidelobes.py

```python
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))
```

```python
57            axs[i].set_yticklabels([])
58        axs[0].set_yticklabels(["$-A$", "", 0, "", "$A$"])
59
60        fname = Path(A1_ROOT, "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(A1_ROOT, "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_tria = 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_tria, 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()
```

A.2 `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():
```

```
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()
```

A.3 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()
```

A.4 `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 A1_ROOT, PLT_CONFIG, SAVEFIG_CONFIG, SNS_STYLE
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
```

```python
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() = }")
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          # Re-set the default Seaborn style, which was changed by part (a)
138          sns.set_style(SNS_STYLE)
139
140          # Re-set the plot text customisation, which gets overridden by set_style
141          plt.rcParams.update(PLT_CONFIG)
142
143          fig, ax = plt.subplots()
144
145          sns.lineplot(x=np.arange(128), y=y_fft, ax=ax, ls="-",  lw=2,
146               label=r"$\texttt{scipy.fft}$")
147          sns.lineplot(x=np.arange(128), y=y_dft, ax=ax, ls="--", lw=1,
148               label=r"Own DFT")
149
150          ax = axes_ratio_scale(ax, ratio=1/4, padto="upper")
151
152          ax.set_title("")
153          ax.set_xlabel("")
154          ax.set_ylabel("$|F(z)|$")
155
156          ax.legend(loc="upper center")
157
158          fname = Path(A1_ROOT, "output", "q5b_dftsample.png")
159          fig.savefig(fname, **SAVEFIG_CONFIG)
160
161     def main():
162
163          poly = Polynomial([1, 5, 3, 4, 4, 2, 1])
164          run_part_a(poly)
165          run_part_b(poly)
166
167
168     if __name__ == "__main__":
169          main()
```

A.5 `interpolation.py`

```python
1  """
2  Script associated with Q6 and 7.
3
4  Performs two methods of upsamping 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 def fourier_interpolate(x: np.array, n: int) -> np.array:
48     """
49     Upsamples the given signal by the specified factor by applying the Fourier
50     transform, zero-padding, then inverse Fourier transforming.
51     """
52     H = fft(x); N = len(H)
53
54     # Pad N*(n-1) zeros between positive and negative frequencies
55     H_upsamp = np.concatenate([H[:N//2], np.zeros(len(x)*(n-1)), H[N//2:]])
56     x_upsamp = ifft(H_upsamp)
57
58     return x_upsamp
59
60 ### VISUALISATION ############################################################
61
62 def time_fourier_plot(t: np.array, x: np.array) -> tuple[Figure, list[Axes]]:
```

26

```
63          """
64          Plot the given signal and its discrete Fourier transform.
65          """
66          f = fftfreq(n=len(t), d=(t[1]-t[0]))[:len(t)//2]
67          H = np.abs(fft(x))[:len(t)//2]
68
69          fig, axs = plt.subplots(2, figsize=(8, 4.5))
70          fig.tight_layout()
71
72          sns.lineplot(x=t, y=x, ax=axs[0])
73          sns.lineplot(x=f, y=H, ax=axs[1])
74
75          axs[0].set_xlabel("Time_[s]")
76          axs[1].set_xlabel("Frequency_[Hz]")
77
78          return fig, axs
79
80  ### ENTRYPOINT ###################################################################
81
82  def run_question_6(x_samp: np.array):
83          """
84          Performs sinc interpolation to upsample the given signal to 80 Hz and plots
85          the results in the time and Fourier domains.
86          """
87          # Upsample from 20 Hz to 80 Hz
88          x_upsamp = sinc_interpolate(x_samp, 4, viz=True)
89          t_upsamp = np.linspace(0, 1, 80)
90
91          fig, axs = time_fourier_plot(t_upsamp, x_upsamp)
92          axs[1].set_xlim(-5, 105)
93
94          fname = Path(A1_ROOT, "output", "q6_upsampled.png")
95          fig.savefig(fname, **SAVEFIG_CONFIG)
96
97  def run_question_7(x_samp: np.array):
98          """
99          Performs zero-padding in the Fourier domain to upsample the given signal to
100         80 Hz and plots the results in the the time and Fourier domains.
101         """
102         # Upsample from 20 Hz to 80 Hz
103         x_upsamp = fourier_interpolate(x_samp, 4)
104         t_upsamp = np.linspace(0, 1, 80)
105
106         # Plot a before/after frequency domain comparison
107         fig, axs = plt.subplots(2, figsize=(8, 4.5))
108         fig.tight_layout()
109
110         f_samp = fftfreq(n=20, d=1/20)[:len(x_samp)//2]
111         H_samp = np.abs(fft(x_samp))[:len(x_samp)//2]
112
113         N_upsamp = len(x_upsamp)
114         f_upsamp = fftfreq(n=80, d=1/80)[:N_upsamp//2]
115         H_upsamp = np.abs(fft(x_upsamp))[:N_upsamp//2]
116
117         sns.lineplot(x=f_samp,   y=H_samp,   ax=axs[0], label="20Hz_sampled")
118         sns.lineplot(x=f_upsamp, y=H_upsamp, ax=axs[1], label="80Hz_sinc_interpolated")
119
120         axs[0].set_xlim([-5, 105])
121         axs[1].set_xlim([-5, 105])
122
123         axs[1].set_xlabel("Frequency_[Hz]")
124
125         fname = Path(A1_ROOT, "output", "q7_freqcompare.png")
```

```
126         fig.savefig(fname, **SAVEFIG_CONFIG)
127
128         # Plot a comparison this method and sinc interpolation
129         x_upsinc = sinc_interpolate(x_samp, 4)
130         H_upsinc = np.abs(fft(x_upsinc))[:N_upsamp//2]
131
132         fig, axs = plt.subplots(2, figsize=(8, 4.5))
133         fig.tight_layout()
134
135         sns.lineplot(x=t_upsamp, y=x_upsamp, ax=axs[0], ls="-",   lw=2,
136             label="Fourier_interpolation")
137         sns.lineplot(x=t_upsamp, y=x_upsinc, ax=axs[0], ls="--", lw=1,
138             label="Sinc_interpolation")
139
140         sns.lineplot(x=f_upsamp, y=H_upsamp, ax=axs[1], ls="-",   lw=2,
141             label="Fourier_interpolation")
142         sns.lineplot(x=f_upsamp, y=H_upsinc, ax=axs[1], ls="--", lw=1,
143             label="Sinc_interpolation")
144
145         axs[0].set_xlabel("Time_[s]")
146         axs[1].set_xlabel("Frequency_[Hz]")
147
148         axs[0].legend(loc="center_right")
149         axs[1].legend(loc="center_right")
150
151         axs[1].set_xlim(-5, 105)
152
153         fname = Path(A1_ROOT, "output", "q7_upsampled.png")
154         fig.savefig(fname, **SAVEFIG_CONFIG)
155
156     def main():
157
158         # "Continuous time" 7 Hz sine wave (actually sampled at 1 kHz)
159         t = np.linspace(0, 1, 1000)
160         x = np.sin(2 * np.pi * 7 * t)
161
162         # Plot in the time and Fourier domains
163         fig, axs = time_fourier_plot(t, x)
164         axs[1].set_xlim(-5, 105)
165
166         fname = Path(A1_ROOT, "output", "q6_sine7hz.png")
167         fig.savefig(fname, **SAVEFIG_CONFIG)
168
169         # Sine wave sampled at 20 Hz
170         t_samp = t[::1000//20]
171         x_samp = x[::1000//20]
172
173         # Plot in the time and Fourier domains
174         fig, axs = time_fourier_plot(t_samp, x_samp)
175         axs[1].set_xlim(-5, 105)
176
177         fname = Path(A1_ROOT, "output", "q6_sampled.png")
178         fig.savefig(fname, **SAVEFIG_CONFIG)
179
180         ### QUESTION 6 ###
181         run_question_6(x_samp)
182
183         ### QUESTION 7 ###
184         run_question_7(x_samp)
185
186
187     if __name__ == "__main__":
188         main()
```

A.6   `sunspots.ipynb`

(Starts on the following page)

Question 8

This script determines the number of years in a solar cycle using annual sunspot data.

```python
from pathlib import Path
from typing import Callable, Tuple

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

from scipy.fft import fft, fftfreq
from scipy.signal.windows import blackman, boxcar, hann

from config import A1_ROOT, SAVEFIG_CONFIG
```
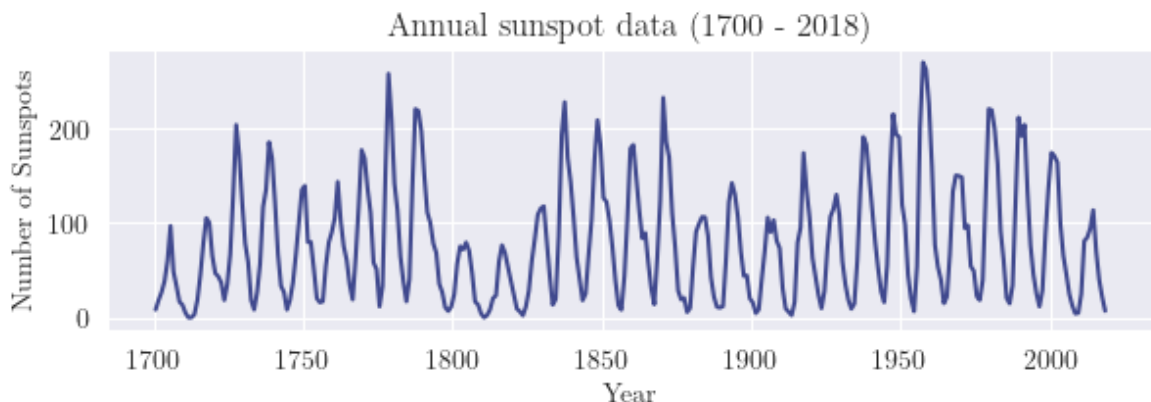
```python
data = pd.read_csv(Path(A1_ROOT, "data", "SunspotData.csv"))
```

```python
# Plot the time series data
fig, ax = plt.subplots(figsize=(6, 2))
fig.tight_layout()

sns.lineplot(data, x="Year", y="Number of Sunspots", ax=ax)
ax.set_title("Annual sunspot data (1700 - 2018)")

fname = Path(A1_ROOT, "output", "q8_timeseries.png")
fig.savefig(fname, **SAVEFIG_CONFIG)
```
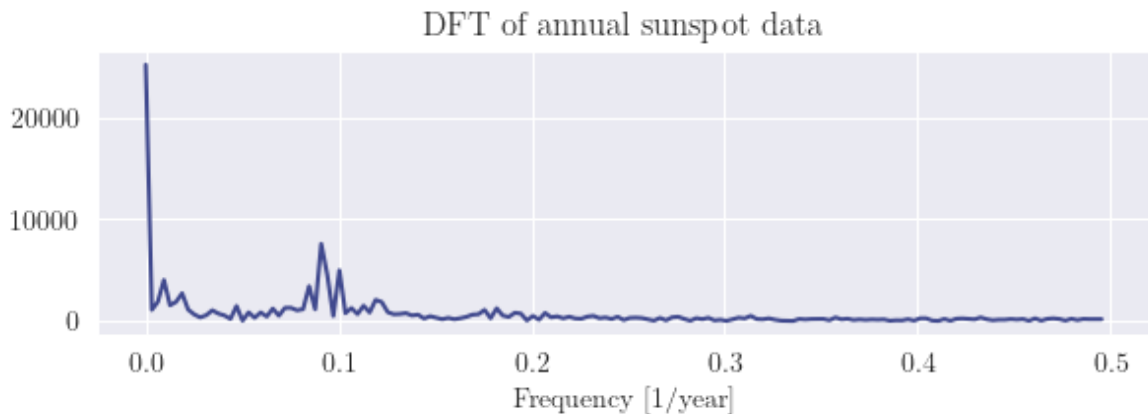


```python
# Plot the DFT
N = data["Year"].count()
f = fftfreq(N, 1)[:N//2]
H = np.abs(fft(data["Number of Sunspots"].to_numpy())[:N//2])

fig, ax = plt.subplots(figsize=(6, 2))
fig.tight_layout()

sns.lineplot(x=f, y=H, ax=ax)

ax.set_title("DFT of annual sunspot data")
ax.set_xlabel("Frequency [1/year]")
```

```python
fname = Path(A1_ROOT, "output", "q8_rawdft.png")
fig.savefig(fname, **SAVEFIG_CONFIG)
```

DFT of annual sunspot data


```python
In [ ]: # Remove the DC component and plot the DFT again
x = (data["Number of Sunspots"] - data["Number of Sunspots"].mean()).to_n
H = np.abs(fft(x)[:N//2])

fig, ax = plt.subplots(figsize=(6, 2))
fig.tight_layout()

sns.lineplot(x=f, y=H, ax=ax)

ax.set_title("DFT with DC component removed")
ax.set_xlabel("Frequency [1/year]")

fname = Path(A1_ROOT, "output", "q8_nodcdft.png")
fig.savefig(fname, **SAVEFIG_CONFIG)
```

DFT with DC component removed
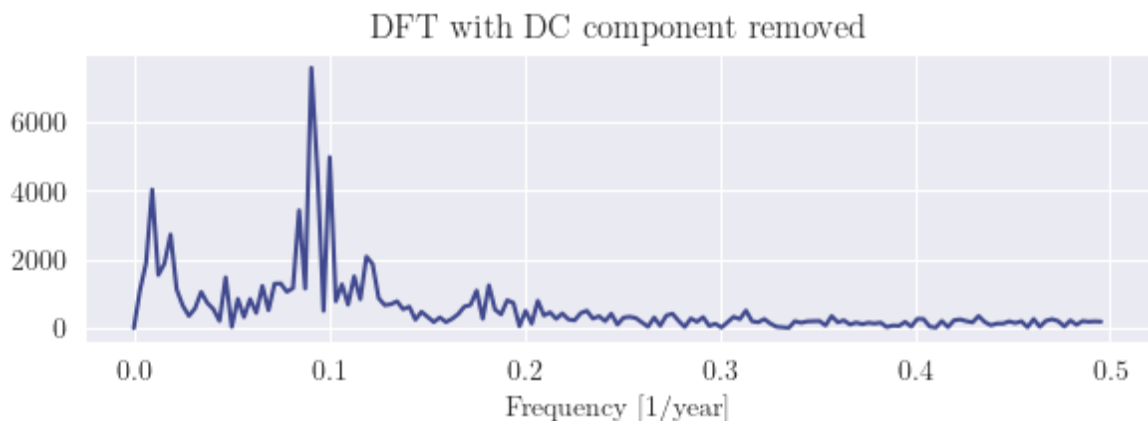

```python
In [ ]: # Try filtering the signal using each of: blackman, boxcar, and hann wind
fig, axs = plt.subplots(3, figsize=(6, 6), sharex=True)
fig.tight_layout()

# 1. Boxcar (rectangular) window
x_boxcar = x * boxcar(N)
H_boxcar = np.abs(fft(x_boxcar)[:N//2])
sns.lineplot(x=f, y=H_boxcar, ax=axs[0], label="Rectangular")

# 2. Hann window
x_hann = x * hann(N)
H_hann = np.abs(fft(x_hann)[:N//2])
sns.lineplot(x=f, y=H_hann, ax=axs[1], label="Hann")
```
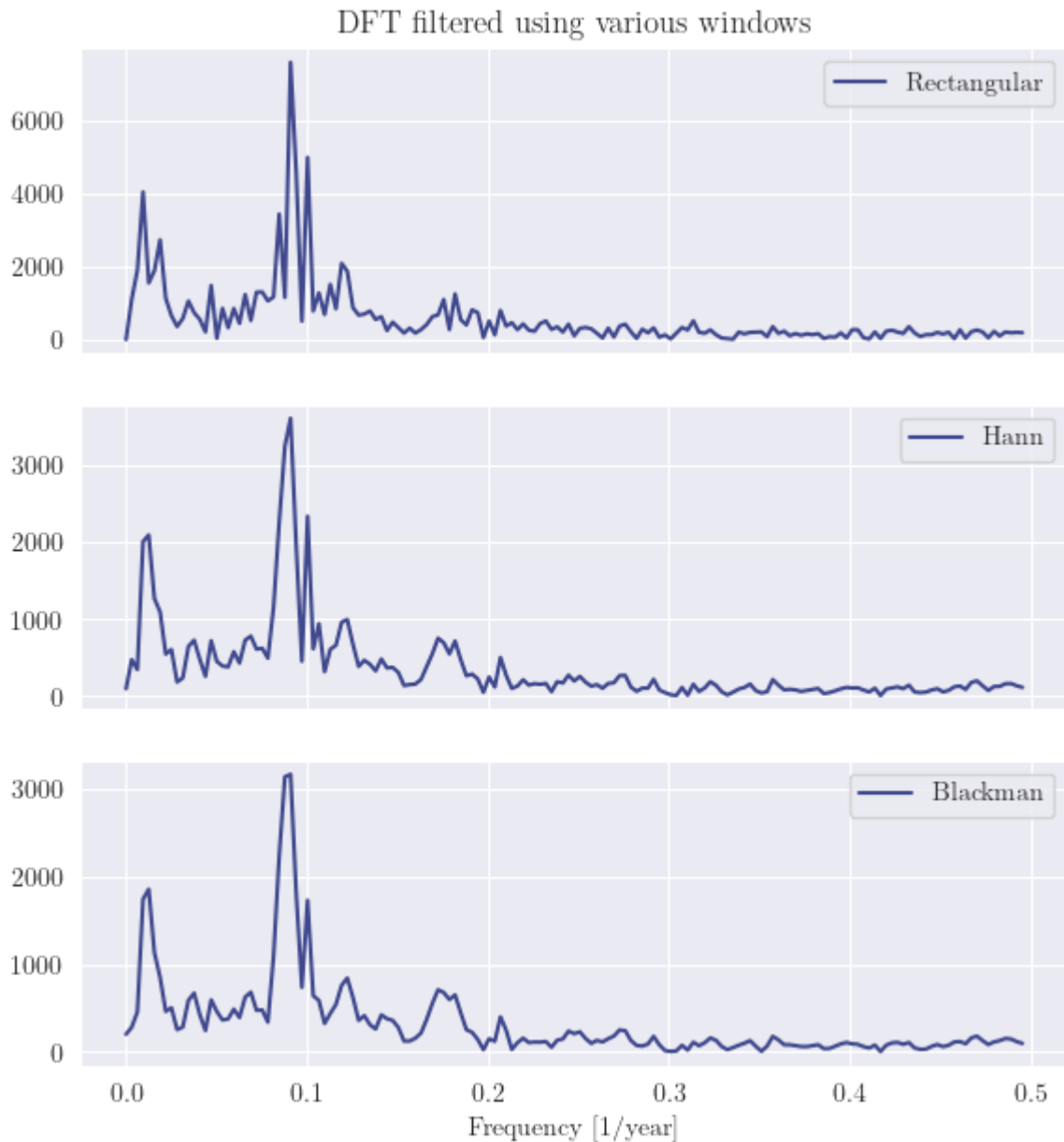
```
# 3. Blackman window
x_blackman = x * blackman(N)
H_blackman = np.abs(fft(x_blackman)[:N//2])
sns.lineplot(x=f, y=H_blackman, ax=axs[2], label="Blackman")

axs[0].set_title("DFT filtered using various windows")
axs[2].set_xlabel("Frequency [1/year]")

fname = Path(A1_ROOT, "output", "q8_windowed.png")
fig.savefig(fname, **SAVEFIG_CONFIG)
```


Find maxima using coarse and fine peak search algorithm[1].

```
In [ ]: def coarse_search(x: np.array, T: float, k: float = 2,
              clip: Tuple[float, float] = None) -> float:
        """
        Performs the coarse search algorithm described by Rife and Boorstyn [
        The factor `k` is equivalent to (M/N) in [1].
        """
        z = np.concatenate([k * x, np.zeros(len(x) * (k - 1))])
        f = fftfreq(len(z), T)[:len(z)//2]
        B = np.abs(fft(z)[:len(z)//2])
```

```python
        B = B if not clip else B[(f >= clip[0]) & (f <= clip[1])]
        f = f if not clip else f[(f >= clip[0]) & (f <= clip[1])]
        return f[(B == B.max())][0]

rough_max_1 = coarse_search(x_blackman, T=1, k=2, clip=(0.00, 0.05))
rough_max_2 = coarse_search(x_blackman, T=1, k=2, clip=(0.05, 0.15))

print("Coarse search:", [rough_max_1, rough_max_2], "[1/years]")
print("              ", [1/rough_max_1, 1/rough_max_2], "[years]")
```

Coarse search: [0.0109717868338558, 0.08934169278996865] [1/years]
               [91.14285714285714, 11.192982456140351] [years]

```python
def root_secant(f: Callable, x0: float, x1: float, maxiter: int = 100) ->
    """
    Performs the secant method root finding algorithm on the function `f`
    with initial guesses `x0` and `x1`. The initial guesses should be clo
    the desired zero. Returns the root found.
    """
    def stop_condition_met(xi: float, xj: float) -> bool:
        """
        Conditions to terminate the root search and return the latest val
        """
        return (xj == 0) or np.isclose(xi, xj, rtol=0, atol=5e-4)

    while ((maxiter := maxiter - 1) > 0):
        x0 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        if (stop_condition_met(x1, x0)):
            return x0
        x1 = x0 - f(x0) * (x0 - x1) / (f(x0) - f(x1))
        if (stop_condition_met(x0, x1)):
            return x1

    raise RuntimeError("secant method did not converge")

def fine_search(x: np.array, T: float, w0: float) -> float:
    """
    Performs the fine search algorithm described by Rife [2].

    Parameters:
        x - set of discrete time observations of length N
        T - sampling period of x
        w0 - initial guess at frequency maximising A(w)

    Returns:
        Fine approximation of frequency maximising A(w).
    """
    step = 1 / (5 * N * T)

    A = lambda w: sum(x[n] * np.exp(-1j * n * w * T) for n in range(N)) /
    B = lambda w: np.abs(A(w))
    dB = lambda w: (B(w + step / 2) - B(w - step / 2)) / step
    dBf = lambda f: dB(2 * np.pi * f)

    step *= np.sign(dBf(w0))

    p = w0
    while (np.sign(dBf(p0 := p)) == np.sign(dBf(p := p0 + step))):
        continue
```

```
        return root_secant(dBf, p0, p)

fine_max_1 = fine_search(x_blackman, T=1, w0=rough_max_1)
fine_max_2 = fine_search(x_blackman, T=1, w0=rough_max_2)
print("Fine search:", [fine_max_1, fine_max_2], "[1/years]")
print("            ", [1/fine_max_1, 1/fine_max_2], "[years]")
```

```
Fine search: [0.011152801870022704, 0.08947384225381592] [1/years]
             [89.6635672052842, 11.176450846530528] [years]
```

In [ ]:
```
# Display the Blackman-windowed signal with identified frequencies overla
fig, ax = plt.subplots(figsize=(6, 2))
fig.tight_layout()

sns.lineplot(x=f, y=H_blackman, ax=ax)

# Draw vertical lines at identified frequencies
ylim = ax.get_ylim()
ax.vlines(x=[fine_max_1, fine_max_2], ymin=ylim[0], ymax=ylim[1], color="
    lw=0.5, ls="--")

ax.set_title("Peaks found by Rife and Boorstyn$^1$ algorithm")
ax.set_xlabel("Frequency [1/year]")

fname = Path(A1_ROOT, "output", "q8_searchresults.png")
fig.savefig(fname, **SAVEFIG_CONFIG)
```
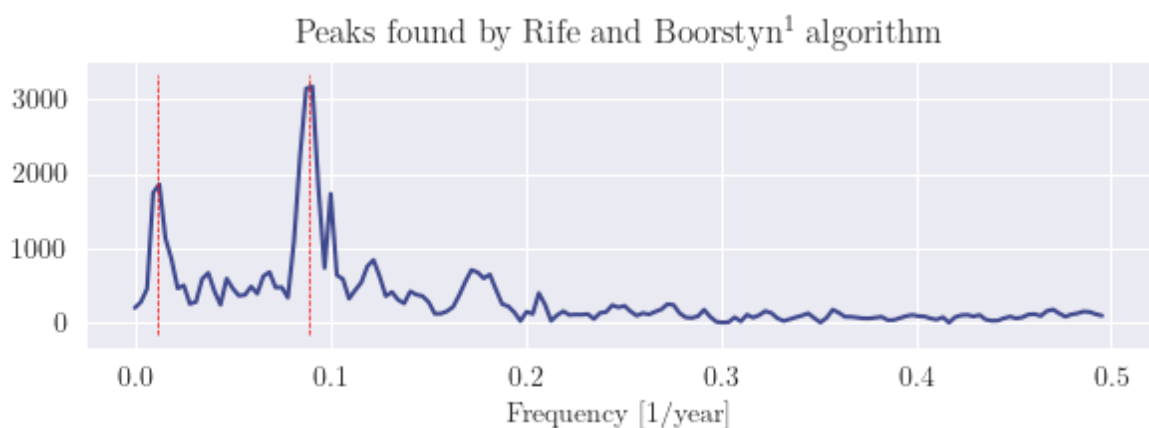

Peaks found by Rife and Boorstyn[1] algorithm

References

[1] D. C. Rife and R. R. Boorstyn, "Single-tone parameter estimation from discrete-time observations," *IEEE Trans. Inf. Theory*, vol. IT-20, no. 5, pp. 591-598, Sep., 1974.

[2] D. C. Rife, "Digital tone parameter estimation in the presence of Gaussian noise," Ph.D. dissertation, Polytech. Inst. Brooklyn, Brooklyn, N.Y., Jun. 1973.

[3] Wikipedia. "Secant method." Wikipedia.org. https://en.wikipedia.org /wiki/Secant_method (accessed Aug. 14, 2023).

A.7 `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 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():
```

```python
63          headings = [11, 15, 350, 330, 23, 347, 17, 356, 6, 358]  # in degrees
64          visualise_headings(headings)
65
66
67  if __name__ == "__main__":
68      main()
```