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

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

Figure 1.1 visualises the rectangular pulse and its first derivative.

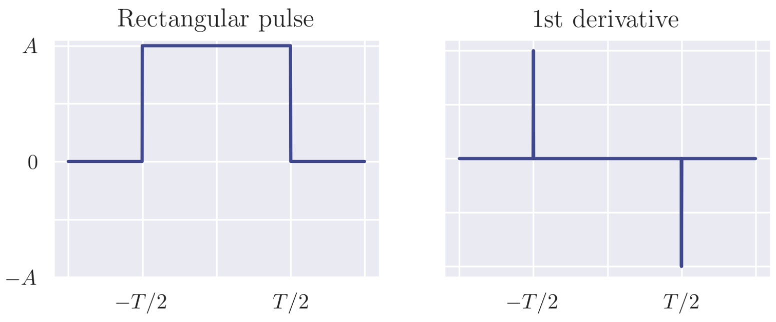


Figure 1.1: A rectangular pulse and its first derivative

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

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

we get the following expression, which has been split into two integrals for simplicity.

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

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

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

Therefore, the Fourier transform of the derivative simplifies to

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

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

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

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

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

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

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

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

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

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

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

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

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

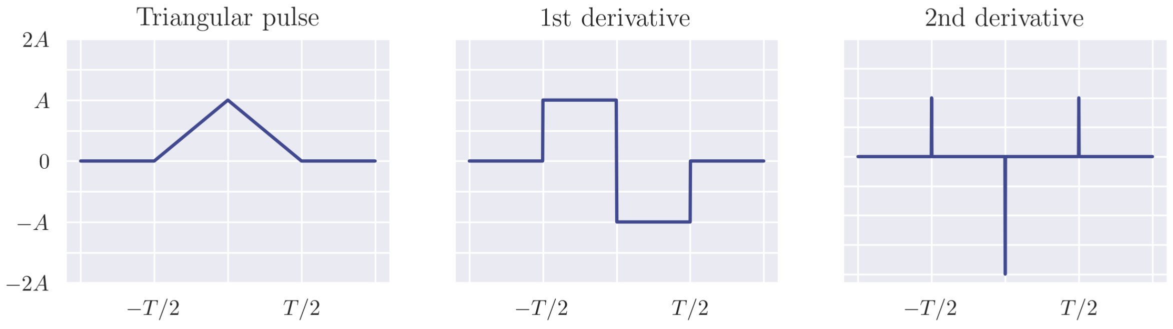


Figure 1.2: A triangular pulse and its first and second derivatives

The Fourier transform of the second derivative is therefore

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

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

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

and further to

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

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

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

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

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

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.

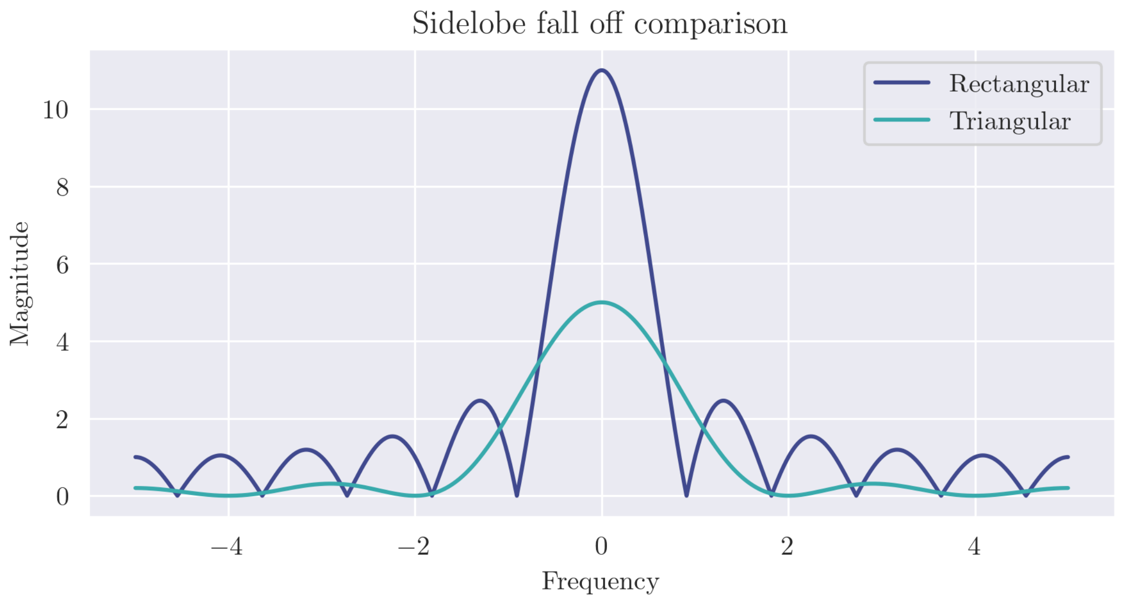


Figure 1.3: Fourier transform sidelobe fall off comparison: rectangular and triangular pulses.

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:

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

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 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 $Y(z)$:

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

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.

```
v_x = np.pad(v_x, (0, len(v_y) - 1))
v_y = np.pad(v_y, (0, len(v_x) - 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(v_x) * fft(v_y))
```

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.

```
v_x = np.pad(v_x, (0, len(v_y) - 1))
v_y = np.pad(v_y, (0, len(v_x) - 1))
```

Then, in the same manner as Question 2:

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

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

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

Question 4

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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,

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

The individual Fourier transforms of x and y are:

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

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

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

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

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

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.

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

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

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

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

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

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

Question 5

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

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

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

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

This finds the following six roots:

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

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

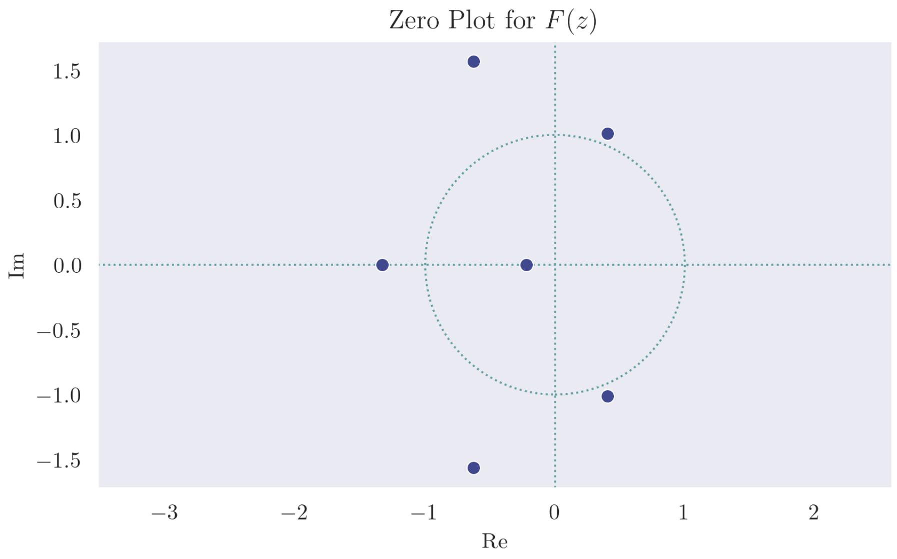


Figure 5.1: Zero plot for $F(z)$; no poles are present.

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

That is,

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

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

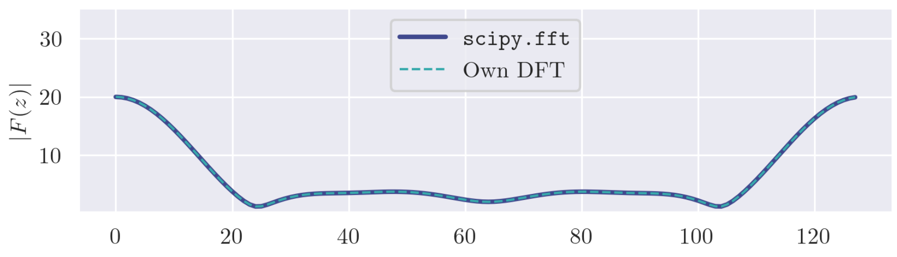


Figure 5.2: $|F(z)|$ evaluated at 128 points around the unit circle.

Question 6

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

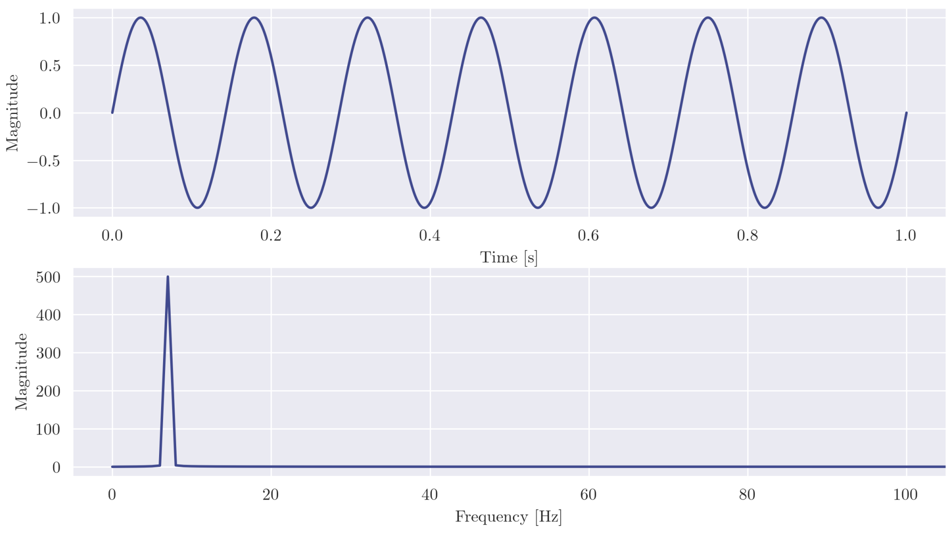


Figure 6.1: Time and frequency views of a 7 Hz sine wave (rendered at 1 kHz)

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

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

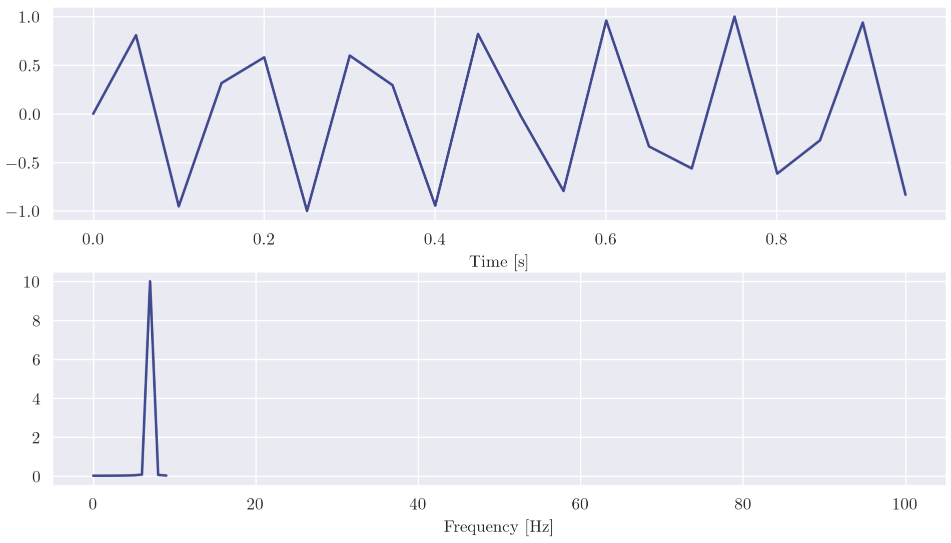


Figure 6.2: Time and frequency views of the 7 Hz sine wave sampled at 20 Hz

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

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

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

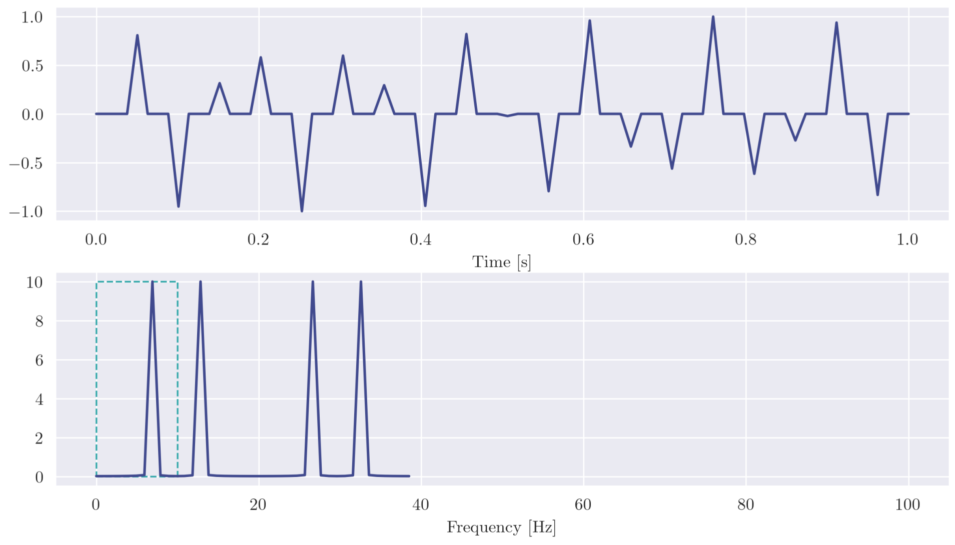


Figure 6.3: Time and frequency views of the intermediate upsampled signal

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

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

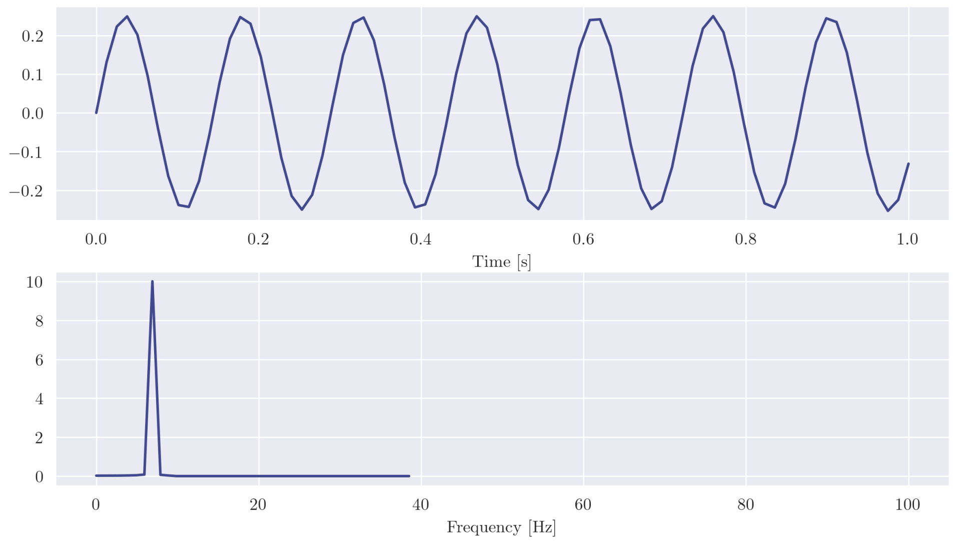


Figure 6.4: Time and frequency views of the 80 Hz sinc interpolated signal

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

Question 7

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

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

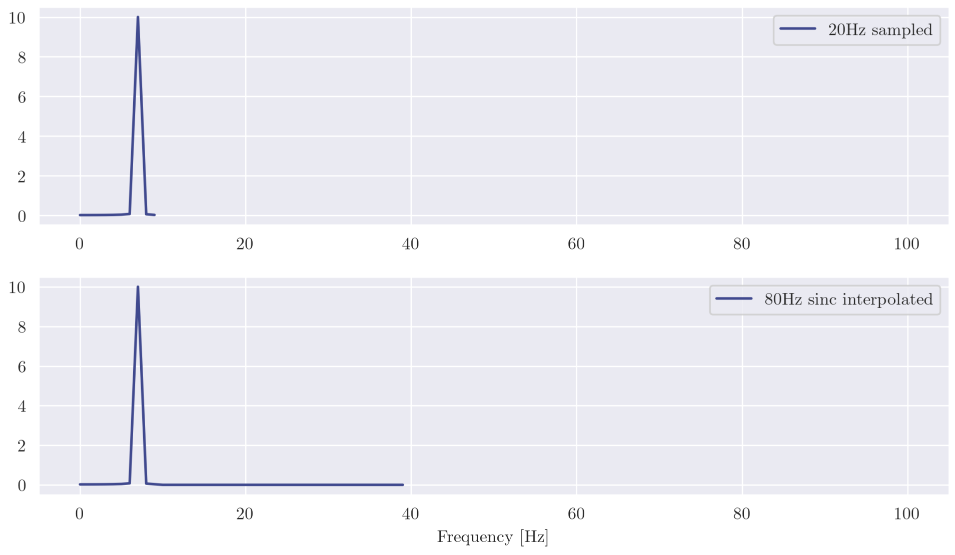


Figure 7.1: Frequency comparison: 20 Hz sampled signal and 80 Hz sinc interpolated signal

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

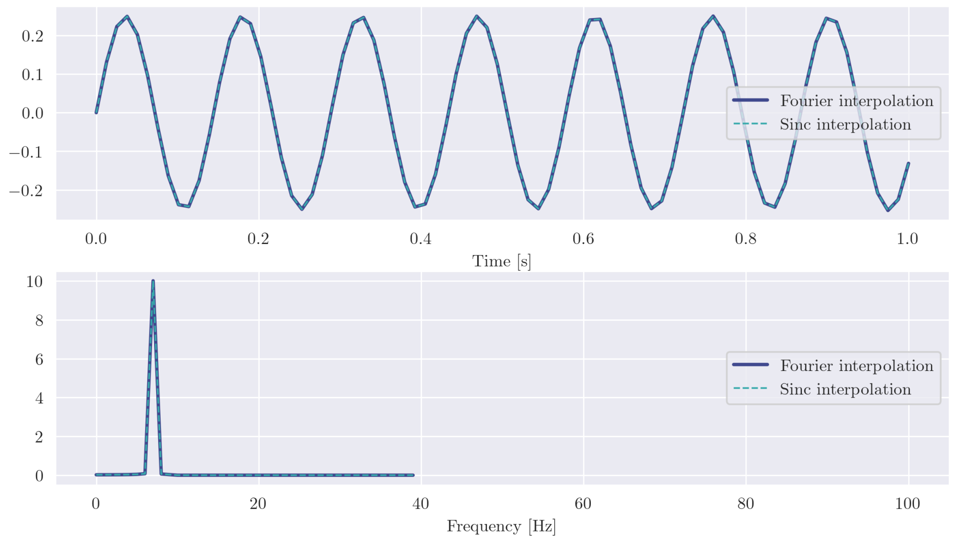


Figure 7.2: Time and frequency views of the 80 Hz Fourier interpolated signal

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.

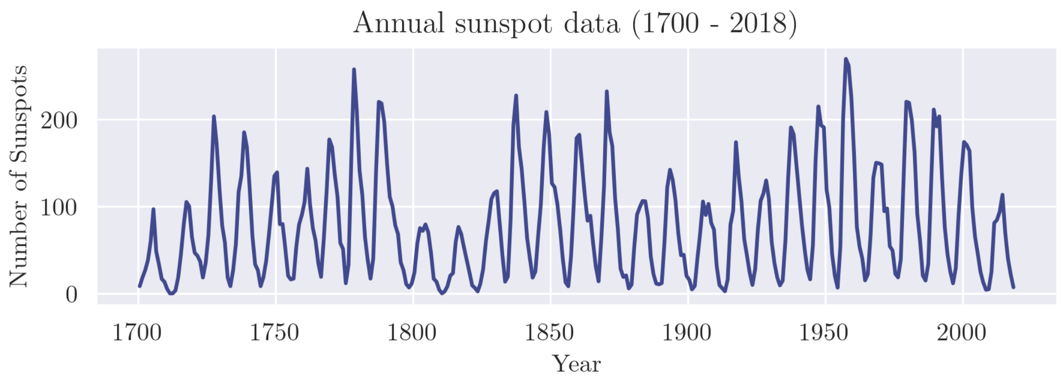


Figure 8.1: Annual sunspot data from 1700 to 2018

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.

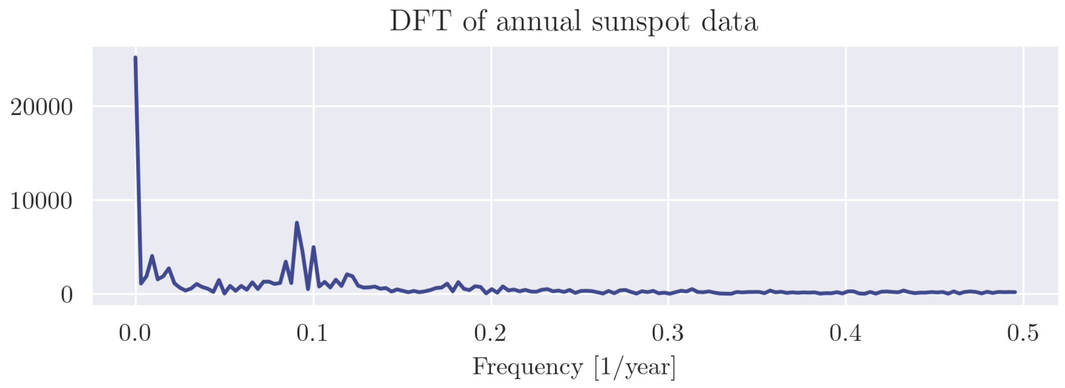


Figure 8.2: Fast Fourier transform applied to the annual sunspot data

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.

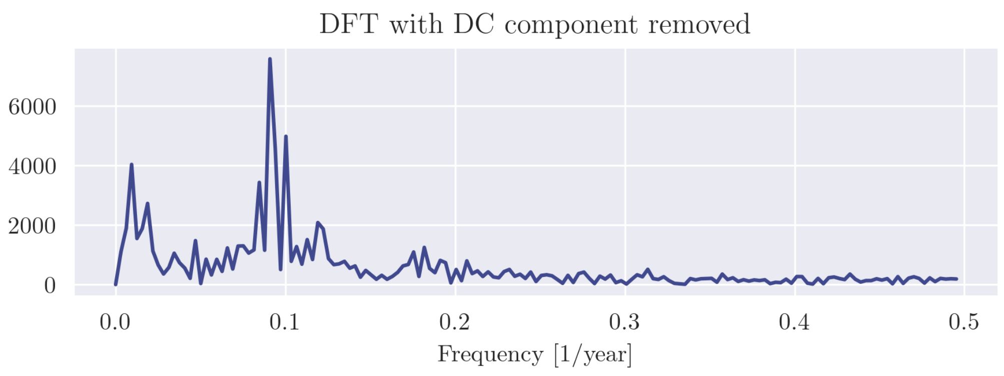


Figure 8.3: Fast Fourier transform applied after removing DC gain

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.

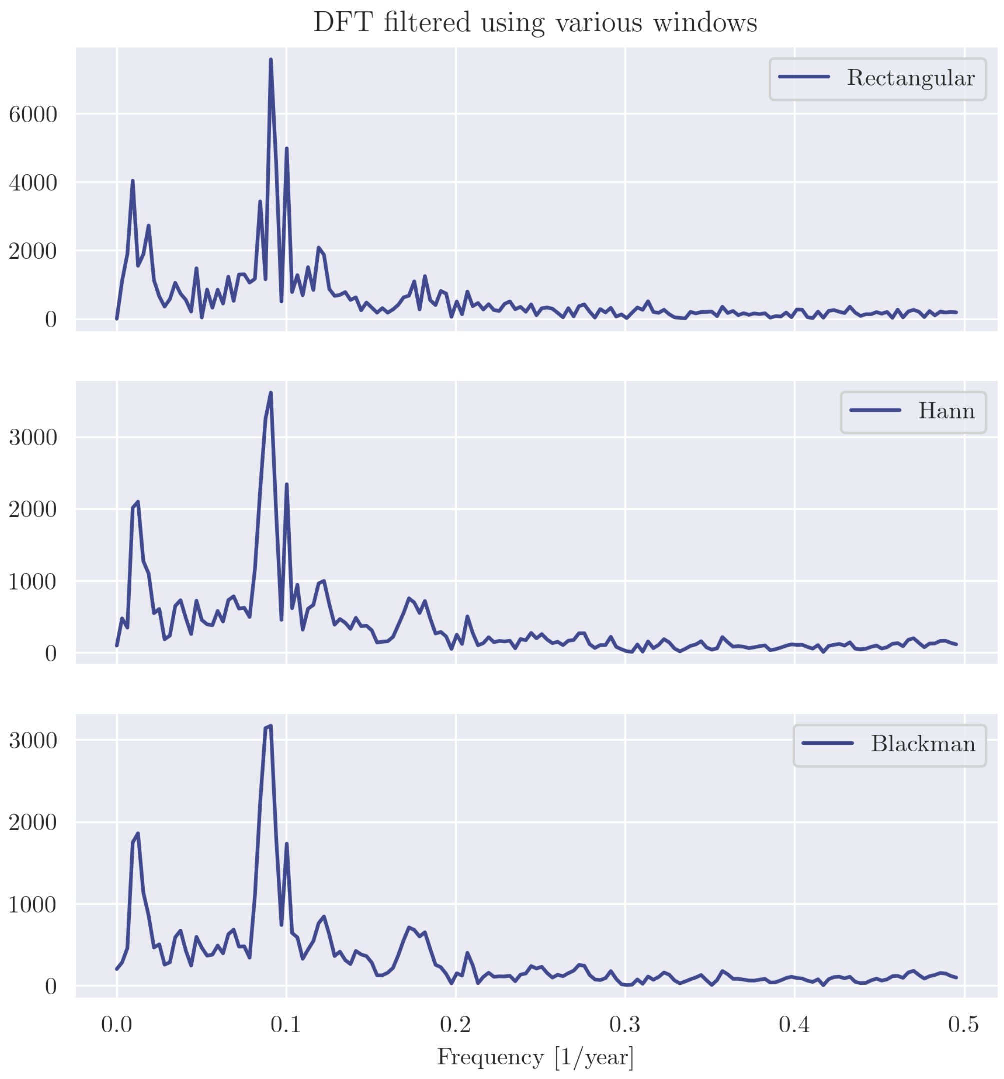


Figure 8.4: Effects of rectangular, Hann and Blackman windows

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.

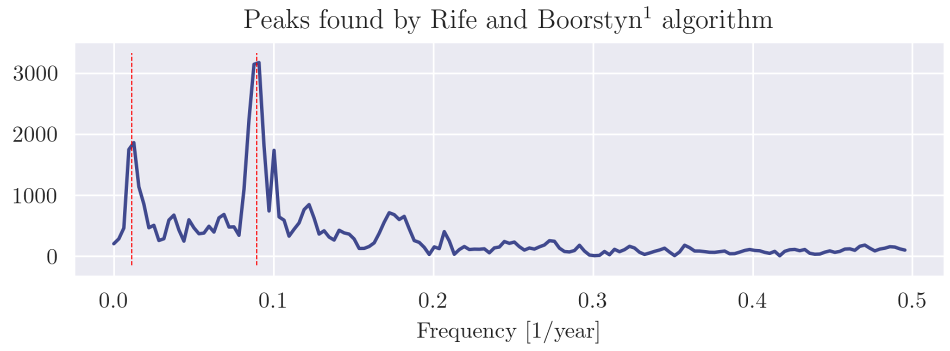


Figure 8.5: DFT of Blackman-windowed signal with identified frequencies in red

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.

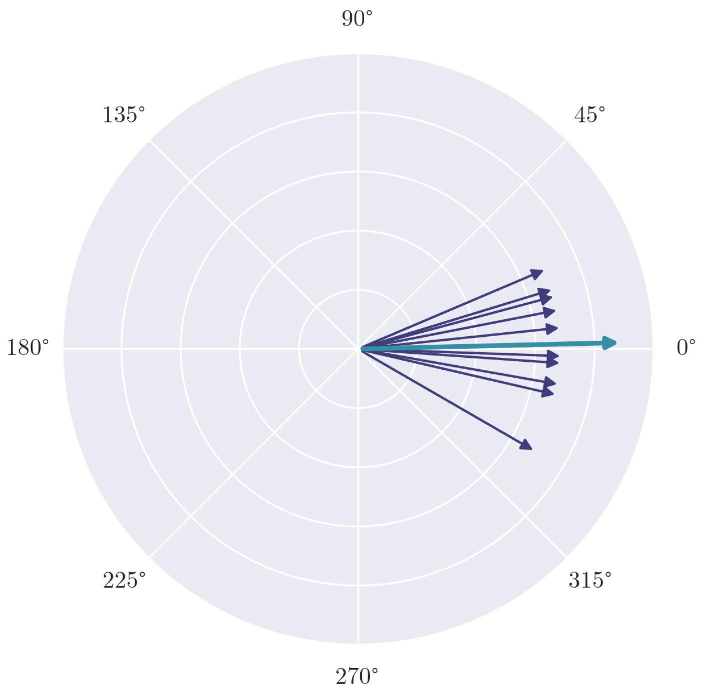


Figure 9.1: Homing pigeon departure headings and average heading

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

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

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:

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

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

Appendix

A Code Listings

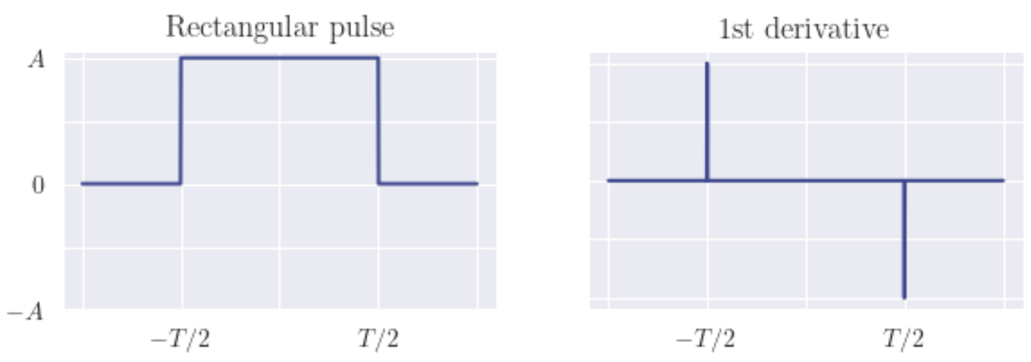
(Starts on the following page)

Question 1

This script visualises the rectangular and triangular pulse functions and their derivatives, and compares the rate at which the sidelobes of their discrete Fourier transforms fall off.

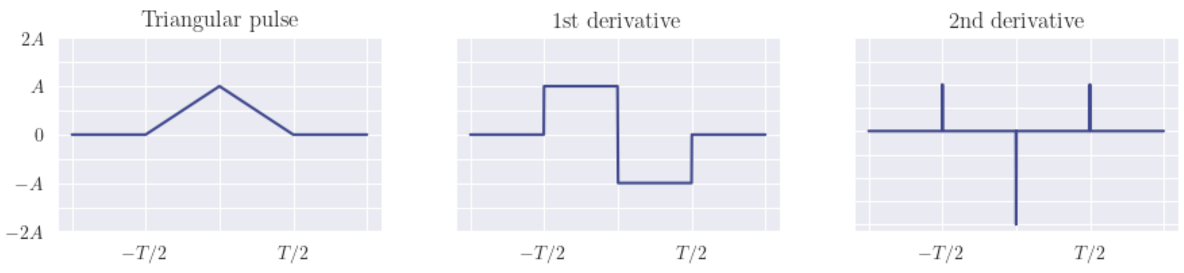
In [ ]:
```python
from pathlib import Path

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

from scipy.fft import fft, fftfreq

from config import A1_ROOT, SAVEFIG_CONFIG
```

In [ ]:
```python
# Plot rectangular pulse and first derivative
t = np.linspace(-1, 1, 1001)

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

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

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

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

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



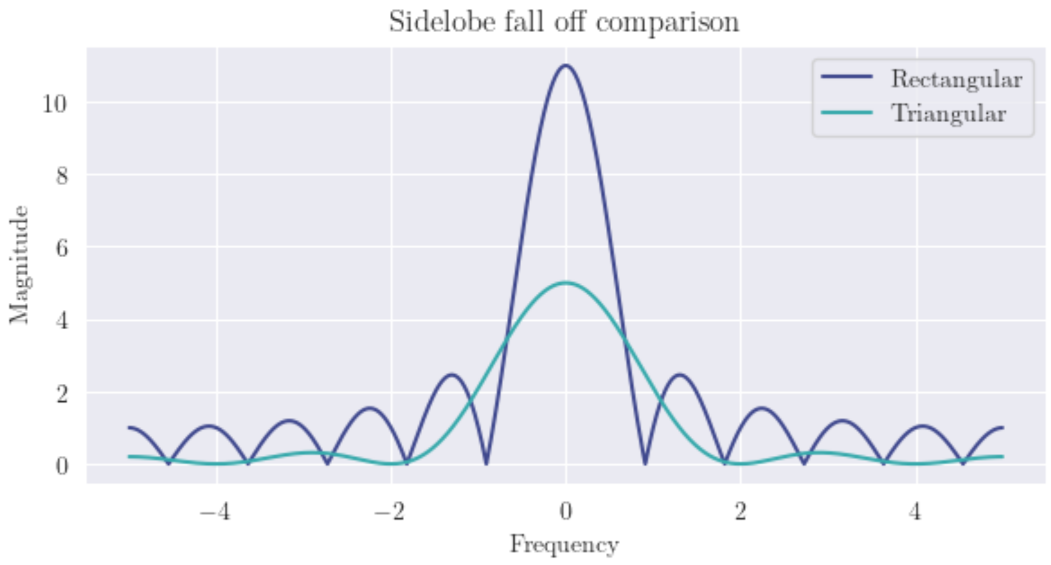
In [ ]:
```python
# Plot triangular pulse and first and second derivatives
t = np.linspace(-1, 1, 1001)

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

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

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

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

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

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



In [ ]:

```python
# Comparison of discrete Fourier transforms (using FFT)
t = np.linspace(-50, 50, 1001)
f = fftfreq(1001, 0.1)

# Create rectangular/triangular pulses of amplitude A and width T on t
rect_pulse = lambda t, A, T: A * (np.abs(t) <= T / 2)
tria_pulse = lambda t, A, T: A * (1 - 2 * np.abs(t) / T) * (np.abs(t) <= T / 2)

H_rect = np.abs(fft(rect_pulse(t, 1, 1)))
H_tria = np.abs(fft(tria_pulse(t, 1, 1)))

fig, ax = plt.subplots(figsize=(6, 3))
fig.tight_layout()

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

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

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


Sidelobe fall off comparison
Rectangular
Triangular
Magnitude
Frequency
0
2
4
6
8
10
−4
−2
0
2
4

Question 2 & 3

This script multiplies polynomials/integers in vector representation using convolution and the discrete Fourier transform.

In [ ]:
```
import numpy as np

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

Question 2

In [ ]:
```
# Define the polynomial coefficients vectors
x = np.array([1, 2, 6, 11, 15, 12], dtype=np.float64)
y = np.array([1, -3, -3, 7, -7, 3], dtype=np.float64)
```

In [ ]:
```
# Multiply by direct convolution
z_conv = convolve(x, y, mode="full", method="direct")

# Multiply by converting to Fourier domain
x_padded = np.pad(x, (0, len(y) - 1))
y_padded = np.pad(y, (0, len(x) - 1))
z_ffts = ifft(fft(x_padded) * fft(y_padded)).real

print("Q2 by convolution: ", z_conv)
print("Q2 by FFT and IFFT:", z_ffts)
```

```
Q2 by convolution:  [  1.  -1.  -3.  -6. -29. -35. -40.  10.  12. -39.  36.]
Q2 by FFT and IFFT: [  1.  -1.  -3.  -6. -29. -35. -40.  10.  12. -39.  36.]
```

Question 3

In [ ]:
```
# Define the integer multiplicands as vectors
x = np.array([8, 7, 5, 5, 7, 9, 0], dtype=np.float64)
y = np.array([1, 3, 6, 7, 2, 6, 7], dtype=np.float64)
```

In [ ]:
```
# Define the "carry" operation used for integer
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])
```

In [ ]:
```
# Multiply by direct convolution
z_conv = multiply_carry(convolve(x, y, mode="full", method="direct"))

# Multiply by converting to Fourier domain
x_padded = np.pad(x, (0, len(y) - 1))
y_padded = np.pad(y, (0, len(x) - 1))
z_ffts = multiply_carry(ifft(fft(x_padded) * fft(y_padded)).real)
```

```
print("Q3 by multiplication:", 8755790 * 1367267)
print("Q3 by convolution:    ", z_conv)
print("Q3 by FFT and IFFT:   ", z_conv)
```

```
Q3 by multiplication: 11971502725930
Q3 by convolution:    [1. 1. 9. 7. 1. 5. 0. 2. 7. 2. 5. 9. 3. 0.]
Q3 by FFT and IFFT:   [1. 1. 9. 7. 1. 5. 0. 2. 7. 2. 5. 9. 3. 0.]
```

Question 4

This script implements the double transform algorithm to apply the discrete Fourier transform to two real, N-point sequences using one complex N-point transform.

```
In [ ]: import numpy as np

        from scipy.fft import fft, ifft
```

Part A: Equal length sequences

```
In [ ]: # Define the vectors
        x = np.array([1, 2, 4, 4, 5, 3, 7, 8])
        y = np.array([1, 5, 3, 1, 3, 5, 3, 7])

        print("DFT of x:", fft(x))
        print("DFT of y:", fft(y))
```

```
DFT of x: [34.        -0.j          -1.87867966+6.53553391j -5.        +7.j
 -6.12132034+0.53553391j  0.        -0.j          -6.12132034-0.53553391j
 -5.        -7.j          -1.87867966-6.53553391j]
DFT of y: [28.        -0.j           2.24264069+4.24264069j -2.        -2.j
 -6.24264069+4.24264069j -8.        -0.j          -6.24264069-4.24264069j
 -2.        +2.j           2.24264069-4.24264069j]
```

```
In [ ]: # Combine x and y into a single complex vector and apply the FFT
        Z = fft(np.array([a+b*1j for a, b in zip(x, y)]))

        print("DFT of z:", Z)
```

```
DFT of z: [ 34.         +28.j           -6.12132034 +8.77817459j
  -3.          +5.j          -10.36396103 -5.70710678j
   0.          -8.j           -1.87867966 -6.77817459j
  -7.          -9.j            2.36396103 -4.29289322j]
```

```
In [ ]: # Extract the odd and even components of the real and imaginary parts of Z

        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)

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

```
ev(np.real(Z)) = array([34.        , -1.87867966, -5.        , -6.12132034,  0.
,
       -6.12132034, -5.        , -1.87867966])
od(np.imag(Z)) = array([ 0.        ,  6.53553391,  7.        ,  0.53553391,  0.
,
       -0.53553391, -7.        , -6.53553391])
od(np.real(Z)) = array([ 0.        , -4.24264069,  2.        , -4.24264069,  0.
,
        4.24264069, -2.        ,  4.24264069])
ev(np.imag(Z)) = array([28.        ,  2.24264069, -2.        , -6.24264069, -8.
,
       -6.24264069, -2.        ,  2.24264069])
```

In [ ]:
```
# Finally, reconstruct the DFTs of x and y
X = ev(np.real(Z)) + 1j * od(np.imag(Z))
Y = ev(np.imag(Z)) - 1j * od(np.real(Z))

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

```
X = array([34.        +0.j        , -1.87867966+6.53553391j,
       -5.        +7.j        , -6.12132034+0.53553391j,
        0.        +0.j        , -6.12132034-0.53553391j,
       -5.        -7.j        , -1.87867966-6.53553391j])
Y = array([28.        +0.j        ,  2.24264069+4.24264069j,
       -2.        -2.j        , -6.24264069+4.24264069j,
       -8.        +0.j        , -6.24264069-4.24264069j,
       -2.        +2.j        ,  2.24264069-4.24264069j])
```

In [ ]:
```
# Inverse Fourier transform X and Y to prove that they are correct
print(f"{ifft(X) = }")
print(f"{ifft(Y) = }")
```

```
ifft(X) = array([1.+0.j, 2.+0.j, 4.+0.j, 4.+0.j, 5.+0.j, 3.+0.j, 7.+0.j, 8.+0.j])
ifft(Y) = array([1.+0.j, 5.+0.j, 3.+0.j, 1.+0.j, 3.+0.j, 5.+0.j, 3.+0.j, 7.+0.j])
```

Part B: Unequal length sequences

In [ ]:
```
# Define the vectors
x = np.array([1, 2, 4, 4, 5, 3, 7, 8]) # this is the same x vector as part (a)
y = np.array([1, 5, 3, 1, 3, 5, 3, 0]) # this y vector is already zero-padded

print("DFT of x:", fft(x))
print("DFT of y:", fft(y))
```

```
DFT of x: [34.        -0.j         -1.87867966+6.53553391j -5.        +7.j
 -6.12132034+0.53553391j  0.        -0.j         -6.12132034-0.53553391j
 -5.        -7.j         -1.87867966-6.53553391j]
DFT of y: [21.        -0.j         -2.70710678-0.70710678j -2.        -9.j
 -1.29289322-0.70710678j -1.        -0.j         -1.29289322+0.70710678j
 -2.        +9.j         -2.70710678+0.70710678j]
```

In [ ]:
```
# Combine x and y into a single complex vector and apply the FFT
Z = fft(np.array([a+b*1j for a, b in zip(x, y)]))

print("DFT of z:", Z)
```

```
DFT of z: [ 34.        +21.j          -1.17157288 +3.82842712j
   4.         +5.j          -5.41421356 -0.75735931j
   0.         -1.j          -6.82842712 -1.82842712j
 -14.         -9.j          -2.58578644 -9.24264069j]
```

In [ ]:
```
# Extract the odd and even components of the real and imaginary parts of 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)) = }")
```

```
ev(np.real(Z)) = array([34.        , -1.87867966, -5.        , -6.12132034,  0.
,
       -6.12132034, -5.        , -1.87867966])
od(np.imag(Z)) = array([ 0.        ,  6.53553391,  7.        ,  0.53553391,  0.
,
       -0.53553391, -7.        , -6.53553391])
od(np.real(Z)) = array([ 0.        ,  0.70710678,  9.        ,  0.70710678,  0.
,
       -0.70710678, -9.        , -0.70710678])
ev(np.imag(Z)) = array([21.        , -2.70710678, -2.        , -1.29289322, -1.
,
       -1.29289322, -2.        , -2.70710678])
```

```
In [ ]: # Finally, reconstruct the DFTs of x and y
        X = ev(np.real(Z)) + 1j * od(np.imag(Z))
        Y = ev(np.imag(Z)) - 1j * od(np.real(Z))

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

```
X = array([34.        +0.j        , -1.87867966+6.53553391j,
       -5.        +7.j        , -6.12132034+0.53553391j,
        0.        +0.j        , -6.12132034-0.53553391j,
       -5.        -7.j        , -1.87867966-6.53553391j])
Y = array([21.        +0.j        , -2.70710678-0.70710678j,
       -2.        -9.j        , -1.29289322-0.70710678j,
       -1.        +0.j        , -1.29289322+0.70710678j,
       -2.        +9.j        , -2.70710678+0.70710678j])
```

```
In [ ]: # Inverse Fourier transform X and Y to prove that they are correct
        print(f"{ifft(X) = }")
        print(f"{ifft(Y) = }")
```

```
ifft(X) = array([1.+0.j, 2.+0.j, 4.+0.j, 4.+0.j, 5.+0.j, 3.+0.j, 7.+0.j, 8.+0.j])
ifft(Y) = array([ 1.00000000e+00+0.j,  5.00000000e+00+0.j,  3.00000000e+00+0.j,
        1.00000000e+00+0.j,  3.00000000e+00+0.j,  5.00000000e+00+0.j,
        3.00000000e+00+0.j, -1.11022302e-16+0.j])
```

Question 5

This script determines the roots of a particular polynomial and produces a pole-zero plot, then evaluates the magnitude of the polynomial around the unit circle using the DFT.

In [ ]:
```python
from pathlib import Path

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

from matplotlib.lines import Line2D
from matplotlib.patches import Circle

from numpy.polynomial.polynomial import Polynomial, polyval

from config import A1_ROOT, PLT_CONFIG, SAVEFIG_CONFIG
```

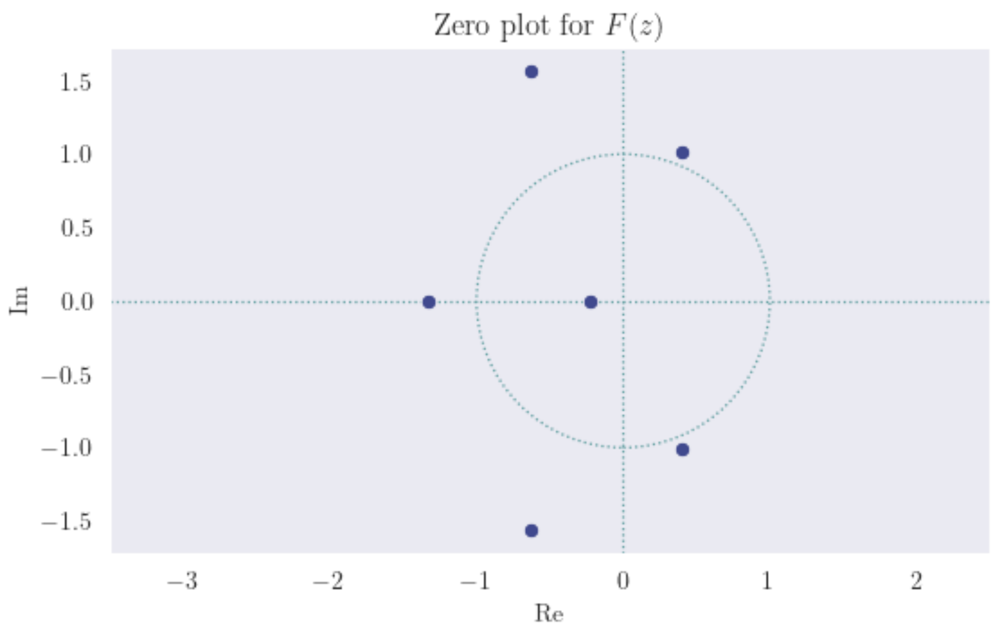
In [ ]:
```python
# Define the polynomial
poly = Polynomial([1, 5, 3, 4, 4, 2, 1])
```

Part A: Pole-zero plot

In [ ]:
```python
# Find the roots of the polynomial
poly_roots = poly.roots()
print("Polynomial roots:", poly_roots)
```

```
Polynomial roots: [-1.33198561+0.j         -0.62775824-1.564597j   -0.62775824+1.564
597j
 -0.22269308+0.j          0.40509758-1.01099437j  0.40509758+1.01099437j]
```

In [ ]:
```python
# Draw the pole-zero plot
sns.set_style("dark")           # override default style to hide grid
plt.rcParams.update(PLT_CONFIG) # re-set plot text customisation

fig, ax = plt.subplots(figsize=(6, 3.375))
fig.tight_layout()

# Plot the zeros
sns.scatterplot(x=np.real(poly_roots), y=np.imag(poly_roots), ax=ax, marker="o")
ax.set_xlim(-3.5, 2.5)

# Draw the unit circle as an underlay
underlay_style = {"ls": "dotted", "lw": 0.9, "color": "cadetblue", "zorder": 0}
unit_circle = Circle(xy=(0, 0), radius=1, fill=False, **underlay_style)
ax.add_patch(unit_circle)

# Draw the x and y axes as another underlay
x_axis = Line2D(xdata=ax.get_xlim(), ydata=(0, 0), **underlay_style)
y_axis = Line2D(xdata=(0, 0), ydata=ax.get_ylim(), **underlay_style)
ax.add_line(x_axis)
ax.add_line(y_axis)

ax.set_aspect("equal")

ax.set_title("Zero plot for $F(z)$")
ax.set_xlabel("Re")
ax.set_ylabel("Im")

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

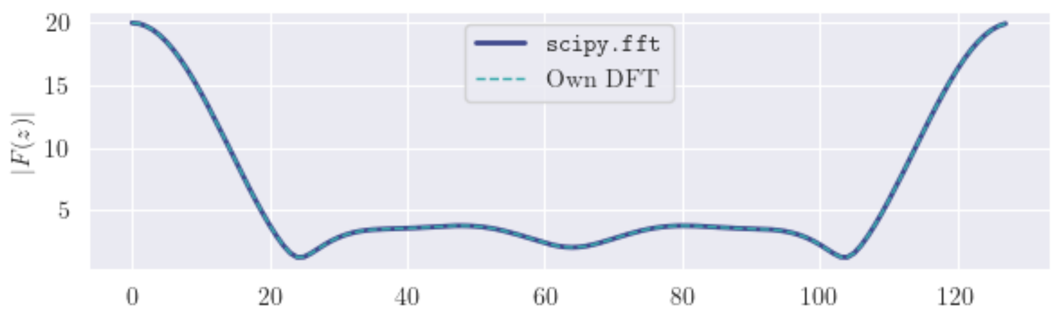


Part B: DFT implementation

In [ ]:
```python
# Define an "own" DFT implementation to compare with scipy
def zdft(poly_coef: np.array, n: int) -> np.array:
    """
    Computes the 1D `n`-point discrete Fourier transform of some sequence from
    its Z transform, given by `poly_coef`.
    """
    return np.array(
        [polyval(np.exp(-1j*2*np.pi*k/n), poly_coef) for k in range(n)])
```

In [ ]:
```python
from scipy.fft import fft

# Evaluate the polynomial coefficients using scipy's and own DFT implementations
y_fft = np.abs(fft(poly.coef, n=128))
y_dft = np.abs(zdft(poly.coef, n=128))
```

In [ ]:
```python
from config import SNS_STYLE

sns.set_style(SNS_STYLE)        # re-set the default style, changed by part (a)
plt.rcParams.update(PLT_CONFIG) # re-set the plot text customisation

# Plot the results of the two implementations together
fig, ax = plt.subplots(figsize=(6, 2))
fig.tight_layout()

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

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

ax.legend(loc="upper center")

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


scipy.fft
Own DFT
$|F(z)|$
20
15
10
5
0
20
40
60
80
100
120

Question 6 & 7

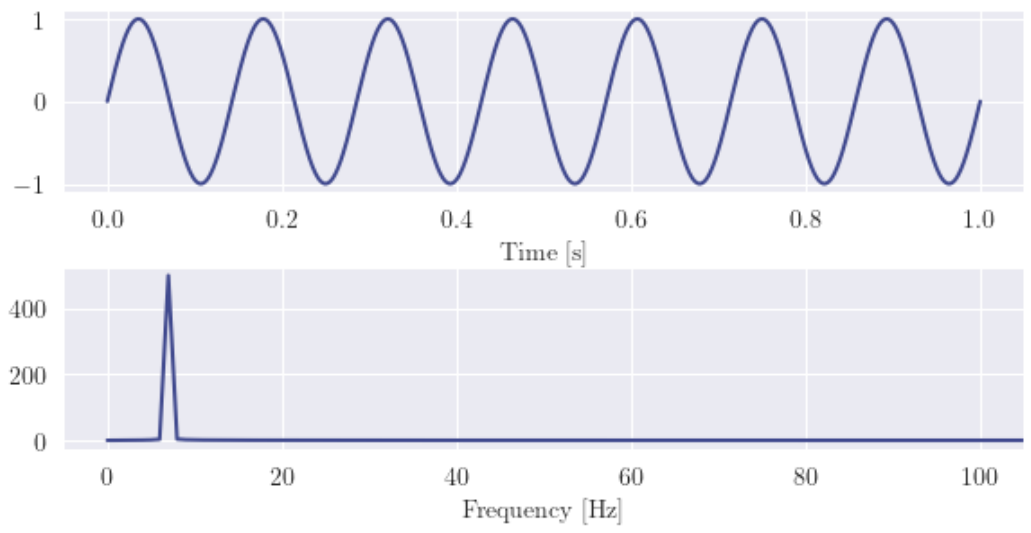
This script demonstrates two methods of interpolating a sine wave sampled above the Nyquist frequency: sinc interpolation and zero-padding in the Fourier domain.

In [ ]:
```python
from pathlib import Path

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

from scipy.fft import fft, fftfreq, ifft

from config import A1_ROOT, SAVEFIG_CONFIG
```

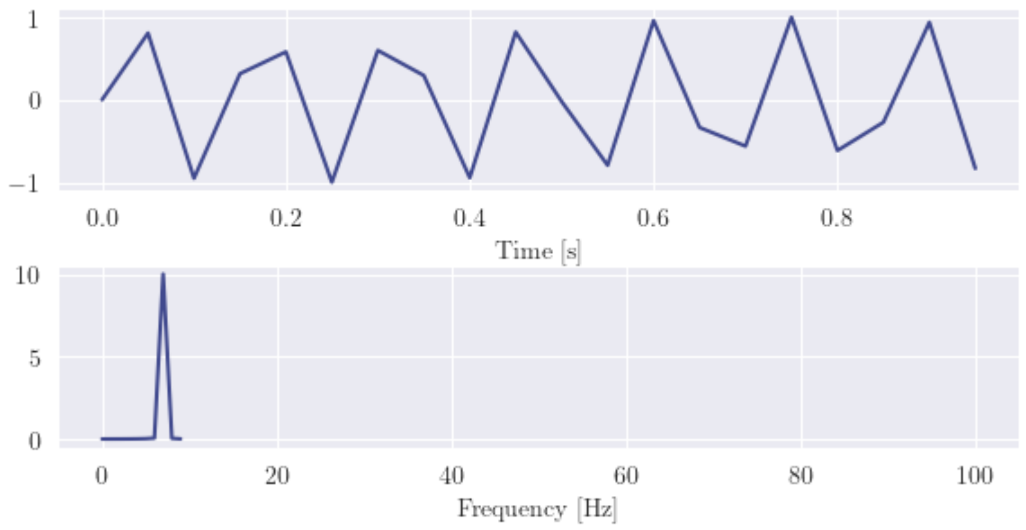
In [ ]:
```python
# Create a "continuous" 7 Hz sine wave (actually 1 kHz)
t = np.linspace(0, 1, 1000)
x = np.sin(2 * np.pi * 7 * t)
```

In [ ]:
```python
from matplotlib.axes import Axes
from matplotlib.figure import Figure

# Define a utility function which we will use on a number of occasions to
# visualise both the time and frequency domain
def time_fourier_plot(t: np.array, x: np.array) -> tuple[Figure, list[Axes]]:
    """
    Plot the given signal and its discrete Fourier transform.
    """
    f = fftfreq(n=len(t), d=(t[1]-t[0]))[:len(t)//2]
    H = np.abs(fft(x))[:len(t)//2]

    fig, axs = plt.subplots(2, figsize=(6, 3))
    fig.tight_layout()

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

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

    return fig, axs
```

In [ ]:
```python
# Visualise the "continuous" 7 Hz sine wave
fig, axs = time_fourier_plot(t, x)
axs[1].set_xlim(-5, 105)

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



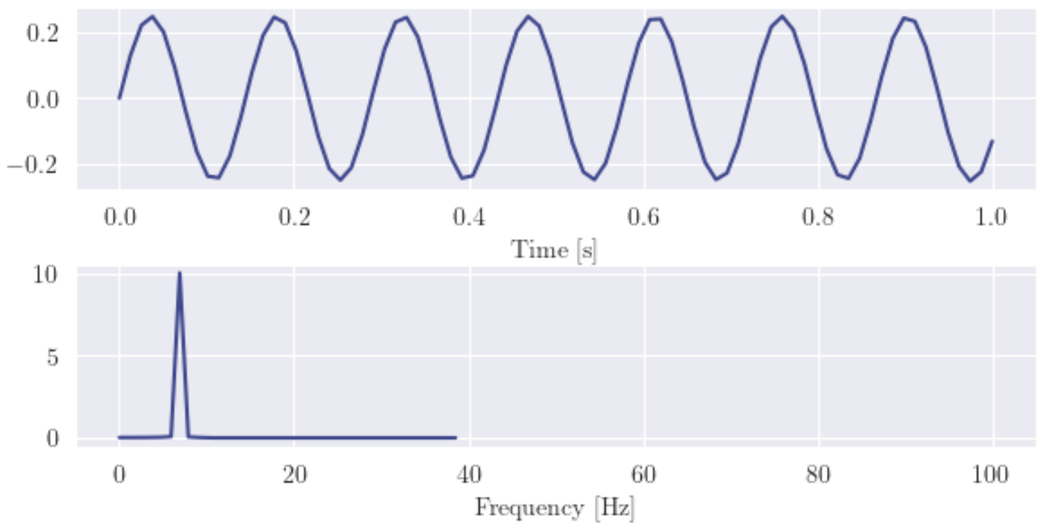
```
In [ ]: # Sample the "continuous" 7 Hz sine wave at 20 Hz (above Nyquist freq. of 14 Hz)
        t_samp = t[::1000//20]
        x_samp = x[::1000//20]

        # Visualise the sampled signal (it looks quite terrible)
        fig, axs = time_fourier_plot(t_samp, x_samp)
        axs[1].set_xlim(-5, 105)

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



Question 6: Sinc interpolation

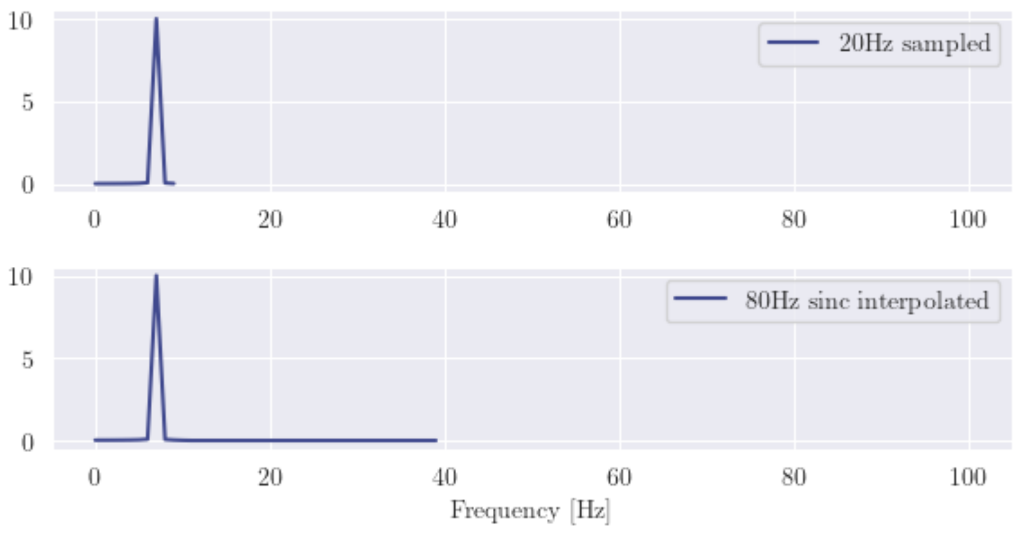
```
In [ ]: # Define a function to perform sinc interpolation on an arbitrary signal
        def sinc_interpolate(x: np.array, n: int, viz: bool = False) -> np.array:
            """
            Upsamples the given signal by the specified factor using sinc interpolation.
            """
            #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

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

            return x_interp
```

In [ ]:
```
# Perform sinc interpolation on the sampled signal
t_interp = np.linspace(0, 1, 80)
x_sinc_interp = sinc_interpolate(x_samp, 4, viz=True)

# Visualise the sinc interpolated signal (it looks much better)
fig, axs = time_fourier_plot(t_interp, x_sinc_interp)
axs[1].set_xlim(-5, 105)

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



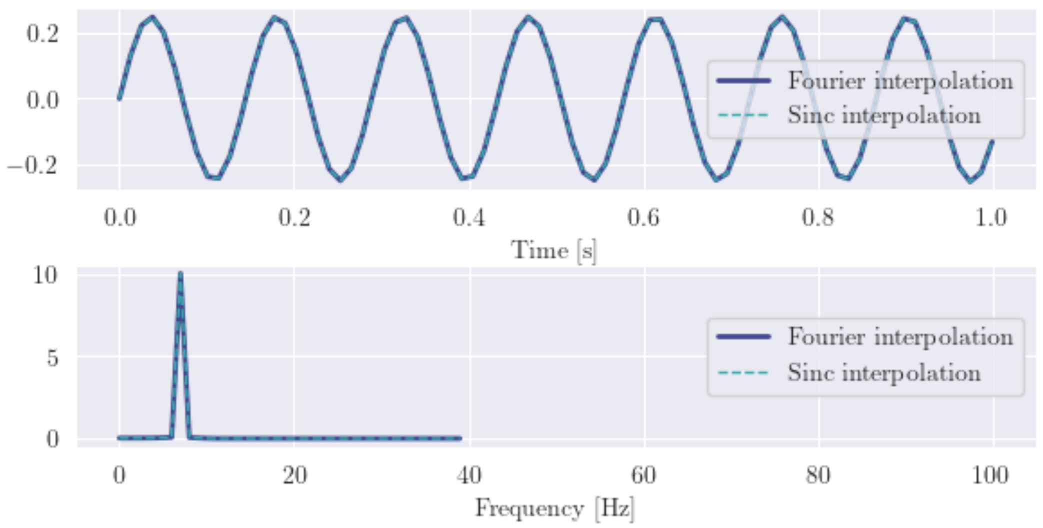
Question 7: Zero-padding in the Fourier domain

In [ ]:
```
# Compare the sampled and sinc interpolated signals in the Fourier domain
fig, axs = plt.subplots(2, figsize=(6, 3))
fig.tight_layout()

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

# Interpolated signal
N_interp = len(x_sinc_interp)
f_interp = fftfreq(n=80, d=1/80)[:N_interp//2]
H_sinc_interp = np.abs(fft(x_sinc_interp))[:N_interp//2]

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

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

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

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



In [ ]:
```
# Define a function to perform interpolation on an arbitrary signal by
# Fourier domain zero-padding
def fourier_interpolate(x: np.array, n: int) -> np.array:
    """
    Upsamples the given signal by the specified factor by applying the Fourier
    transform, zero-padding, then inverse Fourier transforming.
    """
    H = fft(x); N = len(H)

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

    return x_interp
```

In [ ]:
```
# Perform Fourier domain zero-padding to interpolate the sampled signal
x_ffts_interp = fourier_interpolate(x_samp, 4)
H_ffts_interp = np.abs(fft(x_ffts_interp))[:N_interp//2]

# Visualise the interpolated signal alongside the previous sinc interpolation
fig, axs = plt.subplots(2, figsize=(6, 3))
fig.tight_layout()

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

sns.lineplot(x=f_interp, y=H_ffts_interp, ax=axs[1], ls="-",  lw=2,
    label="Fourier interpolation")
sns.lineplot(x=f_interp, y=H_sinc_interp, ax=axs[1], ls="--", lw=1,
    label="Sinc interpolation")

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

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

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

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


0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
Time [s]
Fourier interpolation
Sinc interpolation
10
5
0
0
20
40
60
80
100
Frequency [Hz]
Fourier interpolation
Sinc interpolation

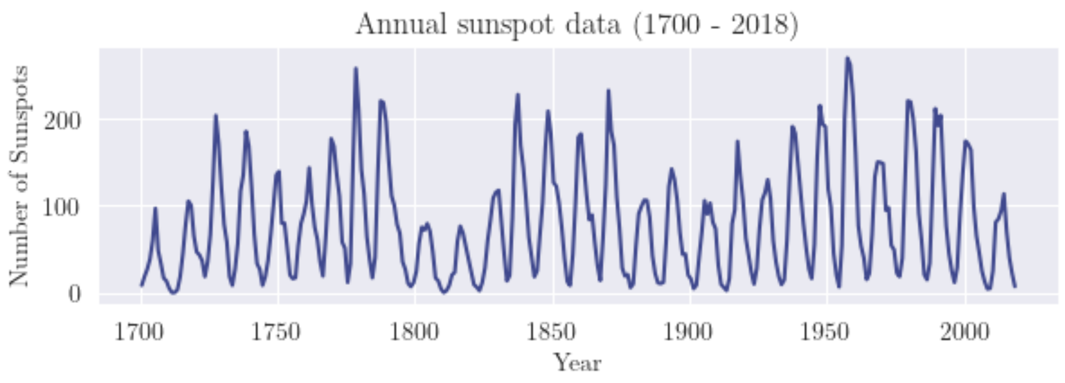
Question 8

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

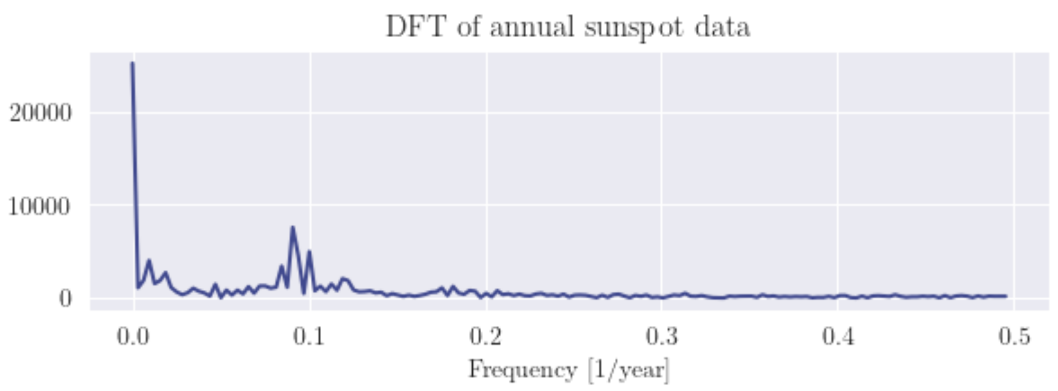
```
In [ ]: from pathlib import Path

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

        from scipy.fft import fft, fftfreq

        from config import A1_ROOT, SAVEFIG_CONFIG
```

```
In [ ]: data = pd.read_csv(Path(A1_ROOT, "data", "SunspotData.csv"))
```

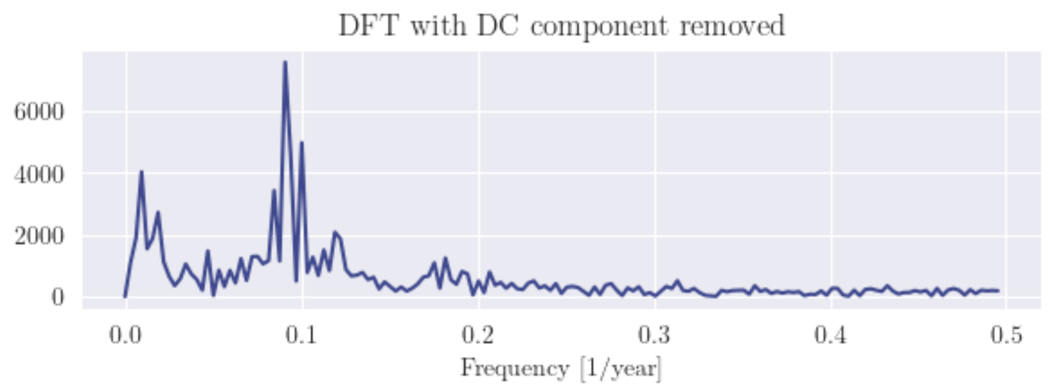
```
In [ ]: # Plot the time series data
        fig, ax = plt.subplots(figsize=(6, 2))
        fig.tight_layout()

        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)
```



```
In [ ]: # 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]")

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

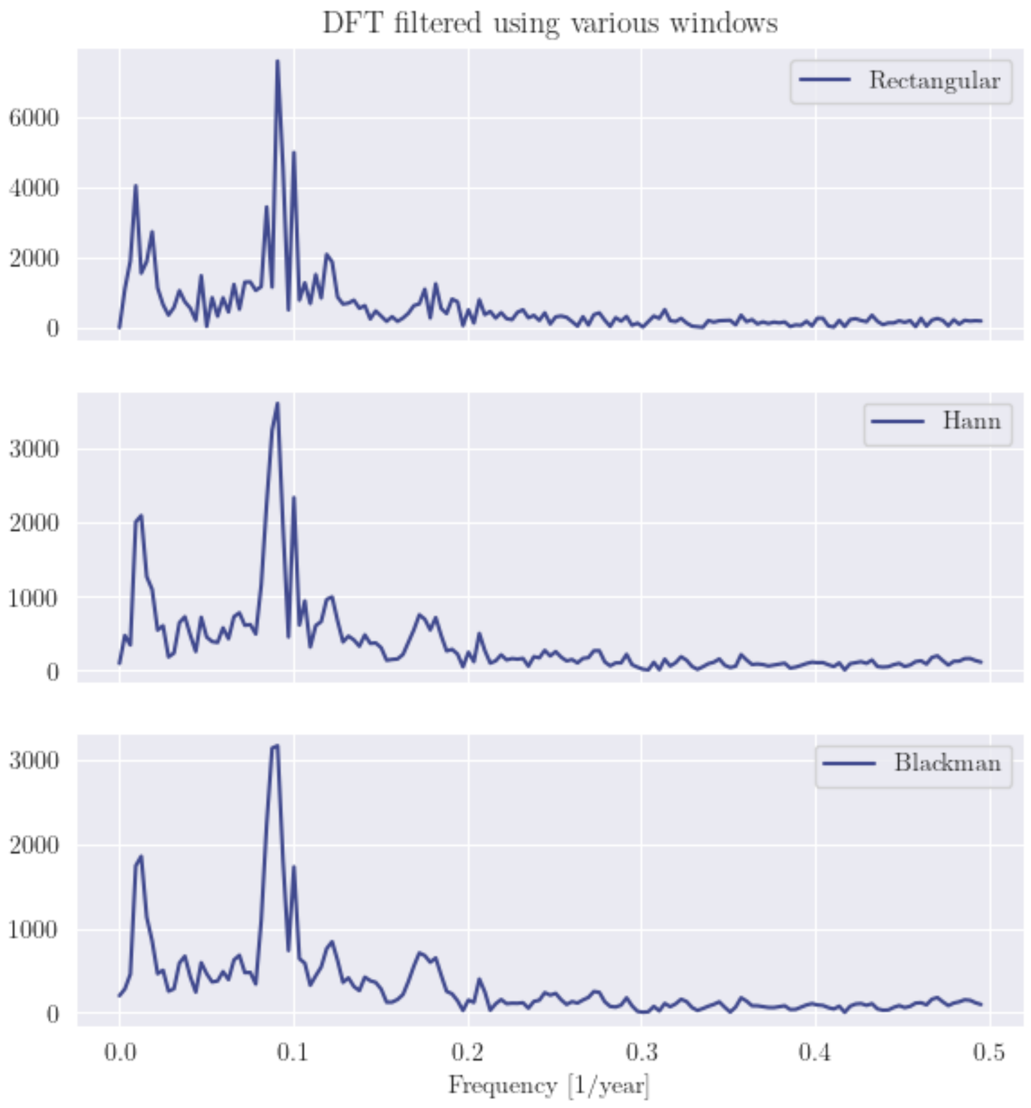


In [ ]:
```python
# Remove the DC component and plot the DFT again
x = (data["Number of Sunspots"] - data["Number of Sunspots"].mean()).to_numpy()
H = np.abs(fft(x)[:N//2])

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)
```



In [ ]:
```python
# Try filtering the signal using each of: blackman, boxcar, and hann windows
from scipy.signal.windows import blackman, boxcar, hann

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 [ ]: from typing import Tuple

        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 [1].
            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])
            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]")
```

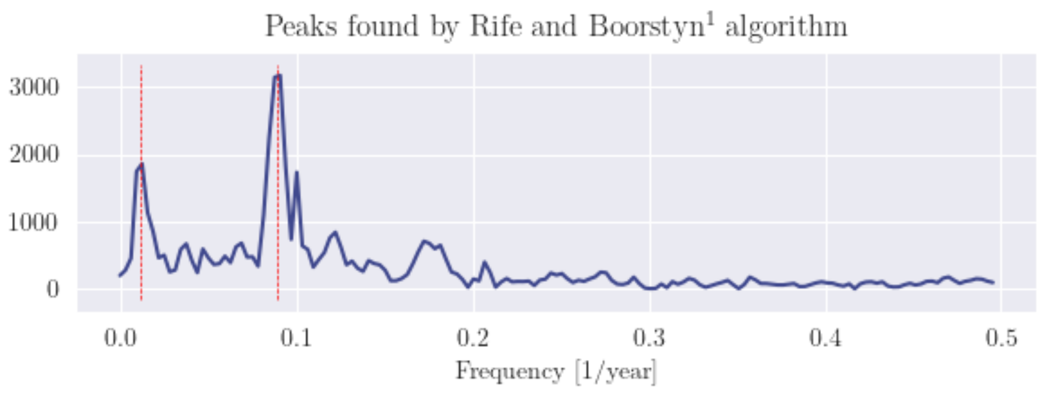
```
Coarse search: [0.0109717868338558, 0.08934169278996865] [1/years]
               [91.14285714285714, 11.192982456140351] [years]
```

In [ ]:
```python
from typing import Callable

def root_secant(f: Callable, x0: float, x1: float, maxiter: int = 100) -> float:
    """
    Performs the secant method root finding algorithm on the function `f`(x),
    with initial guesses `x0` and `x1`. The initial guesses should be close to
    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 value.
        """
        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)) / 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, 0.089473842225381592] [1/years]
             [89.6635672052842, 11.176450846530528] [years]
```

In [ ]:
```python
# Display the Blackman-windowed signal with identified frequencies overlayed
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="r",
    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)
```



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).

Question 9

This script plots a number of headings on a polar plot and calculates the average heading.

```
In [ ]: from pathlib import Path

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

        from config import SNS_PALETTE
```

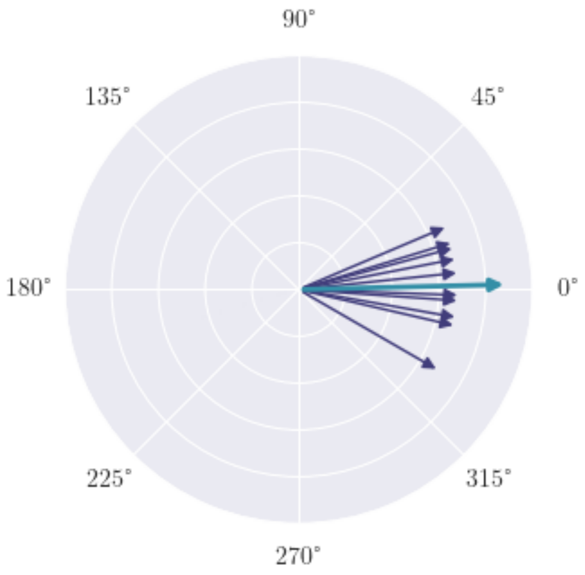
```
In [ ]: # Define the set of headings (in degrees)
        headings = [11, 15, 350, 330, 23, 347, 17, 356, 6, 358]
```

```
In [ ]: # Define a function for calculating average heading as complex numbers
        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)))))
```

```
In [ ]: from matplotlib.axes import Axes

        # Define a utility function for drawing arrows on a polar plot
        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)
```

```
In [ ]: from config import A1_ROOT, SAVEFIG_CONFIG

        # Visualise the headings and their average on a polar plot
        fig = plt.figure(figsize=(6, 3.375))
        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)
```

```
Average heading: 1.39 [deg]
```


90°
135°
45°
180°
0°
225°
315°
270°