Applied signal processing laboratory
(A.Y. 2024/25)

Assignment 1

Fundamentals of spectral analysis

Authors:
Castorina Giovanni (307594)
Fanton Vittoria (308625)
Giordano Andrea (310679)

Contents

Introduction

This report covers the fundamentals of spectral analysis, with key concepts and practical applications.
It begins with an examination of convolution, both linear and circular, followed by an analysis of the Discrete Fourier Transform (DFT) and the Discrete-Time Fourier Transform (DTFT), including the effects of zero padding.
Next, the report discusses the analysis of time- and band- unlimited analog signals, focusing on the role of sampling and the aliasing effect that arises from it. MATLAB's symbolic expressions are also used to verify manually derived results.
The third exercise deals with band-limited signals and further explores the impact of zero padding on the DFT. On the other hand, the fourth exercise delves into delay and frequency estimation, implementing a custom function, `my_xcorr()`, to determine a randomly generated delay and frequency, even in the presence of Gaussian noise at different levels of variance.
Finally, the report examines spectral estimation techniques, comparing different periodograms: Welch, Bartlett, and simple.

Overall, these exercises provide a hands-on approach to understand fundamental aspects of spectral analysis, improving the knowledge of theoretical concepts through practical signal processing applications.

1 Convolution custom function and effect of zero padding on DTFT

The first objective of the exercise is to compute the linear and circular convolution of two sequences creating a custom function called `my_conv()`.

Specifically, the input arguments of the function include the two signals, with an optional specification for the type of convolution: 'c' for circular convolution and 'l' for linear convolution. If no specification is provided, the function, by default, computes the linear convolution. To manage different numbers of input arguments, the function uses `varargin` to allow a variable number of inputs and, inside the function, `nargin` is used to check the actual number of inputs and determine the appropriate convolution type to compute.

Analyzing the actual computation, the linear convolution between the two sequences is defined as:

$$q[n] = x[n] * y[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot y[n-k] \tag{1}$$

According to the definition, the support of the result, in addition, is given by the sum of the two supports minus 1.
Therefore, to perform linear convolution, according to Eq.1, in the MATLAB code, the two sequences are zero-padded to match the length of the convolution result. The second sequence is then flipped, and a `for loop` is used to shift the flipped sequence by one position at each iteration. The scalar product between the shifted sequences is computed at each step and stored in the corresponding position of the output.

On the other hand, for the circular convolution, the length of the result is equal to the maximum length (M) of the two sequences. Therefore, initially, the shorter sequence is zero-padded to match the length of the longer one. Then, a square matrix M_{circ} of size $M \times M$ is created, where each column contains a circularly shifted version of the shorter sequence (using `circshift()`), with shifts ranging from 0 to the length of the longer sequence. Finally, the convolution result is obtained by performing a matrix-vector product between the circular matrix and the longer sequence.

In order to check the functioning of the custom `my_conv()`, both with linear and circular convolution, two random sequences are created with the function `randi()`, in order to have a sequence of random integer numbers in a certain range.
Specifically, just as an example, the first sequence x[n] is made of random numbers between 0 and 50 and has length 20, while the second one y[n] is between 0 and 30 and has length 30. The results of the linear convolution and the circular one are shown, respectively, in Figure 1 (a) and 1 (b). Note that the results obtained with `my_conv()` are superimposed on the ones got with the built-in functions `cconv()` for circular convolution and `conv()` for linear one.

Analyzing Figure 1, it is clear that the results obtained for the two types

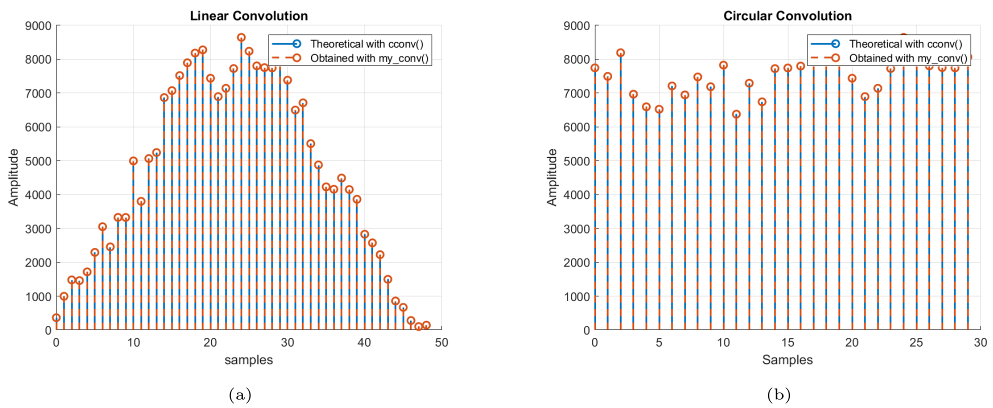


Figure 1: Results of the linear and circular convolution of $x[n]$ and $y[n]$.

of convolution are really different, but they both perfectly coincide with the outputs of the respective built-in functions of MATLAB showing the validity of the custom function.

The following objective is to create a triangular sequence x(n) of N = 45 samples, as the result of the linear convolution (using `my_conv()`) of two rectangle sequences of length M = $\frac{N+1}{2}$ and amplitude $\frac{1}{\sqrt{M}}$. The expected output is a triangular sequence with peak 1 (since it is given by $M \times \frac{1}{\sqrt{M}} \times \frac{1}{\sqrt{M}}$) at sample $\frac{N-1}{2}$ and length equal to $N = M \times 2 - 1$. The result is presented in Figure 2 and it shows perfect compatibility with what was expected both according to the peak and length.

Furthermore, using the functions `fft()` and `fftshift()` it is possible to obtain the spectrum of the triangular signal, as shown in Figure 3.

Using the generated triangular sequence, the final part of the exercise aims

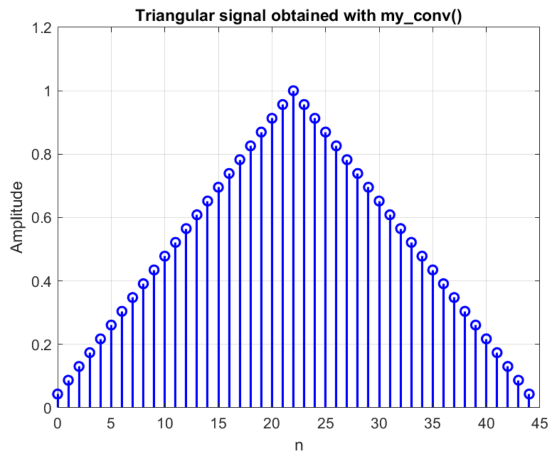


Figure 2: Triangular sequence obtained as the linear convolution of two rectangular sequences using *my_conv*()

to analyze the effect of zero-padding on the DTFT.

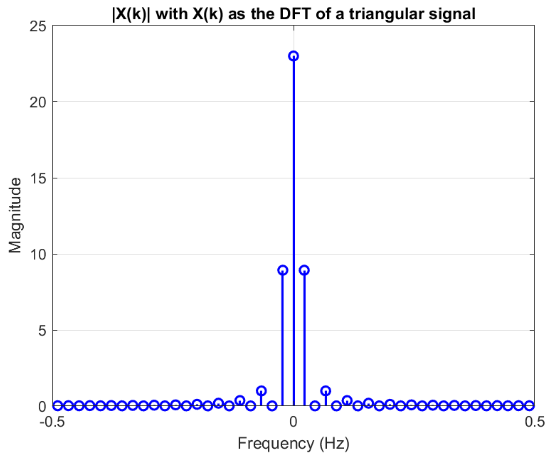
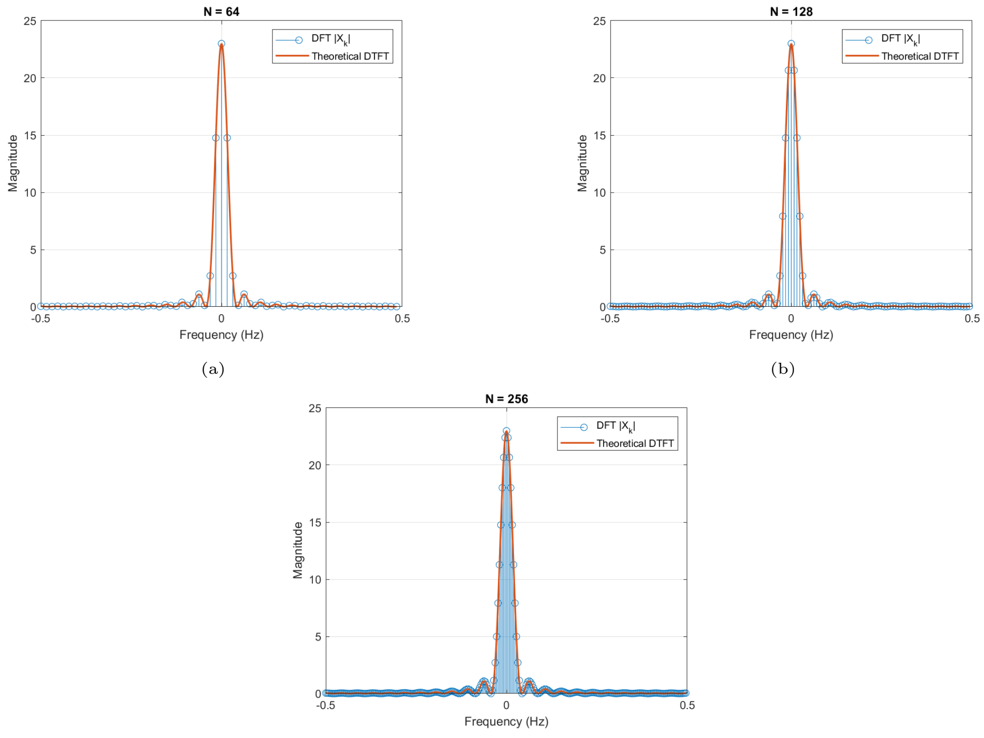


Figure 3: DFT of the triangular sequence obtained as the linear convolution of two rectangular sequences

The expected DTFT of the convolution between two rectangular functions of length M is given by:

$$X(e^{j2\pi f}) = \frac{1}{M}\text{DTFT}\left\{\text{rect}_M(n) * \text{rect}_M(n)\right\} = \frac{\sin^2(\pi f M)}{M\sin^2(\pi f)}$$

This result is then compared with the `fft()` of the convolution between the two rectangular signals to verify the correctness of the method, as illustrated in Figure 4 and the DFT is computed again. Furthermore, to better observe the effect of zero-padding on the spectrum, the convolution result is extended to $N = 64, 128, 256$.



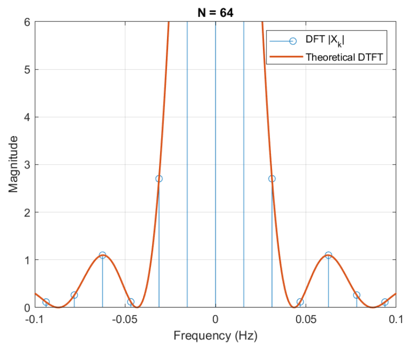
(a)

(b)

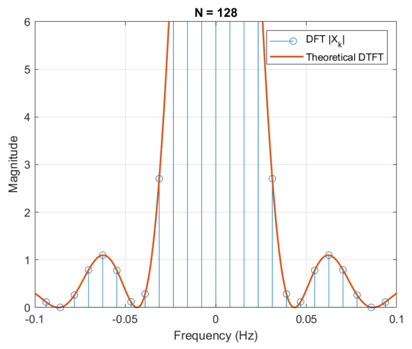
(c)

Figure 4: Comparison between DFT and Theoretical DTFT for different values of N.

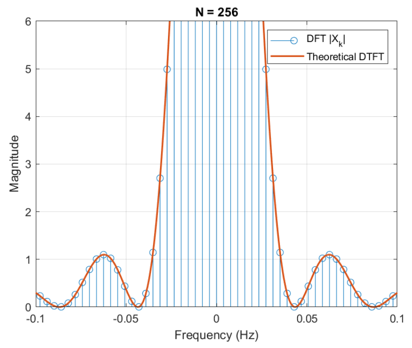
Knowing that zero-padding increases the number of samples (N) without altering the original signal, as the appended values are all zeros, and analyzing the results in Figure 4, it is possible to perform some considerations. Firstly, it can be observed that the shape of the spectrum remains unchanged, while



(a)



(b)



(c)

Figure 5: Comparison between the DFT and the theoretical DTFT, with axes limited to $x \in [-0.1, 0.1]$ and $y \in [0, 6]$, for different N.

the observation time T_0, given by $T_0 = NT_c$, increases as N increases. Extending the observation time while keeping the sampling frequency constant enhances the frequency resolution. As a result, the spectrum maintains its original shape but it is represented with greater accuracy.
This effect is particularly evident in Figure 5, where for $N = 64$, a discrete number of stem points is present, perfectly matching the theoretical spectrum expected from the convolution of two rectangular functions of length M and, as N increases, the number of points grows, progressively approximating the expected $X(e^{j2\pi f})$ in the limit as $N \to \infty$.

2 Analog time- and band-unlimited signal

For the second exercise, the signal to be considered is the following:

$$x(t) = 32t \cdot e^{-8t} H(t)$$

Firstly, it is defined in MATLAB as a symbolic expression using `syms` for the variable t, assuming t real, and `heaviside()` for H(t) $= \begin{cases} 0 & \text{for } t < 0 \\ 1 & \text{for } t \geq 0 \end{cases}$.
Considering that x(t) is different from 0 for $t \geq 0$ due to Heaviside, the energy of x(t) can be computed manually following the definition of the energy of a continuous signal:

$$E = \int_0^\infty \left|32t \cdot e^{-8t}\right|^2 dt = 1024 \int_0^\infty t^2 \cdot e^{-16t}\, dt$$

Integrating by parts:

$$\begin{aligned} E &= 1024 \cdot \left[-\frac{t^2}{16} \cdot e^{-16t} + \int \frac{2t}{16} \cdot e^{-16t}\, dt \right]_0^\infty = \\ &= 1024 \cdot \left[-\frac{t^2}{16} \cdot e^{-16t} + \frac{1}{8} \cdot \left(-\frac{t}{16} \cdot e^{-16t} - \int \frac{1}{16} \cdot e^{-16t}\, dt \right) \right]_0^\infty = \\ &= 1024 \cdot \left[-\frac{t^2}{16} \cdot e^{-16t} + \frac{1}{8} \cdot \left(-\frac{t}{16} \cdot e^{-16t} + \frac{1}{16^2} \cdot e^{-16t} \right) \right]_0^\infty = 1024 \cdot \left[\frac{1}{8} \cdot \frac{1}{16^2} \right] = \frac{1}{2} \end{aligned}$$

Checking this energy value with the MATLAB result given by the function `int()`, it is discovered that the result obtained is exactly the same.

The next task is to find the CTFT X(f) and to check it again with the result obtained in MATLAB using `fourier()`, with parameters [1, -2*pi]. Using the Table of Fourier Transform, it is clear that the function is part of the group in the form $ate^{-at}u(t) \rightarrow \frac{a}{(a+j2\pi f)^2}$ so for our specific case, considering a = 8:

$$CTFT\{32t \cdot e^{-8t}u(t)\} = CTFT\{4\left(8t \cdot e^{-8t}u(t)\right)\} = \frac{4 \cdot 8}{(8 + j2\pi f)^2} = \frac{32}{(8 + j2\pi f)^2}$$

This is the exact same result obtained with the MATLAB function `fourier()`, therefore the result obtained manually is perfectly checked.

After knowing the energy, it is possible to continue estimating the signal bandwidth which contains 99.9% of the energy, defining a new symbolic variable for the bandwidth, called B_x, such that:

$$\int_{-B_x}^{B_x} |X(f)|^2 \, df = 0.999 \int_0^{\infty} |x(t)|^2 \, dt$$

This passage is useful because, since the signal is band unlimited, estimating the bandwidth in this way, it is possible to contain aliasing in frequency, without considering an infinite bandwidth. In order to get the result of the integral in MATLAB, using `int()` to find the symbolic version of the integral and solving with `vpasolve()` for B_x, the result obtained is the following:

$$B_x = 9.5005 \; Hz$$

The sampling frequency f_c is then set to $6 \cdot B_x$, in order to choose a frequnecy which satisfied the Nyquist Theorem.
In order to continue with the exercise, it is fundamental to mention the relation between the CTFT of the sampled signal over N samples ($X_C(f_a)$) and the DFT of the same signal. The relationship, given $T_0 = NT_c$, with T_c sampling time, is:

$$X(\frac{k}{N} f_c) = T_c \cdot DFT[x[n]] = \frac{T_0}{N} \cdot DFT[x[n]]$$

In general, the observation time interval T_0 is defined as the time beyond which the signal $x(t)$ can be considered negligibly small. To determine a precise value, $x(t)$ is assumed to be close to zero when it satisfies the condition $x(t) < 10^{-6}$. The corresponding time is then computed by solving this inequality, considering only the positive solution as physically meaningful. This is achieved using the MATLAB functions `solve()` and `eval()`. Therefore, T_0 is found as:

$$T_0 = 3.262 \; s$$

Then, N is found accordingly as $N = T_0 \cdot f_s = T_0 / t_s$ and it is rounded up to the nearest power of 2 with `nextpow2()`, due to computational complexity reasons:

$$N = 185.95 \quad \text{rounded to} \quad N = 256$$

After having obtained the updated value of N, T_0 is recomputed:

$$T_0 = N/f_c = N \cdot t_c = 4.491 \; s$$

The new Δf is obtained:

$$\Delta f = \frac{1}{T_0} = 0.223\ Hz$$

Afterwards, the discrete-time signal $x[n]$ of $x(t)$ with N samples is obtained by defining a vector n ranging from 0 to $N-1$. The comparison between $x[n]$ and $x(t)$, the latter converted into a MATLAB function using the `matlabFunction()` command and evaluated at $M = 100N$ points, is shown in Figure 6.
The Δf, considering M points, is the following:

$$\Delta f = \frac{f_c}{M} = 0.002\ Hz$$

It is clear that, increasing the number of points (from N to M), the Δf decreases.
Analyzing the figures, it is evident that the two graphs in Figure 6 perfectly

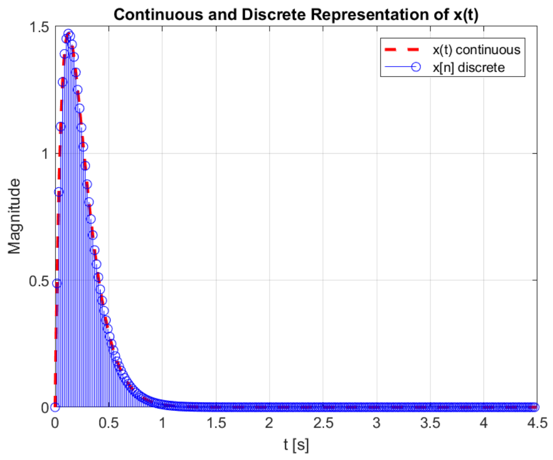


Figure 6: Comparison between the continuous and discrete (x[n]) Representation of x(t)

coincide, demonstrating that sampling in the time domain is simply a process of selecting discrete values from the original continuous signal. This confirms that the sampled signal in time perfectly represents the original function in a discrete set of values.
Then it is possible to do the same comparison between the DFT of x[n] and X(f), given by evaluating the result of the symbolic `fourier()` function. The

results of the comparison are shown in Figure 7. Note that Figure 7(b) is a zoomed version of the result useful in order to allow further analysis on the difference between the two results.

Analyzing Figure 7(a), it is evident that at lower frequencies, the spectrum

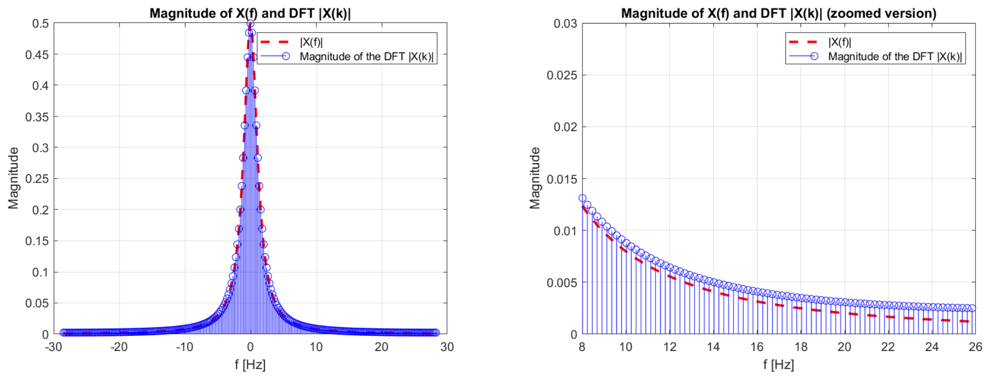


Figure 7: (a) Magnitude of X(f) and DFT $|X(k)|$ and (b) Magnitude of X(f) and DFT $|X(k)|$ with x axis limited to $x \in [8, 26]$

of the sampled signal accurately represents the original one. However, as shown in the zoomed version in Figure 7(b), a slight discrepancy appears at higher frequencies. This difference is due to aliasing.

In fact, sampling introduces copies of the original spectrum at multiples of the sampling frequency. These copies may overlap with the main spectrum, causing distortions, as it is noticeable in the figure.

The main problem is that the signal is band-unlimited. This means that, even though most of its energy is concentrated at lower frequencies, there are still small contributions at very high frequencies. In particular, the fact that 99.9% of the energy is contained within a certain bandwidth B_x suggests that the high-frequency components are small but not entirely absent. Therefore, even these small high-frequency contributions can affect the numerical representation of the signal, leading to distortions caused by aliasing.

3 Band-limited signals

In this exercise, the aim is to compare the continuous-time Fourier transform and discrete Fourier transform. The signals $x[n]$ in the time domain and its equivalent in the frequency domain $Y(f)$ were defined as follows:

$$x[n] = 2B_1 \cdot sinc(2B_1 t) + (4B_2 \cdot sinc(2B_2 t) - 3B_2 \cdot sinc^2(2B_2 t) \cdot cos(2\pi f_0 t)$$

$$Y(f) = \Pi_{2B_1}(f) + \Pi_{2B_2}(f - f_0) + \Pi_{2B_2}(f + f_0) - \frac{3}{2}(\Lambda_{B_2}(f - f_0) + \Lambda_{B_2}(f + f_0)$$

where the bandwidths $B_1 = 16\ Hz$, $B_2 = 6\ Hz$, $f_0 = 64\ Hz$, and $f_c = 64\ Hz$. The signal was represented on $N = 128$ points.

The first two required outputs of this exercise were to calculate the frequency resolution Δf and the observation time interval T_0. Equations 2 and 3 respectively show the formulas and the result obtained.

$$\Delta f = \frac{f_c}{N} = \frac{64\ Hz}{128} = 0.5\ Hz \tag{2}$$

$$T_0 = N \cdot T_c = \frac{N}{f_c} = \frac{128}{64\ Hz} = 2\ s \tag{3}$$

The next task was to compare the CTFT $Y(f)$ with the DFT $X(k)$ computed using the built-in command `fft()`. To plot $Y(f)$ the frequency axis was defined using $M = 100N$ points spaced from $-\frac{f_c}{2}$ to $\frac{f_c}{2} - \frac{f_c}{M}$ with step $\frac{f_c}{M}$; similarly, the frequency axis for $X(k)$ was defined $-\frac{f_c}{2}$ to $\frac{f_c}{2} - \frac{f_c}{N}$ with step $\frac{f_c}{N}$. Figure 8 shows in the same plot both the transforms. It is evident, and highlighted in Figure 8b that the two function do not perfectly coincide.

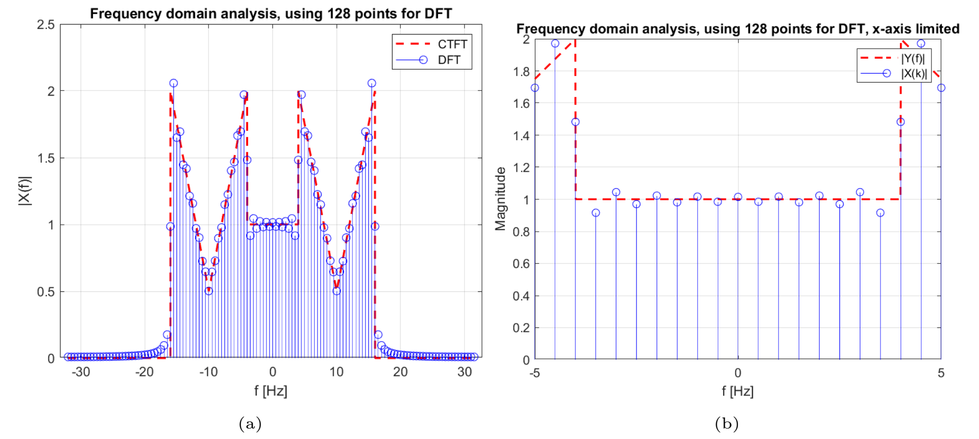


Figure 8: Comparison of $|X(k)|$ and $|Y(f)|$, with $N = 128$, (a) whole axis, (b) frequency axis limited between $[-5, 5]$

We were then asked whether there would be any differences if $x[n]$ were to be considered with a different support, i.e. if before computing the DFT we were to perform zero-padding on the original sequence. In particular, we created two new sequences: $x_a[n]$, $x_b[n]$, where the first consists of 128 additional points and the second of 384, meaning that $N_a = N + 128 = 256$, $N_b = N + 384 = 512$ samples. Since the total number of samples changed, the frequency resolution Δf also has changed:

$$\Delta f_a = 0.25\ Hz \qquad \Delta f_b = 0.125\ Hz$$

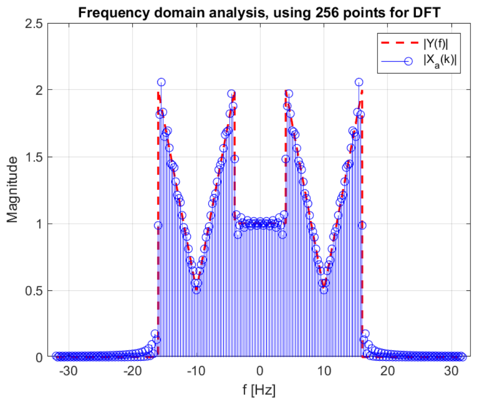
We can easily conclude that, as expected, doubling the number of samples, correspond in halving the frequency resolution, meaning that the discrete frequency axis k needed to be redefined accordingly.

Figure 9 and figure 10 show respectively the DFT for $x_a[n]$ and $x_b[n]$. Considering the whole frequency axis it is difficult to see whether the magnitude of the DFT perfectly matches with the CTFT, hence on sub-figure b it is reported an enlargement of the baseband of the signal. As the number of frequency samples increases (lower Δf) it becomes more evident that the DFT is the sum of sinusoidal components, leading to the observed oscillation pattern. The differences between the DFT $|X(k)|, |X_a(k)|, |X_b(k)|$ resides entirely in the number of points, while the magnitude is the same. To have a perfect match between $Y(f)$ and the DFT we would need to consider an infinite number of samples, only in this way the sum of sinusoids would be perfectly capable of providing an exact match.

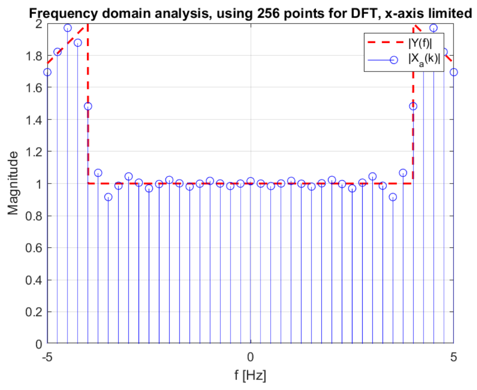
This can be explained by recalling that the continuous-time equivalent of $x[n]$ is $x(t) = x[n] \cdot rect_{T_0}(t)$. Truncating a signal using a rectangular window introduces the Gibbs phenomenon. Mathematically, the Gibbs phenomenon refers to the persistent overshoot and undershoot oscillatory behaviour around jump discontinuities when a function is approximated by a finite sum of sinusoidal components.

Using the MATLAB command `find()` we saved in $\tilde{Y}(f)$ the non-null element of $Y(f)$. using the first and last index returned by the function `find`, we were also able to find the total bandwidth of the signal: $B_w = 32\ Hz$. The last point of this exercise required to compute the analogue convolution in the frequency domain to compute $Z(f) = Y(f) * W(f)$, with

$$W(f) = \mathcal{F}\{\Pi_{T_0}(t)\} = T_0 \cdot sinc(T_0 f)$$



(a)



(b)

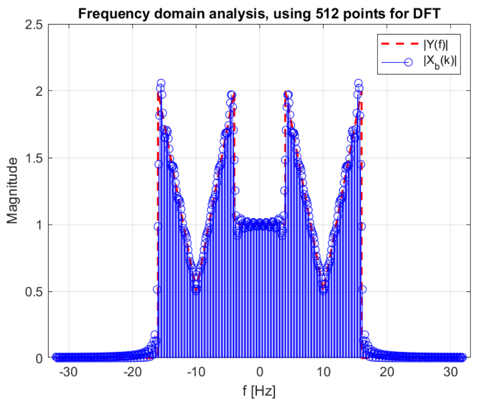
Figure 9: Comparison of $|X_a(k)|$ and $|Y(f)|$, with $N = 256$, (a) whole axis, (b) frequency axis limited between $[-5, 5]$

$W(f)$ is defined with a total bandwidth $f_c + B_w = 96\ Hz$ or equivalently it has a length $Q = M(1+\frac{B_w}{f_c})$. To calculate the convolution, instead of directly using the built-in function, a `for loop` was implemented, integrating, using `trapz`, the product of $\tilde{Y}(f)$ and a slided portion of $W(f)$. To be rigorous, since the definition of convolution integral is

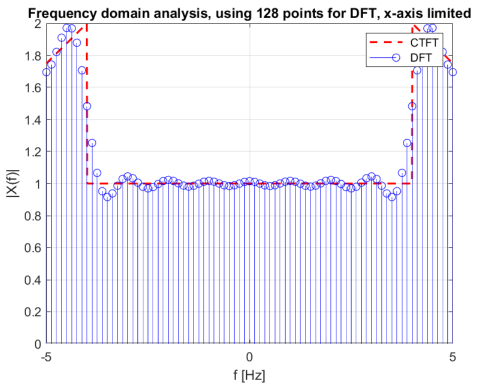
$$\int_{-\infty}^{\infty} \tilde{Y}(k) \cdot W(f-k)\, dk$$

a flipped version of $W(f)$ should be considered, as well the extrema of the integral should be infinite; as a matter of fact, in our implementation, there is no need for the flipping since $W(f)$, being a sinc function, hence even, it is symmetric with respect to the x-axis, and as extrema of integration $\pm\frac{f_c+B_w}{2}$, since the integrated signal is not defined elsewhere.

Analyzing the result shown in figure 11, it becomes evident that this time there is a perfect match between DFT $|X(k)|$ and the convolution result $|Z(f)|$. Looking at figure 11b it is clearly visible the sinusoidal behaviour not only of the DFT, but of the convolution as well.
This is due to the fact that $Z(t)$ also accounts for the presence of the rectangular time window, while $Y(f)$ is the Fourier transform of $x(t)$, defined on the entire time axis. Calculating the convolution between $Y(f)$ and $W(f)$ (sinc function) is equivalent to applying a rectangular window on the time domain. Equation 4 shows the mathematical relationship used to derive this equivalence.



(a)

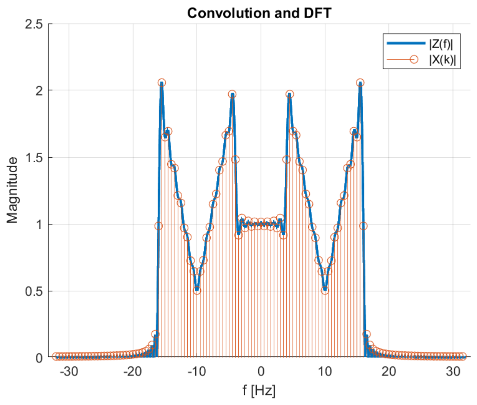


(b)

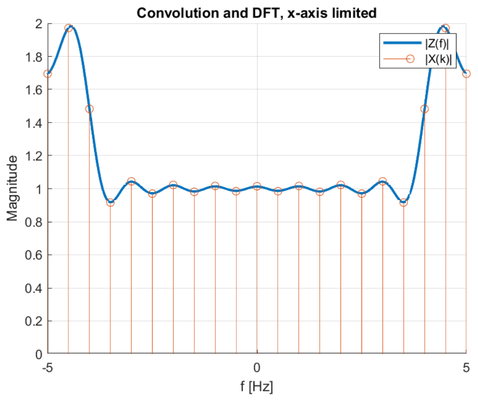
Figure 10: Comparison of $|X_b(k)|$ and $|Y(f)|$, with $N = 512$, (a) whole axis, (b) frequency axis limited between $[-5, 5]$

Summarizing our work, we have proved the equivalence between the time-domain multiplication and the convolution in the time domain, showing that the DFT calculated on a time-limited signal perfectly matches the convolution of the Fourier transform of the band-unlimited signal with a sinc, the Fourier transform of the time window.

$$\mathcal{F}\{x(t) \cdot rect_{T_0}(t)\} = \mathcal{F}\{x(t)\} * \mathcal{F}\{rect_{T_0}(t)\} = Y(f) * W(f) \tag{4}$$



(a)



(b)

Figure 11: Comparison of $|X(k)|$ and $|Z(f)|$ (a) whole axis, (b) frequency axis limited between $[-5, 5]$

4 Delay and frequency estimation through correlation

In this exercise, the following signal was considered:

$$x(t) = s(t) + w(t)$$

where $s(t)$ is a translation of $c(t)$, a pseudorandom binary sequence and $w(t)$ is a zero-mean Gaussian process with $\sigma^2 = 50$. The signal $x(t)$ has a total duration $T_{TOT} = 5\ s$ $c(t)$ has a duration $T_c = 2\ s$, and is defined from the sequence $b[k]$, a Bernoulli process of $B = \lfloor \frac{T_c}{T_b} \rfloor$, with $T_b = 1\ ms$. To obtain the Bernoulli binomial distribution, we used the following command:

```
b = binord(1, 0.5, 1, B)
```

where the first number represents the number of trial we want to simulate, in our case only one, 0.5 is the probability of success, i.e. of obtaining 1, and the last two argument specifies the length of the result vector. The output of the `binord` function is a vector of $\{0, 1\}$, that we mapped into $\{-1, 1\}$; to accomplish that, using `find` we saved the indexes for which the sequence is zero and substitute the 0 with -1.
Having finally obtained our vector $b[k]$ using the function `repmat()` and vectorizing the obtained matrix, $c(t)$ was obtained. Specifically, `repmat(a, y, z)` creates a matrix by repeating the input vector a $y \cdot z$ times; in our case, $y = T_b \cdot f_c = \frac{T_b}{T_c}$ and $z = 1$, corresponding to a total length:

$$M = 8000 \text{ samples}$$

Moreover, $s(t)$ was generated by zero-padding $c(t)$ on the left by $T_d \cdot f_c$ and on the right by $(T_{TOT} - T_c - T_d) \cdot f_c$ samples, where the delay T_d was generated as uniform random variable in the interval $[0, T_{TOT} - T_c]$. In Figure 12 are shown the signal $s(t)$, $x(t)$.

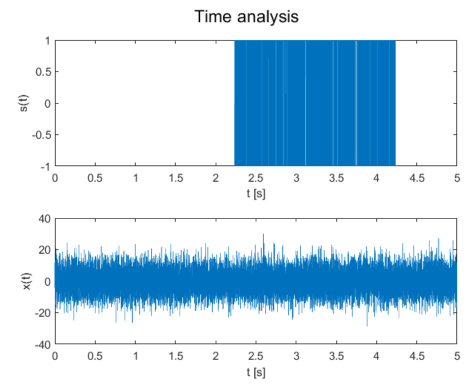


Figure 12: On the top signal $s(t)$, on the bottom $x(t)$ versus time

We proceed by defining the function `my_xcorr()`, which returns the signal $z(t)$ corresponding to the correlation of the two input signals, which has a length equal to the sum of the length of the two input vector minus 1 (i.e. if the input signal x, y have length N, M, respectively, the output will have length $N+M-1$). To compute the correlation, as defined in Equation 5, one `for loop` was used. In particular, to actually compute the correlation, the shorter vector was opportunely zero-padded to reach a number of samples equal to the output vector one, and then, applying a sliding window, the correlation result was saved in the output vector.

$$y(n) = \sum_{k=-\infty}^{\infty} y^*(n) \cdot x(k-n) \tag{5}$$

The goal of the first part of this exercise is to estimate the delay T_d, previously randomly generated using the correlation function. To do so, using the above mentioned `my_xcorr()` the correlation between $x(t)$ and $c(t)$ is calculated. The result is shown in Figure 13.
It is evident that the correlation has a maximum, $\hat{d}$, which, in samples, can be estimated as:

$$\hat{d} = arg \max_m \sum_{n=0}^{N-1} x[nt_c] \cdot c[(n-m)t_c] \tag{6}$$

Having found the sample delay index (with Eq. 6), the time delay is obtained by simply finding the corresponding element in the time axis (lags). In Figure

15, it is possible to see the randomly generated delay and the estimated one: they match.

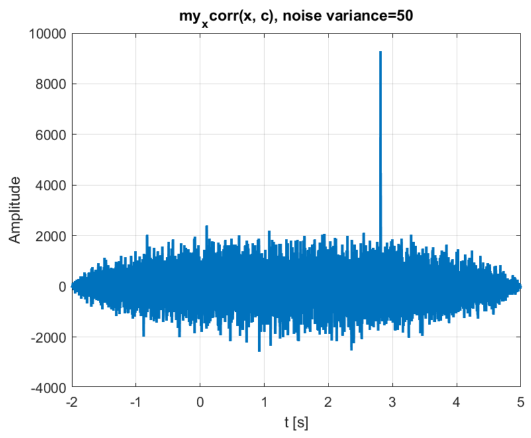


Figure 13: Correlation function calculated using user-defined `my_xcorr` function between the signals $x(t)$ and $c(t)$, for a noise $w(t)$ with a variance $\sigma^2 = 50$

The same process was repeated for a noise signal with a variance $\sigma^2 = 100$, that is double the original one, to see whether our algorithm was still able to estimate the delay. Figure 14 shows the output of the correlation function, having the new noise signal. The plot, as expected, resemble the one in Figure 13, but the increased variance of the noise results in larger fluctuations in amplitude. In Figure 13 almost none of the samples has an amplitude greater than 2000, while in Figure 14 this happens for many samples, reaching values up to 4000.

As it is possible to see in Figure 15, we were able to estimate correctly the delay in both cases. Specifically, to estimate the delay, the `max()` command was used, meaning that until the spike corresponding to the delay T_d has the maximum amplitude in the signal, our algorithm can estimate the delay correctly. While higher noise variance increases the overall amplitude of the cross-correlation signal, potentially masking the peak used for delay estimation and hindering the algorithm's performance in high-noise environments, our results demonstrate that even with a variance of 100, the delay spike remains sufficiently distinct for accurate detection. In fact, the delay is perfectly found by the algorithm and it corresponds to the one expected, therefore it can be considered correct.

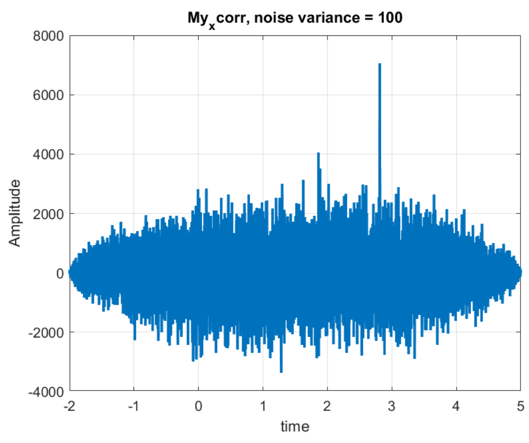
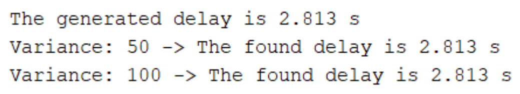


Figure 14: Correlation function calculated using user-defined `my_xcorr` function between the signals $x(t)$ and $c(t)$, for a noise $w(t)$ with a variance $\sigma^2 = 100$



```
The generated delay is 2.813 s
Variance: 50 -> The found delay is 2.813 s
Variance: 100 -> The found delay is 2.813 s
```

Figure 15: T_d original and estimated value for both the noise variance as calculated by MATLAB

In the last section of the exercise, the parameters described initially are slightly changed and the idea is to estimate both the delay and the modulating frequency of a transmitted signal. The pseudorandom binary sequence $c(t)$ is, in fact, first modulated by a cosine carrier of a randomly chosen frequency f_0 (selected from the set $\{10, 20, \ldots, 200\}$ Hz). The modulated signal is then delayed by the random amount T_d previously generated and corrupted by additive noise generated from a zero-mean Gaussian process $w(t)$, resulting in the final signal $y[n]$. Two noise variances are again considered: 50 at first and then 100.

For each candidate frequency in the set, a template signal is generated by modulating $c[n]$ with that frequency. The cross-correlation between the noisy received signal $y[n]$ and each template is computed using the custom function `my_xcorr()`. This produces a cross-correlation matrix A, where each row corresponds to a candidate frequency and each column corresponds to a time lag. The lag vector is defined as

```
lags = (-M+1 : N-1) * t_c,
```

with $t_c = 1/f_c$, ensuring that the lags are expressed in seconds.

The cross-correlation surface is visualized using a 3D mesh plot with the x-axis representing the lag (in seconds), the y-axis representing the candidate frequencies (in Hz), and the z-axis showing the cross-correlation amplitude. To identify the transmitted signal's delay and modulating frequency, we locate the peak of the cross-correlation matrix A by using MATLAB's `max()` function twice.

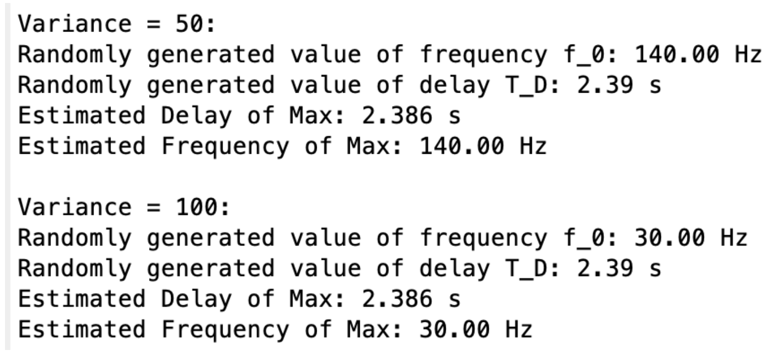
In the first call, the `max()` function is applied along the columns to obtain the maximum cross-correlation value in each row and its corresponding index d (in samples), which represents the lag where the peak occurs for that frequency. In the second call, the function is applied to the vector of maximum values max_col to identify the candidate frequency (indexed by $f0\_idx$) that produced the overall highest cross-correlation peak.

This operation yields the index corresponding to the estimated modulating frequency $\hat{f}_0$ and the associated lag $\hat{d}$ at which the maximum occurs.

The delay in seconds $\hat{T}_D$ is then computed from the lag value at the retrieved index $d(f0\_idx)$ using the relation

```
delay_max = lags(d(f0_idx)-1);
```

where t_c is the sampling period. Finally, the estimated delay $\hat{T}_D$ and frequency $\hat{f}_0$ are compared with the originally generated delay and frequency using print statements, as shown in Figure 16.



```
Variance = 50:
Randomly generated value of frequency f_0: 140.00 Hz
Randomly generated value of delay T_D: 2.39 s
Estimated Delay of Max: 2.386 s
Estimated Frequency of Max: 140.00 Hz

Variance = 100:
Randomly generated value of frequency f_0: 30.00 Hz
Randomly generated value of delay T_D: 2.39 s
Estimated Delay of Max: 2.386 s
Estimated Frequency of Max: 30.00 Hz
```

Figure 16: MATLAB console output displaying the randomly generated frequency (f_0) and delay (T_D), along with their estimated values for two different noise variance scenarios.

Figures 17 and 18 display the cross-correlation surfaces for noise variances of 50 and 100, respectively. When the noise variance increases, the overall amplitude of the cross-correlation surface rises, resulting in a higher noise floor around the peak. Nevertheless, the main peak remains clearly distinguishable, allowing for reliable estimation of both delay and frequency.

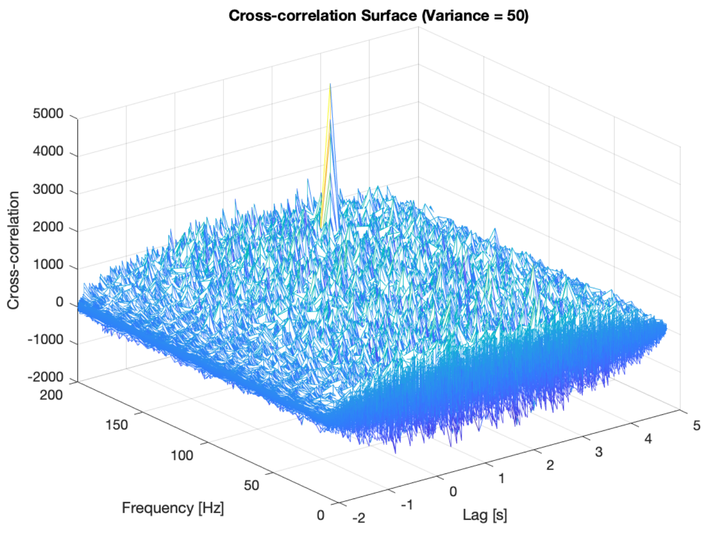


Figure 17: Cross-correlation surface for noise variance = 50. The mesh plot shows the cross-correlation A as a function of time lag and candidate frequency.

5 Spectral estimation

The main goal of this exercise is to perform a classical spectral estimation of the following signal:

$$X(t) = \frac{20}{\sqrt{2\pi\eta}} \exp\Bigl(-\frac{t^2}{2\,\eta^2}\Bigr) \cos\bigl(2\pi f_1\, t\bigr) + \cos\bigl(2\pi f_2\, t\bigr) + \cos\bigl(2\pi f_3\, t\bigr) + W(t),$$

where $\eta = 0.01,\ f_1 = 100\,\text{Hz},\ f_2 = 500\,\text{Hz},\ f_3 = 510\,\text{Hz}$, sampling frequency $f_c = 2\,\text{kHz}$, and $W(t)$ is a zero-mean white Gaussian noise process with variance $\sigma_n^2 = 25$.

We start by generating 20 seconds of the signal. In order to do so, we need to create a 20 seconds time axis whose number of elements is given by the product between the total time and the sampling frequency f_c. Then, $W(t)$ can be developed through the built-in function `randn()`, and since it returns

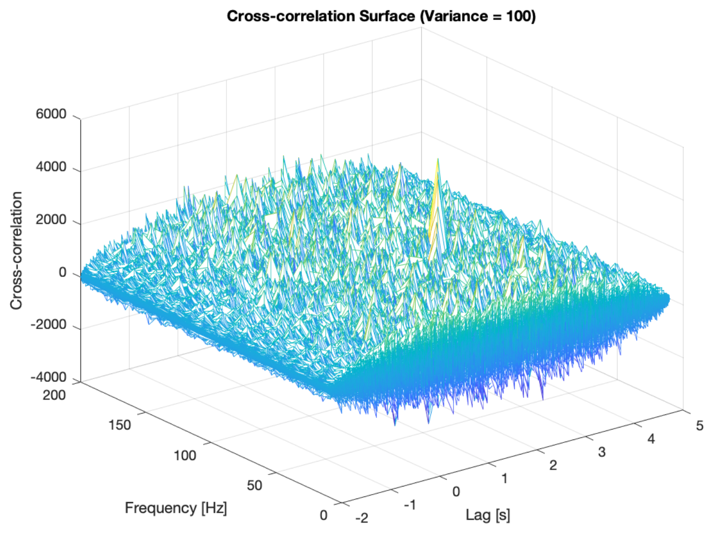


Figure 18: Cross-correlation surface for noise variance = 100. The peak is less pronounced due to increased noise, affecting the clarity of the delay and frequency estimates.

an array of random scalars from the standard normal distribution, it is then multiplied by the given standard deviation ($\sqrt{variance}$).

After this initial setup, the estimate of the power spectral density $S_x(f)$ was obtained using the Welch method, through the following command:

```
[S_x, f] = pwelch(x, window, n_overlap, NFFT, fs, 'centered');
```

This approach involves splitting the time-domain signal $x[n]$ into multiple overlapping segments. Here, NFFT specifies the number of segments, and n_{overlap} is the number of overlapping elements among those segments. Each segment is multiplied by a window function, commonly a Hamming or Hann window, rather than a simple rectangular window, to limit the altering effects of the abrupt edges of a rectangular window on the estimated spectrum. The individual segment periodograms are then averaged to produce the Welch estimate of $S_x(f)$.

Although our signal contains two close sinusoidal components at 500 Hz and 510 Hz, the initial Welch parameters do not provide sufficient frequency resolution to distinguish them as two distinct peaks. Specifically, with a small number of samples per segment ($NFFT$) and no overlap, the main lobes of the windowed segments are too wide, causing the two harmonics to merge into a single peak in the PSD. In addition, the side lobes of the chosen window (when used with short segments) can obscure closely spaced spectral lines.

To obtain a good spectral resolution maintaining a low variance of our PSD estimate, we must balance two primary factors:

1. **Frequency Resolution**, which improves by having more samples in each segment (larger $NFFT$).
2. **Variance Reduction**, which typically requires increasing the number of segments (smaller $NFFT$). Overlapping segments and using an appropriate window can also help reduce the variance of the final estimate.

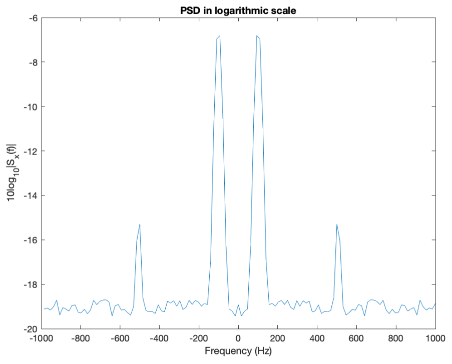
Table 1 shows how we updated the parameters: we slightly reduced $NFFT$ to obtain more segments for averaging (thus lowering variance) and increased the overlap to preserve as many data points in each segment as possible. Additionally, we switched from a Hamming window to a Hann window to reduce spectral leakage while still maintaining a reasonable main-lobe width. In practice, finding the ”optimal parameters” involves trial and error, guided by theoretical trade-offs in spectral resolution versus variance.

Table 1: Summary of initial and adjusted parameters.

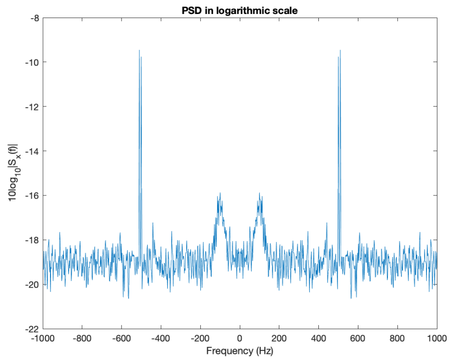
Parameter	**Initial**	**Adjusted**
$NFFT$	128	1024
n_{overlap}	None	512 ($NFFT/2$)
$Window$	Hamming	Hamming
$Variance$	1.29e-03	4.46e-05
Reference Figure	19 (a)	19 (b)

As shown in Figure 19(b), although the two harmonics are still relatively close in frequency, this adjusted parameter set strikes a reasonable balance between resolution (peak separation) and variance reduction in the PSD estimate.

The next step is to set the parameters in order to obtain three different spectral estimates:



(a)



(b)

Figure 19: Plot of the power spectral density, $S_x(f)$, in logarithmic scale, using the initial parameters (a) and the adjusted parameters (b).

- A simple periodogram.
- A Bartlett periodogram with $M = 25$ segments.
- A Welch periodogram with a Hamming window, segment length $D = \frac{N}{M}$, and 50% overlap.

These three methods differ in how they partition and window the data, thus affecting both the variance and the spectral resolution of the resulting power spectral density estimate. A summary of the chosen parameters for each method is provided in Table 2. Figure 20 then compares the three periodograms, plotted in the same figure for ease of comparison.

Table 2: Parameters used for Simple, Bartlett, and Welch periodograms.

Parameter	**Simple**	**Bartlett**	**Welch**
M (Segments)	1 (entire signal)	25	25
$NFFT$	N	N/M	N/M
n_{overlap}	None	None	$NFFT/2$ (50%)
$Window$	Rectangular	Rectangular	Hann
$Variance$	6.27e-02	5.78e-02	6.58e-05
Reference Figure	20 (Blue)	20 (Red)	20 (Orange)

From Table 2, it is evident that the Welch periodogram has the lowest variance (6.58×10^{-5}), confirming the smoothing effect introduced by segment overlap and the Hann window. Conversely, the simple periodogram, while exhibiting the highest variance, preserves the original frequency resolution.

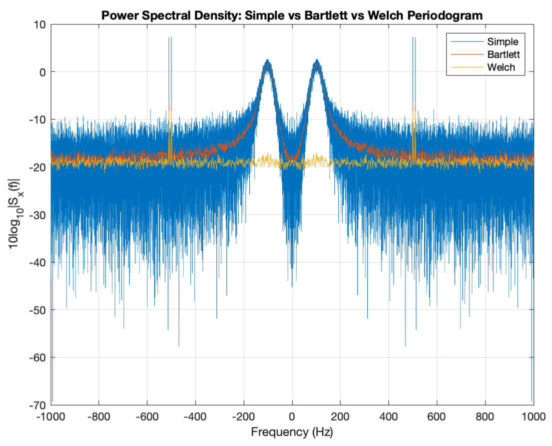


Figure 20: Comparison of the three periodograms (Simple, Bartlett, and Welch) in the same figure.

In general, the variance of periodogram estimates decreases progressively from the simple method to Welch's method. Bartlett's approach reduces variance by averaging over segments, while Welch's method further lowers it by incorporating overlapping segments and windowing. However, this variance reduction comes at the expense of spectral resolution, which is highest in the simple periodogram and progressively degrades in Bartlett and Welch methods. Therefore, the choice of method depends on the trade-off between variance reduction and the desired frequency resolution.

In order to estimate the autocorrelation function of $X(t)$, we can exploit the built-in `xcorr()` command in MATLAB. Specifically, we employ two normalization strategies:

- **Unbiased**: Corrects for the varying number of overlapping samples at each lag, producing $\hat{R}_N[l]$.
- **Biased**: Divides by a constant (full signal length), yielding $\hat{R}'_N[l]$.

In MATLAB, these are invoked as follows:

```
[R_N, lags] = xcorr(x, 'unbiased');
[R_prime_N, lags_prime] = xcorr(x, 'biased');
```

Figure 21 compares the two estimators using a 2×1 subplot. The top plot displays the unbiased autocorrelation in dB, which compensates for the

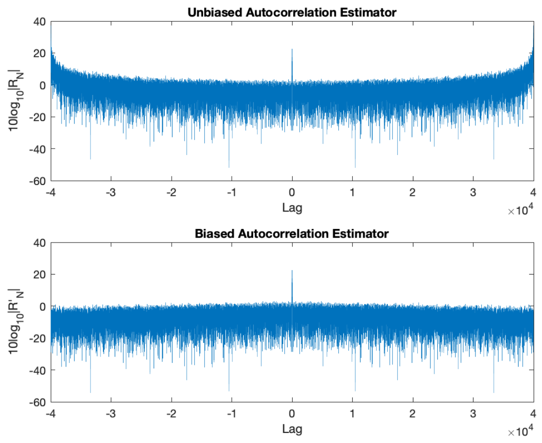


Figure 21: Comparison of unbiased ($\hat{R}_N[l]$) and biased ($\hat{R}'_N[l]$) autocorrelation estimates, shown in dB scale.

reduced number of overlapping samples at large lags. The bottom plot shows the biased autocorrelation in dB, where a constant divisor equal to the total sample length is used for all lags, resulting in a slight underestimation of the autocorrelation at nonzero lags. The biased estimator exhibits reduced variance compared to the unbiased estimator, making it generally preferred in practical applications despite its slight negative bias.

After acquiring the signal $x[n]$, we design and apply a low-pass FIR filter $h[n]$ of length $N = 165$. This filter is constructed by multiplying a scaled `sinc()` function by a Hamming window, ensuring a smoother transition band and reduced side lobes compared to a simple rectangular truncation. The filtered output is $y[n]$.

Next, we estimate the power spectral density (PSD) of $y[n]$ using the Welch method with a segment length $D = 1600$ and 50% overlap ($n_{\text{overlap}} = D/2$). Three different windows are employed: rectangular, Hamming, and Hann. The corresponding periodograms are overlaid in the same figure for comparison.

As shown in Figure 22, the rectangular window typically exhibits the lowest out-of-band attenuation, meaning its side lobes are relatively high compared to the other windows. Both the Hamming and Hann windows offer better side-lobe suppression, resulting in higher out-of-band attenuation. The overlaid periodograms reveal that the rectangular window shows the largest side lobes (i.e., the highest power in the stopband), indicating its comparatively poorer out-of-band attenuation.

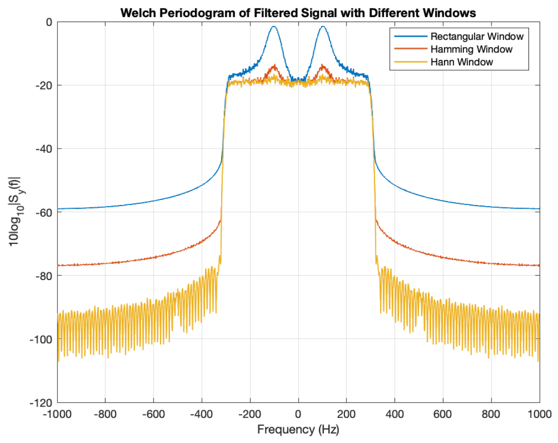


Figure 22: Welch Periodogram of Filtered Signal with Different Windows. Rectangular (Blue), Hamming (Red), and Hann (Orange) (with $D = 1600$ and 50% overlap).

In the last part of the analysis, the power spectral density (PSD) of the filtered signal $Y(t)$ is estimated using two approaches: a Bartlett periodogram and a simple periodogram. Two custom functions, `my_PSD()` and `my_DFT()`, were developed to perform these computations.

The function `my_DFT()` implements the discrete Fourier transform (DFT) using a matrix multiplication approach. A DFT matrix is constructed where each entry is given by an exponential term. The input signal is multiplied by this matrix, and the resulting transform is circularly shifted so that the zero-frequency component is centered. This custom implementation serves as an alternative to MATLAB's built-in `fft()` function.

The function `my_PSD()` computes the PSD of a signal in two modes:

- **Bartlett Periodogram:** When a segmentation parameter M is provided via the optional argument (`varargin`), the signal $Y(t)$ is divided into M segments. If the length of $Y(t)$ is not an integer multiple of M, zero-padding is applied to the signal to ensure equal-length segments. For each segment, the PSD is computed using the custom DFT implemented in "$my\_DFT()$". A `for loop` iterates over all segments, storing the PSD of each segment in a matrix. The final Bartlett estimate is obtained by averaging these individual periodograms.

- **Simple Periodogram:** If no segmentation parameter is provided, the function directly computes the PSD using the entire signal. This is done by applying the built-in `fft()` function, shifting the zero frequency to the center with the `fftshift()` function, and then computing the power.

The use of `varargin` allows the function to be versatile by supporting both PSD computation methods. In the segmented (Bartlett) case, the function returns a vector of normalized frequencies ranging from $-\frac{1}{2}$ to $\frac{1}{2} - \frac{1}{D}$ with a step size of $\frac{1}{D}$, where D is the length of each segment. For the simple periodogram, it returns a vector of normalized frequencies from $-\frac{1}{2}$ to $\frac{1}{2} - \frac{1}{N}$ with a step size of $\frac{1}{N}$, where N is the length of the signal $Y(t)$. The loop in the segmented case ensures that each block of data is processed individually, yielding an estimate that benefits from averaging to reduce variance.

To validate the custom implementations, the PSD estimates obtained from `my_PSD()` are compared against those computed by MATLAB's `pwelch()` function. Two comparisons are made:

1. **Bartlett Periodogram:** The PSD is computed using rectangular windows whose lengths correspond to the segmentation size (length$(y)/M$), with no overlap. Both the custom method and `pwelch()` produce estimates that are averaged to obtain smoother spectral estimates.

2. **Simple Periodogram:** The entire signal $y[n]$ is used, again comparing the built-in `pwelch()` estimate with that from `my_PSD()`.

The resulting plots, as shown in Figures 24 and 23, illustrate the consistency between the custom implementations and MATLAB's built-in functions.

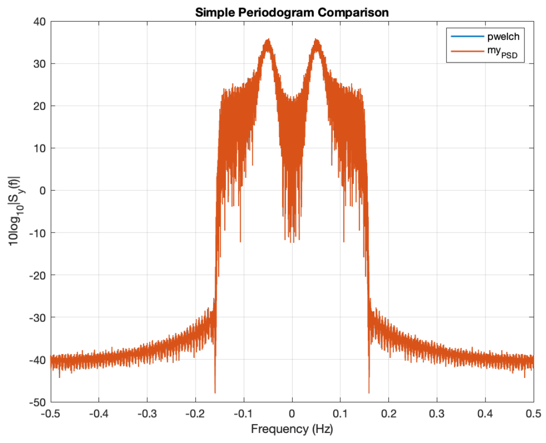


Figure 23: Comparison of the simple periodogram estimates: `pwelch()` (using a rectangular window) versus the custom `my_PSD()` function.

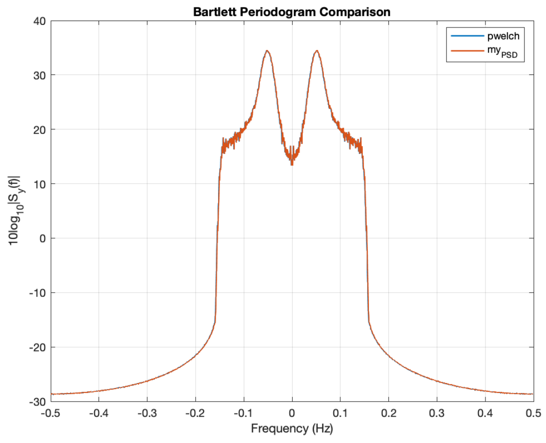


Figure 24: Comparison of the Bartlett periodogram estimates: `pwelch()` (using a rectangular window with segmentation) versus the custom `my_PSD()` function.

In particular, the Bartlett periodogram (obtained by segmenting and averaging) demonstrates a reduction in variance compared to the simple periodogram, which aligns with theoretical expectations since averaging multiple estimates helps to smooth out random fluctuations, leading to a more stable power spectral density estimate.