University of South Wales
Prifysgol De Cymru

Faculty of Computing, Engineering and Science

Final mark awarded:_____

Assessment Cover Sheet and Feedback Form 2017/18

Module Code: **NG4S804**	**Module Title:** **Applied DSP**	**Dr Roula**
Assessment Title and Tasks: DFT and frequency analysis		Assessment No. e.g. 1 of 1
No. of pages submitted in total including this page: **61**		Word Count of submission (if applicable): **9659**

Date Set: 25/10/2017	Submission Date: 15/12/2017	Return Date: 12/01/2018

Part A: Record of Submission (to be completed by Student)

You are required to acknowledge that you have read the above statements by writing your student number(s) in the box:	Student Number(s): **14031329**

IT IS YOUR RESPONSIBILITY TO KEEP RECORDS OF ALL WORK SUBMITTED

[1] University Academic Misconduct Regulations
[2] Information on exclusions to this rule is available from the Advice Centre at each Campus

Part B: Marking and Assessment **(to be completed by Module Lecturer)**
This assignment will be marked out of 100% This assignment contributes to 40% of the total module marks. This assignment is bonded Details:
Assessment Task: See below
Learning Outcomes to be assessed (as specified in the validated module descriptor https://icis.southwales.ac.uk/):
Grading Criteria: **Part 1** **Execution 20** **Analysis 25** **Part 2** **Execution 20** **Analysis 25** **Report quality 10**

Feedback/feed-forward (linked to assessment criteria):

- Areas where you have done well:

- Feedback from this assessment to help you to improve future assessments:

- Other comments

Mark:	**Marker's Signature:**	**Date:**

☐ **Work on this module has been marked, double marked/moderated in line with USW procedures.**

Provisional mark only: subject to change and / or confirmation by the Assessment Board

Part C: Reflections on Assessment **(to be completed by student – optional)**	
Use of previous feedback: In this assessment, I have taken/took note of the following points in feedback on previous work:	
Please indicate which of the following you feel/felt applies/applied to your submitted work • A reasonable attempt. I could have developed some of the sections further. ☐ • A good attempt, displaying my understanding and learning, with analysis in some parts. ☐ • A very good attempt. The work demonstrates my clear understanding of the learning supported by relevant literature and scholarly work with good analysis and evaluation. ☐ • An excellent attempt, with clear application of literature and scholarly work, demonstrating significant analysis and evaluation. ☐	
What I found most difficult about this assessment:	
The areas where I would value/would have valued feedback:	

DFT and Frequency Analysis

University of South Wales

NG4S804 - Applied Digital Signal Processing

Assignment 1

14031329
Miguel Santos

15 December 2017
2017/2018

Contents

Introduction **6**

Part 1 **7**

1.A) 256 DFT on a sine wave . . . 11
1.B) 256 DFT on a mix of three sine waves . . . 16
1.C) 256 DFT on a square wave . . . 18

Part 2 **21**

2.1) 512 DFT on all three input signals . . . 22
2.2) 512 DFT on all three input signals with Hann window . . . 27
2.3) (Research) The Welch and Gaussian windows . . . 32

Conclusion **37**

References **37**

Appendices **40**

A Code **41**

A.1 main.c . . . 42
A.2 dsp_functions.c . . . 44
A.3 dsp_functions.h . . . 47
A.4 question1_a.c . . . 48
A.5 question1_b.c . . . 50
A.6 question1_c.c . . . 52
A.7 output_plot.c . . . 54
A.8 output_plot.h . . . 56

List of Figures

1 Sine wave input [f = 2 kHz f_s = 15 kHz amplitude = 10 and 256 samples] 11
2 DFT spectrum [256 points Single-Sideband (SSB)] of sine wave [f = 2 kHz f_s = 15 kHz amplitude = 10 and 256 samples] 12
3 DFT spectrum [256 points SSB] of sine wave [f = 2 kHz f_s = 15 kHz amplitude = 10 and 256 samples] dB 13
4 Mixed sine wave input [f = 0.5 kHz + 1 kHz + 2 kHz f_s = 15 kHz amplitude = 10 and 256 samples] . . 16
5 DFT spectrum [256 points SSB] of three mixed sine waves [f = 0.5 kHz + 1 kHz + 2 kHz amplitude = 10 and 256 samples] dB 17
6 Square wave input [f = 0.5 kHz f_s = 15 kHz amplitude = 20 and 256 samples] 18
7 DFT spectrum [256 points SSB] of square wave [f = 0.5 kHz f_s = 15 kHz amplitude = [0, 20] and 256 samples] dB 19
8 Sine wave of 256 samples 23
9 256 DFT spectrum of 2 kHz sine wave (dB SSB) 23
10 Sine wave of 512 samples 23
11 512 DFT spectrum of 2 kHz sine wave (dB SSB) 23
12 Mixed sine wave of 256 samples 24
13 256 DFT spectrum of mixed sine wave (dB SSB) 24
14 Mixed sine wave of 512 samples 24
15 512 DFT spectrum of mixed sine wave (dB SSB) 24
16 500 Hz square wave of 256 samples 25
17 256 DFT spectrum of 500 Hz square wave (dB SSB) 25
18 500 Hz square wave of 512 samples 25
19 512 DFT spectrum of 500 Hz square wave (dB SSB) 25
20 The Hann window 27
21 The Hann window in the frequency domain (dB Double-Sideband (DSB)) 27
22 The Hann window in the frequency domain (dB Single-Sideband (SSB)) 27
23 Sine wave of 512 samples without Hann window 28
24 Sine wave of 512 samples with Hann window 28
25 512 DFT spectrum of sine wave without Hann window (dB SSB) 29
26 512 DFT spectrum of sine wave with Hann window (dB SSB) 29
27 Mixed sine wave of 512 samples without Hann window 30
28 Mixed sine wave of 512 samples with Hann window 30
29 512 DFT spectrum of mixed sine wave without Hann window (dB SSB) 30

30 512 DFT spectrum of mixed sine wave with Hann window (dB SSB) 30
31 500 Hz square wave of 512 samples without Hann window 31
32 500 Hz square wave of 512 samples with Hann window 31
33 512 DFT spectrum of square wave without Hann window (dB SSB) 31
34 512 DFT spectrum of square wave wave with Hann window (dB SSB) 31
35 The Welch window 32
36 Welch window spectrum (dB DSB) 32
37 The Hann window 32
38 Hann window spectrum (dB DSB) 32
39 Mix sine sine wave spectrum with Welch window (dB DSB) 33
40 Mix sine wave spectrum with Hann Window (dB DSB) 33
41 The Gaussian window 34
42 Gaussian window spectrum (dB DSB) 34
43 The Hann window 34
44 Hann window spectrum (dB DSB) 34
45 Mix sine sine wave spectrum with Gaussian window (dB DSB)) 35
46 Mix sine wave spectrum with Hann Window (dB DSB) 35

List of Tables

1 Peaks of the 256 DFT - Mixed sine wave 17
2 Peaks of the 256 DFT - Square wave 20
3 Peaks of the 512 DFT - Mixed sine wave 24
4 Peaks of the 512 DFT - Square wave 26
5 256 vs 512 DFT 26
6 Window comparison: Hann vs Welch vs Gaussian 36

Listings

1 Function: create_sinewave() - dsp_functions.c 11
2 Usage: allocating a sine wave - question1_a.c 12
3 Function: DFT() - dsp_functions.c 13
4 Usage: converting the input into the frequency domain - question1_a.c 14
5 Function: convert_to_log_scale() - dsp_functions.c 14
6 Usage: creating and mixing 3 sine waves - question1_b.c 17
7 Function: create_squarewave() - dsp_functions.c 19
8 Usage: allocating a square wave - question1_c.c 19
9 Function: create_window() - dsp_functions.c 28
A.1 main.c 42
A.2 dsp_functions.c 44
A.3 dsp_functions.h 47
A.4 question1_a.c 48
A.5 question1_b.c 50
A.6 question1_c.c 52
A.7 output_plot.c 54
A.8 output_plot.h 56

Introduction

This report will cover the first and last assignment for the module NG4S804 - Applied Digital Signal Processing on the topic of DFT and Frequency Analysis.

The two parts of this report challenge the student to develop a computer application written in the C Language in which a set of three different signals are to be processed and converted from the time domain into the frequency domain by the means of the DFT - Discrete Fourier Transform algorithm. These signals consist of a sine wave, a mix of three sine waves and a square wave.

Part one begins by giving context and briefly introducing the DFT algorithm. It then proceeds by applying the function to the three inputs using 256 points of resolution.

The second part of this report continues in the exact same manner, except the bin count will be increased from 256 to 512 points. The size of the input signals will also increase to 512 samples. After this, a comparison and explanation will be made in regards to the effect of the bin count over the shape of the DFT output.

This same question is then followed by an attempt to further improve the output. In order to do this, a Hann window function will be applied on the input signals before passing them through the DFT algorithm.

Finally, in the topic of filtering and signal processing, two more windowing functions will be researched - the Welch and Gaussian windows.

Part 1

To put into context, the Discrete Fourier Transform function is a simple way of extracting a finite signal's frequency content from its time domain in order to design digital filters or to convey modulated information, such as FM and OFDM.[1]

Intuitively, this is done by correlating the entire signal sample data against a set of multiples of a given frequency for $0 \leq k \leq N$, where k is the nth frequency "bucket" (also called bin) with which the time domain signal is going to be compared against and N being number of samples.[2,3]

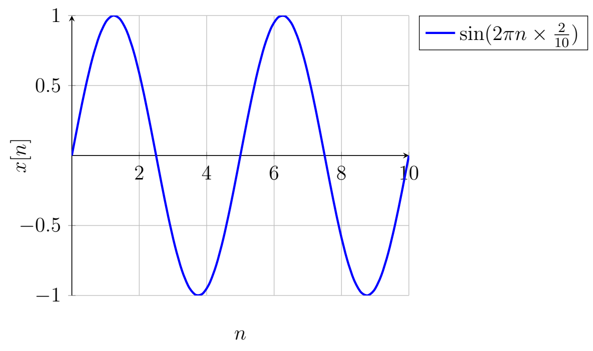
Take for example the finite signal:

$$x[n] = \sin(2\pi n \times \frac{f}{f_s}), 0 \leq n < N - 1$$

Where:

- f is the signal frequency (Hz)
- f_s is the sampling frequency (Hz)
- N is the number of samples

If we make $N = 10$, $f_s = 10$ Hz and $f = 2$ Hz we should observe the following output:



[1]Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 48, 49.

[2]Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 184, 186.

[3]Richard G. Lyons. *Understanding Digital Signal Processing.* Prentice Hall, 1996, pp. 54–63.

Each n sample in this signal is compared/multiplied against a set of coefficients $h[k]$. These are complex values with real and imaginary components (in the context of DFT).[4]
In the DFT algorithm, the contents of such coefficients are made from a cosine and sine as a complex number, such that:

$$h[k] = cos\Big(\frac{2\pi kn}{N}\Big) - j\sin\Big(\frac{2\pi kn}{N}\Big)$$

Where:

- k is the multiple of the signal frequency (bin/bucket)
- n is the nth sample with which this coefficient is going to be "compared"/multiplied against
- N is the total amount of samples in the signal

This can be simplified using Euler's formula:

$$h[k] = cos\Big(\frac{2\pi kn}{N}\Big) - j\sin\Big(\frac{2\pi kn}{N}\Big) = e^{-jnk(\frac{2\pi}{N})}$$

The process continues by accumulating the multiplication of these coefficients with the data samples in order to obtain some value that corresponds to some kind of correlation between each frequency multiple:

Let $Correlation[k]$ be the sum of the weighted data samples,

$$Correlation[k] = \sum_{n=0}^{n=N-1} x[n] \times h[k]$$

So, calculating for $k = 0$ gives us the correlation between the entire signal data and a 0 Hz bin, which clearly is only possible in a signal with DC component.
Similarly, $k = 1$ tells us the frequency information such that: $f_k = k \times \frac{f_s}{N}$ Hz/bin $= 1 \times \frac{10}{10}$ Hz/bin $= 1$ Hz. If the correlation algorithm returns value 0 or approximate, then the signal does not contain any frequency component within that k bin.[5,6]

In other words, it's like comparing the sample's value repeatedly against a *cosine* and *sine* wave tuned for the desired frequency, thus, making a different coefficient for every nth frequency bin. For each comparison, the result is accumulated and by the end of the algorithm, we should have two results which tell us how much of the signal's frequency is contained within each multiple of a sine and cosine function.

[4]Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 56, 60.
[5]Ifeachor and Jervis, *Digital Signal Processing: A Practical Approach.*
[6]Richard G. Lyons. *Understanding Digital Signal Processing.* Prentice Hall, 1996, p. 53.

The higher the value is, the higher the correlation.[7,8]
Finally, we have reached the conclusion that this is exactly the same expression as the DFT formula.[9,10]

$$Correlation[k] = \sum_{n=0}^{n=N-1} x[n] \times e^{-jnk(\frac{2\pi}{N})}$$
$$\Leftrightarrow$$
$$X[k] = \sum_{n=0}^{n=N-1} x[n] \times e^{-jnk(\frac{2\pi}{N})}$$

Lastly, it's important to note that it is common to use the magnitude of the resultant real and imaginary values as the desired output of the DFT.[11] The real and imaginary components could however give more insight into the signal's properties, such as phase.

This calculation is easily done by first converting the exponent into its trigonometric form (Euler's formula again), followed by a sum of the two complex components squared:

$$|X[k]| = \sqrt{\left[\sum_{n=0}^{n=N-1} x[n] \times \cos\left(\frac{2\pi kn}{N}\right)\right]^2 + \left[\sum_{n=0}^{n=N-1} x[n] \times -j\sin\left(\frac{2\pi kn}{N}\right)\right]^2}$$

[7]Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 62, 64.

[8]Lyons, *Understanding Digital Signal Processing.*

[9]Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 56, 57.

[10]Richard G. Lyons. *Understanding Digital Signal Processing.* Prentice Hall, 1996, p. 50.

[11]Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, p. 58.

In regards to the report, this first section will apply the DFT theory that was just described to three different signals:

- A 2 kHz sine wave with amplitude of $A = 10$
- A mix of three sine waves with frequencies 500 Hz, 1 kHz and 2 kHz respectively and amplitudes of $A = 10$
- A square wave with amplitude $0 \leq A \leq 20$

All signals are composed of $N = 256$ samples and these are sampled at a frequency of $f_s = 15$ kHz.

Finally, the DFT function will be done with resolution of $k = 256$ points and this output will be displayed in the logarithmic scale by calculating $20 \log_{10}(Spectrum[k])$.

1.A) 256 DFT on a sine wave

The first signal to analyse is a simple sinusoidal. Its expression:

$$x(t) = 10\sin(2\pi t \times 2000)$$

Since the DFT is a discrete transform, the signal must be sampled first by dividing the frequency.[12]

$$x[n] = 10\sin\left(2\pi n \times \frac{2000}{15000}\right)$$

Which results in the following graph:

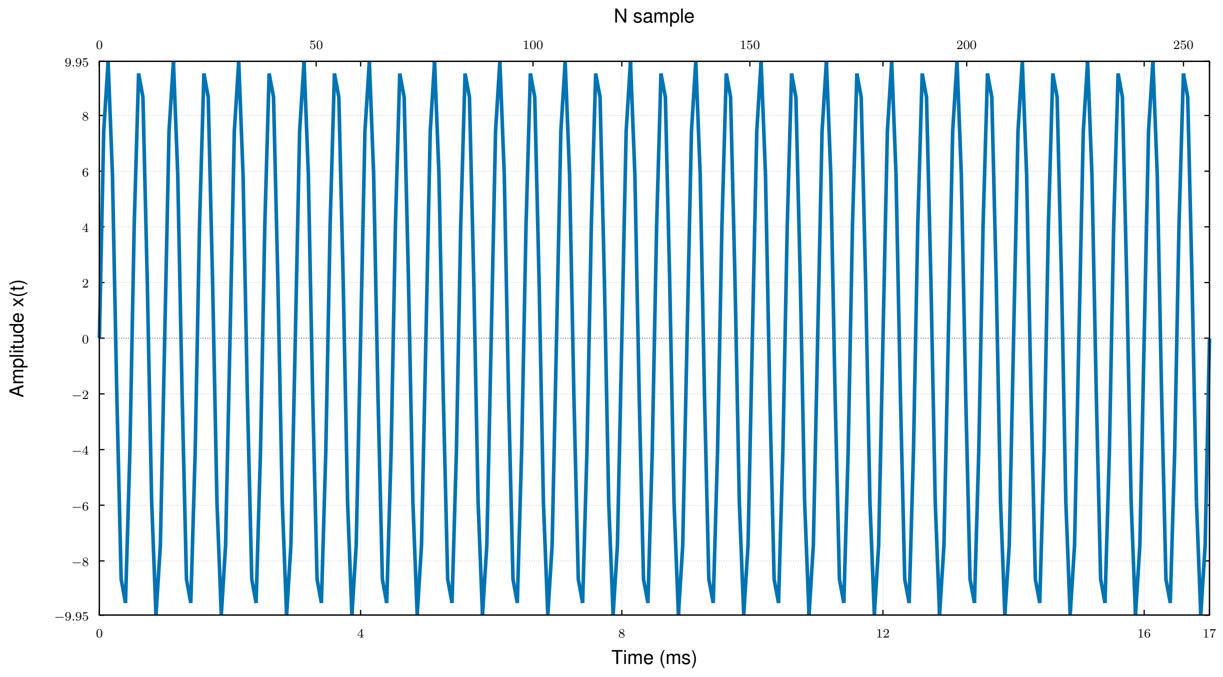


Figure 1: Sine wave input [f = 2 kHz f_s = 15 kHz amplitude = 10 and 256 samples]

This wave form was generated with the C function below:

```c
float * create_sinewave(unsigned int size, float x_low, float x_high, float amplitude, float frequency, float sampling_frequency)
{
    /* Create sine wave array */
    float * output = (float*)malloc(sizeof(float) * size);

    unsigned int n = 0;

    /* Store sine wave data in the array using 'x' as an 'n' variable
       that can go negative and 'n' the positive index to that sample */
    for (float x = x_low; x < x_high && n < size; x++, n++)
        output[n] = (float)(amplitude * sin(2 * PI * (frequency / sampling_frequency) * x));

    return output;
}
```

Code 1: Function: create_sinewave() - dsp_functions.c

[12]Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 55, 56.

Its usage:

```
    /* Create input signal (sine wave) */
    float * input_sinewave = create_sinewave(
        dft_points,               /* The total size of the signal                               */
        0,                        /* The starting point from which the signal begins (starts at x = 0) */
        (float)dft_points,        /* The end point at which the signal ends (ends at n = dft_points)   */
        SIGNAL_AMPLITUDE,         /* Amplitude of the signal                                    */
        SIGNAL_FREQUENCY,         /* Its frequency (Hz)                                         */
        SIGNAL_SAMPLING_FREQUENCY /* And finally, the sampling frequency (Hz) associated with it       */
    );
```

Code 2: Usage: allocating a sine wave - question1_a.c

After passing the signal through the DFT algorithm, we are presented with its single sided spectrum (linear scale):

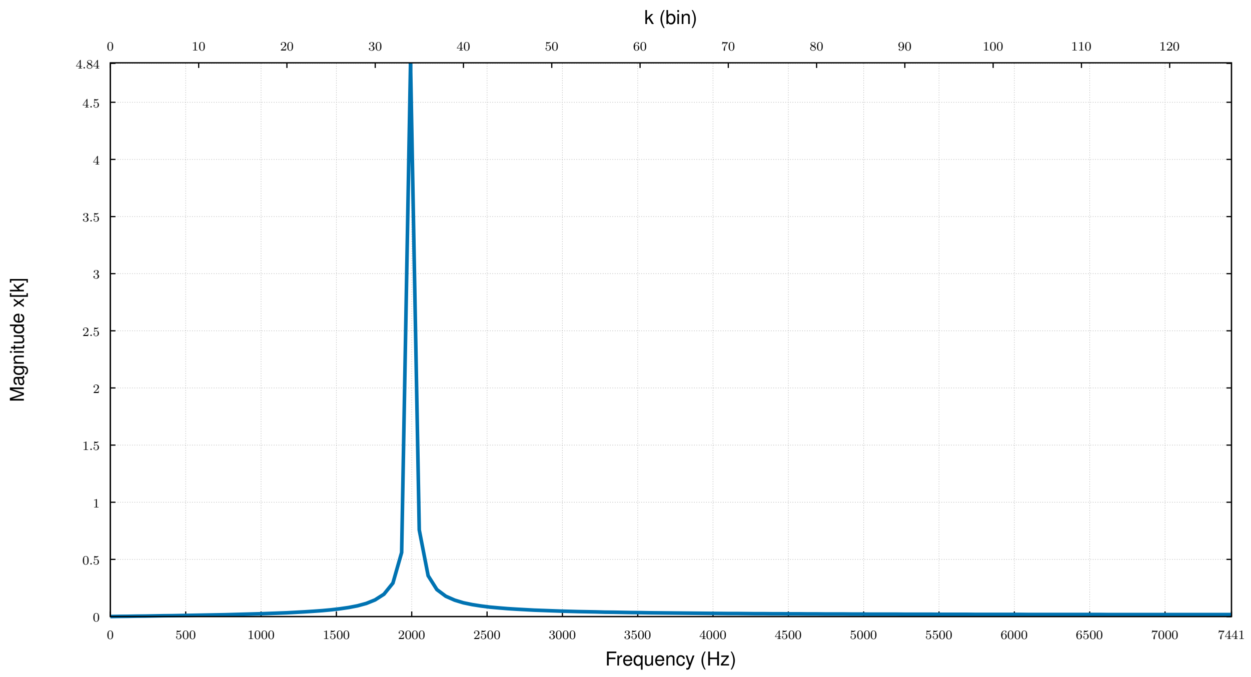


Figure 2: DFT spectrum [256 points Single-Sideband (SSB)] of sine wave [f = 2 kHz f_s = 15 kHz amplitude = 10 and 256 samples]

(**Note** that this is the last time a spectrum with linear scale is shown. The reason is due to the report's requirements. In addition, all spectrums will be Single Sideband (SSB) unless it is specified otherwise.)

In the logarithmic scale:

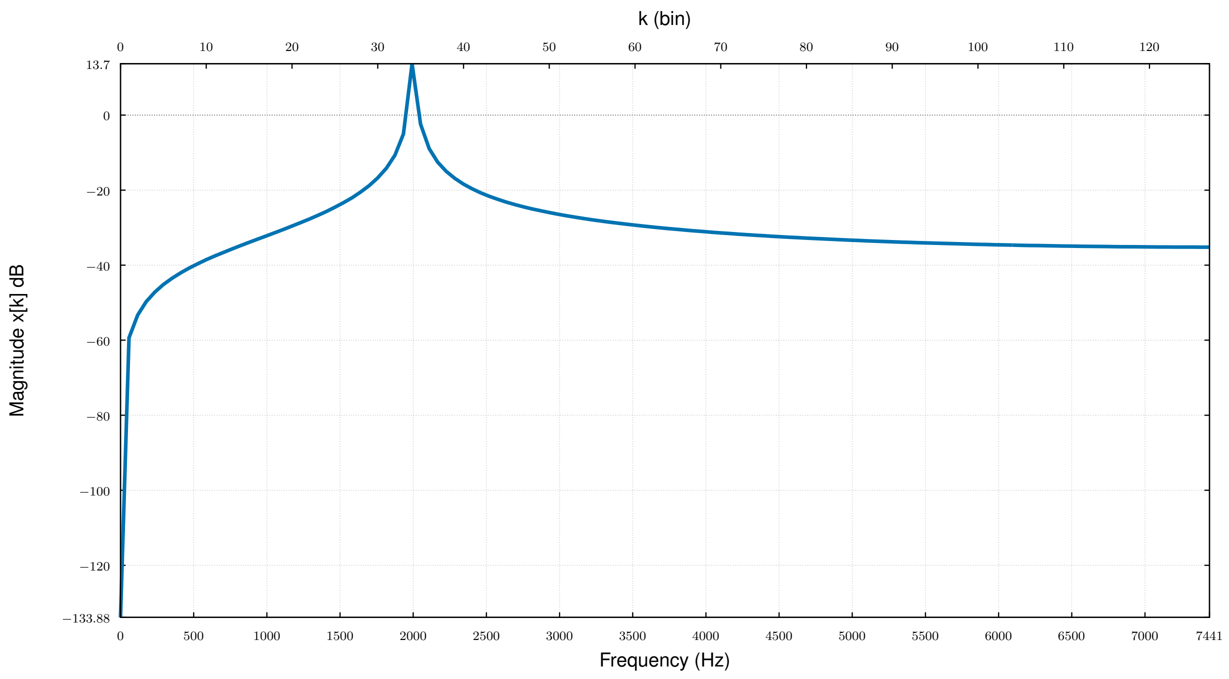


Figure 3: DFT spectrum [256 points SSB] of sine wave [f = 2 kHz f_s = 15 kHz amplitude = 10 and 256 samples] dB

The results were produced using the DFT function below:

```c
float * DFT(float * input, unsigned int input_size, unsigned int bin_count)
{
    /* In order to calculate the DFT, we'll create three signals /

    /* The real component of the DFT */
    float * dft_real = (float*)malloc(sizeof(float) * bin_count);
    /* The imaginary component */
    float * dft_imaginary = (float*)malloc(sizeof(float) * bin_count);
    /* And the magnitude between the two */
    float * dft_spectrum_output = (float*)malloc(sizeof(float) * bin_count);

    /* The frequency bins go to negative values -(N/2), so we'll calculate it here once */
    int bin = (bin_count / 2);

    for (int k = -bin, k_positive = 0; k < bin; k++, k_positive++) {
        /* Clear all magnitudes at this particular frequency bin */
        dft_real[k_positive]      = 0.0;
        dft_imaginary[k_positive] = 0.0;

        /* Add up all the coefficients multiplied with every single data sample */
        for (unsigned int n = 0; n < input_size; n++) {
            dft_real[k_positive]      += (float)(input[n] * cos(((2 * PI * k * n) / input_size)) / input_size);
            dft_imaginary[k_positive] += (float)(input[n] * sin(((2 * PI * k * n) / input_size)) / input_size);
        }

        /* And store the magnitude of the real and imaginary components */
        dft_spectrum_output[k_positive] = (float)((sqrt(pow(dft_real[k_positive], 2) + pow(dft_imaginary[k_positive], 2))));
    }

    /* We only care about the magnitude signal,
       so we'll cleanup the real and imaginary signals */
    free(dft_real);
    free(dft_imaginary);

    /* And finally return the DFT signal */
    return dft_spectrum_output;
}
```

Code 3: Function: DFT() - dsp_functions.c

And its usage:

```c
    /* Perform DFT on the input signal */
    float * dft_spectrum_output    = DFT(input_sinewave, dft_points, dft_points);
    float * dft_spectrum_output_db = convert_to_log_scale(dft_spectrum_output, dft_points);
    display_signal(output_name_dB, dft_spectrum_output_db, dft_points);
    display_spectrum_stats(dft_spectrum_output_db, dft_points, 1);
```

Code 4: Usage: converting the input into the frequency domain - question1_a.c

Also note *convert_to_log_scale*()'s function implementation:

```c
float * convert_to_log_scale(float * input, unsigned int input_size)
{
    /* Create array that will hold a copy of the input
       but with values in the log scale */
    float * input_copy = (float*)malloc(sizeof(float) * input_size);

    /* Convert the data */
    for(unsigned int n = 0; n < input_size; n++)
        input_copy[n] = (float)(20 * log10(input[n]));

    return input_copy;
}
```

Code 5: Function: convert_to_log_scale() - dsp_functions.c

Needless to say, Figure 3 clearly shows a peak at $k = 34$ which can be seen at the top of the horizontal axis. Even though the plot already displays the frequency information for that specific k, it's still important to show how this was calculated[13]:

$$f_k = \frac{k \times f_s}{N}$$

Where:

- N is the sample count ($N = 256$)
- f_s the sampling frequency ($f_s = 15$ kHz)

So, in this case, if the peak occurs at $k = 34$, then the signal frequency must be:

$$f_k = \frac{34 \times 15000}{256} \approx \mathbf{1992.2\ Hz}$$

And indeed the input signal is composed of a 2000 Hz sinusoidal, which clearly indicates that the C function $DFT()$ is implemented correctly.

Also, note the error between the expected value and the returned value, which is **error** $=$ **2000 Hz** $-$ **1992.2 Hz** $=$ **7.8 Hz**. This is one of the trade offs of treating analogue signals as finite discrete data.
Since we have $k = 256$ bins, with frequency ranges of $[0, 15000]$ Hz, and each k being a multiple of $f_k = \frac{f_s}{N} = \frac{15000}{256} = 58.59$ Hz, we notice that some information is going to be lost.

This is called Spectral leakage.[14]

[13]Lyons, *Understanding Digital Signal Processing*.
[14]Richard G. Lyons. *Understanding Digital Signal Processing*. Prentice Hall, 1996, pp. 71–80.

In short, we lack the ability to determine the frequency content below 58.59 Hz of resolution because the bin count is simply too small to represent an entire range of 15 kHz. If we wanted a resolution of 1, then we would increase the DFT point count to $f_s = 15000$, but even then we would not be able to process frequencies with resolution finer than 1 Hz. (e.g. 0.5 Hz, 0.25 Hz or 0.125 Hz.)

In addition, the C program defines k as an integer value, which means this variable can only hold natural numbers (due to the fact that arrays can only be indexed with integers) and therefore the only possible values in this case are:

$$\text{if } k = 34 \text{ then: } f_k = \frac{34 \times 15000}{256} = 1992.2\text{Hz}$$

and

$$\text{if } k = 35 \text{ then: } f_k = \frac{35 \times 15000}{256} = 2050.78\text{Hz}$$

Clearly the result for $k = 35$ goes way beyond 2000 Hz.

If we had the capacity to calculate k as a decimal value, then the real value of k for this signal should be: $k = \frac{f_k \times N}{f_s} = \frac{2000 \times 256}{15000} = 34.13$. However, the value was rounded down to $k = 34$, thus we obtain the error of 7.8 Hz.

By this logic, we conclude that an infinite amount of k bins would be the same as having the ability to process the signal's frequency information at any possible harmonic, with δf for every k being practically 0.

(hint: this infinite process is literally just a continuous Fourier Transform.[15])

[15]Ifeachor and Jervis, *Digital Signal Processing: A Practical Approach.*

1.B) 256 DFT on a mix of three sine waves

In this task the same approach is used for analysing a signal that is similar to the previous exercise. This time three sinusoids with different harmonics are to be added together.

$$x_1(t) = 10\sin(2\pi t \times 500)\ ,\ \mathrm{f} = 500\ \mathrm{Hz}$$

$$x_2(t) = 10\sin(2\pi t \times 1000)\ ,\ \mathrm{f} = 1000\ \mathrm{Hz}$$

$$x_3(t) = 10\sin(2\pi t \times 2000)\ ,\ \mathrm{f} = 2000\ \mathrm{Hz}$$

$$x_{mix}(t) = x_1(t) + x_2(t) + x_3(t)$$

All 3 input signals have amplitude = 10

After discretization (with the sampling frequency being $f_s = 15$ kHz):

$$x_{mix}[n] = 10\left[\sin\left(2\pi n \times \frac{500}{15000}\right) + \sin\left(2\pi n \times \frac{1000}{15000}\right) + \sin\left(2\pi n \times \frac{2000}{15000}\right)\right]$$

And this results in the following output:

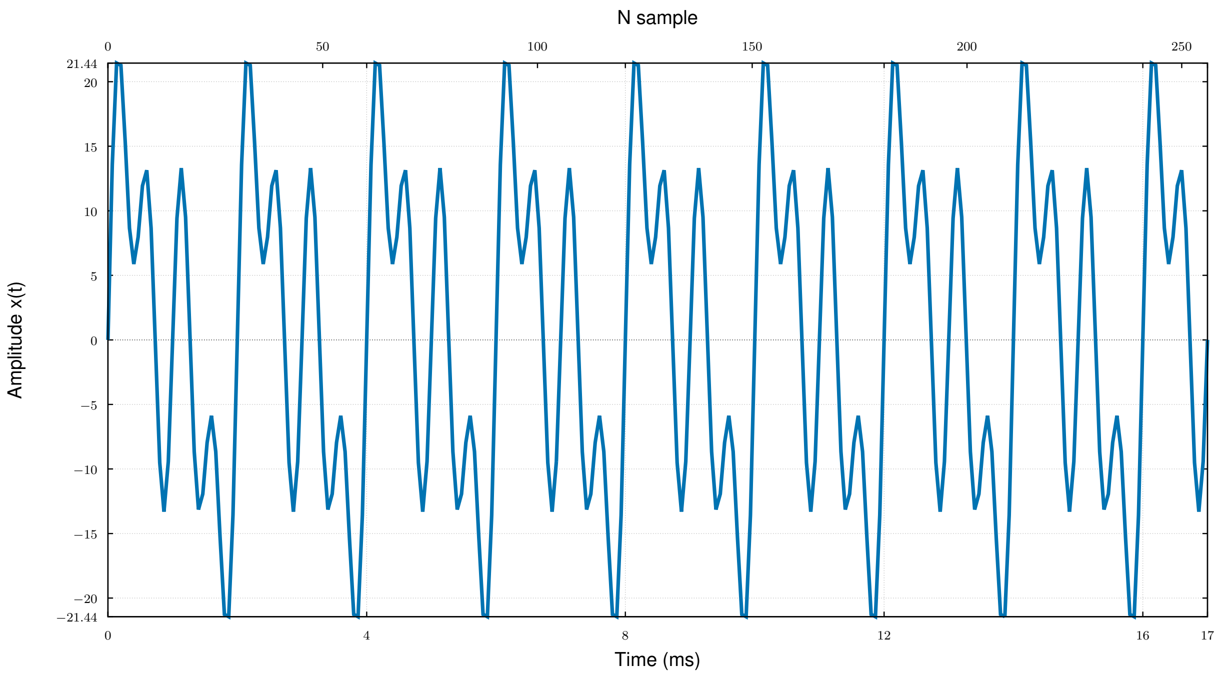


Figure 4: Mixed sine wave input [f = 0.5 kHz + 1 kHz + 2 kHz f_s = 15 kHz amplitude = 10 and 256 samples]

The generator function uses the function found in the listing *Code 1* followed by a simple addition with a for loop:

```c
    /* Create 3 sinewaves of different frequencies and a 4th signal which will store the mixed result */
    float * sin1 = create_sinewave(dft_points, 0, (float)dft_points, SIGNAL_AMPLITUDE, SIGNAL_FREQUENCY1, SIGNAL_SAMPLING_FREQUENCY);
    float * sin2 = create_sinewave(dft_points, 0, (float)dft_points, SIGNAL_AMPLITUDE, SIGNAL_FREQUENCY2, SIGNAL_SAMPLING_FREQUENCY);
    float * sin3 = create_sinewave(dft_points, 0, (float)dft_points, SIGNAL_AMPLITUDE, SIGNAL_FREQUENCY3, SIGNAL_SAMPLING_FREQUENCY);
    float * sin_mixed = (float*)malloc(sizeof(float) * dft_points);

    /* Mix the 3 signals and display the result */
    for (int n = 0; n < dft_points; n++)
        sin_mixed[n] = sin1[n] + sin2[n] + sin3[n];
```

Code 6: Usage: creating and mixing 3 sine waves - question1_b.c

Finally, the frequency spectrum:

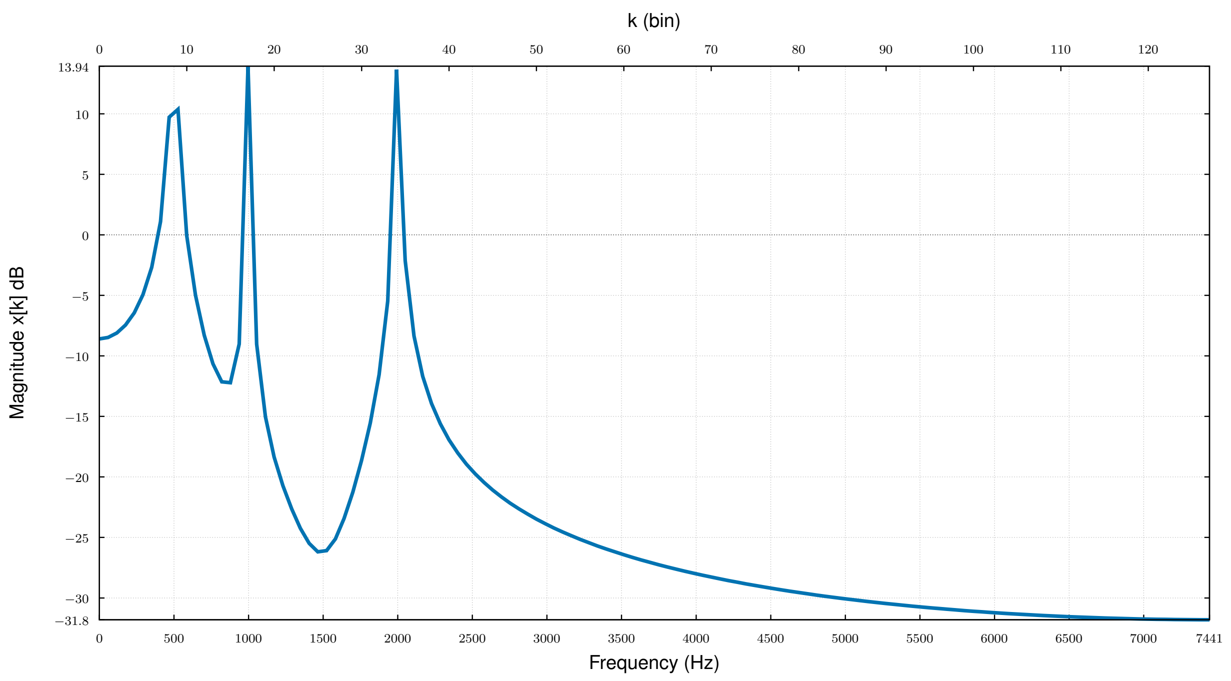


Figure 5: DFT spectrum [256 points SSB] of three mixed sine waves [f = 0.5 kHz + 1 kHz + 2 kHz amplitude = 10 and 256 samples] dB

Again, we clearly see the peaks of the three input signals at their correct frequency values. The k bins for all three peaks are (with approximation to the nearest integer):

Peak #	**k**	**expected k**	f_k **(Hz)**	**expected** f_k **(Hz)**	**error (Hz)**
1	9	$k = \frac{500\times256}{15000} = 8.53$	$f_k = \frac{9\times15000}{256} = 527.34$	500	27.34
2	17	17.1	996.1	1000	3.9
3	34	34.13	1992.2	2000	7.8

Table 1: Peaks of the 256 DFT - Mixed sine wave

1.C) 256 DFT on a square wave

The third and last signal is a square wave with frequency of 500 Hz and amplitude from 0 to 20.

Since a square wave is composed of an infinite set of sinusoids[16] , creating one with a C program becomes much less practical and intuitive as opposed to generating a single sine wave with the $sin()$ function. In this case, the program tries to approximate the square wave with the algorithm:

$$x[n] = \begin{cases} 20, & \text{if } sin(2\pi n \times \frac{500}{15000}) \geq 0 \\ 0, & \text{if } sin(2\pi n \times \frac{500}{15000}) < 0 \end{cases}$$

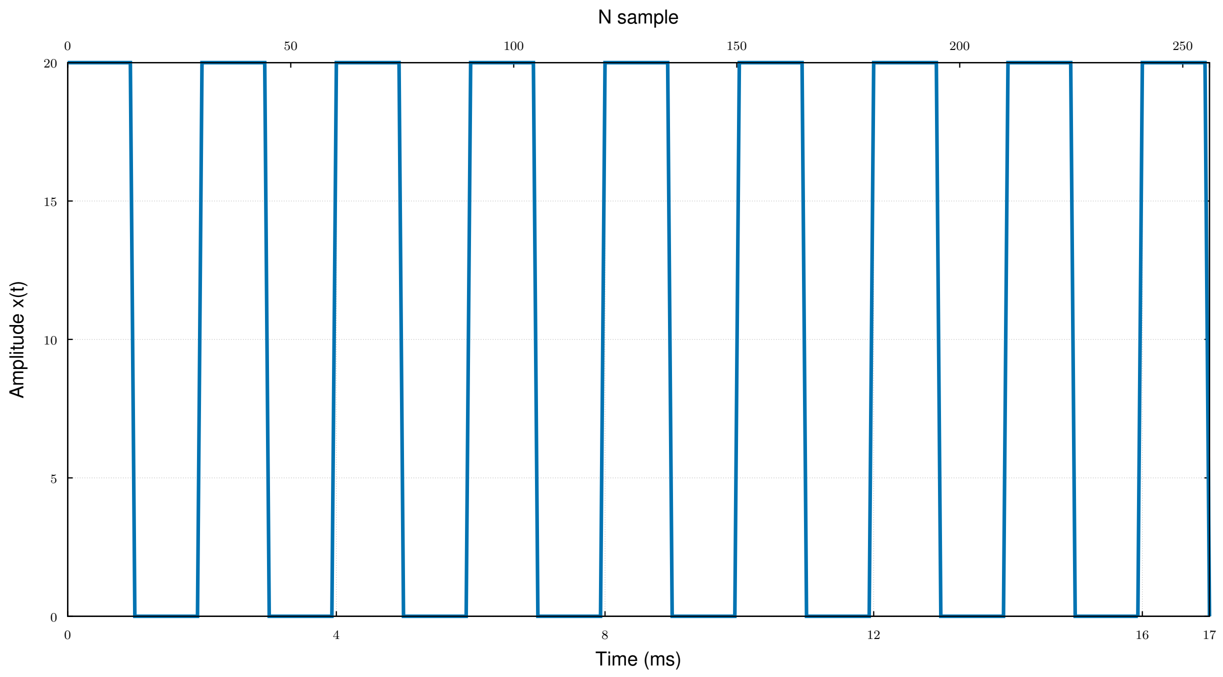


Figure 6: Square wave input [f = 0.5 kHz f_s = 15 kHz amplitude = 20 and 256 samples]

[16]Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, p. 49.

The code for it:

```c
float * create_squarewave(unsigned int size, float x_low, float x_high, float y_low, float y_high, float frequency, float sampling_frequency)
{
    /* Create square wave array */
    float * output = (float*)malloc(sizeof(float) * size);

    unsigned int n = 0;

    /* In this case, the square wave uses a sine wave function to toggle its
       amplitude when the sinusoid goes positive or negative. */
    for (float x = x_low; x < x_high && n < size; x++, n++)
        output[n] = sin(2 * PI * (frequency / sampling_frequency) * x) >= 0.0 ? y_high : y_low;

    return output;
}
```

Code 7: Function: create_squarewave() - dsp_functions.c

And how to use it:

```c
    /* Create input signal (square wave) */
    float * input_squarewave = create_squarewave(dft_points, 0, (float)dft_points, 0, SIGNAL_AMPLITUDE, SIGNAL_FREQUENCY, SIGNAL_SAMPLING_FREQUENCY);
```

Code 8: Usage: allocating a square wave - question1_c.c

Which gives the frequency spectrum:

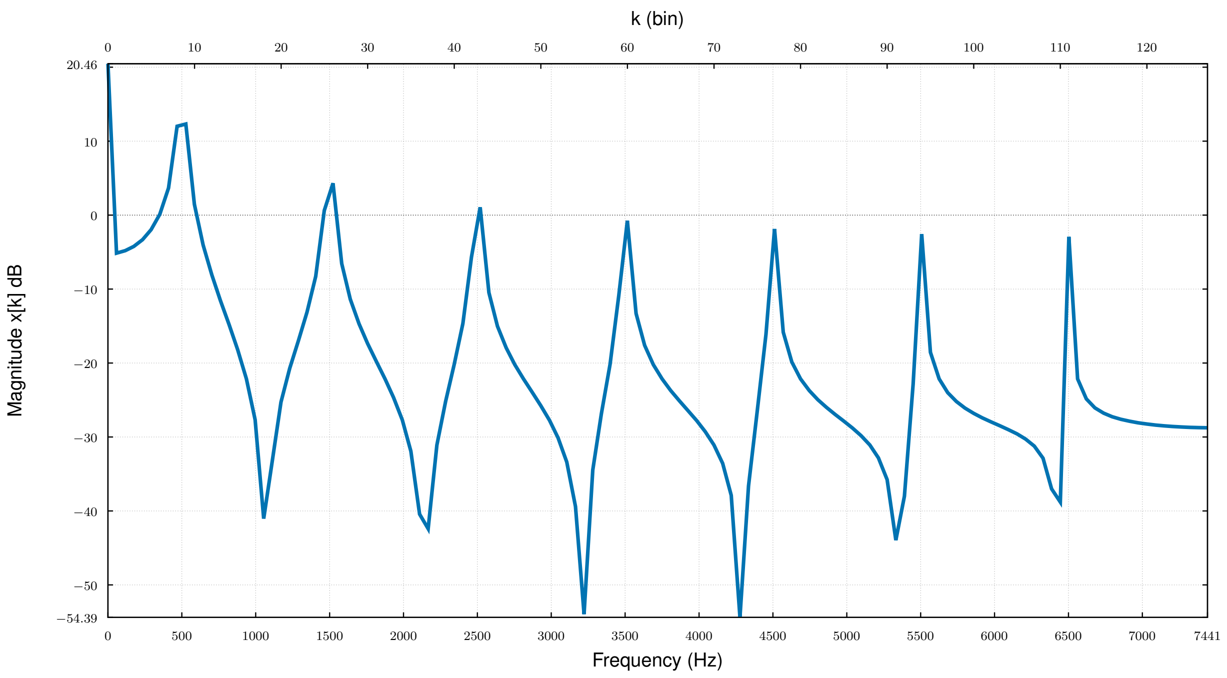


Figure 7: DFT spectrum [256 points SSB] of square wave [f = 0.5 kHz f_s = 15 kHz amplitude = [0, 20] and 256 samples] dB

After analysing the output, we quickly come to the conclusion that there is not only signal being fed through the input, but there are 8 apparent signals, one of them being purely DC at 0 Hz where $k = 0$ (which will be ignored while calculating f_k). As can be seen, this experiment proves that a square wave is indeed just a mix of infinite sinusoids with different frequencies, as described by the Fourier Series.

According to the plot, each peak is expected to be found at frequencies 0, 500, 1500, 2500, 3500, 4500, 5500 and 6500 Hz. The measured k bins are therefore:

Peak #	**k**	**expected k**	f_k **(Hz)**	**expected** f_k **(Hz)**	**error (Hz)**
1	9	$k = \frac{500\times256}{15000} = 8.53$	$f_k = \frac{9\times15000}{256} = 527.34$	500	27.34
2	26	25.6	1523.44	1500	23.44
3	43	42.67	2519.53	2500	19.53
4	60	59.73	3515.63	3500	15.63
5	77	76.8	4511.72	4500	11.72
6	94	93.87	5507.81	5500	7.81
7	111	110.93	6503.9	6500	3.91

Table 2: Peaks of the 256 DFT - Square wave

Also it's important to notice that the highest peak can be found at $k = 9$. This means that despite the fact that this specific square wave is composed of 7 main harmonics (excluding DC), we are still able to determine the actual desired frequency of the original signal, which is clearly indicated by the much larger magnitude at $k = 9$.

Part 2

In this part 2 the three input signals' spectral leakage will be mitigated by increasing the maximum number of frequency bins (k) from 256 to 512 and by using windowing, specifically the Hann window.

Finally, to conclude the report and to give more insight into the theory of DFT and Digital Signal Processing, the topic of windowing will be further developed by analysing and comparing two other windowing techniques against the Hann window.

2.1) 512 DFT on all three input signals

As was previously described in part 1, the number of points in a DFT determines the resolution of the frequency spectrum when the signal is analysed in the time domain. The higher the resolution is the lesser the spectrum leakage.

Before the results are shown for this exercise, some predictions must be made. If each DFT point is a multiple of the sampling frequency ($f_s = 15$ kHz) divided by the number of samples ($N = 256$), then each frequency bin should be associated with every multiple of the harmonic of frequency $f_k = k \times \frac{f_s}{N} = k \times \frac{15000}{256} = k \times 58.6$ Hz.

Any measured δf that is granular than this value will simply not be processed accurately by the DFT algorithm and will lead to spectral leakage. As a consequence, the output spectrum will contain non-zero magnitudes at frequency bins that should have no frequency content at all. Taking this into account, if we double the number of points (from 256 to 512) then the resolution should also double inversely and as a result spectral leakage should be much less impactful in our desired output. In addition, the frequency bins that should be at magnitude 0 will now be much closer to that value (as they should). So, for this new experiment, the new resolution for $N = 512$ will simply be $\frac{58.6}{2} = 29.3$ Hz, which is clearly better. (Another way to find out the resolution: $f_k = \frac{f_s}{N} = \frac{15000}{512} = 29.3$ Hz.)
Despite the benefits of increasing the DFT points, there is still the issue where k is rounded to the nearest integer. The error between adjacent k bins will always be present (as long as k is an integer).

With this in mind, the first input (2 kHz sine wave) shows a similar waveform as the original one with $N = 256$:

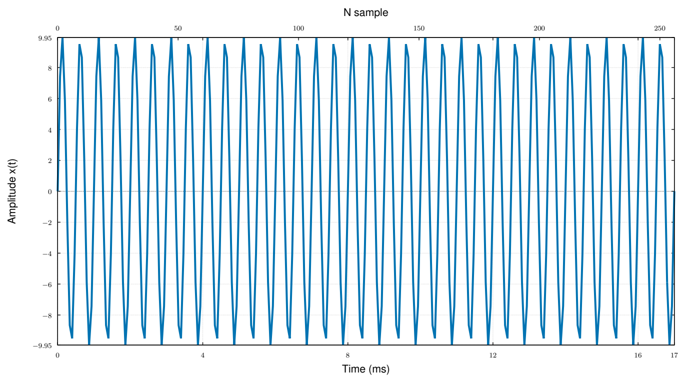


Figure 8: Sine wave of 256 samples

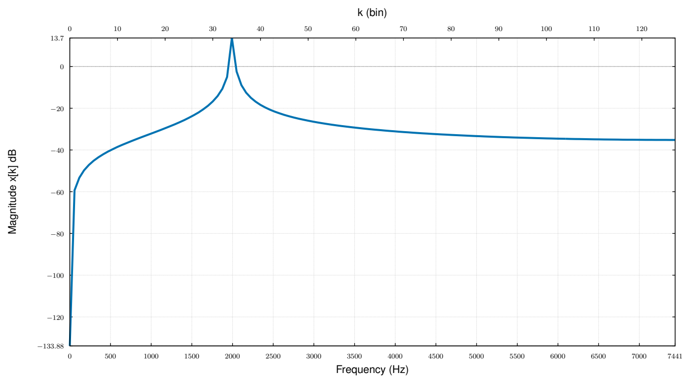


Figure 9: 256 DFT spectrum of 2 kHz sine wave (dB SSB)

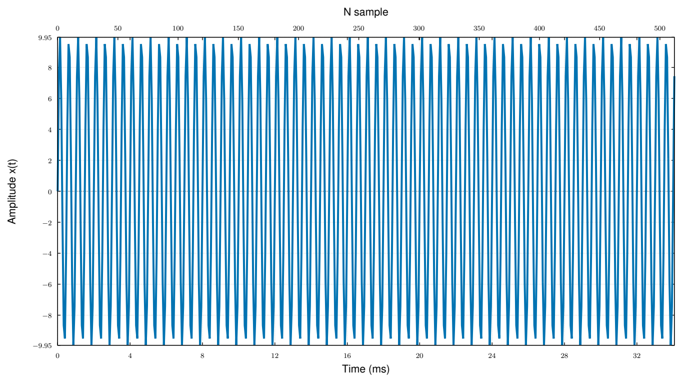


Figure 10: Sine wave of 512 samples

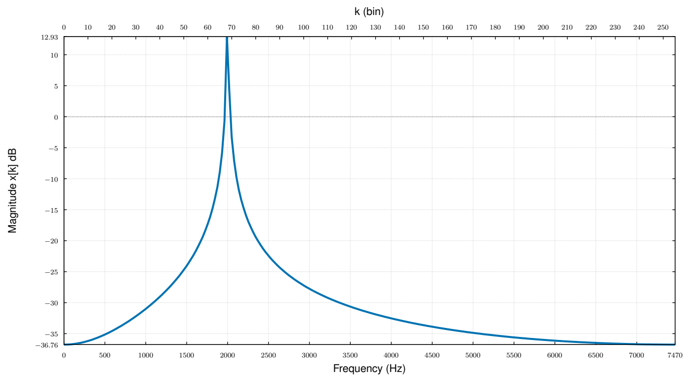


Figure 11: 512 DFT spectrum of 2 kHz sine wave (dB SSB)

The two plots seem to have very distinctive shapes. First, the magnitudes have reduced quite dramatically on both sides around the peak. Also, the peak has become narrower. Secondly, the measured k appears to be **68**, which is equivalent to $\mathbf{f_s} = \frac{\mathbf{68\times15000}}{\mathbf{512}} = \mathbf{1992.2}$ Hz with an error of $\mathbf{2000 - 1992.2} = 7.81$ **Hz**. Unfortunately, this error will always exist due to the fact that k is an integer.

This change in the output may be explained by the fact that the resolution has been improved from **58.6** Hz/bin to **29.3** Hz/bin (lower is better). Now, there is less correlation between $k \neq 68$ and even more correlation on the opposite condition ($k = 68$), which means increasing the number of DFT points will give a more accurate output. Windowing is a preferable alternative to this.

We may now compare the second signal (mix of three sine waves) with its 256 point DFT version.

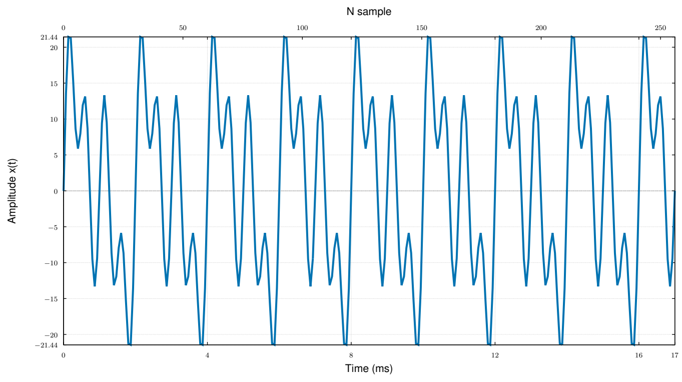


Figure 12: Mixed sine wave of 256 samples

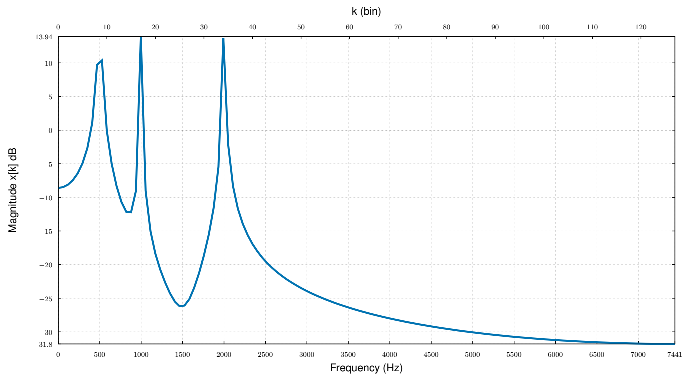


Figure 13: 256 DFT spectrum of mixed sine wave (dB SSB)

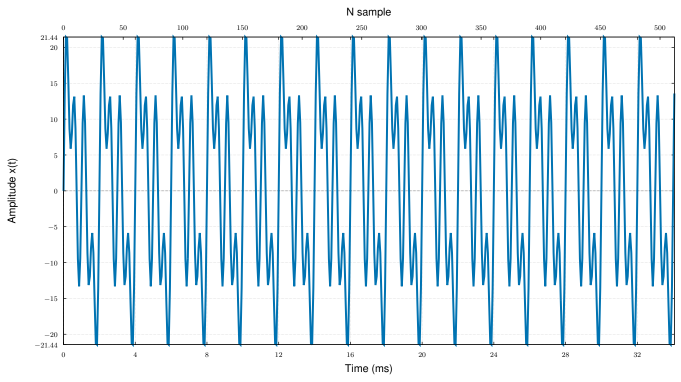


Figure 14: Mixed sine wave of 512 samples

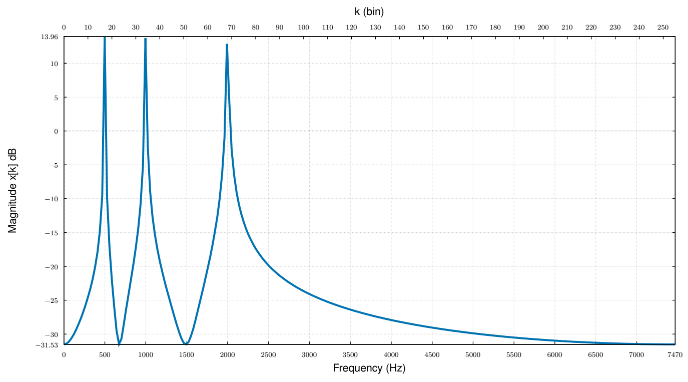


Figure 15: 512 DFT spectrum of mixed sine wave (dB SSB)

The results for the new frequency bins of the 512 DFT are:

Peak #	**k**	**expected k**	f_k **(Hz)**	**expected** f_k **(Hz)**	**error (Hz)**
1	17	$k = \frac{500\times512}{15000} = 17.07$	$f_k = \frac{17\times15000}{512} = 498.05$	500	1.95
2	34	34.13	996.1	1000	3.9
3	68	68.27	1992.2	2000	7.8

Table 3: Peaks of the 512 DFT - Mixed sine wave

And finally, the 500 Hz square wave.

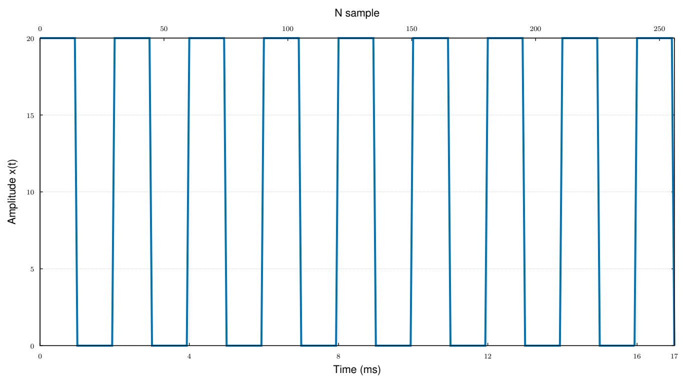


Figure 16: 500 Hz square wave of 256 samples

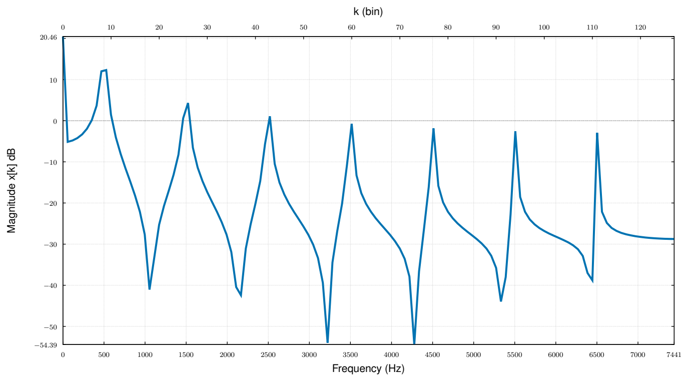


Figure 17: 256 DFT spectrum of 500 Hz square wave (dB SSB)

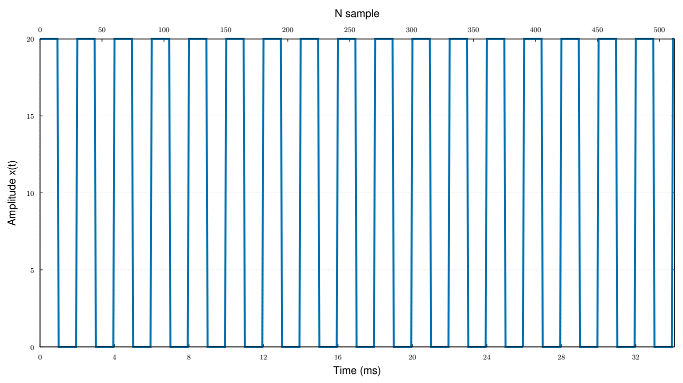


Figure 18: 500 Hz square wave of 512 samples

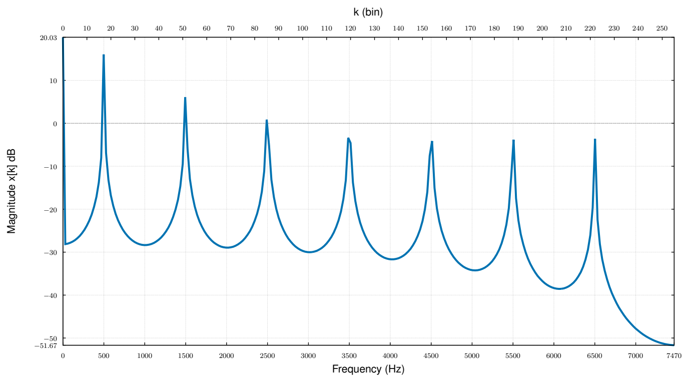


Figure 19: 512 DFT spectrum of 500 Hz square wave (dB SSB)

The peaks happen at k values:

Peak #	**k**	**expected k**	f_k **(Hz)**	**expected** f_k **(Hz)**	**error (Hz)**
1	17	$k = \frac{500\times512}{15000} = 17.07$	$f_k = \frac{17\times15000}{512} = 498.05$	500	1.95
2	51	51.2	1494.14	1500	5.9
3	85	85.3	2490.23	2500	9.8
4	119	119.5	3486.33	3500	13.7
5	154	153.9	4511.72	4500	11.72
6	188	187.7	5507.81	5500	7.81
7	222	221.9	6503.91	6500	3.91

Table 4: Peaks of the 512 DFT - Square wave

Before wrapping up, the k values and their frequency errors for each signal after changing the DFT from 256 to 512 should be compared:

	256 DFT					**512 DFT**			
	Peak #	**k**	**k expected**	$\mathbf{f_k}$	**error**	**k**	**k expected**	$\mathbf{f_k}$	**error**
Sine wave	1	34	34.13	1992.2	7.81	68	68.26	1992.2	7.81
Mix of three sine waves	1	9	8.53	527.34	27.34	17	17.07	498.05	1.95
	2	17	17.1	996.1	3.9	34	34.13	996.1	3.9
	3	34	34.13	1992.2	7.8	68	68.27	1992.2	7.8
Square wave	1	9	8.53	527.34	27.34	17	17.07	498.05	1.95
	2	26	25.6	1523.44	23.44	51	51.2	1494.14	5.9
	3	43	42.67	2519.53	19.53	85	85.3	2490.23	9.8
	4	60	59.73	3515.63	15.63	119	119.5	3486.33	13.7
	5	77	76.8	4511.72	11.72	154	153.6	4511.72	11.72
	6	94	93.87	5507.81	7.81	188	187.7	5507.81	7.81
	7	111	110.93	6503.91	3.91	222	221.9	6503.91	3.91

Table 5: 256 vs 512 DFT

The most obvious conclusion that can be taken from the table is that the larger error values have decreased quite dramatically. The small error values have, however, stayed the same.

For instance, we see that the error 27.34 Hz has decreased to 1.95 Hz, whereas the error 19.53 Hz has reduced only to 9.8 Hz, which indicates that there is a relationship between the error obtained and the resolution of the DFT. (Remembering that the resolution is now 29.3 Hz/bin.)

This means that the closer the measured error value is to that resolution, the better the result. Also, increasing the DFT point count will overall improve the error margin. For example, if the resolution is changed to 1 Hz/bin, then all the errors should be reduced to exactly 0 (given that there are no frequencies below 1), whereas 2 Hz/bin would almost give a perfect output except for odd frequencies. (e.g. harmonics $f = 1, f = 3, f = 5...$ would not be convoluted across the data samples and thus would lead to error.)

2.2) 512 DFT on all three input signals with Hann window

A second method to further improve the output spectrum is by applying a window function on the input prior to the DFT. The Hann window is a very common window and as such this task will apply the window through multiplication, analyse the results and compare them with the previous output spectrums.

$$w(n) = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$$

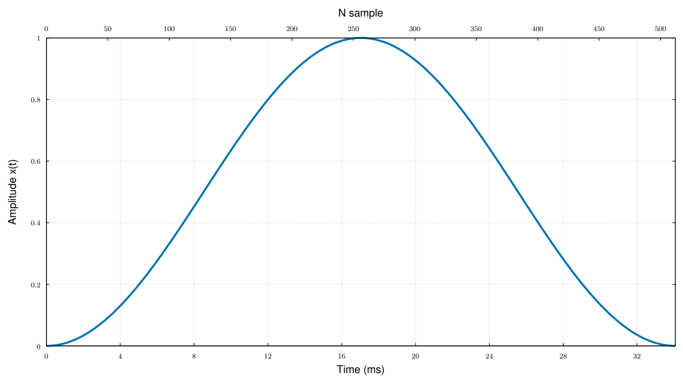


Figure 20: The Hann window

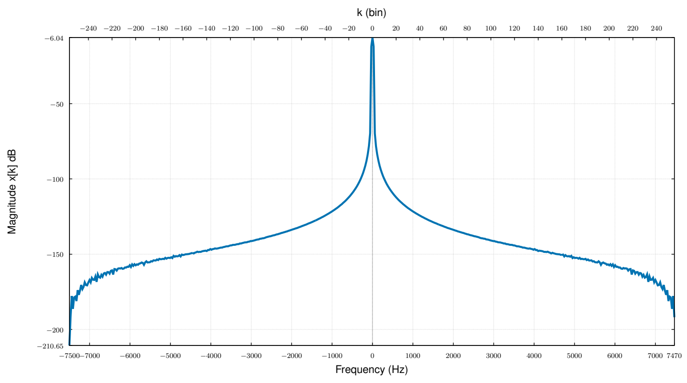


Figure 21: The Hann window in the frequency domain (dB Double-Sideband (DSB))

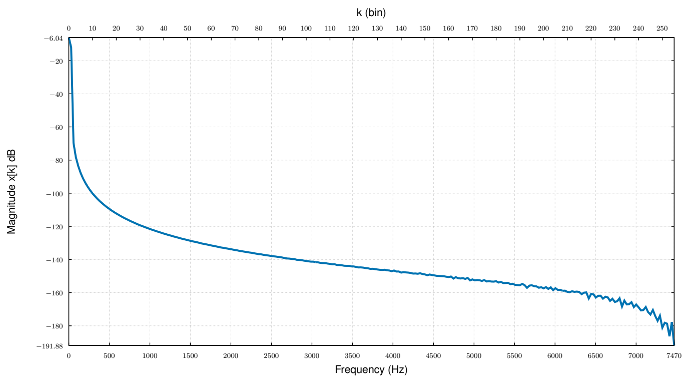


Figure 22: The Hann window in the frequency domain (dB Single-Sideband (SSB))

The generation code for this type of window is simply:

```c
float * create_window(unsigned int window_size, enum WINDOW_TYPE window_type)
{
    /* Create array for the window data */
    float * window = (float*)malloc(sizeof(float) * window_size);

    switch (window_type) {
        case WINDOW_HANN:
            /* Calculate each coefficient based on the
               Hann window formula and store it */
            for (unsigned int n = 0; n < window_size; n++)
                window[n] = (float)(0.5 * (1 - cos((2 * PI * n) / (window_size - 1))));
            break;
    }

    return window;
}
```

Code 9: Function: create_window() - dsp_functions.c

The main idea behind windowing is to remove part of the input signal at the start and at the end to prevent undesired high frequencies from contributing to the convolution of the DFT. This should result in a more accurate and narrower peak at the right frequency. Note, however, that this does not mitigate the frequency error that occurs when k is rounded to the nearest integer. This also means that windowing does not change the k bin frequencies. It simply makes the peaks narrower and reduces spectral leakage.[17]

Below is the result of multiplying/convoluting the input signal with the Hann window.

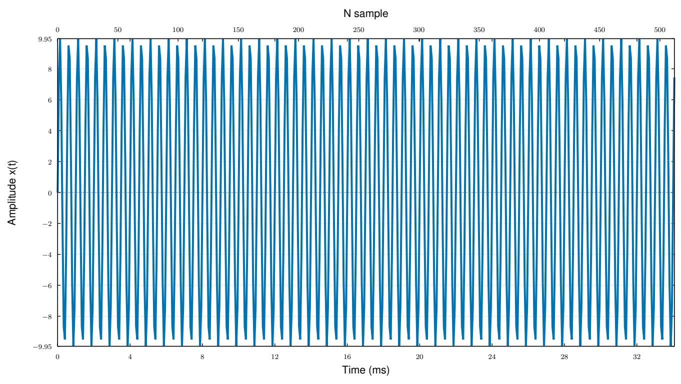


Figure 23: Sine wave of 512 samples without Hann window

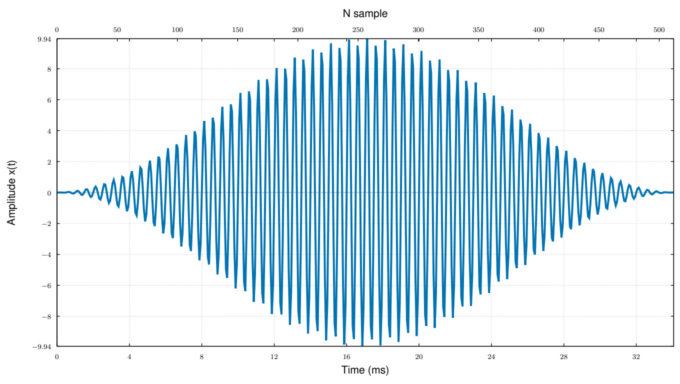


Figure 24: Sine wave of 512 samples with Hann window

[17]Richard G. Lyons. *Understanding Digital Signal Processing*. Prentice Hall, 1996, pp. 80–81.

Which has the following spectrum:

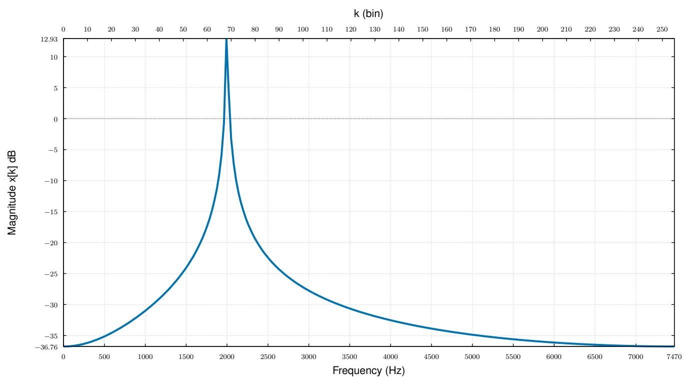


Figure 25: 512 DFT spectrum of sine wave without Hann window (dB SSB)

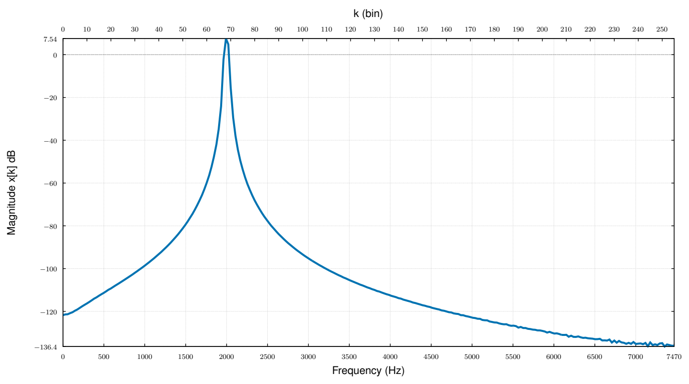


Figure 26: 512 DFT spectrum of sine wave with Hann window (dB SSB)

If we analyse the spectrum on the right correctly, we can see that there is indeed an improvement. The peak has grown shorter in magnitude (which is a disadvantage of windowing) but it has definitely become narrower, and this is without a doubt an improvement.

Applying the same window to the second input signal:

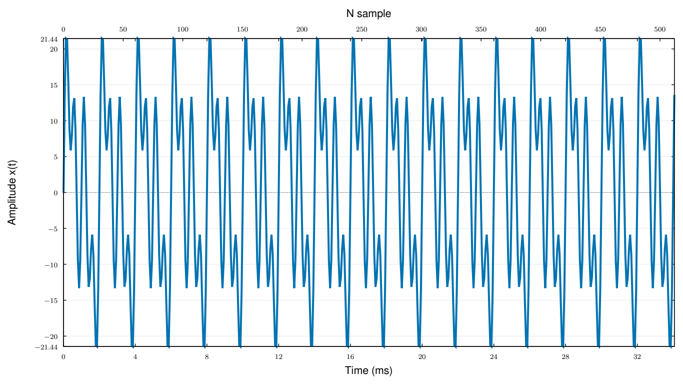


Figure 27: Mixed sine wave of 512 samples without Hann window

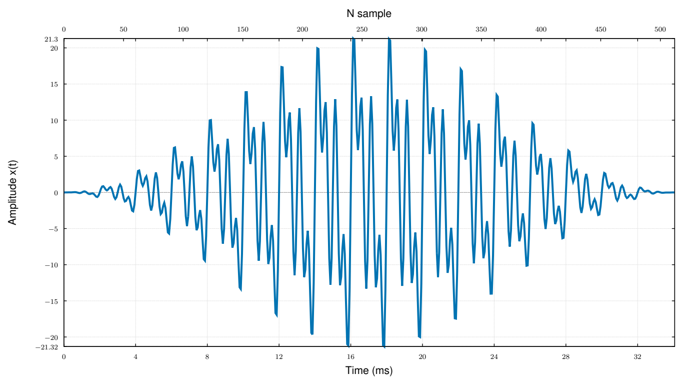


Figure 28: Mixed sine wave of 512 samples with Hann window

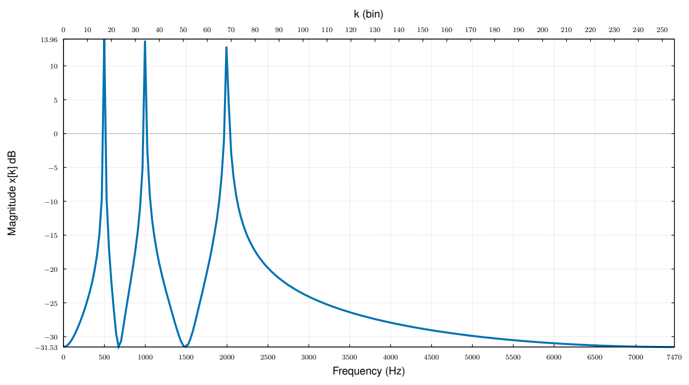


Figure 29: 512 DFT spectrum of mixed sine wave without Hann window (dB SSB)

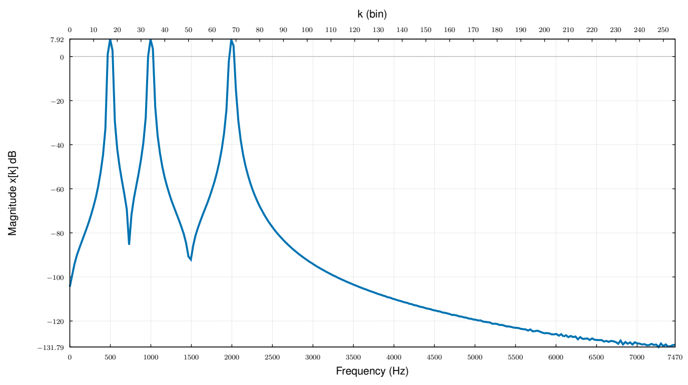


Figure 30: 512 DFT spectrum of mixed sine wave with Hann window (dB SSB)

Again, we observe an improvement over the previous spectrum. The magnitudes at frequencies between 500 and 1000 Hz (on the trough) have moved from -31.53 dB to ≈ -80 dB ≈ -90 dB. Moreover, the peaks became narrower but unfortunately, they have also reduced in magnitude.

Despite this disadvantage, they are still noticeable and it should be easy for a DSP system to retrieve the correct frequency information.

Finally, the third and last signal:

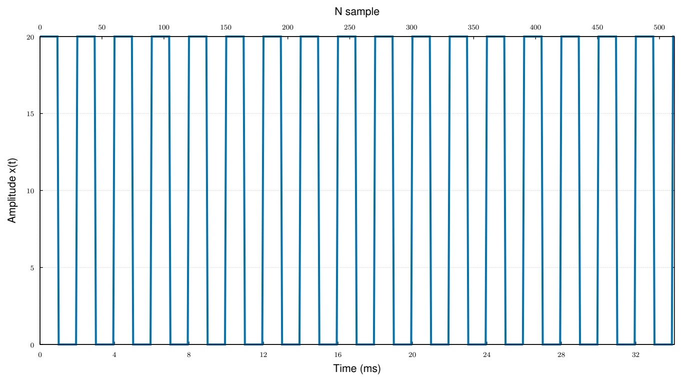


Figure 31: 500 Hz square wave of 512 samples without Hann window

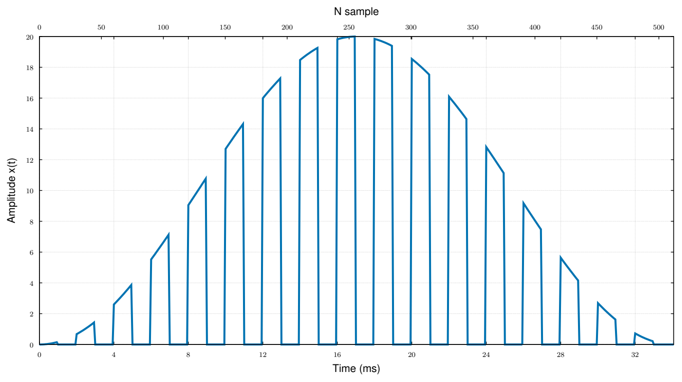


Figure 32: 500 Hz square wave of 512 samples with Hann window

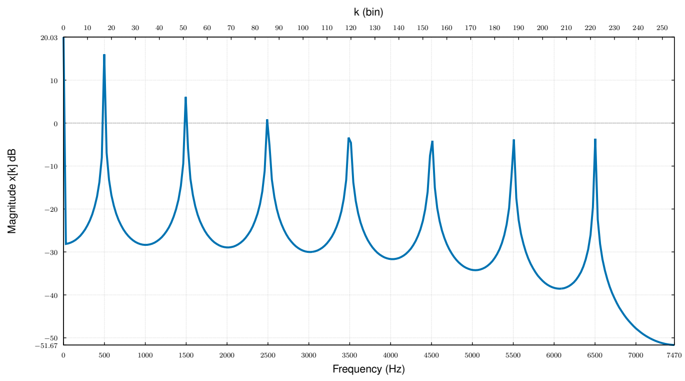


Figure 33: 512 DFT spectrum of square wave without Hann window (dB SSB)

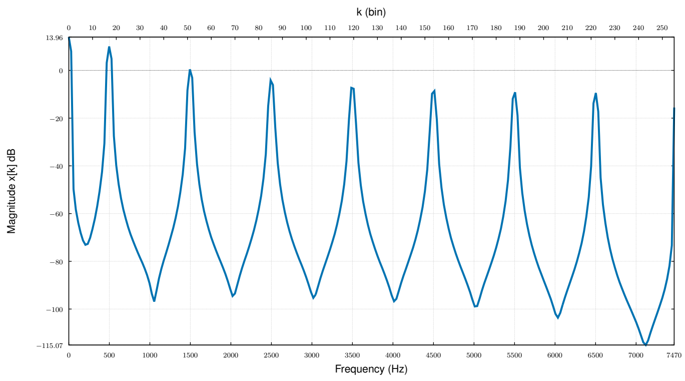


Figure 34: 512 DFT spectrum of square wave wave with Hann window (dB SSB)

Once more, all the frequency bins are in the same place, all peaks were narrowed and the magnitudes at the peaks were reduced only slightly compared to the troughs.

2.3) (Research) The Welch and Gaussian windows

There are many windowing methods used in DSP, each of which has its own application. Some windows are good for general cases while others are adjustable for a specific use. One simple example of such window (besides the Hann window) is the Welch window.[18]

$$w(n) = 1 - \left(\frac{n - \frac{N-1}{2}}{\frac{N-1}{2}} \right)^2$$

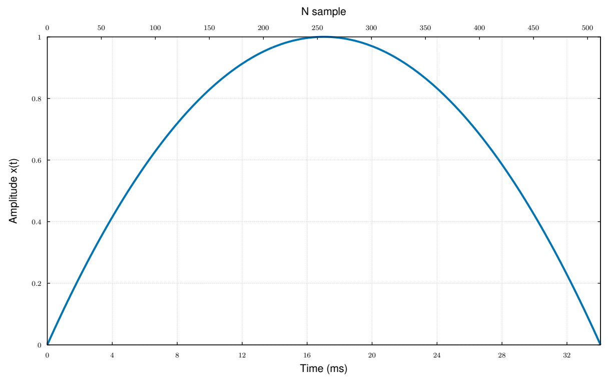


Figure 35: The Welch window

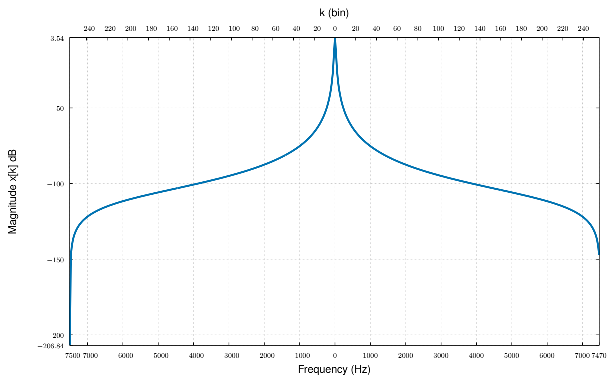


Figure 36: Welch window spectrum (dB DSB)

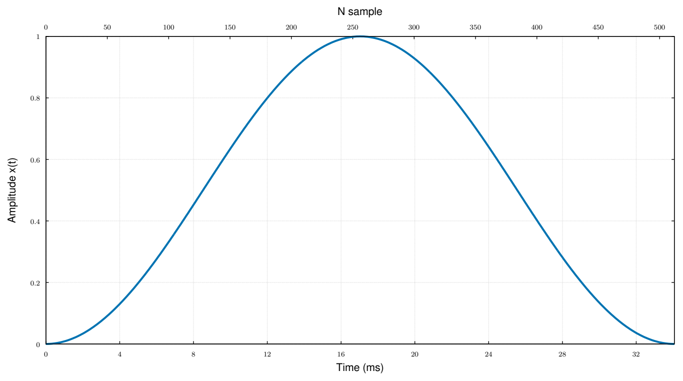


Figure 37: The Hann window

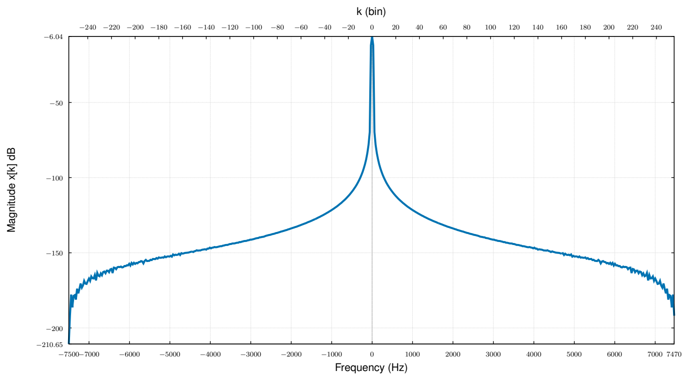


Figure 38: Hann window spectrum (dB DSB)

[18]Heinzel. G., A. Rüdiger, and R. Schilling. "Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows." In: (2002). URL: `http://edoc.mpg.de/395068`.

It takes a shape of a parabola, which when multiplied with the input signal gives the result:

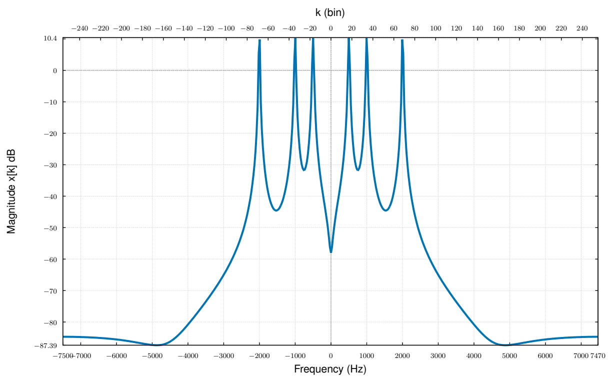


Figure 39: Mix sine sine wave spectrum with Welch window (dB DSB)

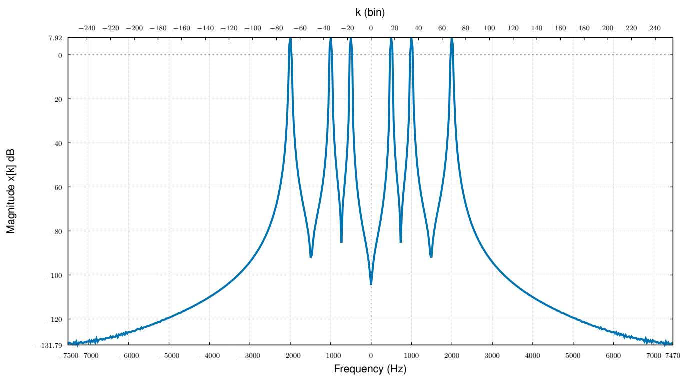


Figure 40: Mix sine wave spectrum with Hann Window (dB DSB)

One clear difference can be seen at the bottom edges of the Welch window spectrum where the curve is much smoother (figures 36 and 38). This seems to result in the input spectrum reaching zero magnitude (approximate) slightly earlier from $f = 2000$ Hz to 4500 Hz, however, as a trade-off, there seems to exist non-zero magnitudes from $f \geq 4750$ Hz on the spectrum.

Finally, the Gaussian window is an adjustable type of window that is simply defined by a standard Gaussian curve.[19]

$$w(n) = e^{-\frac{1}{2}\left(\frac{n-(N-1)/2}{\sigma\times(N-1)/2}\right)^2}$$

Where σ is the standard deviation which determines the width of the bell curve as shown below.

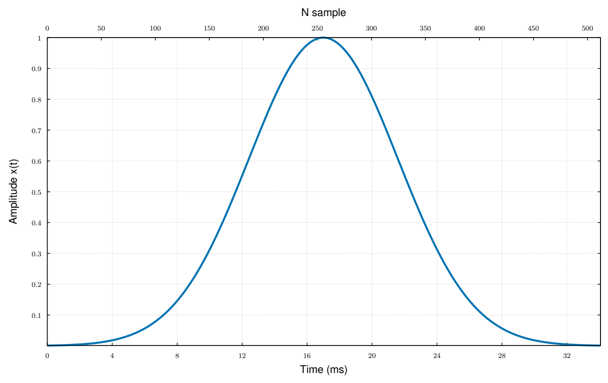


Figure 41: The Gaussian window

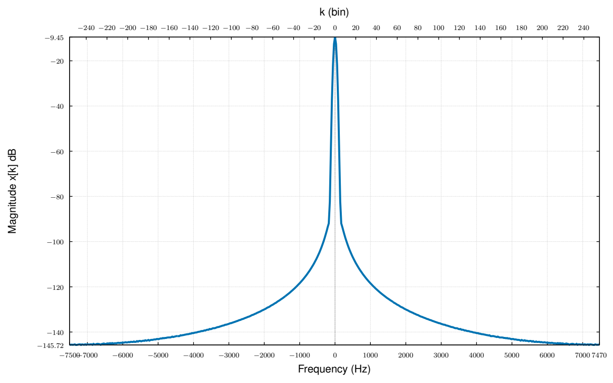


Figure 42: Gaussian window spectrum (dB DSB)

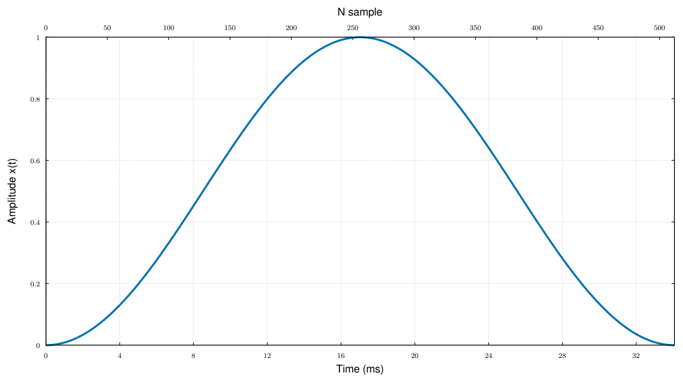


Figure 43: The Hann window

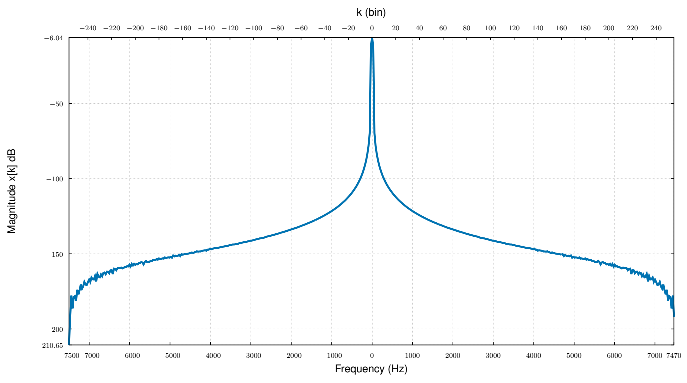


Figure 44: Hann window spectrum (dB DSB)

(In this case, standard deviation takes the value $\sigma = 0.27$.)

[19] Julius O. Smith. *Spectral Audio Signal Processing.* online book, 2011 edition. http://ccrma.stanford.edu/ jos/sasp//`ccrma.stanford.edu/~jos/sasp/`, accessed 11 December 2017.

After multiplying with the second input signal:

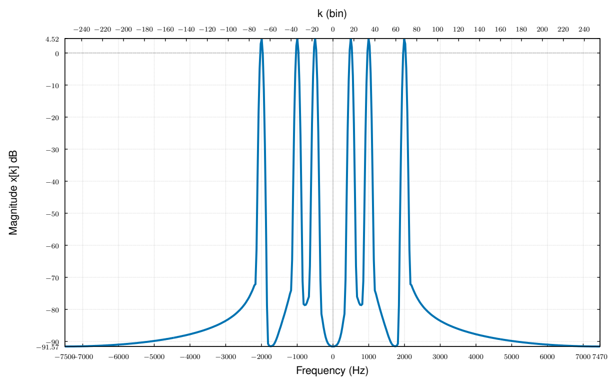


Figure 45: Mix sine sine wave spectrum with Gaussian window (dB DSB))

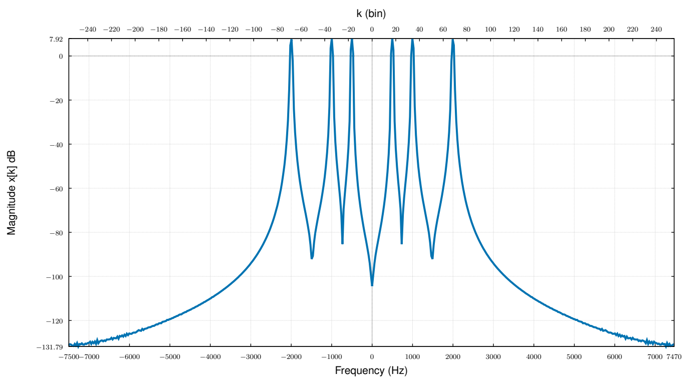


Figure 46: Mix sine wave spectrum with Hann Window (dB DSB)

The difference between the two is quite clear. Although the magnitudes at both ends have decreased from 7.92 dB to 4.52 dB for the peaks and -131.79 dB to -91.57 dB for the troughs, the shape of each peak has become longer and thinner. If the reduction in the peak magnitude is an undesired effect of this window, then it is still possible to change σ to a value higher (or lower) than 0.27. A disadvantage of changing this might be when σ goes below 0.26, where the shape of the peaks will begin to widen quite dramatically.

All in all, we may conclude that out of the three windows the Gaussian seems to be more appropriate. The main reason for this is the fact that it can adjusted for the desired application, whereas the Hann and Welch windows are fixed windows.

Summarising:

Window type	**Expression**	**Properties**	**Advantages**	**Disadvantages**
Hann	$w(n) = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right)$	Sinusoidal shape; Reaches 0 at both ends of the window	Low aliasing	Loss of coherence on main lobe (resolution is lost); Contains Passband ripples.
Welch	$w(n) = 1 - \left(\frac{n - \frac{N-1}{2}}{\frac{N-1}{2}}\right)^2$	Parabolic shape; Reaches 0 at both ends of the window	No Passband ripples	Input's outer side lobes have a higher (seemingly constant) magnitude compared to the Hann window; Troughs became smoother, which is undesirable.
Gaussian	$w(n) = e^{-\frac{1}{2}\left(\frac{n-(N-1)/2}{\sigma \times (N-1)/2}\right)^2}$	Bell shape; Reaches 0 at both ends of the window	Adjustable (Can be targeted for a specific application)	Peak magnitudes have reduced by a great amount compared to the Hann and Welch windows; Peaks become too wide when $\sigma < 0.26$.

Table 6: Window comparison: Hann vs Welch vs Gaussian

Conclusion

After analysing the three signals with the DFT algorithm we come to the conclusion that discrete signals simply cannot be processed without being affected by spectral leakage. It does not matter if the number of frequency bins is equal to the sampling frequency, there will always be non-zero magnitudes at frequency bins that should be at 0 (at the least in the practical sense). Despite this, the peaks were still recognizable and the spectrums displayed the correct frequencies at the right bin frequencies.

This leads to the conclusion that DFT is not supposed to be used in any general application. It needs to be configured specifically for a certain application, with all trade offs being considered. This also includes windowing, which has demonstrated to improve the output quite considerably.

That is the great benefit of DSP and mathematics in general: if a signal is not being processed adequately, just change the number of DFT points or select a different type of window, since there are so many. If no window fits the given application, then it's still possible to design a custom window, because after all, the DFT and Windowing functions are nothing more than just Z transforms and a set of coefficients.

References

[1] Heinzel. G., A. Rüdiger, and R. Schilling. "Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows." In: (2002). URL: `http://edoc.mpg.de/395068`.

[2] Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 48, 49.

[3] Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 184, 186.

[4] Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 56, 60.

[5] Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 62, 64.

[6] Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 56, 57.

[7] Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, p. 58.

[8] Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, pp. 55, 56.

[9] Emmanuel Ifeachor and Barrie Jervis. *Digital Signal Processing: A Practical Approach.* Ed. by Addison-Wesley. Prentice Hall, 1993, p. 49.

[10] Richard G. Lyons. *Understanding Digital Signal Processing.* Prentice Hall, 1996, pp. 54–63.

[11] Richard G. Lyons. *Understanding Digital Signal Processing.* Prentice Hall, 1996, p. 53.

[12] Richard G. Lyons. *Understanding Digital Signal Processing.* Prentice Hall, 1996, p. 50.

[13] Richard G. Lyons. *Understanding Digital Signal Processing.* Prentice Hall, 1996, pp. 71–80.

[14] Richard G. Lyons. *Understanding Digital Signal Processing.* Prentice Hall, 1996, pp. 80–81.

[15] Julius O. Smith. *Spectral Audio Signal Processing*. online book, 2011 edition. http://ccrma.stanford
`ccrma.stanford.edu/~jos/sasp/`, accessed 11 December 2017.

Appendices

Appendix A

Code

A.1 main.c

```c
#include <windows.h>
#include <conio.h>
#include <stdbool.h>
#include <stdio.h>

/* Function prototypes for the functions
   being developed on the assignment */
void question1_a(int dft_points, bool use_hann_window, bool use_double_sided_spectrum);
void question1_b(int dft_points, bool use_hann_window, bool use_double_sided_spectrum);
void question1_c(int dft_points, bool use_hann_window, bool use_double_sided_spectrum);

/* More function prototypes */
void goto_XY(int column, int line);
void print_menu(int cursor_position);
void run_question(int question, bool run_as_dsb);
int  get_adjusted_cursor_xcoord(int cursor_y_position);

#define QUESTION_COUNT  10
#define CURSOR_Y_OFFSET 8

int main(int argc, char ** argv)
{
    int cursor_position = 0;
    bool quit = false;

    print_menu(0);

    while(!quit) {
        bool arrow_pressed = true;
        char key = _getch();

        if (key == -32) {
            switch (_getch()) {
                case 72: /* Up arrow */
                    if(cursor_position > 0)
                        cursor_position--;
                    break;
                case 80: /* Down arrow */
                    if(cursor_position < QUESTION_COUNT)
                        cursor_position++;
                    break;
                case 75: /* Left arrow */
                    if(cursor_position == QUESTION_COUNT)
                        quit = true;
                    else
                        run_question(cursor_position, false);
                    break;
                case 77: /* Right arrow */
                    if (cursor_position == QUESTION_COUNT)
                        quit = true;
                    else
                        run_question(cursor_position, true);
                    break;
                default: arrow_pressed = false; break;
            }

            if(arrow_pressed) {
                printf("\b\b  ");
                goto_XY(get_adjusted_cursor_xcoord(cursor_position), cursor_position + CURSOR_Y_OFFSET);
                printf("<<");
            }
        } else if (key == 13) {
            /* Pressed ENTER */
            if (cursor_position == QUESTION_COUNT)
                quit = true;
            else
                run_question(cursor_position, false);
        } else if (key == 'q' || key == 'Q') {
            quit = true;
        }
    }

    return 0;
}

void goto_XY(int column, int line)
{
    COORD coord;
    coord.X = column;
    coord.Y = line;
    HANDLE consoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    CONSOLE_CURSOR_INFO info;
    info.dwSize = 100;
    info.bVisible = FALSE;
    SetConsoleCursorInfo(consoleHandle, &info);
    SetConsoleCursorPosition(consoleHandle, coord);
}

void print_menu(int cursor_position)
{
    printf("************************************************");
    printf("\n  NG4S804 - Applied Digital Signal Processing");
    printf("\n************************************************");
    printf("\n\t      (Assignment 1)\n        (14031329 - Miguel Santos)\n");
    printf("\n>> Select the question\n");
    printf("\n> Q1.A \t (256 DFT SSB)");
    printf("\n> Q1.B \t (256 DFT SSB)");
    printf("\n> Q1.C \t (256 DFT SSB)");
    printf("\n> Q2.1.A (512 DFT SSB)");
    printf("\n> Q2.1.B (512 DFT SSB)");
    printf("\n> Q2.1.C (512 DFT SSB)");
    printf("\n> Q2.2.A (512 DFT SSB + Hann)");
    printf("\n> Q2.2.B (512 DFT SSB + Hann)");
    printf("\n> Q2.2.C (512 DFT SSB + Hann)");
    printf("\n> Dump all graphs");
    printf("\n>> Quit");
    printf("\n\n(move arrow UP/DOWN to select, LEFT/ENTER to run question OR press RIGHT to run question as DSB (Double Sided Spectrum))\n");

    goto_XY(get_adjusted_cursor_xcoord(cursor_position), cursor_position + CURSOR_Y_OFFSET);
    printf("<<");
}

int get_adjusted_cursor_xcoord(int cursor_y_position)
{
    if     (cursor_y_position < QUESTION_COUNT - 4) return 23;
    else if(cursor_y_position < QUESTION_COUNT - 1) return 30;
    else if(cursor_y_position < QUESTION_COUNT)     return 18;
    else                                            return 8;
}

void run_question(int question, bool run_as_dsb)
{
    goto_XY(0, CURSOR_Y_OFFSET + QUESTION_COUNT + 5);
    switch (question) {
    case 0: printf(">> Executing question 1.a...\n");   question1_a(256, false, run_as_dsb); break;
    case 1: printf(">> Executing question 1.b...\n");   question1_b(256, false, run_as_dsb); break;
    case 2: printf(">> Executing question 1.c...\n");   question1_c(256, false, run_as_dsb); break;
    case 3: printf(">> Executing question 2.1.a...\n"); question1_a(512, false, run_as_dsb); break;
    case 4: printf(">> Executing question 2.1.b...\n"); question1_b(512, false, run_as_dsb); break;
    case 5: printf(">> Executing question 2.1.c...\n"); question1_c(512, false, run_as_dsb); break;
    case 6: printf(">> Executing question 2.2.a...\n"); question1_a(512, true,  run_as_dsb); break;
    case 7: printf(">> Executing question 2.2.b...\n"); question1_b(512, true,  run_as_dsb); break;
    case 8: printf(">> Executing question 2.2.c...\n"); question1_c(512, true,  run_as_dsb); break;
    case 9:
        printf(">> Executing all questions...\n");
        for (char i = 0; i <= 1; i++) {
            question1_a(256, false, (bool)i);
            question1_b(256, false, (bool)i);
            question1_c(256, false, (bool)i);
            question1_a(512, false, (bool)i);
            question1_b(512, false, (bool)i);
            question1_c(512, false, (bool)i);
            question1_a(512, true,  (bool)i);
            question1_b(512, true,  (bool)i);
            question1_c(512, true,  (bool)i);
        }
        break;
    default: break;
    }
    printf("\n>> Finished executing. Press any key to go back up to the menu");
    _getch();
    system("cls");
    print_menu(question);
}
```

Code A.1: main.c

A.2 dsp_functions.c

```c
#include <stdio.h>
#include "dsp_functions.h"

float * DFT(float * input, unsigned int input_size, unsigned int bin_count)
{
    /* In order to calculate the DFT, we'll create three signals /

    /* The real component of the DFT */
    float * dft_real = (float*)malloc(sizeof(float) * bin_count);
    /* The imaginary component */
    float * dft_imaginary = (float*)malloc(sizeof(float) * bin_count);
    /* And the magnitude between the two */
    float * dft_spectrum_output = (float*)malloc(sizeof(float) * bin_count);

    /* The frequency bins go to negative values -(N/2), so we'll calculate it here once */
    int bin = (bin_count / 2);

    for (int k = -bin, k_positive = 0; k < bin; k++, k_positive++) {
        /* Clear all magnitudes at this particular frequency bin */
        dft_real[k_positive]      = 0.0;
        dft_imaginary[k_positive] = 0.0;

        /* Add up all the coefficients multiplied with every single data sample */
        for (unsigned int n = 0; n < input_size; n++) {
            dft_real[k_positive]      += (float)(input[n] * cos(((2 * PI * k * n) / input_size)) / input_size);
            dft_imaginary[k_positive] += (float)(input[n] * sin(((2 * PI * k * n) / input_size)) / input_size);
        }

        /* And store the magnitude of the real and imaginary components */
        dft_spectrum_output[k_positive] = (float)((sqrt(pow(dft_real[k_positive], 2) + pow(dft_imaginary[k_positive], 2))));
    }

    /* We only care about the magnitude signal,
       so we'll cleanup the real and imaginary signals */
    free(dft_real);
    free(dft_imaginary);

    /* And finally return the DFT signal */
    return dft_spectrum_output;
}

float * create_sinewave(unsigned int size, float x_low, float x_high, float amplitude, float frequency, float sampling_frequency)
{
    /* Create sine wave array */
    float * output = (float*)malloc(sizeof(float) * size);

    unsigned int n = 0;

    /* Store sine wave data in the array using 'x' as an 'n' variable
       that can go negative and 'n' the positive index to that sample */
    for (float x = x_low; x < x_high && n < size; x++, n++)
        output[n] = (float)(amplitude * sin(2 * PI * (frequency / sampling_frequency) * x));

    return output;
}

float * create_squarewave(unsigned int size, float x_low, float x_high, float y_low, float y_high, float frequency, float sampling_frequency)
{
    /* Create square wave array */
    float * output = (float*)malloc(sizeof(float) * size);

    unsigned int n = 0;

    /* In this case, the square wave uses a sine wave function to toggle its
       amplitude when the sinusoid goes positive or negative. */
    for (float x = x_low; x < x_high && n < size; x++, n++)
        output[n] = sin(2 * PI * (frequency / sampling_frequency) * x) >= 0.0 ? y_high : y_low;

    return output;
}

float * create_window(unsigned int window_size, enum WINDOW_TYPE window_type)
{
    /* Create array for the window data */
    float * window = (float*)malloc(sizeof(float) * window_size);

    switch (window_type) {
        case WINDOW_HANN:
            /* Calculate each coefficient based on the
               Hann window formula and store it */
            for (unsigned int n = 0; n < window_size; n++)
                window[n] = (float)(0.5 * (1 - cos((2 * PI * n) / (window_size - 1))));
            break;
        case WINDOW_WELCH: {
            float a = (float)(window_size - 1) / 2;
            for (unsigned int n = 0; n < window_size; n++)
                window[n] = (float)(1 - pow((n - a) / a, 2));
            break;
        }
        case WINDOW_GAUSSIAN: {
            float deviation = 0.27f;
            float a = (float)((window_size - 1) / 2);
            for (unsigned int n = 0; n < window_size; n++)
                window[n] = (float)(exp(-0.5f * pow((n-a)/(deviation * a), 2)));
            break;
        }
    }

    return window;
}

float * convert_to_log_scale(float * input, unsigned int input_size)
{
    /* Create array that will hold a copy of the input
       but with values in the log scale */
    float * input_copy = (float*)malloc(sizeof(float) * input_size);

    /* Convert the data */
    for(unsigned int n = 0; n < input_size; n++)
        input_copy[n] = (float)(20 * log10(input[n]));

    return input_copy;
}

float peaks_magnitude[20];
unsigned int peaks[20];
unsigned int peaks_ctr = 0;

void display_spectrum_stats(float * spectrum, unsigned int bin_count, unsigned int maximum_peak_count)
{
    /* Clean the peaks array first */
    for (unsigned int i = 0; i < 20; i++) {
        peaks[i] = 0;
        peaks_magnitude[i] = 0;
        peaks_ctr = 0;
    }

    /* Try to find as many peaks as possible */
    for (unsigned int i = 0; i < maximum_peak_count; i++) {
        float max = spectrum[0];
        unsigned int max_idx = 0;

        for (unsigned int k = 0; k < bin_count; k++) {
            if (spectrum[k] > max) {
                bool is_peak_stored = false;
                for (unsigned int i = 0; i < peaks_ctr; i++)
                    if (peaks_magnitude[i] == spectrum[k])
                        is_peak_stored = true;

                if (!is_peak_stored) {
                    max = spectrum[k];
                    max_idx = k;
                }
            }
        }

        if (max != 0) {
            peaks_magnitude[peaks_ctr] = max;
            peaks[peaks_ctr++] = max_idx;
        }
    }

    printf("\n>> SIGNAL STATISTICS <<\n>> DFT bin count: %d\n>> Peaks found: %d", bin_count, peaks_ctr);

    for (unsigned int k = 0, peak_no = 1; k < bin_count / 2; k++) {
        for (unsigned int j = 0; j < peaks_ctr; j++) {
            if (peaks[j] == k) {
                printf("\n>>> Peak # %d: k = %d | magnitude = %.2f dB",
                    (maximum_peak_count - peak_no++) + 1,
                    (bin_count / 2) - k,
                    peaks_magnitude[j]
                );
            }
        }
    }

    printf("\n");
}
```

Code A.2: dsp_functions.c

A.3 dsp_functions.h

```c
#pragma once

#include <math.h>
#include <stdlib.h>
#include <stdbool.h>

#define PI 3.14159265358979323846

enum WINDOW_TYPE {
    NO_WINDOW,
    WINDOW_HANN,
    WINDOW_WELCH,
    WINDOW_GAUSSIAN
};

float * DFT(float * input, unsigned int input_size, unsigned int bin_count);
float * create_sinewave(unsigned int size, float x_low, float x_high, float amplitude, float frequency, float sampling_frequency);
float * create_squarewave(unsigned int size, float x_low, float x_high, float y_low, float y_high, float frequency, float sampling_frequency);
float * create_window(unsigned int window_size, enum WINDOW_TYPE window_type);
float * convert_to_log_scale(float * input, unsigned int input_size);
void display_spectrum_stats(float * spectrum, unsigned int bin_count, unsigned int maximum_peak_count);
```

Code A.3: dsp_functions.h

A.4 question1_a.c

```c
#include "dsp_functions.h"
#include "output_plot.h"

void question1_a(int dft_points, bool use_hann_window, bool use_double_sided_spectrum)
{
    /* Input signal properties */
    #define SIGNAL_AMPLITUDE          10
    #define SIGNAL_FREQUENCY          2000
    #define SIGNAL_SAMPLING_FREQUENCY 15000
    char input_name[128];
    sprintf(
        input_name, "%s - Sine Wave [f = %d Hz fs = %d amplitude = %d and %d points] %s(input)",
        dft_points == 256 ? (!use_hann_window ? "Q1A1" : "") : (!use_hann_window ? "Q21A1" : "Q22A1"),
        SIGNAL_FREQUENCY, SIGNAL_SAMPLING_FREQUENCY, SIGNAL_AMPLITUDE,
        dft_points, use_hann_window ? "with Hann window " : ""
    );

    /* Create input signal (sine wave) */
    float * input_sinewave = create_sinewave(
        dft_points,                /* The total size of the signal                                  */
        0,                         /* The starting point from which the signal begins (starts at x = 0) */
        (float)dft_points,         /* The end point at which the signal ends (ends at n = dft_points)   */
        SIGNAL_AMPLITUDE,          /* Amplitude of the signal                                       */
        SIGNAL_FREQUENCY,          /* Its frequency (Hz)                                            */
        SIGNAL_SAMPLING_FREQUENCY /* And finally, the sampling frequency (Hz) associated with it       */
    );

    /* Apply a Hann Window filter if the user specifies so */
    #define HANN_WINDOW_NAME          "HannWindow - Hann Window [512 points]"
    #define HANN_WINDOW_SPECTRUM_NAME "HannWindow%s - DFT spectrum of Hann Window [512 points %s]"
    char hann_window_name[128];
    char hann_window_spectrum_name_linear[128];
    char hann_window_spectrum_name_dB[128];
    float * hann_window             = NULL;
    float * hann_window_spectrum    = NULL;
    float * hann_window_spectrum_dB = NULL;

    if(use_hann_window) {
        sprintf(hann_window_name,                 HANN_WINDOW_NAME);
        sprintf(hann_window_spectrum_name_linear, HANN_WINDOW_SPECTRUM_NAME,       SSB_OR_DSB, SSB_OR_DSB);
        sprintf(hann_window_spectrum_name_dB,     HANN_WINDOW_SPECTRUM_NAME " dB", SSB_OR_DSB, SSB_OR_DSB);

        hann_window             = create_window(dft_points, WINDOW_HANN);
        hann_window_spectrum    = DFT(hann_window, dft_points, dft_points);
        hann_window_spectrum_dB = convert_to_log_scale(hann_window_spectrum, dft_points);

        /* Multiply input with window data */
        for(int n = 0; n < dft_points; n++)
            input_sinewave[n] *= hann_window[n];
    }

    /* Display to the console our input signal */
    display_signal(input_name, input_sinewave, dft_points);

    /* Output signal properties (the DFT spectrum) */
    char output_name[256];
    char output_name_linear[256];
    char output_name_dB[256];

    /* Format the plot filenames for each question and for the linear and dB formats */
    sprintf(
        output_name, "%s - DFT spectrum [%d points %s] of Sine Wave [f = %d Hz fs = %d Hz amplitude = %d and %d points] %s(output)",
        SSB_OR_DSB, dft_points, SSB_OR_DSB,
        SIGNAL_FREQUENCY, SIGNAL_SAMPLING_FREQUENCY, SIGNAL_AMPLITUDE,
        dft_points, use_hann_window ? "with Hann window " : ""
    );
    sprintf(output_name_linear, "%s%s",    dft_points == 256 ? (!use_hann_window ? "Q1A2" : "") : (!use_hann_window ? "Q21A2" : "Q22A2"), output_name);
    sprintf(output_name_dB,     "%s%s dB", dft_points == 256 ? (!use_hann_window ? "Q1A3" : "") : (!use_hann_window ? "Q21A3" : "Q22A3"), output_name);

    /* Perform DFT on the input signal */
    float * dft_spectrum_output    = DFT(input_sinewave, dft_points, dft_points);
    float * dft_spectrum_output_db = convert_to_log_scale(dft_spectrum_output, dft_points);
    display_signal(output_name_dB, dft_spectrum_output_db, dft_points);
    display_spectrum_stats(dft_spectrum_output_db, dft_points, 1);

    /* Output all the signals to a CSV file named 'Output.csv' */
    #define CSV_FILENAME "OutputData/Output.csv"

    /* If we are NOT using Hann window then we'll output 3 signals, otherwise, we output 6 (which includes the window) */
    unsigned int signal_output_count = !use_hann_window ? 3 : 6;

    export_to_csv(CSV_FILENAME, signal_output_count, SIGNAL_SAMPLING_FREQUENCY,
        /* For each signal we provide: the name, pointer, its size and the starting sample offset (always in this order) */
        input_name,                       input_sinewave,          dft_points, 0,
        output_name_linear,               dft_spectrum_output,     dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2,
        output_name_dB,                   dft_spectrum_output_db,  dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2,
        hann_window_name,                 hann_window,             dft_points, 0,
        hann_window_spectrum_name_linear, hann_window_spectrum,    dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2,
        hann_window_spectrum_name_dB,     hann_window_spectrum_dB, dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2
    );

    /* Display all plots and export them into .jpg images and .tex */
    show_plots();

    /* Cleanup signals from the heap */
    free(input_sinewave);
    free(dft_spectrum_output);
    free(dft_spectrum_output_db);

    if (use_hann_window) {
        free(hann_window);
        free(hann_window_spectrum);
        free(hann_window_spectrum_dB);
    }
}
```

Code A.4: question1_a.c

A.5 question1_b.c

```c
#include "dsp_functions.h"
#include "output_plot.h"

void question1_b(int dft_points, bool use_hann_window, bool use_double_sided_spectrum)
{
    /* Input signal properties */
    #define SIGNAL_AMPLITUDE          10
    #define SIGNAL_FREQUENCY1         500
    #define SIGNAL_FREQUENCY2         1000
    #define SIGNAL_FREQUENCY3         2000
    #define SIGNAL_SAMPLING_FREQUENCY 15000
    char input_name[256];
    sprintf(
        input_name, "%s - Mixed Sine Wave [f = %d Hz + %d Hz + %d Hz fs = %d Hz amplitude = %d and %d points] %s(input)",
        dft_points == 256 ? (!use_hann_window ? "Q1B1" : "") : (!use_hann_window ? "Q21B1" : "Q22B1"),
        SIGNAL_FREQUENCY1, SIGNAL_FREQUENCY2, SIGNAL_FREQUENCY3,
        SIGNAL_SAMPLING_FREQUENCY, SIGNAL_AMPLITUDE,
        dft_points, use_hann_window ? "with Hann window " : ""
    );

    /* Create 3 sinewaves of different frequencies and a 4th signal which will store the mixed result */
    float * sin1 = create_sinewave(dft_points, 0, (float)dft_points, SIGNAL_AMPLITUDE, SIGNAL_FREQUENCY1, SIGNAL_SAMPLING_FREQUENCY);
    float * sin2 = create_sinewave(dft_points, 0, (float)dft_points, SIGNAL_AMPLITUDE, SIGNAL_FREQUENCY2, SIGNAL_SAMPLING_FREQUENCY);
    float * sin3 = create_sinewave(dft_points, 0, (float)dft_points, SIGNAL_AMPLITUDE, SIGNAL_FREQUENCY3, SIGNAL_SAMPLING_FREQUENCY);
    float * sin_mixed = (float*)malloc(sizeof(float) * dft_points);

    /* Mix the 3 signals and display the result */
    for (int n = 0; n < dft_points; n++)
        sin_mixed[n] = sin1[n] + sin2[n] + sin3[n];

    /* Apply a Hann Window filter on the mixed signal if the user specifies so */
    #define HANN_WINDOW_NAME          "HannWindow - Hann Window [512 points]"
    #define HANN_WINDOW_SPECTRUM_NAME "HannWindow%s - DFT spectrum of Hann Window [512 points %s]"
    char hann_window_name[128];
    char hann_window_spectrum_name_linear[128];
    char hann_window_spectrum_name_dB[128];
    float * hann_window             = NULL;
    float * hann_window_spectrum    = NULL;
    float * hann_window_spectrum_dB = NULL;

    if (use_hann_window) {
        sprintf(hann_window_name,                 HANN_WINDOW_NAME);
        sprintf(hann_window_spectrum_name_linear, HANN_WINDOW_SPECTRUM_NAME,       SSB_OR_DSB, SSB_OR_DSB);
        sprintf(hann_window_spectrum_name_dB,     HANN_WINDOW_SPECTRUM_NAME " dB", SSB_OR_DSB, SSB_OR_DSB);

        hann_window             = create_window(dft_points, WINDOW_HANN);
        hann_window_spectrum    = DFT(hann_window, dft_points, dft_points);
        hann_window_spectrum_dB = convert_to_log_scale(hann_window_spectrum, dft_points);

        /* Multiply input with window data */
        for (int n = 0; n < dft_points; n++)
            sin_mixed[n] *= hann_window[n];
    }

    /* Display to the console our mixed input signal */
    display_signal(input_name, sin_mixed, dft_points);

    /* Output signal properties (the DFT spectrum) */
    char output_name[256];
    char output_name_linear[256];
    char output_name_dB[256];

    /* Format the plot filenames for each question and for the linear and dB formats */
    sprintf(
        output_name, "%s - DFT spectrum [%d points %s] of 3 mixed Sine Waves [f = %d Hz %d Hz %d Hz fs = %d Hz amplitude = %d and %d points] %s(output)",
        SSB_OR_DSB, dft_points, SSB_OR_DSB,
        SIGNAL_FREQUENCY1, SIGNAL_FREQUENCY2, SIGNAL_FREQUENCY3,
        SIGNAL_SAMPLING_FREQUENCY, SIGNAL_AMPLITUDE,
        dft_points, use_hann_window ? "with Hann window " : ""
    );
    sprintf(output_name_linear, "%s%s",    dft_points == 256 ? (!use_hann_window ? "Q1B12" : "") : (!use_hann_window ? "Q21B2" : "Q22B2"), output_name);
    sprintf(output_name_dB,     "%s%s dB", dft_points == 256 ? (!use_hann_window ? "Q1B13" : "") : (!use_hann_window ? "Q21B3" : "Q22B3"), output_name);

    /* Perform DFT on the input signal */
    float * dft_spectrum_output    = DFT(sin_mixed, dft_points, dft_points);
    float * dft_spectrum_output_db = convert_to_log_scale(dft_spectrum_output, dft_points);
    display_signal(output_name_dB, dft_spectrum_output_db, dft_points);
    display_spectrum_stats(dft_spectrum_output_db, dft_points, 3);

    /* Output all the signals to a CSV file named 'Output.csv' */
    #define CSV_FILENAME "OutputData/Output.csv"

    /* If we are NOT using Hann window then we'll output 3 signals, otherwise, we output 6 (which includes the window) */
    unsigned int signal_output_count = !use_hann_window ? 3 : 6;

    export_to_csv(CSV_FILENAME, signal_output_count, SIGNAL_SAMPLING_FREQUENCY,
        /* For each signal we provide: the name, pointer, its size and the starting sample offset (always in this order) */
        input_name,                       sin_mixed,               dft_points, 0,
        output_name_linear,               dft_spectrum_output,     dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2,
        output_name_dB,                   dft_spectrum_output_db,  dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2,
        hann_window_name,                 hann_window,             dft_points, 0,
        hann_window_spectrum_name_linear, hann_window_spectrum,    dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2,
        hann_window_spectrum_name_dB,     hann_window_spectrum_dB, dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2
    );

    /* Display all plots and export them into .jpg images and .tex */
    show_plots();

    /* Cleanup signals from the heap */
    free(sin1);
    free(sin2);
    free(sin3);
    free(sin_mixed);
    free(dft_spectrum_output);

    if (use_hann_window) {
        free(hann_window);
        free(hann_window_spectrum);
        free(hann_window_spectrum_dB);
    }
}
```

Code A.5: question1_b.c

A.6 question1_c.c

```c
#include "dsp_functions.h"
#include "output_plot.h"

void question1_c(int dft_points, bool use_hann_window, bool use_double_sided_spectrum)
{
    /* Input signal properties */
    #define SIGNAL_AMPLITUDE          20
    #define SIGNAL_FREQUENCY          500
    #define SIGNAL_SAMPLING_FREQUENCY 15000
    char input_name[128];
    sprintf(
        input_name, "%s - Square Wave [f = %d Hz fs = %d Hz amplitude = %d and %d points] %s(input)",
        dft_points == 256 ? (!use_hann_window ? "Q1C1" : "") : (!use_hann_window ? "Q21C1" : "Q22C1"),
        SIGNAL_FREQUENCY, SIGNAL_SAMPLING_FREQUENCY, SIGNAL_AMPLITUDE,
        dft_points, use_hann_window ? "with Hann window " : ""
    );

    /* Create input signal (square wave) */
    float * input_squarewave = create_squarewave(dft_points, 0, (float)dft_points, 0, SIGNAL_AMPLITUDE, SIGNAL_FREQUENCY, SIGNAL_SAMPLING_FREQUENCY);

    /* Apply a Hann Window filter if the user specifies so */
    #define HANN_WINDOW_NAME          "HannWindow - Hann Window [512 points]"
    #define HANN_WINDOW_SPECTRUM_NAME "HannWindow%s - DFT spectrum of Hann Window [512 points %s]"
    char hann_window_name[128];
    char hann_window_spectrum_name_linear[128];
    char hann_window_spectrum_name_dB[128];

    float * hann_window             = NULL;
    float * hann_window_spectrum    = NULL;
    float * hann_window_spectrum_dB = NULL;

    if (use_hann_window) {
        sprintf(hann_window_name,                 HANN_WINDOW_NAME);
        sprintf(hann_window_spectrum_name_linear, HANN_WINDOW_SPECTRUM_NAME,       SSB_OR_DSB, SSB_OR_DSB);
        sprintf(hann_window_spectrum_name_dB,     HANN_WINDOW_SPECTRUM_NAME " dB", SSB_OR_DSB, SSB_OR_DSB);

        hann_window             = create_window(dft_points, WINDOW_HANN);
        hann_window_spectrum    = DFT(hann_window, dft_points, dft_points);
        hann_window_spectrum_dB = convert_to_log_scale(hann_window_spectrum, dft_points);

        /* Multiply input with window data */
        for (int n = 0; n < dft_points; n++)
            input_squarewave[n] *= hann_window[n];
    }

    /* Display to the console our input signal */
    display_signal(input_name, input_squarewave, dft_points);

    /* Output signal properties (the DFT spectrum) */
    char output_name[256];
    char output_name_linear[256];
    char output_name_dB[256];

    /* Format the plot filenames for each question and for the linear and dB formats */
    sprintf(
        output_name, "%s - DFT spectrum [%d points %s] of Square Wave [f = %d Hz fs = %d Hz amplitude = %d and %d points] %s(output)",
        SSB_OR_DSB, dft_points, SSB_OR_DSB,
        SIGNAL_FREQUENCY, SIGNAL_SAMPLING_FREQUENCY, SIGNAL_AMPLITUDE,
        dft_points, use_hann_window ? "with Hann window " : ""
    );
    sprintf(output_name_linear, "%s%s",    dft_points == 256 ? (!use_hann_window ? "Q1C2" : "") : (!use_hann_window ? "Q21C2" : "Q22C2"), output_name);
    sprintf(output_name_dB,     "%s%s dB", dft_points == 256 ? (!use_hann_window ? "Q1C3" : "") : (!use_hann_window ? "Q21C3" : "Q22C3"), output_name);

    /* Perform DFT on the input signal */
    float * dft_spectrum_output    = DFT(input_squarewave, dft_points, dft_points);
    float * dft_spectrum_output_db = convert_to_log_scale(dft_spectrum_output, dft_points);
    display_signal(output_name_dB, dft_spectrum_output_db, dft_points);
    display_spectrum_stats(dft_spectrum_output_db, dft_points, 14);

    /* Output all the signals to a CSV file named 'Output.csv' */
    #define CSV_FILENAME "OutputData/Output.csv"

    /* If we are NOT using Hann window then we'll output 3 signals, otherwise, we output 6 (which includes the window) */
    unsigned int signal_output_count = !use_hann_window ? 3 : 6;

    export_to_csv(CSV_FILENAME, signal_output_count, SIGNAL_SAMPLING_FREQUENCY,
        /* For each signal we provide: the name, pointer, its size and the starting sample offset (always in this order) */
        input_name,                       input_squarewave,        dft_points, 0,
        output_name_linear,               dft_spectrum_output,     dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2,
        output_name_dB,                   dft_spectrum_output_db,  dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2,
        hann_window_name,                 hann_window,             dft_points, 0,
        hann_window_spectrum_name_linear, hann_window_spectrum,    dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2,
        hann_window_spectrum_name_dB,     hann_window_spectrum_dB, dft_points, use_double_sided_spectrum ? -dft_points / 2 : dft_points / 2
    );

    /* Display all plots and export them into .jpg images and .tex */
    show_plots();

    /* Cleanup signals from the heap */
    free(input_squarewave);
    free(dft_spectrum_output);

    if (use_hann_window) {
        free(hann_window);
        free(hann_window_spectrum);
        free(hann_window_spectrum_dB);
    }
}
```

Code A.6: question1_c.c

A.7 output_plot.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdbool.h>
#include <windows.h>
#include "output_plot.h"
#include "dsp_functions.h"

/* DISCLAIMER: This file is only used for outputting signals into .csv files.
   It does not contain any information regarding DSP theory. */

#define PLOTS_PATH "..\\..\\..\\LaTeX\\res\\plots"

char spectrum_side_type[2][4] = { "SSB", "DSB" };

bool dir_exists(const char* dirName_in)
{
    DWORD ftyp = GetFileAttributesA(dirName_in);
    if (ftyp == INVALID_FILE_ATTRIBUTES)
        return false;
    if (ftyp & FILE_ATTRIBUTE_DIRECTORY)
        return true;
    return false;
}

void display_signal(char * message, float * signal, unsigned int size)
{
    printf("\n****************** Signal: %s ******************\n", message);
    for (unsigned int n = 0; n < size; n++)
        printf("%.4f\n", signal[n]);
    printf("************************************\n");
}

void export_to_csv(char * csv_filename, unsigned int signal_count, unsigned int sampling_frequency, ...)
{
    char * filename = dir_exists("OutputData\\gnuplot") ? csv_filename : "OutputData.csv";
    FILE * csv_file = fopen(filename, "w");
    va_list va_args;

    char csv_line[256] = "";

    /* Print header cells into the CSV file */
    va_start(va_args, sampling_frequency);
    for (unsigned int i = 0; i < signal_count; i++) {
        char  * signal_name = va_arg(va_args, char*);
        va_arg(va_args, float*);
        va_arg(va_args, unsigned int);
        va_arg(va_args, int);

        fprintf(csv_file, "%s%s%s", i > 0 ? "k,f," : ",n,", signal_name, i < signal_count - 1 ? "," : "");
    }
    va_end(va_args);

    fprintf(csv_file, "\n");

    /* Print the entire sample data for all signals row by row */
    for (unsigned int sample_idx = 0, done_storing = 0; done_storing < signal_count; sample_idx++) {
        done_storing = 0;

        va_start(va_args, sampling_frequency);
        for (unsigned int i = 0; i < signal_count; i++) {

            /* Grab the signal pointers and its size */
            va_arg(va_args, char*);
            float *      signal            = va_arg(va_args, float*);
            unsigned int sample_count      = va_arg(va_args, unsigned int);
            int          sample_offset     = va_arg(va_args, int);
            int          sample_offset_idx = sample_offset < 0 ? 0 : sample_offset;

            int   bin_off       = sample_offset < 0 ? (int)(sample_count / 2) : 0;
            float frequency_off = sample_offset < 0 ? (float)(sampling_frequency / 2) : 0;

            if (sample_idx + sample_offset_idx < sample_count) {
                if(i > 0 && i != 3)
                    sprintf(csv_line, "%s%d,%.4f,%.4f%s", csv_line, sample_idx - bin_off, (float)((sample_idx * sampling_frequency) / sample_count - frequency_off), signal[sample_idx + sample_offset_idx], i < signal_count - 1 ? "," : "");
                else
                    sprintf(csv_line, "%s,%.4f,%.4f%s", csv_line, (float)((float)(sample_idx + sample_offset) / (sampling_frequency) * 1000), signal[sample_idx + sample_offset_idx], i < signal_count - 1 ? "," : "");
            } else {
                sprintf(csv_line, "%s,,%s", csv_line, i < signal_count - 1 ? "," : "");
                done_storing++;
            }
        }
        va_end(va_args);

        if (done_storing < signal_count)
            fprintf(csv_file, "%s\n", csv_line);

        csv_line[0] = '\0';
    }
    fclose(csv_file);

    printf("\n>> All signals were exported to the CSV file: '%s'", filename);
}

void show_plots(void)
{
    if(!dir_exists("OutputData\\gnuplot"))
        return;

    printf("\n>> Displaying plots...");
    system("mkdir res\\plots >nul 2>&1");
    system("OutputData\\gnuplot\\bin\\gnuplot.exe --persist OutputData\\gnuplot\\default.gnuplot");
    printf("\n>> Moving .tex plots into: '" PLOTS_PATH "'...");
    system("move /y res\\plots\\*.tex " PLOTS_PATH " >nul 2>&1");
    system("move /y res\\plots\\*.eps " PLOTS_PATH " >nul 2>&1");
    system("rmdir /s /q res");
    printf("\n>> Exported all plots into separate .jpg and .tex files...");
}
```

Code A.7: output_plot.c

A.8 output_plot.h

```c
#pragma once

#include <stdio.h>

void display_signal(char * message, float * signal, unsigned int size);
void export_to_csv(char * csv_filename, unsigned int signal_count, unsigned int sampling_frequency, ...);
void show_plots(void);

extern char spectrum_side_type[2][4];

#define SSB_OR_DSB (spectrum_side_type[use_double_sided_spectrum])
```

Code A.8: output_plot.h