Marta Szukalska

Advanced Transform Methods
Assignment 1
DFT, FFT and STFT

Introduction

The following report describe the implementation of DFT, FFT, and STFT in Matlab and analyse the results for different input signals. The representation of the signal in different domain is presented.

Discrete Fourier Transform

Discrete Fourier Transform is a method to get frequency representation of the signal. It can be applied to the signal using the Matlab function dft(st) presented below.

```
function sw = dft(st)      %Function takes as an input signal st to calculate its DFT.

M = length(st);            %Get the length of the input sequence.
N = M;                     % Define N as the length of sequence M.
WN = exp(2*pi*j/N);        %Calculate twiddle factor.

%The two for loops below calculate the DFT terms of the input sequence
%of length M for all terms of N.

for n=0:N-1
    temp = 0;  %The variable temp, is cleared to 0, so the sum for new
               %term can be calculated.

    for m=0:M-1

       temp = temp + (st(m+1)*(WN^(-m*n))); %DFT calculation.

    end

    sw(n+1) = temp; %The DFT values are stored in the new vector.


%Plot the real part of the sequence sw.
subplot(4,1,1);     stem(real(sw));      title('Real');
%Plot the imaginary part of the sequence sw.
subplot(4,1,2);     stem(imag(sw));      title('Imaginary');
%Plot the magnitude of the terms in sw.
subplot(4,1,3);     stem(abs(sw));       title('Abs');
%Plot the phase angle of the terms of sw.
subplot(4,1,4);     stem(angle(sw));     title('Angle');
end
```

The results of applying the DFT function to some waveforms are presented as follows.

- Uniform function: s = ones(1,64);

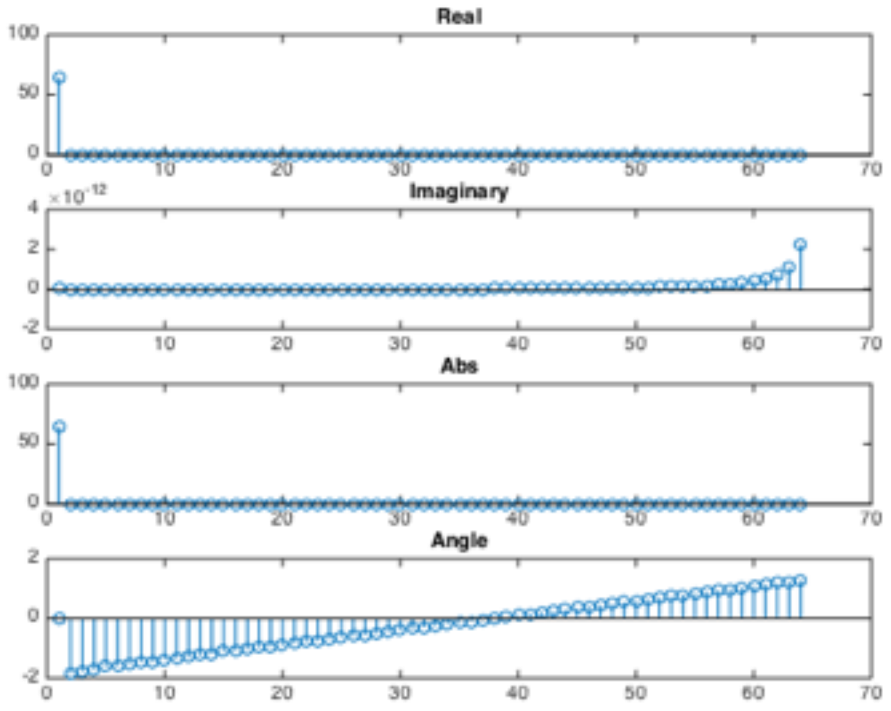


Figure 1: Uniform function DFT plots of real and imaginary parts, magnitude and phase angel.

- Delta Function s = ((1:64)==1);

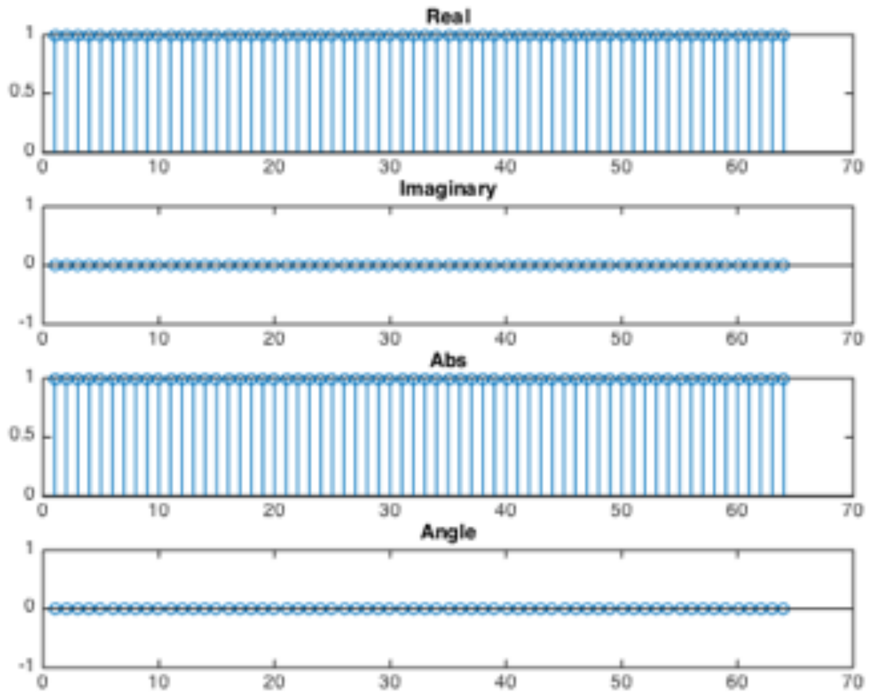


Figure 2: Delta Function DFT plots of real and imaginary parts, magnitude and phase angel.

- Sine Wave: s = sin(((1:64) -1)*2*pi*w/100); for different values of w.

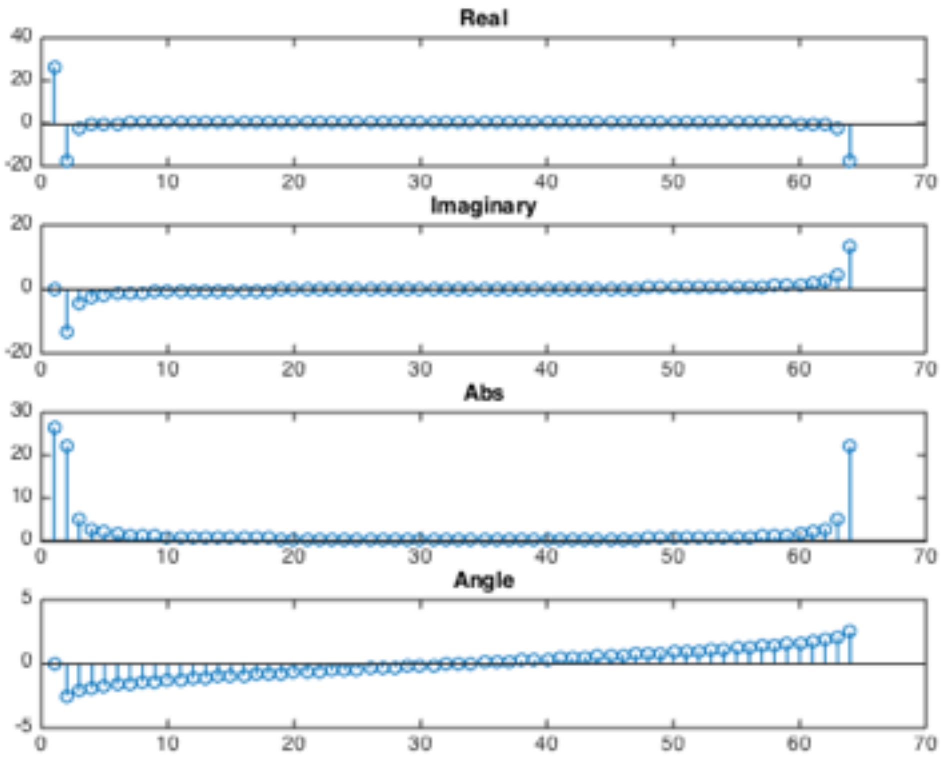


Figure 3: Sinusoid's DFT plots of real and imaginary parts, magnitude and phase angel for w=1.

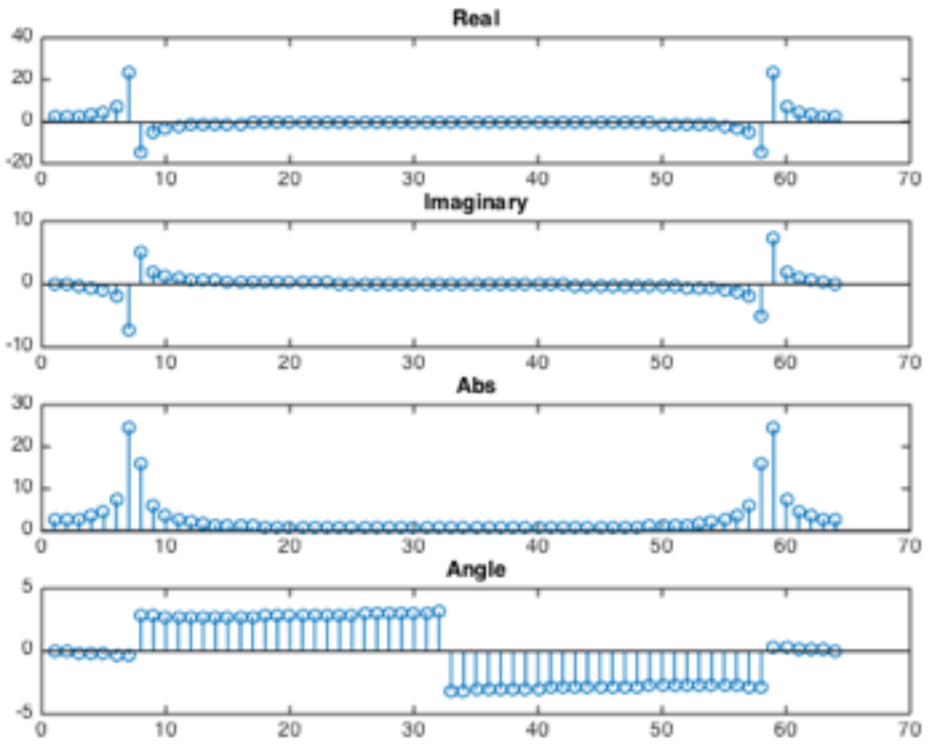


Figure 4: Sinusoid's DFT plots of real and imaginary parts, magnitude and phase angel for w=10.

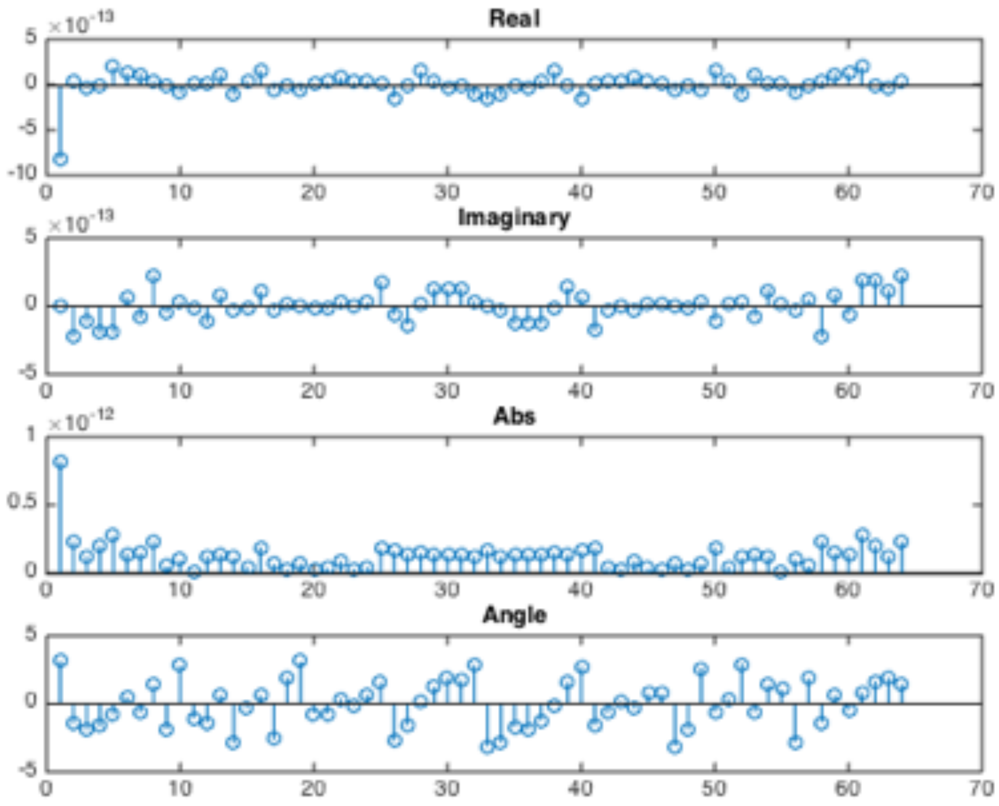


Figure 5: Sinusoid's DFT plots of real and imaginary parts, magnitude and phase angel for w=100.

- **cos** wave: s = cos(((1:64) -1)*2*pi*w/100); for different values of w.

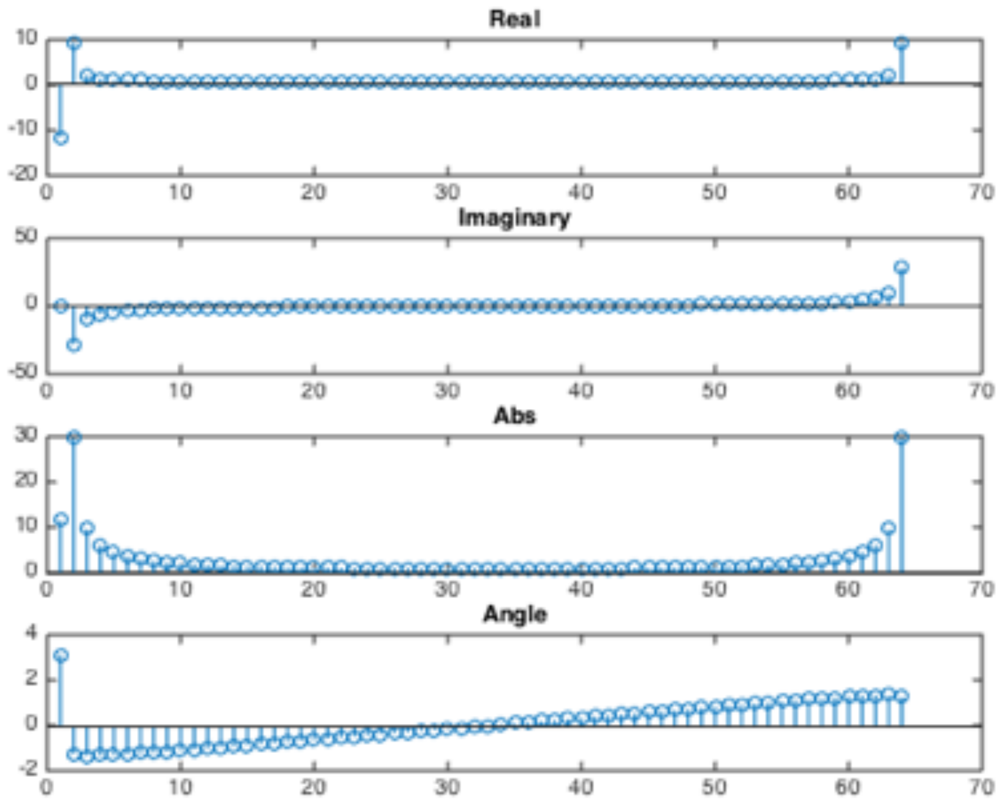


Figure 6: Cos wave's DFT plots of real and imaginary parts, magnitude and phase angel for w=1

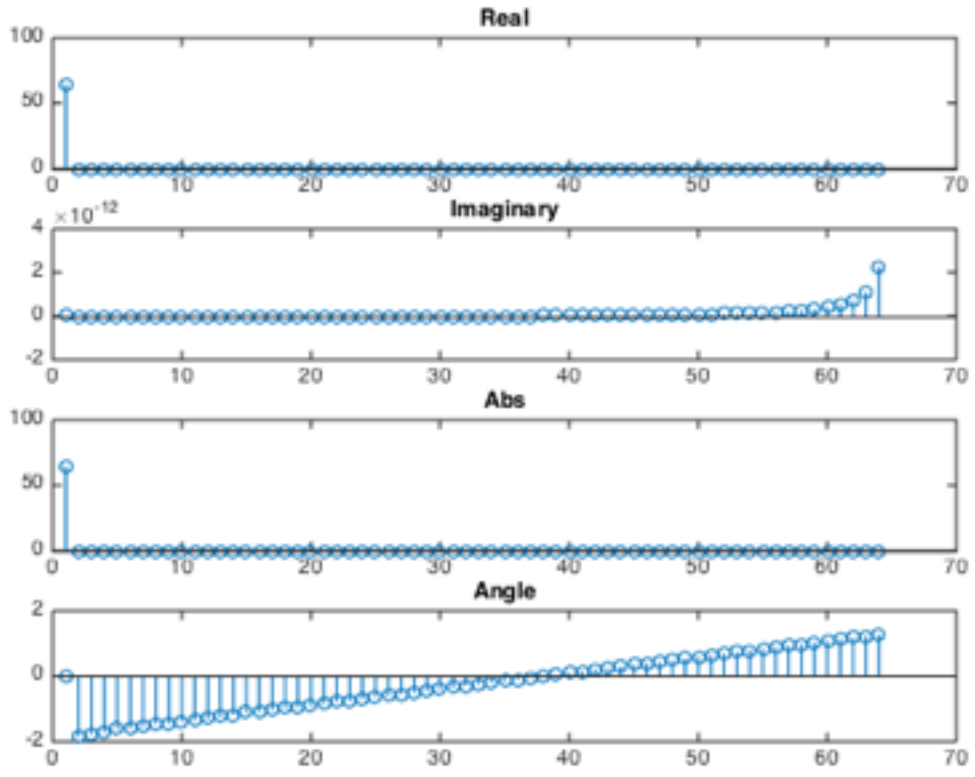


Figure 7: Cos wave's DFT plots of real and imaginary parts, magnitude and phase angel for w=100.

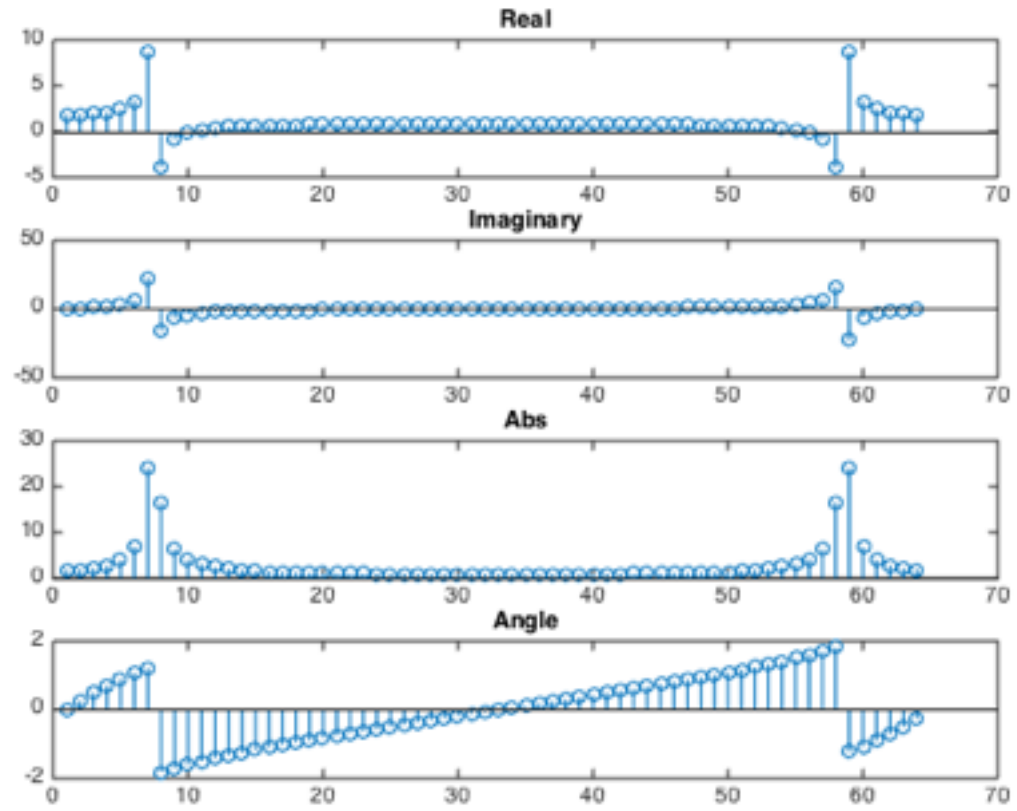


Figure 8: Cos wave's DFT plots of real and imaginary parts, magnitude and phase angel for w=10.

- Symmetrical rectangular pulse: s = [0:31 32: -1:1]<T;

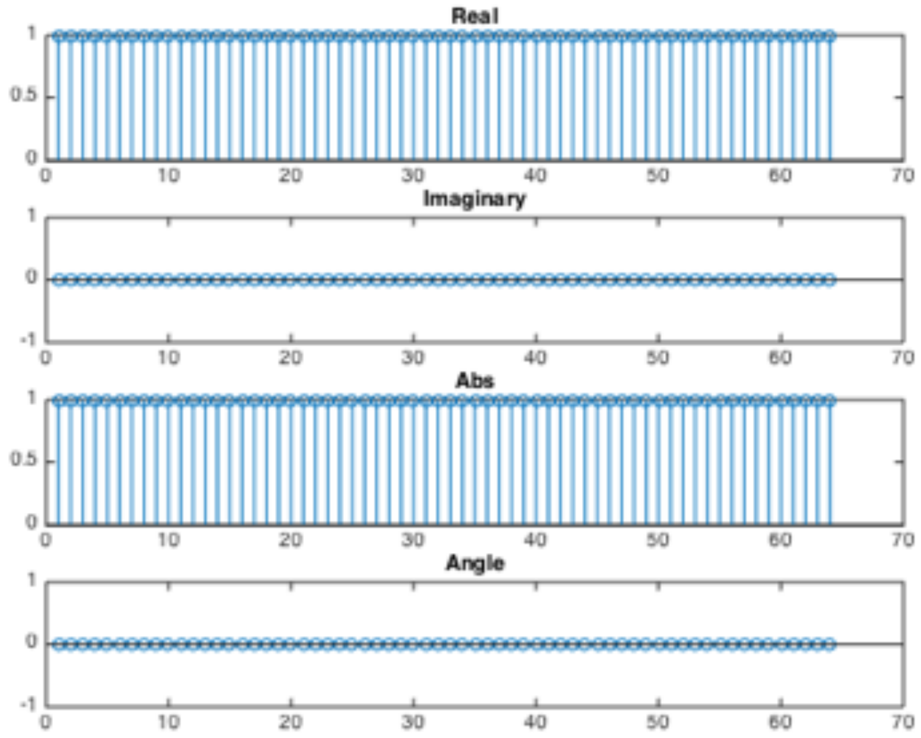


Figure 9: Symmetrical rectangular pulse's DFT plots of real and imaginary parts, magnitude and phase angel for t=1.

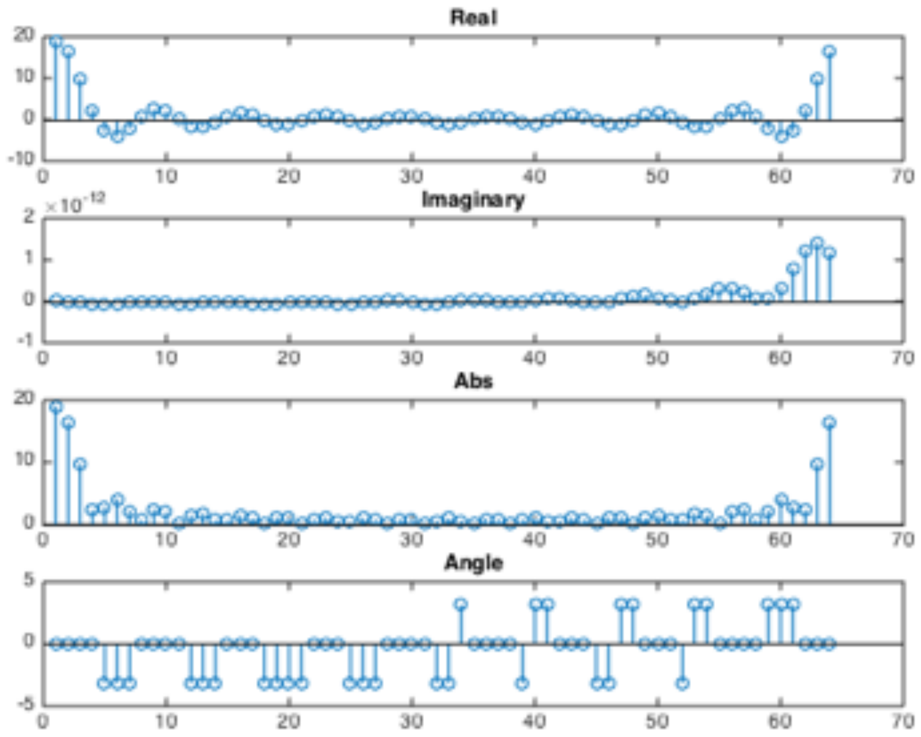


Figure 10: Symmetrical rectangular pulse's DFT plots of real and imaginary parts, magnitude and phase angel for t=10.

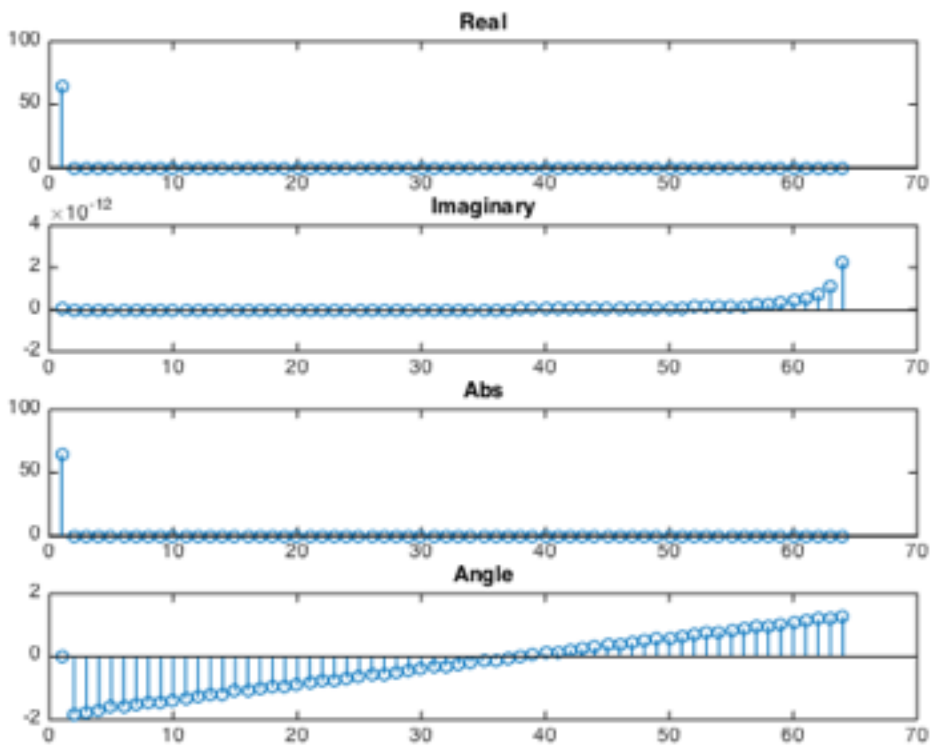


Figure 12: Symmetrical rectangular pulse's DFT plots of real and imaginary parts, magnitude and phase angel for t=100;

For the sine wave signal it is necessary to use (1:64)-1 as we want create the sine wave with the beginning at point 0, and it is known that Matlab is starting its vectors at point 1. That way from every element of the vector one is subtracted, what creates the sequence with values from 0 to 63.

It can be seen, that the best DFT of the sine wave can be achieved by using relatively large value of w(around 100).

Replacing sin with cos gives slight difference between the plots with the same w. There are some similarities and that is caused by the fact, that cos function can be represented as shifted sinusoid.

The symmetrical rectangular pulse is not symmetric, because the plots are not the representation of the input signal, but its different parameters such as: real and imaginary parts, magnitude and phase angle. Their asymmetry is caused by the fact that the pulse goes from 0 to 31 and then from 32 to 1. They have the same size but not the values.

There is as well another method to perform DFT in Matlab. As the software has its own build in function called fft, it is much easier to do. Below are the results.

- Uniform function: s = ones(1,64);

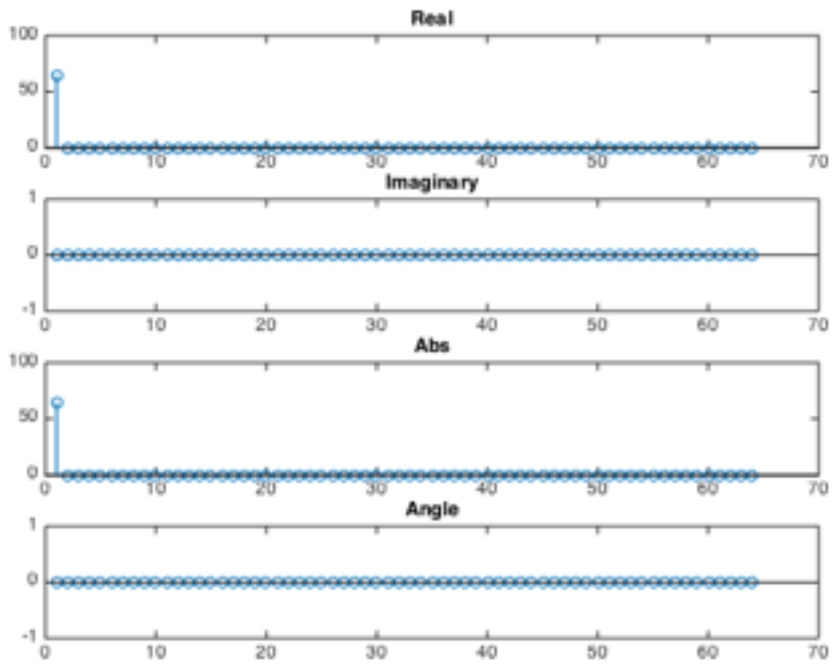


Figure 13: Uniform function FFT plots of real and imaginary parts, magnitude and phase angel.

- Delta Function s = ((1:64)==1);

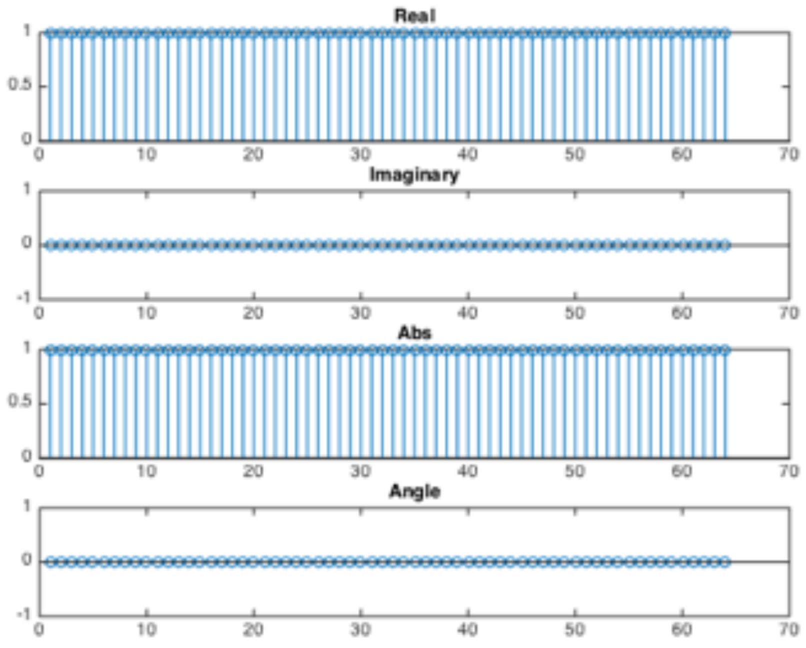


Figure 14: Delta Function FFT plots of real and imaginary parts, magnitude and phase angel.

- Sine Wave: s = sin(((1:64) -1)*2*pi*w/100); for different values of w.

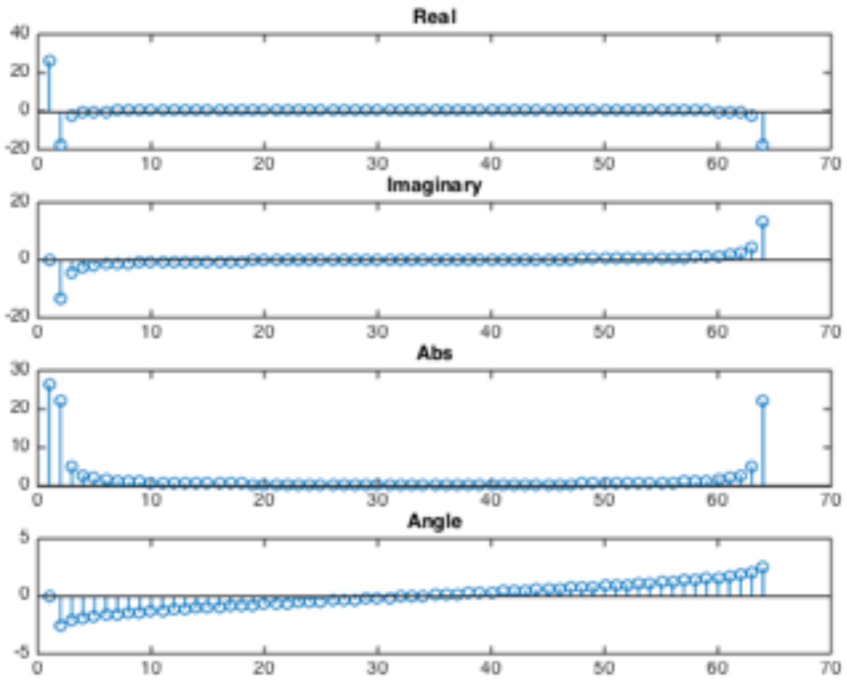


Figure 15: Sinusoid's FFT plots of real and imaginary parts, magnitude and phase angel for w=1.

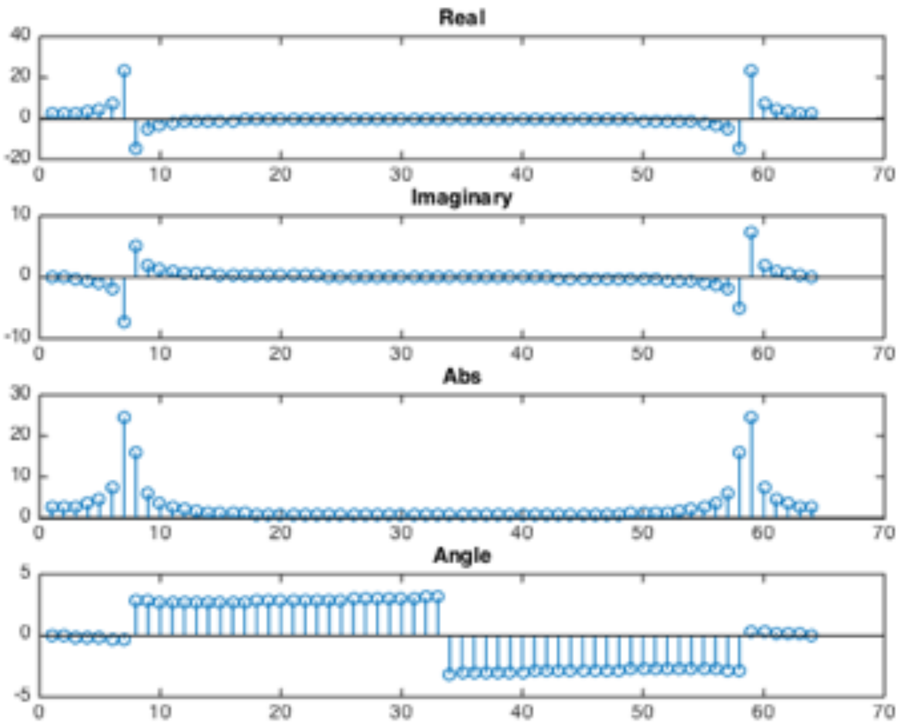


Figure 16: Sinusoid's FFT plots of real and imaginary parts, magnitude and phase angel for w=10.

- Symmetrical rectangular pulse: s = [0:31 32: -1:1]<T;

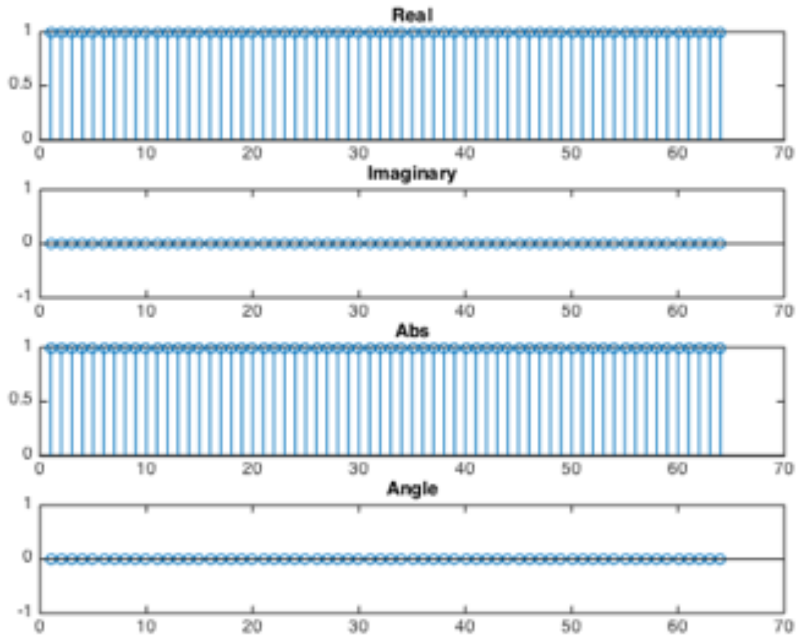


Figure 17: Symmetrical rectangular pulse's FFT plots of real and imaginary parts, magnitude and phase angel for t=1;

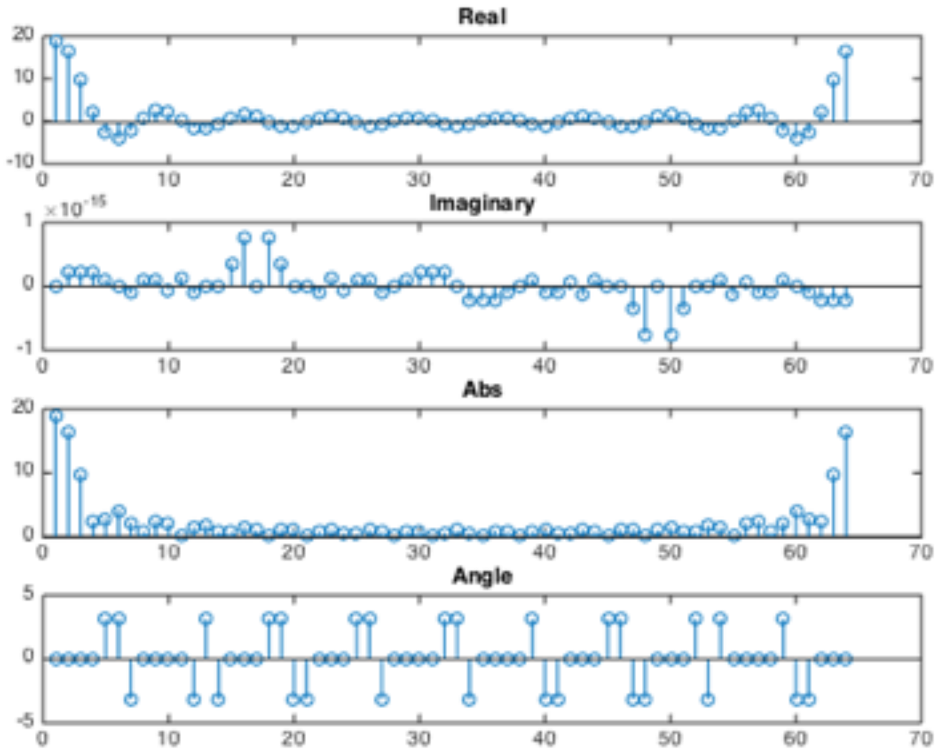


Figure 18: Symmetrical rectangular pulse's FFT plots of real and imaginary parts, magnitude and phase angel for t=10;

Comparing the DFT and FFT results almost all of them give the same output. The only difference is for the symmetrical rectangular pulse signal, where is the difference in imaginary part and phase angle or T bigger than 1.

One of the main differences between presented DFT and built-in FFT is time taken to perform the operation. To show the difference,the function below was used to measure the time of process for the sequences of length from 1 to 10000.

Code to measure the time of DFT:

```
function sw = timeco()

for d =1 : 10000           %For loop crate different length sequences.

st = ones(1,d);            %Create the sequence of ones.
tic;                       %Start calculate the time from here.

%The DFT calculation starts here.
M = length(st);
N = M;
WN = exp(2*pi*j/N);

       for n=0:N-1
       temp = 0;

              for m=0:M-1
              temp = temp + (st(m+1)*(WN^(-m*n)));
              end

       sw(n+1) = temp;

       end
%The DFT calculation ends here.
t(d) = toc;  %The new vector t(d) is created to store the time for
             %different length of sequences st.

end

loglog(t); %Plot the length of the sequence versus the time taken to calculate its DFT.
```

Code to measure the time of FFT:

```
function  fastft()

for d =1 : 10000          %For loop to create the different sequences with length from 1
                          %to 10000.

       st = ones(1,d);    %Define the sequence st of length 1 to d.
       tic;               %Start measuring the time of fft.
       b=fft(st);         %Calculate the fft.
       t(d) = toc;        %Create a vector with the time of fft for every sequence.
end

loglog(t);               %Plot the results in a log scale.
```

The output of the function for both FFT(red) and DFT(blue) is presented on the Figure 19.

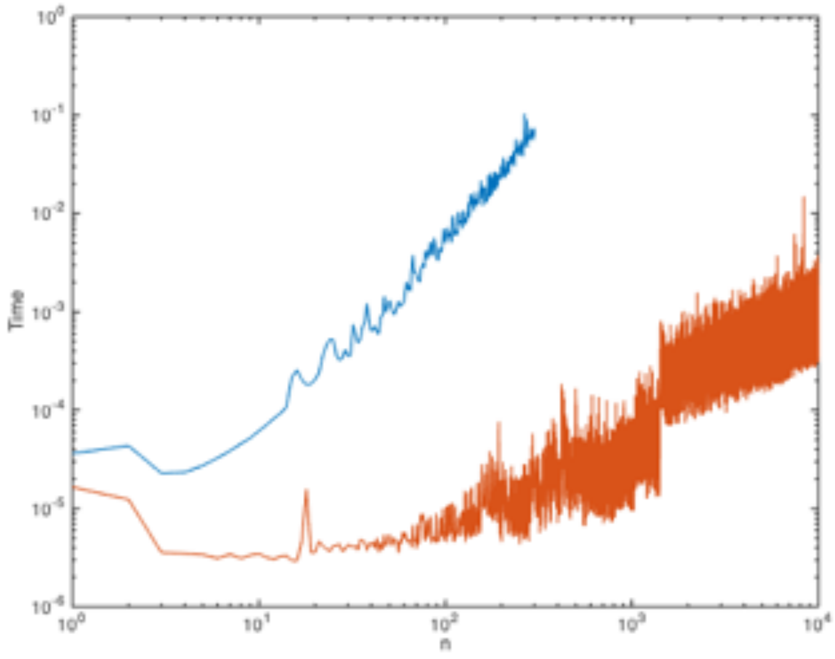


Figure 19: The time of performing DFT(blue plot) and FFT(red plot) on a signal plotted versus the length of the sequence in a logarithmic scale.

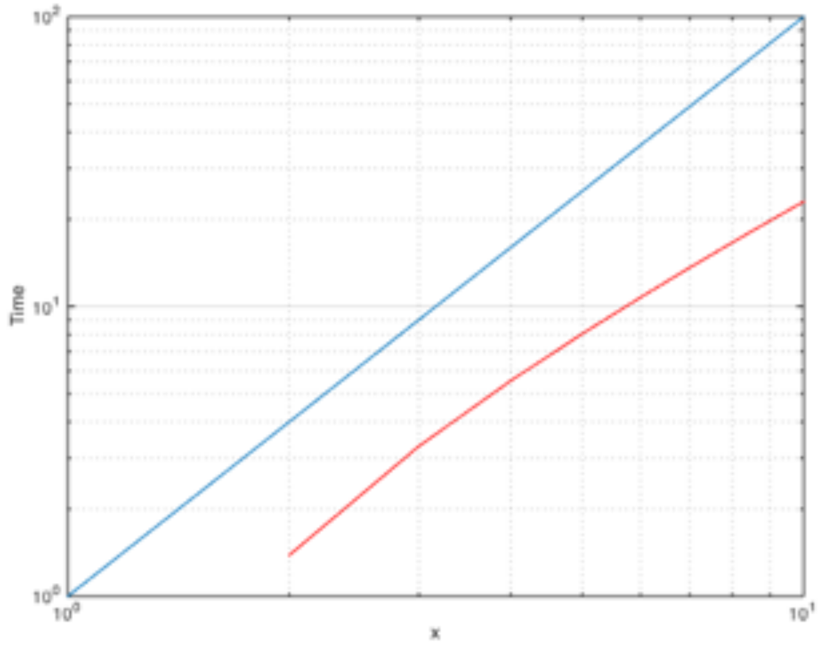


Figure 20: Graphical representation of two functions t = x*x (blue plot) and t = x*log(x) (red plot) in a log scale.

To prove, that the above time operations of DFT and FFT functions can be approximated to t(n*n) and t(n*log(n)) respectively the plot on the Figure 20 was generated.
The input sequence is x=[0,1,2,3,4,5,6,7,8,9,10]. For the blue plot the t terms were calculated by taking the squares of every element of x. The t terms of the red plot were

calculated by using a point wise multiplication x with log(x). As we can see the slopes are similar, the DFT plot is almost a straight line and the FFT plot has a kind of parabolic trend.

The fact that FFT operation time is defined as n*log(n) where n is the length of the sequence, it is obvious, and can be seen on the plots, that in the real life applications where there is necessity to operate on a really big n it is more sufficient to use the FFT not the DFT. As it does not make a significant difference for small values of n, it can save a lot of time and it is computationally less expensive to use FFT.

Single Windowed Fourier Transform

The Figure 21 represents the Fast Fourier Transform magnitude of the audio file 'dbarrett2.wav'.

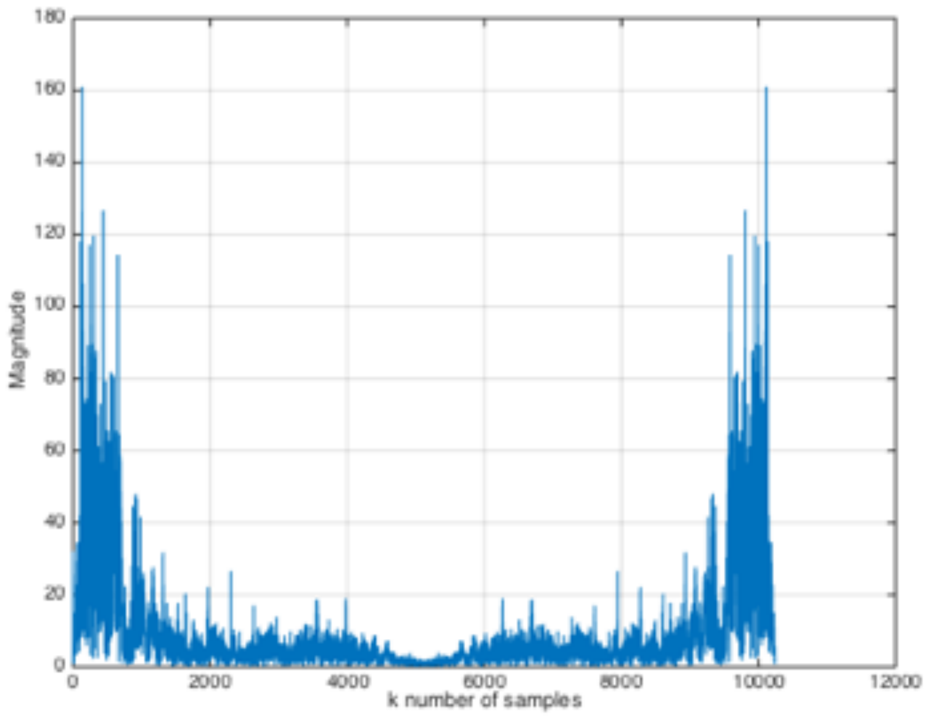


Figure 21: Fast Fourier Transform magnitude representation of 'dbarrett2.wav' audio file plotted versus the time.

The main point of the plot is to show the change in amplitude over a time(which in this case is represented by the number of samples).

The code below was used to generate the plot:

```
[x,Fs] = audioread('dbarrett2.wav); %Read the audio file, and return its number of
                                    %samples x and sampling frequency Fs.
s = abs(fft(x));                    %Calculate the magnitude of every element of the FFT
                                    %of function x.
plot(s);                            %Plot the magnitude versus number of samples(time).
```

The output above is not clear enough, but there exist different methods to change it.The convenient way of doing it is by dividing the plot into small intervals. It is can be done with the following Matlab function.
function

```
y = wft(s,t,n)

c= n+t;          %Calculate the sum of t-time period and n-window length.
b= length(s)     %Get the length of sequence s.
if c>b           %The if statement checks if the sum c is bigger than the input sequence.
                 %If it is,the the new length of n is calculated , so that it dose not
                 %exceed the number of samples.
    n=b-t;       %Calculate the new value of n.
    a =s(t+(1:n));%Create new sequence a which starts at the t point of the input signal
                   %and is the length equal to n.
else                %In the case when the c is equal to b or is smaller than b, the new
                    %sequence a is created and its starting point is t and the length=n.
    a =s(t+(1:n));
end
w = hann(n);     %Creates the hann window with length n.

x = a.*w;        %Multiply the a sequence with the window.

k=fft(x);        %Calculate the pfft of x.

plot(abs(k));    %Plot the Magnitude of FFT of x over the n long sequence.

xlabel(' k - number of samples measured')'
ylabel('Magnitude');
grid

end
```

Plots:

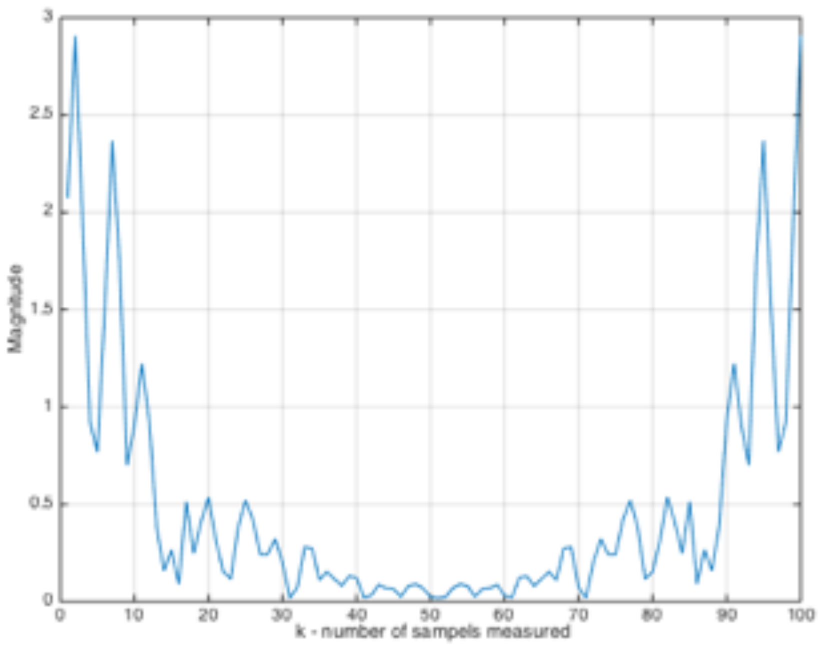


Figure 22: Plot of the signal ‘dbarrett2.wav’ for t=0 and n =100.

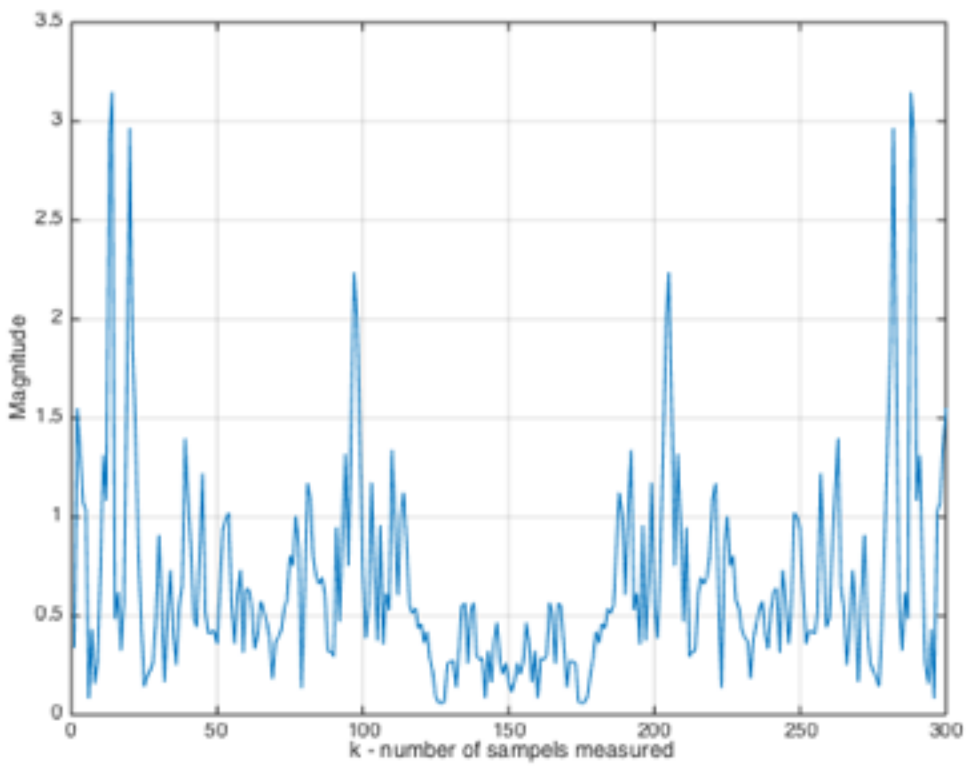


Figure 23: Plot of the signal ‘dbarrett2.wav’ for t=5000 and n =300.

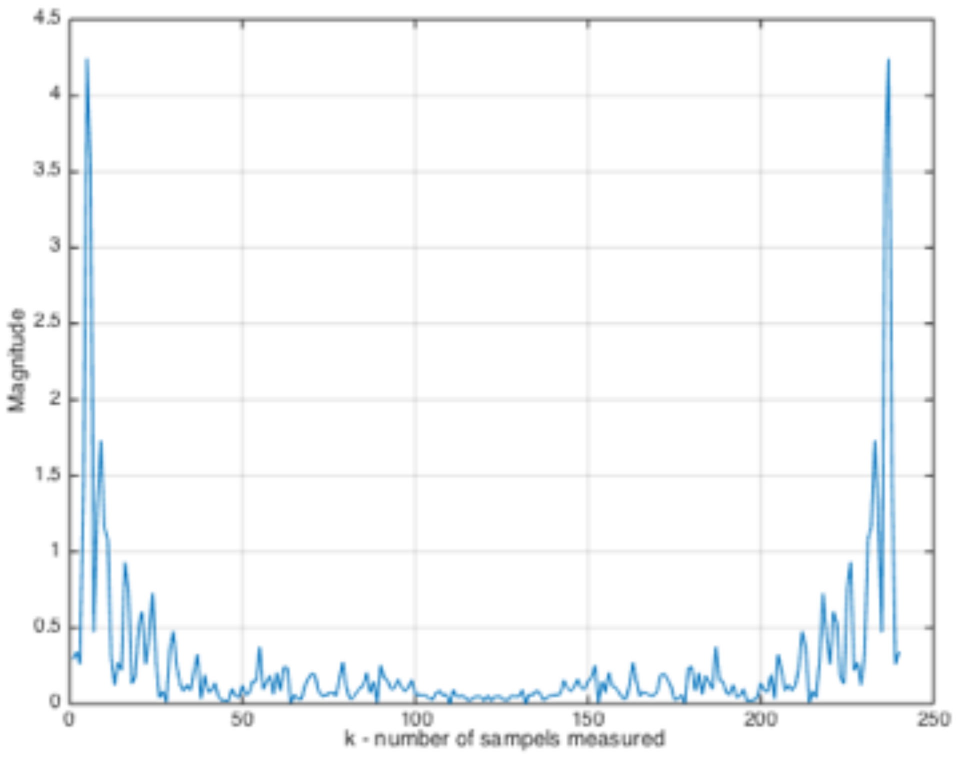


Figure 24: Plot of the signal ‘dbarrett2.wav’ for t=10000 and n =500.

Logarithmic plots:

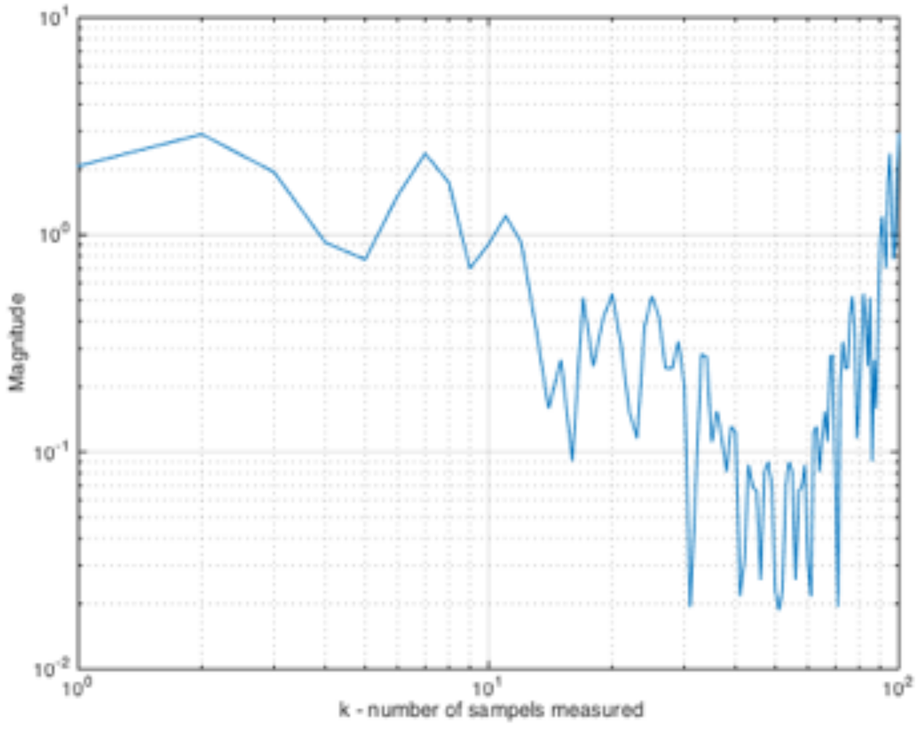


Figure 25: Logarithmic plot of the signal ‘dbarrett2.wav’ for t=0 and n =100.

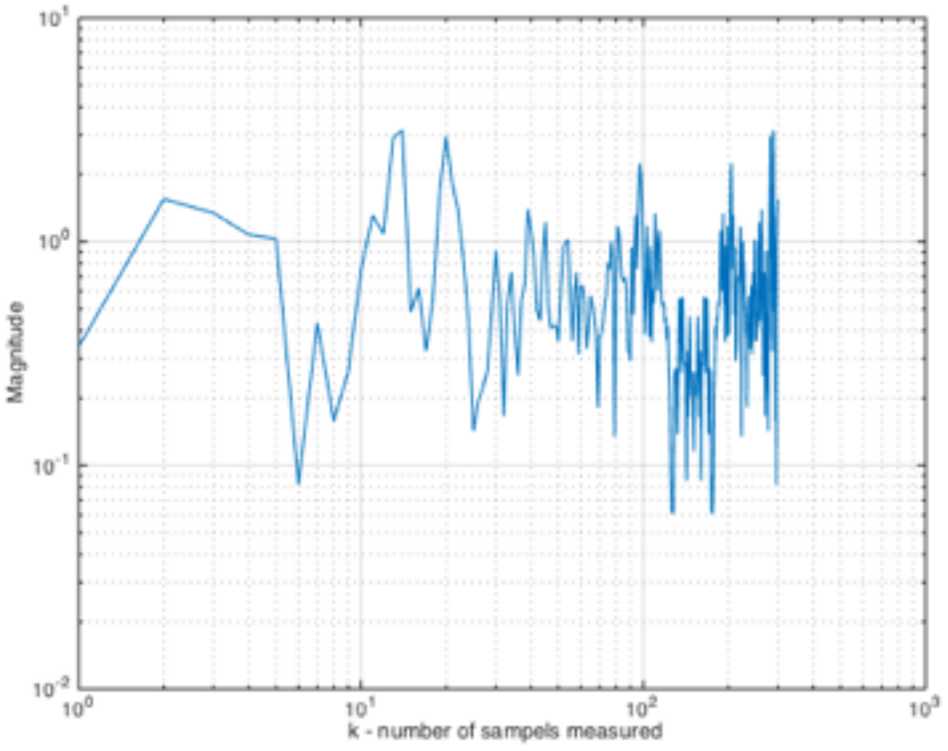


Figure 26: Logarithmic plot of the signal ‘dbarrett2.wav’ for t=5000 and n =300.

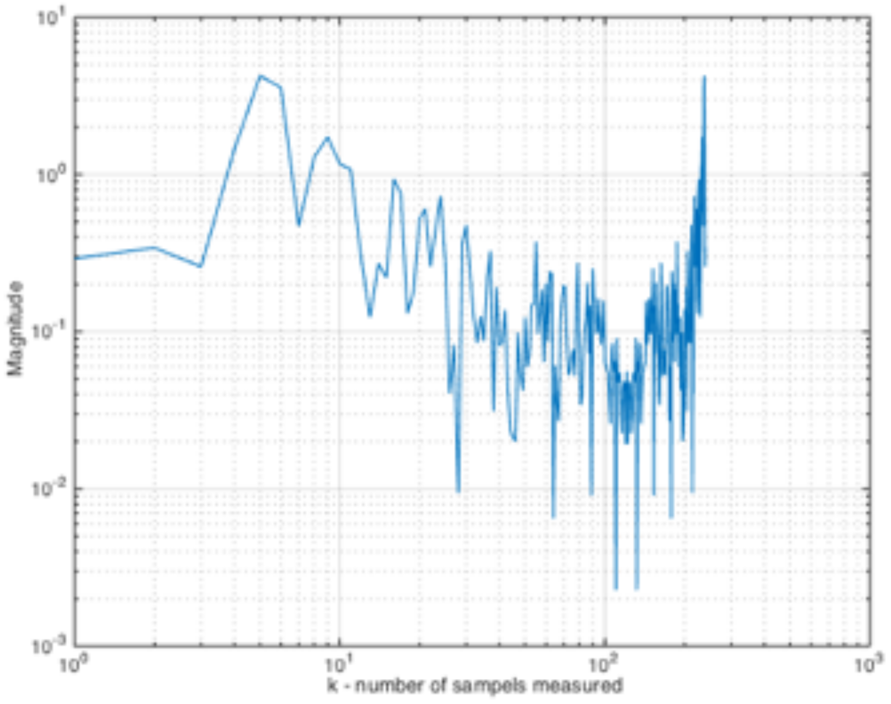


Figure 27:Logarithmic plot of the signal 'dbarrett2.wav' for t=10000and n =500.

Comparing the linear plots with the logarithmic plots it is visible to see that the linear representation in this case is more appropriate. It is cleaner and easier to read.

STFT and Spectrogram

In this section the function to generate spectrogram will be discussed.

Code:

```
function spta(audiofille)

[x,Fs] = audioread(audiofille); %Reads the audio file
length(x)                       %Get the length of x.
nfft=262144;                    %Number of DFT points
window = hamming(512);          %Define the length and type of window
noverlap = 256;                 %Number of overlaped sampels

%Generate Spectrogram.
[S,F,T,P] = spectrogram(x,window,noverlap,nfft,Fs,'yaxis');

%Cretae the surface consisted of Time Frequency and Pressure,without edges
%drawn.
surf(T,F,10*log10(P),'edgecolor','none');

axis tight; %Fit the axes box tightly around the data by setting the axis
            %limits equal to the range of the data.
view(0,90); %Define the angel of view;in this case view directly overhead.
end
```

Plots for four audio files are presented below.

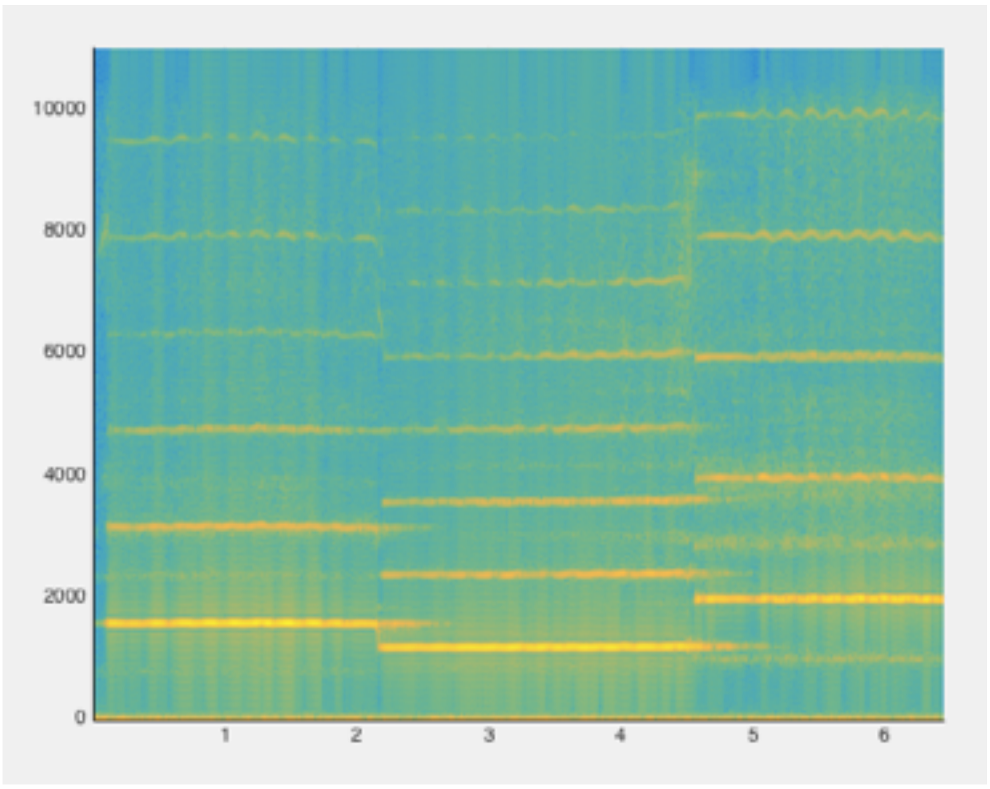


Figure 28: Spectrogram of ‘Picollo.wav’, nfft=262144.

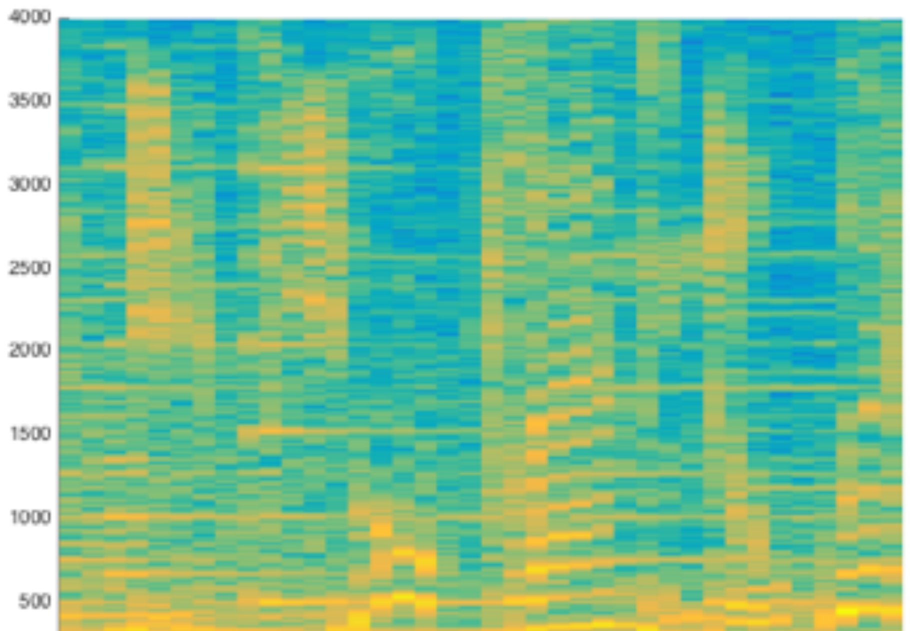


Figure 29: Spectrogram of ‘Picollo.wav’, nfft=16240.

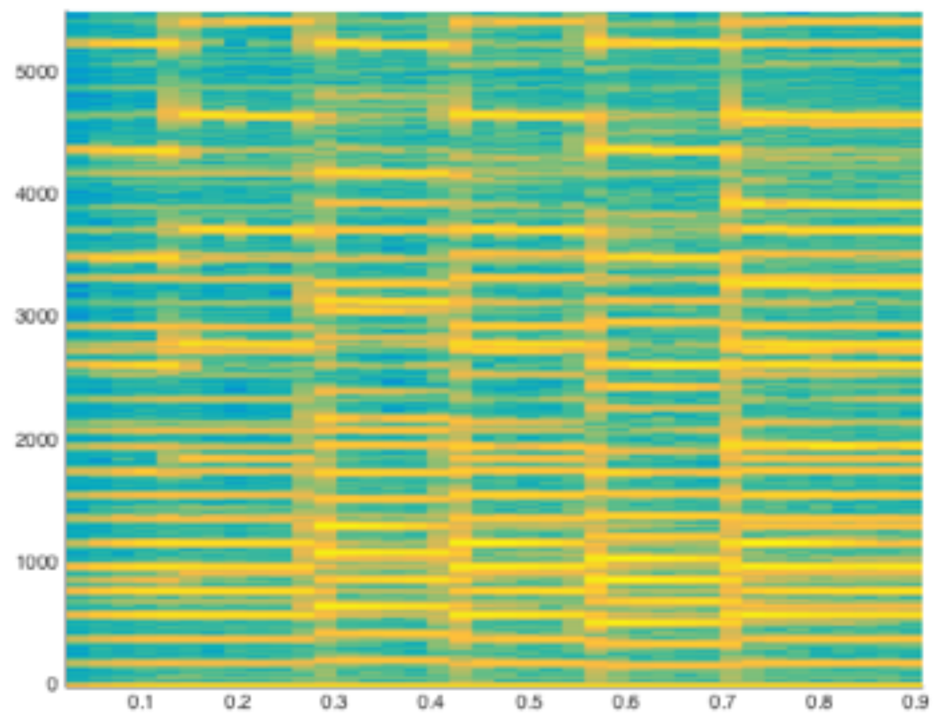


Figure 30: Spectrogram of ‘bwv827b.wav’, nfft=16240.

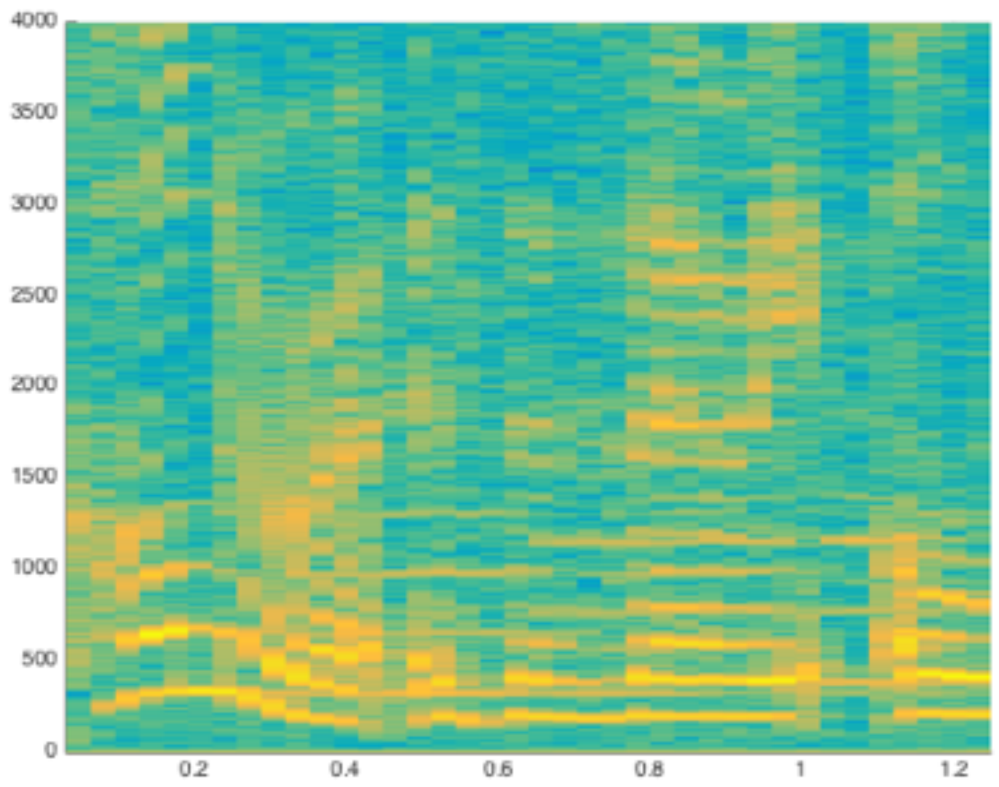
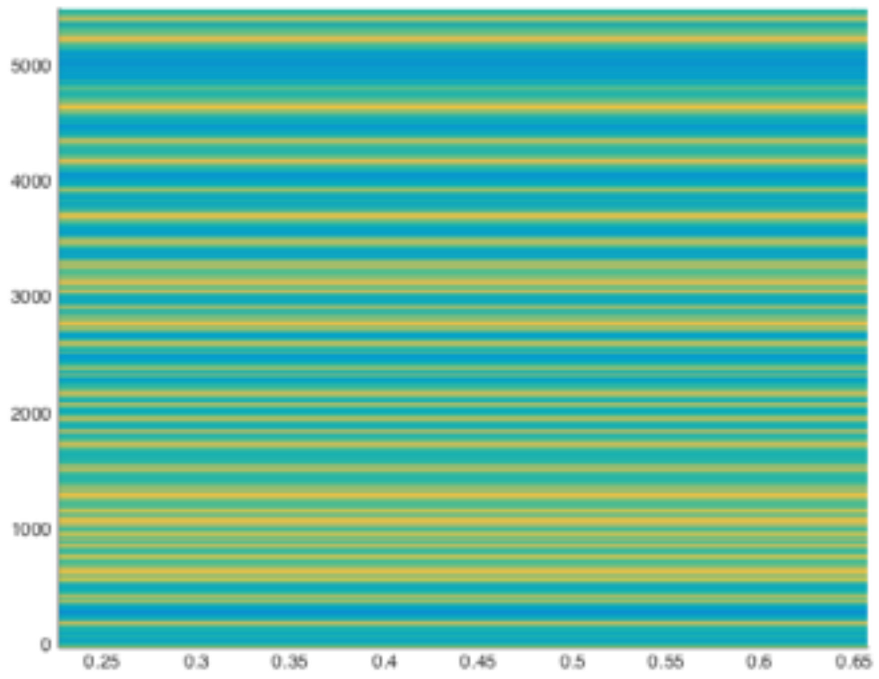


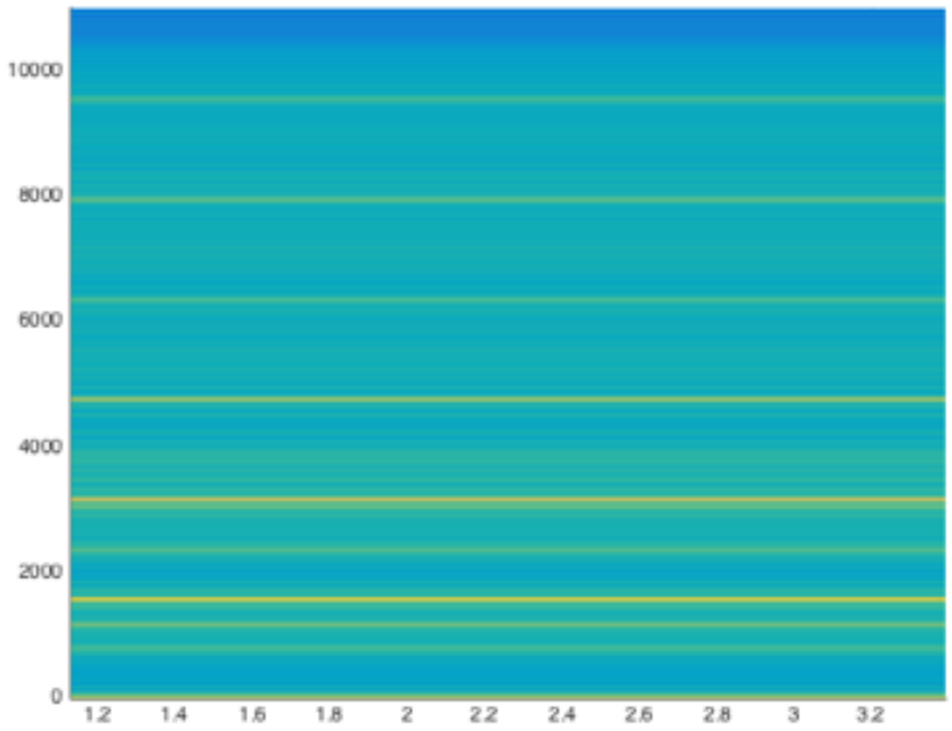
Figure 31: Spectrogram of ‘vmatthew2.wav’, nfft=16240.

All the plots were generated using window size of 512. The bigger the window size is the better the frequency resolution is, but the worse time resolution is. That is because the bigger window size is approximating the values ,and remove the components with lower power intensity.

The ‘bwv827b.wav’ get the best resolution for window of length 5000:



The ‘picollo.wav’ get the best resolution when the window is of the length of 50000:

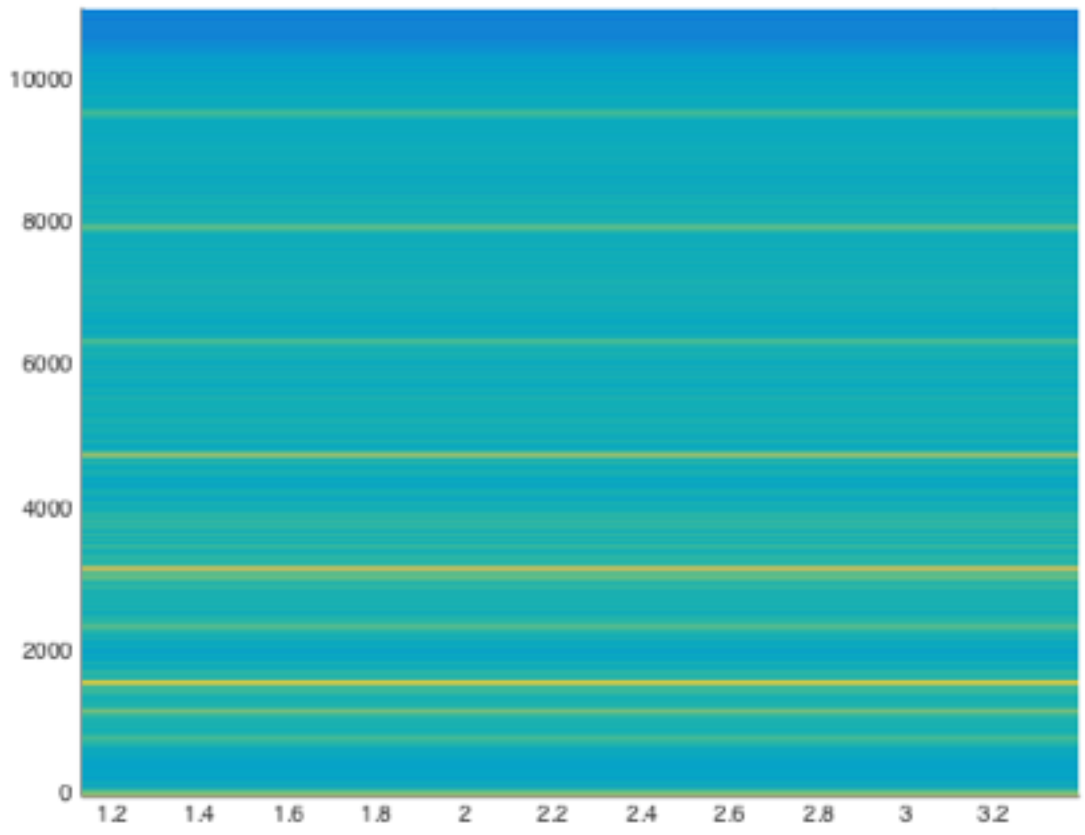


The value of NFFT is chosen, so that it is the closest to the length of the input signal power of 2, and bigger than the input length. The NFFT which is the power of two results in faster performance of the FFT. It is because for such a length the FFT is divided into smaller processes by the Cooley-Tukey algorithm, what result in faster operation. It is more useful for the longer input.

Analysis of the Piccolo sound

The sampling frequency of the 'piccolo.wav' sound is 22500.

The spectrogram of the sound for window size 50000:



By looking at the plot above it essay to see that the fundamental frequency f0 can be approximated to 1600Hz. Then its harmonics are around 3200Hz, 4800Hz, 6400Hz and 8000Hz.
The other method to calculate the fundamental harmonic f0 is to divide the frequency range by the number of harmonics present on the spectrogram.
On the picture above there is six frequencies visible and the frequency range is 10000.

10000/6 = 1666,666

The smaller the window length is, the harder the estimation is ,as there is more and more components and the frequency resolution is worse. the picture is blurred. For the bigger windows the frequencies are smoother..