IMPLEMENTATION OF CONCEPTS OF DSP USING ARDUINO UNO AND FILTERING OF NOISE FROM PULSE SIGNAL

Submitted in partial fulfillment of the requirements of the degree of

Bachelor of Technology

By

Kolla Sree Harshitha
21ECB0F24

Submitted to:

Prof. J. Ravi Kumar

DEPARTMENT OF

ELECTRONICS AND COMMUNICATION ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY, WARANGAL

20223-24

TABLE OF CONTENTS

INTRODUCTION

Digital Signal Processing (DSP) involves the processing and analysis of signals represented in digital form. The processing power of an Arduino Uno is limited compared to dedicated DSP processors, however it can still be used because of its easy accessibility, especially in applications with moderate processing requirements.

Arduino Uno

The Arduino Uno is a microcontroller board that is part of the Arduino platform. It's widely used for prototyping and electronics projects. Here are key features of the Arduino Uno:

a) Microcontroller: The Arduino Uno is based on the ATmega328P microcontroller, which is a member of the AVR family.

b) Digital I/O Pins: It has 14 digital input/output pins, where 6 are used as PWM (Pulse Width Modulation) outputs.

c) Analog Inputs: The board also has 6 analog input pins, allowing us to read analog signals from sensors.

d) Clock Speed: The ATmega328P runs at 16 MHz.

e) Memory:
Flash Memory: 32 KB (0.5 KB is used by the bootloader).
SRAM: 2 KB.
EEPROM: 1 KB

f) USB Connection: The Arduino Uno is equipped with a B type USB connection for programming and interfacing it with a computer.

g) Power: The board can be powered using a USB cable, an external power supply, or a battery operating at 5V.

h) Programming: Arduino Uno is programmed using the Arduino IDE (Integrated Development Environment), a software which supports C programming language.

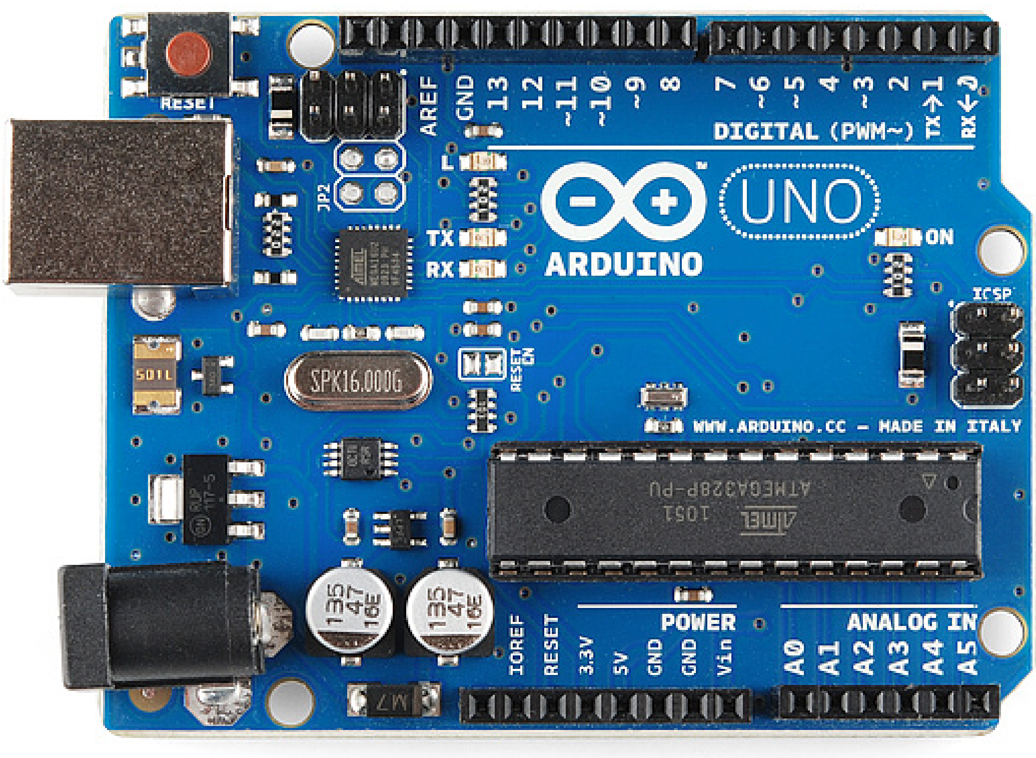
i) Headers: It has digital and analog pins arranged in standard 0.1-inch pitch headers, enabling it to connect the board to peripherals and shields.

j) Compatibility: Arduino Uno is compatible with a wide range of shields (additional boards that can be mounted on top of the Arduino to provide additional functionality).

k) Open Source: The Arduino Uno, like other Arduino boards, is an open-source hardware platform. The design files and software are freely available for users to modify and share.

Typical Usage:

- Prototyping electronic projects and gadgets.
- Learning and teaching electronics in conjunction with programming.
- Developing interactive projects with sensors, actuators, and displays.
- Creating Internet of Things (IoT) applications.



The concepts of DSP which have been implemented on Arduino Uno through this project are:

1) Convolution

Convolution is a mathematical operation that combines two functions in order to produce a third function. In digital signal processing (DSP) and linear systems, convolution is described as the combination of two discrete sequences to produce a third sequence.
Convolution is an essential operation in signal processing and is used in various applications, including analysis of systems, image processing, and filtering signals. It has several properties, such as commutativity and associativity which make it an useful tool in the analysis and manipulation of signals and systems.

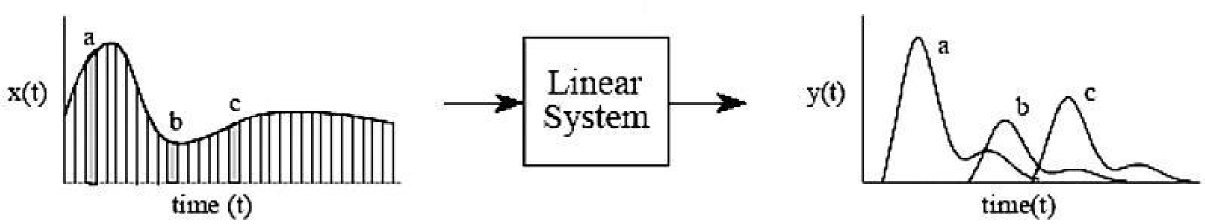


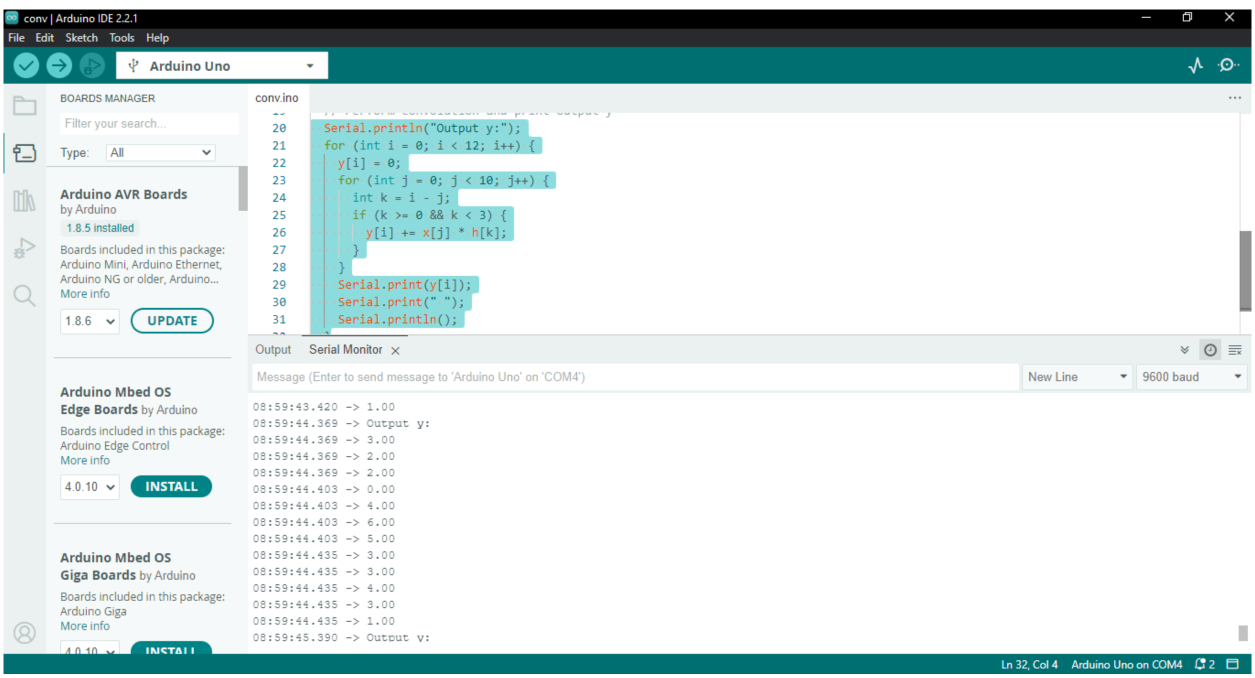
FIGURE 13-2
Convolution viewed from the input side. The input signal, $x(t)$, is divided into narrow segments, each acting as an impulse to the system. The output signal, $y(t)$, is the sum of the resulting scaled and shifted impulse responses. This illustration shows how three points in the input signal contribute to the output signal.

Code

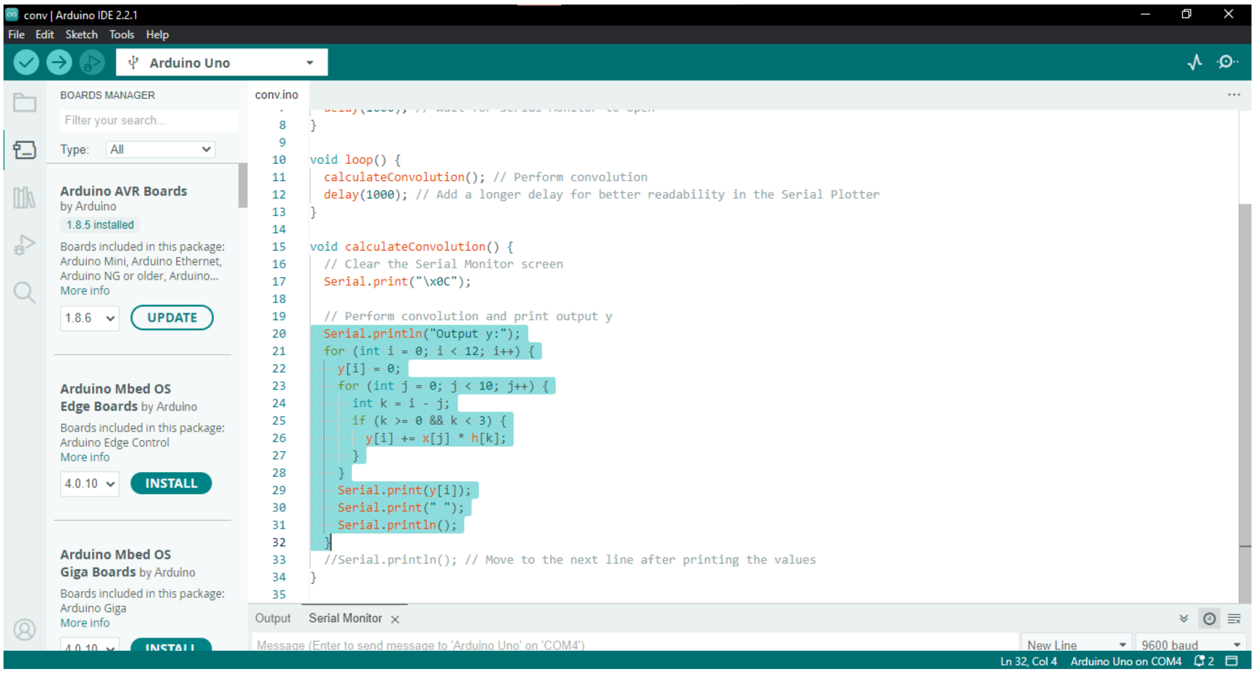
```
double x[10] = {3, -1, 0, 1, 3, 2, 0, 1, 2, 1};
double h[3] = {1, 1, 1};
double y[12];

void setup() {
  Serial.begin(9600); // Initialize serial communication
  delay(1000); // Wait for Serial Monitor to open
}

void loop() {
  calculateConvolution(); // Perform convolution
  delay(1000); // Add a longer delay for better readability in the Serial Plotter
```

```
}

void calculateConvolution() {
  // Clear the Serial Monitor screen
  Serial.print("\x0C");

  // Perform convolution and print output y
  Serial.println("Output y:");
  for (int i = 0; i < 12; i++) {
    y[i] = 0;
    for (int j = 0; j < 10; j++) {
      int k = i - j;
      if (k >= 0 && k < 3) {
        y[i] += x[j] * h[k];
      }
    }
    Serial.print(y[i]);
    Serial.print(" ");
    Serial.println();
  }
}
```

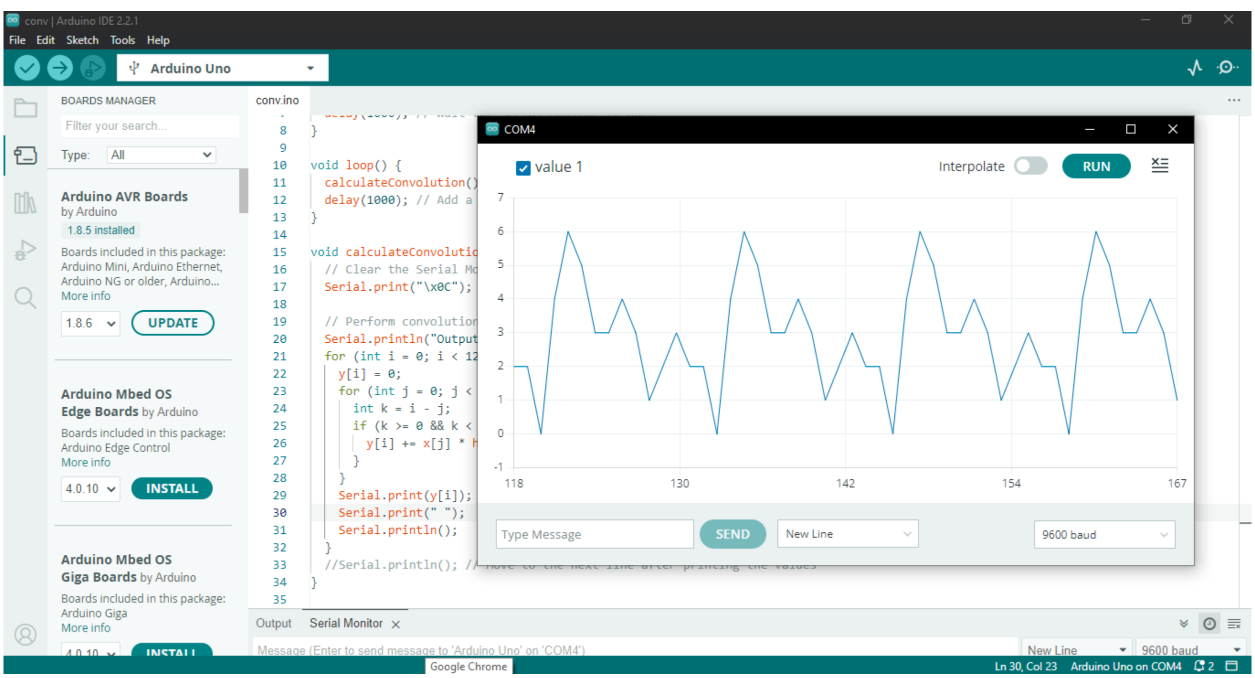
Serial Monitor Output



Code



Graphs Obtained



Hand Calculations

Overlap And Add

$x = \{3, -1, 0, 1, 3, 2, 0, 1, 2, 1\}$

$h = \{1, 1, 1\}$

1. for zero padding , $L(x) = N + M - 1$

$$= 10 + 3 - 1$$

$$= 12$$

$\Rightarrow x = \{3, -1, 0, 1, 3, 2, 0, 1, 2, 1, 0, 0\}$

2. Dividing x into segments of M

$x_1 = \{3, -1, 0\}$

$x_2 = \{1, 3, 2\}$

$x_3 = \{0, 1, 2\}$

$x_4 = \{1, 0, 0\}$

3. Convolution :

$y_1 = x_1 * h$

$= \{3, -1, 0\} * \{1, 1, 1\} = \{3, 2, 2, -1, 0\}$

$y_2 = x_2 * h$

$= \{1, 3, 2\} * \{1, 1, 1\} = \{1, 4, 6, 5, 2\}$

$y_3 = x_3 * h$

$= \{0, 1, 2\} * \{1, 1, 1\} = \{0, 1, 3, 3, 2\}$

$y_4 = x_4 * h$

$= \{1, 0, 0\} * \{1, 1, 1\} = \{1, 1, 1, 0, 0\}$

4. Add the (M-1) part to the successive conv.

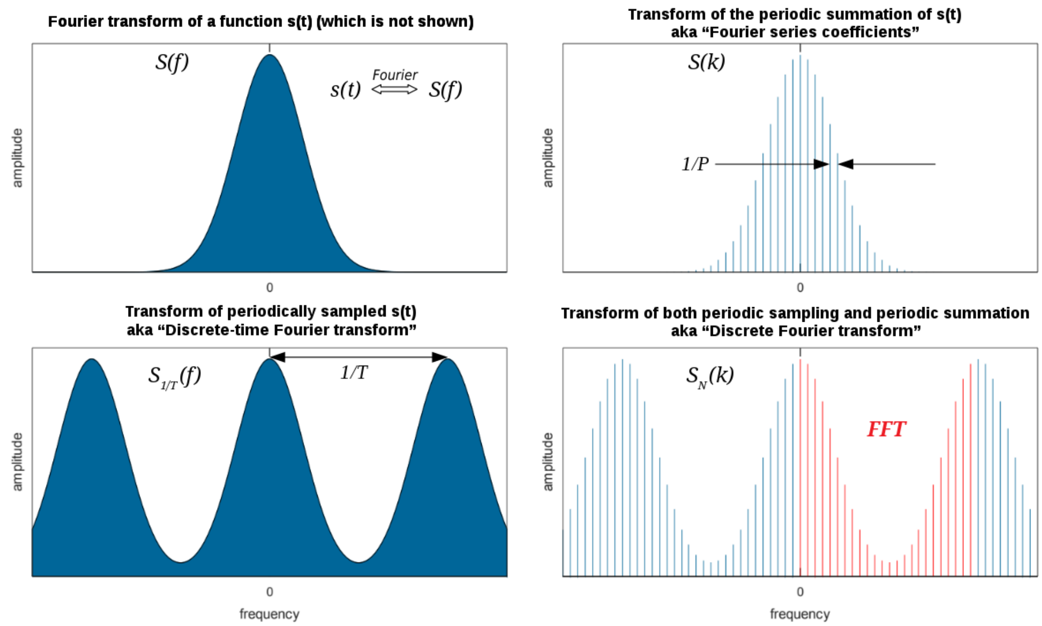
$\therefore y = \{3, 2, 2, 0, 4, 6, 5, 3, 3, 4, 3, 1\}$

$\therefore$ The result of convolution is :

$y = \{3, 2, 2, 0, 4, 6, 5, 3, 3, 4, 3, 1\}$

2) DFT for Continuous Signal

The Discrete Fourier Transform (DFT) is specifically designed for discrete signals, meaning sequences of values that are defined only at distinct points in time. If the signal is a continuous signal, Fourier Transform is used instead of DFT.
If the signal is continuous, mathematical tools like calculus are used to compute its Fourier Transform. However, implementing this on a microcontroller like Arduino (operating with discrete samples in time) might not be straightforward due to the need for integration.

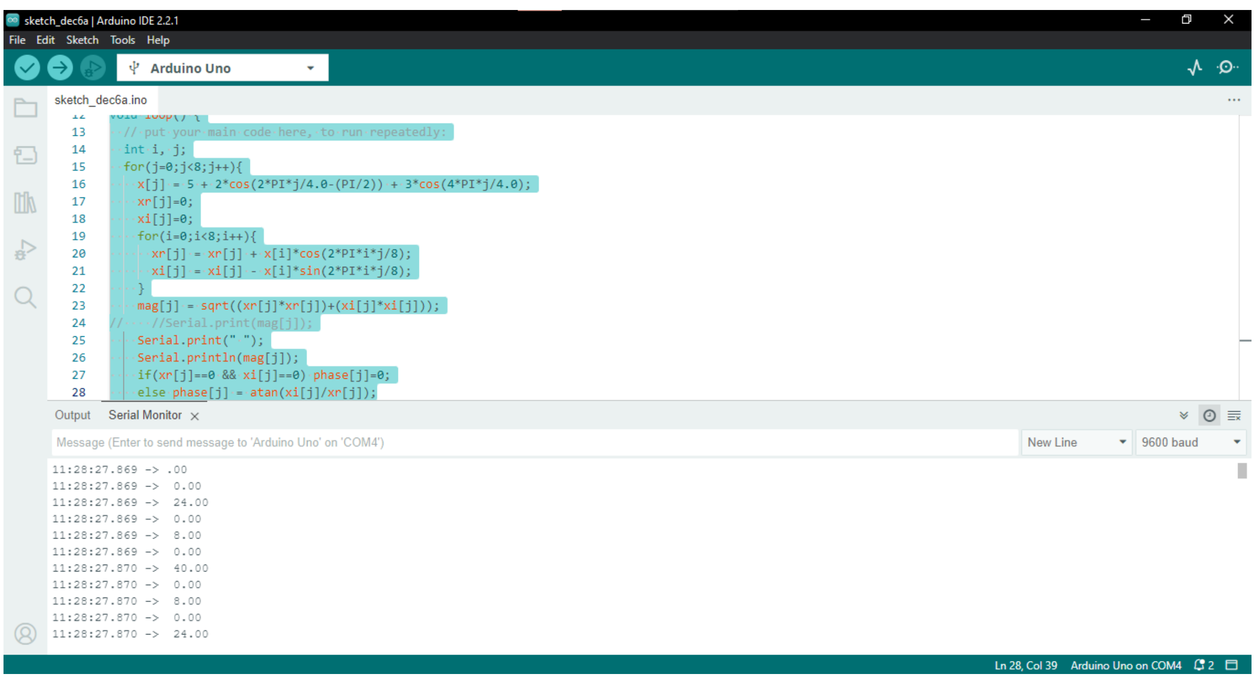


Code

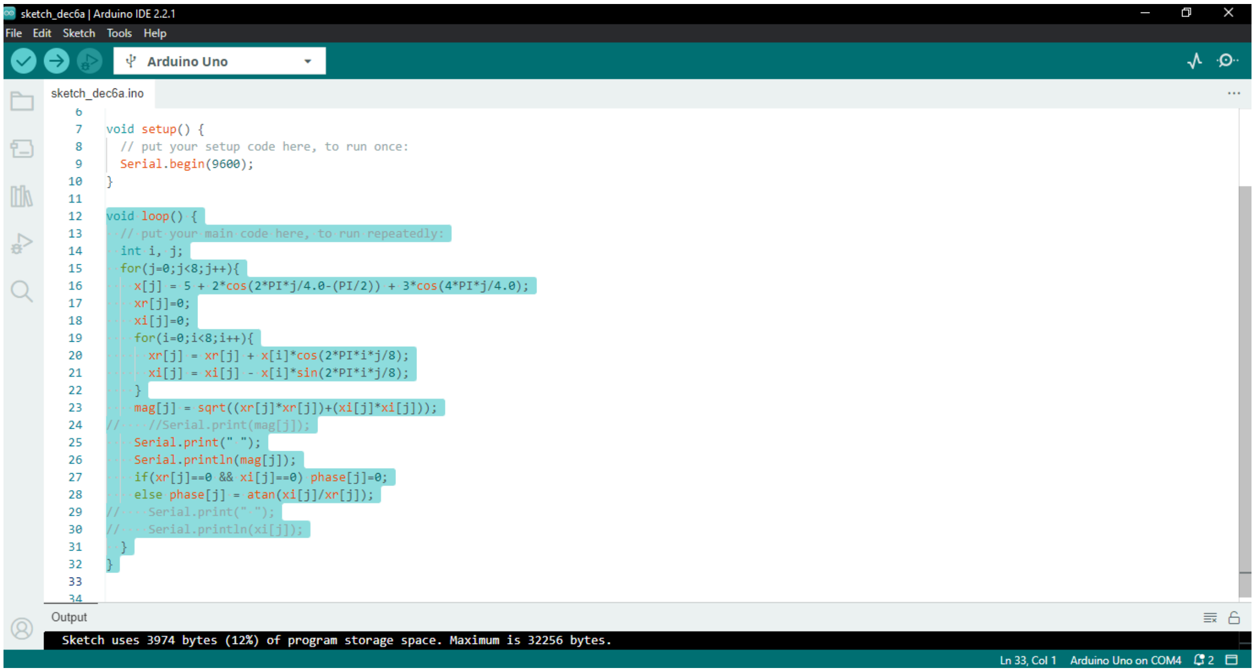
```
int n = 8;
float x[8];
float xr[8], xi[8];
float mag[8];
float phase[8];

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
}

void loop() {
```

```
  // put your main code here, to run repeatedly:
  int i, j;
  for(j=0;j<8;j++){
    x[j] = 5 + 2*cos(2*PI*j/4.0-(PI/2)) + 3*cos(4*PI*j/4.0);
    xr[j]=0;
    xi[j]=0;
    for(i=0;i<8;i++){
      xr[j] = xr[j] + x[i]*cos(2*PI*i*j/8);
      xi[j] = xi[j] - x[i]*sin(2*PI*i*j/8);
    }
    mag[j] = sqrt((xr[j]*xr[j])+(xi[j]*xi[j]));
//    //Serial.print(mag[j]);
    Serial.print(" ");
    Serial.println(mag[j]);
    if(xr[j]==0 && xi[j]==0) phase[j]=0;
    else phase[j] = atan(xi[j]/xr[j]);
//    Serial.print(" ");
//    Serial.println(xi[j]);
  }
}
```

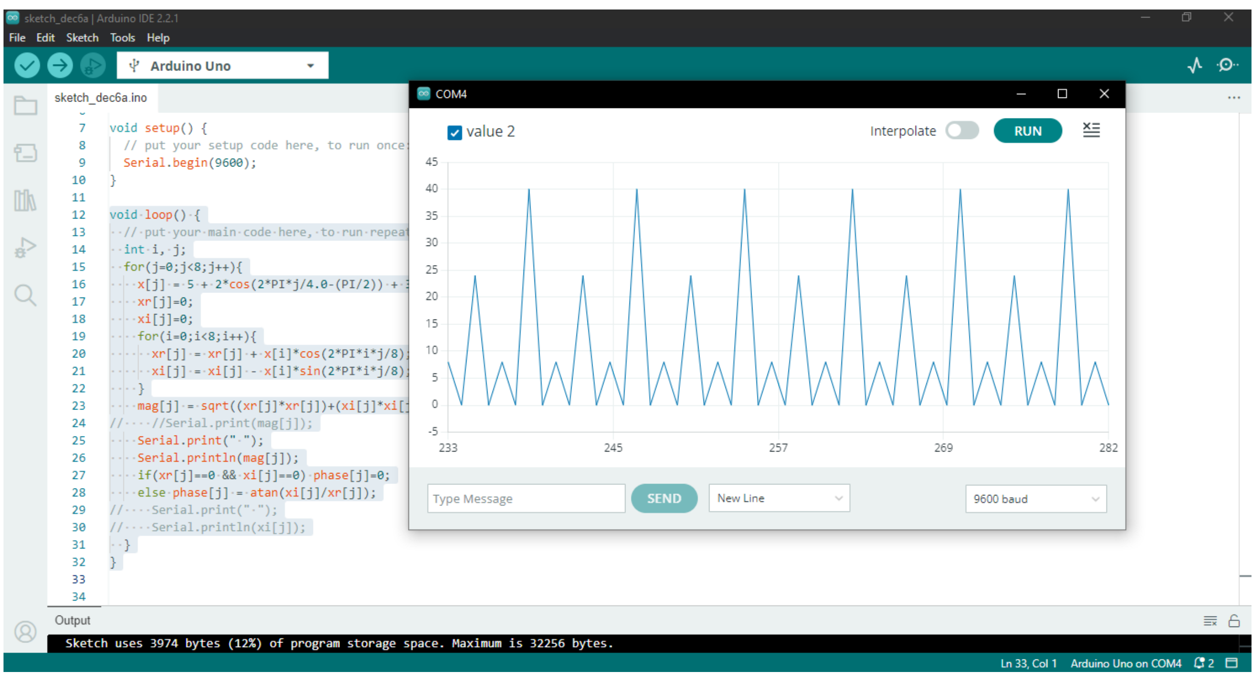
Serial Monitor Output



Code



Graphs Obtained



Hand Calculations

<u>DFT of a continuous signal:</u>

$x(t) = 5 + 2\sin(2\pi t) + 3\cos(4\pi t)$

taking 8 samples:

$x(n) = 5 + 2\sin\left(\frac{2\pi n}{4}\right) + 3\cos\left(\frac{4\pi n}{4}\right)$.

$\Rightarrow x(n) = \{8, 4, 8, 0, 8, 4, 8, 0\}$.

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ X(4) \\ X(5) \\ X(6) \\ X(7) \end{bmatrix} = \begin{bmatrix} W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 & W_8^0 \\ W_8^0 & W_8^1 & W_8^2 & W_8^3 & W_8^4 & W_8^5 & W_8^6 & W_8^7 \\ W_8^0 & W_8^2 & W_8^4 & W_8^6 & W_8^8 & W_8^{10} & W_8^{12} & W_8^{14} \\ W_8^0 & W_8^3 & W_8^6 & W_8^9 & W_8^{12} & W_8^{15} & W_8^{18} & W_8^{21} \\ W_8^0 & W_8^4 & W_8^8 & W_8^{12} & W_8^{16} & W_8^{20} & W_8^{24} & W_8^{28} \\ W_8^0 & W_8^5 & W_8^{10} & W_8^{15} & W_8^{20} & W_8^{25} & W_8^{30} & W_8^{35} \\ W_8^0 & W_8^6 & W_8^{12} & W_8^{18} & W_8^{24} & W_8^{30} & W_8^{36} & W_8^{42} \\ W_8^0 & W_8^7 & W_8^{14} & W_8^{21} & W_8^{28} & W_8^{35} & W_8^{42} & W_8^{49} \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix}$$

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ X(4) \\ X(5) \\ X(6) \\ X(7) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & (1/\sqrt{2} - j/\sqrt{2}) & -j & (-1/\sqrt{2} - j/\sqrt{2}) & -1 & (-1/\sqrt{2} + j/\sqrt{2}) & j & (1/\sqrt{2} + j/\sqrt{2}) \\ 1 & -j & -1 & j & 1 & -j & -1 & j \\ 1 & (-1/\sqrt{2} - j/\sqrt{2}) & j & (1/\sqrt{2} - j/\sqrt{2}) & -1 & (1/\sqrt{2} + j/\sqrt{2}) & -j & (-1/\sqrt{2} + j/\sqrt{2}) \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & (-1/\sqrt{2} + j/\sqrt{2}) & -j & (1/\sqrt{2} + j/\sqrt{2}) & -1 & (1/\sqrt{2} - j/\sqrt{2}) & j & (-1/\sqrt{2} - j/\sqrt{2}) \\ 1 & j & -1 & -j & 1 & j & -1 & -j \\ 1 & (1/\sqrt{2} + j/\sqrt{2}) & j & (-1/\sqrt{2} + j/\sqrt{2}) & -1 & (-1/\sqrt{2} - j/\sqrt{2}) & -j & (1/\sqrt{2} - j/\sqrt{2}) \end{bmatrix} \begin{bmatrix} 8 \\ 4 \\ 8 \\ 0 \\ 8 \\ 4 \\ 8 \\ 0 \end{bmatrix}$$

$\Rightarrow X(k) = \{40, 0, -8j, 0, 24, 0, 8j, 0\}$.

3) <u>DIT - FFT for Continuous Signal</u>

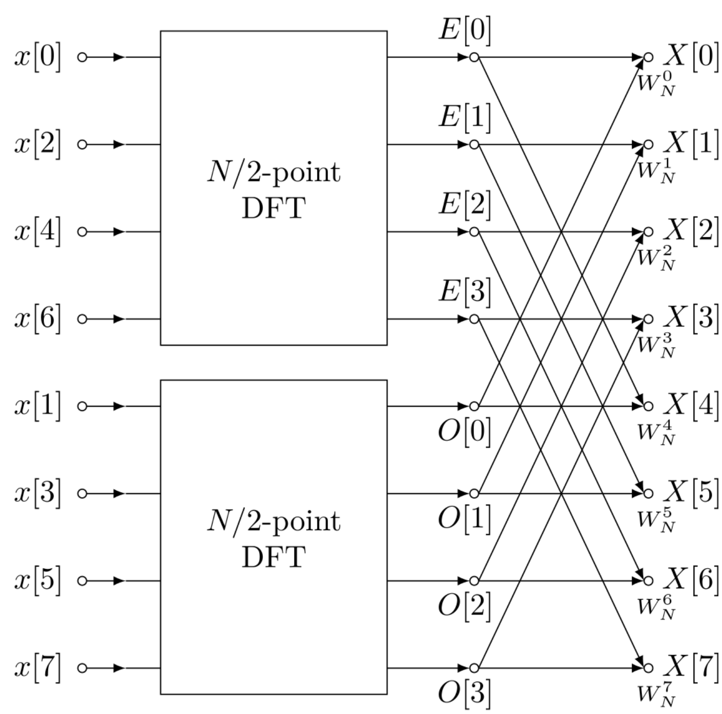
The Decimation in Time FFT (Fast Fourier Transform) is an algorithmic technique used to efficiently compute the Discrete Fourier Transform (DFT). In the DIT FFT, the computations are performed in a recursive, divide-and-conquer fashion. The main idea is to divide the initial DFT into smaller DFTs and combine their results together.

The basic steps of the DIT FFT algorithm are:

a) Decimation in Time: The sequence is divided into even and odd indices, generating two subsequences.
b) Recursive FFT: The FFT is recursively applied to the even and odd subsequences.
c) Combine Results: The results of the FFT on even and odd subsequences are then combined in order to produce the final result.

The DIT FFT is more memory-efficient than its counterpart, the DIF (Decimation in Frequency) FFT, as it doesn't require an additional bit-reversal process.

The Discrete Fourier Transform (DFT) is a mathematical concept that analyzes the frequency parameters of a discrete signal. The Fast Fourier Transform (FFT) is an efficient algorithm for computing the DFT. It's a general term that can refer to both Decimation in Time (DIT) and Decimation in Frequency (DIF) FFT algorithms.



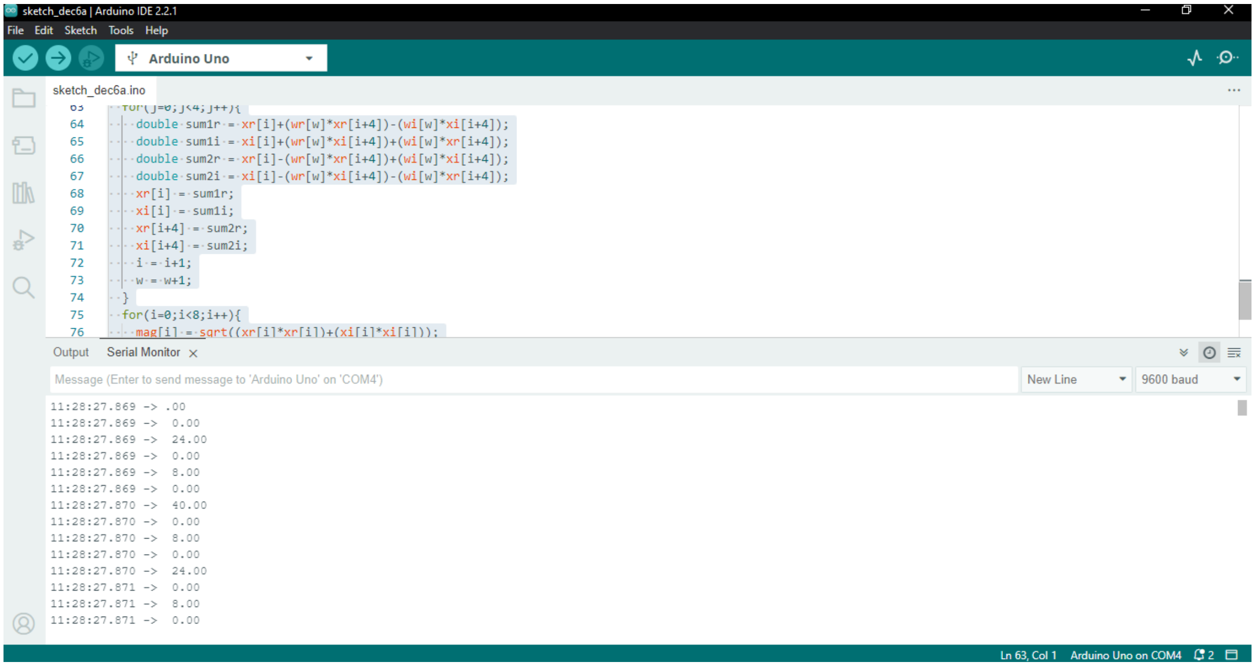
Code

```
int n = 8;
double x[8];
double xr[8], xi[8];
float wr[4] = {1.0, 0.707, 0.0, -0.707};
float wi[4] = {0.0, -0.707, -1.0, 0.707};
double mag[8];
double phase[8];

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
  int i, j;
  for(j=0;j<8;j++){
    x[j] = 5 + 2*cos(2*PI*j/4.0-(PI/2)) + 3*cos(4*PI*j/4.0);
    xr[j]=0;
    xi[j]=0;
  }
  xr[0] = x[0];
  xr[1] = x[4];
  xr[2] = x[2];
  xr[3] = x[6];
  xr[4] = x[1];
  xr[5] = x[5];
  xr[6] = x[3];
  xr[7] = x[7];
  j=0;

  for(j=0;j<=3;j++){
    double sum1r = xr[i] + xr[i+1];
    double sum1i = xi[i] + xi[i+1];
    double sum2r = xr[i] - xr[i+1];
    double sum2i = xi[i] - xi[i+1];
    xr[i] = sum1r;
    xi[i] = sum1i;
```

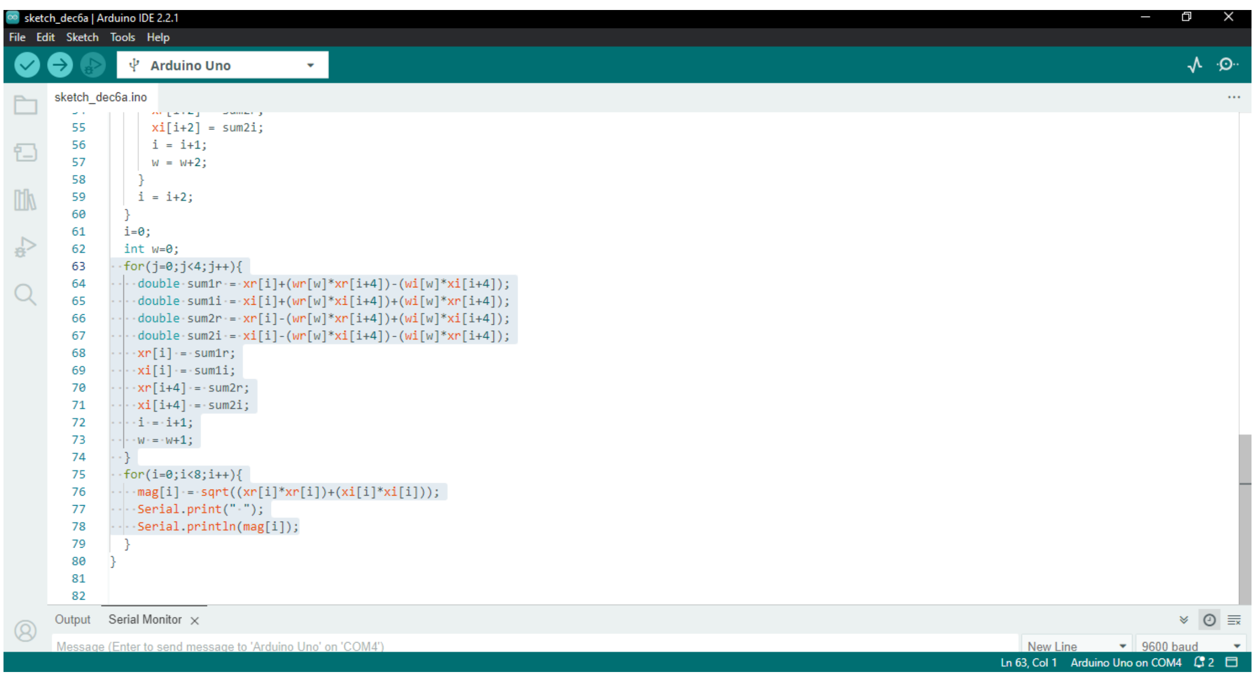
```
  xr[i+1] = sum2r;
  xi[i+1] = sum2i;
  i = i+2;
}
int k;
i=0;
for(k=0;k<2;k++){
  int w=0;
  for(j=0;j<2;j++){
    double sum1r = xr[i]+(wr[w]*xr[i+2])-(wi[w]*xi[i+2]);
    double sum1i = xi[i]+(wr[w]*xi[i+2])+(wi[w]*xr[i+2]);
    double sum2r = xr[i]-(wr[w]*xr[i+2])+(wi[w]*xi[i+2]);
    double sum2i = xi[i]-(wr[w]*xi[i+2])-(wi[w]*xr[i+2]);
    xr[i] = sum1r;
    xi[i] = sum1i;
    xr[i+2] = sum2r;
    xi[i+2] = sum2i;
    i = i+1;
    w = w+2;
  }
  i = i+2;
}
i=0;
int w=0;
for(j=0;j<4;j++){
  double sum1r = xr[i]+(wr[w]*xr[i+4])-(wi[w]*xi[i+4]);
  double sum1i = xi[i]+(wr[w]*xi[i+4])+(wi[w]*xr[i+4]);
  double sum2r = xr[i]-(wr[w]*xr[i+4])+(wi[w]*xi[i+4]);
  double sum2i = xi[i]-(wr[w]*xi[i+4])-(wi[w]*xr[i+4]);
  xr[i] = sum1r;
  xi[i] = sum1i;
  xr[i+4] = sum2r;
  xi[i+4] = sum2i;
  i = i+1;
  w = w+1;
}
for(i=0;i<8;i++){
  mag[i] = sqrt((xr[i]*xr[i])+(xi[i]*xi[i]));
  Serial.print(" ");
  Serial.println(mag[i]);
```

}
}

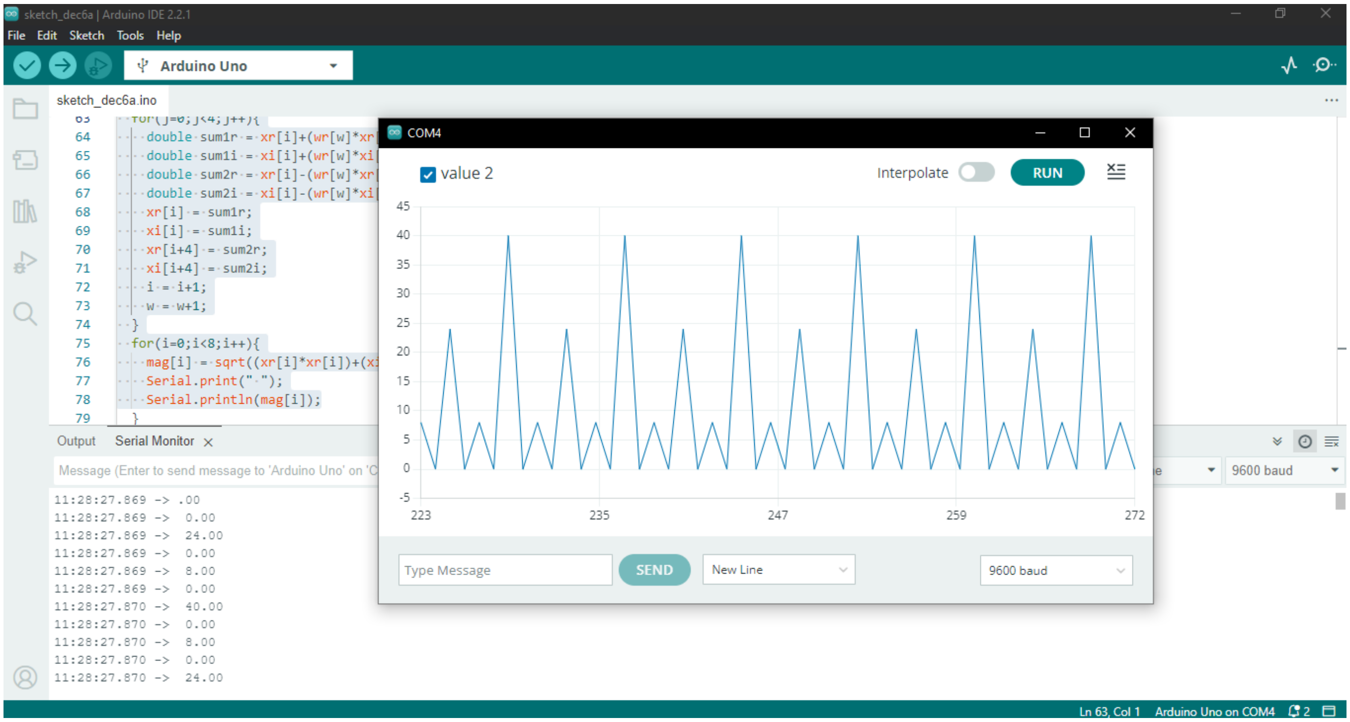
Serial Monitor Output



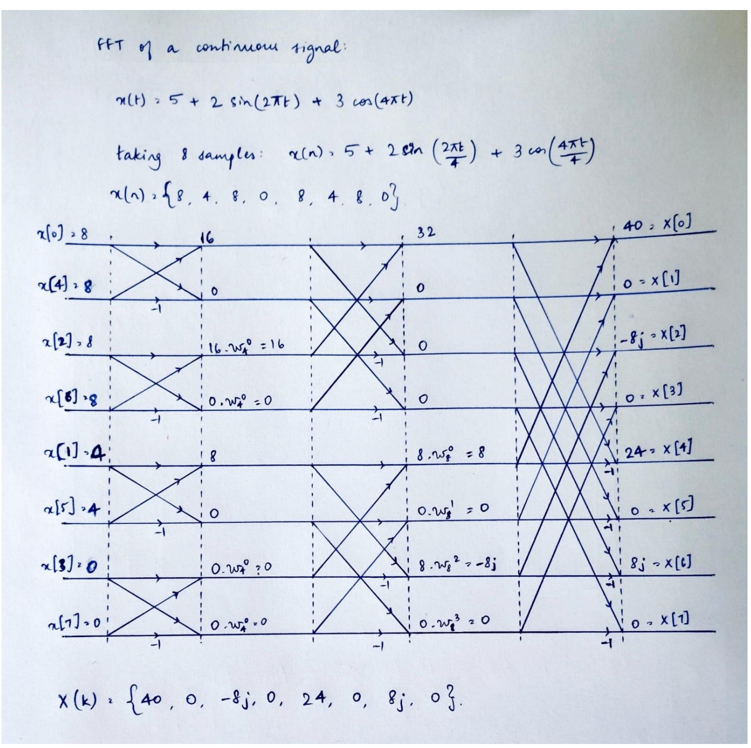
Code



Graph Obtained



Hand Calculations



4) FFT using Hanning Window

The Hanning window, which is also known as the Hann window, is a mathematical function that is commonly used in signal processing to reduce the leakage of spectral components when a portion of the signal is being analyzed, particularly when using the Fourier Transform. The window is named after Julius von Hann, who introduced it.

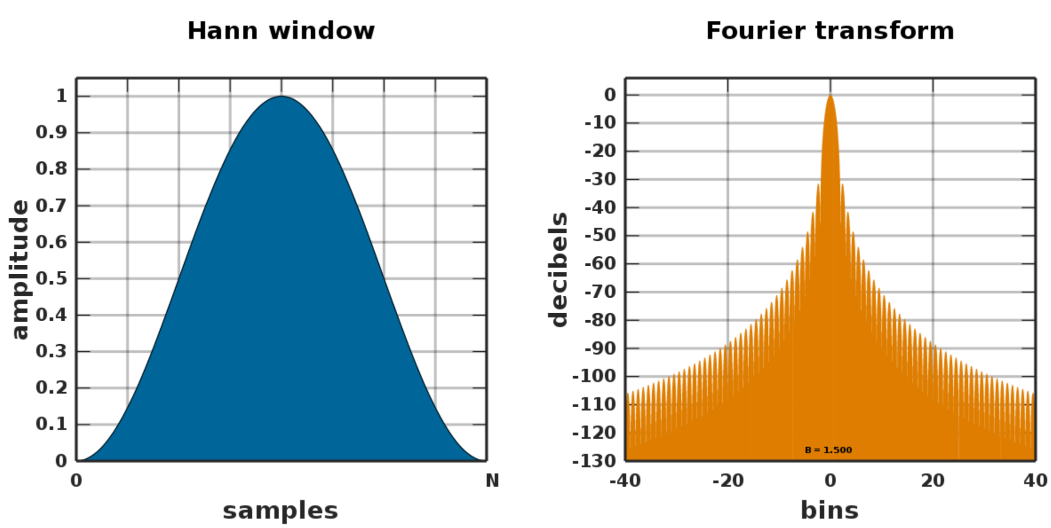
Key properties and characteristics of the Hanning window:

a) Symmetry: It is symmetric around its center.
b) Tapering: The Hann window tapers smoothly to zero at both ends.
c) Main Lobe Width:The main lobe of the window (the central region) is relatively wide, which helps in preserving energy from the signal.
d) Side Lobe Attenuation:The side lobes (the lobes on either side of the main lobe) exhibit relatively low amplitude, reducing spectral leakage.

Mathematical Form:

$$w(n) = 0.5\,(1 \pm \cos(2\pi n N))\ ,\ 0 \leq n \leq N$$

The cosine term in the formula introduces a tapering effect, ensuring that the window smoothly approaches zero at both ends.
The purpose of using the Hanning window is to minimize the effects of spectral leakage, which can occur when analyzing a portion of a signal that does not have an exact integer number of cycles within the window. Spectral leakage can lead to inaccuracies in frequency estimation when using techniques like the Fourier Transform.



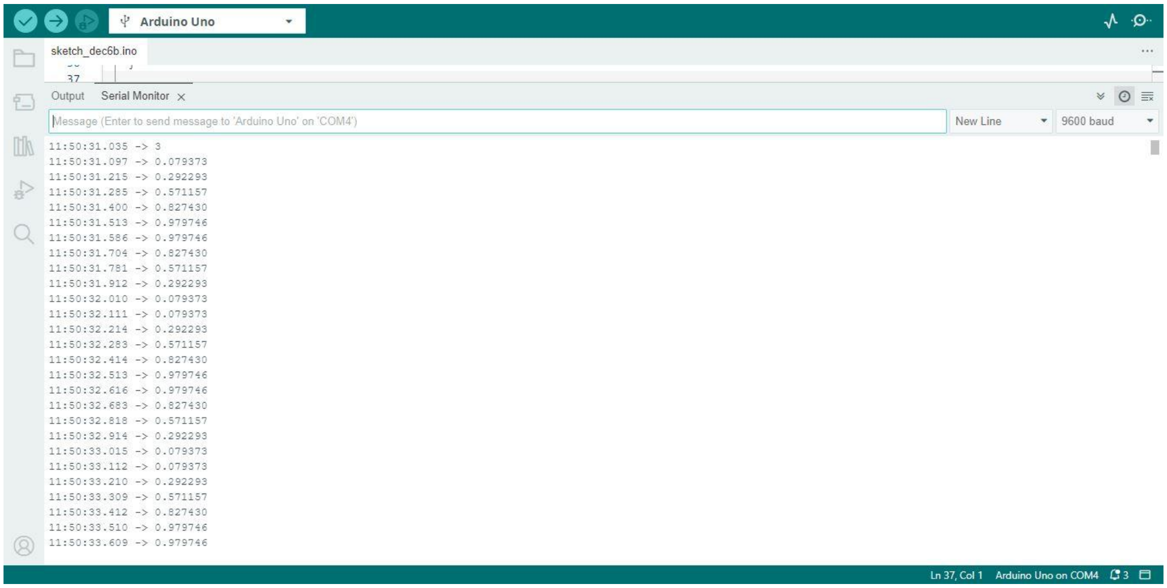
Code

```
#define PI 3.14159265359

float *hanning(int N, short itype)
{
   int half, i, idx, n;
   float *w;

   w = (float*) calloc(N, sizeof(float));
   memset(w, 0, N * sizeof(float));

   if (itype == 1)    // periodic function
      n = N - 1;
   else
      n = N;

   if (n % 2 == 0) {
      half = n / 2;
      for (i = 0; i < half; i++)
         w[i] = 0.5 * (1 - cos(2 * PI * (i + 1) / (n + 1)));

      idx = half - 1;
      for (i = half; i < n; i++) {
         w[i] = w[idx];
         idx--;
      }
   } else {
      half = (n + 1) / 2;
      for (i = 0; i < half; i++)
         w[i] = 0.5 * (1 - cos(2 * PI * (i + 1) / (n + 1)));

      idx = half - 2;
      for (i = half; i < n; i++) {
         w[i] = w[idx];
         idx--;
      }
   }

   if (itype == 1)    // periodic function
```

```
  {
    for (i = N - 1; i >= 1; i--)
      w[i] = w[i - 1];
    w[0] = 0.0;
  }
  return w;
}

void setup() {
  Serial.begin(9600);

  int N = 10;
  short itype = 0;

  float *hannWindow = hanning(N, itype);

  // Printing generated Hanning window coefficients to Serial Plotter
  for (int i = 0; i < N; i++) {
    Serial.println(hannWindow[i], 6); // Adjust the number of decimal
    delay(100); // Adjust the delay
  }

  // Free the allocated memory
  free(hannWindow);
}

void loop() {
  Serial.begin(9600);

  int N = 10;
  short itype = 0;

  float *hannWindow = hanning(N, itype);

  // Printing generated Hanning window coefficients to Serial Plotter
  for (int i = 0; i < N; i++) {
    Serial.println(hannWindow[i], 6); // Adjust the number of decimal
    delay(100); // Adjust the delay
  }
```

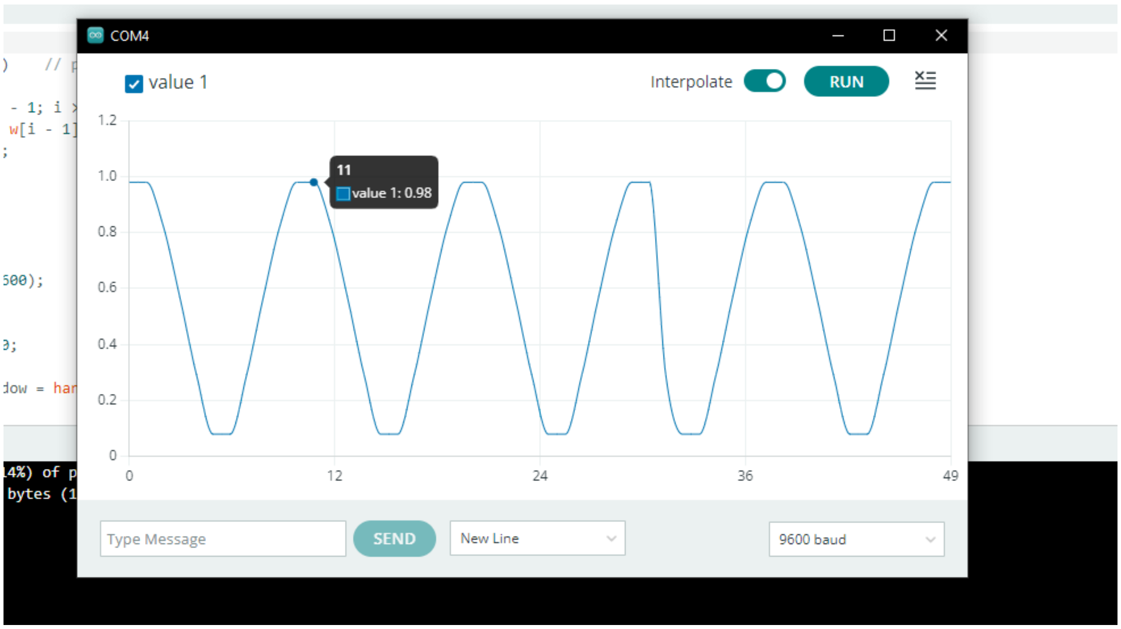
```
    // Free the allocated memory
    free(hannWindow);
}
```

Serial Monitor Output



Code

Graphs Obtained



5) Filter to remove 50Hz Noise

In Digital Signal Processing (DSP), filters are systems or algorithms designed to modify or manipulate the frequency content of a signal. Filters play a crucial role in various applications, including audio processing, image processing, communications, and control systems. There are two main types of filters: analog filters and digital filters.

Digital Filters:
Digital filters operate on discrete-time signals, which are sequences of samples at discrete time intervals.

- FIR Filters (Finite Impulse Response):
  FIR filters are characterized by a finite impulse response. The output is a weighted sum of the current and past input samples. The filter coefficients are the weights.
  FIR filters have linear phase characteristics and are often used when phase linearity is critical.

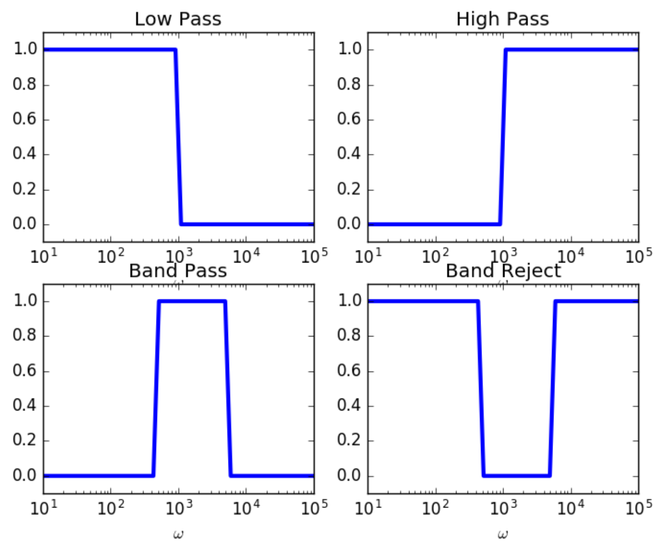
- IIR Filters (Infinite Impulse Response):
  IIR filters have an infinite impulse response, meaning the output depends on the current and past input samples as well as past output samples.

IIR filters are more computationally efficient than FIR filters for achieving similar frequency responses. However, they may exhibit nonlinear phase characteristics.

Filter Types Based on Frequency Response:

`

1) Low-Pass Filter:Allows low-frequency components to pass through while attenuating high-frequency components.
2) High-Pass Filter:Allows high-frequency components to pass through while attenuating low-frequency components.
3) Band-Pass Filter:Allows a specific range of frequencies to pass through, attenuating frequencies outside that range.
4) Band-Stop Filter (Notch Filter):Attenuates a specific range of frequencies, allowing frequencies outside that range to pass through.

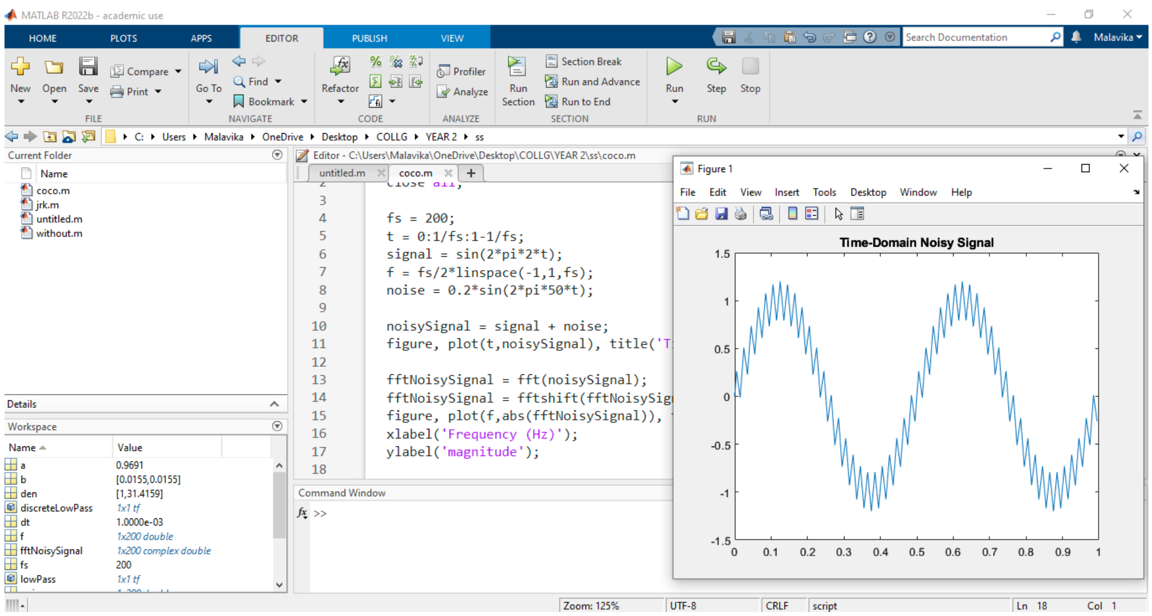


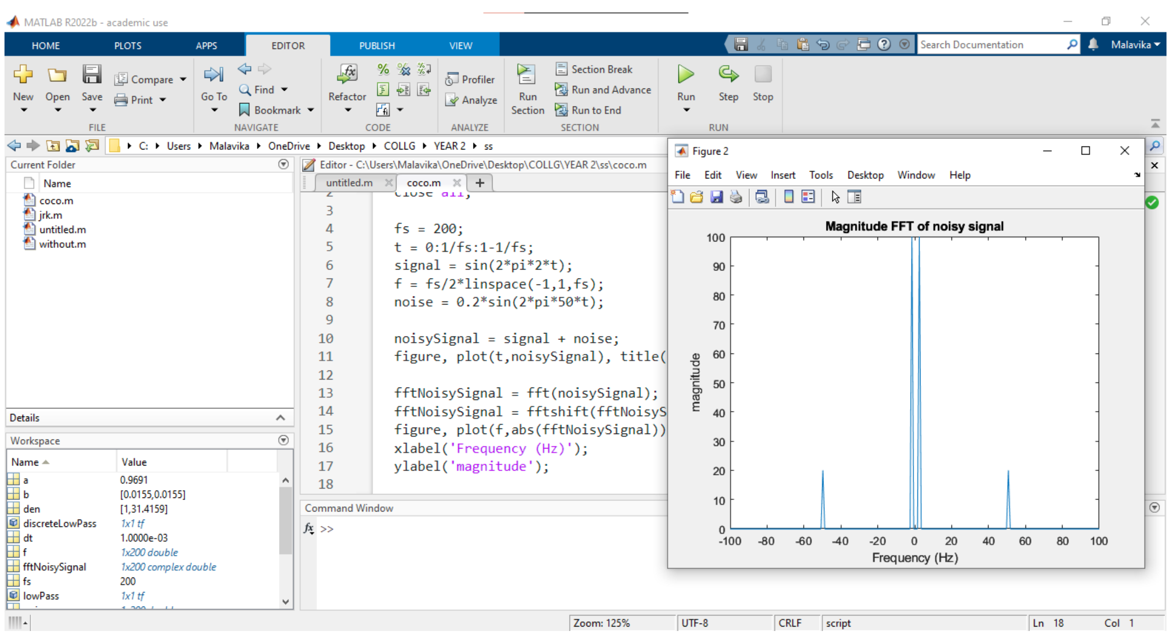
Code

Matlab code and plot of noisy signal

```
clc;
close all;

fs = 200;
t = 0:1/fs:1-1/fs;
signal = sin(2*pi*2*t);
f = fs/2*linspace(-1,1,fs);
noise = 0.2*sin(2*pi*50*t);
```

```
noisySignal = signal + noise;
figure, plot(t,noisySignal), title('Time-Domain Noisy Signal');

fftNoisySignal = fft(noisySignal);
fftNoisySignal = fftshift(fftNoisySignal);
figure, plot(f,abs(fftNoisySignal)), title('Magnitude FFT of noisy signal');
xlabel('Frequency (Hz)');
ylabel('magnitude');
```

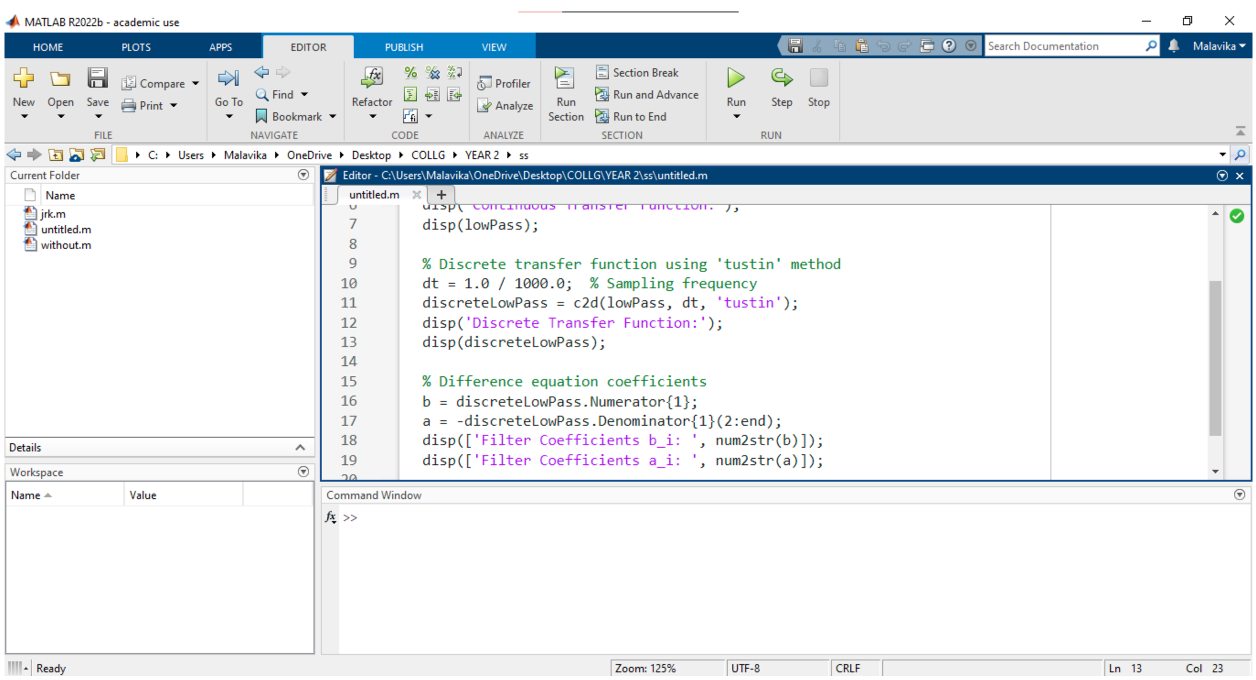




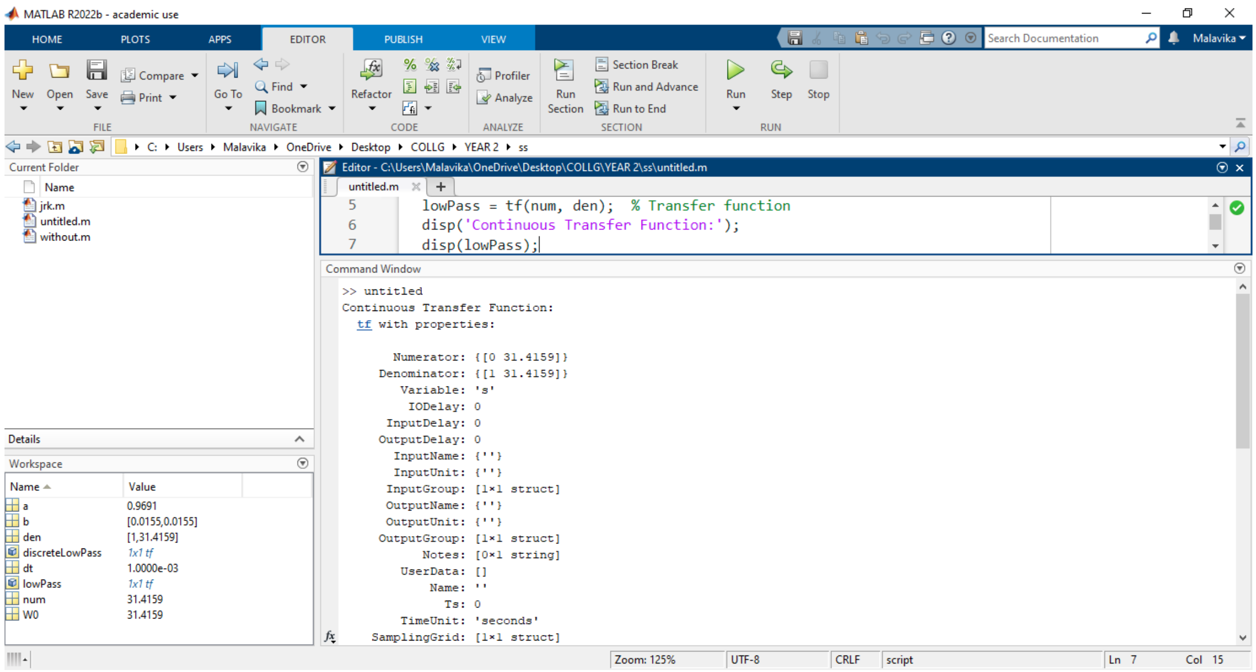
Matlab code to calculate coefficients

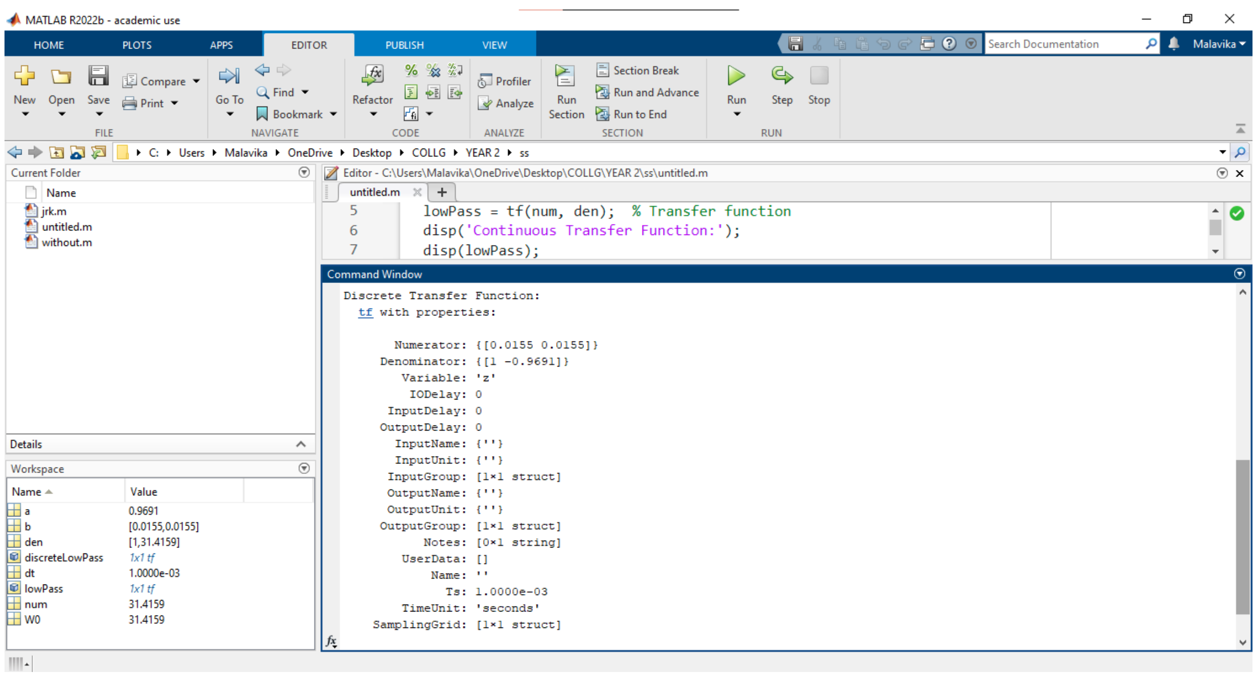
```
% Continuous transfer function
W0 = 2 * pi * 5;  % pole frequency (rad/s)
num = W0;
den = [1, W0];
lowPass = tf(num, den);  % Transfer function
disp('Continuous Transfer Function:');
disp(lowPass);

% Discrete transfer function using 'tustin' method, BLT
dt = 1.0 / 1000.0;  % Sampling frequency
discreteLowPass = c2d(lowPass, dt, 'tustin');
disp('Discrete Transfer Function:');
disp(discreteLowPass);

% Difference equation coefficients
b = discreteLowPass.Numerator{1};
a = -discreteLowPass.Denominator{1}(2:end);
disp(['Filter Coefficients b_i: ', num2str(b)]);
disp(['Filter Coefficients a_i: ', num2str(a)]);
```

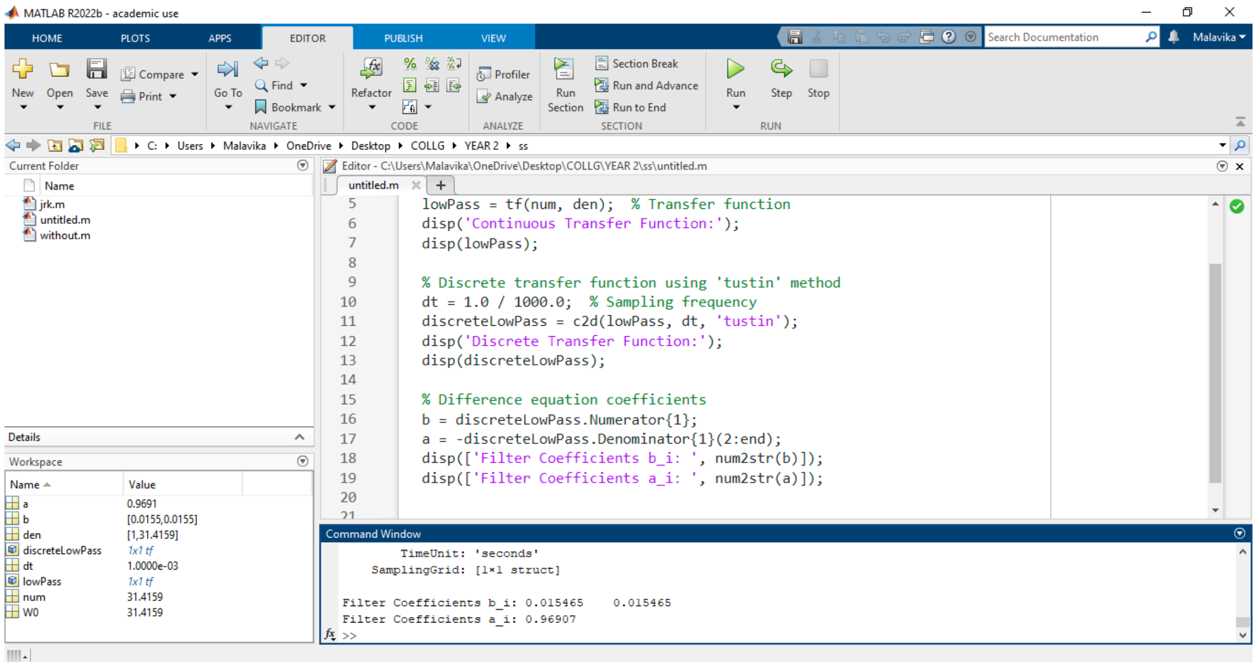
Code



Resulting Coefficients:





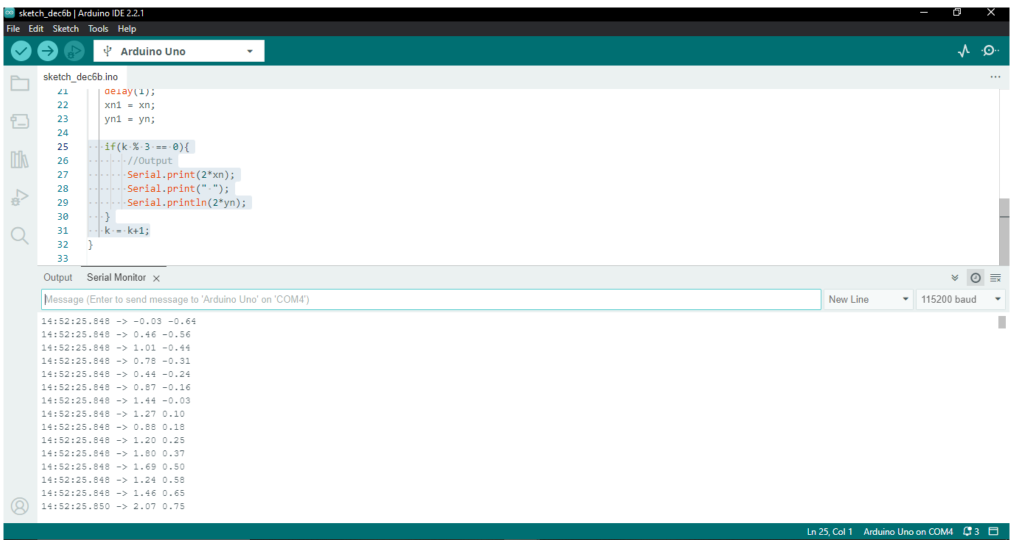


Arduino code and plot

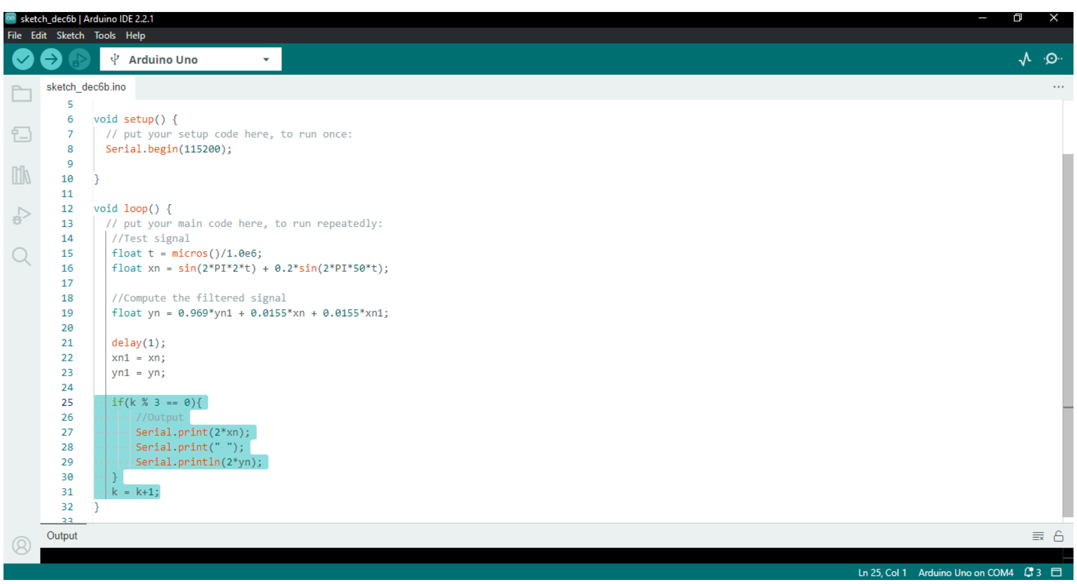
```
float xn1 = 0;
float yn1 = 0;
int k = 0;


void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);

}

void loop() {
  // put your main code here, to run repeatedly:
   //Test signal
   float t = micros()/1.0e6;
   float xn = sin(2*PI*2*t) + 0.2*sin(2*PI*50*t);

   //Compute the filtered signal
   float yn = 0.969*yn1 + 0.0155*xn + 0.0155*xn1;

   delay(1);
   xn1 = xn;
```

```
    yn1 = yn;

    if(k % 3 == 0){
      //Output
      Serial.print(2*xn);
      Serial.print(" ");
      Serial.println(2*yn);
    }
    k = k+1;
  }
```

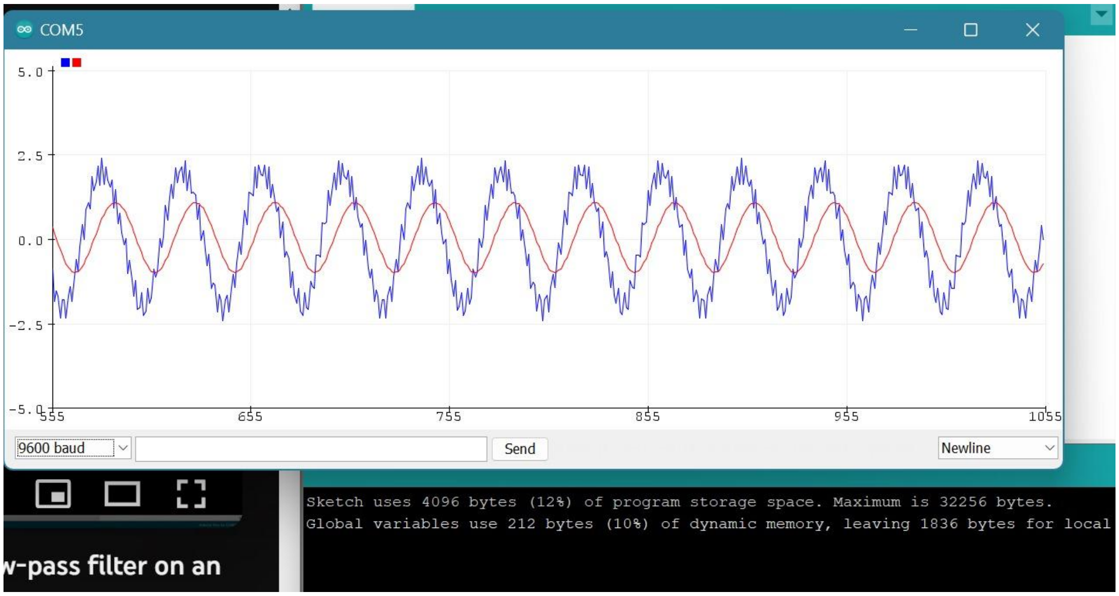
Serial Monitor Output



Code



Graphs Obtained



6) LOW PASS FILTER WITH AUDIO INPUT

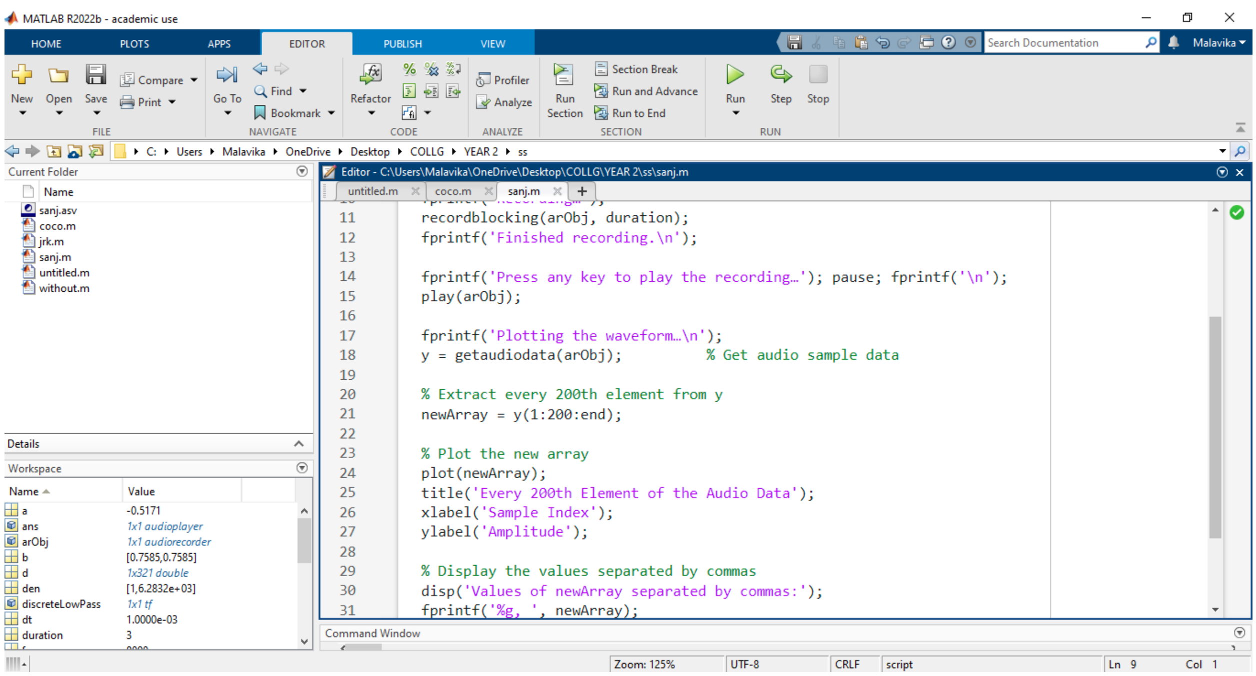
We have used Matlab to take real-time audio for a duration of 3 seconds and plotted the graph and generated the points of the waveform with a sampling frequency of 8000.
Then we created a new array with every 200th element from the waveform.
This new array was then used with a Low Pass Filter code with a cut-off frequency of 1000Hz on Arduino IDE to filter out the elements above this cut-off frequency from the real-time audio and get a final waveform.
Low pass filter has been used as the filter because it helps smoothen the waveform and remove short-term fluctuations while maintaining the long term trend.

Matlab code to generate the audio

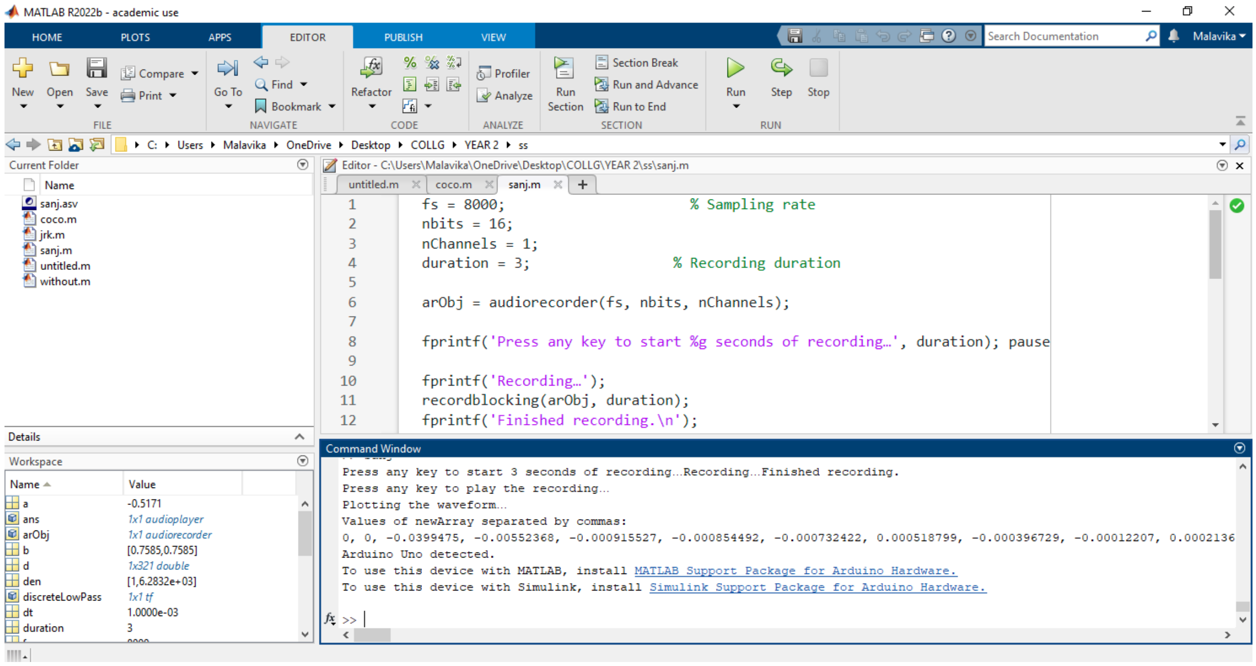
```
fs = 8000;                      % Sampling rate
nbits = 16;
nChannels = 1;
duration = 3;                   % Recording duration
arObj = audiorecorder(fs, nbits, nChannels);
fprintf('Press any key to start %g seconds of recording…', duration); pause
fprintf('Recording…');
recordblocking(arObj, duration);
```

```
fprintf('Finished recording.\n');
fprintf('Press any key to play the recording…'); pause; fprintf('\n');
play(arObj);
fprintf('Plotting the waveform…\n');
y = getaudiodata(arObj);          % Get audio sample data
% Extract every 200th element from y
newArray = y(1:200:end);
% Plot the new array
plot(newArray);
title('Every 200th Element of the Audio Data');
xlabel('Sample Index');
ylabel('Amplitude');
% Display the values separated by commas
disp('Values of newArray separated by commas:');
fprintf('%g, ', newArray);
fprintf('\n');
```

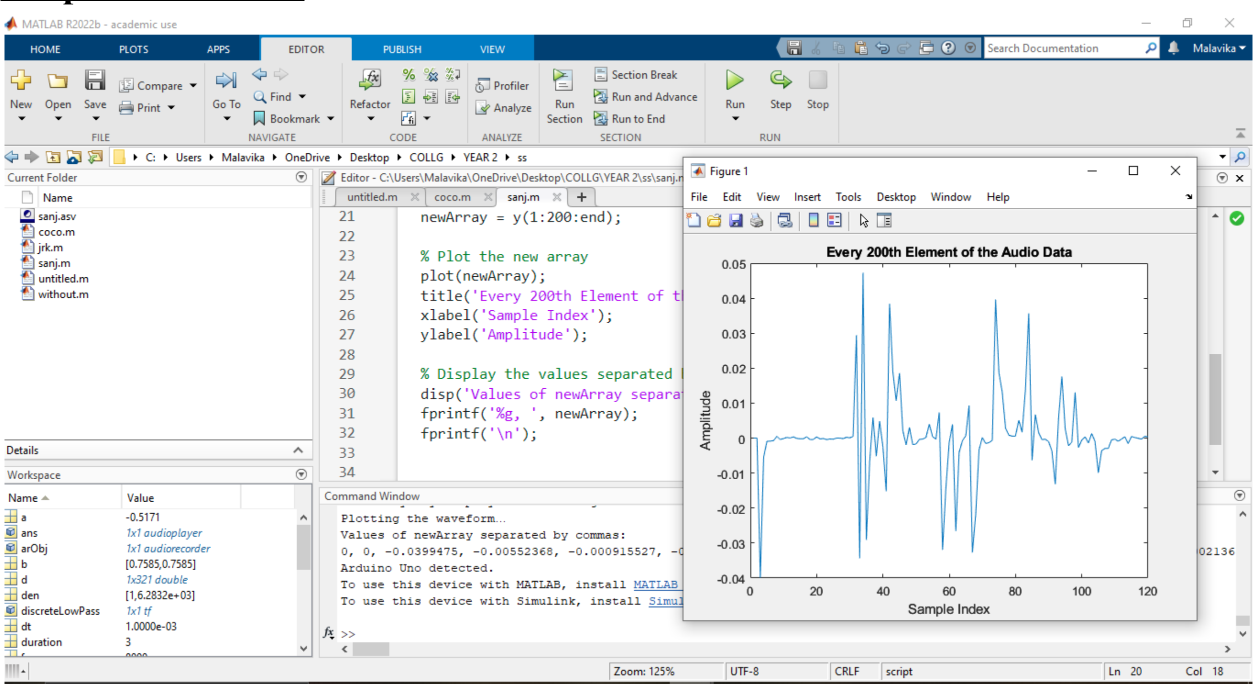
<u>Code</u>



Array of the Audio Recording



Output audio wave

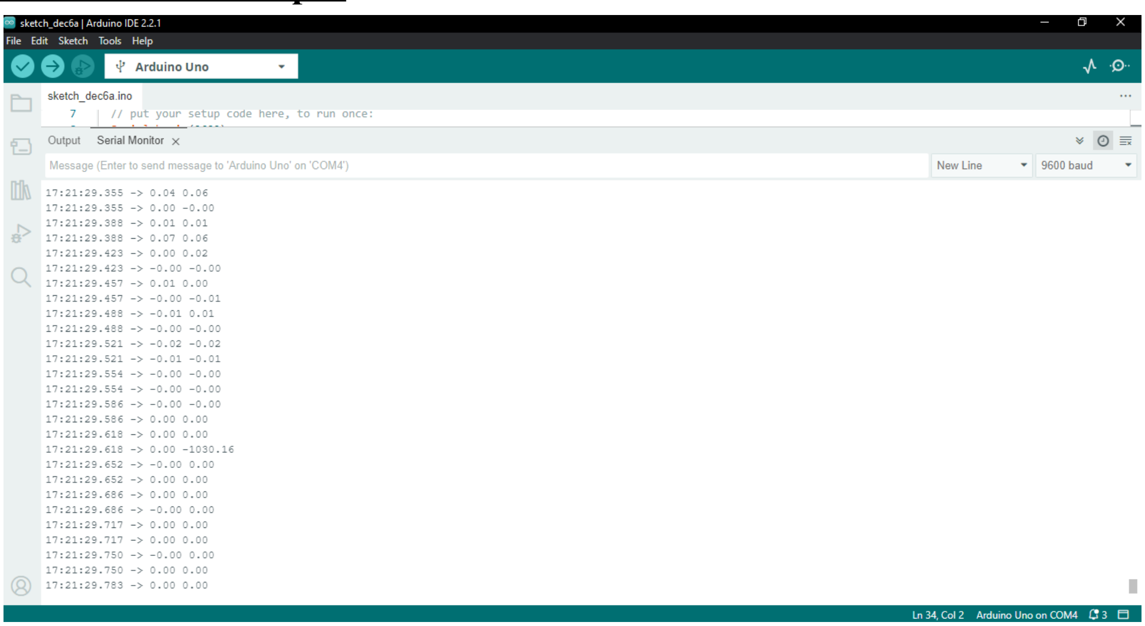


Arduino Code for Low Pass filter

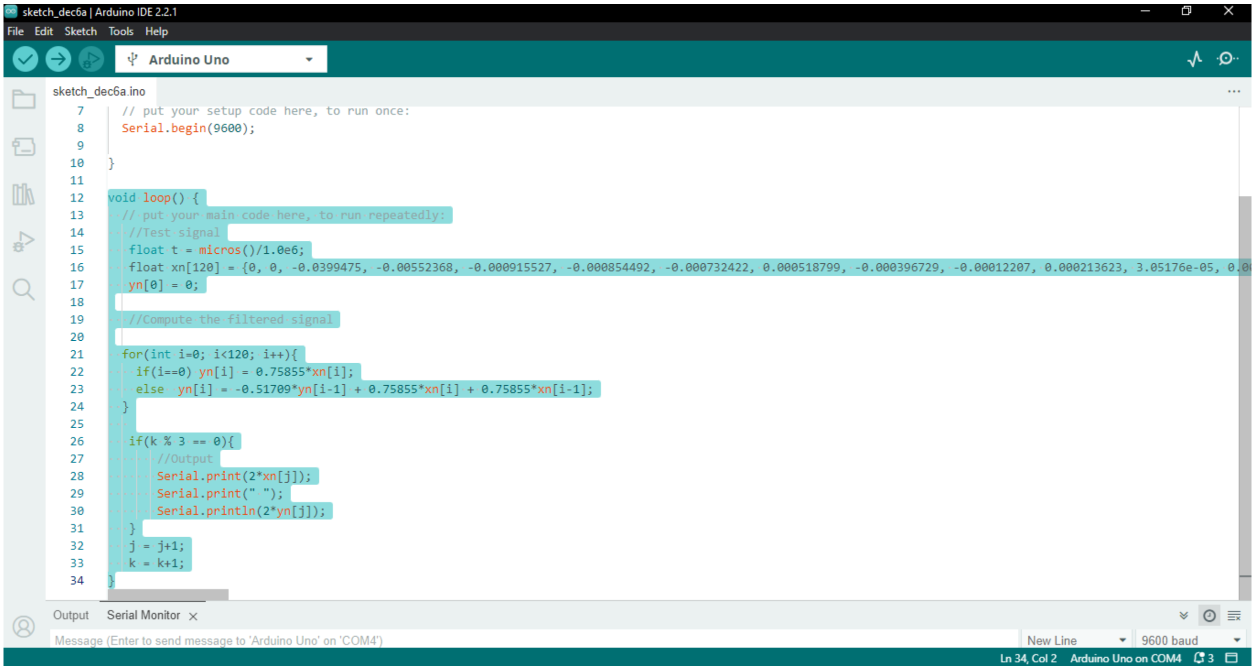
```
float yn[120];
int k = 0;
int j = 0;



void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);

}

void loop() {
  // put your main code here, to run repeatedly:
   //Test signal
   float t = micros()/1.0e6;
   float xn[120] = {0, 0, -0.0399475, -0.00552368, -0.000915527, -0.000854492, -0.000732422,
0.000518799, -0.000396729, -0.00012207, 0.000213623, 3.05176e-05, 0.000366211,
-9.15527e-05, -0.000183105, -0.000183105, 0.000396729, -0.000427246, -0.000396729,
0.000366211, -0.000244141, -0.00012207, -0.000427246, -0.000244141, -0.000305176, 0, 0,
-0.000152588, 0.000244141, 9.15527e-05, 0.000488281, 0.0294189, -0.0343628, 0.0472717,
-0.0291138, -0.0067749, 0.00588989, -0.00515747, 0.00485229, -0.00250244, -0.0151978,
0.0384216, 0.0189514, 0.0106812, 0.0185852, 0.00216675, -0.00192261, 0.00305176,
-0.00186157, -0.00167847, -0.000366211, -0.000274658, 0.000213623, 0.00396729,
0.000396729, -0.000274658, 0.00735474, -0.0319214, -0.0142517, -0.00125122, 0.00393677,
-0.0265808, -0.00405884, -0.000732422, 0.000671387, 0.00933838, -0.0327148, -0.0218811,
-0.00335693, 0.000152588, -0.00152588, -0.00128174, -0.000671387, 0.0396423, 0.0186768,
0.0132141, 0.00286865, 0.000854492, 0.000549316, 0.000579834, 0.00509644, 0.00167847,
0.0137024, 0.035675, -0.00631714, 0.00674438, 0.00152588, -0.000427246, -0.000305176,
-0.00100708, -0.00369263, -0.0131836, 0.00570679, 0.0176086, 0.00268555, -0.00216675,
-0.00115967, 0.0130615, -0.00271606, -0.000579834, 0.000366211, -0.00140381, 0.00131226,
-0.000946045, -0.00991821, -0.00360107, -0.00292969, -0.00292969, -0.000488281,
```

```
-0.000305176, -0.00088501, -0.000274658, 0.000396729, -0.0015564, 0.000488281,
0.000152588, -3.05176e-05, -0.000305176, 0.000549316, 0.000335693};
  yn[0] = 0;

  //Compute the filtered signal

 for(int i=0; i<120; i++){
   if(i==0) yn[i] = 0.75855*xn[i];
   else  yn[i] = -0.51709*yn[i-1] + 0.75855*xn[i] + 0.75855*xn[i-1];
 }

  if(k % 3 == 0){
     //Output
     Serial.print(2*xn[j]);
     Serial.print(" ");
     Serial.println(2*yn[j]);
  }
  j = j+1;
  k = k+1;
}
```

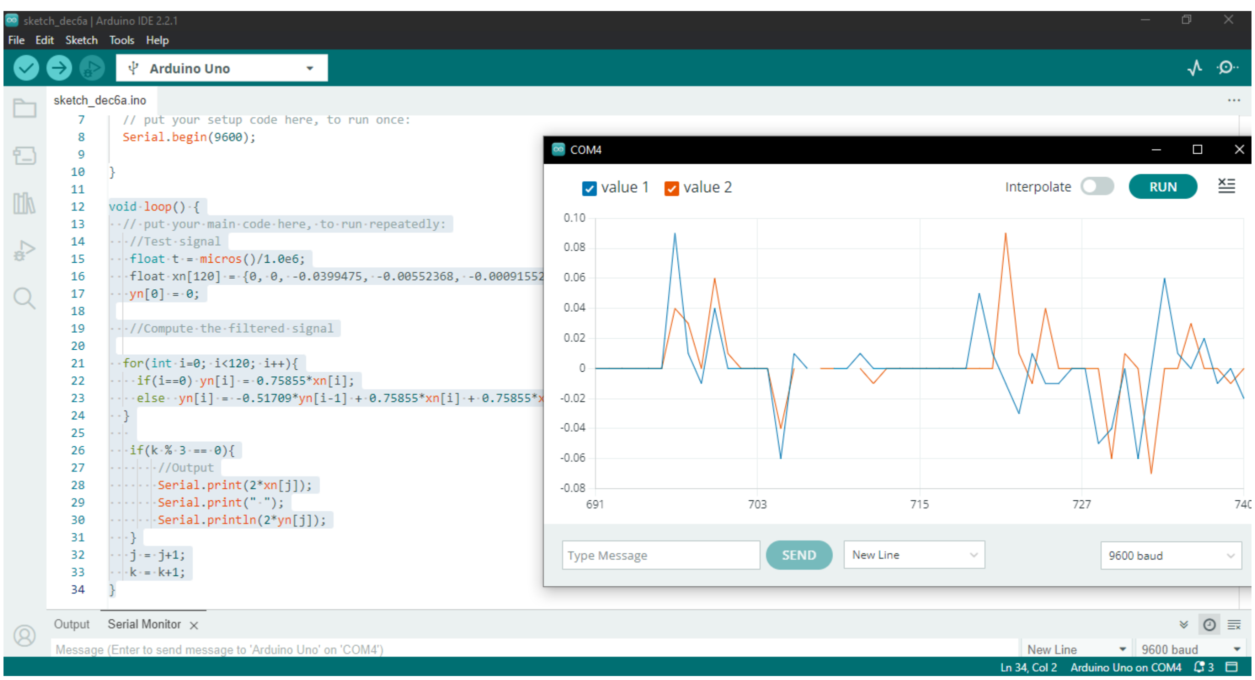
Serial monitor output



Code


sketch_dec6a | Arduino IDE 2.2.1
File Edit Sketch Tools Help
Arduino Uno
sketch_dec6a.ino
// put your setup code here, to run once:
Serial.begin(9600);
}
void loop() {
// put your main code here, to run repeatedly:
//Test signal
float t = micros()/1.0e6;
float xn[120] = {0, 0, -0.0399475, -0.00552368, -0.000915527, -0.000854492, -0.000732422, 0.000518799, -0.000396729, -0.00012207, 0.000213623, 3.05176e-05,
yn[0] = 0;
//Compute the filtered signal
for(int i=0; i<120; i++){
if(i==0) yn[i] = 0.75855*xn[i];
else yn[i] = -0.51709*yn[i-1] + 0.75855*xn[i] + 0.75855*xn[i-1];
}
if(k % 3 == 0){
//Output
Serial.print(2*xn[j]);
Serial.print(" ");
Serial.println(2*yn[j]);
}
j = j+1;
k = k+1;
}
Output
Serial Monitor
Message (Enter to send message to 'Arduino Uno' on 'COM4')
New Line
9600 baud
Ln 34, Col 2 Arduino Uno on COM4

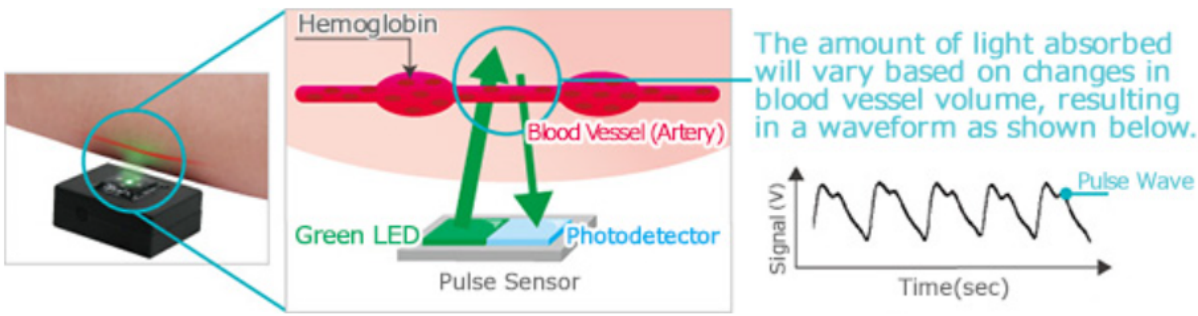
Graph


COM4
value 1
value 2
Interpolate
RUN
0.10
0.08
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
691
703
715
727
740
Type Message
SEND
New Line
9600 baud

7) FILTERING OF PULSE SIGNAL

The sensor used in this project is a Reflection-type pulse sensor. It emits an infrared, red or green light, and it checks the amount of light that is reflected (measured by a photodiode or a phototransistor) to measure the heartbeat. When oxygenated blood is carried to the organs by haemoglobin, the amount of light reflected is different as the oxygen present has a tendency to absorb the incident light. This takes place with time and each time that the amount of absorption increases, it is considered to be one pulse.

These sensors are very prone to error due to interference of any infrared rays (due to sunlight) or red or green lights.

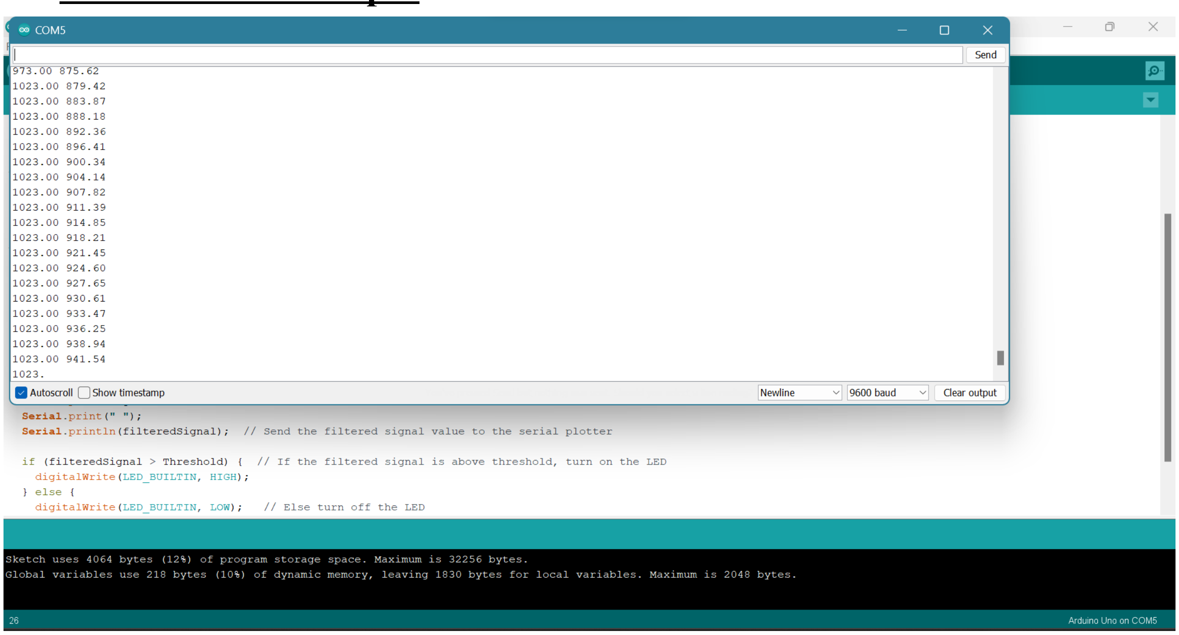


Code

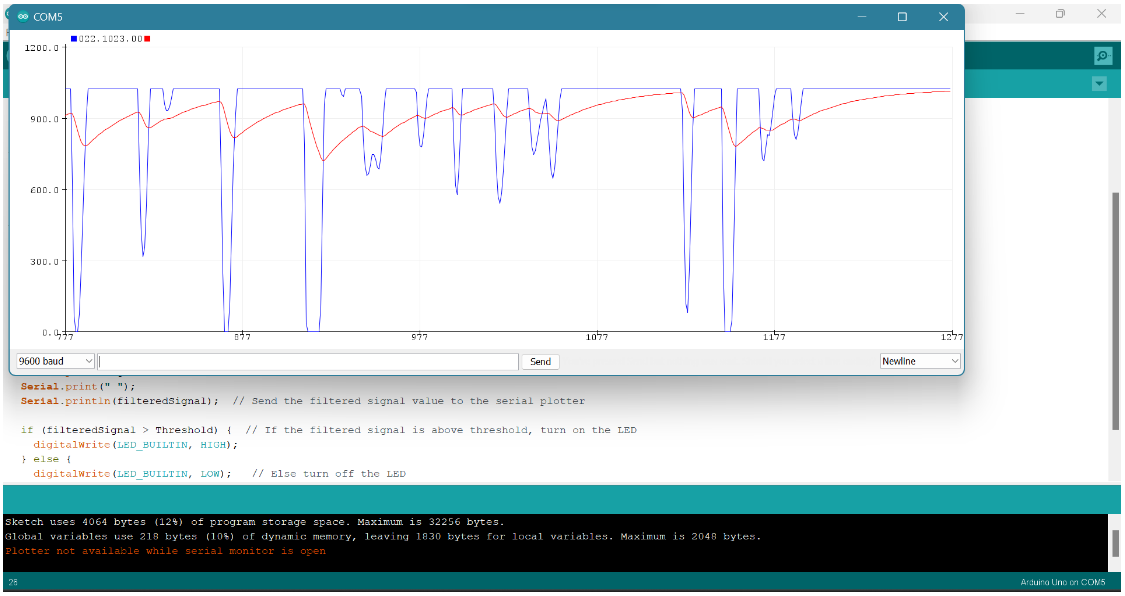
```
const int PULSE_SENSOR_PIN = 0;   // 'S' Signal pin connected to A0

float Signal;                // Store incoming ADC data. Value can range from 0-1024
float Threshold = 550; // Determine which Signal to "count as a beat" and which to
ignore.
float fil;
float sgnl;

const float alpha = 0.1;    // Smoothing factor for the IIR filter

float filteredSignal = 0;   // Variable to store the filtered signal

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);  // Built-in LED will blink to your heartbeat
  Serial.begin(9600);           // Set comm speed for serial plotter window
}
```

```
void loop() {
  Signal = analogRead(PULSE_SENSOR_PIN); // Read the sensor value

  // Apply the IIR filter
  filteredSignal = 0.969*fil + 0.0155*Signal + 0.0155*sgnl;

  delay(1);
  fil = filteredSignal;
  sgnl = Signal;

  Serial.print(Signal);
  Serial.print(" ");
  Serial.println(filteredSignal);  // Send the filtered signal value to the serial plotter

  if (filteredSignal > Threshold) {  // If the filtered signal is above threshold, turn on the
LED
    digitalWrite(LED_BUILTIN, HIGH);
  } else {
    digitalWrite(LED_BUILTIN, LOW);   // Else turn off the LED
  }

  delay(10);
}
```

Serial Monitor Output



Graph Obtained



RESULT

Thus, we have successfully implemented

1) Convolution
2) DFT of a continuous time signal
3) DIT FFT of a continuous time signal
4) FFT using Hanning window
5) IIR filter to remove a noise 50 Hz from 2 Hz signal
6) Low Pass Filter for Audio Signal
7) Filtering of Pulse Signal

using Arduino Uno and plotted their respective graphs. We have also used Matlab to plot the noisy signal in time and frequency domain and generate their coefficients.

APPLICATIONS

There are endless applications of convolution, DFT, FFT and filters in Digital Signal Processing. One major advantage of using Arduino Uno to implement these concepts, is that Arduino Uno is much cheaper than a DSP processor or Arduino Nano, Due etc.
It is easily accessible in many homes and schools across the world. Arduino is easy to use and has rapid prototyping capabilities which make it suitable for quick development and testing of DSP algorithms.

Convolution finds extensive applications across diverse industries, including biomedical, telecommunications, image and video processing, as well as aerospace and defense. Its use is prominent in critical areas such as biomedical signal analysis for electrocardiography (ECG) and medical imaging, channel modeling and signal processing in telecommunications, and advanced techniques within image, video, and audio processing domains.

DFT has numerous applications such as in solving partial differential equations, to perform operations such as convolution, multiplication of large integers, etc.
DFT and FFT can also be used in audio and image processing for compression and enlargement.

<u>REFERENCES</u>

1. Vostrukhin, A., & Vakhtina, E. (2016). Studying Digital Signal Processing on Arduino-Based Platform. Engineering for Rural Development Conference, Jelgava, Latvia. Stavropol Technological Institute of Service, Russia; Stavropol State Agrarian University, Russia. https://www.tf.lbtu.lv/conference/proceedings2016/Papers/N043.pdf

2. Salivahanan, S.(2015). *Digital Signal Processing* (IIIrd ed.). McGraw Hill Education (India) Private Limited

3. https://www.circuitbasics.com/how-to-use-microphones-on-the-arduino/

4. https://in.mathworks.com/help/audio/gs/audio-input-and-audio-output.html