DSP PROJECT

National Institute of Technology Warangal
2021-2025

Audio Filtering using Arduino Nano 33 BLE

Submitted to:

Dr. J Ravi Kumar (Professor, Department of ECE)

Submitted by:

- G S R Raju (21ECB0B20)
- Praveen Kumar(21ECB0B31)
- Ch Sumanth (21ECB0B11)

TABLE OF CONTENTS:

Abstract:

Audio processing with low-pass filters in digital signal processing (DSP) involves the selective manipulation of frequency components within an audio signal. High-pass filters permit the passage of low-frequency content while attenuating or eliminating higher frequencies above a designated cutoff. This paper provides an overview of key concepts associated with low-pass filters, including their frequency response, cutoff frequency, filter order, transfer function, and implementation techniques such as Butterworth and Infinite Impulse Response (IIR) designs. The applications of low-pass filters in audio processing, such as noise reduction, equalization, and speaker protection, are discussed. Overall, the utilization of low-pass filters in DSP offers a powerful means of shaping and enhancing audio signals by effectively managing frequency content.

Introduction:

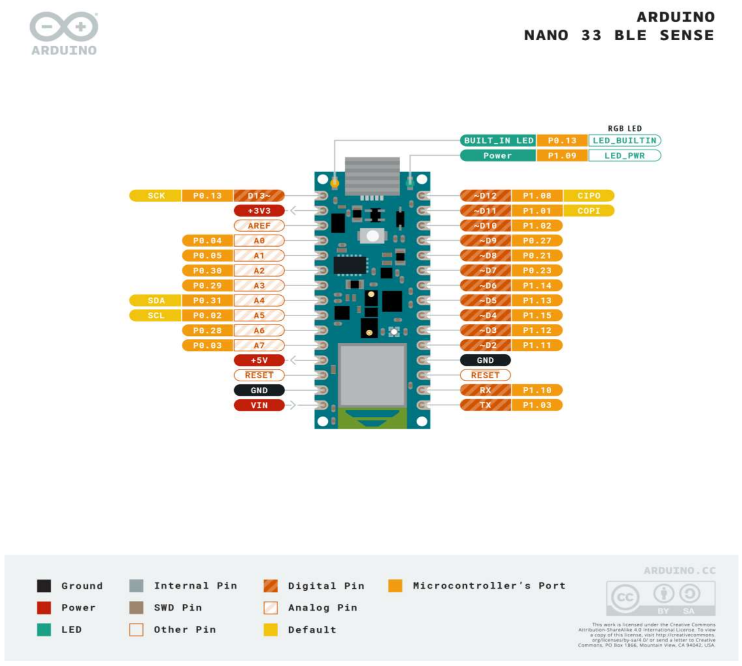
Audio processing is a fundamental aspect of signal processing, playing a pivotal role in various applications such as music production, telecommunications, and speech recognition. Butterworth filters, specifically Infinite Impulse Response (IIR) is widely employed in audio processing to shape, filter, or modify the frequency content of audio signals. This introduction explores the use of Butterworth filters in the context of audio processing. The combination of IIR Butterworth filters provides a powerful toolkit for audio engineers and developers working on applications ranging from basic audio equalization to sophisticated audio effects processing. Whether it's crafting the tonal balance of a musical piece or removing unwanted noise from a recording, the versatility of Butterworth filters makes them a valuable asset in the realm of audio processing. In conclusion, the use of IIR Butterworth filter in audio processing exemplifies the marriage of theoretical filter design with practical applications.

As technologies continue to advance, these filters remain at the forefront of shaping and enhancing the auditory experience across a diverse range of applications.

Arduino BLE – Bluetooth Low Energy:

The Arduino BLE (Bluetooth Low Energy) board represents a cutting-edge evolution in the world of embedded electronics and connectivity. Arduino, a prominent player in the open-source hardware community, has introduced this innovative board to empower developers, hobbyists, and engineers with seamless wireless communication capabilities.

The Arduino BLE board is equipped with the versatility and user-friendly features that have become synonymous with the Arduino platform. It integrates seamlessly with the Arduino IDE, enabling a familiar and accessible development environment for both beginners and experienced developers. With a range of input and output pins, the board facilitates the connection of various sensors, actuators, and peripherals, making it a versatile solution for a wide range of projects.



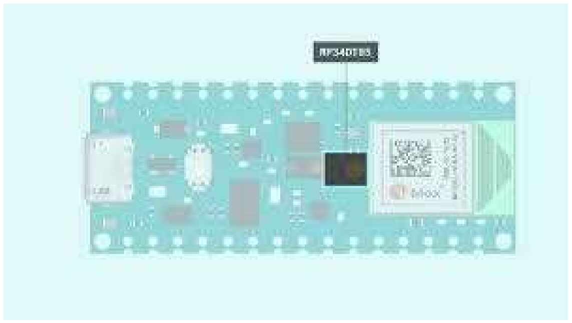
One of the key advantages of the Arduino BLE board is its ability to enable wireless communication between devices in a power-efficient manner. This is achieved through the use of Bluetooth Low Energy technology, allowing for reliable and continuous data exchange while minimizing energy consumption. As a result, the Arduino BLE board is well-suited for battery-powered applications, ensuring longevity and sustainability in projects where power constraints are a primary consideration.

PDM – (Pulse density modulation):

The pulse density modulation (PDM) module enables input of pulse density modulated signals from external audio frontends, for example, digital microphones. The PDM module generates the PDM clock and supports single-channel or dual-channel (Left and Right) data input. Data is transferred directly to RAM buffers using EasyDMA.

PDM libraries are crucial for applications where low- power and high-quality audio processing are essential, such as in portable devices and IoT (Internet of Things) applications. These libraries often support various microcontroller platforms, making them versatile for a range of embedded systems. Developers can leverage PDM libraries to efficiently manage PDM data, enabling the implementation of advanced audio features in their applications. As PDM continues to gain popularity in modern audio systems, PDM libraries play a vital role in simplifying the handling and manipulation of PDM audio streams.

- Here we use 256 samples of input audio per each loop execution using PDM.



FDA TOOL:

The FDA (Filter Design and Analysis) Tool in MATLAB is a powerful tool for designing and analyzing digital filters. To design a low-pass Butterworth IIR filter using the FDA Tool and obtain its coefficients, you can follow these steps:

Step 1: Open the FDA Tool

Open MATLAB and type `fdatool` in the command window to launch the FDA Tool.

Step 2: Choose Filter Type and Specifications

1. In the FDA Tool, select "Design" from the menu and choose "Filter Design" from the dropdown.

2. Choose "Lowpass" as the filter type.

Step 3: Specify Filter Specifications

1. Enter the desired specifications for your low-pass Butterworth filter, including the passband frequency, stopband frequency, and other parameters.

Step 4: Design the Filter

1. Click on the "Design Filter" button.

2. The FDA Tool will generate a filter design based on the specifications you provided.

Step 5: Analyse the Filter

1. After designing the filter, you can analyse its frequency response, phase response, and other characteristics using the available tools in the FDA Tool.

Step 6: Export Filter Coefficients

1. To obtain the filter coefficients, go to the "Export" menu and select "Export Filter Coefficients."

2. Choose the appropriate format for exporting coefficients, such as MATLAB, C, or other supported formats.

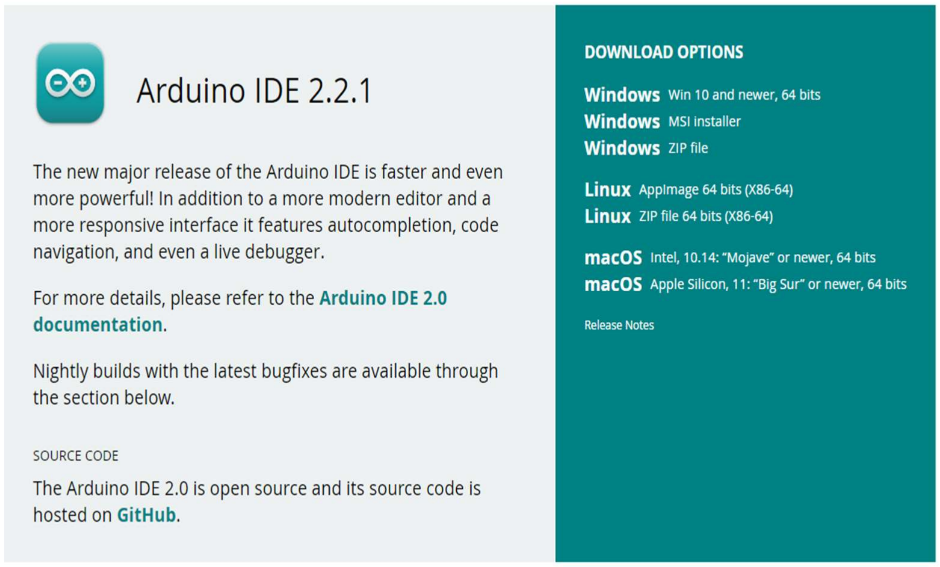
3. Save the coefficients to a file or copy them to the MATLAB workspace.

Step 7: Implement the Filter in MATLAB

Once you have the filter coefficients, you can use them to implement the filter in Arduino Nano 33 BLE using arduino IDE.

Software's Used:

<u>Arduino IDE:</u>



Arduino IDE 2.2.1

The new major release of the Arduino IDE is faster and even more powerful! In addition to a more modern editor and a more responsive interface it features autocompletion, code navigation, and even a live debugger.

For more details, please refer to the **Arduino IDE 2.0 documentation**.

Nightly builds with the latest bugfixes are available through the section below.

SOURCE CODE

The Arduino IDE 2.0 is open source and its source code is hosted on **GitHub**.

DOWNLOAD OPTIONS

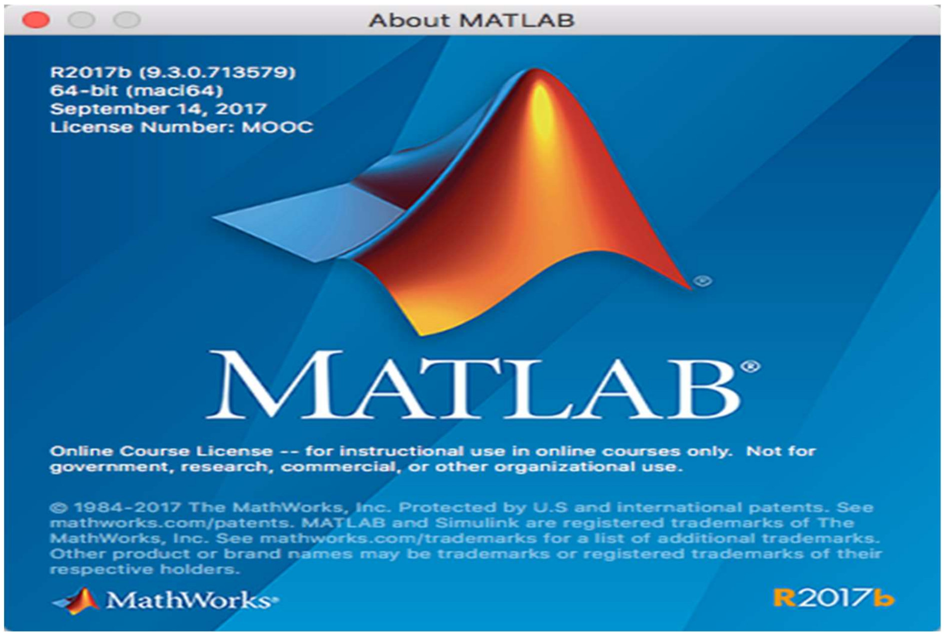
Windows Win 10 and newer, 64 bits
Windows MSI installer
Windows ZIP file

Linux AppImage 64 bits (X86-64)
Linux ZIP file 64 bits (X86-64)

macOS Intel, 10.14: "Mojave" or newer, 64 bits
macOS Apple Silicon, 11: "Big Sur" or newer, 64 bits

Release Notes

<u>MATLAB:</u>



About MATLAB

R2017b (9.3.0.713579)
64-bit (maci64)
September 14, 2017
License Number: MOOC

MATLAB®

Online Course License -- for instructional use in online courses only. Not for government, research, commercial, or other organizational use.

© 1984-2017 The MathWorks, Inc. Protected by U.S and international patents. See mathworks.com/patents. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

MathWorks®

R2017b

Infinite Impulse Response Filter:

An Infinite Impulse Response (IIR) filter is a type of digital filter used in signal processing applications. Unlike Finite Impulse Response (FIR) filters, IIR filters have feedback, allowing the output to depend not only on current inputs but also on past outputs. This characteristic enables IIR filters to achieve similar filtering effects as FIR filters with fewer coefficients, making them computationally efficient.

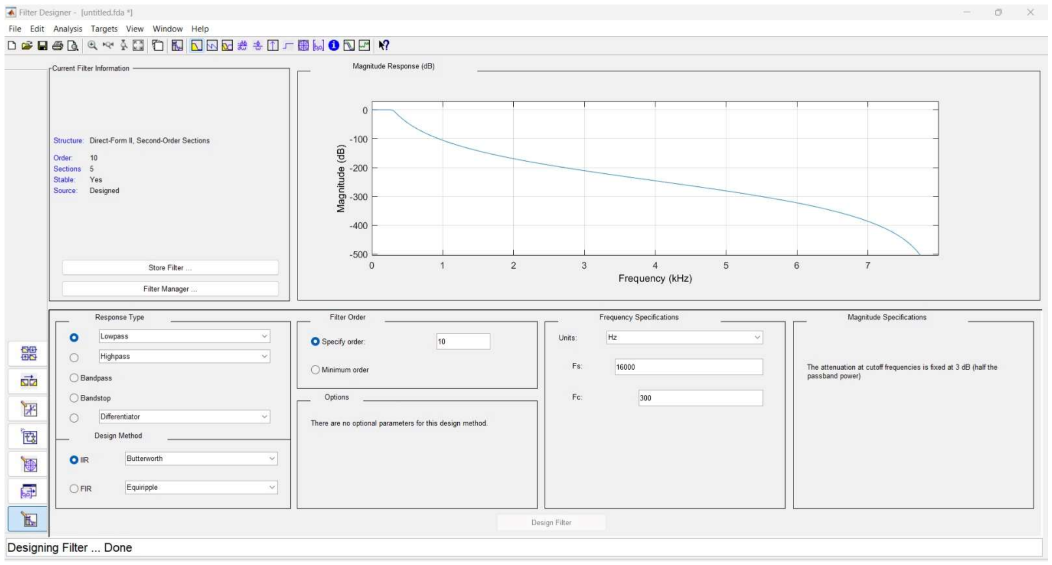
IIR filters are characterized by their transfer function, which describes the relationship between the input and output signals. The transfer function typically includes both numerator and denominator polynomials, representing the feedforward and feedback components, respectively.

One key feature of IIR filters is their ability to exhibit resonant behaviour, emphasizing specific frequencies in the signal. This resonance can be advantageous in applications like audio equalization. However, it also introduces challenges such as potential instability, requiring careful design to maintain filter performance.

IIR filters come in various types, including Butterworth, Chebyshev, and elliptic filters, each with distinct frequency response characteristics. Design parameters such as cutoff frequency, filter order, and ripple influence the filter's behaviour and performance.

While IIR filters offer computational efficiency, their potential for instability and phase distortion should be considered during design. Proper analysis and design techniques are crucial to achieving the desired filtering characteristics without compromising stability .

- The filter used is a cascaded low pass filter of:
  - Order : 10
  - Cutoff frequency : 300 Hz
  - Pass Band gain:3db
  - Stop Band attenuation:60db



Current Filter Information

Structure:	Direct-Form II, Second-Order Sections
Order:	10
Sections	5
Stable:	Yes
Source:	Designed

Response Type

[✓ Lowpass]
Highpass
Bandpass
Bandstop
Differentiator

Design Method

[✓ IIR] Butterworth
FIR Equiripple

Filter Order

[✓ Specify order:] 10
Minimum order

Options

There are no optional parameters for this design method.

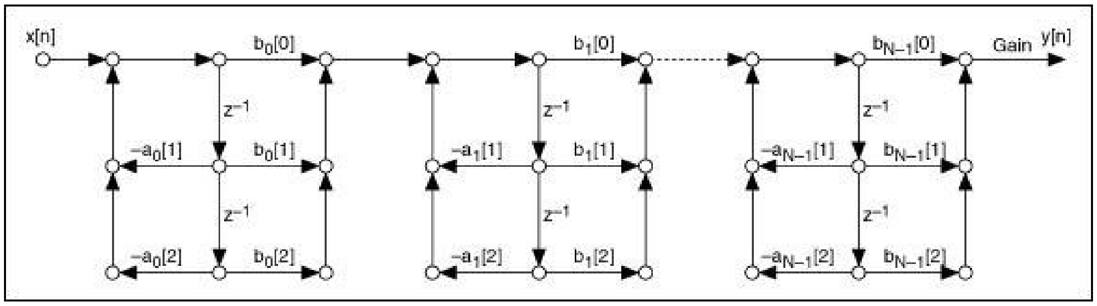
Frequency Specifications

Units:	Hz
Fs:	16000
Fc:	300

Magnitude Specifications

The attenuation at cutoff frequencies is fixed at 3 dB (half the passband power)

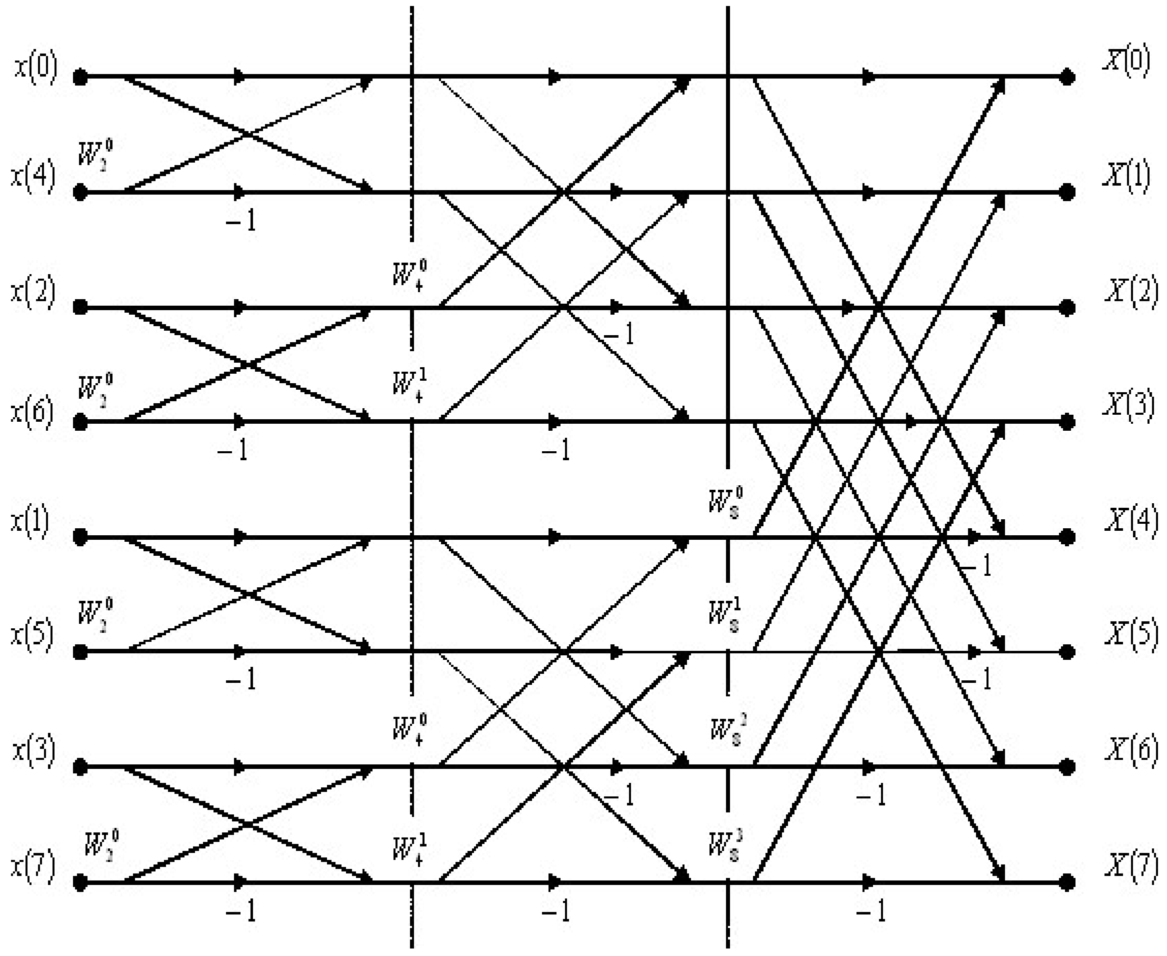
Designing Filter ... Done



Fast Fourier transform:

Fast Fourier Transform (FFT) is a widely used algorithm in signal processing and mathematics for efficiently computing the discrete Fourier transform (DFT) and its inverse. Developed by Cooley and Tukey in the 1960s, FFT significantly reduces the computational complexity of traditional DFT algorithms. This technique is crucial in various applications, including audio processing, image analysis, and communication systems.

The core idea behind FFT is to exploit the symmetry properties of the Fourier transform to reduce the number of computations required.
By recursively breaking down the DFT into smaller sub-problems, FFT achieves a complexity of O (N log N) compared to the O(N^2) complexity of the naive DFT algorithm, making it particularly advantageous for large datasets.



FFT is based on the principle of representing a signal in the frequency domain, providing insights into its spectral components. This is essential in applications such as filtering, where specific frequency components need emphasis or suppression. Understanding FFT is fundamental for engineers and scientists working in fields where signal analysis and processing play a crucial role. Its efficiency and versatility have made FFT a cornerstone in modern computational mathematics and engineering practices.
-> FFT for 256 samples is used.

Algorithmic Steps :

Step 1:

• Take input from inbuilt microphone which is present in Arduino nano BLE 33 sense by playing different sound frequencies.

Step 2:

• Convert them into a one-dimensional array so that it can be given to filters.

Step 3:

• Apply IIR filter to attenuate high frequency content.

• A 1D filtered array will be obtained.

Step 4:

• Apply Fast Fourier Transform to obtain different frequency contents and spectrum.

• Here we apply FFT for 256 samples.

Step 5:

• The Plot the amplitude spectrum for amplitudes obtained using serial plotter.

Applications:

Audio processing is a broad field that involves manipulating and analyzing audio signals for various applications. Here are some common applications of audio processing:

1. Music Production:

i. Equalization (EQ) : Adjusting the balance of frequencies to enhance or modify the sound.

ii. Compression: Controlling the dynamic range of audio to ensure a more consistent output.

iii. Pitch Correction: Correcting or modifying the pitch of vocals or instruments.

2. Speech Processing:

i. Speech Recognition: Converting spoken language into text.

ii. Speech Synthesis: Generating artificial human-like speech.

3. Telecommunications:

i. Audio Coding (e.g., MP3, AAC): Compressing audio data for efficient transmission.

ii. Noise Reduction: Removing unwanted background noise from audio streams.

4. Audio Enhancement:

i. Noise Reduction and Removal: Eliminating unwanted background noise from audio recordings.

ii. Audio Restoration: Improving the quality of old or degraded audio recordings.

5. Entertainment and Gaming:

i. Sound Effects: Adding and manipulating audio effects for enhanced gaming experiences.

ii. Spatial Audio: Creating a sense of direction and space for a more immersive audio experience.

6. Automotive:

i. In-Car Entertainment: Audio processing for in-car music and entertainment systems.

ii. Speech Recognition: Voice control systems for navigation, phone calls, and other vehicle functions.

7. Education:

i. Language Learning: Using speech processing for pronunciation feedback and language learning.

ii. Text-to-Speech (TTS): Converting written text into spoken words for educational materials.

Codes:

Arduino Code: Takes the input audio from Arduino BLE on board microphone, generates the samples and produce the output samples after filtration.

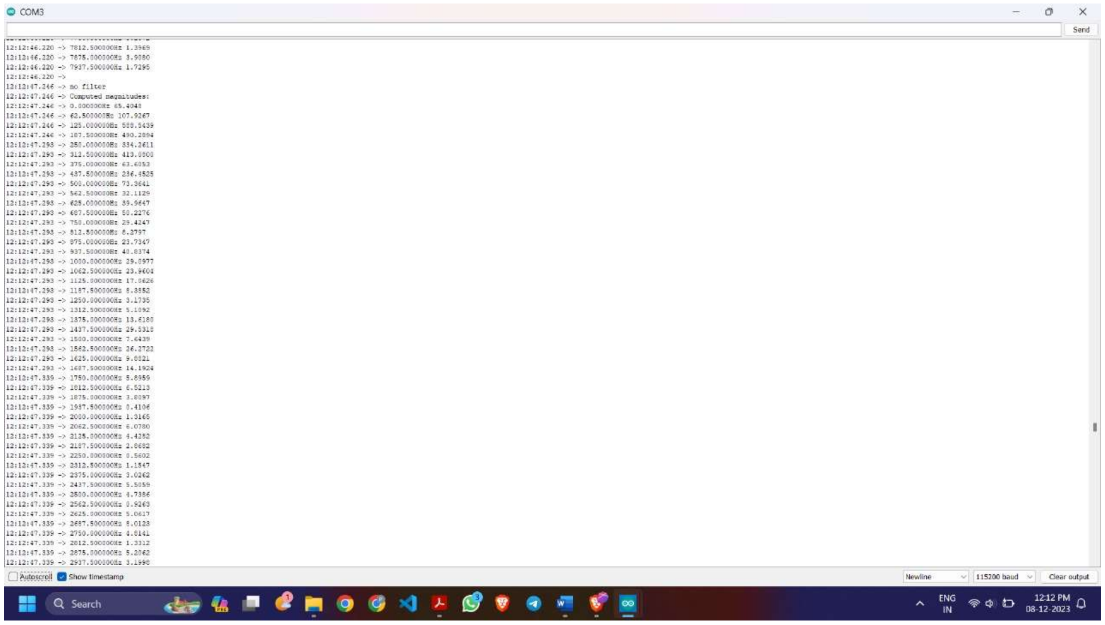
```
#include <PDM.h>
#include "arduinoFFT.h"
const uint16_t samples=256;           //Must be a power of 2
const double sfreq=16000;
short sbuffer[samples];
volatile int samplesRead;
unsigned long microseconds;
```

```
static const int freq = 16000;
double vr[samples];
double vi[samples];
const uint8_t amp = 100;
#define index 0x00
#define tim 0x01
#define freq 0x02
#define plot 0x03
static const char channels = 1;
arduinoFFT FFT;

#define numsamples 256

double x1 = 0.0, x2 = 0.0;
double y_1 = 0.0, y2 = 0.0;

double y[samples];
double a[5][3] = {
	{1, -1.950277359339051885811500142153818160295,
0.963890148438751537263158297719201073O5 },
	{1, -1.88552374206984696414224345062393695116  ,
0.898684555797373163521513106388738378882},
	{1,-1.833732658924647873632807204558048397303 ,
0.84653197479202391129149418702581897378 },
	{1,-1.797853817891356786873302553431130945683 ,
0.81040270215022902622337142020114697516 },
    {1, -1.77954855764198094369987757090711966157  ,
0.791969672562817028627080162550555542111}};


double gain[5]={0.003403197274924847810784189761079687742
,0.00329020343188153379862526115573473362
,0.003199828966843966046584846196765283821
,0.003137221064718014734706841295519552659
,0.003105278730208948807095525879162778438};
double b[5][3] = {
    {1,2,1 },
     {1,2,1 },
      {1,2,1 },
       {1,2,1 },
        {1,2,1 }
  };
double sig_in[numsamples];
double sig_out[numsamples];
void pdmdata(void);
void init() {
  x1 = 0.0;
  x2 = 0.0;
  y_1 = 0.0;
  y2 = 0.0;
}
double iirfilter(double input1,double b0,double b1,double b2,double
a1,double a2,double g) {
  double output = g*(b0 * input1 + b1 * x1 + b2 * x2 - a1 * y_1 - a2 *
y2);
  x2 = x1;
  x1 = input1;
  y2 = y_1;
  y_1 = output;
  return output;
}
void setup() {
  Serial.begin(115200);
  while (!Serial);

  PDM.onReceive(pdmdata);
  if (!PDM.begin(channels, freq)) {
  Serial.println("Failed to start PDM!");
  while (1);
  }
}

void loop() {
  if (samplesRead)
  {
  for (int i = 0; i < samples; i++)
  {
    vr[i] = sbuffer[i];
    vi[i] = 0;
    //Serial.println(vr[i]);
  }
  }
  Serial.println("IIR FILTER");
  for (int i = 0; i < numsamples; i++) {
     sig_in[i] = vr[i];
     sig_out[i]=sig_in[i];


  }


  for(int i=0;i<5;i++)
  {
     init();
     for (int j = 0; j < numsamples; j++)
     {
        sig_out[j] =
iirfilter(sig_out[j],b[i][0],b[i][1],b[i][2],a[i][1],a[i][2],gain[i]);
     }
  }
  for (int j = 0; j < numsamples; j++)
  {
     y[j]=sig_out[j];
  }
     FFT = arduinoFFT(y, vi, samples, sfreq);
      FFT.Windowing(FFT_WIN_TYP_HAMMING, FFT_FORWARD); /*
Weigh data */
```

```
      FFT.Compute(FFT_FORWARD); /* Compute FFT */
      FFT.ComplexToMagnitude(); /* Compute magnitudes */
      Serial.println("Computed magnitudes:");
      printiir(y, (samples >> 1), freq);

    samplesRead = 0;
   delay(1000);
  }
void pdmdata()
{
  int bytesAvailable = PDM.available();
  PDM.read(sbuffer, bytesAvailable);
  samplesRead = bytesAvailable / 2;
}
void printiir(double *vData, uint16_t bufferSize, uint8_t scaleType)
{
  for (uint16_t i = 0; i < bufferSize; i++)
  {
    double absc;
    switch (scaleType)
    {
      case index:
        absc = (i * 1.0);
  break;
      case time:
        absc = ((i * 1.0) / sfreq);
  break;
    case freq:
      absc = ((i * 1.0 * sfreq) / samples);
  break;
   }
   Serial.print(absc, 6);
   if(scaleType==freq)
     Serial.print("Hz");
   Serial.print(" ");
   Serial.println(10000000.00*vData[i], 4);
  }
  Serial.println();
}
```

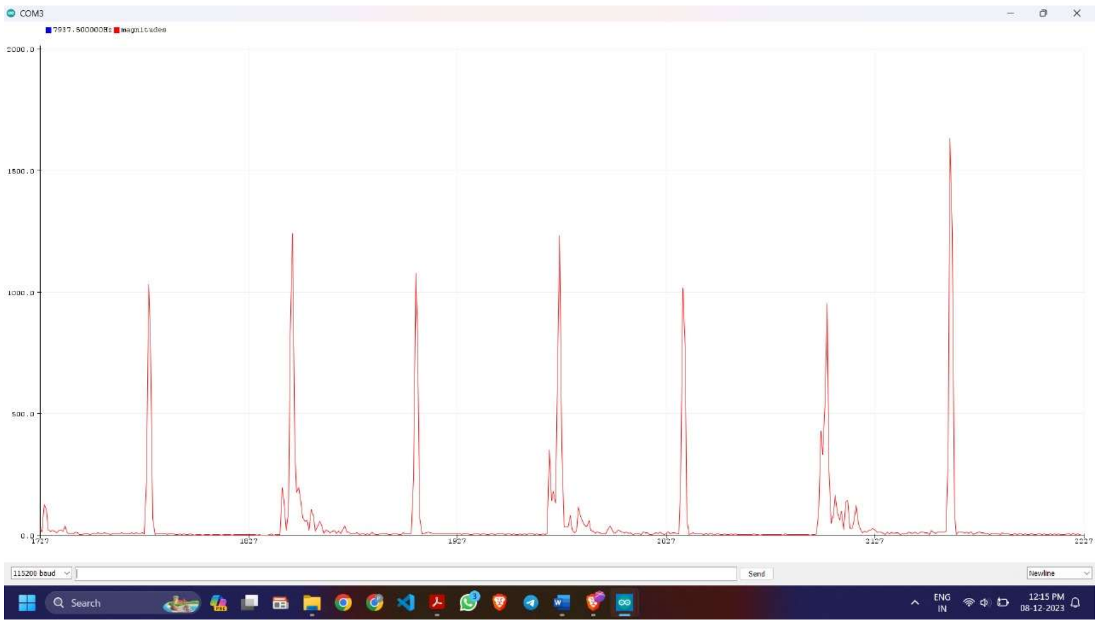
Results:

When 400Hz and 4000Hz frequencies given without applying filter:

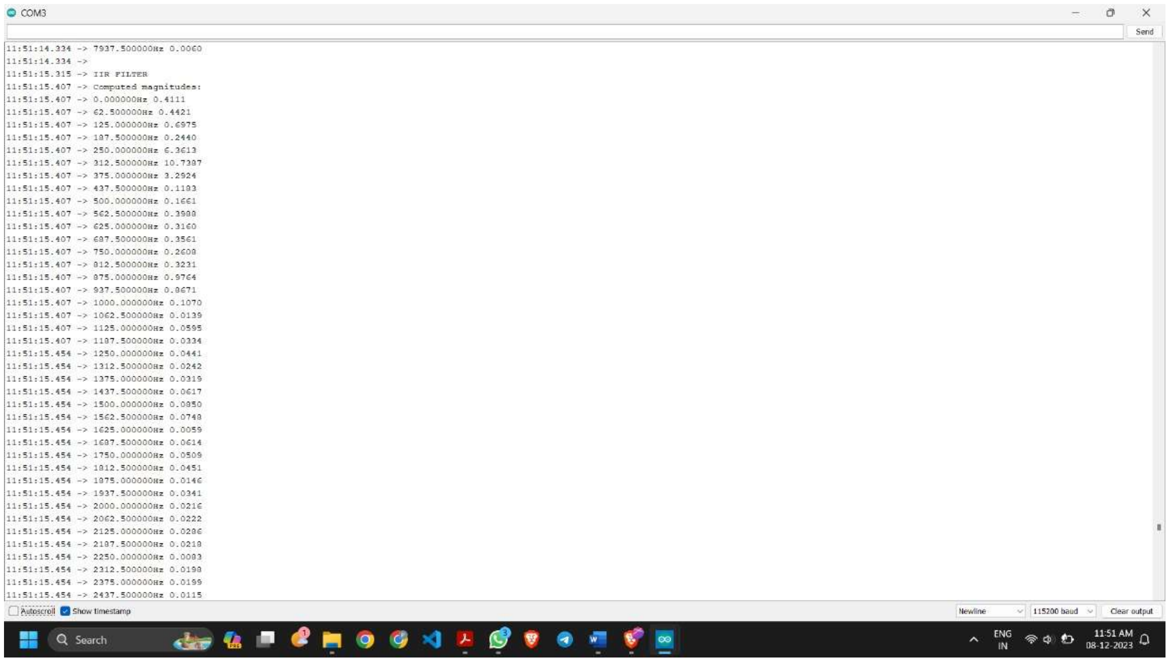


```
12:12:46.220 -> 7812.500000Hz 1.3969
12:12:46.220 -> 7875.000000Hz 3.9080
12:12:46.220 -> 7937.500000Hz 1.7295
12:12:46.220 ->
12:12:47.246 -> no filter
12:12:47.246 -> Computed magnitudes:
12:12:47.246 -> 0.000000Hz 65.4048
12:12:47.246 -> 62.500000Hz 107.9267
12:12:47.246 -> 125.000000Hz 588.5439
12:12:47.246 -> 187.500000Hz 490.2894
12:12:47.293 -> 250.000000Hz 334.2611
12:12:47.293 -> 312.500000Hz 413.0800
12:12:47.293 -> 375.000000Hz 63.6053
12:12:47.293 -> 437.500000Hz 236.4525
12:12:47.293 -> 500.000000Hz 73.3641
12:12:47.293 -> 562.500000Hz 32.1129
12:12:47.293 -> 625.000000Hz 39.9647
12:12:47.293 -> 687.500000Hz 50.2276
12:12:47.293 -> 750.000000Hz 29.4247
12:12:47.293 -> 812.500000Hz 8.2797
12:12:47.293 -> 875.000000Hz 23.7347
12:12:47.293 -> 937.500000Hz 40.8374
12:12:47.293 -> 1000.000000Hz 29.0977
12:12:47.293 -> 1062.500000Hz 23.9604
12:12:47.293 -> 1125.000000Hz 17.0626
12:12:47.293 -> 1187.500000Hz 8.3852
12:12:47.293 -> 1250.000000Hz 3.1735
12:12:47.293 -> 1312.500000Hz 5.1092
12:12:47.293 -> 1375.000000Hz 13.6180
12:12:47.293 -> 1437.500000Hz 29.5318
12:12:47.293 -> 1500.000000Hz 7.6439
12:12:47.293 -> 1562.500000Hz 26.2722
12:12:47.293 -> 1625.000000Hz 9.0821
12:12:47.293 -> 1687.500000Hz 14.1924
12:12:47.339 -> 1750.000000Hz 5.8959
12:12:47.339 -> 1812.500000Hz 6.5213
12:12:47.339 -> 1875.000000Hz 3.8097
12:12:47.339 -> 1937.500000Hz 0.4106
12:12:47.339 -> 2000.000000Hz 1.3165
12:12:47.339 -> 2062.500000Hz 6.0780
12:12:47.339 -> 2125.000000Hz 4.4282
12:12:47.339 -> 2187.500000Hz 2.0682
12:12:47.339 -> 2250.000000Hz 0.5602
12:12:47.339 -> 2312.500000Hz 1.1547
12:12:47.339 -> 2375.000000Hz 3.0262
12:12:47.339 -> 2437.500000Hz 5.5059
12:12:47.339 -> 2500.000000Hz 4.7386
12:12:47.339 -> 2562.500000Hz 0.9263
12:12:47.339 -> 2625.000000Hz 5.0617
12:12:47.339 -> 2687.500000Hz 8.0123
12:12:47.339 -> 2750.000000Hz 4.8141
12:12:47.339 -> 2812.500000Hz 1.3312
12:12:47.339 -> 2875.000000Hz 5.2062
12:12:47.339 -> 2937.500000Hz 3.1998
```

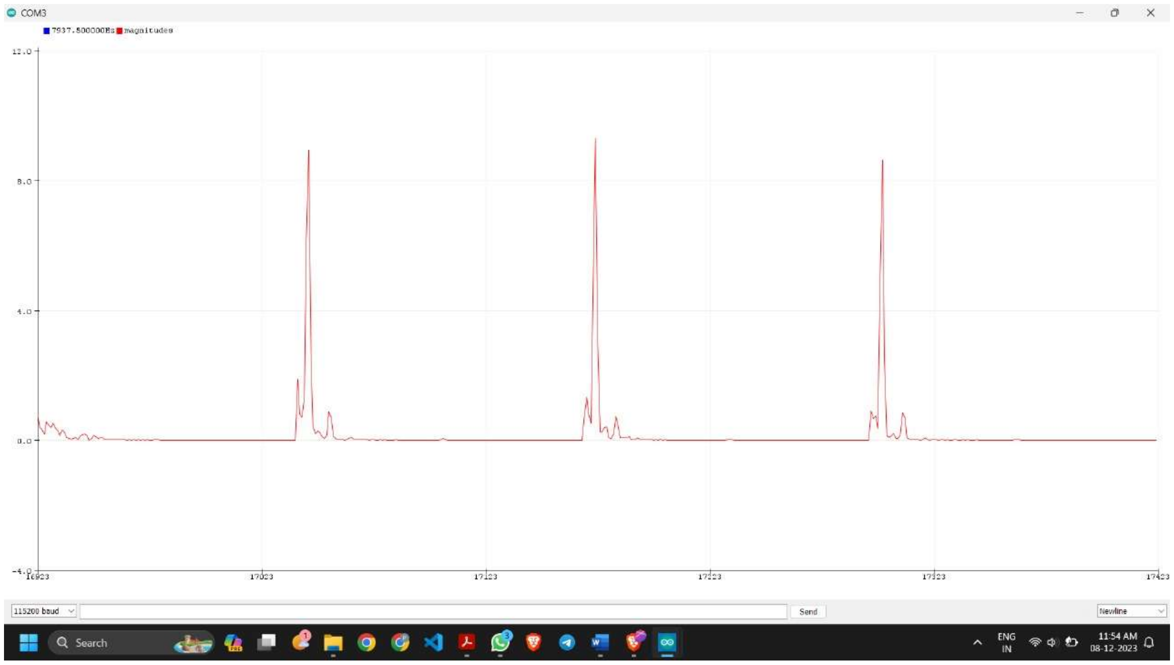
Autoscroll [✓ Show timestamp] Newline 115200 baud



When 400Hz and 4000Hz frequency are given to IIR filter:



```
11:51:14.334 -> 7937.500000Hz 0.0060
11:51:14.334 ->
11:51:15.315 -> IIR FILTER
11:51:15.407 -> Computed magnitudes:
11:51:15.407 -> 0.000000Hz 0.4111
11:51:15.407 -> 62.500000Hz 0.4421
11:51:15.407 -> 125.000000Hz 0.6975
11:51:15.407 -> 187.500000Hz 0.2440
11:51:15.407 -> 250.000000Hz 6.3613
11:51:15.407 -> 312.500000Hz 10.7387
11:51:15.407 -> 375.000000Hz 3.2924
11:51:15.407 -> 437.500000Hz 0.1183
11:51:15.407 -> 500.000000Hz 0.1661
11:51:15.407 -> 562.500000Hz 0.3988
11:51:15.407 -> 625.000000Hz 0.3160
11:51:15.407 -> 687.500000Hz 0.3561
11:51:15.407 -> 750.000000Hz 0.2608
11:51:15.407 -> 812.500000Hz 0.3231
11:51:15.407 -> 875.000000Hz 0.9764
11:51:15.407 -> 937.500000Hz 0.8671
11:51:15.407 -> 1000.000000Hz 0.1070
11:51:15.407 -> 1062.500000Hz 0.0139
11:51:15.407 -> 1125.000000Hz 0.0595
11:51:15.407 -> 1187.500000Hz 0.0334
11:51:15.454 -> 1250.000000Hz 0.0441
11:51:15.454 -> 1312.500000Hz 0.0242
11:51:15.454 -> 1375.000000Hz 0.0319
11:51:15.454 -> 1437.500000Hz 0.0617
11:51:15.454 -> 1500.000000Hz 0.0850
11:51:15.454 -> 1562.500000Hz 0.0748
11:51:15.454 -> 1625.000000Hz 0.0059
11:51:15.454 -> 1687.500000Hz 0.0614
11:51:15.454 -> 1750.000000Hz 0.0509
11:51:15.454 -> 1812.500000Hz 0.0451
11:51:15.454 -> 1875.000000Hz 0.0146
11:51:15.454 -> 1937.500000Hz 0.0341
11:51:15.454 -> 2000.000000Hz 0.0216
11:51:15.454 -> 2062.500000Hz 0.0222
11:51:15.454 -> 2125.000000Hz 0.0286
11:51:15.454 -> 2187.500000Hz 0.0218
11:51:15.454 -> 2250.000000Hz 0.0083
11:51:15.454 -> 2312.500000Hz 0.0198
11:51:15.454 -> 2375.000000Hz 0.0199
11:51:15.454 -> 2437.500000Hz 0.0115
```



Conclusion

This project successfully demonstrated in real-time sound frequency analysis using Arduino Nano BLE 33 Sense. The implementation allows users to capture, filter the external noise and only allows human voice frequencies. The user-friendly interface enables the IIR Butterworth filter and visualizes the amplitude spectrum using the serial plotter.