Microprocessor Lab Project Report

Examining Randomness of Johnson noise using FFT

Ayush Bharadwaj (170260020)

Thoutam Shiva Teja (170260035)

Guidance

Prof. Pradeep Sarin

Department of Physics, IIT Bombay

November 6, 2019

Acknowledgments

We would like to thank Saurabh Mogre, Niladri Chatterji and Karthik Kothari of Batch 2013, Anshul Avasthi, Anushrut Sharma and Rishabh Khandelwal of Batch 2014 for their Excellently documented project, using which as a foundation, we have built ours.
We also thank Prof. Pradeep Sarin for his help throughout the course of this project and Mr. Nitin Pawar for his valuable insights in moments of dire need

1. Introduction

The aim of the project is to Examine the Randomness of the Johnson Noise and use Fast Fourier Transform (FFT) Technique to examine its Randomness. We used resistor as the source of the Johnson noise. We amplify the noise generated and use a microcontroller for processing Fast Fourier Transform (FFT) of the signal and show a Live FFT on a GUI

1.1 Johnson Noise

Johnson–Nyquist noise is the electronic noise generated by the thermal agitation of the charge carriers inside an electrical conductor at equilibrium, which happens regardless of any applied voltage. Thermal noise in an idealistic resistor is approximately white, meaning that the power spectral density is nearly constant throughout the frequency spectrum.

2. Circuitry

For a typical resistor, thermal noise generated will be in the magnitude of μV range, So in order for it to be detected by the microcontroller we use, it should be amplified to around 1 to 2 V range. For that, we used Balanced Wheatstone Bridge (R = 1.2kΩ) as a noise source. The voltage across the terminals should be ideally zero, however, there will be noise generated (in around 50-80μV) which is actually Johnson Noise. We have to amplify this noise. For the Input Voltage, we used IC **MAX 6126** which is ultra-low-noise, high-precision, low dropout voltage reference, so that other Input noise sources won't affect the randomness.

2.1 Two-Stage Amplification

So, for the amplification part, it seems that we have to amplify by a Big number which is around 15k. First thing that comes to mind is we can do it by amplifying 500 times in the first stage and 30 times in the second stage. But it is not that simple, this amplification part needs some attention

By default, we all use **Op-Amp** as the amplifiers in different configurations

The main thing to remember is that we are amplifying a **White noise** whose Power spectral density is almost constant at a Large Frequency Bandwidth. But every Op-Amp has its own Bandwidth limitations in amplification. We have to consider parameter called Gain Bandwidth Product **(GBWP)**, whose formula is given as follows

$$\textbf{GBWP} = \textbf{Gain} \times \textbf{bandwidth}$$

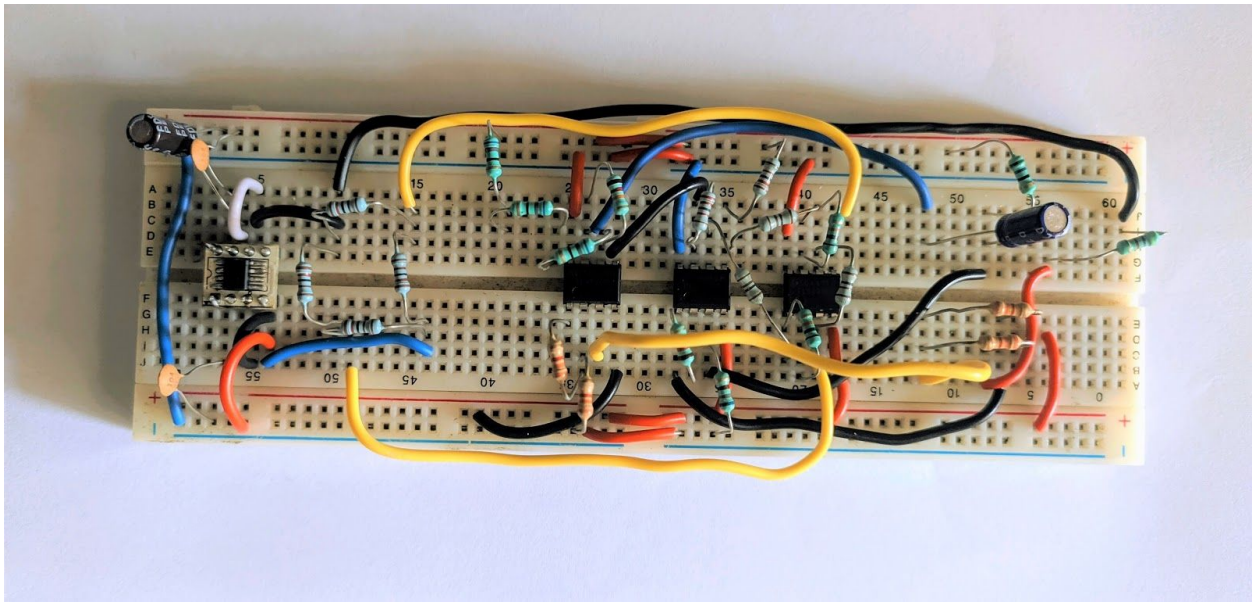
Every Op-Amp is designed with a particular GBWP, so if we increase the Gain the Bandwidth of the output decreases

So Gain at each has to be chosen so that we won't lose this frequency Bandwidth, Considering this we used a **High GBWP Op-Amp MC33078P (16 Mhz)** for the first stage of amplification whose Gain we have chosen is **120 , this** limits the bandwidth to around **120 kHz** (This has been chosen, taking into consideration, Microcontroller's Bandwidth)

We used the **Instrumentation Amplifier Configuration** for the First Stage amplification

In the Second Stage, we used the **Non-Inverting amplifier** of Gain around **6.5**, which is been limited because of the attenuation of the signal

Which amplified our Noise signal from 70 µV to 60 mV



2.2 Filtering of DC offset

There is a small DC component arising from the mismatch in resistor values which gets amplified along with the noise signal. It was filtered out using a high pass cutoff frequency of 60Hz. For which we chose Resistor of R = 560 Ω and Capacitor of C = 4.7 µV Again the values has been chosen taking into consideration the attenuation of the Signal

2.3 Third Stage amplification

The filtered output was then amplified with a Gain of around **17** to give a signal of 1 V pk-pk value, We tried to amplify a bit more but the signal got attenuated, limiting the Signal to 1 V pk-pk

2.4 Noise input to Arduino

The resulting signal from the capacitor could not be fed directly to the microcontroller for processing, as the input range for the Arduino is 0 to 5V. To get the signal in this range, we used Summer Circuit to add a stable **2.5V DC** voltage from **MAX 6126.**

3. Processing of noise on Arduino

Processing is basically executing Fast Fourier Transform (FFT) of the noise using Arduino UNO

3.1 Arduino Uno Description and Sampling

The **Arduino Uno** is a microcontroller board based on the **ATmega328**. It has **20 digital input/output** pins (of which 6 can be used as PWM outputs and **6** can be used as **analog inputs**)

Arduino provides a convenient way to read analog input this using the **analogRead()** function. Without going into much details the analogRead() function takes 100 microseconds leading to a theoretical sampling rate of **9.6 kHz**.

According to the **Nyquist theorem of Sampling :**

A bandlimited continuous-time signal can be sampled and perfectly reconstructed from its samples if the waveform is sampled over **twice as fast as it's highest frequency component**.

Nyquist limit: the highest frequency component that can be accurately represented

$$f_{\max} < f_s/2.$$

So, According to the above description, we can say that Arduino has a **frequency Bandwidth** of **4.8kHz,** which is **pretty less** and all our discussion about using High GBWP Op-Amps to get high Bandwidth Noise was a waste.

So we have to **Speed up the AnalogRead()**

3.2 Speeding up the analogRead() function

In Arduino **analogRead()** works as following

It sends the detected signal to ADC by, which the Analog Signal is converted to 10 bit Digital Information, But this ADC has a clock signal assigned to it, which is the limiting factor of the Bandwidth

The ADC clock is 16 MHz divided by a **Prescale factor**. The prescale is set by **default to 128** which leads to **16MHz/128** = **125 KHz ADC clock**. Since a conversion takes 13 ADC clocks, the default sample rate is about **9600 Hz (125KHz/13**).

But we can access this **Prescaler factor** assigning Register and change its value to 16, which will increase the **Sampling Rate to 76.8kHz** and increase the **Frequency Bandwidth to 38.4kHz** (this is the max we can increase taking into account the tradeoff between Frequency Bandwidth and Resolution of the Detection - which may reduce to 8 or 6 Bits)

ADCSRA Register has 3 bits named ADPS0, ADPS1, ADPS2 which we can access and change the Prescaler Factor (more info - Arduino Code)

Still... there is a **Scope to increase the speed of analogRead()**

3.2.1 AnalogRead with Interrupts

In Reality analogRead() function works very slow

A **better strategy is to avoid calling the analogRead() function** and use the **'ADC Free Running mode'.** This is a mode in which the ADC continuously converts the input and throws an interrupt at the end of each conversion. This approach has two major advantages:

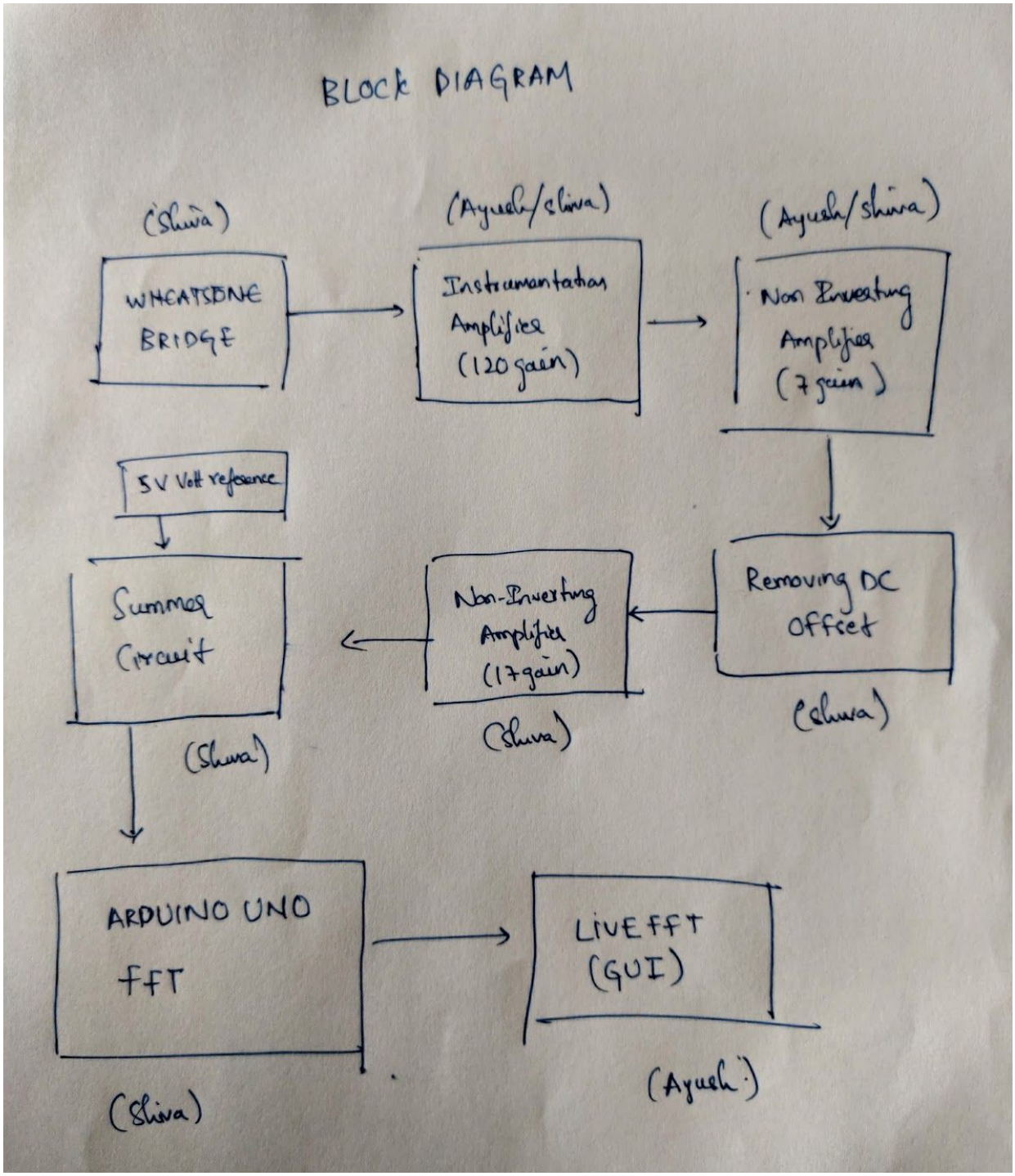
1. Do not waste time waiting for the next sample allowing to execute additional logic in the loop function.

2. Improve accuracy of sampling reducing jitter.

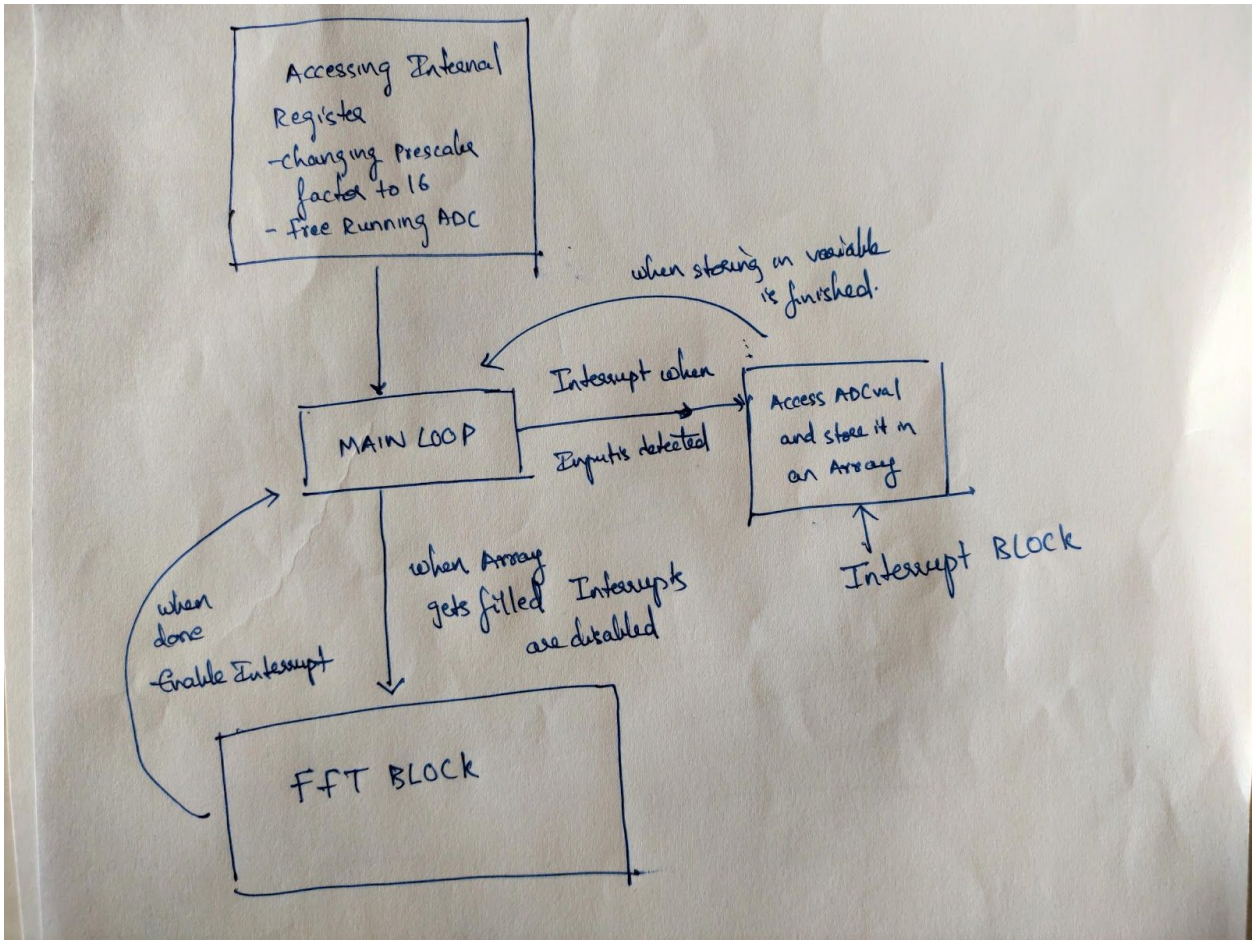
For this we need to access registers **ADCSRA, ADCSRB and ADMUX** and **Reconfigure them** (which is explained in the Arduino code below)

4. List of components

1. Max 6126 - 5V voltage reference
2. MC33078P - Op-amp
3. Arduino UNO
4. Resistors and capacitors


BLOCK DIAGRAM
(Shiva)
WHEATSTONE BRIDGE
(Ayesha/Shiva)
Instrumentation Amplifier (120 gain)
(Ayesha/Shiva)
Non Inverting Amplifier (7 gain)
Removing DC offset
(Shiva)
Non-Inverting Amplifier (17 gain)
(Shiva)
Summer Circuit
(Shiva)
5V Volt reference
ARDUINO UNO FFT
(Shiva)
LIVEFFT (GUI)
(Ayesha)

Flow of Arduino Code



Code

1. Arduino code to calculate the FFT of the input signal

```
#include "arduinoFFT.h" //include the library

arduinoFFT FFT = arduinoFFT(); //Creating FFT object

int ADCval;
const uint16_t samples = 128;
const double samplingfrequency = 76000 ; //19kHz
```

```
double vReal[samples];
double vImag[samples];

uint8_t sampleCount = 0;

void setup() {
  // put your setup code here, to ruwn once:

  cli(); // disable interrupts

  ADCSRA &= ( bit(ADPS0) | bit(ADPS1)| bit(ADPS2) ); //clear prescalar bits
  ADCSRA = 0;  // clear ADCSRA register
  ADCSRB = 0;  // clear ADCSRB register

  // we need the ADC's to be running in the Free running mode
  // This is a mode in which the ADC continuously converts the input and
throws an
  // interrupt at the end of each conversion
  // 1. Do not waste time waiting for the next sample allowing to execute
additional logic
  // 2. Improve accuracy of sampling reducing jitter

  ADMUX |= (0 & 0x07);   // set A0 analog input pin
  ADMUX |= (1 << REFS0); // set reference voltage AVcc as the reference
voltage
  ADMUX &= ~(1 << ADLAR); // clear for 10 bit resolution

// ADCSRA |= (1 << ADPS0); // 2 prescalar instead of 128 -- 608000 Hz
// ADCSRA |= (1 << ADPS1); // 4 prescalar instead of 128 -- 304000 Hz
// ADCSRA |= (1 << ADPS1) | (1 << ADPS0); // 8 prescalar instead of 128 --
152000 Hz


  //We can only bump the bandwidth from 4.50Hz to 38kHz


 ADCSRA |= (1 << ADPS2); // 16 prescalar instead of 128 -- 76000 Hz
// ADCSRA |= (1 << ADPS2) | (1 << ADPS0);  // 32 prescalar instead of 128
-- 38000 Hz
// ADCSRA |= (1 << ADPS2) | (1 << ADPS1);  // 64 prescalar instead of 128
```

```
-- 19000 Hz
// ADCSRA |= (1 << ADPS0) | (1 << ADPS1) | (1 << ADPS2) ;
//  ADMUX = (1 << ADLAR); // left align ADC values for 8 bit which can be
collected from ADCH register
//  ADCval = ADCH // collecting value directly from the ADCH it will be
written in ISR

  ADCSRA |= (1 << ADATE); // enable auto trigger
  ADCSRA |= (1 << ADIE);  // enables interrupts when measurement is
complete
  ADCSRA |= (1 << ADEN);  // enable ADC
  ADCSRA |= (1 << ADSC);  // start ADC measurements

  sei(); //enable interrupts

  Serial.begin(115200);

}

void loop() {
    delay(1000);
  // put your main code here, to run repeatedly:

  while(1){
    while( ADCSRA & _BV(ADIE)); // wait for the samples to be collected


    FFT.Windowing( vReal, samples, FFT_WIN_TYP_HAMMING, FFT_FORWARD );
    FFT.Compute( vReal, vImag, samples, FFT_FORWARD ); // Computes FFT
    FFT.ComplexToMagnitude( vReal,vImag, samples );     // Compute
magnitudes
    double peak = FFT.MajorPeak( vReal, samples, samplingfrequency );

    //Serial Printing Starts

//    Serial.println(peak,6);
    //Serial.println();

    for(uint16_t i=3; i < samples/2; i++){
      float x_coordinate = (i * 1.0 * samplingfrequency) / samples;
      float y_coordinate = vReal[i];
```

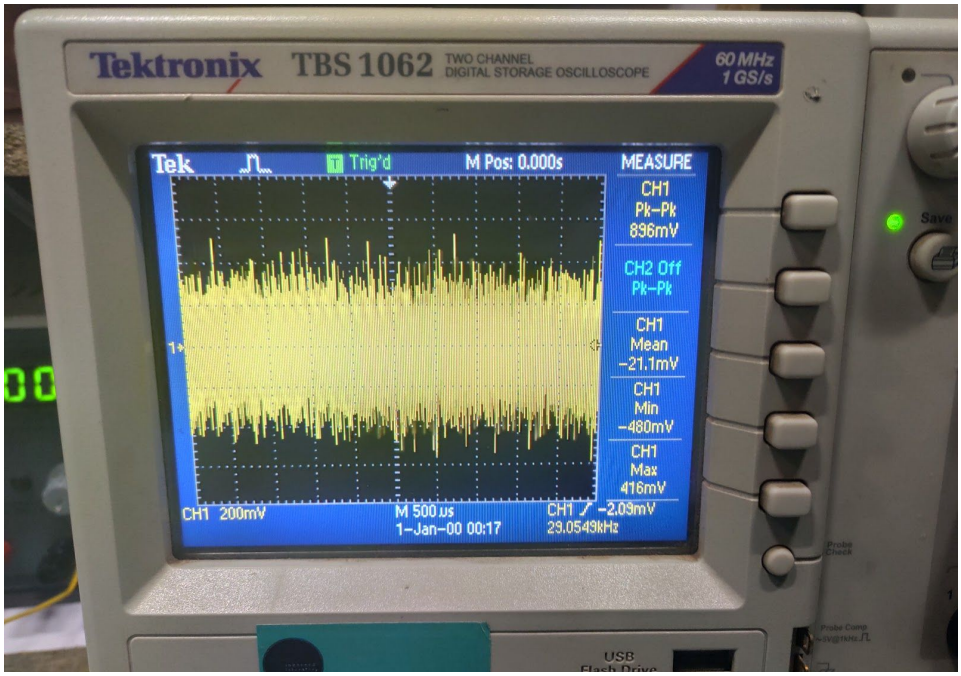
```
      Serial.print(x_coordinate);
      Serial.print(",");
      //Serial.print(" Hz :");
      Serial.print(y_coordinate);
      Serial.print(",");
      //Serial.print("   ");
    }

    Serial.println();

    sampleCount = 0;
    ADCSRA |= (1 << ADIE);  //Interrupt on

    }
}

ISR(ADC_vect){

  //when new ADC value ready
  if(sampleCount<samples){
    ADCval = ADCL;
    ADCval = (ADCH << 8) + ADCval;
    vReal[sampleCount] = ADCval;  //should look whether we need ADCval
variable or not
    vImag[sampleCount] = 0.0;
    sampleCount++;
  }
  else{
  //  ADCSRA &= ~(1 << ADIE);
  ADCSRA &= ~_BV(ADIE); //alternate syntax to the above
  }
}
```

2. Python code used to make a GUI to show live FFT display

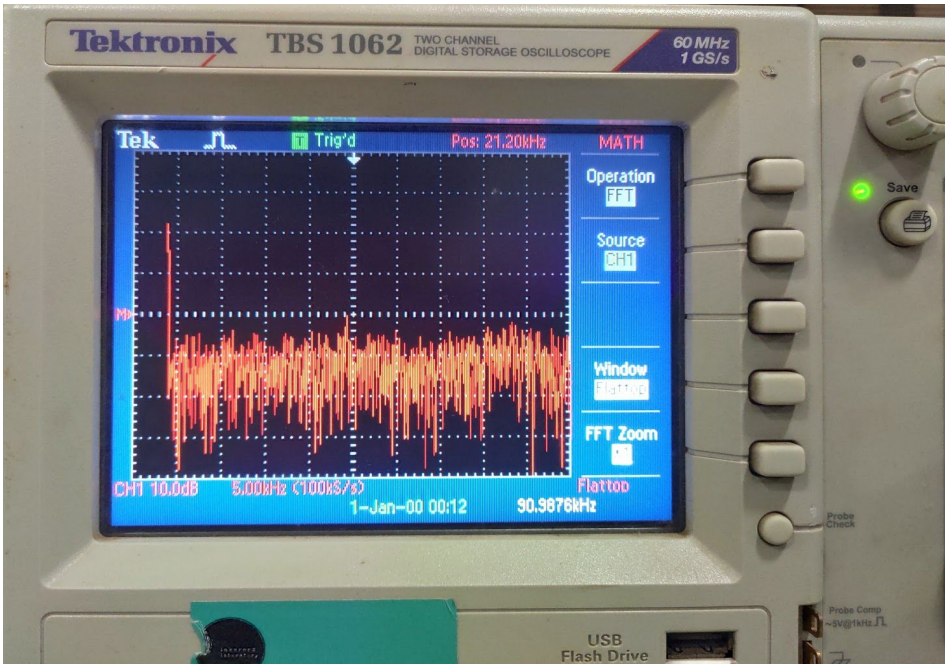
```
import serial
import numpy as numpy
from drawnow import *
from matplotlib import pyplot as plt
fig, ax = plt.subplots()

arduinoData = serial.Serial('/dev/ttyACM0', 115200)

def makeFig(): #Create a function that makes our desired plot
    plt.grid(True)                                  #Turn the grid on
    plt.ylim([0, 1000])                             # setting y limits
    plt.locator_params(axis='x', nbins=500)         #Set y-labels
    plt.plot(x_coor, y_coor)

while True: # While loop that loops forever

    while (arduinoData.inWaiting()==0): #Wait here until there is data
        pass                            #do nothing

    while True :
        try :
            arduinoString = arduinoData.readline().decode("utf-8")
                  # decoding the arduino data to make a string
            break
        except UnicodeDecodeError:
            continue   # to remove this error which was making our GUI
    dataArray = arduinoString.split(',')

    print(dataArray)

    x_coor = []
    y_coor = []
    for i in range(64):
        x_coor.append(593.75*i)    # make a list for all x-values
        x_coor[i] = '{0:.2f}'.format(float(x_coor[i]))
                                   # converting them to string type
        if x_coor[i] in dataArray:
            index = dataArray.index(x_coor[i])
```

```
            y_coor.append(dataArray[index+1])  # We append the value next
                                               # to the x-coor value, it
                                               # depends on the arduino
                                               # output format
        else:
            y_coor.append(0)

        while True :
            try:
                 y_coor[i] = float(y_coor[i])
                break
            except ValueError:
                y_coor[i] = y_coor[i-1]   # no need to use float function
                                          # again, y_coor[i-1] is already a
                                          # float-type

    drawnow(makeFig)                      #Call drawnow to update our graph
    plt.pause(.00001)                     #Pause Briefly. Important to keep
                                          #drawnow from crashing
```

Results



Output of the Whole Circuitry in AC mode



FFT of the Output

Milestones

DATE	MILESTONE	ACTIONS TO BE TAKEN NEXT
2/10/2019	Initially our goal was just to make a random number generator	-
9/10/2019	After talking to TA's and sir, we included the GUI part in our goal. By this week, we were nearly done with the research part.	-
16/10/2019	Finished with the arduino code and got the circuit components delivered. Tried making the GUI in MATLAB first (got stuck in it).	-
23/10/2019	Built prototype of a GUI (in python).	-
30/10/2019	Had a working GUI and the circuit	-
4/11/2019 – 8/11/2019		-

Work distribution

1. Shiva Teja - All the circuitry and arduino part
2. Ayush Bhardwaj - Building live-FFT display in Python