PID Control Integration with Sensor and Motor

Prashant Shushilbhai Gandhi-013712361
Computer Engineering Department, College of Engineering
San Jose State University, San Jose, CA 94303
E-mail: prashantshushilbhai.gandhi@sjsu.edu

Abstract

This paper gives detail explanation of how to design a circuit to interface ADC module, Motor Driver and LSM303 with raspberry pi 3 b+ and program raspberry such that it takes input from that ADC module, do data validation on ADC output, covert that output to target angle for motor and calculate PID algorithm to rotate motor in direction of target angle. This project uses LSM303 sensor to find motor's current position. This paper explains how to read LSM303 data which uses I2C protocol and convert ADC data to 0-360-degree angle and how to calculate Proportional-Derivational and Integral term of PID. Apply 2D Laplace of Gaussian (LoG) to calculated PID to reduce noise and base on that generate PWM signal for motor to rotate it to the targeted angle. The goal of this project is successfully achieved which can be verified in test and verification section.

1. Introduction

This project uses Raspberrypi 3 b+ board which contains CPU ARM-Cortex A53 and it has clock cycle of 1.4 GHz. Raspberry pi 3 b+ does not have inbuilt ADC module. Due to that, to integrate sensors which give analog output with Raspberry Pi 3 b+, we need to connect ADC module externally. This project uses Adafruit MCP3008 10-bit ADC module with 8 channels to convert potentiometer output. Communication between ADC module and Raspberry Pi 3 b+ is done through SPI. This project uses LSM303 magnetometer sensor which works on I2C protocol. To drive stepper motor project uses EasyDrive motor driver. At software side ADC's digital output is converted to appropriate voltage range using ADC resolution formula which this paper coverts later. Due to random noise, measured voltage from multimeter at Potentiometer output and voltage after conversion in software will reflect some fractional error. To overcome this problem, compensation function is used to compensate that error which is done by taking samples of output at fixed frequency which is called sampling frequency. To validate that data and to check if aliasing is occurring or not we use FFT and plot power spectrum graph. After data validation of ADC output, it is converted to 0 to 360 degree to set a target for motor rotation and from LSM303 sensor which works as feedback, error is calculated and based on that PID calculation is done. After that Laplace of Gaussian (LoG) is applied on that data to reduce noises and PWM signal is generated based on the calculated LoG.

2. Methodology

Following section of this paper will explain objective of this paper and what technical challenges were occurred will implementing this project how to overcome from that problems.

2.1. Objectives and Technical Challenges

Objective of this paper is as below:

1. Build a ADC circuit which interface with Raspberry Pi 3b+ through SPI.
2. Convert digital output of ADC circuit to appropriate voltage range using formula.
3. Build compensation function from samples takes at sampling frequency to reduce error.
4. Validate data using FFT program and plot power spectrum to see if aliasing is occurring or not.
5. Convert ADC data to degree to achieve target angle.
6. Take feedback from LSM303 sensor and calculate PID terms and based on that generate PWM signal to rotate motor to targeted angel.

Technical Challenges:

1. Find SPI driver to interface with ADC module.
2. Write PWM driver for stepper motor.
3. Understand I2C protocol.
4. Measure difference between multimeter reading and software calculated reading and calculate compensation function.
5. Plot graph of power spectrum from FFT process.
6. Write C code for PID calculation and design LoG convolution algorithm.

2.2. Problem Formulation and Design

Design block diagram:

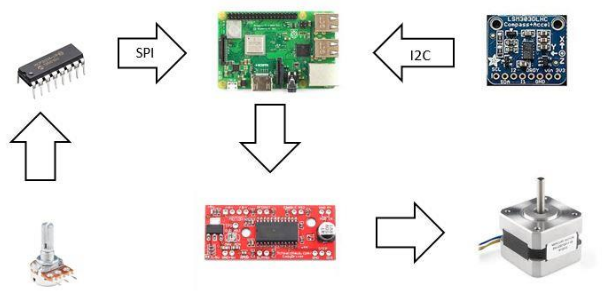


Figure 2.2.1 Design block Diagram

Analog -to-Digital Conversion formula:

$$\frac{Resolution\ of\ the\ ADC}{System\ Voltage} = \frac{ADC\ Reading}{Analog\ Voltage\ Measured}$$

Compensation Formulation steps:

1. First take 10 readings from multimeter and corresponding digital converted values in software.
2. Do interpolation:

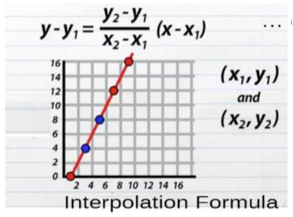


Figure 2.2.2 Interpolation formula and graph

Here, x-axis will be our voltage measured from multimeter and y-axis will be our digital raw reading. Result will be in terms of y and x. Then we apply following formula to calculate compensation function.

F(x)=p*x+q …….(1)

Here, from the result we can extract p and q of above formula:

Using that p and q compensation function is calculated by following formula.

g(x)=a_g +b_g, …….(2)

where**, a_g = -q** and **b_g = (a-p).**

Here, **a = 1024/3.3** (1024 because 10-bit ADC)

G(x) is our compensation function.

Data validation:

To validate our data, we use FFT and DFT process and plot power spectrum to check if aliasing is happening.

The Discrete Fourier Transform can be expressed as

$$\mathbf{F}(n) = \sum_{k=0}^{N-1} f(k)\, e^{-j2\pi nk/N} \qquad (n = 0..1 \ :N-1)$$

The relevant inverse Fourier Transform can be expressed as

$$f(k) = \frac{1}{N}\sum_{n=0}^{N-1} \mathbf{F}(n)\, e^{j2\pi nk/N} \qquad (k = 0, 1, 2...,N-1)$$

PID control formula:

PID (Proportional, Integral, Derivative) controller is feedback system that attempts to control the given system. It calculates error between desired setpoint and measured process carriable.

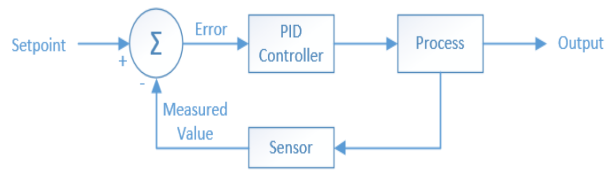


Figure 2.2.3 PID controller schematic

PID Equation:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

PID equation is build around 3 constants, Proportional, Integral and Derivative coefficient. Each of this can be attuned to achieve desire response. Here,

U(t) = output,

E(t) = error value,

Kp = proportional constant that counts for present error.

Ki = Integral error that accounts for historical error values. If output is not strong error will accumulate and controller will response by increasing the output.

Kd = Derivative constant that accounts for future error values.

Note: This project uses square of e(t) in integral term and for derivative, project does central differentiation.

Formula for central differentiation: ½*[e(t)-e(t-2)].

PID response graph:

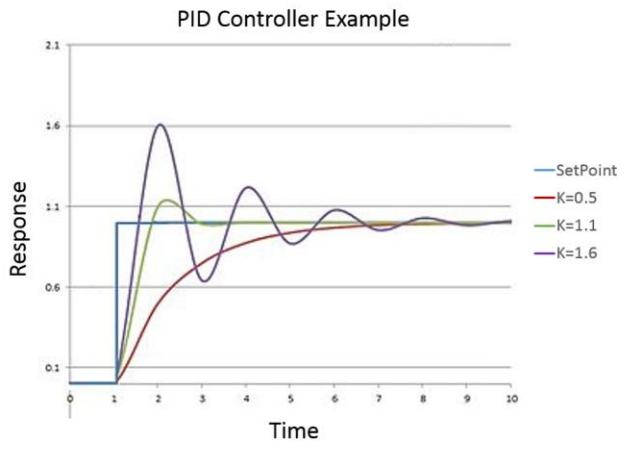


Figure 2.2.4 PID response graph

3. Implementation

Implementation can be categorized in two parts,

1. Hardware part.
2. Software part.

3.1 Hardware Design

Hardware aspect of this project includes designing of ADC circuit using potentiometer. List of components are in the below table.

Table 3.1.1 Bill of Materials

SL.No	Components	Description	Notes

1.	Raspberry Pi 3 B+	CPU	Qty 1
2.	MCP3008	ADC Module	10-bit 8-Channels
3.	Potentiometer	10K Ohm	Qty 1
4.	Jumper wires	-	Qty 2
5.	EasyDriver	Stepper motor driver	Qty 1
6.	Stepper Motor	-	Qty 1
7.	LSM303	Accelerometer+ Magnetometer	Qty 1

System block diagram:

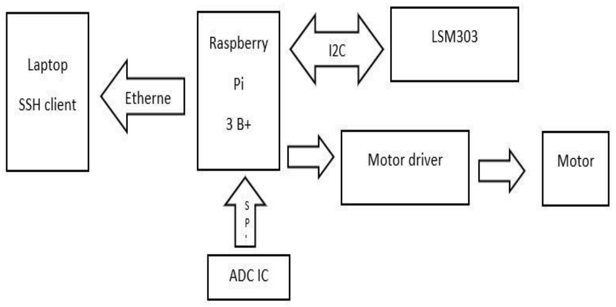


Figure 3.1.1 System block diagram

Raspberry pi 3 b+ is connected to ADC module through SPI protocol whose connection diagram is as below.

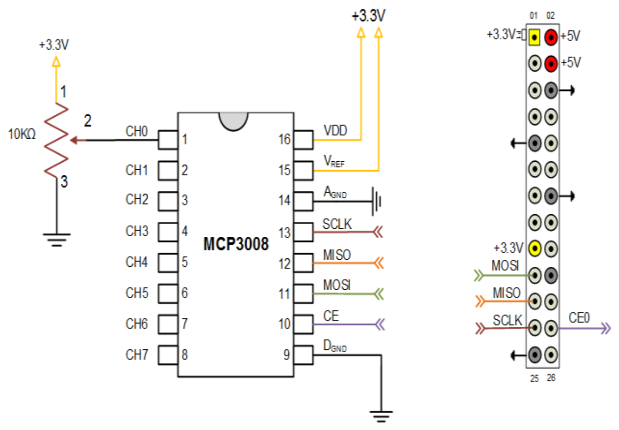


Figure 3.1.2 Raspberrypi and MCP3008 SPI connection

Below is real hardware picture.

Table 3.1.2 Pin connection for MCP3008

Pin	Raspberry Pi	Pin	MCP3008
1	3V3	16	VDD
1	3V3	15	Vref
6	GND	14	Agnd
6	GND	9	Dgnd
23	SPI0_SCLK	13	SCLK
21	SPI0_MISO	12	MISO
19	SPI0_MOSI	11	MOSI
24	SPI0_CE0_N	10	CE
2	Potentiometer	1	CH0

Raspberry pi is connected to LSM303 accelerometer magnetometer through I2C prototcol. Pin connection between raspberrypi and LSM303 is in below table.

Table 3.1.3 Pin connection for LSM303

Pin	Raspberry Pi	Pin	MCP3008
1	3V3	2	VIN
6	GND	3	GND
5	SCL	8	SCLK
3	SDA	7	MISO

Raspberry pi is connected to EasyDriver motor driver as per below table. Pins MS1 and MS2 controls micro step of stepper motor which is describe in table 3.1.5.

Table 3.1.4 Pin connection for driver motor

Pin	Raspberry Pi	MCP3008
2	5V	PWR IN
6	GND	GND
11	GPIO17	ENABLE
15	GPIO22	MS2
16	GPIO23	MS1

18	GPIO24	DIR
22	GPIO25	STEP

Table 3.1.5 Microstep select Resolution truth table

MS1	**MS2**	**Microstep Resolution**
L	L	Full step
H	L	Half step
L	H	Quarter Step
H	H	Eight Step

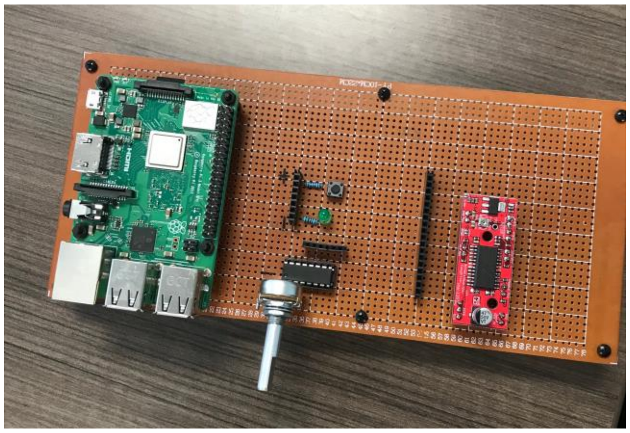


Figure 3.1.3 Prototype Board

3.2. Software Design

Below is the Algorithm, Flow chart of the software implementation.

3.2.1 Algorithm

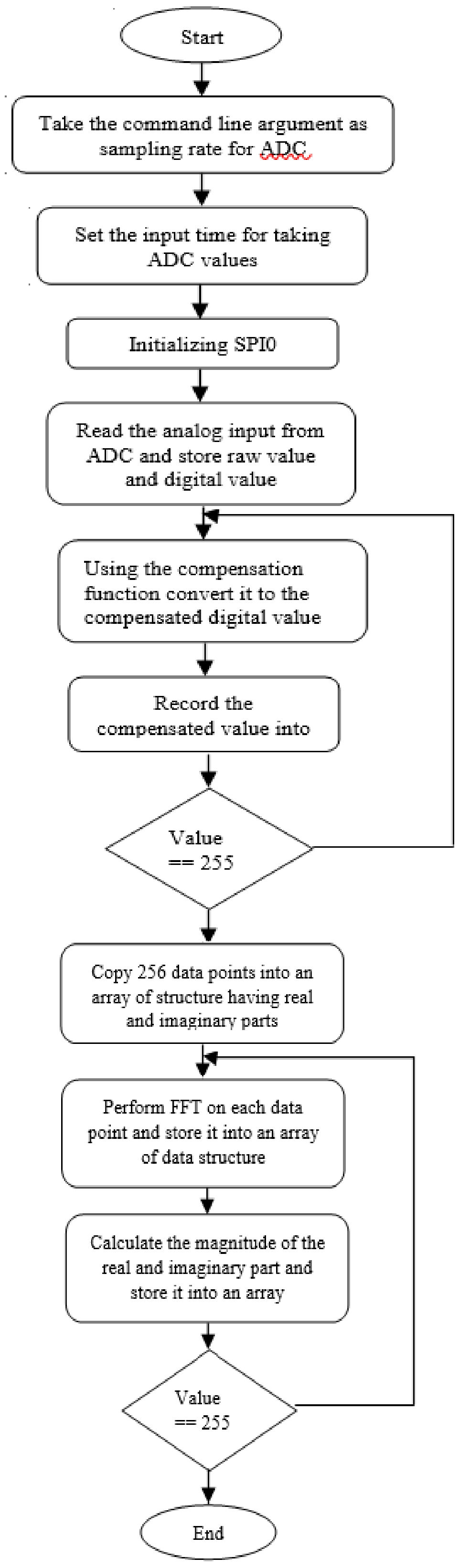
Algorithm for getting ADC output and build compensation function.

1. Take sampling rate from user to take samples of ADC output using command line argument. Using command line argument.
2. Initialize the Raspberry Pi 3 B+ SPI 0 Port.
3. Read ADC module's Channel 1 to get RAW data. Convert it such that its range become 0-3.3V.
4. Calculate compensation function using that data.
5. Repeat steps 3 and 4, introduce delay to take data in accordance of sampling rate.

Algorithm to validate above data using FFT and get power spectrum.

1. Get the 256 raw data from the file in which output is stored.
2. Feed this data to FFT.c file to calculate imaginary and real part
3. Calculate magnitude using that imaginary and real part.
4. After getting output of magnitude we store data in file and using excel we create plot of power spectrum.

3.2.2 Flow-chart _



Start
Take the command line argument as sampling rate for ADC
Set the input time for taking ADC values
Initializing SPI0
Read the analog input from ADC and store raw value and digital value
Using the compensation function convert it to the compensated digital value
Record the compensated value into
Value == 255
Copy 256 data points into an array of structure having real and imaginary parts
Perform FFT on each data point and store it into an array of data structure
Calculate the magnitude of the real and imaginary part and store it into an array
Value == 255
End

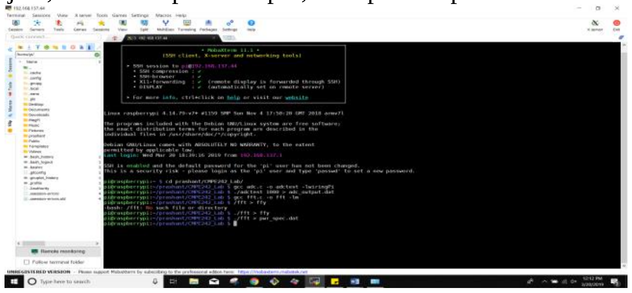
3.2.3 Pseudo Code:

ADC Compensation Function :

```
//Read channel-1 of MCP3008 to get raw data
raw_data = myAnalogRead(0,8,1);

//Convert it into voltage (0-3.3v)
uncompensated_value = ((3.3/1023)*raw_data);

//Apply compensation formula
g_x = (-3.495*uncompensated_value) + 1.032;

//Add that calculate value to raw data and compensate value
//compensated value

new_raw_data = raw_data + g_x;
compensated_value = (3.3/1023)*new_raw_data;
```

4. Testing and Verification

Below are the pictures of Successful compilation of project, ADC 256-point output, FFT power spectrum.



```
* MobaXterm 11.1 *
(SSH client, X-server and networking tools)

➤ SSH session to pi@192.168.137.44
  • SSH compression : ✓
  • SSH-browser     : ✓
  • X11-forwarding  : ✓  (remote display is forwarded through SSH)
  • DISPLAY         : ✓  (automatically set on remote server)

➤ For more info, ctrl+click on help or visit our website

Linux raspberrypi 4.14.79-v7+ #1159 SMP Sun Nov 4 17:50:20 GMT 2018 armv7l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Mar 20 18:39:16 2019 from 192.168.137.1

SSH is enabled and the default password for the 'pi' user has not been changed.
This is a security risk - please login as the 'pi' user and type 'passwd' to set a new password.

pi@raspberrypi:~ $ cd prashant/CMPE242_Lab/
pi@raspberrypi:~/prashant/CMPE242_Lab $ gcc adc.c -o adctest -lwiringPi
pi@raspberrypi:~/prashant/CMPE242_Lab $ ./adctest 1000 > adc_output.dat
pi@raspberrypi:~/prashant/CMPE242_Lab $ gcc fft.c -o fft -lm
pi@raspberrypi:~/prashant/CMPE242_Lab $ /fft > ffy
-bash: /fft: No such file or directory
pi@raspberrypi:~/prashant/CMPE242_Lab $ ./fft > ffy
pi@raspberrypi:~/prashant/CMPE242_Lab $ ./fft > pwr_spec.dat
pi@raspberrypi:~/prashant/CMPE242_Lab $ 
```

Figure 4.1 Successful compilation

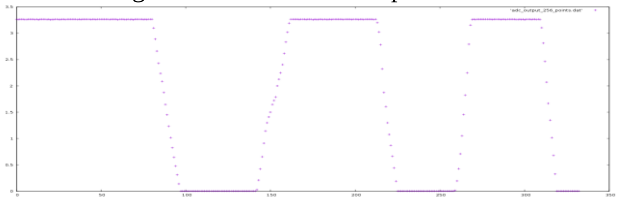


Figure 4.1 256-point ADC digital output

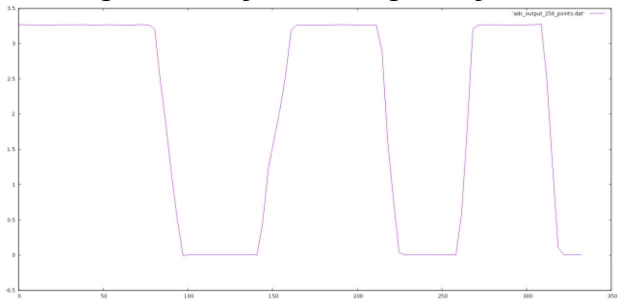


Figure 4.1 256-point ADC digital output connecting dots

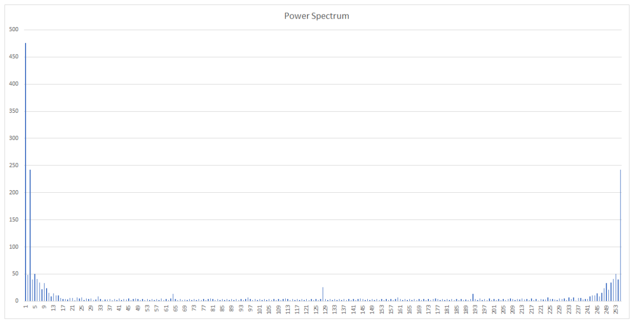


Figure 4.2 Power spectrum

5. Conclusion

Objective of this project was to design ADC circuit for Raspberry Pi 3 B+ module and implement program which can successfully convert digital output into appropriate voltage using compensation function and validate that data using FFT process and plot power spectrum graph for same which is successfully implemented, and project is getting output as expected.

6. Acknowledgement

I would like to express my gratitude and appreciation to our Professor Harry Li for his encouragement, valuable guidance and patient review. His constant suggestion on hardware design helped us to implement good hardware and write effective program to compensation function for ADC module output. Also, I would like to thank him for giving us such a valuable knowledge of how to program Raspberry pi 3 b+. At last, I would also like to thank my project partners for their valuable inputs.

7. References

[1] https://github.com/hualili/CMPE242-Embedded-Systems-/tree/master/2019S
[2] http://wiringpi.com/reference/spi-library/
[3] https://learn.adafruit.com/raspberry-pi-analog-to-digital-converters/mcp3008
[4] https://alvinalexander.com/technology/gnuplot-charts-graphs-examples
[5] https://learn.sparkfun.com/tutorials/analog-to-digital-conversion/all

8. Appendix

8.1 Source Code

adc.c

```
/* ////////////////////////////////////////////////////////
 * LAB-1    : ADC Compensation function and Data Validation
 * Name     : PRASHANT SHUSHILBHAI GANDHI
 * STD-ID   : 013712361
 * Email-ID : prashantshushilbhai.gandhi@sjsu.edu
 * Date     : 03/20/2019
 * ///////////////////////////////////////////////////
 * ///////////////////////////////////////////////////
 * For GPIO and SPI driver we are using wiringPi's Driver
 * Steps to compile:
 *       gcc adc.c -o adctest -lwiringPi
 * To run file:
 *       ./adctest 10000 (Where 10000 is Sampling Rate)
 * ///////////////////////////////////////////////////
*/
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <wiringPi.h>
#include <wiringPiSPI.h>

#define INPUT_ACQ_TIME      5000000
#define SPI_FREQ                                  1000000

static int setup ;

// Setting Up SPI
void spiSetup (int spi_Channel_Number)
{
   if ((setup = wiringPiSPISetup (spi_Channel_Number,
SPI_FREQ)) < 0)
   {
      fprintf (stderr, "Can't open the SPI bus: %s\n",
strerror (errno)) ;
      exit (EXIT_FAILURE) ;
   }
}

int myAnalogRead(int spi_Channel_Number,int
channel_Config,int analog_Channel)
{
   if(analog_Channel<0 || analog_Channel>7){
      return -1;
         }
   unsigned char buffer[3] = {1};
   buffer[1] = (channel_Config+analog_Channel) << 4;
   wiringPiSPIDataRW(spi_Channel_Number, buffer, 3);
   return ( (buffer[1] & 3 ) << 8 ) + buffer[2];     // Extract
last 10-bits
}

// This function is to generate Delay
void _delay(int numberOfSeconds)
{
   int milli_seconds = 1000 * numberOfSeconds;
   clock_t start_time = clock();
   while (clock() < start_time + milli_seconds)
      ;
}

float compensation_function(void){
   float uncompensated_value = 0, compensated_value =
0, g_x = 0,
                        new_raw_data = 0;
   int raw_data = 0;
   raw_data = myAnalogRead(0,8,1);        // Initialize all
channer, Take output from channel 1
   uncompensated_value = ((3.3/1023)*raw_data);
   g_x = (-3.495*uncompensated_value) + 1.032;
   new_raw_data = raw_data + g_x;
   compensated_value = (3.3/1023)*new_raw_data;
   return compensated_value;
}

int main (int agrc, char *argv[])
{
   int int_sampling_rate=0;
   // command line argument will accept sampling rate
                              if(agrc < 2)
   {
      int_sampling_rate = 1000;
   }else{
   char *sampling_rate = argv[1];
   int_sampling_rate = atoi(sampling_rate);
   }
   printf("Sampling Rate given: %dHz\n",
int_sampling_rate);
   printf("Input acquisition time is %d seconds\n",
(INPUT_ACQ_TIME/1000000));
   //Initialization of SPI
   wiringPiSetup();
   spiSetup(0);

   int seconds = INPUT_ACQ_TIME;
          //Take input for 5 seconds
   clock_t start_time = clock();
   while(clock() < start_time + seconds){
      printf("%f\n", compensation_function());
                     _delay(1000 / int_sampling_rate);
   }

   close(setup);
   return 0;
}
```

FFT.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIZE_OF_INPUT 256
#define TWO_TO_THE_POWER 8
```

```
struct Complex
{           double a;        //Real Part
            double b;        //Imaginary Part
}         X[SIZE_OF_INPUT], U, W, T, Tmp;

void FFT(void)
{
            int M = TWO_TO_THE_POWER;
            int N = pow(2, M);

            int i = 1, j = 1, k = 1;
            int LE = 0, LE1 = 0;
            int IP = 0;

            for (k = 1; k <= M; k++)
            {
                        LE = pow(2, M + 1 - k);
                        LE1 = LE / 2;

                        U.a = 1.0;
                        U.b = 0.0;

                        W.a = cos(M_PI / (double)LE1);
                        W.b = -sin(M_PI/ (double)LE1);

                        for (j = 1; j <= LE1; j++)
                        {
                                    for (i = j; i <= N; i = i + LE)
                                    {
                                                IP = i + LE1;
                                                T.a = X[i].a +
X[IP].a;
                                                T.b = X[i].b +
X[IP].b;
                                                Tmp.a = X[i].a -
X[IP].a;
                                                Tmp.b = X[i].b -
X[IP].b;
                                                X[IP].a = (Tmp.a *
U.a) - (Tmp.b * U.b);
                                                X[IP].b = (Tmp.a *
U.b) + (Tmp.b * U.a);
                                                X[i].a = T.a;
                                                X[i].b = T.b;
                                    }
                                    Tmp.a = (U.a * W.a) - (U.b *
W.b);
                                    Tmp.b = (U.a * W.b) + (U.b *
W.a);
                                    U.a = Tmp.a;
                                    U.b = Tmp.b;
                        }
            }

            int NV2 = N / 2;
            int NM1 = N - 1;
            int K = 0;

            j = 1;
            for (i = 1; i <= NM1; i++)
            {
                        if (i >= j) goto TAG25;
                        T.a = X[j].a;
                        T.b = X[j].b;

                        X[j].a = X[i].a;
                        X[j].b = X[i].b;
                        X[i].a = T.a;
                        X[i].b = T.b;
TAG25: K = NV2;
TAG26: if (K >= j) goto TAG30;
                        j = j - K;
                        K = K / 2;
                        goto TAG26;
TAG30: j = j + K;
            }
}

int main(void)
{
             FILE *in_file  =
fopen("adc_output_256_points.dat", "r"); // read only

        // test for files not existing.
        if (in_file == NULL)
          {
            printf("Error! Could not open file\n");
            exit(-1); // must include stdlib.h
          }
         float ab[256];
        // write to file vs write to screen
            for(int j = 0; j < SIZE_OF_INPUT; j++){
            fscanf(in_file, "%f", &ab[j]); // write to file
            }

            int i;
            for (i = 0; i < SIZE_OF_INPUT; i++)
            {
                        X[i].a = ab[i];
                        X[i].b = 0.0;
            }

            printf ("*********Before*********\n");
            for (i = 1; i <= SIZE_OF_INPUT - 1; i++)
                        printf ("X[%d]:real == %f  imaginary
== %f\n", i, X[i].a, X[i].b);
            FFT();

            printf ("\n\n**********After*********\n");
            for (i = 1; i <= SIZE_OF_INPUT - 1; i++)
```

```
                        printf ("X[%d]:real == %f  imaginary
== %f\n", i, X[i].a, X[i].b);

            for(i = 1; i <= SIZE_OF_INPUT - 1; i++){
                        X[i].a = pow(X[i].a, 2);
                        X[i].b = pow(X[i].b, 2);
                        ab[i] = X[i].a + X[i].b;
                        ab[i] = sqrt(ab[i]);
                        printf("Squared value [%d] = %f\n", i,
ab[i]);
//                      printf("%f\n", ab[i]);
            }
            return 0;
}
```