Kerolos Ikladeos (2432195i)
Anirudh Garikapati (2480291g)

Assignment 3: IIR Filters

YouTube Clip: https://youtu.be/2AZ1obUZoQc
GitHub Repository: https://github.com/kerowaseem/ldrgaragesensor

Introduction

In this assignment, the main aim is to do real time filtering for a physical quantity. Unlike the previous assignment where there is an array of the whole data and real time is being simulated, the data is actually being input to the code in real time as they are measured, and the plot is constantly changing due to change in the physical quantity being measured. Therefore, the type of filter that is used is an IIR filter and an Arduino board is used as the data acquisition (DAQ) card that takes measurements from an input sensor. Before mentioning the process of the filtering and the data acquisition, the main application needs to be described.

The aim is to have a parking sensor in an indoor garage that warns the car driver when they get closer to the wall. It is assumed that there is a sensor in each parking space in the garage, and there are lights with different colours that warn the driver that they are getting closer to the wall so they can stop and that they have successfully parked their car. In other words, a sensor is used as an aid for drivers to park their car. The wall is simulated using an LDR sensor that detects the rear lights of a car, and the lights in the garage that emit different colours are simulated by 3 different LED lights with 3 different colours, red, yellow, and green. It is sort of measuring distance of how far away the car is using data output from the LDR. It can be inferred that green means the space is free and there is enough space for the car to park, yellow means that the car is getting closer to the wall and the driver needs to take care, and red means that the driver needs to stop immediately, and the parking space is successfully occupied, and the car is close to the wall as close as possible within the safety limits. Figure 1 below shows the setup of the circuit with the Arduino board. The total list of components used for the whole setup and shown in the figure below are: 3 LEDs for garage lights and their corresponding resistors, LDR sensor, DAQ board (Arduino Uno). Shown in figure 2, another circuit board was assembled in order to control several LED lights that will simulate the rear lights of a car getting closer to the wall which is simulated by an LDR sensor. An ELEGOO Uno R3 board was used to control the 6 LEDs with their corresponding resistors that simulate the rear lights of a car.

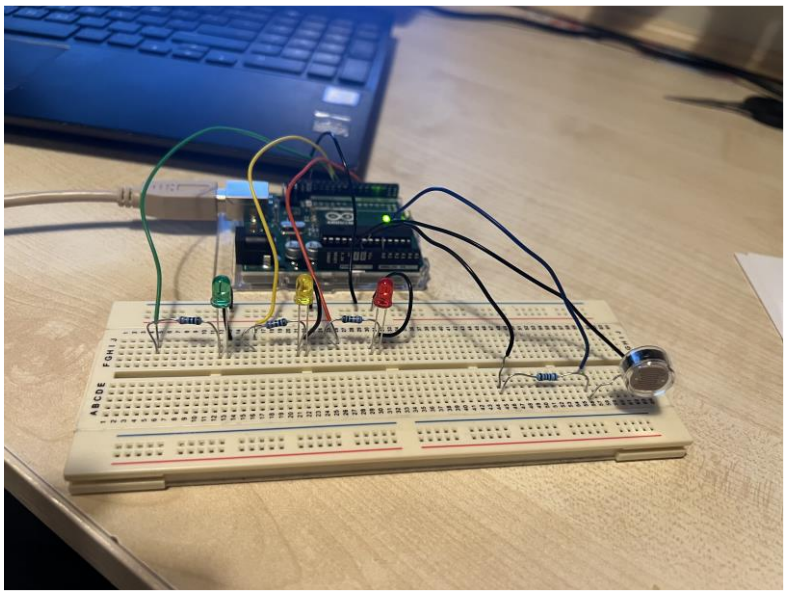

Figure 1: Setup of the LDR Circuit

Data acquisition (DAQ) is the process of measuring an event in terms of voltage, current, pressure, temperature, or sound. A DAQ system includes high-speed data acquisition measurement hardware (a DAQ card or module), input devices such as sensors, and a computer or processor. In this application, Arduino Uno is used as the real time data acquisition cards, measuring voltage output from the LDR, and connected to our personal computers that has Python code that processes the input data which is taken from an LDR (light dependent resistor) sensor. LDRs are light sensitive sensors that are used to indicate the presence or absence of light, or to measure the light intensity. In other words, it is a type of a resistor where its resistance varies depending on the amount of light falling on its surface. When an LDR is kept in the dark place, its resistance is high and, when the LDR is kept in the light its resistance will decrease. In other words, the LDR allows higher voltages to pass through it (low resistance) whenever there is a high intensity of light and passes a low voltage (high resistance) whenever it is dark.

This system works by sensing the intensity of light in its environment using LDR sensor as mentioned. The LDR is connected to VCC (5V), and it outputs an analogue voltage which varies in magnitude in direct proportion to the input light intensity on it. This means that the greater the intensity of light, the greater the corresponding voltage from the LDR will be and vice versa. Because the LDR gives out an analogue voltage, it is connected to the analogue input pin (A0) on the Arduino. The output is an analogue voltage because the LDR is connected to a resistor in series which acts as a voltage divider that the voltage changes as the resistance changes.

A voltage divider configuration is used for the application in order to measure the output analogue voltage of the LDR. In our application, voltage is being plotted and the LDR acts as a voltage divider alongside a 1kΩ resistor. The resistor was chosen by trying different resistor values to see which one will show the effect of voltage division better and changes in the output voltage better. The A0 (analogue pin 0) of the Arduino is connected to the LDR, and this pin is used as the output of the LDR in the Python program. As the LDR's resistance increases in the dark, the less voltage is being output and vice versa. The connection of the breadboard is a voltage divider configuration with the LDR attached the 5 volts pin of the Arduino and a 1kΩ resistor attached to the ground pin. The two resistors are tied together, and the tie point is attached to the analogue input pin A0. The 3 LEDs react to different values of the analogue voltage output. As the value increases, this means that that light intensity increases and the rear light of the car is really close to the LDR (wall); therefore, the red LED lights up. The lower the value and further away it is, depending on the how far and the value of the analogue voltage output, a yellow or green LED lights up. Each LED is connected to a digital pin on the Arduino and lights up at different values/ranges of the input data.

The base value from the surrounding light is at about 0.2. From testing, the car setup at different distances away from the LDR, ranges can be extracted to set exactly which LED and which message to be printed out at the output. Below 0.4, green LED is set to 1, which means it is turned on, and the other two LEDs (yellow and green) are set to zero. At this range, the rear lights are far away from the LDR and "Enough Space" message is printed. More than 0.4 and less than 0.6 which means the car is getting closer and more light is directed towards the LDR, the yellow LED lights up and "Take Caution!" message is printed out. More than 0.6 where the car is really close to the LDR, the red LED lights up and "STOP IMMEDIATELY" message is printed out. Figure 2 below shows the different LEDs lighting up as the rear lights come closer to the LDR.

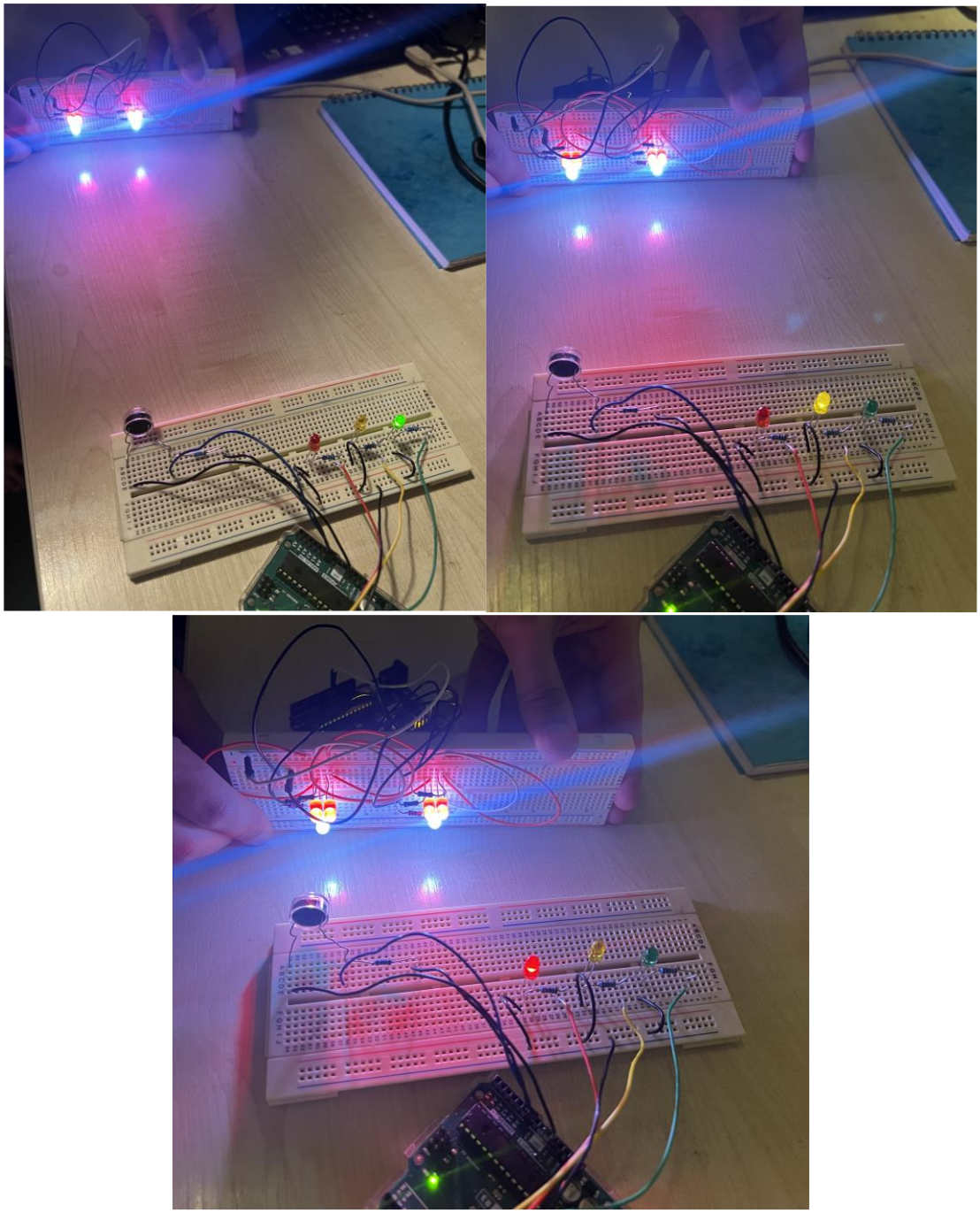

Figure 2: Green, Yellow, and Red LEDs lighting up as the rear lights are coming closer.

Figure 3 below shows the plot at its base value without the car being near and the in the background the green LED is on. The LDR is just reacting to the lighting of the room which simulates lighting in an indoor garage. As mentioned, the output is analogue voltage due to voltage divider configuration; therefore, the plot has a normalised voltage from 0 to 1 which corresponds to the light intensity measured from the LDR. It can be seen that noise from mains that generate a sine wave-like interference needs to be filtered out. In the next section, the acquisition of the data itself will be explained and how a Python code is used. Then, IIR filter will be used to filter the data. The IIR filter coefficients will be generated using Butterworth filter that will output the coefficients in the form of second order structures (SOS).

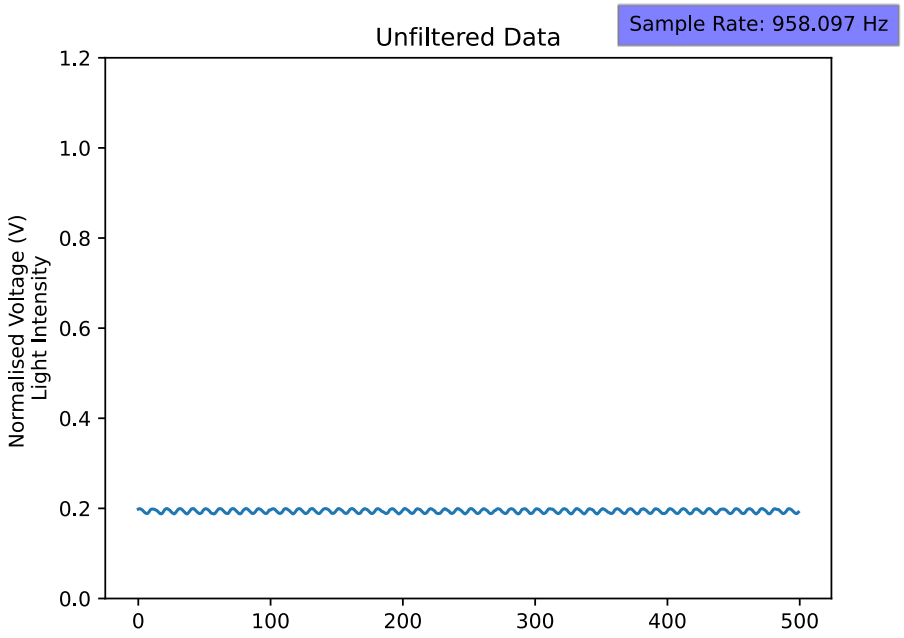

Figure 3: Unfiltered Data at base value 0.2 reacting to room light

Data Acquisition

PyFirmata2 is a module that is being used as it provides event driven real time callbacks and allows the user to control, filter, acquire, and plot the data from the Arduino using Python rather than using the Arduino IDE. In other words, PyFrimata2 turns the Arduino into a data acquisition card controlled by Python. Arduino IDE was just used at the start to upload the standard firmata sketch into the Arduino board to be used later in the Python code. First thing is to define the port in which the Arduino is connected to. Using the "autodetect" in PyFirmata2, this easily detects the Arduino connected to the computer device it is being connected to. From PyFirmata2 GitHub examples that are issued by Dr. Bernd Porr, "RealtimePlotWindow" class is being used in order to plot all the incoming data from the Arduino in real time and observe the changes occurring as the LDR resistance changes due to its light environment. The class sets up the plot window and is plotting the newest 500 samples and discards the older ones.

Thereafter, the most important function that needs to be defined is called callBack. This function has two uses, one of which is what takes the data from the Arduino port that is connected to the LDR and allows manipulating (filtering) the data and plotting it. The other use of this function is to calculate the sampling rate of the data from the LDR in real time. This function is called for every new sample which has arrived from the Arduino. After defining the function, getting the Arduino port is crucial by using "Arduino(PORT)." PORT is auto detected at the beginning of the program. Sampling rate of the Arduino is also set to be 1000 divided the actual sampling rate of the application which is 1 kHz. It will be later on mentioned why 1 kHz was chosen. Then, it is particularly important to "register the callback" from analogue pin 0 which is connected to the LDR which adds the data to the "callBack" function that was defined before and expecting input data to manipulate and plot. The function "register_callback" read from the analogue port (A0) and outputs the data as a float from 0 to 1. Finally, the callback needs to be enabled from analogue pin 0.

IIR Filtering

The infinite impulse response (IIR) filter is a recursive filter. The output from the filter is computed by using the current and previous inputs and previous outputs. This is the general idea behind IIR filters. The IIR filter mimics an analogue filter that processes the data in real time. In other words, the signal is being processed with minimal delay while the signal comes in. In this assignment, IIR filter is being used by using high level design commands such as “butter” to create the IIR coefficients and using those coefficients in the IIR filter class that is filtering the input data in real time.

For IIR filtering in this lab, two important modules are being used. One module is “signal.scipy” which will be used to generate the coefficients for the IIR filter, and the second module is “iir_filter” which will be doing the actual filtering of the input data in real time. First is generating the coefficients for a chain of second order IIR filters. Butterworth filter is used in order to generate those coefficients. Butterworth filter is a standard analogue filter, and it is used as Butterworth filters have a maximally flat response in the passband and that is an ideal filter characteristic.

Fast Fourier Transform was done to analyse and know the frequency content of the data acquired from the sensor. There were two methods used in order to do that. Each method corresponds to different type of plots that were provided to us in order to plot data in real time. The two methods of plotting are using the PyQtGraph module or using Matplotlib. PyQtGraph has a feature to FFT transform the data to show the power spectrum in real time while the data is changing. Figure 4 and figure 5 show the plots for the live FFT transformation using PyQtGraph module to plot them. From figure 4, it can be seen when zoomed in that there is a peak at 100 Hz which correspond to noise from mains that needs to be filtered. The difference between figures 4 and 5 are that light that is being directed towards the LDR is getting closer, and the amplitude at the 0 Hz (DC frequency) can be seen to have increased. This is because the LED lights of the other circuit that are powered digitally and turned all the time, so they do not have any “> 0” Hz frequency content, but the amplitude of the DC frequency (0 Hz) increases as closer as the circuit approaches the LDR sensor.

The other method to find the frequency spectrum of the data before being filtered is to store the data samples in an array and plot its Fourier transform. The method of plotting the Fourier transform was the same as the first assignment. This method was used in order to verify the results achieved using PyQtGraphs. The results are shown in figures 6 and 7. It can be seen they are similar to what was achieved in figures 4 and 5. It is also possible to zoom in and see the amplitude at different frequencies. It can be seen in the zoomed in plot in figure 7 that the main noise is at a frequency of 100 Hz and there are exceedingly small peaks around it at lower and higher frequencies that interferes with the data that they need to be filtered.

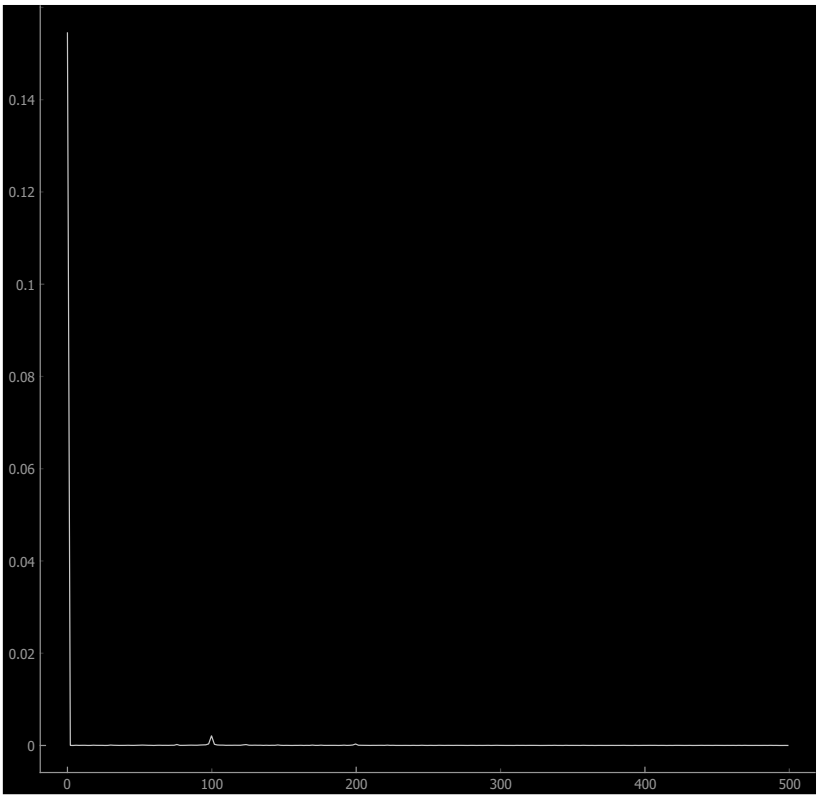

Figure 4: PyQtGraph Power Spectrum FFT Output

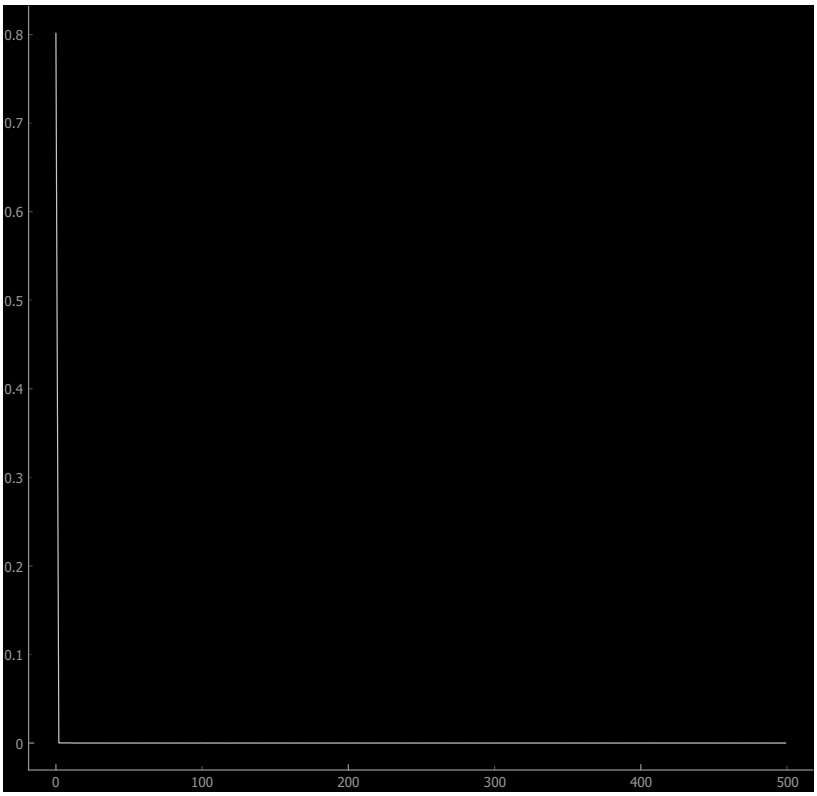

Figure 5: PyQtGraph Power Spectrum FFT Output when more light is directed towards the LDR

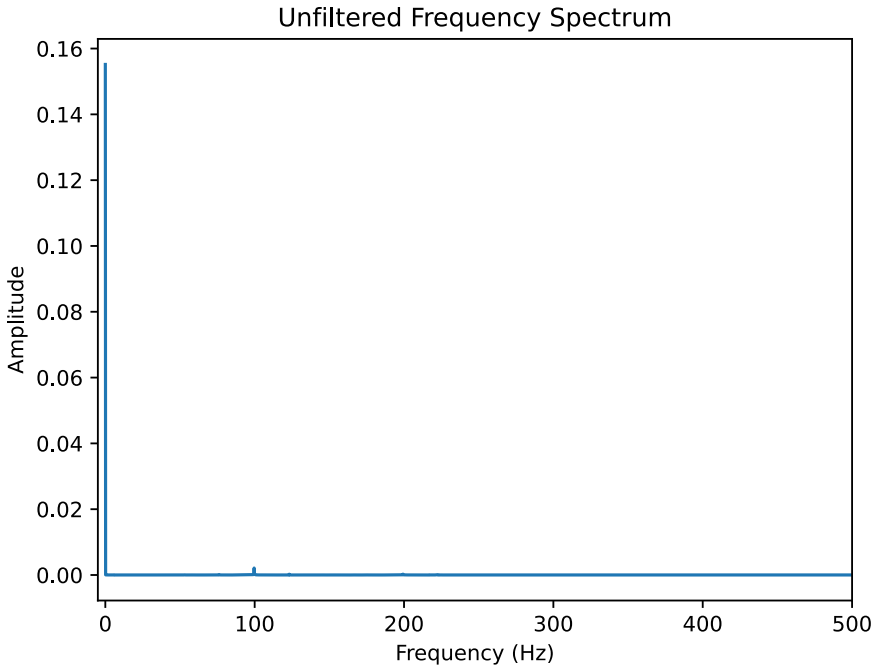

Figure 6: FFT of stored data samples to verify PyQtGraph output

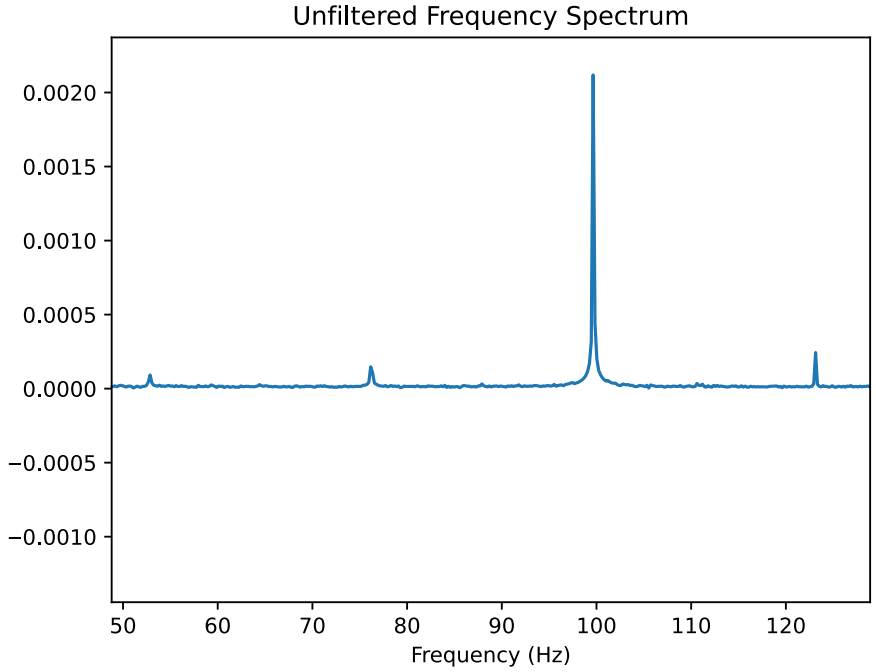

Figure 7: Zoomed In plot of figure 6 to show which frequencies need to be filtered

From figure 7, it can be inferred that the main noise that needs to be filtered is noise from mains which correspond to 100 Hz which is twice the frequency of the mains electricity. The fundamental frequency of this mains noise is usually double that of fundamental 50 Hz. Because the magnetic flux density is strongest twice every electrical cycle, the fundamental frequency of mains noise (100 Hz) will be twice the electrical frequency (50 Hz). Frequencies above 100 Hz are not required for our application as it can be seen at figure 6, amplitude at DC frequency increased as more light was directed towards the LDR. The main frequency range that relates to the signal being measured is a small range at around 0 Hz to 2 Hz. Some small peaks can be also seen below 100 Hz. Therefore, a

lowpass filter has been chosen for the application. A lowpass filter has been set at a cut-off frequency of 45 Hz to remove all noise above including noise from mains that correspond to 100 Hz. Having lowpass filter at 45 Hz gives us very good filtering, and there is no high frequency content that needs to be measured or useful to what is measured for the application, so they are filtered away. The rear lights are made up of static LED lights that do not change and do not turn on and off so that it is steadily affecting the light intensity measurement with no fluctuations or oscillations due to no flickering in LED lights. Frequency of the LED lights correspond to DC frequency which is 0 Hz. Only changes being detected is distance of the LED lights from the LDR which correspond to change in resistance that changes the analogue output voltage. As a result, frequencies above 45 Hz are not needed; therefore, a lowpass filter was chosen. The generation of the coefficients of the IIR filter is better shown in the following line of code:

```
sos1 = signal.butter(4, [fc/fs*2], 'lowpass', output='sos')
```

Using signal.butter from the "SciPy" module, 4th order (first argument in signal.butter) IIR filter coefficients are being generated. Second argument is the cut-off frequency of the lowpass filter. Cutoff frequency "`fc`" is set to 45 Hz and sampling frequency "`fs`" is set to 100 Hz. It is a convention for any high level filter command for the cut-off frequencies to be written in that way: $f_{critical} = \frac{f_{cutoff}}{\frac{f_{sampling}}{2}} = \frac{f_{cutoff}}{f_{sampling}} * 2$. In other words, Nyquist frequency is from 0 to 0.5 in the standard convention, but it is a convention for high level filter commands in the SciPy module to have its Nyquist frequency range from 0 to 1. The cut-off frequency is being normalised to Nyquist frequency which is half of the sampling rate. 3rd argument is the type of filter required as there are four types of filters, highpass, lowpass, bandpass, and bandstop. As mentioned, lowpass is chosen for the application. The 4th and the last argument is `output='sos'`. This is a very important argument to be added in order to output the coefficients in second order structures (SOS); SOS is an array of 2nd order IIR filter coefficients. In other words, this argument makes the output to be a chain of second order IIR filter coefficients. As for our application, a 4th order IIR filter is being used; therefore, two second order structures have been generated in order to create a total of a 4th order IIR filter. This is clearer in the dataflow diagram shown below.

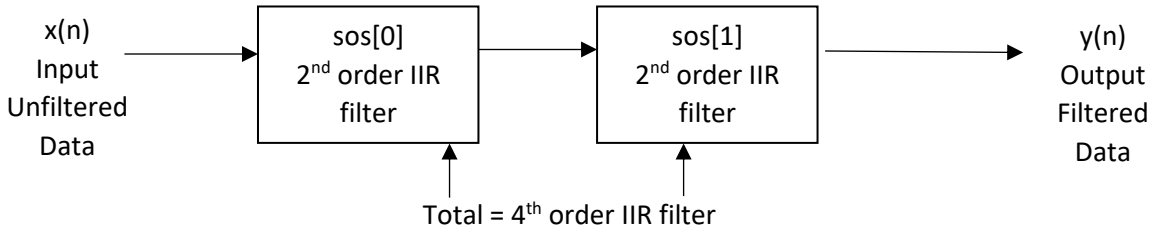

Figure 8: Second Order Structures Dataflow Diagram

The actual IIR filtering is done using an imported module created by Dr. Bernd Porr. This module is based on a Direct Form II IIR filter. This means that it has 2 accumulators and 2 delay steps (buffers). The general form of a 2nd order IIR filter is shown in the following equation:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

The numerator of the equation above is the FIR part, and the denominator is the IIR part. Essential to note, there is no a_0 in the equation as it is always set to 1. The function above is considered to be the transfer function. When it is multiplied by the unfiltered input, the filtered output is the result. The IIR filter module in the program uses the second order structure (SOS) coefficients that are created by "signal.butter" to define the FIR (numerator) and the IIR (denominator) coefficients. The process of filtering is better shown in the data flow diagram (done using MATLAB) below using the general formula.

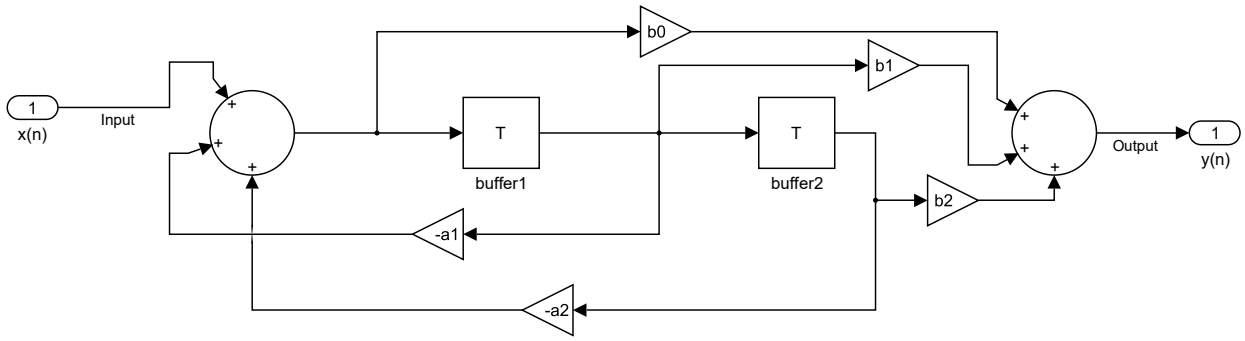

Figure 9: 2nd Order IIR Filter Dataflow Diagram

This dataflow diagram shows how the data is being filtered using a 2nd order IIR filter. In our application, the coefficients that are determined by "signal.butter" depend on the type of filter that is used in the application. As mentioned before, the filter used is lowpass filter and the cut-off frequency of the lowpass filter is defined at 45 Hz. A 4th order IIR filter is created, and the output is second order structures (sos); therefore, there will be two 2nd IIR filters to create a total of a 4th order IIR filter. As shown in the sos data flow diagram, the output of the first IIR 2nd order filter is the input of the second IIR 2nd order filter. It is better to demonstrate this with actual numbers. So, the output of the Butterworth filter for a 4th order lowpass filter are two arrays, with 6 elements each. The first 3 elements of the array are the FIR coefficients and the last 3 elements are the IIR coefficients. The array of the second order IIR filter using the coefficients of the general formula is as following: [b_0, b_1, b_2, a_0, a_1, a_2]. The elements of the first 2nd order IIR filter for the 45 Hz lowpass filter are as following: [2.83144331e-04, 5.66288661e-04, 2.83144331e-04, 1.00000000e+00, -1.52699742e+00, 5.90135850e-01]. The elements of the second 2nd order IIR filter for the 45 Hz lowpass filter are as following: [1. , 2. , 1. , 1. , -1.73531589, 0.8070679]. This shows that a_0 is always set to 1, so it is not required to show it on the block diagram as a separate coefficient. The dataflow diagram of our application will have the same structure as figures 8 and 9. Figure 9 is a more detailed version of each of sos[0] and sos[1] showing the data flow of the signal and the actual FIR and IIR coefficients generated by the Butterworth filter.

Generating the Butterworth lowpass IIR filter and instanciating the IIR filter class should be done only once and outside the callback function. The filtering is done in real time which means it is filtering the input data sample by sample as it does not have the future samples yet, and the IIR filter class has buffers that act as delay lines and memory that store previous samples. Therefore, defining the lowpass filter using Butterworth and instanciating the IIR filter class in the IIR filter module are done outside the callback function. The actual filtering is done inside the callback function. In previous assignments, a for loop was used to simulate real time and take the data of pre-made arrays one by one. In this application, the callback function is the for loop in this assignment, and the data is input one by one from the A0 (Arduino analogue pin). `filtered_output = iir1.filter(data)`. This line of code is inside the callback function, and it sets a new variable called "filtered_output" to the

new input sample after being IIR filtered using the IIR filter class. The filtering process in detail is shown in the data flow diagram in figure 10. Unfiltered input is x(n) and filtered output is y(n).

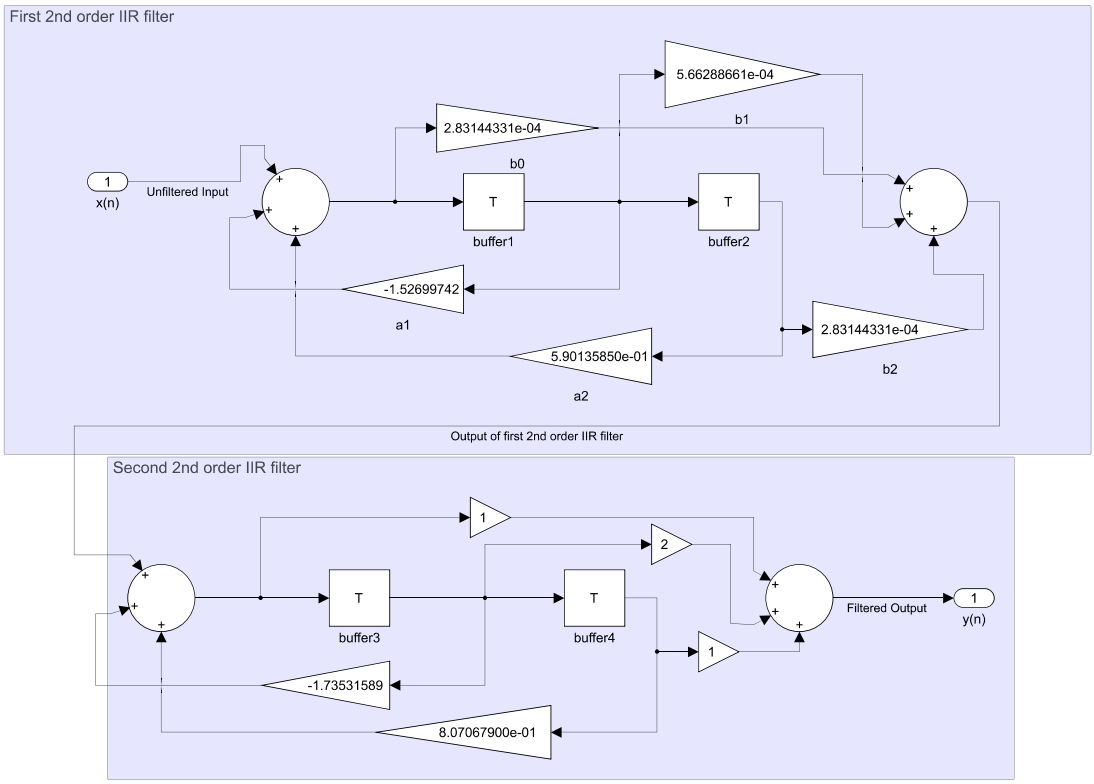

Figure 10: Dataflow Diagram of 4th Order lowpass IIR filter at 45 Hz

Figure 11 shows the unfiltered signals and show how noisy the signal can be when the light is so close to it. Noise is also coming from the other circuit which has been filtered out using the lowpass filter. Figure 12 show the filtered signal after going through the IIR filter, and the performance of the filter is so good that it has filtered the noise that are mainly generated from mains and the other circuit. Figures 13 and 14 show the FFT of the signal after being filtered, using PyQtGraph and Matplotlib respectively, with no peaks at 100 Hz and noise being attenuated.

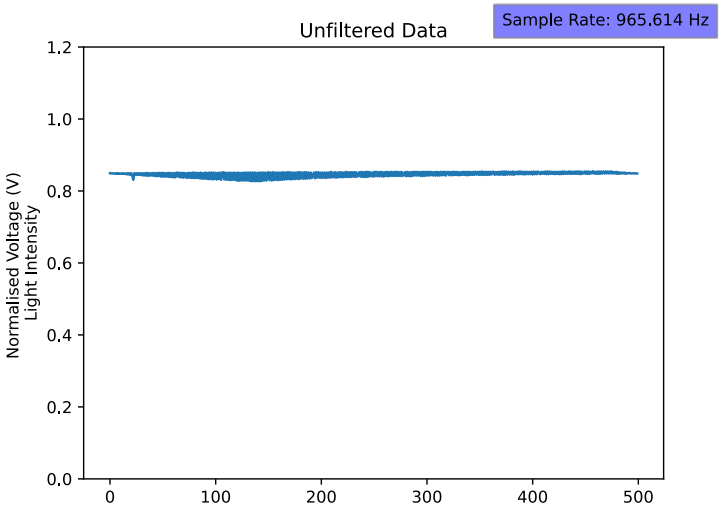

Figure 11: Unfiltered Data with a lot of noise

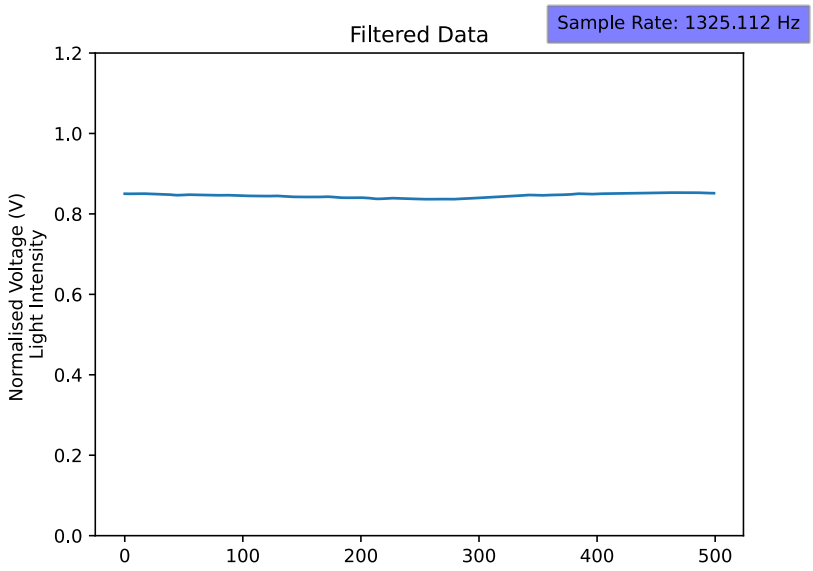

Figure 12: Filtered Data using lowpass 4th order IIR filter

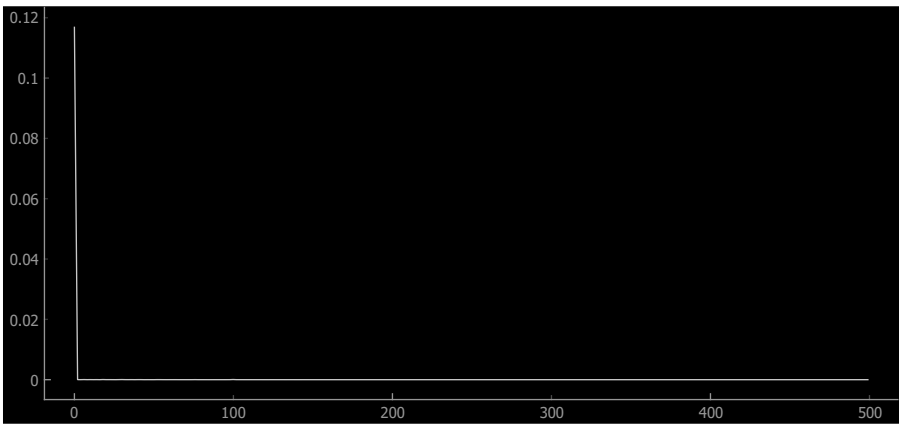

Figure 13: FFT Transformation of Filtered signal using PyQtGraph

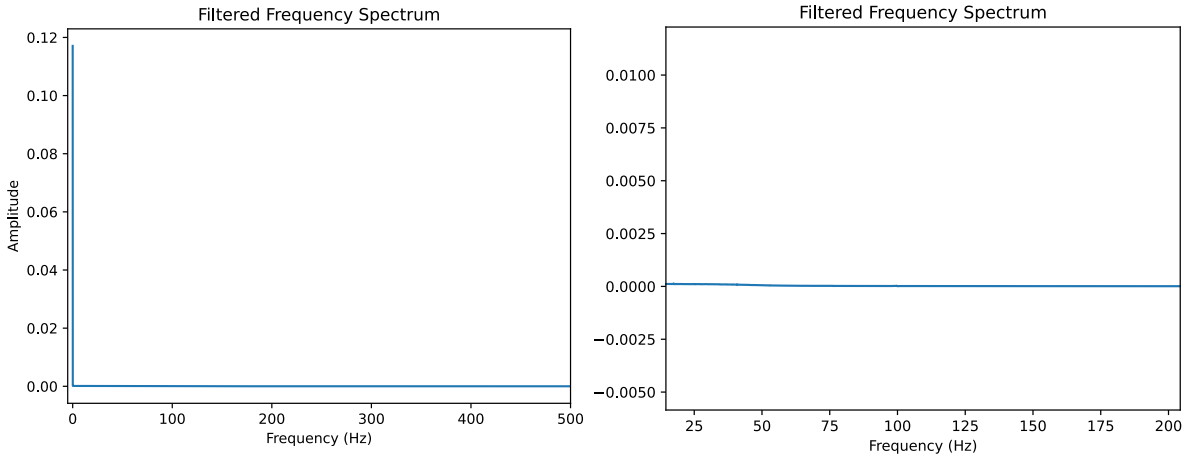

Figure 14: FFT of stored data using Matplotlib with no peaks at 100 Hz or lower frequencies
Right plot shows a zoomed in version of the left plot.

Program Structure

The code was written inspired by realtime_scope.py within the PyFirmata2 GitHub post by Dr. Bernd Porr. The code has two components to it: data filtering and manipulation and plotting the data. Figure 15 is a UML (Unified Modelling Language) diagram for the structure of the code, which showcases these two components through the use of two subsystems: “callBack” and “RealtimePlotWindow”. The callBack subsystem corresponds to the callBack function within the code (which will be explained later) and the latter corresponds to the RealtimePlotWindow class within the program that uses __init__, update and addData functions.

The RealtimePlotWindow class is called within the rest of the code, where the output is an animated plot that showcases the live data collected by the Arduino. Initially, the addData function takes in two arguments: unfiltered/filtered data and samplingFreq, which is the measured sampling frequency within the callBack function. The unfiltered/filtered data is appended onto an array of 500 zeroes that is defined in the __init__ function, which is then plotted. Next, the update function changes the plot as every new element of data is read by the Arduino, to show the last 500 data points.

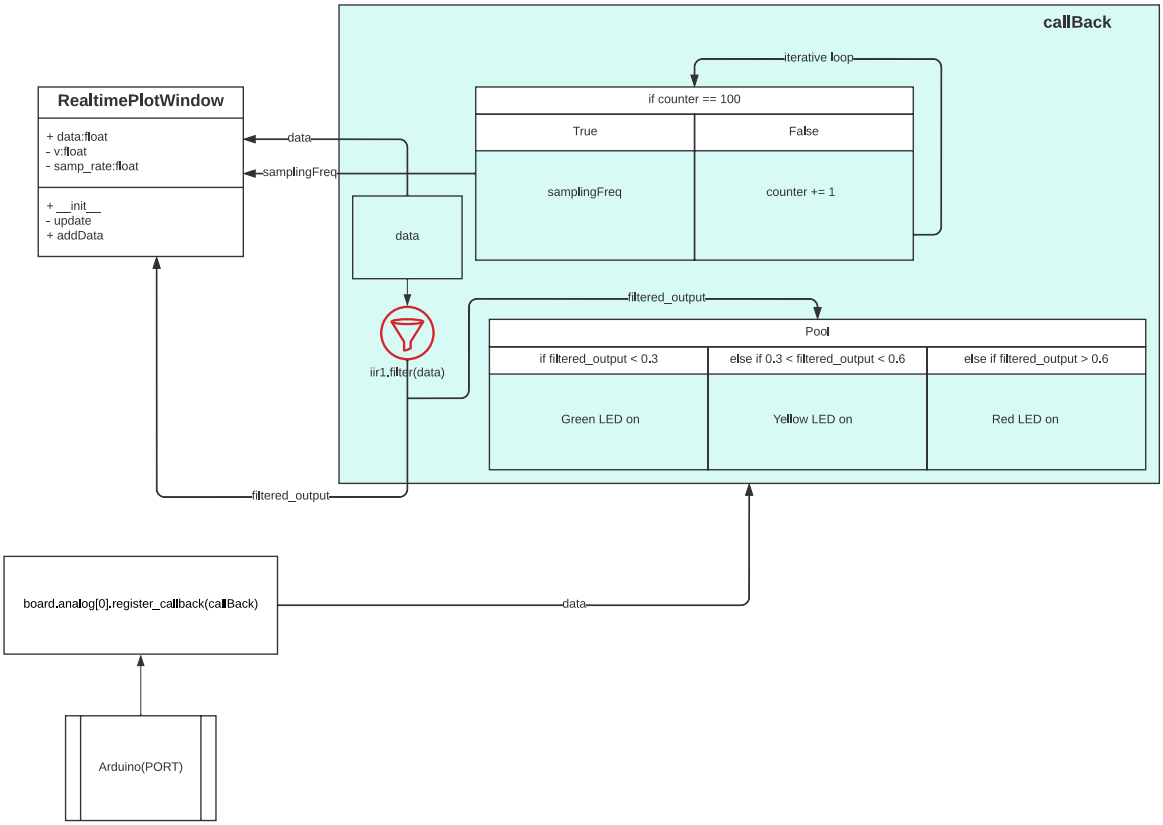

Figure 15: Dataflow UML diagram for the code

Within the callBack function includes an if statement that determines which LED glows up. As the application requires the device to tell the user to stop reversing when the brightness from the car rear lights gets too high, the if statement reads the filtered data and lights up the correct LED themed as a traffic signal. Therefore, as seen in figure 15, as long as the filtered data is below 0.3V, the Green LED is lit up, indicating that the user can keep driving. When the voltage measured through the LDR is

between 0.3V and 0.6V, then the Yellow LED is lit up to indicate that the user must slow down and take caution. Finally, as long as the brightness is above 0.6V, the Red LED is lit up to indicate that the user must stop reversing.

Sampling Rate

In our application, the sampling frequency of the application is 1 kHz. A high sampling rate was needed in order to track fast changes in the data acquired from the input sensor. Higher sampling frequency allows us to observe fast changes occurring in the LDR output as it takes more samples per second and allows the LEDs to change as fast as possible when the rear lights (car) comes closer or move further away from the sensor, therefore changing the light intensity observed on the LDR. This is a hard real time task where the consequence of missing the deadline is catastrophic. It is crucial to have high sampling rate because this helps in capturing the changes while the car is backing into the wall and not keeping track of sudden changes is fatal and can cause a crash.

In order to identify the sampling rate of the application in real time, the function named "callBack" is used. This function has two uses, the first of which is in relation to acquiring the sampling rate of the application. This utilizes an if statement that works as a stopwatch. Initially, the variables samplingFreq, counter and time_prev are set as global variables which have defined values outside of the function that are then brought into the function. The counter is set to 0 so as to have it counted up by integer values with every element of data being input from the sensor. The global variable samplingFreq is set to 0 and time_prev is set to the "perf_counter()" function within the "time" Python module. The reason for these initial values will be explained next.

The stopwatch if statement states that if the value of the counter reaches a specific value, then a series of other actions will take place. This specific value is set to 100 to collect information for every 100 samples. The function works this way because it will collect timestamps every 10^{th} of a second, as the sampling rate was initially set to 1 kHz.

Within the if statement, the series of actions that will take place when the counter reaches 100, then the counter will reset back to 0, current time labelled as "time_now" will be set to the same perf_counter() function within the time module, and time_prev will be set to the value of time_now. After this, the samplingFreq will be calculated as `samplingFreq = (100/(elapsedtime)`. Here, "elapsedtime" is a variable that finds the time passed between time_prev and time_now. This brings us back to the reason why samplingFreq and time_prev were defined outside of the function as 0 and perf_counter() respectively. When perf_counter() is used to define a variable globally, it is initially set to 0, and changes with increments within the data acquisition. However, defining this value outside of the function eliminates the initial transient response of the frequency counter (which can be seen in figure 16), allowing for quicker stabilisation in the frequency content. But samplingFreq being set to 0, solves an arithmetic issue that arises within the if statement. The equation for samplingFreq includes a division by 0 for the first two increments. For the first bit of data acquisition, time_now is 0, while time_prev does not exist (negative time values do not exist). For the first increment (second but of data acquisition), both time_now and time_prev hold the same value, causing the difference to be 0. Both of these increments cause the value of samplingFreq to be undefined, causing an error to occur. In order to solve this, samplingFreq is set to 0 outside of the function, so that the equation will look for a global variable with a defined value for a variable in question in case the actions for the if statement are undefined.

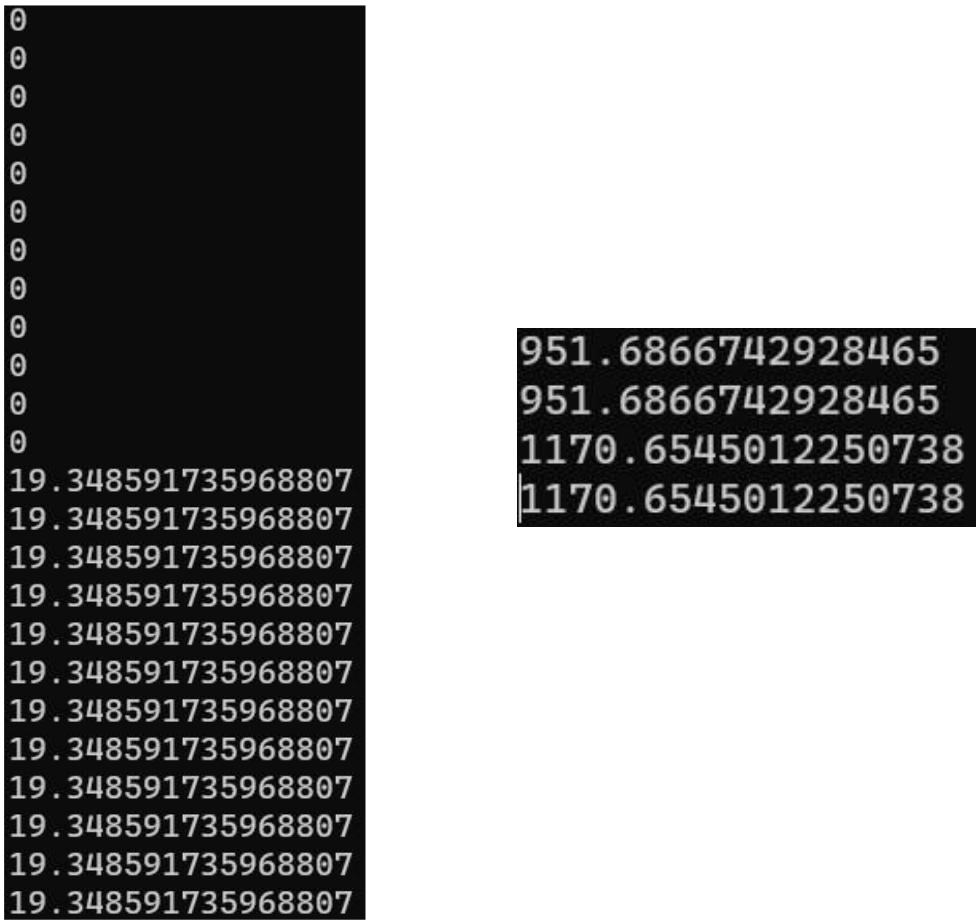

Figure 156: Transient Response for first two increments and average sampling rate

Finally, in order to plot the sampling frequency calculated by this function, the class RealtimePlotWindow was amended. The addData function within the class is now made to have an extra argument, used to append the value of "samp_rate" into the update function. When appended, the update function prints the samp_rate argument as a text using `self.samp_freq.set_text(f'Sample Rate: {self.samprate:.3f} Hz')`. This is then placed in a location at the top right of the plot window within the main __init__ function. This is what allows the Sampling Frequency to be displayed on the main plot and updated as per the counter periodically, as shown in figures 3, 11 and 12.

Conclusion

The initial sampling frequency chosen was 100 Hz, however, the output was too slow and did not simulate a real-time response that would be suitable for an application such as a parking sensor. Therefore, the sampling frequency chosen was 1 kHz, as it was fast enough to be suitable for this application. However, this detected a 100 Hz mains noise that was previously undetected due to the Nyquist limit of 50 Hz for 100 Hz sampling frequency. The IIR filter was used to remove this noise.

IIR filters are superior to FIR filters due to lower computational cost and quicker execution time. This is because IIR filters use fewer coefficients in order to calculate the filter response quicker and with fewer memory space usage. In addition, they provide more attenuation than an FIR filter can for a given filter order.

When comparing the output plots of both unfiltered and filtered data of the LDR through Arduino, the differences between the two sets of data are clear. The unfiltered data has noise as expected, which changes based on the surrounding environment while conducting the test. In the demo shown in the YouTube video (link at the top of the report), the test was conducted in dim yellow lighting that was roughly 1 meter away from the setup, therefore the noise observed by the LDR was very high. The filter does a good job of suppressing unwanted oscillations, as well as eliminating 100 Hz noise from mains. Figures 17 and 18 show a snapshot of the two plots at a random moment in time. It can be observed that both plots are stable, and the purpose of the filter was simply to eliminate oscillations.

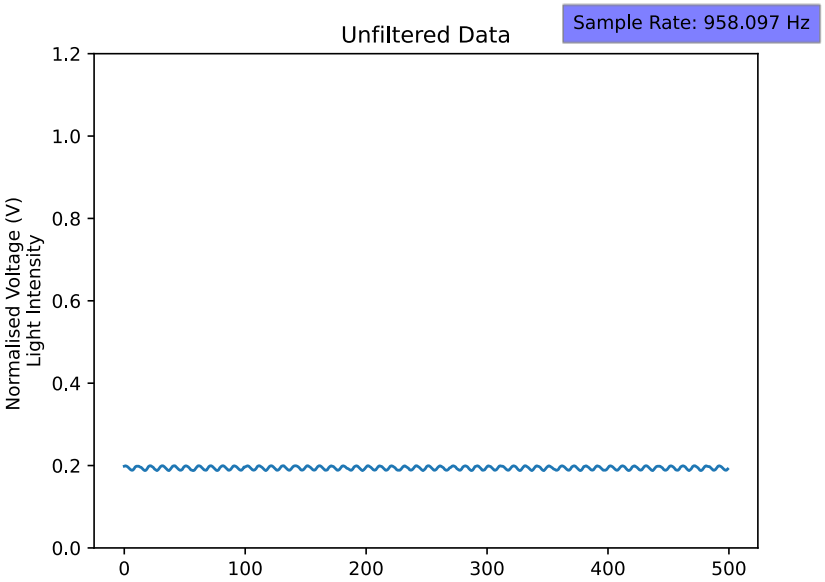

Figure 167: Unfiltered Data at 1 kHz sampling frequency

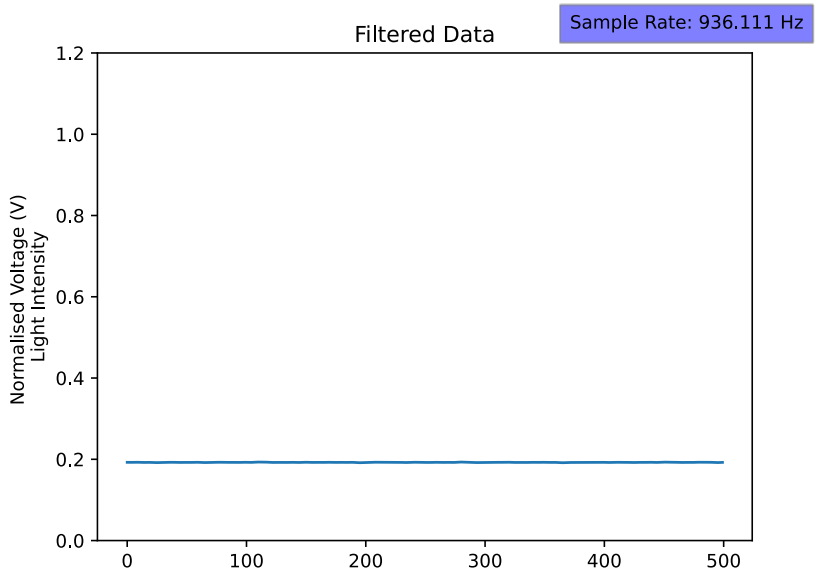

Figure 178: Filtered Data at 1 kHz sampling frequency

Appendix

➔ **main.py**

```
from pyfirmata2 import Arduino
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import iir_filter
from scipy import signal
from time import perf_counter

# Realtime oscilloscope at a sampling rate of 100Hz
# It displays analog channel 0.
# You can plot multiple chnannels just by instantiating
# more RealtimePlotWindow instances and registering
# callbacks from the other channels.
# Copyright (c) 2018-2020, Bernd Porr <mail@berndporr.me.uk>
# see LICENSE file.

PORT = Arduino.AUTODETECT
# PORT = '/dev/ttyUSB0'

# Creates a scrolling data display
class RealtimePlotWindow:

    def __init__(self,title): #xlabel):
        # create a plot window
        self.fig, self.ax = plt.subplots()
        # that's our plotbuffer
        self.plotbuffer = np.zeros(500)
        # create an empty line
        self.line, = self.ax.plot(self.plotbuffer)
        # axis
        self.ax.set_ylim(0, 1.2)
        self.ax.set_ylabel("Normalised Voltage (V) \nLight Intensity")
        self.ax.set_title(title)
        # That's our ringbuffer which accumluates the samples
        # It's emptied every time when the plot window below
        # does a repaint
        self.ringbuffer = []
        # start the animation
        self.ani = animation.FuncAnimation(self.fig, self.update,
interval=100)
        self.samp_freq = self.ax.text(0.9, 1.05, "Sample Rate:%d",
            bbox={'facecolor': 'blue', 'alpha': 0.5, 'pad': 5},
            transform=self.ax.transAxes, ha="center")


    # updates the plot
    def update(self, data):
        # add new data to the buffer
        self.plotbuffer = np.append(self.plotbuffer, self.ringbuffer)
        # only keep the 500 newest ones and discard the old ones
        self.plotbuffer = self.plotbuffer[-500:]
        self.ringbuffer = []
        # set the new 500 points of channel 9
        self.line.set_ydata(self.plotbuffer)
        self.samp_freq.set_text(f'Sample Rate: {self.samprate:.3f} Hz')

        return self.line,
```

```
    # appends data to the ringbuffer
    def addData(self, v,samp_rate): #xlabel):
        self.ringbuffer.append(v)
        self.samprate = samp_rate



# Create twp instances of an animated scrolling window (one for filtered
data and one for unfiltered data)
realtimePlotWindow_unfiltereddata = RealtimePlotWindow("Unfiltered Data")
realtimePlotWindow_filtereddata = RealtimePlotWindow("Filtered Data")


# sampling rate: 1000Hz
samplingRate = 1000 #better sampling and getting all information in real
time coming in


fc = 45 # lowpass cutoff freqeuncy
fs = 1000 # sampling frequency

sos1 = signal.butter(4, fc/fs*2, 'lowpass', output='sos') # creating IIR
filter coefficients using Butterworth filter
iir1 = iir_filter.IIR_filter(sos1) # instanciating the IIR filter using the
sos coefficients created using Butterworth


counter = 0 # used to find the sampling frequency
samplingFreq = 0 # start at zero and is the result of the first two
counters as division is undefined
time_prev = perf_counter() # remove the transient response and to be
considered as the initial value set outside which is 0
array_fft = np.empty([]) # filling an empty array with unfiltered input
data to find its FFT response
filtered_array_fft = np.empty([]) # filling an empty array with filtered
input data to find its FFT response

# called for every new sample which has arrived from the Arduino
def callBack(data):
    global array_fft
    global filtered_array_fft
    global samplingFreq
    global counter
    global time_prev

    # filling array with unfiltered data
    array_fft = np.append(array_fft,data)

    #if condition that works as a stopwatch and calculates the sampling
frequency
    if counter == 100: # calculate sampling frequency every 100 samples
        counter = 0
        time_now = perf_counter()
        elapsedtime = time_now - time_prev
        time_prev = time_now
        samplingFreq = (100/(elapsedtime))

    counter = counter + 1
```

```
    # filtering of input data:
    filtered_output = iir1.filter(data)

    # filling array with filtered data
    filtered_array_fft = np.append(filtered_array_fft,filtered_output)

    # send the sample to the plotwindow:
    realtimePlotWindow_unfiltereddata.addData(data,samplingFreq) # plotting
unfiltered data
    realtimePlotWindow_filtereddata.addData(filtered_output,samplingFreq) #
plotting filtered output



    #digital pin 10 --> green LED
    #digital pin 9 --> yellow LED
    #digital pin 6 --> red LED

    # light source far away
    if filtered_output < 0.4:
        board.digital[10].write(1) # green LED ON
        board.digital[9].write(0) # OFF
        board.digital[6].write(0) # OFF
        print("Enough Space")

    # light source getting closer
    elif 0.4 < filtered_output < 0.6:
        board.digital[10].write(0) # OFF
        board.digital[9].write(1) # yellow LED ON
        board.digital[6].write(0) # OFF
        print("Take Caution!")

    #light source really close
    elif filtered_output > 0.6:
        board.digital[10].write(0) # OFF
        board.digital[9].write(0) # OFF
        board.digital[6].write(1) # red LED ON
        print("STOP IMMEDIATELY!")


# Get the Ardunio board.
board = Arduino(PORT)

# Set the sampling rate in the Arduino
board.samplingOn(1000 / samplingRate)

# Register the callback which adds the data to the animated plot
board.analog[0].register_callback(callBack)

# Enable the callback
board.analog[0].enable_reporting()

# show the plot and start the animation
plt.show()

# needs to be called to close the serial port
board.exit()

print("Mission Parking: Successful!!")
```

```
# Finding the Fourier Transform of the unfiltered data and plotting it:
fourier_transform = np.fft.fft(array_fft) / len(array_fft)
x_axis = np.linspace(0,1000,len(array_fft))

plt.figure()
plt.title("Unfiltered Frequency Spectrum")
plt.xlim(-5,500)
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude")
plt.plot(x_axis,np.abs(fourier_transform))

# Finding the Fourier Transform of the filtered data and plotting it:
fourier_transform_filtered = np.fft.fft(filtered_array_fft) /
len(filtered_array_fft)
x_axis_filtered = np.linspace(0,1000,len(filtered_array_fft))

plt.figure()
plt.title("Filtered Frequency Spectrum")
plt.xlim(-5,500)
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude")
plt.plot(x_axis_filtered,np.abs(fourier_transform_filtered))


plt.show()
```