

Coursework Declaration and Feedback Form


The Student should complete and sign this part

Student Number : 2427176	Student Name : Nicolas Kyriacou
Course Code : ENG5027	Course Name : Digital Signal Processing
Name of Lecturer : Bernd Porr	Name of Demonstrator :
Title of Assignment : Assignment 3(IIR Filters)	
Assignment :	
Declaration of Originality and Submission Information	
<p><i>I affirm that this submission is all my own work in accordance with the University of Glasgow Regulations and the School of Engineering requirements</i></p> <p>Signed (Student) : </p>	 <p>E N G 5 0 2 7</p>
Date of Submission : 16/12/2018	

<i>Feedback from Lecturer to Student – to be completed by Lecturer or Demonstrator</i>	
<p>Grade Awarded:</p> <p>Feedback (as appropriate to the coursework which was assessed):</p>	
Lecturer/Demonstrator:	Date returned to the Teaching Office:

Coursework Declaration and Feedback Form

The Student should complete and sign this part

Student Number : 2095984	Student Name: Lilian Gutierrez
Course Code : ENG4053	Course Name : Digital Signal Processing 4
Name of Lecturer: Bernd Porr	Name of Demonstrator :
Title of Assignment : Assignment 3(IIR Filters)	
Declaration of Originality and Submission Information	
I affirm that this submission is all my own work in accordance with the University of Glasgow Regulations and the School of Engineering requirements Signed (Student) : At the bottom of the sheet	 E N G 4 0 5 3
Date of Submission : 16/12/2018	

Feedback from Lecturer to Student – to be completed by Lecturer or Demonstrator	
Grade Awarded: Feedback (as appropriate to the coursework which was assessed):	
Lecturer/Demonstrator:	Date returned to the Teaching Office:\

Handwritten signature in black ink on a light brown background.

The Central Heating Boiler Smoke Indicator

Assignment 3 report for the Digital Signal Processing course by by Nicolas Kyriacou 2427176, Lilian Gutierrez 2095984

Part A: Problem Definition and Proposed Solution:

Introduction and Motivation:

Central heating systems are usually based on hot water circulating in the building, radiating heat through heat radiators, where water is heated up by a boiler system. Most boilers use oil as the burning fuel. Improper oil burning can cause smoke coming out from the rooftop vent of the boiler's chimney. This smoke is usually grey or dark coloured. As shown in figure 1 below, the smoke is often accompanied by soot or big pieces of ash. Improper fuel burning can be the result of a number of reasons such as improper mixture of air and fuel in the burner, use of low quality oil or blocked oil nozzles. It can also be the result of improper operation of the burner, such as delayed ignition, resulting in accumulated oil in the burner.

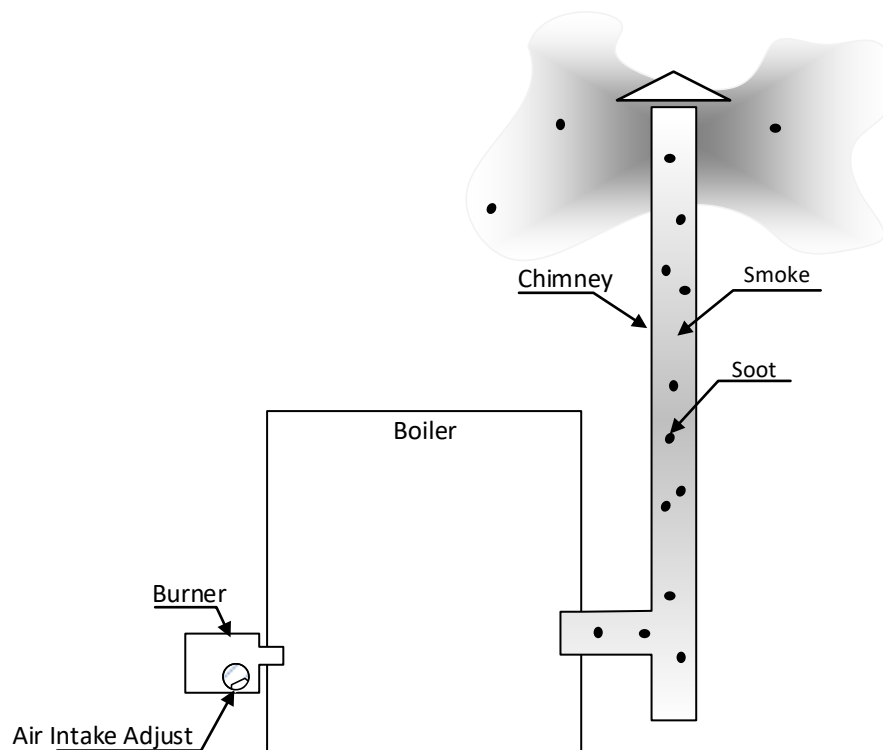


Figure 1: Typical central heating boiler system

Central heating boiler systems must be properly serviced periodically, in order to improve the efficiency of the boiler and avoid the generation of smoke and soot. Excessive smoke and soot coming out of the chimney can create health problems to people and animals, since the smoke and soot might contain toxic substances. They can also contribute to the environment pollution. Furthermore, soot can cause black stains in the area of the chimney and in many times many meters away from the rooftop vent. In extreme cases this improper fuel burning can cause excessive vibrations in the boiler system that can damage the system or even cause a fire.

Central heating technicians, at the last step of the service process examine visually the colour of the smoke coming out of the chimney and try to adjust it by adjusting the air intake to the burner. In many cases the chimney rooftop is not easily visible, creating difficulties in the process of air intake adjustment.

The Central Heating Boiler Smoke Indicator:

The purpose of the application example proposed in this work is to develop a system that will assist the service technician in the adjustment of the boiler air intake, by providing him with a visual indication of the amount and density of smoke produced by the burner. This can be achieved with the use of a light dependant resistor (LDR) and a light source (LED) placed in the chimney, as shown in figure 2 below. The smoke and soot in the chimney will reduce the amount of light hitting the area of the LDR, causing a change in its resistance. The LDR is connected in series with a 10K resistor, forming a voltage divider that provides the analogue input of the microcontroller a voltage that is proportional to the density of the smoke and the amount of soot in the chimney. Based on the voltage level at its analogue input, the microcontroller will change the intensity and colour of a two-colour led connected at its analogue outputs. If the smoke density is low then the led will produce an intense green light. As the density of the smoke increases, the intensity of the green light will decrease. After a set threshold, the light colour will turn to red with its intensity being proportional to the density of the smoke.

It should be noted that for the purpose of this work, the proposed system will provide an indication on the smoke density only. Signals due to the presence of soot or due to possible vibrations of the chimney are ignored (filtered out). A complete solution to the problem could include different indicators for the soot and vibrations.

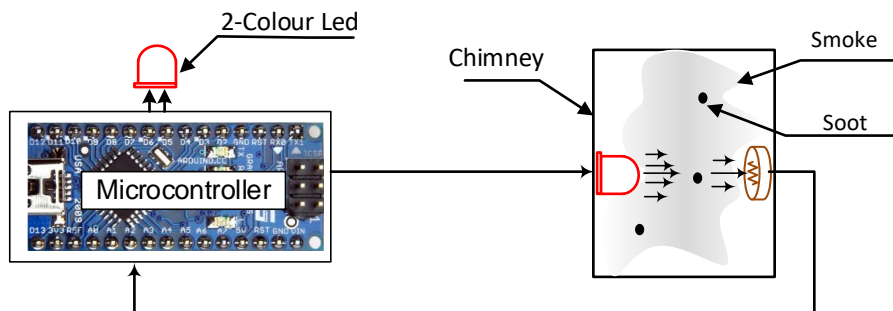


Figure 2: The Central Heating Boiler Smoke Indicator

The flow diagram describing the operation of the system is shown in figure 3. The analogue signal generated by the LDR sensor contains information on the density of the smoke and the soot, as well as possible noise due to vibrations of the chimney. The A/D converter converts the analogue signal into digital codes that are then filtered in order to remove the signal contribution of the soot and the chimney vibrations. Therefore, the signal at the input of the Signal Threshold unit is proportional only to the smoke density. The Signal Threshold unit provides the smoke indicator with analogue voltages that specify the colour and intensity of the light produced by the led.

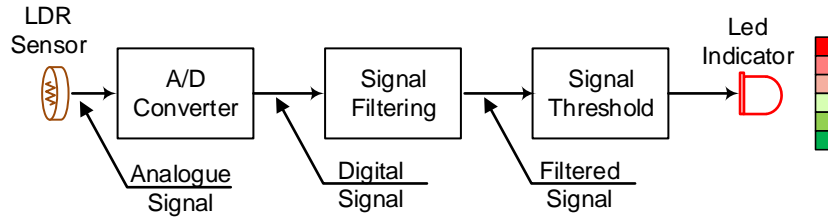


Figure 3: The Smoke Indicator Flow Diagram

It should be noted that in the actual smoke indicator device, the A/D converter unit, the Signal Filter unit and the Signal Threshold unit are parts of the Microcontroller system. For the purpose of this assignment the microcontroller used is the Arduino Nano, while the filtering is performed by a computer running the corresponding filtering commands using Python.

Hardware Used:

The hardware used is based on the Arduino Nano microcontroller system. The circuit diagram of the hardware system is shown in figure 4.

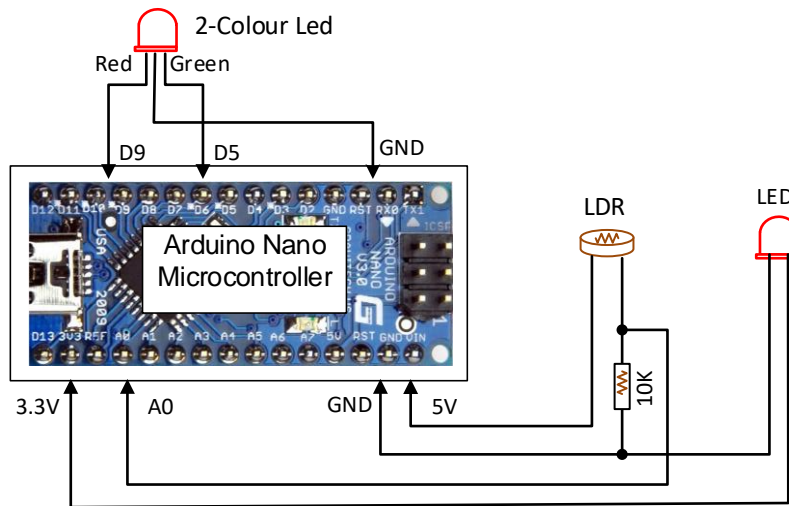


Figure 4: Hardware Circuit

Part B: The A/D Conversion System:

To decide on the appropriate sampling frequency of the A/D converter it is necessary to analyse the frequency characteristics of the signals detected by the LDR sensor in the chimney. This can be achieved either by measuring these signals in a real boiler system, or using analytic calculations based on various assumptions. During the time of the development of this work, there was no access to a real boiler system, therefore, the signal parameters used are primarily based on estimations from previous experience with a faulty central heating boiler system. The signals to be detected by the system are the following:

(a) The signal due to the presence of smoke in the chimney. This signal depends on the density of the smoke. The density of the smoke is relatively constant since it depends primarily on the ratio of fuel to air intake that is preset during the service process. Slow variations in the density of the smoke can be due to possible dirt in the fuel that affects the operation of the oil nozzles, or due to

the frequency of operation of the fuel pump. It is estimated that these fluctuations in the smoke density are of vary amplitude at a frequency near DC, assumed to be 0.1Hz.

(b) The signal due to the presence of soot in the chimney. The frequency of the signal due to the soot depends on the size of the soot particles in relation to the physical size of the LDR and the speed of movement of these particles. For the purpose of this work and based on previous experience it is assumed that these particles travel in the chimney with a speed of approximately 0.25m/s to 0.5m/s. Assuming that the area of the LDR is 5mm, a soot particle will need from 50ms to 100ms to pass by the LDR. Thus the frequency of the signal due to the soot is from 10Hz to 50Hz. The amplitude of these signal depends on the size of the soot particle with respect to the physical size of the LDR.

(c) The signals due to noise created by the possible vibrations of the chimney that could disturb slightly the position of the LDR with respect to the light source. This noise in of very amplitude with a frequency depending on the frequency of the vibration of the chimney. Based on previous experience this vibrations have a frequency of about 10Hz.

Therefore an estimate of the signals present in the chimney is shown figure 5.

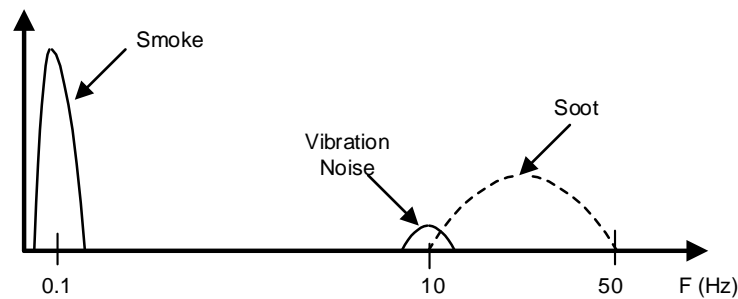


Figure 5: Estimated Frequency Spectrum of Chimney Signals

Based on the above spectrum, the Nyquist frequency can be set to 50Hz, therefore the sampling frequency is set to 100Hz. This sampling frequency is very low compared to the sampling frequency of the A/D converter of the Arduino that has a maximum sampling frequency of 100KHz.

A way to test whether the A/D converter can produce the samples on time without problems, a 50Hz sine wave was generated at the D5 output of the Arduino that was set to generate a PWM analogue voltage, which was the connected to the A0 analogue input of the Arduino through an RC low pass filter, however due to low frequency used by Arduino for the PWM signals, the results were not conclusive. However, since the required sampling frequency is very low compared to the maximum sampling frequency, and since the signal of interest has a frequency of less than one Hz, it is safe to assume that the A/D converter will operate correctly. It should be noted that the expected higher frequencies due to the presence of soot in the smoke will be filtered out anyway.

Part C: IIR Filter Design:

In the proposed system, it is expected to have a smoke signal with a frequency of less than 1Hz (in the range 0.1 Hz approaching the DC), and two unwanted signals, one due to the presence of soot and another due to possible vibrations of the chimney. The frequencies of the unwanted signals is in the range of 10Hz to 50Hz. Therefore a low pass filter is required to remove these unwanted signals. To this end an IIR three stage filter was used.

The main task in the design of the IIR filter involves the calculation of the filter coefficients. These coefficients were obtained using the high level filter functions provided by Python. More specifically,

the function that implements the Chebyshev low pass filter “*cheby2(a,b,c)*” function, where ‘a’ corresponds to the number of group coefficients that specify the filter’s number of stages, ‘b’ corresponds to the required dumping factor in decibels, and ‘c’ corresponds to the normalized sampling frequency.

For the IIR filter designed for this applications the following parameters were used:

- (a) Number of filter stages: 3 (parameter used in filter function = 6)
- (b) Dumping factor: 20db, applied to the cut-off zone since cheby2 is used
- (c) Normalized cut-off frequency = $2 \times \text{Cut-off frequency} / \text{sampling frequency} = 2 \times 1.0 / 100 = 0.02$

The Chebyshev low pass filter was implemented with the code listing shown in Listing 1. To test the produced coefficients, the filter was tested with the “freqz()” python command.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from scipy import signal as signal

# Code segment that generates the coefficients for a 6th order low pass filter based
# on the Chebyshev type 2 filter. The bandstop minimum ripple is set to 20 dbs. The
# cut-off frequency is set to 1Hz with a 100Hz sampling frequency, therefore the
normalized
# frequency is set to  $2 \times 1 / 100 = 0.02$ 
b, a = signal.cheby2(6, 20, 0.02, 'low')
sos = signal.cheby2(6, 20, 0.02, 'low', output = 'sos')
# Code segment that generates the frequency response of the filter to test the generated
# coefficients.
w, h = signal.freqz(b, a)
plt.plot(w/np.pi/2, 20 * np.log10(abs(h)))
plt.xscale('log')
plt.title('Chebyshev frequency response')
plt.xlabel('Normalized Frequency')
plt.ylabel('Amplitude [dB]')
plt.margins(0, 0.1)
plt.grid(which='both', axis='both')
plt.show()
```

Listing 1: Coefficient Generation Code

The normalized frequency response of the filter is shown in figure 6. From the frequency response obtained, it can be verified that the cut-off frequency is at 1Hz (0.01 in the normalized range), where the attenuation is at -20db.

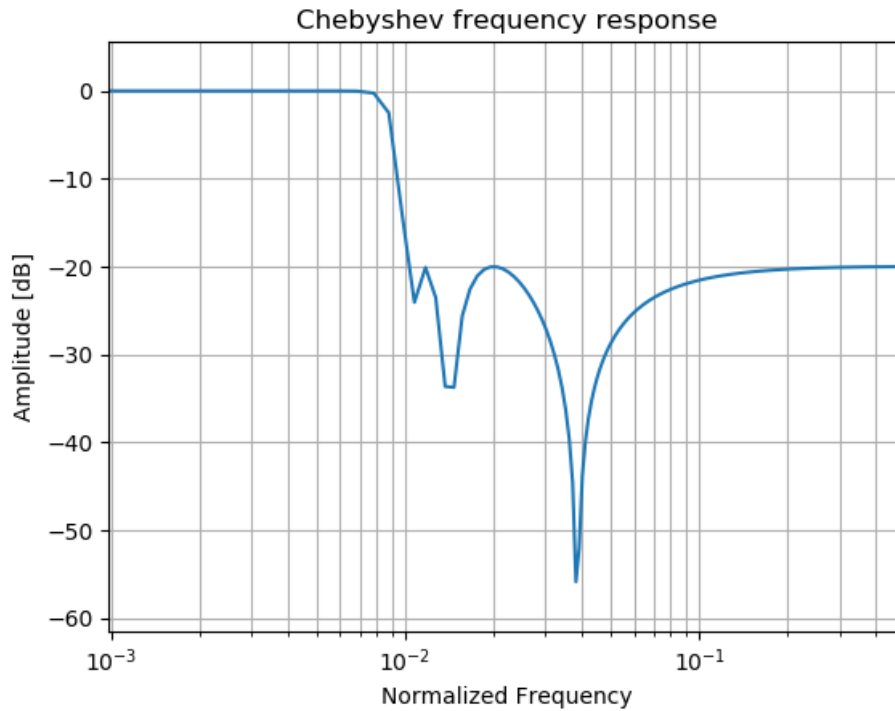


Figure 6: Normalized Frequency of the Chebyshev Filter

The coefficients generated are given below. The first three values correspond to the b0, b1 and b2 coefficients of the first IIR filter stage, followed by the a0, a1 and a2 coefficients. The next values correspond to the coefficients of the other two stages of the filter.

First 2nd Order Coefficients:

0.08930701, -0.17342385, 0.08930701,
1, -1.81861152, 0.82929564,

Second 2nd Order Coefficients:

1, -1.99211469, 1,
1, -1.93685575, 0.9418292,

Third 2nd Order Coefficients:

1, -1.99577041, 1,
1, -1.98279602, 0.98605365

Part D: IIR Filter Classes:

The IIR filter is implemented with two classes. The first one is called 'IIR2Filter' and implements a 2nd order IIR filter. This class takes as input to its constructor the coefficients of the filter and has a method 'IIR2Filter.filter(x)' that computes and returns the IIR filter value for a single sample, using the specified coefficients. The implementation of the 2nd order IIR filter is based on the direct II form IIR filter with a dataflow diagram shown in figure 7.

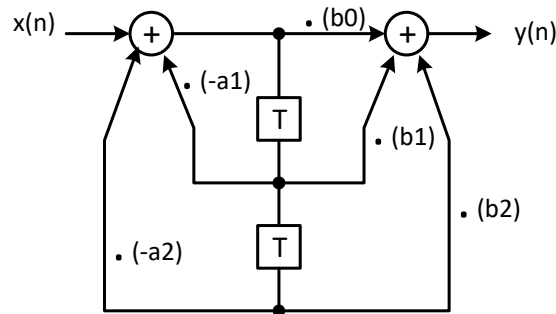


Figure 7: Dataflow Diagram of the 2nd Order IIR Filter

The listing of the IIR2Filter class is shown in listing 2.

```
# Class IIR2Filter computes the output of a 2nd order IIR filter. The filter coefficients  
# are past as parameters during the instantiation of the filter as the constructors  
parameters.  
# The actual filtering operation is carried out by the method 'filter2(x)' that has as input  
# a single sample of the input signal and returns the filtered value.
```

```
class IIR2Filter:
```

```
    def __init__(self, _b0, _b1, _b2, _a1, _a2): # Class constructor
```

```
        self.a1 = _a1    # set filter coefficients
```

```
        self.a2 = _a2    # set filter coefficients
```

```
        self.b0 = _b0    # set filter coefficients
```

```
        self.b1 = _b1    # set filter coefficients
```

```
        self.b2 = _b2    # set filter coefficients
```

```
        self.buffer1 = 0  # Step delay buffer 1
```

```
        self.buffer2 = 0  # Step delay buffer 2
```

```
    def filter2(self, x): # 2nd order IIR filter implementation
```

```
        input_acc = x - self.buffer1*self.a1 - self.buffer2*self.a2
```

```
        output_acc = input_acc*self.b0 + self.buffer1*self.b1 + self.buffer2*self.b2
```

```
        self.buffer2 = self.buffer1 # Forward buffers for next step delay
```

```
        self.buffer1 = input_acc # Forward buffers for next step delay
```

```
        return output_acc # Return filtered value
```

Listing 2: The IIR2Filter Class

To test the IIR2Filter class, an IIR filter was instantiated with the coefficients of the 1st stage of the IIR filter generated in the previous step, and tested with a delta input set in the 10th sample of an array with 100 zeros. The listing for this test program segment is shown in listing 3. The produced impulse response of the IIR 2nd order filter is shown in figure 8.

```
f0 = 0.01
a1 = -1.81861152
a2 = 0.82929564
b0 = 0.08930701
b1 = -0.17342385
b2 = 0.08930701
f=IIR2Filter(b0,b1,b2,a1,a2)
x=np.zeros(100)
x[10] = 1
y=np.zeros(100)
for i in range(len(x)):
    y[i] = f.filter(x[i])
plt.plot(y)
plt.show
```

Listing 3: Testing Code for the IIR2Filter Class

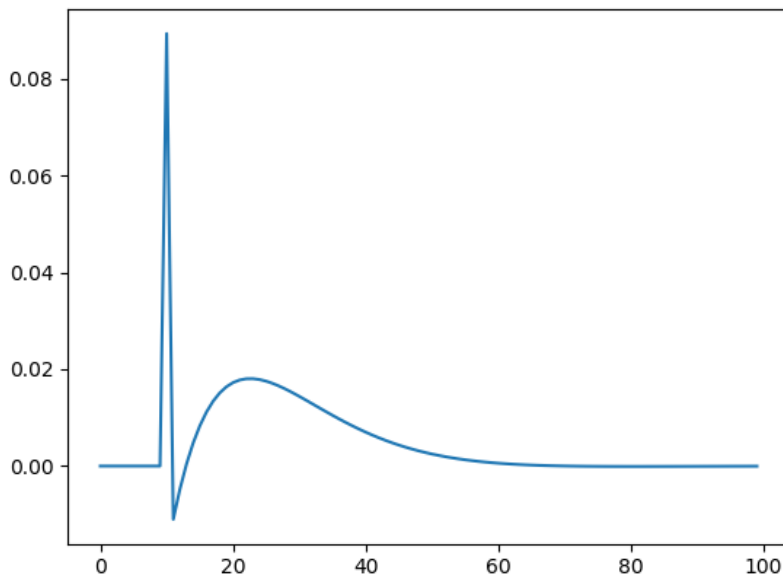


Figure 8: Impulse Response of the 2nd Order IIR Filter

The second class 'IIRFilter' takes the sos coefficients created by the Chebyshev filter design python high level commands as its constructor argument and creates a chain of 2nd order instances of the IIRFilter class. The implementation of the 6th order IIR filter is based on the IIR2Filter class with a dataflow diagram shown in figure 9. The listing of the IIRFilter class is shown in listing 4.

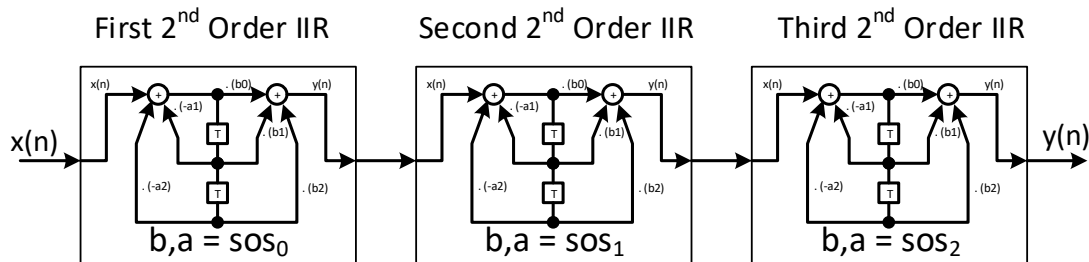


Figure 9: Dataflow Diagram of the 6th Order IIR Filter

```
# Class IIRFilter creates a chain of 2nd order IIR filters specified in class IIR2Filter.
# For each 2nd order filter it updates the filter coefficients and calls the 'filter2' method
# to compute the filtered output.
```

```
class IIRFilter:
```

```
    def __init__(self, _coeffs):
```

```
        self.coeffs = _coeffs
```

```
        self.in_buff = np.zeros(100)
```

```
        self.out_buff = np.zeros(100)
```

```
        self.f=IIR2Filter(b0,b1,b2,a1,a2)
```

```
    def filter(self,x1):
```

```
        self.in_buff[0]=x1 # Assign new input data to the input buffer of the first 2nd order
filter.
```

```
# Set the b,a coefficients and filter the input sample.
```

```
    for i_order in range(int(len(self.coeffs)/5)):
```

```
        IIR2Filter.a1 = self.coeffs[i_order*5+3]
```

```
        IIR2Filter.a2 = self.coeffs[i_order*5+4]
```

```
        IIR2Filter.b0 = self.coeffs[i_order*5+0]
```

```
        IIR2Filter.b1 = self.coeffs[i_order*5+1]
```

```
        IIR2Filter.b2 = self.coeffs[i_order*5+2]
```

```
        self.out_buff[i_order] = self.f.filter2(self.in_buff[i_order])
```

```
# Forward the output buffer of each 2nd order filter to the input buffer of the next one.
```

```
    for i_order in range(int(len(self.coeffs)/5)):
```

```
        self.in_buff[i_order+1] = self.out_buff[i_order]
```

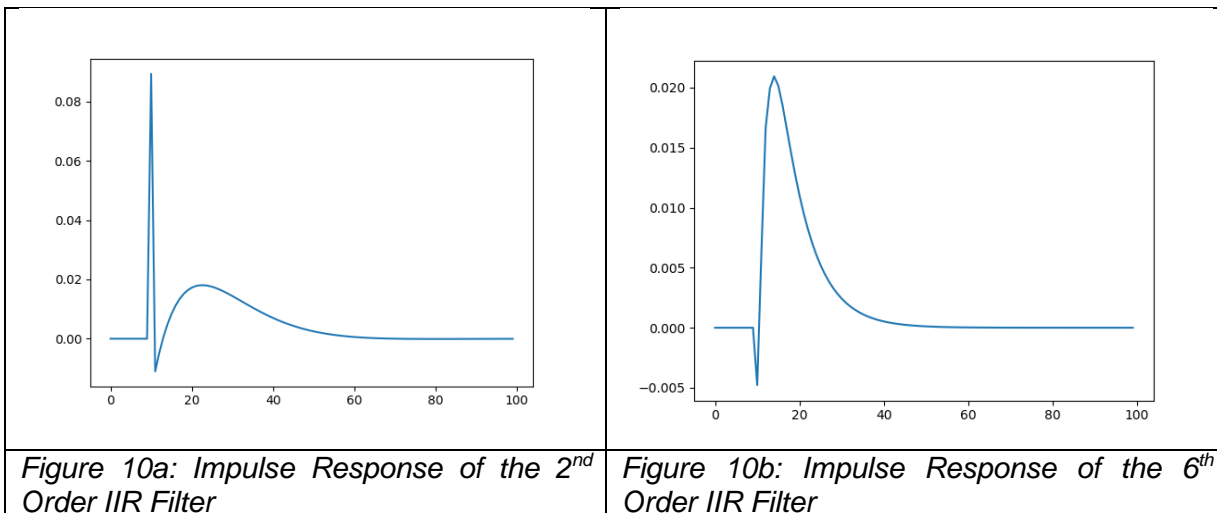
```
    return self.out_buff[i_order]
```

Listing 4: The IIRFilter Class

To test the IIRFilter class, an IIR filter was instantiated with the coefficients of the IIR filter (coefficients in the sos array), and tested with a delta input set in the 10th sample of an array with 100 zeros. The listing for this test program segment is shown in listing 5. The produced impulse response of the IIR filter is shown in figure 10. Figure 10a shows the impulse response of the filter implemented by the class IIRFilter with only the first five coefficients specified, thus a 2nd order filter. The produced impulse response is the same as the one produced with the IIR2Filter class shown in figure 8, therefore the class IIRFilter works correctly. Figure 10b shows the impulse response for a 6th order IIR filter.

```
coeffs = [0.08930701,0.08930701,0.08930701,-1.81861152, 0.82929564,
          -1.93685575,0.9418292,1,-1.9921146,1,
          -1.98279602,0.98605365,1,-1.99577041,1]
f=IIRFilter(coeffs)
x=np.zeros(100)
x[10] = 1
y=np.zeros(100)
for i in range(len(x)):
    y[i] = f.filter(x[i])
plt.plot(y)
plt.show
```

Listing 5: Testing Code for the IIRFilter Class

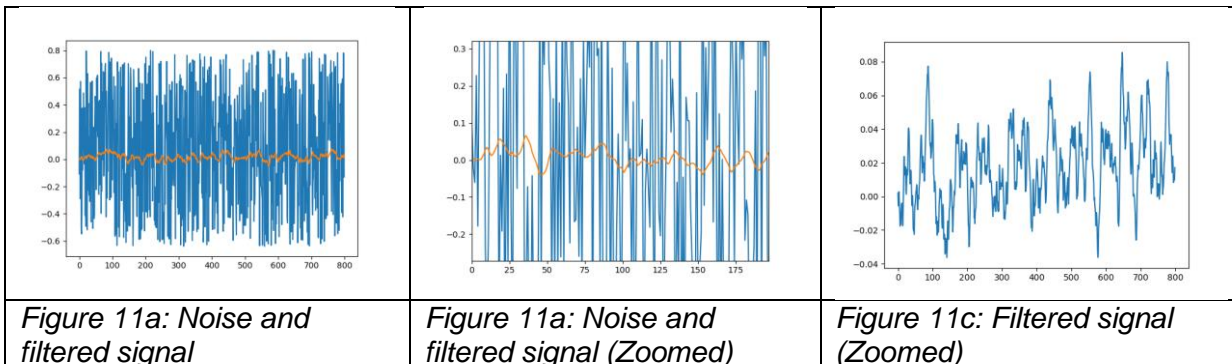


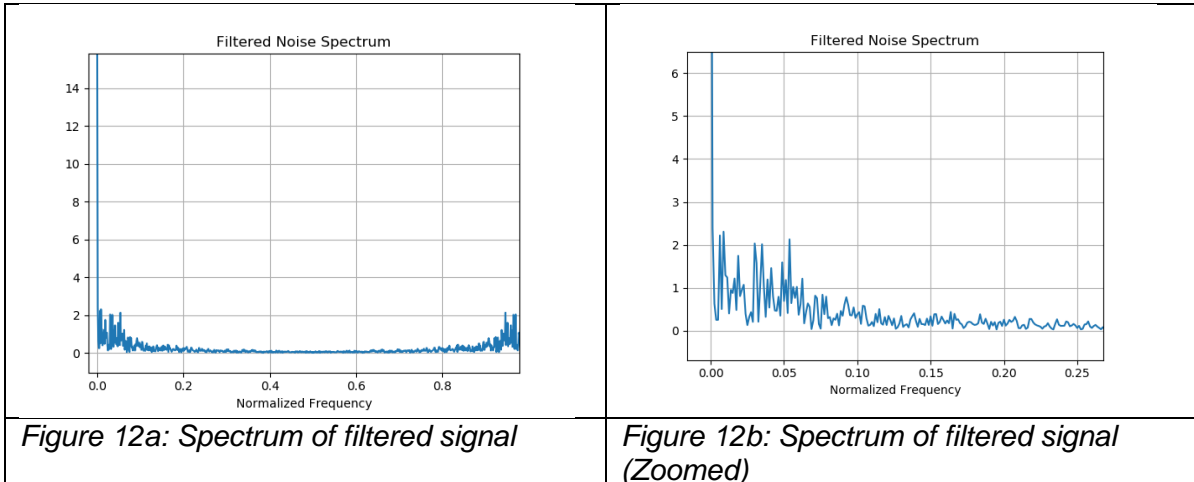
To verify that the developed class implements correctly a low pass filter with a 1Hz cut-off frequency, the filter was tested with the code segment shown in listing 6. This code generates a signal consisting of white noise that contains virtually all frequencies. The time domain signal of the generated noise is shown in figure 11a (blue colour), while the filtered time domain signal is also shown in figure 11a (orange colour). Figure 11b shows a zoomed segment of the time domain signals of figure 11a. Figure 11c shows the filtered signal alone. From this time domain signals it can be seen that the noise signal waveform is very dense, with peak values not very dense, indicating that this signals is composed from a sine waves of a wide range of frequencies. From the

waveform of the filtered signal it can be seen that it changes at a much slower rate than the initial unfiltered signal, indicating that it follows the low frequency changes of the noise signal. Therefore, the developed filter is a low pass filter with a very low cut-off frequency. To investigate further the characteristics of the developed filter, the spectrum of the filtered signal was produced using the FFT function. As indicated by the filtered signal spectrum, in figure 12, the filtered signal consists of primarily a dc component and low frequency components with frequencies less than approximately 5Hz (0.05 in the normalized frequency).

```
# Generate noise with a 0.2 dc offset (range from -800 to 1000)
sample = 800
n = np.arange(sample)
noise = 0.0008*np.asarray(random.sample(range(-800,1000),sample))
y=np.zeros(800)
plt.figure(1)
plt.plot(noise)
plt.show
# Filter the generated noise and plot the time domain output
f=IIRFilter(coeffs)
for i in range(len(noise)):
    y[i] = f.filter(noise[i])
plt.figure(2)
plt.plot(y)
plt.show
N = np.fft.fft(abs(y))
# Use FFT to plot the frequency spectrum of the filtered noise.
plt.figure(3)
plt.plot(n/sample,abs(N))
plt.title('Filtered Noise Spectrum')
plt.xlabel('Normalized Frequency')
plt.margins(0, 0.01)
plt.grid(which='both', axis='both')
plt.show
```

Listing 6: Testing Code for the IIRFilter Frequency Response





Part E: Application Software Testing:

To implement the Central Heating Smoke indicator application, a python software was developed that uses 'pyfirmata2' class to communicate and control the operation of the Arduino platform. The oscilloscope implementation uses the class 'RealtimePlotwindow' taken from the 'analog_realtime_scope.py' program. The listing of the application software is given in Appendix I. The code segment of the application is shown in listing 6. This part of the program configures the Arduino, and reads the signal received from the analogue input [0]. The received signal is real time filtered by a 6th order low pass filter and displayed on the oscilloscope. According to the valued at the output of the filter which is proportional to the density of the smoke, the program switches On or Off a two colour led, with varying light intensities.

```
# Create an instance of an animated scrolling window
# To plot more channels just create more instances and add callback handlers below
realtimePlotWindow = RealtimePlotWindow()
#realtimePlotWindow2 = RealtimePlotWindow()

# sampling rate: 100Hz
samplingRate = 100
gain = 4
# our callback where we filter the data
def callBack(data):
    y = f.filter(data)
    y=gain*y
    #print(y)
    realtimePlotWindow.addData(y)
    if y>=0.70: # Turn on/off the green and red leds according to the signal received.
        LED_green.write(y/1.5)
        LED_red.write(0)
    else:
        LED_green.write(0)
        LED_red.write((1-y)/1.5)

# Get the Ardunio board
board = Arduino("\\.\\COM3')

# Define digital pins D5 and D9 as analogue (PWM) outputs
LED_green = board.get_pin('d:5:p')
LED_red = board.get_pin('d:9:p')
```

```
# 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()

print("finished")
```

Because there was no access to areal central heating system the operation of the system was tested by emulating the smoke using a transparent film painted with colours of varying density. This film was placed in between the LDR and the light source, and moved back and forth to emulate the movement of the smoke in the chimney. The testing processes war recorded in the attached video. Two typical capture instances of the oscilloscope screen is shown in figure 13. From the waveform in figure 13 it can be seen that when the film was moved at a low speed the amplitude of the signal was reduced significantly, indicating the density of the smoke. When the move was moved with a high speed, the variation of the amplitude of the signal was low, due to the operation of the low pass filter.

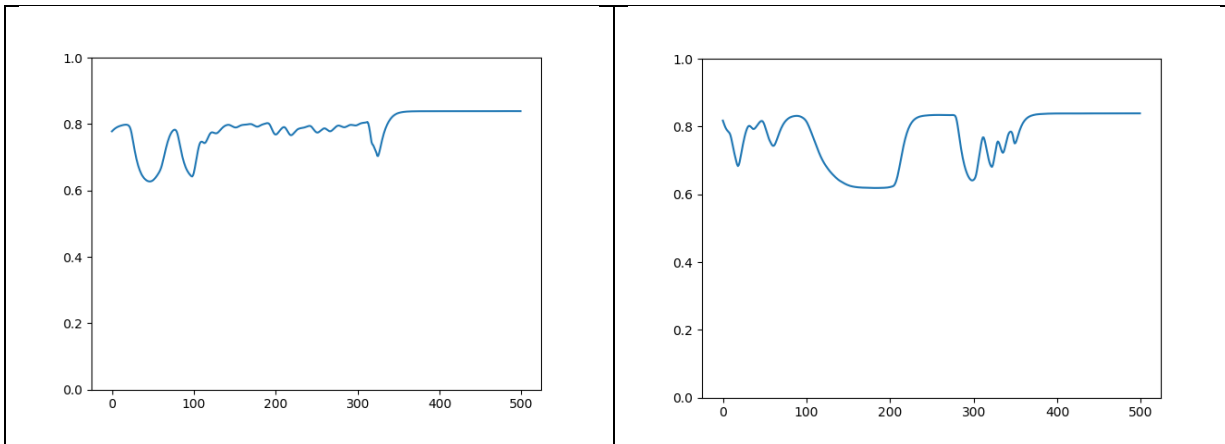


Figure 13: Oscilloscope Captures

The operation of the two-colour led was also tested and found to operate correctly. When the input signal was below a set threshold the colour of the led was red, with its intensity increasing for lower amplitude signals. This indicates the presence of a high density smoke in the chimney. When the input signal was above the set threshold the led colour was green, with its intensity increasing for higher amplitude signals. This indicates that the density smoke in the chimney is low.

Even though the software developed was not tested in a real central heating system, the testing of the operation of the filters and the testing of the final operation of the system indicates that the developed system operates correctly.

Appendix I: Listing of the Application Program

*# This is the application testing software for the ' Central Heating Boiler Smoke Indicator'
Its purpose is to measure the density of the smoke coming out of the central heating boiler
and provide a visual indication, in order to assist the maintenance technician to adjust the
air intake to the burner, and thus reduce the volume of smoke.*

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

*# Class IIR2Filter computes the output of a 2nd order IIR filter. The filter coefficients
are past as parameters during the instantiation of the filter as the constructors parameters.
The actual filtering operation is carried out by the method 'filter2(x)' that has as input
a single sample of the input signal and returns the filtered value.*

```
class IIR2Filter:
    def __init__(self, _b0, _b1, _b2, _a1, _a2): # Class constructor
        self.a1 = _a1 # set filter coefficients
        self.a2 = _a2 # set filter coefficients
        self.b0 = _b0 # set filter coefficients
        self.b1 = _b1 # set filter coefficients
        self.b2 = _b2 # set filter coefficients
        self.buffer1 = 0 # Step delay buffer 1
        self.buffer2 = 0 # Step delay buffer 2
    def filter2(self, x): # 2nd order IIR filter implementation
        input_acc = x - self.buffer1*self.a1 - self.buffer2*self.a2
        output_acc = input_acc*self.b0 + self.buffer1*self.b1 + self.buffer2*self.b2
        self.buffer2 = self.buffer1 # Forward buffers for next step delay
        self.buffer1 = input_acc # Forward buffers for next step delay
        return output_acc # Return filtered value
```

*# Class IIRFilter creates a chain of 2nd order IIR filters specified in class IIR2Filter.
For each 2nd order filter it updates the filter coefficients and calls the 'filter2' method
to compute the filtered output.*

```
class IIRFilter:
    def __init__(self, _coeffs):
        self.coeffs = _coeffs
        self.in_buff = np.zeros(100)
        self.out_buff = np.zeros(100)
        self.f=IIR2Filter(b0,b1,b2,a1,a2)

    def filter(self, x1):
        self.in_buff[0]=x1 # Assign new input data to the input buffer of the first 2nd order filter.
```

```
# Set the b,a coefficients and filter the input sample.
for i_order in range(int(len(self.coeffs)/5)):
    IIR2Filter.a1 = self.coeffs[i_order*5+3]
    IIR2Filter.a2 = self.coeffs[i_order*5+4]
    IIR2Filter.b0 = self.coeffs[i_order*5+0]
    IIR2Filter.b1 = self.coeffs[i_order*5+1]
    IIR2Filter.b2 = self.coeffs[i_order*5+2]
    self.out_buff[i_order] = self.f.filter2(self.in_buff[i_order])
```

```

# Forward the output buffer of each 2nd order filter to the input buffer of the next one.
for i_order in range(int(len(self.coefts)/5)):
    self.in_buff[i_order+1] = self.out_buff[i_order]

return self.out_buff[i_order]

```

```

# RealtimePlotwindow is a realtime oscilloscope that displays the signal received
# from an analogue input of the Arduino board.
# This class is taken from the 'analog_realtime_scope.py' program
# Creates a scrolling data display

```

```

class RealtimePlotWindow:

```

```

    def __init__(self):
# 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)
# That's our ringbuffer which accumulates 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)

# 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)
        return self.line,

# appends data to the ringbuffer
    def addData(self, v):
        self.ringbuffer.append(v)

```

```

# Specify the sos filter coefficients and instantiate the IIR filter.
a1 = -1.81861152 # This part of the code is inserted here to avoid error messages
                # concerning the a,b coefficient definition
a2 = 0.82929564
b0 = 0.08930701
b1 = -0.17342385
b2 = 0.08930701
coefts = [0.08930701,-0.17342385,0.08930701,-1.81861152, 0.82929564,
          -1.93685575,0.9418292,1,-1.9921146,1,
          -1.98279602,0.98605365,1,-1.99577041,1]
f=IIRFilter(coefts)

```

```

# Create an instance of an animated scrolling window
# To plot more channels just create more instances and add callback handlers below
realtimePlotWindow = RealtimePlotWindow()
#realtimePlotWindow2 = RealtimePlotWindow()

# sampling rate: 100Hz
samplingRate = 100
gain = 4
in_array = np.zeros(1000)
filtered_array = np.zeros(1000)
ar_index = 0
# our callback where we filter the data
def callBack(data):
    y = f.filter(data)
    y=gain*y
    realtimePlotWindow.addData(y)
    if y>=0.70: # Turn on/off the green and red leds according to the signal received.
        LED_green.write(y/1.5)
        LED_red.write(0)
    else:
        LED_green.write(0)
        LED_red.write((1-y)/1.5)

# Get the Arduino board
board = Arduino("\\.\\COM3')

# Define digital pins D5 and D9 as analogue (PWM) outputs
LED_green = board.get_pin('d:5:p')
LED_red = board.get_pin('d:9:p')

# 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()

print("finished")

```