Digital Signal Processing Report

Assignment 3 (IIR Filter)

By: Minhao Han(2542073H),
Boxun Tong(2636175T),
Junhao Cheng(2614599C),
Junwen Yao(2629073Y)

Course: Digital Signal Processing
Course Conveners: Dr Bernd Porr, Dr Nicholas Bailey

Date: Jan-3-2022

In the following section of the report, we will show our steps and code with brief explanation.

1. Project Description

This project aims to implement a simple protocol to encode zero-one sequence to a signal emited by a LED light(hw-479), then receive the signal with the help of a LDR(light-dependent resistor) and decode it back to the original sequence.

Setup

An Ardunio Uno board is used to connect all the components up. The setup is shown as the Figure 1. To be specific, red, green and blue pins of the LED are connected separately to Digital pin 9, 10, 11 together with LDR output connected to Analogue Input pin 0. No direct logic is hardcoded in the Arduino board. Instead, a Firmata protocol program is uploaded to Ardunio which enables the board to communicate with PC and thus the PC side takes the responsibility to control the whole logic part. PyFirmata2 is the package used in this project which is implemented to communicate based on the Firmata protocol.

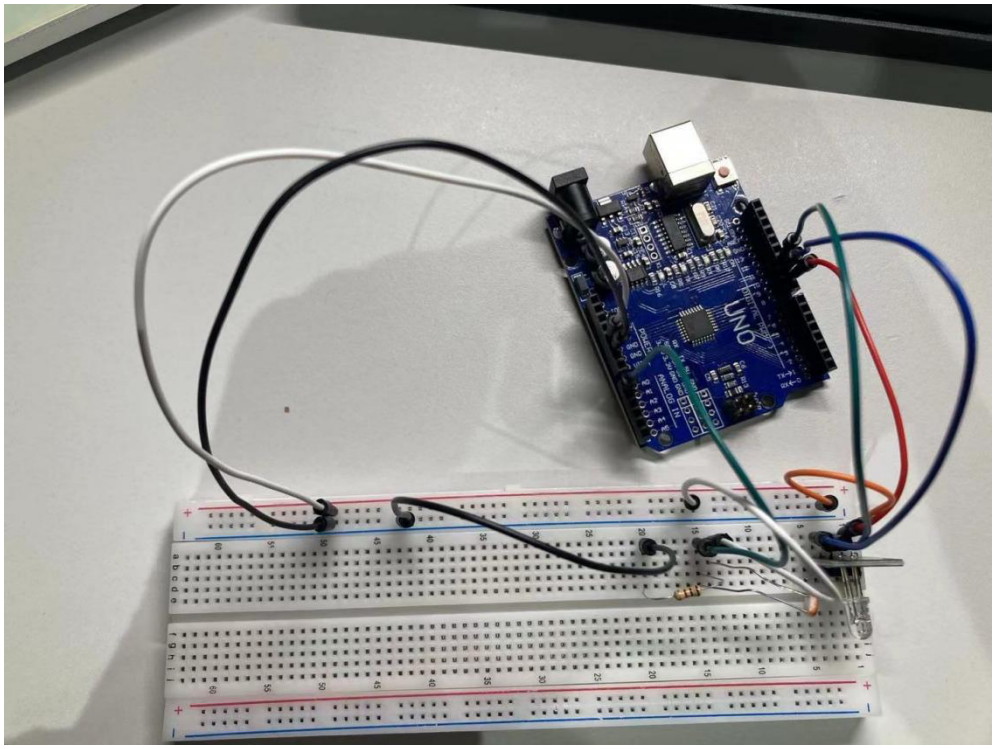

Figure 1. Setup

Dataflow

As shown in Figure 2, signal comes and is encoded from the PC, then emitted by the LED light. After that, the signal is received by the LDR and sent to the IIR filter as analogue input, and finally the filtered signal is processed by the PC.

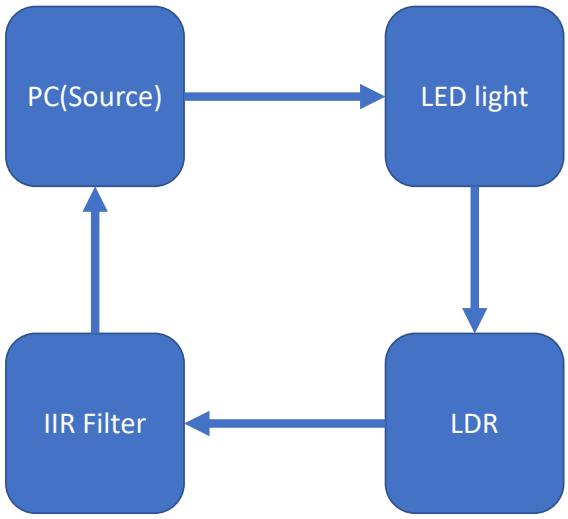

Figure 2. dataflow

Video Clip

A short presentation can be found in the following link:

https://youtu.be/6JyAdl4u0hc.

It shows the basic running process of the whole system.

Unit test

The unit test loads data from an file and then sends data one by one into the IIR filter instance. The result will be stored in a list and later compared with the expected output file. And if all output pairs are equal, the test is passed (as shown in Figure 3).

```
$ python rununittest.py -v
test_filter (__main__.TestIIRFilter) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
```

Figure 3. Unit test result

2. Challenges and solutions

Filter Selection

The original signal from the LED light and the ambient lights is quiet steady although some high frequency noises from light bulbs. And the decoder strategy requires the high voltage output should be steady so that they could be easily recognized as a consecutive sequence of same-valued points. So the main duty of the filter is to

1. cancel the high frequency noises from light bulbs
2. keep the signal steady

A lowpass filtered is introduced here to remove the high frequency noises. However, while trying to replace the lowpass filter with a bandpass filter, which could remove the DC signal as well, the high voltage becomes extremely unstable. So, the final choice is a lowpass filter with a cutoff frequency of 50Hz. The original signal and the filtered signal can be seen in Figure 4.

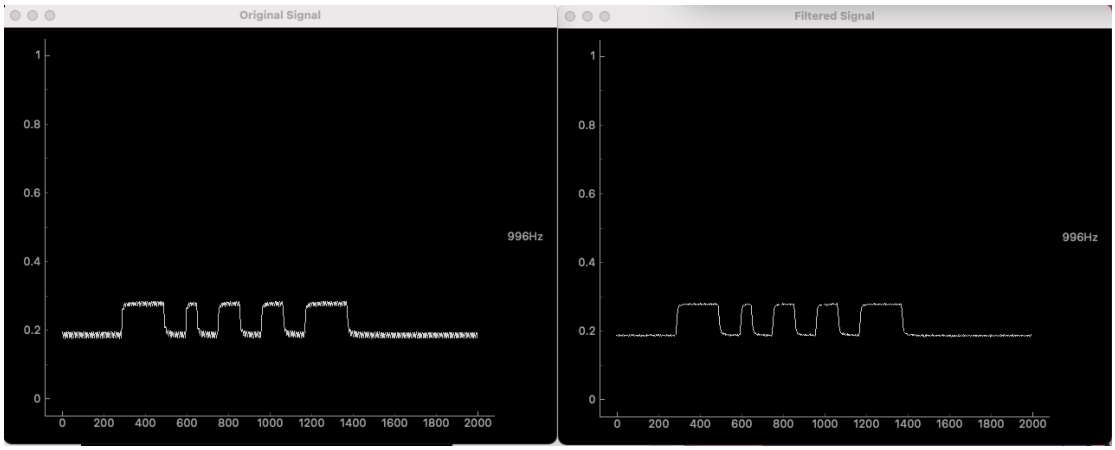

Figure 4. Original Signal(left) and Filtered Signal(right)

Ambient Light Influence

Since it is the LDR adopted in this project, the value of the resistor or the input of the analogue signal is totally depended on the amount of light that is cast to the LDR, while the ambient light level could change by simply turning on or off a light bulb and, as a consequence, it could influence the value of the resistor when LED is turned on and transmitting signals.

The program first implements a function to determine the ambient light level change and uses it as the value of low voltage input.

Then the protocol defines that before starting transmission, a 200ms-long high voltage signal, which is called initiate signal, should be emitted so that the

receiver could record the high voltage value used in this transmission(as the red block in Figure 5).

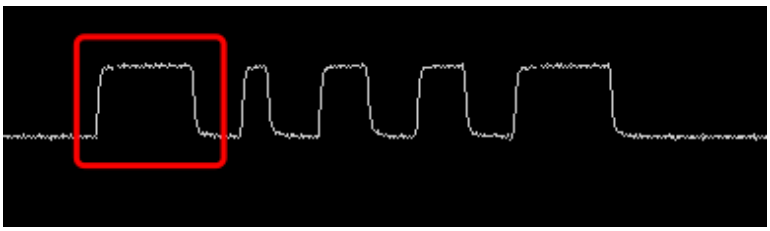

Figure 5. Initiate Signal

3. Protocol Specifications

The protocol decodes the signal based on the duration of a period of high voltage. For example, a 200ms-long, which is 200 points at the sampling rate of 1kHz, will be decoded as the CTRL signal to represent a start or an end flag of a transimission.

Ambient Light Change Detection

The ambient light change detection requires a buffer of 500 samples, which gives the process a delay of 500ms. The detection first calculates the sum of the 500 samples and get the average number within O(1) times, then calculates the deviation of the whole list compared to the average number within O(n) times. Finally, with the help of preset argument deviation_tolerance and level_step, the detection function is implemented. The code is here:

```
def _detect_environment_level_change(self, data, data_to_process):
        if data_to_process == -1:
            self.buffer_sum += data
        else:
            self.buffer_sum = self.buffer_sum + data - data_to_process
        self.buffer_avg = self.buffer_sum / len(self.buffer)

        if data_to_process == -1:
            return

        # calculate deviation
        buffer_deviation = 0
        for num in self.buffer:
            buffer_deviation += abs(num - self.buffer_avg)
        buffer_deviation /= self.buffer_size

        # determine whether env level changes
        if buffer_deviation < self.deviation_tolerance and abs(self.buffer_avg - self.en
    v_level) > self.level_step:
```

```
            self.env_level = self.buffer_avg
            self.log(f"env level changes to: {self.env_level}")
```

Continuous Signal Detection

Continuous signal detection reuses the configuration argument of deviation_tolerance and level_step, all the steady signals except value of ambient light which last longer than 10ms will be recognized as a continuous signal. However, only signals with correct duration would be processed. The code is here:

```
def _detect_continuous_signal(self, data):
        if abs(data - self.env_level) < self.deviation_tolerance:
            return -1, -1

        if abs(data - self.high_vol_avg) > self.deviation_tolerance:
            if self.high_vol_ctr >= 10:
                self.stop_ctr += 1
            else:
                self.high_vol_avg = data
                self.high_vol_ctr = 1
        else:
            self.high_vol_avg = ((self.high_vol_avg * self.high_vol_ctr) + data) / (self.high_vol_ctr + 1)
            self.high_vol_ctr += 1

        if self.stop_ctr >= 3:
            h = self.high_vol_ctr
            self.high_vol_ctr = 0
            self.stop_ctr = 0
            return self.high_vol_avg, h

        return -1, -1
```

4. Analyze and Remarking

The result, as shown in the video, is successful. Compared to original signal, filtered signal is more steady, which makes it easy to recognize as a continuous high voltage signal.

One flash point in this project is the Initiate Signal to determine the high voltage value for transmission. This reduces the influence of the ambient light level. Furthermore, if possible, quantisation can be introduced to enable higher bandwidth.

However, there are quiet a lot of improvements that could be made.

1) Since the ambient light level is detected, the low voltage part could also be used to represent some values (e.g. 0) to increase the bandwidth.
2) Instead of using the difference of duration to distinguish different signals, one simple peak is enough to accomplish this while further effort should be devoted to determine how long it will take for the resistor to reach the peak and conclude higher peaks and lower peaks as one same value.

Appendix I. AnalogPrinter.py

```
import time
from iir_filter import IIR_filter
from scipy import signal

from Decoder import Decoder

class AnalogPrinter:

    def __init__(self, board, original_signal_plot, filtered_signal_plot, decoder: Decoder):
        # sampling rate: 1000Hz
        self.samplingRate = 1000
        self.timestamp = 0
        self.board = board

        # bind output plot
        self.original_signal_plot = original_signal_plot
        self.filtered_signal_plot = filtered_signal_plot

        # bind decoder
        self.decoder = decoder

        # cut off frequency: 50Hz
        cut_off = 50
        sos = signal.butter(2, cut_off/self.samplingRate*2, 'lowpass', output = 'sos')
        self.iir = IIR_filter(sos)

    def start(self):
        self.board.analog[0].register_callback(self.callback)
        self.board.samplingOn(1000 / self.samplingRate)
        self.board.analog[0].enable_reporting()

    def callback(self, data):
        now = time.time()
        filtered_data = self.iir.filter(data)

        self.original_signal_plot.addData(data, now)
        self.filtered_signal_plot.addData(filtered_data, now)
        self.decoder.decode(filtered_data)
```

Appendix II. voweldetector.py

```
1.  #voweldetector:takes an audio file as an input and outputs the vowel
2.  import wave as wv
3.  import numpy as np
4.
5.  class voweldetector():
6.      def __init__(self, filename):
7.          with wv.open(filename) as f:
8.              self.nframes = f.getnframes()
9.              self.framerate = f.getframerate()
10.             self.frames = f.readframes(self.nframes)
11.             self.framerates = np.fromstring(self.frames, dtype=np.short)
12.         # a e i o u
13.         self.vowel = np.array([[275,1095],[128,253],[277,557],[556,276],[269
    ,537]])
14.         self.signals_fft = np.fft.fft(self.framerates)
15.
16.     # function to find fundamental frequency
17.     # sig_flag: FFTsignal sr:framerate sn:totalframes
18.     def fun_fre_seeker(self, sig_flag, sr, sn):
19.         flag1 = 0
20.         flag2 = 0
21.         i = 1
22.         # find the first peak
23.         while i < len(sig_flag) / 2:
24.             if sig_flag[i] > sig_flag[flag1]:
25.                 flag1 = i
26.             i = i + 1
27.         j = 1
28.         # find the second peak also exclude the nearby point of the first pe
    ak
29.         while j < len(sig_flag) / 2:
30.             if sig_flag[j] > sig_flag[flag2] and abs(j - flag1) > 30:
31.                 flag2 = j
32.             j = j + 1
33.         # make the unit of output to be Hz
34.         return int(flag1 * sr / sn), int(flag2 * sr / sn)
35.
36.     def distance_cal(self,x1,x2,vow1,vow2):
37.         return int(((x1-vow1)**2+ (x2-vow2)**2 )**0.5)
38.
39.     # make the output of string
40.     def name_output(self, index):
41.         if(index == 0):
42.             print("This is vowel of a")
43.         if (index == 1):
44.             print("This is vowel of e")
45.         if (index == 2):
46.             print("This is vowel of i")
```

```
        if (index == 3):
            print("This is vowel of o")
        if (index == 4):
            print("This is vowel of u")

    # distinguish the class of input file
    def class_distribution(self):
        peak1, peak2 = self.fun_fre_seeker(self.signals_fft,self.framerate,self.nframes)
        print(peak1, peak2)
        for i in range(len(self.vowel)):
            if(self.distance_cal(peak1, peak2,self.vowel[i][0],self.vowel[i][1]) < 50):
                distance = self.distance_cal(peak1, peak2,self.vowel[i][0],self.vowel[i][1])
                out_close =  i
                #Take the closest value by judging once again
                for j in range(len(self.vowel)-i):
                    if (self.distance_cal(peak1, peak2, self.vowel[i+j][0], self.vowel[i+j][1]) < distance):
                        out_close = i+j
                return self.name_output(out_close)
        print("unknown vowel")

wave = voweldetector('a.wav')
wave.class_distribution()
wave = voweldetector('e.wav')
wave.class_distribution()
wave = voweldetector('i.wav')
wave.class_distribution()
wave = voweldetector('au.wav')
wave.class_distribution()
wave = voweldetector('u.wav')
wave.class_distribution()
```

Appendix Ⅲ. Decoder.py

```
import numpy as np

class Decoder:
    def __init__(self, buffer_size = 500, log_func = print, level_step = 0.003, deviation_tolerance = 0.01,
                 time_offset = 13, init_duration = 200, one_duration = 100, zero_duration = 50):
        self.buffer = []
        self.env_level = 0
        self.level_step = level_step
        self.buffer_size = buffer_size
        self.log = log_func
        self.deviation_tolerance = deviation_tolerance

        self.buffer_sum = 0
        self.buffer_avg = 0
        self.ctr = 0

        self.status = "IDLE"
        self.last_active = 0
        self.output = ""

        # callback functions
        self._on_decode_start = []
        self.on_decode_end = []

        # for detecting high voltage value
        self.high_vol_avg = 0
        self.stop_ctr = 0
        self.high_vol_ctr = 0
        self.high_vol = 0

        # decoding according to signal duration
        self.time_offset = time_offset
        self.init_duration = init_duration
        self.one_duration = one_duration
        self.zero_duration = zero_duration

    def _detect_environment_level_change(self, data, data_to_process):
        if data_to_process == -1:
            self.buffer_sum += data
        else:
            self.buffer_sum = self.buffer_sum + data - data_to_process
        self.buffer_avg = self.buffer_sum / len(self.buffer)

        if data_to_process == -1:
```

```
            return

        # calculate deviation
        buffer_deviation = 0
        for num in self.buffer:
            buffer_deviation += abs(num - self.buffer_avg)
        buffer_deviation /= self.buffer_size

        # determine whether env level changes
        if buffer_deviation < self.deviation_tolerance and abs(self.buffer_avg - self.env_level) > self.level_step:
            self.env_level = self.buffer_avg
            self.log(f"env level changes to: {self.env_level}")

    def _switch_to(self, to_status):
        if to_status == "IDLE":
            self.status = to_status
            for func in self.on_decode_end:
                func(self.output)
            print(self.output)
            self.output = ""
        elif to_status == "RUNNING":
            self.status = to_status
            for func in self._on_decode_start:
                func()
            self.last_active = self.ctr

    def register_on_decode_start(self, func):
        self._on_decode_start.append(func)

    def register_on_decode_end(self, func):
        self.on_decode_end.append(func)

    def decode(self, data):
        data_to_process = self._add(data)
        self.ctr += 1
        self._detect_environment_level_change(data, data_to_process)

        # jump out if no data to process
        if data_to_process == -1:
            return

        self._decode(data_to_process)

    def _decode(self, data):
        if self.status == "IDLE":
            level, duration = self._detect_continuous_signal(data)
```

```
            if (level, duration) != (-1, -1) and abs(duration - self.init_duration) < self.time_offset:
                self.high_vol = level
                self._switch_to("RUNNING")
        elif self.status == "RUNNING":
            if self.ctr - self.last_active >= 2000:
                self.log("decoding timeout, switching to IDLE")
                self._switch_to("IDLE")

            level, duration = self._detect_continuous_signal(data)
              if (level, duration) != (-1, -1) and abs(level - self.high_vol) < self.deviation_tolerance:
                self.last_active = self.ctr
                if abs(duration - self.one_duration) < self.time_offset:
                    self.output += "1"
                elif abs(duration - self.zero_duration) < self.time_offset:
                    self.output += "0"
                elif abs(duration - self.init_duration) < self.time_offset:
                    self._switch_to("IDLE")

    def _detect_continuous_signal(self, data):
        if abs(data - self.env_level) < self.deviation_tolerance:
            return -1, -1

        if abs(data - self.high_vol_avg) > self.deviation_tolerance:
            if self.high_vol_ctr >= 10:
                self.stop_ctr += 1
            else:
                self.high_vol_avg = data
                self.high_vol_ctr = 1
        else:
            self.high_vol_avg = ((self.high_vol_avg * self.high_vol_ctr) + data) / (self.high_vol_ctr + 1)
            self.high_vol_ctr += 1

        if self.stop_ctr >= 3:
            h = self.high_vol_ctr
            self.high_vol_ctr = 0
            self.stop_ctr = 0
            return self.high_vol_avg, h

        return -1, -1

    def _add(self, data):
        """
        add the data into buffer, and throw the first value into the decoder if len(buffer) > buffer_size
        """
```

```
        self.buffer.append(data)

        if len(self.buffer) > self.buffer_size:
            return self.buffer.pop(0)
        return -1
```

Appendix Ⅳ. iir_filter.py

```
import time
from iir_filter import IIR_filter
from scipy import signal

from Decoder import Decoder

class AnalogPrinter:

    def __init__(self, board, original_signal_plot, filtered_signal_plot, decoder: Decoder):
        # sampling rate: 1000Hz
        self.samplingRate = 1000
        self.timestamp = 0
        self.board = board

        # bind output plot
        self.original_signal_plot = original_signal_plot
        self.filtered_signal_plot = filtered_signal_plot

        # bind decoder
        self.decoder = decoder

        # cut off frequency: 50Hz
        cut_off = 50
        sos = signal.butter(2, cut_off/self.samplingRate*2, 'lowpass', output = 'sos')
        self.iir = IIR_filter(sos)

    def start(self):
        self.board.analog[0].register_callback(self.callback)
        self.board.samplingOn(1000 / self.samplingRate)
        self.board.analog[0].enable_reporting()

    def callback(self, data):
        now = time.time()
        filtered_data = self.iir.filter(data)

        self.original_signal_plot.addData(data, now)
        self.filtered_signal_plot.addData(filtered_data, now)
        self.decoder.decode(filtered_data)
```

Appendix Ⅴ. Encoder.py

```
import time

class Encoder:
    def __init__(self, red_pin: Pin, green_pin: Pin, blue_pin: Pin):
        self.red_pin = red_pin
        self.green_pin = green_pin
        self.blue_pin = blue_pin
        self.pulse_delay = 0.1
        self._signal_duration_map = {
            "1": 0.1,
            "0": 0.05
        }

    def rgb(self, red, green, blue):
        self.red_pin.write(red)
        self.green_pin.write(green)
        self.blue_pin.write(blue)

    def encode(self, data: str):
        """
        turn zero-one sequence to signals emited by LED light

        INIT and TERMINATE duration: 200ms
        VALUE ONE duration: 100ms
        VALUE ZERO duration: 50ms
        """
        # INIT
        self._send_signal(0.2)

        # encode the sequence to light signals
        for c in data:
            if c in self._signal_duration_map.keys():
                self._send_signal(self._signal_duration_map[c])

        # TERMINATE
        self._send_signal(0.2)

    def _send_signal(self, length):
        self.rgb(1, 1, 1)
        time.sleep(length)
        self.rgb(0, 0, 0)
        time.sleep(self.pulse_delay)
```

AppendixⅥ. realtime_iir_main.py

```
import sys
from pyfirmata2 import Arduino
from PyQt5.QtWidgets import QApplication
from threading import Thread


from AnalogPrinter import AnalogPrinter
from MainWindow import ControlWidget
from QtPanningPlot import QtPanningPlot
from Encoder import Encoder
from Decoder import Decoder


RED_PIN = 9
BLUE_PIN = 10
GREEN_PIN = 11

PORT = Arduino.AUTODETECT
board = Arduino(PORT)
app = QApplication(sys.argv)
# signals to all threads in endless loops that we'd like to run these
running = True

originalPlot = QtPanningPlot("Original Signal")
filteredPlot = QtPanningPlot("Filtered Signal")

# get pins of the LED light
red = board.get_pin(f"d:{RED_PIN}:p")
blue = board.get_pin(f"d:{BLUE_PIN}:p")
green = board.get_pin(f"d:{GREEN_PIN}:p")

encoder = Encoder(red, green, blue)
mainWindow = ControlWidget(board, encoder)
decoder = Decoder()
analogPrinter = AnalogPrinter(board, originalPlot, filteredPlot, decoder)

decoder.register_on_decode_end(mainWindow.output_to_logbox)

analogPrinter.start()
mainWindow.show()

app.exec_()

running = False
app.exit()
board.exit()
```

AppendixⅦ. Rununittest.py

```
import unittest
import numpy as np

from iir_filter import IIR_filter
from scipy import signal


class TestIIRFilter(unittest.TestCase):
    def __init__(self, methodName: str = ...) -> None:
        super().__init__(methodName=methodName)

        sr = 15     # sampling rate
        cutoff = 5  # cutoff frequency
        sos = signal.butter(2, cutoff/sr*2, 'lowpass', output = 'sos')
        self.filter = IIR_filter(sos)

    def test_filter(self):
        _in = np.loadtxt("assets/test_data.dat")
        expected_out = np.loadtxt("assets/test_out.dat")

        out = []
        for num in _in:
            out.append(self.filter.filter(num))

        self.assertEqual(len(out), len(expected_out))
        for i in range(len(out)):
            self.assertEqual(out[i], expected_out[i])


if __name__ == "__main__":
    unittest.main()
```

AppendixⅧ. QtPanningPlot.py

```
import pyqtgraph as pg
from pyqtgraph import QtGui, QtCore
import numpy as np

class QtPanningPlot:

    def __init__(self,title):
        self.win = pg.GraphicsLayoutWidget()
        self.title = title
        self.win.setWindowTitle(f"{title}")
        self.plt = self.win.addPlot()
        self.plt.setYRange(0,1)
        self.plt.setXRange(0,2000)
        self.curve = self.plt.plot()
        self.data = []
        # any additional initalisation code goes here (filters etc)
        self.timestamps = []
        self.sr_text = pg.LabelItem('0Hz', **{"size": "15"})
        self.win.addItem(self.sr_text)
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.update)
        self.timer.start(100)
        self.layout = QtGui.QGridLayout()
        self.win.setLayout(self.layout)
        self.win.show()

    def update(self):
        self.data=self.data[-2000:]
        if self.data:
            self.curve.setData(np.hstack(self.data))

        # calculate sampling rate and update to plot
        if len(self.timestamps) > 1:
            sr = (len(self.timestamps) - 1) / (self.timestamps[-1] - self.timestamps[0])
            self.sr_text.setText(f"{int(sr)}Hz")

    def addData(self, d, now):
        self.data.append(d)

        # add timestamps
        self.timestamps.append(now)
```

AppendixⅨ. MainWindow.py

```
import typing
from PyQt5.QtGui import QFont

from threading import Thread
from PyQt5.QtWidgets import QGridLayout, QLabel, QPlainTextEdit, QPushButton, QWidget, QApplication

from Encoder import Encoder


class ControlWidget(QWidget):
    def __init__(self, board, encoder: Encoder,
                 parent: typing.Optional['QWidget'] = None,) -> None:
        super().__init__(parent=parent)

        self.CreateItems()
        self.CreateLayout()
        self.CreateSignalSlot()

        self.setFont(QFont('Consolas', 12))

        self.board = board
        self.encoder = encoder

    def CreateItems(self):
        self.encode_label = QLabel("Encode String:")
        self.encode_label.setFixedWidth(200)
        self.encode_label.setFixedHeight(20)
        self.encode_input = QPlainTextEdit()
        self.encode_input.setFixedHeight(self.encode_label.height())
        self.encode_start_btn = QPushButton("Start")
        self.encode_start_btn.setFixedHeight(self.encode_label.height())

        self.decode_label = QLabel("Decode result:")
        self.decode_label.setFixedWidth(200)
        self.decode_label.setFixedHeight(20)
        self.decode_input = QPlainTextEdit()
        self.decode_input.setFixedHeight(self.decode_label.height())
        # self.decode_input.setReadOnly(True)

        self.log_box = QPlainTextEdit()
        self.log_box.setFixedHeight(100)
        self.log_box.setMaximumBlockCount(60000)

    def CreateLayout(self):
        self.mainLayout = QGridLayout()
```

```
46.         self.mainLayout.addWidget(self.encode_label, 0, 0)
47.         self.mainLayout.addWidget(self.encode_input, 0, 1)
48.         self.mainLayout.addWidget(self.encode_start_btn, 0, 2)
49.         # self.mainLayout.addWidget(self.decode_label, 1, 0)
50.         # self.mainLayout.addWidget(self.decode_input, 1, 1)
51.         self.mainLayout.addWidget(self.log_box, 2, 0, 1, 3)
52.         self.mainLayout.setSpacing(5)
53.         self.setLayout(self.mainLayout)
54. 
55.     def CreateSignalSlot(self):
56.         self.encode_start_btn.clicked.connect(self._on_start_btn_clicked)
57. 
58.     def _on_start_btn_clicked(self):
59.         info = self.encode_input.toPlainText()
60.         if not info.strip(): return
61.         t = Thread(target=self.encoder.encode, args=(info,))
62.         t.start()
63. 
64.     def output_to_logbox(self, data):
65.         self.log_box.insertPlainText(f"{data}\n")
66. 
67.     def update_result(self, data):
68.         self.decode_input.clear()
69.         self.decode_input.insertPlainText(f"{data}")
70.         self.decode_input.update()
```