GUI documentation

Elliot Wadge

June 2021

Introduction

This document is meant to give the reader an introduction to GUI programming in python as well as an insight into the inner workings of some of the key functions used in this program.There are two major sides to this code, the parts concerned with displaying information to the viewer which will be briefly summarized in this document and the logic for controlling the spectrometer. The program consists of many files and we will approach them for the most part individually. While I do provide explanations for the key functions it is important that you visit the github links and look at the code in its entirety if you want to really understand how it works. This work was structured to resemble the pre-existing GUI made by Matt Frick using LabVIEW.

Contents

1 Using the Program

The program is operated using the GUI, on the left side are the various controls for activating certain functions and on the right are the plots used to visualize the data in real time. There is also a search function that can be used to search and then plot the samples which utilizes a slightly modified version of the program written by Colton Lohn.

To re-calibrate the spectrometer enter a valid floating point value into the "Actual Value" box in the "Re-calibrate Spectrometer" section then press re-calibrate. This will update the position that the computer is storing as the current position of the spectrometer, it is fast and simple no physical change to the spectrometer occurs.

To move the spectrometer enter a valid floating point value into the "Go to Value" box and press move, this will change the current position of the spectrometer by sending pulses to the stepper motor.

To run a scan the spectrometer should be properly calibrated then you enter a start wavelength an end wavelength, a step size, a count time, file name, and optionally a sample ID then press Scan. The repeat button can also be pressed to repeat the scan forever. It should be noted that the repeat button is essentially just repressing the scan button, this means that changes to the inputs will effect the next scan, I decided to leave this as a feature, however if one wanted to change this it would not be hard. The scan is really just many moves followed by data collection in a row. It will move the step size then collect data for the count time then move again until it reaches the end point. The data is saved under C:/Users/Admin/Documents/PL/Data/YYYY MM DD/File Name.txt.

To utilize the Optimize function simply press go it will continuously generate beeps of 0.15s interval until the stop or abort button is pressed. The frequency of the beeps rises with the number of counts that the PMT is detecting (400 - 800Hz). It will also display these counts on a bar graph in real time for reference. Since the range of counts can be so vast the scale on which the frequency is calculated is re-scaled every factor of ten in increase or decrease. The result of this rescaling will be a drastic drop in frequency if it rescaled by moving up a factor of 10 or a drastic increase in frequency if it is rescaled down by a factor of ten.

To switch between spectrometers being controlled simply click on the double or single radio button depending on which one you would like to control. the single spectrometer does not support optimizing since there is no PMT on it, clicking the optimize button while single is selected will just call the optimize function of the double spectrometer. The position in the top left will reflect the current position of the selected spectrometer, to view the position of the other spectrometer just switch to it using the radiobutton.

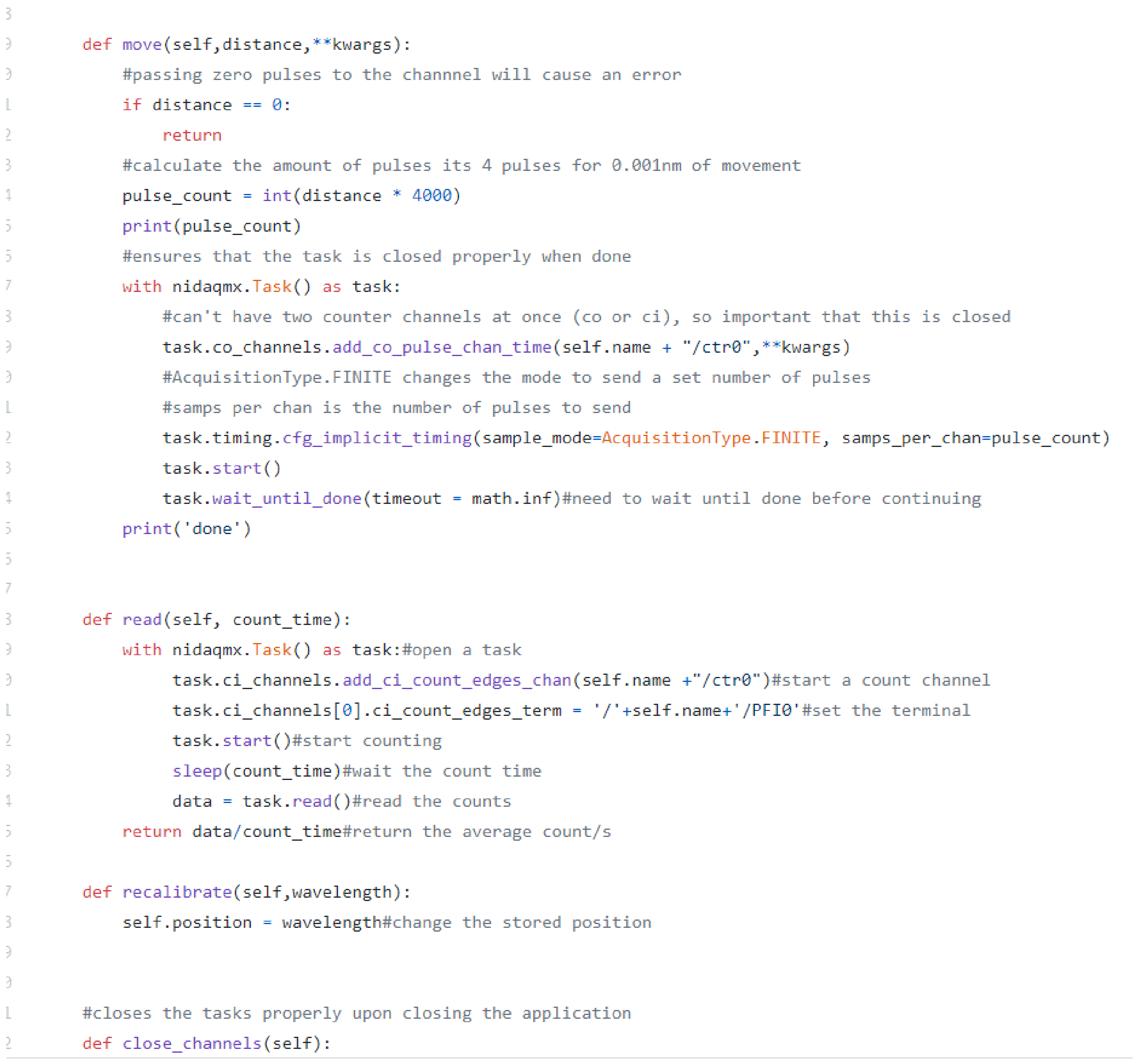
2 Code Overview

The program utilizes the PyQt library to do the vast majority of the heavy lifting. In order to maintain responsiveness of our program it is necessary to thread the long running tasks such as move, scan, and optimize. There is no need to thread the recalibrate button since it happens so quickly. How the code has been laid out is as follows; the main file contains all GUI widgets and functions for altering appearance with the exception of the graphs which have their own seperate files. In seperate files are the different main functions like scan, move and optimize, as well as the spectrometer class. The spectrometer class tracks the position of the spectrometer and does the actual signal sending and receiving. These are imported into the main file and utilized to achieve control of the spectrometer. The program relies heavily on the use of classes and if you are unfamiliar with object oriented programming in python or classes I encourage you to look at the links in the learning resources.

3 Code Specifics

3.1 Spectrometer

Link to github: https://github.com/Elliot-Wadge/Python-GUI/blob/main/spectrometer.py. The code is a bit too long for a picture to be useful but I will specify line numbers changes may be made so they should only be used as a rough number I will also reference what the line says. So first in line 13 there is the "__init__" method this is run everytime an instance of this class is created. It takes a device name a shutter_prt and a direction_prt. The device names specifies the name that the computer has assigned to the daq that is controlling the spectrometer, currently the device controlling double is named 'Dev2' and the one controlling single is named 'Dev3' the ports are the ports that are connected to the shutter and direction pins. In the __init__ the program attempts to read the last position from a file named after the device name if the file has been deleted then it sets the value to zero. It then tries to open the two ports specified as digital outputs using the nidaqmx library. The try and except are simply to make testing easier. Starting at line 61 you can see the open_shutter, close_shutter, and set_direction methods. Since they are digital outs they just write True or False, True corresponding to five volts and False corresponding to zero. Below this is the save method which simply saves the current position of the spectrometer to a file. Next comes the most important functions, which I will include photos of. My documentation for these isn't terrible so I will leave the images to do most of the talking.



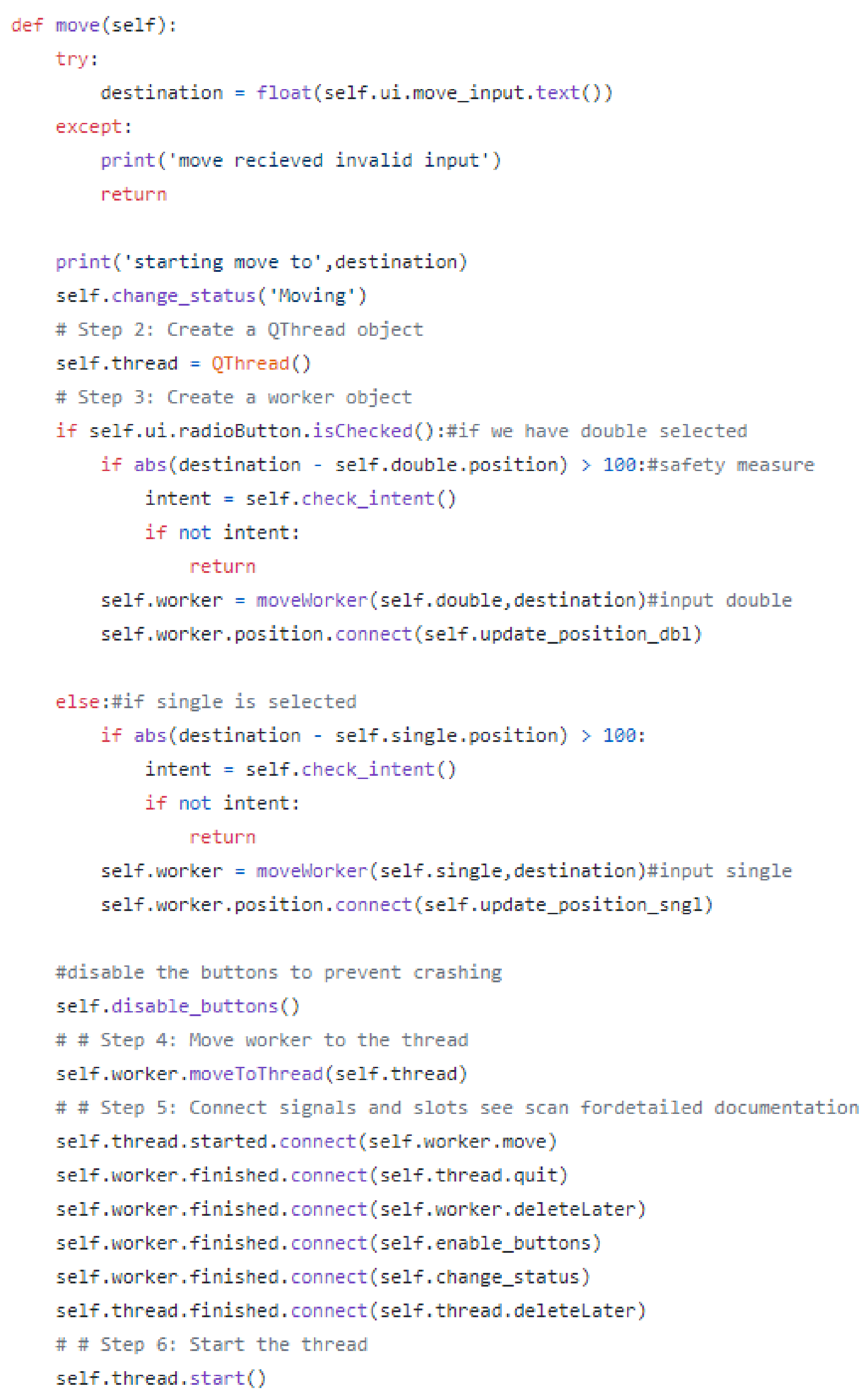
```
    def move(self,distance,**kwargs):
        #passing zero pulses to the channnel will cause an error
        if distance == 0:
            return
        #calculate the amount of pulses its 4 pulses for 0.001nm of movement
        pulse_count = int(distance * 4000)
        print(pulse_count)
        #ensures that the task is closed properly when done
        with nidaqmx.Task() as task:
            #can't have two counter channels at once (co or ci), so important that this is closed
            task.co_channels.add_co_pulse_chan_time(self.name + "/ctr0",**kwargs)
            #AcquisitionType.FINITE changes the mode to send a set number of pulses
            #samps per chan is the number of pulses to send
            task.timing.cfg_implicit_timing(sample_mode=AcquisitionType.FINITE, samps_per_chan=pulse_count)
            task.start()
            task.wait_until_done(timeout = math.inf)#need to wait until done before continuing
        print('done')


    def read(self, count_time):
        with nidaqmx.Task() as task:#open a task
             task.ci_channels.add_ci_count_edges_chan(self.name +"/ctr0")#start a count channel
             task.ci_channels[0].ci_count_edges_term = '/'+self.name+'/PFI0'#set the terminal
             task.start()#start counting
             sleep(count_time)#wait the count time
             data = task.read()#read the counts
        return data/count_time#return the average count/s

    def recalibrate(self,wavelength):
        self.position = wavelength#change the stored position


    #closes the tasks properly upon closing the application
    def close_channels(self):
```

Figure 1: Some things to note, the move function only takes a distance not direction it will just move the distance in the current direction. This is NOT the full move routine that is called when clicking the move button we will get to that next.

The single and double spectrometer are actually two different classes but they are very similar. The single is exactly the same but with different default channels and it doesn't have a shutter function.

3.2 Movement

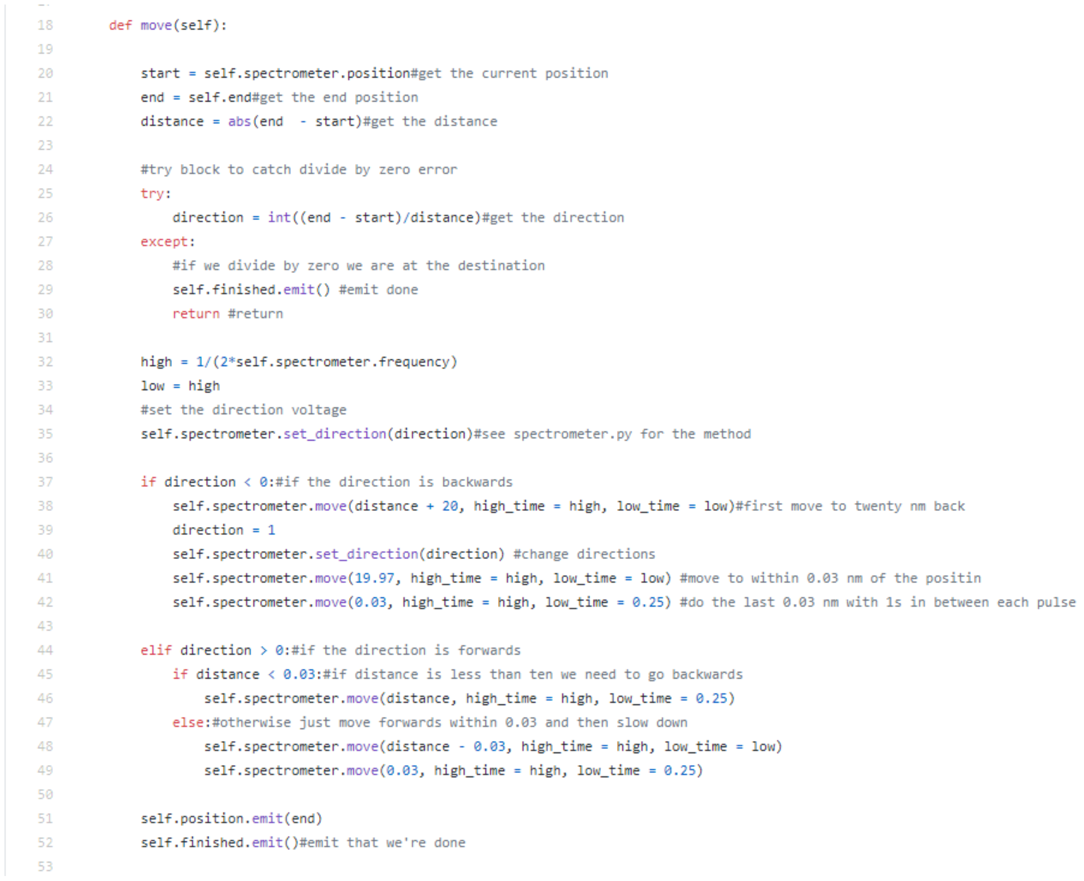
Link to github: https://github.com/Elliot-Wadge/Python-GUI/blob/main/workers/move.py. There are two main parts to the move function, the setting up of the thread to execute the task and the actual execution. First we'll look at setting up the thread, an example on why threading is necessary can be seen in the learning resources section of this document.



```python
def move(self):
    try:
        destination = float(self.ui.move_input.text())
    except:
        print('move recieved invalid input')
        return

    print('starting move to',destination)
    self.change_status('Moving')
    # Step 2: Create a QThread object
    self.thread = QThread()
    # Step 3: Create a worker object
    if self.ui.radioButton.isChecked():#if we have double selected
        if abs(destination - self.double.position) > 100:#safety measure
            intent = self.check_intent()
            if not intent:
                return
        self.worker = moveWorker(self.double,destination)#input double
        self.worker.position.connect(self.update_position_dbl)

    else:#if single is selected
        if abs(destination - self.single.position) > 100:
            intent = self.check_intent()
            if not intent:
                return
        self.worker = moveWorker(self.single,destination)#input single
        self.worker.position.connect(self.update_position_sngl)

    #disable the buttons to prevent crashing
    self.disable_buttons()
    # # Step 4: Move worker to the thread
    self.worker.moveToThread(self.thread)
    # # Step 5: Connect signals and slots see scan fordetailed documentation
    self.thread.started.connect(self.worker.move)
    self.worker.finished.connect(self.thread.quit)
    self.worker.finished.connect(self.worker.deleteLater)
    self.worker.finished.connect(self.enable_buttons)
    self.worker.finished.connect(self.change_status)
    self.thread.finished.connect(self.thread.deleteLater)
    # # Step 6: Start the thread
    self.thread.start()
```

Figure 2: setting up the thread for the move function

in step 2 we create the thread, in step 3 if the move is over 100nm we check that it was not a mistake then make a worker with the selected spectrometer, the selected spectrometer is passed as an argument to be used by the worker, in step 4 we move the worker to the thread which prepares it to be launched, in step 5 we connect the signals to the proper functions, these signals will be emitted from the thread to our main window causing updates to the visuals and to our spectrometer position, in step 6 we begin executing the function that is connected to thread.started which in this case is the workers ".move" method. We will look at the workers ".move" method next. Here we utilize the



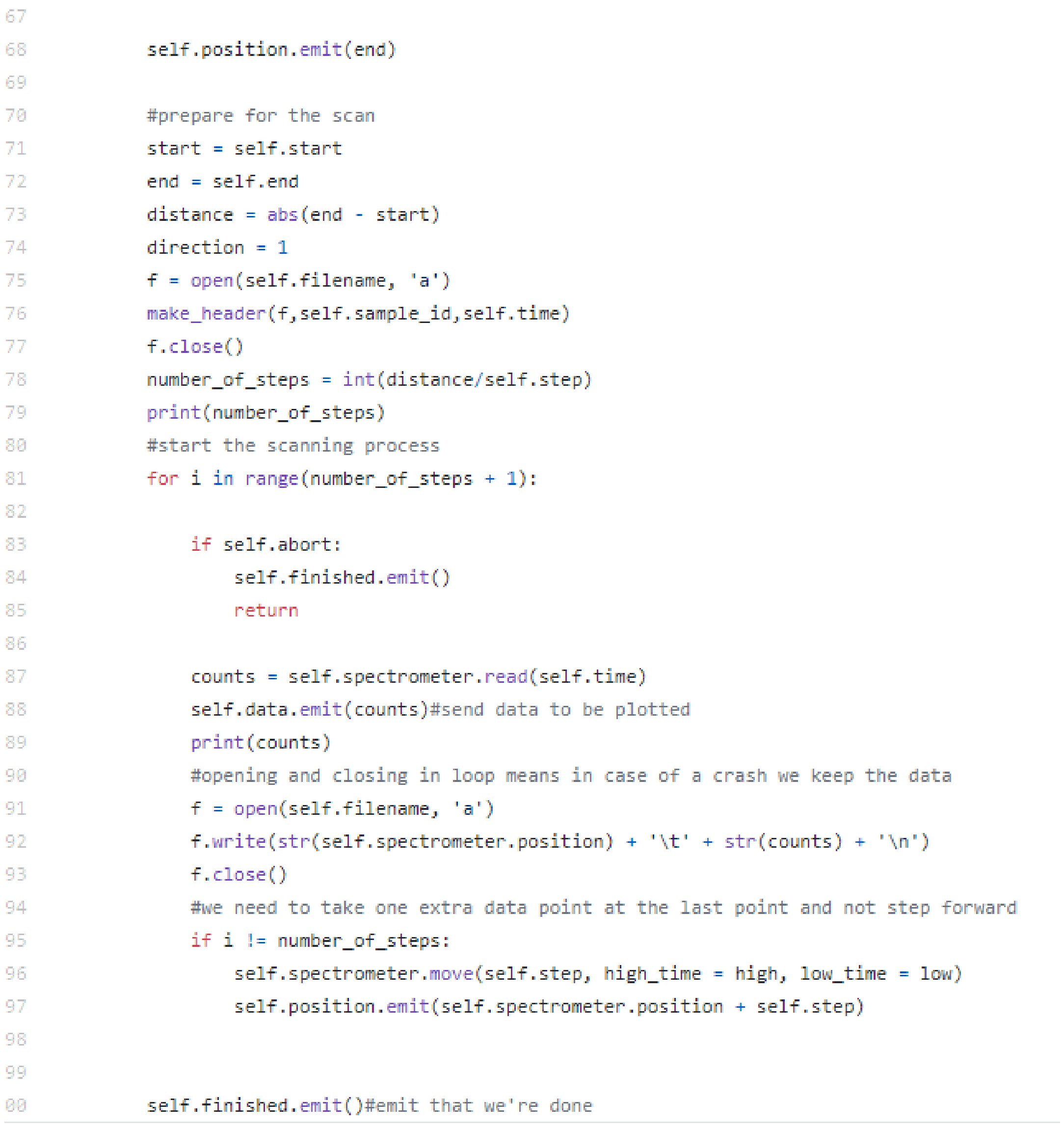
```python
    def move(self):

        start = self.spectrometer.position#get the current position
        end = self.end#get the end position
        distance = abs(end  - start)#get the distance

        #try block to catch divide by zero error
        try:
            direction = int((end - start)/distance)#get the direction
        except:
            #if we divide by zero we are at the destination
            self.finished.emit() #emit done
            return #return

        high = 1/(2*self.spectrometer.frequency)
        low = high
        #set the direction voltage
        self.spectrometer.set_direction(direction)#see spectrometer.py for the method

        if direction < 0:#if the direction is backwards
            self.spectrometer.move(distance + 20, high_time = high, low_time = low)#first move to twenty nm back
            direction = 1
            self.spectrometer.set_direction(direction) #change directions
            self.spectrometer.move(19.97, high_time = high, low_time = low) #move to within 0.03 nm of the positin
            self.spectrometer.move(0.03, high_time = high, low_time = 0.25) #do the last 0.03 nm with 1s in between each pulse

        elif direction > 0:#if the direction is forwards
            if distance < 0.03:#if distance is less than ten we need to go backwards
                self.spectrometer.move(distance, high_time = high, low_time = 0.25)
            else:#otherwise just move forwards within 0.03 and then slow down
                self.spectrometer.move(distance - 0.03, high_time = high, low_time = low)
                self.spectrometer.move(0.03, high_time = high, low_time = 0.25)

        self.position.emit(end)
        self.finished.emit()#emit that we're done

```

Figure 3: the complete move routine

previously mentioned move and set direction methods in the spectrometer Fg.1 to execute the routine properly, if moving backwards we need to remove the kickback so we go an additional 20nm backwards before moving forwards again. We also want to avoid overshoot so the last 0.03nm of the move are slowed down to move only 0.001nm per second.

3.3 Scanning

Link to github: https://github.com/Elliot-Wadge/Python-GUI/blob/main/workers/scan.py. Like the move function the scan function must also be setup on a thread in a very similar way before it can be executed. It takes a few more arguments but is ultimately the same so see Fg.2 or visit the github to see it specifically. To perform a scan we start by repeating the move function 3 to get into the starting position and then executing the routine below.



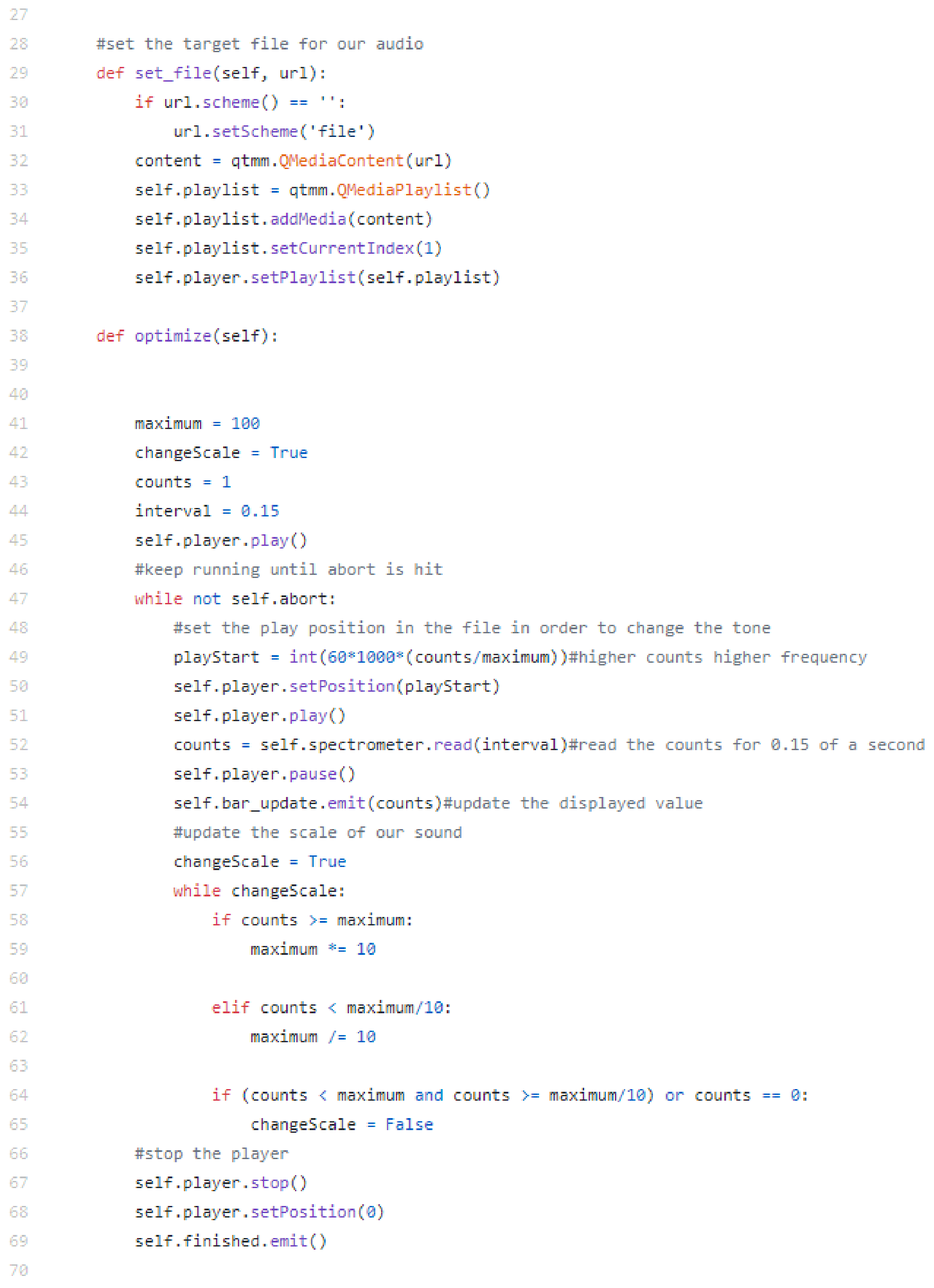
```

        self.position.emit(end)

        #prepare for the scan
        start = self.start
        end = self.end
        distance = abs(end - start)
        direction = 1
        f = open(self.filename, 'a')
        make_header(f,self.sample_id,self.time)
        f.close()
        number_of_steps = int(distance/self.step)
        print(number_of_steps)
        #start the scanning process
        for i in range(number_of_steps + 1):

            if self.abort:
                self.finished.emit()
                return

            counts = self.spectrometer.read(self.time)
            self.data.emit(counts)#send data to be plotted
            print(counts)
            #opening and closing in loop means in case of a crash we keep the data
            f = open(self.filename, 'a')
            f.write(str(self.spectrometer.position) + '\t' + str(counts) + '\n')
            f.close()
            #we need to take one extra data point at the last point and not step forward
            if i != number_of_steps:
                self.spectrometer.move(self.step, high_time = high, low_time = low)
                self.position.emit(self.spectrometer.position + self.step)


        self.finished.emit()#emit that we're done
```

Figure 4: the complete scan routine

the execution is actually pretty simple we just run a for loop for the total number of steps, read the counts for a set amount of time using the read method from Fg.1 append that to a file and then emit that data back to the UI to be displayed in the graph. One of the trickiest parts of this program was getting the graphing to work, adding additional graphs however is trivial due to the object oriented nature of python and hopefully that isn't something anyone else will have to worry about too much.

3.4 Optimizing

Link to github: https://github.com/Elliot-Wadge/Python-GUI/blob/main/workers/optimize.py. Again this function needs to be threaded for responsiveness reasons see Fg.2 for a similar example or visit https://github.com/Elliot-Wadge/Python-GUI/blob/main/GUI.py line 273 to see how this one is set up specifically. The optimize function is as follows.



```

    #set the target file for our audio
    def set_file(self, url):
        if url.scheme() == '':
            url.setScheme('file')
        content = qtmm.QMediaContent(url)
        self.playlist = qtmm.QMediaPlaylist()
        self.playlist.addMedia(content)
        self.playlist.setCurrentIndex(1)
        self.player.setPlaylist(self.playlist)

    def optimize(self):


        maximum = 100
        changeScale = True
        counts = 1
        interval = 0.15
        self.player.play()
        #keep running until abort is hit
        while not self.abort:
            #set the play position in the file in order to change the tone
            playStart = int(60*1000*(counts/maximum))#higher counts higher frequency
            self.player.setPosition(playStart)
            self.player.play()
            counts = self.spectrometer.read(interval)#read the counts for 0.15 of a second
            self.player.pause()
            self.bar_update.emit(counts)#update the displayed value
            #update the scale of our sound
            changeScale = True
            while changeScale:
                if counts >= maximum:
                    maximum *= 10

                elif counts < maximum/10:
                    maximum /= 10

                if (counts < maximum and counts >= maximum/10) or counts == 0:
                    changeScale = False
        #stop the player
        self.player.stop()
        self.player.setPosition(0)
        self.finished.emit()

```

Figure 5: the complete optimize routine

there are some things going on behind the scenes here. The optimize function works by having a .wav file that is 400Hz tone linearly going up to a 800Hz tone. we then read the amount of counts and compare that to the current maximum value in line 49 which is the next largest multiple of ten and then set the position of the player to a place in the sound file that is proportional to that ratio. Thus the closer the counts are to the maximum the higher the pitch. Since the counts can cover such a massive range we have to rescale the maximum in order for us to hear meaningful differences in the counts this is the while loops function in line 57. If the counts are too big or too small it rescales the maximum this will cause a sharp increase or decrease in the tone.

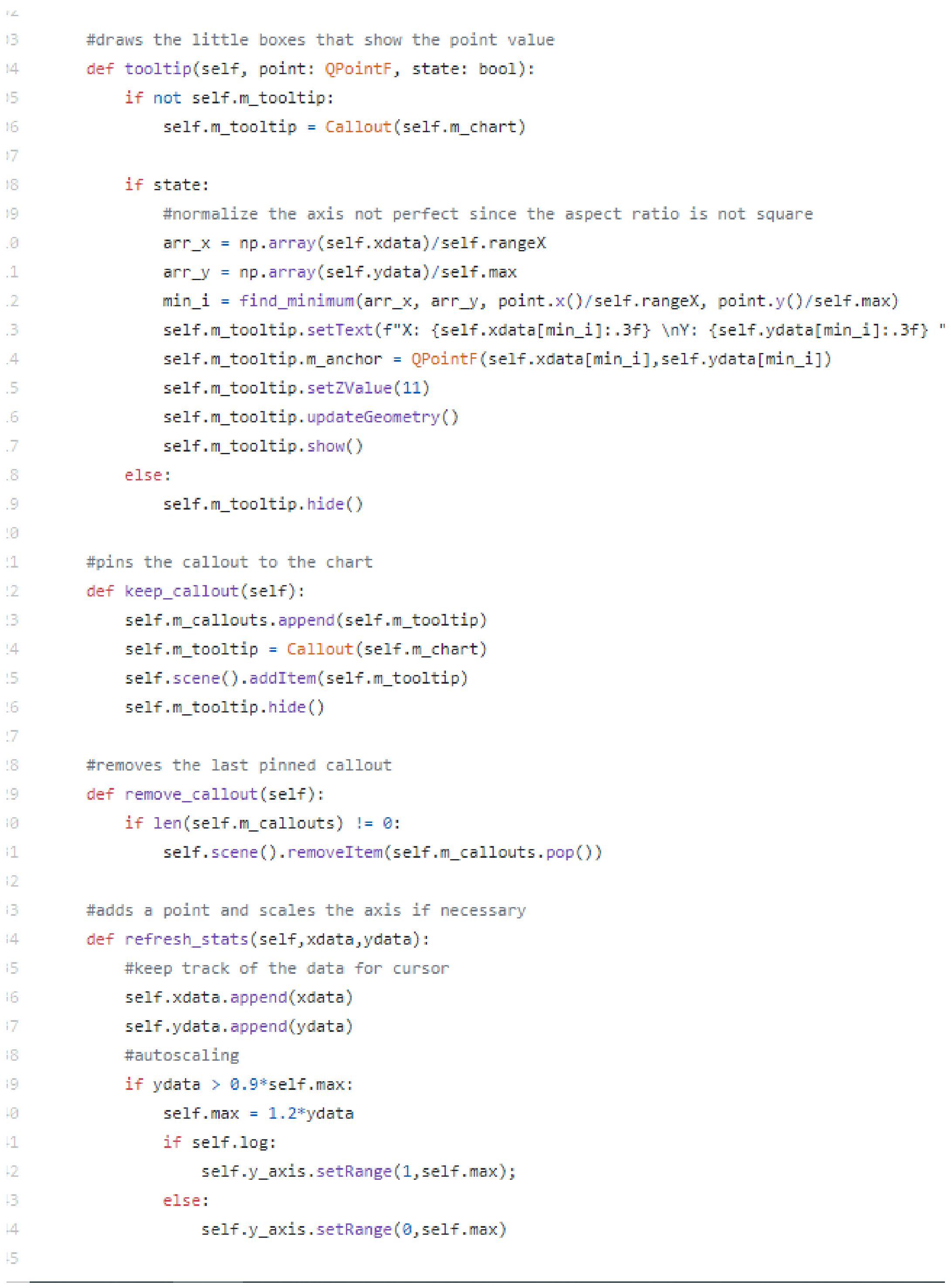
3.5 GUI

The making of the GUI with one exception was done entirely using QtDesigner which is a visual tool that allows you to place widgets onto a screen and then compile that screen into usable python code. It makes developing the visuals of a UI more beginner friendly but unfortunately the compiled code is a bit messy and hard to look at. In this case the code is compiled into a file named "qt_designer.py" which can be seen here https://github.com/Elliot-Wadge/Python-GUI/blob/main/qt_designer.py. However the anyone looking to work on this program should never edit this file directly if a change to the appearance is desired the proper procedure is as follows. Step one download and locate QtDesigner for use, the finding part can be more of a pain then you might think my qtdesigner was located in "C:/Users/ewadge/AppData/Local/Packages/PythonSoftwareFoundation.Python.3.9_qbz5n2kfra8p0/LocalCache/local-packages/Python39/site-packages/QtDesigner" and I installed it with "pip install PyQt5Designer" once you've done that make a shortcut to it on the desktop so you don't have to find it again. Step two open the spectrometer.ui (https://github.com/Elliot-Wadge/Python-GUI/blob/main/qtdesigner_files/spectrometer.ui) file with QtDesigner and make the changes you want using the visual tools and save. Step 3 you'll have to install pyuic5 I don't remember how I did this but I believe "pip install pyuic5-tool" should work, then type the command "pyuic5 -x spectrometer.ui -o qt_designer.py" in the working directory and your changes will be compiled to the proper .py file. If you get the error message "pyuic5 is not a recognized command..." then you can try "python -m pyuic5 -x spectrometer.ui -o qt_designer.py", this issue stems from the fact that it is incredibly challenging to add things to the path on the computers we have here and you can always use another computer if it's not working.

The one exception to things being made in QtDesigner is the graphs and tab section of the GUI. This is created by hand in the GUI.py file (https://github.com/Elliot-Wadge/Python-GUI/blob/main/GUI.py) line 78 and called in line 31. I will leave the reader to familiarize themselves with how to make widgets by hand as it is far to broad a topic to cover here the key concepts to making your own widgets are inheritance and the 'super()' function, I have provided some links in the learning resources section. This was done because the graphs and search part are custom widgets that I was unsure of how to include into QtDesigner. The rest of the GUI file is dedicated to connecting buttons to the right logic and updating the visuals as required. It also is responsible for setting up the threads and is the file that should be run when you want to use the GUI. Including a photo of it wouldn't be helpdul as it is too large. For a simpler introduction into this aspect of PyQt see (https://github.com/Elliot-Wadge/Python-GUI/blob/main/searchUI.py) running this will produce an independent search GUI that uses many of the same tools as the larger program, while remaining much cleaner and simpler. As a side note this might be your first time seeing "If __name__ == '__main__:"' what this does is allow us to have tests and demonstrations in our files that won't be run when we import the code into a different file. In this case not having this line would result in searchUI window opening everytime the GUI.py file was run.

3.6 Graphing

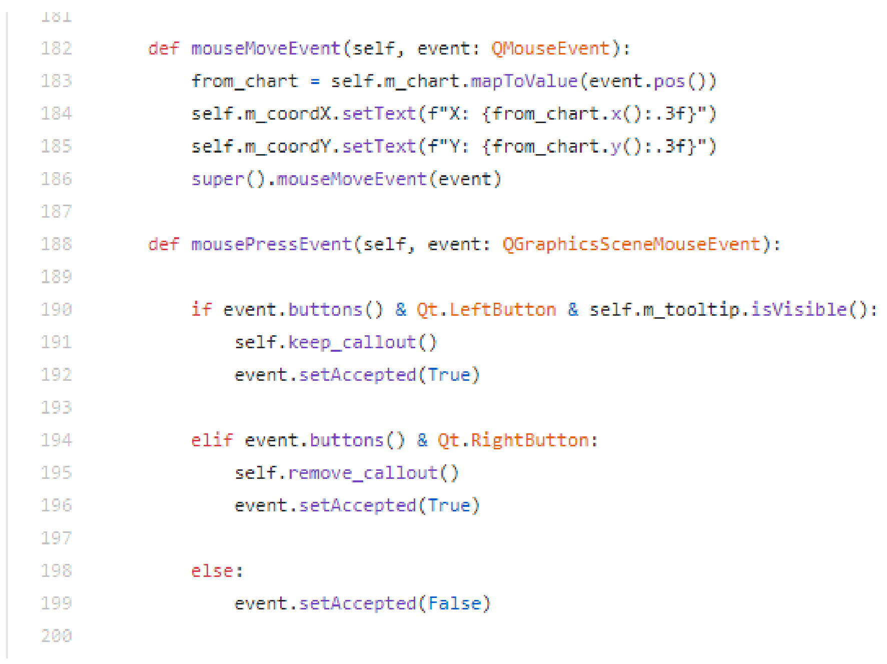
Link to github: https://github.com/Elliot-Wadge/Python-GUI/blob/main/graphing.py. A large amount of code for this section comes from this stack overflow post https://stackoverflow.com/questions/60198436/python-pyqt5-version-of-qt-callout-example so I can't say that I completely understand how it works but I made some modifications that fix some bugs and change the behavior a bit. The most important modification was making the callouts lock on to the nearest data point. The graphs are made using the QChart package which can be installed using "pip install PyQtChart". There are two different kinds of graphs, the bar chart used for displaying the optimize counts and the line charts that plot the counts vs wavelength as a scan progresses. The bar chart is really just formatting that can be learned from visiting the documentation at https://pyqtgraph.readthedocs.io/en/latest/. As I mentioned before the line chart is a bit complicated, what makes it complicated is the hovering feature and left click to keep right click to delete. The callout class at the top of the file is the little text box that comes up when hovering over the line, this is what I understand the least but luckily I don't really have to. The View class is the actual chart and the part that I modified a bit.



```
    #draws the little boxes that show the point value
    def tooltip(self, point: QPointF, state: bool):
        if not self.m_tooltip:
            self.m_tooltip = Callout(self.m_chart)

        if state:
            #normalize the axis not perfect since the aspect ratio is not square
            arr_x = np.array(self.xdata)/self.rangeX
            arr_y = np.array(self.ydata)/self.max
            min_i = find_minimum(arr_x, arr_y, point.x()/self.rangeX, point.y()/self.max)
            self.m_tooltip.setText(f"X: {self.xdata[min_i]:.3f} \nY: {self.ydata[min_i]:.3f} "
            self.m_tooltip.m_anchor = QPointF(self.xdata[min_i],self.ydata[min_i])
            self.m_tooltip.setZValue(11)
            self.m_tooltip.updateGeometry()
            self.m_tooltip.show()
        else:
            self.m_tooltip.hide()

    #pins the callout to the chart
    def keep_callout(self):
        self.m_callouts.append(self.m_tooltip)
        self.m_tooltip = Callout(self.m_chart)
        self.scene().addItem(self.m_tooltip)
        self.m_tooltip.hide()

    #removes the last pinned callout
    def remove_callout(self):
        if len(self.m_callouts) != 0:
            self.scene().removeItem(self.m_callouts.pop())

    #adds a point and scales the axis if necessary
    def refresh_stats(self,xdata,ydata):
        #keep track of the data for cursor
        self.xdata.append(xdata)
        self.ydata.append(ydata)
        #autoscaling
        if ydata > 0.9*self.max:
            self.max = 1.2*ydata
            if self.log:
                self.y_axis.setRange(1,self.max);
            else:
                self.y_axis.setRange(0,self.max)
```

Figure 6: the pinning and unpinning methods

the tooltip is the box that appears and it is connected to the hover function of the line, if the line is not being hovered over it is hidden. when the left mouse button is clicked while hovering over the line the keep_callout method is called which permanently pins the callout to the scene, when the chart is hovered over and right clicked the remove_callout method is called and the last placed call out is removed from the scene. This is set up with the following code.



```
    def mouseMoveEvent(self, event: QMouseEvent):
        from_chart = self.m_chart.mapToValue(event.pos())
        self.m_coordX.setText(f"X: {from_chart.x():.3f}")
        self.m_coordY.setText(f"Y: {from_chart.y():.3f}")
        super().mouseMoveEvent(event)

    def mousePressEvent(self, event: QGraphicsSceneMouseEvent):

        if event.buttons() & Qt.LeftButton & self.m_tooltip.isVisible():
            self.keep_callout()
            event.setAccepted(True)

        elif event.buttons() & Qt.RightButton:
            self.remove_callout()
            event.setAccepted(True)

        else:
            event.setAccepted(False)

```

Figure 7: the mouse events

since view inherits from QChartView we can overwrite its mouse events to do what we want, here when the mouse is moving it shows the position of the cursor in the coordinate system on the bottom of the graph and then calls the super().mouseMoveEvent(event) which just calls the parent functions mousemove, the mousePressEvent is what is handling the pinning and unpinning of callouts. You can see if the the left button is clicked and the tooltip is visible (meaning its hovering over a line) we keep it, if we right click anywhere on the chart we call remove the callout.

4 Dependencies

the following packages are required to run the program

- PyQt5: pip install PyQt5
- numpy: pip install numpy
- nidaqmx: pip install nidaqmx
- QChart: pip install PyQtChart
- Plotly: pip install plotly

5 Conclusion

This document attempts to provide some guidance as to how the current GUI functions and is meant to be used, as well as the provided links for the source code and other learning resources. The benefits of moving to python are many but unfortunately it also comes with its own set of drawbacks. The program is far lighter in terms of how much memory is required and we no longer have to install the massive amount of software that comes with LabVIEW. We also get to move away from the visual language of labview and python is completely free. However Python isn't specifically tailored for this sort of thing unlike LabVIEW so we have to do a lot of things ourselves that LabVIEW does for us such as making real time plots with cursors, or checking if a file exists and making the proper header, this leads to a lot of extra code. The level of python knowledge to work on this code is also intermediate something that a lot of undergraduate students in physics at SFU don't posses so while it may be easier than LabVIEW to learn and debug it is still a challenge. If you do have to work on and improve this code I do apologize for my imperfect and sometimes confusing work (three different move functions? don't ask).

6 Learning Resources

- introduction to classes https://www.w3schools.com/python/python_classes.asp
- more in depth classes https://realpython.com/python3-object-oriented-programming/
- introduction to pyqt (first three videos are a good start) https://www.youtube.com/watch?v=Vde5SH8e1OQ&list=PLzMcB
lB8MZfHPLTEHO9zJDDLpYj
- why you shouldn't edit the qt_designer.py file https://www.youtube.com/watch?v=XXPNpdaK9WA
- @property https://www.programiz.com/python-programming/property
- how to interface with the daq (simple) https://www.youtube.com/watch?v=umXMrr6Z0Og&t=589s
- the github for nidaqmx https://github.com/ni/nidaqmx-python
- Threading https://realpython.com/python-pyqt-qthread/
- book on pyqt https://www.amazon.ca/Mastering-GUI-Programming-Python-cross-platform/dp/178961290X/ref=asc_df_17
20&linkCode=df0&hvadid=80608037717144&hvnetw=o&hvqmt=e&hvbmt=be&hvdev=c&hvlocint=&hvlocphy=&hvtargi
4584207582640366&psc=1
- the github for this project https://github.com/Elliot-Wadge/Python-GUI