NTNU

Norwegian University of
Science and Technology

TKP4555 Advanced Simulation

Developing Graphical User Interfaces in Python using PyQt for Laboratory Use

Brittany Hall

November 30, 2017

Contents

List of Figures

Chapter 1

Introduction

Software development is a huge industry in the western world and is estimated to be worth over US $407.3 billion [9]. One sub-sector of the software industry is the development of graphical user interfaces (GUIs). A GUI allows users to interact with electronic devices through graphical icons and visual indicators instead of using the command line. GUIs were introduced to make computers more user friendly since the command-line interface has a large learning curve. Users perform actions in a GUI by direct manipulation of graphically elements. GUIs can be developed for many different capabilities. Commonly used operating systems utilize GUIs: Windows, macOS, Ubuntu, etc [4]. In this report, we focus on the use of GUIs to control experimental setups in a laboratory.

Constructing GUIs from scratch requires a large knowledge of programming and significant time. This is why some software has been developed to help users create GUIs faster and require less programming knowledge. One such software is LabVIEW; LabVIEW is a relatively well known commercially available software for applications that require test, measurement, and control with rapid access to hardware and data [5]. It advertises itself as a software that helps simplify hardware integration and reduces the complexity of programming required to create a user interface. While this wide range of capabilities is attractive, LabVIEW requires a yearly license to be purchased with prices ranging from 3200 NOK/year to 53640 NOK/year. Thus, finding an open source alternative is attractive to universities and companies.

PyQt, combined with a communication framework, is one such open source alternative; it is a Python compatible version of Qt, which is a cross platform software development kit owned by Nokia that can be used on various software and hardware platforms with little or no change to the underlying codebase [8]. PyQt is distributed under the General Public License (GPL) meaning that we can use the free version of PyQt as long as we don't sell our code; if we want to sell our product, we must purchase a commercial license of PyQt. It has the benefit of being run as a native application; thus, it has the same capabilities and speed as other native applications. PyQt is mainly used for developing multi-platform applications and GUIs, which is why a separate communication framework is required to make it competitive to LabVIEW. There are open source communication frameworks available

in Python so this is not a limiting issue; one example is the Python module `minimalmodbus` which always for communication with instruments from a computer using the Modbus protocol.

PyQt capabilities are discussed within the framework of a GUI for a specific laboratory experiment, known as the Two Tanks experiment, in this report. It is worth noting that PyQt has many more capabilities that were not used to create this particular GUI and thus not discussed in this report. A description of the experiment is given in Chapter 2. The main PyQt modules utilized for the creation of this GUI are explained in general in Chapter 3. How each component was created for the GUI is discussed in detail in Chapter 5.

Chapter 2

Experiment Description

2.1 General Description

A GUI was created for the Two Tank experiment in the Process Control Laboratory and is illustrated in Figure 2.1. The setup consists of two tanks, each of which has an inlet and an outlet stream. A thermocouple measures the temperature inside each tank. The inlet stream consists of room temperature water. On each outlet stream is a solenoid valve; this valve type means that the valve is either completely open or closed. A controller is attached to each solenoid valve.

The two outlet streams connect, after the valves, allowing the outlet streams to mix. A thermocouple is located downstream of the mixing point and measures the temperature of the mixed stream.

A single usb serial analog to digital converter is used for communication. This allows a computer to receive measurements from the thermocouple, send/receive signals to the solenoid valves to open or close, and send/receive signals to the controllers. The purpose of this experiment is to tune the controllers to get the temperature of the mixed stream to a desired set point.

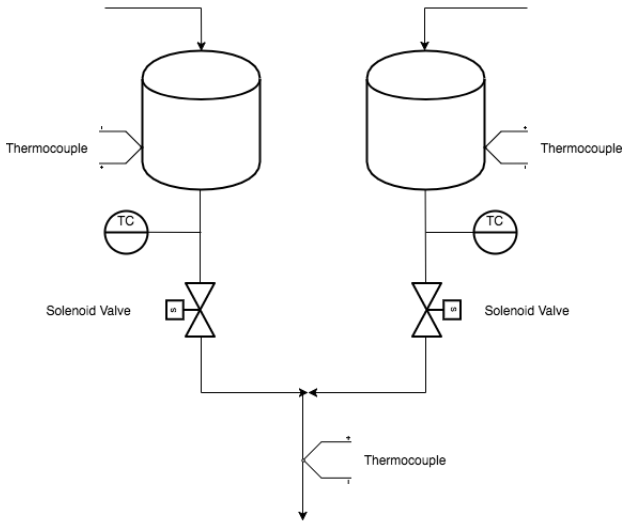

Figure 2.1: Illustration of the two tank experiment

Chapter 3

PyQt Modules

3.1 Introduction

PyQt allows programers to use much of the functionality of Qt in Python; this includes a comprehensive set of widgets, flexible layout managers, standard GUI features for applications, easy communication between application components, threading classes, widget styles, input/output and networking, support for QtDesigner, etc [6].

PyQt4 has a number of Python extension modules that makes it convenient to program GUIs. For the Two Tank experiment, we utilized the two following main modules: `QtCore` and `QtGui`. The `QtCore` module contains the core of non-GUI classes, including event loop and Qt's signal and slot mechanism [6]. The `QtGui` module contains the majority of the GUI classes.

PyQt4 also has a couple of utility programs that are useful:

- `pyuic4` corresponds to the Qt uic utility that converts GUIs created using Qt Designer to Python code.
- `pyrcc4` corresponds to the Qt rcc utility that embeds resources described by a resource collection file in a Python module. This is only included if Qt includes a XML module.
- `pylupdate4` corresponds to the Qt lupdate utility and it extracts all of the translatable strings from Python code and creates or updates translate files. This is only included if Qt includes the XML module.

The `pyuic4` can be used to translate GUIs created in QtDesigner into python code that is then the GUI to be used. How to use `pyuic4` is discussed in more detail in Chapter 5.

Throughout this chapter there are a series of examples that illustrate how to use some PyQt features. Further discussion of how these features were used to construct a GUI for the Two Tanks experiment is conducted in Chapter 5. Also in Chapter 5, illustrative code is discussed for several of the widgets constructed for the GUI.

3.2 Widgets

The Qt widget module provides a set of UI elements that allow for the creation of classic desktop-style user interfaces. Widgets are the primary elements for creating user interfaces in Qt. They can display data and status information, receive user input, and provide a container for other widgets that should be grouped together [3]. A widget not embedded in a parent widget is called a window.

The `QWidget` class gives the basic capability to render the screen and handle user input events. Any UI elements that Qt provides are subclasses of `QWidget` or are used in connection with a `QWidget` subclass.

3.2.1 Window and Dialog Widgets

Every GUI needs to have a main window that provides the screen space upon which the user interface is built. Windows visually separate applications and typically allow users to resize and position the applications as desired. This can be created in PyQt by using the `QMainWindow` class.

`QMainWindow` is used to set up menus, toolbars, and dock widgets. To add a menu bar to the main window, for example, a menu is created and then added to the main window's menu bar using `menuBar()`. `QMainWindow` has its own layout where the programmer can add a menu bar, tool bar, other dockable widgets, and a status bar. Once actions are created they must be added to the main window. Both main windows and dialogs can be created using QtDesigner or by manually coding them in Python.

If a `QWidget` is not assigned a parent, it automatically becomes a window. Typically only one window is allowed to not have a parent: the primary window.

Dialog windows are used as secondary windows that give the user options or choices. These are created using the subclassing `QDialog` along with widgets and layouts. PyQt provides a number of ready-made standard dialogs that can be used for tasks like opening a file or selecting a font. Dialog windows can also be used to create a window that confirms a user actually wants to exit an application when the close button is pushed.

Window Geometry

The `QWidget` object provides several functions that allow the programmer to handle a widget's geometry. Some functions that can be used that include the window frame are: `x(),y()`, `frameGeometry()`, `pos()` and `move()`.

Functions that exclude the window frame (i.e. just affect the widget's geometry) are: `geometry()`, `width()`, `height()`, `rect()` and `size()`. Figure 3.1 illustrates most of the available functions to specify the geometry of the window.

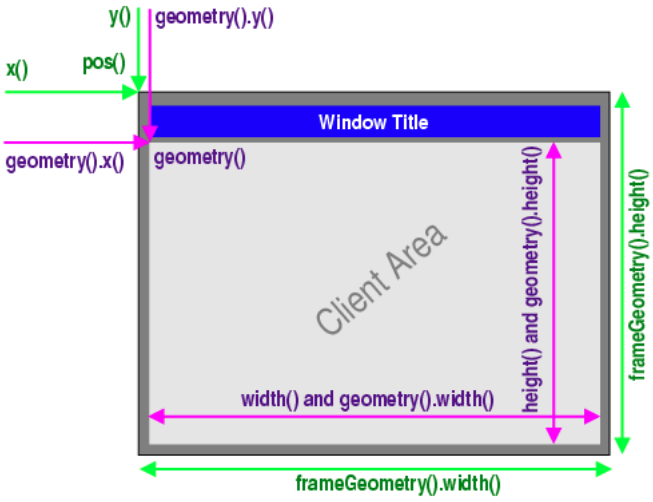

Figure 3.1: Specifying window geometry in PyQt [3]

3.3 Layout Managers

`QLayout` is the base class of geometry managers. It is an abstract base class that is inherited by the concrete classes `QBoxLayout`, `QGridLayout`, `QFormLayout`, and `QStackedLayout`, which all determine where widgets are located in the main window. `QLayout` takes as input the name of a `QWidget` and a parent. This specifies how a widget will appear in the main window.

3.3.1 Style

The style encapsulates the look and feel of a GUI. PyQt utilizes the `QStyle` class to do all the drawing so that they look exactly like native widgets. This means that whatever operating system the GUI is run on, it will adopted the style of that OS. For more customizable appearances, `QStyleSheets` can be used in addition to supplement what can already be done in `QStyle`.

3.3.2 Layouts

Layouts are a way to arrange child widgets within their own container. Each widget gives its size requirements to the layout through the sizeHint and sizePolicy properties and then the layout automatically distributes the available space. The designer can also specify where to position widgets within a window as well.

3.3.3 Example

We look at a simple example of generating a window with a grid layout that has 16 push buttons. This example shows how to create a window, specify the layout, and create push buttons. The code below utilizes the three classes `QApplication, QWidget, QGridLayout` and their methods [7].

```
1  import sys
2  from PyQt4.QtCore import *
3  from PyQt4.QtGui import *
4
5  def window():
6    app = QApplication(sys.argv)
7    win = QWidget()
8    grid = QGridLayout()
9
10   for i in range(1,5):
11    for j in range(1,5):
12      grid.addWidget(QPushButton("B"+str(i)+str(j)),i,j)
13   win.setLayout(grid)
14   win.setGeometry(100,100,200,100)
15   win.setWindowTitle("PyQt")
16   sys.exit(app.exec_())
17
18 if __name__ == '__main__':
19   window()
```

The result is shown in Figure 3.2.

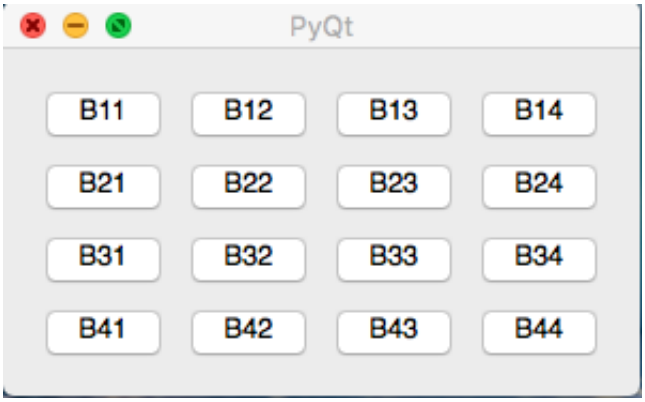

Figure 3.2: Simple example using layout and window widgets

The first three lines define the widget `QWidget` and specify the layout `QGridLayout`. Next, the for loop in lines 10-12 creates the sixteen different buttons `QPushButton`, labels them, and adds them to the grid layout (addWidget). Then we add the layout to the widget (`setLayout`), specify the widget geometry (`setGeometry`) and specify the window title (`setWindowTitle`). Finally we specify what to do when the application is launched. While this is a trivial example, it illustrates how few lines of code are required to create a window, specify properties like the layout, geometry and window tile, and add widgets to the window.

3.4 Threads, Signals and Slots

3.4.1 Threads

A thread contains different tasks to be performed. A software program may contain multiple threads with each one having different tasks to be performed. For most GUIs, it is common to create at least two different threads: one for the main window (main thread), and one that can handle behind the scene tasks (worker thread). This allows users to perform actions in the GUI while the data collection still operates.

For a laboratory GUI, it is important to have at least two threads. One thread that will collect data from the experiment and the other that will handle any user requests emitted from the GUI. The main thread will be idle until an action is performed in the GUI; then the main thread will take priority over the worker thread and send the tasks to be performed. After the tasks from the main thread are performed, the worker thread tasks will then continue. The tasks contained in a thread are specified using signals

and slots.

3.4.2 Signals and Slots

All `QObject` types support the signal and slot mechanism. These objects announce state changes and events like a checkbox being checked/unchecked or a button being clicked. Signals and slots allow for communication between objects. A signal is emitted when something happens in the application, i.e. when a user clicks on a button. A slot is a function that is called in response to a particular signal and contains some specified action to perform. When a signal is connected to a slot, the slot is called when the signal is emitted. If a signal is not connected to a slot, then nothing happens.

The signal/slot mechanism has the following features:

- A signal can be connected to many slots
- A signal may also be connected to another signal
- A slot can be connected to many signals
- Connections can be direct or queued
- Connections can be made across threads
- Signals do not have to be connected to slots

A signal has the methods `connect()`, `disconnect()` and `emit()`; these implement the associated functionality. PyQt4 automatically defines signals for all Qt's built-in signals and new signals can be defined as class attributes using `pyqtSignal()`. You can specify the signal type by writing `int,str,float` inside `pyqtSignal`. This means that the signal will only accept data of the specified type.

Signals are connected to slots using the `connect()` method of a signal. Signals can be disconnected from one or more slot using the `disconnect()` method. Signals are emitted using the `emit()` method of a bound signal. Whenever a signal is emitted, by default PyQt throws it away unless the signal is connected to a slot.

A Python callable (i.e. a function) can be used as a slot by marking it explicitly as a slot using the `pyqtSlot()` function decorator. It is also possible to define slots without marking it explicitly; this works by connecting a signal to a Python callable.

3.4.3 Example

This example demonstrates the use of signal and slots in PyQt [1].

```
1  import sys
2  from PyQt4 import QtGui, QtCore
3
4  class Example(QtGui.QWidget):
5
6      def __init__(self):
7          super(Example, self).__init__()
8
9          self.initUI()
10
11     def initUI(self):
12
13         lcd = QtGui.QLCDNumber(self)
14         sld = QtGui.QSlider(QtCore.Qt.Horizontal, self)
15
16         vbox = QtGui.QVBoxLayout()
17         vbox.addWidget(lcd)
18         vbox.addWidget(sld)
19
20         self.setLayout(vbox)
21         sld.valueChanged.connect(lcd.display)
22
23         self.setGeometry(300, 300, 250, 150)
24         self.setWindowTitle('Signal & slot')
25         self.show()
26
27 def main():
28
29     app = QtGui.QApplication(sys.argv)
30     ex = Example()
31     sys.exit(app.exec_())
32
33 if __name__ == '__main__':
34     main()
```

This creates the GUI shown in Figure 3.3.

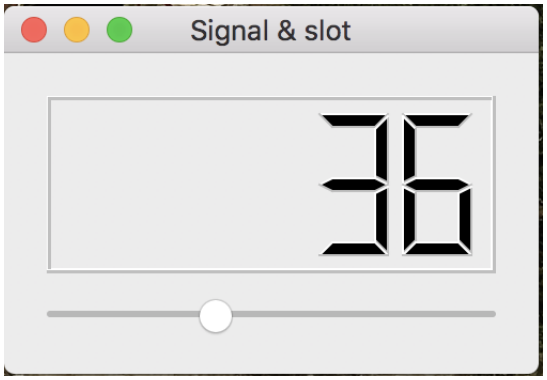

Figure 3.3: Simple example of use of signals and slots

First define the LCD number using `QLCDNumber` and the slider `QSlider` (lines 13-14). Then we define the layout (`QVBoxLayout`) and add the two widgets to it (`addWidget`). We then connect the slider so that when it changes, it emits the new value on the LCD number display (line 21). Finally, we specify the window geometry and the title.

Here the signal of the slider, `sld.valueChanged`, is connected to the `display` slot of the `lcd` number. Therefore, when the user slides the slider bar, the number is emitted and displayed above. Again this trivial example illustrates how few lines of code are required to create a simple GUI thanks to PyQt.

Chapter 4

Using QtDesigner

4.1 What is Qt Designer?

QtDesigner is a software program that allows for visual design and building of graphical user interfaces. You can design widgets, dialogs, or whole main windows using a simple drag and drop interface; Figure 4.1 illustrates the QtDesigner user interface using the Main Window template. This gives the added benefit of being able to preview designs and check that they work as desired before writing any code. QtDesigner has some precoded elements such as: layouts, spacers, buttons and containers. This can be useful for users who are new to programming or to those who need to quickly create a GUI. Additional custom widgets can be added to the QtDesigner, which was done for the Two Tank experiment.

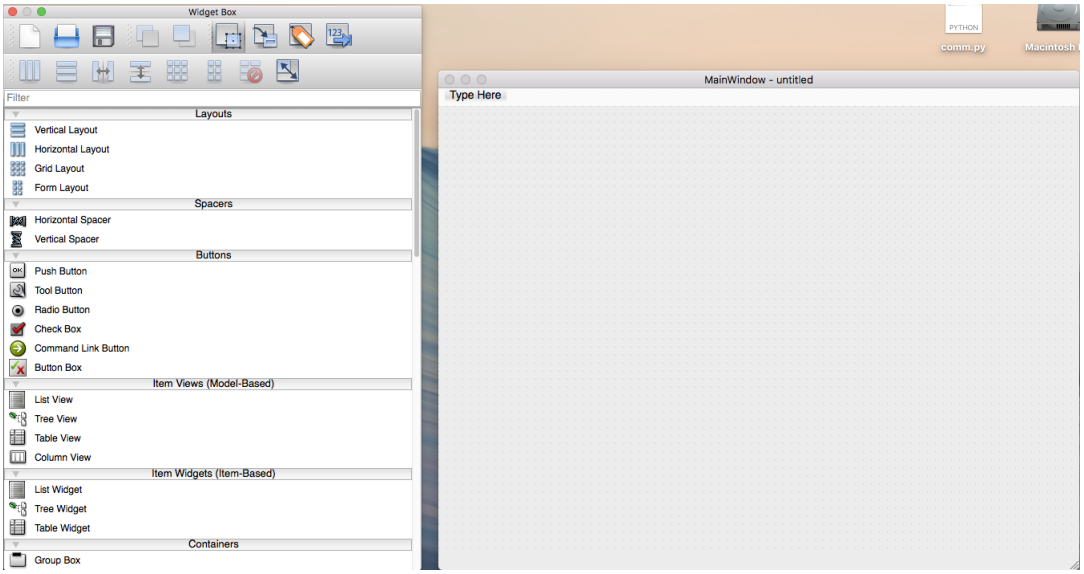

Figure 4.1: Blank Main Window template in QtDesigner

In QtDesigner, you simply drag and drop the desired components into a `MainWindow`. You can change the image used in the GUI for any of the components by changing the pixmap in the Property Editor. To connect the widgets together using signals and slots, click on the "Edit Signals and

Slots" button in the Widget Box. This will allow you to connect the widgets together and select from the available signals and slots in each widget.

QtDesigner generates a XML .ui file to store designs but the `uic` utility can be used to generate code that will create the user interface. To specifically create Python code, `pyuic4` should be used in the command line to translate the design to Python code. Exactly how to use this interface is outlined in further detail below.

4.1.1 `pyuic4`

The `pyuic4` is a command line interface to the uic module and the following syntax should be used:

```
1   pyuic4 [options] .ui-file
```

This allows you to transform any GUIs made in QtDesigner into Python code which can then be launched and used as GUI.

4.2 Installing FellesLab Widgets

Before you can create your own GUI using the widgets that have been created for the Two Tanks experiment, you must set the following path (assuming Python 2.7 is used):

```
1 export PYQTDESIGNERPATH =``[user]/lib/python2.7/site-packages/
2         felleslab/qt_plugins"
```

Then you must launch the QtDesigner from the terminal:

```
1 open -a designer #for Mac OSx
2 designer-qt4 #for Ubuntu (Linux)
```

This will launch the QtDesigner with all available FellesLab widgets. Currently there are five available widgets: thermocouple, real-time plotting of 1 CV and 2 MVS, solenoid valve, PID controller, and a trio of buttons (start, stop, and pause). When loaded into QtDesigner, they should appear in the left hand column in the Widgets Box and be listed under `QFellesLabWidgets` (Figure 4.2).

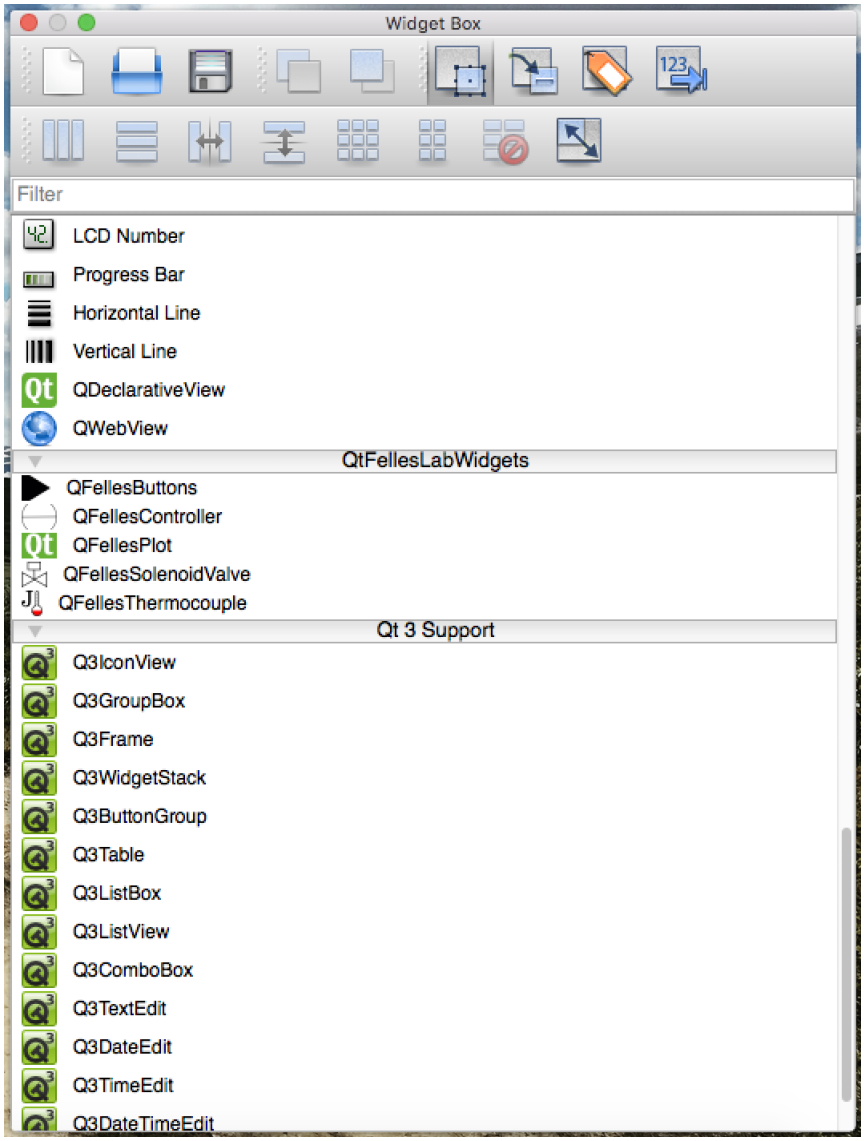

Figure 4.2: QtDesigner Widget Box with QFellesLabWidgets

Then you can use the widgets as you would any other built-in widget to create a GUI.

Chapter 5

Two Tanks GUI

5.1 Installation

To use the Two Tanks GUI, the following dependencies are required (only been tested on Ubuntu and Mac OSx):

- Ubuntu (Linux)

```
sudo apt install python-serial python-pip qt4-designer
  qt4-dev-tools pyqt4-dev-tools pyqtgraph
sudo pip install minimalmodbus
```

- Mac OSx (using Mac Ports)

```
sudo port install qt4-mac qt4-creator-mac py27-pyqt4
  py27-pip py27-serial pyqtgraph
sudo pip install minimalmodbus
```

The felleslab module can be downloaded from GitHub: `https://github.com/sigveka/FellesLab`.

After these dependencies are installed and the module has been downloaded, the following install procedure should be used:

```
python setup.py build
python setup.py install --prefix="${HOME}"
```

Each time you want to run the GUI, you must append the path to the Python path so that you can import the `felleslab` module:

```
export PYTHONPATH=${PYTHONPATH}/[path to felleslab]
```

In addition, if any changes are made to the code, the install must be rebuilt.

5.2 How it Works

At its current stage in development, the GUI for the Two Tanks Experiment is shown in Figure 5.1.

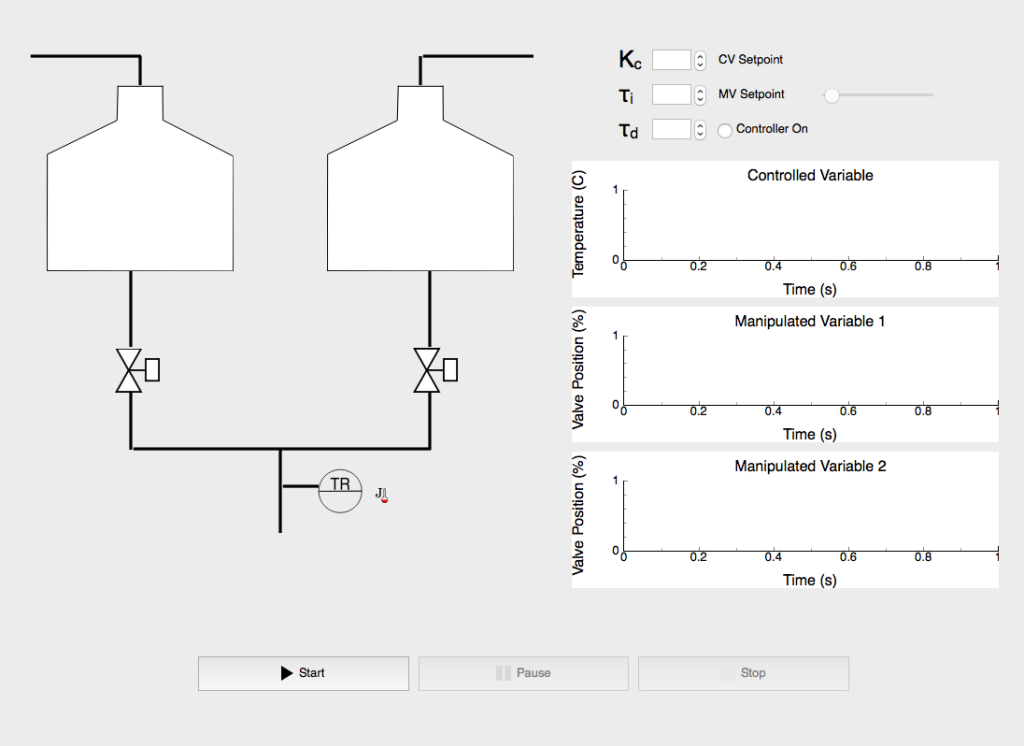

Figure 5.1: Two tanks laboratory GUI

There are two operation modes for the experiment: manual control or automatic control. The operation mode is determined by turning the controller on or off using the radio button located in the controller widget (upper right corner). In manual mode, users can open or close the solenoid valves by clicking on the valve images. If the valve is open, the image will be white; if the valve is closed, the image will be red (Figure 5.2).

Figure 5.2: Open and closed valves, respectively

In manual mode, the user can set the manipulated variable set point in the controller widget. The information flow in manual mode is illustrated in Figure 5.3.

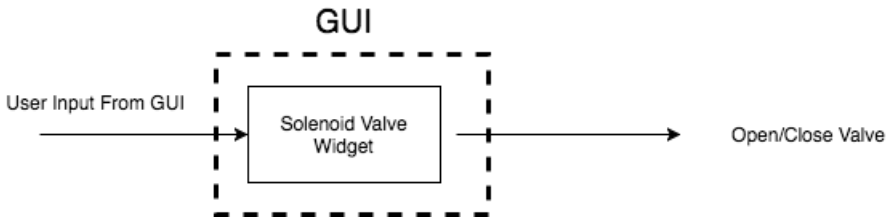

Figure 5.3: GUI Manual Mode

In automatic control, the user can no longer close or open the valves; thus clicking on the valve images in this mode will result in no action being performed. Instead, the state of the valves is manipulated by the controller. Users manually input the desired controller settings in the controller widget box in the GUI. In the same box, users can specify the controlled variable set point. In this mode, the controller will decide what the valve settings should be by using a separate algorithm. Currently this controller algorithm has not been implemented. The information flow in automatic mode is illustrated in Figure 5.4.

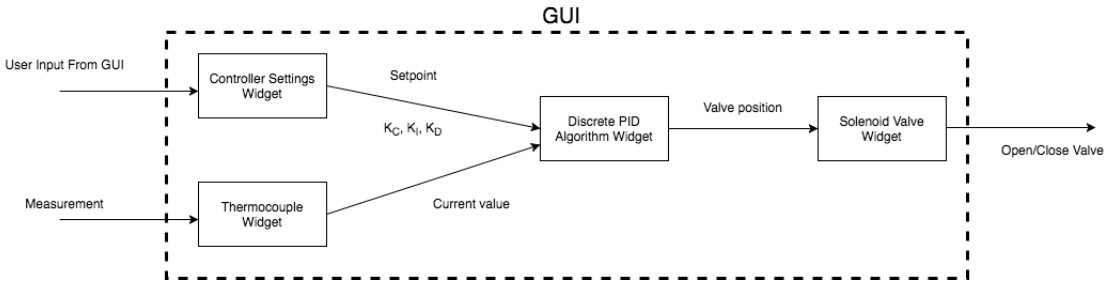

Figure 5.4: GUI Automatic Mode

When the GUI is first launched, users will notice at the bottom of the GUI three buttons: start, stop and pause. Of the three buttons only the start button is enabled initially, i.e. is the only button that can be pressed. Pushing the start button begins the data collection and also changes the state of the button. The real time plotting of the temperature and valve position begins. The start button becomes disabled and the pause and stop buttons become enabled. If the pause button is pressed, the data collection is paused meaning that data is still being recorded but not displayed in the real-time plots. If the stop button is pressed, the data collection is stopped and all

measurements are written out to a file and then cleared from memory. The state space diagram of the three buttons is illustrated in Figure 5.5.

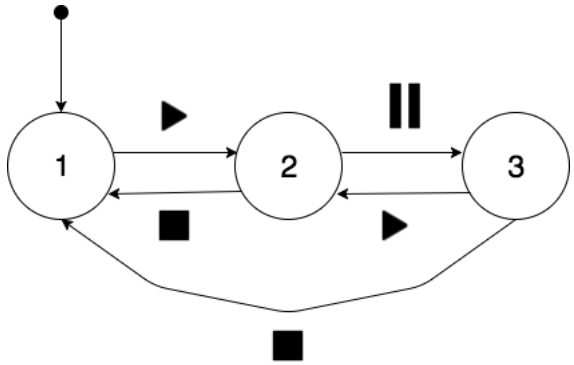

Figure 5.5: State space illustration of start, pause and stop buttons

There are three plots that update in real time; one plot of the temperature versus time and the other two plots show the two valve positions versus time. The data shown in the plots is saved in a data file for convenience. The plot widget is connected to the thermocouple and solenoid valve widgets so that it can receive data from the sampling. When a new sample is received from either widget, it is then added to the appropriate plot.

Behind the scenes, the sampling of the temperature and the solenoid valve positions is being conducted. Here we utilize two Python modules: `pyserial` and `minimalmodbus`. The analog to digital converter used in the Two Tanks experiments uses a USB serial connection that allows a sampling frequency of 10 Hz. Due to the hardware, we can only send or receive one signal at a time. As previously mentioned, the GUI has two threads: one main thread and one worker thread. The main thread gets priority over the worker thread; the sampling is done in the worker thread and the main thread sends any signals from the GUI. This means that get one sample at a time and if we have to send any input to the laboratory, this sampling gets delayed until the action from the main thread is completed.

From the thermocouple, the temperature reading is given as a byte integer. The thermocouple is a JType thermocouple so we know the range and that it is linear in this range. We can use a linear equation to convert this byte integer into a relevant temperature reading. This is done automatically in the felleslab module.

5.3 Custom Widget Examples

The widgets below were created for the Two Tanks experiment. However, they were written as generically as possible so that the widgets could be utilized in other GUIs with no or minor changes required. The code given in the widgets below is for illustrative purposes so that each widget could be run on its own (without needing the felleslab module); the implemented code for the GUI utilizes much of the code below so the idea is still the same. The solenoid valve, thermocouple widgets, and the code required for communication are not discussed here since they were written by Sigve Karolius. The full GUI source code can be found on GitHub as discussed at the beginning of this Chapter.

5.3.1 Start, Stop, Pause Buttons Widget

The start, stop, and pause buttons illustrate a few of the previously metioned built in PyQt widgets such as: `QMainWindow`, `QWidget`, `QPushButton`, `QApplication`, `QPixmap`, `QHBoxLayout`, `QVBoxLayout`, `QIcon`, `QCoreApplication`, and `pyqtSlot`. The buttons are grouped together into one widget; this could be added to QtDesigner since a plugin file has also been coded for it. The widget is coded such that the buttons are enabled or disabled depending on which one is clicked as discussed above. Figure 5.6 shows how the buttons would look in a GUI.

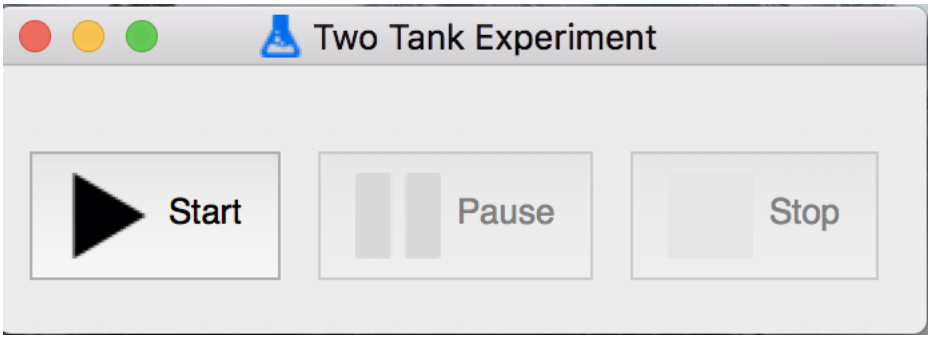

Figure 5.6: Start, Pause and Stop button widget

The code below creates and displays this trio of buttons.

```
1 import sys
2 from PyQt4.QtGui import (QPushButton, QWidget, QIcon,
3         QApplication, QPixmap, QHBoxLayout, QVBoxLayout,
```

```
4           QMainWindow)
5 from PyQt4.QtCore import QCoreApplication, pyqtSlot
6 from felleslab.icons import *
7
8 class MainWindow(QMainWindow):
9
10  def __init__(self, parent = None):
11      super(MainWindow, self).__init__(parent)
12      self.button_widget = ButtonWidget(self)
13      self.setCentralWidget(self.button_widget)
14      self.setWindowTitle('Two Tank Experiment')
15      self.setWindowIcon(QIcon('chemistry-lab-instrument.svg'))
16
17 #Button Widget
18 class ButtonWidget(QWidget):
19
20   def __init__(self, parent):
21     super(ButtonWidget, self).__init__()
22     self._state = 0 #default state of idle
23     self.initUI()
24     self.StartButton.clicked.connect(self.on_click)
25     self.PauseButton.clicked.connect(self.on_click)
26     self.StopButton.clicked.connect(self.on_click)
27
28   @property
29   def state(self):
30     return self._state
31
32   @state.setter
33   def state(self, value):
34     self._state = value
35
36   def initUI(self):
37     #Defining Buttons
38     self.StartButton = QPushButton("Start")
39     self.StartButton.setObjectName('Start')
40     self.PauseButton= QPushButton("Pause")
41     self.PauseButton.setObjectName('Pause')
42     self.StopButton = QPushButton("Stop")
43     self.StopButton.setObjectName('Stop')
44
```

```
45      #Button icons
46      icon_start = QIcon()
47      icon_pause = QIcon()
48      icon_stop = QIcon()
49      icon_start.addPixmap(QPixmap('play.svg'),
50                  QIcon.Normal)
51      icon_start.addPixmap(QPixmap('play_disabled.svg'),
52                  QIcon.Disabled)
53      self.StartButton.setIcon(icon_start)
54      icon_pause.addPixmap(QPixmap('pause.svg'),
55                  QIcon.Normal)
56      icon_pause.addPixmap(QPixmap('pause_disabled.svg'),
57                  QIcon.Disabled)
58      self.PauseButton.setIcon(icon_pause)
59      icon_stop.addPixmap(QPixmap('stop.svg'),
60                  QIcon.Normal)
61      icon_stop.addPixmap(QPixmap('stop_disabled.svg'),
62                  QIcon.Disabled)
63      self.StopButton.setIcon(icon_stop)
64
65      #Specifying button settings for initial state
66      if self.state == 0: #idle (default state)
67          self.StartButton.setEnabled(True)
68          self.PauseButton.setDisabled(True)
69          self.StopButton.setDisabled(True)
70      elif self.state == 1: #sampling
71          self.StartButton.setDisabled(True)
72          self.PauseButton.setEnabled(True)
73          self.StopButton.setEnabled(True)
74      elif self.state == 2: #paused
75          self.StartButton.setEnabled(True)
76          self.PauseButton.setDisabled(True)
77          self.StopButton.setEnabled(True)
78
79      #Defining Layout
80      hbox = QHBoxLayout()
81      hbox.addStretch(1)
82      hbox.addWidget(self.StartButton)
83      hbox.addWidget(self.PauseButton)
84      hbox.addWidget(self.StopButton)
85
```

```
86        self.setLayout(hbox)
87
88
89    @pyqtSlot()
90    def on_click(self, event=None):
91        sending_button = self.sender() #getting button name
92        btn_name = str(sending_button.objectName())
93        if btn_name == 'Start':
94            old_state = self.state
95            self.state = 1 #changing state to sampling
96            self.StartButton.setDisabled(True)
97            self.PauseButton.setEnabled(True)
98            self.StopButton.setEnabled(True)
99        elif btn_name == 'Pause':
100           old_state = self.state
101           self.state = 2 #changing state to paused
102           self.StartButton.setEnabled(True)
103           self.PauseButton.setDisabled(True)
104           self.StopButton.setEnabled(True)
105       elif btn_name == 'Stop':
106           old_state = self.state
107           self.state = 0 #changing state to idle
108           self.StartButton.setEnabled(True)
109           self.PauseButton.setDisabled(True)
110           self.StopButton.setDisabled(True)
111
112 def main():
113       app = QApplication(sys.argv)
114       GUI = MainWindow()
115       GUI.show()
116       sys.exit(app.exec_())
117
118 main()
```

Lines 8-16 specify the GUI main window properties. The class `MainWindow` simply gives instructions for how to create the GUIs main window. In this case, we add the ButtonWidget and make it the main widget by setting it be the central widget (`setCentralWidget`). The other two lines simply set the window title and the icon; this requires the use of the widget `QIcon` to convert an image file to an icon.

Lines 19-111 is the code for the class ButtonWidget; this creates the three buttons, adds images to the buttons, specifies what actions to perform when a button state changes, and sets the layout for the widget. The `__init__` method is a Python method that is a constructor method for a class. `__init__` is called whenever an object of the class is constructed. This means that whenever we create a ButtonWidget object we will perform the actions included in this method. The full details of it will not be discussed here but it is common to use this method in the creation of PyQt GUIs. Here we call another function and connect the buttons to a PyQt slot; this slot specifies what action is taken when each button is clicked.

The `initUI` function is where the buttons are created and their functionality is programmed. First the three buttons, using `QPushButton`, are created and given object names (lines 39-44). Next the buttons are given icons using `QIcon`. We can use the widget `QPixmap` to specify icons that will appear only when the button is in a specific state (i.e., Normal (enabled) or Disabled). Each icon is then added to its respective button (lines 47-64). Next we specify which buttons are enabled or disabled depending on the state value (lines 67-78). Note that we manually set the state value in the `__init__`. Currently the default is to have only the start button enabled but this can be changed to fit the users needs/desires. The layout of how the widget will look in the main window is then defined (lines 81-87). We want all the buttons to be next to each other so we use the widget `QHBoxLayout`; this horizontally aligns all the buttons.

Finally we create a slot using `pyqtSlot` where we specify what happens when a button is clicked. This slot is only used because signals were connected to it; this was done in the `__init__` function in lines 25-27. Once all of this is programmed, we still need to display the GUI; this is what is done in the function `main`. We then call this function and the GUI is displayed.

Note that in this example the buttons are not connected to anything so they don't perform any function. In a full GUI, each button (signal) would be connected to a slot that would then specify some desired action. For example, in the Two Tanks experiment, the start button would begin the data collection and start the real-time plot; the pause button would pause the real-time plot and the data collection; the stop button would end the data collection and result in the data being written out to a file.

5.3.2 PID Controller Input Widget

This example illustrates a GUI where users can set a PID controller's tuning parameters and specify the set points. Users can set the gain, integral time constant, derivative time constant, manipulated variable and controlled vari-

able values. In addition, the user can turn the controller on and off so manual control can also be used. Note that this widget simply provides the desired settings to a PID controller. It needs to be connected to a controller algorithm for it to actually work as a controller. This example illustrates the use of the following widgets and functionalities: `QDoubleSpinBox`, `QGroupBox`, `QRadioButton`, `QSlider`, `pyqtSlot` and `pyqtSignal`. The final product is as shown in Figure 5.7, where the figure on the left is what the widget looks like when the controller is off and the figure on the right is what the widget looks like when the controller is on.

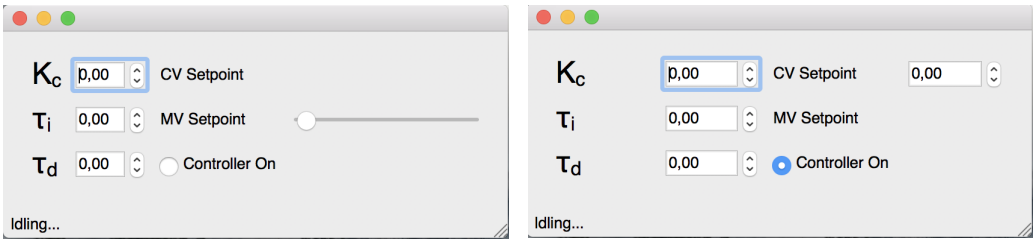

Figure 5.7: Controller GUI

This is generated using the following code.

```
1  import sys
2  from PyQt4.QtGui import (QGroupBox, QLabel, QDoubleSpinBox,
3          QGridLayout, QHBoxLayout, QWidget, QApplication,
4          QRadioButton, QIcon, QMainWindow, QSlider)
5  from PyQt4.QtCore import pyqtSlot, Qt, pyqtSignal
6
7  class Window(QMainWindow):
8    def __init__(self, parent=None):
9      super(Window, self).__init__(parent)
10     self.controller = ControllerWidget(self)
11     self.setCentralWidget(self.controller)
12     self.setWindowTitle('Controller')
13
14 class ControllerWidget(QWidget):
15
16   proportional = pyqtSignal(float)
17   integral = pyqtSignal(float)
18   derivative = pyqtSignal(float)
19   MV_sp = pyqtSignal(float)
20   CV_sp = pyqtSignal(float)
```

```
21
22      def __init__(self,parent):
23          super(ControllerWidget, self).__init__()
24          self.initUI()
25          self.on.toggled.connect(self.btnstate)
26          self.setpoint.valueChanged[str].connect(self.CVSetPoint)
27          self.gain.valueChanged[str].connect(self.controllersettings)
28          self.taui.valueChanged[str].connect(self.controllersettings)
29          self.taud.valueChanged[str].connect(self.controllersettings)
30          self.MV_manual.sliderReleased.connect(self.mv_state)
31
32      def initUI(self):
33          #Group box
34          group_box = QGroupBox('Controller Settings')
35
36          #Spin button Labels
37          gain = "K<sub>c</sub>"
38          Kc = gain.decode('utf-8')
39          self.gain_label = QLabel(Kc)
40          self.gain_label.setStyleSheet("font:25pt")
41          time1 = "&#964;<sub>i</sub>"
42          taui = time1.decode('utf-8')
43          self.taui_label = QLabel(taui)
44          self.taui_label.setStyleSheet("font:25pt ")
45          time2 = "&#964;<sub>d</sub>"
46          taud = time2.decode('utf-8')
47          self.taud_label = QLabel(taud)
48          self.taud_label.setStyleSheet("font:25pt")
49
50          #Spin buttons
51          self.gain = QDoubleSpinBox()#Gain
52          self.taui = QDoubleSpinBox()#Integral time constant
53          self.taud = QDoubleSpinBox()#Derivative time constant
54          self.GAIN_0 = self.gain.value()
55          self.TAUI_0 = self.taui.value()
56          self.TAUD_0 = self.taud.value()
57          self.gain.setKeyboardTracking(False)
58          self.taui.setKeyboardTracking(False)
59          self.taud.setKeyboardTracking(False)
60          self.setpoint = QDoubleSpinBox()#CV setpoint
61          self.setpoint.setHidden(True)
```

```
62        self.setpoint_label = QLabel('CV Setpoint')
63        self.setpoint_label.setBuddy(self.setpoint)
64
65        #On-Off Buttons
66        self.on = QRadioButton("Controller On")
67        self.on.setChecked(False)#Default to have controller off
68
69        #Slider
70        self.MV_manual = QSlider(Qt.Horizontal)
71        self.MV_manual_label = QLabel('MV Setpoint')
72
73        #Layout
74        controlsLayout = QGridLayout()
75        controlsLayout.addWidget(self.gain_label,0,0)
76        controlsLayout.addWidget(self.gain,0,1)
77        controlsLayout.addWidget(self.setpoint_label,0,2)
78        controlsLayout.addWidget(self.setpoint,0,3)
79        controlsLayout.addWidget(self.taui_label,1,0)
80        controlsLayout.addWidget(self.taui,1,1)
81        controlsLayout.addWidget(self.MV_manual_label,1,2)
82        controlsLayout.addWidget(self.MV_manual,1,3)
83        controlsLayout.addWidget(self.taud_label,2,0)
84        controlsLayout.addWidget(self.taud,2,1)
85        controlsLayout.addWidget(self.on,2,2)
86        controlsLayout.setRowStretch(3,1)
87
88        layout = QHBoxLayout()
89        layout.addLayout(controlsLayout)
90        self.setLayout(layout)
91
92
93    @pyqtSlot()
94    def CVSetPoint(self, event=None):
95        self.setpoint_value = self.setpoint.value()
96        self.CV_sp.emit(self.setpoint_value)
97
98    @pyqtSlot()
99    def controllersettings(self, event=None):
100       if self.gain.value() != self.GAIN_0:
101           self.GAIN_0 = self.gain.value()
102           self.gain_value = self.gain.value()
```

```
103             self.proportional.emit(self.gain_value)
104         elif self.taui.value() != self.TAUI_0:
105             self.TAUI_0 = self.taui.value()
106             self.taui_value = self.taui.value()
107             self.integral.emit(self.taui_value)
108         elif self.taud.value() != self.TAUD_0:
109             self.TAUD_0 = self.taud.value()
110             self.taud_value = self.taud.value()
111             self.derivative.emit(self.taud_value)
112 
113     @pyqtSlot()
114     def btnstate(self, event=None):
115         if self.on.isChecked() == True:
116             self.setpoint.setHidden(False)
117             self.MV_manual.setHidden(True)
118         else:
119             self.setpoint.setHidden(True)
120             self.MV_manual.setHidden(False)
121 
122     @pyqtSlot()
123     def mv_state(self, event=None):
124         MV_setpoint = self.MV_manual.value()
125         self.MV_sp.emit(MV_setpoint)
126 
127 application = QApplication(sys.argv)
128 
129 #Making Window
130 window = Window()
131 window.setWindowTitle('Group Box')
132 window.resize(220, 100)
133 window.show()
134 
135 sys.exit(application.exec_())
```

As was the case for the previous example, all the code in class Window is for the creation of the main window and is only used for testing/this example to display the controller widget. The class `ControllerWidget` contains all the necessary code to design a GUI for a PID controller input.

In the class ControllerWidget, six signals are defined using `pyqtSignal`: one for the proportional part of the controller, one for the integral part

of the controller, one for the derivative part of the controller, one for the controller state (off or on) and the other two are for the manipulated variable set point and the controlled variable set point respectively (lines 16-20). Each signal is given a specified object type to accept; in this case all the objects except only floats. Next in the `__init__` function, the signals are connected to slots lines(25-30); the slots are defined in lines 93-128. The first slot `CVSetPoint` takes the controlled variable set point and emits the value whenever the controlled variable set point is changed by the user. The second slot `controllersettings` emits the new tuning parameter if either the gain, integral time constant or derivative time constant is changed. The third slot `btnstate` changes what is displayed if the controller is turned on by clicking the radio button. The final slot `mv_state` emits the manipulated variable set point if it is changed.

In the `initUI` function, the components of the PID controller input are constructed. Using `QGroupBox`, we can collect all the components into one box making it easy to use this as a widget/plugin in QtDesigner. We then create all the labels for the spin buttons; we can use utf-8 and html notation to get greek letters and subscripts (lines 37-48). The spin buttons are then created using `QDoubleSpinBox`; this allows for float values in comparison to `QSpinBox` which only accepts integer values (lines 51-63). We turn the keyboard tracking off so that signals will only be emitted when the user has set the whole desired value; if it was on then a signal would be emitted for every value change instead of just the final value (lines 57-59). For example, if a user wanted to type 1,35 then a signal would be emitted for 1, for 1,3 and for 1,35; since this is not desired, we turn off the keyboard tracking.

Next we create an on button and use a `QRadioButton` and set the default state to be unchecked using `setChecked(False)` since we want the default state of the controller to be off. When the controller is turned off, we can set the manipulated variable value using the slider. This is created using `QSlider` and when the value is changed it emits a signal. This signal is connected to a slot so that the new manipulated variable value is emitted. Finally we specify the layout of the controller; we use `QGridLayout` to create the layout of the controllers and then add this to the overall layout using `QHBoxLayout` (lines 74-90).

5.3.3 Plotting Widget

The plotting widget illustrates the use of `pyqtgraph` which is a graphics and user interface library for Python that makes use of the QtGUI platform (via PyQt) for its high performance graphics and `numpy` for calculations [2]. An example of the plots is shown below for a set of data points.

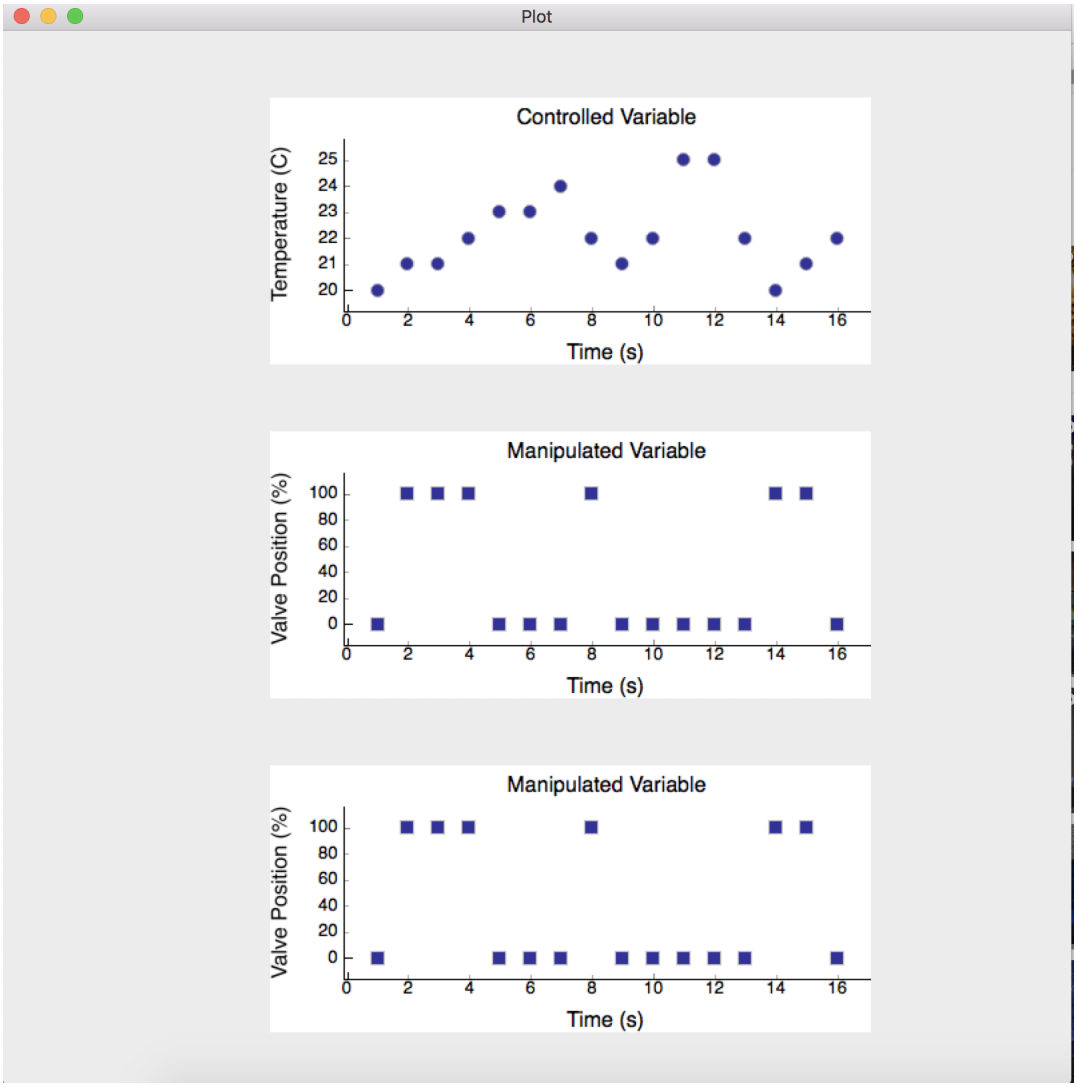

Figure 5.8: Illustration of real-time plotting utilities

This is generated using the code displayed below. This code differs slightly from the code implemented in the GUI but it illustrates the idea of how to use `pyqtgraph` to plot data in real time. The main differences lie in the data that is plotted. In the actual GUI, the plotting widget is connected to the thermocouple and solenoid valve widgets and receives the emitted temperature and valve position. Also note that here we simply programmed the plotting widget to be inside the main window, which is of course not how it is implemented in the lab GUI.

```
1  from PyQt4.QtGui import *
2  from PyQt4.QtCore import *
3  import numpy as np
4  import pyqtgraph as pg
5  import random
6  import sys
7  import datetime
8
9  class Window(QMainWindow):
10   def __init__(self):
11     super(Window, self).__init__()
12     self.setWindowTitle('Plot')
13     self.setGeometry(50,50,800,800)
14     self.home()
15   global i,v,T,s
16   i = 0
17   #Valve position
18   v = [0,100,100,100,0,0,0,100,0,0,0,0,0,100,100,0]
19   #Temperature
20   T = [20,21,21,22,23,23,24,22,21,22,25,25,22,20,21,22]
21   #Time
22   s = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
23
24   def home(self):
25     pg.setConfigOption('background','w')
26     pg.setConfigOption('foreground', 'k')
27
28     #Temperature plot widget
29     labelStyle = {'color': 'k', 'font-size': '16px'}
30     self.tempPlt = pg.PlotWidget(self)
31     self.tempPlt.setTitle('Controlled Variable',size='16px')
32     self.tempPlt.setLabel('left','Temperature',
33         units='C',**labelStyle)
34     self.tempPlt.setLabel('bottom','Time',
35         units='s',**labelStyle)
36     self.tempPlt.move(200,50)
37     self.tempPlt.resize(450,200)
38     self.timer3 = pg.QtCore.QTimer()
39     self.timer3.timeout.connect(self.cUpdate)
40     self.timer3.start(200)
41
```

```
42      #Valve 1 plot widget:
43      self.valve1Plt = pg.PlotWidget(self)
44      self.valve1Plt.setTitle('Manipulated Variable',size='16px')
45      self.valve1Plt.setLabel('left','Valve Position',
46          units='\%',**labelStyle)
47      self.valve1Plt.setLabel('bottom','Time',
48          units='s',**labelStyle)
49      self.valve1Plt.move(200,550)
50      self.valve1Plt.resize(450,200)
51      self.timer3 = pg.QtCore.QTimer()
52      self.timer3.timeout.connect(self.cUpdate)
53      self.timer3.start(200)
54
55      #Valve 2 plot widget:
56      self.valve2Plt = pg.PlotWidget(self)
57      self.valve2Plt.setTitle('Manipulated Variable',size='16px')
58      self.valve2Plt.setLabel('left','Valve Position',
59          units='\%',**labelStyle)
60      self.valve2Plt.setLabel('bottom','Time',
61          units='s',**labelStyle)
62      self.valve2Plt.move(200,300)
63      self.valve2Plt.resize(450,200)
64      self.timer3 = pg.QtCore.QTimer()
65      self.timer3.timeout.connect(self.cUpdate)
66      self.timer3.start(200)
67
68      self.show()
69
70    def cUpdate(self):
71      global i, v, T, s
72      self.tempPlt.plot(np.array([s[i]]),np.array([T[i]]),
73          pen=None,symbol='o')
74      self.valve1Plt.plot(np.array([s[i]]),np.array([v[i]]),
75          pen=None,symbol='s')
76      self.valve2Plt.plot(np.array([s[i]]),np.array([v[i]]),
77          pen=None,symbol='s')
78      i += 1
79
80  def run():
81      app=QApplication(sys.argv)
82      GUI = Window()
```

```
83          sys.exit(app.exec_())
84  run()
```

First we define some values that we can plot for this illustrative example (lines 17-22). Next we set up the three different plots; since all three plots have the same format we will just discuss the details of the temperature plot. First we define the plot configuration by specifying that the background should be white and the foreground should be black (lines 25-26). Next we specify the label style for the plot (line 29) and then create the temperature plot using `pyqtgraph.PlotWidget` (line 30). We then set the plot title using `setTitle` (line 31) and axis label using `setLabel` (line 32-35). The plot location and size is then specified inside the Main Window using `move` and `resize`. Finally we get a timer going for the plot using `QtCore.QTimer`. We then connect the timer to the `pyqtSlot cUpdate` so that whenever the timer timesout the actions specified in the slot are performed (line 39). Finally we specify that the timer starts with a timeout interval of 200 milliseconds.

Chapter 6

Conclusion

PyQt is an attractive open source alternative to LabView for the creation of GUIs for laboratory experiments. While LabView does provide more built-in utilities, the expensive price tag makes it unviable in certain circumstances. PyQt allows for a lot of customization making it applicable to a wide range of applications.

PyQt is a well documented Python module making it relatively easy to figure out how to use it. Using PyQt requires basic knowledge of Python syntax and logic; however, it is also possible to use QtDesigner to create a GUI for people who are not as comfortable with programming. Since PyQt is open source, it is also easier to maintain or change any of the code for the GUI; this is in strong contrast to LabView which can require professional specialists to provide maintence. PyQt has numerous functionalities beyond those discussed in this report making it possible to create GUIs for just about any desired application.

This project proved that it is possible to use PyQt to create a GUI for a laboratory experiment. While some further work needs to be done to improve the GUI, the current functionalities are: receive temperature measurements, plot temperature measurements in real time, send controller settings, open/-close solenoid valves, and start/pause/stop the experiment. Additional work that needs to be done to improve the GUI is outlined below:

- Inherit maximum and minimum values for the set point from the manipulated variable and controlled variable types
- Check that the experiment is connected; check that connection is to correct hardware
- Implement the controller algorithm
- Validate that all signals and slots work as desired
- Validate the plotting widget
- Confirm that data is saved as desired

The GUI widgets were even constructed in such a way as to be generic (with the exception of the plot widget) so that they may be utilized to make other graphic user interfaces.

This project utilized many of the built-in capabilities of PyQt and showed the proof of concept that PyQt could be used to develop a GUI that works with real time data collection and excepts user inputs. Based on this example usage, PyQt is recommended as a feasible replacement for LabView to develop GUI for laboratory experiments.

Bibliography

[1] Jan Bodnar. *Events and Signals in PyQt4*. Nov. 2017. URL: http://zetcode.com/gui/pyqt4/eventsandsignals/.

[2] Luke Campagnola. *PyQtGraph*. Nov. 2017. URL: http://www.pyqtgraph.org/documentation/index.html.

[3] The Qt Company. *Qt Documentation*. Oct. 2017. URL: http://doc.qt.io/qt-5/.

[4] Computer Hope. *GUI*. Oct. 2017. URL: https://www.computerhope.com/jargon/g/gui.htm.

[5] National Instruments. *Labview*. Oct. 2017. URL: http://www.ni.com/en-us/shop/labview.html.

[6] Riverbank Computing Limited. *PyQt4 Reference Guide*. Oct. 2017. URL: http://pyqt.sourceforge.net/Docs/PyQt4/.

[7] Tutorials Point. *PyQt-QGridLayout Class*. Nov. 2017. URL: https://www.tutorialspoint.com/pyqt/pyqt_qgridlayout_class.htm.

[8] Wikipedia. *Qt (software)*.

[9] Wikipedia. *Software Industry*.