

Robot Telemetry

- In this lesson, we will add telemetry to the robot and display it in the remote control program.
- The timer will be used to send telemetry ten times a second. We will add other telemetry later. We will start with the wheel encoders and then add the z-axis Gyro data from the IMU sensor.
- Open **rover7.py** and then save it as **rover11.py**. We do this in case we need the original rover7.py program later.
- We need to import the **Timer** module.
- In the **__init__()** function, add the Timer function. It will be set up to call the telemetry function that will be added to the UGV class..
- The timer is started after the network connection.

```
import ugv
import time
from machine import Pin, I2C, Timer
import select
import connect
from pid import PID

class UGV():
    def __init__(self):
        i2c = I2C(sda=Pin(4), scl=Pin(5))
        drv = ugv.ugv(i2c)
        last_enc = [0,0,0,0]
        inpid = [0,0,0,0]
        outm = [0,0,0,0]
        self.enc = (0,0,0,0)
        pid = []
        for i in range(4):
            pid.append(PID(1.0,0.002,0.0,setpoint=0))
            pid[i].output_limits = (-127,127)
        self.sk,self.addr = connect.connect()
        timer = Timer(period=100,mode=Timer.PERIODIC,\
                      callback=self.telemetry)
```

- In the while loop, add one variable called self.enc.
- It will be assigned the values from enc. Since self.enc is like a global variable in the class, it is a different variable from enc.
- The variable enc is only available in the function `__init__()`.
- self.enc can be accessed from any function within the UGV class. It will be used in the telemetry function.

```

while 1:
    ready = select.select([self.sk],[],[],0.01)
    if ready[0]:
        d,a = self.sk.recvfrom(64)
        print(d)
        d = d.decode().split(' ')
        if len(d) == 2:
            left = float(d[0])
            right = float(d[1])
            pid[0].setpoint = left
            pid[1].setpoint = left
            pid[2].setpoint = right
            pid[3].setpoint = right
            if left == 0:
                pid[0].reset()
                pid[1].reset()
            if right == 0:
                pid[2].reset()
                pid[3].reset()
        enc = drv.get_encoders()
        self.enc = enc
        for i in range(4):
            inpid[i] = enc[i] - last_enc[i]
            last_enc[i] = enc[i]
            outm[i] = pid[i](inpid[i])
        drv.drive(int(outm[0]),int(outm[1]),\
                  int(outm[2]),int(outm[3]))
        time.sleep(0.02)

```

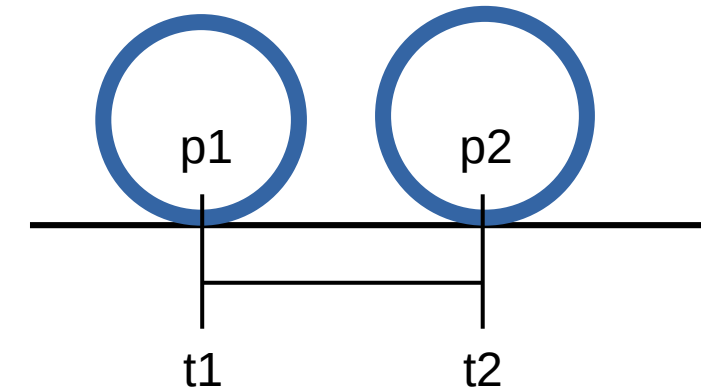
UGV()

- At the end of the class, add the telemetry function.
- This function will send time in seconds since the start of the program and the left and right wheel encoder counts. The time function gets time in milliseconds and divides it by 1000 to convert to seconds. The fractional time will be needed later.
- Make sure the function is indented like the other functions.

```
def telemetry(self,e):  
    ctime = time.time()/1000.0  
    msg = "%f %d %d %d %d\n" % (ctime,self.enc[0],self.enc[1],  
                                self.enc[2],self.enc[3])  
    self.sk.sendto(msg.encode(),self.addr)
```

Robot Program

- The speed of the robot can be calculated. All it requires is knowing the distance traveled, the time of the traveled distance, and the radius of the wheels.
- The robot provides distance information with the wheel encoders. Knowing the encoder count and the distance per encoder tick, the position of the wheel can be calculated by multiplying the encoder count by the encoder tick distance.
- $p2$ and $p1$ can be considered the encoder count at the two times. The distance traveled is the difference between the two encoder counts. pd is the pulse distance which was calculated to be 0.0001476 m.
- The robot can provide a time stamp. Knowing the time at the two positions, the time difference can be calculated.
- The speed is then the distance divided by the time difference.
- Calculate the speed with the following: $p2 = 320$, $p1 = 250$, $t2 = 0.5$, $t1 = 0.3$. Don't forget to use the encoder tick distance from the encoder lesson.



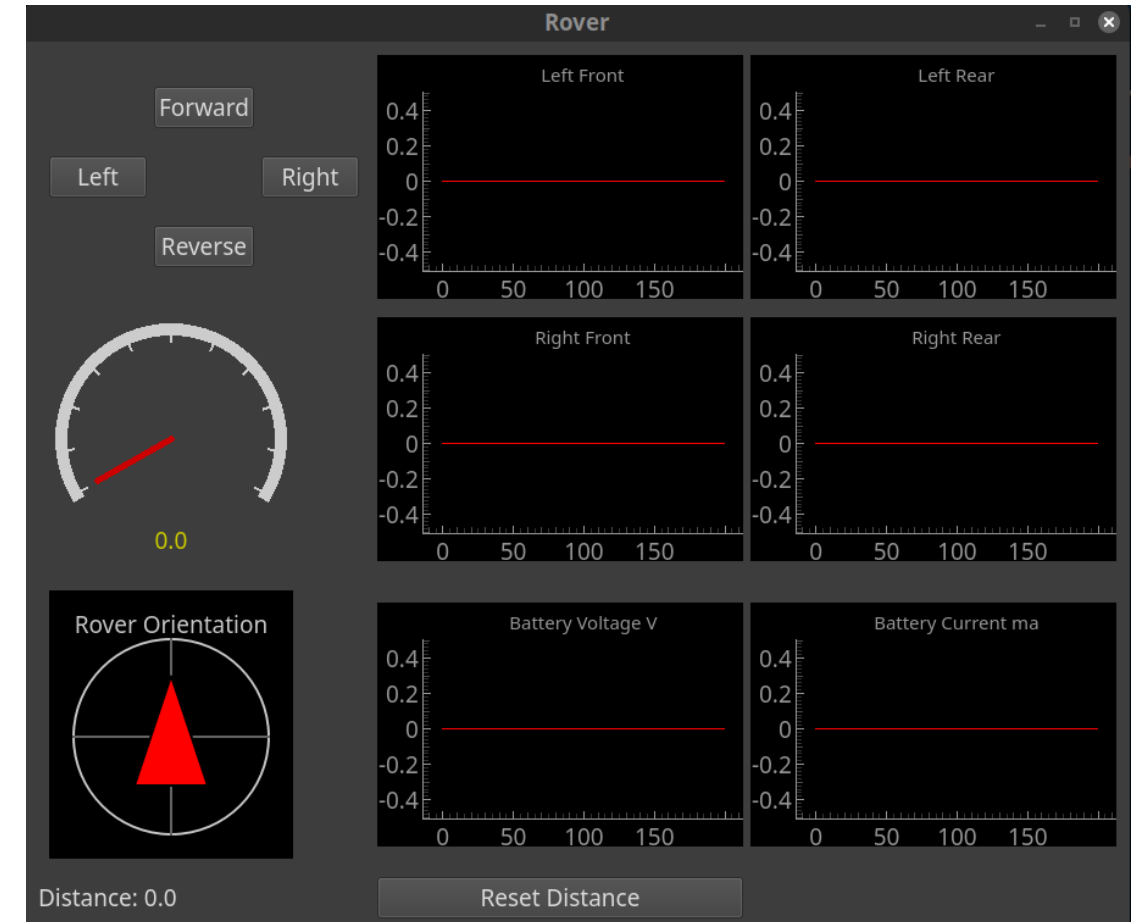
$$\text{distance} = (p2 - p1) * pd$$

$$T_{diff} = t2 - t1$$

$$\text{speed} = \text{distance} / T_{diff}$$

Displaying Telemetry

- We will create a program to display telemetry and control the robot. We will use the buttons program as a widget in this telemetry program.
- The telemetry program will start off with the buttons widget and plotting the speed of each wheel.
- Afterwards, we will add battery telemetry and then create a couple widgets to display the rover speed and orientation.
- We will also use a configuration file that will include ticks per revolution and the wheel radius. This allows different wheels to be used so you do not have to update parameters in the code. JSON will be used which is a standard format for defining parameters. The format is simple.

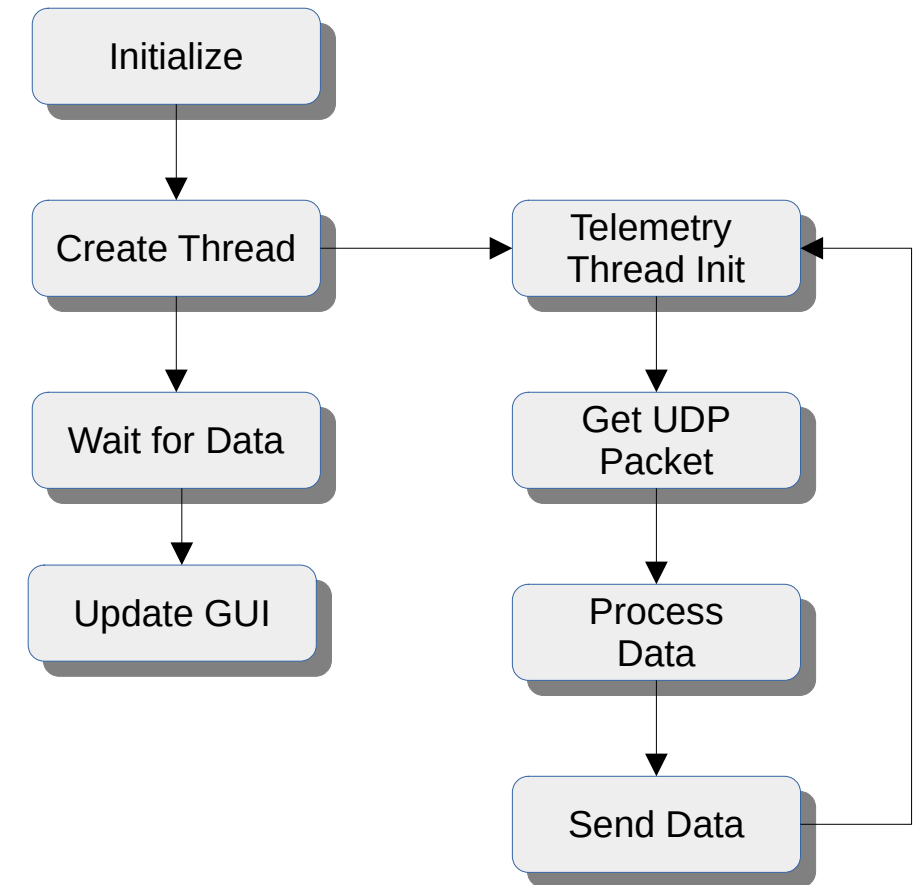


- Create a file in the same location where the laptop python code will be located.
- Name the file **ugv.conf**.
- Enter the text shown to the right. The radius is in meters.
- This is a simple JSON file. Each parameter is named in quotes. A colon is the separator and the text to the right is the parameter.
- The parameter can be a number as shown or it can be text in quotes.
- Each parameter is separated by a comma. The last parameter does not have a comma at the end.
- You can add more parameters if you wish.

```
{  
  "radius" : 0.049,  
  "ticks" : 2086  
}
```

Rover GUI Interface

- The GUI program will be made of three parts
 - The first part is setting up the GUI display, starting a thread which will be a separate class that will execute simultaneously, and setting up data connections with the thread.
 - The second part is the thread which will execute on its own and wait for telemetry data, process it, and send it to the GUI.
 - The third part are the functions that wait for data being sent from the thread and updating the GUI.



Thread Communications

- In PyQt, threads are created as a class. It keeps the program modular.
- PyQt provides a way for threads to communicate with the main program called signals.
- Signals are created in the thread class and the main program can connect to the signals.
- Signals need to be defined and the type of data needs to be declared.
 - Data types can be simple as int, float, and str or complex data like lists tuples, and more complex data structures.
- The thread must send data through the signal using a function name called emit.
- For a thread to receive data from the main program, a function needs to be declared to accept data. The main program calls the function in the thread class and passes arguments. The thread needs to monitor any variables for changes to know when the main program sent anything.
- The thread can be controlled by creating functions in the thread to be called by the main program. One such function can be used to end the execution of the thread.

- Open a new file and enter the code to the right. Save it as **telemetry.py**.
- The class is created just like other classes with some differences.
- Before the **__init__()** function, the signals are declared.
- The signal will pass a list which will contain the time tag and the four encoder values.
- The **__init__()** function initializes the class.

```
import sys
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *
import socket
import json

class Telemetry(QThread):
    encoders = pyqtSignal(list)

    def __init__(self):
        super().__init__()

    @pyqtSlot()
    def run(self):
        fdjson = open('ugv.conf', 'r')
        contents = fdjson.read()
        fdjson.close()
        configs = json.loads(contents)
        TICKS = configs['ticks']
        RADIUS = configs['radius']
        TICKSIZE = (RADIUS * 2 * 3.14159) / TICKS
        last_enc = [0, 0, 0, 0]
        diff_enc = [0, 0, 0, 0]
        enc = [0, 0, 0, 0]
        self.odometer = 0
        last_ttag = 0
        self.is_running = True
        sk = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sk.connect(('8.8.8.8', 80))
        local = sk.getsockname()[0]
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.bind((local, 10000))
```

- Something new is `@pyqtSlot()`. This is called a decorator. It is used to mark a python function as a slot to which PyQt signals can be connected. It isn't necessary but using it reduces the memory usage and makes the function run a little faster.
- The function **`run()`** must be the name. This is the thread that is started. After initialization, the **`run()`** function is automatically executed.
- First, the configuration file called `ugv.conf` is opened and the file contents are read into variable **`contents`**. The file is then closed.
- Next, the contents of the file are processed through a json parser which will generate a dictionary object called **`configs`**.

```
import sys
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *
import socket
import json

class Telemetry(QThread):
    encoders = pyqtSignal(list)

    def __init__(self):
        super().__init__()

    @pyqtSlot()
    def run(self):
        fdjson = open('ugv.conf', 'r')
        contents = fdjson.read()
        fdjson.close()
        configs = json.loads(contents)
        TICKS = configs['ticks']
        RADIUS = configs['radius']
        TICKSIZE = (RADIUS * 2 * 3.14159) / TICKS
        last_enc = [0,0,0,0]
        diff_enc = [0,0,0,0]
        enc = [0,0,0,0]
        self.odometer = 0
        last_ttag = 0
        self.is_running = True
        sk = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sk.connect(('8.8.8.8', 80))
        local = sk.getsockname()[0]
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.bind((local, 10000))
```

- The tick value is extracted from the JSON data.
- The wheel radius is extracted from the JSON data.
- The tick step size is calculated which will be used to calculate speed and distance.
- variables are created. **last_enc** is a list that will contain the previous encoder values for each wheel.
- **diff_enc** is a list that contains the difference between the current wheel encoder values and the previous wheel encoder values.
- **enc** will hold the current wheel encoder values.
- **self.odometer** will accumulate the distance traveled. It is the rover odometer and the units are in meters.
- **last_ttag** is the previous data time tag value.

```
import sys
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *
import socket
import json

class Telemetry(QThread):
    encoders = pyqtSignal(list)

    def __init__(self):
        super().__init__()

    @pyqtSlot()
    def run(self):
        fdjson = open('ugv.conf', 'r')
        contents = fdjson.read()
        fdjson.close()
        configs = json.loads(contents)
        TICKS = configs['ticks']
        RADIUS = configs['radius']
        TICKSIZE = (RADIUS * 2 * 3.14159) / TICKS
        last_enc = [0, 0, 0, 0]
        diff_enc = [0, 0, 0, 0]
        enc = [0, 0, 0, 0]
        self.odometer = 0
        last_ttag = 0
        self.is_running = True
        sk = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sk.connect(('8.8.8.8', 80))
        local = sk.getsockname()[0]
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.bind((local, 10000))
```

- A UDP socket is created and a connection is attempted. This selects the laptop's network interface. The IP address of the laptop can then be acquired.
- A UDP socket is recreated and bound to the laptop port 10000 using the laptop IP address.

```
import sys
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *
import socket
import json

class Telemetry(QThread):
    encoders = pyqtSignal(list)

    def __init__(self):
        super().__init__()

    @pyqtSlot()
    def run(self):
        fdjson = open('ugv.conf', 'r')
        contents = fdjson.read()
        fdjson.close()
        configs = json.loads(contents)
        TICKS = configs['ticks']
        RADIUS = configs['radius']
        TICKSIZE = (RADIUS * 2 * 3.14159) / TICKS
        last_enc = [0, 0, 0, 0]
        diff_enc = [0, 0, 0, 0]
        enc = [0, 0, 0, 0]
        self.odometer = 0
        last_ttag = 0
        self.is_running = True
        sk = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sk.connect(('8.8.8.8', 80))
        local = sk.getsockname()[0]
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.bind((local, 10000))
```

- The code moves on to the while loop. The variable **self.is_running** is checked to make sure it is greater than zero.
- The program then waits for telemetry from the rover.
- The data is decoded into a string and split using the space as a delimiter. Variable d becomes a list with all the telemetry elements.
- The length of the list is checked to make sure it includes the time tag and the four wheel encoder values.

```
while self.is_running:
    d,a = s.recvfrom(256)
    d = d.decode().split(' ')
    if len(d) == 5:
        ttag = float(d[0])
        diff_t = ttag - last_ttag
        last_ttag = ttag
        dist = 0.0
        for i in range(4):
            enc[i] = int(d[i+1])
            diff_enc[i] = enc[i] - last_enc[i]
            last_enc[i] = enc[i]
            dist += diff_enc[i]
        dist = dist/4
        dist = (dist * TICKSIZE)
        self.odometer += abs(dist)
        msg = [ttag,diff_enc,self.odometer,dist/diff_t]
        self.encoders.emit(msg)
```

- The time tag is converted from a string to a floating point value.
- The time difference is calculated.
- **last_ttag** is updated with the current time tag.
- Variable **dist** is set to zero. It will be used in the for loop and needs to be reset.

```
while self.is_running:
    d,a = s.recvfrom(256)
    d = d.decode().split(' ')
    if len(d) == 5:
        ttag = float(d[0])
        diff_t = ttag - last_ttag
        last_ttag = ttag
        dist = 0.0
        for i in range(4):
            enc[i] = int(d[i+1])
            diff_enc[i] = enc[i] - last_enc[i]
            last_enc[i] = enc[i]
            dist += diff_enc[i]
        dist = dist/4
        dist = (dist * TICKSIZE)
        self.odometer += abs(dist)
        msg = [ttag,diff_enc,self.odometer,dist/diff_t]
        self.encoders.emit(msg)
```

- The for loop will sequence through the four wheel encoders.
- First, the wheel encoder value is converted from a string to an integer.
- The wheel encoder difference is calculated.
- the **last_enc** value is updated with the current encoder value.
- The wheel encoder difference value is accumulated in the **dist** variable.

```
while self.is_running:
    d,a = s.recvfrom(256)
    d = d.decode().split(' ')
    if len(d) == 5:
        ttag = float(d[0])
        diff_t = ttag - last_ttag
        last_ttag = ttag
        dist = 0.0
        for i in range(4):
            enc[i] = int(d[i+1])
            diff_enc[i] = enc[i] - last_enc[i]
            last_enc[i] = enc[i]
            dist += diff_enc[i]
        dist = dist/4
        dist = (dist * TICKSIZE)
        self.odometer += abs(dist)
        msg = [ttag,diff_enc,self.odometer,dist/diff_t]
        self.encoders.emit(msg)
```

- Next **dist** is divided by four which gives the average of all the wheel encoder difference values.
- **dist** is then multiplied by the encoder tick size. This is the distance traveled within the time difference.
- the **dist** value is added to the **odometer** variable which accumulates the distance traveled at each telemetry interval.
- **msg** is a list variable that contains the current time tag, the encoder difference list, the **odometer** variable, and the speed of the rover.
- The list is emitted so the other code that uses it will update the GUI.

```
while self.is_running:
    d,a = s.recvfrom(256)
    d = d.decode().split(' ')
    if len(d) == 5:
        ttag = float(d[0])
        diff_t = ttag - last_ttag
        last_ttag = ttag
        dist = 0.0
        for i in range(4):
            enc[i] = int(d[i+1])
            diff_enc[i] = enc[i] - last_enc[i]
            last_enc[i] = enc[i]
            dist += diff_enc[i]
        dist = dist/4
        dist = (dist * TICKSIZE)
        self.odometer += abs(dist)
        msg = [ttag,diff_enc,self.odometer,dist/diff_t]
        self.encoders.emit(msg)
```

- The next function is used to stop the thread. It sets the **self.is_running** variable to False. This will stop the while loop. The main GUI program calls this function.
- The last function lets you reset the odometer value. A button in the GUI will be included to trigger the reset.
- Save this as **telemetry.py**

```
while self.is_running:
    d,a = s.recvfrom(256)
    d = d.decode().split(' ')
    if len(d) == 5:
        ttag = float(d[0])
        diff_t = ttag - last_ttag
        last_ttag = ttag
        dist = 0.0
        for i in range(4):
            enc[i] = int(d[i+1])
            diff_enc[i] = enc[i] - last_enc[i]
            last_enc[i] = enc[i]
            dist += diff_enc[i]
        dist = dist/4
        dist = (dist * TICKSIZE)
        self.odometer += abs(dist)
        msg = [ttag,diff_enc,self.odometer,dist/diff_t]
        self.encoders.emit(msg)

def stop(self):
    self.is_running = False

def reset_distance(self):
    self.odometer = 0.0
```

Main GUI Program

- Open a new file on the laptop and save it as **idle11.py**.
- Aside from the usual modules that get imported, pyqtgraph will be imported to plot the speed of the wheels.
- The idle9a.py program is imported as a widget.
- telemetry.py is imported so this program can run it as a thread.

```
import sys
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *
import pyqtgraph as pg
import idle9a
import telemetry

class Rover(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Rover')
        lfplot = pg.PlotWidget()
        lrplot = pg.PlotWidget()
        rfplot = pg.PlotWidget()
        rrplot = pg.PlotWidget()
        lfplot.setFixedSize(300,200)
        lrplot.setFixedSize(300,200)
        rfplot.setFixedSize(300,200)
        rrplot.setFixedSize(300,200)
        lfplot.setTitle("Left Front")
        lrplot.setTitle("Left Rear")
        rfplot.setTitle("Right Front")
        rrplot.setTitle("Right Rear")
        self.dist = QLabel("Distance: 0.0")
        buttons = idle9.Buttons()
        pen = pg.mkPen(color=(255,0,0))
        self.timeline = list(range(200))
        self.lfy = [0.0] * 200
        self.lry = [0.0] * 200
        self.rfy = [0.0] * 200
        self.rry = [0.0] * 200
        self.lfline = lfplot.plot(self.timeline, self.lfy, pen=pen)
        self.lrline = lrplot.plot(self.timeline, self.lry, pen=pen)
        self.rfline = rfplot.plot(self.timeline, self.rfy, pen=pen)
        self.rrline = rrplot.plot(self.timeline, self.rry, pen=pen)
```

Laptop Program

Main GUI Program

- In the `__init__()` function, the Window title is set.
- Four plots are created and set to a size of 300 by 200 pixels high and wide.
- Titles are created for the four plots.
- The distance label widget is created and will display the rover distance travelled.
- The buttons widget is added.
- The pen color is set to red. This will be the color of the plot lines.
- **self.timeline** is a list with element values from 0 to 199. This is for the X-axis of the plots.
- The four other lists are the y-axis for each wheel plot. The list is initialized with all zeros and is 200 elements long.
- The plots are then configured to plot x and y with red lines.

```
import sys
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *
import pyqtgraph as pg
import buttons
import telemetry

class Rover(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle('Rover')
        lfplot = pg.PlotWidget()
        lrplot = pg.PlotWidget()
        rfplot = pg.PlotWidget()
        rrplot = pg.PlotWidget()
        lfplot.setFixedSize(300,200)
        lrplot.setFixedSize(300,200)
        rfplot.setFixedSize(300,200)
        rrplot.setFixedSize(300,200)
        lfplot.setTitle("Left Front")
        lrplot.setTitle("Left Rear")
        rfplot.setTitle("Right Front")
        rrplot.setTitle("Right Rear")
        self.dist = QLabel("Distance: 0.0")
        buttons = idle9.Buttons()
        pen = pg.mkPen(color=(255,0,0))
        self.timeline = list(range(200))
        self.lfy = [0.0] * 200
        self.lry = [0.0] * 200
        self.rfy = [0.0] * 200
        self.rry = [0.0] * 200
        self.lfline = lfplot.plot(self.timeline, self.lfy, pen=pen)
        self.lrline = lrplot.plot(self.timeline, self.lry, pen=pen)
        self.rfline = rfplot.plot(self.timeline, self.rfy, pen=pen)
        self.rrline = rrplot.plot(self.timeline, self.rry, pen=pen)
```

Laptop Program

Main GUI Program

- Continuing in the `__init__()` function, the buttons widget is declared. it is declared by specifying the module name and the class name in the module.
- Next, the GUI layout is created. The layout will use the grid.
- The buttons widget is positioned in the top left corner with row 0, column 0.
- The four plots are positioned to the right of the buttons widget with the left wheels on top and the right wheels below.
- The distance label is positioned at the bottom left.

```
layout = QGridLayout()
layout.addWidget(buttons, 0, 0)
layout.addWidget(lfplot, 0, 1)
layout.addWidget(lrplot, 0, 2)
layout.addWidget(rfplot, 1, 1)
layout.addWidget(rrplot, 1, 2)
layout.addWidget(self.dist, 3, 0)
self.setLayout(layout)
self.thread = telemetry.Telemetry()
self.thread.start()
self.thread.encoders.connect(self.update_encoders)
```

Laptop Program

Main GUI Program

- Next, the telemetry thread is set up. First, the thread object is declared by referencing the Telemetry class in telemetry.py.
- The thread is started.
- The encoders signal is connected to the function update_encoders. Any time the telemetry thread emits the data, the function will be executed.

```
layout = QGridLayout()
layout.addWidget(buttons,0,0)
layout.addWidget(lfplot,0,1)
layout.addWidget(lrplot,0,2)
layout.addWidget(rfplot,1,1)
layout.addWidget(rrplot,1,2)
layout.addWidget(self.dist,3,0)
self.setLayout(layout)
self.thread = telemetry.Telemetry()
self.thread.start()
self.thread.encoders.connect(self.update_encoders)
```

Laptop Program

Main GUI Program

- Add the function **update_encoders()** to the GUI program.
- **update_encoders()** gets the **data** from the signal as an argument.
- The time tag and encoder difference values, distance and speed are extracted from the list.
- First, the plot y lists are updated. The first element is removed and all elements are shifted left one element.
- Next, the current data is appended to the list for each wheel.
- Next, the plots are updated with the updated lists.
- Lastly, the distance label is updated with the distance value. The distance will be displayed to three decimal places. It is in meters.

```
def update_encoders(self, data):
    ttag = data[0]
    enc = data[1]
    dist = data[2]
    speed = data[3]
    self.lfy = self.lfy[1:]
    self.lry = self.lry[1:]
    self.rfy = self.rfy[1:]
    self.rry = self.rry[1:]
    self.lfy.append(enc[0])
    self.lry.append(enc[1])
    self.rfy.append(enc[2])
    self.rry.append(enc[3])
    self.lfline.setData(self.timeline, self.lfy)
    self.lrline.setData(self.timeline, self.lry)
    self.rfline.setData(self.timeline, self.rfy)
    self.rrline.setData(self.timeline, self.rry)
    self.dist.setText("Distance: %0.3f" % dist)

def closeEvent(self, event):
    self.thread.stop()
    event.accept()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = Panel()
    win.show()
    app.exec()
```

Laptop Program

Main GUI Program

- PyQt has a standard function called **closeEvent()**. This is called when you click on the GUI window close button.
- This function is used to stop the thread. The thread **stop()** function is called. and then **event.accept()** is called which starts the shutdown of the program.
- Lastly is the code to check if the this program is being used as a widget or being run as a program. The program is started.

```
def update_encoders(self, data):
    ttag = data[0]
    enc = data[1]
    dist = data[2]
    speed = data[3]
    self.lfy = self.lfy[1:]
    self.lry = self.lry[1:]
    self.rfy = self.rfy[1:]
    self.rry = self.rry[1:]
    self.lfy.append(enc[0])
    self.lry.append(enc[1])
    self.rfy.append(enc[2])
    self.rry.append(enc[3])
    self.lfline.setData(self.timeline, self.lfy)
    self.lrline.setData(self.timeline, self.lry)
    self.rfline.setData(self.timeline, self.rfy)
    self.rrline.setData(self.timeline, self.rry)
    self.dist.setText("Distance: %0.3f" % dist)

def closeEvent(self, event):
    self.thread.stop()
    event.accept()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    win = Rover()
    win.show()
    app.exec()
```

Laptop Program

