Summary

The series of programs in this folder consist of the **MController.py**, **SetSpeeds.py**, **ImageConverter.py**, **Lab3Display.py**, **Lab4Grid.py**, and **Lab.py** and **LabTasks.py** files for each of the lab tasks.

MController.py is a program that can/should be run alongside the other programs to troubleshoot issues and display information from the Robot's sensors, encoders, and can manually stop the robot in the event of the Lab programs crashing.

SetSpeeds.py creates a **ROS publisher** for sending speed messages to the robot and holds functions for setting linear and angular speeds.

ImageConverter.py file creates a **ROS subscriber** for the video frames sent by the **RaspiCam** node running on the Robot, and converts the frames to **opencv** images.

Lab3Display.py holds the GUI for displaying the information from the Triangulation Lab task.

The **Lab4Grid.py** holds the GUI for displaying the location of the robot and its direction in the grid maze.

The **Lab.py** files consist of a GUI for selecting the Lab Tasks, and call upon the corresponding functions from the **LabTasks.py** file. The students generally should not have to modify the **Lab.py** files, but will use them to select the Task functions that they will write.

The function ***rate.sleep()*** should be called almost always at the end of a while loop. This is because if there are communications with the robot inside of this loop, there is a risk of sending out an extremely large amount of messages through **ROS** which results in the robot handling them one at a time. This could lead to the robot being flooded in messages for setting the robot's speed, which will essentially lock the robot at that speed until the controller program on the robot is turned off. Sleeping for the rate at which the robot receives/sends messages will prevent this from happening.

"*while True:*" loops should be replaced with

while not rospy.is_shutdown():

This allows the program to terminate properly if it is killed via **ctrl-c.**

Tkinter is used for the GUI. It is suggested that the students have a light understanding on how it works. For instance

root = Tk()
app = Application(master=root)

```
app.mainloop()
```

Is equivalent to.

```
root = Tk()
app = Application(master=root)
while not rospy.is_shutdown():
        try:
            app.update_idletasks()
            app.update()
            rate.sleep()
        except Exception as e:
            break
```

If the function ***mainloop()*** is called, the GUI is essentially in a while loop updating the GUI and performing functions called by the GUI until the window is closed. The **Tkinter** functions ***update_idletasks()*** and ***update()*** in a while loop essentially have the same functionality. If the user would like to have a GUI running, alongside other tasks(without using multi-threading), the user can make frequent calls to these functions, however, the GUI will only respond to input upon execution of these update functions.

Running the GUI on the main thread and running the lab tasks on another thread is the ideal solution, however, the code complexity increases. There are sample implementations for the few lab tasks that require a GUI during execution of the task. It is important to note that the **MController** program can always be run concurrently with a Task program as it is a separate process.

Calibrating the robot
In order to calibrate the robot, two terminals should be opened on the robot with ***ROS_IP***, ***ROS_SERVER_URI,*** and ***ROS_MASTER_URI*** variables set (see installation documentation). One one terminal run,

```
roscore
```

On the other terminal run

```
rosrun pi3_robot_2019 MyServos.py
```

When calibration is complete, ***roscore*** can be stopped, and the controller program can be started.

```
roslaunch pi3_robot_2019 pi3_robot_2019.launch
```

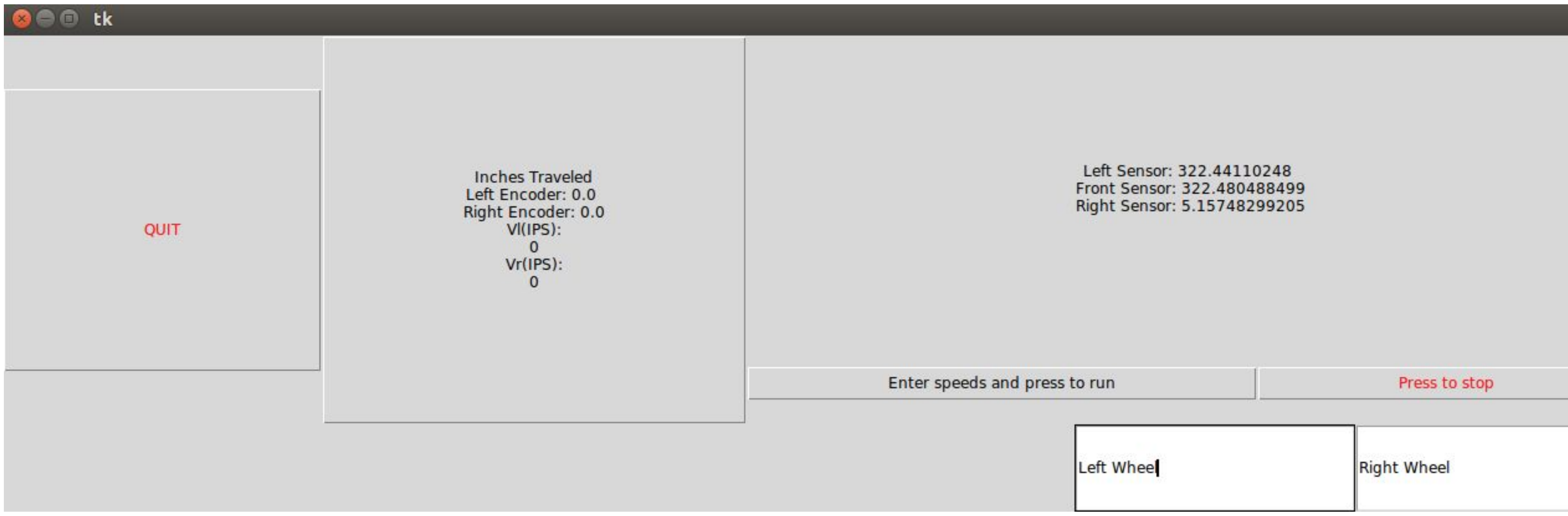
MController

MController.py is a **ROS node** that can be run in order to read the distance traveled, instantaneous speed, sensor readings, and manually control the robot's speed.

The distance travelled can be reset by clicking the encoder widget.

This can be run concurrently with the Lab tasks in order to help pinpoint the issues and observe readings that we would have to print via the terminal otherwise.

MController2.py has additional buttons for turning 90 degrees left, and 90 degrees right. Implementing this turning function could possibly be one of the first lab tasks as it could familiarize the students to working with **Tkinter** if they need to modify the GUI for any reason.



```
Inches Traveled
Left Encoder: 0.0
Right Encoder: 0.0
Vl(IPS):
0
Vr(IPS):
0
```

```
Left Sensor: 322.44110248
Front Sensor: 322.480488499
Right Sensor: 5.15748299205
```

Lab1 and Lab1Tasks

Lab1:

Lab1.py consists of the Application class which is the main GUI that will allow the students to select from the lab tasks and additional GUI's used for entering data needed for the Lab Tasks. Pressing a button will set the string variable **function** with a tag that the main function will use to call the tasks function from the ***Lab1Tasks.py*** file.

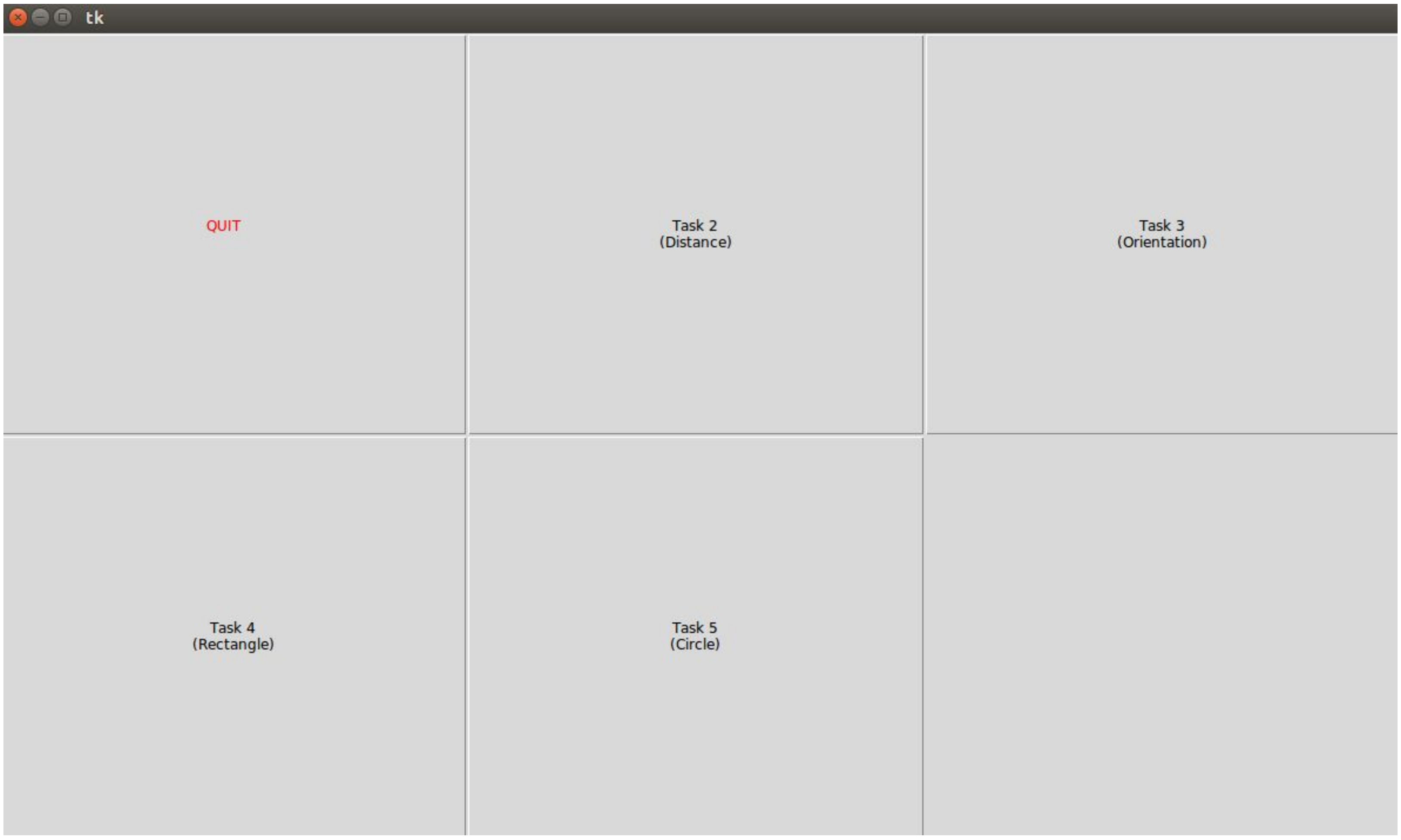
```
root = Tk()
app = Application(master=root)
app.mainloop()
root.destroy()
if(app.function == "Task2"):
        ...
```

```
elif(app.function == "Task3"):
        ...
```

Inside of these if statements, if the Task requires additional information it will create another GUI that will take in the data entry which will be used to call the function from ***Lab1Tasks***.

```
elif(app.function == "Task3"):
        root = Tk()
        app = Orientation_GUI(master=root)
        app.mainloop()
        root.destroy()
        Lab1Tasks.Task3(app.degrees.get(), app.secs.get())
```

When the program terminates it calls ***on_shutdown*** which sets the robots speeds to 0.



Lab1Tasks:

For Lab 1 the **encoder** information is required, so the following statements should be included to receive the Encoder Messages from the Robot

```
from robot_client.srv import GetEncoder
from robot_client.srv import GetEncoderRequest
from robot_client.srv import GetEncoderResponse

rospy.wait_for_service('pi3_robot_2019/r1/get_encoder')
get_encoder = rospy.ServiceProxy('pi3_robot_2019/r1/get_encoder', GetEncoder)
```

This creates a **ServiceProxy** to allow the program to request encoder information from the robot. In order to call the service and retrieve the encoder information ***get_enconder().result*** should be used.

```
encod =  get_encoder().result
l = encod[0]    (left encoder)
r = encod[1]    (right encoder)
```

Because the **MController** program actively displays the encoder information, resetting the tick count on the robot would result in the **MController's encoder widget** resetting as well. Therefore, instead of resetting the encoder values, we take the initial encoder values when starting the program to subtract from the current encoder values when measuring a distance. The controller program on the robot may have to be reset if the robot is used for several hours and is not giving accurate encoder and/or sensor information.

```
encod =  get_encoder().result
lInit = encod[0] # GET INITIAL ENCODER VALUES
rInit = encod[1]
l=0
r=0
while (r-rInit<c and l-lInit<c):
    encod =  get_encoder().result
    l = encod[0]
    r = encod[1]
```

SetSpeeds should be imported to use

```
SetSpeeds.setspeeds(linSpeed,linSpeed)                    (from Task2)
SetSpeeds.setspeedsvw(0,Degrees*math.pi/(180*Seconds)) (from Task3)
```

Lab2 and Lab2Tasks

Lab2:

Lab2.py holds the Application class which, similarly to **Lab1.py**, is the GUI that allows the user to select from the Lab 2 Tasks. There is no data entry required for the Lab 2 Tasks.

Lab2Tasks:
In addition to requiring the **encoders**(see Lab 1) for discrete turns, the **sensors** are also required for Lab 2.

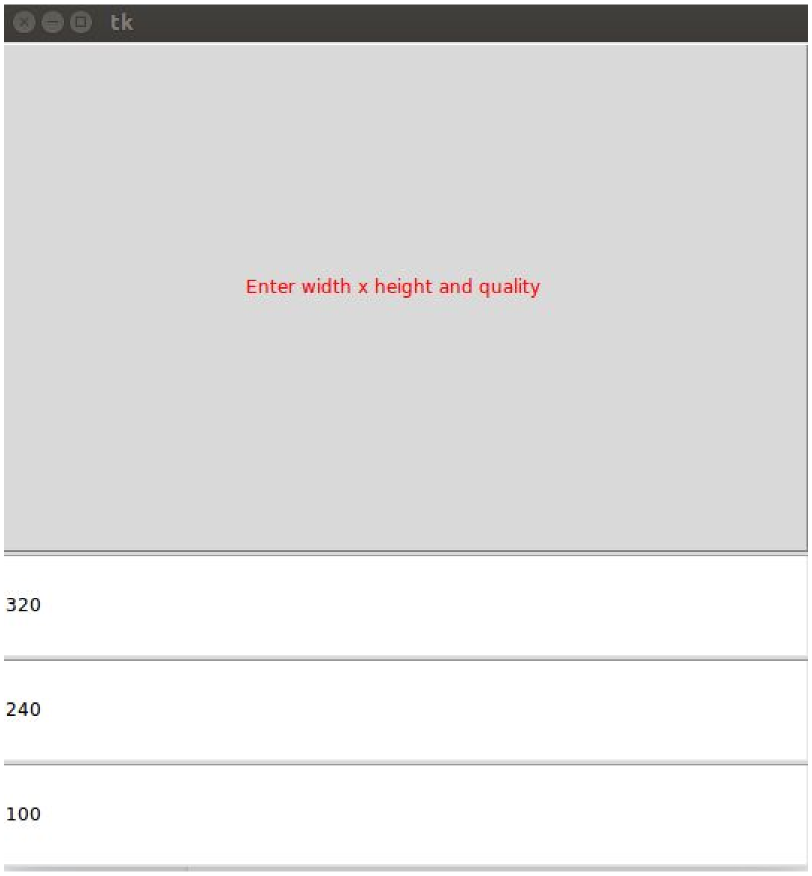
```
from std_msgs.msg import Float32MultiArray
from std_msgs.msg import MultiArrayLayout
from std_msgs.msg import MultiArrayDimension

sense = (0.0, 0.0, 0.0)

def distance_sensors(msg):
    global sense
    sense = msg.data
rospy.Subscriber("/pi3_robot_2019/r1/distance_sensors/d_data",
                        Float32MultiArray, distance_sensors)
```

This creates a **Ros subscriber** to receive the readings from the distance sensors. In order to use the sensor information, the readings must be converted to inches.

```
l = sense[0]*39.3701
f = sense[1]*39.3701
r = sense[2]*39.3701
```

Lab3 and Lab3Tasks

Lab3:
This contains the **Resolution_GUI** class which allows the user to enter in the resolution and quality for the video. I have tested various settings, but have not noticed what the resolution setting does. More information can be found here:
(https://github.com/UbiquityRobotics/raspicam_node)



Enter width x height and quality

320

240

100

The information will be sent to the robot and the camera will be initialized with the specified settings. If the camera is already running it will be restarted with the specified settings, and upon terminating the program the camera will be killed. This information is sent using the **RunFunction** service.

from robot_client.srv import RunFunction
from robot_client.srv import RunFunctionRequest
from robot_client.srv import RunFunctionResponse

rospy.wait_for_service('pi3_robot_2019/r1/run_function')
run_function = rospy.ServiceProxy('pi3_robot_2019/r1/run_function', RunFunction)

Run_function can then be used as
result = run_function("init_camera",[str(res.w.get()),str(res.w.get()),str(res.q.get())])
print(result)

If the camera is successfully opened it should print ‘ok’ in the terminal.
When the program terminates, it will shut off the camera with the following lines.

result = run_function("init_camera",["kill"])
print(result)

Lab3Tasks:
Lab 3 requires the **sensors**(see lab 2), the **ImageConverter**, **cv2**, and the **Lab3Display**.

```
import cv2
import Lab3Display
import ImageConverter
from threading import Thread, Lock
import threading
```

Inside of the tasks, an **ImageConverter** should be created and its variable mutex should be acquired. **Mutex** acts as a **Lock**, and when the **ImageConverter** receives a new frame, it will release the lock. The lock should be acquired after creating it so that once it reaches the while loop, it cannot continue until the **ImageConverter** receives a frame.

```
ic = ImageConverter.image_converter()
ic.mutex.acquire()
        ...
while not rospy.is_shutdown():
      ic.mutex.acquire()
        …
```

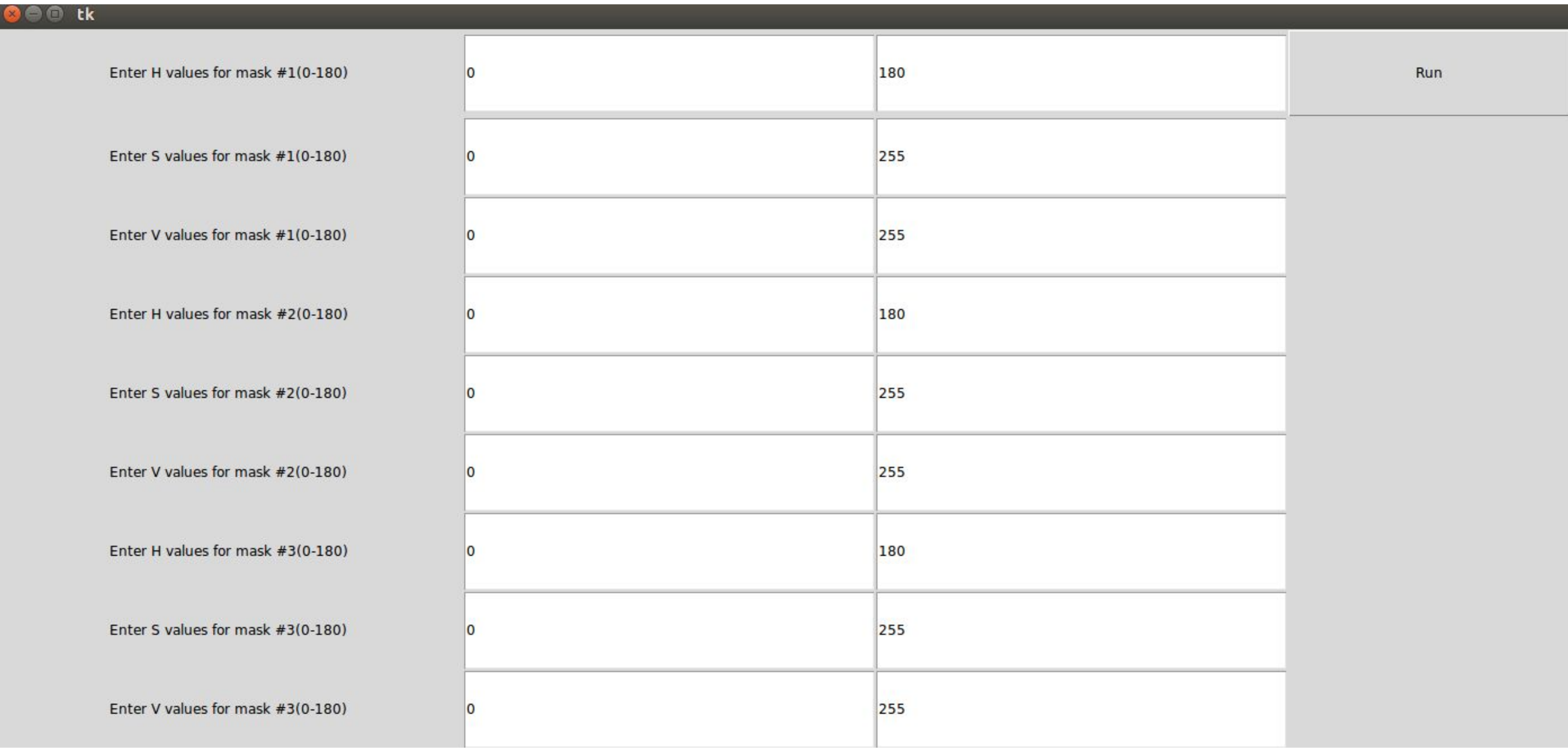
There are other lines of code(from previous semesters) that are used, such as creating a blob detector and reading from the params.yaml file. All of these lines have comments explaining what they do. The only thing that has changed is acquiring the lock at the beginning of the while loop and using

```
frame = ic.cv_image
```

Instead of

```
frame = camera.read()
```

Lab3Display:
Lab3Display contains two GUI classes for the **Triangulation exercise**: **Entry_GUI** and **Application**. **Entry_GUI** should be called at the beginning of Task 3, and allows the user to enter the **HSV** values for the three masks needed for the Triangulation exercise.



Enter H values for mask #1(0-180)	0	180	Run
Enter S values for mask #1(0-180)	0	255	
Enter V values for mask #1(0-180)	0	255	
Enter H values for mask #2(0-180)	0	180	
Enter S values for mask #2(0-180)	0	255	
Enter V values for mask #2(0-180)	0	255	
Enter H values for mask #3(0-180)	0	180	
Enter S values for mask #3(0-180)	0	255	
Enter V values for mask #3(0-180)	0	255	

The **HSV** values can be retrieved with the following for loop.

```
minH=[0,0,0]; minS=[0,0,0]; minV=[0,0,0]; maxH=[0,0,0]; maxS=[0,0,0]; maxV=[0,0,0]
  for i in range (3):
    minH[i] = entry.minH[i].get(); minS[i] = entry.minS[i].get(); minV[i] = entry.minV[i].get();
    maxH[i] = entry.maxH[i].get(); maxS[i] = entry.maxS[i].get(); maxV[i] = entry.maxV[i].get();
```

The **Application** class allows the user to display the values by updating the following variables and calling the function ***updateDisplay()***

```
R1=0
R2=0
R3=0
loc =[0,0]
```

Task 3:
Task 3 can be implemented on a **single** thread or with **multiple** threads. If using a **single** thread, at the end of the while loop call

```
display.update_idletasks()
display.update()
```

And between while loops, call

```
display.R1 = r1
```

display.updateDisplay()
display.update_idletasks()
display.update()

These variables should be set as they are identified, and calling ***updateDisplay()*** will set the updated values to the display. ***update_idletasks()*** and ***display.update()*** update the GUI. Otherwise the GUI would not update and would not respond to input.

The issue with running both the GUI and task on a single thread, is that the GUI will only respond when ***update_idletasks()*** and ***display.update()*** are called. If the user wants to exit the program by clicking the exit button on the GUI, it will respond the next time these functions are called. Therefore, it is ideal to run the GUI on the **main** thread, and run the actual task on a **seperate** thread. The version of **opencv** installed with **ROS Kinetic** must also be run on the main thread, so the GUI and videos will be displayed on the main thread, with the lab task and processing running on the separate thread.

The ***Task3Main()*** function has been provided to display the GUI and video frames. There are a few global variables required to tell the main thread when to display a new video frame or when to stop the program from a separate thread.

mask = None; frame_with_keypoints = None; newFrame = False
quitFlag=0; finished = False

In order to start the thread, the following lines must be called.

t3Thread = threading.Thread(target=Task3, args=(display,minH, minS, minV, maxH, maxS, maxV,))
t3Thread.start()

The following lines of code are used in the while loop running on the main thread. If the flag ***newFrame*** is true, the new frames will be displayed and the flag will be set to false. If the flag ***finished*** is true, the while loop will terminate. Outside of the while loop the flag ***quitFlag*** is set to true.
Note: Since *mask*, *frame_with_keypoints,* and *newFrame* are modified in both threads, we surround them in a lock called mutex(not to get confused with the ImageConverters variable called mutex) to avoid race condition.

```
while not rospy.is_shutdown():
    try:
        display.updateDisplay()
        display.update_idletasks()
        display.update()
        if(finished==True):
                cv2.destroyAllWindows()
                break
            elif(newFrame==True):
                 mutex.acquire()
                cv2.imshow(WINDOW1, mask)
                cv2.imshow(WINDOW2, frame_with_keypoints)
                c = cv2.waitKey(1)
                newFrame=false
                mutex.release()

            rate.sleep()
        except Exception as e:
            break
quitFlag=1
```

Task3 implemented on a seperate thread is very similar to the single thread implementation except for a few changes. Since we are calling ***cv2.imshow(WINDOW1, mask)*** and ***cv2.imshow(WINDOW2, frame_with_keypoints)*** on the main thread, they will not be called on the separate thread. Instead ***mask*** and ***frame_with_keypoints*** will be global variables, and ***newFrame=True*** will be called to notify the main thread to display a new frame.

```
global finished; global mask; global frame_with_keypoints; global newFrame;
        ...
        while quitFlag==0:
                ic.mutex.acquire()
                    ...
                frame = ic.cv_image
                    ...

                mutex.acquire()
                mask = cv2.inRange(frame_hsv, (minH[2], minS[2], minV[2]), (maxH[2], maxS[2],
maxV[2]))
                keypoints = detector.detect(mask)
                ...
                newFrame=True
                mutex.release()



                    ...
```

Note that Instead of

```
        while not rospy_is_shutdown():
```

The while loop is

while quitFlag==0:

This is so that if the **main** thread is terminated and exits the while loop, the separate thread will stop running as well.

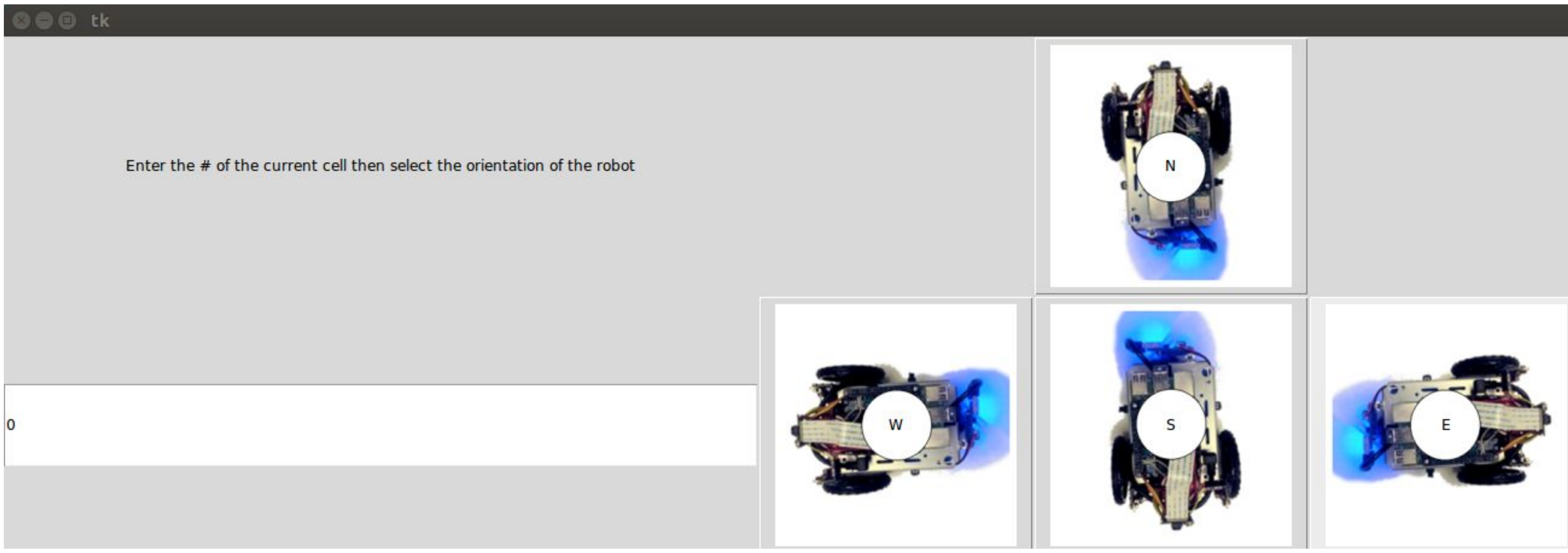
At the end of the thread running Task3, the flag ***finished*** is set to true. This tells the main thread to close the opencv windows since now new frames will be displayed.

finished = True

Lab4 and Lab4Tasks

Lab4:

Lab4.py contains the Application class which is the GUI for selecting between the Lab Tasks and the Entry_GUI class which allows the student to enter the current location of the robot and its orientation. This information is then passed to the corresponding Lab 4 Task function.



Enter the # of the current cell then select the orientation of the robot

0

Lab4Grid:

Lab4 uses the sensors used in previous labs and the **Lab 4 Grid GUI**. The Grid GUI has 3 variables that should be set

maze
curcell
dir

maze is a 16x6 list where each row contains whether the tile has been **visited**, whether it has a **North** wall, whether it has a **East** wall, whether it has a **South** wall, whether it has a **West** wall, the cost to get to that tile in path planning. The 6th entry in each row is not used by the GUI, so technically a 16x5 list would work as well. A 1 indicates that the tile has been visited or that a wall is at the indicated direction.

[Visited, N, E, S, W, Cost]

A tile that has been visited and has a North and South wall would be marked as

[1, 1, 0, 1, 0, -1]

For example, a maze that has not been traveled should appear as such.

```
maze = [[0, 1, 0, 0, 1, -1], [0, 1, 0, 0, 0, -1], [0, 1, 0, 0, 0,-1],[0, 1, 1, 0, 0, -1],
        [0, 0, 0, 0, 1, -1], [0, 0, 0, 0, 0, -1], [0, 0, 0, 0, 0, -1],[0, 0, 1, 0, 0, -1],
        [0, 0, 0, 0, 1, -1], [0, 0, 0, 0, 0, -1], [0, 0, 0, 0, 0, -1],[0, 0, 1, 0, 0, -1],
        [0, 0, 0, 1, 1, -1], [0, 0, 0, 1, 0, -1], [0, 0, 0, 1, 0, -1],[0, 0, 1, 1, 0, -1]]
```

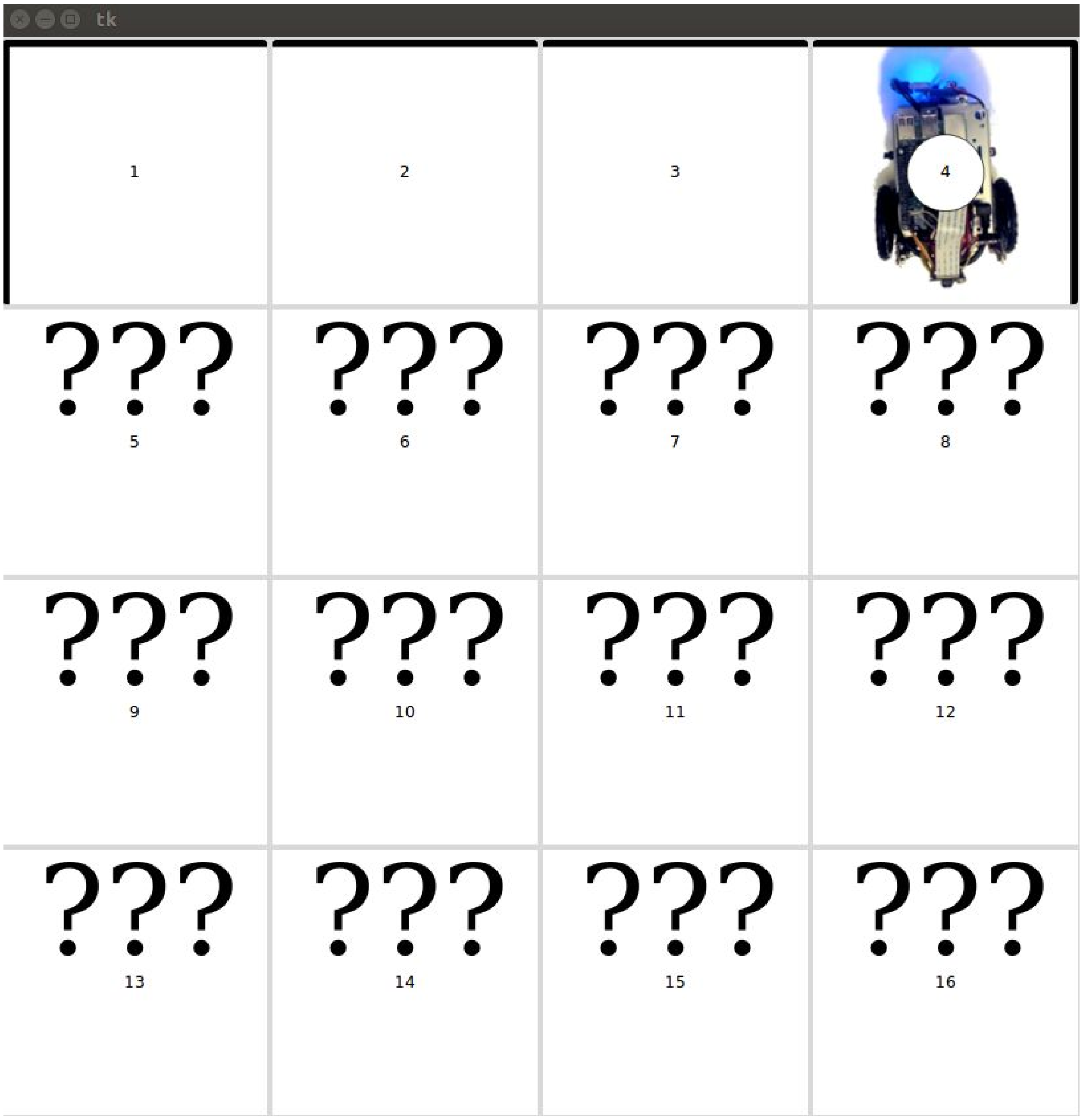
The **curcell** variable contains the index of the list entry where the robot is currently at(0-15). The **dir** variable contains a character indicating the direction the robot is facing('N','E','S','W').

The GUI works by creating a string out of the integer values for the walls and adding the direction then creating a GUI tile using a PNG from the **grid_images** folder.

```
pngName = str(self.maze[i][1]) + str(self.maze[i][2]) +str(self.maze[i][3]) +str(self.maze[i][4])
if(self.curcell ==i):
        pngName += self.dir
pngName+=".png"
```

When updating the GUI from the Task function, there would be code similar to the following lines

```
gui.maze = maze1
gui.curcell = curcell
gui.dir = dir_Enum[dir_Num]
gui.updateGrid()
gui.update_idletasks()
gui.update()
```



Lab4Tasks:
Lab 4 can be implemented much like Lab 3 Task 3, using either a **single** thread or **multiple** threads. The issue with using a **single** thread for Lab 4, is that if there is a while loop containing several functions such as move cell or mark cell, and the GUI is updated at the end of this while loop, there would be several seconds that the GUI is unresponsive. Therefore, it is far more ideal to run the Tasks for Lab 3 using **multiple** threads.

Main Thread

while not rospy.is_shutdown(): #GUI must be ran on main thread. Task2 ran on seperate thread.

```
        try:
            if(cellMoved==1):
                gui.updateGrid()
                cellMoved=0
            gui.update_idletasks()
            gui.update()
            rate.sleep()
        except Exception as e:
            break
quitFlag=1
```

Task 2 Thread

```
while quitFlag == 0:
        gui.maze = maze1
        gui.curcell = curcell
        gui.dir = dir_Enum[dir_Num]
        cellMoved=1
            … logic for turning and moving cells, marking cells etc...
        rospy.sleep(1)
```

A single threaded implementation would look like

```
while not rospy.is_shutdown():
        gui.maze = maze1
        gui.curcell = curcell
        gui.dir = dir_Enum[dir_Num]
        gui.updateGrid()
        gui.update_idletasks()
        gui.update()
        … logic for turning and moving cells, marking cells etc...
        rospy.sleep(1)
```

In the **multithreaded** implementation, the program would immediately terminate upon clicking the exit button on the GUI, whereas the single thread would have to complete the current action, which could be moving to another cell, or turning.