

# **TurtleBot 3 Burger Final Project Report**

AuE 8930: Autonomy Science and Systems

Spring 2020

Shubham Horane, Chase Gurcan, Manikanda Balaji Venkatesan, Rahil  
Modi, Akshay Hole (Group 6)



## **Preface:**

Over the duration of the course, we have worked on several assignments and homework that have led us up to this final project. This project is a culmination of all our efforts combined.

The knowledge and experience gained from each assignment have been applied in the final project, ranging from developing control programs in python for a TurtleBot simulator in ROS to applying the same code on a real-life TurtleBot 3 Burger. The main highlights of our implementation are image detection and processing with OpenCV, Apriltag detection, wall following, line-following, obstacle avoidance and stop sign detection using a pre-trained object detection model.

## Contents

Preface: .....	2
List of Figures: .....	5
Task 0: Initial setup activities .....	6
Setting up the world file correctly .....	6
Challenges faced with setting up the world in Gazebo: .....	7
1. Adding camera to TurtleBot 3 Burger model in gazebo .....	7
Task 1: Wall following and obstacle avoidance .....	8
Problem Statement: .....	8
Preparation .....	8
Tasks completed: .....	8
Problems and Solutions: .....	8
Code: .....	9
Result: .....	9
Task 2: Line following .....	13
Problem Statement: .....	13
Preparation: .....	13
Tasks completed: .....	13
Problems and Solutions: .....	13
Code: .....	14
Results: .....	14
Task 3: Stop sign detection: .....	16
Problem Statement: .....	16
Preparation: .....	16
Tasks completed: .....	16
Problems and solutions: .....	16
Code: .....	17
Results: .....	17
Task 4: Human tracking through leg detection .....	19
Problem Statement: .....	19
Preparation: .....	19

Tasks completed:.....	19
Problems and Solutions:.....	19
Code: .....	20
Results: .....	20
Task 5: Code integration .....	22
Conclusion: .....	24
Appendix.....	26
Camera implementation on the TurtleBot Burger model in Gazebo simulator environment:..	26
Final integrated code in python:.....	27
Launch file used: .....	34

## List of Figures:

Figure 1. Final word correctly loaded in Gazebo .....	6
Figure 2: Wall following image before turn .....	9
Figure 3: Wall following image after turn .....	10
Figure 4: Obstacle avoidance image at start of section.....	10
Figure 5: Obstacle avoidance image right before end of section.....	11
Figure 6 Hector SLAM output using Rviz.....	12
Figure 7: Problems faced with camera height in line following.....	13
Figure 8: Robot hitting nearby stop sign object due to inaccurate line following.....	14
Figure 9: Line following image beginning of line .....	15
Figure 10: Line following image middle of line.....	15
Figure 11: Stop sign detection .....	18
Figure 12: Human leg tracking in Gazebo and Rviz.....	21
Figure 13: The robot loses the human in instances of sharp turning .....	21

## Task 0: Initial setup activities

### Setting up the world file correctly

The organization of each file was crucial to the success of our group. This involved ensuring the proper set up of the models and affiliated files. This task involved setting up the models and world files correctly in the catkin workspace. The aue final world and the person sim packages were added to the workspace and their relevant model and world files were added to the turtlebot3 simulation package before running catkin\_make to compile these packages.

The command `roslaunch turtlebot3_auefinals_pkg final_launch.launch` command launches the below world along with all the dependant packages:

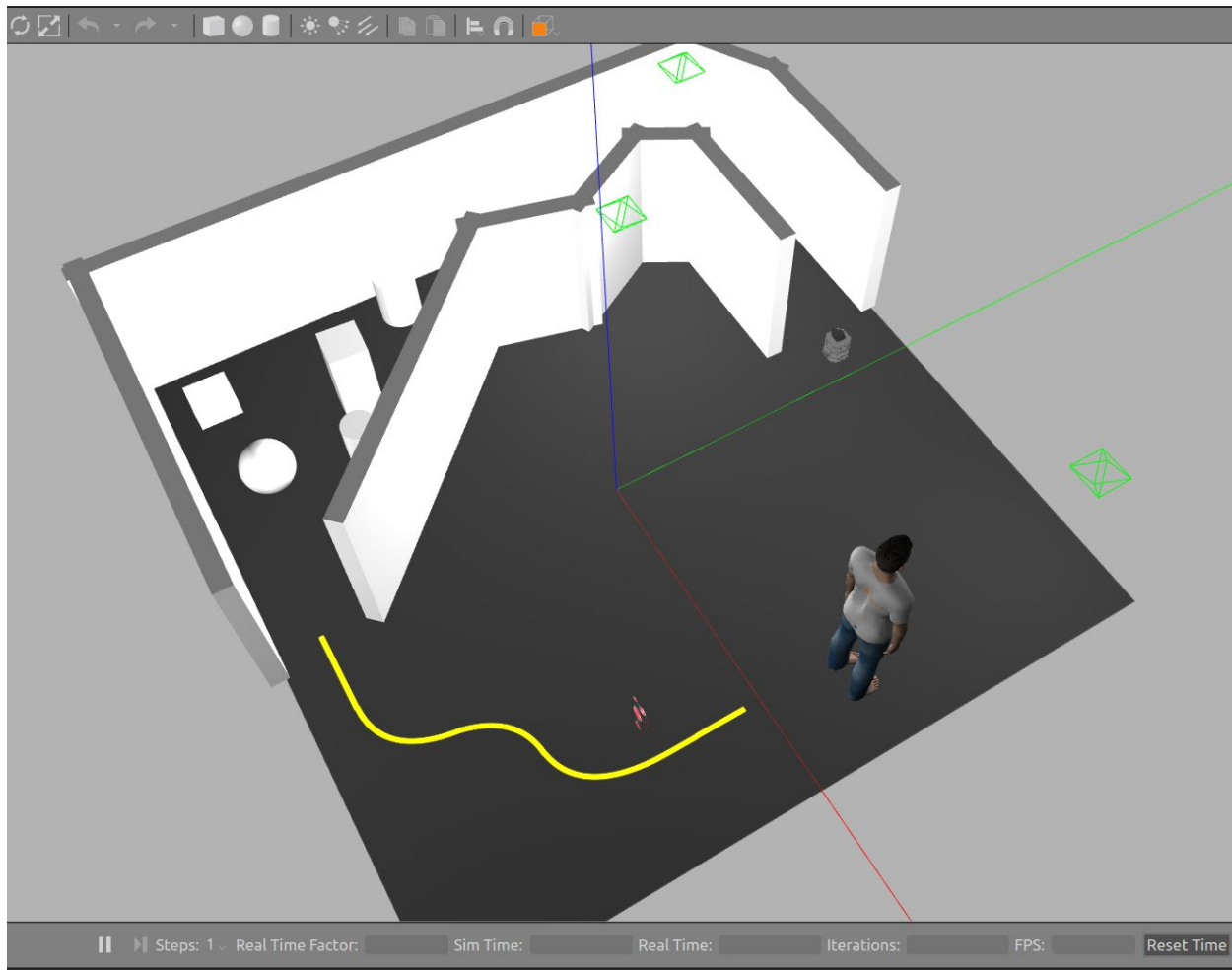


Figure 1. Final word correctly loaded in Gazebo

## Challenges faced while setting up the world in Gazebo:

1. Adding a camera to the TurtleBot 3 Burger model in gazebo

A camera configuration section similar to that of the one used in the .xacro files of the waffle\_pi model in the turtlebot3 description files was added to the .xacro files for the burger model. This allowed us to use camera input with the burger model in gazebo.

Changes were made to the following files:

1. Filename: “turtlebot3\_burger.gazebo.xacro” ( changes included in [Appendix section 1](#))
2. File name: “turtlebot3\_burger.urdf.xacro” (changes included in [Appendix section 1](#))

2. Loading up the person correctly in the final world

The model and world files were added to the main folder of the turtlebot3 package as well as the auefinals package. This helped the world load up correctly.

## Task 1: Wall following and obstacle avoidance

### Problem Statement:

The TurtleBot starts here. It must successfully follow the wall and avoid the obstacles until it reaches the yellow line.

### Preparation

The code developed during the submission for Assignment 4 was modified and implemented for this task. The SLAM packages hector slam and g-mapping introduced as part of course handout 6 were considered.

### Tasks completed:

1. Manipulating laser scanner data for navigation.
2. Implementing P-controller in Python to maintain a constant distance from the bot to the wall.
3. To avoid a collision in the narrow corridor and avoid obstacles on the way to the line following area.
4. To start a SLAM node and keep it running till the end of the simulation. The SLAM process must be visualized using Rviz.

### Problems and Solutions:

Problem 1: Bot not remaining at a set distance from the wall and eventually crashing with the wall at times.

Solution: Re-tuning of the P-controller to adjust the performance according to the area of the new world.

Problem 2: Bot failing to successfully navigate through all obstacles and reach the beginning of the line following section.

Solution: Adjusting the azimuth of the cone angles in the left direction and the right direction.

Problem 3: The bot was not maintaining a middle line between both walls.

Solution: Take the minimum measurement (closest area of an obstacle to the bot) from both the left and right-side value, add them and divide by two to get the middle.

Problem 4: Selection of appropriate SLAM package

Solution: After having tried the g-mapping SLAM package as well as the hector SLAM package on the real TurtleBot 3 Burger as part of the class exercise for handout 6, we observed in Rviz that the g-mapping function was slower than hector mapping. Hector mapping was able to create a map of the environment much faster at the normal running speed of the robot.



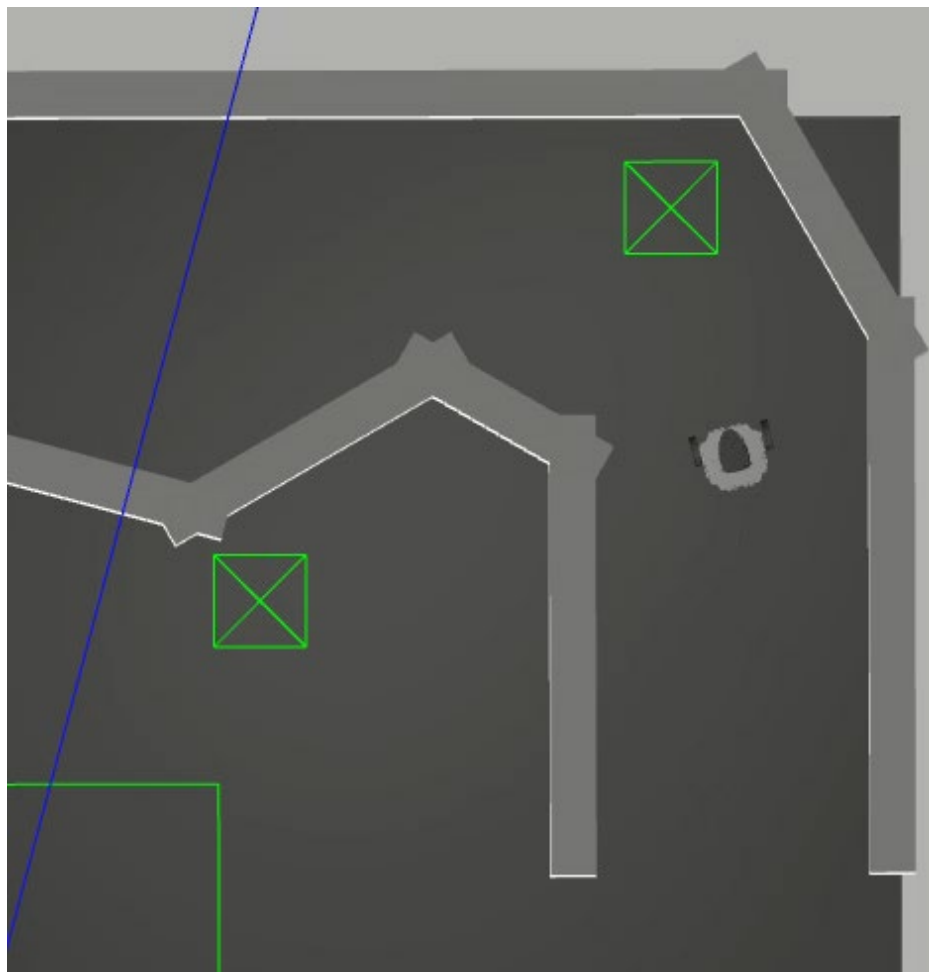
**Code:**

The final code for this functionality was written in the form of a callback function for the laser scanner data and for the proportional controller implementation. The python code can be found in [Appendix section 2](#).

The launch file for Hector slam was added to the main launch file. The launch file is included in [Appendix section 3](#).

**Result:**

After applying the above solutions, we were able to obtain good results in the wall following and obstacle avoidance areas. The below images show the top-down view of the bot moving through the final world while maintaining a perfect distance from both the walls.



*Figure 2: Wall following image before the turn*

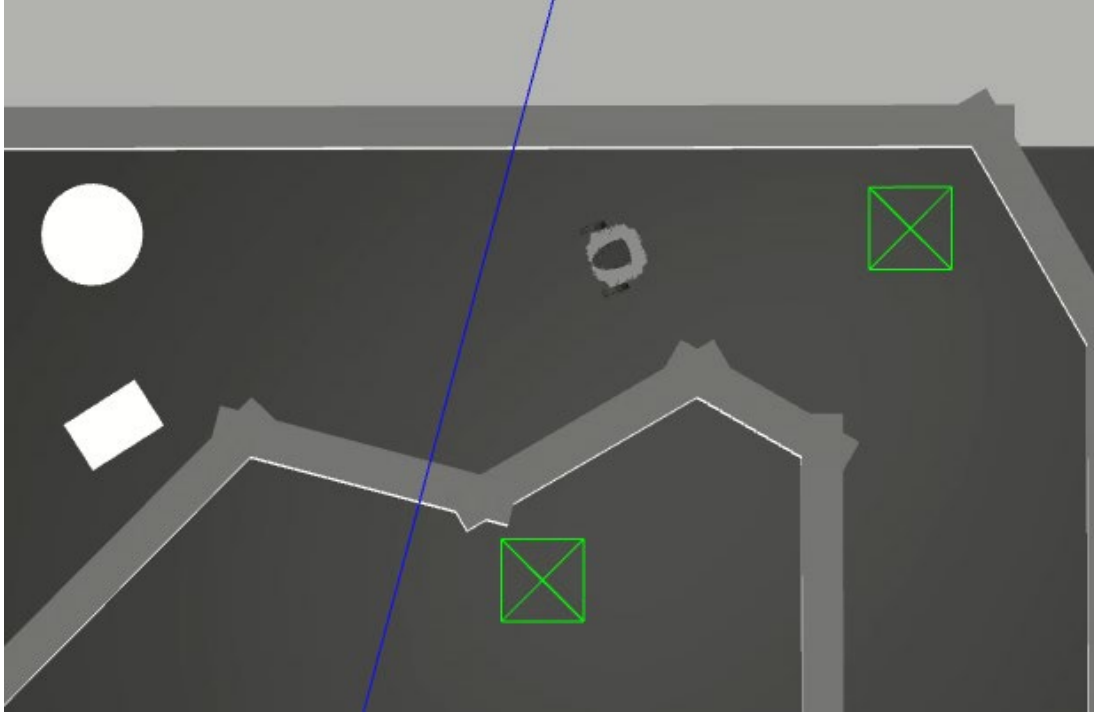


Figure 3: Wall following image after the turn

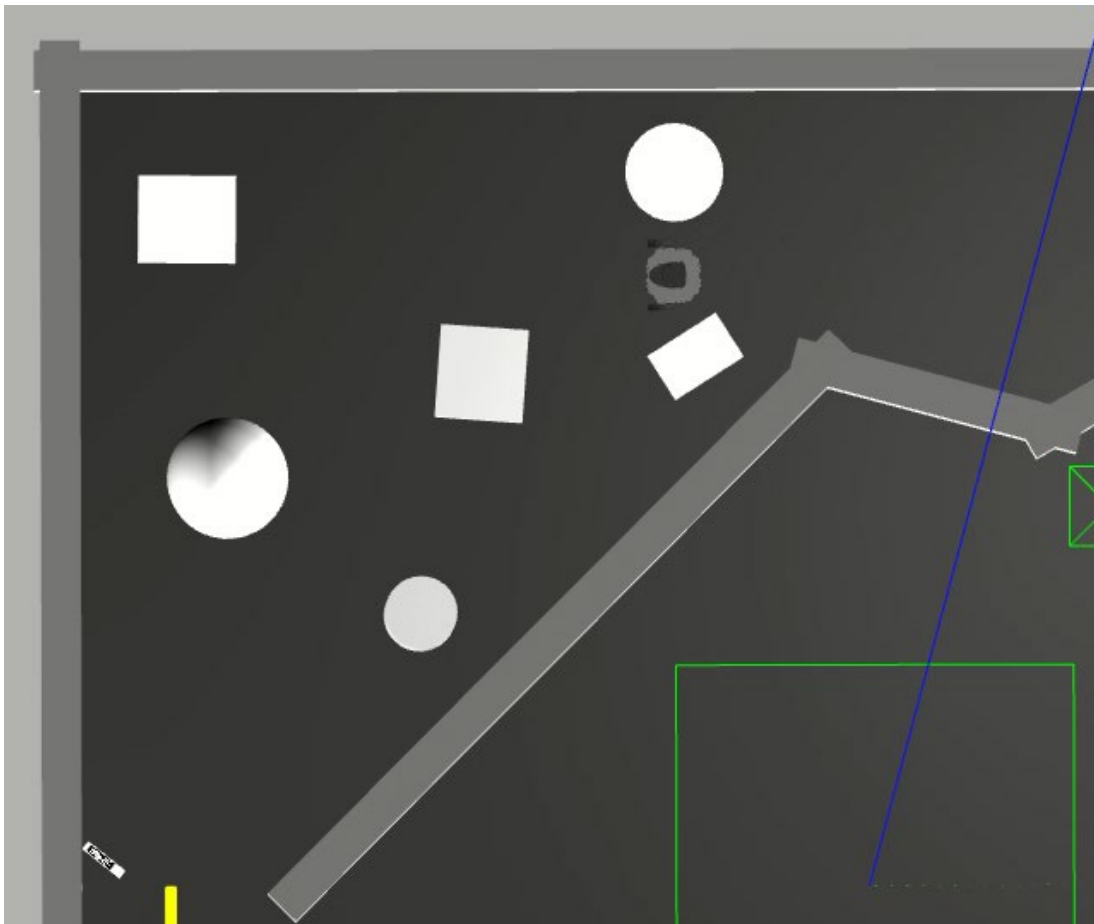
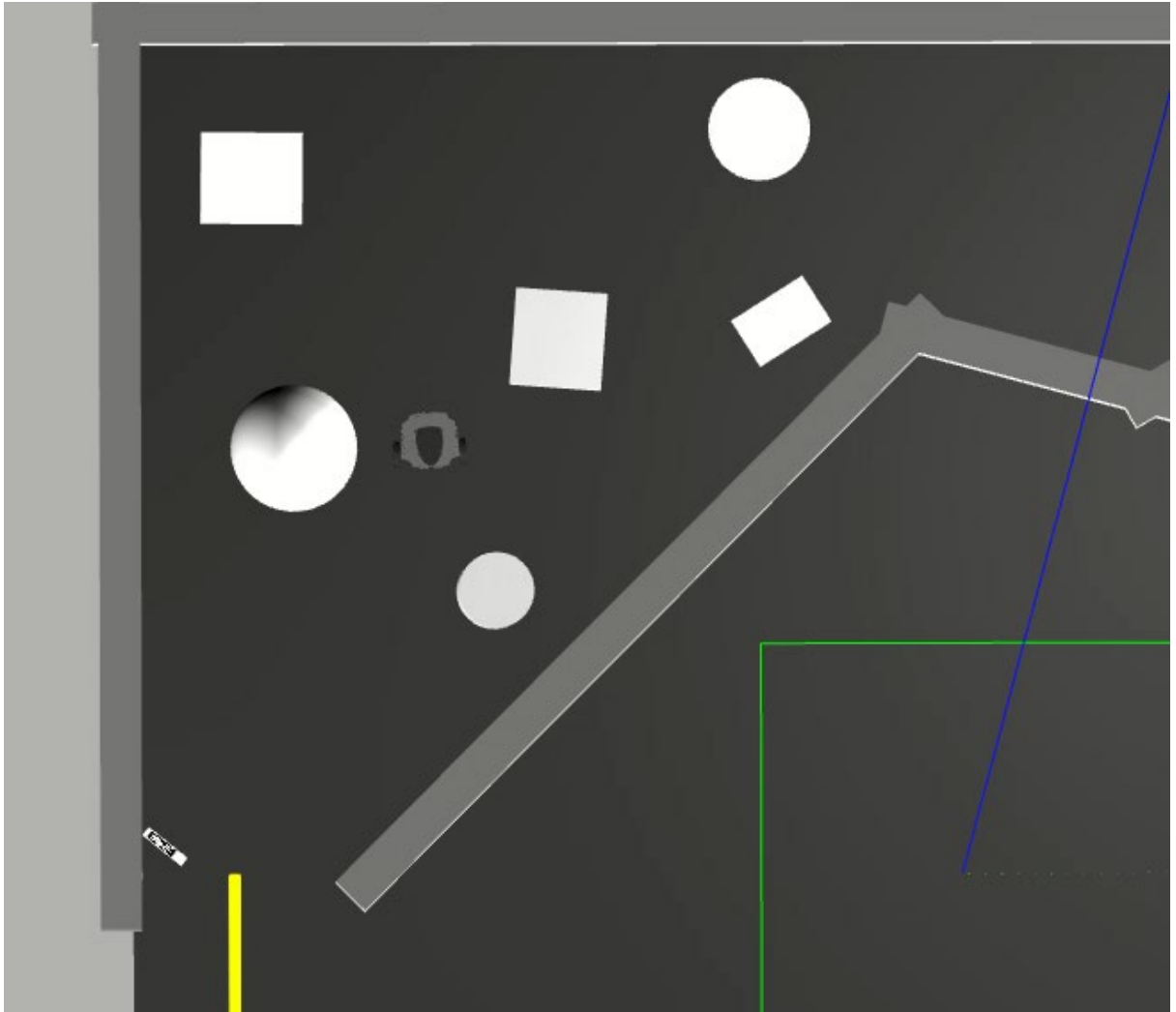


Figure 4: Obstacle avoidance image at the start of section



*Figure 5: Obstacle avoidance image right before the end of section*



*Figure 6 Hector SLAM output using Rviz*

Here we can see the bottom left corner looks incomplete but due to the lack of wall/obstacle, the LiDAR does not record anything there. The same theory applies to the bottom right corner as well.

## Task 2: Line following

### Problem Statement:

The TurtleBot must successfully identify and follow the yellow line.

### Preparation:

- Install OpenCV package in ROS workspace
- Tune and utilize line tracking code from assignment 5

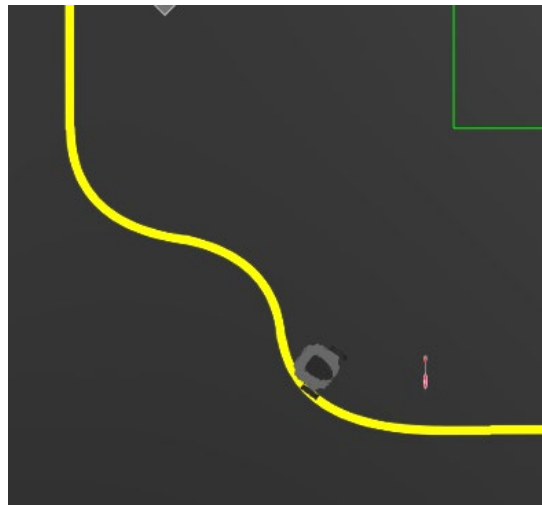
### Tasks completed:

1. Manipulating image data by using a mask to identify the lane center and ensure that the bot will stay as close to this center until completion.
2. Implement a P-controller to keep the bot following the center of the line.

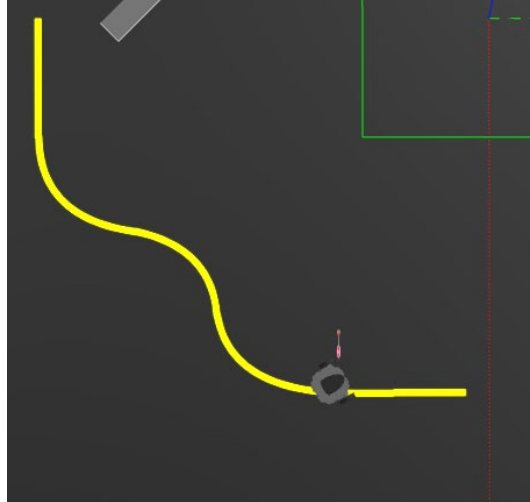
### Problems and Solutions:

Problem 1: Bot was not able to follow the line and was picking lane center much ahead of its position.

Solution: The mask position was adjusted to the lowest possible position in the image to allow the bot to take a closer reference for its P-controller. Another solution for this was to lower the camera position on the robot model in its configuration files. Even the lowest position on the mask did not solve the problem completely, and therefore we chose to lower the camera. After both solutions were applied, the robot was able to detect the line correctly.



*Figure 7: Problems faced with camera height in line following*



*Figure 8: Robot hitting nearby stop sign object due to inaccurate line following*

**Problem 2:** Bot turning too much when it detected high curvature areas.

**Solution:** This was also due to the height of the camera. But reducing the height did not solve this problem entirely, so we reduced the proportional gain value to obtain a better performance of the line following code.

**Problem 3:** All group members are working in Ubuntu 16.04 and ROS Kinetic which uses OpenCV3 but OpenCV2 is required for this task.

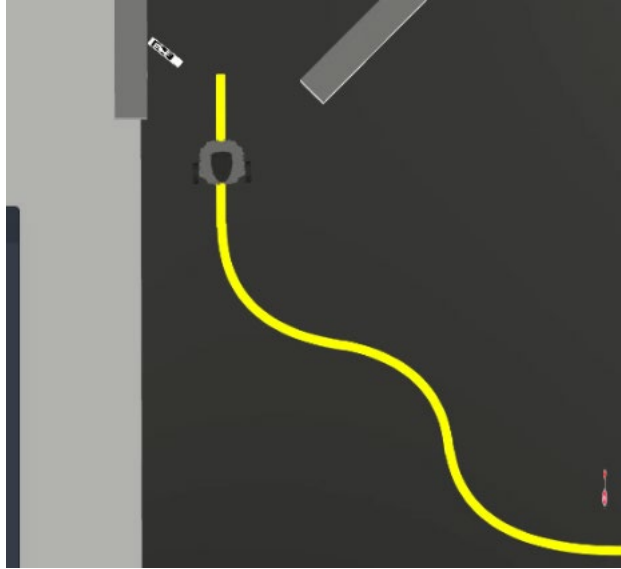
**Solution:** Utilize CVBridge to access OpenCV2 for our camera

### **Code:**

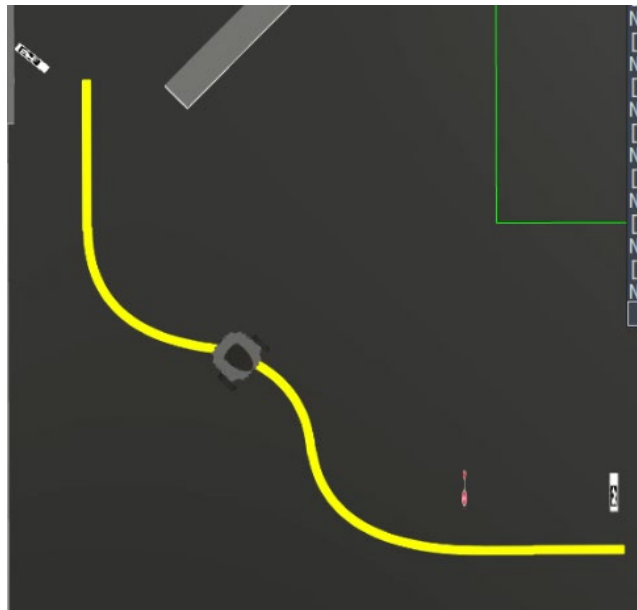
The final code for this section was written as a callback function for the camera sensor on the robot. The image processing and subsequent line following function were included in the same function. The integrated code can be found in [Appendix section 2](#).

### **Results:**

The robot was able to start line following when the yellow line came into the field of view of the extreme lower section of the camera. After adjusting the camera and tuning the P-controller, the performance of the line following functionality was up to the mark. The speed of the robot was set at 1.2 units.



*Figure 9: Accurate line following image beginning of the line*



*Figure 10: Accurate line following image middle of the line*

### Task 3: Stop sign detection:

#### Problem Statement:

This task involves recognizing the “Stop” sign by the bot and stopping for 3 seconds before it moves on. This is done using the visual data stream generated by a raspberry-pi camera on the bot and using a pre-trained TinyYolo dataset which will give intimate detection of stop signs via ROS topic. Once the sign has been detected the bot will halt and after 3 seconds it will start to roll. We then ensured that the bot was only to stop one time, as opposed to multiple times from varying times of detection.

#### Preparation:

- Setting up the camera on the TurtleBot Burger model (Steps in [Appendix section 1](#))
- Installing the TinyYolo ROS package mentioned in handout 8.

#### Tasks completed:

1. Manipulating the data published by the darknet-ros node and use it to detect the stop sign and implement the desired behavior.
2. Detect the stop sign using the tiny-yolo package, stop the bot for 3 secs, and resume line following until the end of the line is reached.

#### Problems and solutions:

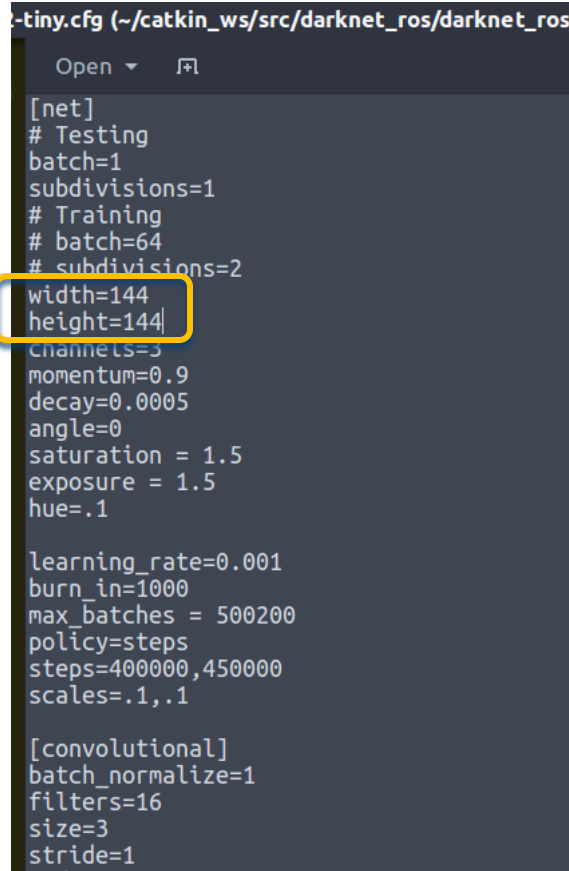
Problem 1: The Tiny-Yolo package was producing extremely low frames per second (FPS: 0.2).

Solution: Changed the frame height and width in the configuration file that was being used by the Tiny-Yolo package. The FPS was increased to approximately 10 times the original FPS (2 FPS). The frame rate was now acceptable and allowed us to work further.

Problem 2: The stop sign was detected multiple times throughout the line following process, causing multiple stops to occur

Solution: We added a flag which would be set to True once the bot detects the stop sign and stops for 3 secs for the first time. And this flag is used as a condition to prevent the bot from stopping during further instances of stop sign detection. More about this integration is mentioned in the [code integration](#) section of this report.





```
tiny.cfg (~/.catkin_ws/src/darknet_ros/darknet_ros)
Open ▾  📄
[net]
# Testing
batch=1
subdivisions=1
# Training
# batch=64
# subdivisions=2
width=144
height=144
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
max_batches = 500200
policy=steps
steps=400000,450000
scales=.1,.1

[convolutional]
batch_normalize=1
filters=16
size=3
stride=1
```

Figure 11: Configuration file of darknet\_ros package

The above configuration file was edited. Default frame height and width were reduced from 416x416 to 144x144.

This increased the FPS from 0.2 to 2.0

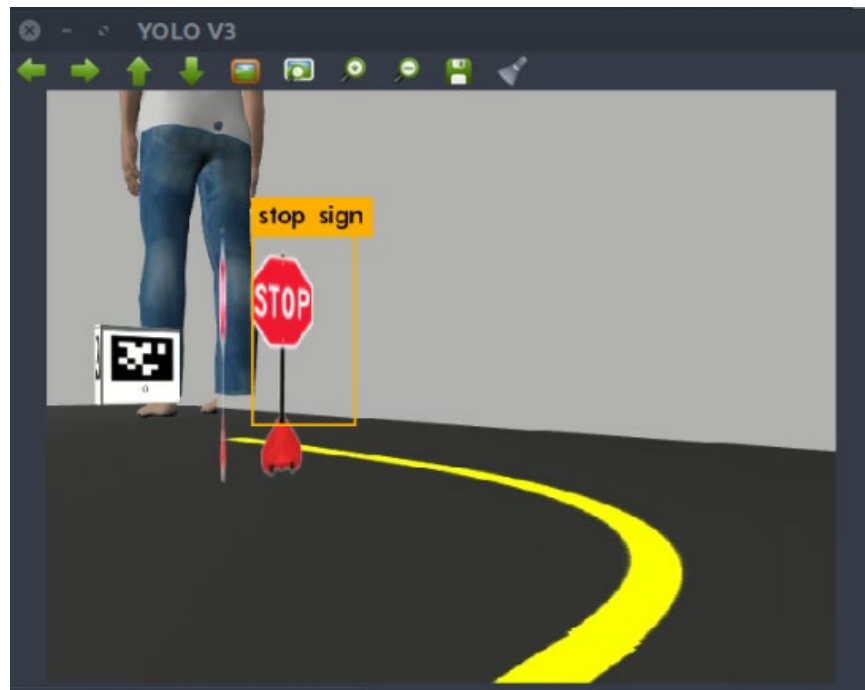
Although this comes at the cost of reduced performance of the tiny-yolo model. Yet it was enough to detect the stop sign at close range.

### Code:

Stop sign detection was done by subscribing to the darknet-ros node that published the bounding boxes topic. This topic contains the name of any object that is successfully detected in each frame. The callback function of this subscriber was used for the detection and handling of its corresponding global flag to achieve the desired functionality. The code for this can be found in the `det_callback` function of [Appendix section 2](#).

### Results:

The Tiny-Yolo package was able to clearly identify the stop sign in a timely manner and the vehicle came to a complete stop for three seconds.



*Figure 12: Stop sign detection*

## Task 4: Human tracking through leg detection

### Problem Statement:

The purpose of this task was to identify the obstacle as a human, track the position of human legs, and follow the human while it is being teleoperated. The first part of the task will be to detect human legs and hence the human's position in the global frame. The second task will be to control the command velocity of the bot while following the human. The tracking depends on the LiDAR data to identify the U shapes created by human legs in scan data of LiDAR.

### Preparation:

- The people detection package provided in handout 8 was used for this purpose
- People's perception - People tracking chapter from "ROS perception in 5 days" book was referred to understand the functioning of the leg detector package.
- This package subscribes to the LiDAR data and uses the "trained\_leg\_detector.yaml" file which contains the configuration parameters for detection purposes.
- The "leg\_detector\_start.launch" file is used to launch the leg detection package. It publishes the coordinates of leg position in the global frame on multiple ROS topics. These topics provide slightly varying results, and a reliability factor as well. The reliability factor indicates the confidence level of the value determined and the covariance of the Kalman filter used in the package.

### Tasks completed:

-Implement pre-trained Deep Learning for the robot to identify human legs.

-Follow the human while it is being teleoperated while maintaining a set distance away from the human.

### Problems and Solutions:

Problem 1: When teleoperating the human, if the human turns too sharply then the bot scan fails to identify both legs and loses track of the human.

Solution: One way of ensuring continuous detection was to move the human slowly and execute wider turns. Another solution was to skip the leg detector package and directly use the laser scanner data to detect the two closest obstacles in the frontal cone of the robot and follow those by trying to keep them at a set distance and in the center of the robot's field of view. However, this solution was also found to be inaccurate at times, and hence we continued with the leg detection package provided with handout 8.

Problem 2: The bot failed to maintain a set distance from the human and kept a constant velocity resulting in collision with the human.

Solution: The frontal lidar scan cone was activated in the leg detection function in parallel with the leg detection and leg following task. The robot was programmed to stop 0.3 units behind any obstacle is detected. Hence, the bot would stop moving if the human came within 0.3 units of the human.

Problem 3: The angular orientation feedback information of the robot was available in quaternion form whereas the controller required the yaw information to calculate the angular error.

Solution: The quaternion orientation, that gives the orientation information of the robot in x,y,z,w form, was transformed into the roll, pitch and yaw (R-P-Y) form and the yaw value was used. The below command was used for this:

```
"tf.transformations.euler_from_quaternion(quaternion)"
```

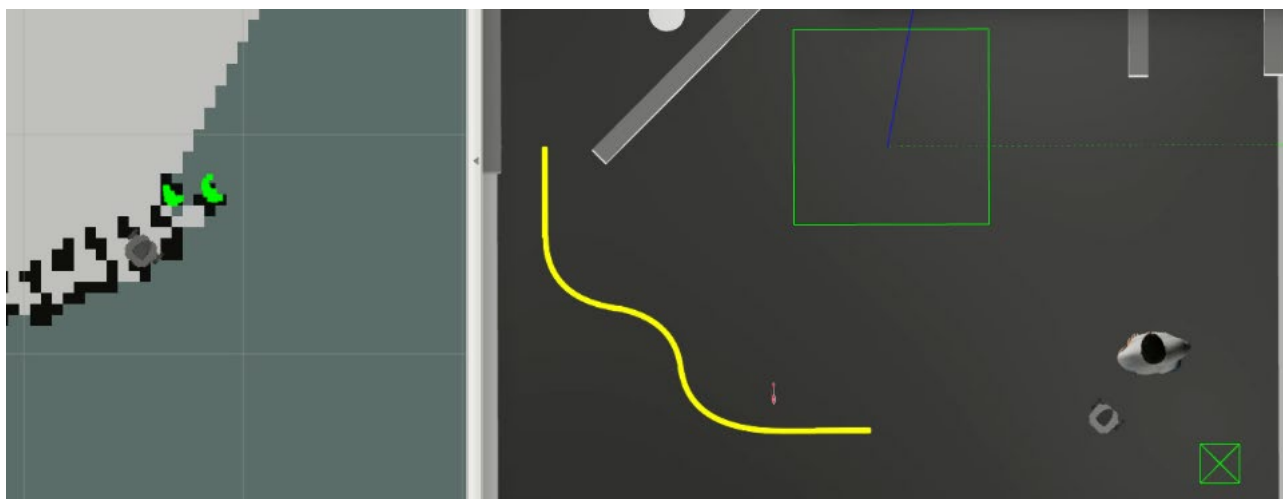
The output of this command gives a 1D array with the R-P-Y values. The quaternion given as input to this command must be in a 1D array.

### **Code:**

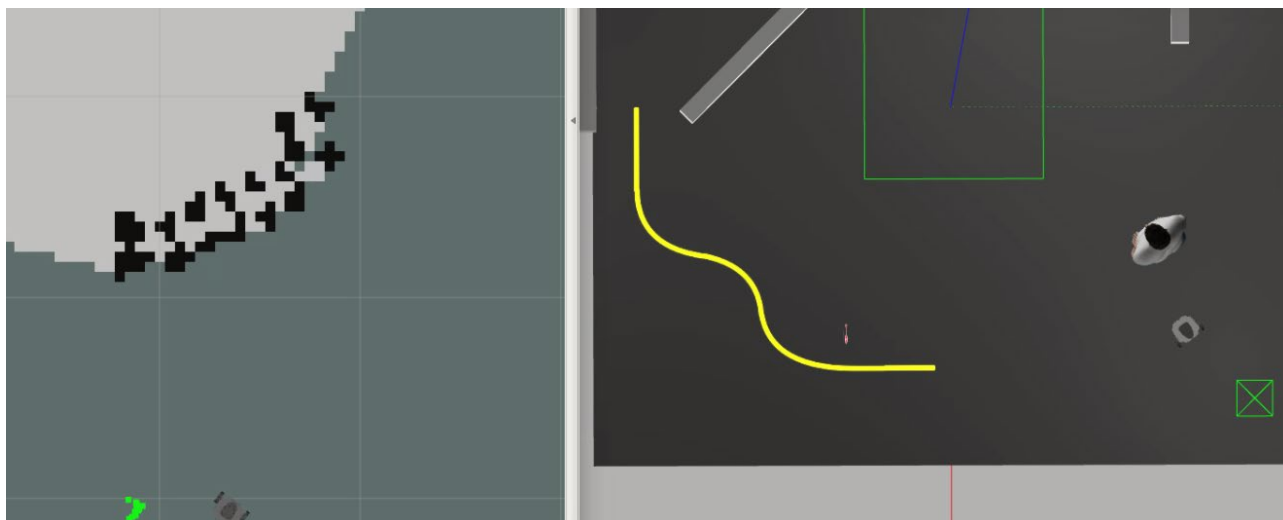
For this task, we implemented two subscribers in our code for the leg detection functionality. The topics `"/people_tracker_measurements"` and `"/to_pose_array"` were subscribed to and pointed to the same callback function in the code. Only one subscriber was uncommented at one time so that both would not run at the same time and cause the code to malfunction. The callback function took data from whichever topic was uncommented and implemented linear and angular proportional controllers to track the human. The feedbacks to the controllers were the actual yaw and position coordinates of the robot. These were available from the `/odom` ROS topic. The code for this can be found in the `leg_callback` function of [Appendix section 2](#).

### **Results:**

According to our observations, the `to_pose_array` topic allowed for better performance of the leg tracking function. The package was only partially successful in tracking the human. During instances of the sharp turning of the human, the robot loses the human and keeps moving straight instead of following the human.



*Figure 13: Human leg tracking in Gazebo and Rviz*



*Figure 14: The robot loses the human in instances of sharp turning*

## Task 5: Code integration

For the most part, much of the necessary code was either built as assignments throughout the semester or given to us. However, this does not mean that all we had to do was to take them and put them together and we were finished. The blending and debugging of our code were integral to our group's success.

Our initial plan was to use multiple Apriltags placed strategically at the beginning of separate sections to switch between codes and create a fluid simulation of our final project. We had placed apriltags at the beginning of the line following and human tracking areas. However, we eventually settled on a global flag-based implementation to determine which function it should be running. This allowed us to be independent of additional objects being placed in the world. The global flag system we applied uses the pre-existing objects in the world like frontal obstacles, the yellow line, the stop-sign and the human to switch between codes, as necessary. Our overall result provided a fluid and autonomous path. However, although our blending worked fluidly for our exact project scenario, it is not the most optimal option. A detailed analysis of the flag system we used is shown below.

Flag descriptions:

Flag\_1: This flag is controlled by the line following function. It is initialized at a zero value. It changes to the 2 when a yellow line is detected. It reverts to 1 if there is no yellow line is detected at any time.

Flag\_2: This flag is controlled by the stop sign detection function. It is initialized at a zero value and increases by 1 every time the function detects a stop sign.

Sr. No.	Function	Flag_1	Flag_2	Comments
1	Wall following and obstacle avoidance	0 or 1	0	This function works when there is no line in the image AND when the stop sign has not been detected.
2	Line following	2	any value	This is activated every time the robot detects a yellow line in the lowermost part of its image feed.
3	Stop sign detection	any value	any value	This is always active.
4	Stopping after stop sign detection	any value	0	The robot stops only when it detects the stop sign for the first time.
5	Human tracking	1	$\geq 1$	For this function to be active, the bot must have seen the stop sign at least once and should not be seeing a yellow line.

NOTE: the above flags have been implemented by using if conditions and while loops just before publishing each velocity. By doing this, we have ensured that all packages are always running from the start and their dependent calculations are continuously active. Since loading these packages takes a considerable amount of time, we felt that it was better to control only the publishing of each of the function's velocity with the flags. To see the actual code integration, please refer to [Appendix section 2](#).

The global flags system ensures that only one function is being run at any one time. If multiple functions are running at any one point in time, the bot is very likely to underperform, since multiple functions governing the velocity of the robot lead to slowing down of the robot and it does not react to any of the functions in the desired manner.

To maintain the correct flow of functions in the current world scenario where human tracking starts after stop sign detection, we have adjusted the flags for running leg tracking only after the required sequence of events. The flag system can be modified by changing the flag triggers or introducing more flags in the code to make it more robust to the sequence of events.

Finally, below is a list of all ROS topics used in this project:

1. Subscribed topics:

```
/camera/rgb/image_raw  
/scan  
/darknet_ros/bounding_boxes  
/to_pose_array/leg_detector  
/odom
```

2. Published topics:

```
/cmd_vel
```

## **Conclusion:**

To conclude, this project implemented many key aspects of Linux and ROS. This project quickly became a perfect test of “who was paying attention in class and actually performed the extracurricular work assigned by Dr. Krovi.”. Although the end tasks completed was to get the robot to navigate its way around a created path, there were so many more lessons learned on the path to implementing just this project. Relying heavily on collaboration and in times of quarantine, this project demanded the utmost communication from every member on the team. Utilizing the tools at our reaches such as zoom and other messenger applications, the group was able to overcome these additional difficulties and produce a proper result. Although being able to see a finished result in person would be nice, the Gazebo simulations do a great job of visualizing the fruits of our effort. We would like to thank our professor Dr. Krovi, and TAs Mugdha and Adhiti for the hard work and diligence throughout the duration of this course.



## References

1. “Operations with Images.” *OpenCV*, docs.opencv.org/master/d5/d98/tutorial\_mat\_operations.html.
2. Redmon, Joseph. “YOLO.” *YOLO: Real-Time Object Detection*, pjreddie.com/darknet/yolo/.
3. Leggedrobotics. “Leggedrobotics/darknet\_ros.” *GitHub*, 14 Apr. 2020, github.com/leggedrobotics/darknet\_ros.
4. Robotis. “ROBOTIS e.” *Manual*, emannual.robotis.com/docs/en/platform/turtlebot3/.
5. “Wiki.” *Ros.org*, wiki.ros.org/ROS/Tutorials/.
6. Dr. Krovi, Venkat. “Clemson University ICAR”. AuE 8930: Autonomy Science and Systems.Spring 2020.
7. Osrf. “Beginner: GUI.” *Gazebo*, gazebosim.org/tutorials?cat=guided\_b&tut=guided\_b2.
8. U of Michigan. “Apriltag - Software.” *AprilTag*, april.eecs.umich.edu/software/apriltag.
9. Téllez Ricardo, et al. *ROS Basics in 5 Days: Entirely Practical Robot Operating System Training*. The Construct, 2017.

## Appendix

### Camera implementation on the TurtleBot Burger model in Gazebo simulator environment:

Changes were made to the following two files. The below code snippets must replace the existing code in the files.

1. File name: "turtlebot3\_burger.gazebo.xacro"
2. File name: "turtlebot3\_burger.urdf.xacro"

The below code snippet was added in the end of this file:

1. File name: "turtlebot3\_burger.gazebo.xacro"

```
<!--link : https://www.raspberrypi.org/documentation/hardware/camera/-->
<gazebo reference="camera_rgb_frame">
  <sensor type="camera" name="Pi Camera">
    <always_on>true</always_on>
    <visualize>$(arg camera_visual)</visualize>
    <camera>
      <horizontal_fov>1.085595</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.03</near>
        <far>100</far>
      </clip>
    </camera>
    <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>30.0</updateRate>
      <cameraName>camera</cameraName>
      <frameName>camera_rgb_optical_frame</frameName>
      <imageTopicName>rgb/image_raw</imageTopicName>
      <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>
      <hackBaseline>0.07</hackBaseline>
      <distortionK1>0.0</distortionK1>
      <distortionK2>0.0</distortionK2>
      <distortionK3>0.0</distortionK3>
      <distortionT1>0.0</distortionT1>
      <distortionT2>0.0</distortionT2>
    </plugin>
  </sensor>
```

```
</gazebo>
```

3. File name: Turtlebot3 burger.urdf.xacro:

```
<joint name="camera_joint" type="fixed">
  <origin xyz="0.073 -0.011 0.044" rpy="0 0 0"/>
  <parent link="base_link"/>
  <child link="camera_link"/>
</joint>

<link name="camera_link">
  <collision>
    <origin xyz="0.005 0.011 0.013" rpy="0 0 0"/>
    <geometry>
      <box size="0.015 0.030 0.027"/>
    </geometry>
  </collision>
</link>

<joint name="camera_rgb_joint" type="fixed">
  <origin xyz="0.003 0.011 0.009" rpy="0 0 0"/>
  <parent link="camera_link"/>
  <child link="camera_rgb_frame"/>
</joint>
<link name="camera_rgb_frame"/>

<joint name="camera_rgb_optical_joint" type="fixed">
  <origin xyz="0 0 0" rpy="-1.57 0 -1.57"/>
  <parent link="camera_rgb_frame"/>
  <child link="camera_rgb_optical_frame"/>
</joint>
<link name="camera_rgb_optical_frame"/>
```

**Final integrated code in python:**

```
#!/usr/bin/env python

import rospy
import time
from sensor_msgs.msg import LaserScan
```

```

from geometry_msgs.msg import Twist
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
from darknet_ros_msgs.msg import BoundingBoxes
from people_msgs.msg import PositionMeasurementArray
from nav_msgs.msg import Odometry
from decimal import *
import numpy as np
import cv2
import tf
import math
from geometry_msgs.msg import PoseArray

global d
right_min = 0.3
left_min = 0.3
front_min = 10
d = 0
det_flag = 0
bot_x = 0; bot_y = 0; bot_yaw = 0
leg_track = 0
print("Starting script")

class LineFollower(object):
    def __init__(self):
        self.bridge_object = CvBridge()
        self.image_sub = rospy.Subscriber("/camera/rgb/image_raw", Image, self.camera_callback)
        self.scan_sub = rospy.Subscriber('/scan', LaserScan, self.callback_wall)
        self.det_sub = rospy.Subscriber("/darknet_ros/bounding_boxes", BoundingBoxes, self.det_callback)
        self.leg_sub = rospy.Subscriber('/to_pose_array/leg_detector', PoseArray, self.leg_callback)
        #self.leg_sub = rospy.Subscriber('/people_tracker_measurements', PositionMeasurementArray, self.leg_callback)
        self.bot_sub = rospy.Subscriber('/odom', Odometry, self.bot_callback)
        self.vel_pub = rospy.Publisher('/cmd_vel', Twist, queue_size = 10)

        self.rate = rospy.Rate(225)

    def callback_wall(self, data):
        # print("in 1")
        global right_min, left_min, front_min
        left = data.ranges[90]
        right = data.ranges[270]
        front = data.ranges[0]

```

```

right_cone = []
left_cone = []
front_cone = []
if right == float("inf"): right_min = 0.3
else:
    for i in range(300,341):
        right_cone.append(data.ranges[i])

if left == float("inf"): left_min = 0.3
else:
    for o in range(20,61):
        left_cone.append(data.ranges[o])

for p in range(len(data.ranges)):
    if p<20 or p>340:
        front_cone.append(data.ranges[p])
    if len(right_cone)!=0:
        right_min = min(right_cone)
    if len(left_cone)!= 0:
        left_min = min(left_cone)

    front_min = min (front_cone)

def camera_callback(self, data):
    global d, det_flag, leg_track
    # print("in 2")
    # print("d: ", d, "det_flag: ", det_flag)
    if d ==3 and det_flag == 1:
        # print("line sleep")
        rospy.sleep(3)
        d = 4
        # print(d)
    # We select bgr8 because its the OpneCV encoding by default
    cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding="bgr8")

    # We get image dimensions and crop the parts of the image we dont need
    height, width, channels = cv_image.shape
    crop_img = cv_image[(height/2)+230:(height/2)+240][1:width]

    # Convert from RGB to HSV
    hsv = cv2.cvtColor(crop_img, cv2.COLOR_BGR2HSV)

    # Define the Yellow Colour in HSV

```

```

"""
    To know which color to track in HSV use ColorZilla to get the color registered by the
    camera in BGR and convert to HSV.
"""

# Threshold the HSV image to get only yellow colors
lower_yellow = np.array([20,100,100])
upper_yellow = np.array([50,255,255])
mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

# Calculate centroid of the blob of binary image using ImageMoments
m = cv2.moments(mask, False)
# d = 0
try:
    cx, cy = m['m10']/m['m00'], m['m01']/m['m00']
    d = 2
except ZeroDivisionError:
    cx, cy = height/2, width/2
    d = 1

# Draw the centroid in the resultut image
# cv2.circle(img, center, radius, color[, thickness[, lineType[, shift]]])
cv2.circle(mask,(int(cx), int(cy)), 10,(0,0,255),-1)

cv2.imshow("Original", cv_image)
cv2.imshow("MASK", mask)
cv2.waitKey(1)

"""
Enter controller here.
"""

twist_object = Twist()
twist_object.linear.x = 0.12
diff = width/2 - cx
twist_object.angular.z = diff/900

if d==2:
    # print("line following active")
    self.vel_pub.publish(twist_object)

def det_callback(self,data):
    global d, det_flag
    #print("in 3")
    twist_obj = Twist()

```

```

for i in range(len(data.bounding_boxes)):
    if data.bounding_boxes[i].Class == "stop sign":
        d = 3
        det_flag += 1
        # print(d)
        print("stop sign detected")
        if det_flag == 1 and leg_track == 0:
            print("Stopping for 3 seconds")
            twist_obj.linear.x = 0
            twist_obj.angular.z = 0
            self.vel_pub.publish(twist_obj)
            rospy.sleep(3)

def wall_avoid(self):
    global right_min, left_min, front_min, d, det_flag
    vel_msg = Twist()
    vel_msg.linear.x = 0.1
    vel_msg.linear.y = 0
    vel_msg.linear.z = 0
    vel_msg.angular.x = 0
    vel_msg.angular.y = 0
    vel_msg.angular.z = 0

    while det_flag < 1 and d < 2 and leg_track == 0:
        # print("wall/obs active")

        error = (left_min - right_min)/2

        # print("error: ", error)

        if front_min <= 0.23:
            vel_msg.linear.x = 0
            vel_msg.angular.z = 0.15
        if front_min > 0.23: vel_msg.linear.x = 0.1

        if error != float("inf") and error != float("-inf"):
            vel_msg.angular.z = error * 3
        if d != 2 and d != 3:
            self.vel_pub.publish(vel_msg)
        self.rate.sleep()

```

```

def leg_callback(self, data):
    global d, det_flag, bot_x, bot_y, bot_yaw, leg_track

    if det_flag>=1 and d ==1:
        print("Leg detection active")
        leg_track = 1

    # Leg following using pose array topic

    if len(data.poses) !=0:
        print("Person detected. Following person.")
        vel_msg = Twist()
        Kp_dist = 0.3 ; Kp_ang = 0.7
        leg_x = data.poses[0].position.x
        leg_y = abs(data.poses[0].position.y)
        theta_desired = math.atan2(leg_y - bot_y , leg_x - bot_x)
        distance_error = math.sqrt((leg_x - bot_x)**2 + (leg_y - bot_y)**2)

        # Uncomment following lines for debugging:

        # rospy.loginfo("Bot x %f      Bot y %f      Bot yaw %f",bot_x,bot_y,bot_yaw)
        # rospy.loginfo("leg x %f      leg y %f",leg_x,leg_y)
        # rospy.loginfo("theta is %f",theta_desired)
        # rospy.loginfo("distance error %f",distance_error)

        angular_error = theta_desired - bot_yaw
        if distance_error > 0.32:
            vel_msg.linear.x = Kp_dist * distance_error
            vel_msg.angular.z = Kp_ang * angular_error
        if front_min < 0.3:
            vel_msg.linear.x = 0
            vel_msg.angular.z = 0
        self.vel_pub.publish(vel_msg)
        self.rate.sleep()

    # Leg following using people_tracker_measurements topic

    # if len(data.people) !=0:
    #     self.rate_tracker = rospy.Rate(10)
    #     vel_msg = Twist()
    #     Kp_dist = 10 ; Kp_ang = 0.1
    #     leg_x = data.people[0].pos.x
    #     leg_y = data.people[0].pos.y
    #     theta_desired = math.atan2(leg_y - bot_y , leg_x - bot_x)

```



```

        # distance_error = math.sqrt((leg_x - bot_x)**2 + (leg_y - bot_y)**2)
        # rospy.loginfo("Bot x %f      Bot y %f      Bot yaw %f",bot_x,bot_y,bot_yaw)
        # rospy.loginfo("leg x %f      leg y %f",leg_x,leg_y)
        # rospy.loginfo("theta is %f",theta_desired)
        # rospy.loginfo("distance error %f",distance_error)
        # angular_error = theta_desired - bot_yaw
        # if distance_error > 0.1:
        #     vel_msg.linear.x = Kp_dist * distance_error
        #     vel_msg.angular.z = Kp_ang * angular_error
        #     self.vel_pub.publish(vel_msg)
        #     self.rate.sleep()

def bot_callback(self, msg):
    global bot_x, bot_y, bot_yaw
    pose = Twist()
    #print("in 6")
    quaternion = (
        msg.pose.pose.orientation.x,
        msg.pose.pose.orientation.y,
        msg.pose.pose.orientation.z,
        msg.pose.pose.orientation.w)
    euler = tf.transformations.euler_from_quaternion(quaternion)
    roll = euler[0]
    pitch = euler[1]
    bot_yaw = euler[2]
    bot_x = msg.pose.pose.position.x
    bot_y = msg.pose.pose.position.y

if __name__ == '__main__':
    # global right_min, left_min, front_min
    # global d, det_flag
    rospy.init_node('gazebo_scan_sub')
    tobj = LineFollower()
    tobj.wall_avoid()

    rospy.spin()

```

## Launch file used:

```
<launch>

  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle
, waffle_pi]"/>
  <arg name="x_pos" default="-0.2"/>
  <arg name="y_pos" default="1.6"/>
  <arg name="z_pos" default="0"/>
  <arg name="yaw_pos" default="3.14"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find turtlebot3_auefinals)/worlds/turtlebot3_fina
l.world" />
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>

  <include file="$(find person_sim)/launch/init_small_standing_person.launch">
  </include>

  <include file="$(find turtle_tf_3d)/launch/person_standing_keyboard_move.launch">
  </include>

  <include file="$(find turtlebot3_auefinals)/launch/leg_detector.launch">
  </include>

  <include file="$(find darknet_ros)/launch/darknet_ros.launch">
  </include>

  <include file="$(find turtlebot3_slam)/launch/turtlebot3_slam.launch">
    <arg name="slam_methods" value="hector"/>
  </include>

  <param name="robot_description" command="$(find xacro)/xacro --
inorder $(find turtlebot3_description)/urdf/turtlebot3_$(arg model).urdf.xacro" />
```

```

    <node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf" args="-urdf -
model turtlebot3_$(arg model) -x $(arg x_pos) -y $(arg y_pos) -z $(arg z_pos) -
Y $(arg yaw_pos) -param robot_description" />

    <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_pu
blisher1">
        <param name="publish_frequency" type="double" value="50.0" />
        <param name="tf_prefix" value=""/>
    </node>
<!--
    <node pkg="turtlebot3_auefinals" type="wall_line_combined_iter10.py" name="master_s
cript" />
-->
</launch>

```