|  |  |  |
| --- | --- | --- |
| C:\Users\mohammad\Downloads\image(1).png | به نام خدا | C:\Users\mohammad\Pictures\amirkabirLogo.png |
| **دانشگاه صنعتی امیرکبیر**  **دانشکده‌ مهندسی کامپیوتر**  **اصول علم ربات**  **تمرین سری چهارم بخش پیاده سازی** | | |

|  |  |
| --- | --- |
| مهدی رحمانی | نام و نام خانوادگی |
| 9731701 | شماره‌ دانشجویی |
| 03/04/1402 | تاریخ ارسال گزارش |

­

**فهرست گزارش سوالات**

[بخش صفرم (آماده سازی Workspace) 3](#_Toc138461992)

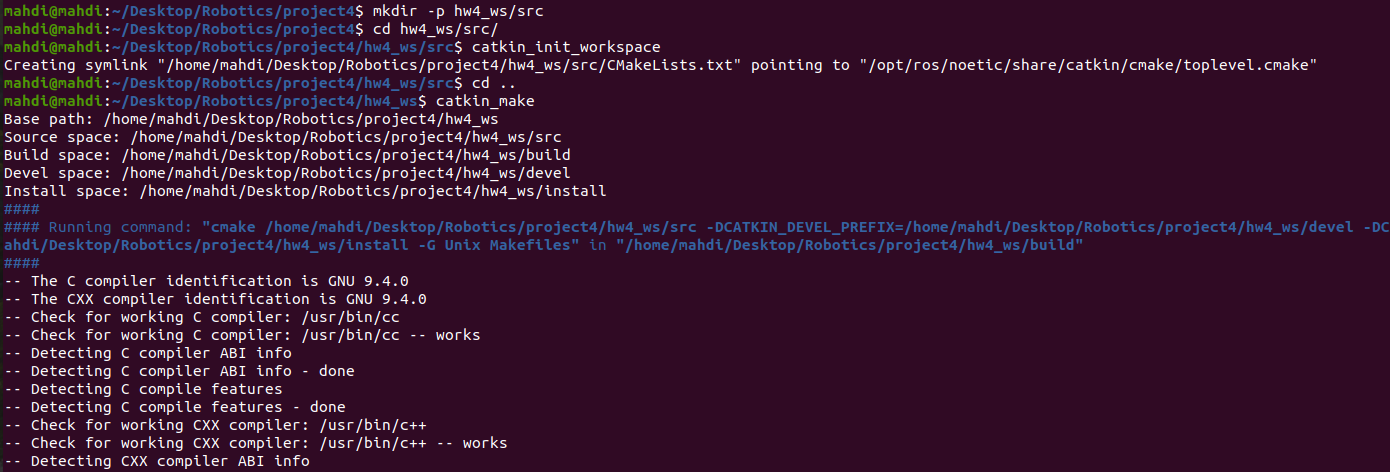
[گام اول- توضیحات پیاده‌سازی 6](#_Toc138461993)

[گام اول- اجرا و نتایج 15](#_Toc138461994)

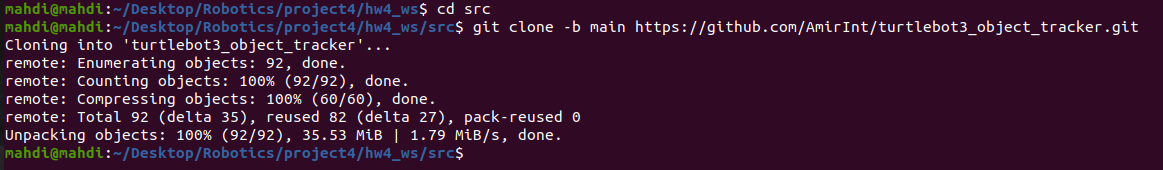
[گام دوم) امتیازی(-توضیحات پیاده سازی 16](#_Toc138461995)

[گام دوم) امتیازی( - اجرا و نتایج 23](#_Toc138461996)

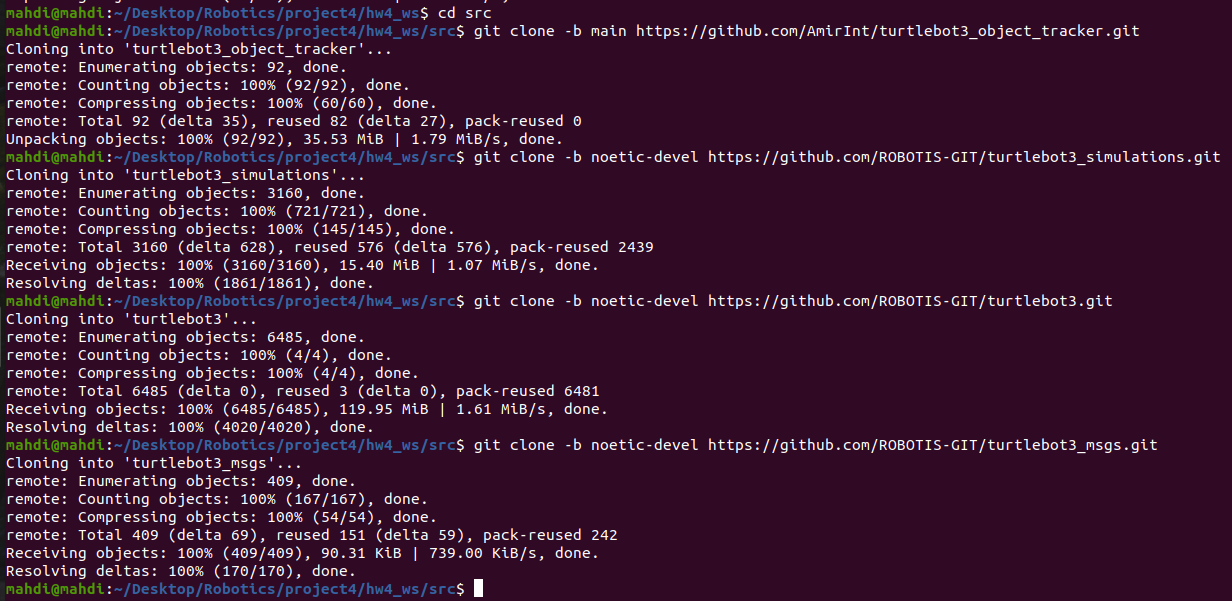
# بخش صفرم (آماده سازی Workspace)

ابتدا یک Work space برای این پروژه میسازیم و آن را initialize میکنیم:

برای گام اول ما طبق توضیحات دستورکار کد بخش‌های پایه‌ای نودها برای ما قرار داده شده است و کافی است آن را در کنار پکیج‌های دیگر خود در Workspace کلون کنیم.

* git clone -b main https://github.com/AmirInt/turtlebot3\_object\_tracker.git

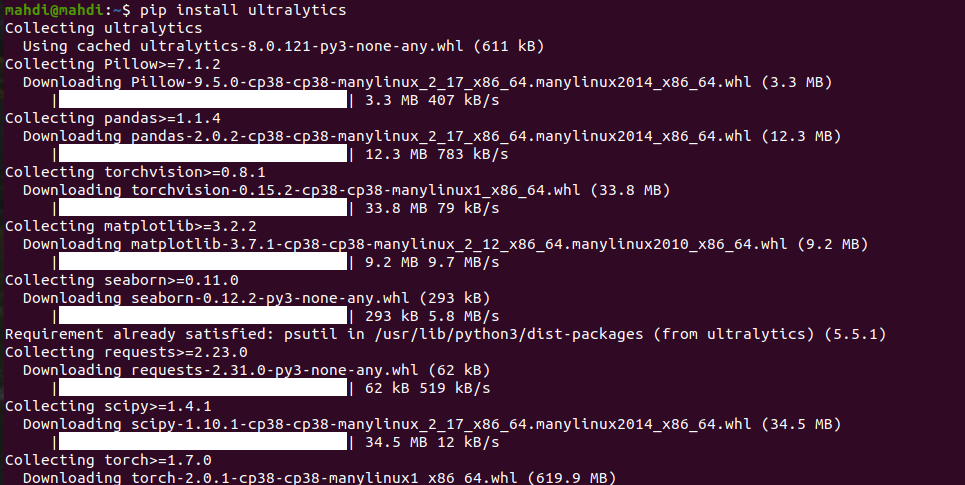
همچنین در این پروژه به ربات turtlebot3 و شبیه‌ساز gazebo و rviz نیاز می‌شود. پس برای این منظور لازم است تا پکیج‌های زیر را نیز در فولدر src مربوط به workspaceمان کلون کنیم.

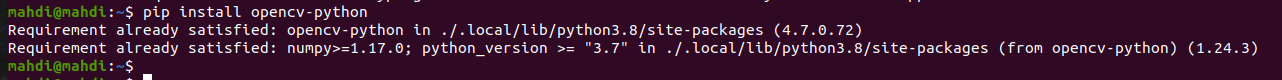
* git clone -b noetic-devel https://github.com/ROBOTIS-GIT/turtlebot3\_simulations.git
* git clone -b noetic-devel https://github.com/ROBOTIS-GIT/turtlebot3.git
* git clone -b noetic-devel https://github.com/ROBOTIS-GIT/turtlebot3\_msgs.git

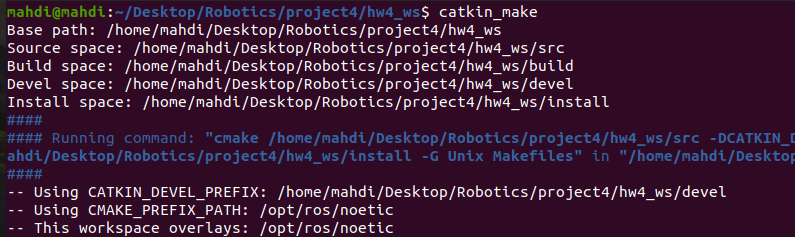
لازم است کتابخانه‌های زیر را نصب کنیم:

* numpy
* ultralytics
* opencv-python

numpy را من از قبل نصب داشتم. لازم است تا ultralytics را نصب کنیم که حجم نسبتا زیادی هم دارد.



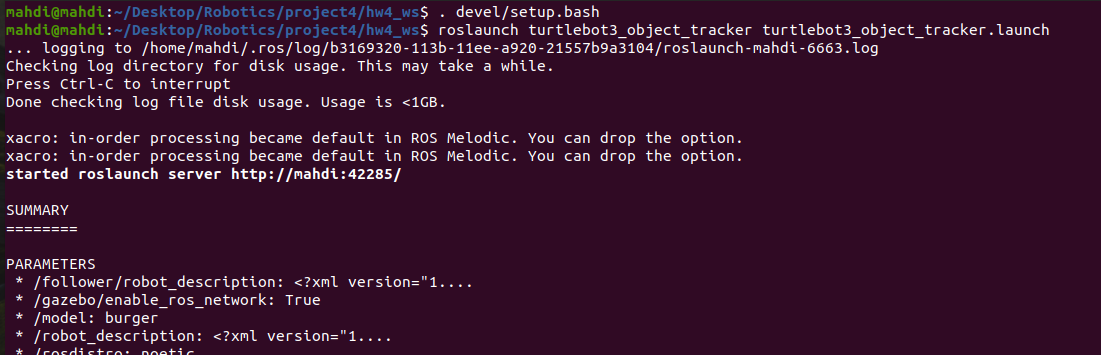
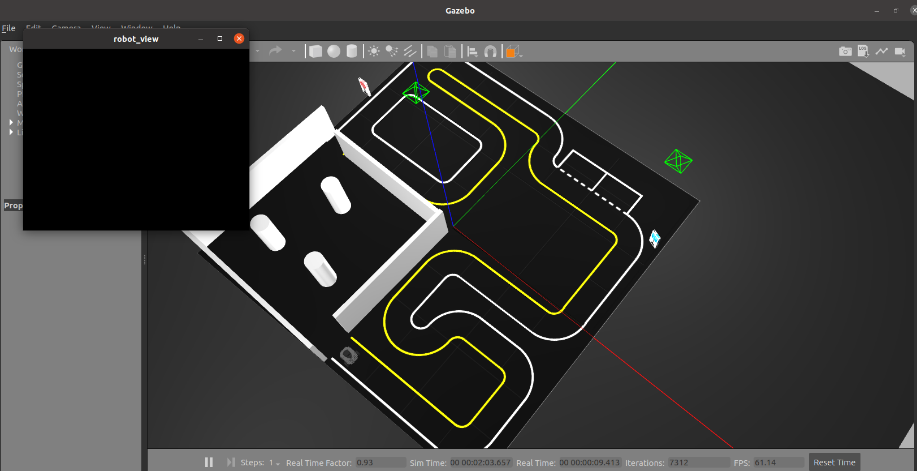
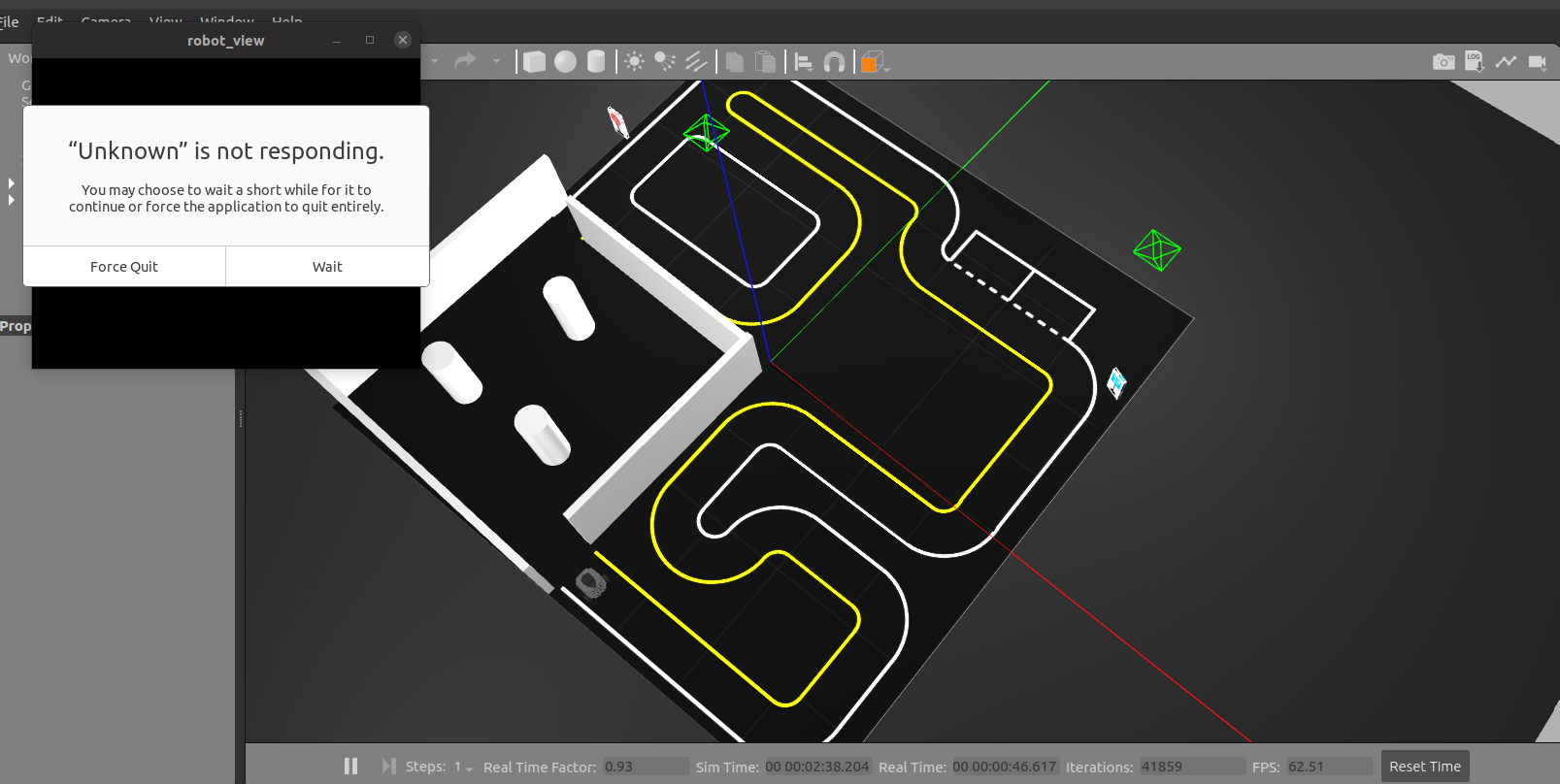
سپس لازم است open-CV را هم نصب کنیم:

پس از نصب تمامی Requirementها به دایرکتوری root مربوط به workspaceمان میرویم و catkin\_make را اجرا میکنیم.

سپس برای شروع میتوانید برنامه را اجرا کرده ولی خب طبیعتا چون پیاده‌سازی نکرده ایم و همچنین تاپیکی را subscribe نکردیم خروجی خاصی نداریم.

برای این منظور در همان دایرکتوری root مربوط به workspace رفته و کاندهای زیر را اجرا کنید:

* . devel/setup.bash
* roslaunch turtlebot3\_object\_tracker turtlebot3\_object\_tracker.launch

خروجی به صورت زیر خواهد بود و البته ممکنه پیام not responding هم برای شما بیاید.

# گام اول- توضیحات پیاده‌سازی

در این قسمت در ابتدا لازم است تا از طریق rqt graph بتوانیم تاپیک مناسب را برای subscribe کردن بیابیم. اگر قبل از این که چیزی پیاده سازی کنیم مراجعه کنیم با تصویر زیر مواجه میشویم:

A screenshot of a computer

Description automatically generated

سپس اگر subscribe را به صورت زیر یبنویسیم میتوانیم تصویر دوربین را مشاهده کنیم:

* self.camera\_subscriber = rospy.Subscriber("/follower/camera/image",Image,self.camera\_listener)

A screenshot of a computer program

Description automatically generated with low confidence

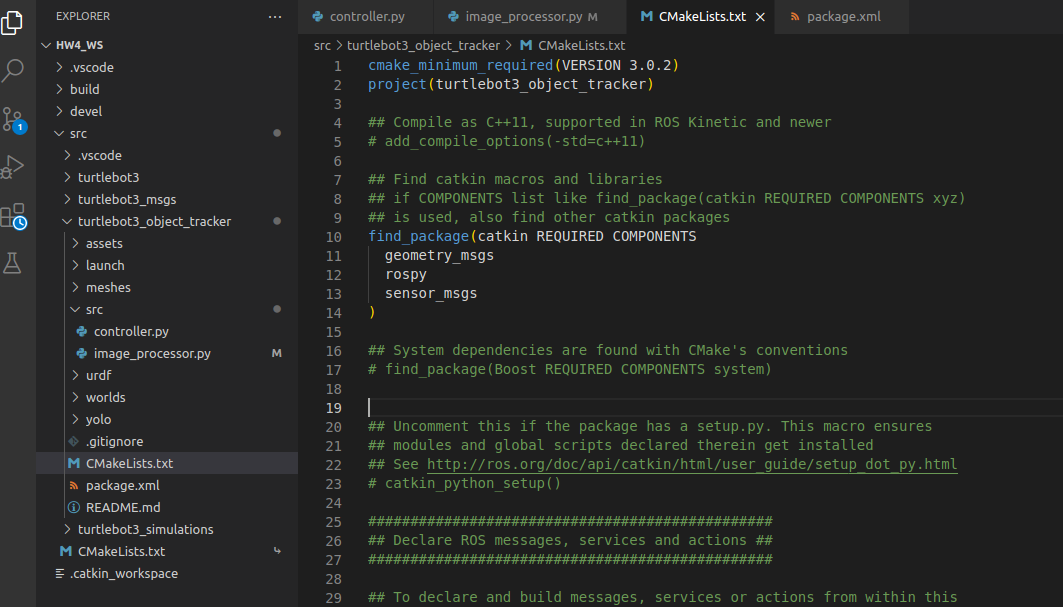
A picture containing text, screenshot, font, white

Description automatically generatedگراف نیز به صورت زیر خواهد شد:

سپس لازم است تا مدل را به متغیر self.model تخصیص دهیم. برای این منظور باتوجه به مسیر ذخیره شدن مدل خواهیم داشت:

* self.model: YOLO = YOLO('../yolo/yolov8n.pt')

حال لازم است تا یک سرویس برای detection بسازیم. برای این منظور ابتدا فایل‌های CMakeLists.txt و package.xml را بازکرده تا تغییرات لازم را در آن‌ها بدهیم.



1. ابتدا در package.xml به سراغ 55 و 61 میرویم و به ترتیب در این خطوط موارد زیر را اضافه میکنیم:

* >build\_depend>message\_generation</build\_depend<
* A screen shot of a computer program

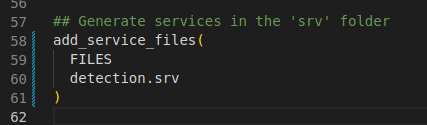
  Description automatically generated with low confidence>exec\_depend>message\_runtime</exec\_depend<

1. سپس در CMakeLists ابتدا باید در خط 14 عبارت message\_generation را اضافه نماییم:

A screenshot of a computer program

Description automatically generated with medium confidence

1. سپس به خط 58 رفته و باید add\_service\_files را uncomment کنیم و نام سرویس مورد نظر خود را اضافه نماییم:



1. A black screen with green text

   Description automatically generated with low confidenceسپس خطوط 71 تا 75 که مربوط به generate\_messages هست را uncomment میکنیم:
2. برای آنکه کارمان تکمیل شود، در همان پوشه turtlebot3\_object\_tracker یک پوشه به نام srv درست میکنیم و بعد داخل آن یک فایل با نام اونی که در استپ 3 مشخص کردیم میذاریم .

A screenshot of a computer program

Description automatically generated with medium confidence

1. A screen shot of a computer

   Description automatically generated with medium confidenceتوی فایل فوق لازمه که ورودی ها و خروجی های سرویس را بگوییم. بین ورودی ها و خروجی ها با --- جدا میشود. اگرم ورودی نداره خط اول همون --- میشود.
2. A picture containing text, screenshot, font

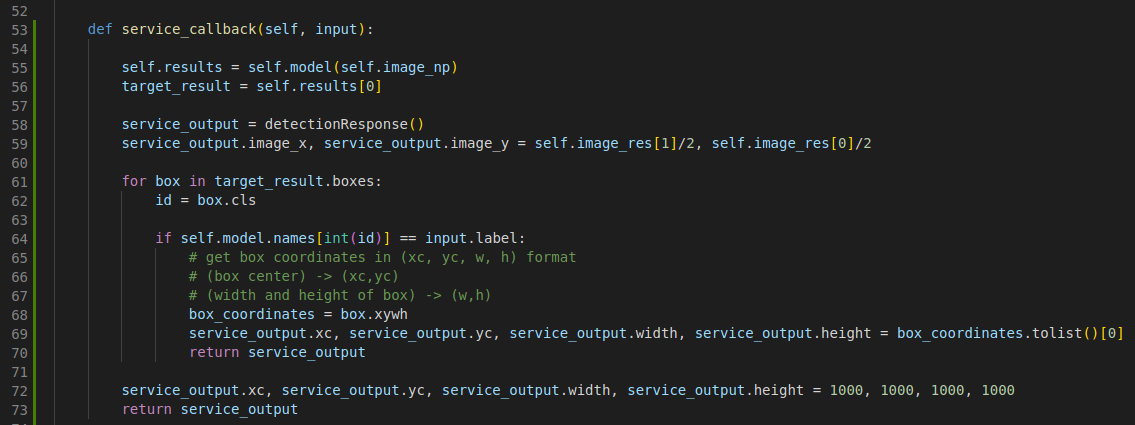
   Description automatically generatedحال به پوشه hw4\_ws در ترمینال برگشته و catkin\_make میکنیم.

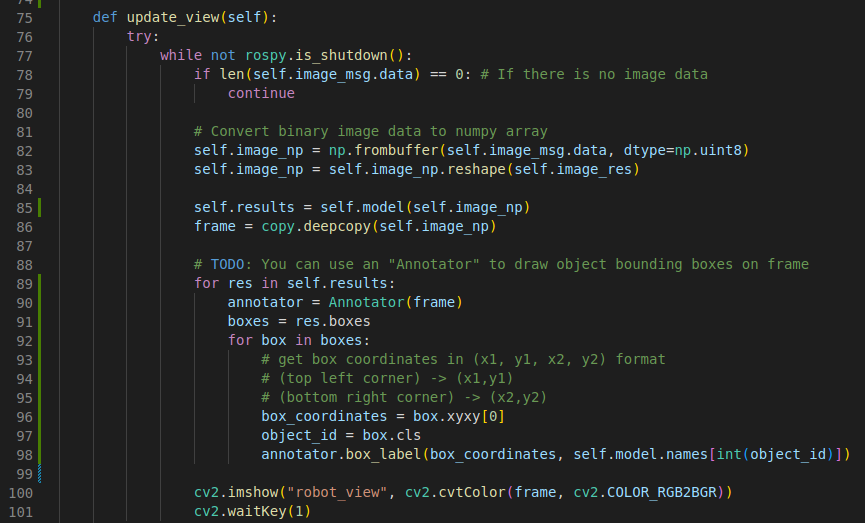
حال لازم است سرویس مربوط به human detection را در image\_processor پیاده سازی کنیم.

ابتدا خط زیر را در init اضافه میکنیم:

* A screen shot of a computer

  Description automatically generated with low confidenceself.human\_detection\_server = rospy.Service('detection', detection, self.service\_callback)

همچنین لازم است تا متدی که قرار است سرور سرویس اجرا کند یعنی service\_callback را پیاده کنیم. ابتدا باید image\_np را به مدل بدهیم و self.results را آپدیت کنیم. سپس یک detectionResponse میسازیم و باید فیلدهای جوابی که قرار است به کلاینت دهیم را پر کنیم. حال باید باکس‌هایی که داخل result هست را بیابیم و از بین آن‌ها یکی مربوط به person است البته آن هم به شرط اینکه person در کادر باشد. در این صورت مشخصات آن باکس را در قالب پیام پاسخ میگذاریم و برای کلاینت ارسال میکنیم. چنانچه person که همان label ماست به عنوان label باکسی حضور نداشت باید یک مقداری برای فیلدها قرار دهیم که گیرنده با دیدن آن‌ها متوجه شود person داخل کادر نبوده. برای این منظور من مقدار 1000 قرار دادم که باتوجه به سایز تصویر غیرقابل دسترس است.

سپس لازم است تغییرات اندکی در update\_view ایجاد کنیم که بتواند باکس‌ها را به کمک annotator دور objectها ترسیم کند و label آن‌ها را نیز قرار دهد.

دقت شود برای یافتن ابعاد باکس به کمک مختصات ‌‎ها دو روش است یکی box.xyxy که مختصات دو گوشه‌ی سمت چپ بالا و سمت راست پایین را میدهد. دیگری box.xywh که مختصات نقطه مرکزس باکس و عرض و ارتفاع آن را میدهد.

کد مربوط به این قسمت در صفحه بعد کامل آمده است:

#!/usr/bin/python3

# Python

import copy

# Object detection

import cv2

import numpy as np

from ultralytics import YOLO

from ultralytics.yolo.utils.plotting import Annotator

from ultralytics.yolo.engine.results import Results

import torch

# ROS

import rospy

from sensor\_msgs.msg import Image

# detection service

from turtlebot3\_object\_tracker.srv import detection, detectionResponse

class ImageProcessor:

    def \_\_init\_\_(self) -> None:

        # Image message

        self.image\_msg = Image()

        self.image\_res = 240, 320, 3 # Camera resolution: height, width

        self.image\_np = np.zeros(self.image\_res) # The numpy array to pour the image data into

        # TODO: Subscribe on your robot's camera topic

        # NOTE: Make sure you use the provided listener for this subscription

        self.camera\_subscriber = rospy.Subscriber("/follower/camera/image",Image,self.camera\_listener)

        # TODO: Instantiate your YOLO object detector/classifier model

        self.model: YOLO = YOLO('../yolo/yolov8n.pt')

        # TODO: You need to update results each time you call your model

        self.results: Results = None

        self.cv2\_frame\_size = 400, 320

        cv2.namedWindow("robot\_view", cv2.WINDOW\_NORMAL)

        cv2.resizeWindow("robot\_view", \*self.cv2\_frame\_size)

        # TODO: Setup your "human detection" service

        self.human\_detection\_server = rospy.Service('detection', detection, self.service\_callback)

        self.update\_view()

    def camera\_listener(self, msg: Image):

        self.image\_msg.data = copy.deepcopy(msg.data)

    def service\_callback(self, input):

        self.results = self.model(self.image\_np)

        target\_result = self.results[0]

        service\_output = detectionResponse()

        service\_output.image\_x, service\_output.image\_y = self.image\_res[1]/2, self.image\_res[0]/2

        for box in target\_result.boxes:

            id = box.cls

            if self.model.names[int(id)] == input.label:

                # get box coordinates in (xc, yc, w, h) format

                # (box center) -> (xc,yc)

                # (width and height of box) -> (w,h)

                box\_coordinates = box.xywh

                service\_output.xc, service\_output.yc, service\_output.width, service\_output.height = box\_coordinates.tolist()[0]

                return service\_output

        service\_output.xc, service\_output.yc, service\_output.width, service\_output.height = 1000, 1000, 1000, 1000

        return service\_output

    def update\_view(self):

        try:

            while not rospy.is\_shutdown():

                if len(self.image\_msg.data) == 0: # If there is no image data

                    continue

                # Convert binary image data to numpy array

                self.image\_np = np.frombuffer(self.image\_msg.data, dtype=np.uint8)

                self.image\_np = self.image\_np.reshape(self.image\_res)

                self.results = self.model(self.image\_np)

                frame = copy.deepcopy(self.image\_np)

                # TODO: You can use an "Annotator" to draw object bounding boxes on frame

                for res in self.results:

                    annotator = Annotator(frame)

                    boxes = res.boxes

                    for box in boxes:

                        # get box coordinates in (x1, y1, x2, y2) format

                        # (top left corner) -> (x1,y1)

                        # (bottom right corner) -> (x2,y2)

                        box\_coordinates = box.xyxy[0]

                        object\_id = box.cls

                        annotator.box\_label(box\_coordinates, self.model.names[int(object\_id)])

                cv2.imshow("robot\_view", cv2.cvtColor(frame, cv2.COLOR\_RGB2BGR))

                cv2.waitKey(1)

        except rospy.exceptions.ROSInterruptException:

            pass

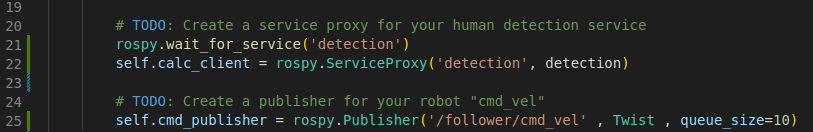
if \_\_name\_\_ == "\_\_main\_\_":

    rospy.init\_node("image\_processor", anonymous=True)

    rospy.on\_shutdown(cv2.destroyAllWindows)

    image\_processor = ImageProcessor()

    rospy.spin()

در قسمت کنترلر لازم است تا در تابع init هم service proxy را تعریف کنیم و هم نود را به عنوان پابلیشر cmd\_vel برای follower تعریف کنیم.

سپس در قسمت ران لازم است بگوییم که درخواستی برای human detection service بفرستد و چنانچه person در کادر بود و اطلاعات آن را برگرداند سرعت زاویه‌ای را به کمک کنترلر محاسبه کنیم.

برای محاسبه خطایی که باید در kp که اینجا angular\_vel\_coef تعریف شده است، در آن ضرب شود میایم فاصله x را برای وسط باکس تا وسط تصویر میابیم و سپس سرعت زاویه‌ای را محاسبه کرده و پابلیش میکنیم.

همچنین اگر انسان در تصویر نبود من اینگونه تعریف کردم که ربات freeze شود و حرکت نکند.

درنهایت کد کامل آن در ادامه آمده است:

#!/usr/bin/python3

# ROS

import rospy

from geometry\_msgs.msg import Twist

# detection service

from turtlebot3\_object\_tracker.srv import detection, detectionRequest

class Controller:

    def \_\_init\_\_(self) -> None:

        # Use these Twists to control your robot

        self.move = Twist()

        self.move.linear.x = 0.1

        self.freeze = Twist()

        # The "p" parameter for your p-controller, TODO: you need to tune this

        self.angular\_vel\_coef = 0.002

        # TODO: Create a service proxy for your human detection service

        rospy.wait\_for\_service('detection')

        self.calc\_client = rospy.ServiceProxy('detection', detection)

        # TODO: Create a publisher for your robot "cmd\_vel"

        self.cmd\_publisher = rospy.Publisher('/follower/cmd\_vel' , Twist , queue\_size=10)

    def run(self) -> None:

        try:

            while not rospy.is\_shutdown():

                # TODO: Call your service, ride your robot

                req = detectionRequest()

                req.label = 'person'

                resp = self.calc\_client(req)

                if resp.xc == 1000:

                    self.cmd\_publisher.publish(self.freeze)

                else:

                    person\_x, person\_y = resp.xc, resp.yc

                    image\_center\_x, image\_center\_y = resp.image\_x, resp.image\_y

                    error = person\_x-image\_center\_x

                    angular\_velocity = self.angular\_vel\_coef\*(error)

                    self.move.angular.z = -angular\_velocity

                    self.cmd\_publisher.publish(self.move)

        except rospy.exceptions.ROSInterruptException:

            pass

if \_\_name\_\_ == "\_\_main\_\_":

    rospy.init\_node("controller", anonymous=True)

    controller = Controller()

    controller.run()

# گام اول- اجرا و نتایج

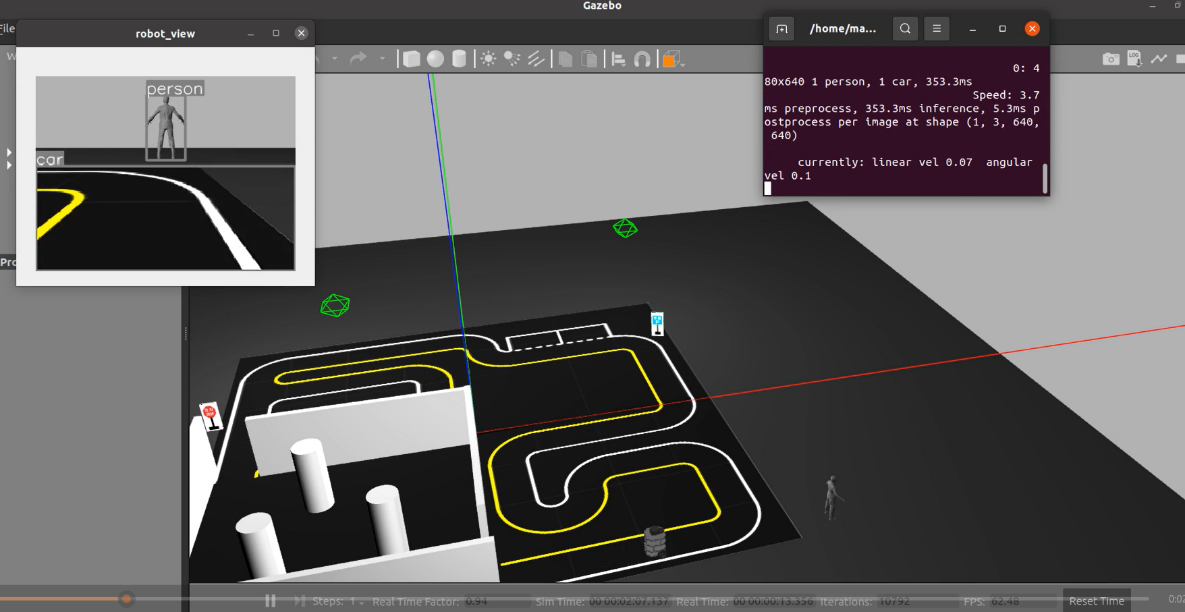
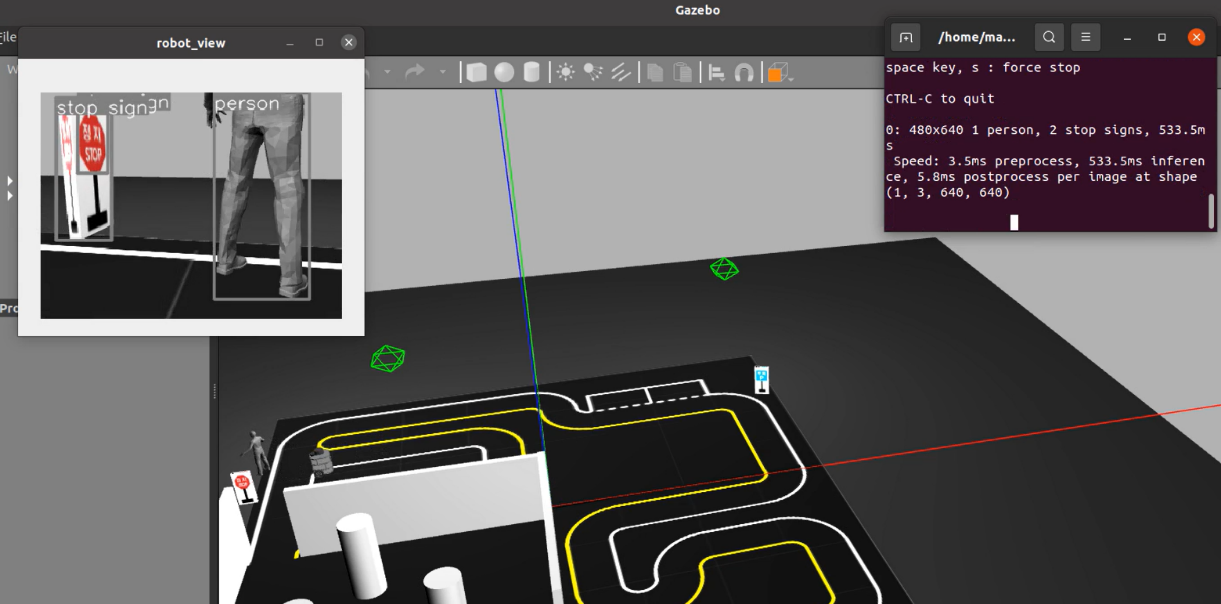
در نهایت برای اجرا دستورات زیر را وارد کرده:

* . devel/setup.bash
* roslaunch turtlebot3\_object\_tracker turtlebot3\_object\_tracker.launch

سپس ربات باتوجه به سرعت خطی خود شروع به حرکت کرده و با توجه به فاصله‌اش تا person میزان سرعت زاویه‌ای خود را تنظیم میکند.

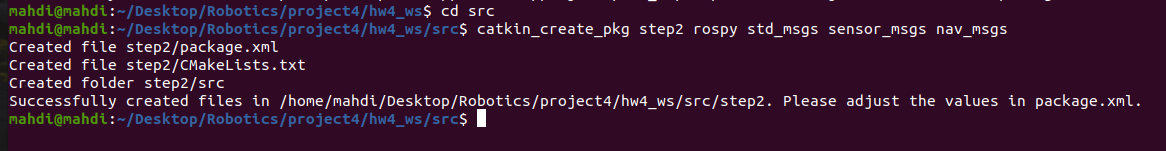
میتوان به کمک کلیدهای wو d و a و s و x شخص را هدایت کرد و پیش برد تا ربات هم آن را دنبال کند. فیلمی جهت صحت و درستی اجرا قرار داده شده است.

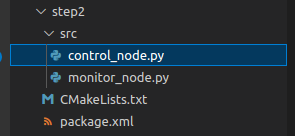
در زیر نمایی از دنبال کردن انسان توسط ربات را در Gazebo و robot view مشاهده میکنید:



# گام دوم) امتیازی(-توضیحات پیاده سازی

برای این قسمت یک پکیج جداگانه درست میکنیم به نام step2 و میکنیم و dependencyهایی که ممکن است به کار بیایند را هم اضافه کنیم.

* catkin\_create\_pkg step2 rospy std\_msgs sensor\_msgs nav\_msgs

حال به سراغ ساخت نودها میرویم. برای این منظور در فولدر src مربوط به step2 میرویم و دوفایل پایتون با نام های control\_node.py و monitor\_node.py میسازیم. نودmonitor\_node.py نیز برای نمایش مسیر ربات در rviz به کار گرفته خواهد شد.

حال در گام بعد به سراغ نوشتن کد مربوط به هریک از نودها میرویم. کد مربوط به نود monitor\_node مانند تمرین‌های قبل میباشد. در monitor کد پایتون مربوط به رسم مسیری که ربات آن را طی میکند میباشد که از تاپیک path استفاده میکند. مکان هایی که میرود را در قالب یک آرایه یا لیست ذخیره میکنیم و در rviz آن را نشان میدهیم.

حال به کد نود control\_node میپردازیم. در این جا یک متغیر direction داریم که باتوجه به متغیر follow\_type مقدارش میتواند 1 یا 1- باشد. چنانچه ربات دنبالگر راست باشد این ضریب منفی 1 است و به عنوان ضریبی در حاصل جمع P+I+D ضرب میشود. همچنین در این جا یک تابع distance\_from\_wall داریم که با توجه به اسکن‌های لیزر لایدار میتواند فاصله تا دیوار را برگرداند. دقت شود اگر دیوار در سمت راست باشد باید بین مقادیر در range بین 180 تا 360 مینیمم بگیریم و اگر سمت چپ باشد بین 0 تا 180 را بررسی کنیم. باتوجه به اینکه فاصله مطلوب تا دیوار را 1.5 گرفتیم بنابراین اختلاف فاصله تا دیوار و این مقدار 1.5 را باید سعی کنیم که 0 کنیم و PID در این جهت تلاش میکند. برای tune کردن PID ابتدا ترم P زیاد و D کم بود و باعث میشد که ربات به دیوار برخورد کند. در نهایت با سعی و خطا به مقدارهایی که دلخواه است رسیدیم. دقت شود از مقادیر wall follower در تمرین قبلی هم ایده گرفتم. همچنین توضیحات کافی با کامنت داخل کد داده شده و کل کد هم در ادامه قرار داده شده است.

#!/usr/bin/python3

import rospy

from geometry\_msgs.msg import Twist

from sensor\_msgs.msg import LaserScan

import matplotlib.pyplot as plt

class PIDController():

    def \_\_init\_\_(self):

        rospy.init\_node('controller', anonymous=False)

        rospy.on\_shutdown(self.on\_shutdown)

        self.cmd\_vel = rospy.Publisher('/cmd\_vel', Twist, queue\_size=10)

        # if we want to follow wall on right hand the robot should rotate

        # in the inverse direction of default state. so we multiply -1 coefficient

        # to (P+I+D)

        self.follow\_type = rospy.get\_param("/controller/follow\_type")

        if self.follow\_type == "right":

            self.direction = -1

        else:

            self.direction = 1

        # angular velocity PID gains

        self.k\_p = 1

        self.k\_i = 0.0001

        self.k\_d = 30

        self.dt = 0.005

        self.v = 0.06

        self.D = 0.5

        rate = 1/self.dt

        self.r = rospy.Rate(rate)

        self.errs = []

    def distance\_from\_wall(self):

        '''

        this method give us the distance from wall with the help of scans

        comming from the lidar sensor. if direction is -1 means that we

        want to have on the right hand of robot and ranges [180:] is important to

        be checked and if direction is 1 it is vice versa.

        '''

        laser\_data = rospy.wait\_for\_message("/scan" , LaserScan)

        if self.direction==1:

            rng = laser\_data.ranges[:180]

        else:

            rng = laser\_data.ranges[180:]

        d = min(rng)

        return d

    def control(self):

        '''

        this function is the main function of this code. we calculate the

        distance error and try to calculate P, I, D terms. the summation of these

        terms define our angular velocity.

        '''

        d = self.distance\_from\_wall()

        sum\_i\_theta = 0

        prev\_theta\_error = 0

        move\_cmd = Twist()

        move\_cmd.angular.z = 0

        move\_cmd.linear.x = self.v

        while not rospy.is\_shutdown():

            self.cmd\_vel.publish(move\_cmd)

            err = d - self.D

            self.errs.append(err)

            sum\_i\_theta += err \* self.dt

            P = self.k\_p \* err

            I = self.k\_i \* sum\_i\_theta

            D = self.k\_d \* (err - prev\_theta\_error)

            move\_cmd.angular.z = self.direction \* (P + I + D)

            prev\_theta\_error = err

            move\_cmd.linear.x = self.v

            #rospy.loginfo(f"P : {P} I : {I} D : {D}")

            #rospy.loginfo(f"error : {err} speed : {move\_cmd.linear.x} theta : {move\_cmd.angular.z}")

            d = self.distance\_from\_wall()

            self.r.sleep()

    def on\_shutdown(self):

        '''

        this method plot error of linear and angular velocity separately.

        '''

        rospy.loginfo("Stopping the robot...")

        self.cmd\_vel.publish(Twist())

        plt.plot(list(range(len(self.errs))),self.errs, label='errs')

        plt.axhline(y=0,color='R')

        plt.draw()

        plt.legend(loc="upper left", frameon=False)

        plt.savefig(f"errs\_{self.k\_p}\_{self.k\_d}\_{self.k\_i}.png")

        plt.show()

        rospy.sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

    try:

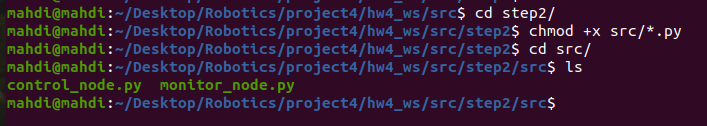
        pidc = PIDController()

        pidc.control()

    except rospy.ROSInterruptException:

        rospy.loginfo("Navigation terminated.")

در مرحله بعد باید تمامی کدهای پایتون را executable کنیم. برای این کار لازم است در ترمینال در پکیج step2 کد زیر را اجرا کنیم:

* chmod +x src/\*.py

سپس به سراغ نوشتن لانچ فایل میرویم. در همین پکیج step3 لازم است تا یک فولدر launch ایجاد کنیم. قبلا لانچ فایل با نام my\_empty\_world را که در پوشه لانچ مربوط به turtle\_bot بود را include میکردیم. در این جا یک فایل مشابه لانچ فایل با نام empty\_world مستقیما همینجا اضافه میکنیم. ( این لانچ فایل را میتوانید در لانچ فایل‌های turtlebot پیدا کنید.) اسم این لانچ فایل را turtlebot3\_maze.launch میگذاریم. محتوای آن به صورت زیر است. دقت شود که زاویه اضافه شدن ربات به map را هم در اینجا تعریف کردیم. همچنین world\_name را هم مشخص کردیم.

<launch>

  <arg name="model" default="$(env TURTLEBOT3\_MODEL)" doc="model type [burger, waffle, waffle\_pi]"/>

  <arg name="x\_pos" default="0.0"/>

  <arg name="y\_pos" default="0.0"/>

  <arg name="z\_pos" default="0.0"/>

  <arg name="yaw" value="0.0"/>

  <include file="$(find gazebo\_ros)/launch/empty\_world.launch">

    <arg name="world\_name" value="$(find step2/worlds/maze.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>

  <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) -param robot\_description" />

</launch>

همچنین یک control.launch ایجاد کنیم. لانچ فایل به صورت زیر است. ابتدا لانچ فایل قبلی را که نوشتیم include میکنیم و میگوییم در آن آرگومان های ورودی مثل مکان اولیه ربات و زاویه اولیه چه باشند. طبق خواسته سوال در این جا ربات در مکان (0و0.5-) و با زاویه 0 درجه باید اضافه شود. سپس launch file مربوط به rviz را include میکنیم تا آن را هم برای نمایش مسیر ربات بالا بیاورد.

همچنین نودهای control\_node.py و همچنین monitor\_node.py را با نام‌های controller و monitor بالا بیاورد. نود monitor که برای رسم مسیر لازم است و آن هم باید بالا بیاید.

<launch>

    <arg name="follow\_type" default="right"/>

    <include file="$(find step2)/launch/turtlebot3\_maze.launch">

        <arg name="x\_pos" value="-0.5"/>

        <arg name="y\_pos" value="0.0"/>

        <arg name="z\_pos" value="0.0"/>

        <arg name="yaw" value="0.0"/>

    </include>

    <include file="$(find turtlebot3\_gazebo)/launch/turtlebot3\_gazebo\_rviz.launch"/>

    <node pkg="step2" type="control\_node.py" name="controller" output="screen">

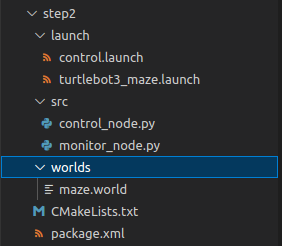
        <param name="follow\_type" value="$(arg follow\_type)" />

    </node>

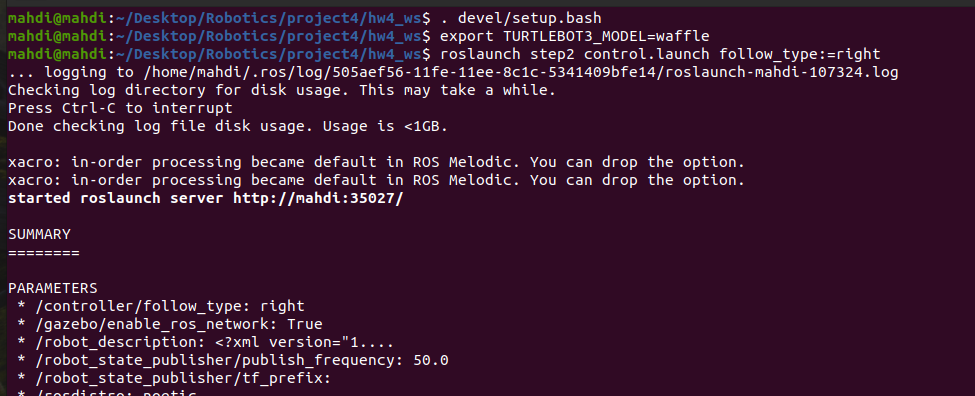
    <node pkg="step2" type="monitor\_node.py" name="monitor"></node>

</launch>

همچنین لازم است یک فولدر به نام worlds در src مربوط به پکیج step3 ایجاد کنیم و فایل square.world را در آنجا اضافه کنیم:



سپس در آخر لازم است تا به دایرکتوری ورک اسپیس برویم و catkin\_make را صدا بزنیم. سپس برای استفاده لازم است تا ابتدا سورس کنیم و سپس ربات را اکسپورت کنیم و در نهایت roslaunch را صدا بزنیم:

* . devel/setup.bash
* export TURTLEBOT3\_MODEL=waffle
* roslaunch step3 control.launch follow\_type:=right

در رابطه با tune کردن PID من ابتدا باتوجه به تمرین قبلی مقادیر زیر را برای gainها انتخاب کردم:

A screenshot of a computer

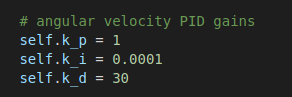
Description automatically generated with low confidence

سپس نتیجه آن شد که ربات به دیوار برخورد کرد:

A screenshot of a computer

Description automatically generated

باتوجه به اینکه اینجا به دیوار نزدیک هستیم و مستقیم به سمت آن میرویم لازم است که ربات سریعتر به خطایی که از حد آستانه و مطلوب دارد، پاسخ دهد. پس لازم است gain مربوط به P را زیاد کنیم همچنین چون P زیاد شود به همراه خود ممکن است باعث شود Overshoot زیاد شود لازم است تا مقدار gain مربوط به D را هم زیاد کنیم. در نهایت با سعی و خطا به مقادیر زیر رسیدم:



در نهایت میتوانید نتایج در قسمت بعدی مشاهده کنید.

# گام دوم) امتیازی( - اجرا و نتایج

اگر دقت شود دو مسیر خروج از maze وجود دارد. چنانچه ربات ما دنبالگر راست باشد و یعنی همیشه دیوار را در سمت راست خود نگه دارد آنگاه از خروجی بالایی خارج شده و چنانچه دنبالگر چپ باشد از خروجی پایینی اجرا شده است.

پیاده سازی به گونه‌ای است که در ترمینال میتوان به عنوان ورودی نوع دنبالگر را به کمک left یا right معرفی کرد. حال در ادامه نتیجه اجرا برای هر حالت آمده است. لازم به ذکر است که مقدار follow\_type برابر با right میباشد.

* **دنبالگر راست:**

پس از source کردن و export کردن نوع ربات باید دستور زیر را وارد نمایید:

* roslaunch step3 control.launch follow\_type:=right

A screenshot of a computer

Description automatically generatedخروجی به صورت زیر خواهد شد:

* **دنبالگر چپ:**

پس از source کردن و export کردن نوع ربات باید دستور زیر را وارد نمایید:

* roslaunch step3 control.launch follow\_type:=right

A screenshot of a computer

Description automatically generatedخروجی به صورت زیر خواهد شد: