|  |  |  |
| --- | --- | --- |
| 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 سوم** | | |

|  |  |
| --- | --- |
| مهدی رحمانی | نام و نام خانوادگی |
| 9731701 | شماره‌ دانشجویی |
| 14/03/1402 | تاریخ ارسال گزارش |

­

**فهرست گزارش سوالات**

[بخش صفرم (آماده سازی ورک اسپیس برای پروژه) 3](#_Toc136736885)

[گام اول- بخش اول (توضیحات کد و آماده سازی) 4](#_Toc136736886)

[گام اول- بخش اول (اجرا و نتایج) 13](#_Toc136736887)

[گام اول-بخش دوم(توضیحات کد و آماده سازی) 20](#_Toc136736888)

[گام اول-بخش دوم (اجرا و نتایج) 32](#_Toc136736889)

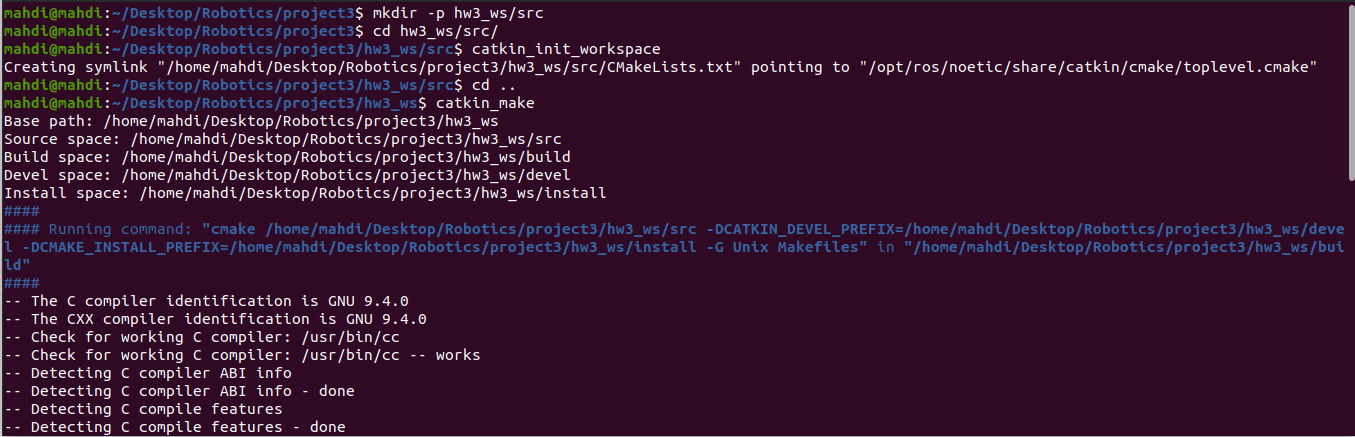
[گام دوم(توضیحات کد و آماده سازی) 34](#_Toc136736890)

[گام دوم (اجرا و نتایج) 44](#_Toc136736891)

[گام سوم(توضیحات کد و آماده سازی) 53](#_Toc136736892)

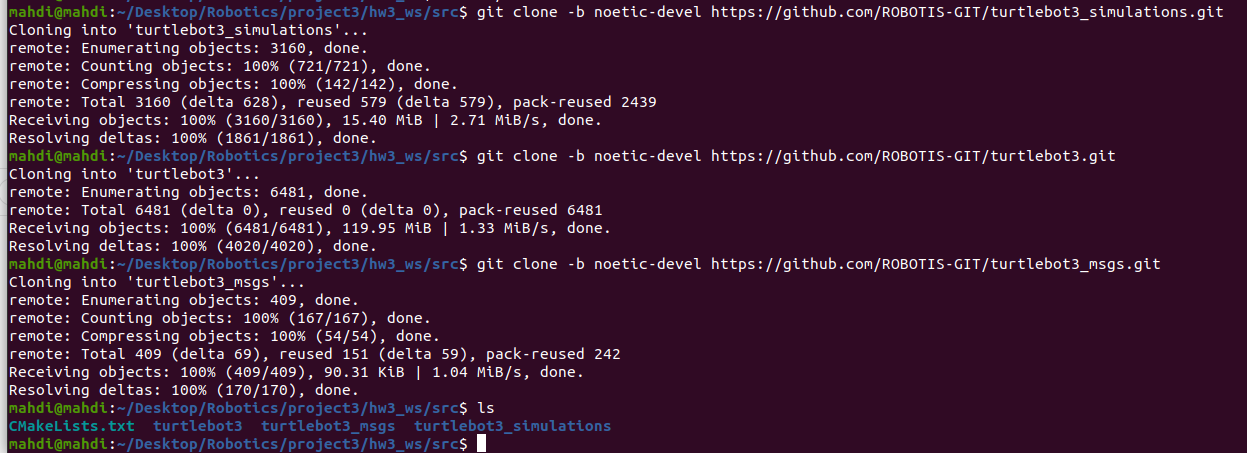
[گام سوم(اجرا و نتایج) 59](#_Toc136736893)

# بخش صفرم (آماده سازی ورک اسپیس برای پروژه)

ابتدا لازم است تا یک work space برای پروژه بسازیم و آن را initialize کنیم:

چون از turtlebot و شبیه ساز gazebo ممکن است استفاده کنیم لازم است ابتدا از لینک های زیر یک سری پکیج‌ها را در فولدر src خودمان دانلود 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>

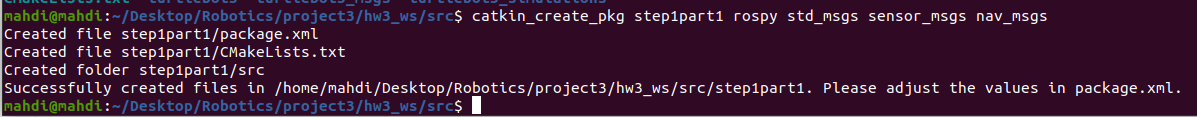
پس از دانلود اگر ls بزنیم لیست پکیج‌های داخل فولدر سورس را خواهیم دید:

همچنین در مسیر زیر یک فایل لانچ با نام my\_empty\_world.launch ایجاد میکنیم که محتوای آن شبیه فایل empty\_world که به صورت دیفالت در این مسیر است میباشد ولی کمی تغییرات دارد.

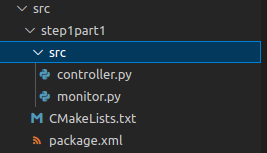
* hw3\_ws/src/turtlebot3\_simulations/turtlebot3\_gazebo/launch

# گام اول- بخش اول (توضیحات کد و آماده سازی)

حال لازم است که یک پکیج با نام step1part1 برای گام اول در فولدر سورس بسازیم. همچنین dependencyهای لازم را که ممکن است در نوشتن نودها و کد زنی نیاز شود به آن میدهیم:

* catkin\_create\_pkg step1part1 rospy std\_msgs sensor\_msgs nav\_msgs

حال به سراغ ساخت نودها میرویم. برای این منظور در فولدر src مربوط به step1part1 میرویم و دوفایل پایتون با نام های controller.py و monitor.py میسازیم. نودmonitor.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. ابتدا کد مربوط به controller را مینویسیم. در این قسمت، در تابع init خود نود را initialize کرده و در گام بعد این نود باید سرعت خطی و زاویه ای را برای ربات در قالب twist پابلیش کند پس نود را به عنوان پابلیشر معرفی میکنیم. سپس ضرایب Kp و Kd و Ki را برای کنترلرهای مختلف تعریف میکنیم تا آن‌ها را tune کنیم.

سپس مقدار dt که time step را برای گرفتن فیدبک مشخص میکند، را تعیین کردیم. مقدار D را هم 0 تعیین کردیم و میخواهیم در حقیقت به مقدار فاصله 0 از هدف و همچنین ارور زاویه‌ای 0 برسیم. ما در اینجا کنترلر را برای سرعت خطی و هم سرعت زاویه‌ای مینویسیم و میخواهیم در نهایت فاصله ما تا هدف برابر 0 باشد و ارور زاویه‌ای هم به مقدار 0 برسد. بنابراین gainهایی مثل k\_d\_l، k\_p\_l و k\_i\_l که به زیروند l مشخص شدند مربوط به سرعت خطی میباشند و gainهایی مانند k\_d\_a و k\_p\_a و k\_i\_a که با زیروند a مشخص شدند برای سرعت زاویه ای هستند. همچنین مقدار متغیر threshold برای آن است که بگوییم اگر تا این حد ( چه از نظر زاویه و چه از نظر فاصله) ربات به مقصد و مقدار دلخواه نزدیک شد برنامه را تمام کن و نمودارها را نمایش بده. نمیتوان شرط دقیقا 0 گذاشت چراکه اولا همیشه یک مقدار کمی خطا وجود خواهد داشت و اگر هم به نحوی قابل دسترس باشد زمانبر خواهد بود.

سپس یک تابع get\_heading و همچنین get\_pose داریم که به ترتیب زاویه yaw ربات را به رادیان و دیگری مکان x و y ربات را برحسب متر با خواندن مقادیر از تاپیک odom برمیگرداند.

یک تابع distance\_from\_goal داریم که فاصله اقلیدسی تا هدف را حساب میکند و خروجی میدهد.

تابع دیگر angle\_from\_goal میباشد. در این تابع ما ابتدا زاویه heading ربات را میابیم و همچنین زاویه نقطه هدف نسبت به ربات را هم می‌یابیم. اختلاف زاویه هدینگ و این زاویه desired میشود ارور زاویه‌ای ما. باید دقت شود چنانچه این مقدار بیشتر از 180 درجه یا کمتر از 180- درجه شد باید آن را از 360 کم کنیم و به عنوان ارور اعلام کنیم. چراکه به جای آنکه ربات از اسمت زاویه منفرجه به چرخش در بیاید که خطا را بالا میبرد و ربات ناپایدار میشود میتوان از زاویه حاده استفاده کرد.

در تابع controller مقدار ارور را برای فاصله و محاسبه سرعت خطی برابر با اختلاف distance حساب شده بین نقطه فعلی تا مقصد و همچپنین مقدار D تعریف میکنیم که 0 بود. یک مقدار sum\_i داریم که برابر خطای جمع شونده است و درواقع مساحت و انتگرال زیر نمودار err را حساب میکند که زمانی که بخواهیم از ترم انتگرال گیر استفاده کنیم به کار می اید. همین پارامترها را برای زاویه و محاسبه سرعت زاویه‌ای نیز جداگانه تعریف میشوند.

سپس باتوجه به مطالب داخل درس و فیلم‌های قرار داده شده، مقدار هریک از ترم‌های P و I و D را تعیین میکنیم. باتوجه به تعریف ارروی که داشتیم مقدار ترم P میشود حاصل ضرب ضریب Kp در آن ارور و همچنین مقدار ترم D میشود حاصل ضرب، ضریب Kd در نرخ تغییرات ارور( حالت مشتقی دارد) و مقدار I میشود ضربِ ضریب Ki در مقدار انتگرال گیری شده روی خطا که همان sum\_i بود. مجموع این 3 ترم در حالتی که مربوط به فاصله باشند سرعت خطی و درحالتی که مربوط به زاویه باشند سرعت زاویه‌ای را میسازند.

در نهایت هم میگوییم پس از تمام شدن اجرا مقدار خطا را در طول اجرا پلات کند. این کار به کمک تابع on\_shutdown انجام میشود. در کد زیر مقادیر نهایی ضرایب آمده است و در ادامه طریقه به دست آوردن هریک را میگوییم. دقت شود که برای استفاده از هر کنترلر باید مقدار ضرایب آن uncomment شود و بقیه کامنت شوند.

#!/usr/bin/python3

import rospy

import tf

from geometry\_msgs.msg import Twist

import matplotlib.pyplot as plt

from nav\_msgs.msg import Odometry

from math import atan2, sqrt, radians

class PIDController():

    def \_\_init\_\_(self):

        # initialize node and define it as a publisher

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

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

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

        # (linear Velocity)

        # P controller

        #self.k\_p\_l = 0.08

        #self.k\_i\_l = 0.0

        #self.k\_d\_l= 0.0

        # PD controller

        # self.k\_p\_l = 0.1

        # self.k\_i\_l = 0.0

        # self.k\_d\_l = 0.2

        # PID controller

        self.k\_p\_l = 0.1

        self.k\_i\_l = 0.0005

        self.k\_d\_l = 0.2

        # (angular Velocity)

        # P controller

        #self.k\_p\_a = 0.3

        #self.k\_i\_a = 0.0

        #self.k\_d\_a = 0.0

        # PD controller

        # self.k\_p\_a = 0.3

        # self.k\_i\_a = 0.0

        # self.k\_d\_a = 0.9

        # PID controller

        self.k\_p\_a = 0.3

        self.k\_i\_a = 0.121

        self.k\_d\_a = 0.9

        # goal pose

        self.x\_goal = 10

        self.y\_goal = 0

        self.D = 0

        self.threshold = 0.005

        self.dist\_errs = []

        self.angle\_errs = []

        self.dt = 0.005

        rate = 1/self.dt

        self.r = rospy.Rate(rate)

    def get\_pose(self):

        '''

        get x and y coordinate of position of the robot

        '''

        # waiting for the most recent message from topic /odom

        msg = rospy.wait\_for\_message("/odom" , Odometry)

        position = msg.pose.pose.position

        return position.x, position.y

    def get\_heading(self):

        '''

        get the yaw angle of robot in world.

        We call it, heading of the robot.

        '''

        # waiting for the most recent message from topic /odom

        msg = rospy.wait\_for\_message("/odom" , Odometry)

        orientation = msg.pose.pose.orientation

        # convert quaternion to odom

        roll, pitch, yaw = tf.transformations.euler\_from\_quaternion((

            orientation.x ,orientation.y ,orientation.z ,orientation.w

        ))

        return yaw

    def distance\_from\_goal(self):

        '''

        this function get us the current Euclidean distance from goal

        '''

        x\_curr, y\_curr = self.get\_pose()

        distance = sqrt((self.x\_goal-x\_curr)\*\*2 + (self.y\_goal-y\_curr)\*\*2)

        return distance

    def angle\_from\_goal(self):

        '''

        function below first calculate the heading angle and then the desired angle from current pose to goal pose.

        it means that robot heading must be equal to this angle for being in a

        correct direction. then return the difference between heading and desired angle.

        '''

        # find x and y of current position and find relative x and y to goal point

        x\_curr, y\_curr = self.get\_pose()

        relative\_x = self.x\_goal - x\_curr

        relative\_y = self.y\_goal - y\_curr

        # get heading of robot in radian

        heading = self.get\_heading()

        # now we should find the desired angle

        # desired angle tell us the angle of goal point relative to current point

        desired\_angle = 0

        if (relative\_x == 0 and relative\_y ==0):

            # this state (x=0 , y =0) is undefined so we handle it separately

            desired\_angle = heading

        else:

            desired\_angle = atan2(relative\_y, relative\_x)

        # the angle express howmuch we should rotate to reach the desired angle

        angle = heading - desired\_angle

        # but we design controller and if the angle is bigger than 180 or less than -180

        # the robot must rotate alot so we should find its complementary to 360 degrees

        if angle < radians(-180):

            angle = radians(360)-abs(angle)

        elif angle > radians(180):

            angle = angle-radians(360)

        return angle

    def control(self):

        '''

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

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

        terms define our linear and angular velocity.

        also we define a threshold and if the robot close to goal point and the goal and

        angle distance are less than that threshold the programe is terminated.

        '''

        distance = self.distance\_from\_goal()

        sum\_i\_dist = 0

        prev\_error\_dist = 0

        angle = self.angle\_from\_goal()

        sum\_i\_angle = 0

        prev\_error\_angle = 0

        move\_cmd = Twist()

        while not rospy.is\_shutdown():

            # linear velocity

            err\_dist = distance - self.D

            self.dist\_errs.append(err\_dist)

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

            P\_l = self.k\_p\_l \* err\_dist

            I\_l = self.k\_i\_l \* sum\_i\_dist

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

            move\_cmd.linear.x = P\_l + I\_l + D\_l

            prev\_error\_dist = err\_dist

            distance = self.distance\_from\_goal()

            rospy.loginfo(f"linear velocity")

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

            # angular velocity

            err\_angle = angle - self.D

            self.angle\_errs.append(err\_angle)

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

            P\_a = self.k\_p\_a \* err\_angle

            I\_a = self.k\_i\_a \* sum\_i\_angle

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

            move\_cmd.angular.z = -(P\_a + I\_a + D\_a)

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

            prev\_error\_angle = err\_angle

            angle = self.angle\_from\_goal()

            rospy.loginfo(f"angular velocity")

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

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

            if err\_dist < self.threshold and err\_angle < self.threshold:

                break

            self.r.sleep()

    def on\_shutdown(self):

        '''

        this method plot error of linear and angular velocity separately.

        '''

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

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

        # linear

        plt.plot(list(range(len(self.dist\_errs))), self.dist\_errs, label='dist\_errs')

        plt.axhline(y=0,color='R')

        plt.draw()

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

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

        plt.show()

        # angular

        plt.plot(list(range(len(self.angle\_errs))), self.angle\_errs, label='angle\_errs')

        plt.axhline(y=0,color='R')

        plt.draw()

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

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

        plt.show()

        rospy.sleep(1)

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

    try:

        pidc = PIDController()

        pidc.control()

    except rospy.ROSInterruptException:

        rospy.loginfo("Navigation terminated.")

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.

#!/usr/bin/python3

import rospy

from nav\_msgs.msg import Odometry, Path

from geometry\_msgs.msg import PoseStamped

class PathMonitor:

    def \_\_init\_\_(self) -> None:

        rospy.init\_node("monitor" , anonymous=False)

        self.path = Path()

        self.odom\_subscriber = rospy.Subscriber("/odom" , Odometry , callback=self.odom\_callback)

        self.path\_publisher = rospy.Publisher("/path" , Path , queue\_size=10)

    def odom\_callback(self, msg : Odometry):

        self.path.header = msg.header

        pose = PoseStamped()

        pose.header = msg.header

        pose.pose = msg.pose.pose

        self.path.poses.append(pose)

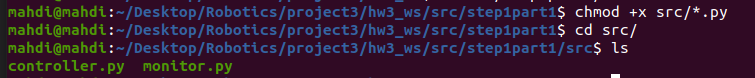
        self.path\_publisher.publish(self.path)

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

    path\_monitor = PathMonitor()

    rospy.spin()

در مرحله بعد باید تمامی کدهای پایتون را executable کنیم. برای این کار لازم است در ترمینال در پکیج step1part1 کد زیر را اجرا کنیم:

chmod +x src/\*.py

سپس به سراغ نوشتن لانچ فایل میرویم. در همین پکیج step1part1 لازم است تا یک فولدر launch ایجاد کنیم و داخل آن یک step1part1.launch ایجاد کنیم. لانچ فایل به صورت زیر است:

<launch>

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

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

        <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="step1part1" type="controller.py" name="controller" output="screen">

        <param name="angular\_speed" value="0.02"/>

    </node>

    <node pkg="step1part1" type="monitor.py" name="monitor"></node>

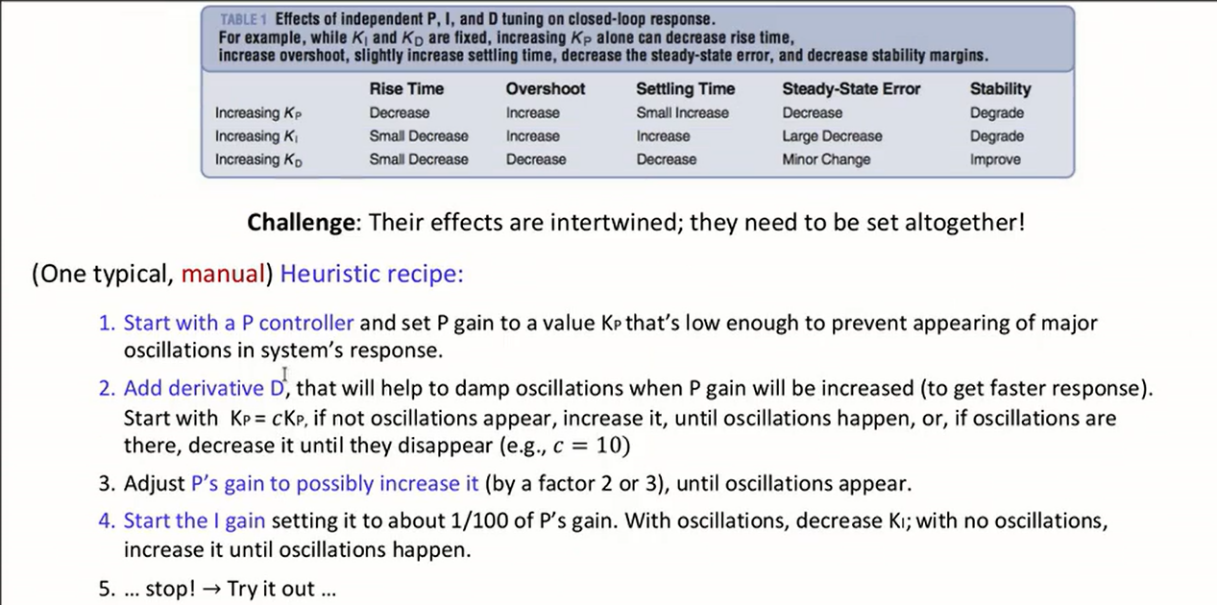
</launch>

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 step1part1 control.launch

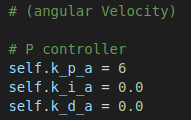
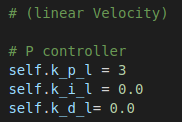
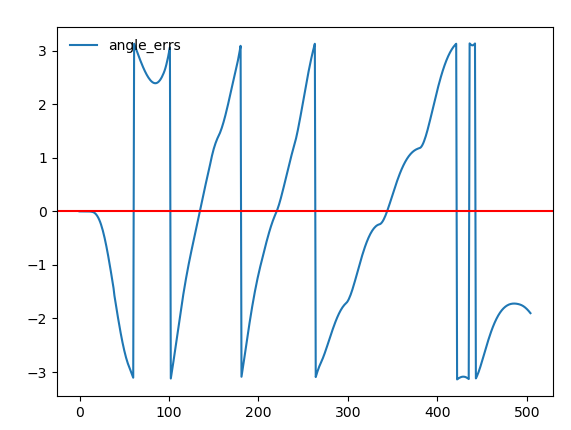
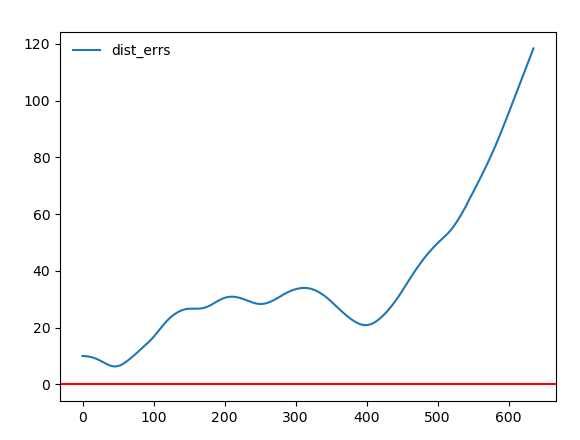
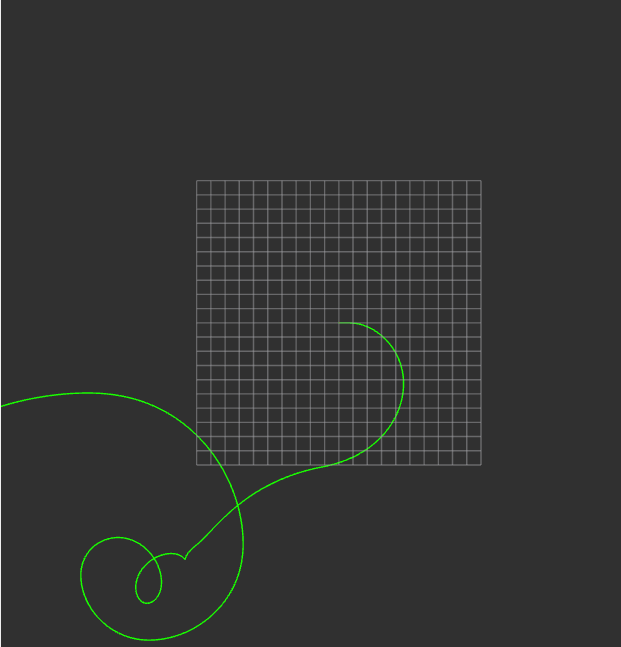
# گام اول- بخش اول (اجرا و نتایج)

**توضیحات به دست آوردن ضرایب برای هر کنترلر:**

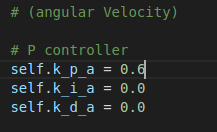
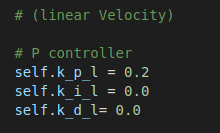
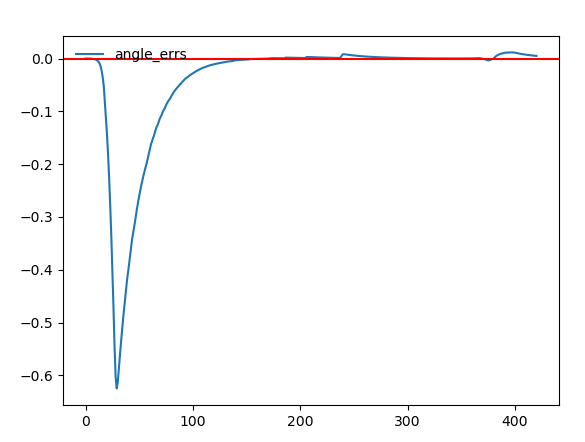
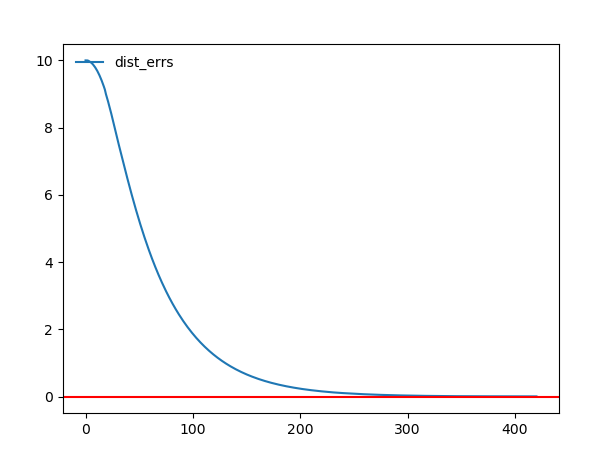
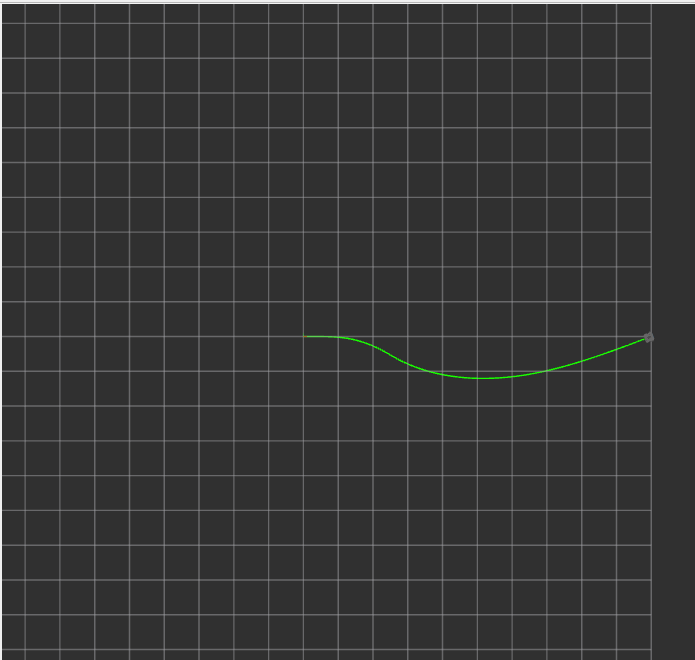
باتوج به اسلاید زیر مقدار ضرایب را تعیین میکنیم.

* **کنترلر P**

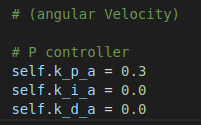
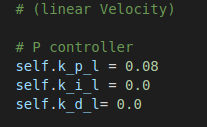
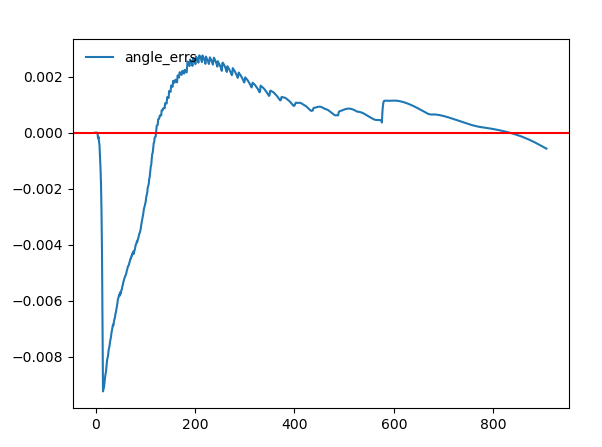
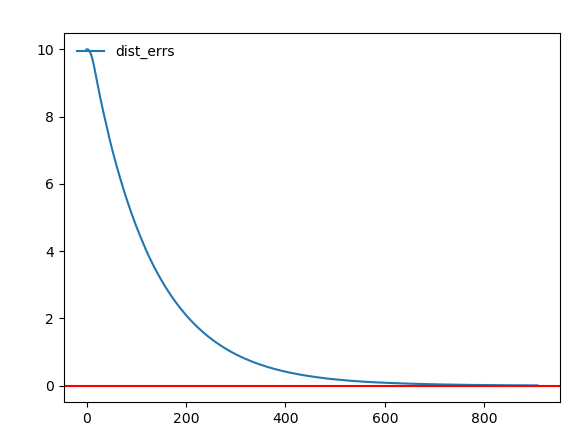
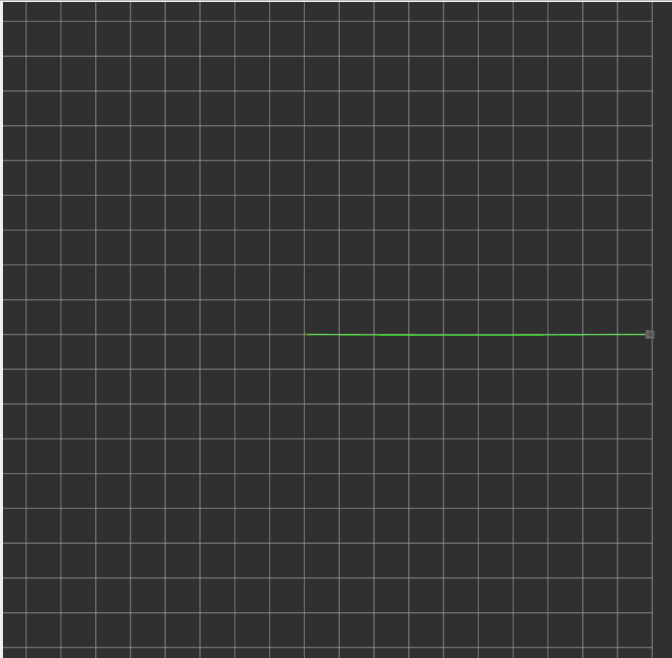
**حالت اول)** طبیعتا در این کنترلر مقادیر مربوط به K\_i و K\_d برابر با 0 میباشد. برای تعیین ترم proportional دقت شود که در اسلایدهای هم اشاره شده بود که از مقدار زیاد K\_p پرهیز شود. اگر مثلا مقدار K\_p\_l ( برای سرعت خطی) و K\_p\_a ( برای سرعت زاویه ای) را به ترتیب برابر 3و6 بگیریم، در این صورت خواهیم دید که oscillation زیاد خواهد بود و ارور ما زیاد است. این مورد را میتوانید در شکل مشاهده کنید.



**حالت دوم)** پس مقدار آنها را کمتر کرده و برای سرعت خطی و زاویه‌ای به ترتیب برابر با 0.2 و 0.6 میگذاریم. سپس نتیجه را مشاهده میکنیم. همانطور که دیده میشود ابتدای مسیر انحراف از مسیر مستقیم زیاد شده است ولیس در نهایت به مقصد رسیده است و ارور نزدیک به 0 شده است.



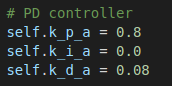
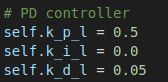
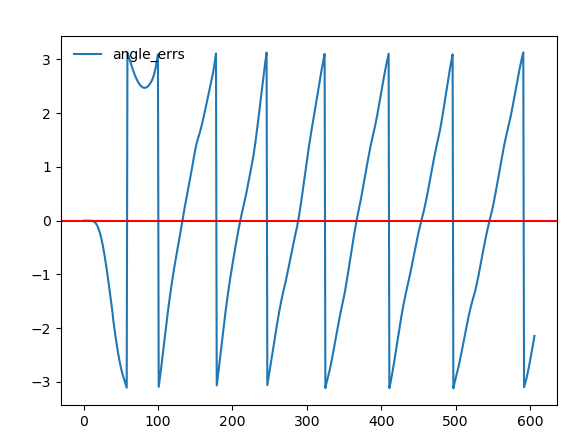
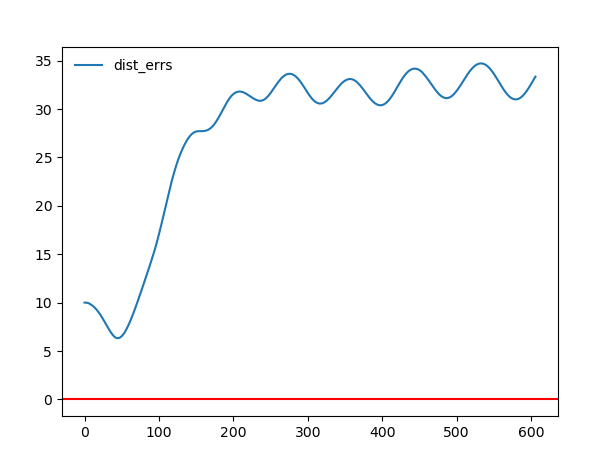
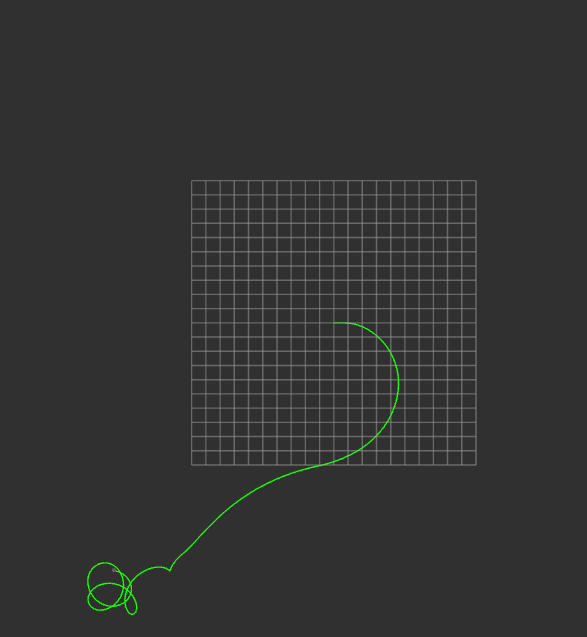
**حالت سوم (نهایی))** در حالت قبل به مقصد رسیدیم ولی در مسیر انحرافاتی داشتیم. مقدار را خیلی کمتر کرده و پس از کمی سعی و خطا به مقادیر 0.08 و 0.3 برای سرعت خطی و زاویه‌ای میرسیم. همانطور که میبینیم مقدار خطا به مرور کم شده و نمودار به 0 همگرا شده. دراین جا همانطور که دیده میشود ربات روی یک خط صاف حرکت کرده و از مسیر مستقیم منحرف نشدیم. برای دقت بیشتر میتوان threshold و همچنین مقادیر gainها را کمتر کرد تا ربات اهسته تر و با دقت بیشتر پیش برود.



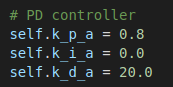
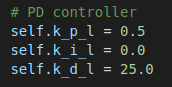
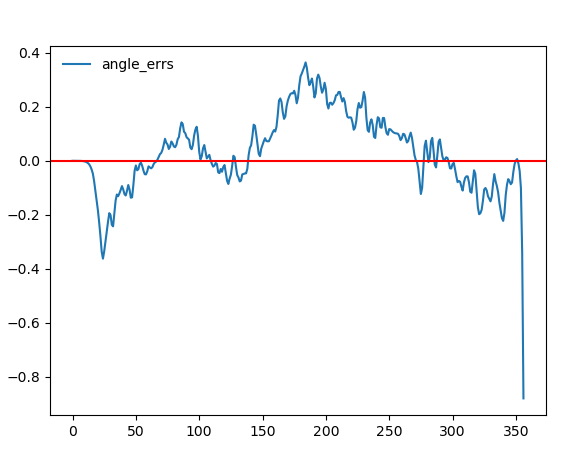
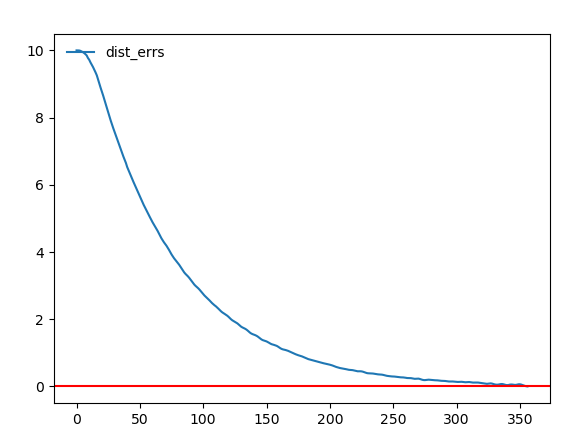
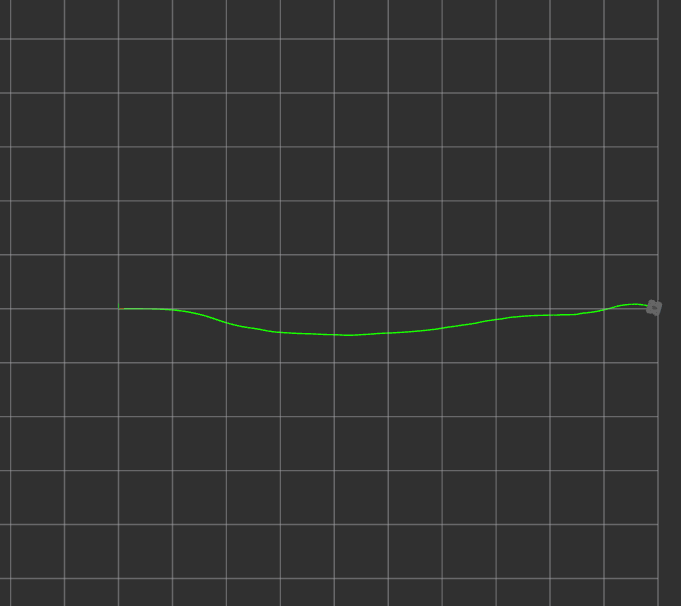
* **کنترلر PD**

میدانیم کاربرد D برای جلوگیری از Overshoot میباشد. طبق توضیحات اسلاید یک مقدار اولیه مناسب برای مقدار D برابر با 0.1 مقدار k\_p میباشد. در قسمت قبل چون صرفا کنترلر P داشتیم باید gain را کم میکردیم تا به دقت کافی برسیم اما اکنون که میتوان D داشت پس میتوان کمی ضرایب P را بالاتر برد و جلوی اورشوت را با D گرفت. درواقع هم در زمان کمتری به مقصد برسیم و هم دقت نسبتا خوبی داشته باشیم. ( طبیعتا با مقادیر K\_p پایین دقت بهتر است و مقدار K\_d کم هم جوابگوست ( حتی نبودشان هم اوکیه 😊 ))

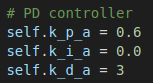
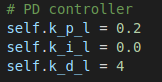
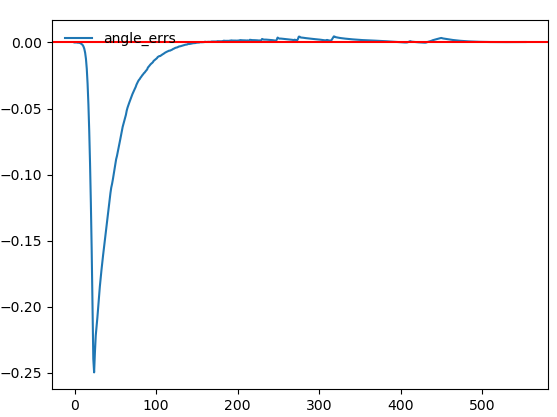
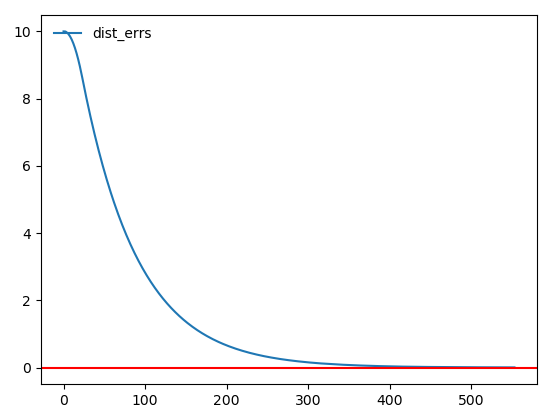
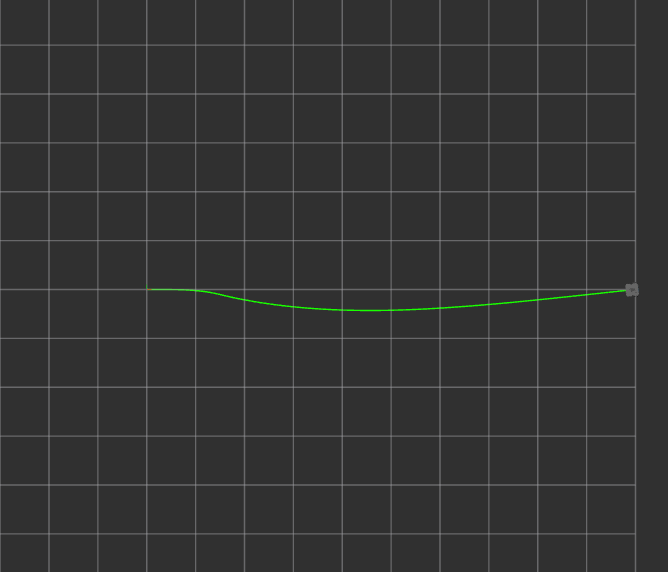
**حالت اول)** در این حالت ما مقدار K\_p را برای سرعت خطی و زاویه ای به ترتیب برابر با 0.5 و 0.8 میگذاریم و مقدارK\_d را برابر با 0.05 و 0.08 برای شروع تنظیم میکنیم. خواهیم دید که نتایج قابل قبول نیستند مقدار ارور بالا و نوسانی میباشد و درحقیقت مقدار gain مربوط به K\_d برای این مقادیر proportional خیلی کم است.



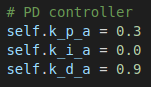
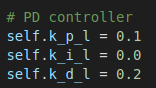
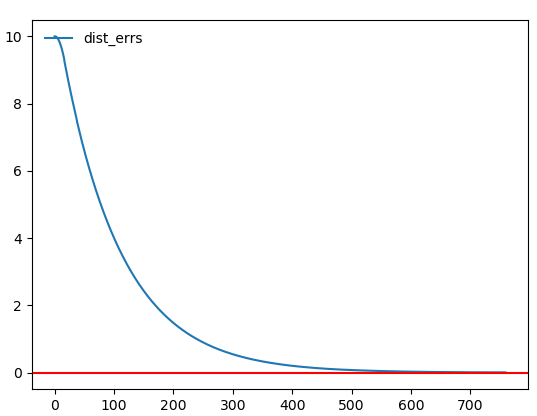
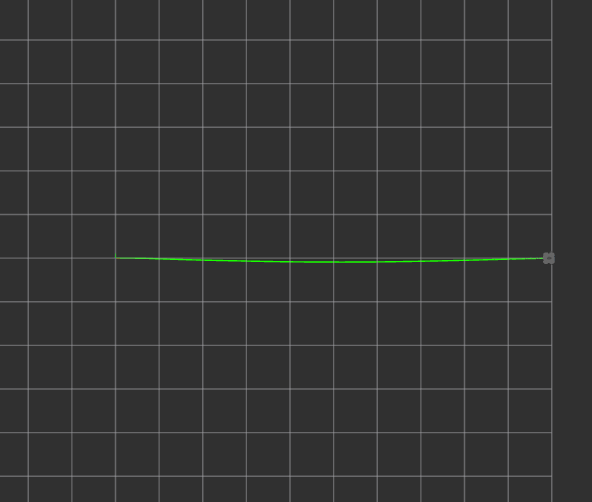
**حالت دوم)** حال با سعی و خطا اگر برای همین مقادیر K\_p مقادیر مربوط به K\_d را زیادتر کنیم و به ترتیب برای سرعت خطی و زاویه‌ای برابر با 25 و 20 شکل و نمودارهای زیر را مشاهده خواهید کرد. در این صورت در عین حال که سرعت بالا هست به مقصد هم میرسیم ولی همانطور که دیده میشود از مسیر انحراف داریم و همچنین نوسانات زیاد است. درواقع در اینجا چون مقادیر gainهای K\_d بالا هست نویز پذیری سیستم نیز افزایش میابد و درواقع ترم مشتق گیر اهمیت پیدا میکند و به تغییرات خیلی حساس است. بنابراین لازم است سرعت را کمتر کرده و از gainهای K\_d کمتری استفاده کنیم.



**حالت سوم)** برای همان حالتی که در P-controller داشتیم که مقادیر gainهای K\_p برای سرعت خطی و زاویه‌ای به ترتیب برابر با 0.2 و 0.6 بودند اگر gainهای K\_d را به ترتیب 0.02 و 0.06 گذاشته و با حدس و خطا آن را زیاد کنیم با مقادیر K\_d کمتر نسبت به قبل که برابر با 4 و3 هستند میتوان نتیجه زیر را گرفت. میتوان دید که مقدار Overshoot و انحراف از مسیر نسبت به حالتی که فقط K\_p داشتیم کمتر شده است. در نمودار ارور زاویه ای هم مقدار اورشوت از -0.6 در قسمت قبل به -0.25 کاهش یافته است.



**حالت چهارم(نهایی))** اگر بخواهیم روی مسیر مستقیم پیش برویم و در عین حال سیتم ما نویز پذیر نباشد لازم است مقادیر gianهای K\_p و K\_d کمتر باشند. در این حالت اگرچه سرعت کمتر است ولی دقت بیشتر است هرچند مقادیر gain کمتر از این و مقدار threshold کمتر در رسیدن به نقطه انتهایی کمتر از این هم باشند دقت بیشتر میشود. در اینجا برای ارور زاویه اگر دقت شود نسبت به حالت سوم برای P-controller مقدار overshoot نصفشده است.( با وجود اینکه مقدار K\_p هم زیاد شده و سرعت بیشتر است) همچنین در این حالت مقادیر K\_d مناسب است و خیلی بزرگ نیست که خیلی به نویز حساس باشد.

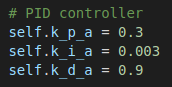
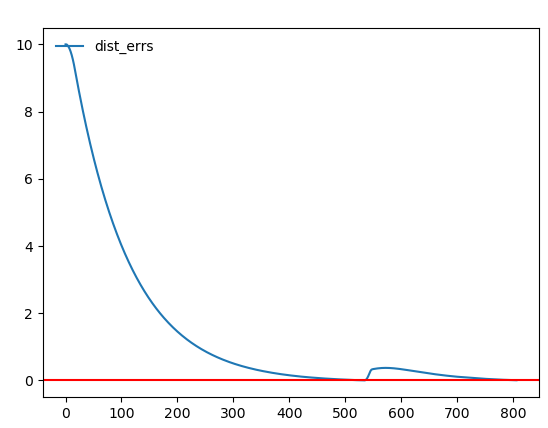
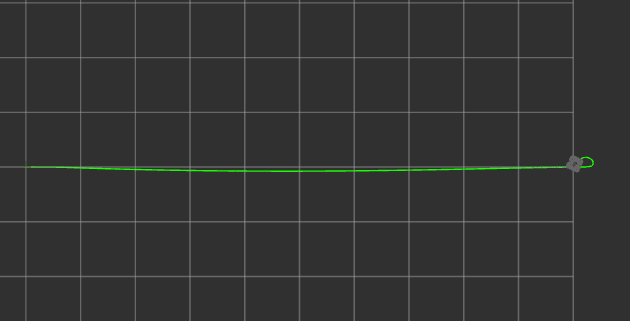


* **کنترلر PID**

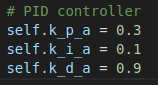
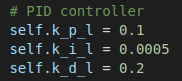
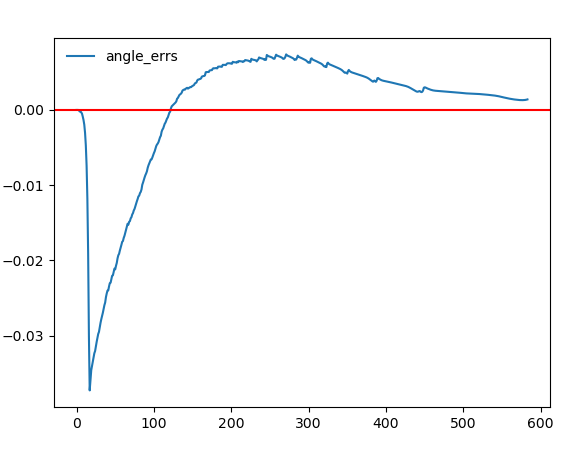
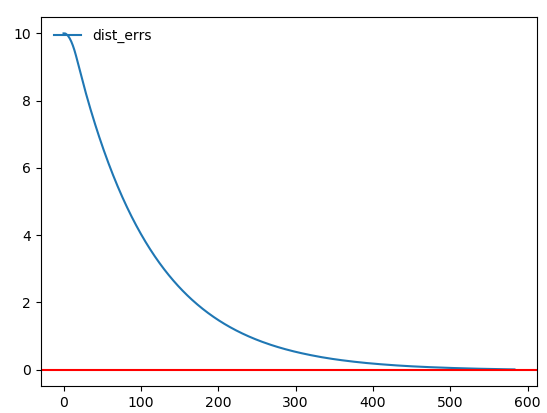
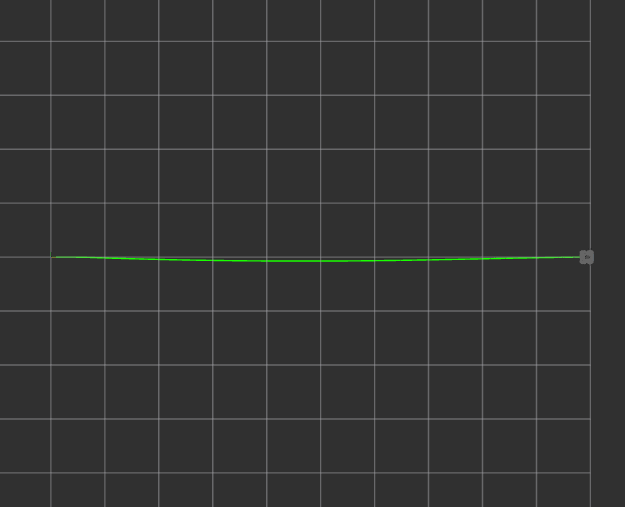
میدانیم که ترم I در PID برای برطرف کردن steady state error میباشد. همانطور که در فیلم هندزان هم دیدیم در اینجا ما خیلی با steady state error روبرو نیستیم و مثلا توی یک کوادکوپتر که گرانش g داریم شاید بیشتر مطرح باشد. به هرحال به stability روبات کمک میکند. همچنین دقت شود که I میتواند منجر به overshoot هم شود پس باید در استفاده از آن دقت داشت. طبق اسلاید مقدار اولیه مناسب برای شروع 0.01 مقدار K\_p میباشد.

**حالت اول)** مقادیر K\_p و K\_d مانند حالت چهارم کنترلر PD میماند و مقدار K\_i برای سرعت خطی و زاویه‌ای به ترتیب برابر با 0.001 و 0.003 میگیریم.

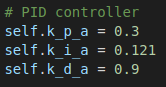
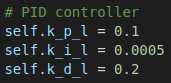
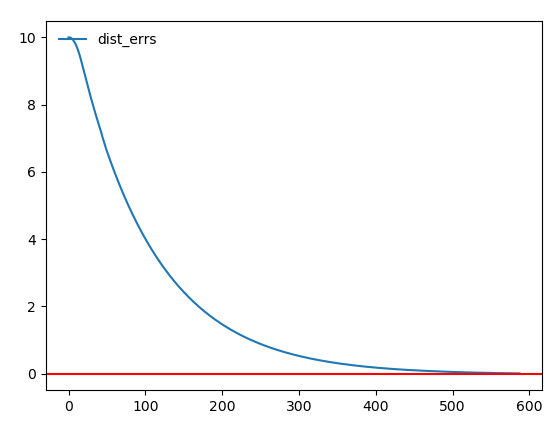
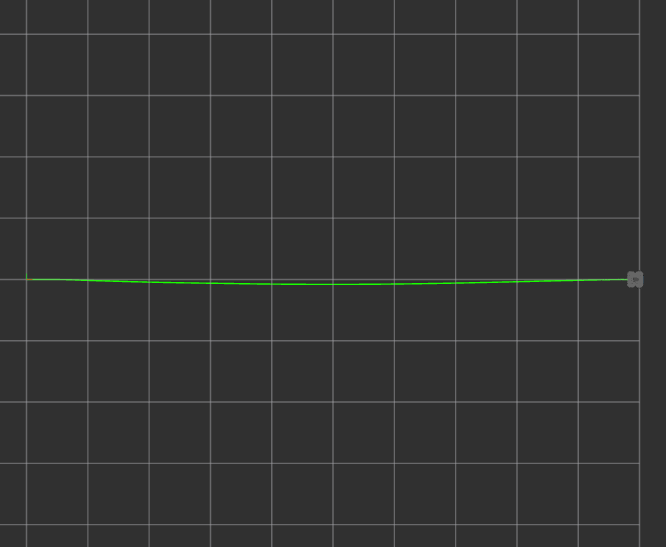
نتایج در صفحه بعد قابل مشاهده است. در این حالت به نظر میرسد مقدار k\_i\_l باید کمتر شود و به علت انباشت خطاهای قبلی باعث شده است که در انتها ربات به آرامی پیش نرود و مقصد را رد کند.



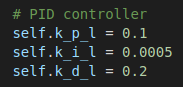
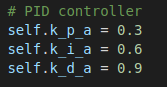
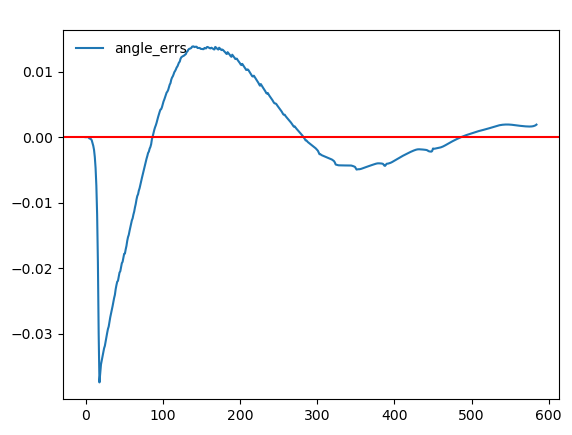
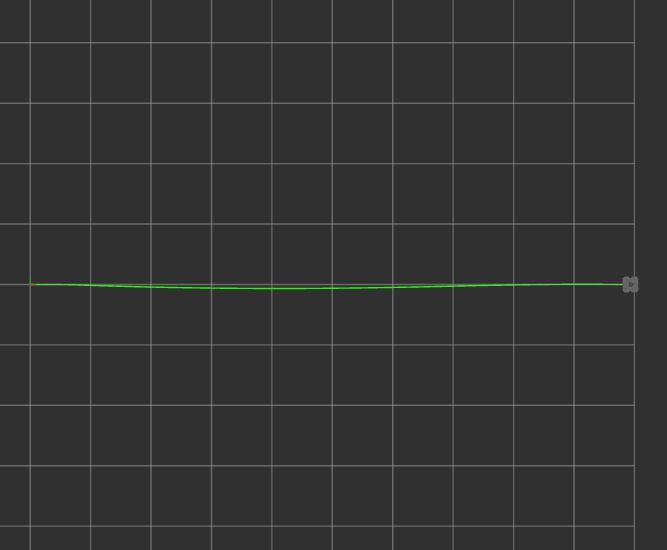
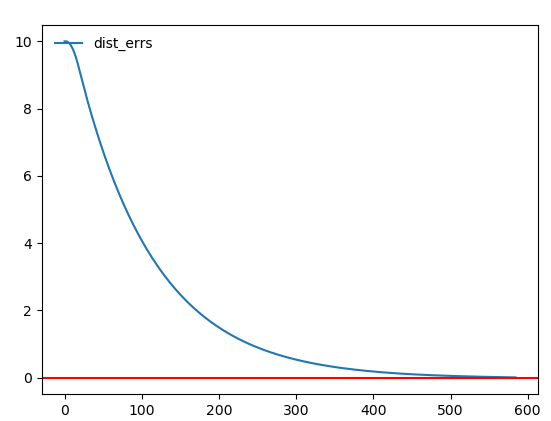
**حالت دوم)** دقت شود همانطور که از نمودارهای خطا میتوان متوجه شد خطای فاصله خیلی نیازی به اصلاح ندارد و باید مقدار k\_i\_l کمتر شود. مقدار k\_i\_a را بیشتر میکنیم. به ترتیب این دو را برابر با 0.0005 و 0.1 میگذاریم. در این صورت خروجی به صورت زیر میشود. همانطور که میبینیم کمی خطای steady state در زاویه داریم.



**حالت سوم ( نهایی) ):** حال برای برطرف کردن خطای steady state در زاویه با سعی و خطا کمی آن را افزایش میدهیم و به مقدار 0.121 میرسیم. اگر دقت شود در نهایت به آرامی خطای زاویه به 0 میرسد و مقادیر overshoot هم کم میباشد.

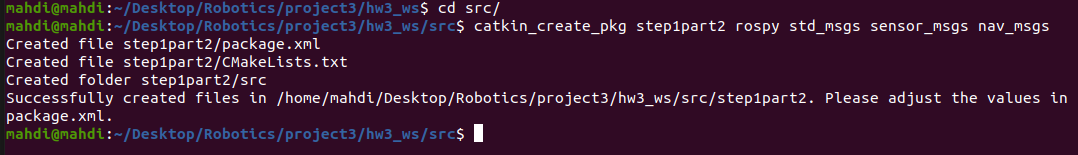


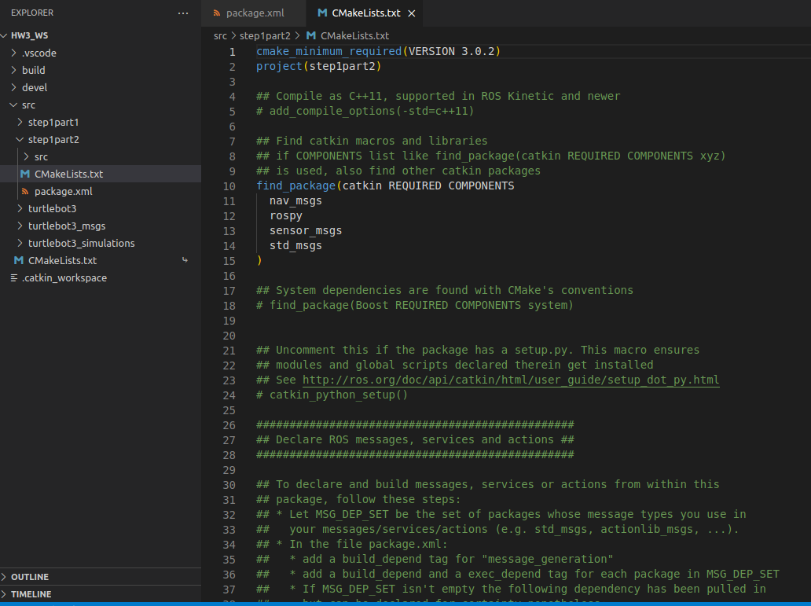
**حالت چهارم)** حال اگر مقدار k\_i\_a را بیشتر کنیم و برابر با 0.6 بگذاریم خواهیم دید که هم overshoot در نمودار خطای زاویه‌ای بیشتر میشود و هم ربات کمی حول خطای 0 نوسان کرده است و به آرامی همگرا نشده است که خوب نیست.



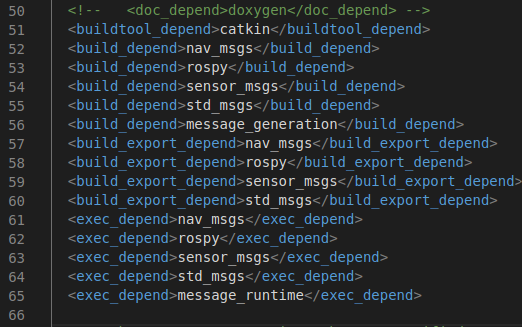
# گام اول-بخش دوم(توضیحات کد و آماده سازی)

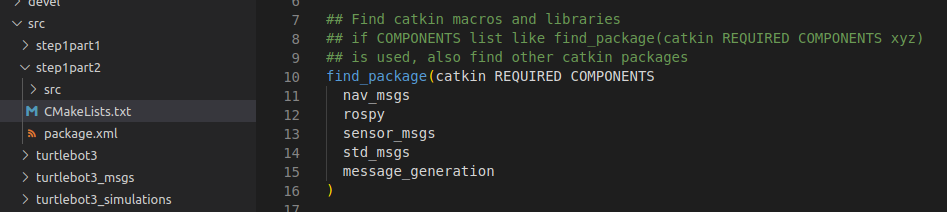
ابتدا لازم است یک پکیج برای این بخش نیز بسازیم و dependencyهایی که ممکن است به کار بیایند را هم اضافه کنیم.

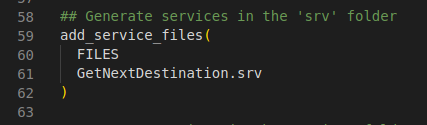
* catkin\_create\_pkg step1part2 rospy std\_msgs sensor\_msgs nav\_msgs

حال باید سرویس مورد نظر را درون پکیج step1part2 ایجاد کنیم. برای این کار کل ورک اسپیس را درون vs code باز میکنیم. از دورن پکیج step1part2 باید دو فایل pakages.xml و CMakeLists.txt را باز کنیم تا تغییرات لازم را اعمال کنیم:

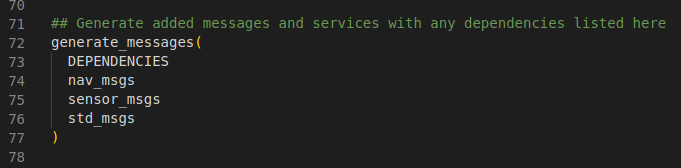
1. ابتدا در فایل package.xml در خط 55 اینتر زده و بعد build\_depend جدید برای message\_generation ایجاد کنیم و در خط 65 هم یک exec\_depend ایجاد میکنیم:



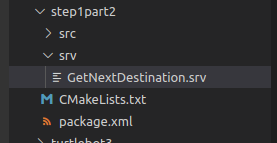
1. سپس در CMakeLists هم در خط 15 message\_generation را میگذاریم:
2. خطوط 59 تا 63 که مربوط به سرویس هست را uncomment میکنیم و تغییرات را اعمال میکنیم:



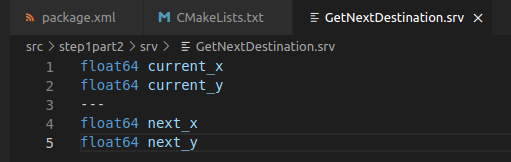
1. خطوط 72 تا 77 هم uncomment شوند:

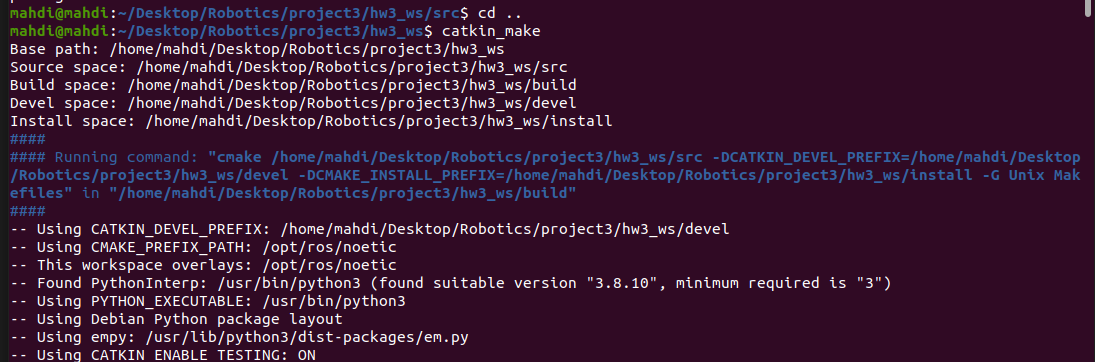


1. برای آنکه کارمان تکمیل شود، در همان پوشه step1part2 یک پوشه به نام srv درست میکنیم و بعد داخل آن یک فایل با نام اونی که در استپ 3 مشخص کردیم میذاریم.

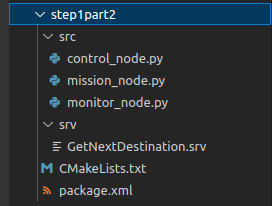


1. توی فایل فوق لازمه که ورودی ها و خروجی های سرویس را بگوییم. بین ورودی ها و خروجی ها با --- جدا میشود. اگرم ورودی نداره خط اول همون --- میشود.



1. حال به پوشه hw3\_ws در ترمینال برگشته و catkin\_make میکنیم.

حال به سراغ ساخت نودها میرویم. برای این منظور در فولدر src مربوط به step1part2 میرویم و دوفایل پایتون با نام های mission\_node.py و control\_node.py میسازیم. همچنین یک نود monitor\_node.py نیز برای نمایش مسیر ربات در rviz میسازیم.



حال در گام بعد به سراغ نوشتن کد مربوط به هریک از نودها میرویم. ابتدا کد مربوط به mission را مینویسیم. کد آن به صورت زیر است. در آن یک تابع get\_next\_dest را داریم که به عنوان call\_back برای سرویس ما میباشد. در این متد با توجه به اینکه میخواهیم مقدار پردازش کمینه باشد. ابتدا x را در بازه 20- تا 20 به طور رندوم تولید میکنیم و برای y باتوجه به اینکه استیت بعدی باید حداقل 10 متری این یکی باشد پس باید در بیرون از یک دایره با شعاع 10 متری آن باشد. روش دیگر این بود که دائما عدد رندوم تولید کنیم به نحوی که فاصله بیشتر از 10 شود ولی خب ممکن است بدشانس باشیم و خیلی طولانی شود و میدانیم که برای هندل کردن سرویس نباید کار پردازشی زیادی داشت.

#!/usr/bin/python3

import rospy

from step1part2.srv import GetNextDestination, GetNextDestinationResponse

import random

class Mission():

    def \_\_init\_\_(self):

        # minimum distance from next point

        self.min\_distance = 10

        # maximum and minimum x and y in grid

        self.max\_x\_y = 20

        self.min\_x\_y = -20

    def get\_next\_dest(self, current\_loc):

        cur\_x = current\_loc.current\_x

        cur\_y = current\_loc.current\_y

        rospy.loginfo(f"Service : NEW CALL: {cur\_x, cur\_y}")

        next\_x = random.uniform(self.min\_x\_y, self.max\_x\_y)

        # if random x is far enough so we don't need to calculate Euclidean distance and cosidering y

        if abs(next\_x - cur\_x)>=self.min\_distance:

            next\_y = random.uniform(self.min\_x\_y, self.max\_x\_y)

        else:

            y\_dist = (self.min\_distance\*\*2 - (next\_x - cur\_x)\*\*2)\*\*0.5

            valid\_min\_y1 = cur\_y + y\_dist

            valid\_min\_y2 = cur\_y - y\_dist

            if valid\_min\_y1 >= self.max\_x\_y:

                next\_y = random.uniform(self.min\_x\_y, valid\_min\_y2)

            elif valid\_min\_y2 <= self.min\_x\_y:

                next\_y = random.uniform(valid\_min\_y1, self.max\_x\_y)

            else:

                next\_y = random.choice([random.uniform(self.min\_x\_y, valid\_min\_y2), random.uniform(valid\_min\_y1, self.max\_x\_y)])

        result = GetNextDestinationResponse()

        result.next\_x = next\_x

        result.next\_y = next\_y

        return result

def listener():

    rospy.init\_node('mission', anonymous=True)

    mi = Mission()

    s = rospy.Service('get\_destination', GetNextDestination, mi.get\_next\_dest)

    rospy.spin()

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

    listener()

کد بعدی مربوط به کنترلر میباشد. کد کنترلر نیز تا حد زیادی شبیه به بخش قبل میباشد. در این قسمت، در تابع init خود نود را initialize کرده و سپس این نود را به عنوان client برای سرویس معرفی میکنیم و در گام بعد این نود باید سرعت خطی و زاویه ای را برای ربات در قالب twist پابلیش کند. همچنین متغیرهایی برای نگهداری موقعیت فعلی و همچنین هدف فعلی تعریف میشوند و همچنین متغیرهایی برای ضرایب gainهای PID مربوط به هریک از کنترلرهای سرعت خطی و زاویه‌ای تعریف میشوند. دو تابع get\_pose و get\_heading به ترتیب مکان فعلی و همچنین زاویه هدینگ ربات را مشخص میکنند و برمیگردانند. تابع distance\_from\_goal در حقیقت فاصله فعلی ربات را تا نقطه هدف تعیین شده به کمک فرمول فاصله اقلیدسی به دست می‌آورد و همچنین تابع angle\_from\_goal ابتدا زاویه ای که ربات باید نسبت به نقطه هدف داشته باشد را میابد و سپس مقدار زاویه هدینگ را از آن کم میکند و حاصل را برمیگرداند که در حقیقت زاویه‌ای است که ربات باید بچرخد تا در جهت درست قرار گیرد. در تابع control لازم است تا در 4 مرتبه مقصد جدید تعیین کنیم که شرط تعیین مقصد جدید آن است که فاصله و زاویه از هدف از یک thresholdای کمتر باشد. همچنین باتوجه به gainهای تعیین شده و خطای فاصله و زاویه‌ای که داریم در هر time step مقادیر P و I وD تعیین شده و از جمع این مقادیر، مقدار سرعت خطی و زاویه‌ای تعیین میشود. برای هر مسیر جدید نیز دقت شود که پارامترهای مربوط به ارورهای قبلی و ارورهای جمع شوند باید reset شوند و در مسیر جدید از 0 شروع شوند.

#!/usr/bin/python3

import rospy

import tf

import math

from nav\_msgs.msg import Odometry

from geometry\_msgs.msg import Twist

import matplotlib.pyplot as plt

from step1part2.srv import GetNextDestination, GetNextDestinationRequest

class PIDController():

    def \_\_init\_\_(self) -> None:

        rospy.init\_node("controller" , anonymous=False)  # initialize node

        rospy.wait\_for\_service('get\_destination')        # wait for response from service

        self.calc\_client = rospy.ServiceProxy('get\_destination', GetNextDestination)

        self.cmd\_publisher = rospy.Publisher('/cmd\_vel' , Twist , queue\_size=10)  # this node also is a publisher

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

        self.current\_x = 0

        self.current\_y = 0

        # linear velocity PID gains

        self.k\_p\_l = 0.05

        self.k\_i\_l = 0.0005

        self.k\_d\_l = 0.1

        # angular velocity PID gains

        self.k\_p\_a = 0.2

        self.k\_i\_a = 0.005

        self.k\_d\_a = 0.4

        # threshold for getting next point

        self.dist\_threshold = 0.035

        self.angle\_threshold = 0.7

        # define goal variables

        self.x\_goal = 0

        self.y\_goal = 0

        # arrays contains all errors in simulation time used for plotting

        self.dist\_errs = []

        self.angle\_errs = []

        # time step of get feedback

        self.dt = 0.005

        # the desired value

        self.D = 0

        rate = 1/self.dt

        self.r = rospy.Rate(rate)

    def get\_heading(self):

        '''

        get the yaw angle of robot in world.

        We call it, heading of the robot.

        '''

        # waiting for the most recent message from topic /odom

        msg = rospy.wait\_for\_message("/odom" , Odometry)

        orientation = msg.pose.pose.orientation

        # convert quaternion to odom

        roll, pitch, yaw = tf.transformations.euler\_from\_quaternion((

            orientation.x ,orientation.y ,orientation.z ,orientation.w

        ))

        return yaw

    def get\_pose(self):

        '''

        get x and y coordinate of position of the robot

        '''

        # waiting for the most recent message from topic /odom

        msg = rospy.wait\_for\_message("/odom" , Odometry)

        position = msg.pose.pose.position

        return position.x, position.y

    def distance\_from\_goal(self):

        '''

        this function get us the current Euclidean distance from goal

        '''

        x\_curr, y\_curr = self.get\_pose()

        distance = math.sqrt((self.x\_goal-x\_curr)\*\*2 + (self.y\_goal-y\_curr)\*\*2)

        return distance

    def angle\_from\_goal(self):

        '''

        function below first calculate the heading angle and then the desired angle from current pose to goal pose.

        it means that robot heading must be equal to this angle for being in a

        correct direction. then return the difference between heading and desired angle.

        '''

        # find x and y of current position and find relative x and y to goal point

        x\_curr, y\_curr = self.get\_pose()

        relative\_x = self.x\_goal - x\_curr

        relative\_y = self.y\_goal - y\_curr

        # get heading of robot in radian

        heading = self.get\_heading()

        # now we should find the desired angle

        # desired angle tell us the angle of goal point relative to current point

        desired\_angle = 0

        if (relative\_x == 0 and relative\_y ==0):

            # this state (x=0 , y =0) is undefined so we handle it separately

            desired\_angle = heading

        else:

            desired\_angle = math.atan2(relative\_y, relative\_x)

        # the angle express howmuch we should rotate to reach the desired angle

        angle = heading - desired\_angle

        # but we design controller and if the angle is bigger than 180 or less than -180

        # the robot must rotate alot so we should find its complementary to 360 degrees

        if angle < math.radians(-180):

            angle = math.radians(360)-abs(angle)

        elif angle > math.radians(180):

            angle = angle-math.radians(360)

        return angle

    def get\_next\_goal(self):

        '''

        This method first find current x and y and then

        use them as inputs for send request to service and

        get next goal. The goal must be in range (-20,20) for

        x and y and also has minimum distance of 10 at least.

        '''

        self.current\_x, self.current\_y = self.get\_pose()

        req = GetNextDestinationRequest()

        req.current\_x = self.current\_x

        req.current\_y = self.current\_y

        rospy.loginfo(f"Client : current pose: {req.current\_x, req.current\_y}")

        resp = self.calc\_client(req)

        self.x\_goal, self.y\_goal = resp.next\_x, resp.next\_y

        rospy.loginfo(f"Client : Goal pose : {resp.next\_x, resp.next\_y}")

    def control(self):

        '''

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

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

        terms define our linear and angular velocity.

        also we define two thresholds and if the robot close to goal point and the

        goal distance and angle distance are less than that thresholds then we should

        find new goal and move toward it. This process is happened 4 times and then

        the program is terminated.

        '''

        distance = self.distance\_from\_goal()

        sum\_i\_dist = 0

        prev\_error\_dist = 0

        err\_dist = distance - self.D

        angle = self.angle\_from\_goal()

        sum\_i\_angle = 0

        prev\_error\_angle = 0

        err\_angle = angle - self.D

        move\_cmd = Twist()

        counter = 0

        while not rospy.is\_shutdown():

            # If distance from current goal is less than threshold then define new goal

            if err\_dist < self.dist\_threshold and err\_angle < self.angle\_threshold:

                counter += 1

                if counter > 1:

                    rospy.loginfo(f"Goal{counter-1} final error dist: {err\_dist } final err\_angle : {err\_angle}")

                    rospy.loginfo(f"------------------------------------")

                # In this task we should go to a new goal for 4 times

                if counter == 5:

                    break

                self.get\_next\_goal()

                # the previous path errors aren't important for us.

                # So initialize related parameters again

                sum\_i\_angle = 0

                prev\_error\_angle = 0

                sum\_i\_dist = 0

                prev\_error\_dist = 0

            # linear velocity

            distance = self.distance\_from\_goal()

            err\_dist = distance - self.D

            self.dist\_errs.append(err\_dist)

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

            P\_l = self.k\_p\_l \* err\_dist

            I\_l = self.k\_i\_l \* sum\_i\_dist

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

            move\_cmd.linear.x = P\_l + I\_l + D\_l

            prev\_error\_dist = err\_dist

            #rospy.loginfo(f"linear velocity")

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

            # angular velocity

            angle = self.angle\_from\_goal()

            err\_angle = angle - self.D

            self.angle\_errs.append(err\_angle)

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

            P\_a = self.k\_p\_a \* err\_angle

            I\_a = self.k\_i\_a \* sum\_i\_angle

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

            move\_cmd.angular.z = -(P\_a + I\_a + D\_a)

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

            prev\_error\_angle = err\_angle

            #rospy.loginfo(f"angular velocity")

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

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

            #rospy.loginfo(f"error\_angle : {err\_angle} error\_dist: {err\_dist}")

            self.r.sleep()

    def on\_shutdown(self):

        '''

        this method plot error of linear and angular velocity separately.

        '''

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

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

        # linear

        plt.plot(list(range(len(self.dist\_errs))), self.dist\_errs, label='dist\_errs')

        plt.axhline(y=0,color='R')

        plt.draw()

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

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

        plt.show()

        # angular

        plt.plot(list(range(len(self.angle\_errs))), self.angle\_errs, label='angle\_errs')

        plt.axhline(y=0,color='R')

        plt.draw()

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

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

        plt.show()

        rospy.sleep(1)

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

    try:

        pidc = PIDController()

        pidc.control()

    except rospy.ROSInterruptException:

        rospy.loginfo("Navigation terminated.")

در monitor کد پایتون مربوط به رسم مسیری که ربات آن را طی میکند میباشد که چون کد آن تغییری نکرده است و در بخش 1 توضیح دادیم از توضیح مجدد آن خودداری میکنم.

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

* A screen shot of a computer code

  Description automatically generated with low confidencechmod +x src/\*.py

سپس به سراغ نوشتن لانچ فایل میرویم. در همین پکیج step1part2 لازم است تا یک فولدر launch ایجاد کنیم و داخل آن یک control.launch ایجاد کنیم. لانچ فایل به صورت زیر است. ابتدا mission\_node.py را لازم است بالا بیاوریم. میدانیم پکیج آن step1part2 است و همچنین خروجی‌های آن را میخواهیم در ترمینال چاپ کند پس screen مینویسیم.

سپس یک لانچ فایل دیگر را include میکنیم و میگوییم در آن آرگومان های ورودی مثل مکان اولیه ربات و زاویه اولیه چه باشند . اصلا در کدام نقشه ( که اینجا empty\_world بود) را بالا بیاورد. و ربات و gazebo را آماده کند. همچنین لانچ فایل مربوط به rviz را هم صدا میزنیم.

سپس mission\_node.py سپس control\_node.py را لازم است بالا بیاوریم. در نود کنترلر پارامتر linear\_speed که میتواند توسط کاربر وارد شود را مینویسیم. در نهایت لازم است نود monitor که برای رسم مسیر لازم بود بالا بیاوریم.

در زیر میتوانید لانچ فایل را مشاهده کنید:

<launch>

    <node pkg="step1part2" type="mission\_node.py" name="mission" output="screen"></node>

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

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

        <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="step1part2" type="control\_node.py" name="controller" output="screen"></node>

    <node pkg="step1part2" type="monitor\_node.py" name="monitor"></node>

</launch>

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
* A screen shot of a computer

  Description automatically generated with medium confidenceroslaunch step1part2 control.launch

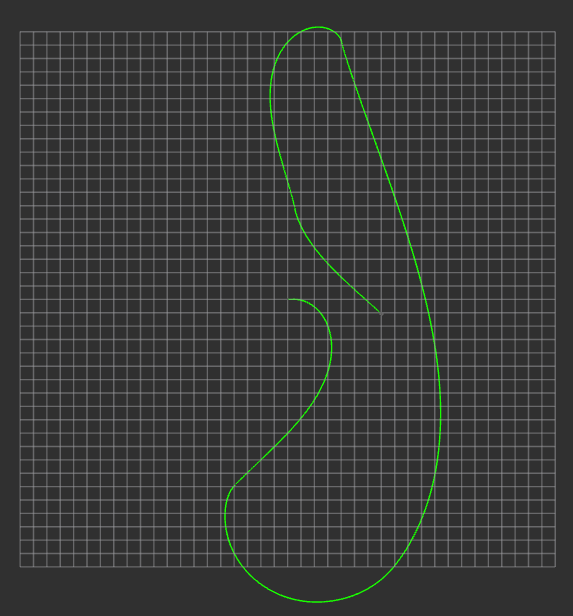
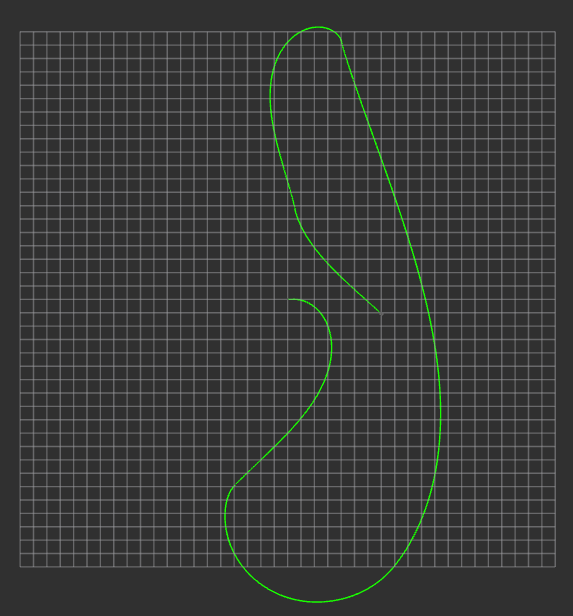
# گام اول-بخش دوم (اجرا و نتایج)

در این قسمت gainهای PID مربوط به بخش 1 را مبنای کار خود قرار دادیم ولی خب در بخش قبل چون ربات با زاویه yaw=0 قرار داده شده بود و روبروی goal point بود کمی حالت خاص محسوب میشد و اینجا ممکن است زاویه دوران و همچنین فاصله برای رسیدن به نقطه هدف بیشتر باشد لازم است تا کمی gainها را تغییر دهیم. برای آنکه دقت بیشتر شود و مسیرهای طی شده تا رسیدن به مقصد خیلی عجیب و پر از پیچ نباشد کمی مقدار gainهای k\_p\_l و k\_p\_a را کم کردیم تا ربات با سرعت خطی و دورانی کمتری پیش برود و تا خطا کمتر شود. به همین نسبت مقادیر مربوط به K\_d و k\_i برای هردو سرعت خطی و زاویه‌ای نیز تغییر میکند و کمتر میشود. در نهایت مقادیر gainهای زیر به نظر خوب میباشند. لازم به ذکر است که این مقادیر با سعی و خطا به دست آمدند.

A screenshot of a computer screen

Description automatically generated with low confidence

حال اگر لانچ فایل مربوطه را ران کنیم خروجی زیر را میگیریم. ( فیلم ضبط شده نیز مربوط به همین اجرا میباشد.) در تصویر راست نقاط goal به همراه مختصات مشخص شده اند. همانطور که میبنید ربات با دقت خوبی به هدف ها رسیده است.



G\_1: (-4.02, -13.98)

G\_2: (4.03, 19.25)

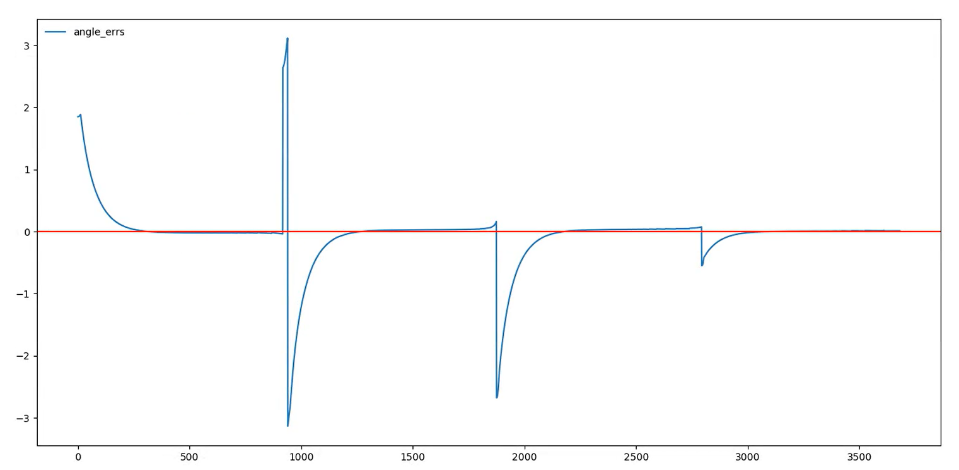
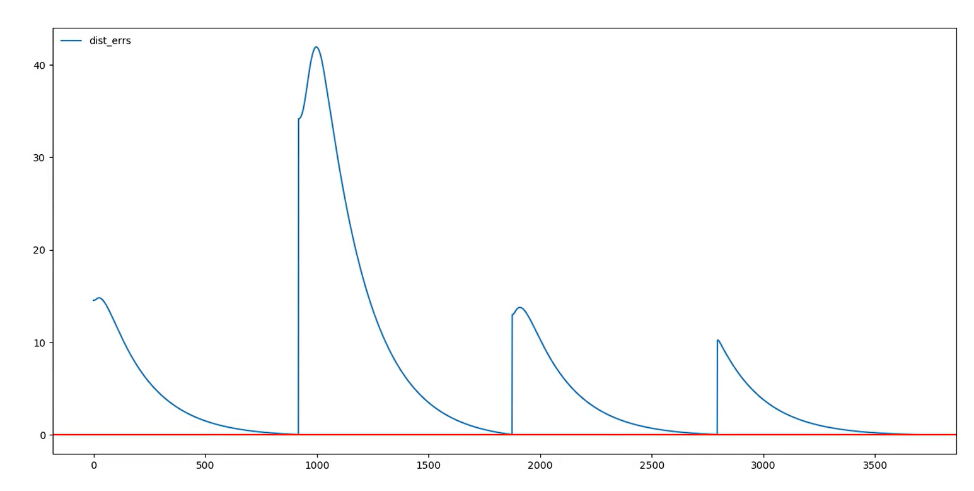
G\_3: (0.55, 6.7)

G\_4: (7.07, -1.15)

A picture containing text, screenshot, menu, font

Description automatically generatedدر تصویر زیر میتوانید نقاط هدفی که توسط سرویس به ربات داده شده است، به همراه خطای فاصله و زاویه (البته توجه شود که در این بخش اینکه با چه زاویه‌ای به مقصد برسد مهم نبوده است) در نقطه‌ای که ربات به threshold مد نظر رسیده است را مشاهده کنید. این فرآیند 4 بار تکرار شده است.

در نهایت نیز نمودارهای خطای فاصله‌ای و زاویه‌ای به صورت زیر میباشند. با رسیدن به مقصد خطای زاویه و فاصله هردو به سمت 0 میروند.



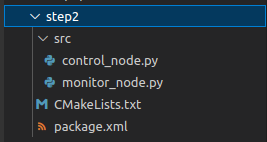
# گام دوم(توضیحات کد و آماده سازی)

در این قسمت نیز ابتدا یک پکیج با نام step2 ایجاد میکنیم و dependencyهایی که ممکن است به کار بیایند را هم اضافه کنیم.

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

A picture containing text, screenshot, font

Description automatically generated

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 قبل میباشد و توضیح آن در قسمت‌های قبل داده شد. در این جا فقط به کد نود control\_node میپردازیم. اکثر توضیحات کنترلر نیز مانند قبل میباشد. تغییراتی که این کد دارد به این قرار است: در اینجا یک متغیر shape داریم که کاربر به عنوان ورودی در ترمینال وارد میکند و مقدار دیفالت آن rectangle میباشد. بر اساس اینکه چه شکلی انتخاب شود توابع مرتبط با آن شکل یک سری way point روی آن شکل با ابعاد مشخص ایجاد میکنند و x و y نقاط را در قالب یک لیست برمیگردانند و در متغیر points ذخیره میشود. کلیات کار مانند قبل است. در اینجا کمی تغییرات در مقادیر gainهای سرعت زاویه‌ای و سرعت خطی داشتیم که در قسمت اجرا و نتایج علت آن‌ها توضیح داده شده است. در تابع control کلیات مانند قبل است با این تفاوت که در اینجا ابتدا ربات باید نزدیک ترین نقطه روی شکل را پیدا کند و به سمت آن برود که به کمک تابع find\_nearest\_point انجام میشود و زمانی که از یک حد thresholdای به شکل نزدیک شد دیگر باید نقاط روی شکل را دنبال کند. دقت شود که تابع find\_nearest\_point را باید دائما تا زمانیکه به آن threshold برسیم صدا بزنیم چراکه ربات ما وقتی شروع به حرکت میکند لزوما روی یک خط مستقیم به سمت هدف نمیرود و بنابراین ممکن است نزدیکترین نقطه دائما تغییر کند. همچنین برای دنبال کردن نقاط روی شکل لزومی به دنبال کردن تک تک نقاط نیست و میتوان 2 تا یکی نقاط را دنبال کرد. همچنین برای آنکه بتواند دوباره از نقاطی که یک بار رد شده باز هم رد شود ما برای iterate کردن روی لیست از index%len(points) استفاده کردیم و مقدار index نیز به مرور زیاد میشود. فقط در حالتی که شکل ما logarithmic\_spiral میباشد در صورت سوال ذکر نشده که مجدد به مسیرش ادامه دهد ( که منطقا یک شکل بسته هم نیست که این کار را انجام دهد) پس وقتی به انتهای لیست نقاط رسید متوقف میشود.

#!/usr/bin/python3

import rospy

import tf

import math

from nav\_msgs.msg import Odometry

from geometry\_msgs.msg import Twist

import matplotlib.pyplot as plt

import numpy as np

class PIDController():

    def \_\_init\_\_(self) -> None:

        rospy.init\_node("controller" , anonymous=False)                           # initialize node

        self.cmd\_publisher = rospy.Publisher('/cmd\_vel' , Twist , queue\_size=10)  # this node also is a publisher

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

        # linear velocity PID gains

        self.k\_p\_l = 0.1

        self.k\_i\_l = 0.001

        self.k\_d\_l = 0.02

        # angular velocity PID gains

        self.k\_p\_a = 0.3

        self.k\_i\_a = 0.003

        self.k\_d\_a = 0.1

        # its not necessary to check all points if we have

        # alot of sample on path.

        self.index\_increment = 2

        # the points list on a desired shape

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

        if self.shape == "rectangle":

            self.points = self.make\_rectangle()

        elif self.shape == "star":

            self.points = self.make\_star()

        elif self.shape == "logarithmic\_spiral":

            self.points = self.make\_logarithmic\_spiral()

            self.index\_increment = 1

        else:

            rospy.loginfo(f"please enter a valid shape")

        # threshold for getting next point

        self.dist\_threshold = 0.25

        self.angle\_threshold = 1.2

        # define goal variables

        self.x\_goal = 0

        self.y\_goal = 0

        # arrays contains all errors in simulation time used for plotting

        self.dist\_errs = []

        self.angle\_errs = []

        self.index = 0

        # time step of get feedback

        self.dt = 0.005

        # the desired value

        self.D = 0

        rate = 1/self.dt

        self.r = rospy.Rate(rate)

    def get\_heading(self):

        '''

        get the yaw angle of robot in world.

        We call it, heading of the robot.

        '''

        # waiting for the most recent message from topic /odom

        msg = rospy.wait\_for\_message("/odom" , Odometry)

        orientation = msg.pose.pose.orientation

        # convert quaternion to odom

        roll, pitch, yaw = tf.transformations.euler\_from\_quaternion((

            orientation.x ,orientation.y ,orientation.z ,orientation.w

        ))

        return yaw

    def get\_pose(self):

        '''

        get x and y coordinate of position of the robot

        '''

        # waiting for the most recent message from topic /odom

        msg = rospy.wait\_for\_message("/odom" , Odometry)

        position = msg.pose.pose.position

        return position.x, position.y

    def get\_distance(self, point\_target, point\_curr):

        '''

        this function calculate Euclidean distance between

        a given current and target point.

        '''

        x1, y1 = point\_curr

        x2, y2 = point\_target

        return math.sqrt((x2 - x1)\*\*2 + (y2 - y1)\*\*2)

    def distance\_from\_goal(self):

        '''

        this function get us the current Euclidean distance from goal

        '''

        x\_curr, y\_curr = self.get\_pose()

        distance = math.sqrt((self.x\_goal-x\_curr)\*\*2 + (self.y\_goal-y\_curr)\*\*2)

        return distance

    def angle\_from\_goal(self):

        '''

        function below first calculate the heading angle and then the desired angle from current pose to goal pose.

        it means that robot heading must be equal to this angle for being in a

        correct direction. then return the difference between heading and desired angle.

        '''

        # find x and y of current position and find relative x and y to goal point

        x\_curr, y\_curr = self.get\_pose()

        relative\_x = self.x\_goal - x\_curr

        relative\_y = self.y\_goal - y\_curr

        # get heading of robot in radian

        heading = self.get\_heading()

        # now we should find the desired angle

        # desired angle tell us the angle of goal point relative to current point

        desired\_angle = 0

        if (relative\_x == 0 and relative\_y ==0):

            # this state (x=0 , y =0) is undefined so we handle it seperately

            desired\_angle = heading

        else:

            desired\_angle = math.atan2(relative\_y, relative\_x)

        # the angle express howmuch we should rotate to reach the desired angle

        angle = heading - desired\_angle

        # but we design controller and if the angle is bigger than 180 or less than -180

        # the robot must rotate alot so we should find its complementary to 360 degrees

        if angle < math.radians(-180):

            angle = math.radians(360)-abs(angle)

        elif angle > math.radians(180):

            angle = angle-math.radians(360)

        return angle

    def find\_nearest\_point(self):

        '''

        When the robot is placed in a world it should find the closest point

        of given path and move forward it. this function find the closest point.

        also we consider that, the robot when starting to move the nearest point

        maybe changed so we should update the nearest point.

        '''

        curr\_point = self.get\_pose()

        goal\_point = self.points[0]

        my\_min\_dist = self.get\_distance(goal\_point, curr\_point)

        counter = 0

        for point in self.points:

            dist = self.get\_distance(point, curr\_point)

            if dist < my\_min\_dist:

                my\_min\_dist = dist

                goal\_point = point

                self.index = counter

            counter += 1

        self.x\_goal, self.y\_goal = goal\_point

    def make\_rectangle(self):

        '''

        below code make sample points on a rectangle shape and

        if we plot them show us a rectangle. we return the list of

        this points and each point also is a list like [x,y].

        length: 6

        width: 4

        '''

        X1 = np.linspace(-3, 3 , 100)

        Y1 = np.array([2]\*100)

        Y2 = np.linspace(2, -2 , 100)

        X2 = np.array([3]\*100)

        X3 = np.linspace(3, -3 , 100)

        Y3 = np.array([-2]\*100)

        Y4 = np.linspace(-2, 2 , 100)

        X4 = np.array([-3]\*100)

        X, Y = np.concatenate([X1,X2, X3 , X4]), np.concatenate([Y1,Y2,Y3,Y4])

        points = [[x, y] for x, y in zip(X, Y)]

        return points

    def make\_star(self):

        '''

        below code make sample points on a star shape and

        if we plot them show us a star. we return the list of

        this points and each point also is a list like [x,y].

        '''

        X1 = np.linspace(0, 3 , 100)

        Y1 = - (7/3) \* X1  + 12

        X2 = np.linspace(3, 10 , 100)

        Y2 = np.array([5]\*100)

        X3 = np.linspace(10, 4 , 100)

        Y3 = (5/6) \* X3  - (10/3)

        X4 = np.linspace(4, 7 , 100)

        Y4 = -(3) \* X4  + 12

        X5 = np.linspace(7, 0 , 100)

        Y5 = -(4/7) \* X5  - 5

        X6 = np.linspace(0, -7 , 100)

        Y6 = (4/7) \* X6  - 5

        X7 = np.linspace(-7, -4 , 100)

        Y7 = 3 \* X7  + 12

        X8 = np.linspace(-4, -10 , 100)

        Y8 = -(5/6) \* X8  - (10/3)

        X9 = np.linspace(-10, -3 , 100)

        Y9 = np.array([5]\*100)

        X10 = np.linspace(-3, 0 , 100)

        Y10 = (7/3) \* X10  + 12

        X, Y = np.concatenate([X1, X2, X3, X4, X5, X6, X7, X8, X9, X10]), np.concatenate([Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8, Y9, Y10])

        points = [[x, y] for x, y in zip(X, Y)]

        return points

    def make\_logarithmic\_spiral(self):

        '''

        below code make sample points on a logarithmic spiral shape and

        if we plot them show us a logarithmic spiral. we return the list of

        this points and each point also is a list like [x,y].

        '''

        a = 0.17

        k = math.tan(a)

        X , Y = [] , []

        for i in range(150):

            t = i / 20 \* math.pi

            dx = a \* math.exp(k \* t) \* math.cos(t)

            dy = a \* math.exp(k \* t) \* math.sin(t)

            X.append(dx)

            Y.append(dy)

        points = [[x, y] for x, y in zip(X, Y)]

        return points

    def control(self):

        close\_enough = False

        self.find\_nearest\_point()

        distance = self.distance\_from\_goal()

        sum\_i\_dist = 0

        prev\_error\_dist = 0

        err\_dist = distance - self.D

        angle = self.angle\_from\_goal()

        sum\_i\_angle = 0

        prev\_error\_angle = 0

        err\_angle = angle - self.D

        move\_cmd = Twist()

        while not rospy.is\_shutdown():

            # linear velocity

            distance = self.distance\_from\_goal()

            err\_dist = distance - self.D

            self.dist\_errs.append(err\_dist)

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

            P\_l = self.k\_p\_l \* err\_dist

            I\_l = self.k\_i\_l \* sum\_i\_dist

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

            move\_cmd.linear.x = P\_l + I\_l + D\_l

            prev\_error\_dist = err\_dist

            #rospy.loginfo(f"linear velocity")

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

            # angular velocity

            angle = self.angle\_from\_goal()

            err\_angle = angle - self.D

            self.angle\_errs.append(err\_angle)

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

            P\_a = self.k\_p\_a \* err\_angle

            I\_a = self.k\_i\_a \* sum\_i\_angle

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

            move\_cmd.angular.z = -(P\_a + I\_a + D\_a)

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

            prev\_error\_angle = err\_angle

            #rospy.loginfo(f"angular velocity")

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

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

            # If distance from current goal is less than threshold then define new goal

            if err\_dist < self.dist\_threshold and err\_angle < self.angle\_threshold:

                close\_enough = True

                self.index += self.index\_increment

                # Note that for logarithmic\_spiral if arrive to last point we dont continue

                # but for star and rectangle we continue until user make interrupt

                if self.shape == "logarithmic\_spiral" and self.index > len(self.points)-1:

                    break

                self.x\_goal, self.y\_goal = self.points[self.index%len(self.points)]

            elif not close\_enough:

                self.find\_nearest\_point()

            self.r.sleep()

    def on\_shutdown(self):

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

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

        # linear

        plt.plot(list(range(len(self.dist\_errs))), self.dist\_errs, label='dist\_errs')

        plt.axhline(y=0,color='R')

        plt.draw()

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

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

        plt.show()

        # angular

        plt.plot(list(range(len(self.angle\_errs))), self.angle\_errs, label='angle\_errs')

        plt.axhline(y=0,color='R')

        plt.draw()

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

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

        plt.show()

        rospy.sleep(1)

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

    try:

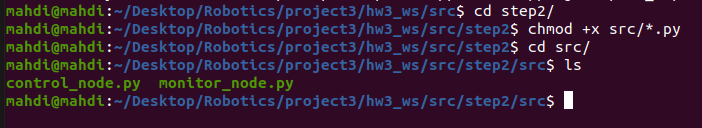
        pidc = PIDController()

        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 step2 لازم است تا یک فولدر launch ایجاد کنیم و داخل آن یک control.launch ایجاد کنیم. لانچ فایل به صورت زیر است.

ابتدا یک لانچ فایل دیگر را include میکنیم و میگوییم در آن آرگومان های ورودی مثل مکان اولیه ربات و زاویه اولیه چه باشند . اصلا در کدام نقشه ( که اینجا empty\_world بود) را بالا بیاورد و ربات و gazebo را آماده کند. 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. همچنین در نود controller ما نام شکل مسیر را به عنوان ورودی از کاربر میگیریم و در این لانچ فایل مشخص شده است. نود monitor که برای رسم مسیر لازم است و آن هم باید بالا بیاید.

<launch>

    <arg name="shape" default="rectangle"/>

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

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

        <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="shape" value="$(arg shape)" />

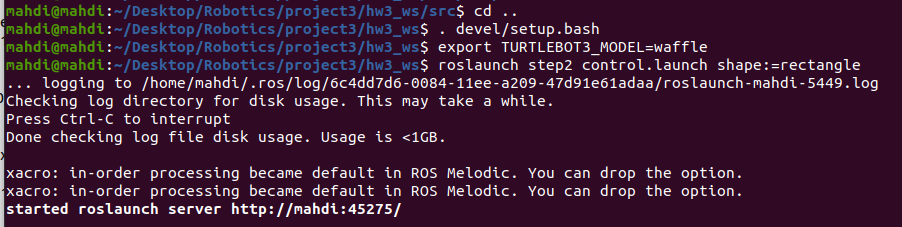
    </node>

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

</launch>

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 step2 control.launch shape:=rectangle



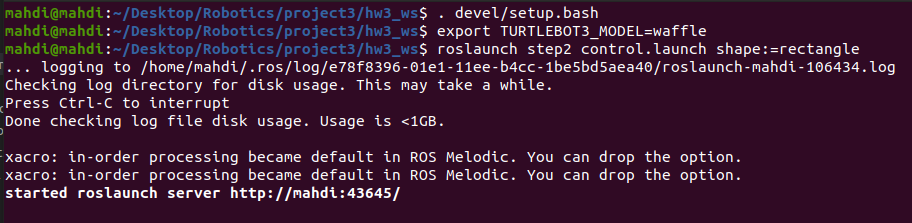
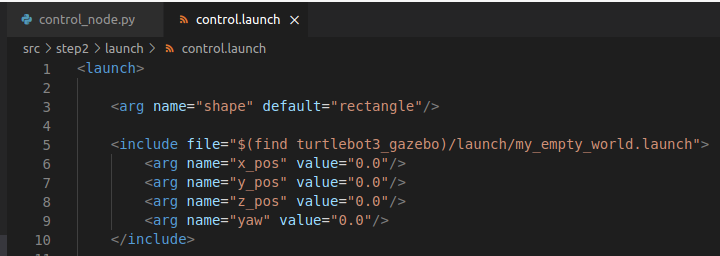
# گام دوم (اجرا و نتایج)

**موارد زیر را در گزارش خود قرار دهید:**

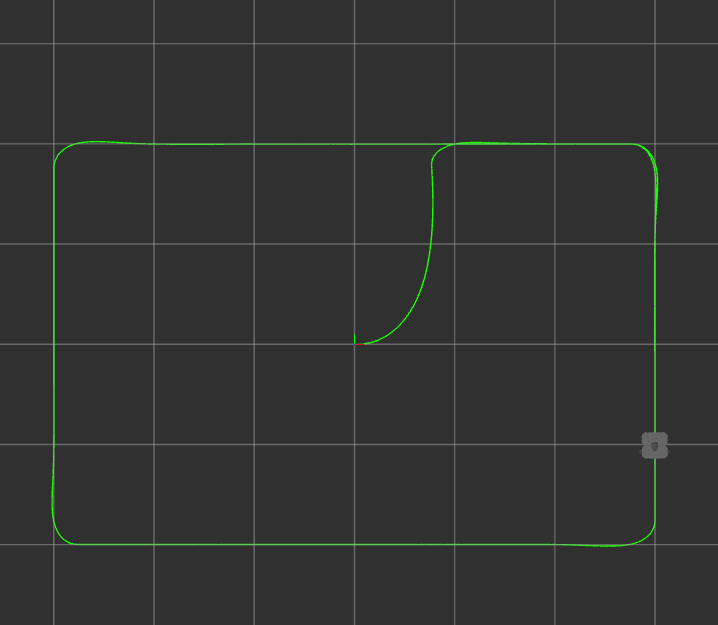
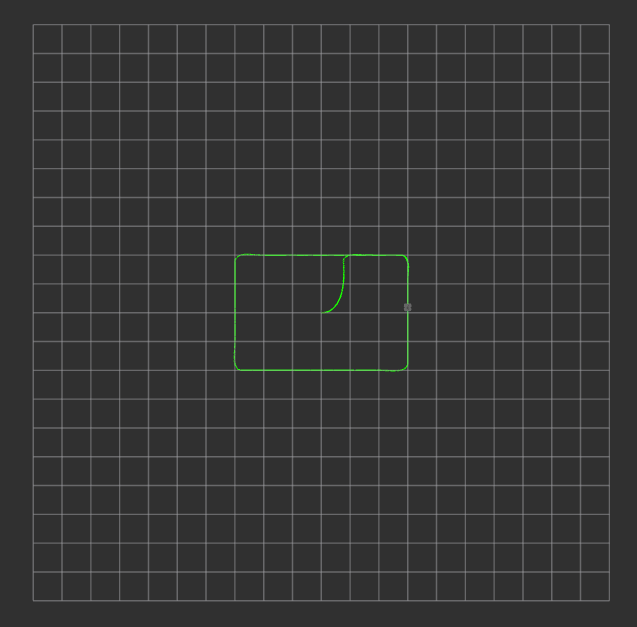
1. **نمای شکل تولید شده از حرکت ربات در شبیه سازRviz را نشان دهید.**

* **مسیر اول: مستطیل**

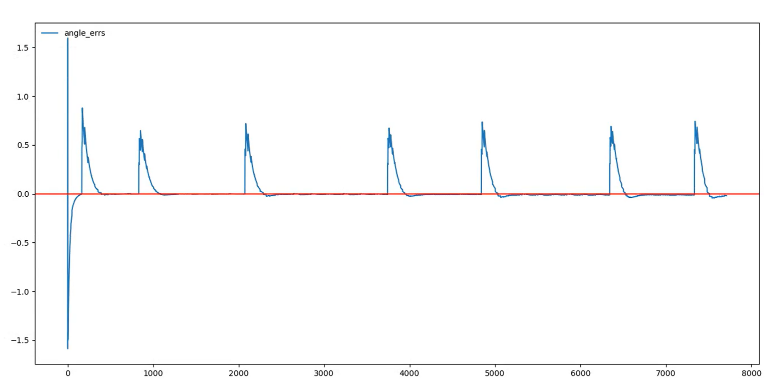
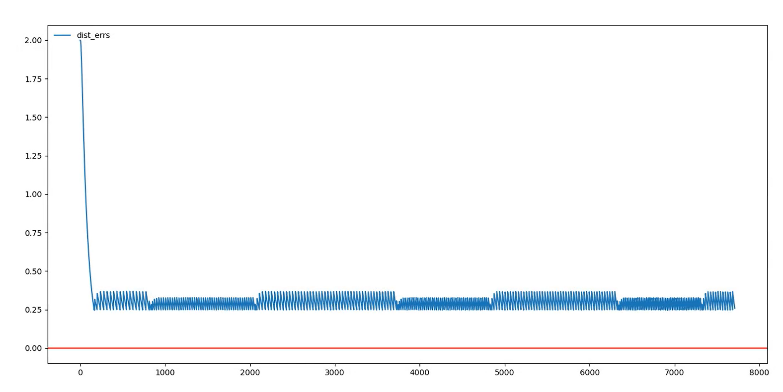
**حالت اول) ( مختصات اولیه (0,0) )** ابتدا مختصات اولیه را در لانچ فایل برابر با (0,0) قرار میدهیم و آن را اجرا میکنیم:



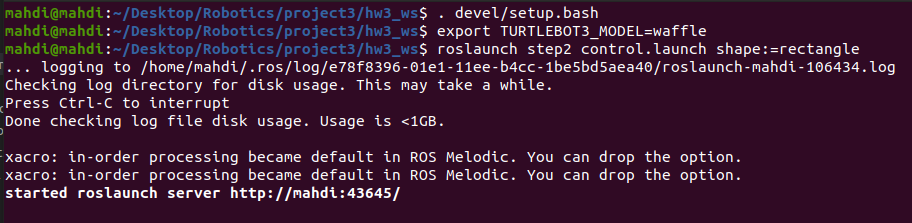
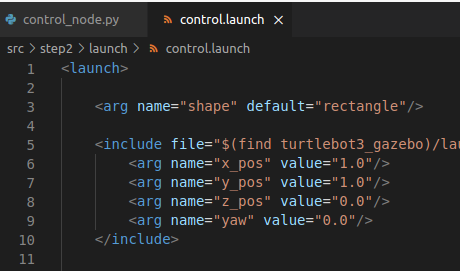
همانطور که دیده میشود ربات به نزدیک ترین نقطه حرکت کرده و سپس حرکت خود را تا زمانی که برنامه متوقف نشود، ادامه می‌دهد.



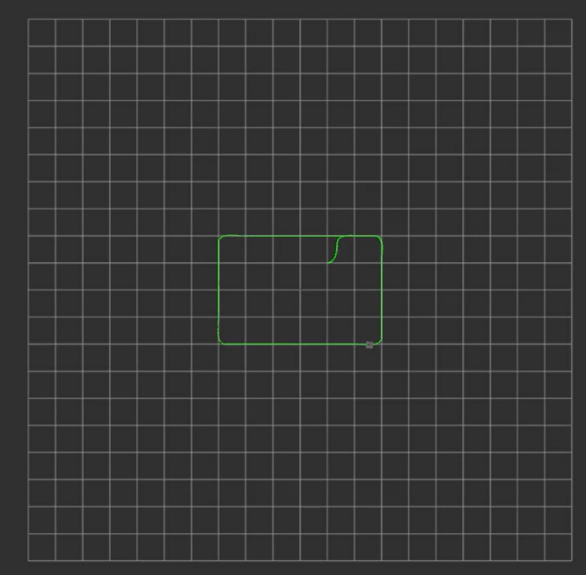
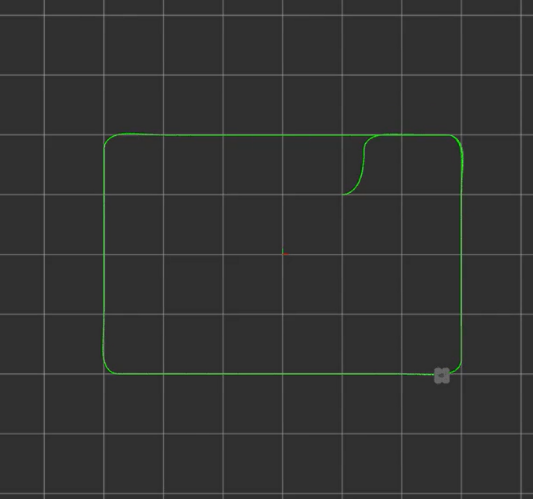
نمودارهای خطا به صورت زیر میشوند. در خطای فاصله ابتدا که به سمت نزدیکترین نقطه میرویم خطا بالاس ولی بعد که روی شکل قرار میگیریم باتوجه به threshold تعریف شده خطای فاصله در اوردر threshold میباشد. threshold برای آن است که اگر مقدار فاصله تا نقطه goal که جزء مختصات‌های شکل است کمتر از آن شد برنامه به ما مختصات بعدی را بدهد و به دنبال آن برویم. در نمودار خطای زاویه خطای زیاد برای زمانیست که ربات میواهد بچرخد و مثلا گوشه‌های مستطیل را طی کند.



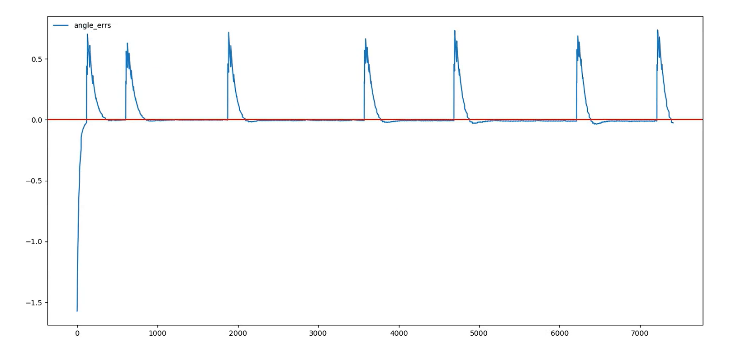
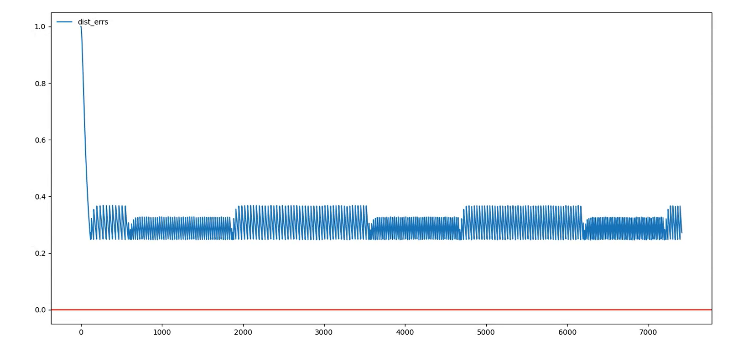
**حالت دوم) (مختصات اولیه (1,1) )** در لانچ فایل مختصات اولیه را برابر با (1,1) میگذاریم و آن را اجرا میکنیم:



همانطور که دیده میشود ربات به نزدیک ترین نقطه حرکت کرده و سپس حرکت خود را تا زمانی که برنامه متوقف نشود، ادامه می‌دهد.

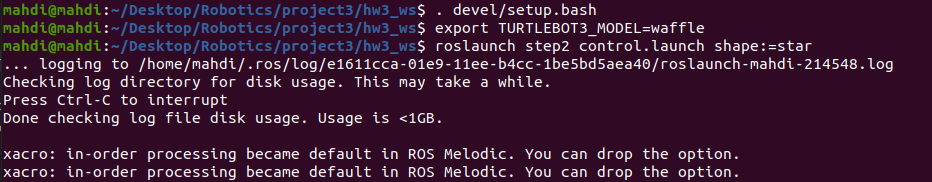
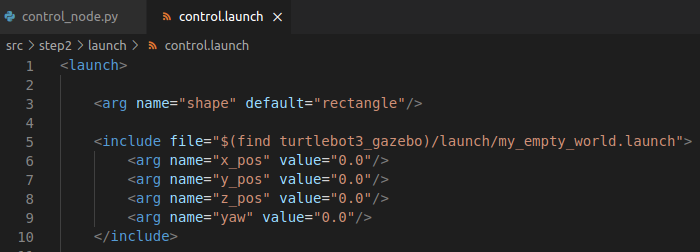


نمودارهای خطا نیز به صورت زیر خواهد بود. توضیحات نیز مانند قبل میباشد. در اینجا در خطای فاصله چون روی نقطه 1و1 بودیم خطای اولیه کمتر است.



* **مسیر دوم: ستاره**

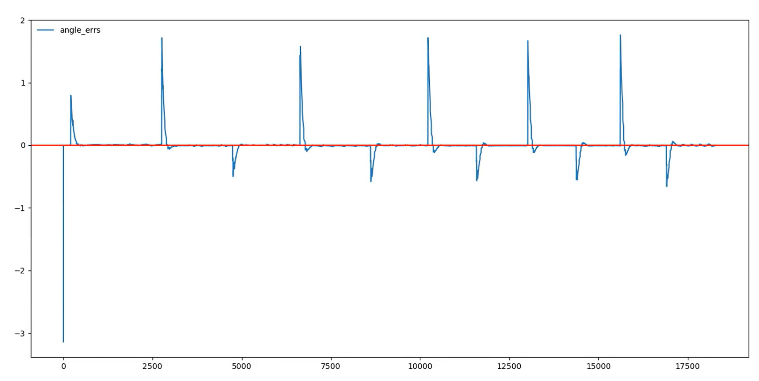
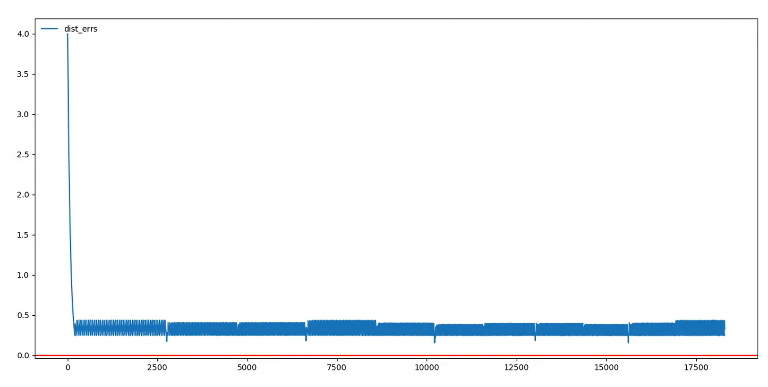
در این جا یک حالت بررسی میشود و مختصات اولیه برابر با (0,0) میباشد. همچنین ربات تا زمانیکه برنامه متوقف نشده به حرکت خود ادامه میدهد.



A star on a graph

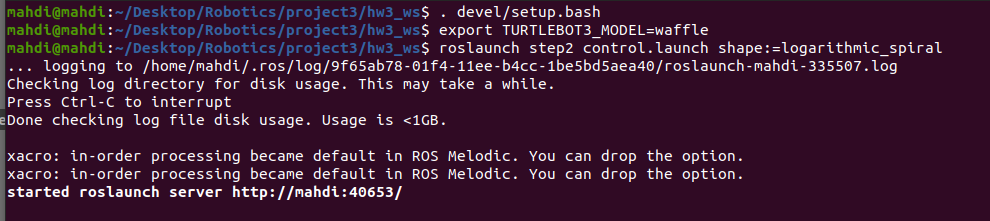
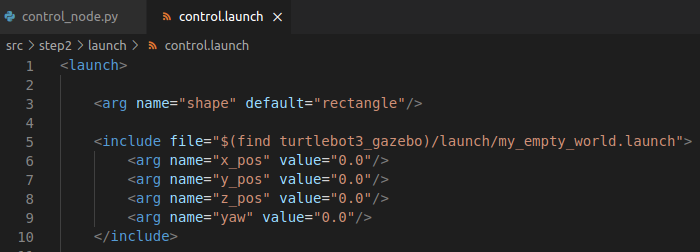
Description automatically generated with low confidenceهمانطور که دیده میشود ربات به نزدیک ترین نقطه حرکت کرده و سپس حرکت خود را تا زمانی که برنامه متوقف نشود، ادامه می‌دهد.

نمودارهای خطا نیز به صورت زیر خواهد بود. توضیحات نیز مانند قبل (در مسیر مستطیل توضیح داده شد) میباشد. نمودار سمت چپ نمودار خطای فاصله تا هر goal میباشد و نمودار سمت راست نمودار خطای زاویه‌ای ( فاصله بین heading و زاویه desired که در قسمت کد توضیحات داده شده) میباشد.

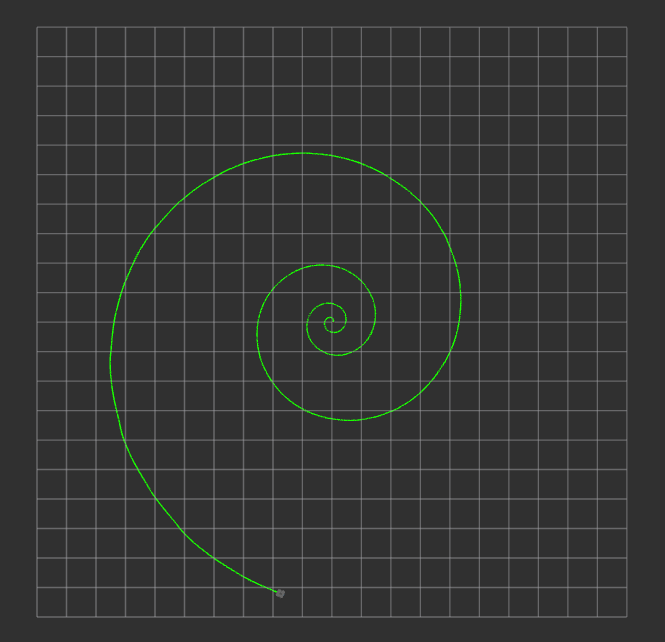
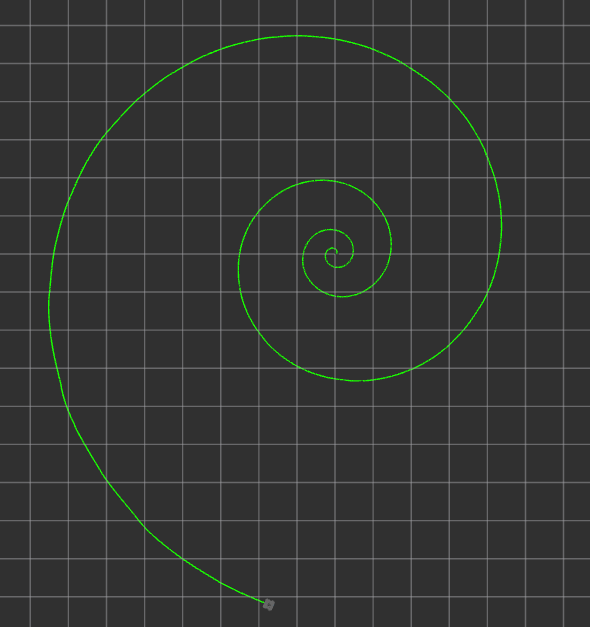


* **مسیر سوم: مارپیچ لگاریتمی**

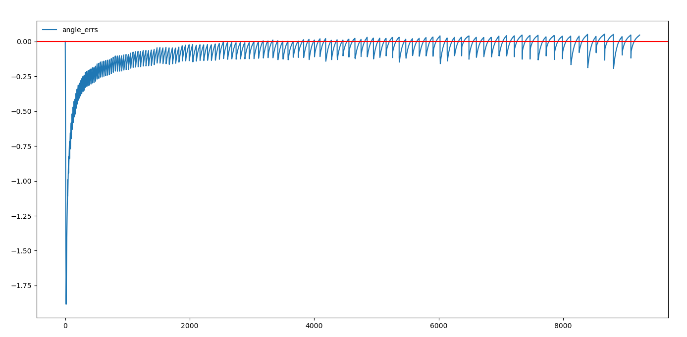
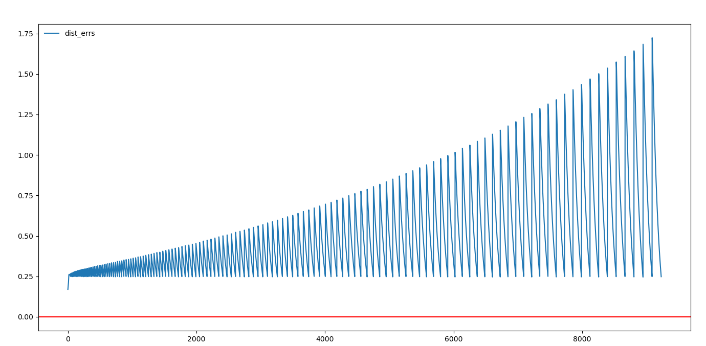
در این جا مختصات اولیه برابر با (0,0) و از در تابع تولید نقاط روی مارپیچ a=0.17 ( در دستور کار نوشته 17 که وقتی رسم کردم شکلش عجیب و خیلی بزرگ شد و به نظر غلطه) میباشد. همچنین در اینجا ذکر نشده که ربات دائما به حرکتش ادامه دهد و درنتیجه وقتی به اخرین نقطه رسید، می ایستد.



در نهایت نتیجه به صورت زیر میشود:

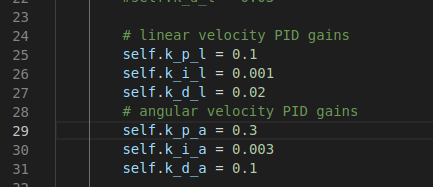


نمودارهای خطا نیز به صورت زیر خواهد بود. در نمودار سمت چپ که برای خطای فاصله هست، به علت اینکه به مرور فاصله نقاط روی شکل زیاد میشود خطای فاصله تا مقصد بیشتر شده ولی با نزدیک شدن به آن نقطه خطا کم میشود تا به threshold برسد و بعد نقطه بعدی روی شکل به عنوان goal جدید اعلام میشود. در نمودار سمت راست نیز خطای زاویه است که به خاطر شکل منحنی که داریم این مقدار دائما درحال کم و زیاد شدن است و حالت نوسانی به خود گرفته است.



1. **تشریح کنید که چه ضرایبی برای P، I و D پاسخ مناسبی را ارائه میکند. (ضرایب مناسب را به صورت تجربی به دست آورید) و درباره تاثیرات افزایش و کاهش هر کدام از ضرایب بحث کنید و چند نمونه آن را تشریح کنید.**

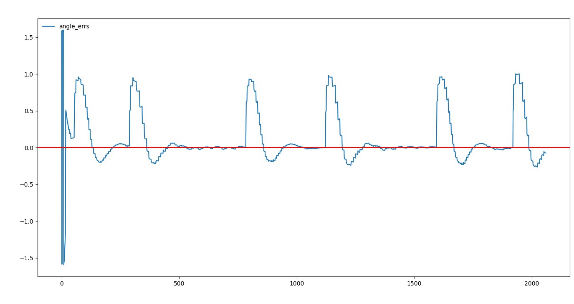
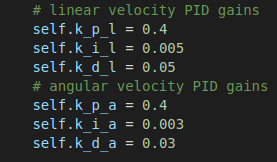
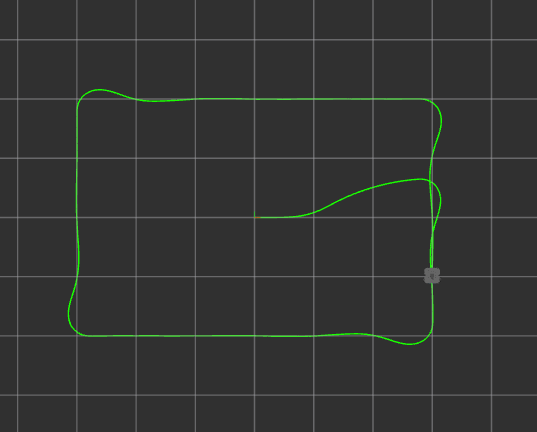
در این گام ضرایب k\_p و k\_i و k\_d مربوط به سرعت خطی و زاویه‌ای به گونه‌ای تنظیم شدند که بتوانیم به کمک آن‌ها تمامی شکل‌های خواسته شده را تولید کنیم. ضرایب به صورت زیر میباشند:



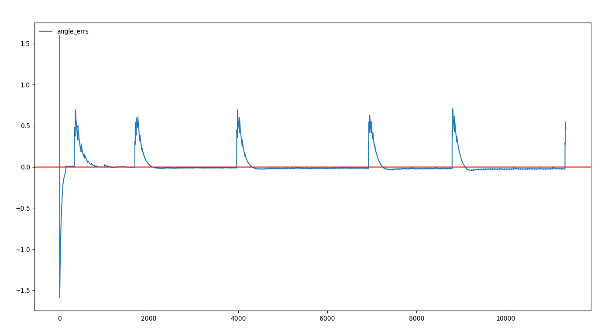
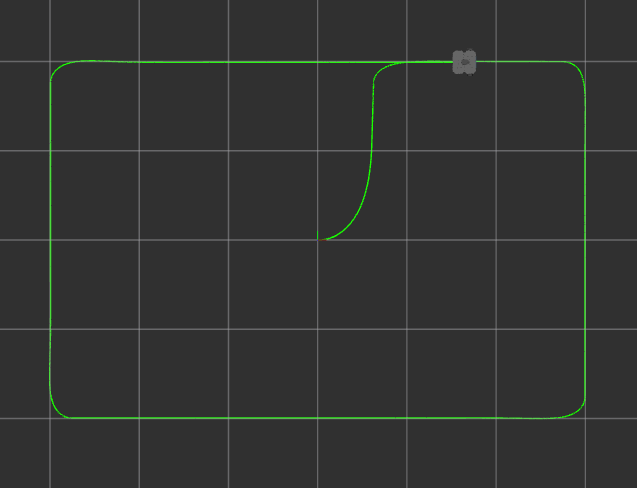
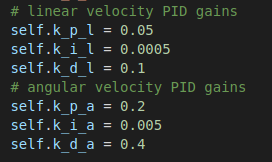
دقت شود که اگر gainها را کمی بیشتر میگرفتیم ( به خصوص gainهای proportional) باز هم جواب قابل قبولی میداد و سرعت هم بیشتر میشد ولی برای آنکه شکل موردنظر تا حد ممکن دقیقتر شود و در گروه درس هم مطرح شد که دقت مهم هست، gainها را کم گرفتیم. همچنین میخواستم که یک کنترلر بنویسم که برای هر سه شکل کار کند.

* **مراحل رسیدن به gainهای فوق**

ابتدا مقادیر را همینطوری set کردم و خروجی زیر را دریافت کردم. به نظر میرسد که اولا باید k\_p مربوط به سرعت زاویه‌ای بیششتر از سرعت خطی باشد تا به ارور زاویه‌ای سریعتر پاسخ دهد و بتواند به نقطه نزدیکتر که ضلع بالا و پایین است برود و در پیچ‌ها هم بهتر بپیچد و خیلی منحنی نشود. همچنین نیاز است مقدار k\_d ها نیز بیشتر شوند و برای زاویه کمی بیشتر، بیشتر شود 😊 در نمودار خطای زاویه هم میتوان نوسانات و overshoot را مشاهده کرد که به کمک همین gain دیفرانسیلی میتوان بهبود بخشید. در ادامه با جزئیات بیشتر بررسی میشود.

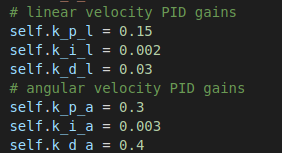
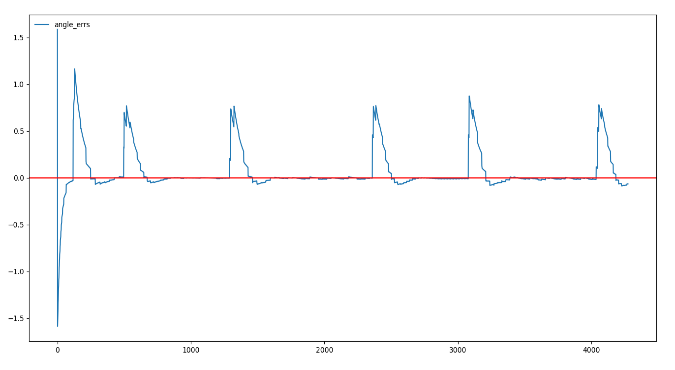
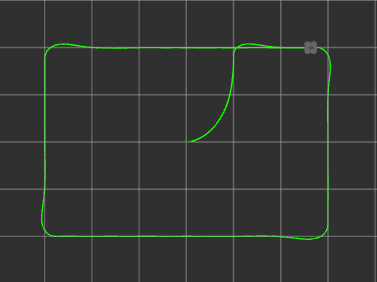


باتوجه به تجربه قسمت های قبل اگر gain های proportional کمتر شود باعث میشود ربات به خطاها با سرعت کمتری پاسخ دهد و راحت تر و دقیقتر بتوان آن را کنترل کرد. از طرفی هم طبق توضیح بالا باید k\_p مربوط به سرعت زاویه ای از خطی بیشتر است. پس با این اوصاف همان ضرایب به دست آمده از گام 1 بخش دوم را که فاصله نقاط بیشتر بود و جواب گرفتیم، معیار خود قرار دادم و از آن‌ها استفاده کردم و تا حد خوبی جواب میگرفتم چراکه ضرایب k\_p مقادیر کمی داشتند و به آرامی ربات حرکت میکرد و Overshoot چندانی نداشتم. این ضرایب و نتایج آن در زیر امده اند. خروجی خیلی دقیق است ولی چون gainهای P خیلی کم است اجرای آن خیلی طول کشید و ربات سرعت کمی داشت.



همانطور که اشاره شد، در اینجا نقاط goal نزدیک به هم دیگر هستند و از رفتن از یکی به دیگری خیلی ارور فاصله و زاویه زیاد نمیشد. بیترین ارور در هنگام شروع برای رفتن از مثلا (0,0) به نزدیکترین نقطه به خصوص در شکل مستطیل بود. که با توجه به نحوه پیاده سازی و اینکه سرعت‌های خطی و زاویه‌ای زیاد نبود به خوبی کنترل شد. در بعضی مواقع در سر پیچ‌ها کمی overshoot زیاد بود و ربات خیلی فاصله میگرفت که به کمک gain مشتقی یعنی k\_d کنترل شد و مقدارش را بیشتر کردم. همانطور که قبلا هم گفتم gain انتگرالی یعنی k\_i ممکن است در این بخش خیلی به چشم نیایند چون steady state erorای نداریم و مخصوصا یک thresholdای تعریف کردیم که ربات به آن برسد کافی است و به هر جهت مقدار آن‌ها طبق توضیحات اسلاید برابر با 0.01 مقدار k\_p در نظر گرفته شد. با این حال این ترم به stability و پایداری ربات کمک میکند ولی خب مقدارش را کم گذاشتیم که برعکس موجب Overshoot نشود. حتی میشد مقدارش را کمتر هم بگذاریم.

پس از کمی سعی و خطا برای شکل مستطیل مقادیر gainها را به صورت زیر برای سرعت خطی و زاویه‌ای گذاشتیم و نتیجه نسبتا خوبی میداد ولی خب در گوشه ها به خصوص در شکل ستاره کمی به مشکل میخوردیم و چون ترم p زیاد میشد سرعت بیشتر بود و خطای کار بالا میرفت.



با سعی و خطای بیشتر مقدار k\_p\_l را به 0.1 رساندیم و مقدار k\_iها را برابر با 0.01 مقدار k\_pها گرفتیم و دقت شود در این مسئله خیلی ترم انتگرال گیر کمکی نمیکند ولی زیاد شدنش منجر به overshoot و خطا میشود. ترم k\_d ها را هم کمی کمتر کردیم چراکه تا جای ممکن باید کم باشند که نویز پذیری و نوسانات سیستم کاهش یابد.

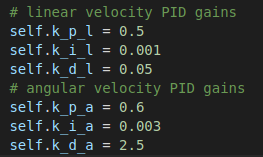
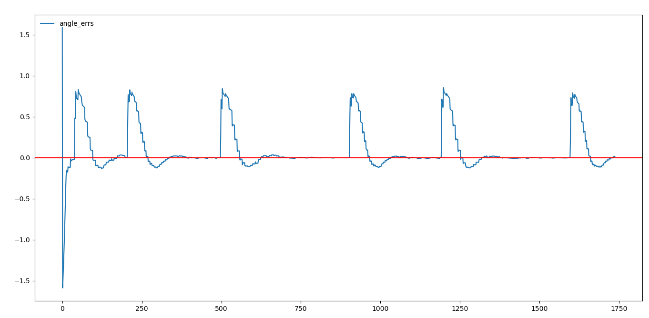
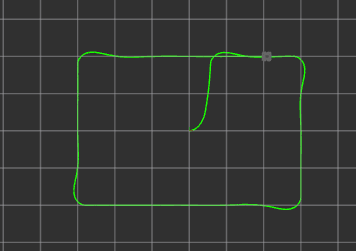
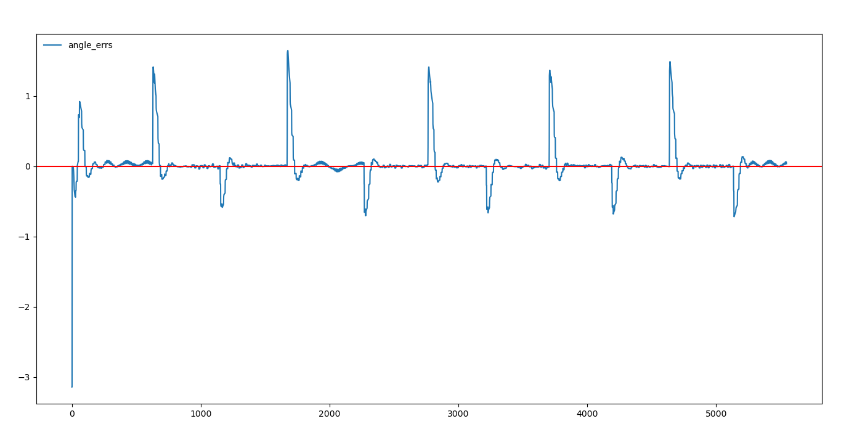
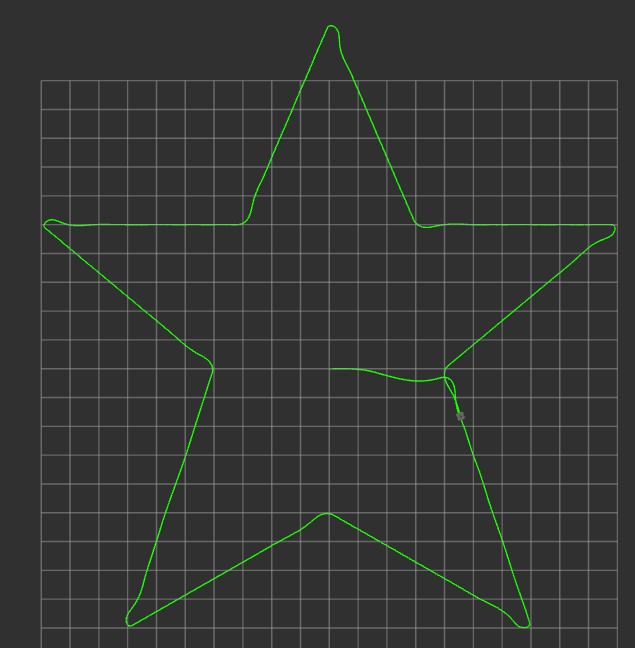
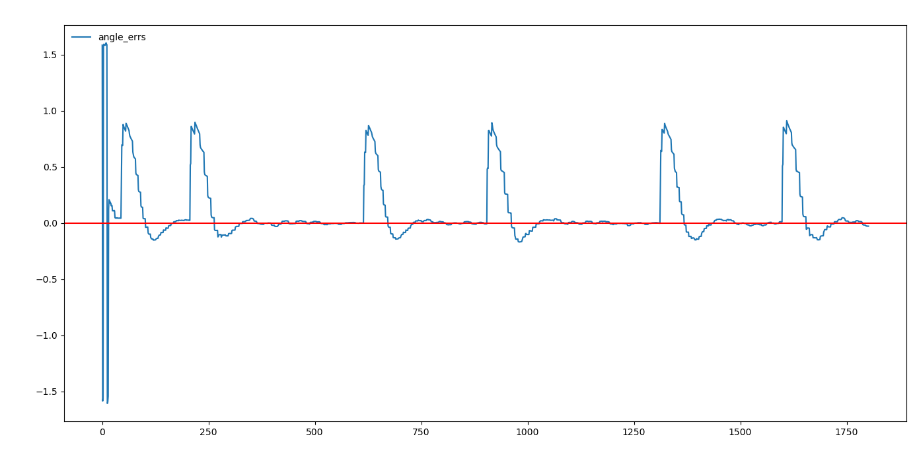
* **در ادامه توضیح کم و زیاد کردن هر ترم آمده است که منجر به رسیدن به پاسخ نهایی شد:**

حال برای مثال اگر gain مربوط به k\_p را زیاد کنیم خب باعث میشود که ربات سعی کند سریعتر ارور را کم کند و سریعتر ری اکشن دهد که طبیعتا موجب Overshoot میشود. برای مثال با gainهای زیر خروجی حاصله در شکل مستطیل و ستاره به صورت زیر میشود. در هنگام شروع چون سرعت بالاس به خوبی نمیتواند نزدیکترین نقطه را برای شروع که در ضلع‌های بالا و پایین است پیدا کند و چون سریع میرود به جایی میرسد که ضلع چپ و راست نزدیکترین میشوند. در گوشه‌ها هم به خوبی میتوان Overshoot را دید. همچنین نمودار خطای زاویه مربوط به مستطیل به خوبی بیانگر این Overshootها و نوسانات میباشد.

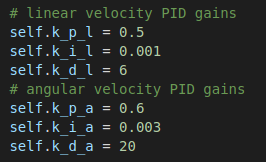
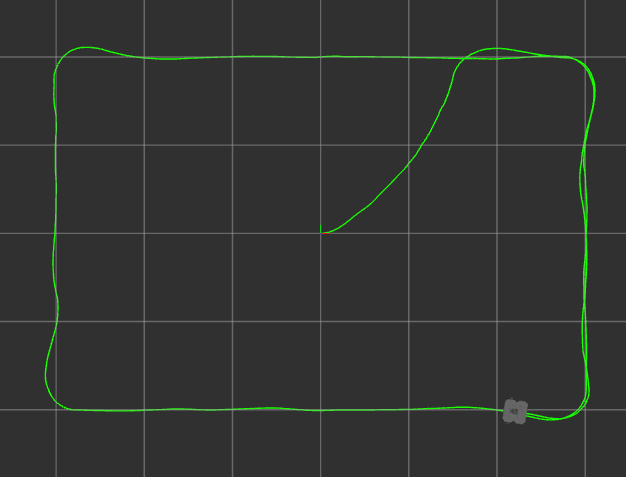
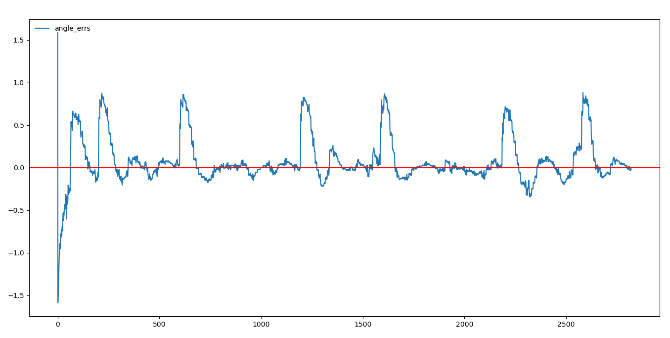
A screenshot of a computer screen

Description automatically generated with low confidence

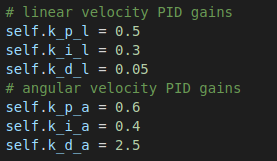
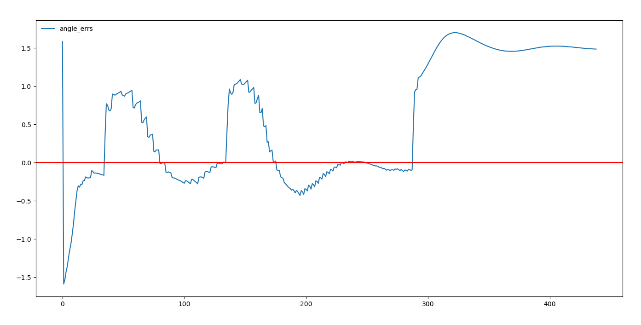
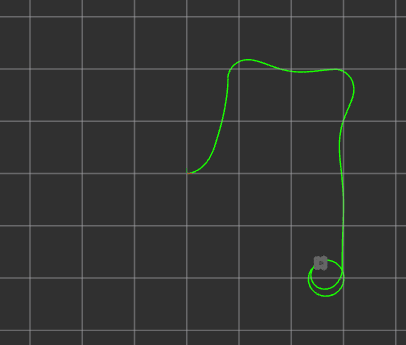
اگر مقدار K\_pها کمتر شود باعث میشود که سرعت کمتر شود و overshoot کمتر شود. همچنین دقت شود سعی میشود که k\_p مربوط به سرعت زاویه‌ای بیشتر از خطی باشد تا سرعت خطی زیاد موجب نشود که حالت curve بزرگی داشته باشیم و درواقع در مواقع پیچیدن نقطه‌ای که باید بپیچیم را رد کنیم. اگر مقادیر k\_d را زیاد کنیم بخشی از overshootها را میتواند جبران کند برای مثال در مستطیل به صورت زیر میشود. اگر نمودار خطا با قبلی مقایسه شود کتوجه میشویم هم نوسانات کمتر شده است و هم دامنه اورشوت‌ها کمی کم شده است و در شروع کار نیز به خوبی به سمت نقطه نزدیکتر میچرخد. به هرحال چون سرعت بالاس ضریب d هم تا حدی میتواند جلوی overshoot را بگیرد و بنابراین باید k\_p را کمتر کرد.



اگر مقدار k\_d بیش از حد زیاد شود باعث میشود که ربات به نویز و تغییرات خیلی حساس شود و دائما نوسان کند و مسیر از smooth بودن خارج شود.

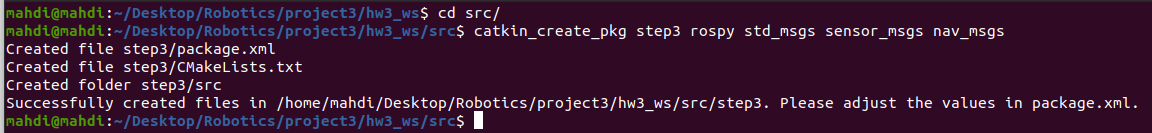


اگر مقدار k\_i زیاد باشد باعث میشود که برعکس مقدار overshoot بیشتر شود و سیستم ناپایدار شود و ارور جمع شونده ای که داریم باعث به نقطه هدف نرسیم و دائما هم خطا بیشتر شود. مثلا در شکل مستطیل در گوشه انگار که ربات نقطه هدف را به نوعی رد میکند و سعی میکند که برگردد به آن برسد و دائما تلاش میکند و خطای جمع شونده هم بیشتر میشود. لذا مقدار k\_i را خیلی کم میکنیم و با مقادیری که ما انتخاب کردیم از این دست مشکلات نمیگیریم.

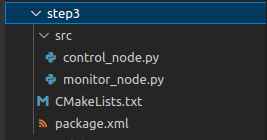


# گام سوم(توضیحات کد و آماده سازی)

این تمرین بسیار شبیه به ویدیو و کد موجود در هندزان میباشد و از آن کمک گرفتم. در این قسمت نیز ابتدا یک پکیج با نام step3 ایجاد میکنیم و dependencyهایی که ممکن است به کار بیایند را هم اضافه کنیم.

* catkin\_create\_pkg step3 rospy std\_msgs sensor\_msgs nav\_msgs

حال به سراغ ساخت نودها میرویم. برای این منظور در فولدر src مربوط به step3 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 قبل میباشد و توضیح آن در قسمت‌های قبل داده شد. در این جا فقط به کد نود control\_node میپردازیم. توضیحات تا حد زیادی مانند قبل است. 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. بقیه روال کار مانند قسمت‌های قبل میباشد.

#!/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)

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

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

        # 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 = 0.8

        self.k\_i = 0.0005

        self.k\_d = 25

        self.dt = 0.005

        self.v = 0.1

        self.D = 1.5

        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)

            rospy.loginfo(f"d : {d}")

            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

            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 step3 کد زیر را اجرا کنیم:

* A picture containing text, screenshot, font

  Description automatically generatedchmod +x src/\*.py

سپس به سراغ نوشتن لانچ فایل میرویم. در این جا به روشی کمی متفاوت تر با قبل کار میکنیم. 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 آن مستقیما همینجا اضافه میکنیم. اسم این لانچ فایل را turtlebot3\_square\_wall.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 step3/worlds/square.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) و با زاویه 90 درجه یا همان 1.57 رادیان 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="left"/>

    <include file="$(find step3)/launch/turtlebot3\_square\_wall.launch">

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

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

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

        <arg name="yaw" value="1.5708"/>

    </include>

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

    <node pkg="step3" type="control\_node.py" name="controller" output="screen">

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

    </node>

    <node pkg="step3" type="monitor\_node.py" name="monitor"></node>

</launch>

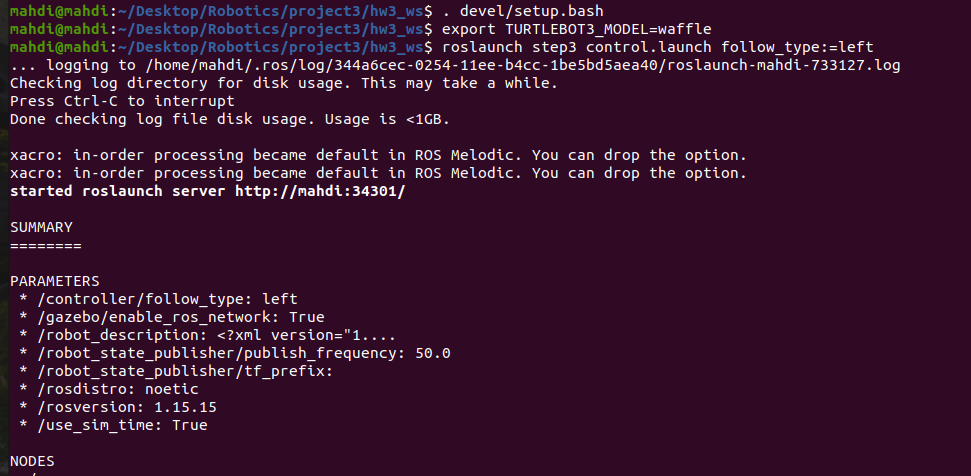
همچنین لازم است یک فولدر به نام worlds در src مربوط به پکیج step3 ایجاد کنیم و فایل square.world را در آنجا اضافه کنیم:

A screenshot of a computer

Description automatically generated

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:=left



# گام سوم(اجرا و نتایج)

**موارد زیر را در گزارش خود قرار دهید:**

1. **نمای شکل تولید شده از حرکت ربات در شبیه ساز Rviz را نشان دهید.**

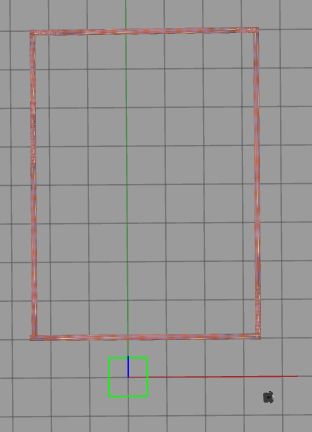
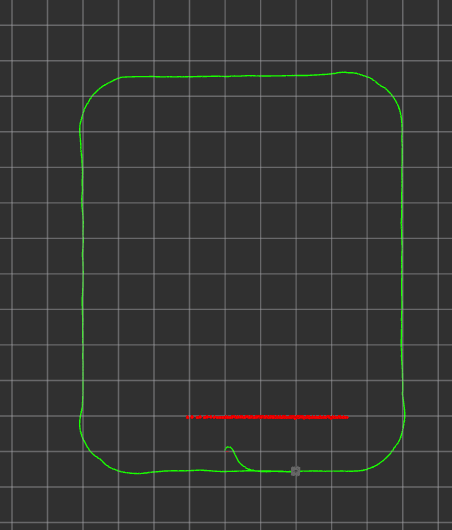
پارامترها : 1- فاصله از دیوار: 1.5m 2- سرعت خطی ثابت: 0.1m/s 3- کنترلر روی سرعت زاویه‌ای

در لانچ فایل براساس follow\_type میتوان ربات را دنبالگر چپ یا راست تعریف کرد. این مقدار را از طریق ورودی در cmd تعریف میکنیم که میتواند مقدار left یا right بگیرد.

A screen shot of a computer

Description automatically generated with low confidenceبرای دنبالگر چپ دستور زیر را به صورت زیر اجرا کردم:

نتیجه به صورت زیر شد:



همچنین نمودار خطا به صورت زیر است. سعی شد تا حد ممکن جلوی اورشوت را با gain دیفرانسیلی (k\_d) بگیرم ولی به هرحال کامل ازبین نرفت و در گوشه‌ها و چرخش‌ها کمی اورشوت داریم.

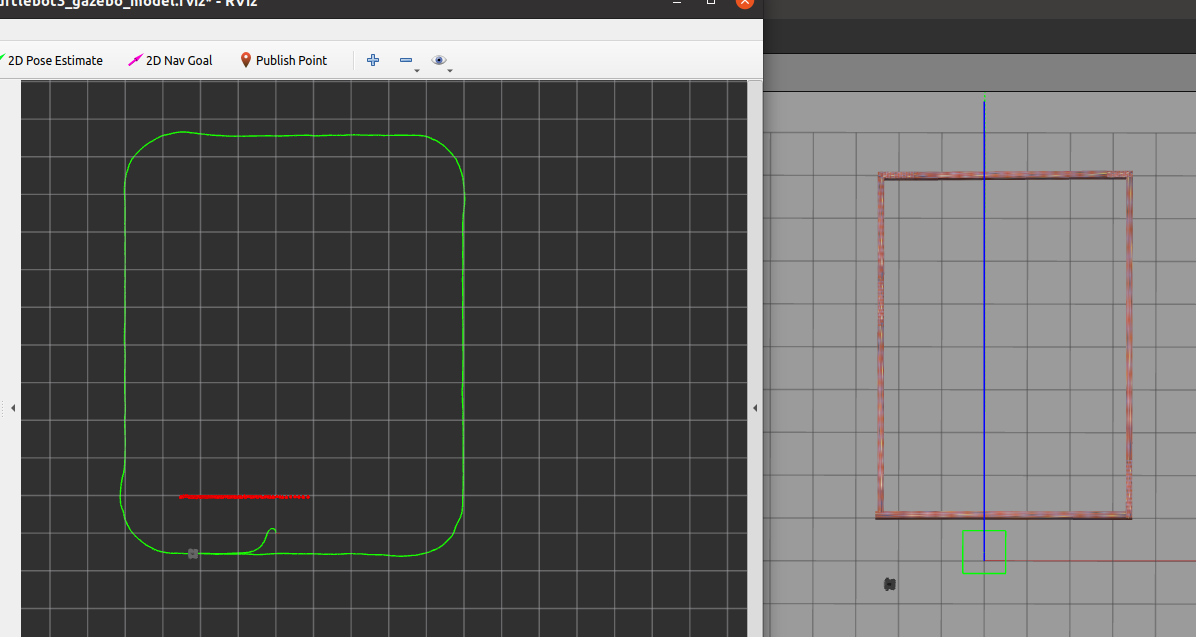
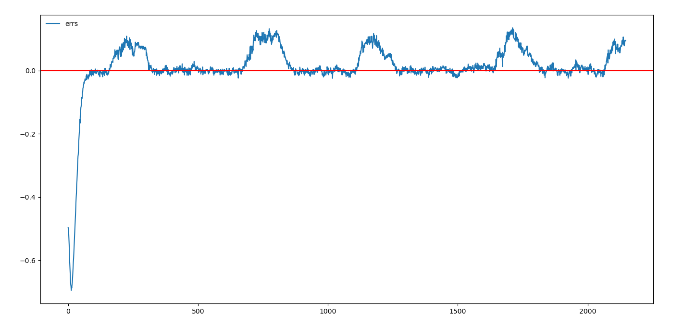
A picture containing text, line, plot, screenshot

Description automatically generated

A screen shot of a computer

Description automatically generated with low confidenceبرای دنبالگر راست به صورت زیر دستور را اجرا میکنیم. دقت شود که برای این حالت از همان gainهای قبلی استفاده کردیم .

نتیجه در Rviz و نمودار خطا را در زیر میتوانید مشاهده کنید.



1. **تشریح کنید چه ضرایبی برای P، I و D پاسخ مناسبی را ارائه میکند.**

ضرایب مناسب برای gainها به صورت زیر میباشد که با آن‌ها توانستیم نتایج بالا را بگیریم که تقریبا نتایج مطلوبی هستند.

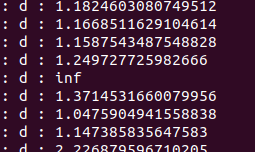
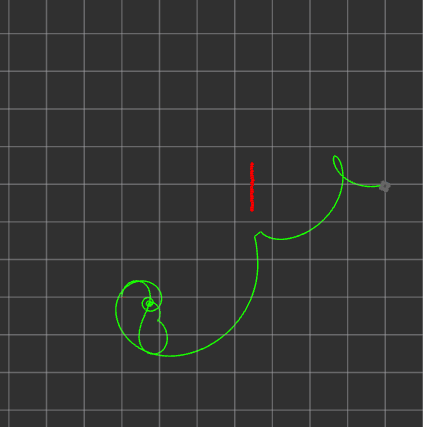
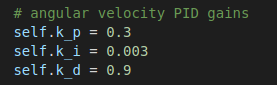
A picture containing text, font, screenshot, clock

Description automatically generated

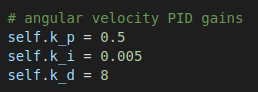
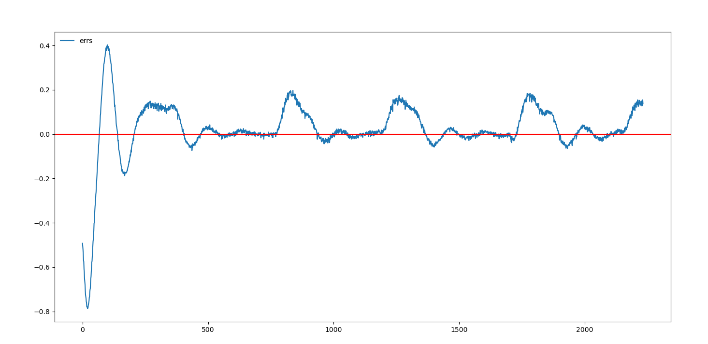
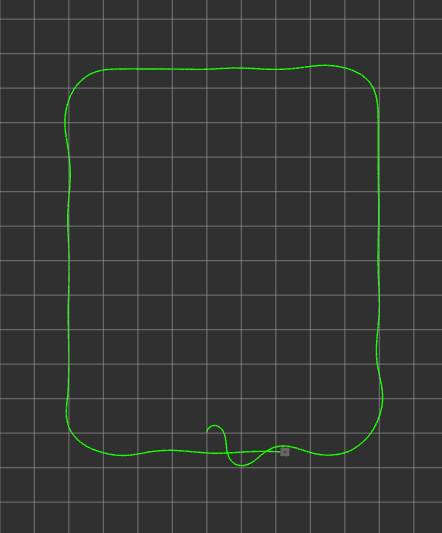
**مراحل رسیدن به gain مناسب:**

پارامترها : 1- فاصله از دیوار: 1.5m 2- سرعت خطی ثابت: 0.1m/s 3- کنترلر روی سرعت زاویه‌ای

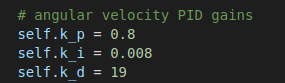
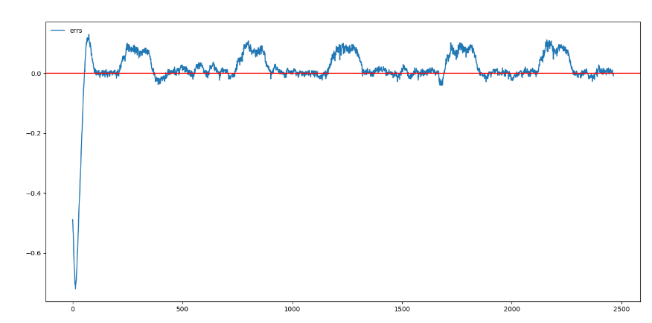
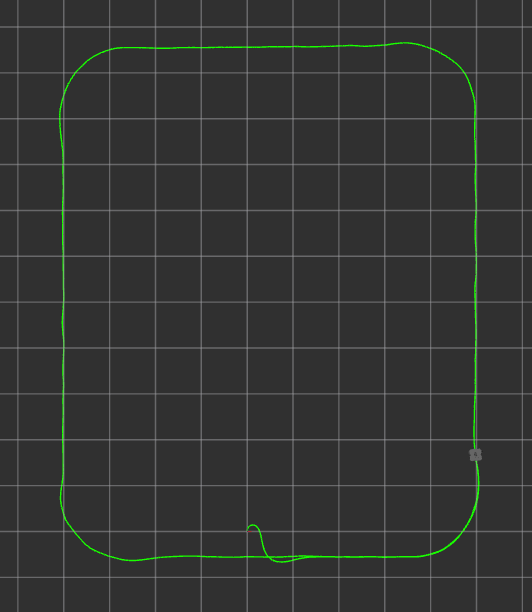
برای رسیدن به این gainها ابتدا با توجه به کارهای قبلی برای سرعت زاویه‌ای مقادیر زیر را برای gain انتخاب کردیم و نتیجه را هم در عکس سمت راست مشاهده میکنید.



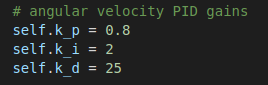
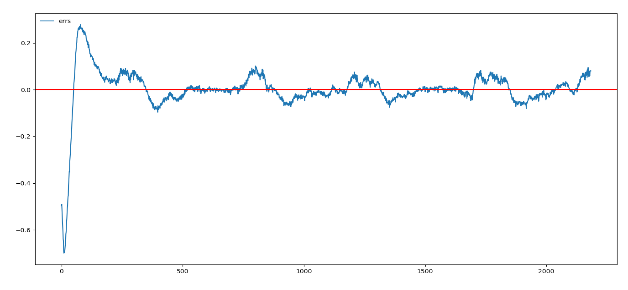
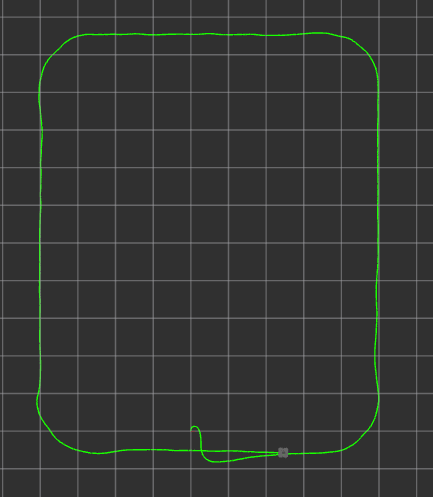
چون k\_p به اندازه کافی نبود و از طرفی overshoot داشتیم که باز چون k\_d کم بود باعث شد از دیوار فاصله بگیرد و حتی مقادیر inf در فاصله ظاهر شود ( چون دیگر سمت چپ روبه مانع نبوده که فاصله برگرداند) طبیعتا سرعت زاویه‌ای به بینهایت میل کند و ربات ناپایدار شود. پس مقدار k\_p را کمی بیشتر میکنیم که در مواقعی که باید بپیچد باتوجه به آنکه سرعت خطی 0.1 و ثابت است بتواند به خوبی بپیچد و همچنین این بالا رفتن سرعت زاویه‌ای نیازمند آن است که k\_d بیشتر شود تا جلوی overshoot را بگیرد. پس مقادیر زیر را گذاشتیم و نتیجه زیر را گرفتیم:



همانطور که دیده میشود اولا در رسیدن به فاصله مناسب از دیوار در حالت اولیه مقدار ارور زیاد بوده و به نظر میرسد که کنترلر با سرعت مناسب به ارور پاسخ نمیدهد و باید کمی مقدار k\_p را بیشتر کرد. از اون طرف در پیچیدن‌ها اورشوت زیاد است و کلا نوسانات زیاد است پس باید k\_d را هم بیشتر کنیم. ولی خب درکل اکنون ربات به یک پایداری نسبی رسیده است. حال مقادیر زیر با ادامه سعی و خطاها امتحان میکنیم که نتیجه نزدیک به مطلوب ماست.



فقط همانطور که دیده میشود در ابتدای کار وقتی از دیوار برمیگردد به فاصله 1.5 یک overshoot داریم. برای این منظور مقدار k\_d را بالاتر برده و برابر با 25 میگذاریم که نتیجه مطلوب نهایی به دست بیاید. فقط نکته دیگر ترم انتگرال گیر میباشد. این ترم خیلی در اینجا تاثیر گذار نیست و ما خطای steady state نداریم. ولی اگر مقدار این gain را زیاد بگیریم، خواهیم داشت:



همانطور که دیده میشود در این مسئله اگر مقدار k\_i زیاد باشد بدتر باعث overshoot و همچنین خارج شدن از وضعیت هموار میشود. بنابراین تا جای ممکن مقدار آن را کوچک میگیرم. در نهایت به همان مقادیر گفته شده میرسیم:

A picture containing text, font, screenshot, clock

Description automatically generated