|  |  |  |
| --- | --- | --- |
| 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 دوم** | | |

|  |  |
| --- | --- |
| مهدی رحمانی | نام و نام خانوادگی |
| 9731701 | شماره‌ دانشجویی |
| 05/02/1402 | تاریخ ارسال گزارش |

­

**فهرست گزارش سوالات**

[گام اول – به کار گیری سرویس و تعیین مقصد در empty map 3](#_Toc133080192)

[گام اول – نتایج 16](#_Toc133080193)

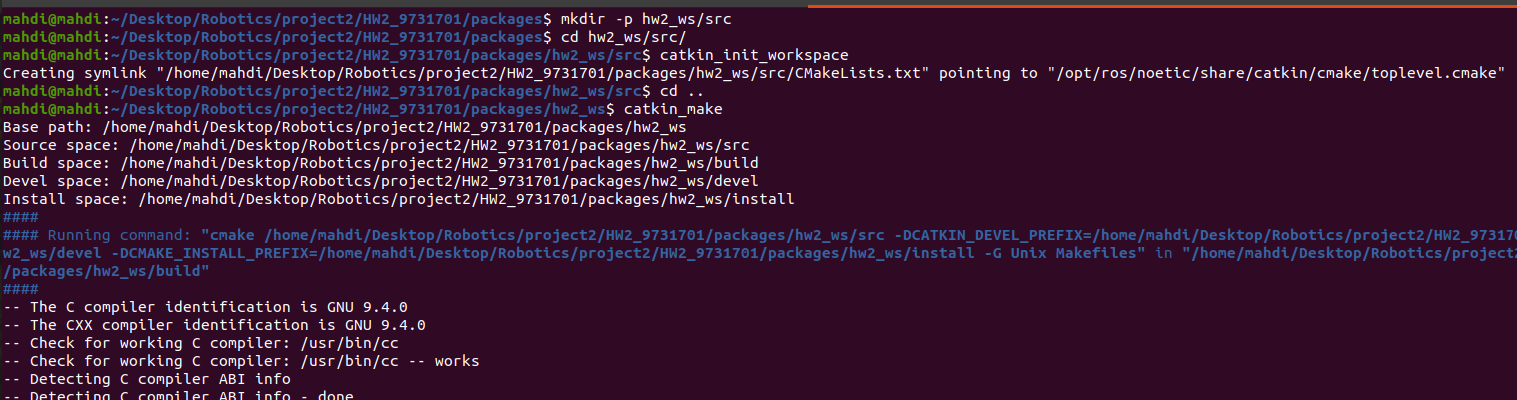
[گام دوم – تشخیص مانع به کمک لایدار 20](#_Toc133080194)

[گام دوم – نتایج 29](#_Toc133080195)

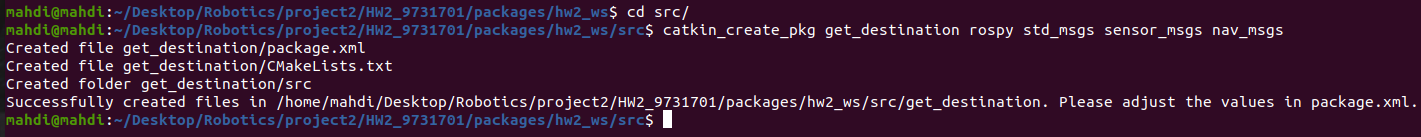
[گام سوم – کار با Pointcloud 30](#_Toc133080196)

[گام سوم – نتایج 37](#_Toc133080197)

# گام اول – به کار گیری سرویس و تعیین مقصد در empty map

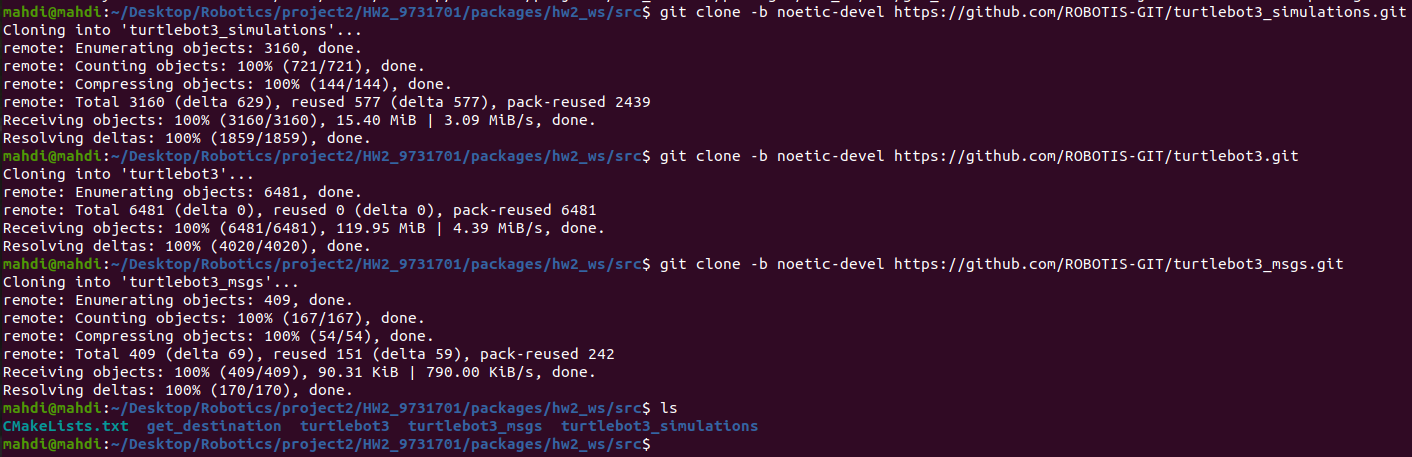
ابتدا لازم است تا یک work space برای گام اول بسازیم و آن را initialize کنیم:

حال لازم است که یک پکیج مثلا با نام get\_destination 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 get\_destination rospy std\_msgs sensor\_msgs nav\_msgs

چون قرار است از turtlebot و شبیه ساز gazebo 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:

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

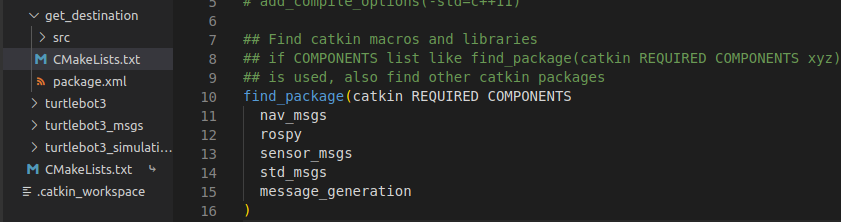
Text

Description automatically generated

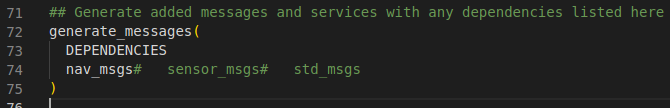
1. در خط 54 اینتر زده و بک build\_depend جدید ایجاد کنیم و در خط 62 هم یک exec\_depend ایجاد میکنیم:

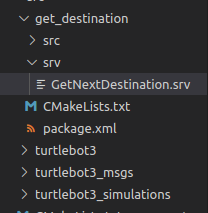
Text

Description automatically generated

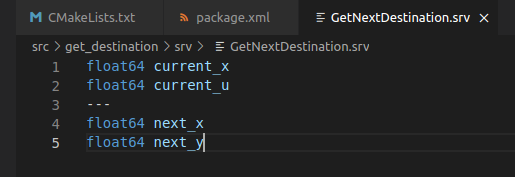
1. سپس در CMakeLists هم در خط 15 message\_generation را میگذاریم:
2. خطوط 59 تا 63 که مربوط به سرویس هست را uncomment میکنیم و تغییرات را اعمال میکنیم:

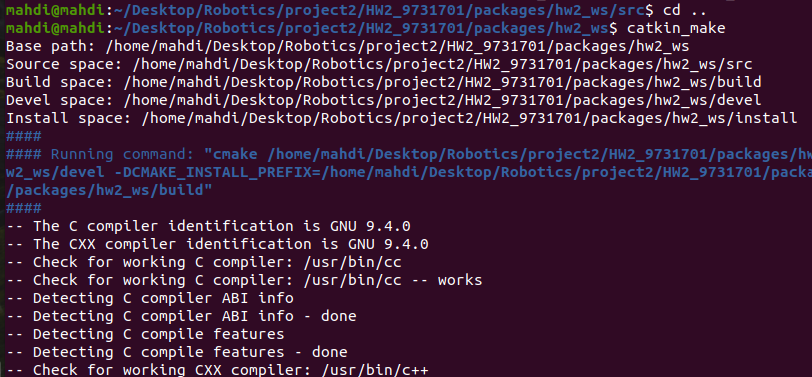


1. خطوط 71 تا 74 هم uncomment شوند:
2. برای آنکه کارمان تکمیل شود، در همان پوشه get\_destination یک پوشه به نام srv درست میکنیم و بعد داخل آن یک فایل با نام اونی که در استپ 3 مشخص کردیم میذاریم.

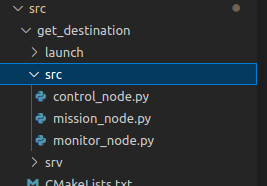


1. توی فایل فوق لازمه که ورودی ها و خروجی های سرویس را بگوییم. بین ورودی ها و خروجی ها با --- جدا میشود. اگرم ورودی نداره خط اول همون --- میشود.



1. حال به پوشه hw2\_ws در ترمینال برگشته و catkin\_make میکنیم.

حال به سراغ ساخت نودها میرویم. برای این منظور در فولدر src مربوط به get\_destination 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 10- تا 10 به طور رندوم تولید میکنیم و برای y باتوجه به اینکه استیت بعدی باید حداقل 5 متری این یکی باشد پس باید در بیرون از یک دایره با شعاع 5 متری آن باشد. روش دیگر این بود که دائما عدد رندوم تولید کنیم به نحوی که فاصله بیشتر از 5 شود ولی خب ممکن است بدشانس باشیم و خیلی طولانی شود.

#!/usr/bin/python3

import rospy

from get\_destination.srv import GetNextDestination, GetNextDestinationResponse

import random

import math

class Mission():

    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(-10,10)

        if abs(next\_x - cur\_x)>=5:

            next\_y = random.uniform(-10,10)

        else:

            y\_dist = (25 - (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 >=10:

                next\_y = random.uniform(-10, valid\_min\_y2)

            elif valid\_min\_y2 <= -10:

                next\_y = random.uniform(valid\_min\_y1, 10)

            else:

                next\_y = random.choice([random.uniform(-10, valid\_min\_y2), random.uniform(valid\_min\_y1, 10)])

        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. 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 در این قسمت پارامترهای مختلف را معرفی میکنیم و linear\_speed را کاربر میتواند به عنوان ورودی از طریق تگ param در لانچ فایل بدهد.

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

در تابع ران که کل کار را در بر دارد، ابتدا لازم است تا یک پیام ریکوئست بسازیم که شامل مکان فعلی ربات است و آن را برای سرویس بفرستیم و در جواب آن به ما next\_x و next\_y را بدهد. در گام بعد با توجه به مکان فعلی و مکان بعدی باید زاویه مناسب را بیابیم که به آن جهت بچرخیم و روبروی آن قرار بگیریم. برای این منظور باید دائما هدینگ ربات را بگیریم تا ببینیم در جهت مناسب قرار گرفتیم یا نه. سپس لازم است تا مستقیم به سمت هدف حرکت کرده. برای اینکه ببینیم به مکان موردنظر رسیدیم یا نه میتوان فاصله نقطه فعلی تا هدف را حساب کرده و هدف را رسیدن به آن فاصله بگذاریم. سپس به مرور با جلو رفتن ربات دائما مکان ربات را گرفته و فاصله اش را تا نقطه مبدا حساب میکنیم.

لازم به ذکر است وقتی Twist() را پابلیش میکنیم همه مقادیرش 0 است و لذا ربات می ایستد. دذر انتهای هر ایتریشن هم ارور که فاصله نقطه هدف تا نقطه ای هست که به آن رسیدیم را حساب میکنیم و به یک لیست اضافه مینماییم. در پایان کار بین اعضای لیست میانگین گرفته و به عنوان ارور نهایی اعلام میکنیم. کد آن در ادامه آمده است.

#!/usr/bin/python3

import rospy

import tf

import math

from nav\_msgs.msg import Odometry

from geometry\_msgs.msg import Twist

from get\_destination.srv import GetNextDestination, GetNextDestinationRequest

class Controller():

    def \_\_init\_\_(self) -> None:

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

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

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

        self.linear\_speed = rospy.get\_param("/controller/linear\_speed") # m/s

        self.angular\_speed = 0.08

self.epsilon = 0.17

self.current\_x = 0

self.current\_y = 0

self.yaw = 90

    # heading of the robot

    def get\_heading(self):

        # 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 math.degrees(yaw)

    # position of robot

    def get\_pose(self):

        # 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 run(self):

        error\_array = []

        counter = 0

        while not rospy.is\_shutdown():

            counter += 1

            if counter == 6:

                break

            # first we should send our current location to service for

            # getting next destination

            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)

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

            # according to the next destination we should calculate our desired angle

            # for finding desired angle we should find next location relative robot position

            new\_next\_x = resp.next\_x - self.current\_x

            new\_next\_y = resp.next\_y - self.current\_y

            sin = new\_next\_y/math.sqrt(new\_next\_x\*\*2+new\_next\_y\*\*2)

            cos = new\_next\_x/math.sqrt(new\_next\_x\*\*2+new\_next\_y\*\*2)

            sin\_inv = math.degrees(math.asin(sin))

            if cos >= 0:

                desired\_angle = sin\_inv

            elif cos <= 0 and sin >=0:

                desired\_angle = 180 - sin\_inv

            else:

                desired\_angle = -180 - sin\_inv

            rospy.loginfo(f"Client: desired angle : {desired\_angle}")

            # our robot start to rotate

            twist = Twist()

            if 90<=desired\_angle<=180 or -180<=desired\_angle<=-90:

twist.angular.z = self.angular\_speed

else:

twist.angular.z = - self.angular\_speed

            while abs(self.yaw - desired\_angle) >= self.epsilon:

                self.yaw = self.get\_heading()

                self.cmd\_publisher.publish(twist)

            # our robot reached to its desired angle and now should be stopped

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

            rospy.sleep(1)

            # now we should go sraight to reach our destination

            twist = Twist()

            twist.linear.x = self.linear\_speed

            self.cmd\_publisher.publish(twist)

            desired\_distance = math.sqrt((self.current\_x-resp.next\_x)\*\*2 + (self.current\_y-resp.next\_y)\*\*2)

            prev\_x = self.current\_x

            prev\_y = self.current\_y

            walked\_distance = math.sqrt((self.current\_x-prev\_x)\*\*2 + (self.current\_y-prev\_y)\*\*2)

            while abs(desired\_distance-walked\_distance) >= self.epsilon:

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

                walked\_distance = math.sqrt((self.current\_x-prev\_x)\*\*2 + (self.current\_y-prev\_y)\*\*2)

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

            # calculate error and append to list

            error\_array.append(math.sqrt((self.current\_x-resp.next\_x)\*\*2+(self.current\_y-resp.next\_y)\*\*2))

            rospy.sleep(1)

        rospy.loginfo(f"TOTAL ERROR (in meter): {sum(error\_array)/len(error\_array)}")

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

    controller = Controller()

    controller.run()

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 get\_destinations کد زیر را اجرا کنیم:

chmod +x src/\*.py

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

<launch>

    <arg name="linear\_speed" default="0.2"/>

    <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="1.57075"/>

    </include>

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

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

        <param name="linear\_speed" value="$(arg linear\_speed)" />

    </node>

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

    <include file="$(find turtlebot\_rviz\_launchers)/launch/view\_robot.launch"/>

</launch>

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 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 و همچنین لانچ فایل مربوط به rviz را هم صدا بزنیم.

برای rviz لازم است که مانند پکیج‌هایی که در ابتدای کار در ورک اسپیس نصب کردیم، این پکیج را نیز به کمک دستور زیر نصب کنیم:

* git clone -b indigo <https://github.com/turtlebot/turtlebot_interactions.git>

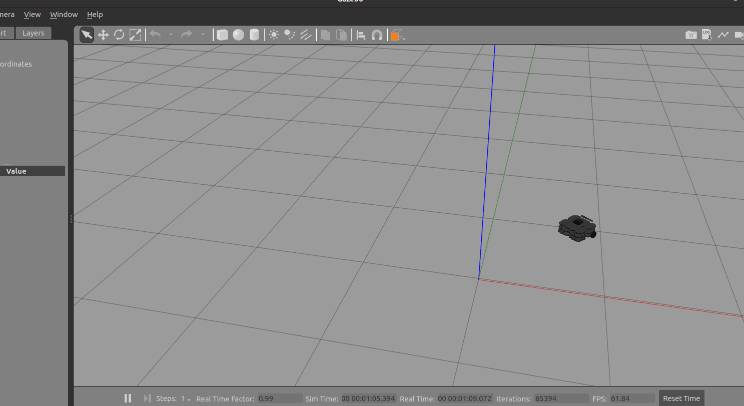
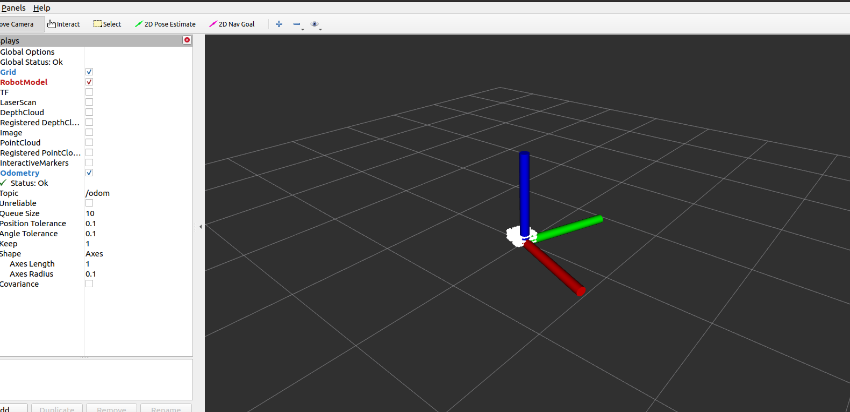
سپس در آخر لازم است تا به دایرکتوری ورک اسپیس برویم و catkin\_make را صدا بزنیم.

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
* Text

  Description automatically generatedroslaunch get\_destination get\_destination.launch linear\_speed:=0.4

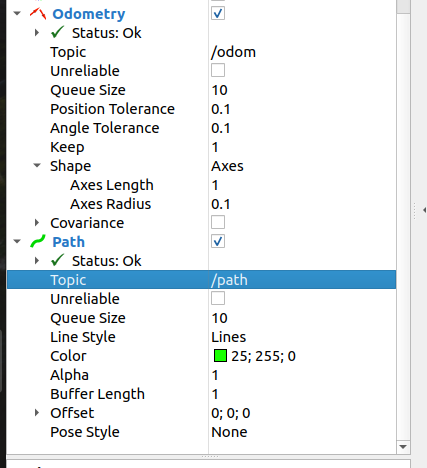
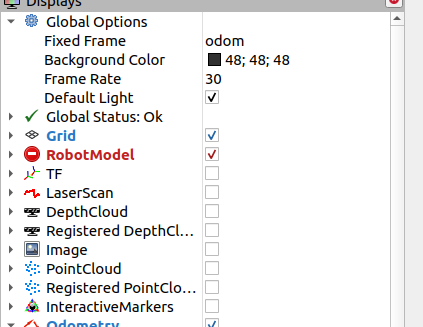
در تصویر سمت راست نمونه ای از rviz و در چپ نمونه ای از تصویر در gazebo را میبینید.



Graphical user interface, text, application

Description automatically generatedسپس باید در rviz تاپیک مد نظرمان را معرفی کنیم که path را رسم کند. برای این کار باید ابتدا path را اضافه کنیم:

سپس تنظیمات را به صورت زیر تنظیم کنیم:

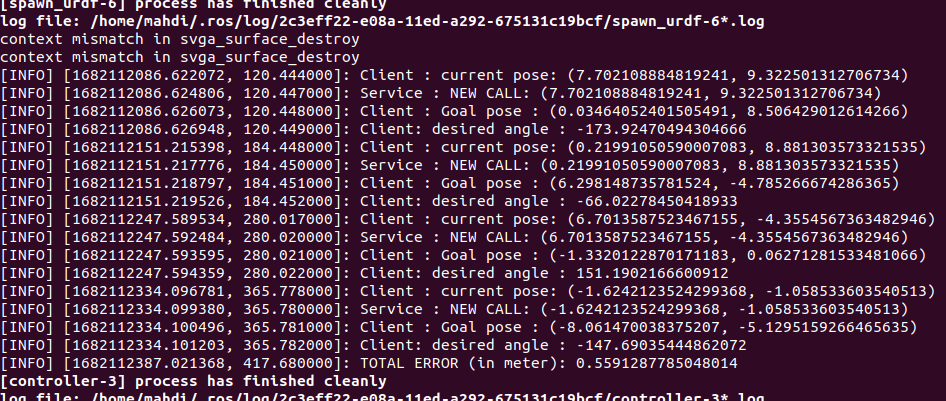


در نهایت شکلی مانند زیر را مشاهده خواهیم کرد و خطای مورد نظر پس از 5 ایتریشن در ترمینال نمایش داده میشود:

Chart

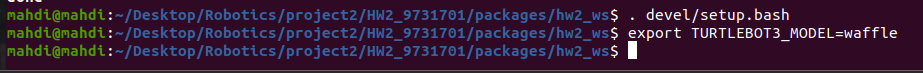
Description automatically generated

برای مثال برای سرعت خطی 0.4 متر بر ثانیه خطا برابر 0.5 متر به طور میانگین میباشد:

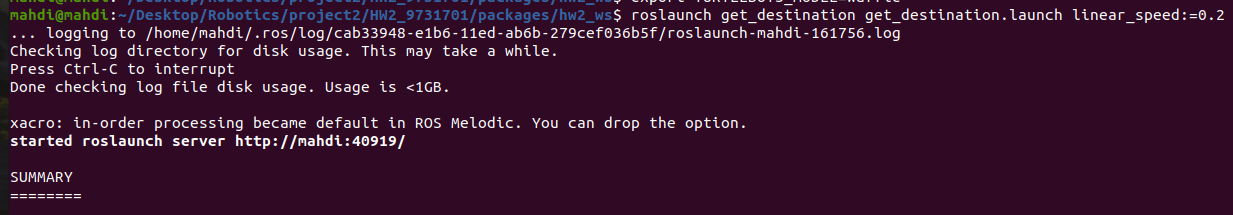


# گام اول – نتایج

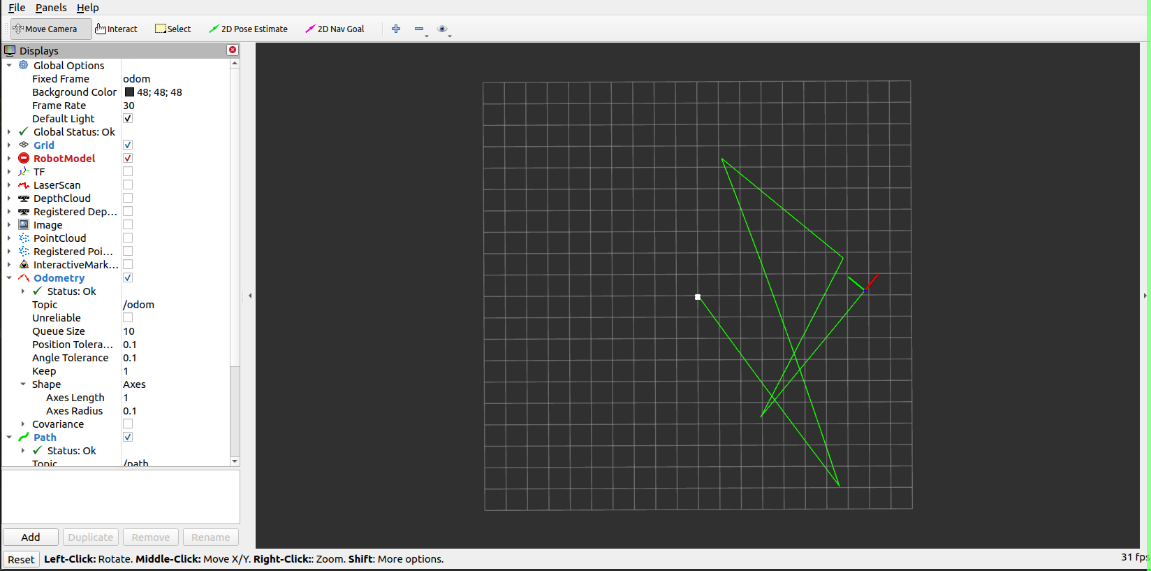
1. **ویدیوی کارکرد ربات برای هر سرعت خطی:** در فولدر مربوطه 3 فیلم که هر یک مربوط به یکی از سرعت‌های 0.2 و 0.4 و 0.8 متر بر ثانیه است، قرار گرفته است. دقت شود به علت خواسته سوال که باید حداقل 5 ایترشن نمایش داده شود و همچنین برای بالا بردن دقت، که مجبور به کم کردن سرعت زاویه ای شدیم، زمان فیلم‌ها با عرض پوزش بیشتر شد.
2. **نمای مسیر ربات در شبیه ساز Rviz به ازای هر سرعت خطی:**

برای این منظور در ترمینال در ورودی لازم است سرعت خطی را بدهیم. باتوجه به آنکه در دستورکار گفته شده بود حداقل 5 بار فرآیند یافتن مقصد جدید ادامه یابد، من هم برای طولانی تر نشدن ویدیوها دقیقا همان 5 بار را ست کردم. ابتدا لازم است که ترمینال را سورس کنیم و نوع ربات را اکسپورت کنیم:

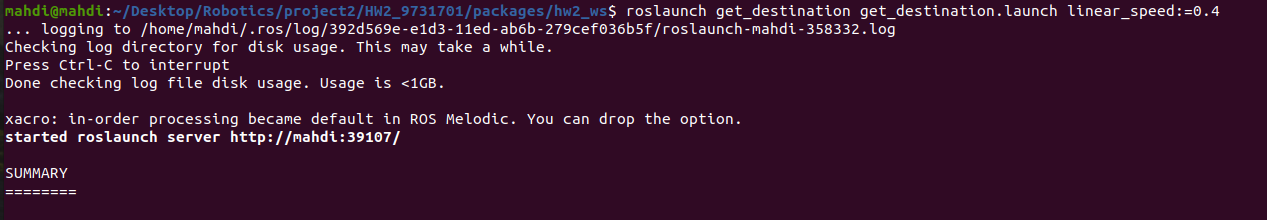
* **برای سرعت 0.2 m/s :**

ابتدا دستور زیر را وارد میکنیم:

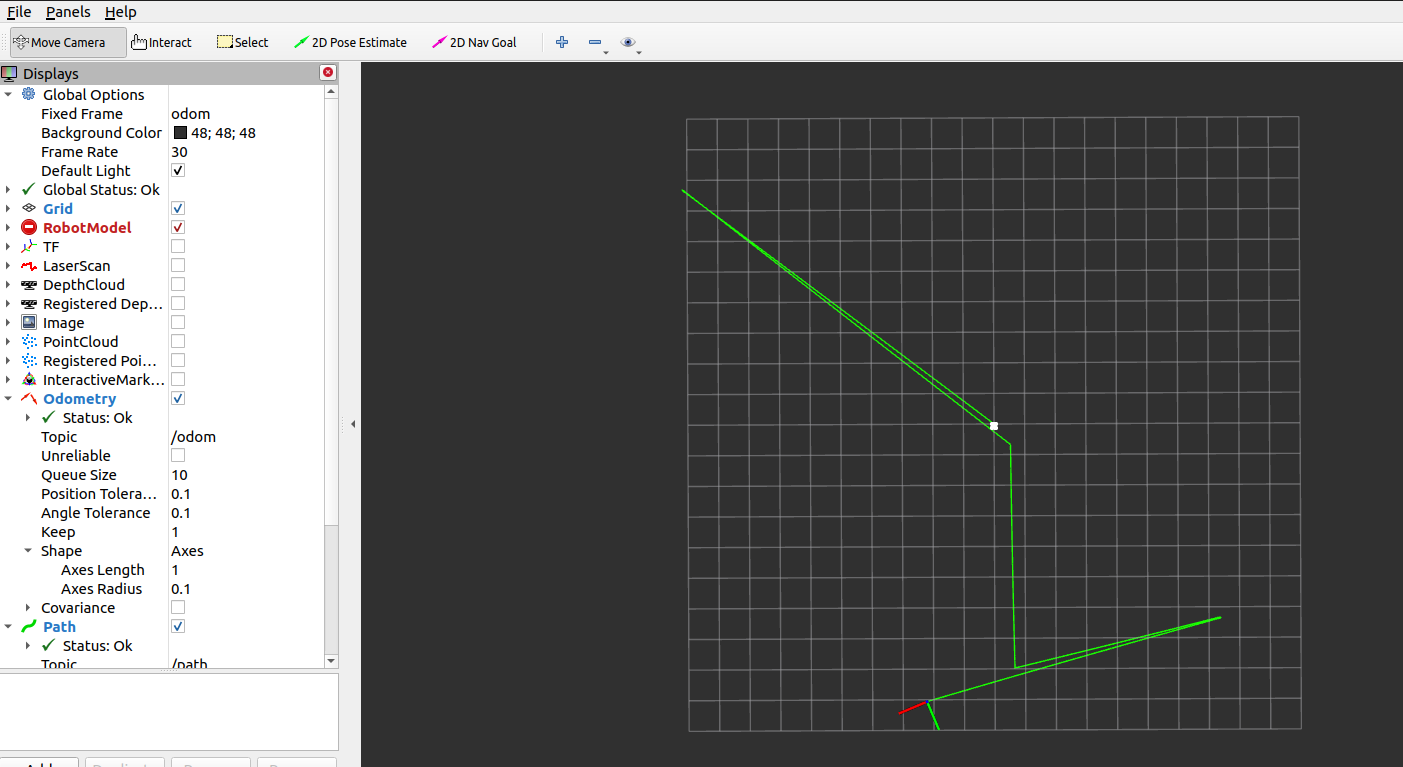
در نهایت نمای مسیر ربات در Rviz پس از 5 iteration به صورت زیر خواهد بود:



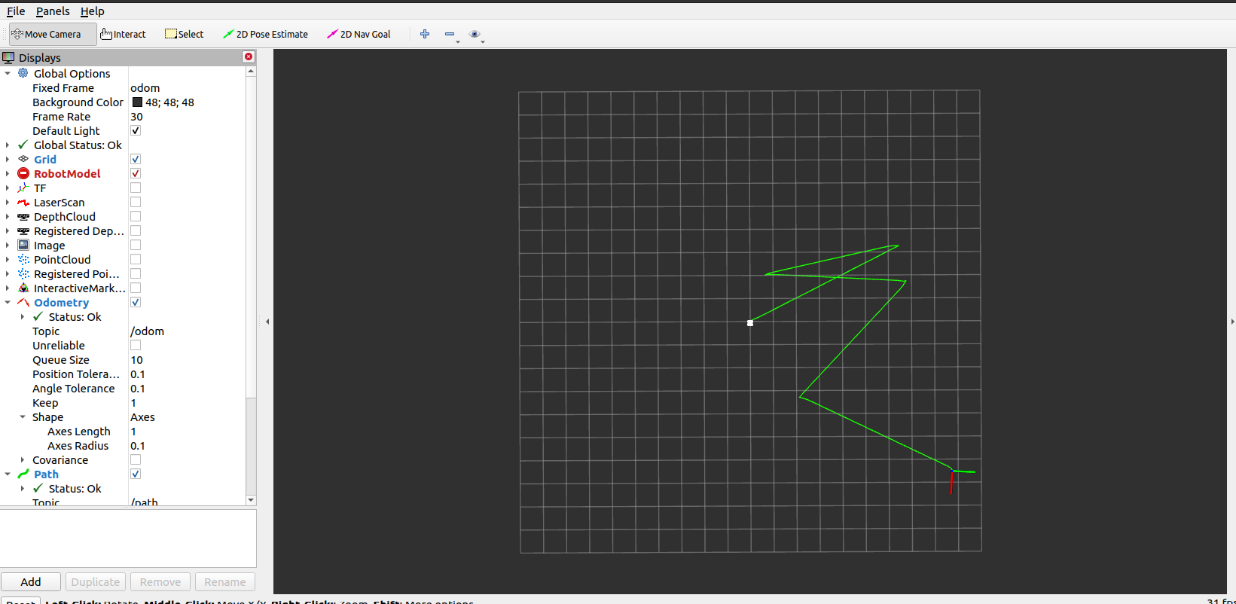
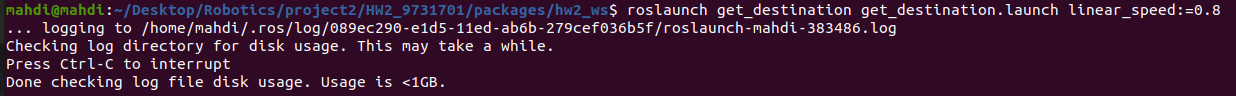
* **برای سرعت 0.4 m/s :**

ابتدا دستور زیر را وارد میکنیم:

در نهایت نمای مسیر ربات در Rviz پس از 5 iteration به صورت زیر خواهد بود:

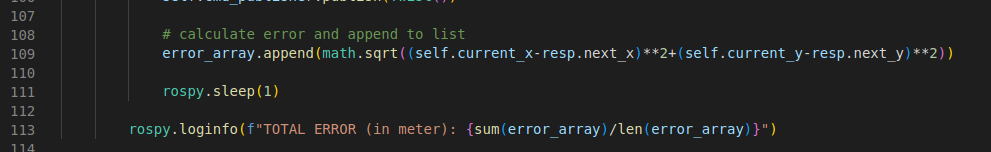


* **برای سرعت 0.8 m/s :**

ابتدا دستور زیر را وارد میکنیم:

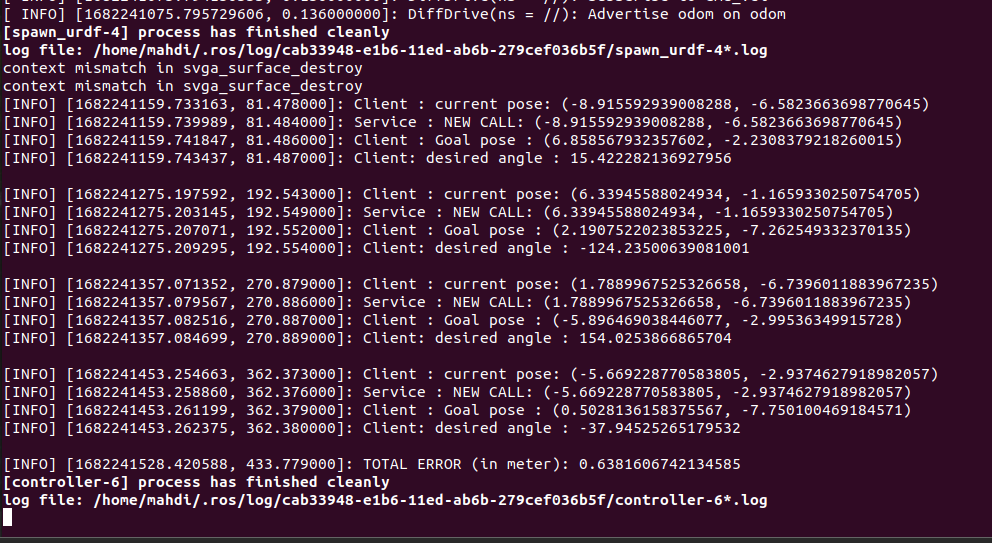
در نهایت نمای مسیر ربات در Rviz پس از 5 iteration به صورت زیر خواهد بود:

1. **خطای انحراف از مقصد به ازای هر سرعت خطی. برای این کار می توانید پس از ایست ربات فاصله‌اش را تامقصد کنونی به دست آورده و درنهایت میانگین فواصل را گزارش دهید.**

طبق راهکار ارائه شده در کد نیز به این صورت عمل کردیم که پس از پیمودن مسافت مد نظر، فاصله مقصد مورد نظر را از مقصدی که فعلا به آن رسیدیم، میابیم. در انتهای هر ایتریشن باتوجه به توضیحات ارور را حساب کردیم و به یک لیست آن را اضافه کردیم. در نهایت جمع ارورها را بر تعدادشان تقسیم کرده تا میانگین خطا به دست آید.

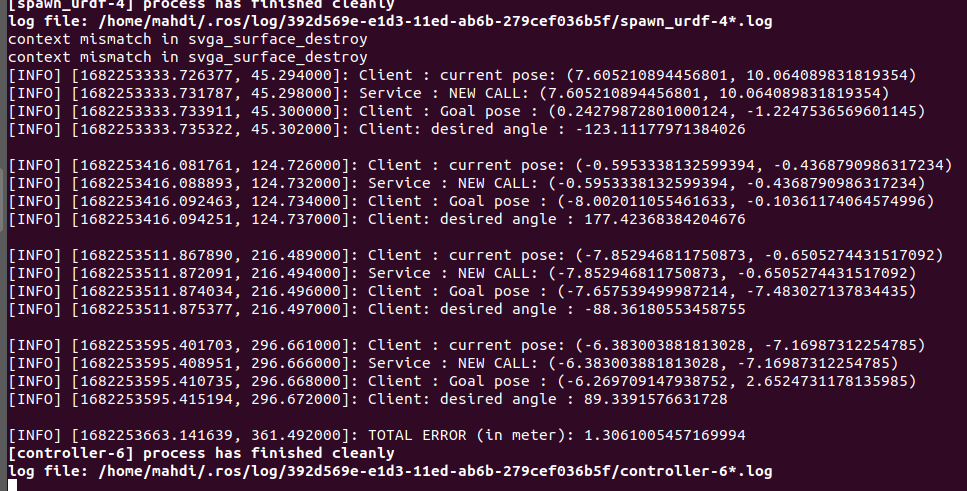
* **برای سرعت 0.2 m/s :**

همانطور که مشاهده میکنید برای این حالت میانگین خطا برابر 0.68 m میباشد.



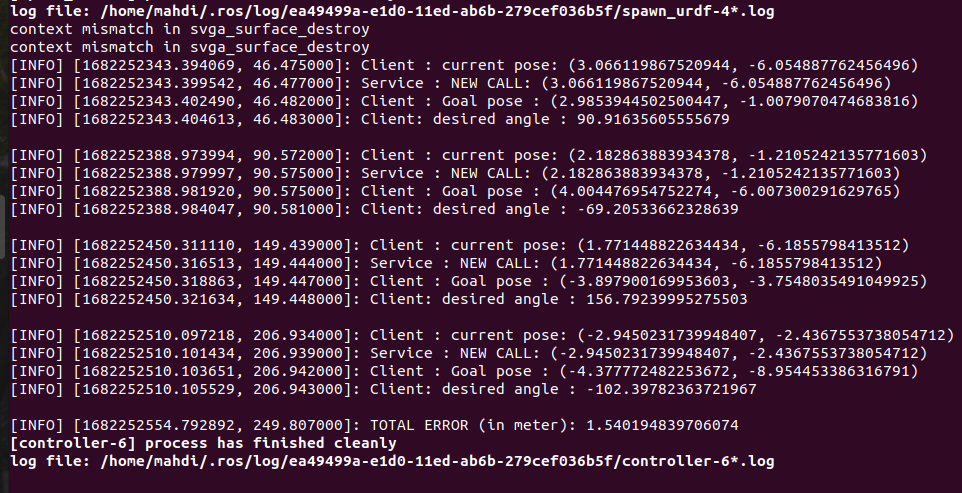
* **برای سرعت 0.4 m/s :**

همانطور که مشاهده میکنید برای این حالت میانگین خطا برابر 1.3 m میباشد. که نسبت به قبل بیشتر شد.



* **برای سرعت 0.8 m/s :**

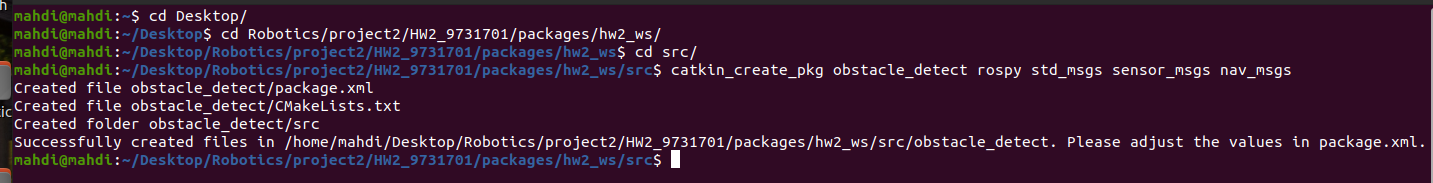
همانطور که مشاهده میکنید برای این حالت میانگین خطا برابر 1.54 m میباشد. که نسبت به قبل بیشتر شد.



همانطور که دیده میشود با زیاد شدن سرعت میزان خطا نیز افزایش میابد چراکه میزان لغزش بیشتر شده و دقت odometry پایین میاید. البته در مقادیر فوق، به علت رندم بودن مقصدها، میزان مسافت طی شده هم تاثیر دارد و ممکن است در یک اجرا نقاط مقصد نزدیک به هم باشند ولی به هرحال آنچه مشهود است با افزایش سرعت خطی دقت کاهش میابد. اگر در ویدیو هم نگاه شود حتی در هنگام شروع به حرکت خطی، ربات یک دریفتی دارد و هر دوچرخ به یک اندازه نمیچرخند که دقیقا روی مسیر تعیین شده بروند.

# گام دوم – تشخیص مانع به کمک لایدار

حال لازم است که یک پکیج مثلا با نام get\_destination در فولدر سورس بسازیم. همچنین dependencyهای لازم را که ممکن است در نوشتن نودها و کد زنی نیاز شود به آن میدهیم:

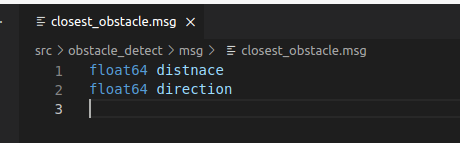
* catkin\_create\_pkg obstacle\_detect rospy std\_msgs sensor\_msgs nav\_msgs

حال در گام بعد در مسیر ورک اسپیس میایم و vs code را باز میکنیم. لازم است در پوشه obstacle\_detect یک پوشه به نام msg ایجاد کنیم. سپس در آن باید custom message را ایجاد نماییم.

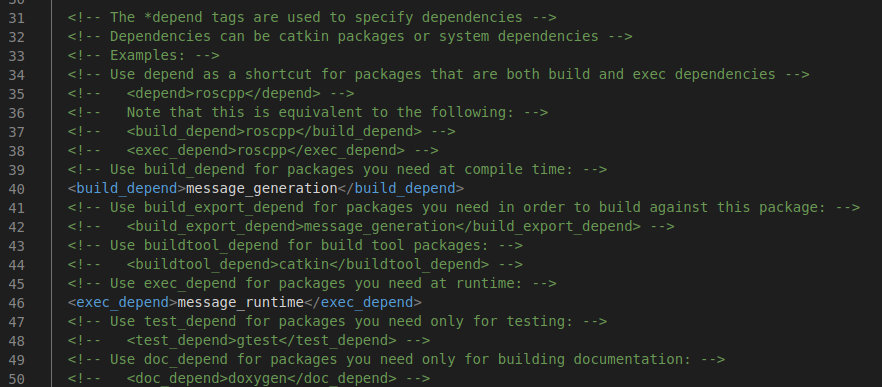
Text

Description automatically generated

همچنین محتوای داخل این پیام به صورت زیر میباشد:



حال لازم است تا تنظیمات مربوط به custom message را انجام دهیم. برای این منظور CMakeLists.txt و همچنین package.xml را باز میکنیم. برای این منظور میتوانید به قسمت tutorials سطح beginner در wiki.ros مراجعه میکنیم وقسمت CreatingMsgAndSrv میرویم. ([لینک](http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv)) براساس آن قرار است پیش برویم.

اولین تغییر در package.xml میباشد. دو خطی که در خطوط 40 و 46 هستند را uncomment کنید.

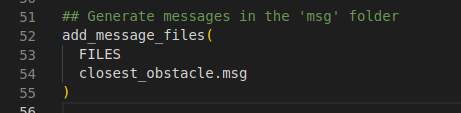
Text

Description automatically generatedحال باید به سراغ تغییرات داخل CMakeLists برویم. یکی اینکه در find\_package در خط 10 باید message\_generation را اضافه نماییم .

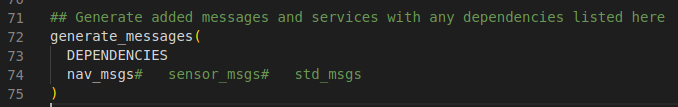
Graphical user interface, text

Description automatically generatedتغییر دیگر در catkin\_package در خط 107 میباشد که باید uncomment نماییم و بقیه موارد را پاک کرده و message\_runtime را جلوی CATKIN\_DEPENDS اضافه مینماییم.

سپس به دنبال add\_message\_files میگردیم و آن را uncomment میکنیم و closest\_obstacle.msg را به جای Message1.msg و Message2.msg میگذاریم.



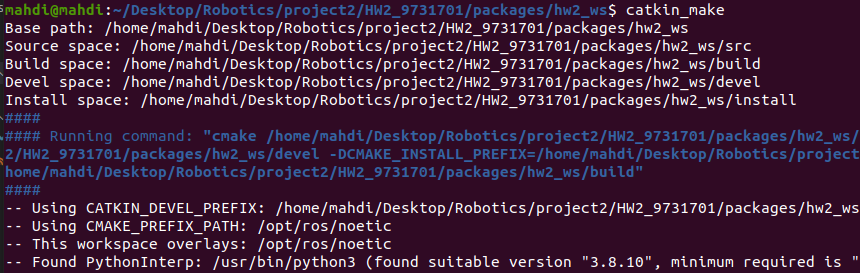
حال باید generate\_message را از comment در بیاریم.



کار دیگری که باید بکنیم اینه که در خط 115 اون include را هم uncomment کنیم

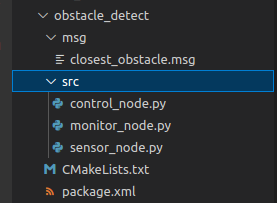
Text

Description automatically generated

حال به علت تغییراتی که در این دو فایل اعمال کردیم لازم است در work space یک بار catkin\_make را بزنیم:

حال یک بار هم vscode را بسته و دوباره باز میکنیم. حال میتوان import را انجام داد. ممکن است هنوز مسیج ساخته شده را در غیر اینصورت نشناسد.

خب حال به سراغ ساخت نودها میرویم. در فایل src لازم است 3 نود ایجاد کنیم. یکی sensor\_node.py که باید با خواندن laser scan کمترین فاصله از موانع را پیدا کرده و در تاپیک مناسب برگرداند. یکی control\_node.py که وظیفه کنترل ربات اعم از چرخش مناسب و حرکت روبه جلو را دارد. و یک نود monitor\_node.py که برای نمایش مسیر ربات در Rviz لازم میباشد.



حال به سراغ نوشتن کد در sensor\_node.py میرویم:

#!/usr/bin/python3

import rospy

from obstacle\_detect.msg import closest\_obstacle

from sensor\_msgs.msg import LaserScan

class Sensor():

    def \_\_init\_\_(self) -> None:

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

        self.pub = rospy.Publisher('ClosestObstacle', closest\_obstacle, queue\_size=10)

        self.laser\_subscriber = rospy.Subscriber("/scan" , LaserScan , callback=self.laser\_callback)

        self.rate = rospy.Rate(0.2) #Hz

        self.closest\_msg = closest\_obstacle()

    def laser\_callback(self, msg: LaserScan):

        min\_dist = min(msg.ranges)

        direction = msg.ranges.index(min\_dist)

        self.closest\_msg.distnace = min\_dist

        self.closest\_msg.direction = direction

    def run(self):

        while not rospy.is\_shutdown():

            rospy.loginfo(f' closest obtacle {self.closest\_msg}')

            self.pub.publish(self.closest\_msg)

            self.rate.sleep()

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

    my\_sensor = Sensor()

    my\_sensor.run()

کار نود فوق این است مقدار minimum فاصله ای که از آرایه ranges از پیام scan دریافت میکند را به همراه ایندکس این فاصله در این آرایه که نشان دهنده همان زاویه مورد نظر است را به ترتیب در فیلدهای distance و direction جاگذاری کند و publish کند.

حال به سراغ نود control\_node.py میرویم:

#!/usr/bin/python3

import rospy

import tf

import math

from nav\_msgs.msg import Odometry

from geometry\_msgs.msg import Twist

from obstacle\_detect.msg import closest\_obstacle

class Controller():

    def \_\_init\_\_(self) -> None:

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

        rospy.Subscriber('ClosestObstacle', closest\_obstacle, self.closest\_callback)

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

        self.linear\_speed = 0.2 # m/s

        self.angular\_speed = 0.2

        self.closest\_obstacle = closest\_obstacle()

    def closest\_callback(self, data):

        self.closest\_obstacle = data

    # heading of the robot

    def get\_heading(self):

        # 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 math.degrees(yaw)

    def move(self, mode, direction=0):

        twist = Twist()

        self.cmd\_publisher.publish(twist)

        rospy.sleep(1)

        if mode == "rotate":

            first\_heading = self.get\_heading()

            if direction <= 180:

                while abs(self.get\_heading()-first\_heading) < (180-direction):

                    twist.angular.z = -self.angular\_speed

                    self.cmd\_publisher.publish(twist)

            else:

                while abs(self.get\_heading()-first\_heading) < (direction-180):

                    twist.angular.z = -self.angular\_speed

                    self.cmd\_publisher.publish(twist)

        elif mode == "go\_straight":

            twist.linear.x = self.linear\_speed

            self.cmd\_publisher.publish(twist)

        else:

            rospy.loginfo('the mode is not correct')

    def run(self):

        sensor\_is\_ready = False

        self.move("go\_straight")

        while not rospy.is\_shutdown():

            dist, direction = self.closest\_obstacle.distnace, self.closest\_obstacle.direction

            if dist == 0.0 and direction == 0.0 and not sensor\_is\_ready:

                continue

            sensor\_is\_ready = True

            if dist < 2 and not (120 < direction < 240):

                rospy.loginfo(f'controller : {dist, direction}')

                self.move("rotate", direction)

                self.move("go\_straight")

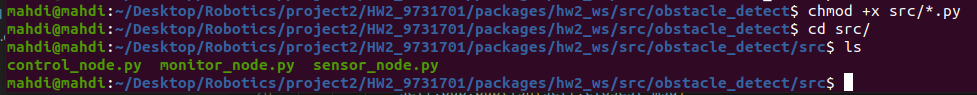
if \_\_name\_\_ == "\_\_main\_\_":

    controller = Controller()

    controller.run()

در نود فوق وظیفه داریم، که پیام‌های publish شده از نود sensor را دائما بخوانیم چنانچه در زاویه دیدمان ( که من برای مثال یک زاویه دید 240 درجه تعریف کردم) مانعی بود که فاصله ارسال شده از سنسور بینگر این بود که کمتر از 2 متری است باید بچرخیم و پشتمان به آن بشود. این زاویه دید برای آن تعریف شده، چون ما زمانی که به مانع میرسیدیم و میچرخیدیم و پشتمان به مانع میشد دوباره برای ما پیامی میامد که مانعی در فاصله کمتر از دو متری پیدا شده و در پشت شما است و دوباره میچرخیدیم و در یک لوپ می افتادیم. میشد با یک بیت flag و تعریف کردن state هم آن را هندل کرد اما باز کمی جای کار داشت چراکه ما دقیقا در دو متری مانع نمی ایستادیم و کمی خطا داریم و مثلا در 1.8 m آن می ایستیم برای مثال. پس در اون حالت هم چنانچه کمی حرکت میکرد باز میچرخید چون در دو متری آن یک مانع یافت میشد. همچنین هر direction ای که به ما بدهد به اندازه 180 – direction باید چرخش کنیم که پشتمان به آن مانع شود. موارد مربوط به چرخش و حرکت خطی در تابع move هندل شده است.

نود monitor\_node.py مانند قسمت قبل میباشد و بنابراین از آوردن مجدد آن خودداری کردم.

حال تمامی فایل‌های پایتون را به صورت زیر executable میکنیم.

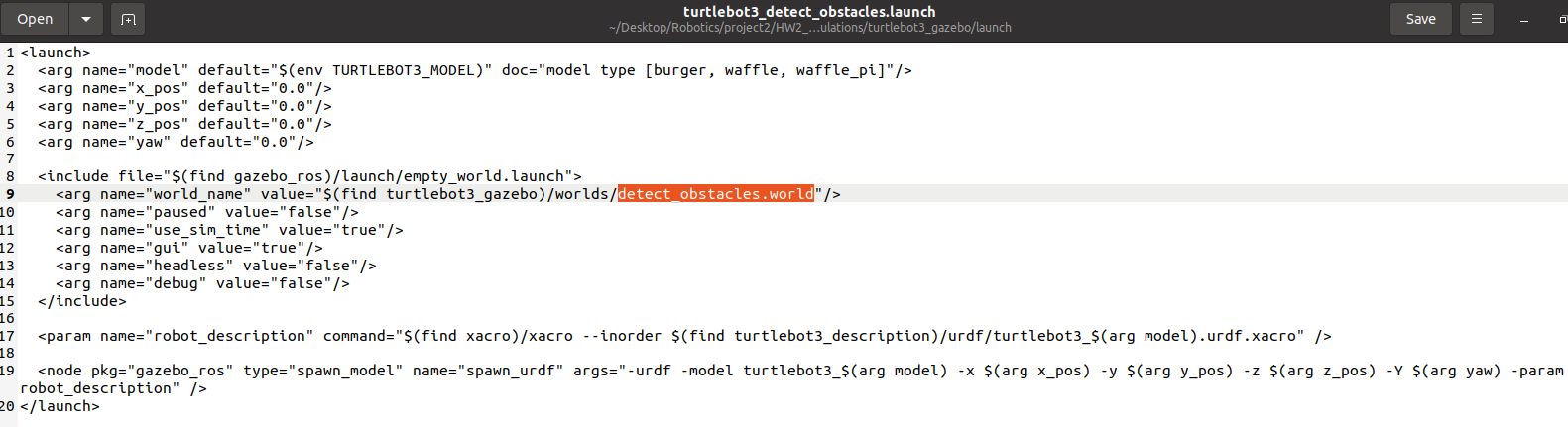
درگام بعدی به سراغ نوشتن لانچ فایل میرویم. طبق خواسته سوال مکان اولیه ربات 0 , 0 و با زاویه 0.75 رادیان میباشد که درون لانچ فایل این مقادیر را اعمال میکنیم.

ابتدا برای راحتی detect\_obstacles.world را در مسیر زیر کپی میکنیم:

/hw2\_ws/src/turtlebot3\_simulations/turtlebot3\_gazebo/worlds

حال لازم است تا در دایرکتوری زیر یک launch file پایه برای بالا آوردن ربات در نقشه بنویسیم.

/hw2\_ws/src/turtlebot3\_simulations/turtlebot3\_gazebo/launch$

ابتدا یک کپی از my\_empty\_world.launch که آن را در گام یک خودمان از روی turtlebot3\_emoty\_world.launch ساختیم، تهیه میکنیم. سپس اسم آن را عوض میکنیم به turtlebot3\_detect\_obstacles.launch و آن را باز میکنیم و به صورت زیر تغیر میدهیم:

حال به سراغ launch فایل اصلی که داخل پکیج detect\_obstacle ایجاد کردیم، میرویم.

<launch>

    <include file="$(find turtlebot3\_gazebo)/launch/turtlebot3\_detect\_obstacles.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="0.75"/>

    </include>

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

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

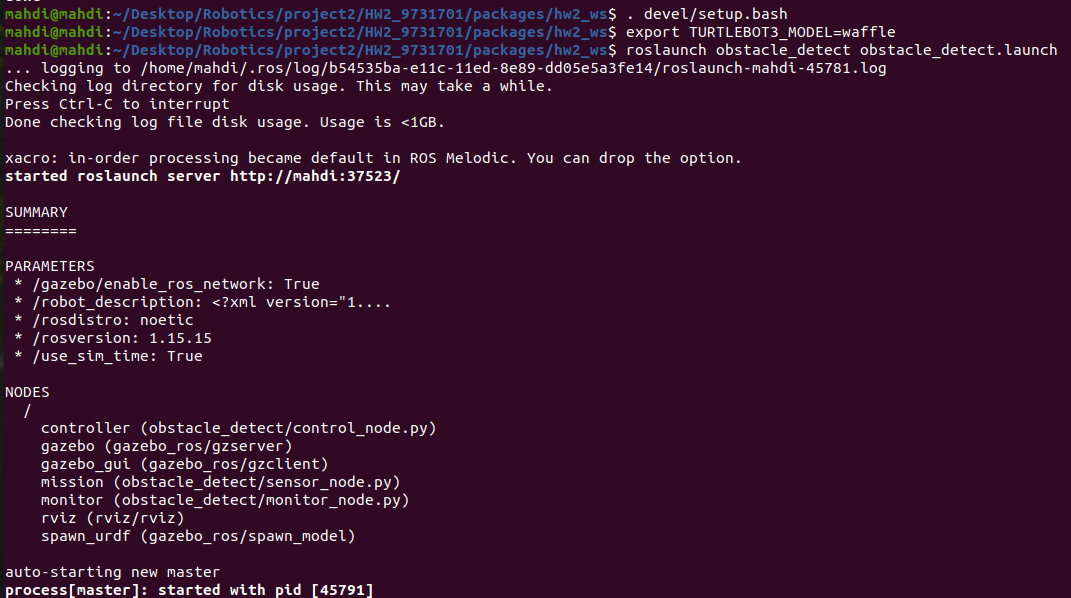
    <node pkg="obstacle\_detect" type="monitor\_node.py" name="monitor"></node>

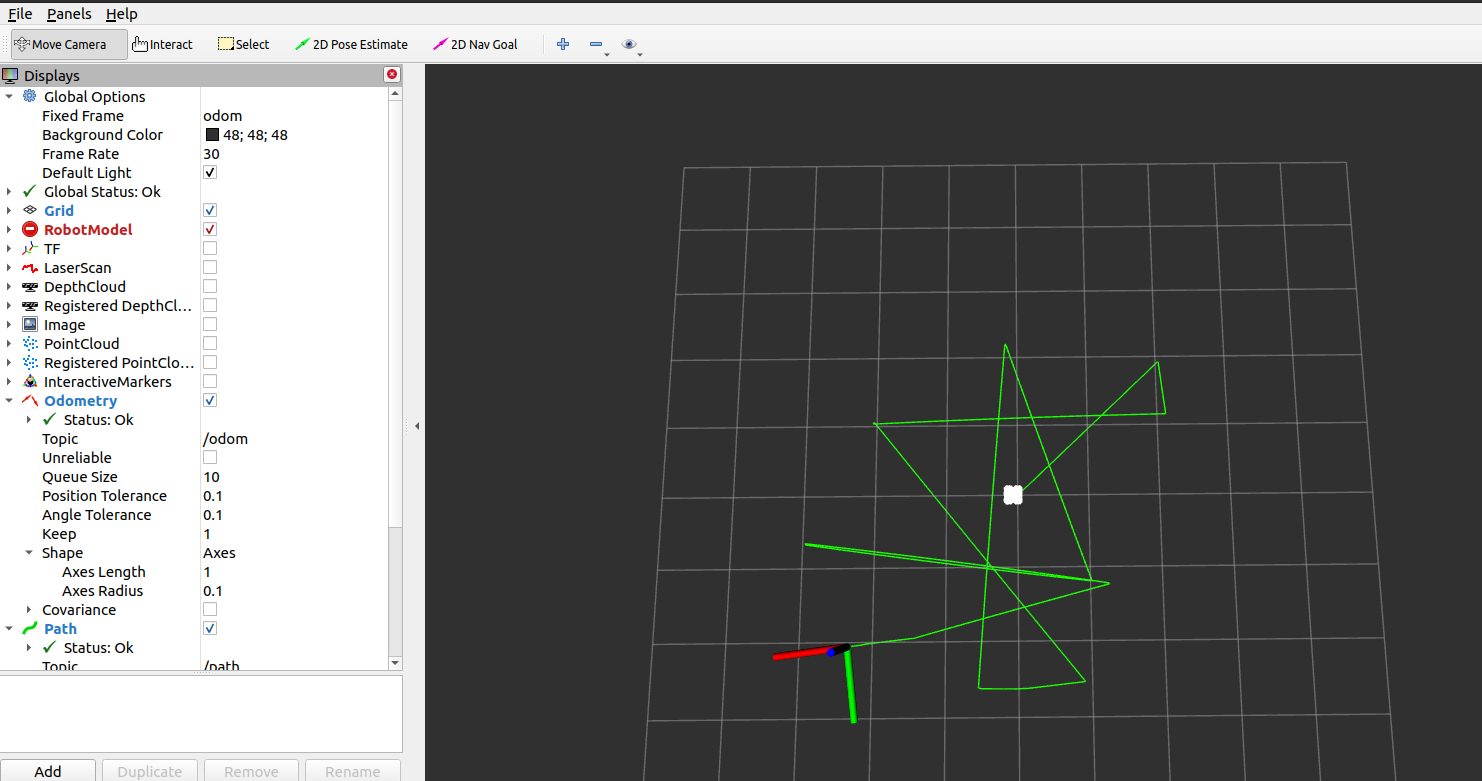
    <include file="$(find turtlebot\_rviz\_launchers)/launch/view\_robot.launch"/>

</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 obstacle\_detect obstacle\_detect.launch

برای مثال نمونه ای از مسیری که ربات پیموده در تصویر زیر قابل مشاهده میباشد.

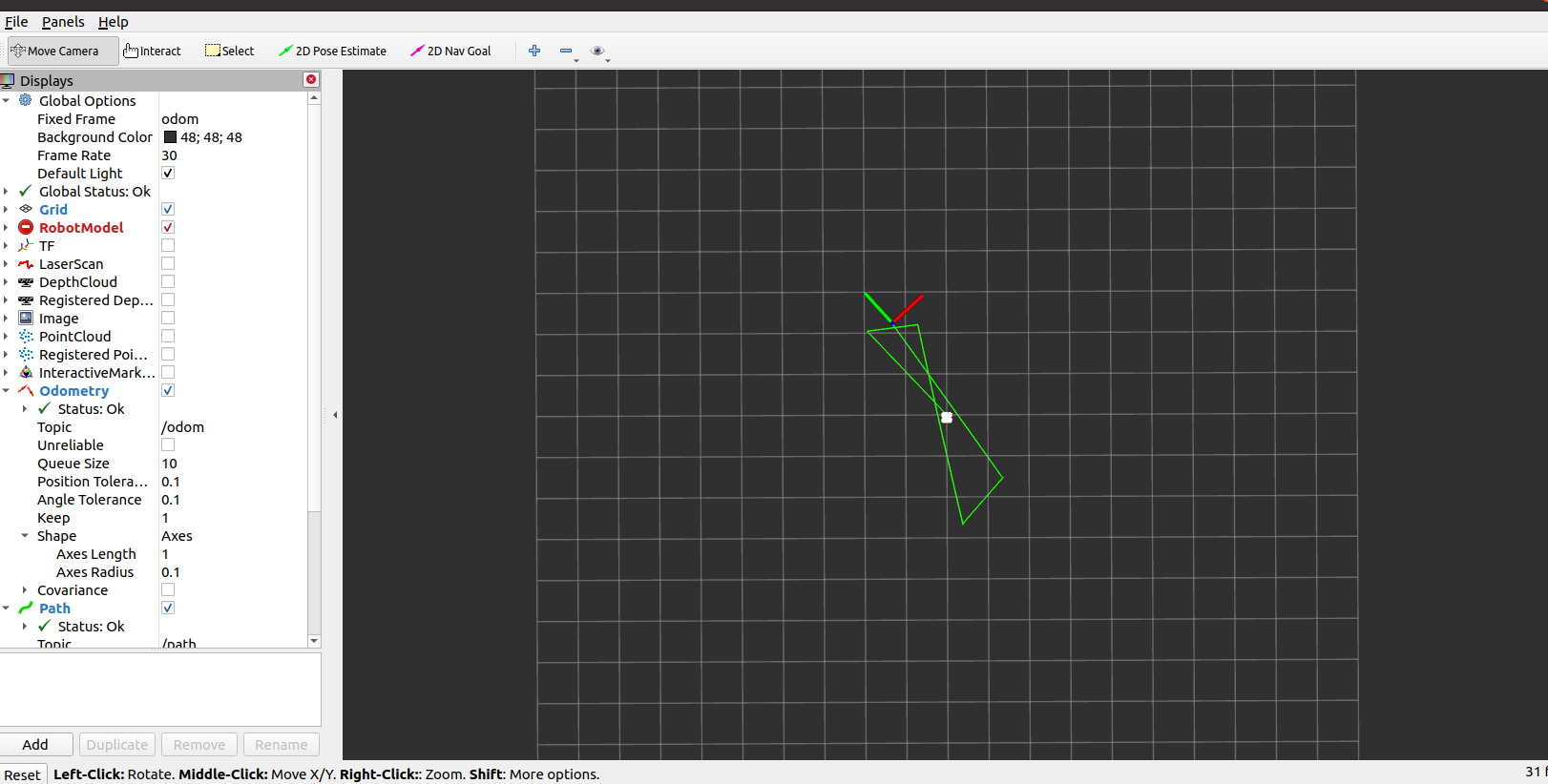
# گام دوم – نتایج

1. ویدیوی کارکرد ربات:

**ویدیوی مذکور ضبط شده و با فرمت گفته شده در یک پوشه ارسال شده است.**

**دقت شود که برای آنکه کمی سرعت بالاتر رود که زمان فیلم کم شود، سرعت خطی و زاویه ای را کمی زیاد کردم که باعث میشود دقت پایین تر بیاید و برای مثلا 2 متر را کمی رد میکند و سپس متوجه میشود با اینکه در کد شرط را روی کمتر از 2 متر گذاشته بودم.**

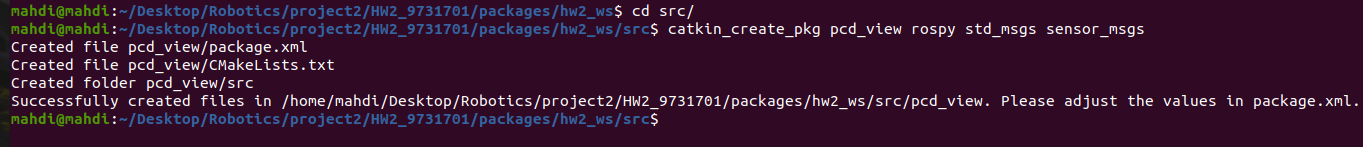
1. نمای مسیر ربات در شبیه ساز Rviz

**نمونه ای از آن را در صفحه قبل میتوانید مشاهده کنید. نمونه دیگری که نمونه ضبط شده در ویدیو هست هم در زیر آمده است.**

# گام سوم – کار با Pointcloud

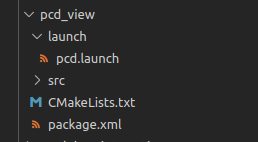
**بخش اول:**

پکیجی به نام pcd\_view بسازید.

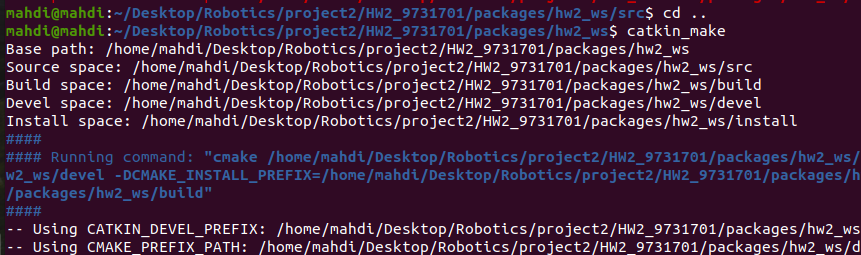
* ****catkin\_create\_pkg pcd\_view rospy std\_msgs sensor\_msgs

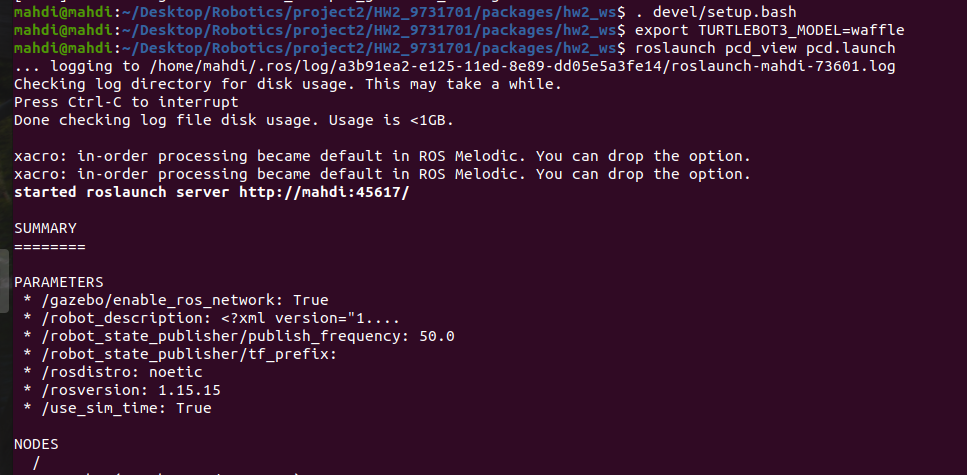
یک launch file در آن گذاشته و launch file های turtlebot3\_stage\_4.launch و turtlebot3\_gazebo\_rviz.launch را به روش زیر در فایل خود include نمایید.

نام لانچ فایل را pcd.launch گذاشتم:

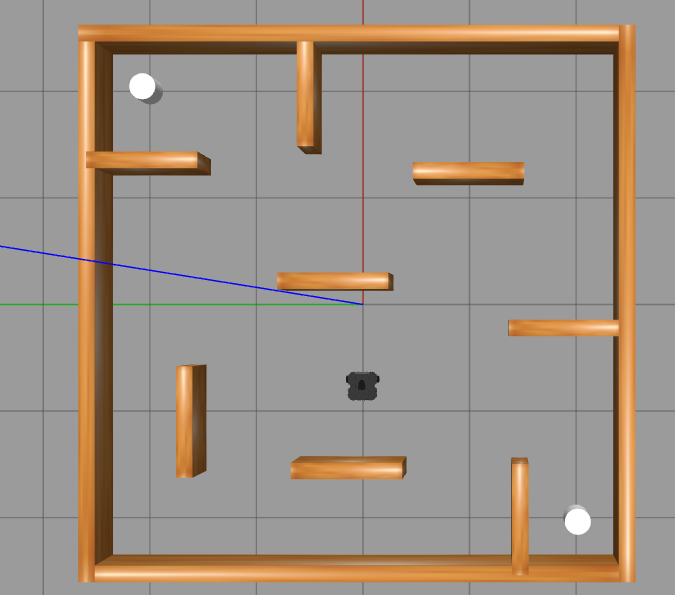
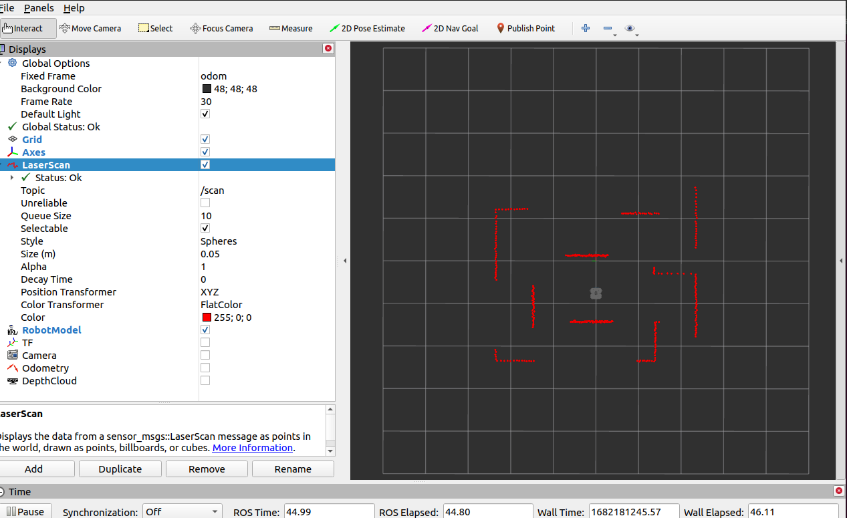


محتوای لانچ فایل به صورت زیر میباشد:

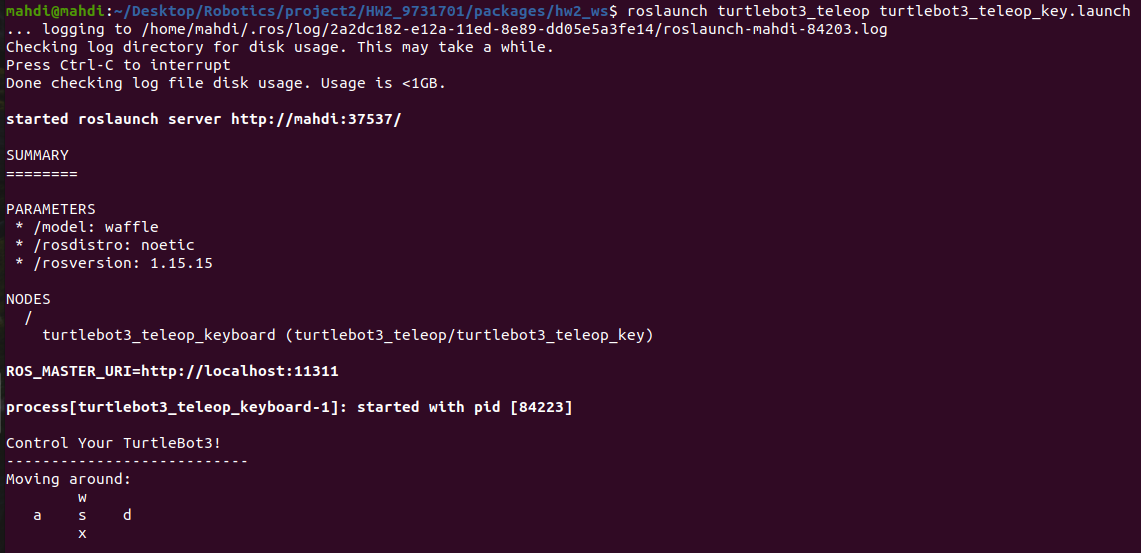
حال باید در ترمینال برویم و در work space دستور catkin\_make را اجرا کنیم:

سپس باید ترمینال را سورس کنیم و مدل ربات را اضافه کنیم و سپس به کمک دستور roslaunch لانچ فایلی که نوشتیم را اجرا کنیم.

با این دستور Gazebo و RViz اجرا می شوند.



برای نمای بهتر می توانید از منوی سمت چپ، Style مربوط به LaserScan را به Points تغییر دهید. اگر برای شما هم ، مثل من 😊، rviz درصورت انجام این کار terminate شد میتوانید مثلا بذارید روی حالت sphere و اندازه نقاط را مثلا 0.05m بذارید. ( برای این موضوع سرچ کردم ظاهرا یک باگ rviz میباشد و نقاط به شدت بزرگ میشوند و برنامه terminate میشود.) میتوانید [لینک1](https://answers.ros.org/question/358930/pointcloud2-large-points/?answer=364514" \l "post-id-364514) و [لینک2](https://github.com/ros-visualization/rviz/issues/1508) را برای این مشکل بخوانید.

سپس در یک ترمینال دیگر turtlebot3\_teleop\_key.launch را از پکج turtlebot3\_teleop ران کنید و ربات را با کیبورد کنترل نمایید.

**بخش دوم:**

در این بخش میتوان به [لینک](https://www.theconstructsim.com/merge-laser-scans-single-pointcloud/) مراجعه کرده و روال کار را متوجه شد. همچنین ویدیوی یوتوب در این [لینک](https://www.dideo.ir/v/yt/6O3QKhkg8zo/%D8%AA%D9%85%D8%A7%D8%B4%D8%A7%DB%8C-%D9%88%DB%8C%D8%AF%D8%A6%D9%88-%D8%A7%D8%B2-%D8%AF%DB%8C%D8%AF%D8%A6%D9%88-watch-video-from-dideo) نیز مراحل کار را گفته است.

در این بخش از داده ی LaserScan به PointCloud می ر سیم.

برای این کار به سرویس laser\_scan\_assembler از پکج laser\_assembler نیاز داریم. این سرویس داده های مربوط به LaserScan را جمع آوری کرده و مجموع آنها را در فرمت یک پیام PointCloud2 از پکج sensor\_msgs باز میگرداند. دستور اجرای نود این سرویس را به انتهای launch file خود اضافه کنید.

دقت کنید که این نود دو پارامتر دریافت می کند:

1. max\_scans : که تعداد بیشینه اسکن‌ها را در یک پیام PointCloud2 مشخص می‌کند. مقدار آنرا " 400 " بگذارید.
2. Fixed\_frame : که فریم publish کردن پیام ها مشخص می کند. برای کاربرد ما این مقدار را " odom "ست کنید

همچنین ما به یک نود برای فرستادن درخواست به سرویس بالا، دریافت و publish کردن پیغام PointCloud2 نیاز داریم. برای نوشتن این نود از AssembleScan2 از پکج laser\_assembler استفاده نمایید.

با توجه به توضیحات ابتدا یک فایل پایتون با نام laser\_to\_pcd.py در قسمت src مربوط به پکیج pcd\_view ایجاد میکنیم. سپس کد زیر را در آن مینویسیم:

#!/usr/bin/python3

import rospy

from laser\_assembler.srv import AssembleScans2

from sensor\_msgs.msg import PointCloud2

class PointCloud():

    def \_\_init\_\_(self) -> None:

        rospy.init\_node("laser\_to\_pcd")

        rospy.wait\_for\_service("assemble\_scans2")

        self.assemble\_scans = rospy.ServiceProxy('assemble\_scans2', AssembleScans2)

        self.pub = rospy.Publisher ("/laser\_pointcloud", PointCloud2, queue\_size=1)

    def run(self):

        while not rospy.is\_shutdown():

            try:

                resp = self.assemble\_scans(rospy.Time(0,0), rospy.get\_rostime())

                #rospy.loginfo("Got cloud with %u points" % len(resp.cloud.data))

                self.pub.publish(resp.cloud)

            except rospy.ServiceException as e:

                rospy.loginfo("Service call failed: %s" %e)

            rospy.sleep(1)

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

    pcd = PointCloud()

    pcd.run()

حال به سراغ لانچ فایل میرویم و تغییرات زیر را اعمال میکنیم. دوتا include اول که توضیحاتش در بخش1 داده شد برای بالا آوردن ربات در gazebo و rviz میباشد.

<launch>

    <include file="$(find turtlebot3\_gazebo)/launch/turtlebot3\_stage\_4.launch"/>

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

    <node pkg="laser\_assembler" type="laser\_scan\_assembler" name="my\_assembler">

        <param name="max\_scans" type="int" value="400" />

        <param name="fixed\_frame" type="string" value="odom" />

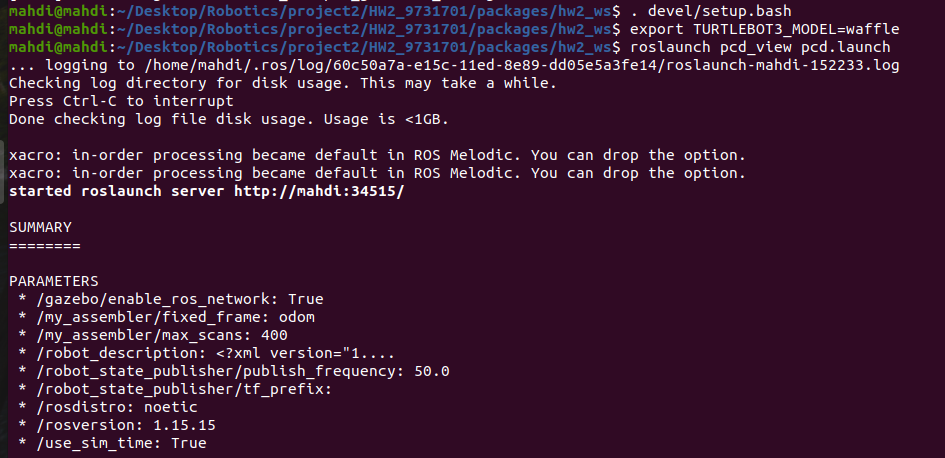
    </node>

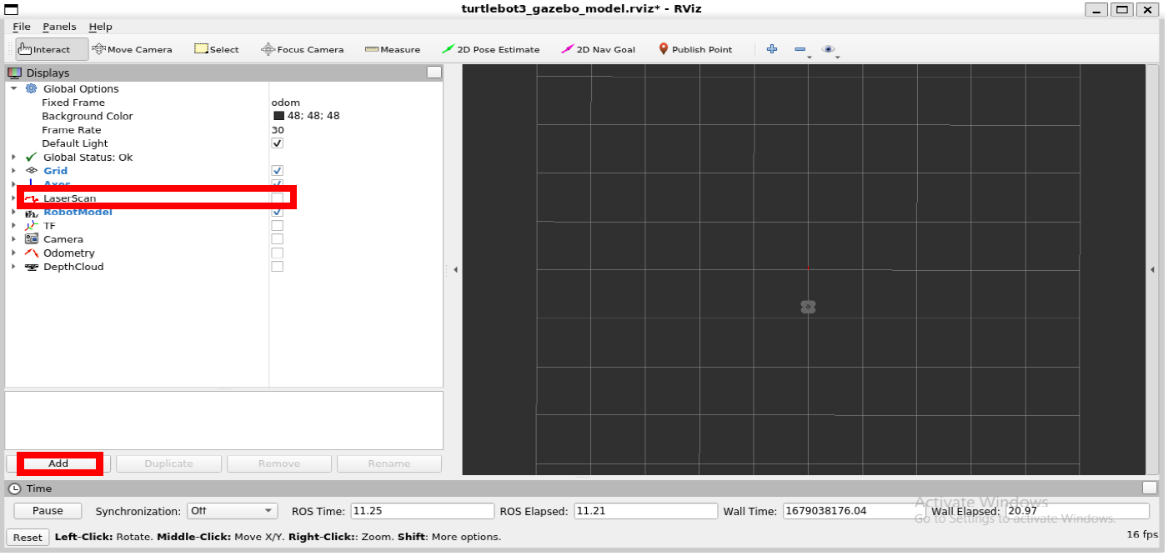
    <node pkg="pcd\_view" type ="laser\_to\_pcd.py" name="laser\_to\_pcd" output="screen"/>

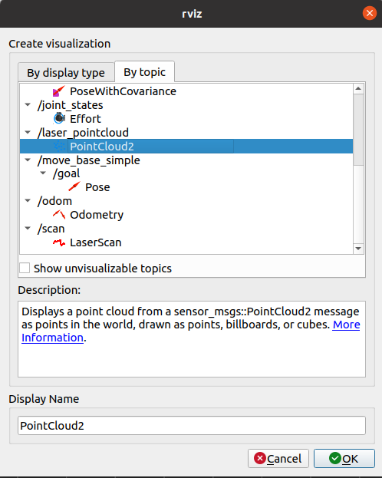
</launch>

حال ابتدا لازم است که فایل پایتون مربوطه را executable کنیم:

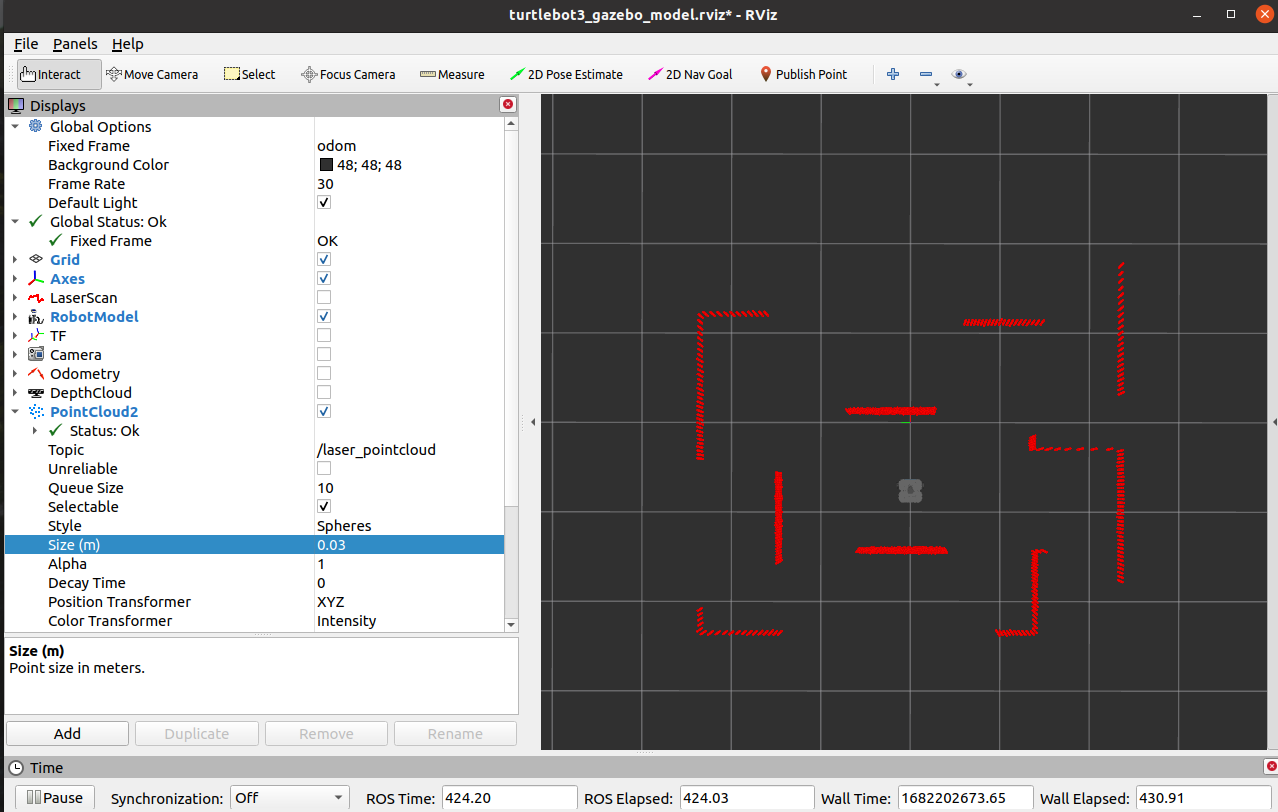
سپس در work space رفته و catkin\_make را میزنیم. سپس لازم است تا در این ترمینال سورس کنیم و ربات را معرفی نماییم و سپس لانچ فایل را اجرا نماییم:



در RViz از منوی سمت چپ با Uncheck کردن LaserScan ، آن را Disable نمایید. سپس گزینه ی Add را انتخاب کنید.

در منوی باز شده، در بخش By Topic بر روی PointCloud2 کلیک نمایید.

پس از این کار RViz داده های دریافتی از تاپیک مربوط به PointCloud2 را نمایش می دهد.



اکنون در یک ترمینال دیگر turtlebot3\_teleop\_key.launch را از پکج turtlebot3\_teleop ران کنید و ربات را با کیبورد کنترل نمایید. با حرکت ربات می توانید نقشه ای از محیط بسازید و کامل نمایید.

A screenshot of a computer

Description automatically generated with medium confidenceالبته چون کمی کنترل ربات سخت بود و کار نقشه برداری به کندی انجام میشد مجبور شدم مقدار max\_scans را در لانچ فایل افزایش دهم و برابر 2000 قرار دهم.

# گام سوم – نتایج

**بخش اول:**

1. **تصویری از کارکرد ربات:**

همانطور که در شکل زیر مشاهده میکنید تصویری از کارکرد ربات را نشان میدهد که در آن به کمک teleop ربات را جا به جا کردیم و خروجی اسکن لیرز را در rviz نشان میدهد.



1. **دلیل لرزش نقاط داده در شبیه ساز چیست؟ دلیل خود را در گزارش تمرین بیاورید.**

یکی از علت های آن که در کلاس هم توضیح داده شد، خطای موجود در اسکن کردن محیط میباشد. برای مثال زمانی که یک تابش لیزر به یک نقطه از محیط میخورد و ما فاصله آن را پیدا میکنیم و متناظر با آن در Rviz یک نقطه نشان میدهیم، لزومی ندارد که اگر مجدد پرتو لیزر به همان نقطه بخورد باز در Rviz دقیقا همان نقطه را نشان دهیم و ممکن است کمی جلوتر یا عقب تر نشان دهیم که به دلیل نرخ بالای اسکن کردن از محیط به نظر میرسه که نفاط داده در حال لرزش هستند. پس در حقیقت نویزی که در سنسور وجود دارد میتواند باعث این موضوع شود.

حال اگر ربات درحال حرکت باشد و سنسورهم همراه با آن در حال حرکت هست و این باعث میشود این اثر لرزش کمی بیشتر هم بشود.

نکته دیگر بحث پردازش این دیتاها در ROS میباشد که ممکن است در هنگام تبدیل نقاط به فریم لوکال لیزر اسکنر، خطایی اندک وجود داشته باشد و حتی بحث های پردازش سخت افزاری و زمان پراسس موجب پدیده jittering شود.

همچنین برخی نقاط ممکن است بازتابشان در لحظاتی به درستی به سنسور نرسد و آن نقطه متناظر به درستی در یک لحظه تشخیص داده نشود و در لحظه بعد مجدد در rviz نمایش داده شود که این هم خب باز میتواند تاثیر گذار باشد**.**

در این [لینک](https://answers.ros.org/question/338112/vibrating-point-cloud-in-rviz/) نیز در این رابطه توضیحاتی دادند.

**بخش دوم:**

1. **ویدیوی کارکرد ربات :** ویدیو ضبط شده و در فولدر مربوطه قرار داده شد.
2. **تصویر نهایی ربات:**

A picture containing text, red, clock, black

Description automatically generatedدرزیر میتوانید تصویر نهایی از نقشه برداری را مشاهده کنید. ( چون کمی کنترل ربات سخت بود و کار نقشه برداری به کندی انجام میشد مجبور شدم مقدار max\_scans را در لانچ فایل افزایش دهم و برابر 2000 قرار دهم که تعداد اسکن های بیشتری ذخیره شود و تا رسیدن به پایان کار قبلی ها پاک نشوند.)