گزارش کار پروژه‌ی اصول علم ربات

نویسنده:

شکیبا امیرشاهی ۹۷۳۱۰۷۴

دلارام رجایی ۹۷۳۱۰۸۴

استاد راهنما:

دکتر جوانمردی

تاریخ:

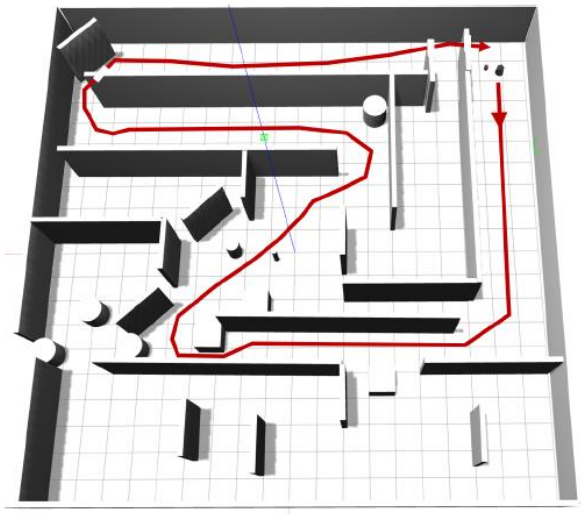
تیر ۱۴۰۰

فهرست

مقدمه

در این پروژه سعی داریم تا رباتی را در محیطی خاص حرکت دهیم. محیط از قبل تعیین شده است و به صورت فایل آمده به اسم funky-maze.world آماده وجود دارد. این فایل شامل توضیحاتی از موانع است که در کنار هم شکلی که در پایین می‌بینیم را تشکیل می‌دهد. ربات باید در این محیط به مدت ۱۵ دقیقه حرکت کند و به جایگاه اولیه خود یا origin برگردد. مسیریابی ربات باید دو ویژگی داشته باشد:

۱. امن باشد: مسیریابی ربات نباید هیچ برخوردی با موانع داشته باشد.
۲. بهینه باشد: مسیریابی ربات، باید در مسیر مد نظر که در شکل زیر نشان داده است حرکت کند و از مسیر خارج نشود.



ربات تنها دو حرکت می‌تواند داشته باشد:

۱. حرکت مستقیم
۲. چرخش

اصطلاحا به ربات reflex یا reactive controller گفته می‌شود چون ربات ما با توجه به اطلاعاتی که از سنسورها دریافت می‌کند، یکی از حرکات بالا را انتخاب کرده و عملی را انجام می‌دهد.

ربات دو سنسور دارد که در جهت‌یابی به آن کمک می‌کند:

۱. لیزر: به وسیله‌ی لیزر، فاصله تا موانع را تشخیص می‌دهد و اندازه گیری می‌کند.
۲. bumper: سنسوری است که تعداد دفعاتی که ربات به مانعی برخورد کند را می‌شمارد و تشخیص می‌دهد.

ربات ما نیمه‌مشاهده‌پذیر است؛ به این معنی که دنیای اطراف خود را به طور کامل نمی‌شناسد و به صورت محلی جست و جو می‌کند و تصمیم می‌گیرد و تنها محدوده‌ی اطراف خود را مشاهده می‌کند.

پیاده‌سازی

۴ فایل پایتون پیاده‌سازی شده است که در ادامه هر کدام به صورت جداگانه توضیح داده می‌شود:

Odom2.py

این node در هر مرحله موقعیت ربات را اندازه‌گیری می‌کند و بدست می‌آورد و سپس به node دیگر، جهت اندازه‌گیری فاصله‌ی آن با موانع می‌فرستد. یک متغیر coke-count هم وجود دارد که تعداد دفعاتی که ربات از موقعیت اولیه خود عبور می‌کند را می‌شمارد.
کد به صورت زیر است:

```
import rospy
from nav_msgs.msg import Odometry
from nav_msgs.msg import Path
from geometry_msgs.msg import PoseStamped
import math
path = Path()
global coke_count
coke_count=0
def callback(msg):
    global path
    global coke_count
    print (msg.pose.pose)
    if(msg.pose.pose.position.x==0  and msg.pose.pose.position.y==0 and
msg.pose.pose.position.z==0 ):
       coke_count+=1
    print("coke count is {}".format( coke_count))


if __name__ == '__main__':
    rospy.init_node('check_odometry')
    odom_sub = rospy.Subscriber('/odom' ,Odometry,callback)
    path_pub = rospy.Publisher('/path', Path, queue_size=10)
     rospy.spin()
```

Bumper.py

در اینجا در صورت برخورد با مانع تشخیص می‌دهد و flag را برابر True قرار می‌دهد سپس اطلاعات خود را به node دیگری منتشر می‌کند.

کد آن به صورت زیر است:

```
from kobuki_msgs.msg import BumperEvent
import rospy
from std_msgs.msg import String
def process_Bumper(data):
    global bump
    bumper=""
    if (data.state == BumperEvent.PRESSED):
        bump = True
        bumper="True"
    else:
        bump = False
        bumper="False"
    message_publisher.publish(bumper)
    rospy.loginfo("Bumper Event")
    rospy.loginfo(data.bumper)
#END MEASUREMENT
rospy.init_node('bumper',anonymous=True)
rospy.Subscriber('/bumper', BumperEvent, process_Bumper)
message_publisher = rospy.Publisher("bumper_topic", String, queue_size=10)
rospy.spin()
```

obstacle-detector.py

با توجه به اطلاعاتی که از دو سنسور قبل می‌گیرد. فاصله‌ی خود تا موانع را اندازه‌گیری می‌کند و در آرایه‌ای با اندازه‌ی n به نام range ذخیره می‌کند. که در این آرایه خانه‌ی ۰ نشان دهنده‌ی راست ربات – فاصله تا مانع راست، n نشان دهنده‌ی چپ ربات – فاصله تا مانع چپ ربات و n/۲ نشان دهنده‌ی رو به رو است. اگر هم هیچ مانعی قرار نداشته باشد، عددی ذخیره نمی‌شود. بعد از محاسابه‌ی این مقادیر آنها در قالب یک string به node بعدی منتقل می‌کند.

کد به صورت زیر است:

```
#!/usr/bin/env python
import rospy
from sensor_msgs.msg import LaserScan
from std_msgs.msg import String
import math
# BEGIN MEASUREMENT
def scan_callback(msg):
    print("hello")
    string=""
    range_center = msg.ranges[math.floor(len(msg.ranges)/2)]
    range_left = msg.ranges[len(msg.ranges)-1]
    range_right = msg.ranges[0]
    string=str(range_center)+"#"+str(range_left)+"#"+str(range_right)
    print(string)
    message_publisher.publish(string)
#END MEASUREMENT
rospy.init_node('range_ahead',anonymous=True)
scan_sub = rospy.Subscriber('/scan', LaserScan, scan_callback)
message_publisher = rospy.Publisher("messageTopic", String, queue_size=10)
rospy.spin()
```

velocity-controller.py

در این node با توجه به فاصله‌های اندازه‌گیری شده از node قبل، یکی از حرکات ربات، چرخش یا حرکت به صورت مستقیم، را انتخاب می‌کند. در اینجا سرعت را تعیین می‌کنیم. بعد از چند بار اجرا با سرعت‌های متفاوت عدد مناسبی را بدست می‌آوریم. ربات تا زمانی که مانعی را در جلوی خود نبیند به صورت مستقیم حرکت می‌کند و اگر مانعی در هر کدام از جهت های چپ ، راست یا روبرو مشاهده کند اقدام به چرخش میکند. دو آرایه‌ی دیگر نیز در این تابع وجود دارد؛ x-vel و z-vel که سرعت‌های ربات را در خود ذخیره می‌کنند، هنگام تخمین زدن میانگین سرعت، به صورت راندوم یکی از این عددها را انتخاب می‌کنیم زیرا عددی که بیشتر ظاهر شده باشد احتمال انتخاب آن نیز بیشتر خواهد بود بنابراین به این صورت می‌توان سرعت را تخمین زد.
هر مرحله از کد نیز شامل یک سری if وelse است که با توجه با وجود یا عدم وجود مانع در چپ ،راست و روبروی ربات سرعت خطی یا زاویه ای را تعیین میکنیم.
کد به صورت زیر است:

```
from std_msgs.msg import String
from geometry_msgs.msg import Twist
global BUMPERCOUNT
BUMPERCOUNT=0
global x_vel
global z_vel
x_vel=[]
z_vel=[]
def callback_str(subscribedData):
     global x_vel
     global z_vel
     rospy.loginfo('Subscribed: ' + subscribedData.data
     string=str(subscribedData.data)
     ranges=string.split("#")
     range_center=float(ranges[0])
     range_left=float(ranges[1])
     range_right=float(ranges[2])
     obstacle_description =""
     if range_center > 1 and range_left > 1 and range_right > 1:
        obstacle_description = 'case 1 - nothing'
        linear_x = 0.6
        angular_z = 0
        x_vel.append(linear_x )
        z_vel.append(angular_z )
     elif range_center < 1 and range_left > 1 and range_right > 1:
        obstacle_description = 'case 2 - front'
        linear_x = 0
        angular_z = -0.3
     elif range_center > 1 and range_left > 1 and range_right < 1:
        obstacle_description = 'case 3 - right'
        linear_x = 0
        angular_z = -0.3
        x_vel.append(linear_x )
        z_vel.append(angular_z )
     elif range_center > 1 and range_left < 1 and range_right > 1:
        obstacle_description = 'case 4 - left'
        linear_x = 0
```

```
            angular_z = 0.3
            x_vel.append(linear_x )
            z_vel.append(angular_z )
        elif range_center < 1 and range_left > 1 and range_right < 1:
            obstacle_description = 'case 5 - front and right'
            linear_x = 0
            angular_z = -0.3
            x_vel.append(linear_x )
            z_vel.append(angular_z )
        elif range_center < 1 and range_left < 1 and range_right > 1:
            obstacle_description = 'case 6 - front and left'
            linear_x = 0
            angular_z = 0.3
            x_vel.append(linear_x )
            z_vel.append(angular_z )
        elif range_center < 1 and range_left < 1 and range_right < 1:
            obstacle_description = 'case 7 - front and left and right'
            linear_x = 0
            angular_z = -0.3
            x_vel.append(linear_x )
            z_vel.append(angular_z )
        elif range_center > 1 and range_left < 1 and range_right < 1:
            obstacle_description = 'case 8 - left and right'
            linear_x = 0
            angular_z = -0.3
            x_vel.append(linear_x )
            z_vel.append(angular_z )
        rospy.loginfo( obstacle_description)
        vel_msg.linear.x = linear_x
        vel_msg.angular.z = angular_z
        velocity_publisher.publish(vel_msg)
        average_linear=random.choices(x_vel)
        average_angular=random.choices(z_vel)
        print("average Linear Velocity {} ".format(average_linear))
        print("average angular Velocity {} ".format(average_angular))
def callback_bumper(subscribedData):
        global BUMPERCOUNT
        rospy.loginfo('Subscribed: ' + subscribedData.data
        string=str(subscribedData.data)
        if string=="True":
            BUMPERCOUNT+=1
            print(BUMPERCOUNT)
def messageSubscriber():
    #initialize the subscriber node called 'messageSubNode'
    rospy.init_node('velocity_controller', anonymous=False)
    rospy.Subscriber('messageTopic', String, callback_str)
    rospy.Subscriber('bumper_topic', String, callback_bumper
    rospy.spin()
if __name__ == '__main__':
    try:
        velocity_publisher = rospy.Publisher('/cmd_vel',Twist, queue_size=10)
        vel_msg =Twist()
        messageSubscriber()
    except rospy.ROSInterruptException:
        pass
```

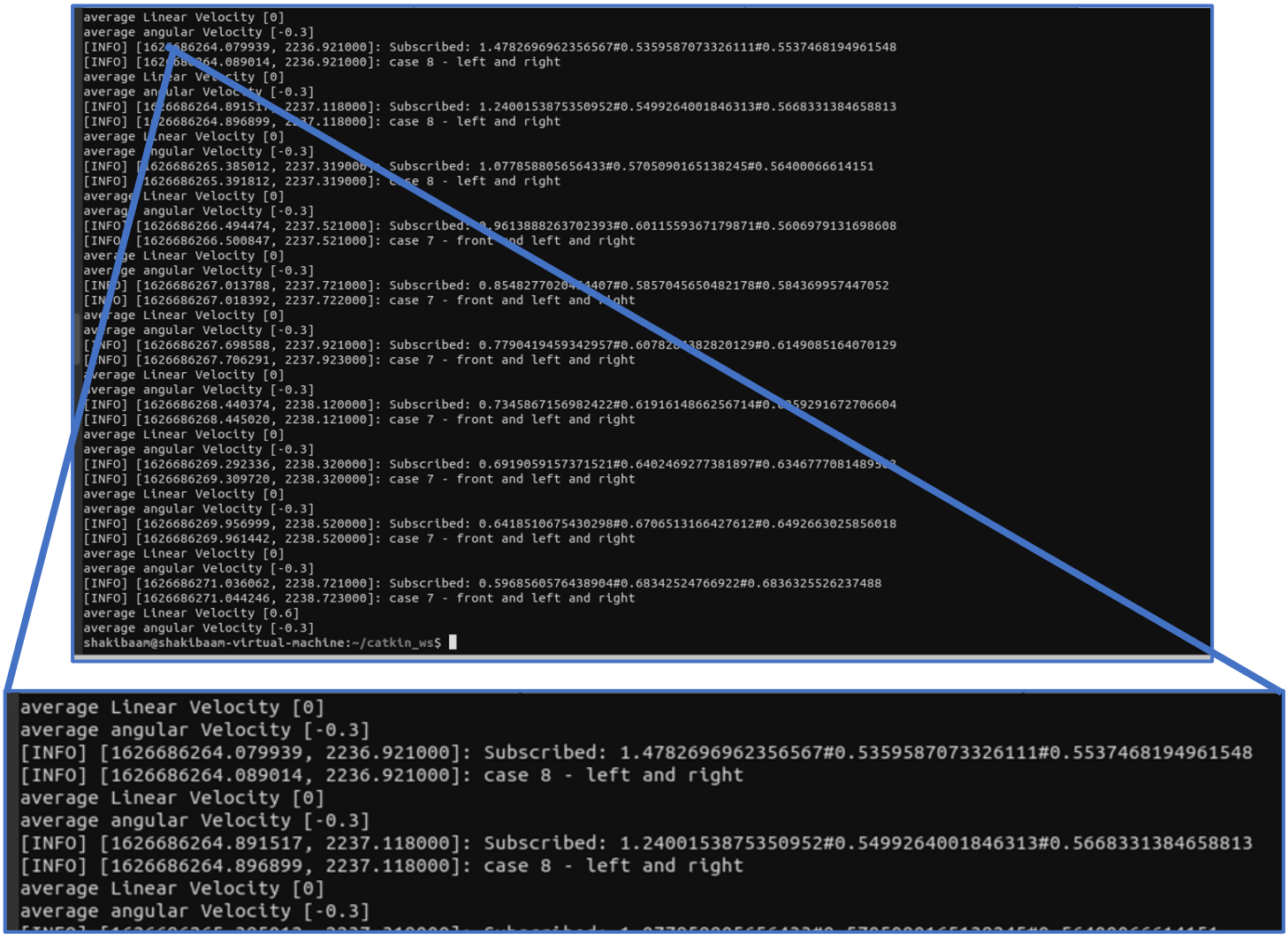
سوالات

برای اینکه مطمئن شویم آیا دو تا خواسته‌ی اصلی مسئله یعنی امن و بهینه بودن را رعایت می‌کند به تعدادی از سوالات پاسخ می‌دهیم.

تعداد دفعاتی که ربات از نقطه‌ی اول می‌گذرد و تخمینی از سرعت میانگین ربات؟

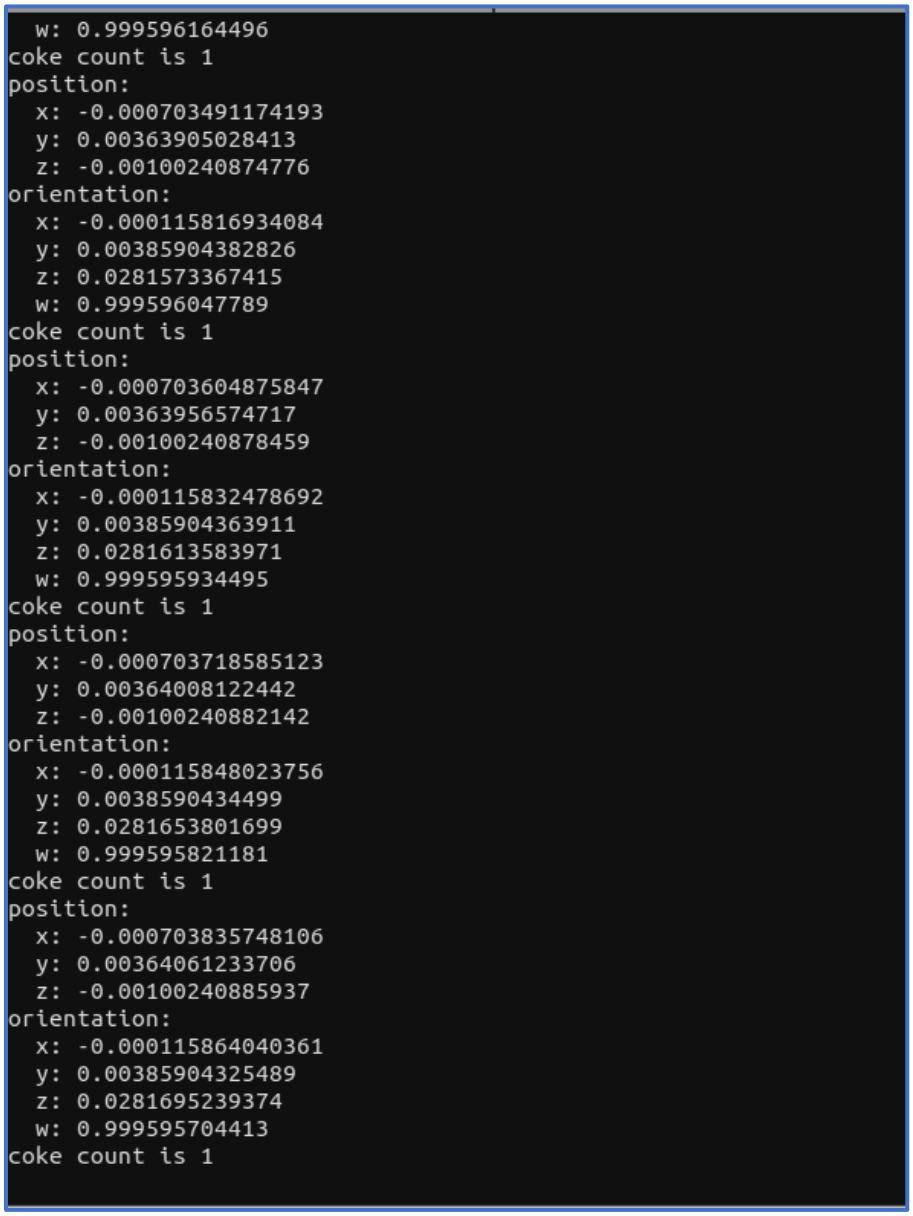
در تصویر اول، میانگین سرعت تخمینی ربات را نشان میدهد که با average linear و average angular مشخص شده است.

هم چنین فاصله تا موانع راست ، چپ و روبرو هم در قالب یک string نشان داده میشود که با # از هم جدا شده اند.



average Linear Velocity [0]
average angular Velocity [-0.3]
[INFO] [1626686264.079939, 2236.921000]: Subscribed: 1.4782696962356567#0.5359587073326111#0.5537468194961548
[INFO] [1626686264.089014, 2236.921000]: case 8 - left and right
average Linear Velocity [0]
average angular Velocity [-0.3]
[INFO] [1626686264.891517, 2237.118000]: Subscribed: 1.2400153875350952#0.5499264001846313#0.5668331384658813
[INFO] [1626686264.896899, 2237.118000]: case 8 - left and right
average Linear Velocity [0]
average angular Velocity [-0.3]

در تصویر دوم نیز موقعیت ربات در هر مرحله و اینکه چند بار تا این لحظه از مبدا عبور کرده است را نشان میدهد.



تعداد دفعاتی که ربات به مانعی برخورد می‌کند؟

عملکرد کد در بالا توضیح داده شده است. ربات turtlebot۳ سنسور bumper را ندارد. در نتیجه عملیات را نمی‌توان به صورت کامل درستی کد را چک کرد.

آیا عملکرد ربات رضایتمندانه است؟

همانطور که از اعداد بدست آمده متوجه می‌شویم، تعداد دفعات برخورد ربات با مانع و تعداد دفعاتی که از نقطه‌ی اولیه می‌گذرد، می‌توان تا حدودی عملکرد ربات را سنجید. در اینجا می‌توان گفت که عملکرد ربات رضایت‌مندانه است، ربات در مسیری که به آن می‌دهیم تا حد خوبی حرکت می‌کند، بهنیه است و امن است و کمترین تعداد برخورد را با موانع دارد زیرا همونطور که بالاتر گفته شد در یک سری if وelse تمام اطراف ربات چک میشود تا حرکت طوری تعیین شود که کمترین برخورد با موانع اتفاق بیفتد. اما همچنان با بالا بردن دقت controller می‌توان عملکرد ربات را بهبود بخشید.

مشکلاتی که در پیاده‌سازی controller وجود داشتند به چه صورت بودند؟

یکی از مشکلات در محاسبه‌ی فاصله‌ی ربات تا مانع بود، تا بتواند دقت کافی را داشته باشد و سپس بر اساس آن حرکت ربات را انتخاب کنیم. یکی دیگر از مشکلات چرخش ربات با سرعت مناسب بود. ربات باید در جهت خاصی برای حرکت قرار بگیرد. چرخش ربات با تغییر سرعت ربات و زیاد شدن سرعت تغییر می‌کند. پیدا کردن سرعت مناسب یکی دیگر از مشکلات موجود بوده است.

Controllerپیاده سازی شده چه مشکلاتی داشت؟

همانطور که در سوال قبل نیز اشاره شد یکی از مشکلات این controller دقیق نبودن سرعت چرخش است ، هر چند با آزمون و خطا سعی شد سرعت مناسب به دست بیاید اما همچنان این سرعت دقیق نیست و در برخی موارد هنگام چرخش باعث برخورد به مانع میشود.

همچنین با اینکه سعی شد با if , else های فراوان تمامی شرایط ممکن لحاظ شود تا کمترین برخورد صورت گیر اما ممکن است حالت خاصی در این شروط قرار نگرفته باشد و باعث برخورد شود.

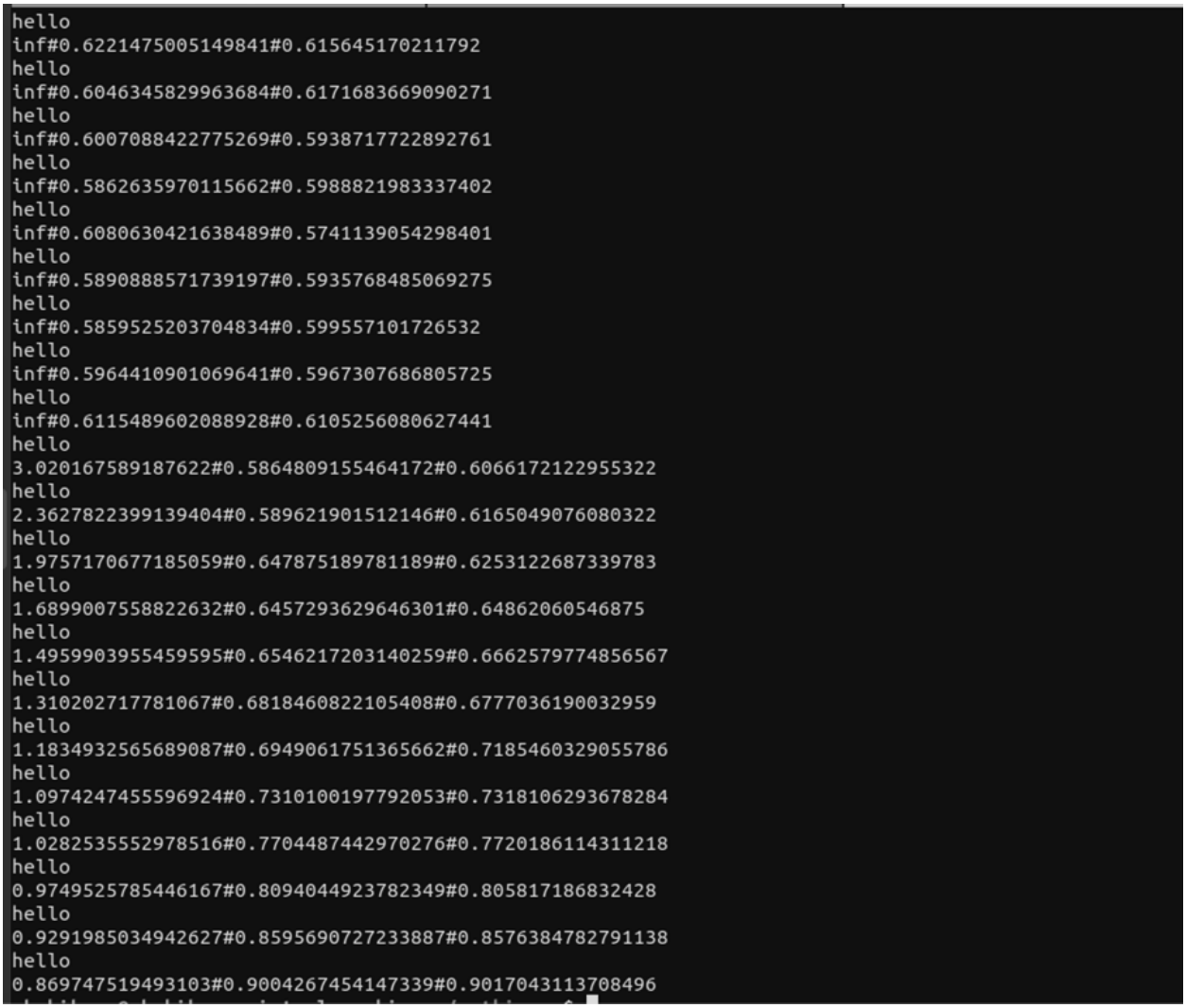
پیشنهاد شما برای بهتر کردن مسیریابی ربات چیست؟

داشتن حافظه اضافه می‌تواند کمک کند تا ربات مسیری را که طی کرده است به خاطر بسپارد؛ با چندین بار طی کردن مسیر می‌تواند بهترین مسیر را نگهداری کند و هر دفعه به محاسبه‌ی موانع نپردازد. البته این در صورتی است که هزینه‌ی اضافی مشکلی نباشد و محیط پویا نباشد و تغییر نکند.

پیشنهاد دیگر این است اگر امکان اضافه کردن حرکت دیگری داشته باشیم، ربات بتواند به عقب حرکت کند در این صورت نیاز به دو بار چرخیدن نیست و پردازش کاهش پیدا می‌کند.

نمونه‌ای از اجرای کد

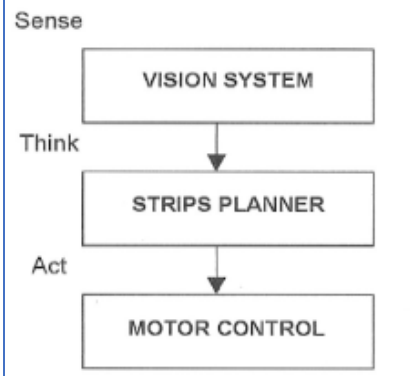
فاصله‌ی ربات تا موانع راست، چپ و جلو که با # از یکدیگر جدا شده‌اند.

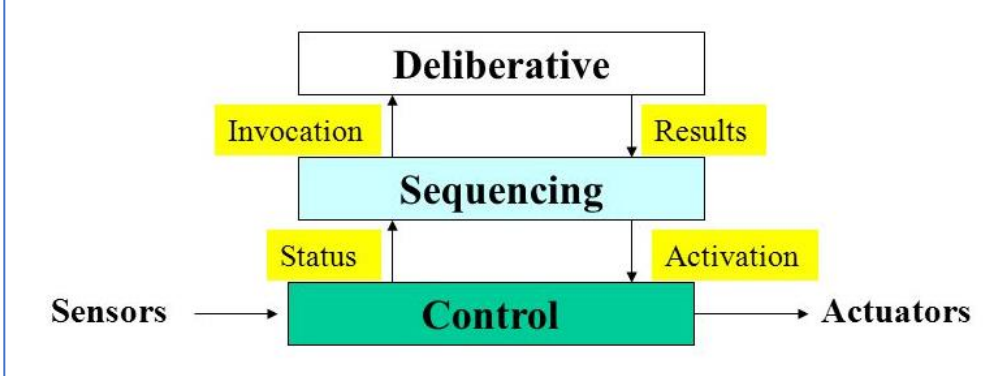


قسمت امتیازی

ربات در اینجا باید کل مکان و location را طی کند تا نقشه‌ی مورد نظر را بدست آورد. با کمک دو سنسوری که در مراحل قبل توضیح داده شد، سنسور bumper و لیزر، می‌توان در مسیری که تا حد خوبی امن و بهنیه است حرکت کند با بدست آوردن فاصله تا اشیاء آنها را به صورت نقشه در حافظه نگهداری کند و به این ترتیب کل نقشه‌ی محلی که در آن قرار دارد را بدست آورد. سپس ربات ما مشاهده‌پذیر خواهد بود و می‌توان به کمک الگوریتم‌هایی مانند BFS و DFS مسیری بهینه تا مقصد را پیدا کند و به سمت آن حرکت کند. اما این راه تا زمانی پاسخ نیاز ما را رفع می‌کند که محیط پویا نباشد و تغییر نکند زیرا اگر تغییر کند، ربات نمی‌تواند نقشه‌ی مفیدی را بدست آورد.Deliberative controller به این صورت عمل می‌کند که،"Think hard, then act" . و مراحل آن به صورت زیر است:

- درست کردن نقشه - Making maps
- انتخاب رفتار درست – Selecting behaviors
- کنترل عملکرد – Monitor performance
- برنامه‌ریزی – Planning





همانطور که پیش‌تر گفتیم ربات تمام اطلاعاتی که سنسورها می‌گیرد را به همراه اطلاعات اولیه‌ای که خودش دارد در کنار هم قرار می‌دهد و بر روی آنها به اصطلاح تفکر انجام می‌دهد تا به نتیجه‌ای برسد و نقشه‌ای درست کند.

برای اینکه ربات بتواند چنین کاری را انجام دهد باید تمام اتفاقاتی که ممکن است بیافتد را در نظر بگیرد و در نهایت آن مسیری که به هدف می‌رسد را انتخاب کند.

این کار ممکن است زمانبر باشد به همین دلیل است که برای کارهایی که باید عملکرد سریعی داشت این ربات مناسب نیست. اما اگر زمان مشکل نباشد این ربات می‌تواند به خوبی عمل کند.

deliberative-robot.py

این node از سه قسمت اصلی تشکیل شده است. sensing،path planning و acting.

در تابع sensing ما ابتدا اطلاعات مورد نیاز خود را از node، velocity-controller می‌گیریم. اطلاعاتی شامل range و bumper. اینکار را آنقدر تکرار می‌کنیم تا ربات به جایگاه اولیه‌ی خود برگردد به این معنی که یکبار حداقل دنیای اطراف خود را طی کرده باشد. مقادیر ذخیره شده در range به این صورت است که می‌توان position را به آن داد و موانع اطراف آن را بدست آورد.

```
    def callback(self, msg):
        self.position["x"] = msg.pose.pose.position.x
        self.position["y"] = msg.pose.pose.position.y
        self.position["z"] = msg.pose.pose.position.z
        if msg.pose.pose.position.x == 0 and msg.pose.pose.position.y == 0 and
msg.pose.pose.position.z == 0:
            self.count += 1
    def callback_message(self, subscribedData):
        rospy.loginfo('Subscribed: ' + subscribedData.data)
        string = str(subscribedData.data)
        odom_sub = rospy.Subscriber('/odom', Odometry, self.callback)
        # data = bumper-ranges
        str_position = self.position["x"], self.position["y"], self.position["z"]
        data = string.split("-")
        self.bumper[str_position] = data[0]
        # data[1:4] = right#center#left
        range = data.split("#")
        self.ranges[str_position] = range[1:4]
    def sensing(self):
        while self.count != 2:
            rospy.init_node('deliberative_controller', anonymous=False)
            rospy.Subscriber("velocity_topic", String, self.callback_message)
```

بعد از اینکه اطلاعات را بدست آوریم و تقریبا نقشه‌ی اولیه مورد نیاز را تولید کردیم. به سراغ path planning می‌رویم. در این تابع به کمک یکی از الگوریتم‌هایی که می‌شناسیم مانند A^* مسیری از نقطه‌ی اولیه‌ی ربات تا

مقصد بدست می‌آوریم. در این مسیر باید موانع را در نظر بگیریم و سعی کنیم در عین حال کوتاهترین مسیر و بهینه‌ترین (کم هزینه‌ترین) مسیر را برای رسیدن به هدف انتخاب کنیم.

```
def get_end_position(self):
    return input("Goal position: ")
def planing(self):
    astar = Astar()
    end = self.get_end_position()
    start = (0, 0, 0)
    self.path = astar.astar_search(self.ranges, start, end)
```

بعد از پیدا کردن بهترین مسیر به سراغ acting می‌رویم. در اینجا با توجه به خروجی قسمت قبل ربات حرکت می‌کند تا به مقصد برسد.

```
def acting(self):
    x_vel = []
    z_vel = []
    for i in range(len(self.path)):
        x_vel.append(self.path("linearX"))
        z_vel.append(self.path("angularZ"))
        self.deliberative_msg.linear.x = self.path("linearX")
        self.deliberative_msg.angular.z = self.path("angularZ")
        self.deliberative_publisher.publish(self.deliberative_msg)
```

تابع main نیز به صورت زیر است:

```
def main(self):
    try:
        self.deliberative_publisher = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
        self.deliberative_msg = Twist()
        # Sense
        self.sensing()
        # Plan
        self.planing()
        # Act
        self.acting()
    except rospy.ROSInterruptException:
        pass
```

مراجع

لینک مراجعی که در این پروژه مورد استفاده قرار گرفتند به شرح زیر است:

[1] https://web.fe.up.pt/~lpreis/ir2007_08/documents/IR0708-3-AgentArchitectures.pdf

[2] https://www.annytab.com/a-star-search-algorithm-in-python/

[3] https://github.com/enansakib/obstacle-avoidance-turtlebot/blob/master/src/naive_obs_avoid_tb3.py

[4] https://github.com/aditya-167/ROS-Turtlebot-maze/tree/master/src/src

[5] https://canvas.harvard.edu/courses/37276/pages/getting-started-2-ros-turtlebot-sensors-and-code

[6] https://www.youtube.com/watch?v=q3Dn5U3cSWk