دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

پروژه نهایی درس مبانی علم ربات

استاد

دکتر جوانمردی

نگارش

محمدجواد رجبی

علیرضا کریمی

سناریو اول – گام اول

یک پکیج با نام Final_project_slam ایجاد می‌کنیم و پروژه را در این پکیج انجام می‌دهیم.

دو پکیج زیر را نصب می‌کنیم که یکی برای ساخت map از محیط است و دیگری برای ذخیره کردن map به دست آمده:

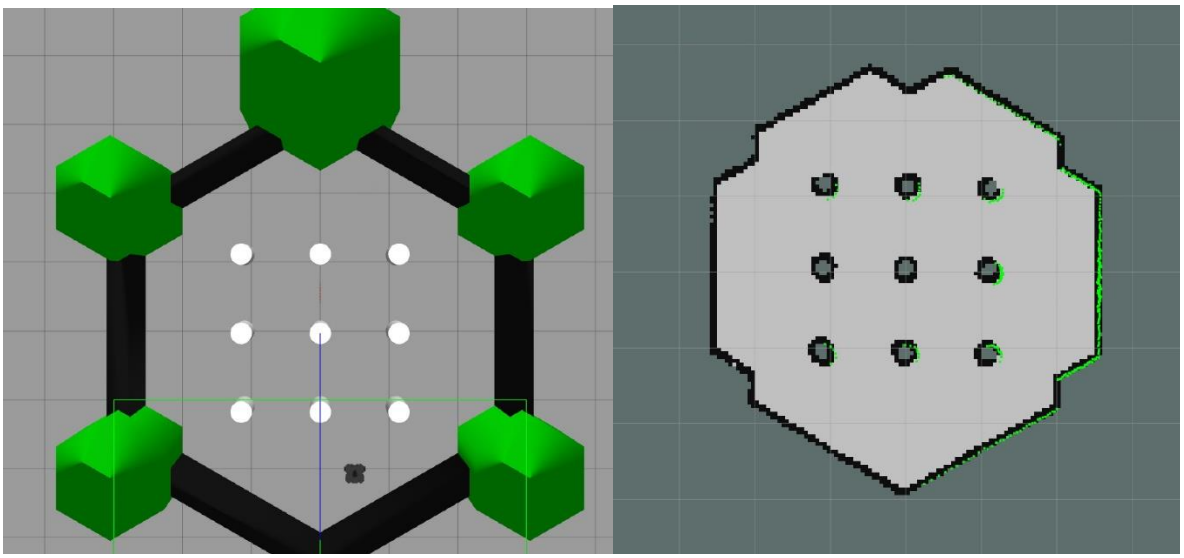
```
sudo apt install ros-noetic-slam-gmapping
sudo apt install ros-noetic-slam-map-server
```

برای اجرای RVIZ و رسم نقشه در آن دستور زیر را پس از اجرای گزبو وارد می‌کنیم:

```
roslaunch turtlebot3_slam turtlebot3_slam.launch
```

برای کامل شدن نقشه نیاز است که با teleop و دستورات کیبورد ربات را حرکت دهیم تا کل محیط را ببینند.

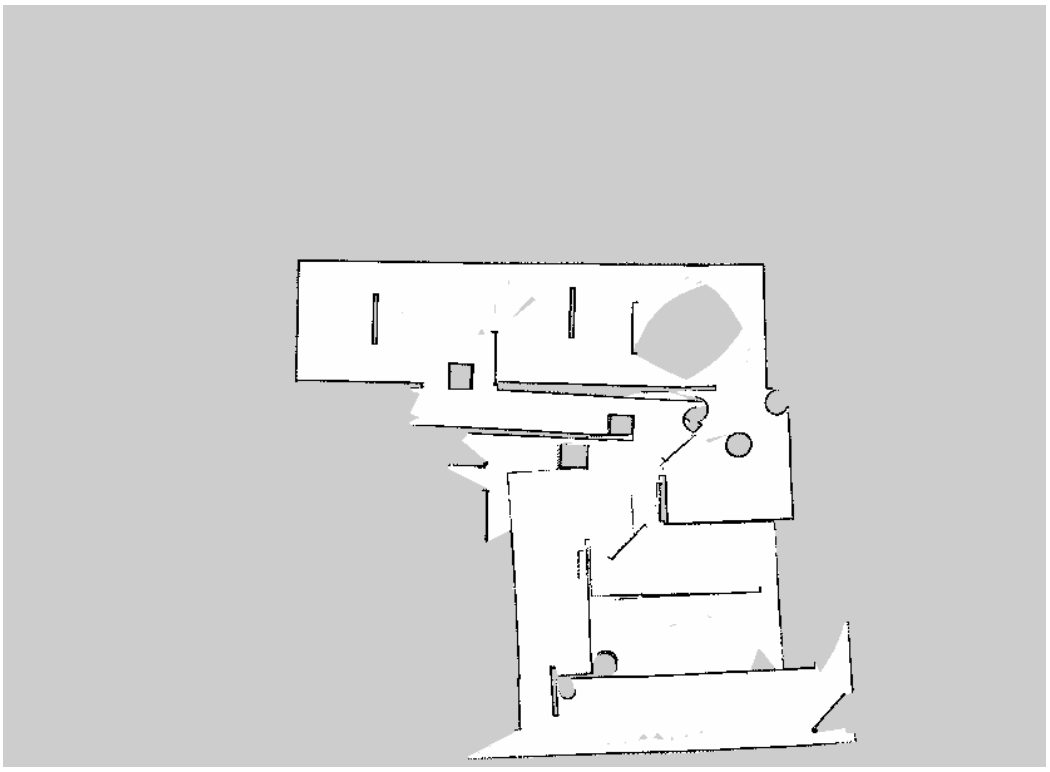
با اجرای دستور rosrun map_server map_saver -f ~/map نقشه ذخیره می‌شود.

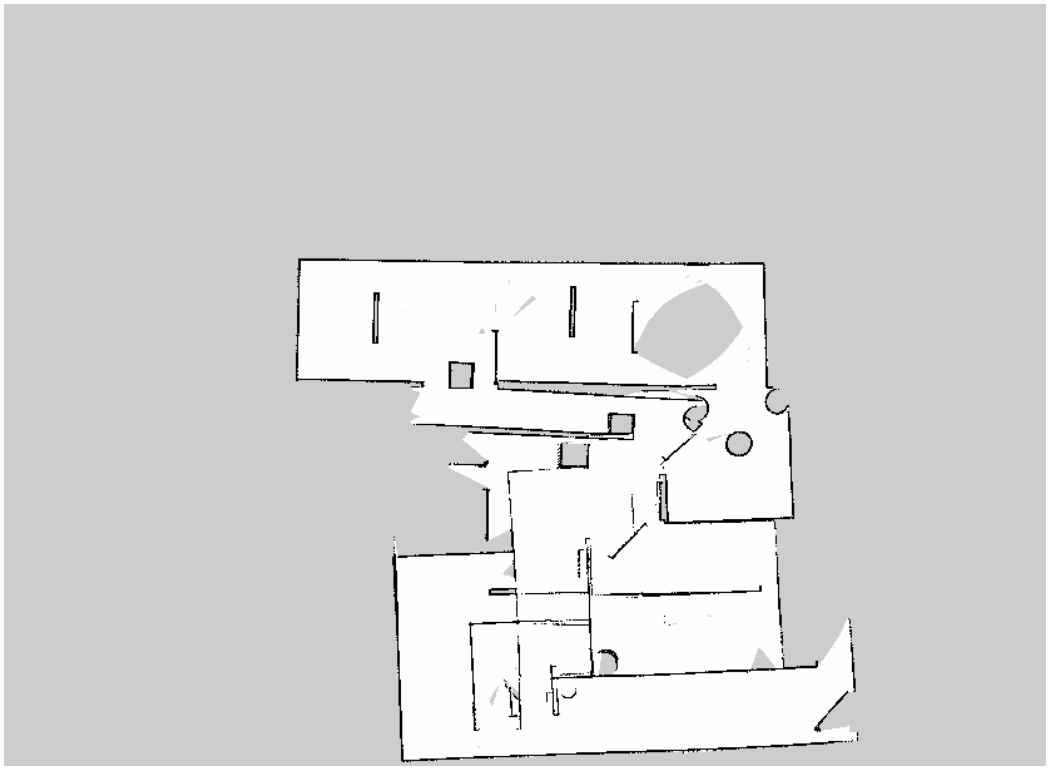


سناریو اول – گام دوم

در این مرحله بایستی با استفاده از یک کنترلر کاری کنیم که ربات به طور خودکار محیط را Explore کند.

خروجی این مرحله در نهایت نقشه‌های زیر شد (در دو اجرای مختلف):





انجام اکتشاف در محیط توسط کدی که در فایل wall_follow قرار دارد انجام می‌شود. این فایل را در ادامه توضیح می‌دهیم. اما نکته‌ای که قابل ذکر است آن است که با اینکه ربات محیط را به خوبی اکتشاف می‌کند، حتی با چندین و چند بار اجرا گرفتن و حتی با زمان‌های مختلف exploration، نقشه خروجی slam خطاهایی دارد.

این خطاها احتمالا به این علت رخ می‌دهد که slam از odometry استفاده می‌کند و به مرور زمان خطا زیاد می‌شود.

توضیح الگوریتم Exploration:

این کد در حقیقت یک الگوریتم wall following است که مشابه آن را در تمرین‌های قبلی داشتیم. یک PID Controller داریم که خطای آن فاصله تا دیوار است و سعی داریم این فاصله را کم نگه داریم. بنابراین همواره دیواری را در جهان دنبال می‌کند و به این ترتیب کل جهان را explore می‌کند. از آن جایی که الگوریتم wall following در حالت معمولا ممکن است در لوپ گیر کند و در همه محیط نرود، با اعمال یک عامل رندوم سعی کردیم در 10 درصد مواقع به جای اعمال زاویه با PID زاویه را صفر دهیم تا ربات مسیر قبلی را ادامه دهد و حرکتی نکند.

یک الگوریتم دیگری که میتوان برای این قسمت استفاده کرد آن است از VFH Controller قسمت بعدی استفاده کنیم با این تفاوت که به جای یک goal، به آن مجموعه‌ای از goalها را دهیم که در جاهای مختلفی مختلفی از صفحه قرار دارند.

سناریو دوم

در این سناریو الگوریتم VFH را مطابق توضیحات پیاده‌سازی کردیم. نکات مهم در پیاده‌سازی ما عبارتند از:

- ابتدا و در تعریف کلا متغیرهای گفته شده در دستور کار را به عنوان اتریبیوت ست می‌کنیم. مثلا a و b یا محل هدف یا threshold در اینجا تعریف می‌شوند.

```
class VFH_Controller():

    def __init__(self):

        rospy.init_node('vfh', anonymous=False)

        self.target_x = -8
        self.target_y = -8

        self.angular_speed = 0.1
        self.angular_epsilon = 0.1
        self.linear_spead = 0.1
        self.linear_epsilon = 0.05
        self.linear_lenght = 1

        self.sector_size = 5
        self.a = 1
        self.b = 0.25
        self.threshold = 2.2
        self.s_max = 9
```

لازم به ذکر است که linear_length طول مسیری است که ربات طی می‌کند تا مجددا الگوریتم VFH مسیریابی را دوباره انجام دهد و بهترین زاویه حرکت به دست آید. همچنین sector_size همان اندازه هر سکتور است که طبق پیشنهاد دستور کار 5 درجه در نظر گرفته شده است.

- با اجرای کنترلر VFH، تابع global_path_planning مادامی که برنامه در حال اجراست اجرا می‌شود.

```
def global_path_planning(self):

    sectors = self.calculate_Histogram()

    target_sector = self.find_target_sector()
    selected_sectors = self.thresholding(sectors)

    if sectors[target_sector] < self.threshold:
        best_sector = target_sector
    else:
        best_sector = self.select_valley(selected_sectors, target_sector)

    if best_sector > 36:
        best_sector -= 72

    # print(best_sector)
    angle = math.radians(best_sector * 5)

    self.controller(angle)
```

در این تابع ابتدا با فراخوانی تابع calculate_Histogram هیستوگرام را رسم می‌کنیم و سپس با توجه به محل قرارگیری ربات و نقطه هدف، تابع find_target_sector هم سکتوری که در نهایت به هدف می‌رسد را پیدا می‌کند. تابع thresholding نیز با گرفتن سکتورها دره‌ها را به دست می‌آورد. پس از انجام کارهای ذکرشده بررسی می‌کنیم که سکتوری که هدف در راستای آن است، آیا مقدار نرمال شده هیستوگرامش کمتر از threshold قرار می‌گیرد یا نه؛ اگر قرار بگیرد این سکتور به عنوان بهترین سکتور برای حرکت انجام می‌شود. در غیر این صورت تابع select_valley سکتور مناسب را به دست می‌آورد. در نهایت نیز سکتور را به زاویه تبدیل کرده و با استفاده از تابع move_side زاویه را به ربات اعمال می‌کنیم.

در ادامه به صورت مختصر توابع ذکرشده را توضیح می‌دهیم.

- تابع calculate_Histogram:

این تابع با iterate کردن بر روی داده‌های laser_scan، در هر سکتور با استفاده از فرمول

$$m_{i,j} = (c^*_{i,j})^2 \, (a - bd_{i,j})$$

اندازه هیستوگرام آن سکتور را به دست می‌آورد. در این جا از آن جایی که فقط در یک زاویه می‌دانیم در چه فاصله‌ای مانع قرار دارد، c* را همواره 1 در نظر می‌گیریم و d هم می‌شود اندازه آن عنصر آرایه laser_scan.

```
def calculate_Histogram(self):
    histogram = []
    self.get_laser_scan()


    self.number_of_sector = int(len(self.laser_scan.ranges)/self.sector_size)

    for i in range(self.number_of_sector):

        tmp_histogram = 0
        for j in range(i*self.sector_size,(i+1)*self.sector_size):
            magnitude = self.a - self.b * min(6,self.laser_scan.ranges[j])
            tmp_histogram += magnitude

        histogram.append(tmp_histogram)


    return self.smoothing_histogram(histogram)
#     return histogram
```

در نهایت آرایه histogram ساخته‌شده را با تابع smoothing_histogram نرمال می‌کنیم و برمی‌گردانیم. علت استفاده از این تابع آن است که در صورت وجود یک مانع با magnitude زیاد در وسط یک دره، کل دره نادیده گرفته نشود.

- تابع find_target_sector:

این تابع ابتدا بررسی می‌کند که زاویه محل قرارگیری هدف نسبت به زاویه هدینگ ربات چقدر تفاوت دارد و سپس متناسب با این تفاوت سکتور در راستای هدف را به دست می‌آورد.

```
def find_target_sector(self):

    position, yaw = self.get_pose()
    angle = math.atan2(self.target_x - position.y, self.target_y - position.x)

    if angle < 0:
        angle += 2 * math.pi

    dif = angle - yaw

    if dif < 0:
        dif += 2 * math.pi

    target_index = int(math.degrees(dif) / self.sector_size)

    return target_index % self.number_of_sector
```

- تابع thresholding:

این تابع با گرفتن سکتورها و با بررسی مقادیر هیستوگرام سکتورها، دره‌های موجود را به دست می‌آورد.

```
def thresholding(self ,sectors):

    thresholded = []
    for i in range(self.number_of_sector):

        if sectors[i] < self.threshold:
            thresholded.append(i)

    return thresholded
```

- تابع select_valley:

این تابع در مواقعی استفاده می‌شود که سکتور هدف یک دره نیست و مجبوریم از بین دره‌های دیگر بهترین گزینه را انتخاب کنیم. دره مناسب آن دره‌ای است که کمترین زاویه را با زاویه ربات تا هدف داشته باشد. پس با مقایسه ایندکس سکتورهای دره، بهترین دره را برای حرکت انتخاب می‌کنیم.

```
def select_valley(self,selected_sectors,target_sector):

    my_min= 999
    my_index = 0
    # my_index2 = 0


    valleys = self.vallye_clstering(selected_sectors)
    # print(valleys)
    for i in range(len(valleys)):

        for j in range(len(valleys[i])):

            distances = abs(valleys[i][j]  - target_sector)

            if distances > 36:
                distances = 72 - distances

            # print(str(valleys[i][j]) +' '+str(distances))

            if distances < my_min:
                my_min = distances
                my_index = i

                # my_index2 = j


    closest_valley = valleys[my_index]
    # print(valleys[my_index][my_index2])

    if len(closest_valley) <= self.s_max:
        return closest_valley[int(len(closest_valley)/2)]
    else :
        return closest_valley[my_index+int(self.s_max/2)]
```

- تابع move_side و تابع move_forward:

این دو تابع عملکرد بسیار ساده‌ای دارند و تنها به اندازه زاویه ورودی یا میزان حرکت خطی ورودی ربات را حرکت می‌دهند.