South Dakota School of Mines and Technology

Advanced Robotic Algorithms, Spring 2022

CSC 692 - M01

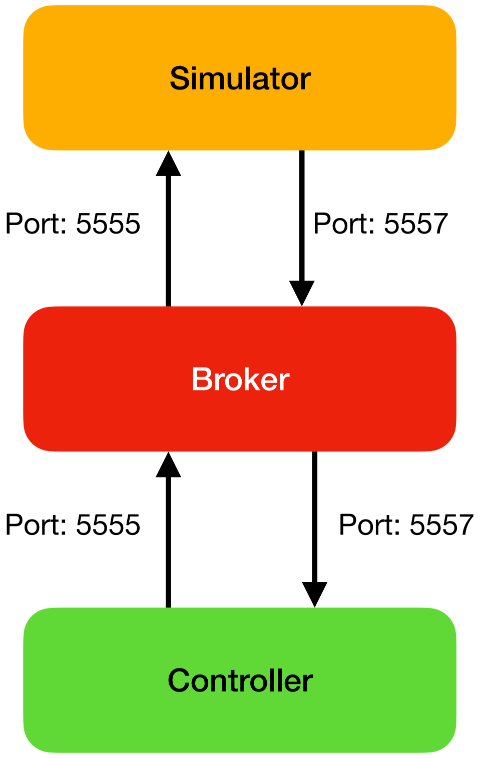
Final Project Report

The Project

The aim of this project was to design a SLAM system that communicated with a Simulator, that simulated a Differential Drive Robot, over ZMQ. The Simulator accepted wheel speed values from a controller and then simulated the robot (albeit with some noise). The Robot simulated a Lidar Sensor that recorded distance values up to 0.5 units, every 9 degrees in a 180 degree sweep and published them to the bus with some Gaussian Noise.

Communication Bus

The communication bus was designed on the base of a ZMQ PUB-SUB Bus with a broker acting as a middle grounds, connecting the simulator to the controller as shown below.



In our model we integrated the SLAM Module with the Controller. This Controller was also responsible of processing the data and creating a map. We decided to go with this approach as we wanted to reduce bloat over the Communication BUS.

Parts of our SLAM Module:

- The Controller
- The Localizer
- The Filter
- The Mapper

The Controller

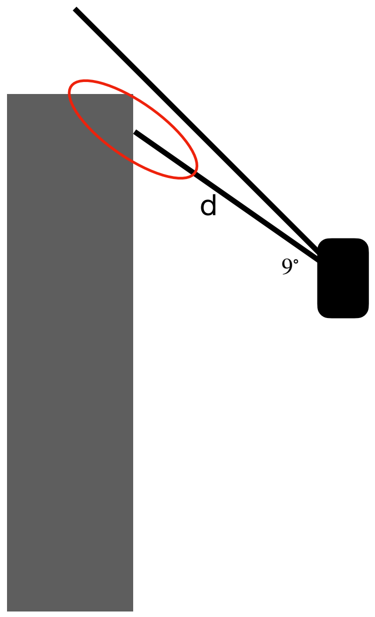
The controller was the main 'brain' of the operations. Its primary responsibility was handling the data received and passing it on over to the other 'sub modules'. It was also responsible in publishing wheel velocities. On a broad scale, it implemented a really low level bug algorithm - if the robot sensed itself getting a little too close to an object, it would increase the speed of its right wheel, thereby turning clockwise till it saw open ground. The controller was implemented as while loop and the code to it can be found here - slam_mapper.py

The Localizer

The aim of the Localizer was to identify landmarks and approximate the position of the robot via the Landmarks.

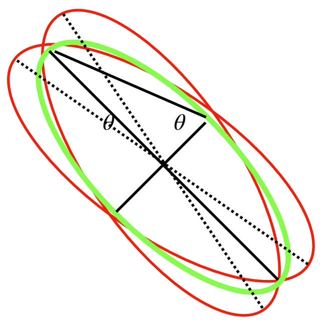
The Landmark identification was done by the identify_landmark function shown below. It did a sweep among the distances received from the Lidar sensor, looking for an anomaly. An anomaly in this case is a situation in which the difference between two consecutive angle values is greater than 0.5 * max_sensed_distance.

For example consider the diagram below. The following situation would be considered an anomaly. What's evident from this approach is that our model is proficient in detecting corners and thats something we want to exploit in the future sections.



The only problem here is that we don't know the exact position of the edge of the corner. So each position is represented by an ellipse whose major axis is equal to the uncertainty or 2 times the standard deviation of the Lidar distance measurement and whose minor axis is equal to $d * sin(9^{\circ})$ where d is the distance measured by the Lidar sensor. We assume the position of the center of the ellipse to be w.r.t the predicted state, which will be then updated once we get the estimated stat from the filter.

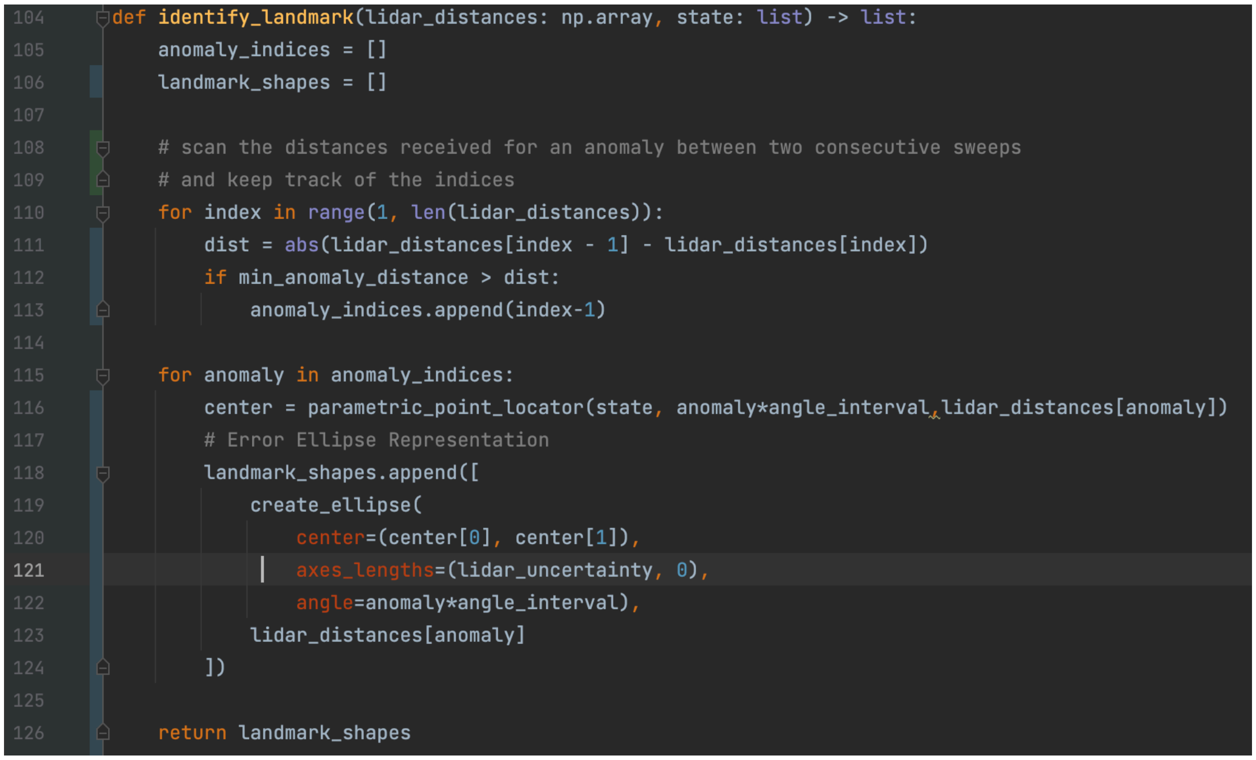
Consecutive measurements of the same anomaly will reduce the uncertainty as follows in the next diagram. (Based on the principles of set theory) The green ellipse is the resultant whose major axis and minor axis can be calculated using the intersections of both the ellipses.



If two ellipses have their centres offset, the new centre will be the insterction between the newly found major and minor axis.

The main assumption made here is that between 2 consecutive states [x, y, theta], an anomaly detected will be the same landmark and it will have relatively the same coordinates w.r.t the origin (the point at which the robot started from) as the previous state. And so using that information we can calculate the ‘observed’ or ‘sensed’ state of the next iteration with the new lidar distances and at what angle it was measured.

The code shown below implements of the anomaly detector.

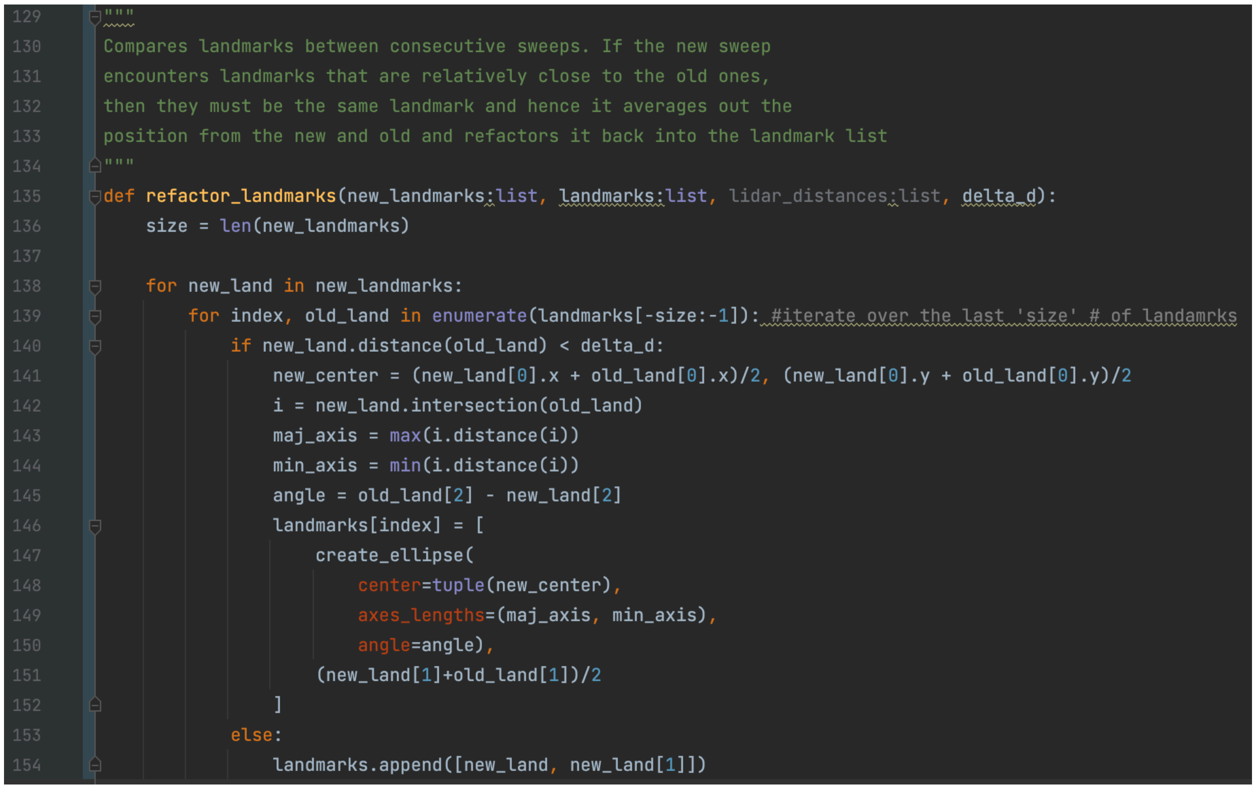


```
104 def identify_landmark(lidar_distances: np.array, state: list) -> list:
105     anomaly_indices = []
106     landmark_shapes = []
107 
108     # scan the distances received for an anomaly between two consecutive sweeps
109     # and keep track of the indices
110     for index in range(1, len(lidar_distances)):
111         dist = abs(lidar_distances[index - 1] - lidar_distances[index])
112         if min_anomaly_distance > dist:
113             anomaly_indices.append(index-1)
114 
115     for anomaly in anomaly_indices:
116         center = parametric_point_locator(state, anomaly*angle_interval,lidar_distances[anomaly])
117         # Error Ellipse Representation
118         landmark_shapes.append([
119             create_ellipse(
120                 center=(center[0], center[1]),
121                 axes_lengths=(lidar_uncertainty, 0),
122                 angle=anomaly*angle_interval),
123             lidar_distances[anomaly]
124         ])
125 
126     return landmark_shapes
```

Line 110 to 113 checks for anomalies and keeps track of the indices. The following lines create a Shapely Ellipse. Line 116 calls a function I designed to get the location of the center using the parametric formula of a line (point along with slope tan(pi*index/20) at a distance <lidar_distance(index)>).

A landmark element is a three element list consisting of [ellipse_object, last_distance, angle]. The last_distance is in finding the sensed state. Other than that it has no broader scope use.

The following function refactor_landmarks is used to compare the newly found landmarks with the existing ones to see if a previously recorded landmark was found.



```
129 """
130 Compares landmarks between consecutive sweeps. If the new sweep
131 encounters landmarks that are relatively close to the old ones,
132 then they must be the same landmark and hence it averages out the
133 position from the new and old and refactors it back into the landmark list
134 """
135 def refactor_landmarks(new_landmarks:list, landmarks:list, lidar_distances:list, delta_d):
136     size = len(new_landmarks)
137 
138     for new_land in new_landmarks:
139         for index, old_land in enumerate(landmarks[-size:-1]): #iterate over the last 'size' # of landamrks
140             if new_land.distance(old_land) < delta_d:
141                 new_center = (new_land[0].x + old_land[0].x)/2, (new_land[0].y + old_land[0].y)/2
142                 i = new_land.intersection(old_land)
143                 maj_axis = max(i.distance(i))
144                 min_axis = min(i.distance(i))
145                 angle = old_land[2] - new_land[2]
146                 landmarks[index] = [
147                     create_ellipse(
148                         center=tuple(new_center),
149                         axes_lengths=(maj_axis, min_axis),
150                         angle=angle),
151                     (new_land[1]+old_land[1])/2
152                 ]
153             else:
154                 landmarks.append([new_land, new_land[1]])
```

Line 143 and 144 show the usage of the Shapely intersection function to get the min and maj axis lengths. The angle of the resultant is calculated in line 145.

Now for measuring the sensed value, we just used my parametric form function to find a point thats in the direction of $\pi - angle$ of the centre at a distance given by the value in landmark[2]. One interesting parameter is delta_d, which corresponds to the maximum error in distance traversed due to error in wheel speed which is given by

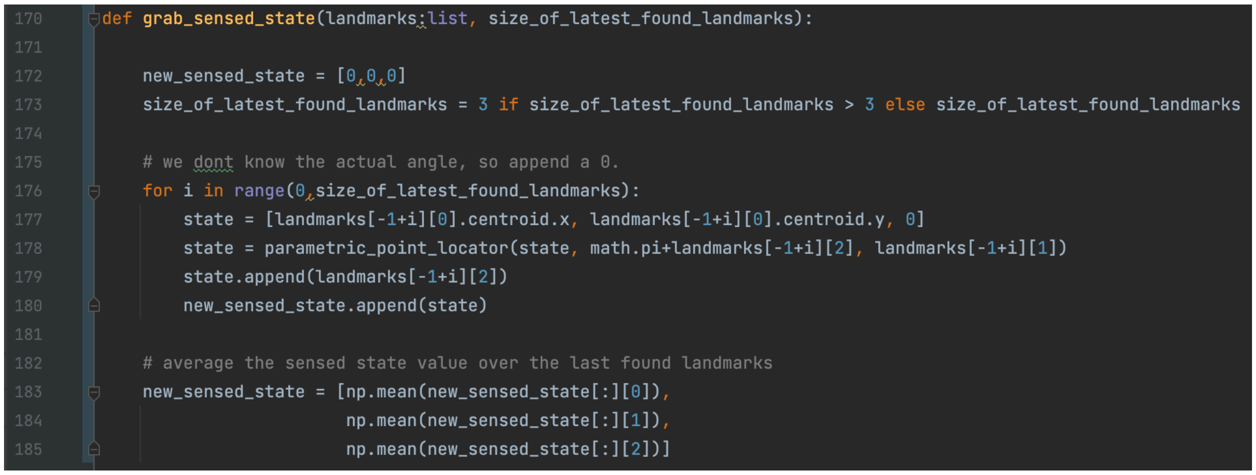
$$\frac{r}{2} * (\delta\omega_1 + \delta\omega_2) \ * \Delta t$$

This consequently is the maximum Euclidean distance error between the predicted value and sensed value. This translates to the to obstacles and hence if the sam object was detected, the maximum distance between their centers can be what the above formula equates to.

This function below shows the method for obtaining the sensed stat from the last found landmarks. Basically, iterate of the landmarks you just identified and refactored in the current state and average the values of the sensed state you get.

* landmarks is a list of a 3 list object -> [shapely_object, distance, angle] where distance and angle are the last updated distance and angle values.

As of now, our sensed state approximation is a fairly accurate estimate of our actual position with uncertainty in the position spread about an ellipse of major axis length = the uncertainty in the lidar measurement value and minor axis = maximum error in distance traversed (formula on the previous page).

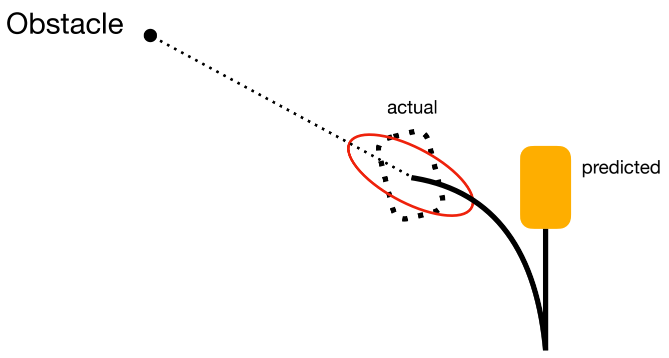


```python
def grab_sensed_state(landmarks:list, size_of_latest_found_landmarks):

    new_sensed_state = [0,0,0]
    size_of_latest_found_landmarks = 3 if size_of_latest_found_landmarks > 3 else size_of_latest_found_landmarks

    # we dont know the actual angle, so append a 0.
    for i in range(0,size_of_latest_found_landmarks):
        state = [landmarks[-1+i][0].centroid.x, landmarks[-1+i][0].centroid.y, 0]
        state = parametric_point_locator(state, math.pi+landmarks[-1+i][2], landmarks[-1+i][1])
        state.append(landmarks[-1+i][2])
        new_sensed_state.append(state)

    # average the sensed state value over the last found landmarks
    new_sensed_state = [np.mean(new_sensed_state[:][0]),
                        np.mean(new_sensed_state[:][1]),
                        np.mean(new_sensed_state[:][2])]
```



The Filter

For small time steps our unfiltered sensed state estimation is a fairly accurate localization of our actual position. However, one state variable that was really difficult to sense was orientation (and I still haven't found the best trigonometric way to calculate).

Fortunately, the Extended Kalman Filter works in magical ways. It can get us a pretty good estimate of the actual orientation of an unobserved variable from the P Covariance Matrix.

The Kalman process is fairly simple as shown below

1. Predicted state:
   $$\hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k)$$
2. Predicted estimate covariance:
   $$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + V_k$$
3. Optimal Kalman gain:
   $$K_k = P_{k|k-1} H_k^{\mathrm{T}} \left(H_k P_{k|k-1} H_k^{\mathrm{T}} + W_k \right)^{-1}$$
4. Updated state estimate:
   $$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \left(z_k - h(\hat{x}_{k|k-1}) \right)$$
5. Updated estimate covariance:
   $$P_{k|k} = (I - K_k H_k) P_{k|k-1}$$

The Predicted state is simply this equation written in matrix form.

$$x_{k+1} = x_k + \tfrac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k})\cos(\theta_k)$$

$$y_{k+1} = y_k + \tfrac{r\Delta t}{2}(\omega_{1,k} + \omega_{2,k})\sin(\theta_k)$$

$$\theta_{k+1} = \theta_k + \tfrac{r\Delta t}{2L}(\omega_{1,k} - \omega_{2,k})$$

The P matrix is $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ initially and gets changed as the state updates.

The H matrix is the observation matrix and is $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ as we can only observe 2 state variable out of the 3.

The V matrix is the covariance of the input values (omega1 and omega2). I couldn't really arrive at set values for this as I didn't know the standard deviation of the noise. But if I did the matrix would be $\begin{bmatrix} \sigma^2 & 0 \\ 0 & \sigma^2 \end{bmatrix}$

The W matrix is the covariance of the measured or sensed values and is defined as follow $\begin{bmatrix} l & 0 \\ 0 & l * sin(pi/20) \end{bmatrix}$ where l is the variance in the lidar noise.

I started with the approach of designing the Kalman Filter from scratch in my code but then realized I had wasted time on that and could have just imported it from filter.py. Unfortunately due to lack of time I couldn't get either of them done.

Once we get the state update from the Kalman Filter, that is gonna be our most accurate estimate; we then use this to iteratively calculate successive approximations.

The Mapper

I mentioned earlier that our ability to identify corners would be very useful to us; here's where we can exploit this. Since we know now that all the stored landmarks are corners, connecting these points should give us the obstacles. This methodology however gives rise to two hurdles:

- Identifying if an edge connection crosses a traversed path or open space
- Identifying when an the polygon we connect is closed

I haven't arrived at coding up this module yet but my projected approach would be as follows in resolving the hurdles.

Identifying if an edge connection crosses a traversed path or open space :

- Connect all the path points we stored to create a Shapely Multipoint Object.
- Create another Shapely MultiPoint Object connecting the points of the edge we want to form.
- Use the shapely intersects function (a.intersects(b)) to check if edge traverses the path. If the function returns a Multipoint Point Empty Object, we know that it doesn't then and this edge probably corresponds to a polygon.

Identifying when an the polygon we connect is closed:

- Connected edges will be stored as a MultiPoint object.
- Use self.area( ) function to check for area. If the area is non 0, then we have completed a polygon.

About the Code

The code can be found in this repository: slam_repo

The **main branch** has the last tested code. Currently, when this report was written, this branch contained code implementing Obstacles as no-noise Point representations. There are few algorithmic inconsistencies with this approach. But it showcases plotting and a general idea of how the more developed algorithm works.

The **untested branch** has the current discussed implementation, however, as of now it's got a few bugs that makes it crash. I am actively working on fixing it. I created this branch as a 'testing branch' of different implementations.