

# CSE276A\_HW1

Authors: Weixiao Zhan (A59023453), Somanshu Singla (A59023795)

## Summary

We started this homework with just one python file with hard coded way points. This is present in `control.py` file.

Later we upgrade to a two-roscnode setup ( `control_node.py` , `path_planner_node.py` ):

$$\text{path planner node} \xrightarrow[\text{(} t_{\text{linear}}, t_{\text{angular}} \text{)}]{\text{waypoint}} \text{control node}$$

The path planner node ( `path_planner_node.py` ) reads in `waypoints.txt` , and computes what is the time needed to go a certain distance/ rotate to a certain orientation. Then path planner node encode this information in Joy format, and publish to a topic, named `/waypoint` .

The control node ( `control_node.py` ) subscribe to `/waypoint` topic. Upon receiving a message, the control node would use our kinematic model and our calibration parameters to compute each wheel speed, and use `megapi` to set actual wheel speeds.

## Calibration

1. Wheel Orientation: Our robot is wired that the motors are going counter-clockwise with positive values.
2. Calibrate each wheel: We measured each wheel's free rotation speed in both clockwise and counter-clockwise directions. A calibration constant was applied in the `setFourMotors()` function so that each wheel and direction to ensure that all wheels rotate at an approximately equal speed. After this, with some fine tuning, our car can go reasonable straight in forward and backward directions.

Wheel	Clock Wise factor	Counter-Clock Wise factor
Back Left	0.86	0.86
Back Right	0.852	0.85

Wheel	Clock Wise factor	Counter-Clock Wise factor
Front Left	0.86	0.88
Front Right	0.86	0.84

### 3. Real-world unit Calibration:

- For rotational calibration, due to the absence of a straightforward method to directly measure degrees, we adjusted the time to produce the required data points. we timed the robot's rotations at 45, 90, 180, and 360 degrees using a fixed motor command (45). The equation is  $\Theta_{w=45} = 73.19 * t - 19.71$
- For straight calibration, using similar methods, we have fitted  $D_{w=40} = 0.136 * t - 0.0412$

All unites are in `meters` , `seconds` , and `degrees`

## Kinematic Model

$$\begin{bmatrix} w_{fl} \\ w_{fr} \\ w_{bl} \\ w_{br} \end{bmatrix} = \frac{1}{0.03} \begin{bmatrix} 1 & -1 & -0.1225 \\ 1 & 1 & 0.1225 \\ 1 & 1 & -0.1225 \\ 1 & -1 & 0.1225 \end{bmatrix} \begin{bmatrix} v_{straight} \\ v_{side} \\ w \end{bmatrix}$$

## Path algorithm

Our path algorithm is very simple. For each way point, It publishes three messages.

We used constant angular velocity ( $w = 40$  for linear motion and  $w = 45$  for angular motion) and estimated the time to allow the robot to travel certain distance and rotate to certain orientations.

- rotate the car towards that way point
- move forward (carStraight) to the target point
- rotate the car to the orientation of the way point.

In addition, path planner would estimated how long the control node need to complete the operation. And publishes the next messages afterwards to prevent message got ignored in topic queue.

### Note:

We have also developed a method to control the robot by updating its linear and angular velocities at a frequency of 10Hz. We control each of the  $v_x$ ,  $v_y$  and  $\omega$  using PID controllers. Currently, we are able to move the robot using this method reasonably well but the gains need some more tuning. We plan to use this method starting HW2 (by when we would have completed the tuning).

The structure for this method can be found below (Code for this method is attached in `path_planner_node_org.py` and `control_node_org.py`):

path planner node org  $\xrightarrow[\text{(\textit{v}_x, \textit{v}_y, \omega)}]{\text{waypoint}}$  control node org

## Results

For our experiments we have scaled down the distances by a factor of two.

276\_HW1

 [https://youtu.be/bJY\\_b8Zovw4](https://youtu.be/bJY_b8Zovw4)



<https://github.com/Singla17/CSE276A.git>

Our overall results look reasonable when we ran the robot at the battery level which we calibrated it for. It doesn't move perfectly but our final distance error was reasonable (well within a foot) and there was also some orientation error (<30 degrees).

## Challenges

1. Broken robot: our first robot is broken with unstable frames. We changed the robot.

2. Calibration: Even though we calibrated our robot to move straight due to different friction on different wheels, it may not always move straight.
3. Time delay in ros-node communication: Subscribing to a topic can cause delays in communication between publisher and subscriber, due to which our subscriber often missed the first message. We by-passed this by posting a dummy-first message which can be ignore.
4. Battery level introduces in-consistency: with the exact code used in result video, the robot behaves differently depends on battery levels.

CSE276A\_HW1 (Different Battery Level)

 <https://youtu.be/TelDaLTTJJg>



# CSE276A\_HW2

Authors: Weixiao Zhan (A59023453), Somanshu Singla (A59023795)

[Closed-loop control](#)

[Landmarks](#)

[Estimate Robot Position](#)

[No Landmark Control:](#)

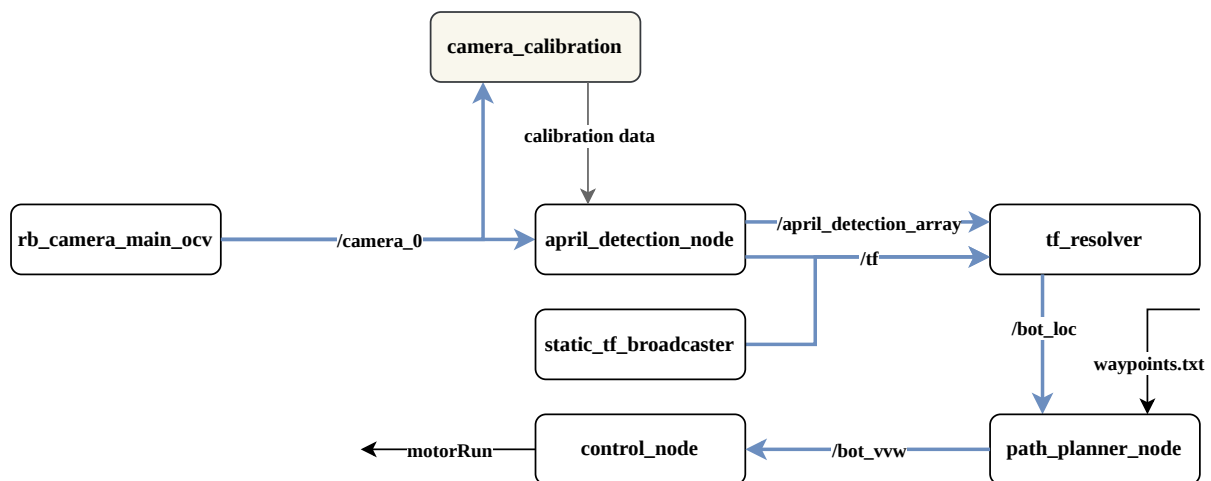
[Results:](#)

[Analysis](#)

[Challenges](#)

## Closed-loop control

This diagram outlines our closed-loop control system. Rectangles represent ROS nodes, blue arrows represent topics and arrows represent subscribers.



- The **rb\_camera\_main\_ocv** node streams video to the **/camera\_0** topic. Initial camera calibration is performed using the **camera\_calibration** node, and the resulting calibration matrix is hardcoded into the **april\_detection\_node**.
- The **april\_detection\_node** subscribes to the **/camera\_0** topic. When it detects April tags, it publishes pose messages and spatial translations (from markers to cameras to robot body) to **/april\_detection\_array** and **/tf** respectively. If no tags are detected, only the pose message is published.
- The **static\_tf\_broadcaster** node periodically broadcasts the spatial translations from the world frame to markers on **/tf** topic.

- The `tf_resolver` node subscribes to `/april_detection_array` and `/tf`. Upon receiving a pose message from `/april_detection_array`, It estimates the robot's position by aggregating spatial translations in `/tf` and then publishes the result to `/bot_loc`. This estimation method is explained further in the "estimate car pose" section.
- The `path_planner_node` operates on two threads. One reads in `waypoints.txt`, runs a PID path-planning algorithm, and publishes desired velocities to `/bot_vvw`. The second thread updates the PID's current location based on data from `/bot_loc`. More detail is elaborated on in the "No Landmark Estimate" section.
- Finally, the `control_node` is simply inherited from HW1.

## Landmarks

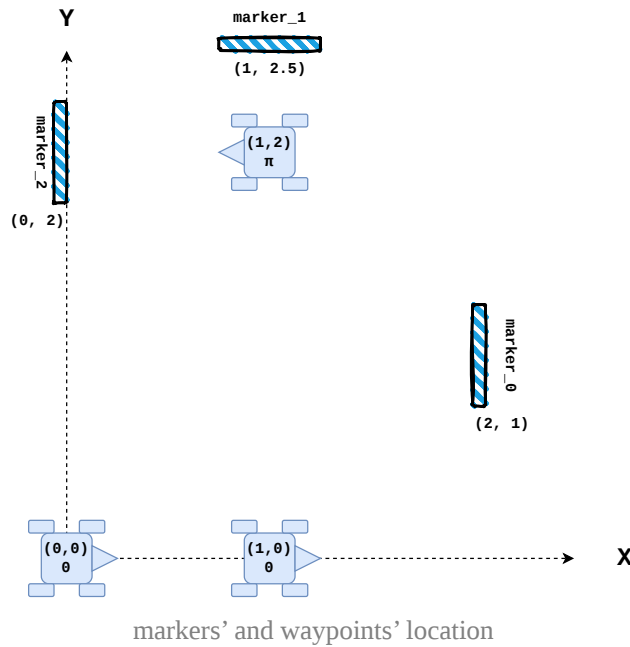
We print each April tags on a piece of A4 paper and tape them to the side of used boxes. We also tried to align the center of April tags to the same height as the robot camera.

- Three markers were used ( Their locations are shown in the graph below ).
- These locations were chosen in the following considerations:
  - marker\_0 at  $(2, 1)$  allows the robot to navigate to  $(1, 0, 0)$  and back to  $(0, 0, 0)$  properly.
  - marker\_1 at  $(1, 2.5)$  and marker\_2 at  $(0, 2)$  help the robot reach waypoint  $(1, 2, \pi)$ .
  - Our robot rotates and moves at the same time so we placed all three landmarks in different orientations so that the robot has enough visual guidance whatever pose it is in.

When the robot traveled from waypoint  $(1, 2, \pi)$  back to waypoint  $(0, 0, 0)$ , there was a certain period of time that the robot couldn't see any marker. In this situation, we rely on no landmark control, i.e. open-loop control, in our path planner node. Once the robot sees marker 0 again, the closed-loop control takes over, allowing the robot to return the waypoint  $(0, 0, 0)$  precisely.

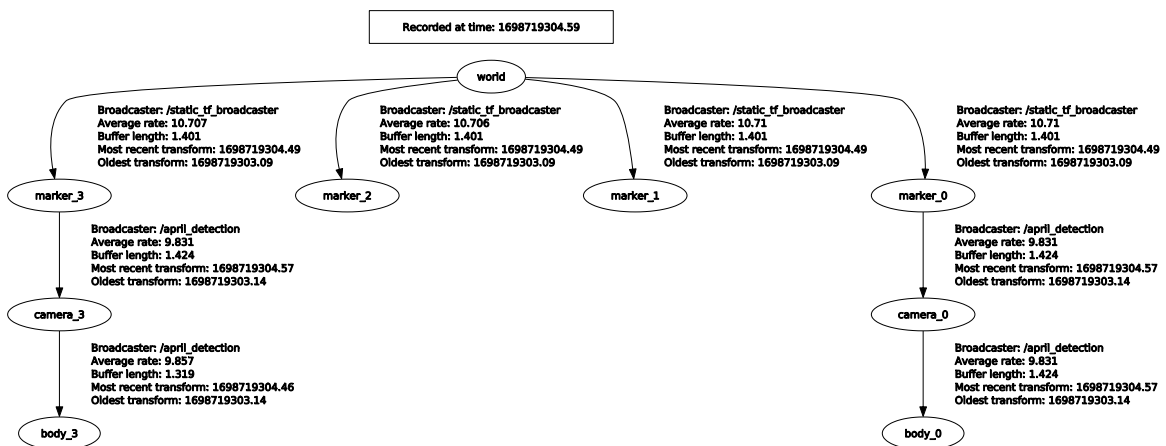
We also developed a weighted average mechanism to reduce the noise in location estimation when the robot can see multiple markers. (More detail in "Estimate Robot Position" section).

Also, since we take measurements from April Tags repeatedly at a high frequency the effect of noise should be minimal as our underlying is that it's white noise.



## Estimate Robot Position

The diagram below shows an example of `tf_tree`, which we used to estimate our car's position.



- world → marker\_i: the `static_tf_broadcaster` broadcasts these translations. We built our `static_tf_broadcaster` to use four markers. But later in the project, marker\_3 is removed, so it doesn't show up in our landmark graph but still exists in our tf\_tree.
- marker\_i → camera\_i: April tag library takes care of this.
- camera\_i → body\_i: We assume the camera frame's center is the same as the body frame's center. This assumption makes camera-to-body translation only consisting of rotations and allows us to avoid measuring distance and angles inside the robot, which is

difficult and prone to errors. In addition, we align the center of April tags to the same height of the robot camera, so that we can approximate the camera-to-body rotation has all  $90^\circ$  Euler angles.

All translations above are present in `/tf`. Our `tf_resolver` uses ROS build-in functionality to compose these translation to get world  $\rightarrow$  body<sub>i</sub> translation  $\mathcal{T}_i = \begin{bmatrix} R_i & T_i \end{bmatrix}$ .

- body  $\rightarrow$  /bot\_loc: the 2D robot location representation,  $[x, y, ori]$ , is simply the projection of the 3D translation.

$$\begin{cases} x_i = T_i.x \\ y_i = T_i.y \\ ori_i = R_i.z \end{cases}$$

If there are multiple markers have been detected, the `tf_resolver` would report an averaged estimation weighted by the inverse of each marker's distance.

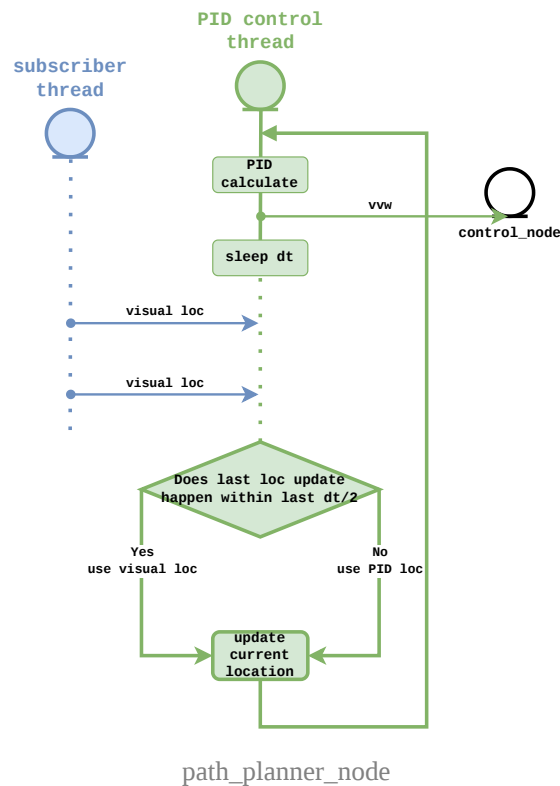
$$\begin{cases} x = \frac{\sum_i w_i x_i}{\sum_i w_i} \\ y = \frac{\sum_i w_i y_i}{\sum_i w_i} \\ ori = \frac{\sum_i w_i ori_i}{\sum_i w_i} \end{cases}, w_i := \frac{1}{\sqrt{x_i^2 + y_i^2}}$$

## No Landmark Control:

Our strategy for no landmark control is to do open-loop control from the location, which is last updated by closed-loop, and hope the closed-loop (visual feedback) can be re-established before the open-loop exit. Because we have the freedom to choose the position of landmarks, we believe that implementing an algorithm that actively seeks landmarks through random routes or rotations when visual feedback is lost is unnecessary.

We recognize the key difference between open-loop (no landmark) and closed-loop in the PID controller is how the current location got updated. If the current location is estimated from visual or other sensors, the PID controller is closed-loop; if the current location is calculated from an internal model, the PID controller is open-loop. Based on this insight, we implemented our `path_planner_node`. It is illustrated in the following diagram.





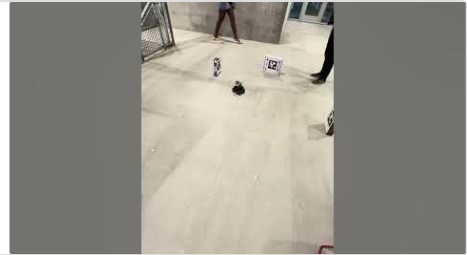
The `path_planner_node` runs in two threads. The subscriber thread subscribes to `/bot_loc` topic and writes the visual location estimation to a shared location data structure.

The PID control thread would first use the current location and target location to calculate and publish the desired  $v_x$ ,  $v_y$ ,  $w$ . Then, the PID thread sleeps  $dt$ . During this period, the subscriber thread may or may not update the shared location data structure. Upon waking up, the PID control thread needs to decide how to update the current location. The decision condition we implemented is only to use the shared location data structure, i.e. the visual location, when the subscriber thread made the update in the past  $\frac{dt}{2}$ ; otherwise, use the kinetics model of distance = velocity \* time. After the update, repeat this process.

Our simple design allows the robot to seamlessly switch between open-loop and closed-loop PID control in each iteration. We used a higher refresh rate for PID at 20Hz, compared to the visual loop's 10Hz, to make our robot move more smoothly.

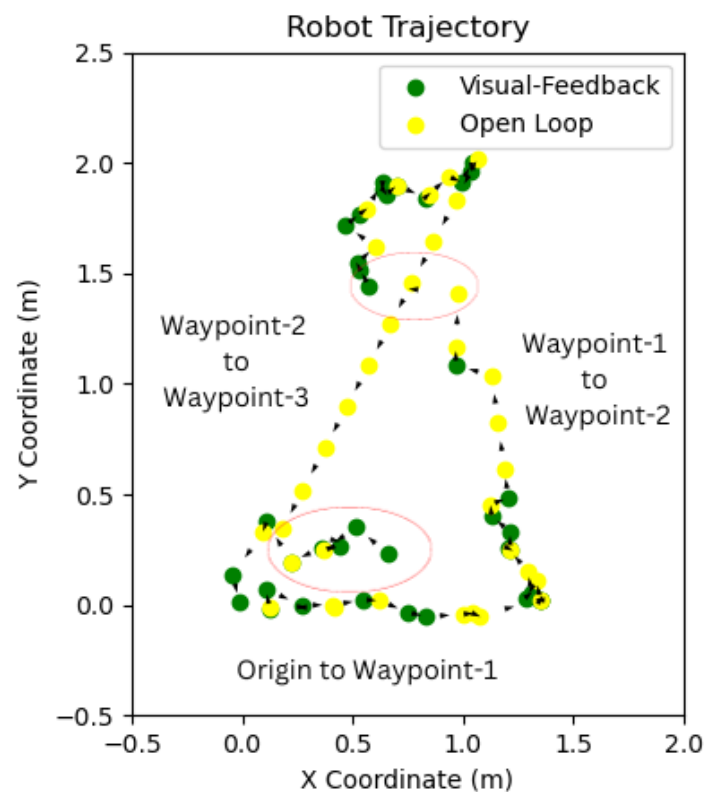
We also acknowledge this design has some flaws in corner cases. For instance, if our robot can't any landmarks and the open loop thinks it has reached the target location, the `path_planner_node` will halt. Our solution is to ensure that the area around each waypoint, with a radius of the open-loop control error, can all see the landmark. so that the robot will have a chance to correct its location from the closed-loop before exiting.

## Results:



## Analysis

### 1. Trajectory:



- This is the trajectory of our robot.
  - The green dots represent the points where we localized using April tags.
  - The yellow dots represent the locations where the robot thinks where it's at.
  - The small arrow represents the orientation of the robot.
- When traveling from waypoint  $(1, 2, \pi)$  to waypoint  $(0, 0, 0)$ , the robot lost visual on all markers near the top red ovals, and later regained visual near the bottom ovals. During this period, the robot relied on open-loop control, thus it logged a perfectly straight line of yellow dots. After regaining visual on marker\_0, the robot is able to

correct its location estimation to the green dots in the bottom red oval, and successfully return to waypoint  $(0, 0, 0)$

2. Smoothness: Our robot's motion was fairly smooth for most of the trajectory, except for the part where it corrects small localization errors using closed-loop control. When making corrections based on  $v_x$  or  $\omega$ , our bot's motion is fairly smooth and confident (can be seen when it tries to move back to the origin in the video). The smoothness could be improved when the bot is sliding, but due to the significantly higher friction, we can't improve much unless we change the PID control algorithm to the bicycle model.

Note: we deliberately parked the robot on each marker for 5 seconds so that we may have time to measure the location error at each waypoint.

3. Moving Distance: Our Robot moved approximately 6.3 m and it rotated approximately 6.5 radians (calculated by only the location estimates from April Tag). The ideal moving distance was about 5.23 m and the ideal rotation was about 6.28 radians, which is 20.5% more in distance and 3.5% more in rotation. We are fairly satisfied with our result as the significant sliding friction caused our robot's control less accurate.

#### 4. Location Error:

For all three waypoints, as shown in the video, our robot fully covered every waypoint marker. At this precision, it is hard for us to measure the exact location error, but to conclude that the location error at each waypoint is less than the size of the robot:

- x and y localization errors are both much smaller than 5.5 (11/2) cm (that is the distance from one side of the robot to its center).
- angular errors were less than 10 degrees.

We also provide the localization errors reported in the log: format is  $(\Delta x, \Delta y, \Delta ori)@(x, y, ori)$

$(0.027, 0.001, 0.002)@(1, 0, 0)$   
 $(0.030, 0.020, 0.030)@(1, 2, \pi)$   
 $(0.001, 0.007, 0.003)@(0, 0, 0)$

Overall, we believe that our robot performed reasonably well given that it was able to reach the waypoints fairly accurately. Something to improve on in future works would be make the sliding motion better.

## Challenges

1. Sliding is a lot harder for the bot due to higher friction, we had to adjust for this issue in our controller by keeping sliding velocities a little higher than angular/straight velocities.
2. Tuning PID controller parameters takes up a lot of time. The parameters for the PID controller after tuning were:  $K_p = 0.4$ ,  $K_I = 0.01$ ,  $K_d = 0.03$ . We greatly reduced  $K_I$  to prevent overshoot.
3. We had to set up and calibrate April tags' position multiple times. The orientation has a big impact on the accuracy of the estimated location.

# CSE276A\_HW3

Authors: Weixiao Zhan (A59023453), Somanshu Singla (A59023795)

[Changes in Closed-loop Control Structure](#)

[Ground-truth Map](#)

[Kalman Filter Algorithm](#)

[notation](#)

[helper lists](#)

[definition](#)

[initialization](#)

[handling landmarks](#)

[Results](#)

[square and octagon edge length](#)

[square \(1 lap\)](#)

[square \(2 laps\)](#)

[octagon \(1 lap\)](#)

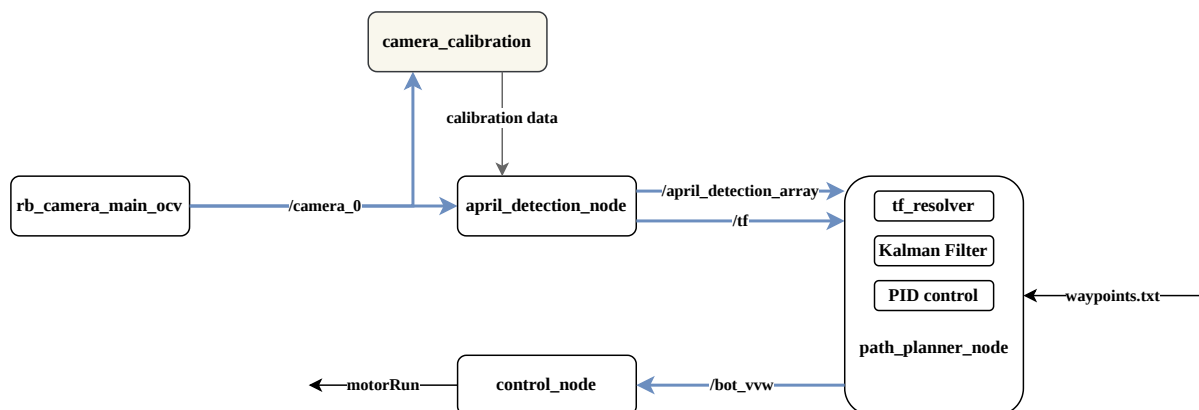
[octagon \(2 laps\)](#)

[Comparison](#)

[square VS octagon](#)

[single-run VS multi-run](#)

## Changes in Closed-loop Control Structure



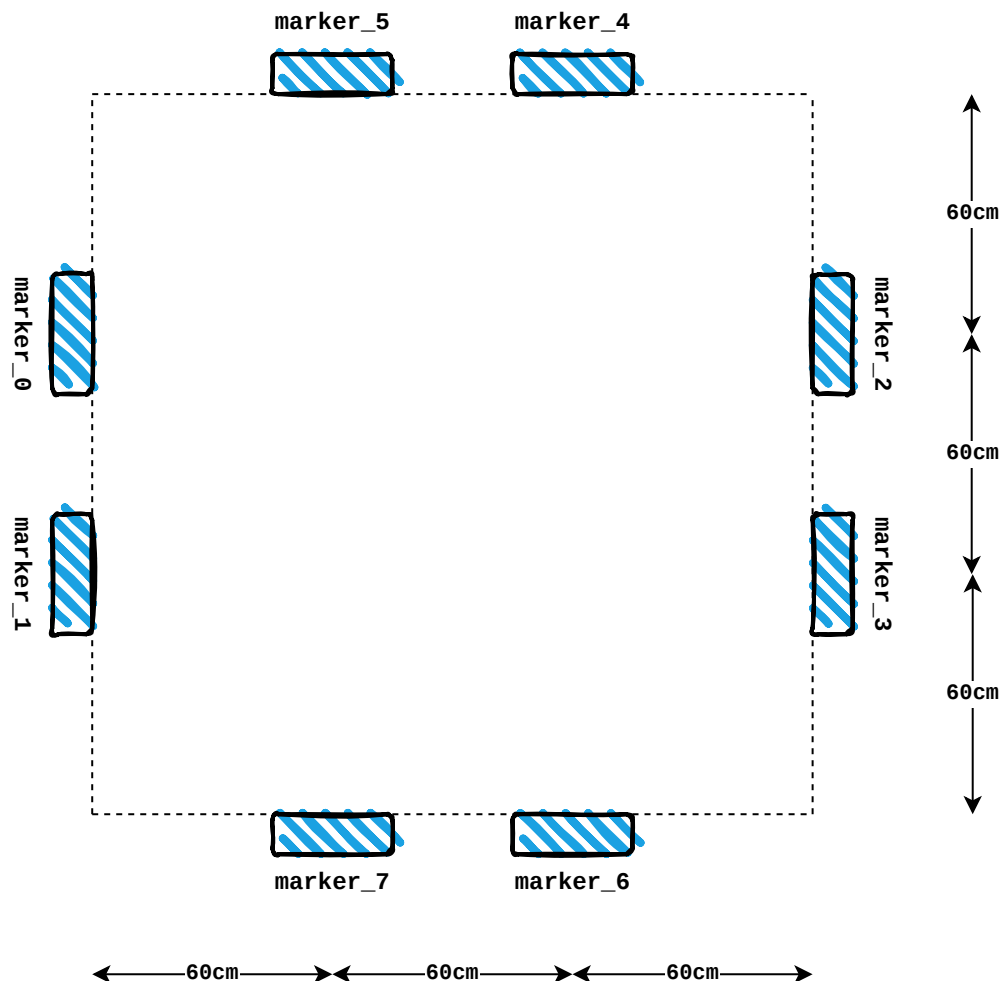
We started HW3 from our HW2's implementation, but we also made a few changes to cope with the new requirement of HW3:

1. We removed `static_tf_broadcaster`, since the robot shouldn't know the markers' location.

2. We modified the `april_detection_node` to publish translation from `body` to `camera` to `marker_i`.
3. We also migrated `tf_resolver` and implemented Kalman Filter and PID control as python classes rest inside `plath_planner_node` process. Because of their frequent exchange of location data, we believe resting them within one process and utilize shared memory and multi-thread lock can reduce the computation delay and thus reduce the system noise.

## Ground-truth Map

Eight markers are places on the edge of an 1.8m x 1.8m rectangle with equal space between markers. Note that the origin is defined at the initial starting position of the robot.



## Kalman Filter Algorithm

## notation

$x_r, y_r, \theta_r$  are robot's location and orientation in world frame.

$x_i, y_i$  are landmark  $i$ 's location in world frame.

$x_i^r, y_i^r$  are landmark  $i$ 's location in robot frame.

## helper lists

`lm_ordering` is a helper list that represents all the seen April tag's indices and maps the indices in state vector to April tag's indices.

`s` is a list that includes the currently seeing April tag's indices. `s` always is a subset of `lm_ordering` and keeps the same order as `lm_ordering`

## definition

1. state vector

$$x := \begin{bmatrix} x_r & y_r & \theta_r & \cdots & x_i & y_i & \cdots \end{bmatrix}^T$$

For instance, `lm_ordering = [2, 7]` means the state vector  $x =$

$$\begin{bmatrix} x_r & y_r & \theta_r & x_2 & y_2 & x_7 & y_7 \end{bmatrix}^T$$

2. measurement vector

$$z := \begin{bmatrix} x_{s_0}^r & y_{s_0}^r & x_{s_1}^r & y_{s_1}^r & \cdots \end{bmatrix}^T$$

For instance, `lm_ordering = [2, 7]`, `s = [7]` means we have seen marker\_2 and marker\_7, but currently only sees marker\_7. The measurement vector  $z = \begin{bmatrix} x_7^r & y_7^r \end{bmatrix}$

3. system matrix

$$F := I_n$$

When there is no control signal, i.e. the robot stops, the state vector should not change.

4. control matrix

$$G := \begin{bmatrix} dt & & & & \\ & dt & & & \\ & & dt & & \\ & & & 0 & \\ & & & & \ddots \end{bmatrix}$$

$dt$  is the time step of our controller. The robot movement control signal after multiplying  $G$  only changes  $[x_r \ y_r \ \theta_r]^T$  and landmarks' locations stay the same.

##### 5. measurement matrix

$$H = \begin{bmatrix} \vdots \\ H_{s_i} \\ \vdots \end{bmatrix}$$

$$H_{s_i} = \begin{bmatrix} -\cos(\theta_r) & -\sin(\theta_r) & 0 & \cdots & \cos(\theta_r) & \sin(\theta_r) & \cdots \\ \sin(\theta_r) & -\cos(\theta_r) & 0 & \cdots & -\sin(\theta_r) & \cos(\theta_r) & \cdots \end{bmatrix}$$

The coefficients in  $H_i$  are derived from the transformation:

$$x_i^r = +x_i \cos(\theta_r) - x_r \cos(\theta_r) + y_i \sin(\theta_r) - y_r \sin(\theta_r)$$

$$y_i^r = -x_i \sin(\theta_r) + x_r \sin(\theta_r) + y_i \cos(\theta_r) - y_r \cos(\theta_r)$$

$$\therefore \begin{bmatrix} x_i^r \\ y_i^r \end{bmatrix} = H_i \begin{bmatrix} x_r & y_r & \theta_r & \cdots & x_i & y_i & \cdots \end{bmatrix}^T$$

- we tried Extended Kalman filter, however, we weren't able to tune the noise matrices properly for that. So we kept using the above definition of  $H$  in our program.

### initialization

1. state vector  $x_{0|0} := [0 \ 0 \ 0]^T$

we define the initial position of the robot as the origin of world frame.

2. covariance  $\Sigma_{0|0} := \begin{bmatrix} 0.0001 & & \\ & 0.0001 & \\ & & 0.0001 \end{bmatrix}$

We chose very small values to initialize covariance because at  $t=0$  we are very confident about the robot being at origin.

3. system noise  $Q := \begin{bmatrix} 0.05 & & & & \\ & 0.05 & & & \\ & & 0.005 & & \\ & & & 0.03 & \\ & & & & 0.03 & \ddots \end{bmatrix}$



These values were chosen based on our experience from HW-1, given a command to move 1m, our robot committed errors within the range of 0.2m, so we took variance of noise in  $x$  and  $y$  a little higher than  $(0.2)^2$ . Our robot's rotation errors were smaller, around 5 degrees based on experience. So we took a smaller value for system noise for  $\theta_r$ .

We also took the system noise terms for  $x, y$  of April Tags as 0.03 because of the error in April tags' placement. The exact values were determined by empirical testing.

4. measurement noise  $R := 0.0001 * I_n$

We took these values based on our experience from HW-2, we saw that our bot was able to move at cm level accuracy if it could clearly see an April Tag, so we took the variance of noise in  $x$  and  $y$  as  $(0.01)^2$ .

## handling landmarks

### 1. new landmark

When our bot detects any new landmark (landmark index not in `lm_ordering`), we waited until this new landmark been detected 5 times before really adding it to our Kalman filter.

Some times the April detection reports false tags, ones that don't even exists in our setup environment. This patience filter helps prevent those false reports enter our Kalman filter. We also noticed that the new April tags always seem to enter the field of view from the edges of the camera frame. Additionally, based on our experience with HW2, when the April tag is near the edge of the field of view, the estimation of distance is not accurate due to camera distortion. This patience filter allows April tags to move into the view, so that we can have a better initial estimation of its locations.

When adding a new landmark, we 1) append its index to `lm_ordering`, 2) extend the state vector  $x$ , system noise matrix  $Q$ , measurement noise  $R$ , and covariance matrix  $\Sigma$ .

### 2. out-of view landmark and re-appearing landmark

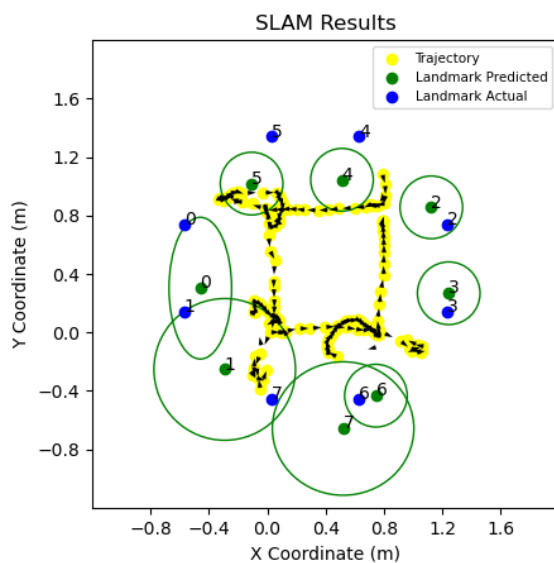
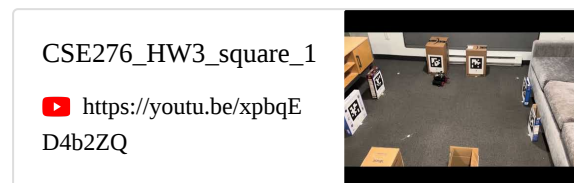
In each iteration, we first update the seeing set `s`, then compose  $H$  on the fly, finally perform the Kalman filter update algorithm. So whether there is landmarks going out of view or reentering, our Kalman filter always have correct matrices. Out of view landmarks' states are not updated, and re-appearing landmarks are updated just like all the other landmarks we are seeing.

## Results

## square and octagon edge length

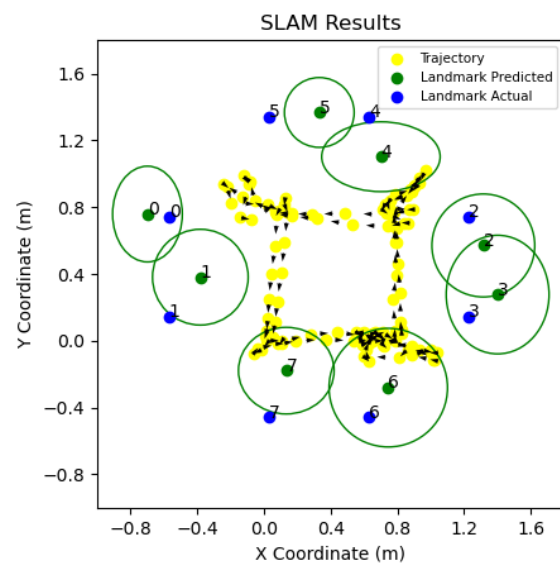
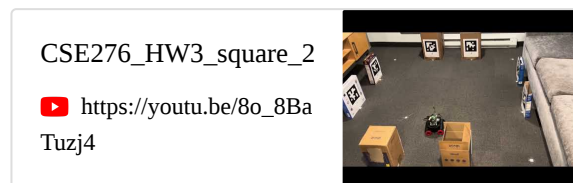
1. Square: We took edge length as 0.8m
2. Octagon: We took the edge length as 0.3313m
3. Factors that led to the choices:
  - a. We wanted the robot to travel enough distance so that the Kalman filter has enough iterations to actually get reasonable estimates of the landmarks.
  - b. We wanted to leave some space for the robot to correct itself before hitting the landmarks.
  - c. We are limited by the dorm size.

### square (1 lap)



average error in x	average error in y
0.1709m	0.2397m

### square (2 laps)



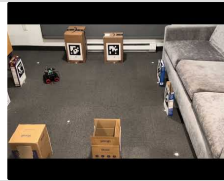
average error in x	average error in y
0.1457m	0.1610m

### octagon (1 lap)

### octagon (2 laps)

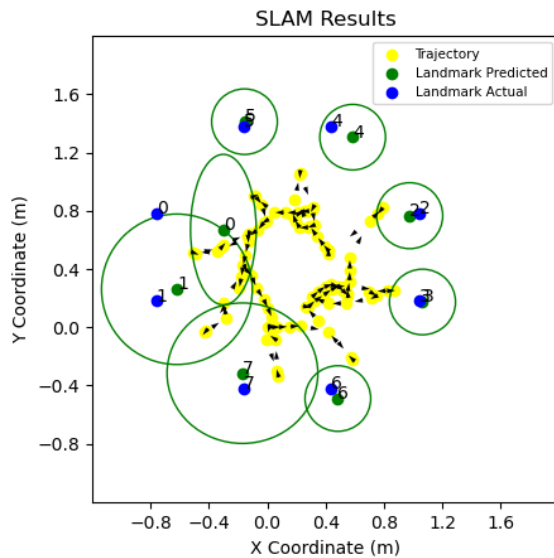
CSE276\_HW3\_octagon\_1

<https://youtu.be/ty9jG-VpDmo>

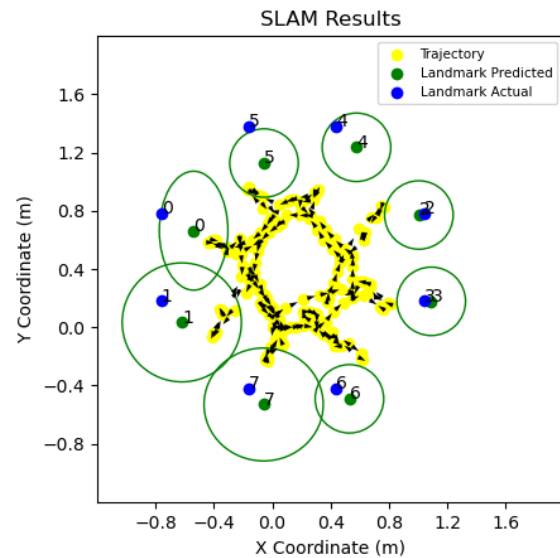


CSE276\_HW3\_octagon\_2

<https://youtu.be/uo239bRnGq8>



average error in x	average error in y
0.1111m	0.0614m



average error in x	average error in y
0.1091m	0.1072m

*The green ellipses in the plots show the uncertainty in the mapping by bot.*

## Comparison

### square VS octagon

average error and uncertainties of landmarks' x and y are both smaller in octagon motion than square motion

We recognize the cause is: in octagon movement, the robot is making smaller rotation. When it detects a new landmark, the robot can still see a lot of old landmarks to help localize the new landmark.

In addition, the rotating speed would be smaller in octagon movement, so the robot traveled or rotated less during the delay between the image been captured till Kalman filter got updated, which reduced the system noise.

### single-run VS multi-run

1. The robot typically reports higher uncertainties for marker 0, 1, and 7 in 1 lap movement. We recognize the cause is these landmarks are detected at the very end, so the robot doesn't see them long enough to lower the uncertainty. With a second lap, the uncertainty of landmarks 0, 1, 7 indeed decreased, but still less certain in these landmarks compared to others.
2. In square motion, we saw a material improvement in mapping by the robot and the uncertainties also reduced for a good number of landmarks.
3. In octagon motion, although the average error of localization didn't improve, the ground truth of Marker\_0 is updated to within the uncertainty zone of its estimation.

# CSE276A\_HW4

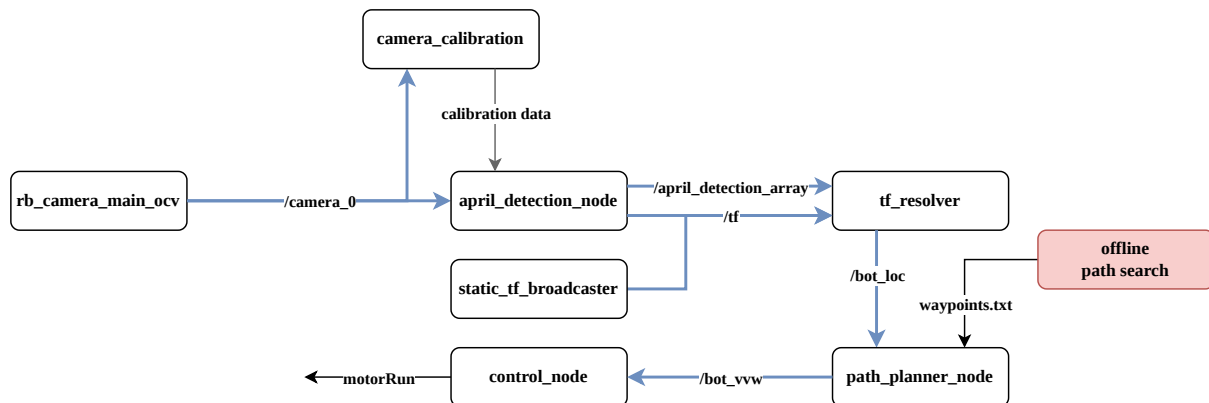
Authors: Weixiao Zhan (A59023453), Somanshu Singla (A59023795)

## Control Architecture

This diagram outlines our closed-loop control system.

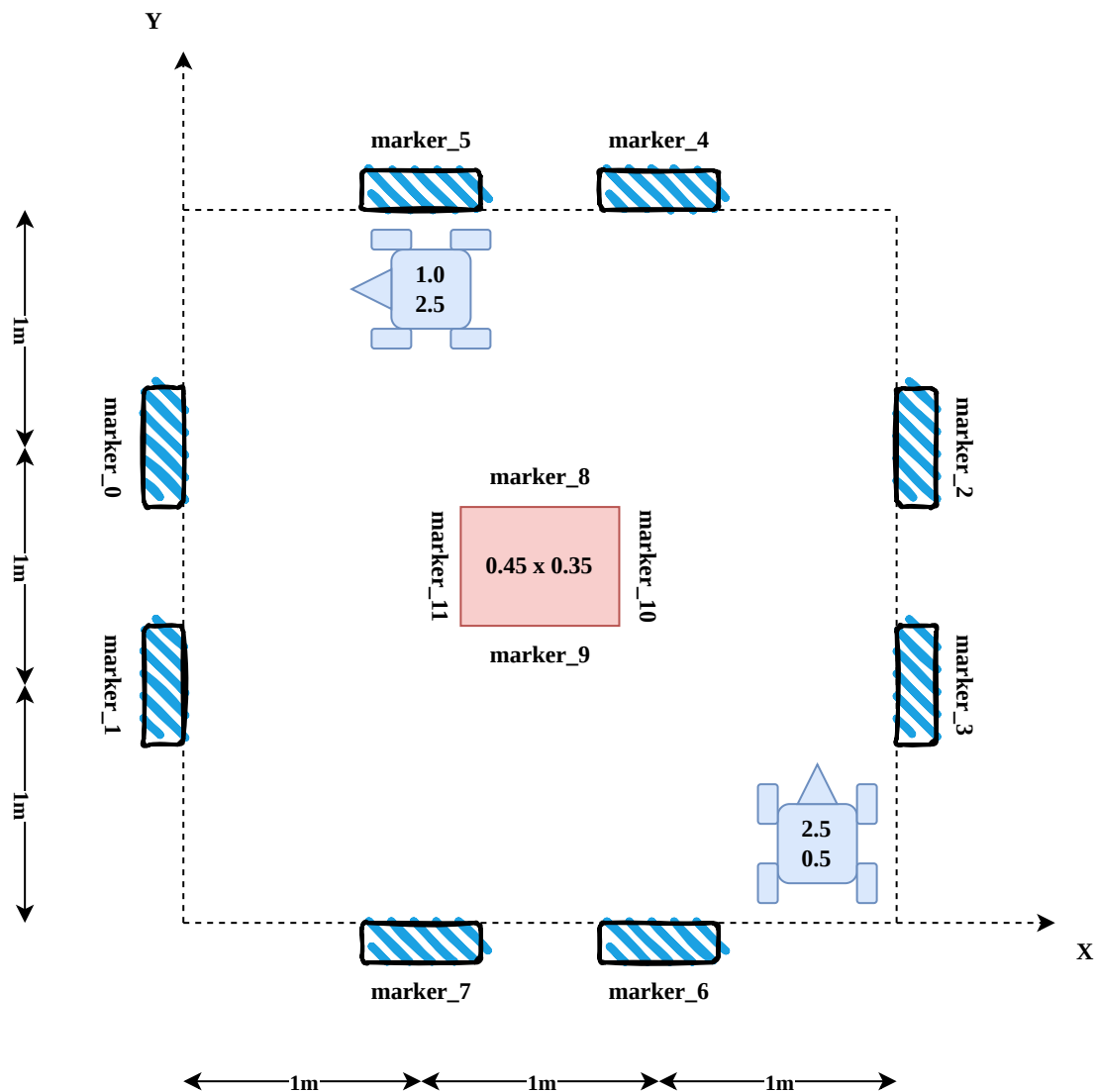
The only differences we made on our HW2's solution are: 1) using offline path search algorithms to generate the waypoints.txt, 2) `static_tf_broadcaster` broadcasting new locations of landmarks.

Rectangles represent ROS nodes, blue arrows represent topics and arrows represent subscribers.



## Environment Setup

The outer boundary of the environment is set to be 3m by 3m, with two markers on spread out on evenly each side. The origin is defined to the bottom left corner. The obstacle, with size 0.45m by 0.35m is placed at the center. Four additional April tags are added to each side of the obstacle.

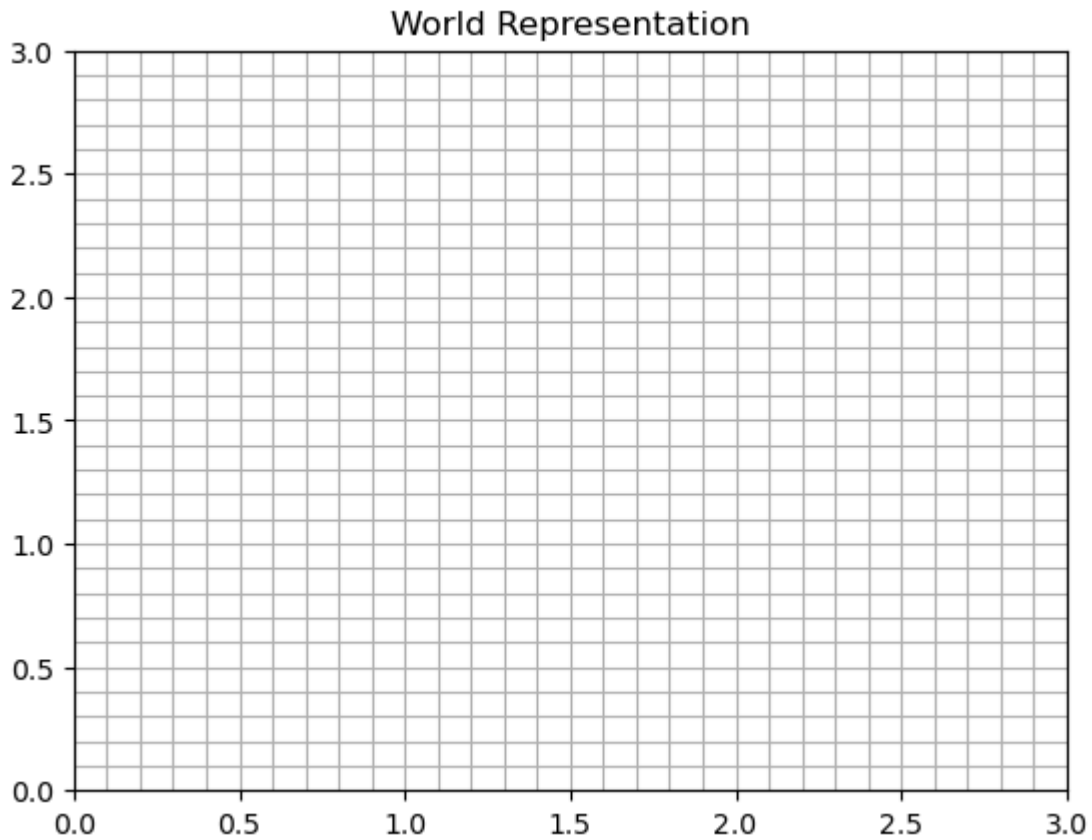


# Maximum Safety

## World Representation

We divided the 3m by 3m space into a grid where each square was 1cm x 1cm (Ignore the image unit they are for representative purposes only).

Each grid cell is connected to the 8 grid cell next to it (including the diagonal ones).



## Planning Algorithm

We used A-star search algorithm for planning maximum safety path.

1. A-star is a good choice because it is an informed search algorithm, so converges very quickly. Moreover, we can plan for multiple scenarios with different heuristic functions.

2. The Heuristic we used:

$$h(x_i, y_i) = \text{Euclidean distance to the goal} - \text{Perpendicular Distance from Obstacle}$$

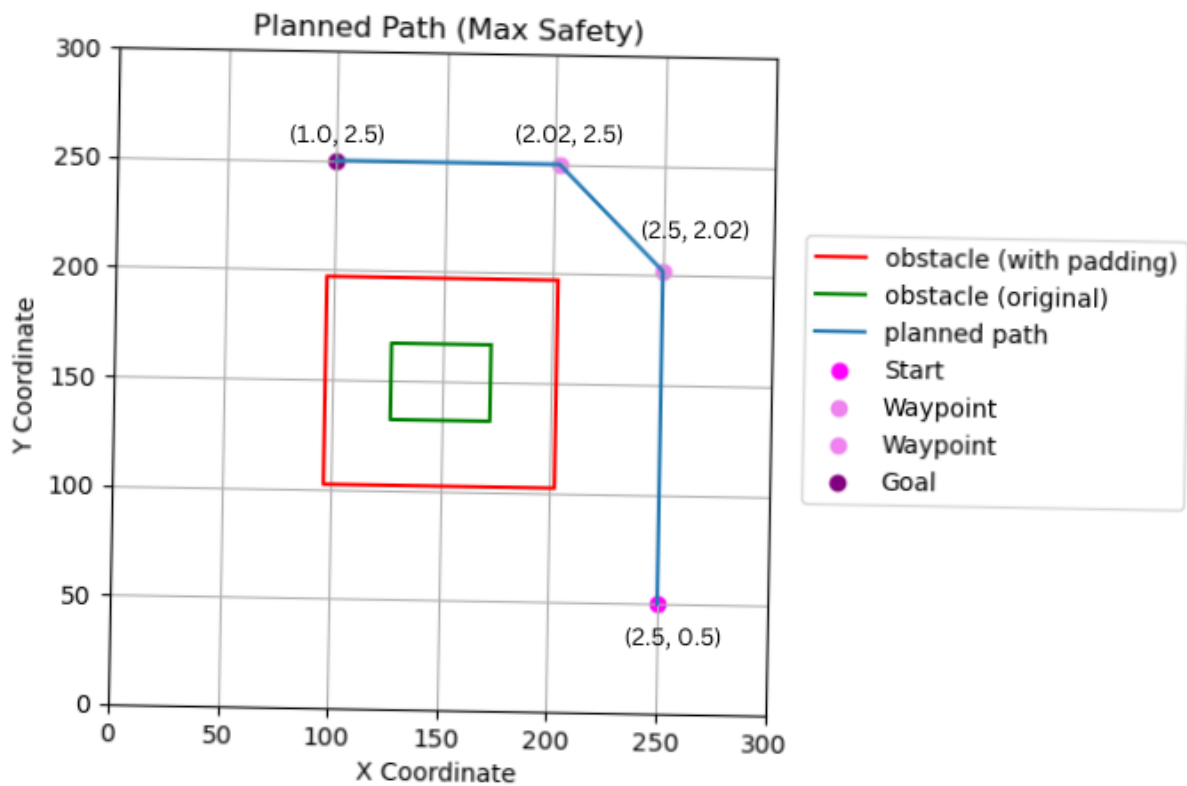
3. The cost function we used was: Euclidean distance between centers of two grid cells.

3. Our choice of heuristic is based on the fact that for Optimality of the path returned by A-star we need our heuristic to be admissible and if subtract the perpendicular distance from obstacle from the Euclidean distance to the goal, our heuristic will never overestimate the cost to goal.

## Planning Algorithm Implementation

We implemented A-star search using heapq module in python.

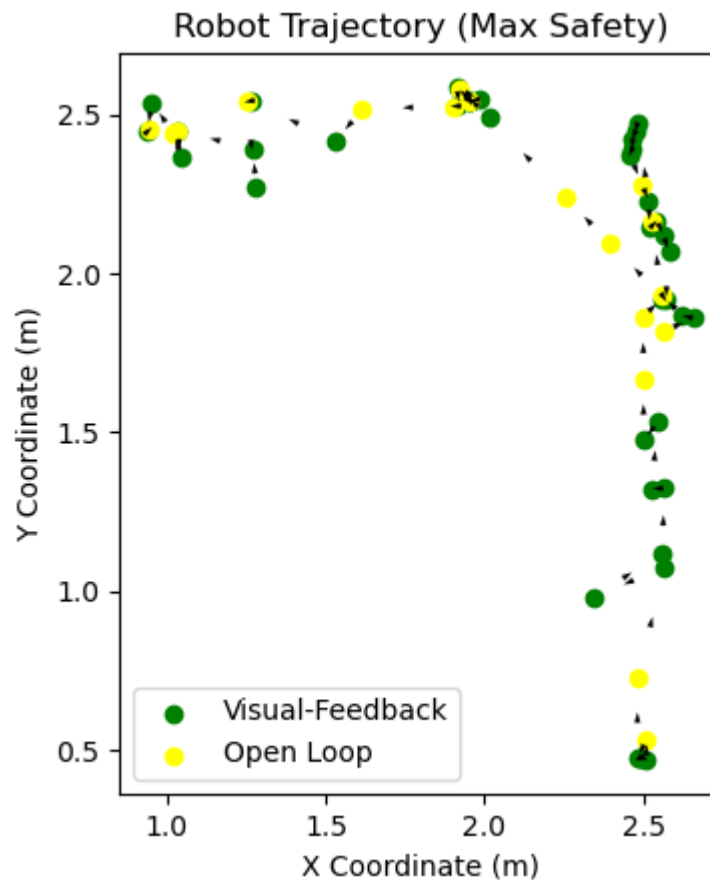
## Planned Path



1. Since in our representation we use 1cm x 1cm squares, the path planned by our algorithm contains very small steps which are not possible to execute. We subsampled the path returned by A-star whenever the robot was supposed to change direction and used those as the waypoints.
2. We added padding to our obstacle to ensure that the planning algorithm never returns any points which are too close to the obstacle.


## Actual Trajectory

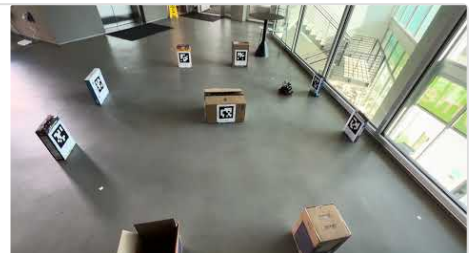




## Results

CSE276\_HW4\_maxSafety

 <https://youtu.be/hnyloesmyr4>



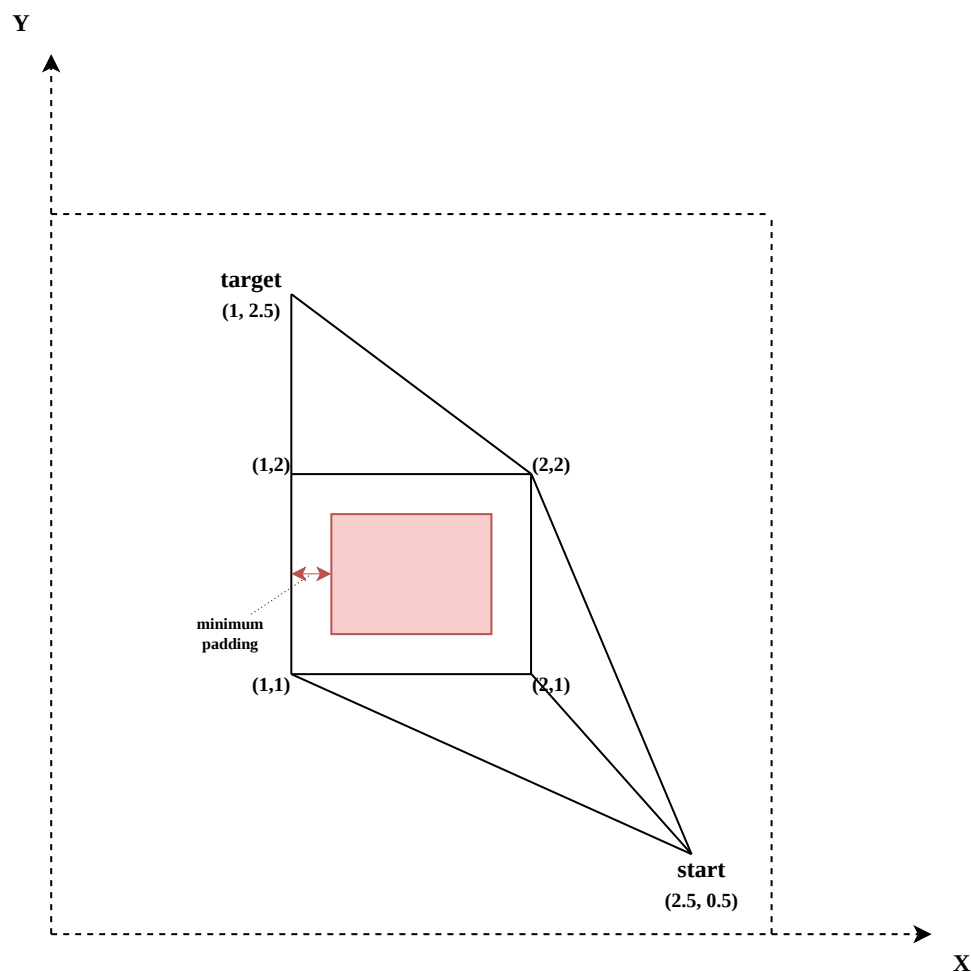
Our robot was able to execute the planned path with very high precision. It did make very large corrections but was able to reach the target within cm level accuracy using the visual feedback.

## Minimum Distance

### World representation

For minimum distance path search, we build a visibility graph to represent the world. The graph is shown below. We defined 6 vertices and the edge weight is defined to the Euclidean

distance between two ends. Note, We still kept a minimum padding around the obstacle to leave some margin for the robot to correct itself before hitting the obstacle.



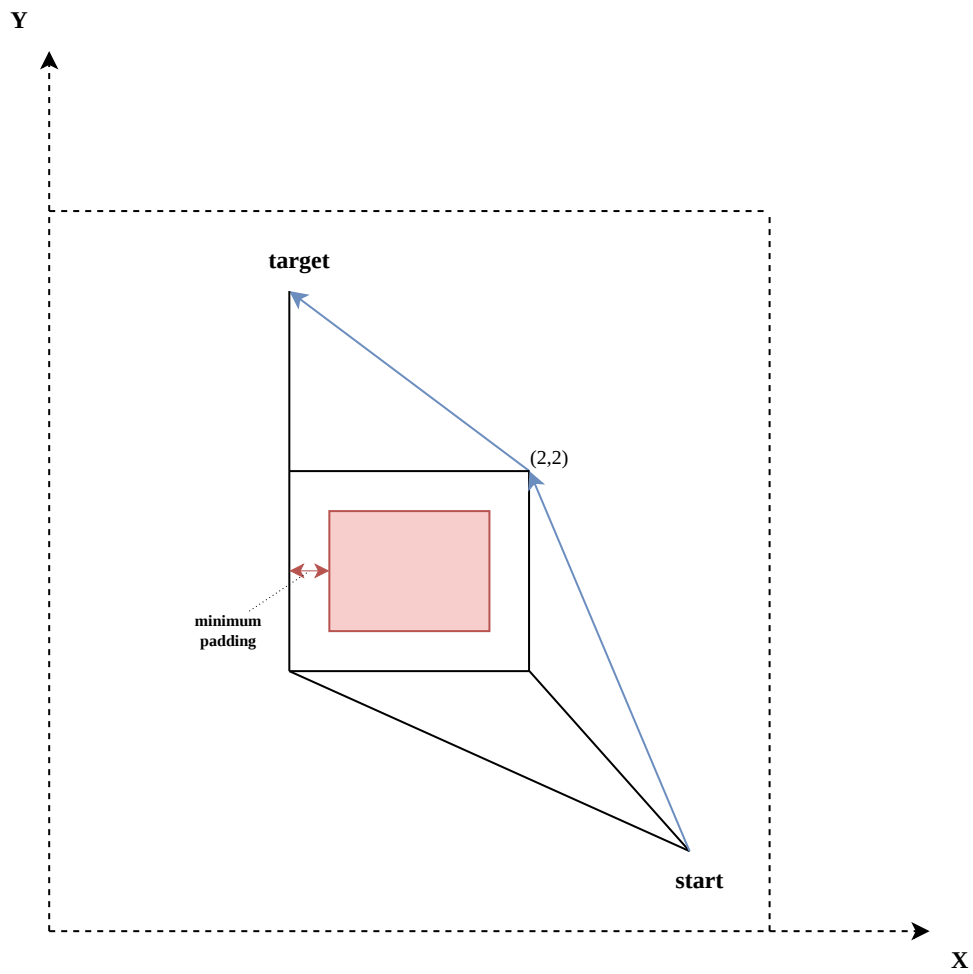
## Planning Algorithm

1. By using visibility graph we can encode information about the environment very efficiently
2. Our robot is capable of movements in every direction, capturing that in a grid based representation using A-star was not feasible.
3. The path returned by this algorithm is much shorter then what we could generate using A-star because this representation utilizes the fact that robot can move at any random angle and not just  $45n^\circ$ ,  $n \in [0, 8]$

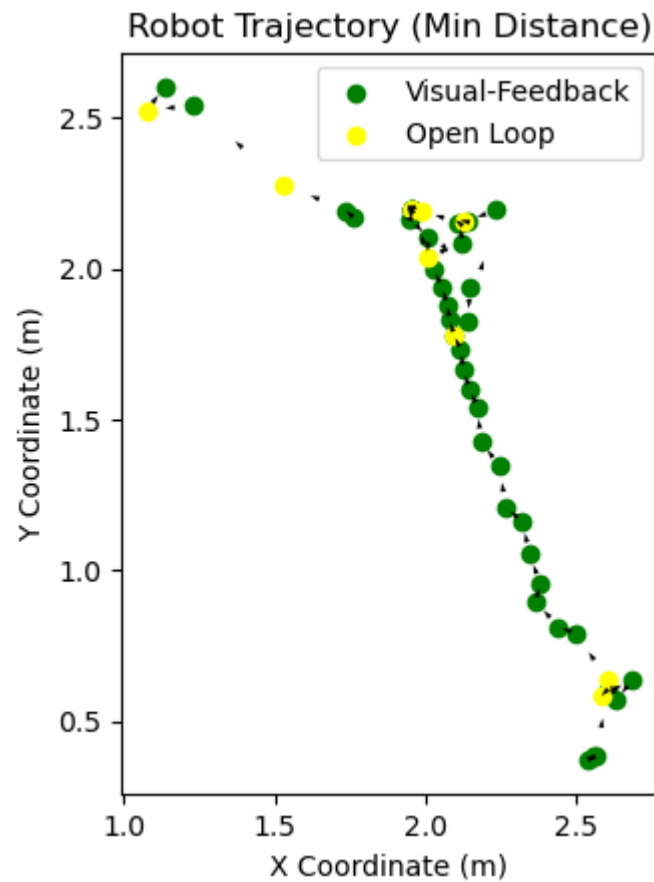
## Planning algorithm implementation

We used networkX library and its build-in bellman-ford algorithm to find the shortest path.

## Planned Path




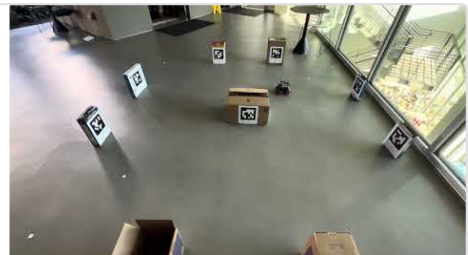
## Actual Trajectory



## Results

CSE276\_HW4\_minDist

 <https://youtu.be/-N2AQ9LTj64>



In our observations, the robot executed numerous adjustments as it navigated around the central waypoint. This behavior is likely attributed to the tilted view angle to the April tag on the rear side of the obstacle, which may have resulted in inconsistent and unreliable estimations of the robot's positional data.

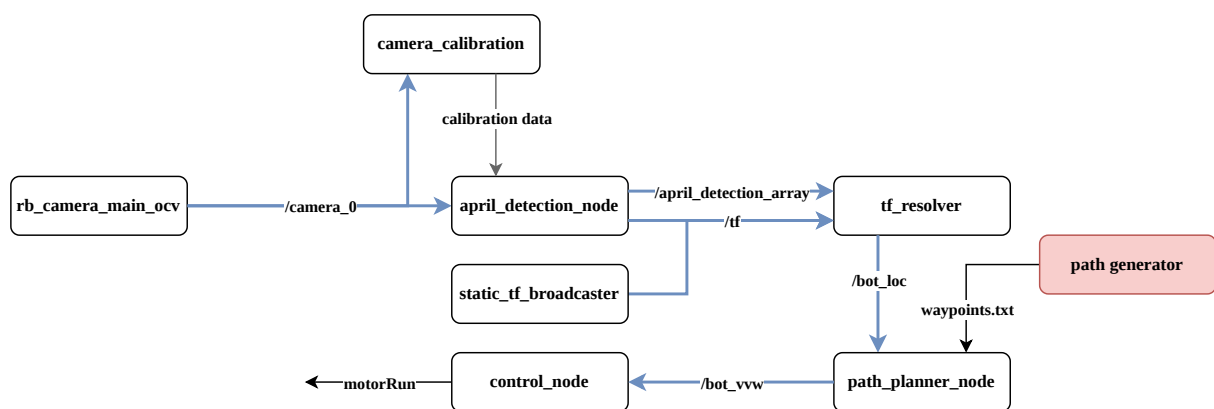
# CSE276A\_HW5

Authors: Weixiao Zhan (A59023453), Somanshu Singla (A59023795)

## Control Architecture

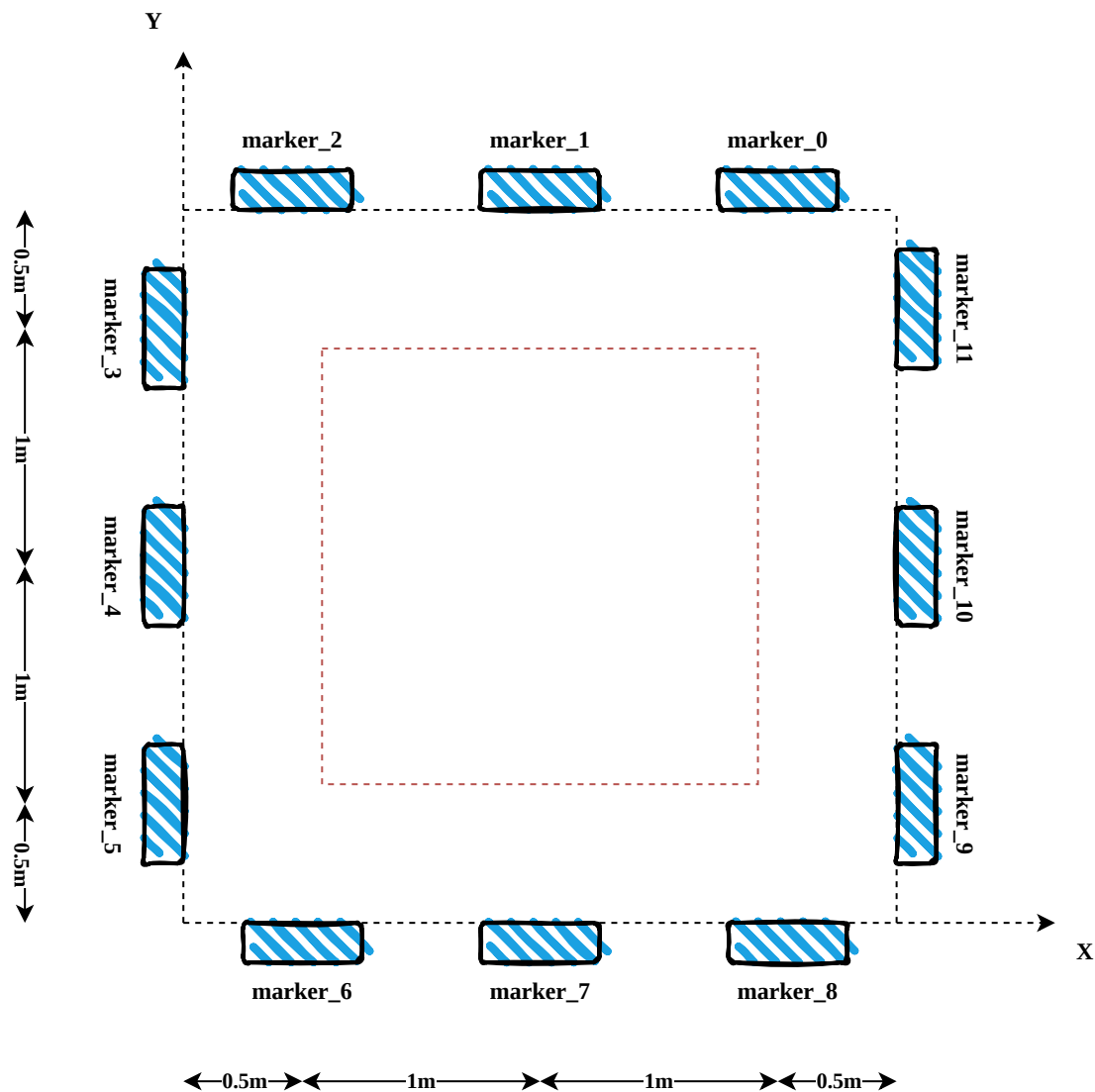
This diagram outlines our closed-loop control system. Rectangles represent ROS nodes, blue arrows represent topics and arrows represent subscribers.

The only differences we made on our HW4's solution are: 1) using coverage algorithm in path generator to generate the waypoints.txt, 2) `static_tf_broadcaster` broadcasting new locations of landmarks.



## Environment Setup

The outer boundary of the environment is set to be 3m by 3m, with three markers on spread out on each side. The origin is defined to the bottom left corner. The area to be covered is marked by the red dot line.



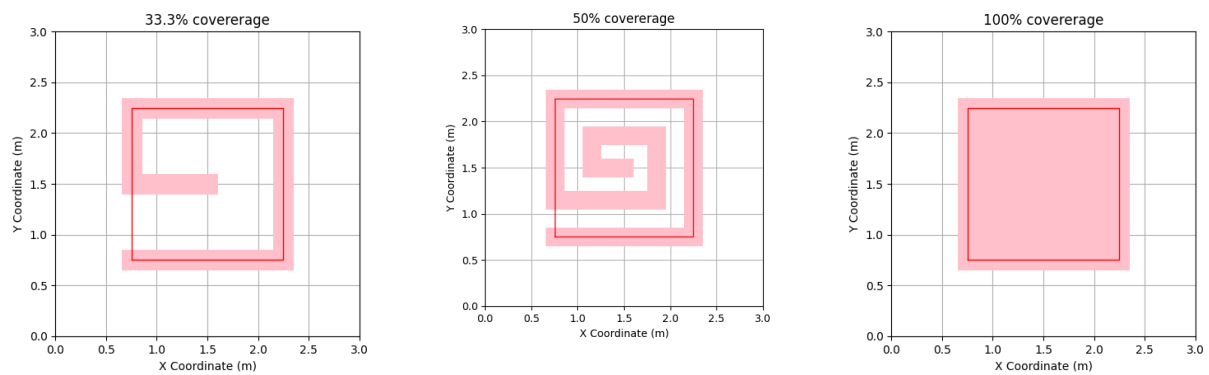
## Behavior to provide coverage/avoidance

For our case since the environment and landmarks are fixed we could make use of an offline planning algorithm which designs waypoints in a way that cover the entire area. For avoidance since we already knew location of all landmarks we added some padding near the landmarks and ensured the bot never enters that area for avoidance.

For a more general case, we can use SLAM and then add some padding based on the SLAM's state to establish avoidance.

## Coverage Algorithm

We developed the path generator to generate path with different coverage targets. Graphs below show a few examples.



The 3x3 outer box is our complete environment area. The inner red 1.5x1.5 box is the area robot trying to cover. Pink path is the planned trajectory. Our robot would start from bottom left corner and work its way to the middle. The trajectory line width authentically reflects the width of robot in the map. The space between trajectory, called stride  $s$ , is adjustable to achieve different coverage target.

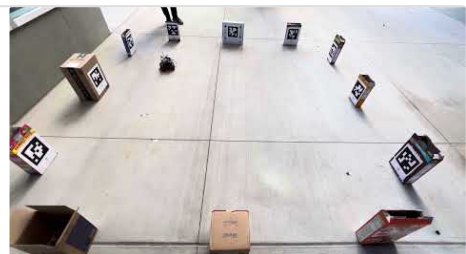
For an ideal robot, with stride set to the same as the robot width, the trajectory achieved 100% coverage. However, For a robot like ours, we can chose a stride less than the width of the robot ( $<0.2\text{m}$ ) to have some redundancies/overlap in coverage. This ensures even after making errors and correcting for them the bot covers the complete area.

Lastly, we kept a margin between the inner to-cover area and outer edge of the environment, where the April tags locate on, to give robot some space to adjust before running into the landmarks and to prevent too close April tag causing bad distance estimation.

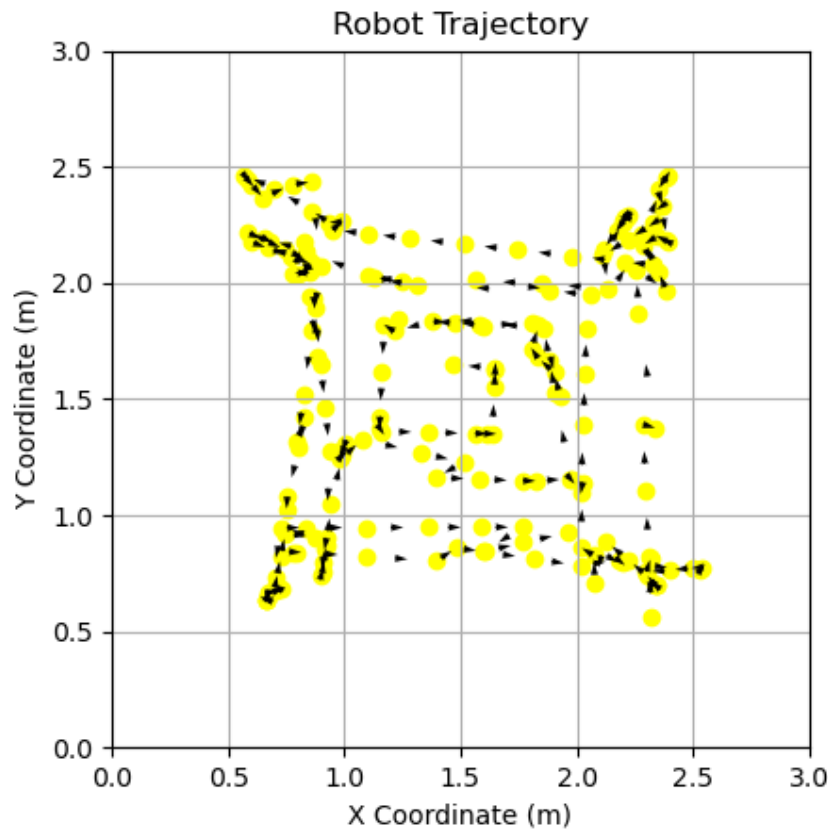
## Results

CSE276\_HW5

 <https://youtu.be/fRGVf9gam4E>



Plot of the executed path:



## Coverage Guarantee

### 1. Forward and Rotation

Despite our robot and PID support sliding, we found sliding motion is generally less accurate due to the friction. Thus the waypoints for coverage is always rotate towards next waypoint first then forward to next waypoint. In addition, at each waypoints, we let the robot stops for certain time of time to allow us observe and measure the error at each step. The stop is completely for debug purpose and can be easily removed with one line of code. But we will still include the stop time in our guarantee equations.

In equations, Total time = time doing rotation + time doing forward + time stopping

$$t = t_{\theta} + t_d + (\# \text{ of waypoints})t_0$$

Nevertheless, sliding motion still happens when robot uses its visual feed back to correct it's location. And we will bound such motion later in our models.

### 2. Upper bound of ideal path

Let the robot width is  $W$ , stride distance is  $s$ ,

$$\text{then coverage } c := \frac{W}{s} \Leftrightarrow s = \frac{W}{c}$$



Let the to-cover area is a  $e \times e$  square

we have:

$$m = \# \text{ of squares} = \max \left\{ \left\lfloor \frac{e}{2s} \right\rfloor, 1 \right\}$$
$$n = \# \text{ of waypoints} = 8m$$

and the ideal length and rotation is

$$d_{\min} = 4e + 4(e - 2s) + \dots$$
$$= m(e - (m - 1)s)$$
$$\theta_{\min} = m \cdot 2\pi$$

### 3. Robot error

Our robot can't execute the ideal path perfectly, thus we will assume that the robot will execute the forward and/or rotation motion with in  $\lambda$  times of the ideal motion ( $\lambda > 1$ ):

$$d_{\min} < d < (d_{\max} = \lambda_d d_{\min})$$
$$\theta_{\min} < \theta < (\theta_{\max} = \lambda_{\theta} \theta_{\min})$$

We would consider the sliding motion in actual execution as part of the  $d$ .

### 4. Max and min velocities

In our PID control (rest in path planner node), we used to minimum and maximum velocity and rotation velocity,  $v_{\min}$ ,  $v_{\max}$ ,  $\omega_{\min}$ ,  $\omega_{\max}$ , so that we don't get stuck when raw PID desired very little velocities and don't overwhelm the robot when traveling long distance.

thus minimum time is  $t_{\min} = \frac{d_{\min}}{v_{\max}} + \frac{\theta_{\min}}{\omega_{\max}}$

and maximum time is  $t_{\max} = \frac{d_{\max}}{v_{\min}} + \frac{\theta_{\max}}{\omega_{\min}}$

### 5. Supply in numbers

In our experiment (shown in video):

target coverage:  $c = 110\%$ ,

environment side:  $e = 1.5m$ ,

robot width:  $W = 0.2m$ ,

stop time:  $t_0 = 1s$

PID velocity:  $v_{\max} = 0.15m/s$ ,  $v_{\min} = 0.09m/s$

PID rotation velocity:  $\omega_{\max} = 2.449rad/s$ ,  $\omega_{\min} = 0.7347rad/s$

Based on our experience, we are confident that the robot will be able complete the motion within 2 times of the correction, i.e.  $\lambda_d = \lambda_\theta = 2$

supply the numbers to the equations above:

$$t_{\min} = 67.721 \text{ sec}$$

$$t_{\max} = 185.265 \text{ sec}$$

Note: there are total 32 seconds of stop time in both numbers.

## 6. Summary

We have build a proofing model above, which yield confident lower and upper bound of the time the robot needs to cover a certain area with desired coverage rate. In our experiment (shown in video), our robot finished the path in approximately 176 seconds, which matches our performance guarantee.