ME780 Assignment 2

Stan Brown & Chris Choi

1 Motion Model of a Bicycle

Define the bicycle motion model for an Ackermann steered robot as shown in the configuration below(and presented in the motion modelling lecture notes).

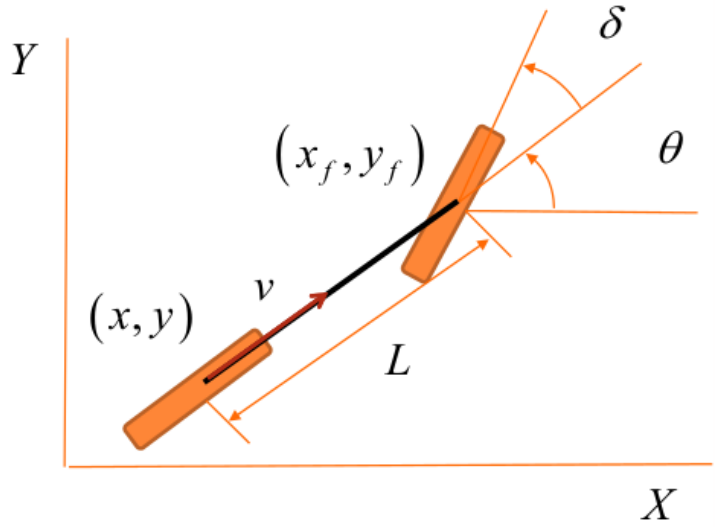


Figure 1: Bicycle in World Frame

From the lecture notes, the motion of front wheel can be written as follows:

$$x_f = x + L\cos(\theta) \tag{1}$$

$$y_f = y + L\sin(\theta) \tag{2}$$

$$\begin{bmatrix} x_{f,t} \\ y_{f,t} \end{bmatrix} = \begin{bmatrix} x_{f,t-1} + v_{f,t}\cos(\theta_{f,t-1} + \delta_t)dt \\ y_{f,t-1} + v_{f,t}\sin(\theta_{f,t-1} + \delta_t)dt \end{bmatrix} \tag{3}$$

Similarly the motion of the rear wheel is as follows:

$$\begin{bmatrix} x_{r,t} \\ y_{r,t} \end{bmatrix} = \begin{bmatrix} x_{r,t-1} + v_{r,t}\cos(\theta_{r,t-1})dt \\ y_{r,t-1} + v_{r,t}\sin(\theta_{r,t-1})dt \end{bmatrix} \tag{4}$$

These two equations can be linked and simplified by using the Instantaneous Center of Rotation (ICR) to combine equations of motion for the front and rear wheels as follows.

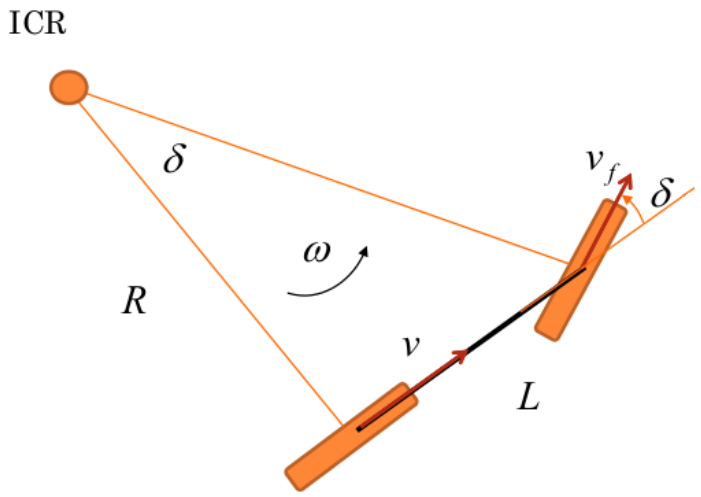


Figure 2: Simplifying the Bicycle Model with ICR

From Fig 2 the relationship between the steering angle δ relative to the bicycle's heading can be written as:

$$\tan(\delta) = \frac{L}{R} \tag{5}$$

Moreover, we know that the angular velocity ω from the point ICR of both the front and rear wheel is:

$$\omega = \frac{v\tan(\delta)}{L} = \frac{v}{R} \tag{6}$$

Now that we have the angular velocity of the bicycle over time, the bicycle's change in heading over time is simply:

$$\theta_t = \theta_{t-1} + \frac{v_t\tan(\delta_t)}{L}dt \tag{7}$$

For the complete motion model of the bicycle we have:

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} + v_t \cos(\theta_{t-1})dt \\ y_{t-1} + v_t \sin(\theta_{t-1})dt \\ \theta_{t-1} + \dfrac{v_t \tan(\delta_t)}{L}dt \end{bmatrix} \tag{8}$$

Using Equation 8, the bicycle motion model was simulated in Matlab over a period of 20 seconds with the following parameters:

- Simulation time step dt: 0.1s
- Velocity v: 3ms^{-1}
- Length of bicycle L: 0.3m
- Steering angle δ: $10 - t$ degrees
- Steering angle limit: ± 30 degrees
- Standard deviation on x and y: 0.02m
- Standard deviation on θ: 1 degree

The resulting trajectory from the simulation inputs is shown in Fig 3. Based on the simulation inputs the the observed trajectory appears to be valid as the heading starts at 10 degrees and then decreases at a rate of -1 degree per second. This means during the first half of the run the steering angle is positive, causing it to steer to the left, straighten out for a portion and then begin starting to turn to the right as the steering angle becomes negative. These motions result in the "S" shape that is depicted in Fig 3

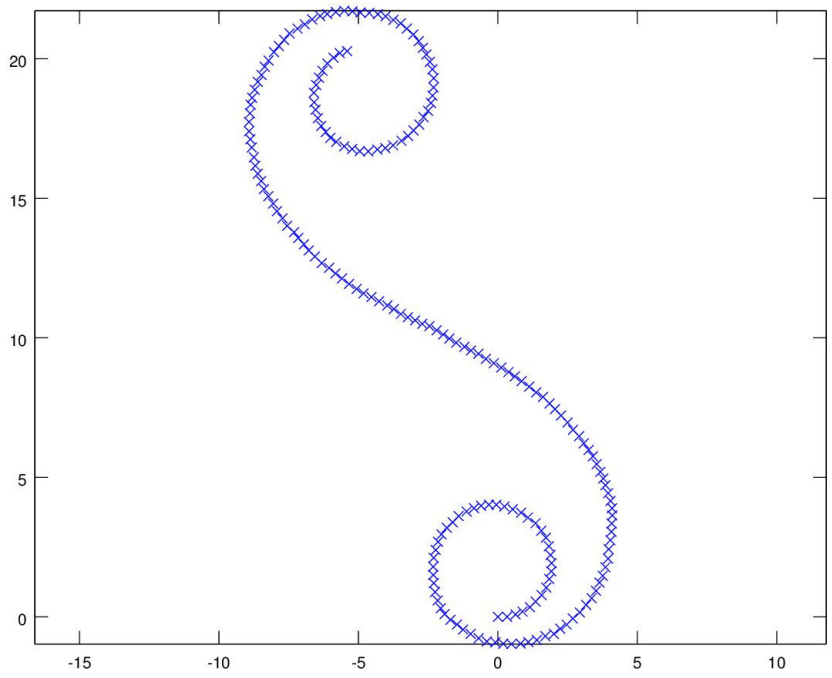


Figure 3: Simulating the Bicycle Model for 20s starting from (0, 0)

2 Carrot Controller for Bicycle

Develop a carrot following controller. The way this controller works is you define a carrot, or point on the desired trajectory line a fixed distance, r, ahead of the closest point on the line to the current robot position.

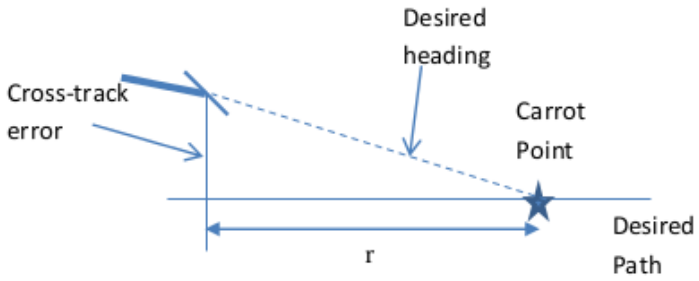


Figure 4: Carrot Controller

2.1 Closest Point on the Trajectory Line to the Robot

The first step in implementing the the carrot controller for the outlined problem is to first determine a way to calculate the correct location of the carrot point given the robot's current position. To do this we can use the projection equation between two vectors.

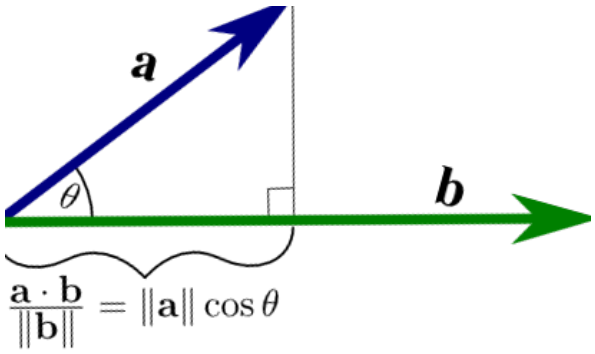


Figure 5: Dot Product Projection)

Let the vector a denote the vector from the start of the trajectory line to the position of the robot, and vector b denote the start and end of the trajectory line. To calculate the closest point on vector b to the robot position one simply has to project vector a onto vector b, this can be calculated via:

$$\text{proj}_{\vec{a}\vec{b}} = \frac{\vec{a}\cdot\vec{b}}{|\vec{a}|}\frac{\vec{a}}{|\vec{a}|} = \frac{\vec{a}\cdot\vec{b}}{|\vec{a}|^2}\vec{a} \tag{9}$$

Next we need to determine where to place the carrot point. Having already found the closest point on

the trajectory line p_{closest}, the carrot is simply calculated by adding the look ahead distance r times the unit vector of the trajectory line b to the value of p_{closest}.

$$\text{carrot} = p_{\text{closest}} + r\frac{\vec{b}}{|\vec{b}|} \tag{10}$$

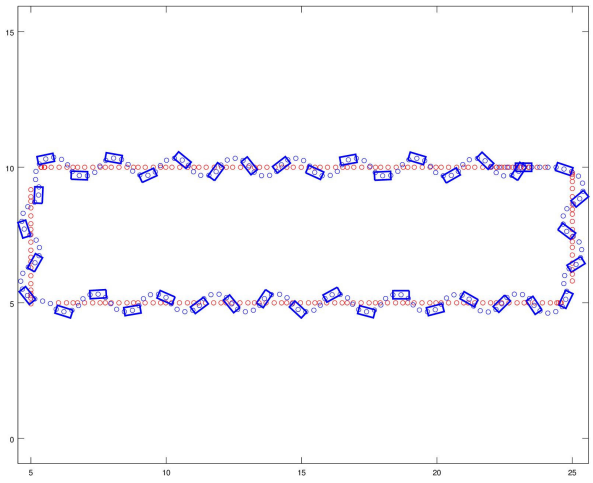
2.2 Carrot Controller Implementation

We implemented a proportional (P) carrot controller and traced a rectangle of $20m$ by $5m$ at $3ms^{-1}$. This is the resulting motion of the bicycle (See Fig 6), we used a proportional gain $k_p = 2$ and tested look ahead distances r at 2, 0.2 and 5.

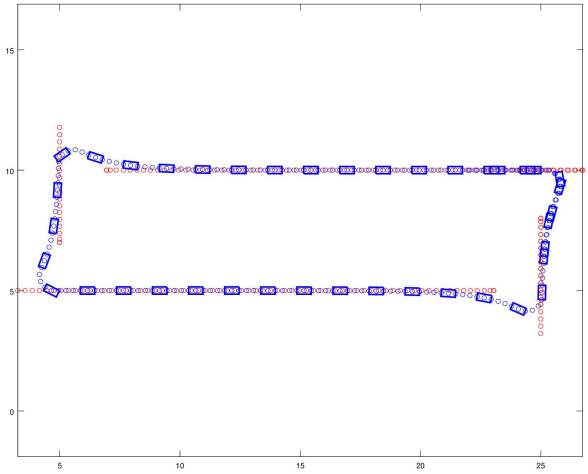
From Fig 6 we can see that different look ahead distance r greatly impact the way the robot traces a rectangle. If the look ahead distance is too low (i.e. Fig 6a where $r = 0.2$), the carrot is always very small distance ahead to the robot which in turn causes the angle between the carrot's location and the robot to be relatively large. This combined with the coarse time step of $dt = 0.1s$ causes the robot to overshoot and cross over the desired path line at each time step, inducing a large error in the heading which then causes it to overshoot again, leading to the oscillatory motion observed in Fig 6a. This problem can be avoided if one was to use a PD controller, reduce the P gains or decrease the time step.

When the carrot was set to a look ahead distance of $r = 5m$ we found that robot had trouble hitting way points and we had to increase the way point reached threshold from $0.2m$ to $0.5m$ to compensate for this problem. The way point reached threshold represents a virtual circle that the robot must enter in order for the carrot controller to consider the way point has been reached. This difficulty of hitting way points, along with larger overshoots compared to the carrot distances were $r = 0.2$ and $r = 2m$ are caused by the large distance between the carrot and the robot which reduces the calculated angle between the robot and the carrot at each time step, making it less sensitive to heading errors and in turn dampening the turning rate of the robot.

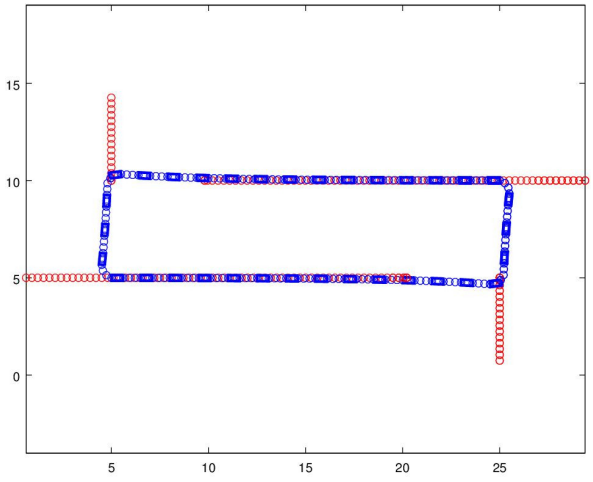
Overall we found that a lookahead distance of $r = 2m$ provided the best results for the time steps and proportional gains provided. We were able to use a detection radius of $0.2m$ at each way point to detect if the robot had reached the way point goal. Moreover, after passing a way point and slightly overshooting, the robot's trajectory matched the on the next path segment within approximately $5m$ of last way point compared to the runs where $r = 5m$ in which case the robot took over $10m$ to pass directly on the path segments.



(a) $k_p = 2$, $r = 0.2$



(b) $k_p = 2$ and $r = 2$



(c) $k_p = 2$, $r = 5$

Figure 6: Bicycle motion using carrot controller starting at (23, 10)

3 IGVC Planner

Using the map provided with the homework (IGVCmap.mand IGVCmap.jpg), identify a planning strategy that can be used in conjunction with the carrot planner of part 2)to navigate the Intelligent Ground Vehicle Competition auto-navigation course.

The provided map of the Intelligent Ground Vehicle Competition in the form of a 926×716 element grid, where each cell represents a 10 cm X 10 cm area of the course, resulting in a 92.6 m X 71.6 m environment.

Our goal is to implement a planner and present a combined motion planning solution that finds a straight-line path through the environment, then uses the carrot controller to provide the inputs required to drive the robot to the start and end locations which are ocated at ($x = 4.0m$, $y = 0.5m$, $\theta = 3.14159rad$) and ($x = 5.0m$, $y = 1.0m$, $\theta =$ not defined) respectively.

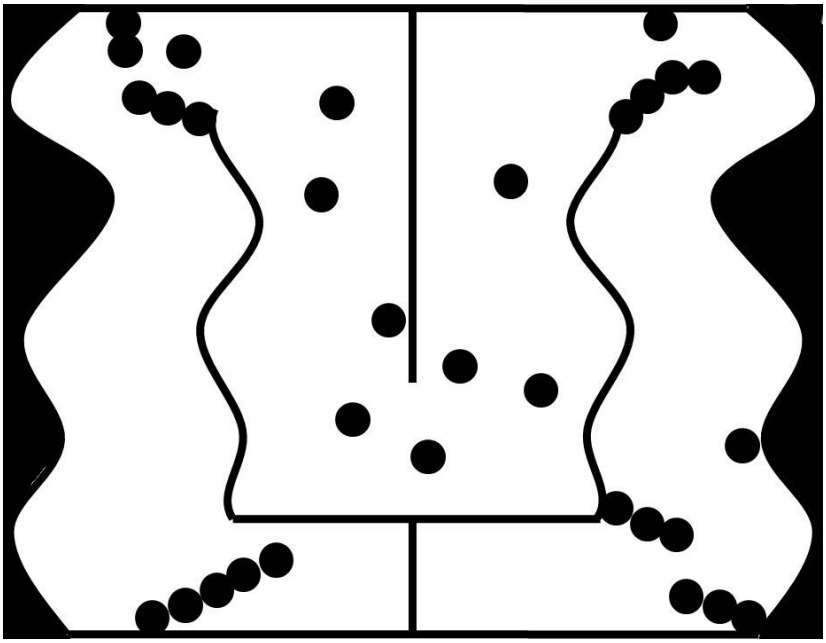


Figure 7: IGVC Map

3.1 Wavefront Planner

Since the map given represents an occupancy grid of free space and obstacles, our initial approach to the planning portion was to apply the wavefront a path that finds an optional path between start to goal point. Fig 8 highlights the results of the wavefront planner and highlights the resulting path found by gradient decent.

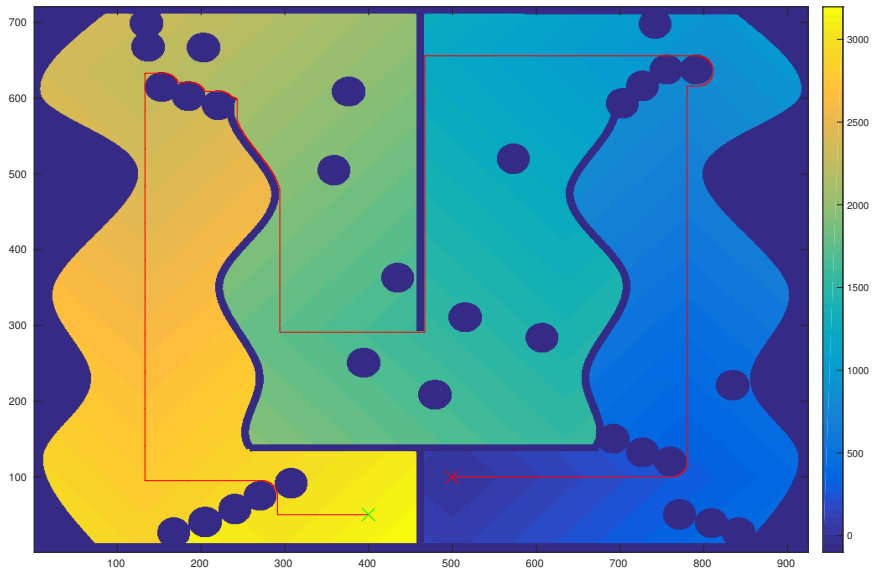


Figure 8: Wavefront IGVC Map

While the wavefront method successfully found an optimal solution on the grid based environment, it is far from optimal then a path generated in the continuous space. This problem stems from the fact that wavefront method is a grid based method based on a breadth first search that starts and the end goal and calculates a distance between the end goal and all points in the map. Because the distance is based on the number of grids between the start and end point the distance calculated actually reflects the Manhattan Distance between the start and end goals rather than the Euclidean Distance. This means that driving at a 45^o angle has the same distance as two paths that first travel horizontal and then upward. Furthermore the robots trajectory is restricted to the exploration directions used to generate the wavefront which in this case were a set 4 moves, mainly move up, move right, move left and move down.

Additional problems are realized when one takes into account that the fact that robot is likely larger than the grid cells which makes collision avoidance and detection difficult. In many cases the grid cells are assumed to be the size of the robot but this has the additional drawback of causing obstacles to become very course and poorly represented. In Fig 8 it can be noted that the wavefront planner slips between a very small gap between two pylons on the course, which unless the robot is 10cm by 10cm, would be an impossible path for the robot to follow. Additionally the robot also tends to hug obstacles very closely which means that any errors in possession or heading are likely to result in collisions.

Lastly, for this large map, the run time was over 1 minute without any collision checking procedures and therefore this approach was abandoned in favour of a Probabilistic Roadmap Planner (PRM).

3.2 Probabilistic Roadmap Planner

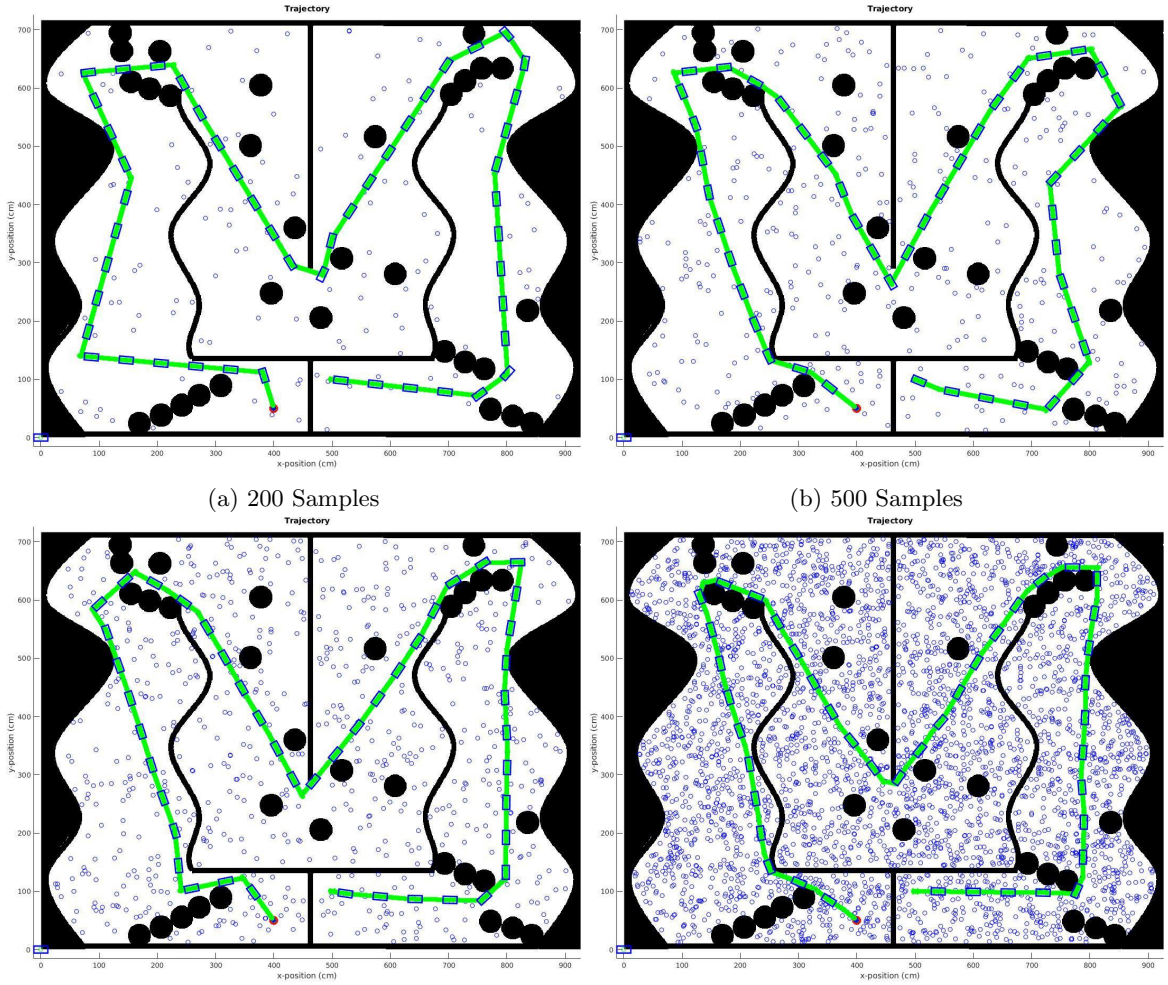
In our next approach we used the Probabilistic Roadmap (PRM) planner, purely out of curiosity and it seemed like an intuitive alternative to wavefront planner.

For the PRM implementation on a grid based environment, we had to make modifications to the provided PRM example, which assumes the obstacles are in the form of set of polygon points. This allows the PRM planner implemented in the ME 597 example code to exploit Matlab's *inpolygon* function to detect whether a sample point located within a polygon.

However the map given in this assignment is represented an occupancy grid instead of set of polygons, and therefore the collision checking portion of the PRM code to be replaced by a ray tracing algorithm. In our implementation we utilized a modified version of Bresenham's line algorithm, to find create collision free edges between sample points.

We implemented a PRM planner and then applied the carrot controller outlined in Question 2 to the resulting paths which are shown in Fig 9 which also highlights the effect of using varying sample sizes in the PRM. From testing it appears that a sample size of 200 will yield a valid solution in about 50 percent of the time, a sample size of 500 will yield a solution is 90 percent of cases and a sample sizes with more than 1000 sample points always resulted in a solution.

Higher sample sizes also tended to lead to smoother and more optimal paths which can be observed by comparing the 200 sample case to the 5000 sample case highlighted in Fig 10. The run time of the PRM when using 5000 points less than 3 seconds which is a significant improvement over the wavefront planner which took over a minute to find a solution.



(a) 200 Samples

(b) 500 Samples

(c) 1000 Samples

(d) 5000 Samples

Figure 9: PRM Planner and Carrot Controller

3.2.1 PRM Planner with Sample Points Padded

While the PRM yielded a complete path between teh start and end goals, it was noted that at some locations along the path the robot will pass extremely close to obstacles due to the PRM cutting the corners extremely tightly. This behaviour is due to a lack of particles in locations near choke points. An example of the PRM cutting a corner to tightly is highlighted in Fig 9.

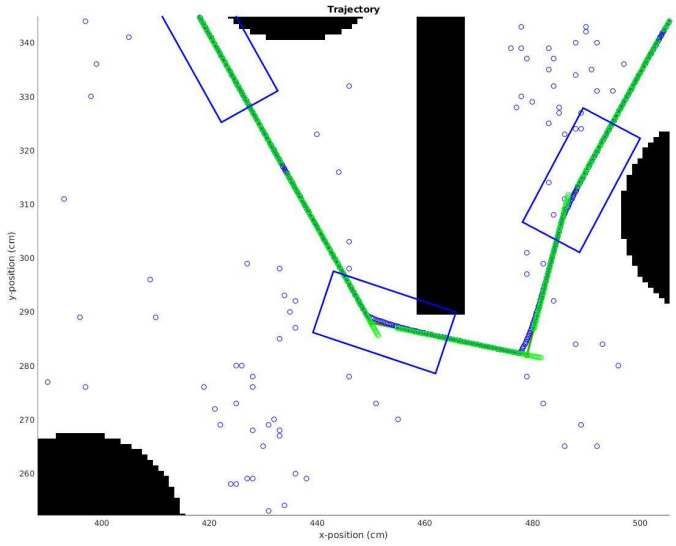


Figure 10: Collision Example

We attempted to address this issue by adjusting the way that sample points were created in the PRM planner by checking for any obstacles within that are within 0.5 m circle of each sample point. If an obstacle is detected with 0.5 m of a point, the point is deleted and a set of 5 additional points are created and randomly distributed within a circle centred at the deleted sample point with a radius of 1.5 m. These points are again checked for proximity to obstacles and pruned in a the same manner described as above.

This approach resulted in areas near walls and obstacles to be sampled much more heavily compared to open spaces and also resulted in smoother paths near obstacles and are highlighted in Fig 11

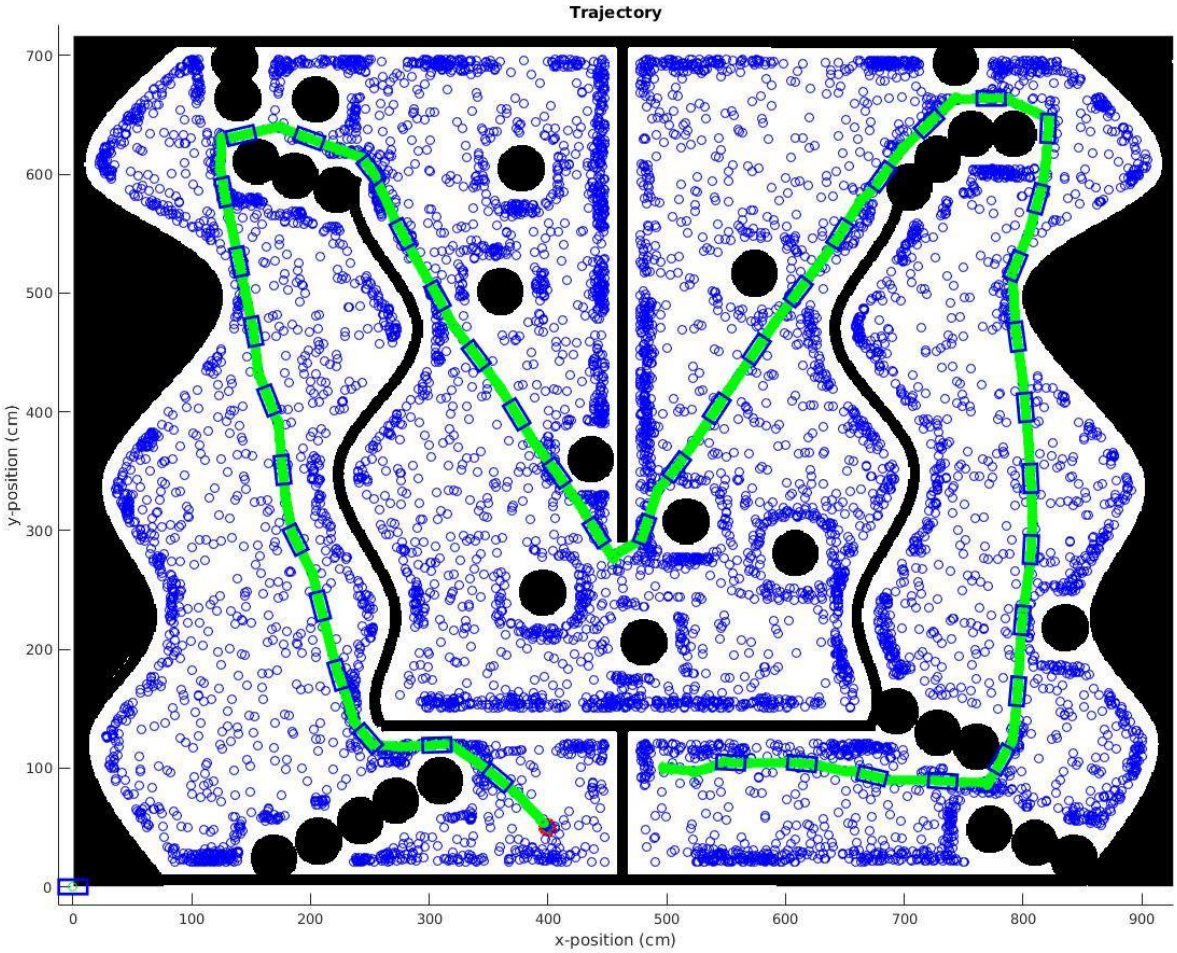


Figure 11: PRM Resampling with 1000 points and a collision check raduis of 0.5 m

4 Limitations and Future Improvements to the PRM Planner

Previously in Section 3 explored both wavefront and PRM path planning methods, both methods alone are not satisfactory in planning a path for the bicycle through the IGVC course, we will discuss three limitations below along with possible solutions:

- **Path is not smooth**: The paths generated by both wavefront and PRM do not account motion model of the bicycle. Some portions of the planned paths have very sharp edges or corners, particularly in locations near very sharp obstacles such as the wall located in the middle of the course. Such an angle is to sharp for the controller and therefore results in a non optimal turn around the obstacle that could be avoided if the planner could take into account the robots turning radius.
- **No collision detection**: None of the explored methods explicitly perform collision detection of a robot with an actual size. We approximated some collision checking using our re-sampling approach which did eliminate a significant portion of collisions but not all of them. One possible

way to perform collision checking in the PRM approach would be to modify the A* search algorithm to check for collisions with obstacles when selecting an edge to transverse. One option for collision checking would be to check parallel line segments located at a distance 0.5 robot widths above and below the line segment formed by a selected edge for collision. If one of these lines was to found to interest an obstacle the edge should not be added to the final path.

- **Map is assumed to be known, accurate and static**: Probably the largest limitation to our approach is the assumption that the map is accurate and static. A local planner such as trajectory roll-out combined with some kind of range sensor, such as lidar, would allow the robot to determine paths that avoid uncounted for obstacles. In the case of non-static obstacles, such as other robots (since it is a race) a real time object tracker could be implemented and used to make predictions as to where moving obstacles will be in future which could then be used by the local planner to come up with a collision free trajectory.