UNIVERSITY OF WATERLOO
Department of Systems Design Engineering

ME 597: Assignment 1

prepared by

Sarah Elliott
Leigh Pauls
November 1, 2013

1.0 Bicycle Model

For the bicycle model, we basically need to find the x_t and y_t for each time step. We add Gaussian noise to the position with $\sigma_{xy} = 0.02\ m$.

$$x_t = x_{t-1} + d_t * cos(\mu_{heading}) + N(0, \sigma_{xy}^2) \tag{1}$$

$$y_t = y_{t-1} + d_t * sin(\mu_{heading}) + N(0, \sigma_{xy}^2) \tag{2}$$

Within that we get d_t, which is the distance traveled since the last time step by multiplying the speed $v = 3\ m/s$ by the time step $dt = 0.1\ s$:

$$d_t = v * dt \tag{3}$$

We now need to get the heading. The heading, h_t, is shown below. We add Guassian noise to the angle calculated with $\sigma_\theta = 1°$.

$$h_t = h_{t-1} + dheading * dt + N(0, \sigma_\theta^2) * dt \tag{4}$$

The change in heading, *dheading*, for the time step is found by:

$$dheading = v * tan(\delta) \tag{5}$$

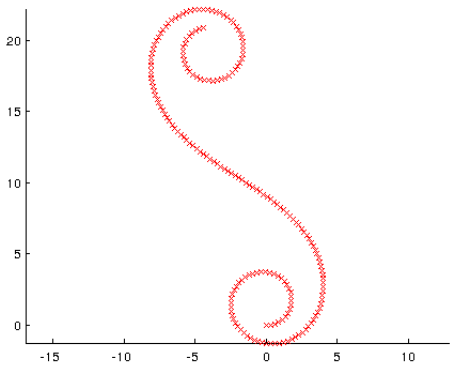
The average heading, $\mu_{heading}$, used in the first equation is:

$$\mu_{heading} = \frac{h_t + h_{t-1}}{2} \tag{6}$$

The above equations all rely on δ. The instructions give a max steering angle of +/-30°.

$$\delta = \begin{cases} 30° & t > 40 \\ 10 - t & t < 40 \end{cases}$$

The resulting path for a start position of (0,0) and heading of 0° is shown below:



2.0 Carrot Planner

The robot needs to track a rectangular path. The path P is defined by 4 points.

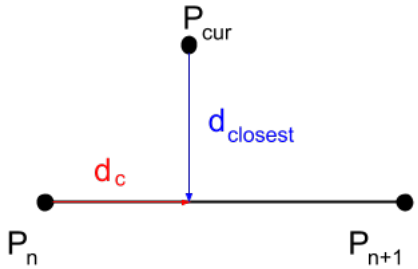
$$P = [\ p_1 p_2 p_3 p_4\ ] \tag{7}$$

To find the closest point on the path you need to know which part of the path is closest. We can imagine the path as having 4 segments, one for each side of the rectangle. Segment $S(p_n, p_{n+1})$ is the segment with start and end points at p_n and p_{n+1} respectively, where $n = 1, 2, 3$. For each S, we need to find the closest point on that segment. Then we will compare the closest points on each segment to see which of the segments is the closest overall.

Let p_{cur} be the current point. Let $r_{p_n/p_{n+1}}$ be the distance between those two points. This notation is extended to all other distances between points. The distance calculation is fairly trivial and not included in this report. Let's define the distance from the first waypoint, p_n, of the current segment to the closest point on the line as d_c.

$$d_c = r_{p_{cur}/p_n} * \frac{r^2_{p_{n+1}/p_{cur}} - r^2_{p_n/p_{cur}} - r^2_{p_{n+1}/p_n}}{-2 * r_{p_n/p_c ur} * r_{p_{n+1}/p_n}} \tag{8}$$

The figure below shows an example of d_c. Other point configurations are possible. To



find the closest point on the line we have to consider a number of cases. If the points are oriented as above, finding the closest point is fairly straight forward. You just add d_c to

the location of p_n. However, what if p_{cur} is closest to the top edge of the rectangle and moving counterclockwise? This would mean you need to subtract d_c from the location of p_n. Other cases include if d_c is negative or greater than r_{p_{n+1}/p_n}; for example if p_{cur} is outside the rectangle. To take all these situations into account:

$$p_{closest} = \begin{cases} p_n & d_c < 0 \\ p_n + d_c * \frac{p_{n+1}=p_n}{r_{p_{n+1}/p_n}} & r_{p_{n+1}/p_n} > d_c > 0 \\ p_{n+1} & d_c > r_{p_{n+1}/p_n} \end{cases}$$

This is repeated for every line segment and the line segment where the distance between p_{cur} and $p_{closest}$ is chosen. Then that $p_{closest}$ is the overall closest point. To set the actual desired carrot position, check how much distance is left on the line until the next waypoint. If the distance left is less than r *=1* then the next waypoint is the carrot position. Otherwise, it computes the carrot position as follows:

$$p_{carrot} = \begin{cases} p_{n+1} & r > r_{p_{n+1}/p_{closest}} \\ p_{closest} + r * \frac{p_{n+1}+p_{closest}}{r_{p_{n+1}/p_{closest}}} & r < r_{p_{n+1}/p_{closest}} \end{cases}$$

To steer towards that position we find the the vector distance between the current position and carrot position, $r_{p_{carrot}/p_{current}}$. Then the desired heading is:

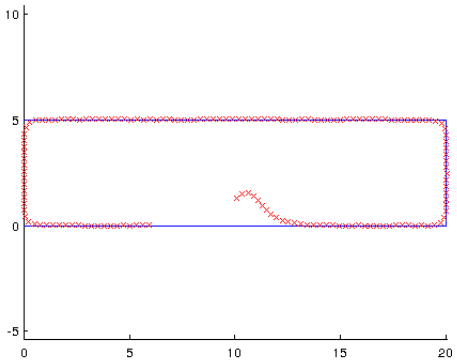
$$h_{desired} = arctan(\frac{r_{p_{carrot}/p_{current}x}}{r_{p_{carrot}/p_{current}y}}) \tag{9}$$

$$h_{error} = \begin{cases} h_{error} - 2 * \pi & h_{error} > \pi \\ (h_{desired} - h_{t-1})\%2 * \pi & h_{error} < \pi \end{cases}$$

We make the steering angle using proportional control (in this case we multiply our steering angle error by 1 to get a new steering angle).

$$\delta = 1 \ * h_{error} \tag{10}$$

Then we just feed that steering angle into the bike model from the previous question. The resulting plot of the robot driving around the rectangle is shown below. It starts with an abitrary position at (10, 1) with a heading of 90°.



3.0 IGVC Navigation

To generate a path that the carrot planner tries to follow, we generated waypoints. We used the bridge point strategy of generating two points, p_1 and p_2, close together. If one of the points is in an occupied part of the map then we throw that point away and keep the other one. If both points are in free space or both in occupied space we throw them both away. We repeat this until we have generated a satisfactory number of waypoints. This method helps us navigate through tighter spaces because it helps prevents the problem of only having waypoints in wide open areas and increases the number in tight spaces.

$$p_1 = [rand(0,1)*map\_size_x,\ rand(0,1)*map\_size_y\ ] \tag{11}$$

$$p_2 = [p_{1x} + N(0,\ \sigma_{bridge},\ p_{1y} + N(0,\ \sigma_{bridge}] \tag{12}$$

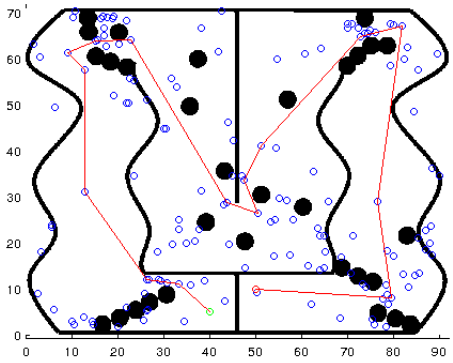
Then we check if either of the points are occupied in the map. This amounts to retrieving the value of the map at each point. Next we try to find connections between each of the generated points. This is part of our algorithm that will take the most time to execute. It takes the distance vector, d, between two points and then steps along it at a step size, $step$, of the roughly the size of map resolution.

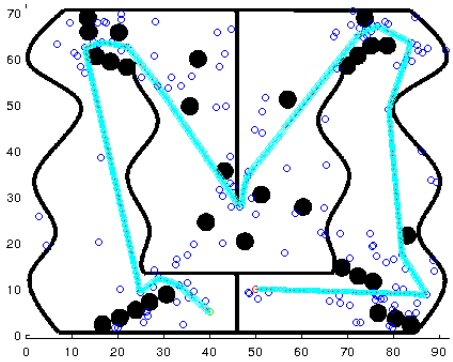
$$d = p_b \text{ - } p_b \tag{13}$$

$$step = \frac{d}{r_{b/a}/resolution} \tag{14}$$

$$p_c = p_a + \frac{d}{step} \tag{15}$$

At each point along the line, represented in the above equations as p_c, we check if the map is occupied. If so, then that line collides with an obstacle and does not represent a valid connection between points. Next we use A* to find the shortest path through the resulting connected points. The A* algorithm is fairly standard. The equations will not be reproduced here as it is mostly a matter of implementation. The selected path is shown below in red. The path then gets fed to the carrot planner in the form of the ordered waypoints. The carrot planner uses the waypoints in the same way it used the rectangle's waypoints in the last question. The output of the carrot planner is shown below in blue.





4.0 Discussion

One limitation of this approach (or at least the way we implemented this approach) is that it would only actually work for a robot of negligible size. The collision checks are done only for each point, which is the size of the resolution of the map. This approach could still work if the obstacles in the map are dilated appropriately. Alternatively, we could check in a radius around each point. This would ensure that we did not plan paths through areas that are, for exmaple, only one pixel wide.

We also assume that the ground is perfectly flat. To take into account changes in elevation we would need a much more sophisticated map and model. A tool like Gazebo would be more suited for that task.

Another limitation is that the robot is not guaranteed to be able to drive the path that the sampling-based planner generates. The planner does not take into account the robot's kinematics and dynamics. For example, it might suggest a series of sharp turns to get through an area that the robot ultimately cannot navigate. To solve this, we could use inverse kinematics to adapt the connections between the nodes to follow kinematically sound paths. Another way would be to use an approach like an RRT and generate somewhat random trajectories and accept the trajectory if it includes the next waypoint.

Related to this, the motion model assumes you have perfect traction and can always maintain a certain speed. The noise in the model does not perfectly take into account the fact that the wheels would slip more on turns than straightaways, for example. To remedy this, you could add momentum as a state variable.

The path generated is also not an optimal path. It uses semi-random (bridge) points. The path is only optimal for that set of waypoints as we used A* to find the shortest possible path between those points. A sampling-based planner such as this does not guarantee an optimal path. However, by sampling more you can improve the chances of finding a better path. The probability of finding an optimal path converges to 1 as the number of samples goes to infinity. We could also change the approach to use a visibility graph instead of the generated points. This has its own challenges with proximity to obstacles, etc.