Vehicle Path Planning

Jared Ticotin

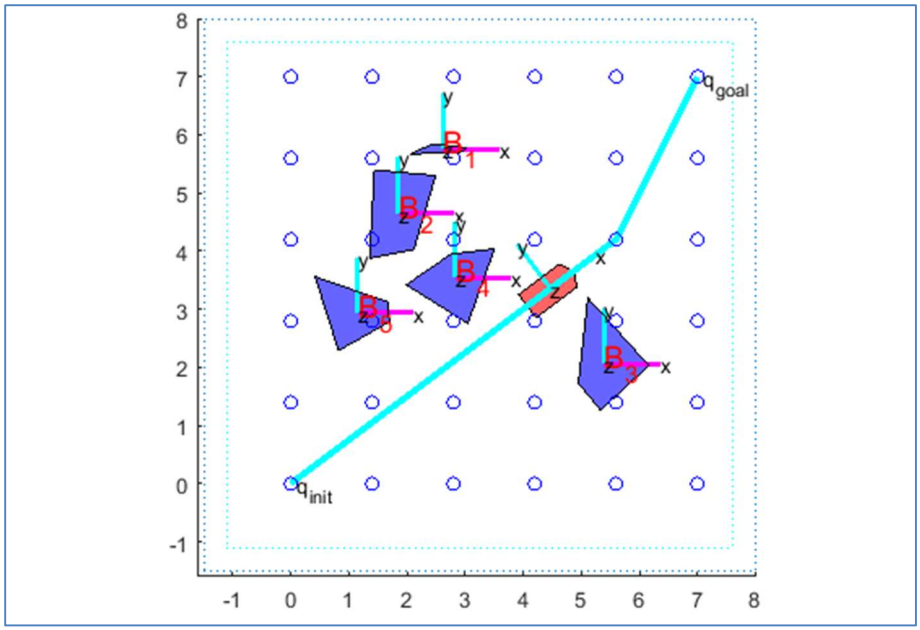
The challenge was to bring a robot from some initial position to a goal position without hitting any obstacles. The obstacles were randomly generated each time.

Two methods were used to automatically plan the path of a vehicle:
Method 1: Stop and Turn
Method 2: Kinematic Constraints

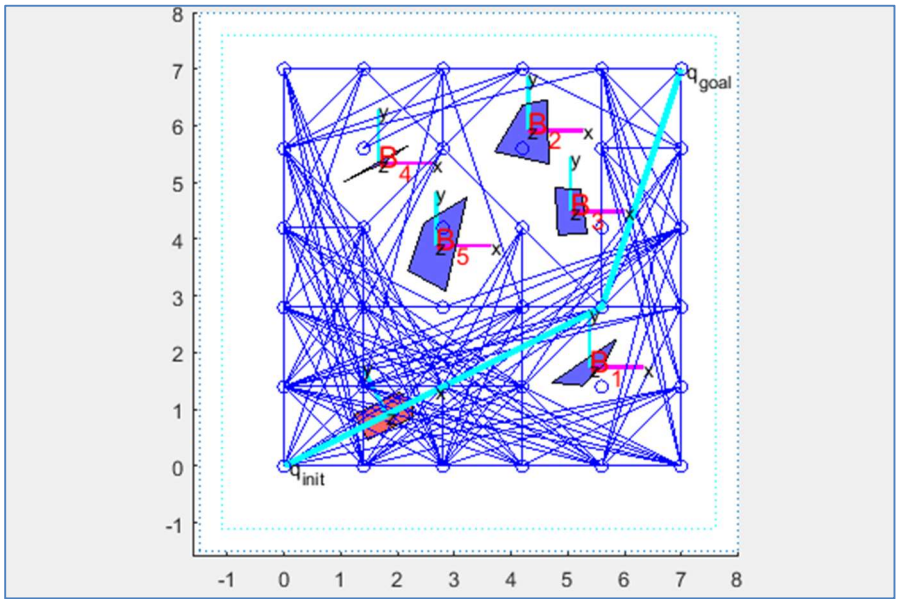
Method 1 is simpler and assumes the robot can rotate in place but can only move forward and backward in the direction it is facing.
Method 2 will drive in a smooth path so that it doesn't need to stop to make a turn.

Method 1: Stop and Turn Method:

The algorithm incorporates both position and orientation. The robot now needs to always orient in the direction of travel. The results:



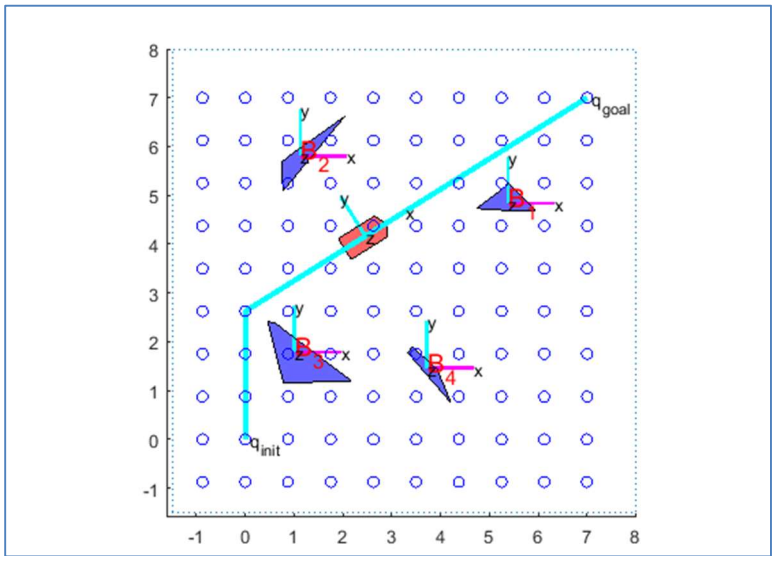
At each travel point and each angle turn, the Cobstacles need to be taken into account. This algorithm makes sure the robot never hits an obstacle (or boundary line) whether it is travelling forward or turning. Note that here I am only showing the Obstacles since the Cobstacle changes shape at each angle.



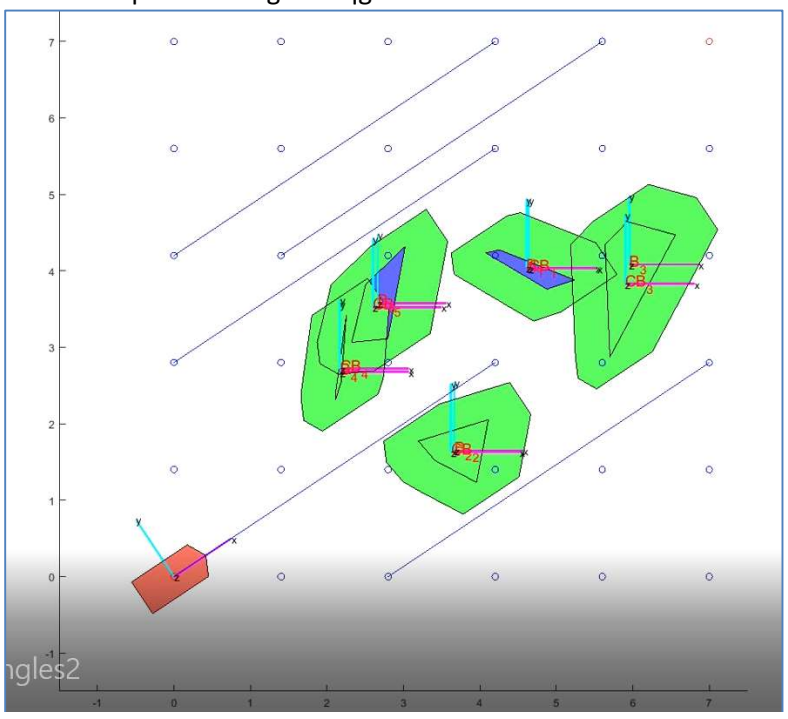
Here I'm showing all possible paths between nodes. The paths are only possible where there's no intersection with the Cobstacle at that angle. Once again, I'm only showing the obstacles since the Cobstacles are different for each angle. Please refer to the animations in the zip to see what the Cobstacle and paths look like at each angle.

Looking at this also gives an idea of all the possible angles the robot can make. Note that the angle increments are NOT constant. They are reliant on the angle to travel from one node to another.

(continued next page…)



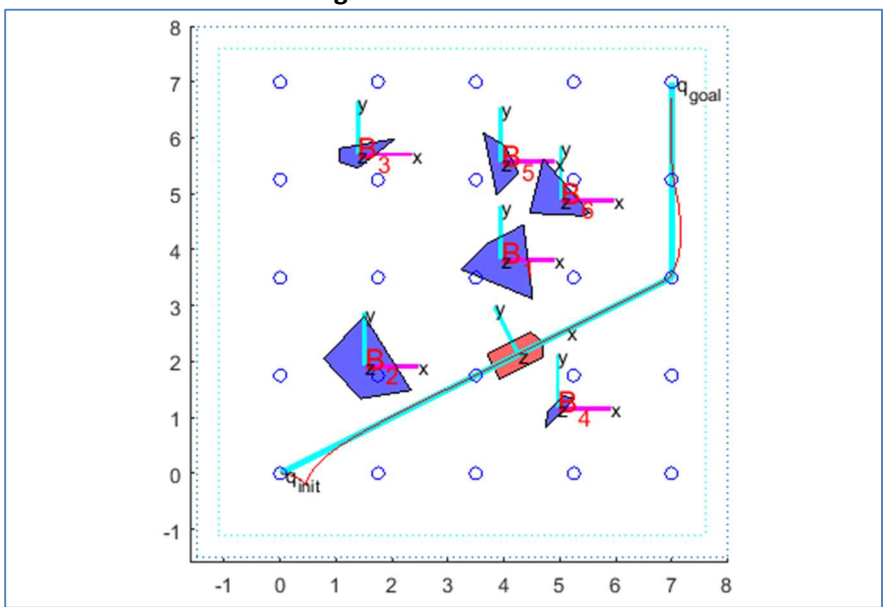
I wanted to show what it looks like with 10x10 grid points. Mainly, I wanted to show that it will create grid points outside of qinit and qgoal if possible (it depends on the distance between grid points and the distance between qinit/qgoal and the boundaries).
This is important because there may be some cases when the robot must go around and travel outside its current position to get to qgoal.



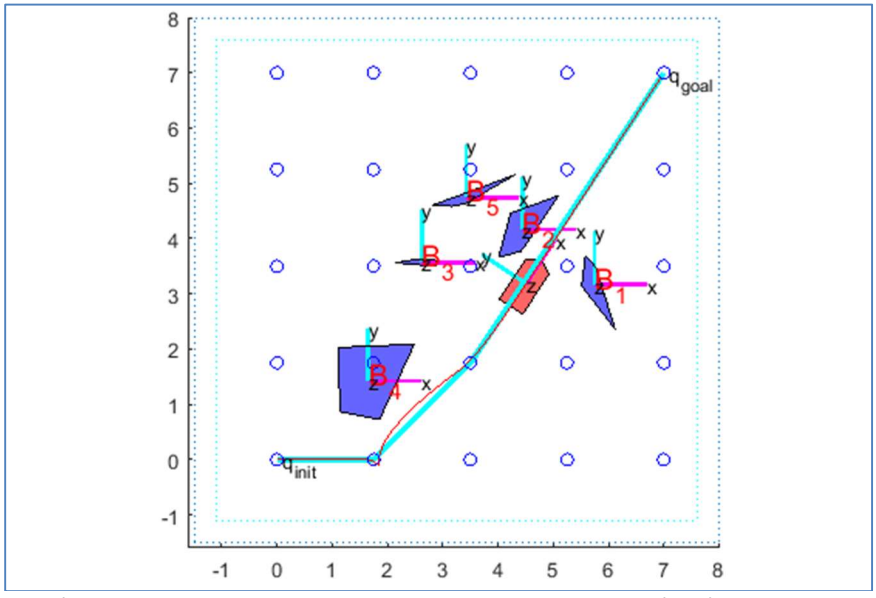
This is a quick preview of the animation “FinalProjectProblem3c_CObstacleAngles2”. It shows what the Cobstacles look like as the robot rotates. It also shows all valid paths when facing that direction.
The red circle at the top right corner is qgoal.

For all problems, my animaterobot() function does 2 things: it discretizes/interpolates the paths between nodes and animates the motion. This function returns the discretized qpath.

Method 2: Kinematic driving constraints for a 2 wheeled car:



I used the same robot appearance for this two wheeled car. The car has a cover and the wheels are underneath.
To get this to work, I started with the optimal path that came from Grid Paths and extracted all the way points as well as the start and end angle. I adjusted the wheel radius, distance between wheels and reference point d according to the scale of the robot. Then the scale factor s was tuned to get a path that wraps reasonably close to the original path.



Here's another example comparing the kinematic path (red) and the GridPaths path (cyan).
Note that the kinematic path diverges slightly from the GridPaths path so there's a small chance that it could collide with the obstacles. However, I've found that the vast majority of the time, it does not. One way to prevent all collisions here is to increase the Cobstacle sizes based on the radius of curvature of the kinematic path.

The pseudocode for accounting for orientation and position was broken into 3 segments: gridCoords,gridPath, and gridPaths.
The approach was to set up a grid of points that spans inside the boundaries while also containing a point exactly located at qinit and qgoal. These are all possible positions it will travel to for any angle. Now for the angles: the angles are all possible angles created when traveling from one node to any other node in the grid.
Now that there's a set number of possible angles and set number of nodes to position the robot, I can create a set number of CObstacles. I had a cell CB{i,a} where i is the obstacle # and a was the angle #. For each angle, the robot can only travel in that direction. So I set up all paths in that direction so long as it didn't collide with a CObstacle at that angle.
I also set up a rule where it can only rotate by one angle increment at a time. This handles the turning of the robot while also checking to make sure there's no collision with obstacles while turning.
Finally, I set up the small rotation required to get from qinit to the closest 2 available angles. Similarly, the closest 2 available angles were set up so they can rotate to qgoal. At the end I returned large adjacency matrices and qpath which includes x,y,theta. Now Astar can be utilized to find the best path. I only considered x,y distance when calculating my heuristic. However, a weight was added for rotations.

Grid Coords Pseudocode:
1: get max and min x and y coordinates based on the boundaries
2: generate a grid (of user specified amount of points) that spans these boundaries
3: scale smaller and shift such that one grid point lands on qinit and one lands on qgoal
4: find all possible angles between grid points
5: make a cell gdir{a} that contains information on how many grid points a node at angle a can travel down and how many points it can travel right.
-I ordered up to down and then left to right so I could utilize matlab's sub2ind and ind2sub
-to reduce the # of travel points, only allow travel that doesn't intersect another node

Grid Path Pseudocode:
1: based on CB and gdir (for a specific angle), grid coordinates and boundaries: generate all the paths that go in this one specific direction. Do not save any path the collides with obstacles. Return adjacency and weighted adjacency matrices that reflect the paths found.

Grid Paths Pseudocode:
1: get grid coordinates using Grid Coords above.
2: make 3D Adj and wAdj matrices and collect all paths at each angle using Grid Path from above.
3: start Adj and wAdj (adjacency matrices covering all angles) at size LA*Lg+2
-where LA= number of angles, Lg= number of grid coordinates, and 2 to account for qinit and qgoal
4: collect all adjacency information from the 3D Adj and wAdj (forward travel only, no rotation yet)
5: add rotation to Adj, only allowing it to rotate one angle away (including from the largest angle to the smallest angle). The weight for wAdj is proportional to the absolute value of the change in angle.
6: find the closest angles to qinit and qgoal. Find the Grid Coords that match their positions. Now update Adj and wAdj to allow rotation from qinit to these 2 angles as well as from the closest angles to qgoal.
7: if a path from qinit to qgoal doesn't exist, then try again from step 1 but with more grid points

Please see the zip for the script and all code for this. I included several animations both to show it working and also to give a visual representation of how it works.

Two Wheeled Robot One Step Pseudocode (handles only 1 start and 1 end point)

1: start loop:

1a: calculate the change in position and normalize by the distance to qgoal

1b: Calculate the translational and rotational Jacobian:

$J_T = \frac{r}{2}\begin{bmatrix} cos\theta & \cos\theta \\ sin\theta & sin\theta \end{bmatrix} + \frac{dr}{l}\begin{bmatrix} sin\theta & -sin\theta \\ -cos\theta & cos\theta \end{bmatrix}; J_R = \frac{r}{l}[-1,1]$

1c: calculate the change in wheel angles: $d\phi = s * J_T^{-1} dX$

1d: calculate the change in robot orientation: $d\theta = J_R d\phi$

1e: update the change in position from the rotation of the robot body:

$dX = \frac{r}{2}[1,1]\begin{bmatrix} cos\theta \\ sin\theta \end{bmatrix} d\phi$

1f: find the next position and orientation: X=X+dX and $\theta = \theta + d\theta$

2: repeat step 1 until within a reasonable distance from qgoal

3: if this is the absolute end position, then rotate the robot's wheels such that it corrects the end orientation

Two Wheeled Robot Pseudocode (qinit,qgoal, and all way points in between)

1: save qinit (first point) and qgoal (last point)

2: get all the unique x,y coordinates (ignoring angles).

3: update qpath to include the indexof the unique coordinates (and keep the angles this time)

4: loop through the new set of qpath and interpolate the path using Two Wheeled Robot One Step

5: at the very last step, have the robot rotate in place into the qgoal end orientation

Please see the script as well as twoWheeledRobotOneStep() and twoWheeledRobot() functions in the zip. The zip also contains animations of the robot moving from qinit to qgoal

Future Implementations

The methods used worked successfully but it is important to think of possible improvements. One limitation is that this approach doesn't scale well with the number of nodes (computation time increases exponentially). Some possible improvements include:

-eliminate all rotation angles at nodes where nothing can travel to it or away from it.

-break the atlas down to smaller charts: charts containing the obstacles and empty charts.

--clearly, the empty charts don't need to be so dense with nodes

-grid points inside the obstacles or outside boundaries can be removed all together.

When a gridsize of 6x6 is used, it calculates very fast. But it takes noticeably longer as the grid size is increased to 9x9. With some more sophisticated modifications (such as those mentioned), this approach can be both time efficient and good at finding optimal paths.