Path Planning and Deployment of Mobile Robot Networks

Shengdong Liu
University of California, San Diego
Spring 2015

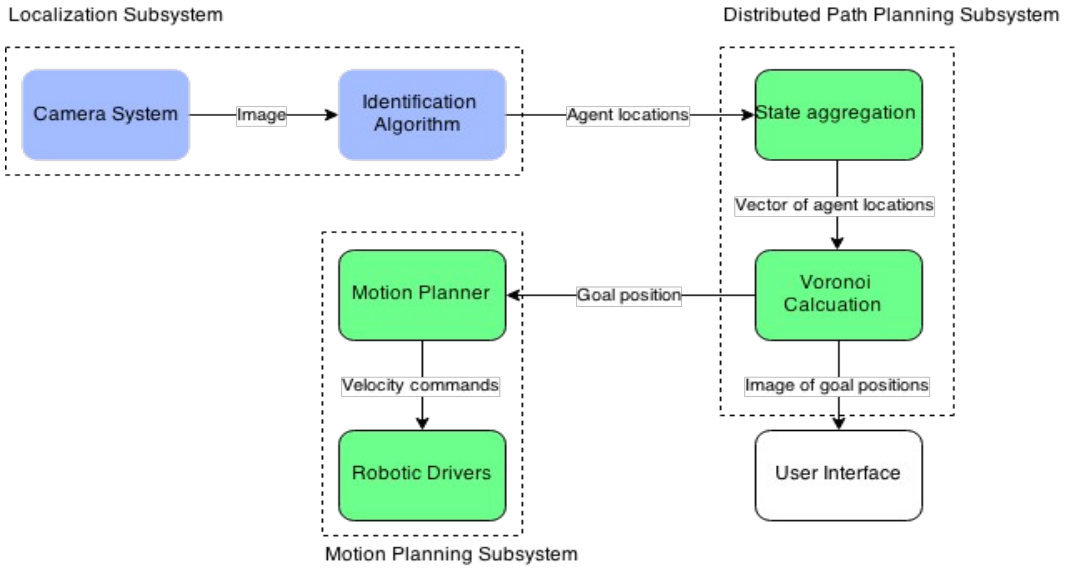
Abstract

This paper focuses on the implementation of a path planning algorithm – Dubins Curve. This algorithm is adopted to simulate a simple car on TurtleBot, a small ground robot consists of a mobile base, 3D Sensor, and a laptop computer. The simple car is assumed to move at constant forward speed and is constrained by a maximum steering angle, which results in a minimum turning radius. With the implementation of this algorithm, TurtleBots can simulate a network of autonomous vehicles for the the setting up and testing of inter-robot communication and coordination.

1 Big Picture

Advancements in robotics technology are increasing the number of autonomous robots in our daily lives. This boom in robots calls for not only more robot-user interaction, but also more robot to robot communication. By communicating, robots can complete tasks in groups. Tasks such as localization, collision avoidance, and traffic control can be done more accurately and efficiently through inter-robot communication and coordination. The Cortes and Martinez labs at UCSD have been incubating a testbed to test potential multi-agent control algorithms for deployment. The labs currently possess ten TurtleBot platforms as a multi-agent robotic network. The open source ROS (Robotic Operating System) is used as the software platform for its publisher-subscriber model to allow for inter-robot communication and coordination. The objective of the ROS TurtleBot project is to provide a stable system to which distributed control system can be readily deployed for testing and validation on hardware. There are a plethora of interesting algorithms that could potentially be deployed using the TurtleBot network, such as cooperative task completion. Of particular interest to the group is cooperative simultaneous localization and mapping (SLAM).

The current system set up for the testbed consists of three subsystems. The localization subsystem provides the position and orientation of each of deployed TurtleBot through two overhead cameras that spend video stream to a laptop as the base station. The distributed path planning system applies multi-agent control algorithms and uses the video stream to calculates the goal position of each TurtleBot. The motions planning subsystem on each TurtleBot then uses the goal position to create a path and send velocity commands to the mobile base for navigation. Since the whole system is implement in ROS, information are easily passed on by means of publishers and subscribers between the subsystems. Visualizations of the multi-agent control algorithm are created and displayed to the users through the user interface for monitoring the correctness and effectiveness of the multi-agent control algorithm being tested.



Drawing 1

2 Personal Contribution

This quarter, I worked with Daniel Heideman on the motion planning subsystem, specifically on the motion planner. Two main components of the motion planner are path planing and path following. Path planning is the process that calculates an optimal path, given the robot's current location, a goal location, and some constrains. Path following is the process that uses the optimal path as a guide to produce the speed and turning commands sent to the robot's wheels. Currently the motion control algorithm for deployment is first-turn-then-move. The path planer basically finds the line between the current position of the TurtleBot and the goal position, then send commands to the TurtleBot to turn to the orientation of the goal position and move forward until the goal is reached. Although this algorithm works on TurtleBot, it's cannot be applied to many mobile machinery in real life. Our goal for this quarter was to upgrade the motion planning algorithm to be more sophisticated, and thus a better simulation of a real life environment.

Daniel and I chose to implement the Dubins Curve algorithm to simulate a simple car. We divided the work up so that I would work on the motion planer and he would work on the path follower. In the next few sections, I'll will explain the nature of Dubins Curve and how I implemented for TurtleBot in ROS. (Please refer to report-Sp15-dHeideman.pdf to see more on the path follower.)

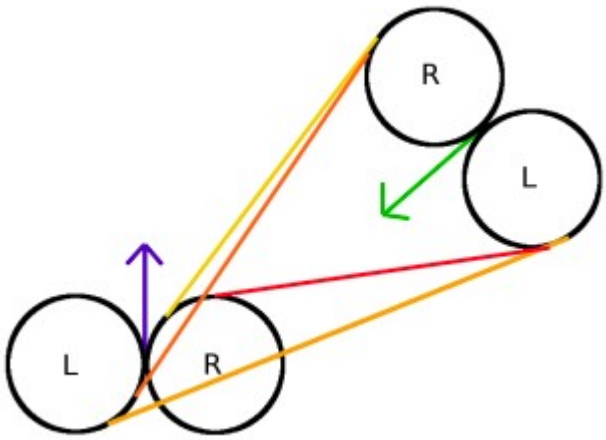
3 Preliminaries

This section explains the idea of a Dubins Car and how to construct a Dubins Path. Dubins path refers to the shortest curve that connects two points in the two-dimensional Euclidean plane with the constraint on the maximum steering angle. In 1957, Lester Eli Dubins showed using geometrical arguments that any such path can be made by joining circular arcs of maximum curvature and straight lines[1]. In his paper, he proved that there are only 6

combinations of arcs and straight lines can describe all the shortest path. Each path consist of 3 segments, and each segment has one of the three controls: L (a left turn of maximum curvature), R (a right turn of maximum curvature, and S (a straight line). The 6 combinations of there controls are that makes up the shortest path are **RSR**, **LSL**, **RSL**, **LSR**, **RLR**, **and LRL**. For the purpose of explaining the paths, we can categorize them into two classes: CSC (Curve-Straight-Curve) and CCC (Curve-Curve-Curve)

3.1 CSC (Curve-Straight-Curve)

CSC is a path consist of a curve of maximum turning radius, a straight line, then another curve of maximum turning radius. Essentially, we are looking at **RSR, LSL, RSL,** and **LSR**. Intuitively, the shortest path is a straight line from the current location of the vehicle to the goal location. Because our simple car have a maximum steering angle, resulting in a minimum turning radius, it cannot turn in spot and move forward to the goal location. However, what can be done is drawing a arc to the left or right from the initial position, and drawing a arc to the left or right into the goal position, then finding the line that is tangent to the arcs drawn. This gives us the shortest straight line distance given that the car has a minimum turning radius. Let's take a look at this example:

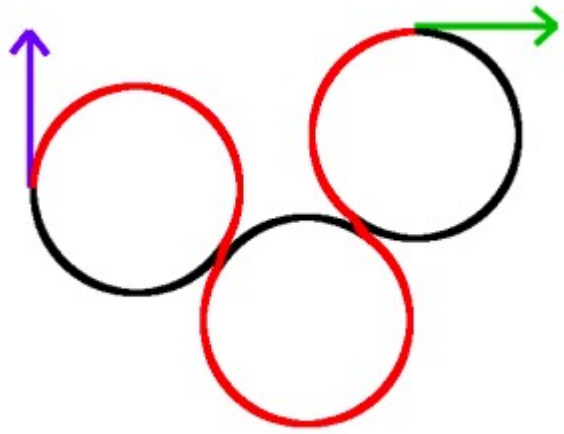


Drawing 2

In Drawing 2, the purple arrow denotes the initial position and orientation of the simple car, and the green arrow is the goal position and orientation the car wants to be. From each position, 2 arcs are drawn, one turning left and one turning right at maximum curvature. Then tangent lines are drawn to connect the arcs from the initial position to the arcs at the goal position. Now let take a look at the path **RSR**, which consist of the initial arc R, the yellow line and final arc R. If the car were to turn any less than where the yellow line is before moving straight forward, the path would not be optimal, which can be shown using Pythagorean theorem. If the car were to turn more that where the yellow line is before moving straight forward, the path would miss the minimal arc that ends at the goal position, thus this path is again, not optimal. Therefore, the yellow line that is tangent to the initial and final arc is the optimal path for **RSR**. Of course, this does not guarantee that **RSR** is optimal path, but we know that this path is indeed optimal for the combination right-straight-right.

3.2 CCC (Curve-Curve-Curve)

CCC is a path consist of 3 curves of maximum curvature. The first curve turns in one direction, followed by turn in the opposite direction, and lastly, another turn in the original direction. Essentially, we are looking at **LRL**, and **RLR**. These path are only valid when the initial and the goal position are relatively close to one another.



Drawing 3

Drawing 4

Drawing 3 is an example of CCC from the purple arrow to the green arrow via the red path. After seeming this drawing, it is reasonable to wonder why would we need the CCC approach? The CSC path looks more straight forward and works even when the goal position is farther away. The answer lies in that CSC are less than optimal when the goal position is really close to the initial position. Now consider the following scenario: navigate from one position to the exact same position but with the opposite orientation. An CSC solution would look like Drawing 4, can you think of a CCC solution that is better? (see Drawing # for solution)

For more in depth information on Dubins Curve, please read Planning Algorithms by Steven M. LaValle, 2006, Section 13.1.2.1 on simple car [2], and Section 15.3.1 on Dubins Curves [3]. Another good resource is A Comprehensive, Step-by-Step Tutorial to Computing Dubin's Paths on Andy G's Blog Page[4].

4 Methodology

4.1 Algorithm Adoption

I adopted Andrew Walker's implementation of Dubins-Curve in C++[5], as to not reinvent the wheel. His code, dubin.c contains the entirty of the Dubins algorithm to generate a Dubins path from 3 input parameters: an initial position, a goal position, and a minimum turning radius. Each position is composed of three floating point numbers: (x, y, theta), where x and y are the position on a 2D Cartesian plane, and theta is the orientation in radians. I conducted some tests to verify the correctness of Andrew's implementation. Here are the test results of each of the six (**LRL**, **LSL**, **LSR**, **RLR**, **RSL**, and **RSR**) scenarios.

```
mike@GalaxyFXB: ~/Documents/school/10turtleBot/Dubins-Curves/src
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$ ./dubins
This Program takes in 7 input parameters:
x0
y0
alpha (in radians)
x1
y1
beta (in radians)
mtr (minimum turning radius)
0 0 0
2 0 4.71
1
# (0.000000 0.000000 0.000000)    (2.000000 0.000000 4.710000)
# best path = 5 = LRL
  segments = (1.907079 4.459093 0.978829) in mtr

(0.000000,0.000000,0.000000),0.000000
(0.479426,0.122417,0.500000),0.500000
(0.841471,0.459698,1.000000),1.000000
(0.997495,0.929263,1.500000),1.500000
(0.917442,1.418994,1.814159),2.000000
(0.920727,1.913791,1.314159),2.500000
(1.160827,2.346442,0.814159),3.000000
(1.578959,2.611018,0.314159),3.500000
(2.072749,2.642742,6.097344),4.000000
(2.521300,2.433848,5.597344),4.500000
(2.814791,2.035478,5.097344),5.000000
(2.881365,1.545170,4.597344),5.500000
(2.704723,1.082966,4.097344),6.000000
(2.338056,0.747167,3.864999),6.500000
(2.059733,0.338056,4.364999),7.000000
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$
```

Initial position: (0, 0, 0)
Goal position: (2, 0, 4.71)
Min turning radius: 1
Expected path: **LRL**

LRL Test

```
mike@GalaxyFXB: ~/Documents/school/10turtleBot/Dubins-Curves/src
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$ ./dubins
This Program takes in 7 input parameters:
x0
y0
alpha (in radians)
x1
y1
beta (in radians)
mtr (minimum turning radius)
0 0 0
0 7 3.1415926
2
# (0.000000 0.000000 0.000000)    (0.000000 7.000000 3.141593)
# best path = 0 = LSL
  segments = (1.570796 1.500000 1.570796) in mtr

(0.000000,0.000000,0.000000),0.000000
(0.494808,0.062175,0.250000),0.500000
(0.958851,0.244835,0.500000),1.000000
(1.363278,0.536622,0.750000),1.500000
(1.682942,0.919395,1.000000),2.000000
(1.897969,1.369355,1.250000),2.500000
(1.994990,1.858526,1.500000),3.000000
(2.000000,2.358407,1.570796),3.500000
(2.000000,2.858407,1.570796),4.000000
(2.000000,3.358407,1.570796),4.500000
(2.000000,3.858407,1.570796),5.000000
(2.000000,4.358407,1.570796),5.500000
(2.000000,4.858407,1.570796),6.000000
(1.967972,5.356492,1.750000),6.500000
(1.818595,5.832294,2.000000),7.000000
(1.556146,6.256347,2.250000),7.500000
(1.196944,6.602287,2.500000),8.000000
(0.763322,6.848605,2.750000),8.500000
(0.282240,6.979985,3.000000),9.000000
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$
```

Initial position: (0, 0, 0)
Goal position: (0, 7, 3.14)
Min turning radius: 2
Expect path: **LSL**

LSL Test

```
mike@GalaxyFXB: ~/Documents/school/10turtleBot/Dubins-Curves/src
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$ ./dubins
This Program takes in 7 input parameters:
x0
y0
alpha (in radians)
x1
y1
beta (in radians)
mtr (minimum turning radius)
0 0 0
5 5 0
2
# (0.000000 0.000000 0.000000)    (5.000000 5.000000 0.000000)
# best path = 1 = LSR
  segments = (1.099228 1.581139 1.099228) in mtr

(0.000000,0.000000,0.000000),0.000000
(0.494808,0.062175,0.250000),0.500000
(0.958851,0.244835,0.500000),1.000000
(1.363278,0.536622,0.750000),1.500000
(1.682942,0.919395,1.000000),2.000000
(1.918701,1.360065,1.099228),2.500000
(2.145843,1.805493,1.099228),3.000000
(2.372985,2.250921,1.099228),3.500000
(2.600127,2.696350,1.099228),4.000000
(2.827269,3.141778,1.099228),4.500000
(3.054411,3.587207,1.099228),5.000000
(3.285820,4.030333,1.029594),5.500000
(3.594018,4.422397,0.779594),6.000000
(3.989633,4.726024,0.529594),6.500000
(4.448068,4.922335,0.279594),7.000000
(4.940820,4.999124,0.029594),7.500000
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$
```

LSR Test

Initial position: (0, 0, 0)
Goal position: (5, 5, 0)
Min turning radius: 2
Expected path: **LSR**

```
mike@GalaxyFXB: ~/Documents/school/10turtleBot/Dubins-Curves/src
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$ ./dubins
This Program takes in 7 input parameters:
x0
y0
alpha (in radians)
x1
y1
beta (in radians)
mtr (minimum turning radius)
0 0 0
2 0 1.57
1
# (0.000000 0.000000 0.000000)    (2.000000 0.000000 1.570000)
# best path = 4 = RLR
  segments = (1.908446 4.459914 0.981469) in mtr

(0.000000,0.000000,0.000000),0.000000
(0.479426,-0.122417,5.783185),0.500000
(0.841471,-0.459698,5.283185),1.000000
(0.997495,-0.929263,4.783185),1.500000
(0.917201,-1.418922,4.466294),2.000000
(0.919133,-1.913726,4.966294),2.500000
(1.158051,-2.347031,5.466294),3.000000
(1.575458,-2.612749,5.966294),3.500000
(2.069159,-2.645822,0.183109),4.000000
(2.518279,-2.438154,0.683109),4.500000
(2.812858,-2.040589,1.183109),5.000000
(2.880772,-1.550463,1.683109),5.500000
(2.705393,-1.087779,2.183109),6.000000
(2.339290,-0.751438,2.419829),6.500000
(2.060296,-0.342785,1.919829),7.000000
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$
```

RLR Test

Initial position: (0, 0, 0)
Goal position: (2, 0, 1.57)
Min turning radius: 1
Expected path: **RLR**

```
mike@GalaxyFXB: ~/Documents/school/10turtleBot/Dubins-Curves/src
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$ ./dubins
This Program takes in 7 input parameters:
x0
y0
alpha (in radians)
x1
y1
beta (in radians)
mtr (minimum turning radius)
0 0 0
4 -6 0
2
# (0.000000 0.000000 0.000000)    (4.000000 -6.000000 0.000000)
# best path = 2 = RSL
  segments = (1.570796 1.000000 1.570796) in mtr

(0.000000,0.000000,0.000000),0.000000
(0.494808,-0.062175,6.033185),0.500000
(0.958851,-0.244835,5.783185),1.000000
(1.363278,-0.536622,5.533185),1.500000
(1.682942,-0.919395,5.283185),2.000000
(1.897969,-1.369355,5.033185),2.500000
(1.994990,-1.858526,4.783185),3.000000
(2.000000,-2.358407,4.712389),3.500000
(2.000000,-2.858407,4.712389),4.000000
(2.000000,-3.358407,4.712389),4.500000
(2.000000,-3.858407,4.712389),5.000000
(2.032028,-4.356492,4.891593),5.500000
(2.181405,-4.832294,5.141593),6.000000
(2.443854,-5.256347,5.391593),6.500000
(2.803056,-5.602287,5.641593),7.000000
(3.236678,-5.848605,5.891593),7.500000
(3.717760,-5.979985,6.141593),8.000000
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$
```

Initial position: (0, 0, 0)
Goal position: (4, -6, 0)
Min turning radius: 2
Expected path: **RSL**

RSL Test

```
mike@GalaxyFXB: ~/Documents/school/10turtleBot/Dubins-Curves/src
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$ ./dubins
This Program takes in 7 input parameters:
x0
y0
alpha (in radians)
x1
y1
beta (in radians)
mtr (minimum turning radius)
0 0 0
0 -7.5 3.1415
2.5
# (0.000000 0.000000 0.000000)    (0.000000 -7.500000 3.141500)
# best path = 3 = RSR
  segments = (1.570704 1.000000 1.570982) in mtr

(0.000000,0.000000,0.000000),0.000000
(0.496673,-0.049834,6.083185),0.500000
(0.973546,-0.197348,5.883185),1.000000
(1.411606,-0.436661,5.683185),1.500000
(1.793390,-0.758233,5.483185),2.000000
(2.103677,-1.149244,5.283185),2.500000
(2.330098,-1.594106,5.083185),3.000000
(2.463624,-2.075082,4.883185),3.500000
(2.500007,-2.573009,4.712482),4.000000
(2.500053,-3.073009,4.712482),4.500000
(2.500099,-3.573009,4.712482),5.000000
(2.500146,-4.073009,4.712482),5.500000
(2.500192,-4.573009,4.712482),6.000000
(2.499166,-5.072999,4.683185),6.500000
(2.434851,-5.568005,4.483185),7.000000
(2.273475,-6.040367,4.283185),7.500000
(2.021473,-6.471253,4.083185),8.000000
(1.688890,-6.843484,3.883185),8.500000
(1.288985,-7.142222,3.683185),9.000000
(0.837702,-7.355556,3.483185),9.500000
(0.353032,-7.474981,3.283185),10.000000
mike@GalaxyFXB:~/Documents/school/10turtleBot/Dubins-Curves/src$
```

Initial position: (0, 0, 0)
Goal position: (0, -7.5, 3.14)
Min turning radius: 2.5
Expected path: **RSR**

RSR Test

The result of these tests shows that Andrew's implementation correctly generate a Dubins Path ifor each of the six types of path. In addition to the path type, is also a vector in the format of {(x, y, theta), l}, where l is the length traveled along the path. We actually do not need l to generate a path for the TurtleBot, but it demonstrate the progression of the path for these tests. Also for testing purpose, the sample size of the steps along the path is set at 0.5 times the length of the minimum turning radius. This is a constant we can adjust when we produce the path for TurtleBot. One problem in sampling the path is that the goal position is not included, because the path length is not always a perfect multiple of the step size. (As you can see from the tests above, the end point just short of the actual goal position.) This is something I kept in mind when I wrote my wrapper class to integrate his algorithm for the TurtleBot.

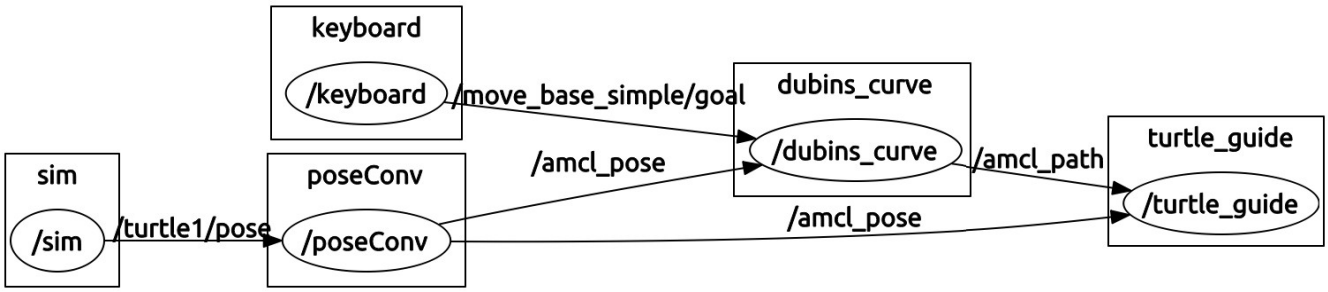
4.2 Algorithm Integration

I wrote an dubins_curve.cpp which has 2 subscribers and 1 publisher. In ROS (Robotic Operation System), information are communicated via “topics”, which functions like public chartrooms. A subscribers is a “listener” that gets information from the topic and a publisher is the “speaker” that presents the information to the topic. The two subscribers I used are subscribed to topics that have information on the current position of the TurtleBot and the goal position respectively. The positions retrieved from those are used as the input parameters for Andrew's implementation of Dubins Curve algorithm, which takes an initial position, and goal position, and a mtr (minimum turning radius). The mtr is currently define by a global constant for easy adjustment later on. We want the path to be outputted as an vector of positions, the format of the nav_msgs::Path object in ROS, once the Dubins Path object is built, we call on the “dubins_path_sample_many” method which returns a vector of positions, like those show in the tests in section 4.1. Due to the lack of an end point in the sample_many method, I manually added the goal position to the vector of positions to ensure the path would not stop short. When the path vector is completely built, it will be published to a topic so Daniel's path following algorithm can use that information to drive the TurtleBot. For the full implementation, please see source code for dubins_curve.cpp on Github. (Note: the Github repository is private, so if you are not one of the collaborators, please email me at shl202@ucsd.edu for more info).

4.3 Algorithm Testing Through TurtleSim

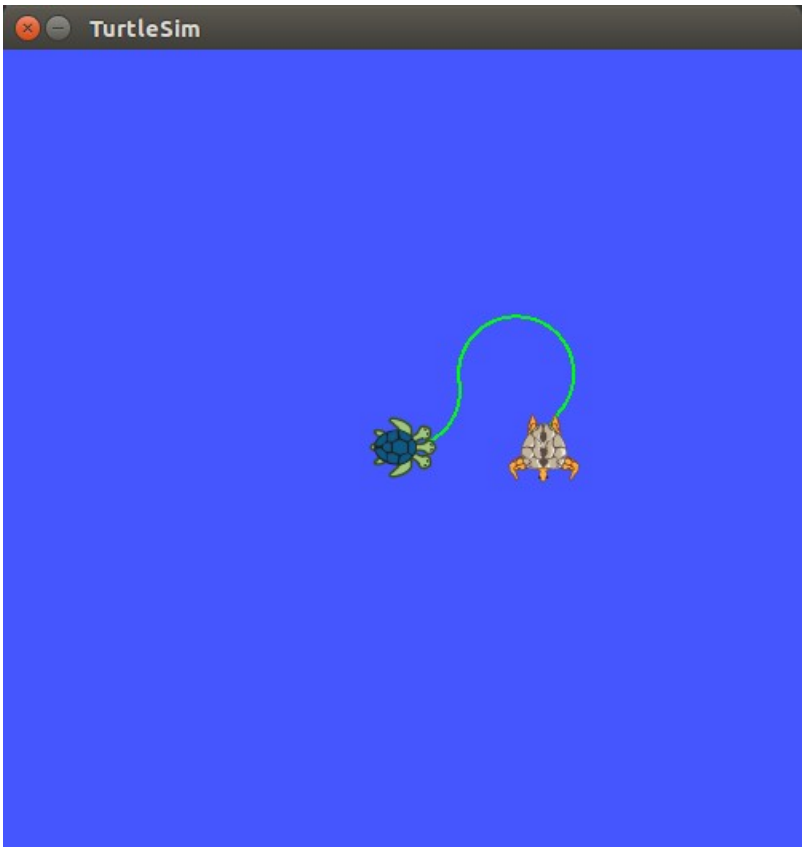
Having the algorithm run on the TurtleBots is the end goal, but not the most efficient way to test the algorithm due to hardware and calibration issues that could potentially create a lot of noise during the tests. Instead, Daniel and I decided to use TurtleSim as the testing environment. TurtleSim is very simplistic simulator for TurtleBots and is much easier to work with. To setup for the testing environment, there are a few things that needs to be done. First, dubins_curve gets current position of the TurtleBot from the overhead camera, but we what it to subscribe to the position that TurtleSim publishes. For this, I wrote a poseConv.cpp which subscribes to TurtleSim, converts the current position to the format that the camera would publish, and publish that information to the topic dubins_curve subscribes to. Next, dubins_curve gets the goal position from the multi-agent control algorithm. For simplicity and convenience, I wrote a keyboard.cpp which gets keyboard input of the goal position (x, y, theta) and send the goal information the correct topic. Lastly, we need the control algorithm to send the velocity command to TurtleSim. Since Daniel is working on the path following

algorithm, it would be difficult to debug if we test the two pieces of code together. Thus, Another method is needed to unit test the path planning algorithm. Fortunately, in TurtleSim, there is a service that teleports the Turtle to an absolute position. By teleporting the turtle along each of the positions in the path vector, a nice path can be drawn to represent the Dubins Curve. So I wrote a turtle_guide.cpp that subscribes to the vector of path published by dubins_curve, and teleport the newly spawned turtle along that path. Now we have all the prerequisite to test the dubins_curve.cpp, and the network looks like this:



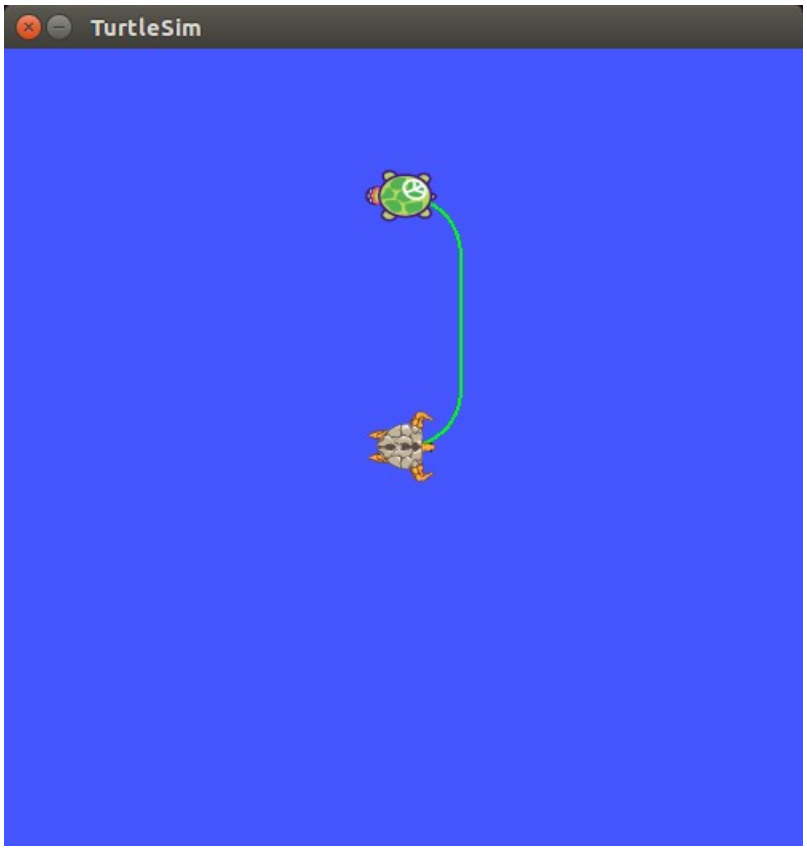
Drawing 5

In Drawing 5, each box represent a ROS node and the arrows are the topics. The nodes correspond with the cpp files that process information and topics are channels to pass the information to the next node. With the testing system set up like this, I ran some simulations on TurtleSim, again for each of the six types of Dubins Curve, and here are the test results:



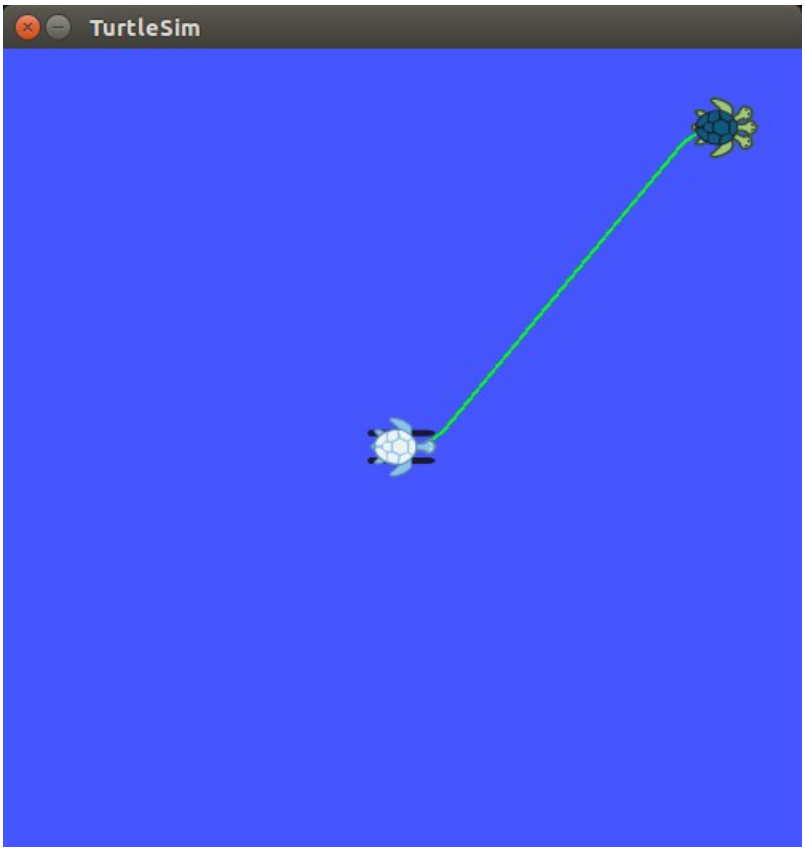
Initial position: (5.54, 5.54, 0)
Goal position: (7.54., 5.54, 4.71)
Min turning radius: 0.8
Expected path: **LRL**

LRL Test



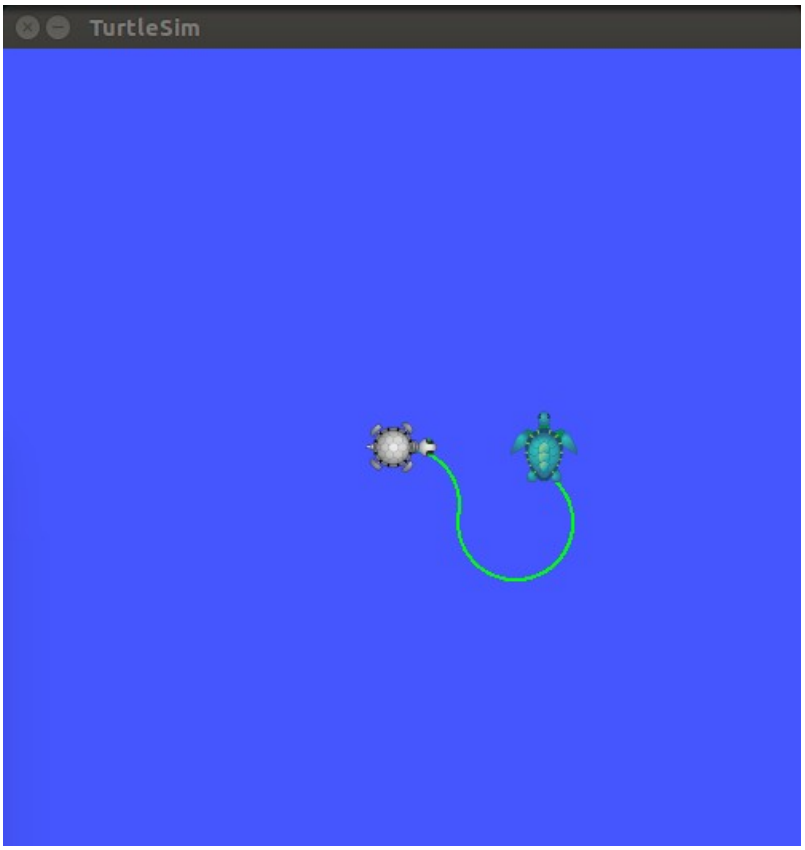
Initial position: (5.54, 5.54, 0)
Goal position: (5.54, 8, 3.14)
Min turning radius: 0.8
Expect path: **LSL**

LSL Test



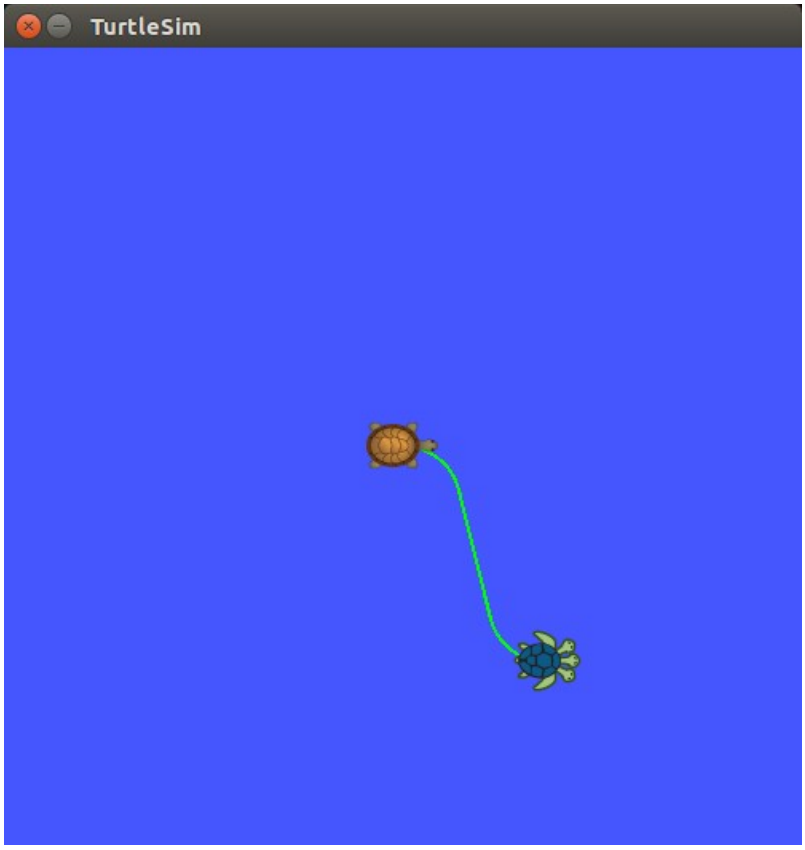
Initial position: (5.54, 5.54, 0)
Goal position: (10, 10, 0)
Min turning radius: 0.8
Expect path: **LSR**

LSR Test



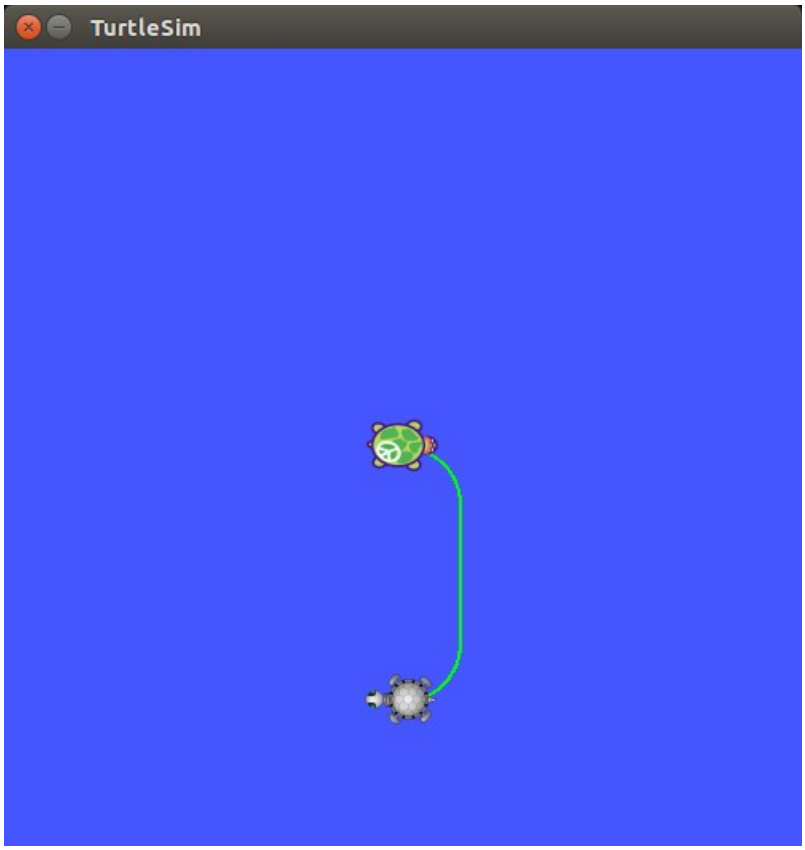
Initial position: (5.54, 5.54, 0)
Goal position: (7.54, 5.54, 1.57)
Min turning radius: 0.8
Expected path: **RLR**

RLR Test



Initial position: (5.54, 5.54, 0)
Goal position: (7.54, 2.54, 0)
Min turning radius: 0.8
Expected path: **RSL**

RSL Test



Initial position: (5.54, 5.54, 0)
Goal position: (5.54, 2, 3.14)
Min turning radius: 0.8
Expect path: **RSR**

RSR Test

These test results show that the path planing algorithm, Dubins Path works for each of the six path type, and it is ready to be integrated to the motion planner, in conjunction with the path following algorithm Daniel has, to replace the first-turn-then-move algorithm currently in use.

5 Tips

5.1 For Dubins Curve

Planning Algorithms by Steven M. LaValle, 2006, Section 13.1.2.1 on simple car [2], and Section 15.3.1 on Dubins Curves [3] provides great information to understand Dubins Curve and its application of it in Robotics. A Comprehensive, Step-by-Step Tutorial to Computing Dubin's Paths on Andy G's Blog Page[4] provides great in depth explanation of Dubins Curve and its implementation. Since the implementation used is adopted from Andrew Walker's implementation[5], a good understanding of his implementation is also very helpful.

5.2 For ROS

Going through beginner's tutorials[6] for ROS will help with understanding the structure of ROS. The particular tutorials that should be attempt before handling this project are #1-8, 11, 13, 14, 16 in the beginner's tutorials section. I would also recommend reading

#19 on Navigating the ROS wiki as you will be using the ROS Wiki pages very often.

Understanding the ROS API for geometry_msgs and nav_msgs[7] will be of great help for updating the subscribers and publishers for motion control.

The TurtleSim Tutorial[8] will help with understanding and modifying the tests conducted in TurtleSim, but it is optional as there are many ways to test the result other than using Turtlesim.

6 Pitfalls

6.1 ROS Distribution

The ROS distribution used for the project is “hydro”, so some of the more recent tutorial on the ROS Wiki page for distributions such are “indigo” and “jade” might not apply to the current project. As far as this motion planning process described in this paper, there's no problem, but nevertheless, we should look out for this problem when reading the Wiki pages and consulting the Internet.

6.2 Information Overload

There was a phase where dubins_curve node would crash after a few seconds after it is launched. I suspected that the reason for the crash was because in the duration between dubins_curve get the goal positions, it needs to calculate and construct a vector of many positions to publish. It might be interrupted mid calculation or vector construction because a goal position is received and thus a path needs to be recalculated. The problem was solved after a condition was added tells dubins_curve node to recalculate the path if the goal position is different from the previous goal received. This problem of dubins_curve crashing might reappear if the goal position is changing very rapidly.

7 Conclusion

We adopted Dubins Curve as the motion planing algorithm to simulate a simple car on TurtleBot, with the goal of setting up the testbed for inter-robot communication and coordination. We used Andrew's Walker's implementation of Dubin's Curve, which was shown to work for each of the six scenarios (**LRL**, **LSL**, **LSR**, **RLR**, **RSL**, and **RSR**) to generate a Dubins Path. A dubins_curve ROS node was written as a wrapper class to integrate Andrew's algorithm for TurtleBot. Simulation has been conducted in TurtleSim and the results show that the path is constructed properly as a navigate message and is ready to be tested on the TurtleBot. The next possible step is to combine this effort with Daniel's path follower to implement the motion control algorithm on the TurtleBot, Also it is in our best interest to implement collision avoidance so that the TurtleBots in a formation would not collide with each other.

References

[1] Dubins, L.E. (July 1957). "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents". *American Journal of Mathematics* **79** (3): 497–516. doi:10.2307/2372560.

[2] Steven M. LaValle, “Section 13.1.2.1 A Simple Car.” *Planning Algorithms*. 2006 http://planning.cs.uiuc.edu/node658.html

[3] Steven M. LaValle, “Section 15.3.1 Dubins Curves.” *Planning Algorithms*. 2006 http://planning.cs.uiuc.edu/node821.html

[4] Andy G, A Comprehensive, Step-by-Step Tutorial to Computing Dubin’s Paths https://gieseanw.wordpress.com/2012/10/21/a-comprehensive-step-by-step-tutorial-to-computing-dubins-paths/

[5] Andrew Walker, Path generation for the Dubin's car https://github.com/AndrewWalker/Dubins-Curves

[6] ROS Tutorials http://wiki.ros.org/ROS/Tutorials

[7] ROS Common Messages http://wiki.ros.org/common_msgs

[8] ROS TurtleSim http://wiki.ros.org/turtlesim