Cooperative Parking for Self-Driving Cars

Anton Lukyanenko, Damoon Soudbakhsh, Heath Camphire,
Avery Austin, Samuel Schmidgall

Mason Experimental Geometry Lab

December 7th, 2018

Introduction

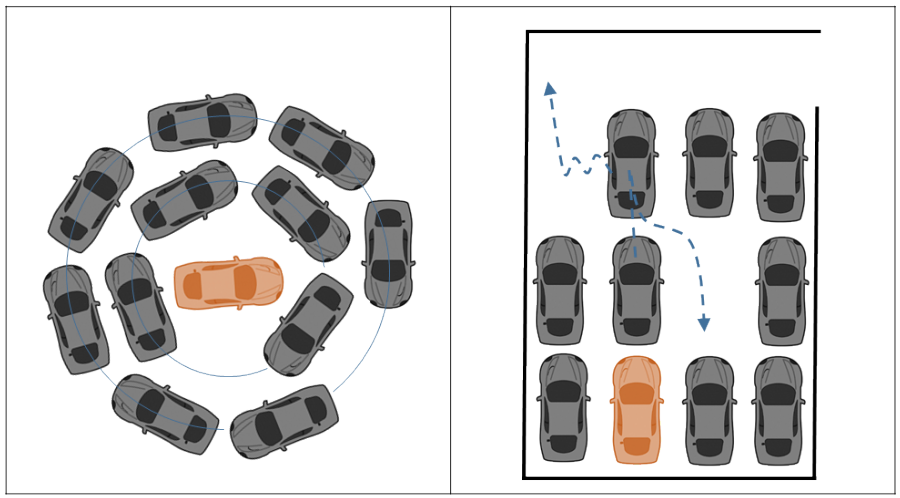
Self-Driving cars are quickly being integrated into society, but there are still lacking any motion-planning algorithms that effectively find paths for multiple cars.

The Broad Question

If we have a self-driving car filled parking garage and we want to get one of the cars out, how can we maneuver the cars around it so that the car in question leaves the garage?

The Not as Broad Question

Can we build an efficient algorithm that optimally maneuvers cars into specified positions?



Research Goals from this Semester

Simulation

- Build a simulation of self-driving cars in C++.
- Build an efficient algorithm to plan self-driving car motion.
- Make the code fast.

Robotic Implementation

- Get robots to a functional state.
- Have robots follow a specified path.
- Have multiple robots all follow a path.

Rapidly-Exploring Random-Tree*

To predict motion for a single car, an algorithm titled RRT* is commonly used. Our team chose to use RRT* for the path-finding aspect of our project.

Overview of RRT*

RRT*, which is short for Rapidly-Exploring Random-Tree*, is the current state-of-the-art path-finding algorithm that iteratively explores a state space.

Why RRT*?

- RRT* is unique in that it converges to the optimal solution.
- RRT* allows for differential constraints – such as a turning radius.

AutoPark RRT* Algorithm

In our case, RRT* was not a sufficient because it only considered the path of one car. So, our team extended the original RRT* algorithm to work with multiple cars.

RRT* Algorithm

Let N = Number of Iterations

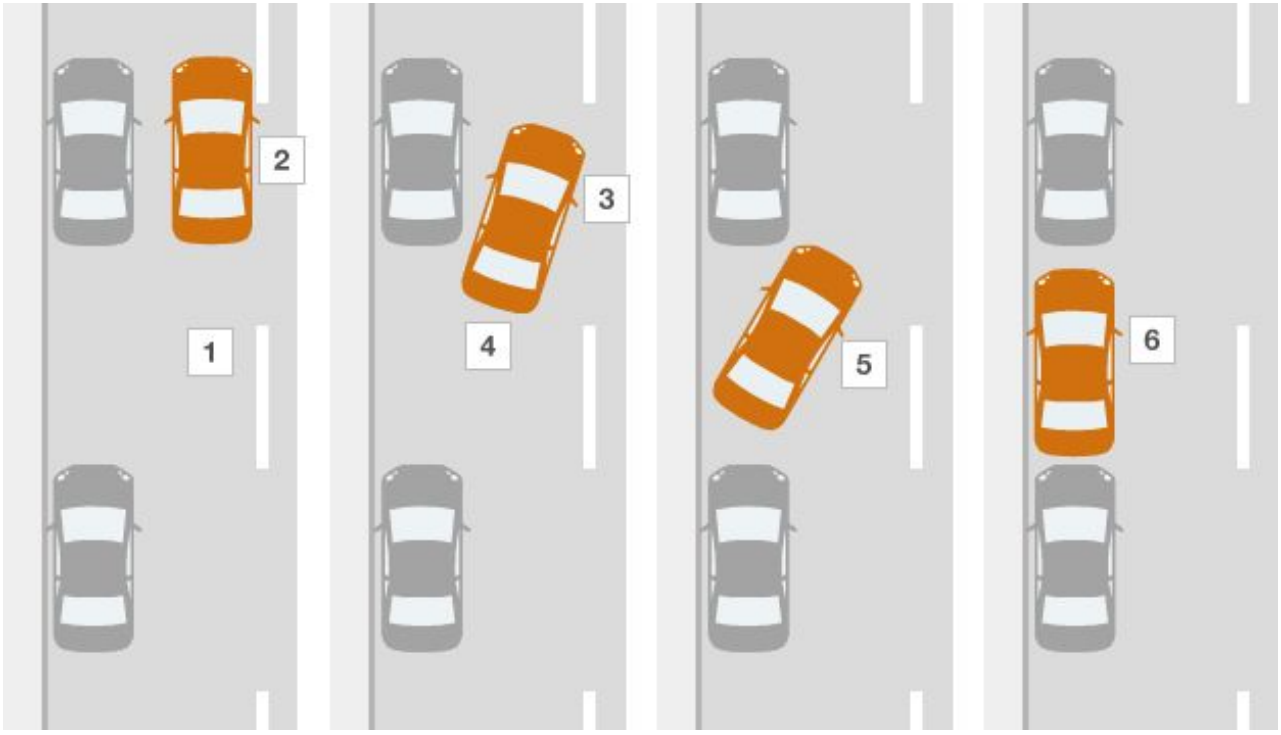
$T = (V, E) \leftarrow RRT^*(z_{init})$;
$T \leftarrow InitializeTree()$;
$T \leftarrow InsertNode(\varnothing, z_{start}, T)$;
for ($i = 0;\ i < N;\ i = i + 1$) {
    $z_{random} \leftarrow RandomSample()$;
    $z_{nearest} \leftarrow ClosestParent(T, z_{random})$;
    $z_{new} \leftarrow Drive(z_{random}, z_{nearest})$;
    if $ObstacleFreePath(z_{nearest}, z_{new})$ **then**
        $T \leftarrow InsertNode(z_{new}, T)$;
        $T \leftarrow Rewire(T, z_{nearest}, z_{new})$;
    end
}
Return T;

Algorithm 1: RRT*

Reeds-Shepp Paths

How do you get from one point to another?

Imagine you're building a program that parallel parks a car. How many different combinations of motions (turning the car, going straight) are required to guarantee that your program will park as efficiently as possible.



Reeds-Shepp Paths

What is a path anyways?

- There are two types of paths commonly discussed in path-finding; Holonomic and Non-Holonomic.
- A Non-Holonomic path has differential constraints.
- The types of motions must also be considered when defining what a path is.

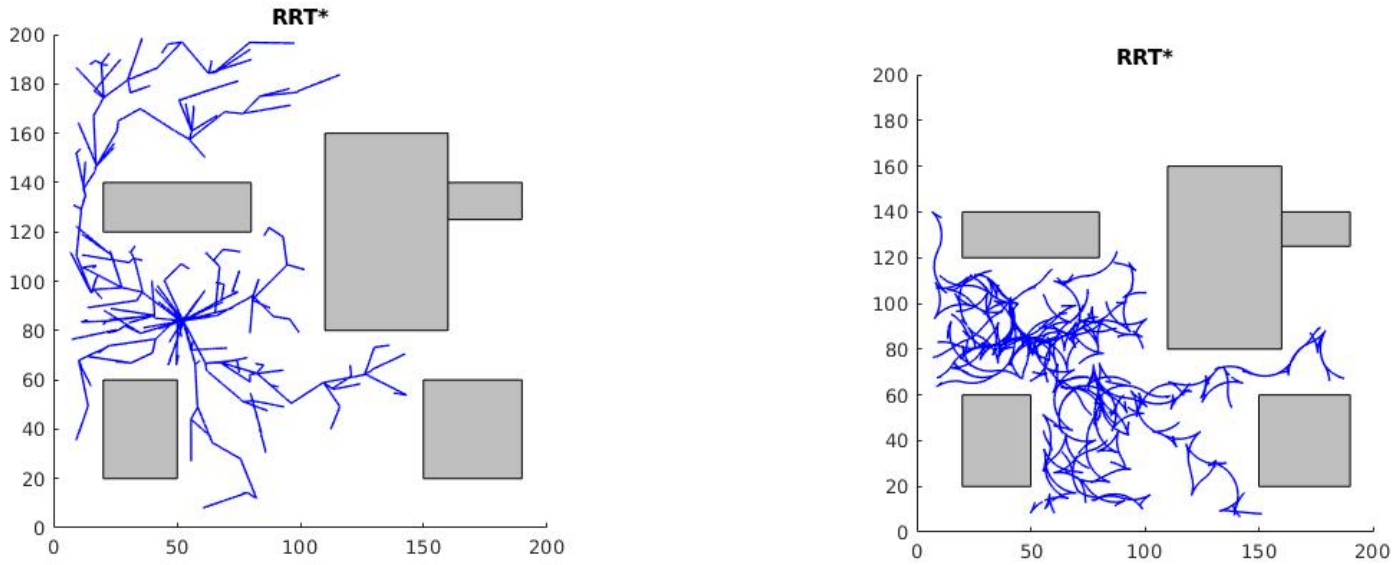


Figure 1: 200 Node Holonomic(Left) and Non-Holonomic(Right) Paths

Reeds-Shepp Paths (cont.)

Constructing Reeds-Shepp Paths

This problem was solved in 1990 by Reeds and Shepp. They concluded that there are 48 unique motion sequences that guarantee your path will be optimal. The motion sequences are constructed from up to 5 different combinations of curves and straight lines.

Base	α	β	γ	d
$C_\alpha \vert C_\beta \vert C_\gamma$	$[0, \pi]$	$[0, \pi]$	$[0, \pi]$	–
$C_\alpha \vert C_\beta C_\gamma$	$[0, \beta]$	$[0, \pi/2]$	$[0, \beta]$	–
$C_\alpha C_\beta \vert C_\gamma$	$[0, \beta]$	$[0, \pi/2]$	$[0, \beta]$	–
$C_\alpha S_d C_\gamma$	$[0, \pi/2]$	-	$[0, \pi/2]$	$(0, \infty)$
$C_\alpha C_\beta \vert C_\beta C_\gamma$	$[0, \beta]$	$[0, \pi/2]$	$[0, \beta]$	–
$C_\alpha \vert C_\beta C_\beta \vert C_\gamma$	$[0, \beta]$	$[0, \pi/2]$	$[0, \beta]$	–
$C_\alpha \vert C_{\pi/2} S_d C_{\pi/2} \vert C_\gamma$	$[0, \pi/2]$	-	$[0, \pi/2]$	$(0, \infty)$
$C_\alpha \vert C_{\pi/2} S_d C_\gamma$	$[0, \pi/2]$	-	$[0, \pi/2]$	$(0, \infty)$
$C_\alpha S_d C_{\pi/2} \vert C_\gamma$	$[0, \pi/2]$	-	$[0, \pi/2]$	$(0, \infty)$

Symbol	Gear: u_1	Steering: u_2
S^+	1	0
S^-	-1	0
L^+	1	1
L^-	-1	1
R^+	1	-1
R^-	-1	-1

Figure 2: Reeds-Shepp Curve Combinations and Motion Primitives

Putting it all together

With all of our new path-finding gadgets, we are ready to generate non-holonomic paths for our self-driving cars.

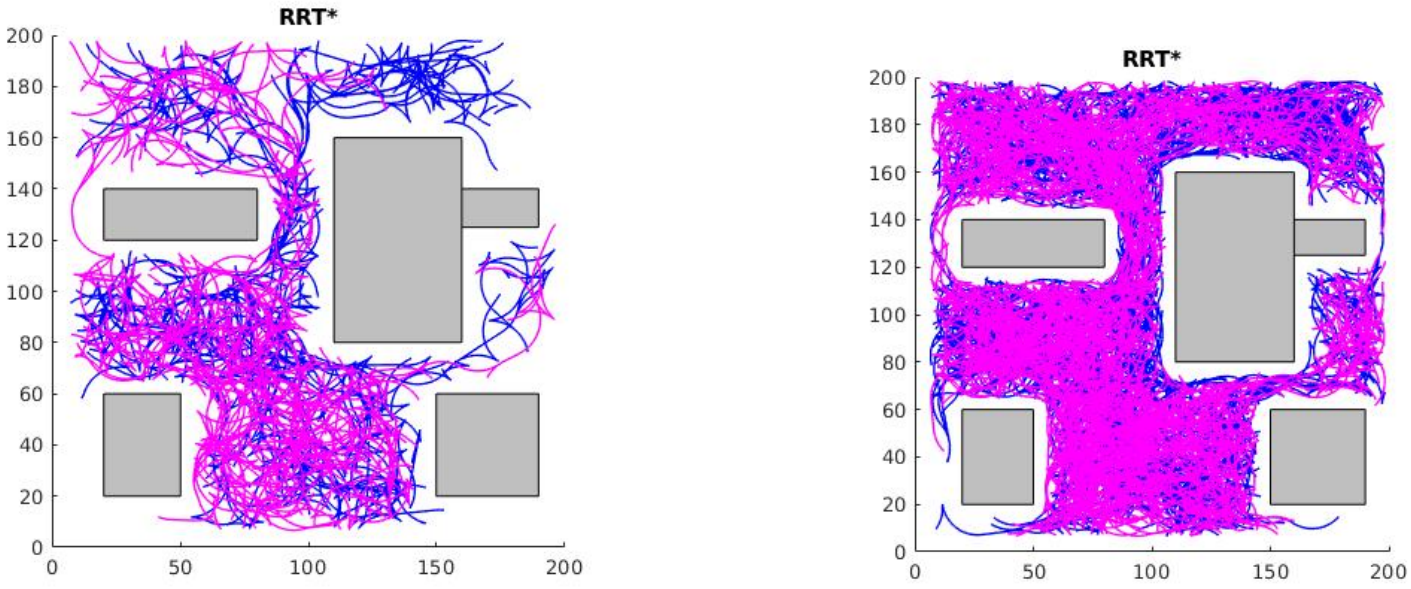


Figure 3: Reeds-Shepp RRT* Tree Visualization with 500 and 1500 nodes respectively

The next question is, once we have it built how can we make it fast?

Optimization

The Battle for Efficiency

- RRT* and Multiple-Car RRT* are really slow.
- To make it faster, we had to be more clever than the algorithm.

K-D Tree

A K-D Tree is an efficient way of storing points so that you can quickly search for the nearest points around a specified tree node.

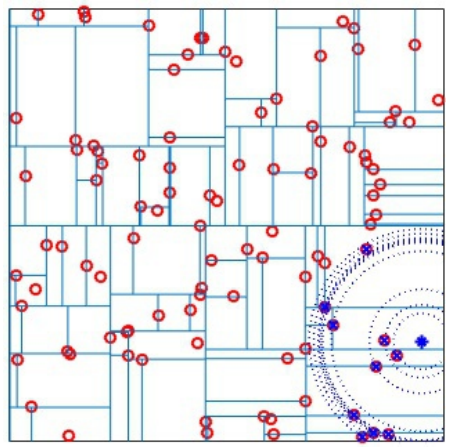


Figure 4: 2-Dimensional K-D Tree using a nearest neighbor search

Algorithmic Progress

The Journey From MatLab to C++

The code that we had previous to this semester was written in MatLab, therefore it was naturally really slow. To speed up the code, we converted everything into C++.

Improvements in time

- In the previous semester we spent 66 hours computing a path for two cars, and the path ended up being sub-optimal.
- This semester we were able to compute a similar path in under a second.

Optimal Path Generation

The original MatLab code had been selecting an optimal motion sequence from four options, when the total number of options required to guarantee our path would converge to the optimal solution was 48, which we saw earlier. During this semester we implemented all 48 motion sequences into our C++ code.

Algorithmic Goals for the Future

- Implement a K-D Tree that computes actual distance, rather than an approximation (Warning: Involves Sub-Riemannian Geometry).
- Implement different variants of RRT*.
- Make the code even faster.

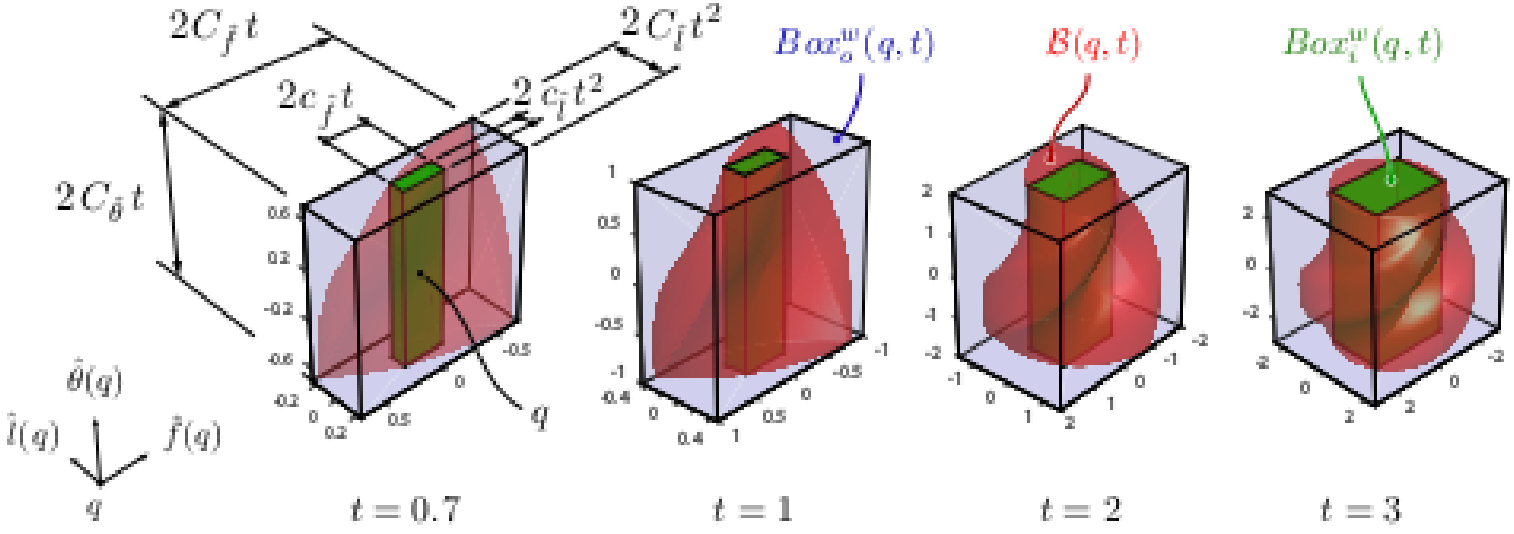


Figure 5: . Reachable sets and bounding boxes for a Reeds-Shepp vehicle around a configuration q and for different values of t. (Frazzoli)

Goal and Process

The goal of this research project is to represent the motion path generated by the RRT* path in a real world system. (Not Trivial)

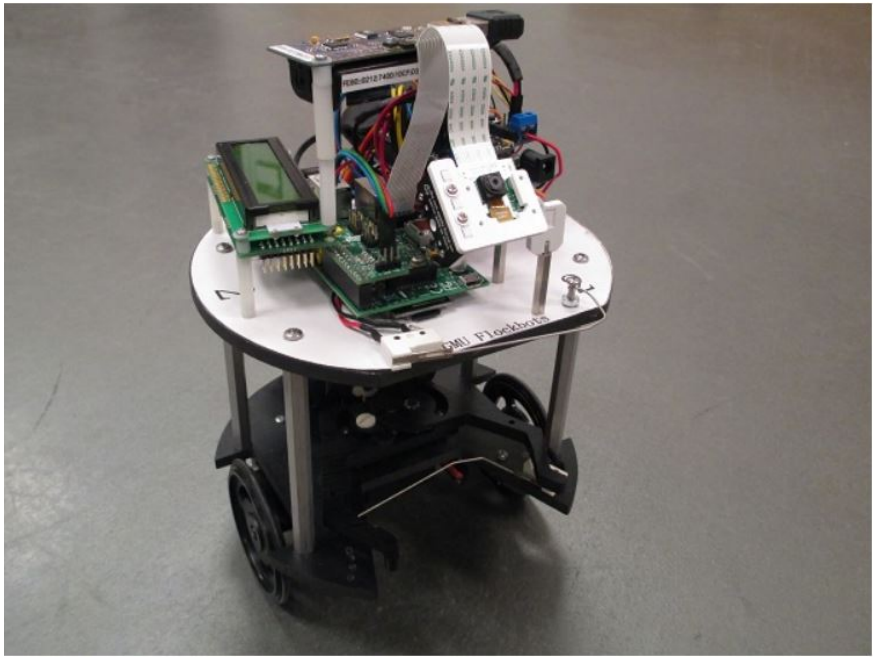


Figure 6: Image of Flockbot and AR tag

FlockBots: Wandering Robot

Raspberry Pi

The raspberry pi functions as the communication point between the laptop code and Arduino. It sends commands to the Arduino based on the commands it is sent from a laptop.

Arduino

The Arduino is a low cost microcontroller and it is used to operate the motor. The boards are equipped with (I/O) pins that interface with a system to control various aspects.

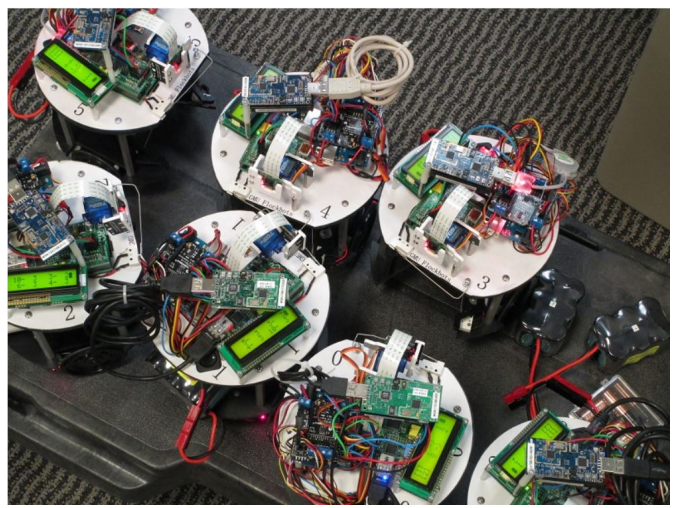
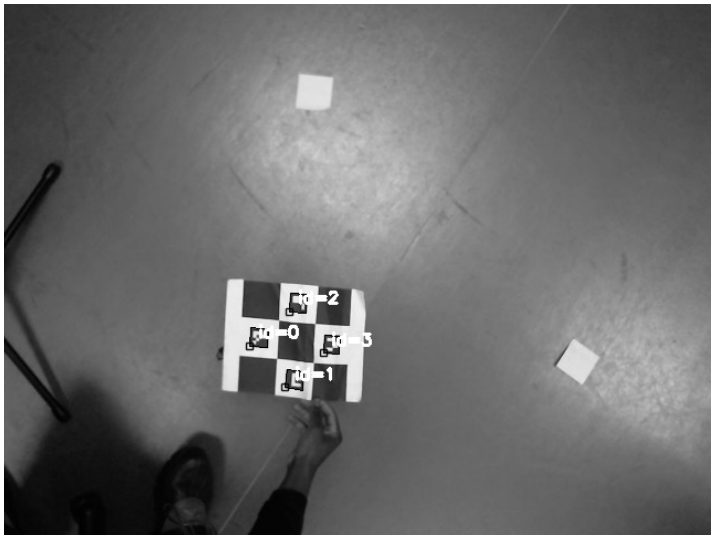


Figure 7: Image of multiple Flockbots resting

AR Tracking System: Where are the Robots



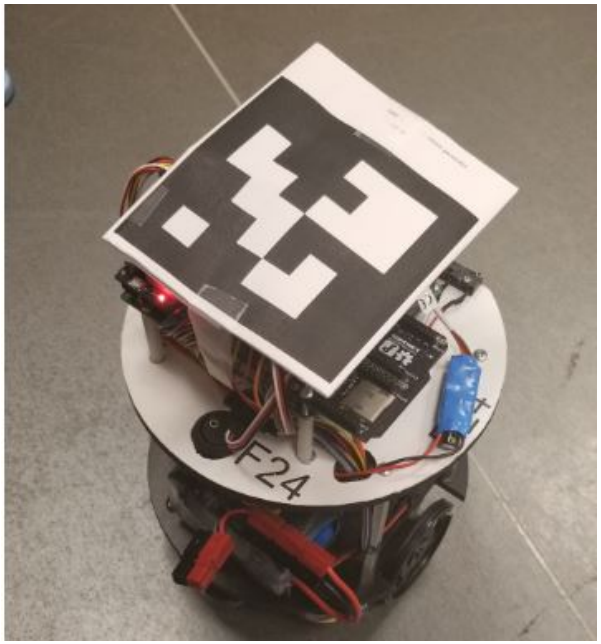


Figure 8: Image of webcam calibration and AR tag taped to Flockbot

Basic Information

The ideal was to set up a global reference frame so that we could always know the current FB position and orientation. In order to connect the Flockbot's code and AR tracking code we had to use a TCP/IP server client connection.

Early Progress

Status

Initially the Flockbots rotated towards the heading to the next point in a path, then it would move along that line. The AR tracking system was also completed but it was not implemented to give feedback on Flockbot motion.

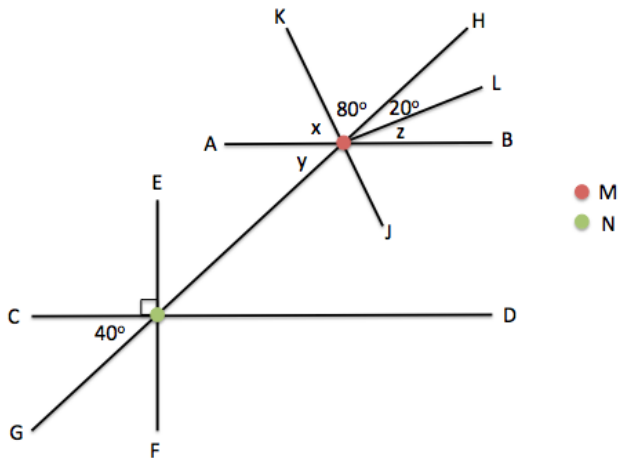


Figure 9: Image of previous motion planning path and AR tag

Key Problems and Solution

Problem: Old Flockbot Motion Model

- The old model described does not account for a minimum turning radius.
- The old model makes it hard to control velocity.
- Requires flockbot to come to a complete stop at every point in the path.

Solution: Closed Loop Feedback Control System

$$V = (VR + VL)/2$$

$$\omega = (VR - VL)/L$$

$$R = L(VL + VR)/2(VR - VL)$$

Key Problems and Solution Cont.

Problem: Latency

Latency is a delay before the transfer of data following an instruction for its transfer. The Flockbot would wait about .2 to .4 seconds before it would perform a command sent.

Solution: Reassemble Path

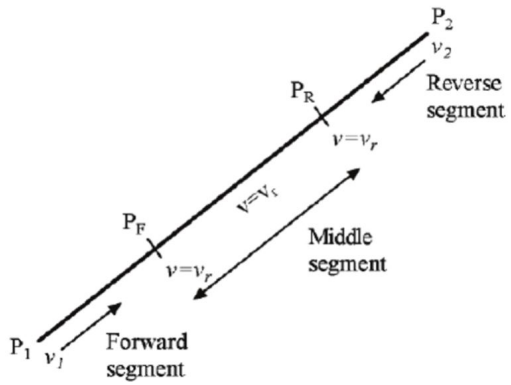


Figure 10: Image of line segments describing path

Block Diagram of System

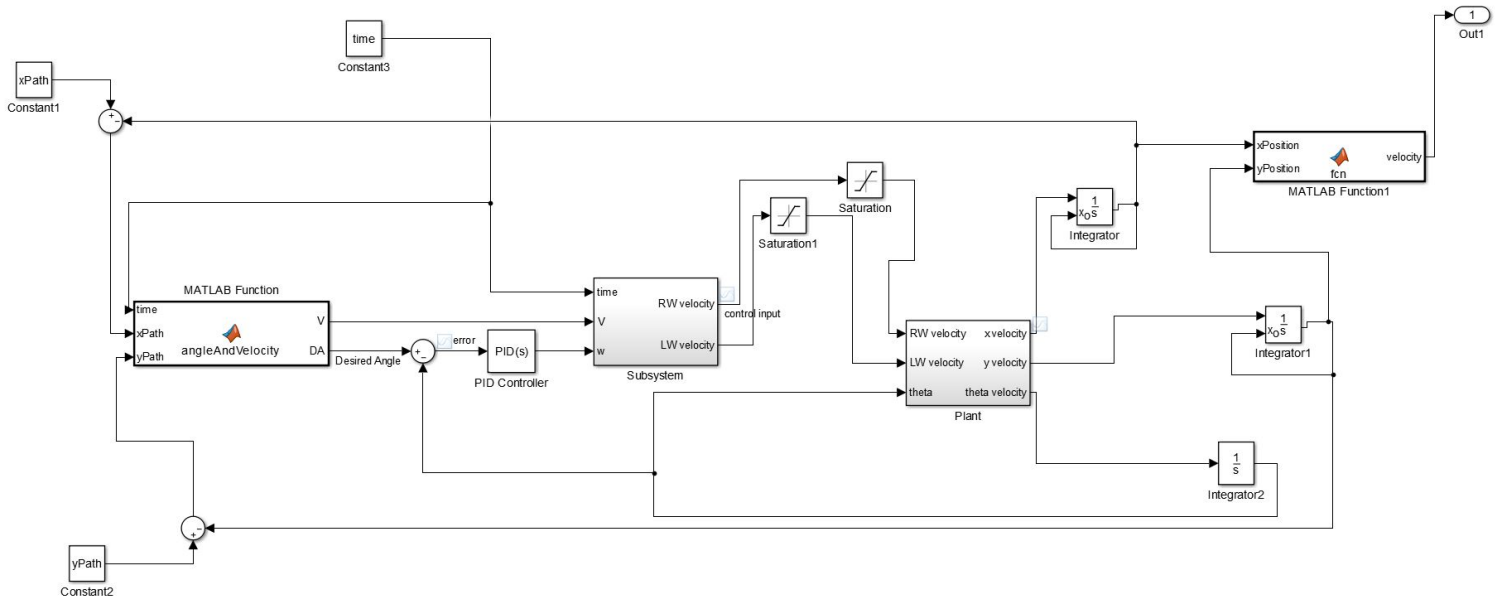


Figure 11: This is a model of the Flockbot's kinematic motion along a given path

Real World Future Implementation Goals

- Edit code to account for multiple flockbots
- Push most of the code onto the Arduino to decrease the effects of latency
- Tune model to account for smaller distances between points