Adaptive Pure Pursuit

Ethan Frank Kimberlee I. Model Paul Gehman

Dawgma Robotics

September 15, 2019

Table of Contents

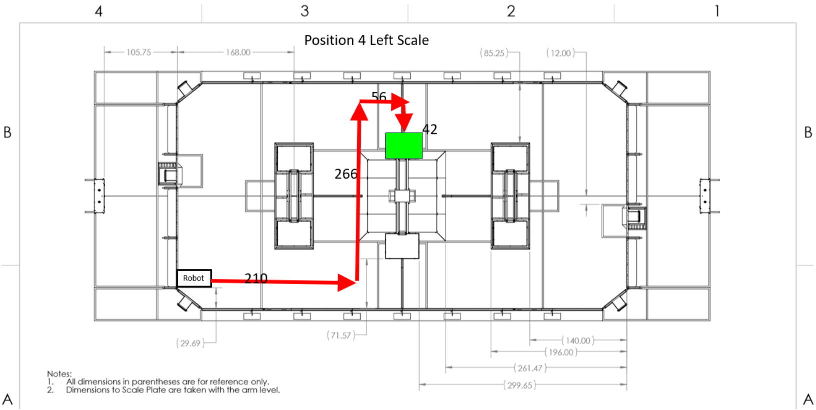
Introduction

- An overview of Pure Pursuit as used by team 1712 during the 2018 season
- Architectural and Mathematical overview. Staying away from code
- Please raise your hand to ask questions as you have them.

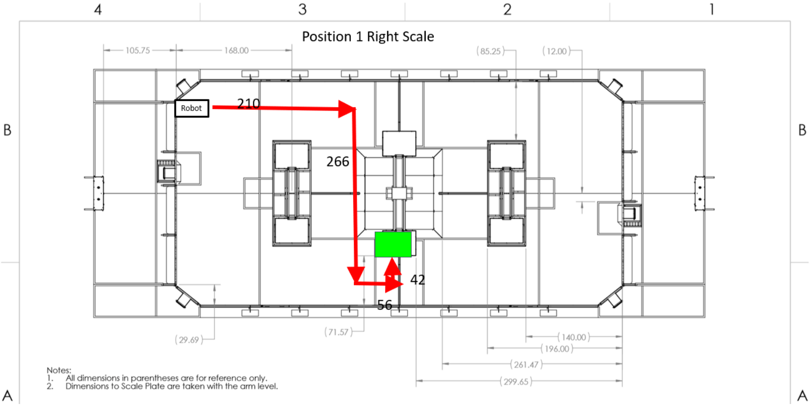
Brief History

- 16 Possible Paths
- Pure Pursuit Algorithm
- File-Encoded Routines

16 Possible Paths



16 Possible Paths



What is Pure Pursuit

- Path Follower

What is Pure Pursuit

- Path Follower
- Path Generator

What is Pure Pursuit

- Path Follower
- Path Generator
- JSON based File Encoding

What is Pure Pursuit

- Path Follower
- Path Generator
- JSON based File Encoding
- **A bunch of Mathematics Expressions in a trench coat**

What is Pure Pursuit

- Path Follower
- Path Generator
- JSON based File Encoding
- **A bunch of Mathematics Expressions in a trench coat**

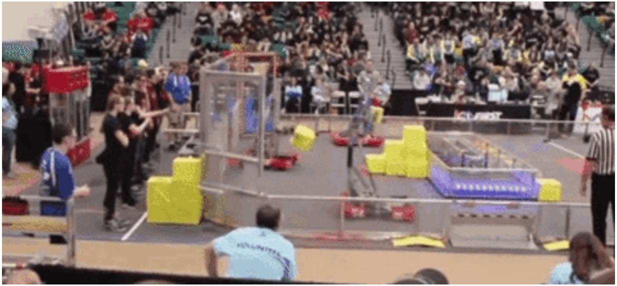
Analogy

Think of path generation as **drawing a virtual line**.
And think of path following as **walking along the virtual line**.

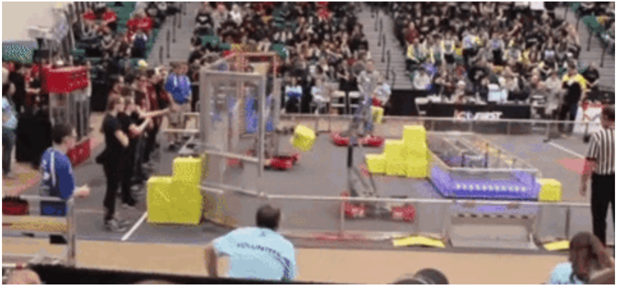
Odometry

- Use sensors to track the location of the robot
- Plot on a Cartesian Plain
- Pure Pursuit requires accuracy

Importance of Odometry



Importance of Odometry



- NavX failed
- Robot attempting to turn slightly left
- No input causes RoboRIO to believe that it is not turning at all
- Increasing control to attempt left turn

Sensors involved

- Rotary Encoders (one on each side of the drive train)
- NavX MXP for accurate angle

Sensors involved

- Rotary Encoders (one on each side of the drive train)
- NavX MXP for accurate angle
- Preset starting location
- Long term summation of changes to the position

Odometric Calculations

- D_l and D_r Distance traveled by the left and right wheels since previous iteration
- A angle robot is facing relative to the field.
- X_{prev} and Y_{prev} location from previous calculation
- X and Y location of the robot relative to starting position.

$$D = (D_l + D_r)/2 \tag{1}$$

$$X = X_{prev} + D * cos(A) \tag{2}$$

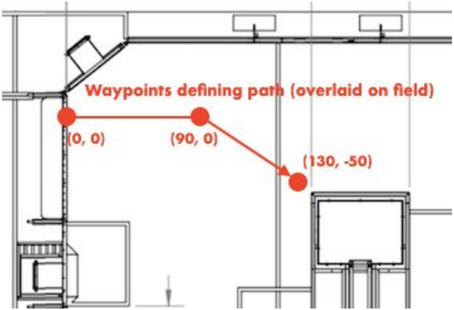
$$Y = Y_{prev} + D * sin(A) \tag{3}$$

Generating Paths

- Define start point, destination and way points
- Inject additional way points
- Smooth the path
- Curves and maximum speed along the path.

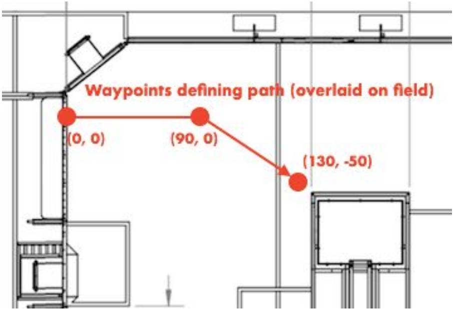
Define Way points

- Path Drawer Tool

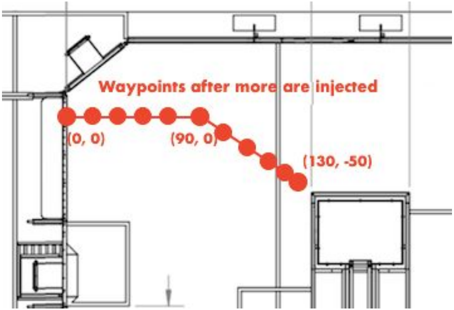


Define Way points

- Path Drawer Tool
- Start Point: one of four set positions.
- Way Points, to avoid obstacles.
- Destination.



Injecting Points



Injecting Points

- Path is a collection of way points
- Also a collection of line segments
- To inject points drop breadcrumbs at regular intervals

Injecting Points

- Path is a collection of way points
- Also a collection of line segments
- To inject points drop breadcrumbs at regular intervals

```
interval := the distance between injected points;
segments := the lines between the way points;
newpoints := [ empty list of points ];
for each segment in segments:
    walker := segment.start;
    while (walker < segment.end):
        newpoints.append(walker);
        walker.advanceOnLine(segment, interval);
```

Injecting Points

- Path is a collection of way points
- Also a collection of line segments
- To inject points drop breadcrumbs at regular intervals

```
interval := the distance between injected points;
segments := the lines between the way points;
newpoints := [ empty list of points ];
for each segment in segments:
    walker := segment.start;
    while (walker < segment.end):
        newpoints.append(walker);
        walker.advanceOnLine(segment, interval);
```

- Dawgma used 6 inch sub-segments

Smoothing the Path

- Dawgma used the same algorithm as Team 2168
- Each point is a weighted combination of:
  - the original point
  - the midpoint of the previous and next points
- Repeats calculation of small increments
- Finishes when calculation results in sufficiently small changes (Tolerance)

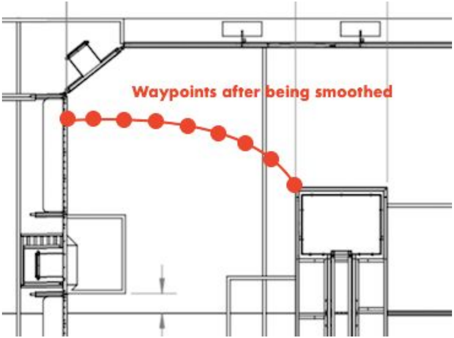
Smoothing the Path

```
let og, nc := original path, smoothed path (copy og);
let a, b := original weight, smoothing weight;
let t, c := tolerance, 0.0;
while(c >= t):
    c := 0;
    for each x, y in nc, og:
        let tmp := nc;
        nc +=
```

$$a(og - nc) + b(nc_{prev} + nc_{next} - 2(nc))$$

```
        c += absval(tmp - nc);
```

Smoothing the Path



Smoothing Alternatives

- Quintic Splines for smoothing way points

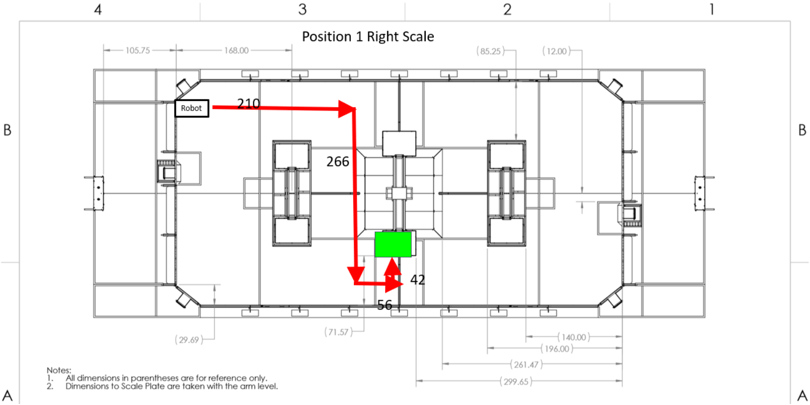
Smoothing Alternatives

- Quintic Splines for smoothing way points
- Bezier Curves for directly generating a path

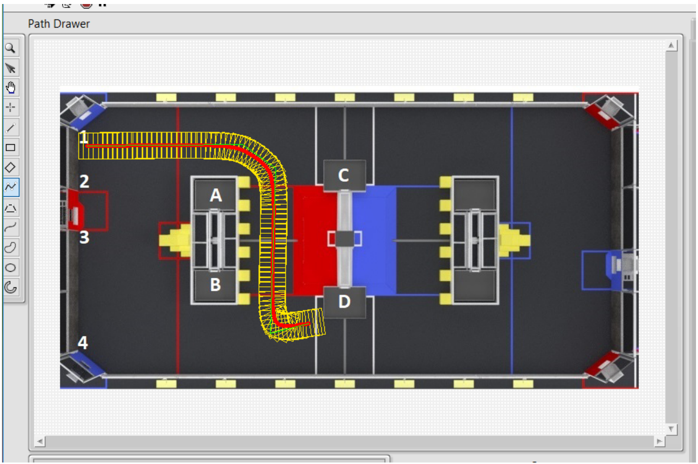
Smoothing Alternatives

- Quintic Splines for smoothing way points
- Bezier Curves for directly generating a path
- Generate some points by hand

Visualizing Paths



Visualizing Paths



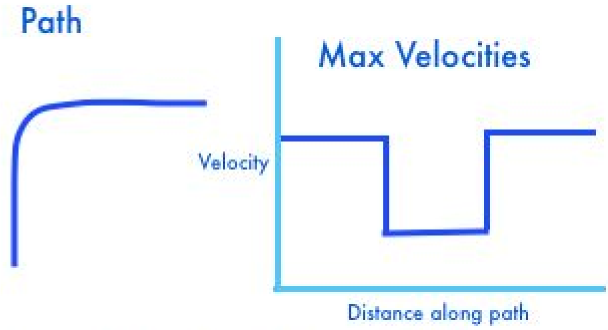
Curvatures and Velocities

- Story time.

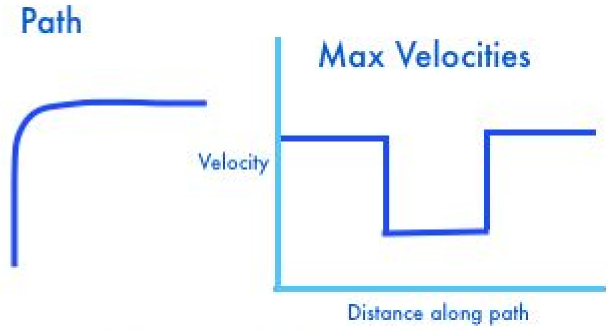
Curvatures and Velocities

- Story time.
- Slow down around turn
- Determine the curvature (rate of turn)
- How much to slow down
- Check the white paper for details on these steps

Velocity Profiles

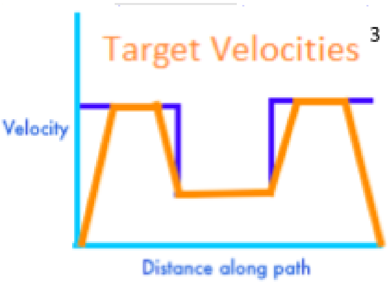


Velocity Profiles

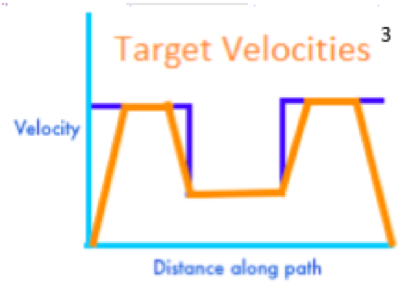


- Slow the maximum velocity during turns to prevent tipping
- Introduce sudden acceleration

Velocity Profiles

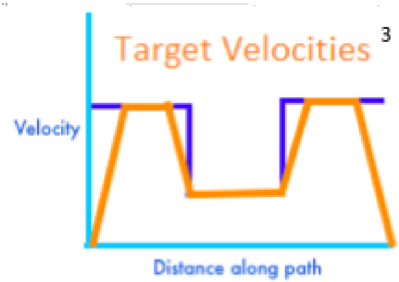


Velocity Profiles



- Decelerate **before** the curve
- Re-accelerate **after** the curve
- Changing Velocity within the curve could cause tipping

Velocity Profiles



- Decelerate **before** the curve
- Re-accelerate **after** the curve
- Changing Velocity within the curve could cause tipping
- Each point has a target velocity and target acceleration

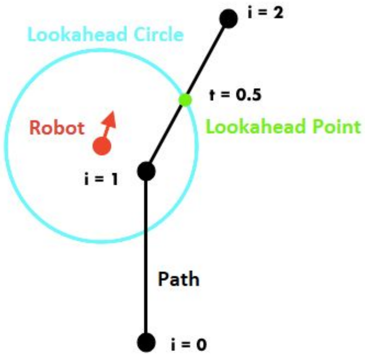
Encoding Velocity and Acceleration

- Zero (0) max velocity at the starting line

Encoding Velocity and Acceleration

- Zero (0) max velocity at the starting line
- Rate limit acceleration during runtime instead of path generation
- Use max velocity of next point instead of current point

Following the Path



Following The Path

- Know the current location using odometry
- Find the closest way point along the path
- Find the lookahead point
- Drive in an arc from current location to lookahead point
- Calculate the target left and right wheel velocities
- Use a control loop to achieve the target left and right wheel velocities

Closest Point

- cur the current location of the robot
- min the point with minimal distance from cur
- prev the previous point of minimal distance
- distance the Cartesian distance formula:
  $distance(A, B) \rightarrow \{\sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}\}$

```
min := prev;
for point in path from prev to end:
    if(distance(min, cur) > distance(point, cur)):
        min := point;
```

Lookahead Point

- Robot attempts to drive towards this point
- Follows the path as the point keeps moving forward

Lookahead Point

- Robot attempts to drive towards this point
- Follows the path as the point keeps moving forward
- `lookahead distance` is the distance in front of the robot where the lookahead point is calculated
- Intersection of a "lookahead" circle with the path.
- two intersection points, choose the one farther in front of the robot

Lookahead Point

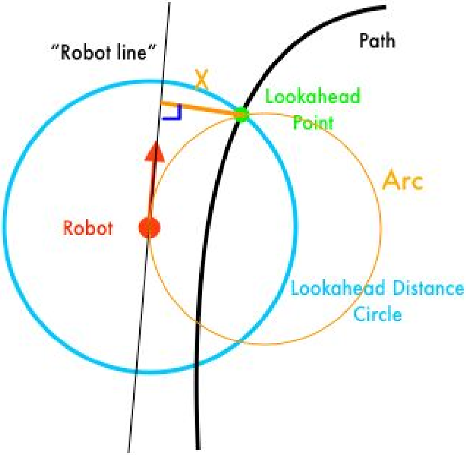
```
loc := the Robot's current location (odometry);
d := the Lookahed Distance;
n := the nearest point in the path;
segments := the lines between points in the path;
intersections := [ empty list ];
for each segment in segments from n to end:
    a, b := intersection(segment, loc, d);
    if(a != null): intersections.append(a);
    if(b != null): intersections.append(b);
lookahead_point := segments.last();
```

Check the white paper for details on the intersection of a circle and line.

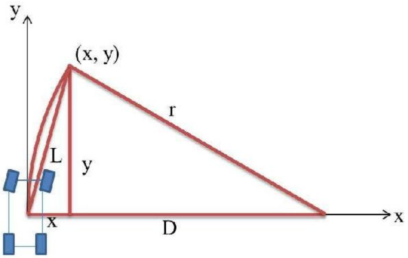
Choosing a Lookahead Distance

- Shorter lookahead for curvy paths
- Longer lookahead for smoothing a bit
- Dawgma used a distance of 12 to 25 inches
- Consider varying the lookahead distance within the path

Arc Towards the Lookahead Point



Curvature of the Arc



- Robot at the origin traveling along the Y-axis
- (X, Y) is the lookahead point
- L is the direct path to the lookahead point
- but we want to drive the arc around L
- r is the radius of the arc

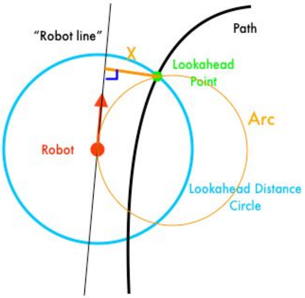
Stemming from the Pythagorean Equation

$$L = \sqrt{X^2 + Y^2} \tag{4}$$

$$r = L^2/(2X) \tag{5}$$

Curvature (C) is $1/r$

On the Field



- P is the lookahead point
- R is the robot location (from odometry)
- A is the robots angle

The robot is traveling along the line:

$$0 = -tan(A)x + y + tan(A)R_x - R_y \quad (6)$$

We can calculate X as:

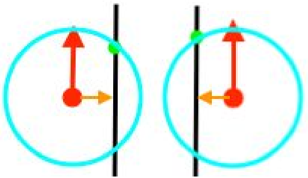
$$X = \frac{|-tan(A)P_x + P_y + tan(A)R_x - R_y|}{\sqrt{-tan(A)^2 + 1}} \quad (7)$$

and Y as:

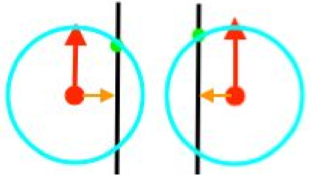
$$Y = \sqrt{\sqrt{(P_x - R_x)^2 + (P_y - R_y)^2} - X^2} \quad (8)$$

From here we can use curvature as calculated earlier.

Which direction to turn



Which direction to turn



The direction to turn can be taken as the sign of the vector cross product:

$$Red \times Orange \tag{9}$$

Left if negative, right if positive.

Robot Velocity

- As fast as the robot can go

Robot Velocity

- As fast as the robot can go
- Without falling over.

Robot Velocity

- As fast as the robot can go
- Without falling over.
- Early in the season, we traveled the entire path at the maximum velocity of the sharpest turn.
- Later in the season, we added the ability to change velocities throughout the path.
- At velocity transitions we would calculate an acceleration or deceleration.

Wheel Velocities

Have:

- Target Velocity (V) of the robot
- Target curvature (ω) of the robot
- Track Width (T) of your robot

Want:

- Left wheel velocity (L)
- Right wheel velocity (R)

Wheel Velocities

Mathematical Model of a Tank Drive:

$$V = (L + R)/2 \tag{10}$$

$$\omega = (L - R)/T \tag{11}$$

$$V = \omega/C \tag{12}$$

Now we isolate L and R

$$L = V\frac{2 + CT}{2} \tag{13}$$

$$R = V\frac{2 - CT}{2} \tag{14}$$

Controlling the Wheels

- Combined Feed Forward and Feed Backward Controller
- Individually control left and right wheel speed based on Rotary Encoder velocities.
- PWM output
- Desired Velocity and Desired Acceleration

Feed Forward

- K_v proportional constant for target velocity (V)
- K_a proportional constant for target acceleration (A)

$$FF = K_v * V + K_a * A \tag{15}$$

Feed Backward

- Corrects error between actual velocity (M) and target velocity.
- K_p is the feed backwards proportional constant.

$$FB = K_p * (V - M) \tag{16}$$

Combined to get PWM output (O):

$$O = FF + FB = K_v * V + K_a * A + K_p * (V - M) \tag{17}$$

Choosing Proportional Constants

- start with a straight line path
- Set K_v approximately equal to $1/V_{max}$
- Set K_a and K_p to zero (0)
- Adjust K_v until a target velocity and the actual velocity match.

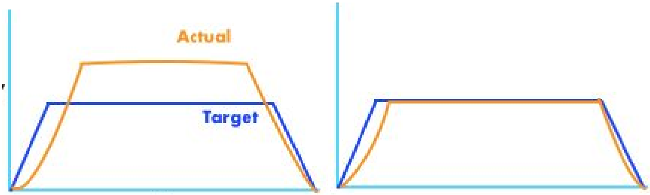


Figure: graphs of velocity vs. time. Left is at the start. right is with K_v tuned

Choosing Proportional Constants

- Set K_a to 0.002
- Adjust K_a until the acceleration lines on the graph are roughly straight

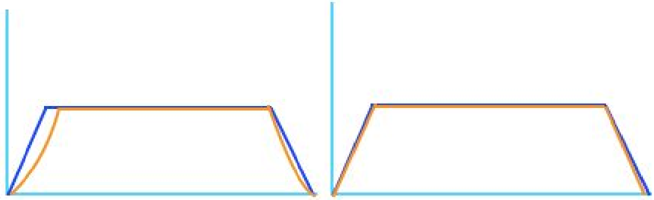


Figure: graphs of velocity vs. time. Left is at the start. right is with K_v tuned

Choosing Proportional Constants

- Set K_p to 0.01
- Adjust K_a as needed until the actual line covers the desired line
- Too much feed backwards will cause “jitteryness”

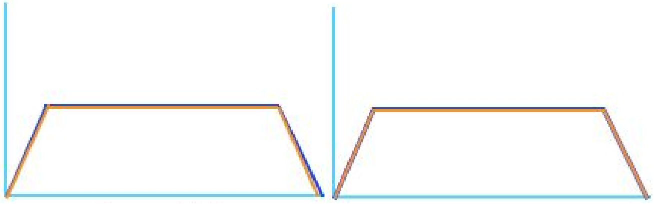
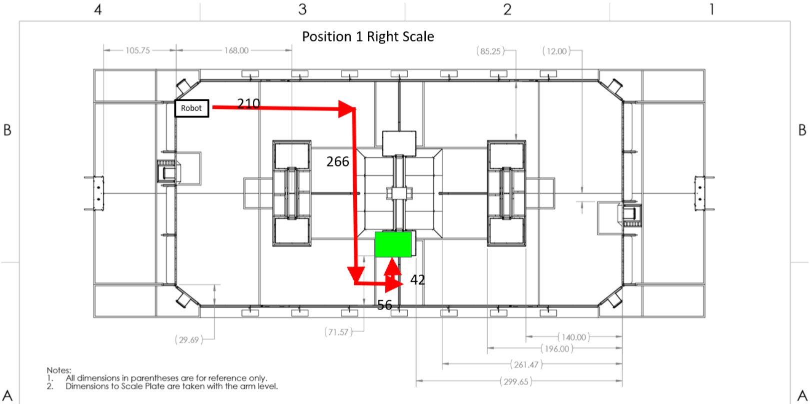


Figure: graphs of velocity vs. time. Left is at the start. right is with K_v tuned

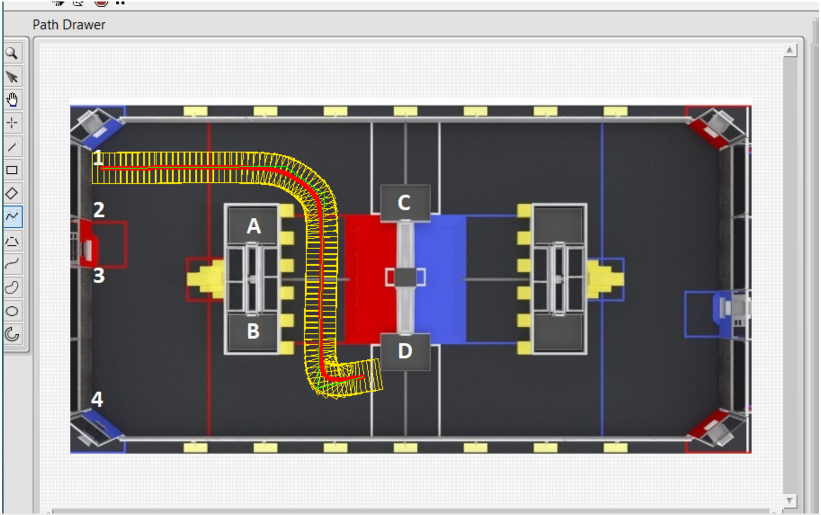
Visualizing a Path

- PowerPoint with arrows
- Path Drawing Tool
- Path Simulation Tool

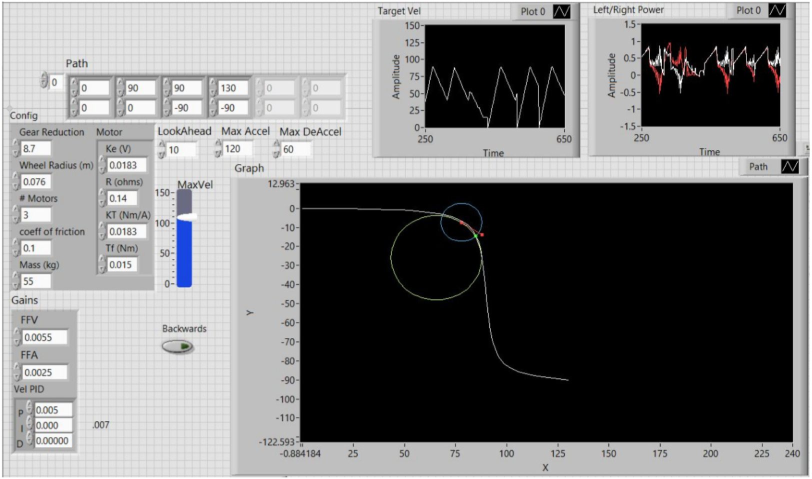
Visualizing Paths in PowerPoint



Path Drawing Tool



Path Simulation Tool



Contact and Links

- The 1712 Pure Pursuit White Paper: https://www.chiefdelphi.com/media/papers/3488
- Dawgma's 2018 Code Repository: https://github.com/Dawgma-1712/new-FRC-2018
- Chief Delphi Discussion: https://www.chiefdelphi.com/forums/showthread.php?t=166214
- Dawgma Email: frc1712@gmail.com
- My Email: kimee.i.model@gmail.com