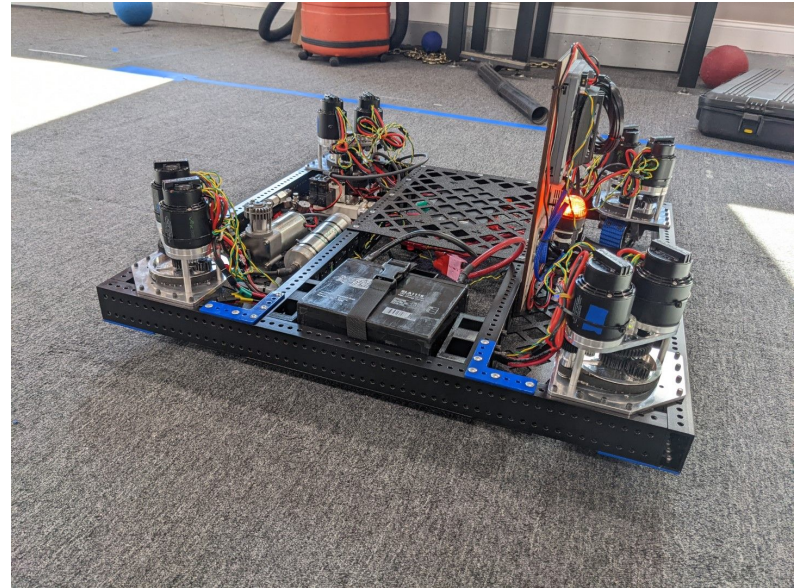

Swerve Drive and Autonomous Programming

The Highlanders



Breaking Down the Problem

1. Single Module Control
 - a. Position angle control
 - b. Velocity speed control
 - c. Optimizing module performance
2. Integrating multiple modules together
 - a. Strafing
 - b. Turning
3. Referencing the robot to reality
 - a. Coordinate Systems
 - b. Kinematics Model
 - c. Odometry
4. Pathing
 - a. Basic paths and motions
 - b. Advanced Pathing
 - i. Constant acceleration interpolation
 - ii. Simulation
 - iii. Visualizing and debugging paths





**Individual
Module
Control**

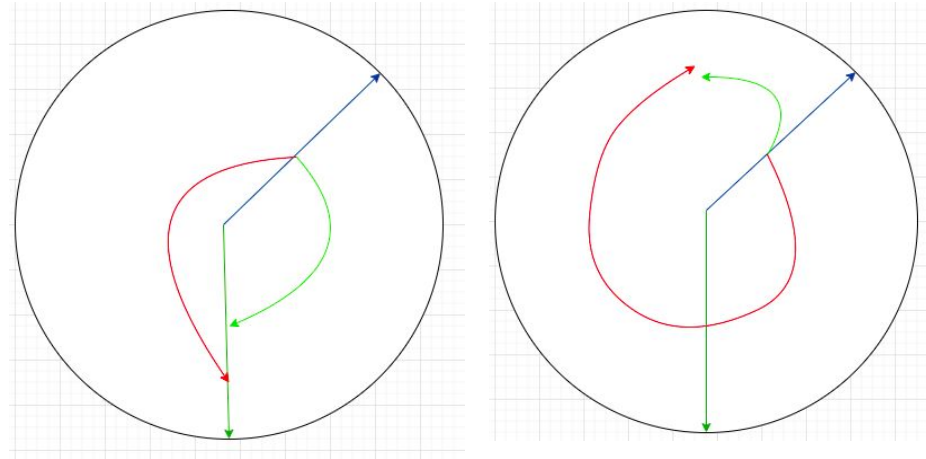


Programming a Swerve Module - Angle Motor

- Controlled with position PID related to angle of wheel
- Need to convert angle(in radians) to encoder ticks
 - $\text{Encoder Ticks} = (\text{angle} * \text{FalconTicsPerRotation} * \text{SteerGearRatio}) / (2\pi)$
- Use Position control on motor controller to set position
- Use absolute encoders at the start to calibrate the initial module position.
 - Ideally swerve modules should be aligned at the start of the match. However sometimes this doesn't happen. It is nice to have a backup just in case.

Optimizing Direction / Angle Selection

- For a given angle and direction, there are four possible ways a swerve module can turn and drive to achieve that angle and direction:
 - Angle rotates clockwise, wheel drives forward
 - Angle rotates counterclockwise, wheel drives forward
 - Angle rotates counterclockwise, wheel drives backwards
 - Angle rotates clockwise, wheel drives backwards
- This often leads to issues, particularly when rolling over the wheel position (359- 0). As calculating angle and speed independently will cause the wheel to spin 359 degrees instead of 1.
- To solve this problem, pick the movement solution that minimizes the amount the module needs to spin.



Programming a Swerve Module - Drive Motor

- Controlled with a velocity PID running on motor controller
- Need to convert a velocity(in m/s) to encoder ticks/100ms(native units)
 - $\text{Ticks/s} = (\text{Velocity} * \text{DriveGearRatio} * \text{FalconTicksPerRotation}) / (\text{WheelCircumference})$
 - $\text{Ticks/100ms} = (\text{Ticks/s}) / 10$
- Use velocity control on motor controller to set velocity on motor

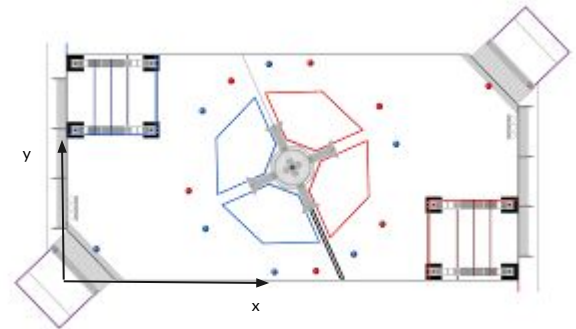




**Integrating
Multiple
Modules**

Velocity Drive - Overview

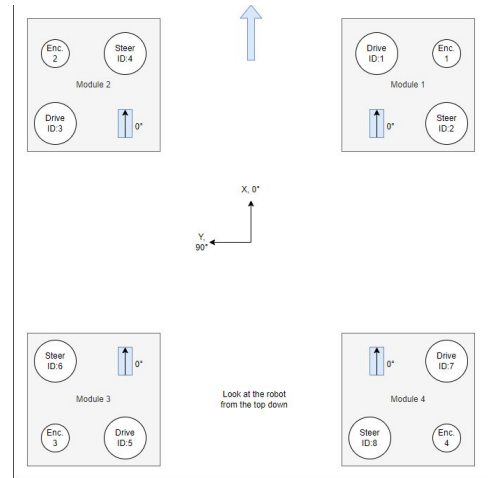
- The velocity drive is a method that runs for each swerve module and converts a desired x , y , and theta velocity into individual angles and speeds for each module.
 - Inputs: Desired X , Y , Theta Velocities.
 - Outputs: Module wheel speed and direction for each module
- Note, we are only providing the module a vector it should spin towards. It is up to the module to decide how best to achieve that motion.
- Positive X - towards opposing alliance wall
- Positive Y - Towards hangar zone
- Theta - Direction robot is facing on the field
- X and Y velocities are given in m/s . Theta velocity is given in radians/second



Velocity Drive - Strafing

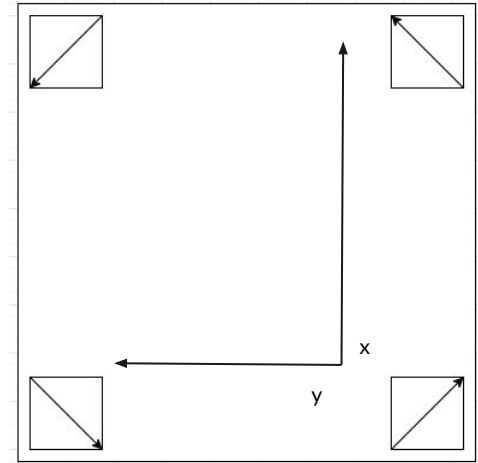
- We want to create a vector that represents the strafing components of a robots motion
- Wheel angle = $\arctangent2(y,x)$
 - To have field centric control of swerve, subtract the imu angle from the calculated wheel angle*
 - For most programming languages, arctangent only provides a mathematical tangent with a range of $-\pi/2$ to $\pi/2$. Use atan2 to get a coordinate conversion that spans all quadrants.
- To include this input into our x and y velocities, we need to calculate the magnitude of the two velocities and then split it into x and y components
 - Magnitude = $\sqrt{x^2 + y^2}$
 - StrafeX = Magnitude * $\cos(\text{wheelAngle})$
 - StrafeY = Magnitude * $\sin(\text{wheelAngle})$

* Note: We are using a right handed coordinate system with the Z axis pointed up. This means that positive rotation is defined to be counterclockwise. FRC IMU's such as the NavX use a left handed coordinate system, so positive rotation is defined to be clockwise. There are a number of implications of this discrepancy, but for our purposes here, it is sufficient to negate the value of the yaw rotation when reading from the sensor.



Velocity Drive - Turn

- We need to create a turn vector for each module.
 - The turn vector is the vector at which the module will provide the most rotation for the robot in the counterclockwise direction.
 - Turn vectors are constant and do not change as the robot moves and spins.
- To start we need to convert our theta velocity into a tangential ground velocity of the module in meters/s
 - $\text{AngleVelMetersPerSec} = \text{thetaVel} * \text{robotRadius}$
 - For the purpose of this calculation, the robotRadius is the distance between the center of the robot and the center of the wheel on the module.
- We need to multiply this velocity by a vector that determines the direction of the wheel
- For a square swerve drive each turn vector will be the AngleVelMetersPerSec multiplied by:
 - Front Right: $(\sqrt{2}/2, \sqrt{2}/2)$
 - Front Left: $(-\sqrt{2}/2, \sqrt{2}/2)$
 - Back Right: $(\sqrt{2}/2, -\sqrt{2}/2)$
 - Back Left: $(-\sqrt{2}/2, -\sqrt{2}/2)$
- The final wheel speed for any given wheel inputs then becomes:
 - $\text{SpinX} = \text{thetaVel} * \text{robotRadius} * \text{turnVectorX}$
 - $\text{SpinY} = \text{thetaVel} * \text{robotRadius} * \text{turnVectorY}$





Velocity Drive - Combining Strafe and Turn Vectors

- Now that we have two vectors, a strafe vector and a spin vector, we can add them together
 - $\text{CombinedX} = \text{StrafeX} + \text{SpinX}$
 - $\text{CombinedY} = \text{StrafeY} + \text{SpinY}$
- Here we have a vector that represents the magnitude and direction of each swerve module
- Next we will find the magnitude of this vector(wheel velocity), as well as the direction(angle)
 - $\text{Magnitude} = \sqrt{x^2 + y^2}$
 - $\text{Direction} = \text{arctangent2}(y,x)$
- Here we have a velocity that we can pass to our module control method which will then compute the optimal way of achieving the given magnitude and direction.



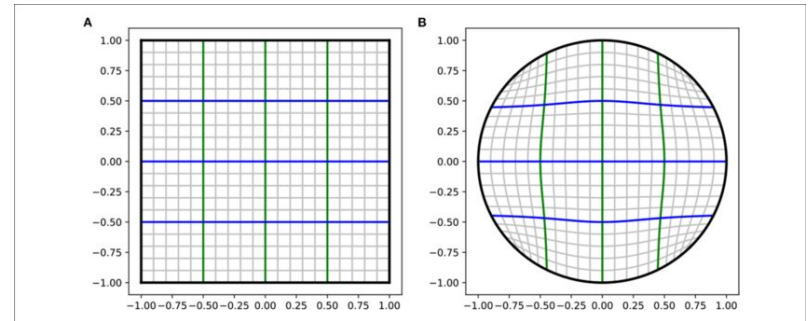
Teleoperated Driving

- At this point, driving the swerve is as simple as supplying the robot the desired X, Y and theta mapping to achieve a desired motion.
- This lends itself quite nicely to a standard Xbox controller. Simply map the desired joysticks to X, Y and Theta.
 - Joysticks read from -1 to 1 on their axis, multiply these by the top speed of the robot to get corresponding values of X and Y and theta.
- So long as the input X, Y and Theta do not exceed the max speed of the module, the robot should move in a consistent manner.
- Since the system is built on velocity PID's the robot will be inherently voltage compensated. However, the PID's will need to be carefully tuned for the robots weight on the ground. Minor adjustments may be needed after adding large mechanisms.

Notes on Joystick Mapping

- Joysticks traditionally read in a value from -1 to 1 in X and -1 to 1 in Y. This means that every value of a joystick can be represented by a point on a square that is 2 x 2 units across and centered at the origin.
- Directly mapping points on the square to swerve speeds will cause issues since certain points on the square are more than 1 unit away from the origin. To correct for this considering using a square to circle mapping function.

$$(x', y') = \left(x\sqrt{1 - \frac{y^2}{2}}, y\sqrt{1 - \frac{x^2}{2}} \right)$$

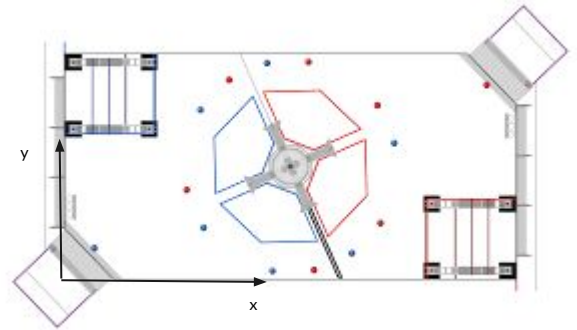


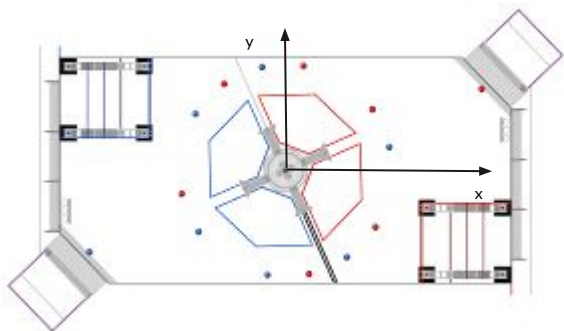
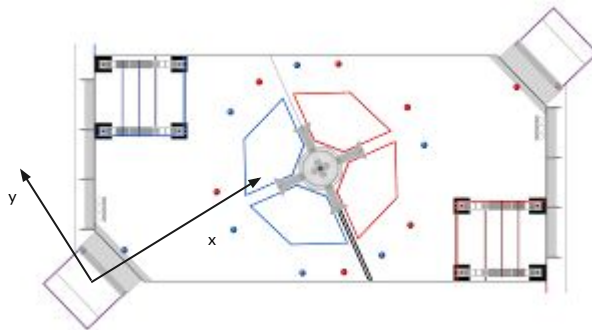
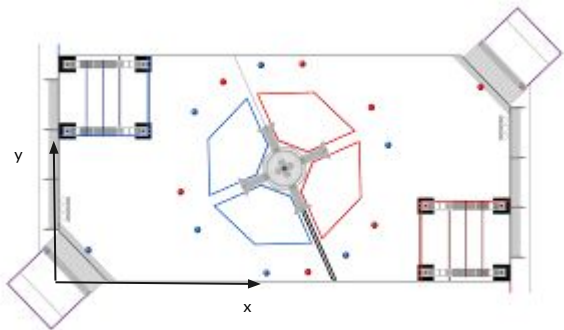


**Referencing
the robot to
reality**

Coordinate Systems

- Coordinate systems are how you define where a robot is on a field.
- There is no single correct answer for how you define your coordinate system. However, your choice of coordinate system will make certain problems easier or harder.
- Be mindful of left vs right hand coordinate system. We chose to use a right handed coordinate system for all of systems, as this is how vectors are typically taught in most physics classes. However many other applications such as computer graphics and IMU units often use a left hand system





Tips:

- Pick something intuitive
- Pick coordinate frames that are referenced to important things (the goal)
- This year, the FRC field is rotationally symmetric. This allows you to pick a single coordinate system that works for both red and blue. Sometimes this is not the case (Steamworks)
- BE CONSISTENT. More than anything else be consistent. (This can be deceptively difficult)



Kinematics Model

- Since all drive commands are given to the robot in terms of $x(m/s)$ $y(m/s)$ and $\theta(rad/s)$. The robots expected location and orientation at any given point in time is simply the integral of the X, Y and theta velocities up to that time.
- For example, if the rover is given the instruction to move 1m/s in the X direction for 1 second, it should have a displacement of 1 meter in the x direction.
- We have found that this is actually quite accurate at computing robot position. Consequently, we use it as a kinematics model that is integrated with our odometry for computing position.
 - Note this only holds so long as the robot doesn't hit anything!



Fusing Odometry with Kinematics

- Started with SwerveDriveOdometry class from WPILIB
 - We noticed that the accuracy was not as good as we wanted, likely due to relying purely on encoder measurements
- To fix this, we averaged the results of the SwerveDriveOdometry class with our kinematics model to better predict the location of the robot on the field.
- Kinematics model worked great in autonomous because there was no interference from other robots, and greatly improved our accuracy.

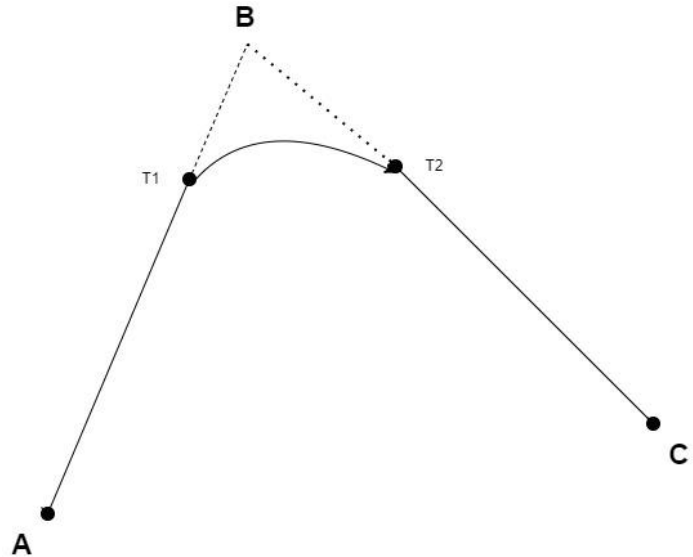


Pathing



Constant Acceleration Interpolation - Overview

- Time-based algorithm
- Easy to define velocities and timing of movements
- Smooth curves allow for minimal slipping
- Easy to visualize and optimizable





Segment One - A to T1

- In this segment, we are focusing on driving straight from point A to T1, from time $t = 0$, to $t = t1$
- To do so, we will calculate the velocities that the robot must achieve to make it from A to T1 in time
 - $\text{VelocityX} = (T1x - Ax)/(t1 - 0)$
 - $\text{VelocityY} = (T1y - Ay)/(t1 - 0)$
 - $\text{ThetaVelocity} = (T1\text{Theta} - A\text{Theta})/(t1 - 0)$
- From here, we can return our velocities to the AutoDrive function!



Segment Two - T₁ to T₂

- Once the robot reach T₁, we need to begin to change our velocity from the velocity at T₁, to the velocity at T₂
- To do so, we need to change our velocity by constantly accelerating between the points
- $Acceleration = (T_2Vel - T_1Vel) / (T_2 - T_1)$
- $AdjustedVel = (Acceleration * loopTime) + T_1Vel$

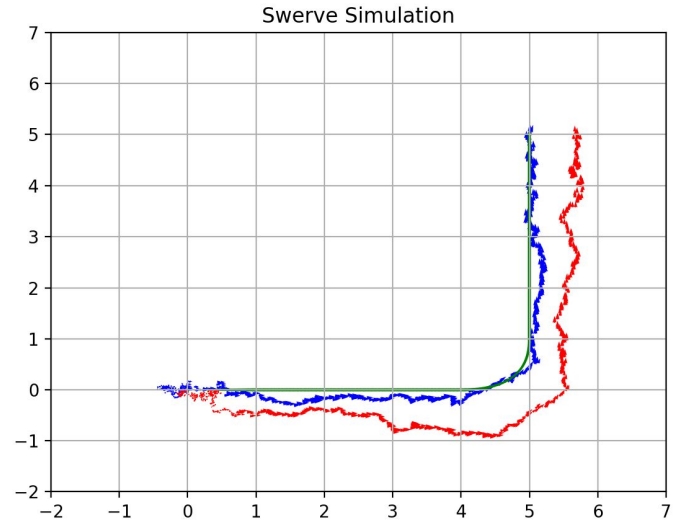


Segment Three - T2 to C

- This segment is very similar to segment one - just generate the velocities in between T2 and C
 - $\text{VelocityX} = (C_x - T2_x)/(t_C - t_2)$
 - $\text{VelocityY} = (C_y - T2_y)/(t_C - t_2)$
 - $\text{ThetaVelocity} = (C_{\text{theta}} - T2_{\text{theta}})/(t_C - t_2)$
- Then set the velocities through the auto drive function!

Simulation

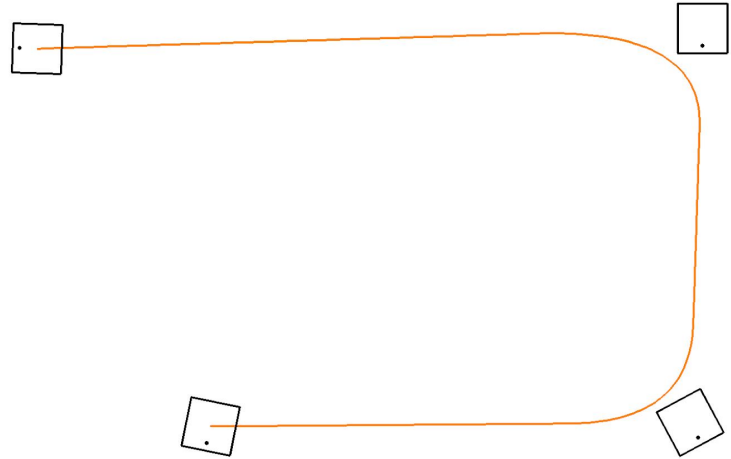
- Writing simulations of the robot has numerous advantages, and allows you to validate your algorithms and strategies without the need for a physical robot.
- Simulations allow you to control noise and error in the system.
- Enables rapid experimentation with new algorithms.



Green - Ideal Path
Red - Pure Const Accel Interp
Blue - Const Accel Interp with dynamic repathing

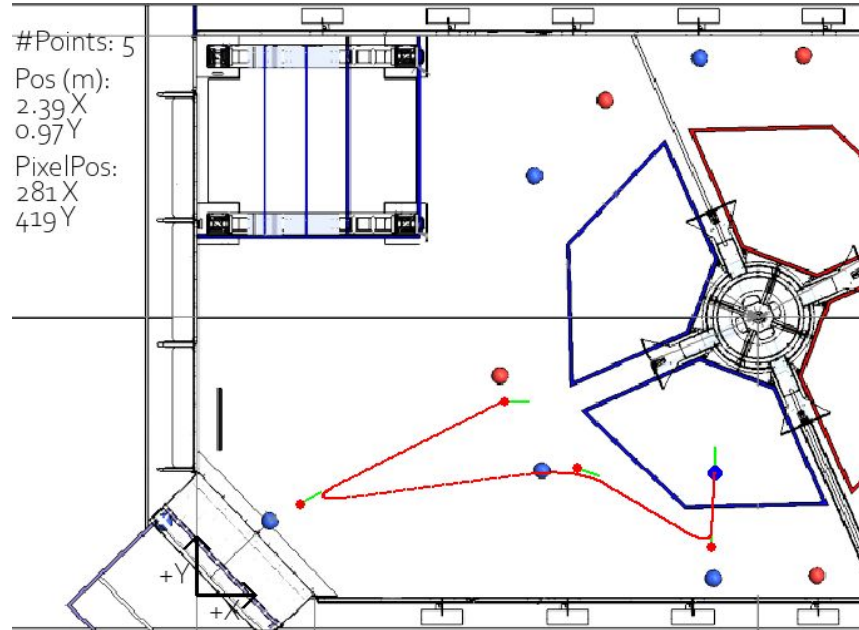
Simulation

- The pathing algorithms you will use are fairly complicated. Even if you understand them, you will not program them perfectly on the first try.
- Use simulation to fix logical errors in your code. It is much faster to debug and run than setting up a robot.
- Additionally, simulation can be done remotely.



Path Generation

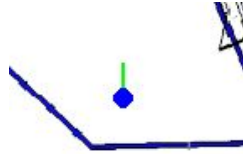
- Path format
- GUI
- Path visualization
- Path debugging tools
- Uploading / Downloading from robot





Path Format

- Paths are saved as JSON files
- Each path file is a list of points
- Each point contains:
 - Index (starting at 0)
 - X and Y coordinate (meters)
 - Angle (radians)
 - Time (seconds)
 - Interpolation range (0 - 0.5)
 - Timestamp (seconds since epoch)
 - And other fields for display purposes




`{}` ExamplePoint.json

```
1  [
2    {
3      "x": 7.48811212458287,
4      "y": 1.8125259176863182,
5      "pixelX": 559.0,
6      "pixelY": 373.0,
7      "angle": 1.5707963267948966,
8      "speed": 0.0,
9      "time": 0.0,
10     "deltaTime": 0.0,
11     "interpolationRange": 0.0,
12     "color": [
13       255,
14       0,
15       0
16     ],
17     "index": 0,
18     "timeStamp": "1659019956.982787"
19   }
20 ]
```

Graphical User Interface

- Point selection
- Tools for editing points
 - Angle selector
 - X-Y input and adjuster
 - Time from previous point selector
 - Delete point button
 - Interpolation selector

Angle: 90.0 ExamplePath



Path Saved

Speed: 0.0

Time from prev: 0.0

X: 7.48

Y: 1.81

Index: 0

Path Time: 0.0

Time to point: 0.0

Interp: 0.0

ExamplePoint

Color...

UPLOAD

Accel

UPLOAD ALL

Whl Path

Veloc

DEL ALL

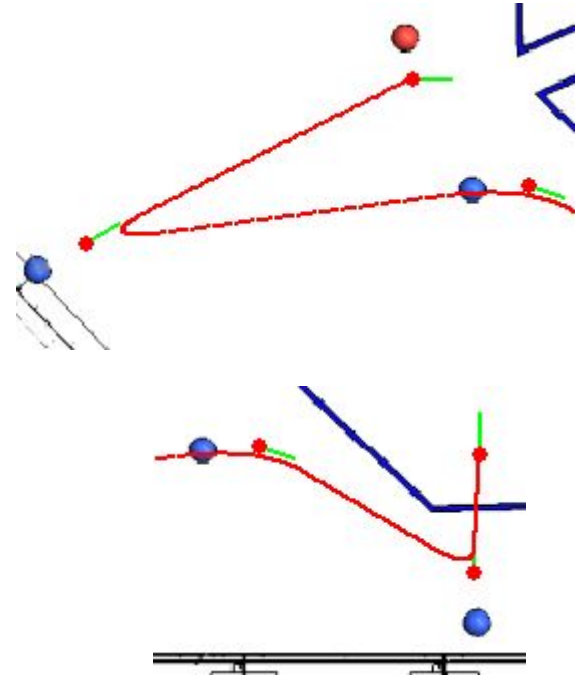
DOWNLND ALL

SAV DEL

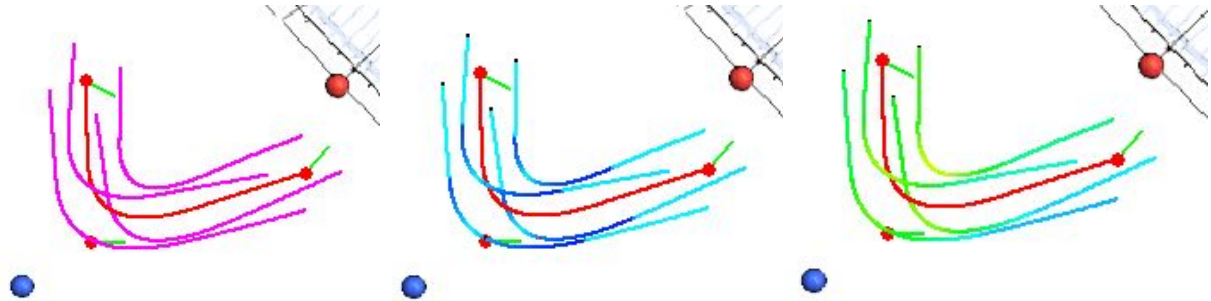


Path Visualization

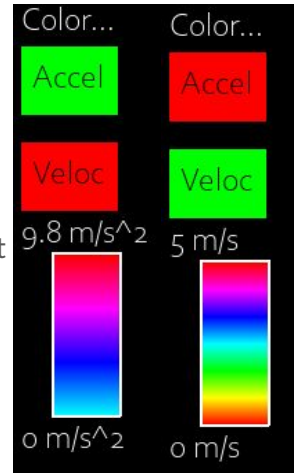
- Field in background
- Path display
 - Uses same constant acceleration interpolation algorithm
 - Samples path at a given time interval
 - Angle at path point shown by green line



Path Debugging Tools



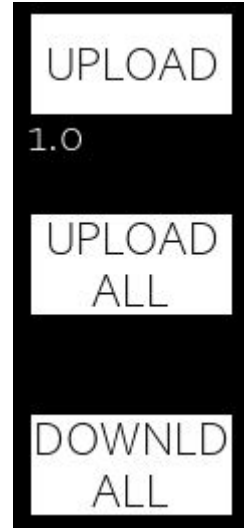
- Wheel paths
 - Path of each wheel can be plotted along with the robot center
 - Useful for knowing the physical bounds of the path (space the robot will pass through)
- Wheel velocity and acceleration
 - The path of each wheel can be colored to show its instantaneous velocity or acceleration
 - Useful for knowing where the robot is trying to push its driving capabilities
 - Example: High acceleration at a corner shows that the robot will try to take a corner too fast
 - Example: Low velocity shows where time can be saved in an autonomous path





Uploading and Downloading

- Paths can be uploaded and downloaded to the robot from the path tool via SSH
- Paths are stored in the 'deploy' folder
- No code re-deploy needed; only restart code





Planned improvements

- Improve logging and debugging features to enable debugging paths faster
- Add MQTT data stream to relay real time robot position to the path planning tool for analysis
- Integrate computer vision position system into odometry as an absolute correction system
 - Develop Kalman filter to fuse Kinematics, Vision, wheel encoders and IMU to produce better location
- Upgrade the path planning tool UI to improve user interaction
- Develop an ground truth robot tracking solution to debug everything



Questions?



Resources

- <https://github.com/HighlandersFRC/2022-Maverick/tree/dev-IRI>
- https://docs.google.com/document/d/19eBLhe2_HO-rruhovWB305U-AfO3vUngpONt3NE_dhY/edit?usp=sharing
- <https://www.xarg.org/2017/07/how-to-map-a-square-to-a-circle/>
- <https://docs.wpilib.org/en/stable/docs/software/kinematics-and-odometry/swerve-drive-odometry.html>