PathWeaver and Robot Path Planning

Last Updated: 01-25-2019

Table of Contents

Introduction to Path Planning

Introduction to Path Planning

Autonomous is an important section of the match; it is exciting when robots do impressive things in autonomous. In order to score, the robot usually need to go somewhere. The faster the robot arrives at that location, the sooner it can score points! The traditional method for autonomous is driving in a straight line, turning to a certain angle, and driving in a straight line again. This approach works fine, but the robot spends a non-negligible amount of time stopping and starting again after each straight line and turn.

A more advanced approach to autonomous is called “path planning”. Instead of driving in a straight line and turning once the line is complete, the robot continuously moves, driving with a curve-like motion. This can reduce turning stoppage time.

Paths have position and velocity values. A path has a list of position and velocity values for the robot every x seconds (usually 0.02 seconds or less). The robot then attempts to follow these points, adjusting using PID control when needed.

Many teams and organizations have created path planning libraries for FRC, but the officially supported library for WPILib is Pathfinder, which can be added as a vendor library by adding this vendor JSON using the **Manage Libraries->Install New Libraries (online)** function in VSCode (see the 3rd Party Libraries article for more details): http://dev.imjac.in/maven/jaci/pathfinder/PathfinderOLD-latest.json

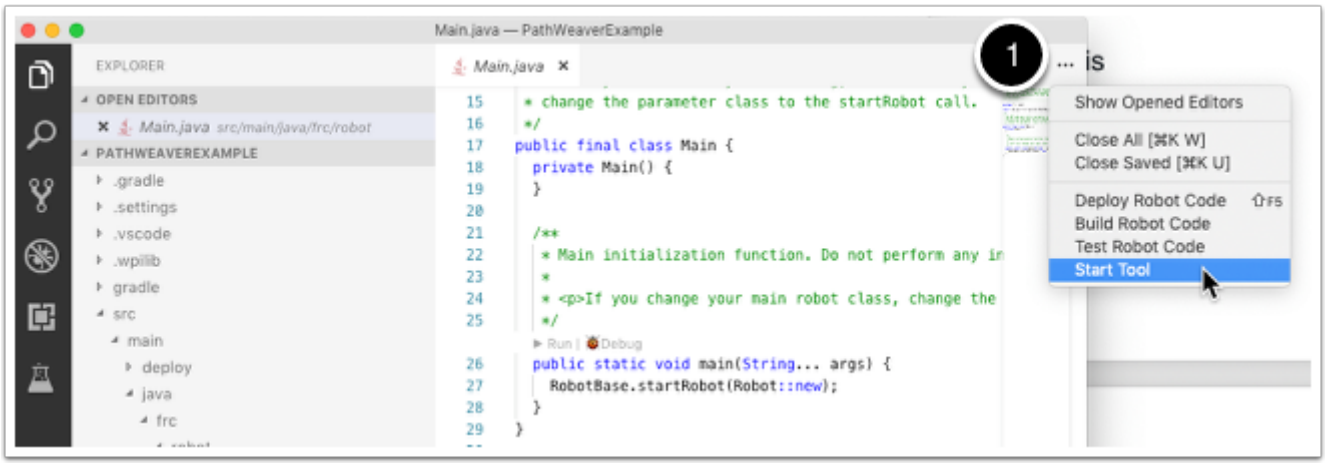
Creating paths for your robot to follow

Creating a PathWeaver project

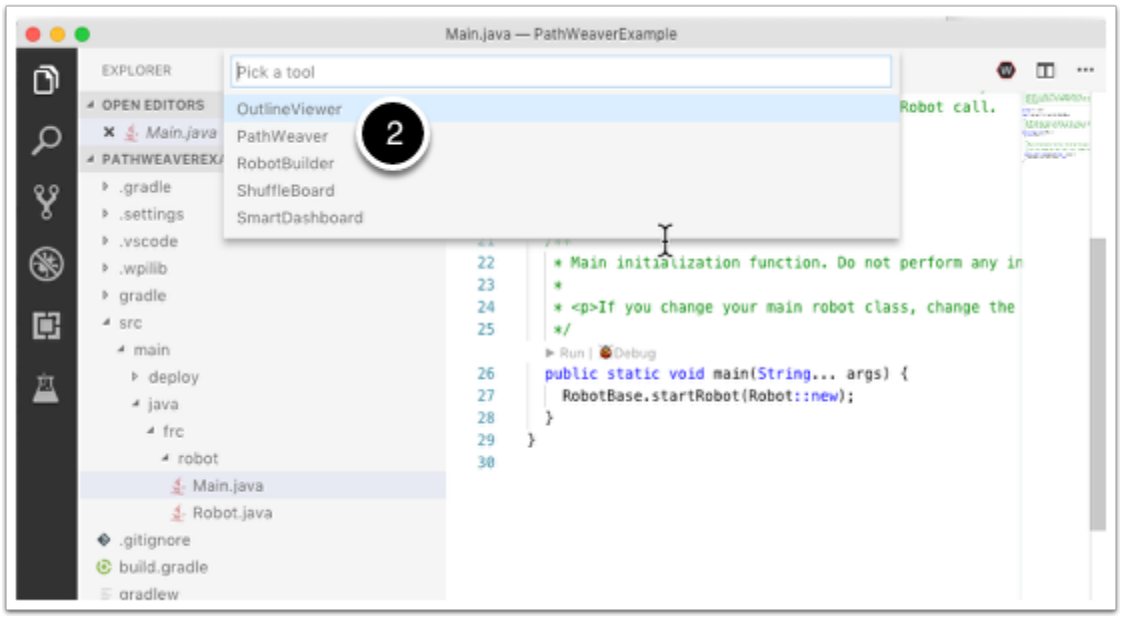
PathWeaver is the tool used to draw the paths for a robot to follow. The paths for a single program are stored in a PathWeaver project.

Starting PathWeaver

PathWeaver is started by clicking on the ellipsis icon in the top right of the corner of the VSCode interface. You must select a source file from the WPILib project to see the icon. Then click on "Start tool" and then click on"PathWeaver" as shown below.



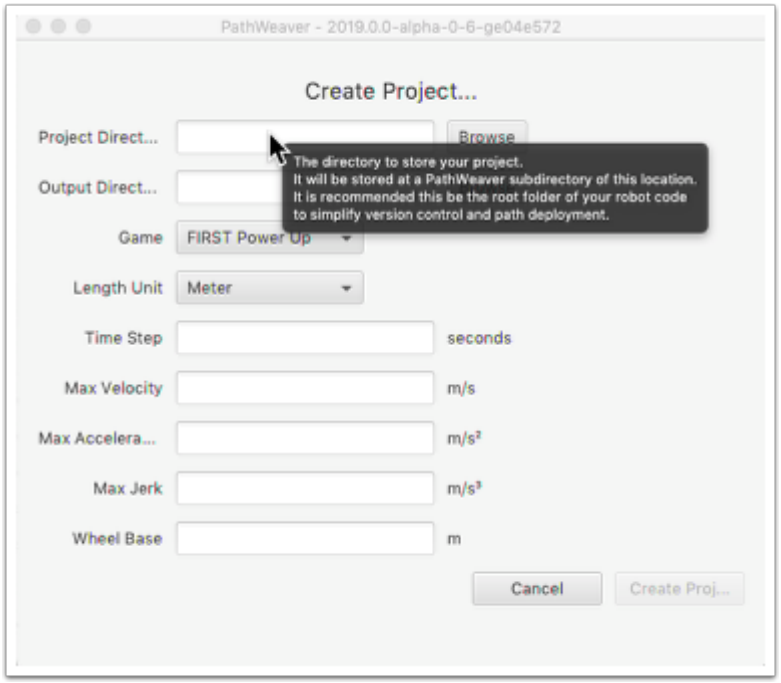
PathWeaver and Robot Path Planning



Creating a PathWeaver project

To create a PathWeave project, click on "Create project" then fill out the project creation form. Notice that hovering ove any of the fields in the form will display more information about what is required.

PathWeaver and Robot Path Planning



Project directory: this is normally top level project directory that contains the build.gradle and src files for your robot program. Choosing this directory is the expected way to use PathWeaver and will cause it to locate all the output files in the correct directories for automatic path deployment to your robot.

Output directory: the directory where the paths are stored for deployment to your robot. If you specified the top level project folder of our robot project in the previous step (as recommended) filling in the project directory is optional.

Game: the game (which FRC game is being used) will cause the correct field image overlay to be used. You can also create your own field images and the procedure will be described later in this document.

Length Unit: the units to be used in describing your robot and for the field measurements when using PathWeaver. It's best to use units that match the rest of the field documentation to avoid errors in creating paths.

Time Step: is the time that each generated point should be driven. Shorter time steps cause more points to be generated and may significantly improve the accuracy of the followed path. If you are using the default TimedRobot class, we recommend using a time step of 20ms or 0.02 seconds.

Max Velocity: the maximum speed attainable by your robot. This is the most important parameter of the robot profile since it causes generated paths to never exceed this speed. You can either

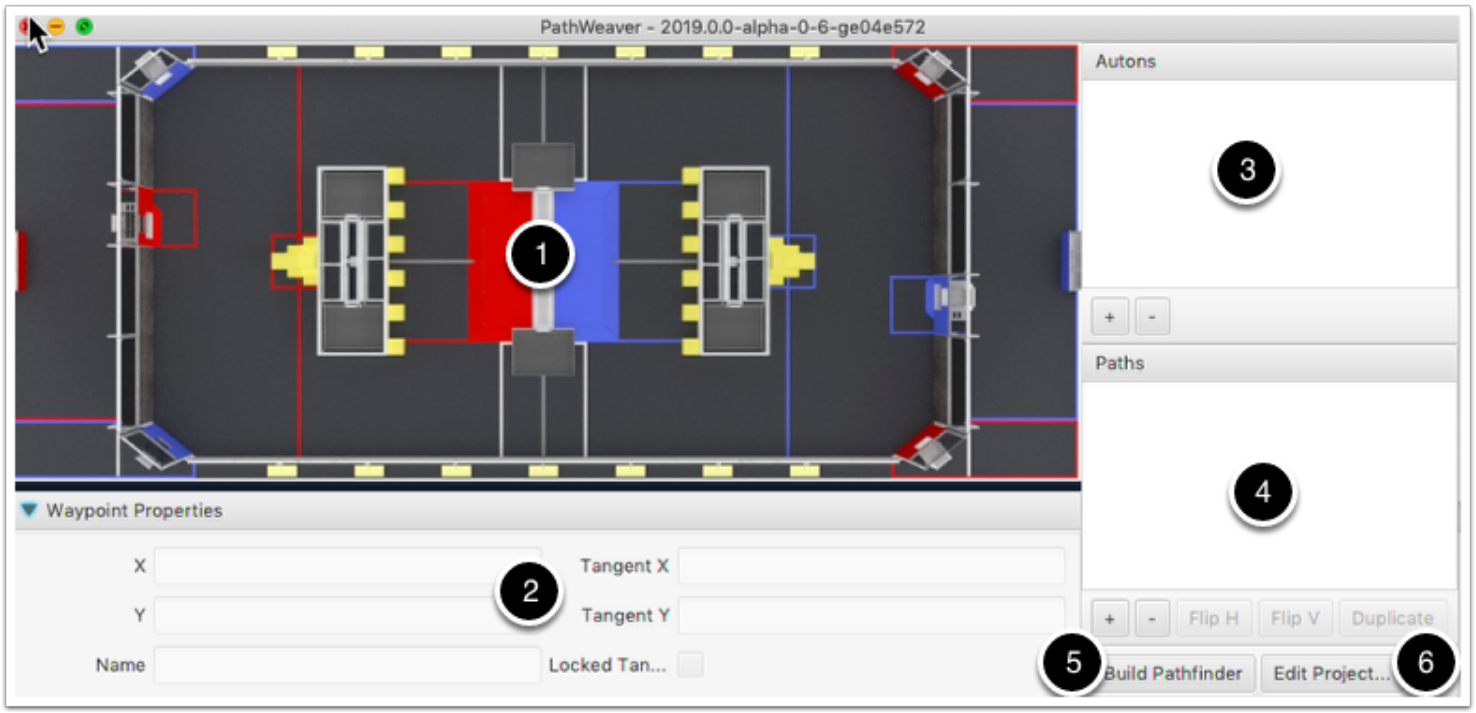
calculate this velocity using knowledge of your robot design or by measuring it. One method of measuring the Max Velocity is driving in a straight line then measuring the speed after the robot has reached full speed. Another method is spinning in place at full speed for 10 rotations and using the formula for circumference divided by 10 for total distance traveled and the time to complete for the distance traveled.

Max Acceleration: this is the maximum acceleration your robot is capable of achieving. A good starting point is 6.56 ft/sec^2 or 2m/sec^2 for a kit of parts drivetrain.

Max Jerk: this is the rate of change of the robots acceleration expressed as dist/sec^3. This is mostly a user preference and a recommended starting point is 197 ft/sec^3 or 60m/sec^3.

Wheel Base: the distance between the left and right wheels of your robot.

PathWeaver user interface



The PathWeaver user interface consists of:

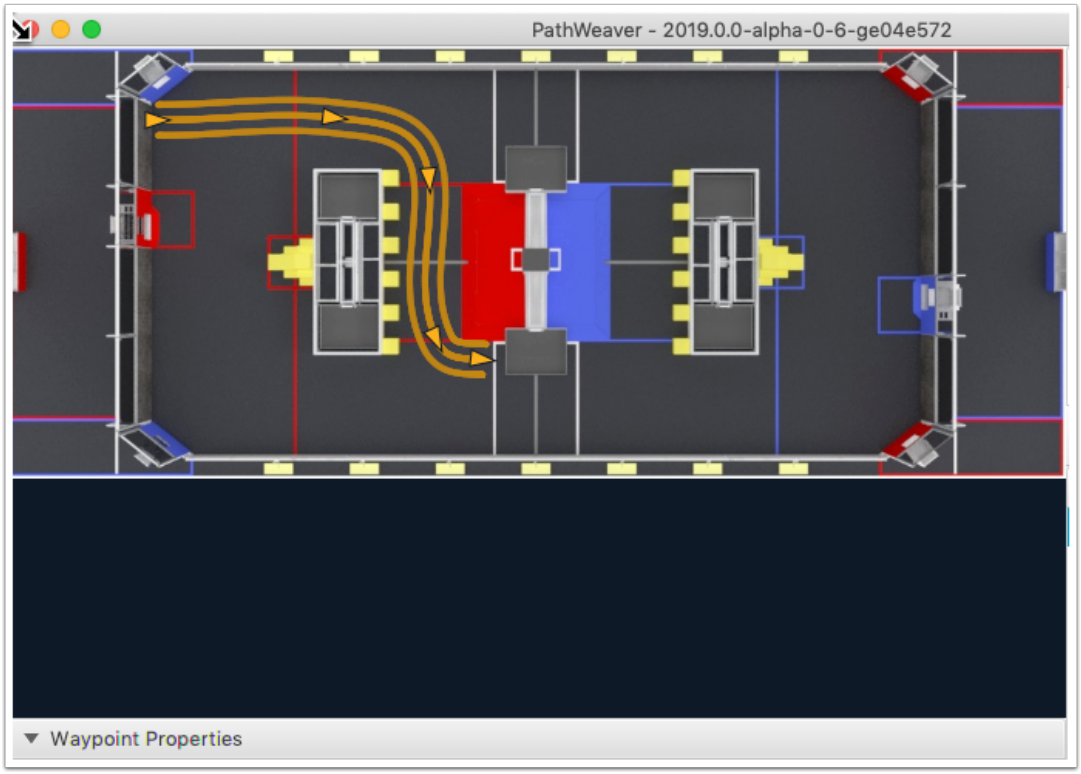
1. the field area where the robot will travel. It is on this image the robot paths will be drawn to visualize the motion.
2. the properties of the currently selected waypoint

3. the groups of paths that will be used together. It is a way of seeing all the paths the robot might travel in a single run.
4. the individual paths that a robot will follow, The paths may be grouped together as a Path Group in order to better visualize the total robot movement in a multi-path sequence.
5. the drawn paths are used to create the set of wheel velocities the robot will use when following your path. There is one velocity generated for every unit of time as described in the Time Base parameter of your project.
6. allows the PathWeaver project properties to be edited (includes Project directory, time step, field drawing, etc.) as described at the beginning of this article.

Drawing the path the robot will follow

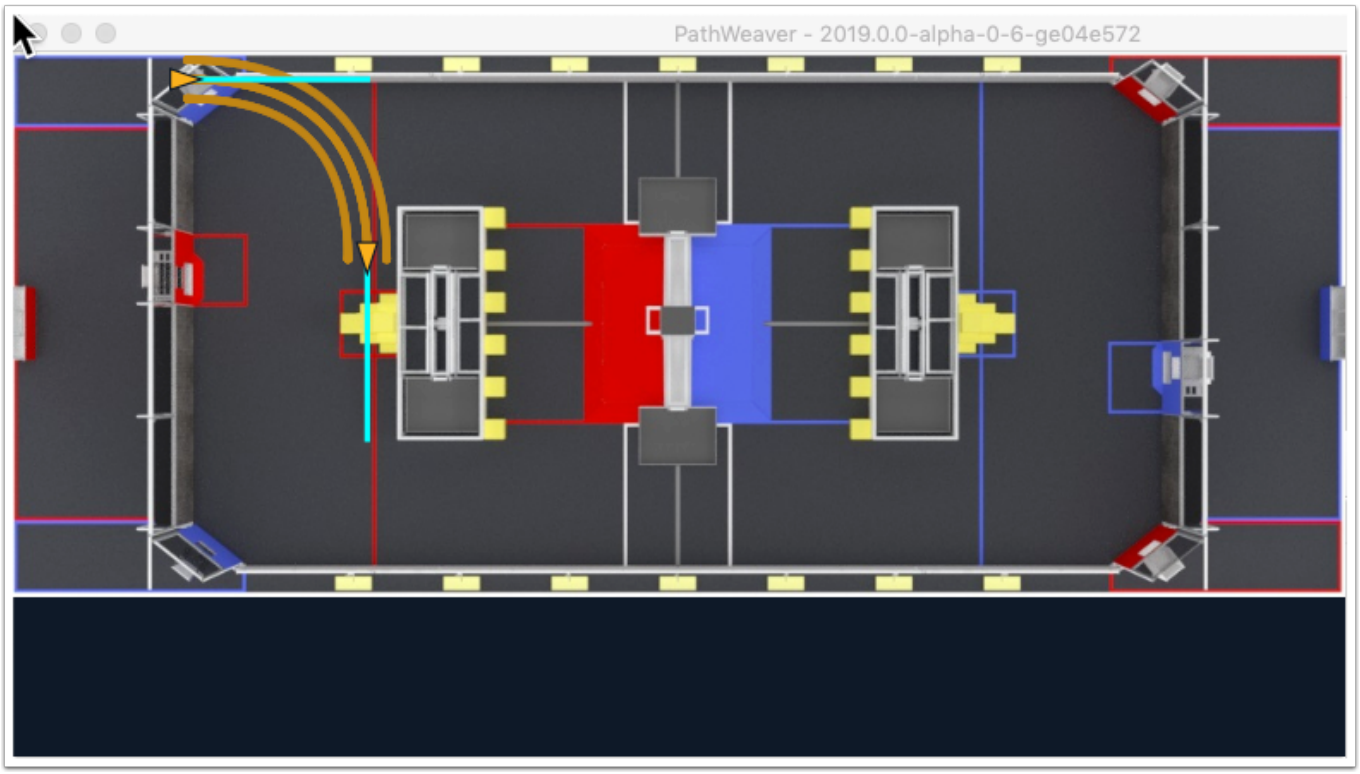
The main feature of PathWeaver is that it allows the path the robot will follow to be easily visualized. In the following picture you can see a path that goes from a robot starting point to a finish point that follows a smooth path with minimum accelerating and decelerating. Paths can have any number of waypoints that can allow more complex driving to be described. In this case there are 5 waypoints (including the start and stop) depicted with the triangle icons. Each waypoint consists of a X, Y position on the field as well as a robot heading described as the X and Y tangent lines.

Notice also that the path shows the left and right wheel tracks as well as the robot centerline to make it easier to visualize the robot on the path and avoid obstacles. To get better performance, PathWeaver draws the path when you are interactively manipulating it using an estimation algorithm. The three depicted paths (left, right, and center) are computed using a PathFinder generated path when you stop editing the waypoints. This is the same data that the robot will use to actually follow the path while driving. The following sections will show how to create this path.

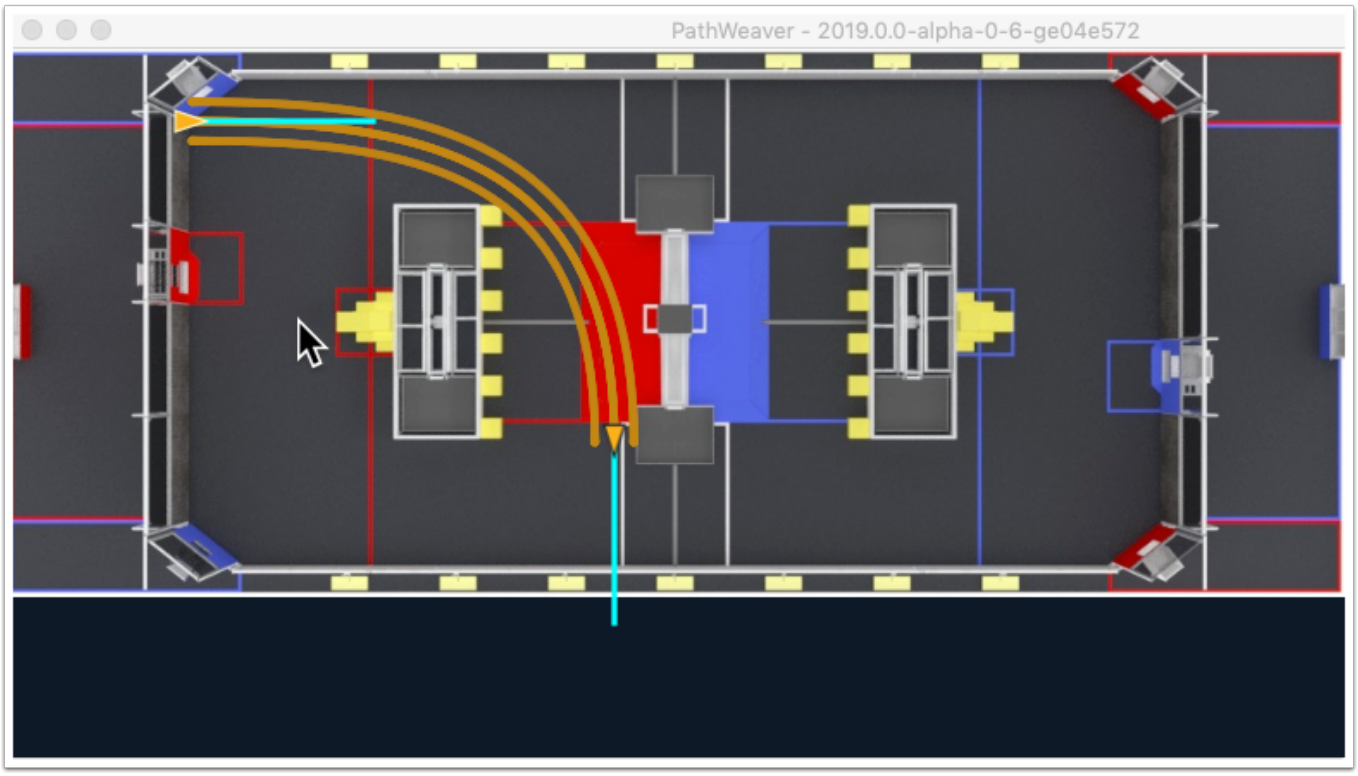


Creating the initial path

To start creating a path, click the + (plus) button in the path window. A default path will be created that probably does not have the proper start and end points. The path also shows the tangent vectors (teal lines) for the start and end points. By changing the angle of the tangent vectors, the path followed is affected.

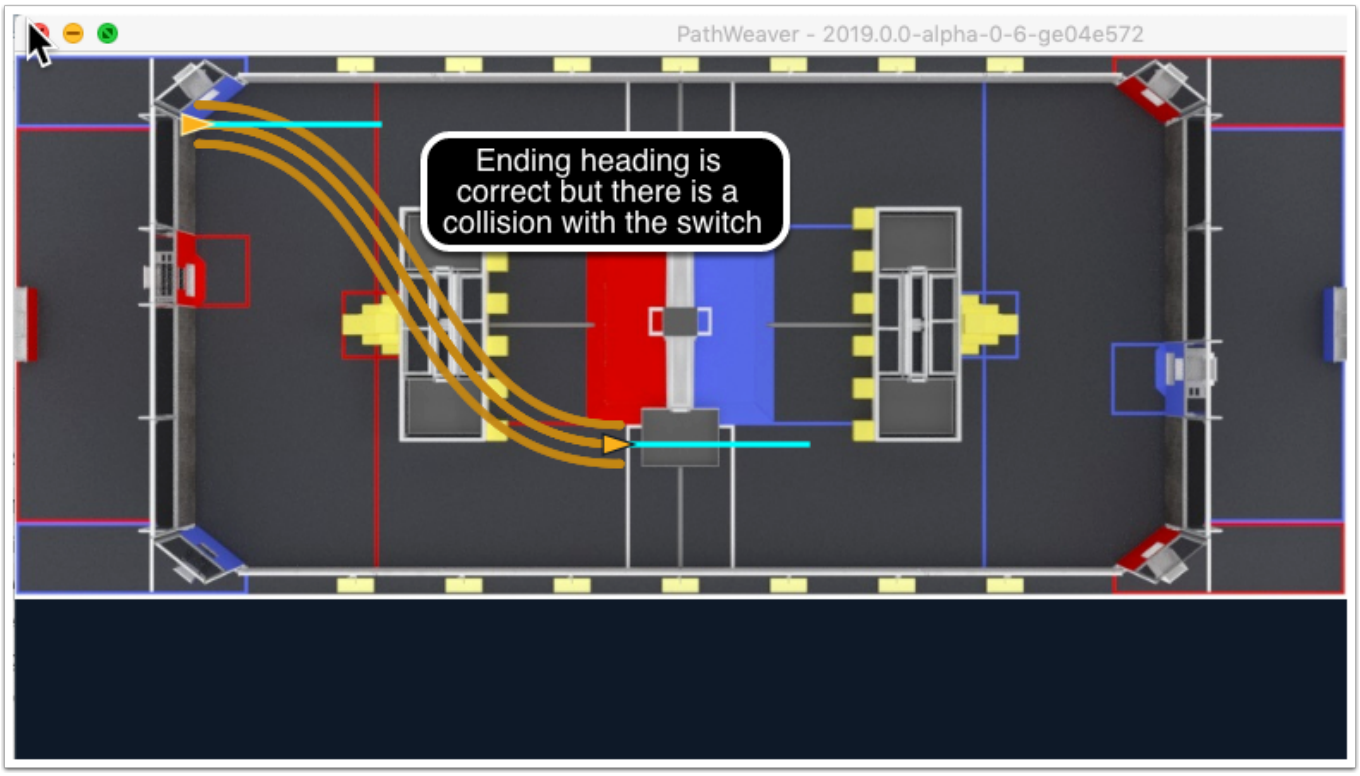


Drag the start and end points of the path to the desired locations. Notice that in this case, the path drives through the edge of a field element (Switch from the FRC 2018 game) and ends with the robot facing towards the bottom of the field drawing. The path should instead drive around the Switch and end up pointing towards the Scale.



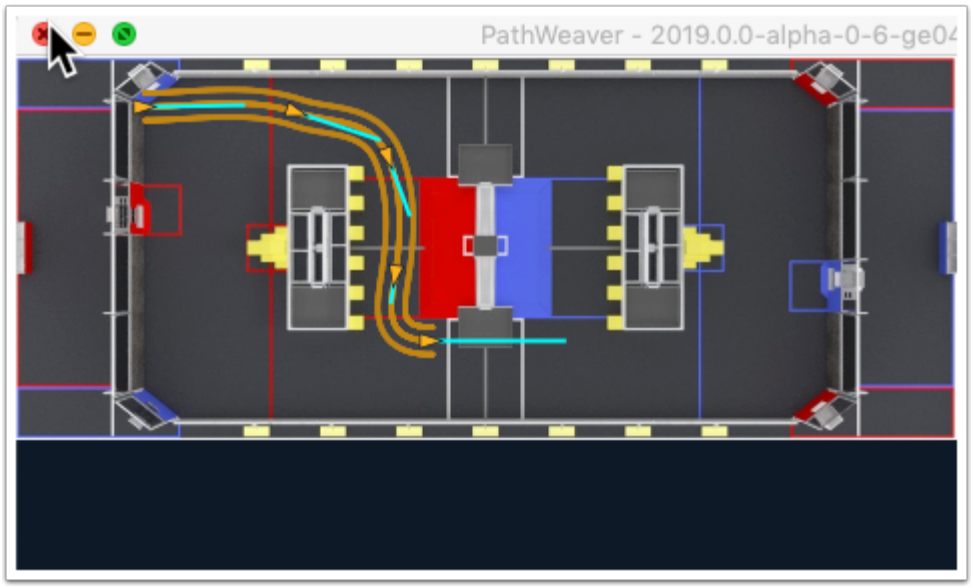
Changing a waypoint heading

The robot heading can be changed by dragging the tangent vector (teal) line to that it faces the Scale side. That causes the robot to end the path facing in the right direction, but it now is driving through the center of the Switch.



Adding additional waypoints to control the robot path

Adding additional waypoints and changing their tangent vectors can affect the path that is followed. Additional waypoints can be added by dragging in the middle of the path. In this case, adding two additional waypoints for the first turn and another waypoint for the final turn will help the robot make the turns more smoothly without collisions.



Locking the tangent lines

Locking tangent lines prevents them from changing when the path is being manipulated. The tangent lines will also be locked when the point is moved.

More precise control of waypoints

While PathWeaver makes it simple to draw paths that the robot should follow, it is sometimes hard to precisely set where the waypoints should be placed. In this case, setting the waypoint locations can be done by entering the X and Y value which might come from an accurate CAD model of the field. The points can be entered in the X and Y fields when a waypoint is selected.

Adding additional related paths

There are times when multiple related paths might be necessary. For example, a red vs. blue side where something isn't symmetric or transpose of the X or Y axis would be needed. In this case a path can be duplicated, then the Horizontal or Vertical axis can be flipped between the old path and the duplicate. To do this create a duplicate path using the "Duplicate" button, then use the "Flip H" or "Flip V" buttons for the new path to modify it.

Creating Path Groups

Path Groups are a way of visualizing where one path ends and the next one starts. An example is when the robot program drives one path, does something after the path has completed, drives to another location to obtain a game piece, then back again to score it. It's important that the start and end points of each path in the group have common end and start points. By adding all the paths to a single path group and selecting the group, all paths in that group will be shown. Then each path can be edited while viewing all the paths.

Creating a Path Group

Press the "plus" button underneath Path Groups. Then drag the Paths from the Paths section into your Path Group.

Each path added to a path group will be drawn in a different color making it easy to figure out what the name is for each path.

If there are multiple paths in a group, selection works as follows:

1. Selecting the group displays all paths in the group making it easy to see the relationship between them. Any waypoint on any of the paths can be edited while the group is selected and it will only change the path containing the waypoint.
2. Selecting on a single path in the group will only display that path, making it easy to precisely see what all the waypoints are doing and preventing clutter in the interface if multiple paths cross over or are close to each other.

What paths can and cannot do

What kind of paths can PathFinder follow?

Paths and path groups work best when the robot generally drives in the same direction and multiple paths continue straight line driving.

Can Pathfinder generate paths where the robot drives in reverse?

PathFinder paths always go in the forward direction, however, for any path the robot can drive backward by negating the velocities, then exchanging left for right. You should also exchange the left and right encoder values.

Can I have a 90-degree turn in my path?

A 90-degree turn requires a point-turn and is discontinuous in curvature and therefore not possible to create in PathFinder.

Following generated paths

Integrating path following into a robot program

Known Issue

PathWeaver currently has a known issue. The left and right paths are being swapped. This will be fixed in PathWeaver v2019.3.1. In the meantime, this can be corrected in the follower (what this article describes). Once this update is published, the following fix will not be needed. The fix is as follows:

Replace:

```
Trajectory left_trajectory = PathfinderFRC.getTrajectory(k_path_name + ".left");
```

```
Trajectory right_trajectory = PathfinderFRC.getTrajectory(k_path_name + ".right"
```

With:

```
Trajectory left_trajectory = PathfinderFRC.getTrajectory(k_path_name + ".right")
```

```
Trajectory right_trajectory = PathfinderFRC.getTrajectory(k_path_name + ".left")
```

Depending on the orientation of your gyro, you may also need to invert the desired heading with the following fix:

```
double desired_heading = -Pathfinder.r2d(m_left_follower.getHeading());
```

Overview

This article describes the process of writing a program that can use a stored path and have the robot follow it. The robot for this example is based on the TimedRobot template class and has left and right driven 4" wheels connected to motors with encoders on each side. In addition the robot has an Analog Gyro that is used to help make sure the robot hardware is keeping up with each generated point. The following code is heavily taken from Jaci's Pathfinder's wiki. This is because the paths PathWeaver generates are Pathfinder paths and Pathfinder must be added as a vendor dependency.

Java import directives

Each class in WPILib requires that an import declaration to help the compiler resolve the references to WPILib libraries and Pathfinder code included as a vendor library.

```
package frc.robot;

import edu.wpi.first.wpilibj.AnalogGyro;
import edu.wpi.first.wpilibj.Encoder;
import edu.wpi.first.wpilibj.Notifier;
import edu.wpi.first.wpilibj.Spark;
import edu.wpi.first.wpilibj.SpeedController;
import edu.wpi.first.wpilibj.TimedRobot;
import jaci.pathfinder.Pathfinder;
import jaci.pathfinder.PathfinderFRC;
import jaci.pathfinder.Trajectory;
import jaci.pathfinder.followers.EncoderFollower;
```

Start of program and constant declarations

The program is based on the TimedRobot class - a class where the appropriate initialization and periodic methods for each state (disabled, autonomous, test, and teleop) that the program could be in. For example, the teleopPeriodic() method is called periodically, every 20mS by default, and the same is true for autonomous, test, and disabled.

Symbolic constants are used throughout the program to name each one to match its purpose in the program. By grouping the constants together at the start of the module it makes it easier to see the assumptions the program is making where each is reflected as some constant value.

```
public class Robot extends TimedRobot {
  private static final int k_ticks_per_rev = 1024;
  private static final double k_wheel_diameter = 4.0 / 12.0;
  private static final double k_max_velocity = 10;

  private static final int k_left_channel = 0;
  private static final int k_right_channel = 1;
```

```
private static final int k_left_encoder_port_a = 0;
private static final int k_left_encoder_port_b = 1;
private static final int k_right_encoder_port_a = 2;
private static final int k_right_encoder_port_b = 3;

private static final int k_gyro_port = 0;

private static final String k_path_name = "example";
```

Member variables used for the Robot class

The Robot class (inherited from TimedRobot) contains the periodic methods. It also has a number of variables required for the Robot class.

```
private SpeedController m_left_motor;
private SpeedController m_right_motor;

private Encoder m_left_encoder;
private Encoder m_right_encoder;

private AnalogGyro m_gyro;

private EncoderFollower m_left_follower;
private EncoderFollower m_right_follower;

private Notifier m_follower_notifier;
```

k_ticks_per_rev - number of encoder counts per wheel revolution

k_wheel_diameter - diameter of the wheels

k_max_velocity - maximum velocity of the robot

k_left_channel, k_right_channel - the port numbers for the left and right speed controllers

k_left_encoder_port_a, k_left_encoder_port_b, k_right_encoder_port_a, k_right_encoder_port_b - the port numbers for the encoders connected to the left and right side of the drivetrain

k_gyro_port - the analog input for the gyro (other gyros might be connected differently)

k_path_name - name of this path

Initialize the robot sensors and actuators

```
@Override
public void robotInit() {
  m_left_motor = new Spark(k_left_channel);
  m_right_motor = new Spark(k_right_channel);
  m_left_encoder = new Encoder(k_left_encoder_port_a, k_left_encoder_port_b);
  m_right_encoder = new Encoder(k_right_encoder_port_a, k_right_encoder_port_b);
  m_gyro = new AnalogGyro(k_gyro_port);
}
```

Initialize the EncoderFollower objects

At the start of the autonomous period we do the following operations:

1. Create the trajectories for the left and right sides of the drivetrain. This will look for paths in the /home/lvuser/deploy/paths folder on the roboRIO. If you choose the output directory in PathWeaver (as shown in the previous instructions), PathWeaver will automatically place the paths in the proper folder. The full filename for the path is: /home/lvuser/deploy/paths/PathName.left.pf1.csv and /home/lvuser/deploy/paths/PathName.right.pf1.csv for the left and right paths.
2. Create encoder followers from the left and right trajectories. The encoder followers compute the motor values based on where the robot is in the path.
3. Configure the encoders used by the followers with the number of counts per wheel revolution and diameter and PID constants to tune how fast the follower reacts to changes in velocity.
4. Create the notifier that will regularly call the followPath() method that computes the motor speeds and send them to the motors.

```
@Override
public void autonomousInit() {
  Trajectory left_trajectory = PathfinderFRC.getTrajectory(k_path_name + ".left");
  Trajectory right_trajectory = PathfinderFRC.getTrajectory(k_path_name + ".right");

  m_left_follower = new EncoderFollower(left_trajectory);
  m_right_follower = new EncoderFollower(right_trajectory);
```

```
    m_left_follower.configureEncoder(m_left_encoder.get(), k_ticks_per_rev,
k_wheel_diameter);
    // You must tune the PID values on the following line!
    m_left_follower.configurePIDVA(1.0, 0.0, 0.0, 1 / k_max_velocity, 0);

    m_right_follower.configureEncoder(m_right_encoder.get(), k_ticks_per_rev,
k_wheel_diameter);
    // You must tune the PID values on the following line!
    m_right_follower.configurePIDVA(1.0, 0.0, 0.0, 1 / k_max_velocity, 0);

    m_follower_notifier = new Notifier(this::followPath);
    m_follower_notifier.startPeriodic(left_trajectory.get(0).dt);
  }
```

Notifier method that actually drives the motors

Each delta time (value programmed into the notifier in the previous code segment) get the current wheel speeds for the left and the right side. Use the predicted heading at each point and the actual robot heading from the gyro sensor. The difference between the actual and predicted heading is the heading error that is factored into the motor speed setting to help ensure the robot tracks the path direction.

```
  private void followPath() {
    if (m_left_follower.isFinished() || m_right_follower.isFinished()) {
      m_follower_notifier.stop();
    } else {
      double left_speed = m_left_follower.calculate(m_left_encoder.get());
      double right_speed = m_right_follower.calculate(m_right_encoder.get());
      double heading = m_gyro.getAngle();
      double desired_heading = Pathfinder.r2d(m_left_follower.getHeading());
      double heading_difference = Pathfinder.boundHalfDegrees(desired_heading - heading);
      double turn =  0.8 * (-1.0/80.0) * heading_difference;
      m_left_motor.set(left_speed + turn);
      m_right_motor.set(right_speed - turn);
    }
  }

  /**
```

```
   * This function is called periodically during autonomous.
   */
  @Override
  public void autonomousPeriodic() {
  }
```

Stop the motors at the start of the Teleop period

After the autonomous period ends and the teleop period begins, be sure to stop the notifier from running the followPath() method (above) and stop the motors in case they were still running.

```
  @Override
  public void teleopInit() {
    m_follower_notifier.stop();
    m_left_motor.set(0);
    m_right_motor.set(0);
  }
}
```

Advanced topics

Adding field images to PathWeaver

The initial release of PathWeaver contained the field image for the FRC 2018 Game. The next update will include *FIRST* Destination Deep Space. Here are instructions for adding the 2019 game now or adding your own field image for some other application.

Games are loaded from the~/PathWeaver/Games on Linux and Mac or %USERPROFILE%/PathWeaver/Games directory on Windows. The files can be in either a game-specific subdirectory, or in a zip file in the Games directory. The ZIP file must follow the same layout as a game directory; the JSON file must be in the root of the ZIP file (cannot be in a subdirectory).

FIRST Destination Deep Space field definition.

File Layout

```
~/PathWeaver
  /Games
    /Custom Game
      custom-game.json
      field-image.png
    OtherGame.zip
```

JSON Format

```
{
  "game": "game name",
  "field-image": "relative/path/to/img.png",
  "field-corners": {
    "top-left": [x, y],
    "bottom-right": [x, y]
  },
  "field-size": [width, length],
  "field-unit": "unit name"
}
```

The path to the field image is relative to the JSON file. For simplicity, the image file should be in the same directory as the JSON file.

The field corners are the X and Y coordinates of the top-left and bottom-right pixels defining the rectangular boundary of the playable area in the field image. Non-rectangular playing areas are not supported.

The field size is the width and length of the playable area of the field in the provided units.

The field units are case-insensitive and can be in meters, cm, mm, inches, feet, yards, or miles. Singular, plural, and abbreviations are supported (eg"meter","meters", and"m"are all valid for specifying meters)