squiggles Documentation

Release 0.1.0

Jonathan Bayless

Feb 21, 2021

BASICS

A library for generating spline-based paths for robots.

The "squiggles" created by this path generation library allow for smooth, fast autonomous movements. Robots can follow the generated paths through the use of the wheel velocities calculated at each point *along with an appropriate feedback controller*.

CHAPTER

ONE

GETTING STARTED

1. Install the Library

The list of installation options and their instructions can be found in *Installation*.

2. Find Your Robot's Constraints

The guide in *Constraints* should help you identify the size and speed of your robot.

3. Generate Some Paths

You can follow along with the examples in *Example Paths* or jump straight into the *Library API*.

4. Add a Closed Loop Controller

It is possible to directly command the generated wheel velocities to a robot but any mismatch between your robot measurements and reality will cause your robot to go off course. Writing a closed-loop path following controller is an exercise left to the reader but the *Controller Suggestions* document has some tips to help get you started.

CHAPTER

TWO

TABLE OF CONTENTS

2.1 Installation

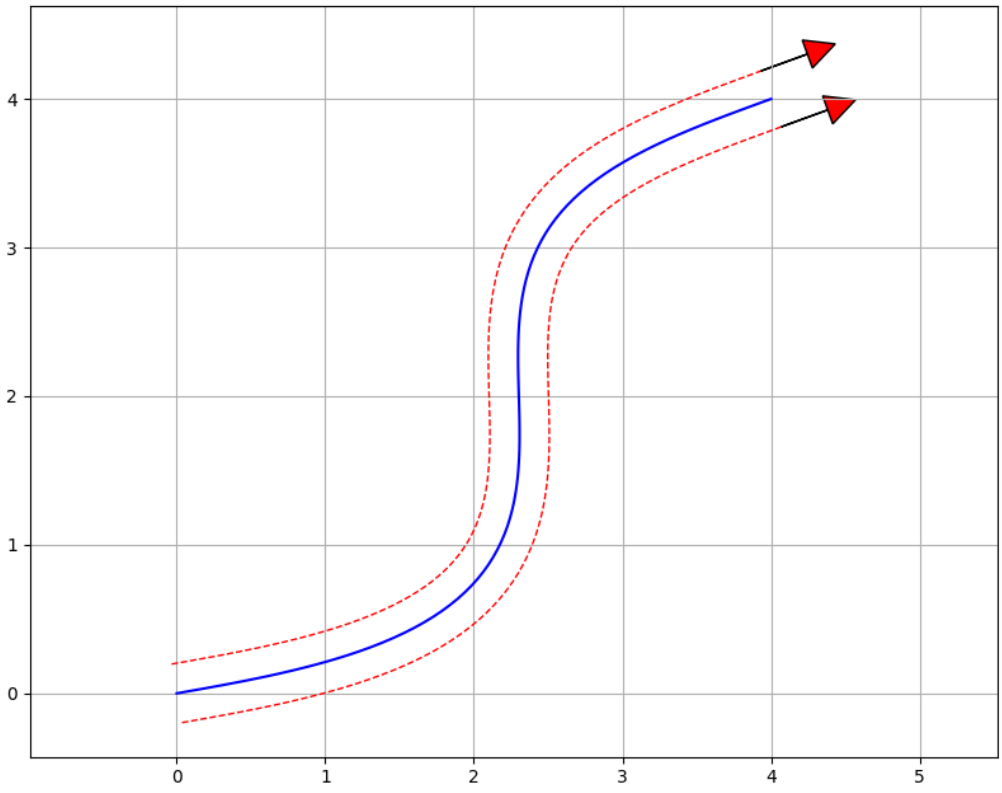
The Squiggles source code contains no external dependencies. You can include Squiggles in any existing project that uses C++20 standard by adding the contents of the *include* and *src* directories to your project.

You can also add Squiggles to your project as a static library by downloading the latest release from Github.

For instructions on building the library in a development environment see the Github CONTRIBUTING doc.

2.2 Example Paths

2.2.1 Basic Path



The above path can be created in three simple steps. First, define the Constraints with the robot's maximum velocity, acceleration, and jerk when driving:

```
#include "squiggles.hpp"

const double MAX_VEL   = 2.0; // in meters per second
const double MAX_ACCEL = 3.0; // in meters per second per second
const double MAX_JERK  = 6.0; // in meters per second per second per second

squiggles::Constraints constraints = squiggles::Constraints(MAX_VEL, MAX_ACCEL, MAX_
↪JERK);
```

Then measure the width between the robot's wheels and create a SplineGenerator with the constraints and the width measurement:

```
#include "squiggles.hpp"

const double MAX_VEL     = 2.0; // in meters per second
```

(continues on next page)

(continued from previous page)

```
const double MAX_ACCEL   = 3.0; // in meters per second per second
const double MAX_JERK    = 6.0; // in meters per second per second per second
const double ROBOT_WIDTH = 0.4; // in meters

squiggles::Constraints constraints = squiggles::Constraints(MAX_VEL, MAX_ACCEL, MAX_
↪JERK);
squiggles::SplineGenerator generator = squiggles::SplineGenerator(
  constraints,
  std::make_shared<squiggles::TankModel>(ROBOT_WIDTH, constraints));
```

And finally let's set the starting and ending poses as shown in the image above:

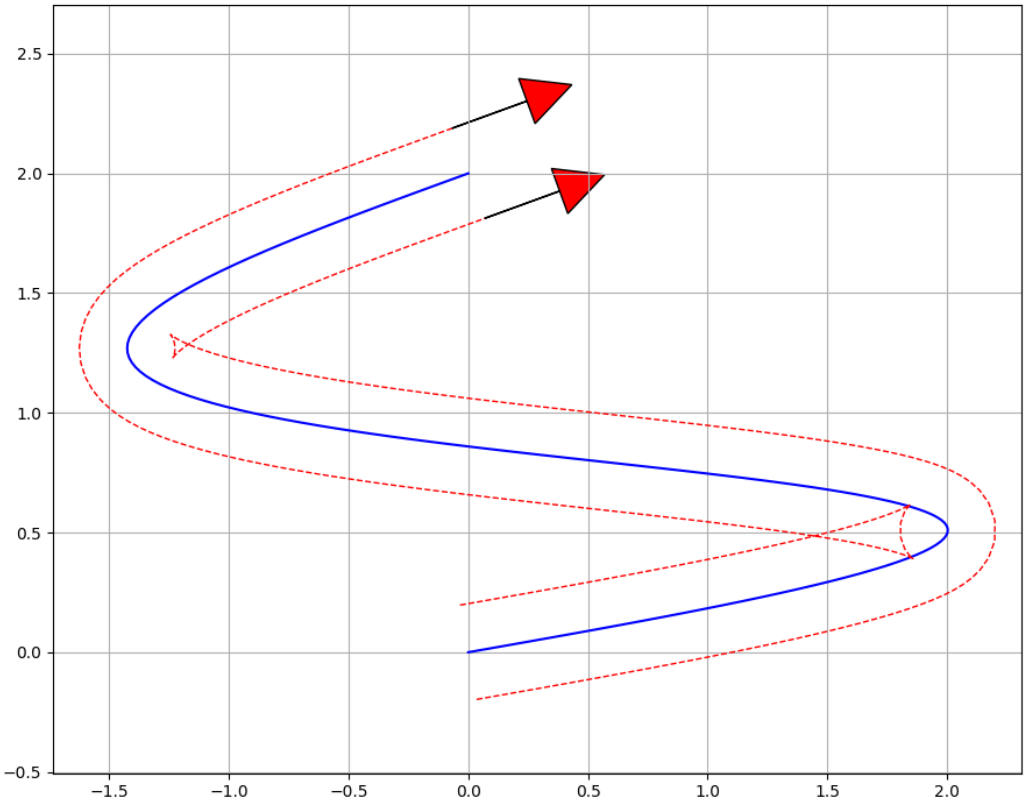
```
#include "squiggles.hpp"

const double MAX_VEL     = 2.0; // in meters per second
const double MAX_ACCEL   = 3.0; // in meters per second per second
const double MAX_JERK    = 6.0; // in meters per second per second per second
const double ROBOT_WIDTH = 0.4; // in meters

squiggles::Constraints constraints = squiggles::Constraints(MAX_VEL, MAX_ACCEL, MAX_
↪JERK);
squiggles::SplineGenerator generator = squiggles::SplineGenerator(
  constraints,
  std::make_shared<squiggles::TankModel>(ROBOT_WIDTH, constraints));

std::vector<squiggles::ProfilePoint> path = generator.generate({
  squiggles::Pose(0.0, 0.0, 1.0),
  squiggles::Pose(4.0, 4.0, 1.0)});
```

2.2.2 Tight Path

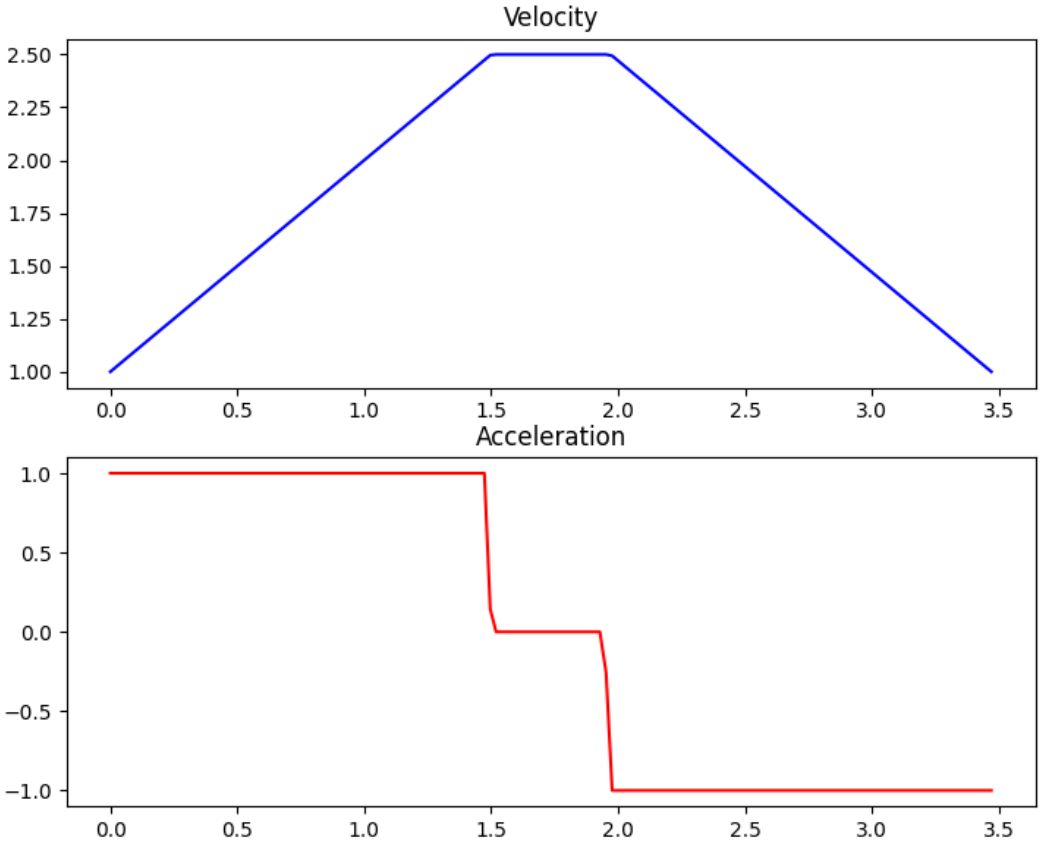


The generated paths can get a bit more interesting when trying to make tight turns. The path shown above sends negative velocities to the inner wheel during the turns in order to make turns in a small space.

We can reuse the `SplineGenerator` from the previous section for this second path. It is often easiest, though not required, to create the `SplineGenerator` once and call its `generate()` method as many times as needed.

```
std::vector<squiggles::ProfilePoint> path = generator.generate({
  squiggles::Pose(0.0, 0.0, 1.0),
  squiggles::Pose(0.0, 2.0, 1.0)});
```

2.3 Constraints



The robot's Constraints provide the maximum allowable dynamics for the generated paths. Careful measurement and configuration of these parameters ensures that the path will not expect the robot to move more quickly than it actually can. Resolving such discrepancies in the generated path and reality is an important first step in ensuring that the robot performs reliably.

2.3.1 Maximum Velocity

The simplest constraint for the robot's motion is its **Maximum Velocity**. This value can be found by one of two methods: calculation and measurement. There are a lot of options for measuring the maximum velocity; we'll focus on the calculation option here.

Calculating the theoretical maximum velocity can be done easily without using the actual robot. So long as you know the **wheel diameter** and **motor velocity** for the drivetrain you can calculate the robot's maximum velocity.

Let's say we have 4 inch wheels and 200rpm motors on the robot's drive. We'll first convert these values into the right units:

```
4 inches * 0.0254 inches/meter = 0.1016 meters
```

```
200 rpm / 60 seconds/minute = 3.333 revolutions per second
```

We then find the *circumference* of the wheels and multiply by the rate of rotation of the motor to get the velocity.

```
Circumference = PI * Diameter = 3.14 * 0.1016 = 0.319 meters
Velocity = Circumference * Rotation Rate = 0.319 * 3.333 = 1.063 meters per second
```

2.3.2 Maximum Acceleration

Let's assume that we're using the same robot as above with the 4 inch wheels and 200 rpm motors. If we're using the *VEX V5 Smart Motors <https://www.vexrobotics.com/276-4840.html#additional>* on the drive then our 200 rpm motors will have a stall torque of 1.05 Nm. We won't be able to hit that value while keeping the robot moving and we don't want to push the motors *that* hard all the time. Let's set our maximum torque at 0.5 Nm for the robot's movements to keep the current down and the motors happy.

We can use the physics equations for torque and force to find the maximum acceleration for our robot. First, let's find the maximum force that the robot can deliver.

```
Torque = wheel radius * force
0.5 Nm = (0.1016 meters / 2) * force
Force = 9.843 N
```

We can then use Newton's second law and the mass of our fictional robot to find the maximum acceleration. It is important to remember that we are looking for the *sum* of the forces so we will account for each motor on the drive. Let's give our fictional robot 4 drive motors for a total of `9.843 * 4 = 39.372` Newtons.

```
Sum of Force = mass * acceleration
39.372 = 10 Kg * acceleration
acceleration = 3.937 meters per second per second
```

2.3.3 Maximum Jerk

The calculations for maximum jerk are not nearly as convenient as the above calculations for velocity and acceleration. The easiest option for this parameter is to set it to an arbitrary value that's a bit larger than your acceleration, like twice as large. This can be a good fiddle-factor to get the robot's movement to be smoother or more aggressive than default.

2.3.4 Resources

- Class Reference

2.4 Controller Suggestions

The *motion profiles* help the robot's path-following abilities considerably but there are many factors that could prevent the robot from following the generated path. It is recommended to pair the output of Squiggles with a feedback controller.

A velocity PID controller is an easy start but a controller optimized for path following is the best choice. Small errors in a velocity controller are fine with systems like a flywheel but can cause a path-following robot to go wildly off course.

2.4.1 Pure Pursuit Controller

The Pure Pursuit Controller is the defacto standard for closed loop path following. Instead of trying to make the robot move to the nearest point along the path when it diverges the Pure Pursuit Controller anticipates moving to a point ahead on the path. The target point is the nearest point plus a *look ahead distance*. The Pure Pursuit Controller closes the control loop by using the robot's measured position – calculated by something like odometry – and finds the look ahead point from that measured position.

The Pure Pursuit Controller has been used extensively in FRC. A couple examples of its use for FRC are listed below:

- Team 254's Implementation
- XiaoXie's Implementation

The Pure Pursuit algorithm is best explained by Alonzo Kelly's white paper. It is often helpful to reference this document in conjunction with an example Pure Pursuit implementation when writing your own take on the controller.

2.4.2 Ramsete Controller

The Ramsete Controller is another controller that is used for following paths. It does not look ahead to follow the path like the Pure Pursuit Controller but is best suited for correcting small errors.

Like Pure Pursuit, the Ramsete Controller has become quite popular in FRC. The WPILib library includes an implementation for the Ramsete Controller:

- WPILib Implementation

The following white papers are excellent resources to use when constructing your Ramsete Controller:

- Tyler Veness' Controls Engineering in the FIRST Robotics Competition
- The original paper detailing the algorithm

2.5 Physical Models

Physical Models define the translation from a velocity and curvature into wheel velocities. This additional limit is imposed to prevent conditions where the robot's linear velocity is within the velocity constraint but one of its wheels would need to exceed the velocity constraint to match the curvature at the point. The generated wheel velocities can also be used as a feedforward value in the control loop for following the path.

2.5.1 Tank Drive Model

The TankModel defines a Tank Drive model according to the unicycle model. This model translates the linear velocity and curvature at the point into two values: a velocity for the left side of the robot and a velocity for the right side. These velocities are used when constraining the robot's linear velocity. Additionally, the acceleration of each wheel is accounted for using the robot's linear acceleration and the curvature at the point. This additional constraint on the acceleration is applied during the motion profile phase of the path generation.

2.6 The Math

2.6.1 Quintic Polynomials

Each point along the generated path is determined by a Quintic Polynomial, or a polynomial function of the form $x(t) = a*t^5 + b*t^4 + c*t^3 + d*t^2 + e*t + f$. t is a unitless parameter that represents the length along the path that the point occurs at, in the range of $[0, 1]$. There is one polynomial for each dimension that the robot travels through; we have an x polynomial and a y polynomial for our 2D paths.

The source for the Quintic Polynomial coefficients is Atsushi Sakai's Python Robotics. The coefficients are set in accordance with the laws of physics regarding linear movement. The t parameter takes the place of time in the physics equations.

The example code in Python Robotics solves the equation for the coefficients dynamically with numpy but it can be computed statically. The convenient Symbolab Matrix Equations Calculator solved the matrix of coefficients as a function of the t parameter.

The t parameter maps directly to a theoretical duration for the path. The generation process starts with a short duration, as a faster path would be ideal, and incrementally generates longer and longer paths until the robot's *Constraints* are met.

Note: This duration will differ from the end duration of the path after motion profiling takes place.

2.6.2 Motion Profiles

The trapezoidal motion profile is applied to the path after the 2D positions are computed with the polynomials. This motion profile constrains the maximum velocity and the maximum acceleration for the robot while leaving jerk unconstrained. The *Quintic Polynomials* constrain the path's acceleration and jerk but do not constrain the velocity so adding the motion profile is a necessary step.

The profile generates the target velocity and acceleration for the robot at each point along the path through two passes: a forward pass and then a backward pass. After the velocity and acceleration are determined we reference the physics equations for linear motion again to set a more accurate time stamp for each of the positions. The forward pass first sets the starting velocity to the preferred starting velocity set by the user in place of the "dummy" velocity used when calculating the polynomial. This pass then limits the velocity at each point to no greater than the maximum and then uses that new velocity value to calculate the necessary acceleration value at the previous point. The backward pass first sets the ending velocity to the preffered ending velocity and then performs roughly the same limiting as the forward pass but in reverse. These two passes get the starting and ending velocities matching the velocities set by the user and keep the path velocities within the limits.

These new velocities and accelerations are used to find new timestamps for each point along the path given the linear distance between each. These new timestamps do not last long, though, as the next step is to create new points at each increment of the dt value by interpolating between the points at the aforementioned timestamps.

2.7 Library API

2.7.1 Class Hierarchy

2.7.2 File Hierarchy

2.7.3 Full API

2.8 Release Notes

2.8.1 0.1.0

Initial Public Release

2.9 Resources

The following resources were referenced during the writing of Squiggles:

- Team 254's contributions to WPILib
- Jaci Brunning's Pathfinder Library
- Atsushi Sakai's Python Robotics