squiggles Documentation

Release 0.1.0

Jonathan Bayless

Jun 19, 2022

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

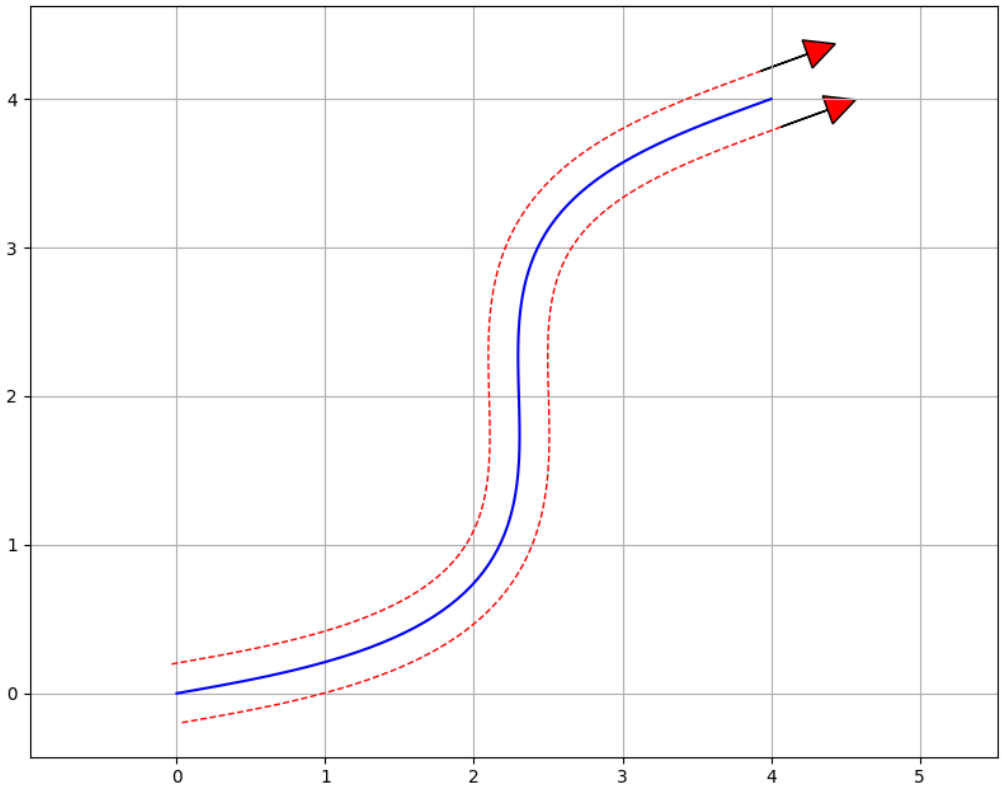
The Squiggles source code contains no external dependencies. You can include Squiggles in any existing project that uses C++20 standard by adding the contents of the *include* and *src* directories to your project.

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"

```

(continues on next page)

(continued from previous page)

```
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));
```

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

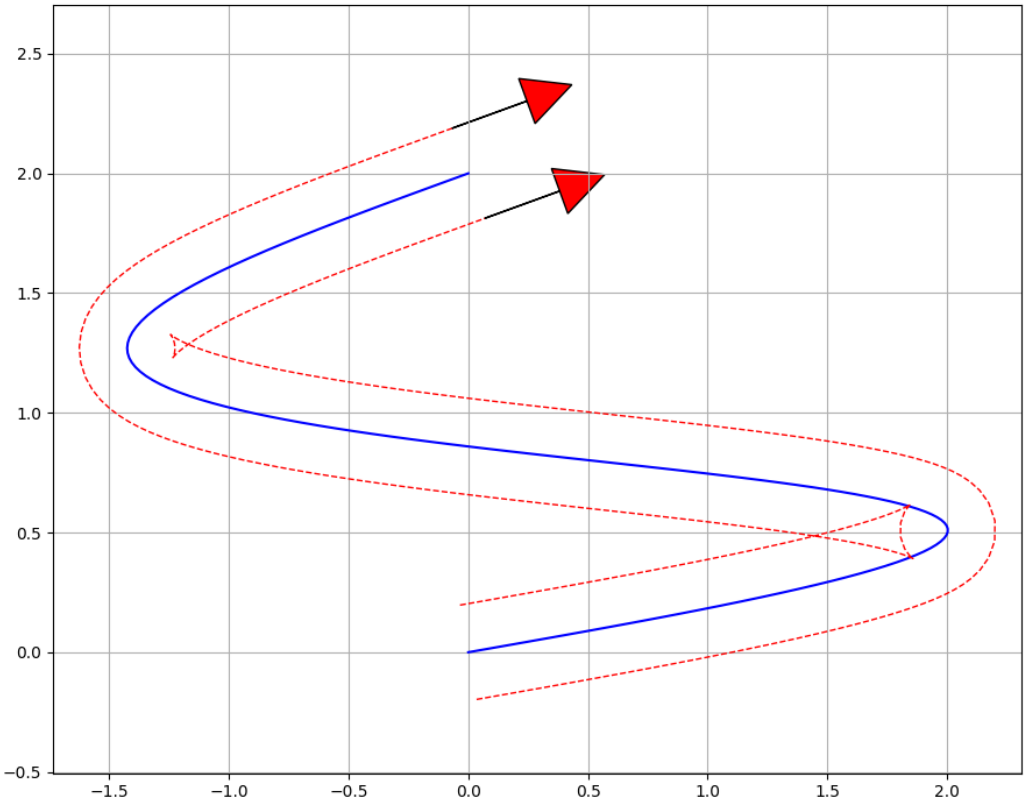
```
#include "squiggles.hpp"

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

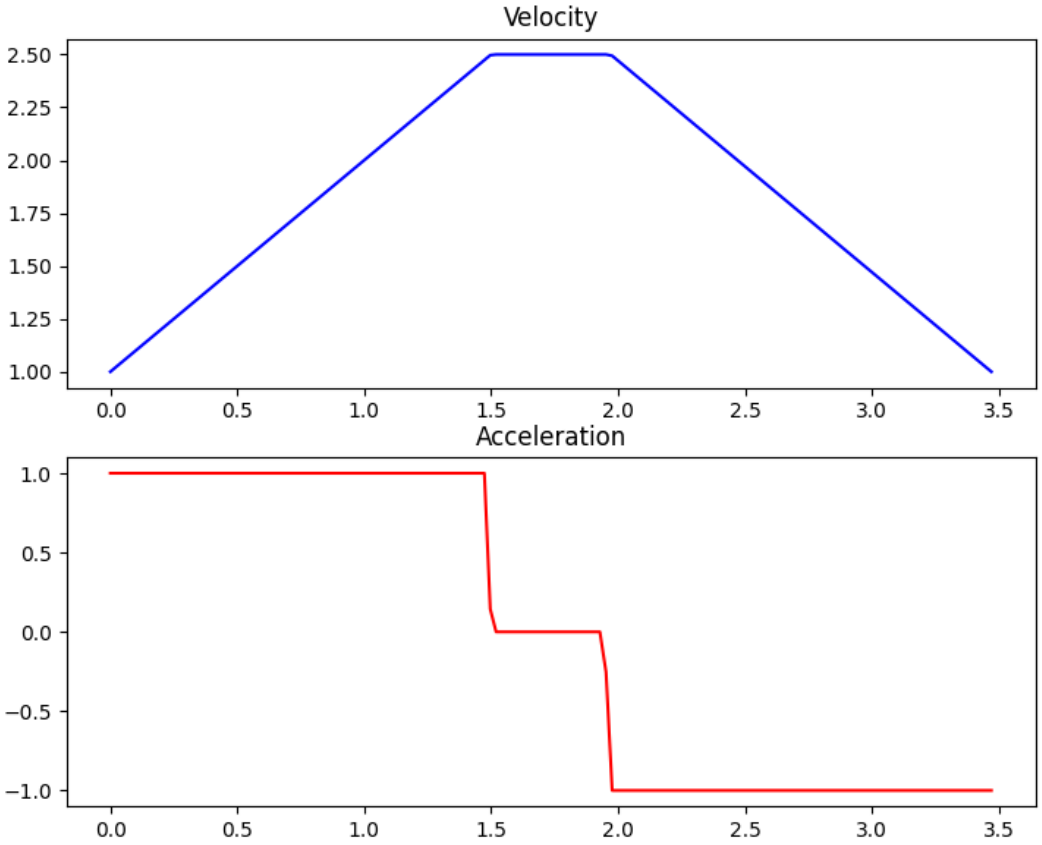


The generated paths can get a bit more interesting when trying to make tight turns. The path shown above sends negative velocities to the inner wheel during the turns in order to make turns in a small space.

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 the Purdue SIGBots Wiki. 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 resources are helpful to use when constructing your Ramsete Controller:

- Purdue SIGBots Wiki Article
- 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

Namespaces

Namespace squiggles

Contents

Detailed Description

Classes

Functions

- *Function squiggles::deserialize_path*
- *Function squiggles::deserialize_pathfinder_path*
- *Function squiggles::nearly_equal*
- *Function squiggles::serialize_path*
- *Template Function squiggles::sgn*

Namespace std

Classes and Structs

Struct Constraints

- Defined in file_main_include_constraints.hpp

Struct Documentation

struct `squiggles::Constraints`

Public Functions

inline **`Constraints`**(double imax_vel, double imax_accel = std::numeric_limits<double>::max(), double imax_jerk = std::numeric_limits<double>::max(), double imax_curvature = 1000, double imin_accel = std::nan(""))

Defines the motion constraints for a path.

Parameters

- **`imax_vel`** – The maximum allowable velocity for the robot in meters per second.
- **`imax_accel`** – The maximum allowable acceleration for the robot in meters per second per second.
- **`imax_jerk`** – The maximum allowable jerk for the robot in meters per second per second per second (m/s^3).
- **`imax_curvature`** – The maximum allowable change in heading in radians per second. This is not set to the numeric limits by default as that will allow for wild paths.
- **`imin_accel`** – The minimum allowable acceleration for the robot in meters per second per second.

inline std::string **`to_string`**() const

Serializes the *Constraints* data for debugging.

Returns The *Constraints* data.

Public Members

double **max_vel**

double **max_accel**

double **max_jerk**

double **min_accel**

double **max_curvature**

Struct ProfilePoint

- Defined in file_main_include_geometry_profilepoint.hpp

Struct Documentation

struct `squiggles::ProfilePoint`

Public Functions

inline **ProfilePoint**(*ControlVector* ivector, std::vector<double> iwheel_velocities, double icurvature, double itime)

Defines a state along a motion profiled path.

Parameters

- **ivector** – The pose and associated dynamics at this state in the path.
- **iwheel_velocities** – The component of the robot's velocity provided by each wheel in meters per second.
- **icurvature** – The degree to which the curve deviates from a straight line at this point in 1 / meters.
- **itime** – The timestamp for this state relative to the start of the path in seconds.

ProfilePoint() = default

inline std::string **to_string**() const

Serializes the Profile Point data for debugging.

Returns The Profile Point data.

inline std::string **to_csv**() const

inline bool **operator==**(const *ProfilePoint* &other) const

Public Members

ControlVector **vector**

std::vector<double> **wheel_velocities**

double **curvature**

double **time**

Friends

inline friend std::ostream &**operator<<**(std::ostream &os, const *ProfilePoint* &p)

Struct SplineGenerator::ConstrainedState

- Defined in file_main_include_spline.hpp

Nested Relationships

This struct is a nested type of *Class SplineGenerator*.

Struct Documentation

struct `squiggles::SplineGenerator::ConstrainedState`

An intermediate value used in the parameterization step. Adds the constrained values from the motion profile to the output from the "naive" generation step.

Public Functions

inline **ConstrainedState**(*Pose* ipose, double icurvature, double idistance, double imax_vel, double imin_accel, double imax_accel)

ConstrainedState() = default

inline std::string **to_string**() const

Public Members

Pose **pose** = *Pose*()

double **curvature** = 0

double **distance** = 0

double **max_vel** = 0

double **min_accel** = 0

double **max_accel** = 0

Struct SplineGenerator::GeneratedPoint

- Defined in file_main_include_spline.hpp

Nested Relationships

This struct is a nested type of *Class SplineGenerator*.

Struct Documentation

struct `squiggles::SplineGenerator::GeneratedPoint`

The output of the initial, "naive" generation step. We discard the derivative values to replace them with values from a motion profile.

Public Functions

inline **GeneratedPoint**(*Pose* ipose, double icurvature = 0.0)

inline std::string **to_string**() const

Public Members

Pose **pose**

double **curvature**

Struct SplineGenerator::GeneratedVector

- Defined in file_main_include_spline.hpp

Nested Relationships

This struct is a nested type of *Class SplineGenerator*.

Struct Documentation

struct `squiggles::SplineGenerator::GeneratedVector`

An intermediate value used in the "naive" generation step. Contains the final *GeneratedPoint* value that will be returned as well as the spline's derivative values to perform the initial check against the constraints.

Public Functions

inline **GeneratedVector**(*GeneratedPoint* ipoint, double ivel, double iaccel, double ijerk)

inline std::string **to_string**() const

Public Members

GeneratedPoint **point**

double **vel**

double **accel**

double **jerk**

Class ControlVector

- Defined in file_main_include_geometry_controlvector.hpp

Class Documentation

class `squiggles::ControlVector`

Public Functions

inline **ControlVector**(*Pose* ipose, double ivel = std::nan(""), double iaccel = 0.0, double ijerk = 0.0)

A vector used to specify a state along a hermite spline.

Parameters

- **ipose** – The 2D position and heading.
- **ivel** – The velocity component of the vector.
- **iaccel** – The acceleration component of the vector.
- **ijerk** – The jerk component of the vector.

ControlVector() = default

inline std::string **to_string**() const

Serializes the Control Vector data for debugging.

Returns The Control Vector data.

inline std::string **to_csv**() const

inline bool **operator==**(const *ControlVector* &other) const

Public Members

Pose **pose**

double **vel**

double **accel**

double **jerk**

Class PassthroughModel

- Defined in file_main_include_physicalmodel_passthroughmodel.hpp

Inheritance Relationships

Base Type

- public squiggles::PhysicalModel (*Class PhysicalModel*)

Class Documentation

class `squiggles::PassthroughModel` : public squiggles::*PhysicalModel*

Public Functions

`PassthroughModel()` = default

Defines a Physical Model that imposes no constraints of its own.

inline *Constraints* **constraints**([[maybe_unused]] const *Pose* pose, [[maybe_unused]] double curvature, double vel) override

inline std::vector<double> **linear_to_wheel_vels**([[maybe_unused]] double lin_vel, [[maybe_unused]] double curvature) override

inline virtual std::string **to_string**() const override

Class PhysicalModel

- Defined in file_main_include_physicalmodel_physicalmodel.hpp

Inheritance Relationships

Derived Types

- `public squiggles::PassthroughModel` (*Class PassthroughModel*)
- `public squiggles::TankModel` (*Class TankModel*)

Class Documentation

class `squiggles::PhysicalModel`

Subclassed by *squiggles::PassthroughModel*, *squiggles::TankModel*

Public Functions

virtual *Constraints* **constraints**(const *Pose* pose, double curvature, double vel) = 0

Calculate a set of stricter constraints for the path at the given state than the general constraints based on the robot's kinematics.

Parameters

- **pose** – The 2D pose for this state in the path.
- **curvature** – The change in heading at this state in the path in 1 / meters.
- **vel** – The linear velocity at this state in the path in meters per second.

virtual std::vector<double> **linear_to_wheel_vels**(double linear, double curvature) = 0

Converts a linear velocity and desired curvature into the component for each wheel of the robot.

Parameters

- **linear** – The linear velocity for the robot in meters per second.
- **curvature** – The change in heading for the robot in 1 / meters.

virtual std::string **to_string**() const = 0

Class Pose

- Defined in file_main_include_geometry_pose.hpp

Class Documentation

class **squiggles::Pose**

Public Functions

inline **Pose**(double ix, double iy, double iyaw)

Specifies a point and heading in 2D space.

Parameters

- **ix** – The x position of the point in meters.
- **iy** – The y position of the point in meters.
- **iyaw** – The heading at the point in radians.

Pose() = default

inline double **dist**(const *Pose* &other) const

Calculates the Euclidean distance between this pose and another.

Parameters **other** – The point from which the distance will be calculated.

Returns The distance between this pose and Other.

inline std::string **to_string**() const

Serializes the *Pose* data for debugging.

Returns The *Pose* data.

inline std::string **to_csv**() const

inline bool **operator==**(const *Pose* &other) const

Public Members

double **x**

double **y**

double **yaw**

Class QuinticPolynomial

- Defined in file_main_include_math_quinticpolynomial.hpp

Class Documentation

class squiggles::**QuinticPolynomial**

Public Functions

QuinticPolynomial(double s_p, double s_v, double s_a, double g_p, double g_v, double g_a, double t)

Defines the polynomial function for a spline in one dimension.

Parameters

- **s_p** – The starting position of the curve in meters.
- **s_v** – The starting velocity of the curve in meters per second.
- **s_a** – The starting acceleration of the curve in meters per second per second.
- **g_p** – The goal or ending position of the curve in meters.
- **g_v** – The goal or ending velocity of the curve in meters per second.
- **g_a** – The goal or ending acceleration of the curve in meters per second per second.
- **t** – The desired duration for the curve in seconds.

double **calc_point**(double t)

Calculates the values of the polynomial and its derivatives at the given time stamp.

double **calc_first_derivative**(double t)

double **calc_second_derivative**(double t)

double **calc_third_derivative**(double t)

inline std::string **to_string**() const

Serializes the Quintic Polynomial data for debugging.

Returns The Quintic Polynomial data.

Protected Attributes

double **a0**

The coefficients for each term of the polynomial.

double **a1**

double **a2**

double **a3**

double **a4**

double **a5**

Class SplineGenerator

- Defined in file_main_include_spline.hpp

Nested Relationships

Nested Types

- *Struct SplineGenerator::ConstrainedState*
- *Struct SplineGenerator::GeneratedPoint*
- *Struct SplineGenerator::GeneratedVector*

Class Documentation

class `squiggles::`**SplineGenerator**

Public Functions

SplineGenerator(*Constraints* iconstraints, std::shared_ptr<*PhysicalModel*> imodel = std::make_shared<*PassthroughModel*>(), double idt = 0.1)

Generates curves that match the given motion constraints.

Parameters

- **iconstraints** – The maximum allowable values for the robot's motion.
- **imodel** – The robot's physical characteristics and constraints
- **idt** – The difference in time in seconds between each state for the generated paths.

std::vector<*ProfilePoint*> **generate**(std::vector<*Pose*> iwaypoints, bool fast = false)

Creates a motion profiled path between the given waypoints.

Parameters

- **iwaypoints** – The list of poses that the robot should reach along the path.
- **fast** – If true, the path optimization process will stop as soon as the constraints are met. If false, the optimizer will find the smoothest possible path between the points.

Returns A series of robot states defining a path between the poses.

std::vector<*ProfilePoint*> **generate**(std::initializer_list<*Pose*> iwaypoints, bool fast = false)

std::vector<*ProfilePoint*> **generate**(std::vector<*ControlVector*> iwaypoints)

Creates a motion profiled path between the given waypoints.

Parameters **iwaypoints** – The list of vectors that the robot should reach along the path.

Returns A series of robot states defining a path between the vectors.

std::vector<*ProfilePoint*> **generate**(std::initializer_list<*ControlVector*> iwaypoints)

std::vector<*GeneratedVector*> **gen_single_raw_path**(*ControlVector* start, *ControlVector* end, int duration, double start_vel, double end_vel)

std::vector<*GeneratedPoint*> **gradient_descent**(*ControlVector* &start, *ControlVector* &end, bool fast)

Runs a Gradient Descent algorithm to minimize the linear acceleration, linear jerk, and curvature for the generated path.

This is used when there is not a start/end velocity specified for a given path.

template<class **Iter**>
std::vector<*ProfilePoint*> **_generate**(*Iter* start, *Iter* end, bool fast)

The actual function called by the "generate" functions.

Parameters

- **start** – An iterator pointing to the first *ControlVector* in the path
- **end** – An iterator pointting to the last *ControlVector* in the path

Returns The points from each path concatenated together

std::vector<*GeneratedPoint*> **gen_raw_path**(*ControlVector* &start, *ControlVector* &end, bool fast)

Performs the "naive" generation step.

This step calculates the spline polynomials that fit within the *SplineGenerator*'s acceleration and jerk constraints and returns the points that form the curve.

std::vector<*ProfilePoint*> **parameterize**(const *ControlVector* start, const *ControlVector* end, const std::vector<*GeneratedPoint*> &raw_path, const double preferred_start_vel, const double preferred_end_vel, const double start_time)

Imposes a linear motion profile on the raw path.

This step creates the velocity and acceleration values to command the robot at each point along the curve.

std::vector<*ProfilePoint*> **integrate_constrained_states**(std::vector<*ConstrainedState*> constrainedStates)

Finds the new timestamps for each point along the curve based on the motion profile.

ProfilePoint **get_point_at_time**(const *ControlVector* start, const *ControlVector* end, std::vector<*ProfilePoint*> points, double t)

Finds the *ProfilePoint* on the profiled curve for the given timestamp.

This with interpolate between points on the curve if a point with an exact matching timestamp is not found.

ProfilePoint **lerp_point**(*QuinticPolynomial* x_qp, *QuinticPolynomial* y_qp, *ProfilePoint* start, *ProfilePoint* end, double i)

Linearly interpolates between points along the profiled curve.

QuinticPolynomial **get_x_spline**(const *ControlVector* start, const *ControlVector* end, const double duration)

Returns the spline curve for the given control vectors and path duration.

QuinticPolynomial **get_y_spline**(const *ControlVector* start, const *ControlVector* end, const double duration)

void **enforce_accel_lims**(*ConstrainedState* *state)

Applies the general constraints and model constraints to the given state.

void **forward_pass**(*ConstrainedState* *predecessor, *ConstrainedState* *successor)

Imposes the motion profile constraints on a segment of the path from the perspective of iterating forwards through the path.

void **backward_pass**(*ConstrainedState* *predecessor, *ConstrainedState* *successor)

Imposes the motion profile constraints on a segment of the path from the perspective of iterating backwards through the path.

double **vf**(double vi, double a, double ds)

Calculates the final velocity for a path segment.

double **ai**(double vf, double vi, double s)

Calculates the initial acceleration needed to match the segments' velocities.

Public Members

const double **K_DEFAULT_VEL** = 1.0

This is factor is used to create a "dummy velocity" in the initial path generation step one or both of the preferred start or end velocities is zero. The velocity will be replaced with the preferred start/end velocity in parameterization but a nonzero velocity is needed for the spline calculation.

This was 1.2 in the WPILib example but that large of a value seems to create wild paths, 0.12 worked better in testing with VEX-sized paths.

Public Static Attributes

static constexpr double **K_EPSILON** = 1e-5

Values that are closer to each other than this value are considered equal.

Protected Attributes

Constraints **constraints**

The maximum allowable values for the robot's motion.

std::shared_ptr<*PhysicalModel*> **model**

Defines the physical structure of the robot and translates the linear kinematics to wheel velocities.

double **dt**

The time difference between each value in the generated path.

const int **T_MIN** = 2

The minimum and maximum durations for a path to take. A larger range allows for longer possible paths at the expense of a longer path generation time.

const int **T_MAX** = 15

const int **MAX_GRAD_DESCENT_ITERATIONS** = 10

struct **ConstrainedState**

An intermediate value used in the parameterization step. Adds the constrained values from the motion profile to the output from the "naive" generation step.

Public Functions

inline **ConstrainedState**(*Pose* ipose, double icurvature, double idistance, double imax_vel, double imin_accel, double imax_accel)

ConstrainedState() = default

inline std::string **to_string**() const

Public Members

Pose **pose** = *Pose*()

double **curvature** = 0

double **distance** = 0

double **max_vel** = 0

double **min_accel** = 0

double **max_accel** = 0

struct **GeneratedPoint**

The output of the initial, "naive" generation step. We discard the derivative values to replace them with values from a motion profile.

Public Functions

inline **GeneratedPoint**(*Pose* ipose, double icurvature = 0.0)

inline std::string **to_string**() const

Public Members

Pose **pose**

double **curvature**

struct **GeneratedVector**

An intermediate value used in the "naive" generation step. Contains the final *GeneratedPoint* value that will be returned as well as the spline's derivative values to perform the initial check against the constraints.

Public Functions

inline **GeneratedVector**(*GeneratedPoint* ipoint, double ivel, double iaccel, double ijerk)

inline std::string **to_string**() const

Public Members

GeneratedPoint **point**

double **vel**

double **accel**

double **jerk**

Class TankModel

- Defined in file_main_include_physicalmodel_tankmodel.hpp

Inheritance Relationships

Base Type

- `public squiggles::PhysicalModel` (*Class PhysicalModel*)

Class Documentation

class `squiggles::TankModel` : public squiggles::*PhysicalModel*

Public Functions

`TankModel`(double itrack_width, *Constraints* ilinear_constraints)

Defines a model of a tank drive or differential drive robot.

Parameters

- **itrack_width** – The distance between the the wheels on each side of the robot in meters.
- **ilinear_constraints** – The maximum values for the robot's movement.

virtual *Constraints* **constraints**(const *Pose* pose, double curvature, double vel) override

Calculate a set of stricter constraints for the path at the given state than the general constraints based on the robot's kinematics.

Parameters

- **pose** – The 2D pose for this state in the path.
- **curvature** – The change in heading at this state in the path in 1 / meters.
- **vel** – The linear velocity at this state in the path in meters per second.

virtual std::vector<double> **linear_to_wheel_vels**(double lin_vel, double curvature) override

Converts a linear velocity and desired curvature into the component for each wheel of the robot.

Parameters

- **linear** – The linear velocity for the robot in meters per second.
- **curvature** – The change in heading for the robot in 1 / meters.

virtual std::string **to_string**() const override

Functions

Function squiggles::deserialize_path

- Defined in file_main_include_io.hpp

Function Documentation

std::optional<std::vector<*ProfilePoint*>> `squiggles::deserialize_path`(std::istream &in)

Converts CSV data into a path.

Parameters `in` – The input stream containing the CSV data. This is usually a file.

Returns The path specified by the CSV data or std::nullopt if de-serializing the path was unsuccessful.

Function squiggles::deserialize_pathfinder_path

- Defined in file_main_include_io.hpp

Function Documentation

std::optional<std::vector<*ProfilePoint*>> `squiggles::deserialize_pathfinder_path`(std::istream &left, std::istream &right)

Converts CSV data from the Pathfinder library's format to a Squiggles path.

NOTE: this code translates data from Jaci Brunning's Pathfinder library. The source for that library can be found at: https://github.com/JaciBrunning/Pathfinder/

Parameters

- `left` – The input stream containing the left wheels' CSV data. This is usually a file.
- `right` – The input stream containing the right wheels' CSV data. This is usually a file.

Returns The path specified by the CSV data or std::nullopt if de-serializing the path was unsuccessful.

Function squiggles::nearly_equal

- Defined in file_main_include_math_utils.hpp

Function Documentation

inline bool `squiggles::nearly_equal`(const double &a, const double &b, double epsilon = 1e-5)

Function squiggles::serialize_path

- Defined in file_main_include_io.hpp

Function Documentation

int `squiggles::serialize_path`(std::ostream &out, std::vector<*ProfilePoint*> path)

Writes the path data to a CSV file.

Parameters

- **out** – The output stream to write the CSV data to. This is usually a file.
- **path** – The path to serialized

Returns 0 if the path was serialized succesfully or -1 if an error occurred.

Template Function squiggles::sgn

- Defined in file_main_include_math_utils.hpp

Function Documentation

template<class **T**>
inline int `squiggles::sgn`(*T* v)

Returns the sign value of the given value.

Returns 1 if the value is positive, -1 if the value is negative, and 0 if the value is 0.

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

INDEX

S