robotpy-pathfinder Documentation

Release 0.2.6.post0.dev2

RobotPy

Mar 09, 2020

CONTENTS:

Note: pathfinder is deprecated, we highly recommend using the WPILib Trajectory Generation/Following support introduced in the 2020 season. All trajectory related features should be usable from RobotPy.

These are python bindings around Jaci B's PathFinder library. From the original documentation:

Cross-Platform, Multi-Use Motion Profiling and Trajectory Generation.

Pathfinder is a library for generating Motion Profiles, a way to smoothly fit and follow a trajectory based upon given waypoints.

Note: This requires C++ 11 and Python 3.5+

CHAPTER

ONE

USAGE

Installation and usage documentation can be found at http://robotpy-pathfinder.readthedocs.io

1.1 Using Pathfinder

Note: This documentation is a Python translation of Jaci's Java notes that can be found at https://github.com/JacisNonsense/Pathfinder/wiki/Pathfinder-for-FRC—Java

1.1.1 Installation (RobotPy on a RoboRIO)

Use robotpy-installer to install the precompiled package.

```
robotpy-installer download-opkg python37-robotpy-pathfinder
robotpy-installer install-opkg python37-robotpy-pathfinder
```

1.1.2 Installation (Windows)

We now publish Windows binaries for pathfinder, you should be able to install them via pip on supported versions of Python 3:

```
py -m pip install robotpy-pathfinder
```

1.1.3 Installation (other)

Note that this requires a C++11 compiler to be present on your system, as I'm not currently publishing non-Windows wheels of this library.

```
pip3 install 'pybind11>=2.2'
pip3 install robotpy-pathfinder
```

1.1.4 Generating a Trajectory

Whenever you want to generate a trajectory, you can do so by defining a set of waypoints and calling *pathfinder.generate()*:

```
import pathfinder as pf

points = [
    pf.Waypoint(-4, -1, math.radians(-45.0)),    # Waypoint @ x=-4, y=-1, exit angle=-
↪45 degrees
    pf.Waypoint(-2, -2, 0),                      # Waypoint @ x=-2, y=-2, exit angle=0␣
↪radians
    pf.Waypoint(0, 0, 0),                        # Waypoint @ x=0, y=0,   exit angle=0␣
↪radians
]

info, trajectory = pf.generate(points, pf.FIT_HERMITE_CUBIC, pf.SAMPLES_HIGH,
                               dt=0.05, # 50ms
                               max_velocity=1.7,
                               max_acceleration=2.0,
                               max_jerk=60.0)
```

You can also modify the trajectory for either Swerve or Tank drive:

```
modifier = pf.modifiers.TankModifier(trajectory).modify(0.5)
# OR
modifier = pf.modifiers.SwerveModifier(trajectory).modify(0.5, 0.6)
```

Note: It can take a really long time to generate a trajectory on a RoboRIO, but very little time on a modern computer. You can take advantage of this by pre-generating the trajectory before deployment. When using with RobotPy, you can use the following pattern to pregenerate them automatically before you deploy code to the robot:

```
import os.path
import pickle

# because of a quirk in pyfrc, this must be in a subdirectory
# or the file won't get copied over to the robot
pickle_file = os.path.join(os.path.dirname(__file__), 'trajectory.pickle')

if wpilib.RobotBase.isSimulation():
    # generate the trajectory here

    # and then write it out
    with open(pickle_file, 'wb') as fp:
        pickle.dump(trajectory, fp)
else:
    with open('fname', 'rb') as fp:
        trajectory = pickle.load(fp)
```

This works because whenever you run `robot.py deploy`, your robot code modules are imported and executed.

1.1.5 Following a Trajectory

To get your robot to follow a trajectory, you can use the *EncoderFollower* object. As the name suggests, this will use encoders as feedback to guide your robot along the trajectory. It is important that your time step passed into your generate call is the same as the time difference between control loop iterations, otherwise you may find your path tracking inaccurately.

Tank Drive

Create two *EncoderFollower* objects, one for the left and one for the right:

```
from pathfinder.followers import EncoderFollower

left = EncoderFollower(modifier.getLeftTrajectory())
right = EncoderFollower(modifier.getRightTrajectory())
```

When you're ready to start following:

Setup your encoder details:

```
# Encoder Position is the current, cumulative position of your encoder. If
# you're using an SRX, this will be the 'getEncPosition' function.
# 1000 is the amount of encoder ticks per full revolution
# Wheel Diameter is the diameter of your wheels (or pulley for a track system) in␣
↪meters
left.configureEncoder(encoder_position, 1000, wheel_diameter)
```

Set your PID/VA variables:

```
# The first argument is the proportional gain. Usually this will be quite high
# The second argument is the integral gain. This is unused for motion profiling
# The third argument is the derivative gain. Tweak this if you are unhappy with the␣
↪tracking of the trajectory
# The fourth argument is the velocity ratio. This is 1 over the maximum velocity you␣
↪provided in the
#      trajectory configuration (it translates m/s to a -1 to 1 scale that your␣
↪motors can read)
# The fifth argument is your acceleration gain. Tweak this if you want to get to a␣
↪higher or lower speed quicker
left.configurePIDVA(1.0, 0.0, 0.0, 1 / max_velocity, 0)
```

Inside your control loop, you can add the following code to calculate the desired output of your motors:

```
output = left.calculate(encoder_position);
```

Now, keep in mind this doesn't account for heading of your robot, meaning it won't track a curved path. To adjust for this, you can use your Gyroscope and the desired heading of the robot to create a simple, proportional gain that will turn your tracks. A full example, including the calculations for each side of the drive train is given below.

```
l = left.calculate(encoder_position_left)
r = right.calculate(encoder_position_right)

gyro_heading = ... your gyro code here ...    # Assuming the gyro is giving a value␣
↪in degrees
desired_heading = pf.r2d(left.getHeading())   # Should also be in degrees

```

(continues on next page)

(continued from previous page)

```
angleDifference = pf.boundHalfDegrees(desired_heading - gyro_heading)
turn = 0.8 * (-1.0/80.0) * angleDifference

setLeftMotors(l + turn)
setRightMotors(r - turn)
```

The `boundHalfDegrees()` function simply binds a degrees angle to `-180..180`, making sure we don't end up with an absurdly large turn value.

Note that for the desired heading of the robot, we're only using the left follower as a comparison. This is because both the left and right sides of a tank drive are parallel, and therefore always face in the same direction.

Swerve Drive

Swerve Drive following is very similar to Tank Drive, except each wheel can have a different trajectory and heading. To make things simple, I will be showing how to do it for a single wheel. For all 4 wheels, just do the exact same thing 4 times.

Create an EncoderFollower object for your wheel:

```
from pathfinder.followers import EncoderFollower

flFollower = EncoderFollower(modifier.getFrontLeftTrajectory())   # Front Left wheel
```

When you're ready to start following:

Setup your encoder details:

```
# Encoder Position is the current, cumulative position of your encoder. If
# you're using an SRX, this will be the 'getEncPosition' function.
# 1000 is the amount of encoder ticks per full revolution
# Wheel Diameter is the diameter of your wheel in meters
flFollower.configureEncoder(fl_encoder_position, 1000, wheel_diameter)
```

Set your PID/VA variables:

```
# The first argument is the proportional gain. Usually this will be quite high
# The second argument is the integral gain. This is unused for motion profiling
# The third argument is the derivative gain. Tweak this if you are unhappy with the␣
↪tracking of the trajectory
# The fourth argument is the velocity ratio. This is 1 over the maximum velocity you␣
↪provided in the
#      trajectory configuration (it translates m/s to a -1 to 1 scale that your␣
↪motors can read)
# The fifth argument is your acceleration gain. Tweak this if you want to get to a␣
↪higher or lower speed quicker
flFollower.configurePIDVA(1.0, 0.0, 0.0, 1 / max_velocity, 0)
```

Inside your control loop, you can add the following code to calculate the desired output of your motor:

```
output = flFollower.calculate(fl_encoder_position)
```

The above *`EncoderFollower.calculate`* call won't account for the heading of your wheel. If you run this as is, you will be permanently going in a straight line. To fix this, we need to know the heading of your swerve wheel. For most teams, this will be done with an encoder. Some example code for dealing with heading is given below:

```
output = flFollower.calculate(fl_encoder_position)
desiredHeading = pf.boundHalfDegrees(pf.r2d(flFollower.getHeading()))    # Bound to -
↪180..180 degrees

frontLeftWheel.setDirection(desiredHeading)
frontLeftWheel.setSpeed(output)
```

The `setDirection` implementation is up to you. Usually, for a swerve drive, this will be some kind of PID control loop.

1.1.6 Example code

The RobotPy examples repository has a pathfinder example program in it, which also contains a working physics module so you can experiment with pathfinder using the pyfrc simulator.

1.2 Pathfinder API

class `pathfinder.`**`Segment`**

- **property acceleration**
- **property dt**
- **property heading**
- **property jerk**
- **property position**
- **property velocity**
- **property x**
- **property y**

class `pathfinder.`**`Waypoint`**

- **property angle**
- **property x**
- **property y**

class `pathfinder.`**`TrajectoryInfo`**

- **property dt**
- **property filter1**
- **property filter2**
- **property impulse**
- **property length**
- **property u**
- **property v**

`pathfinder.`**`generate`**`()`

pathfinder_generate(path: List[pathfinder._pathfinder.Waypoint], fit: capsule, sample_count: int, dt: float, max_velocity: float, max_acceleration: float, max_jerk: float) -> Tuple[pathfinder._pathfinder.TrajectoryInfo, List[pathfinder._pathfinder.Segment]]

Generate a motion profile trajectory using the given waypoints and configuration.

Parameters

- **`path`** – A list of waypoints (setpoints) for the trajectory path to intersect
- **`fit`** – A fit function; use FIT_HERMITE_CUBIC or FIT_HERMITE_QUINTIC
- **`sample_count`** –
- **`dt`** –
- **`max_velocity`** –
- **`max_acceleration`** –
- **`max_jerk`** –

Returns A tuple of TrajectoryInfo, and a generated trajectory (a list of segments)

`pathfinder.`**`d2r`**`()`

Convert angle x from degrees to radians.

`pathfinder.`**`r2d`**`()`

Convert angle x from radians to degrees.

`pathfinder.`**`boundHalfDegrees`**`(`*degrees*`)`

Bound an angle (in degrees) to -180 to 180 degrees.

1.2.1 Followers

`class` `pathfinder.followers.`**`DistanceFollower`**`(`*trajectory*`)`

The DistanceFollower is an object designed to follow a trajectory based on distance covered input. This class can be used for Tank or Swerve drive implementations.

`calculate``(`*distance_covered*`)`

Calculate the desired output for the motors, based on the distance the robot has covered. This does not account for heading of the robot. To account for heading, add some extra terms in your control loop for realignment based on gyroscope input and the desired heading given by this object.

Parameters **`distance_covered`** (`float`) – The distance covered in meters

Return type `float`

Returns The desired output for your motor controller

`configurePIDVA``(`*kp*, *ki*, *kd*, *kv*, *ka*`)`

Configure the PID/VA Variables for the Follower

Parameters

- **`kp`** (`float`) – The proportional term. This is usually quite high (0.8 - 1.0 are common values)
- **`ki`** (`float`) – The integral term. Currently unused.
- **`kd`** (`float`) – The derivative term. Adjust this if you are unhappy with the tracking of the follower. 0.0 is the default

- **kv** (`float`) – The velocity ratio. This should be 1 over your maximum velocity @ 100% throttle. This converts m/s given by the algorithm to a scale of -1..1 to be used by your motor controllers
- **ka** (`float`) – The acceleration term. Adjust this if you want to reach higher or lower speeds faster. 0.0 is the default

Return type `None`

getHeading()

Return type `float`

Returns the desired heading of the current point in the trajectory

getSegment()

Return type `Segment`

Returns the current segment being operated on

isFinished()

Return type `bool`

Returns whether we have finished tracking this trajectory or not.

reset()

Reset the follower to start again. Encoders must be reconfigured.

Return type `None`

setTrajectory(*trajectory*)

Set a new trajectory to follow, and reset the cumulative errors and segment counts

Return type `None`

class pathfinder.followers.**EncoderFollower**(*trajectory*)

The EncoderFollower is an object designed to follow a trajectory based on encoder input. This class can be used for Tank or Swerve drive implementations.

calculate(*encoder_tick*)

Calculate the desired output for the motors, based on the amount of ticks the encoder has gone through. This does not account for heading of the robot. To account for heading, add some extra terms in your control loop for realignment based on gyroscope input and the desired heading given by this object.

Parameters **encoder_tick** (`int`) – The amount of ticks the encoder has currently measured.

Return type `float`

Returns The desired output for your motor controller

configureEncoder(*initial_position*, *ticks_per_revolution*, *wheel_diameter*)

Configure the Encoders being used in the follower.

Parameters

- **initial_position** (`int`) – The initial 'offset' of your encoder. This should be set to the encoder value just before you start to track
- **ticks_per_revolution** (`int`) – How many ticks per revolution the encoder has
- **wheel_diameter** (`float`) – The diameter of your wheels (or pulleys for track systems) in meters

Return type `None`

configurePIDVA(*kp*, *ki*, *kd*, *kv*, *ka*)
Configure the PID/VA Variables for the Follower

Parameters

- **kp** (`float`) – The proportional term. This is usually quite high (0.8 - 1.0 are common values)
- **ki** (`float`) – The integral term. Currently unused.
- **kd** (`float`) – The derivative term. Adjust this if you are unhappy with the tracking of the follower. 0.0 is the default
- **kv** (`float`) – The velocity ratio. This should be 1 over your maximum velocity @ 100% throttle. This converts m/s given by the algorithm to a scale of -1..1 to be used by your motor controllers
- **ka** (`float`) – The acceleration term. Adjust this if you want to reach higher or lower speeds faster. 0.0 is the default

Return type `None`

getHeading()

Return type `float`

Returns the desired heading of the current point in the trajectory

getSegment()

Return type `Segment`

Returns the current segment being operated on

isFinished()

Return type `bool`

Returns whether we have finished tracking this trajectory or not.

reset()
Reset the follower to start again. Encoders must be reconfigured.

Return type `None`

setTrajectory(*trajectory*)
Set a new trajectory to follow, and reset the cumulative errors and segment counts

Return type `None`

1.2.2 Modifiers

class `pathfinder.modifiers.`**SwerveModifier**(*source*)
The Swerve Modifier will take in a Source Trajectory and spit out 4 trajectories, 1 for each wheel on the drive. This is commonly used in robotics for robots with 4 individual wheels in a 'swerve' configuration, where each wheel can rotate to a specified heading while still being powered.

The Source Trajectory is measured from the centre of the drive base. The modification will not modify the central trajectory

getBackLeftTrajectory()
Get the trajectory for the back-left wheel of the drive base

Return type `List[Segment]`

getBackRightTrajectory()
Get the trajectory for the back-right wheel of the drive base

Return type `List[Segment]`

getFrontLeftTrajectory()
Get the trajectory for the front-left wheel of the drive base

Return type `List[Segment]`

getFrontRightTrajectory()
Get the trajectory for the front-right wheel of the drive base

Return type `List[Segment]`

getSourceTrajectory()
Get the initial source trajectory

Return type `List[Segment]`

modify(*wheelbase_width*, *wheelbase_depth*)
Generate the Trajectory Modification

Parameters

- **wheelbase_width** (`float`) – The width (in meters) between the individual left-right sides of the drivebase
- **wheelbase_depth** (`float`) – The width (in meters) between the individual front-back sides of the drivebase

Return type *SwerveModifier*

Returns self

class pathfinder.modifiers.**TankModifier**(*source*)
The Tank Modifier will take in a Source Trajectory and a Wheelbase Width and spit out a Trajectory for each side of the wheelbase. This is commonly used in robotics for robots which have a drive system similar to a 'tank', where individual parallel sides are driven independently

The Source Trajectory is measured from the centre of the drive base. The modification will not modify the central trajectory

getLeftTrajectory()
Get the trajectory for the left side of the drive base

Return type `List[Segment]`

getRightTrajectory()
Get the trajectory for the right side of the drive base

Return type `List[Segment]`

getSourceTrajectory()
Get the initial source trajectory

Return type `List[Segment]`

modify(*wheelbase_width*)
Generate the Trajectory Modification

Parameters **wheelbase_width** (`float`) – The width (in meters) between the individual sides of the drivebase

Return type *TankModifier*

Returns self

1.2.3 Serialization

For serializing/deserializing in python programs, it's probably easiest to use Python's `pickle` module to directly serialize a trajectory:

```
import pickle

with open('fname', 'wb') as fp:
    pickle.dump(trajectory, fp)

with open('fname', 'rb') as fp:
    trajectory = pickle.load(fp)
```

One advantage to this approach is that you could put multiple trajectories in a data structure such as a dictionary, and serialize them all in a single file. The pathfinder compatibility serialization routines only support a single trajectory per file.

However, for compatibility with other pathfinder implementations, the following functions are made available.

`pathfinder.`**`deserialize`**`()`

pathfinder_deserialize(fname: str) -> List[pathfinder._pathfinder.Segment]

Read a Trajectory from a Binary (non human readable) file

`pathfinder.`**`deserialize_csv`**`(`*fname*`)`

Read a Trajectory from a CSV File

`pathfinder.`**`serialize`**`()`

pathfinder_serialize(fname: str, trajectory: List[pathfinder._pathfinder.Segment]) -> bool

Write the Trajectory to a Binary (non human readable) file

`pathfinder.`**`serialize_csv`**`()`

pathfinder_serialize_csv(fname: str, trajectory: List[pathfinder._pathfinder.Segment]) -> bool

Write the Trajectory to a CSV File

INDEX