pyfrc Documentation

Release 2017.1.5

Dustin Spicuzza

Apr 11, 2017

Robot Programming

pyfrc is a python 3 library designed to make developing python code using WPILib for FIRST Robotics Competition easier.

This library contains a few primary parts:

- A built-in uploader that will upload your robot code to the robot
- Integration with the py.test testing tool to allow you to easily write unit tests for your robot code.
- A robot simulator tool which allows you to run your code in (vaguely) real time and get simple feedback via a tk-based UI

CHAPTER 1

PyFRC API

Tests that come with pyfrc

pyfrc comes with testing functions that can be used to test basic functionality of just about any robot, including running through a simulated practice match. These generic test modules can be applied to `wpilib.IterativeRobot` and `wpilib.SampleRobot` based robots. The primary purpose of these tests is to run through your code and make sure that it doesn't crash. If you actually want to test your code, you need to write your own custom tests to tease out the edge cases.

To use these, add the following to a python file in your tests directory:

```
from pyfrc.tests import *
```

pyfrc.tests.basic.**test_autonomous**(*control*, *fake_time*, *robot*)
Runs autonomous mode by itself

pyfrc.tests.basic.**test_disabled**(*control*, *fake_time*, *robot*)
Runs disabled mode by itself

pyfrc.tests.basic.**test_operator_control**(*control*, *fake_time*, *robot*)
Runs operator control mode by itself

pyfrc.tests.basic.**test_practice**(*control*, *fake_time*, *robot*)
Runs through the entire span of a practice match

Fuzz tests

The purpose of the fuzz 'test' is not exactly to 'do' anything, but rather it mashes the buttons and switches in various completely random ways to try and find any possible control situations and such that would probably never *normally* come up, but.. well, given a bit of bad luck, could totally happen.

Keep in mind that the results will totally different every time you run this, so if you find an error, fix it – but don't expect that you'll be able to duplicate it with this test. Instead, you should design a specific test that can trigger the bug, to ensure that you actually fixed it.

pyfrc.tests.fuzz_test.**test_fuzz**(*hal_data*, *control*, *fake_time*, *robot*)
Runs through a whole game randomly setting components

Docstring tests

pyfrc.tests.docstring_test.**ignore_object**(*o*, *robot_path*)
Returns true if the object can be ignored

pyfrc.tests.docstring_test.**test_docstrings**(*robot*, *robot_path*)
The purpose of this test is to ensure that all of your robot code has docstrings. Properly using docstrings will make your code more maintainable and look more professional.

Custom Test Support

Contents

pytest fixtures

class pyfrc.test_support.pytest_plugin.**PyFrcPlugin**(*robot_class*, *robot_file*, *robot_path*)
Pytest plugin. Each documented member function name can be an argument to your test functions, and the data that these functions return will be passed to your test function.

control()
A fixture that provides control over the robot

Return type *TestController*

fake_time()
A fixture that gives you control over the time your robot is using

Return type *FakeTime*

hal_data()
Provides access to a dict with all the device data about the robot

See also:

For a listing of what the dict contains and some documentation, see https://github.com/robotpy/robotpy-wpilib/blob/master/hal-sim/hal_impl/data.py

robot()
Your robot instance

robot_file()
The absolute filename your robot code is started from

robot_path()
The absolute directory that your robot code is located at

wpilib()
The wpilib module. Provided for backwards compatibility

Controlling the robot's state

class pyfrc.test_support.controller.**TestController**(*fake_time_inst*)
This object is used to control the robot during unit tests. You do not need to create an instance of this, instead use the controller fixture.

get_mode()
Returns the current mode that the robot is in

Returns 'autonomous', 'teleop', 'test', or 'disabled'

run_test(*controller=None*)
Call this to execute the robot code. Cannot be called more than once in a single test.

If the controller argument is a class, it will be constructed and the instance will be returned.

Parameters **controller** – This can either be a function that takes a single argument, or a class that has an 'on_step' function. If it is a class, an instance will be created. Either the function or the on_step function will be called with a single parameter, which is the the current robot time. If None, an error will be signaled unless *set_practice_match()* has been called.

set_autonomous(*enabled=True*)
Puts the robot in autonomous mode

set_operator_control(*enabled=True*)
Puts the robot in operator control mode

set_practice_match()
Call this function to enable a practice match. Must only be called before *run_test()* is called.

set_test_mode(*enabled=True*)
Puts the robot in test mode (the robot mode, not related to unit testing)

class pyfrc.test_support.fake_time.**FakeTime**
Keeps track of time for robot code being tested, and makes sure the DriverStation is notified that new packets are coming in.

Note: Do not create this object, your testing code can use this object to control time via the fake_time fixture

---

get()

Returns The current time for the robot

increment_new_packet()
Increment time enough to where the new DriverStation packet comes in

increment_time_by(*time*)
Increments time by some number of seconds

Parameters **time** (*float*) – Number of seconds to increment time by

initialize()
Initializes fake time

reset()
Resets the fake time to zero, and sets the time limit to default

set_time_limit(*time_limit*)
Sets the amount of time that a robot will run. When time is incremented past this time, a TestRanTooLong is thrown.

The default time limit is 500 seconds

Parameters **time_limit** (*float*) – Number of seconds

exception pyfrc.test_support.fake_time.**TestEnded**
This is thrown when the controller has been signaled to end the test

This exception inherits from BaseException, so if you want to catch it you must explicitly specify it, as a blanket except statement will not catch this exception.'

Generally, only internal pyfrc code needs to catch this

exception pyfrc.test_support.fake_time.**TestFroze**
This happens when an infinite loop of some kind in one of your non-robot threads is detected.

exception pyfrc.test_support.fake_time.**TestRanTooLong**
This is thrown when the time limit has expired

This exception inherits from BaseException, so if you want to catch it you must explicitly specify it, as a blanket except statement will not catch this exception.'

Generally, only internal pyfrc code needs to catch this

Robot 'physics model'

pyfrc supports simplistic custom physics model implementations for simulation and testing support. It can be as simple or complex as you want to make it. We will continue to add helper functions (such as the *pyfrc.physics.drivetrains* module) to make this a lot easier to do. General purpose physics implementations are welcome also!

The idea is you provide a *PhysicsEngine* object that overrides specific pieces of WPILib, and modifies motors/sensors accordingly depending on the state of the simulation. An example of this would be measuring a motor moving for a set period of time, and then changing a limit switch to turn on after that period of time. This can help you do more complex simulations of your robot code without too much extra effort.

Note: One limitation to be aware of is that the physics implementation currently assumes that you are only calling `wpilib.delay()` once per main loop iteration. If you do it more than that, you may get some rather funky results.

By default, pyfrc doesn't modify any of your inputs/outputs without being told to do so by your code or the simulation GUI.

See the physics sample for more details.

Enabling physics support

You must create a python module called `physics.py` next to your `robot.py`. A physics module must have a class called *PhysicsEngine* which must have a function called `update_sim`. When initialized, it will be passed an

instance of this object.

You must also create a 'sim' directory, and place a `config.json` file there, with the following JSON information:

```
{
  "pyfrc": {
    "robot": {
      "w": 2,
      "h": 3,
      "starting_x": 2,
      "starting_y": 20,
      "starting_angle": 0
    },
    "field": {
      "w": 25,
      "h": 27,
      "px_per_ft": 10
    }
  }
}
```

class `pyfrc.physics.core.`**PhysicsEngine**(*physics_controller*)

Your physics module must contain a class called `PhysicsEngine`, and it must implement the same functions as this class.

Alternatively, you can inherit from this object. However, that is not required.

The constructor must take the following arguments:

Parameters **physics_controller** (*PhysicsInterface*) – The physics controller interface

initialize(*hal_data*)

Called with the hal_data dictionary before the robot has started running. Some values may be overwritten when devices are initialized... it's not consistent yet, sorry.

update_sim(*hal_data*, *now*, *tm_diff*)

Called when the simulation parameters for the program need to be updated. This is mostly when `wpilib.delay()` is called.

Parameters

- **hal_data** – A giant dictionary that has all data about the robot. See `hal-sim/hal_impl/data.py` in robotpy-wpilib's repository for more information on the contents of this dictionary.
- **now** (*float*) – The current time
- **tm_diff** (*float*) – The amount of time that has passed since the last time that this function was called

class `pyfrc.physics.core.`**PhysicsInterface**(*robot_path*, *fake_time*, *config_obj*)

An instance of this is passed to the constructor of your *PhysicsEngine* object. This instance is used to communicate information to the simulation, such as moving the robot on the field displayed to the user.

add_analog_gyro_channel(*ch*)

If you want to enable a wpilib `AnalogGyro` object to be updated when the robot rotates, add the channel number via this function.

Parameters **ch** (*int*) – Analog input channel that the gyro is on

add_device_gyro_channel(*angle_key*)

Parameters **angle_key** – The name of the angle key in `hal_data['robot']`

add_gyro_channel(*ch*)
If you want to enable a wpilib `AnalogGyro` object to be updated when the robot rotates, add the channel number via this function.

Parameters **ch** (*int*) – Analog input channel that the gyro is on

drive(*speed*, *rotation_speed*, *tm_diff*)
Call this from your *PhysicsEngine.update_sim()* function. Will update the robot's position on the simulation field.

You can either calculate the speed & rotation manually, or you can use the predefined functions in *pyfrc.physics.drivetrains*.

The outputs of the *drivetrains.** functions should be passed to this function.

Note: The simulator currently only allows 2D motion

Parameters

- **speed** – Speed of robot in ft/s
- **rotation_speed** – Clockwise rotational speed in radians/s
- **tm_diff** – Amount of time speed was traveled (this is the same value that was passed to update_sim)

get_position()

Returns Robot's current position on the field as *(x,y,angle)*. *x* and *y* are specified in feet, *angle* is in radians

vector_drive(*vx*, *vy*, *vw*, *tm_diff*)
Call this from your *PhysicsEngine.update_sim()* function. Will update the robot's position on the simulation field.

This moves the robot using a vector relative to the robot instead of by speed/rotation speed.

Parameters

- **vx** – Speed in x direction relative to robot in ft/s
- **vy** – Speed in y direction relative to robot in ft/s
- **vw** – Clockwise rotational speed in rad/s
- **tm_diff** – Amount of time speed was traveled

Camera 'simulator'

The 'vision simulator' provides objects that assist in modeling inputs from a camera processing system.

class pyfrc.physics.visionsim.**VisionSim**(*targets*, *camera_fov*, *view_dst_start*, *view_dst_end*, *data_frequency=15*, *data_lag=0.05*, *physics_controller=None*)
This helper object is designed to help you simulate input from a vision system. The algorithm is a very simple approximation and has some weaknesses, but it should be a good start and general enough to work for many different usages.

There are a few assumptions that this makes:

•Your camera code sends new data at a constant frequency

- The data from the camera lags behind at a fixed latency
- If the camera is too close, the target cannot be seen
- If the camera is too far, the target cannot be seen
- You can only 'see' the target when the 'front' of the robot is around particular angles to the target
- The camera is in the center of your robot (this simplifies some things, maybe fix this in the future...)

To use this, create an instance in your physics simulator:

```
targets = [
    VisionSim.Target(...)
]
```

Then call the *compute()* method from your `update_sim` method whenever your camera processing is enabled:

```
# in physics engine update_sim()
x, y, angle = self.physics_controller.get_position()

if self.camera_enabled:
    data = self.vision_sim.compute(now, x, y, angle)
    if data is not None:
        self.nt.putNumberArray('/camera/target', data[0])
else:
    self.vision_sim.dont_compute()
```

Note: There is a working example in the examples repository you can use to try this functionality out

There are a lot of constructor parameters:

Parameters

- **targets** – List of target positions (x, y) on field in feet
- **view_angle_start** – Center angle that the robot can 'see' the target from (in degrees)
- **camera_fov** – Field of view of camera (in degrees)
- **view_dst_start** – If the robot is closer than this, the target cannot be seen
- **view_dst_end** – If the robot is farther than this, the target cannot be seen
- **data_frequency** – How often the camera transmits new coordinates
- **data_lag** – How long it takes for the camera data to be processed and make it to the robot
- **physics_controller** – If set, will draw target information in UI

Target
alias of *VisionSimTarget*

compute(*now*, *x*, *y*, *angle*)
Call this when vision processing should be enabled

Parameters

- **now** – The value passed to `update_sim`
- **x** – Returned from physics_controller.get_position
- **y** – Returned from physics_controller.get_position

- **angle** – Returned from physics_controller.get_position

Returns

None or list of tuples of (found=0 or 1, capture_time, offset_degrees, distance). The tuples are ordered by absolute offset from the target. If a list is returned, it is guaranteed to have at least one element in it.

Note: If your vision targeting doesn't have the ability to focus on multiple targets, then you should ignore the other elements.

dont_compute()
Call this when vision processing should be disabled

get_immediate_distance()
Use this data to feed to a sensor that is mostly instantaneous (such as an ultrasonic sensor).

Note: You must call *compute()* first.

---

class pyfrc.physics.visionsim.**VisionSimTarget**(*x*, *y*, *view_angle_start*, *view_angle_end*)
Target object that you pass the to the constructor of *VisionSim*

Parameters

- **x** – Target x position
- **y** – Target y position
- **view_angle_start** –
- **view_angle_end** – clockwise from start angle

View angle is defined in degrees from 0 to 360, with 0 = east, increasing clockwise. So, if the robot could only see the target from the south east side, you would use a view angle of start=0, end=90.

Drivetrain support

Based on input from various drive motors, these helper functions simulate moving the robot in various ways. Many thanks to Ether for assistance with the motion equations.

When specifying the robot speed to the below functions, the following may help you determine the approximate speed of your robot:

- Slow: 4ft/s
- Typical: 5 to 7ft/s
- Fast: 8 to 12ft/s

Obviously, to get the best simulation results, you should try to estimate the speed of your robot accurately.

pyfrc.physics.drivetrains.**four_motor_drivetrain**(*lr_motor*, *rr_motor*, *lf_motor*, *rf_motor*, *x_wheelbase=2*, *speed=5*)
Four motors, each side chained together. The motion equations are as follows:

```
FWD = (L+R)/2
RCW = (L-R)/W
```

•L is forward speed of the left wheel(s), all in sync

•R is forward speed of the right wheel(s), all in sync

•W is wheelbase in feet

If you called "SetInvertedMotor" on any of your motors in RobotDrive, then you will need to multiply that motor's value by -1.

Note: WPILib RobotDrive assumes that to make the robot go forward, the left motors must be set to -1, and the right to +1

Parameters

- **lr_motor** – Left rear motor value (-1 to 1); -1 is forward
- **rr_motor** – Right rear motor value (-1 to 1); 1 is forward
- **lf_motor** – Left front motor value (-1 to 1); -1 is forward
- **rf_motor** – Right front motor value (-1 to 1); 1 is forward
- **x_wheelbase** – The distance in feet between right and left wheels.
- **speed** – Speed of robot in feet per second (see above)

Returns speed of robot (ft/s), clockwise rotation of robot (radians/s)

pyfrc.physics.drivetrains.**mecanum_drivetrain**(*lr_motor*, *rr_motor*, *lf_motor*, *rf_motor*, *x_wheelbase=2*, *y_wheelbase=3*, *speed=5*)

Four motors, each with a mechanum wheel attached to it.

If you called "SetInvertedMotor" on any of your motors in RobotDrive, then you will need to multiply that motor's value by -1.

Note: WPILib RobotDrive assumes that to make the robot go forward, all motors are set to +1

Parameters

- **lr_motor** – Left rear motor value (-1 to 1); 1 is forward
- **rr_motor** – Right rear motor value (-1 to 1); 1 is forward
- **lf_motor** – Left front motor value (-1 to 1); 1 is forward
- **rf_motor** – Right front motor value (-1 to 1); 1 is forward
- **x_wheelbase** – The distance in feet between right and left wheels.
- **y_wheelbase** – The distance in feet between forward and rear wheels.
- **speed** – Speed of robot in feet per second (see above)

Returns Speed of robot in x (ft/s), Speed of robot in y (ft/s), clockwise rotation of robot (radians/s)

pyfrc.physics.drivetrains.**two_motor_drivetrain**(*l_motor*, *r_motor*, *x_wheelbase=2*, *speed=5*)

Two center-mounted motors with a simple drivetrain. The motion equations are as follows:

```
FWD = (L+R)/2
RCW = (L-R)/W
```

•L is forward speed of the left wheel(s), all in sync

•R is forward speed of the right wheel(s), all in sync

•W is wheelbase in feet

If you called "SetInvertedMotor" on any of your motors in RobotDrive, then you will need to multiply that motor's value by -1.

Note: WPILib RobotDrive assumes that to make the robot go forward, the left motor must be set to -1, and the right to +1

Parameters

- **l_motor** – Left motor value (-1 to 1); -1 is forward
- **r_motor** – Right motor value (-1 to 1); 1 is forward
- **x_wheelbase** – The distance in feet between right and left wheels.
- **speed** – Speed of robot in feet per second (see above)

Returns speed of robot (ft/s), clockwise rotation of robot (radians/s)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

S

T

U

V

W