pyfrc Documentation

Release 2019.1.0.post0.dev7

Dustin Spicuzza

Oct 21, 2019

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

ONE

PYFRC API

1.1 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*, *gamedata*)
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

1.1.1 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

1.1.2 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.

1.2 Custom Test Support

Contents

1.2.1 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

exception pyfrc.test_support.pytest_plugin.**ThreadStillRunningError**

1.2.2 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.

To set a game specific message for autonomous mode, just set the 'game_specific_message' property of this object, and it will be setup correctly upon transition into autonomous mode.

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

1.3 Simulation Physics

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 interacts with the simulated HAL, 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.

1.3.1 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
    }
  }
}
```

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

exception pyfrc.physics.core.**PhysicsInitException**

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*)

This function is no longer needed

add_device_gyro_channel(*angle_key*)

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

add_gyro_channel(*ch*)

This function is no longer needed

distance_drive(*x*, *y*, *angle*)

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

This moves the robot some relative distance and angle from its current position.

Parameters

- **x** – Feet to move the robot in the x direction
- **y** – Feet to move the robot in the y direction
- **angle** – Radians to turn the robot

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_offset(*x*, *y*)

Computes how far away and at what angle a coordinate is located.

Distance is returned in feet, angle is returned in degrees

Returns distance,angle offset of the given x,y coordinate

New in version 2018.1.7.

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 velocity 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

1.3.2 Drivetrain support

Warning: These drivetrain models are not particularly realistic, and if you are using a tank drive style drivetrain you should use the *TankModel* instead.

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.

Here's an example usage of the drivetrains:

```
from pyfrc.physics import drivetrains

class PhysicsEngine:

    def __init__(self, physics_controller):
        self.physics_controller = physics_controller
        self.drivetrain = drivetrains.TwoMotorDrivetrain(deadzone=drivetrains.linear_
↪deadzone(0.2))

    def update_sim(self, hal_data, now, tm_diff):
        # TODO: get motor values from hal_data
        speed, rotation = self.drivetrain.get_vector(l_motor, r_motor)
        self.physics_controller.drive(speed, rotation, tm_diff)

        # optional: compute encoder
        # l_encoder = self.drivetrain.l_speed * tm_diff
```

class pyfrc.physics.drivetrains.**FourMotorDrivetrain**(*x_wheelbase=2*, *speed=5*, *deadzone=None*)

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

New in version 2018.2.0.

Parameters

- **x_wheelbase** (float) – The distance in feet between right and left wheels.
- **speed** (float) – Speed of robot in feet per second (see above)
- **deadzone** (Optional[Callable[[float], float]]) – A function that adjusts the output of the motor (see *linear_deadzone()*)

get_vector(*lr_motor*, *rr_motor*, *lf_motor*, *rf_motor*)

Parameters

- **lr_motor** (float) – Left rear motor value (-1 to 1); -1 is forward
- **rr_motor** (float) – Right rear motor value (-1 to 1); 1 is forward
- **lf_motor** (float) – Left front motor value (-1 to 1); -1 is forward
- **rf_motor** (float) – Right front motor value (-1 to 1); 1 is forward

Return type Tuple[float, float]

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

l_speed = 0

Use this to compute encoder data after get_vector is called

class pyfrc.physics.drivetrains.**MecanumDrivetrain**(*x_wheelbase=2*, *y_wheelbase=3*, *speed=5*, *deadzone=None*)

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

New in version 2018.2.0.

Parameters

- **x_wheelbase** (float) – The distance in feet between right and left wheels.
- **y_wheelbase** (float) – The distance in feet between forward and rear wheels.
- **speed** (float) – Speed of robot in feet per second (see above)
- **deadzone** (Optional[Callable[[float], float]]) – A function that adjusts the output of the motor (see *linear_deadzone()*)

get_vector(*lr_motor*, *rr_motor*, *lf_motor*, *rf_motor*)

Given motor values, retrieves the vector of (distance, speed) for your robot

Parameters

- **lr_motor** (float) – Left rear motor value (-1 to 1); 1 is forward
- **rr_motor** (float) – Right rear motor value (-1 to 1); 1 is forward
- **lf_motor** (float) – Left front motor value (-1 to 1); 1 is forward
- **rf_motor** (float) – Right front motor value (-1 to 1); 1 is forward

Return type Tuple[float, float, float]

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

lr_speed = 0
Use this to compute encoder data after get_vector is called

class pyfrc.physics.drivetrains.**TwoMotorDrivetrain**(*x_wheelbase=2*, *speed=5*, *deadzone=None*)
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

New in version 2018.2.0.

Parameters

- **x_wheelbase** (float) – The distance in feet between right and left wheels.
- **speed** (float) – Speed of robot in feet per second (see above)
- **deadzone** (Optional[Callable[[float], float]]) – A function that adjusts the output of the motor (see *linear_deadzone()*)

get_vector(*l_motor*, *r_motor*)
Given motor values, retrieves the vector of (distance, speed) for your robot

Parameters

- **l_motor** (float) – Left motor value (-1 to 1); -1 is forward
- **r_motor** (float) – Right motor value (-1 to 1); 1 is forward

Return type Tuple[float, float]

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

pyfrc.physics.drivetrains.**four_motor_drivetrain**(*lr_motor*, *rr_motor*, *lf_motor*, *rf_motor*, *x_wheelbase=2*, *speed=5*, *deadzone=None*)
Deprecated since version 2018.2.0: Use *FourMotorDrivetrain* instead

pyfrc.physics.drivetrains.**four_motor_swerve_drivetrain**(*lr_motor*, *rr_motor*, *lf_motor*, *rf_motor*, *lr_angle*, *rr_angle*, *lf_angle*, *rf_angle*, *x_wheelbase=2*, *y_wheelbase=2*, *speed=5*, *deadzone=None*)
Four motors that can be rotated in any direction

If any motors are inverted, then you will need to multiply that motor's value by -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
- **lr_angle** – Left rear motor angle in degrees (0 to 360 measured clockwise from forward position)
- **rr_angle** – Right rear motor angle in degrees (0 to 360 measured clockwise from forward position)
- **lf_angle** – Left front motor angle in degrees (0 to 360 measured clockwise from forward position)
- **rf_angle** – Right front motor angle in degrees (0 to 360 measured clockwise from forward position)
- **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)
- **deadzone** – A function that adjusts the output of the motor (see *linear_deadzone()*)

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

pyfrc.physics.drivetrains.**linear_deadzone**(*deadzone*)

Real motors won't actually move unless you give them some minimum amount of input. This computes an output speed for a motor and causes it to 'not move' if the input isn't high enough. Additionally, the output is adjusted linearly to compensate.

Example: For a deadzone of 0.2:

- Input of 0.0 will result in 0.0
- Input of 0.2 will result in 0.0
- Input of 0.3 will result in ~0.12
- Input of 1.0 will result in 1.0

This returns a function that computes the deadzone. You should pass the returned function to one of the drivetrain simulation functions as the `deadzone` parameter.

Parameters

- **motor_input** – The motor input (between -1 and 1)
- **deadzone** (`float`) – Minimum input required for the motor to move (between 0 and 1)

Return type `Callable[[float], float]`

pyfrc.physics.drivetrains.**mecanum_drivetrain**(*lr_motor*, *rr_motor*, *lf_motor*, *rf_motor*, *x_wheelbase=2*, *y_wheelbase=3*, *speed=5*, *deadzone=None*)

Deprecated since version 2018.2.0: Use *MecanumDrivetrain* instead

pyfrc.physics.drivetrains.**two_motor_drivetrain**(*l_motor*, *r_motor*, *x_wheelbase=2*, *speed=5*, *deadzone=None*)

Deprecated since version 2018.2.0: Use *TwoMotorDrivetrain* instead

1.3.3 Motor configurations

pyfrc.physics.motor_cfgs.**MOTOR_CFG_775PRO = MotorModelConfig(name='775Pro', nominalVoltage=**
Motor configuration for 775 Pro

pyfrc.physics.motor_cfgs.**MOTOR_CFG_775_125 = MotorModelConfig(name='RS775-125', nominalVolt**
Motor configuration for Andymark RS 775-125

pyfrc.physics.motor_cfgs.**MOTOR_CFG_AM_9015 = MotorModelConfig(name='AM-9015', nominalVoltag**
Motor configuration for Andymark 9015

pyfrc.physics.motor_cfgs.**MOTOR_CFG_BAG = MotorModelConfig(name='Bag', nominalVoltage=<Quant**
Motor configuration for Bag Motor

pyfrc.physics.motor_cfgs.**MOTOR_CFG_BB_RS550 = MotorModelConfig(name='RS550', nominalVoltage**
Motor configuration for Banebots RS 550

pyfrc.physics.motor_cfgs.**MOTOR_CFG_BB_RS775 = MotorModelConfig(name='RS775', nominalVoltage**
Motor configuration for Banebots RS 775

pyfrc.physics.motor_cfgs.**MOTOR_CFG_CIM = MotorModelConfig(name='CIM', nominalVoltage=<Quant**
Motor configuration for CIM

pyfrc.physics.motor_cfgs.**MOTOR_CFG_MINI_CIM = MotorModelConfig(name='MiniCIM', nominalVolta**
Motor configuration for Mini CIM

class pyfrc.physics.motor_cfgs.**MotorModelConfig**
Configuration parameters useful for simulating a motor. Typically these parameters can be obtained from the manufacturer via a data sheet or other specification.

RobotPy contains MotorModelConfig objects for many motors that are commonly used in FRC. If you find that we're missing a motor you care about, please file a bug report and let us know!

Note: The motor configurations that come with pyfrc are defined using the pint units library. See *Unit conversions*

Create new instance of MotorModelConfig(name, nominalVoltage, freeSpeed, freeCurrent, stallTorque, stallCurrent)

property freeCurrent
No-load motor current

property freeSpeed
No-load motor speed (`1 / [time]`)

property name
Descriptive name of motor

property nominalVoltage
Nominal voltage for the motor

property stallCurrent
Stall current

property stallTorque
Stall torque (`[length]**2 * [mass] / [time]**2`)

1.3.4 Motion support

class pyfrc.physics.motion.**LinearMotion**(*name*, *motor_ft_per_sec*, *ticks_per_feet*, *max_position=None*, *min_position=0*)

Helper for simulating motion involving a encoder directly coupled to a motor.

Here's an example that shows a linear motion of 6ft at 2 ft/s with a 360-count-per-ft encoder, coupled to a PWM motor on port 0 and the first encoder object:

```
from pyfrc.physics import motion

class PhysicsEngine:

    def __init__(self, physics_controller):
        self.motion = pyfrc.physics.motion.LinearMotion('Motion', 2, 360, 6)

    def update_sim(self, hal_data, now, tm_diff):

        motor_value = hal_data['pwm'][0]['value']
        hal_data['encoder'][0]['value'] = self.motion.compute(motor_value)
```

New in version 2018.3.0.

Parameters

- **name** (str) – Name of motion, shown in pyfrc simulation UI
- **motor_ft_per_sec** (float) – Motor travel in feet per second (or whatever units you want)
- **ticks_per_feet** (int) – Number of encoder ticks per feet
- **max_position** (Optional[float]) – Maximum position that this motion travels to
- **min_position** (Optional[float]) – Minimum position that this motion travels to

position_ft = 0

Current computed position of motion, in feet

position_ticks = 0

Current computed position of motion (encoder ticks)

1.3.5 Tank drive model support

New in version 2018.4.0.

Note: The equations used in our *TankModel* is derived from Noah Gleason and Eli Barnett's motor characterization whitepaper. It is recommended that users of this model read the paper so they can more fully understand how this works.

In the interest of making progress, this API may receive backwards-incompatible changes before the start of the 2019 FRC season.

---

class pyfrc.physics.tankmodel.**MotorModel**(*motor_config*, *kv*, *ka*, *vintercept*)

Motor model used by the *TankModel*. You should not need to create this object if you're using the *TankModel* class.

Parameters

- **motor_config** (*MotorModelConfig*) – The specification data for your motor
- **kv** (*tm_kv*) – Computed `kv` for your robot
- **ka** (*tm_ka*) – Computed `ka` for your robot
- **vintercept** (*volt*) – The minimum voltage required to generate enough torque to overcome steady-state friction (see the paper for more details)

acceleration = None
Current computed acceleration (in ft/s^2)

compute(*motor_pct*, *tm_diff*)

Parameters

- **motor_pct** (`float`) – Percentage of power for motor in range [1..-1]
- **tm_diff** (`float`) – Time elapsed since this function was last called

Return type `float`

Returns velocity

position = None
Current computed position (in ft)

velocity = None
Current computed velocity (in ft/s)

class pyfrc.physics.tankmodel.**TankModel**(*motor_config*, *robot_mass*, *x_wheelbase*, *robot_width*, *robot_length*, *l_kv*, *l_ka*, *l_vi*, *r_kv*, *r_ka*, *r_vi*, *timestep=<Quantity(5, 'millisecond')>*)

This is a model of a FRC tankdrive-style drivetrain that will provide vaguely realistic motion for the simulator.

This drivetrain model makes a number of assumptions:

- N motors per side
- Constant gearing
- Motors are geared together
- Wheels do not 'slip' on the ground
- Each side of the robot moves in unison

There are two ways to construct this model. You can use the theoretical model via *TankModel.theory()* and provide robot parameters such as gearing, total mass, etc.

Alternatively, if you measure `kv`, `ka`, and `vintercept` as detailed in the paper mentioned above, you can plug those values in directly instead using the *TankModel* constructor instead. For more information about measuring your own values, see the paper and this thread on ChiefDelphi.

Note: You must specify the You can use whatever units you would like to specify the input parameter for your robot, RobotPy will convert them all to the correct units for computation.

Output units for velocity and acceleration are in ft/s and ft/s^2

Example usage for a 90lb robot with 2 CIM motors on each side with 6 inch wheels:

```
from pyfrc.physics import motors, tankmodel
from pyfrc.physics.units import units

class PhysicsEngine:

    def __init__(self, physics_controller):
        self.physics_controller = physics_controller

        self.drivetrain = tankmodel.TankModel.theory(motors.MOTOR_CFG_CIM_IMP,
                                                     robot_mass=90 * units.lbs,
                                                     gearing=10.71, nmotors=2,
                                                     x_wheelbase=2.0*feet,
                                                     wheel_diameter=6*units.inch)

    def update_sim(self, hal_data, now, tm_diff):
        # TODO: get motor values from hal_data
        velocity, rotation = self.drivetrain.get_vector(l_motor, r_motor)
        self.physics_controller.drive(velocity, rotation, tm_diff)

        # optional: compute encoder
        # l_encoder = self.drivetrain.l_position * ENCODER_TICKS_PER_FT
        # r_encoder = self.drivetrain.r_position * ENCODER_TICKS_PER_FT
```

Use the constructor if you have measured `kv`, `ka`, and `Vintercept` for your robot. Use the `theory()` function if you haven't.

`Vintercept` is the minimum voltage required to generate enough torque to overcome steady-state friction (see the paper for more details).

The robot width/length is used to compute the moment of inertia of the robot. Don't forget about bumpers!

Parameters

- **motor_config** (*MotorModelConfig*) – Motor specification
- **robot_mass** (`Quantity`) – Mass of robot
- **x_wheelbase** (`Quantity`) – Wheelbase of the robot
- **robot_width** (`Quantity`) – Width of the robot
- **robot_length** (`Quantity`) – Length of the robot
- **l_kv** (`Quantity`) – Left side `kv`
- **l_ka** (`Quantity`) – Left side `ka`
- **l_vi** (*volt*) – Left side `Vintercept`
- **r_kv** (`Quantity`) – Right side `kv`
- **r_ka** (`Quantity`) – Right side `ka`
- **r_vi** (*volt*) – Right side `Vintercept`
- **timestep** (`Quantity`) – Model computation timestep

get_distance (*l_motor*, *r_motor*, *tm_diff*)

Given motor values and the amount of time elapsed since this was last called, retrieves the x,y,angle that the robot has moved. Pass these values to `PhysicsInterface.distance_drive()`.

To update your encoders, use the `l_position` and `r_position` attributes of this object.

Parameters

- **l_motor** (float) – Left motor value (-1 to 1); -1 is forward
- **r_motor** (float) – Right motor value (-1 to 1); 1 is forward
- **tm_diff** (float) – Elapsed time since last call to this function

Return type Tuple[float, float]

Returns x travel, y travel, angle turned (radians)

Note: If you are using more than 2 motors, it is assumed that all motors on each side are set to the same speed. Only pass in one of the values from each side

---

property inertia
The model computes a moment of inertia for your robot based on the given mass and robot width/length. If you wish to use a different moment of inertia, set this property after constructing the object

Units are `[mass] * [length] ** 2`

property l_position
The linear position of the left side wheel (in feet)

property l_velocity
The velocity of the left side (in ft/s)

property r_position
The linear position of the right side wheel (in feet)

property r_velocity
The velocity of the right side (in ft/s)

classmethod theory(*motor_config*, *robot_mass*, *gearing*, *nmotors=1*, *x_wheelbase=<Quantity(21.0, 'inch')>*, *robot_width=<Quantity(27.5, 'inch')>*, *robot_length=<Quantity(36.5, 'inch')>*, *wheel_diameter=<Quantity(6, 'inch')>*, *vintercept=<Quantity(1.3, 'volt')>*, *timestep=<Quantity(5, 'millisecond')>*)
Use this to create the drivetrain model when you haven't measured `kv` and `ka` for your robot.

Parameters

- **motor_config** (*MotorModelConfig*) – Specifications for your motor
- **robot_mass** (Quantity) – Mass of the robot
- **gearing** (float) – Gear ratio .. so for a 10.74:1 ratio, you would pass 10.74
- **nmotors** (int) – Number of motors per side
- **x_wheelbase** (Quantity) – Wheelbase of the robot
- **robot_width** (Quantity) – Width of the robot
- **robot_length** (Quantity) – Length of the robot
- **wheel_diameter** (Quantity) – Diameter of the wheel
- **vintercept** (*volt*) – The minimum voltage required to generate enough torque to overcome steady-state friction (see the paper for more details)
- **timestep_ms** – Model computation timestep

Computation of `kv` and `ka` are done as follows:

- ω_{free} is the free speed of the motor

- τ_{stall} is the stall torque of the motor
- n is the number of drive motors
- m_{robot} is the mass of the robot
- d_{wheels} is the diameter of the robot's wheels
- $r_{gearing}$ is the total gear reduction between the motors and the wheels
- V_{max} is the nominal max voltage of the motor

$$velocity_{max} = \frac{\omega_{free} \cdot \pi \cdot d_{wheels}}{r_{gearing}}$$
$$acceleration_{max} = \frac{2 \cdot n \cdot \tau_{stall} \cdot r_{gearing}}{d_{wheels} \cdot m_{robot}}$$
$$k_v = \frac{V_{max}}{velocity_{max}}$$
$$k_a = \frac{V_{max}}{acceleration_{max}}$$

1.3.6 Unit conversions

pyfrc uses the pint library in some places for representing physical quantities to allow users to specify the physical parameters of their robot in a natural and non-ambiguous way. For example, to represent 5 feet:

```
from pyfrc.physics.units import units

five_feet = 5 * units.feet
```

Unfortunately, actually using the quantities is a huge performance hit, so we don't use them to perform actual physics computations. Instead, pyfrc uses them to convert to known units, then performs computations using the magnitude of the quantity.

pyfrc defines the following custom units:

- `counts_per_minute` or `cpm`: Counts per minute, which should be used instead of pint's predefined `rpm` (because it is rad/s). Used to represent motor free speed
- `N_m`: Shorthand for N-m or newton-meter. Used for motor torque.
- `tm_ka`: The kA value used in the tankmodel (uses imperial units)
- `tm_kv`: The kV value used in the tankmodel (uses imperial units)

Refer to the pint documentation for more information on how to use pint.

`pyfrc.physics.units.`**`units = <pint.registry.UnitRegistry object>`**
All units that pyfrc uses are defined in this global object

1.3.7 Custom labels

If you write data to the 'custom' key of the `hal_data` dictionary, the data will be shown on the pyfrc simulation UI. The key will be the label for the data, and the value will be rendered as text inside the box:

```
# first time you set the data
hal_data.setdefault('custom', {})['My Label'] = '---'

# All other times
hal_data['custom']['My Label'] = 1
```

New in version 2018.3.0.

1.3.8 Custom drawing

pyfrc.sim.**get_user_renderer**()
This retrieves an object that can be used to draw on the simulated field when running the robot in simulation.

Returns None if no renderers are available, else a *UserRenderer* object

class pyfrc.sim.field.user_renderer.**UserRenderer**

draw_line(*line_pts*, *color='#ff0000'*, *robot_coordinates=False*, *relative_to_first=False*, *arrow=True*, *scale=(1, 1)*, ***kwargs*)

Parameters

- **line_pts** – A list of (x,y) pairs to draw. (x,y) are in field units which are measured in feet
- **color** – The color of the line, expressed as a 6-digit hex color
- **robot_coordinates** – If True, the pts will be adjusted such that the first point starts at the center of the robot and that x and y coordinates are rotated according to the robot's current heading. If a tuple, then the pts are adjusted relative to the robot center AND the x,y in the tuple
- **relative_to_first** – If True, the points will be adjusted such that the first point is considered to be (0,0)
- **arrow** – If True, draw the line with an arrow at the end
- **scale** – Multiply all points by this (x,y) tuple
- **kwargs** – Keyword options to pass to tkinter.create_line

draw_pathfinder_trajectory(*trajectory*, *color='#ff0000'*, *offset=None*, *scale=(1, 1)*, *show_dt=False*, *dt_offset=0.0*, ***kwargs*)
Special helper function for drawing trajectories generated by robotpy-pathfinder

Parameters

- **trajectory** – A list of pathfinder segment objects
- **offset** – If specified, should be x/y tuple to add to the path relative to the robot coordinates
- **scale** – Multiply all points by this (x,y) tuple
- **show_dt** – draw text every N seconds along path, or False
- **dt_offset** – add this to each dt shown
- **kwargs** – Keyword options to pass to tkinter.create_line

draw_text(*text*, *pt*, *color='#000000'*, *fontSize=10*, *robot_coordinates=False*, *scale=(1, 1)*, ***kwargs*)

Parameters

- **text** – Text to render
- **pt** – A tuple of (x,y) in field units (which are measured in feet)
- **color** – The color of the text, expressed as a 6-digit hex color
- **robot_coordinates** – If True, the pt will be adjusted such that the point starts at the center of the robot and that x and y coordinates are rotated according to the robot's current heading. If a tuple, then the pt is adjusted relative to the robot center AND the x,y in the tuple
- **arrow** – If True, draw the line with an arrow at the end
- **scale** – Multiply all points by this (x,y) tuple
- **kwargs** – Keyword options to pass to tkinter.create_text

1.3.9 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.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

T

U

V

W