pyfrc Documentation

Release 2020.1.5

Dustin Spicuzza

Mar 13, 2020

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.
- Simulation and 'robot physics' support

CHAPTER 1

PyFRC API

1.1 Tests that come with pyfrc

Note: pyfrc's testing functionality has not yet been updated to work with RobotPy 2020

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

Test support has not yet been upgraded for 2020.

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.

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.

`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

update_sim(*now*, *tm_diff*)
Called when the simulation parameters for the program should be updated. This is called after robotPeriodic is called.

Parameters

- **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**(*physics_module*)
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.

drive(*speeds*, *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 chassis speeds yourself, or you can use the predefined functions in *pyfrc.physics.drivetrains*.

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

Parameters

- **speeds** (ChassisSpeeds) – Represents current speed/angle of robot travel
- **tm_diff** (float) – Amount of time speed was traveled (this is the same value that was passed to update_sim)

Return type Pose2d

Returns current robot pose

Changed in version 2020.1.0: Input parameter is ChassisSpeeds object

get_pose()

Returns current robot pose

New in version 2020.1.0.

move_robot(*transform*)
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 **transform** (Transform2d) – The distance and angle to move the robot

Return type Pose2d

Returns current robot pose

New in version 2020.1.0.

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:

```
import hal.simulation
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))

        self.l_motor = hal.simulation.PWMSim(1)
        self.r_motor = hal.simulation.PWMSim(2)

    def update_sim(self, now, tm_diff):
        l_motor = self.l_motor.getSpeed()
        r_motor = self.r_motor.getSpeed()

        speeds = self.drivetrain.calculate(l_motor, r_motor)
        self.physics_controller.drive(speeds, tm_diff)

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

Changed in version 2020.1.0: The input speeds and output rotation angles were changed to reflect the current WPILib drivetrain/field objects. Wheelbases and default speeds all require units.

class pyfrc.physics.drivetrains.**FourMotorDrivetrain**(*x_wheelbase=<Quantity(2, 'foot')>*, *speed=<Quantity(5, 'foot_per_second')>*, *deadzone=None*)

Four motors, each side chained together. The motion equations are as follows:

```
FWD = (L+R)/2
RCCW = (R-L)/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

Note: `wpilib.DifferentialDrive` 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** (`Quantity`) – The distance between right and left wheels.
- **speed** (`Quantity`) – Speed of robot (see above)
- **deadzone** (`Optional[Callable[[float], float]]`) – A function that adjusts the output of the motor (see *`linear_deadzone()`*)

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

Given motor values, computes resulting chassis speeds of robot

Parameters

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

Return type `ChassisSpeeds`

Returns ChassisSpeeds that can be passed to 'drive'

New in version 2020.1.0.

wheelSpeeds = None

Wheel speeds you can use for encoder calculations (updated by calculate)

class pyfrc.physics.drivetrains.**MecanumDrivetrain**(*x_wheelbase=<Quantity(2, 'foot')>*, *y_wheelbase=<Quantity(3, 'foot')>*, *speed=<Quantity(5, 'foot_per_second')>*, *deadzone=None*)

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

Note: `wpilib.MecanumDrive` assumes that to make the robot go forward, the left motor outputs are 1, and the right motor outputs are -1

New in version 2018.2.0.

Parameters

- **x_wheelbase** (`Quantity`) – The distance between right and left wheels.
- **y_wheelbase** (`Quantity`) – The distance between forward and rear wheels.
- **speed** (`Quantity`) – Speed of robot (see above)

- **deadzone** (Optional[Callable[[float], float]]) – A function that adjusts the output of the motor (see *linear_deadzone()*)

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

Parameters

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

Return type ChassisSpeeds

Returns ChassisSpeeds that can be passed to 'drive'

New in version 2020.1.0.

wheelSpeeds = None

Use this to compute encoder data after calculate is called

class pyfrc.physics.drivetrains.**TwoMotorDrivetrain**(*x_wheelbase=<Quantity(2, 'foot')>*, *speed=<Quantity(5, 'foot_per_second')>*, *deadzone=None*)

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

```
FWD = (L+R)/2
RCCW = (R-L)/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

Note: wpilib.DifferentialDrive assumes that to make the robot go forward, the left motor output is 1, and the right motor output is -1

New in version 2018.2.0.

Parameters

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

calculate(*l_motor*, *r_motor*)

Given motor values, computes resulting chassis speeds of 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 ChassisSpeeds

Returns ChassisSpeeds that can be passed to 'drive'

New in version 2020.1.0.

wheelSpeeds = None
Wheel speeds you can use for encoder calculations (updated by calculate)

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** (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
- **lr_angle** (float) – Left rear motor angle in degrees (0 to 360 measured clockwise from forward position)
- **rr_angle** (float) – Right rear motor angle in degrees (0 to 360 measured clockwise from forward position)
- **lf_angle** (float) – Left front motor angle in degrees (0 to 360 measured clockwise from forward position)
- **rf_angle** (float) – 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()*)

Return type ChassisSpeeds

Returns ChassisSpeeds that can be passed to 'drive'

Changed in version 2020.1.0: The output rotation angle was changed from CW to CCW to reflect the current WPILib drivetrain/field objects

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]`

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_FALCON_500 = MotorModelConfig(name='Falcon 500', nominal**
Motor configuration for Falcon 500 Brushless Motor

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

pyfrc.physics.motor_cfgs.**MOTOR_CFG_NEO_550 = MotorModelConfig(name='NEO 550', nominalVoltag**
Motor configuration for NEO 550 Brushless Motor

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)

freeCurrent
No-load motor current

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

name
Descriptive name of motor

nominalVoltage
Nominal voltage for the motor

stallCurrent
Stall current

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:

```
import hal.simulation
from pyfrc.physics import motion

class PhysicsEngine:

    def __init__(self, physics_controller):
        self.motion = pyfrc.physics.motion.LinearMotion('Motion', 2, 360, 6)
        self.motor = hal.simulation.PWMSim(0)
        self.encoder = hal.simulation

    def update_sim(self, now, tm_diff):
        motor_value = self.motor.getValue()
        count = self.motion.compute(motor_value, tm_diff)
        self.encoder.setCount(count)
```

New in version 2018.3.0.

Parameters

- **name** (`str`) – Name of motion, shown in 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** (Unit) – Computed kv for your robot
- **ka** (Unit) – Computed ka for your robot
- **vintercept** (Unit) – 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.l_motor = hal.simulation.PWMSim(1)
        self.r_motor = hal.simulation.PWMSim(2)

        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, now, tm_diff):
        l_motor = self.l_motor.getSpeed()
        r_motor = self.r_motor.getSpeed()

        transform = self.drivetrain.calculate(l_motor, r_motor, tm_diff)
        self.physics_controller.move_robot(transform)

        # 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** (Unit) – Left side Vintercept
- **r_kv** (Quantity) – Right side kv
- **r_ka** (Quantity) – Right side ka
- **r_vi** (Unit) – Right side Vintercept
- **timestep** (Quantity) – Model computation timestep

calculate(*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 Transform2d

Returns transform containing x/y/angle offsets of robot travel

---

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

---

New in version 2020.1.0.

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

l_position

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

l_velocity

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

r_position

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

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** (Unit) – 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 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 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index