robotpy-websim Documentation

Release 2019.0.0a5.post0.dev2

RobotPy development team

Jan 16, 2019

Physics

1 Documentation 3

2 Quick Install + Demo 5

3 Authors 7

4 Contents 9
- 4.1 Introduction 9
- 4.2 Enabling Physics Support 9
- 4.3 Matter.js 12
- 4.4 Drivetrain Support 12
- 4.5 Motor Configurations 15
- 4.6 Tank Drive Model Support 17
- 4.7 Unit Conversion 20
- 4.8 Installation 21
- 4.9 Starting a simulation 21
- 4.10 Interface 22
- 4.11 Built in Modules 23
- 4.12 Custom Modules 34

5 Indices and tables 37

This is an web interface for controlling low fidelity FRC robot simulations. As the control/simulation interface is created using HTML/javascript, one of the goals of this project is to make it very simple to create your own custom animations and extensions to help simulate your robot more effectively.

Currently, the only backend for the interface interacts with python based FRC robots using the RobotPy library, and is a replacement for the simulator that comes with pyfrc.

However, the HTML/Javascript portion of the code is not designed to be specific to python, but can be reused with C++ or Java backends using a similar simulated HAL library for those languages. Those have not been implemented yet, but that would be awesome if someone did it.

Note: The simulator and its extension APIs are still very experimental and are expected to vary until the start of the 2016 FRC season.

CHAPTER 1

Documentation

For usage, detailed installation information, and other notes, please see our documentation at http://robotpy-websim.readthedocs.org

CHAPTER 2

Quick Install + Demo

If you have python3 and pip installed, then do:

```
pip3 install --pre robotpy-websim
```

Once this is done, you can run a quick demo by running:

```
cd examples/simple
python3 robot.py websim
```

Your default browser (or Chrome) should be launched and show the control interface. If it does not show automatically, you can browse to http://localhost:8000/

CHAPTER 3

Authors

- Dustin Spicuzza came up with the original concept
- Amory Galili has done much of the actual work and webdesign

CHAPTER 4

Contents

4.1 Introduction

The websim supports 2D rigid body physics for simulation and testing support using Matter.js. It can be as simple or complex as you want to make it. We will continue to add helper functions (such as the `Field`, `Robot`, and `Models` modules) to make this a lot easier to do. By default the websim supports an overhead view of the robot, but you can also create a side profile of your robot as well to better simulate how end effectors interact with game pieces.

The idea is you provide a `MyUserPhysics` 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, the websim doesn't modify any of your inputs/outputs without being told to do so by your code or the simulation GUI.

4.2 Enabling Physics Support

4.2.1 Config.json

To enable physics, you must create a 'sim' directory and place a `config.json` file there, with the following JSON information:

```
{
  "websim": {
    "robot": {
      "w": 2,
      "h": 3,
      "starting_x": 2,
      "starting_y": 20,
      "starting_angle": 0
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

Values without units will default to feet and degrees, but you can optionally add units to your JSON:

```
{
  "websim": {
    "robot": {
      "w": ".66m",
      "h": "1m",
      "starting_x": "24in",
      "starting_y": "20ft",
      "starting_angle": "0rad"
    }
  }
}
```

4.2.2 Physics.js

class UserPhysics(*Matter*, *engine*, *canvas*, *config*)

You must create a `physics.js` file in your 'sim' directory and create a `MyUserPhysics` class that extends `UserPhysics`. The `UserPhysics` class has several methods you can override to customize your simulation:

```
class MyUserPhysics extends UserPhysics {
  ...
  ...
  ...
}
```

Note: Do **NOT** call the `constructor` function yourself. This is automatically called by the websim. If you need to do any custom initialization override the `init` function.

`UserPhysics.`**addDeviceGyroChannel**(*angleKey*)

Call this in the `init` function if you need to add a non-analog gyro device to the hal data.

Arguments

- **angleKey** (*String*) – The name of the angle key in `halData.robot`

`UserPhysics.`**createField**(*fieldConfig*)

Override this function if you want to create a custom field. By default it creates a rectangular field using the dimensions provided in the `config.json` file.

Arguments

- **fieldConfig** (*Object*) – The config JSON from the `config.json` file.

`UserPhysics.`**createRobot**(*robotConfig*)

Override this function if you want to create a custom robot. By default it creates a rectangular robot using the dimensions provided in the `config.json` file.

Arguments

- **robotConfig** (*Object*) – The config JSON from the `config.json` file.

`UserPhysics.`**`createRobotModel`**`(`*robotConfig*`)`

Override this if you want to create a custom robot model. Robot models are used to help simulate the movement of different types of drivetrains. They have a `getVector` method which are used to update the robot's velocity and angular velocity in the `updateSim` method.

Arguments

- **`robotConfig`** (*) – The config JSON from the `config.json` file.

`UserPhysics.`**`disableRobot`**`(`*halData*, *dt*`)`

Override this if you want to customize the physics for disabling the robot. By default the robot is stopped instantaneously.

Arguments

- **`halData`** (*`Object`*) – 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.
- **`dt`** (*`Number`*) – The time that has passed since the last update.

`UserPhysics.`**`init`**`()`

Override this function if you need to perform custom initialization steps in your simulation.

`UserPhysics.`**`reset`**`()`

Resets the simulation. Calls the `createField`, `createRobot`, and `createRobotModel` functions. If you need to add some custom behavior to the reset function, override it and call `super.reset`:

```
reset() {
  // Some custom behavior goes here:
  ...
  ...
  ...

  super.reset();
}
```

`UserPhysics.`**`updateGyros`**`()`

This function is called automatically. It updates the gyros in the hal data based on the angle of the robot in the simulation.

`UserPhysics.`**`updateHalDataIn`**`(`*key*, *value*`)`

Use this to update the hal data. Usually this is called in the `updateSim` function.

Arguments

- **`key`** (*`String`*) – The path to the value you want to update in halData.
- **`value`** (*`String|Number|Boolean|Array`*) – The new value.

`UserPhysics.`**`updateSim`**`(`*halData*, *dt*`)`

Called when the simulation parameters for the program need to be updated. This is called approximately 60 times per second. Override this to update the velocity and angular velocity of the robot, as well as any other custom objects created in the Matter.js world that the robot can interact with. Also use this function to update the hal data if anything like sensor values change using the `updateHalDataIn` method.

Arguments

- **`halData`** (*`Object`*) – 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.
- **`dt`** (*`Number`*) – The time that has passed since the last update.

4.3 Matter.js

Matter-js is a 2D rigid body physics engine which is used to implement the simulation physics in the websim. Documentation on Matter.js can be found here.

4.4 Drivetrain Support

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

```
class MyUserPhysics extends UserPhysics {

  createRobotModel(robotConfig) {
    return new this.Models.TwoMotor();
  }

  updateSim(halData, dt) {
    const dataOut = halData.out;
    const can = dataOut.CAN;

    let lMotor = can[1].value;
    let rMotor = can[2].value;

    let {rcw, fwd} = this.model.getVector(lMotor, rMotor);

    let xSpeed = fwd * Math.cos(this.robot.angle);
    let ySpeed = fwd * Math.sin(this.robot.angle);

    this.Matter.Body.setVelocity(this.robot, { x: xSpeed, y: ySpeed });
    this.Matter.Body.setAngularVelocity(this.robot, rcw);
  }
}
```

linearDeadzone(*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.

Arguments

- **deadzone** (*Number*) –

class TwoMotorDrivetrain(*xWheelbase=2*, *speed=5*, *deadzone=null*)

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

Arguments

- **xWheelbase** (*Number*) – The distance in feet between right and left wheels.
- **speed** (*Number*) – Speed of robot in feet per second (see above)
- **deadzone** (*function*) – A function that adjusts the output of the motor (see `linear_deadzone()`)

TwoMotorDrivetrain.**getVector**(*lMotor*, *rMotor*)

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

Arguments

- **lMotor** (*Number*) – Left motor value (-1 to 1); -1 is forward
- **rMotor** (*Number*) – Right motor value (-1 to 1); 1 is forward

class FourMotorDrivetrain(*xWheelbase=2*, *speed=5*, *deadzone=null*)

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

Arguments

- **xWheelbase** (*Number*) – The distance in feet between right and left wheels.
- **speed** (*Number*) – Speed of robot in feet per second (see above)
- **deadzone** (*function*) – A function that adjusts the output of the motor (see `linear_deadzone()`)

FourMotorDrivetrain.**getVector**(*lrMotor*, *rrMotor*, *lfMotor*, *rfMotor*)

Arguments

- **lrMotor** (*Number*) – Left rear motor value (-1 to 1); -1 is forward
- **rrMotor** (*Number*) – Right rear motor value (-1 to 1); 1 is forward
- **lfMotor** (*Number*) – Left front motor value (-1 to 1); -1 is forward
- **rfMotor** (*Number*) – Right front motor value (-1 to 1); 1 is forward

Returns Object – Speed of robot (ft/s), clockwise rotation of robot (radians/s)

class MecanumDrivetrain(*xWheelbase=2*, *yWheelbase=3*, *speed=5*, *deadzone=null*)

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

Arguments

- **xWheelbase** (*Number*) – The distance in feet between right and left wheels.
- **yWheelbase** (*Number*) – The distance in feet between forward and rear wheels.
- **speed** (*Number*) – Speed of robot in feet per second (see above)
- **deadzone** (*function*) – A function that adjusts the output of the motor (see `linear_deadzone()`)

MecanumDrivetrain.**getVector**(*lrMotor*, *rrnotor*, *lfNotor*, *rfMotor*)

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

Arguments

- **lrMotor** (*Number*) – Left rear motor value (-1 to 1); 1 is forward
- **rrnotor** (*Number*) – Right rear motor value (-1 to 1); 1 is forward
- **lfNotor** (*Number*) – Left front motor value (-1 to 1); 1 is forward
- **rfMotor** (*Number*) – Right front motor value (-1 to 1); 1 is forward

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

class FourMotorSwerveDrivetrain(*xWheelbase=2*, *yWheelbase=2*, *speed=5*, *deadzone=null*)

Arguments

- **xWheelbase** (*Number*) – The distance in feet between right and left wheels.

- **yWheelbase** (*Number*) – The distance in feet between forward and rear wheels.
- **speed** (*Number*) – Speed of robot in feet per second (see above)
- **deadzone** (*function*) – A function that adjusts the output of the motor (see `linear_deadzone()`)

`FourMotorSwerveDrivetrain.`**`getVector`**(*lrMotor*, *rrMotor*, *lfMotor*, *rfMotor*, *lrAngle*, *rrAngle*, *lfAngle*, *rfAngle*)

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.

Arguments

- **lrMotor** (*Number*) – Left rear motor value (-1 to 1); 1 is forward
- **rrMotor** (*Number*) – Right rear motor value (-1 to 1); 1 is forward
- **lfMotor** (*Number*) – Left front motor value (-1 to 1); 1 is forward
- **rfMotor** (*Number*) – Right front motor value (-1 to 1); 1 is forward
- **lrAngle** (*Number*) – Left rear motor angle in degrees (0 to 360 measured clockwise from forward position)
- **rrAngle** (*Number*) – Right rear motor angle in degrees (0 to 360 measured clockwise from forward position)
- **lfAngle** (*Number*) – Left front motor angle in degrees (0 to 360 measured clockwise from forward position)
- **rfAngle** (*Number*) – Right front motor angle in degrees (0 to 360 measured clockwise from forward position)

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

4.5 Motor Configurations

Configuration parameters useful for simulating a motor. Typically these parameters can be obtained from the manufacturer via a data sheet or other specification. The websim contains config 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!

MOTOR_CFG_CIM

Motor configuration for CIM

- *Name*: CIM
- *Nominal Voltage*: 12 Volts
- *Free Speed*: 5310 RPM
- *Free Current*: 2.7 Amps
- *Stall Torque*: 2.42 N-m
- *Stall Current*: 133 Amps

MOTOR_CFG_MINI_CIM

Motor configuration for Mini CIM

- *Name*: MiniCIM

- *Nominal Voltage*: 12 Volts
- *Free Speed*: 5840 RPM
- *Free Current*: 3.0 Amps
- *Stall Torque*: 1.41 N-m
- *Stall Current*: 89.0 Amps

MOTOR_CFG_BAG
Motor configuration for Bag Motor

- *Name*: Bag
- *Nominal Voltage*: 12 Volts
- *Free Speed*: 13180 RPM
- *Free Current*: 1.8 Amps
- *Stall Torque*: 0.43 N-m
- *Stall Current*: 53.0 Amps

MOTOR_CFG_775PRO
Motor configuration for 775 Pro

- *Name*: 775Pro
- *Nominal Voltage*: 12 Volts
- *Free Speed*: 18730 RPM
- *Free Current*: 0.7 Amps
- *Stall Torque*: 0.71 N-m
- *Stall Current*: 134 Amps

MOTOR_CFG_775_125
Motor configuration for Andymark RS 775-125

- *Name*: RS775-125
- *Nominal Voltage*: 12 Volts
- *Free Speed*: 5800 RPM
- *Free Current*: 1.6 Amps
- *Stall Torque*: 0.28 N-m
- *Stall Current*: 18.0 Amps

MOTOR_CFG_BB_RS775
Motor configuration for Banebots RS 775

- *Name*: RS775
- *Nominal Voltage*: 12 Volts
- *Free Speed*: 13050 RPM
- *Free Current*: 2.7 Amps
- *Stall Torque*: 0.72 N-m
- *Stall Current*: 97.0 Amps

MOTOR_CFG_AM_9015
Motor configuration for Andymark 9015

- *Name*: AM-9015
- *Nominal Voltage*: 12 Volts
- *Free Speed*: 14270 RPM
- *Free Current*: 3.7 Amps
- *Stall Torque*: 0.36 N-m
- *Stall Current*: 71.0 Amps

MOTOR_CFG_BB_RS550
Motor configuration for Banebots RS 550

- *Name*: RS550
- *Nominal Voltage*: 12 Volts
- *Free Speed*: 19000 RPM
- *Free Current*: 0.4 Amps
- *Stall Torque*: 0.38 N-m
- *Stall Current*: 84.0 Amps

4.6 Tank Drive Model Support

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 MotorModel(*motorConfig*, *kv*, *ka*, *vintercept*)
Motor model used by the `TankModel`. You should not need to create this object if you're using the `TankModel` class.

Arguments

- **motorConfig** (*Object*) – 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)

MotorModel.**compute**(*motorPct*, *tmDiff*)

Arguments

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

class TankModel(*motorConfig*, *robotMass*, *xWheelbase*, *robotWidth*, *robotLength*, *lKv*, *lKa*, *lVi*, *rKv*, *rKa*, *rVi*)

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 can use whatever units you would like to specify the input parameter for your robot, the websim 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 40kg robot with 2 CIM motors on each side with 6 inch wheels:

```
class MyUserPhysics extends UserPhysics {

  createRobotModel(robotConfig) {
    let math = this.Math;

    let model = this.Models.TankModel.theory(
      this.Models.MotorConfigs.MOTOR_CFG_CIM,
      math.unit(40, 'kg'),
      10.71,
      2,
      math.unit(2, 'ft'),
      math.unit(6, 'inch')
    );

    return model;
  }

  updateSim(halData, dt) {

    const dataOut = halData.out;
    const pwm = dataOut.pwm;

    let lrMotor = pwm[1].value;
    let rrMotor = pwm[2].value;

    let {rcw, fwd} = this.model.getVector(lrMotor, rrMotor, dt);

    let xSpeed = fwd * Math.cos(this.robot.angle);
    let ySpeed = fwd * Math.sin(this.robot.angle);
```

(continues on next page)

(continued from previous page)

```
    this.Matter.Body.setVelocity(this.robot, { x: xSpeed, y: ySpeed });
    this.Matter.Body.setAngularVelocity(this.robot, rcw);
  }
}
```

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!

Arguments

- **motorConfig** (*Object*) – Motor specification
- **robotMass** (*Quantity*) – Mass of robot
- **xWheelbase** (*Quantity*) – Wheelbase of the robot
- **robotWidth** (*Quantity*) – Width of the robot
- **robotLength** (*Quantity*) – Length of the robot
- **lKv** (*Quantity*) – Left side `kv`
- **lKa** (*Quantity*) – Left side `ka`
- **lVi** (*volt*) – Left side `Vintercept`
- **rKv** (*Quantity*) – Right side `kv`
- **rKa** (*Quantity*) – Right side `ka`
- **rVi** (*Quantity*) – Right side `Vintercept`

`TankModel.`**getVector**(*lMotor*, *rMotor*, *tmDiff*)

Given motor values and the amount of time elapsed since this was last called, retrieves the velocity and anglular velocity of the robot.

To update your encoders, use the `lPosition` and `rPosition` attributes of this object.

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

Arguments

- **lMotor** (*Number*) – Left motor value (-1 to 1); -1 is forward
- **rMotor** (*Number*) – Right motor value (-1 to 1); 1 is forward
- **tmDiff** (*Number*) – Elapsed time since last call to this function

`TankModel.`**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`

`TankModel.`**lPosition**

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

`TankModel.`**`lVelocity`**
The velocity of the left side (in ft/s)

`TankModel.`**`rPosition`**
The linear position of the right side wheel (in feet)

`TankModel.`**`rVelocity`**
The velocity of the left side (in ft/s)

`TankModel.`**`theory`**(*motorConfig*, *robotMass*, *gearing*, *nmotors*, *xWheelbase*, *robotWidth*, *robotLength*, *wheelDiameter*, *vintercept*)
Use this to create the drivetrain model when you haven't measured `kv` and `ka` for your robot.

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

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

Arguments

- **`motorConfig`** (*`Object`*) – Specifications for your motor
- **`robotMass`** (*`Quantity`*) – Mass of the robot
- **`gearing`** (*`Number`*) – Gear ratio .. so for a 10.74:1 ratio, you would pass 10.74
- **`nmotors`** (*`Number`*) – Number of motors per side
- **`xWheelbase`** (*`Quantity`*) – Wheelbase of the robot
- **`robotWidth`** (*`Quantity`*) – Width of the robot
- **`robotLength`** (*`Quantity`*) – Length of the robot
- **`wheelDiameter`** (*`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)

4.7 Unit Conversion

pyfrc uses the math.js 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:

```
five_feet = math.unit(5, 'ft');
```

The websim defines the following custom units:

- `cpm`: Counts per minute. Used to represent motor free speed
- `Nm`: Shorthand for N-m or newton-meter. Used for motor torque.
- `tmka`: The kA value used in the tankmodel (uses imperial units)
- `tmkv`: The kV value used in the tankmodel (uses imperial units)

Refer to the math.js unit documentation for more information.

4.8 Installation

If you have python3 and pip installed, then do:

```
pip3 install robotpy-websim
```

Alternatively, you can install from a git checkout:

```
git clone https://github.com/robotpy/robotpy-websim.git
cd robotpy-websim
pip3 install .
```

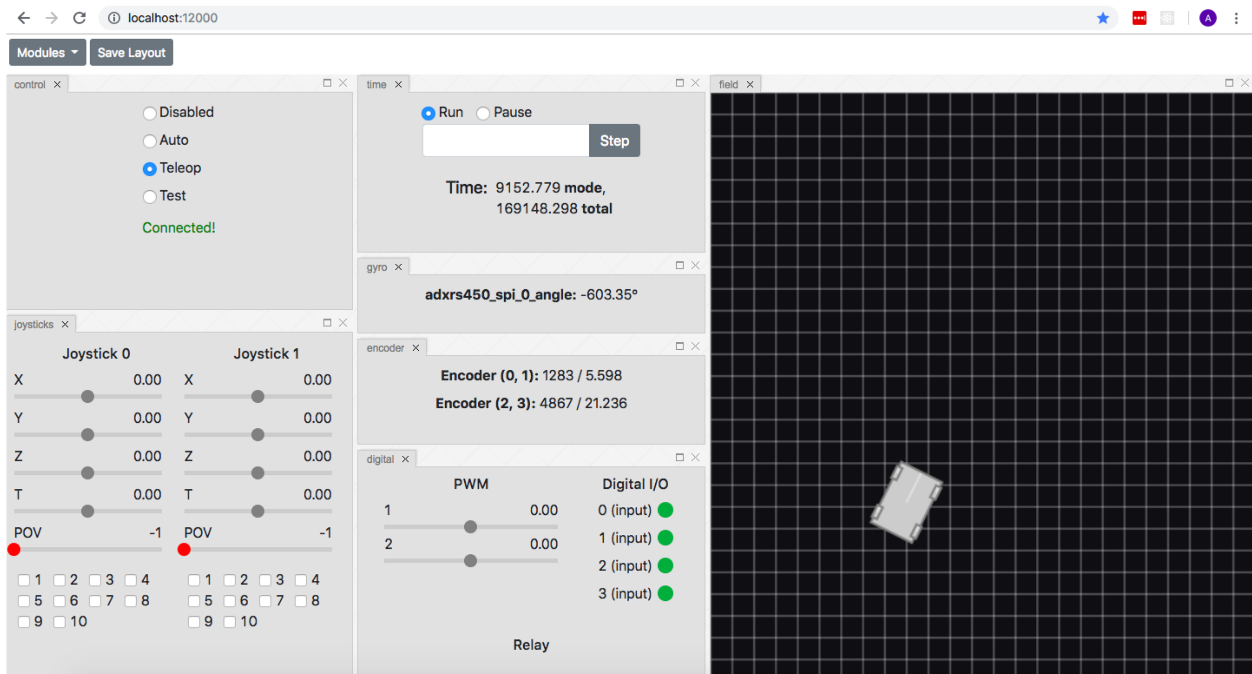
4.9 Starting a simulation

You can run your robot.py directly like so:

```
python3 robot.py websim
```

Your default browser (or Chrome) should be launched and show the control interface. If it does not show automatically, you can browse to http://localhost:8000/

4.10 Interface



The websim interface is made up of containers called modules that can be dragged and dropped, exited, and organized into rows, columns and stacks. Modules contain inputs, controls, labels, and other components that allow you to interact and get information from a simulated robot.

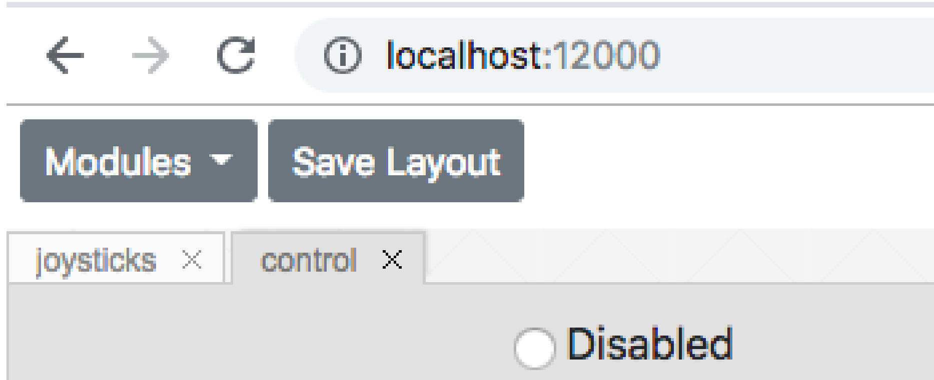
4.10.1 Dragging and Dropping

To add modules to the interface you must open the module menu and drag and drop them onto the interface.

4.10.2 Organizing Layout

Modules placed on the layout can be organized and rearanged into rows, columns and stacks.

4.10.3 Saving Layout



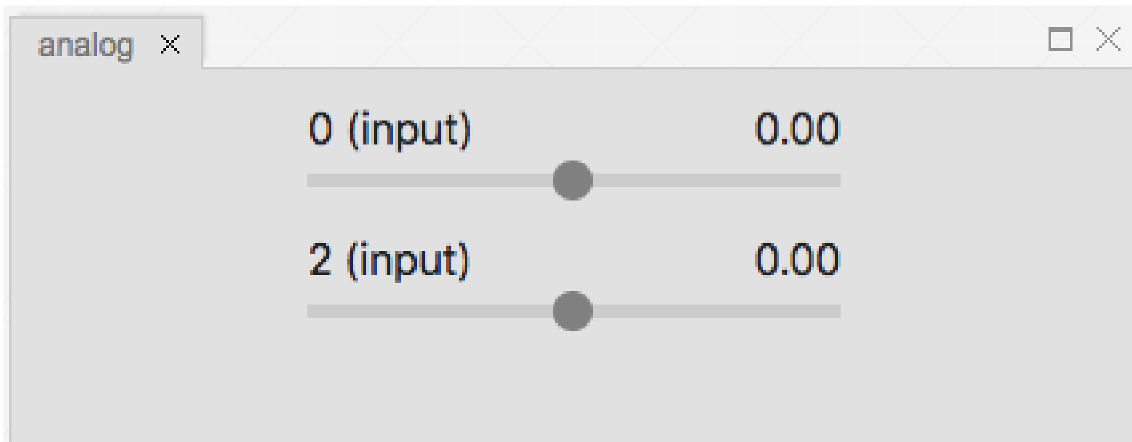
To save the layout click the `Save Layout` button next to the module menu. The layout data is saved to the `user-config.json` file in the `sim` folder parallel to your `robot.py` file.

4.10.4 Golden Layout

The websim uses a mult-window javascript layout manager called **Golden Layout**. Visit their site to learn more about how **Golden Layout** works.

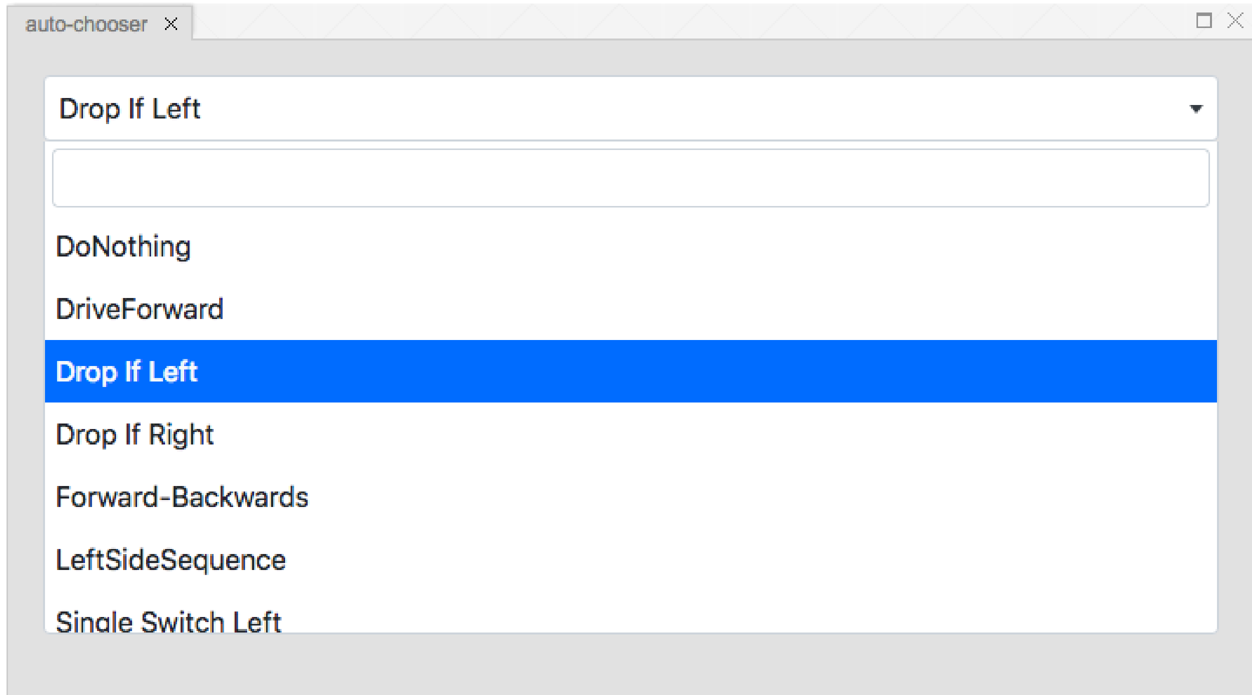
4.11 Built in Modules

4.11.1 Analog



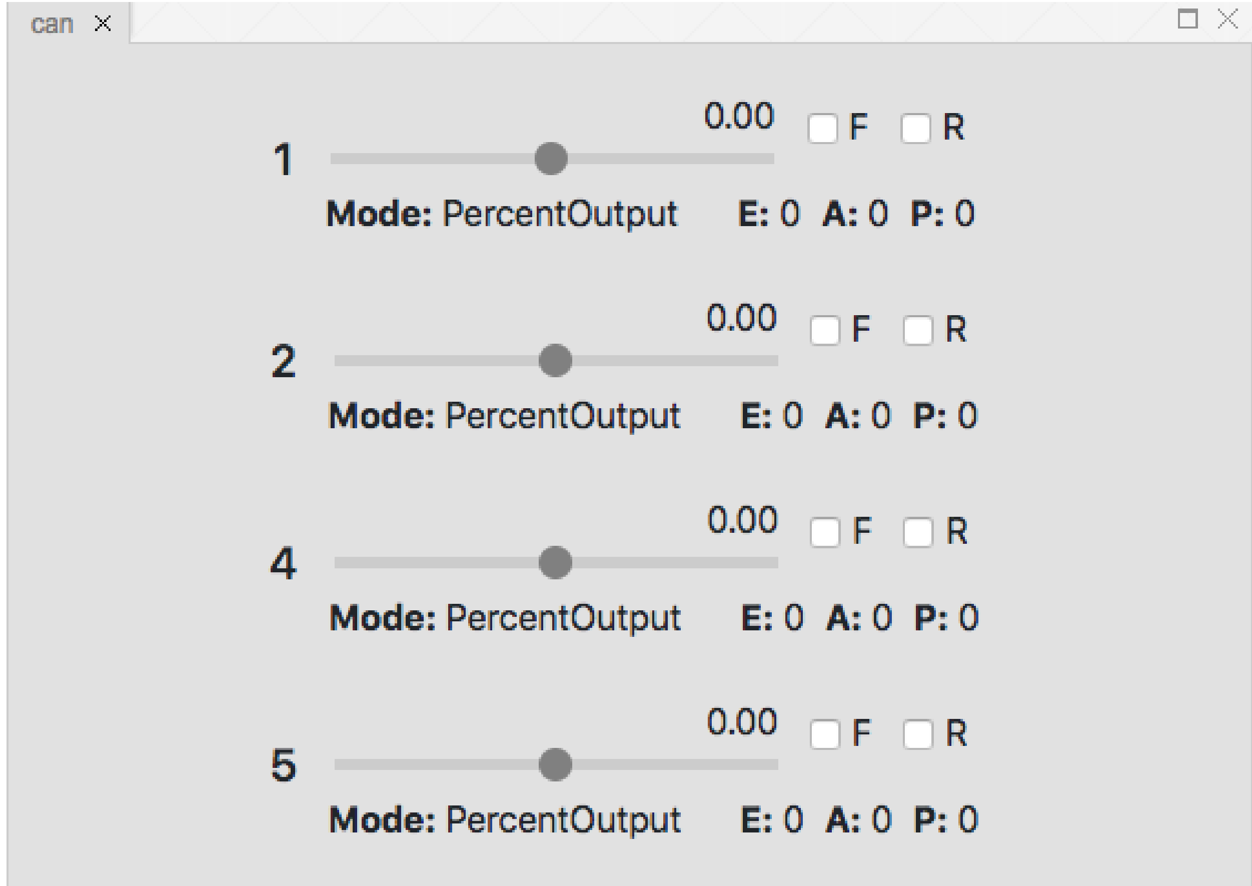
The analog module is used to display digital outputs and control the value of digital inputs with sliders.

4.11.2 Auto Chooser



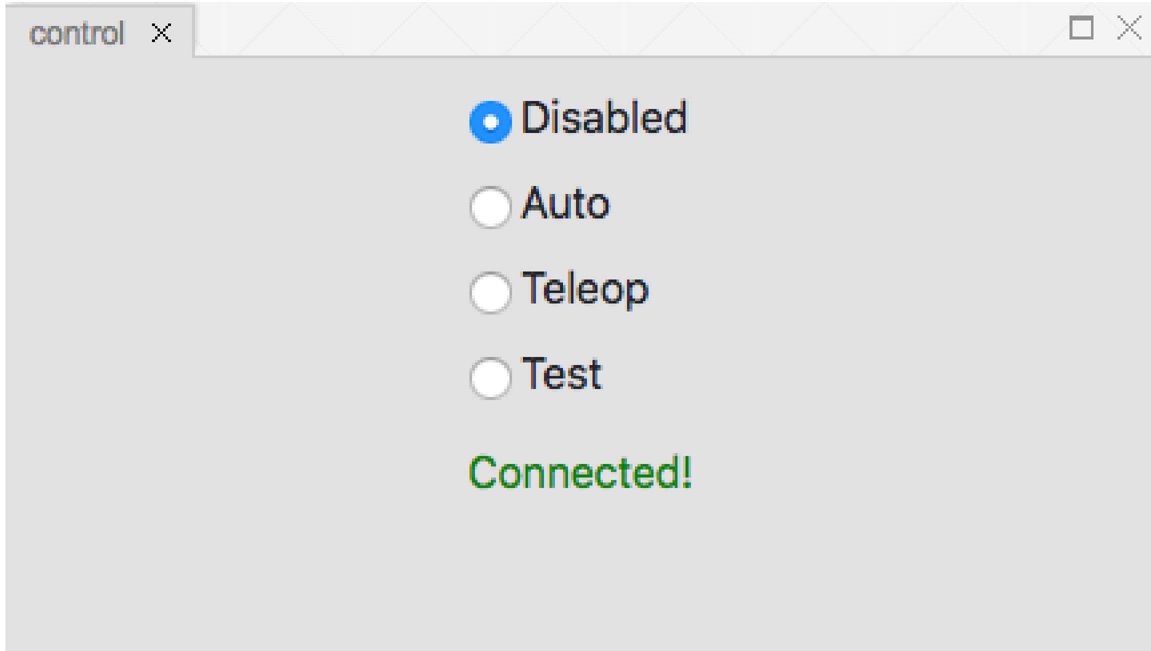
If you have multiple autonomous modes you can pick one using the auto chooser module.

4.11.3 CAN



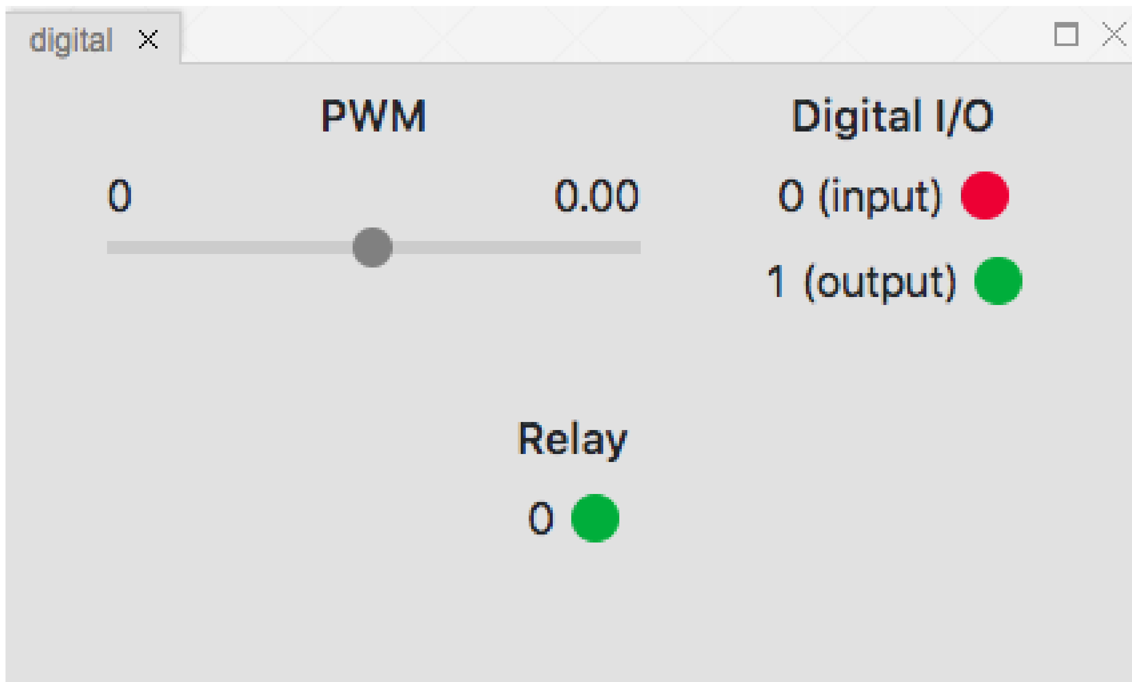
CAN devices such as the Talon SRX are simulated using the CAN module.

4.11.4 Mode Picker



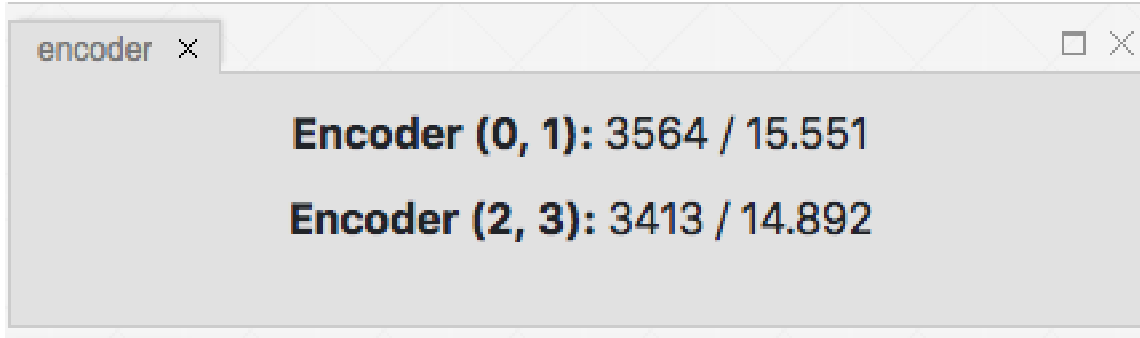
You can select the robot mode using the mode picker. For example, if you want to test your autonomous code you can select the *Auto* radio button.

4.11.5 Digital



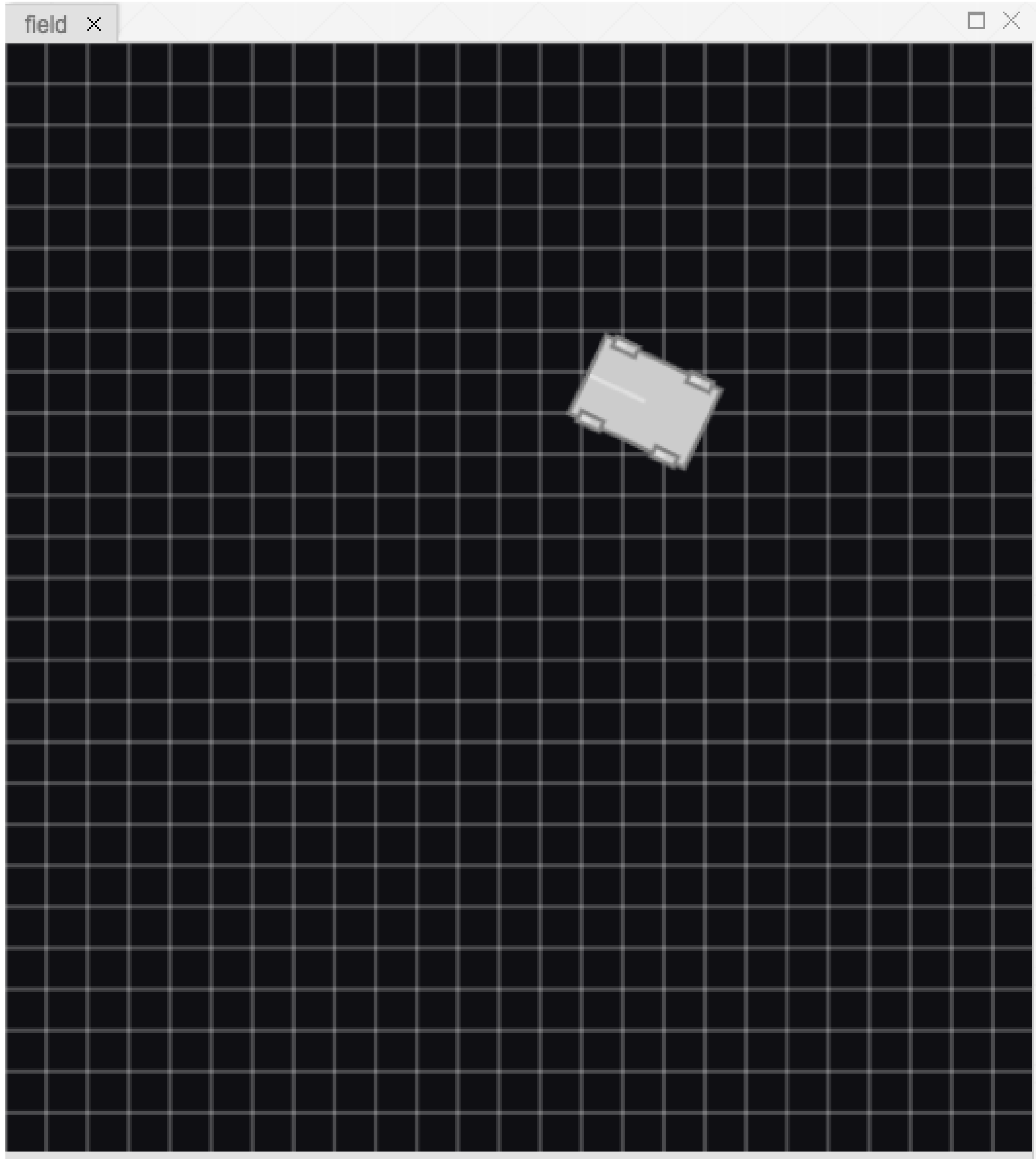
The digital module simulates digital devices such as PWMs, Digital I/Os, and Relays.

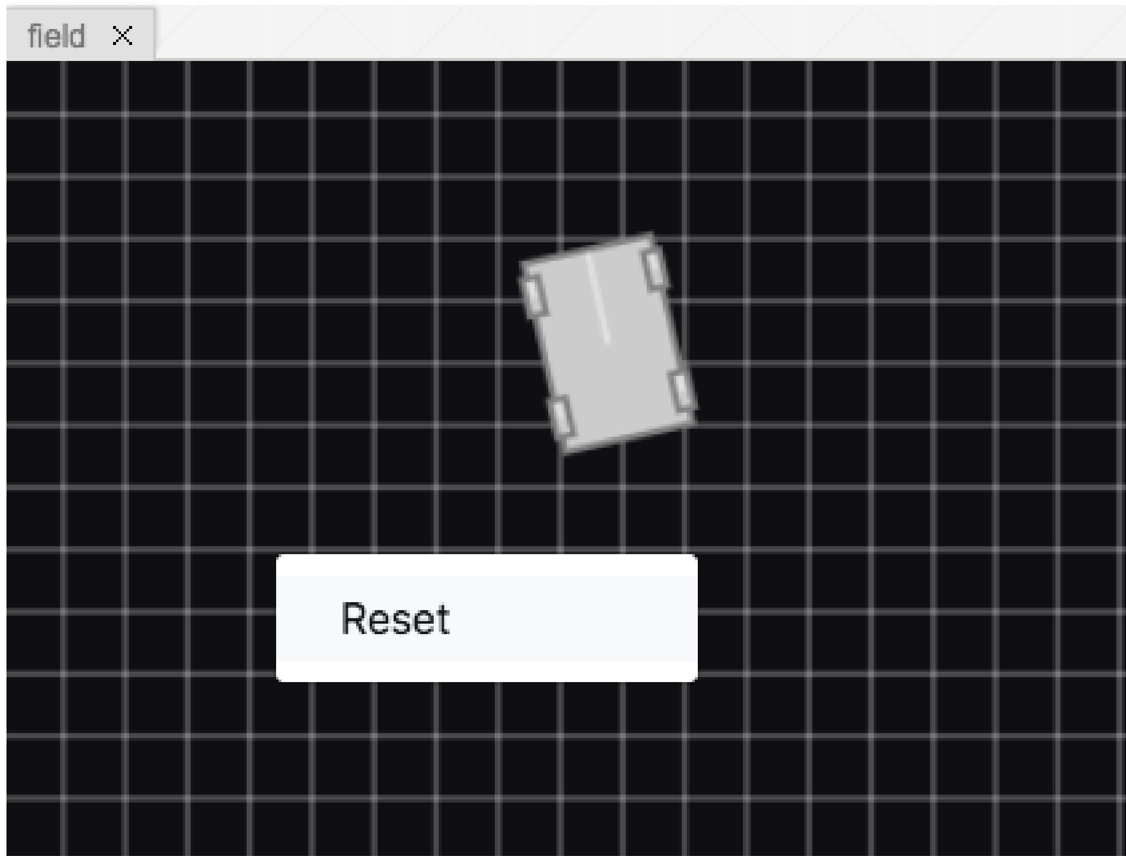
4.11.6 Encoder



Shows source channels, count, and distance traveled for simulated encoders. Encoder counts can be set in the `MyUserPhysics` class in the `physics.js` file.

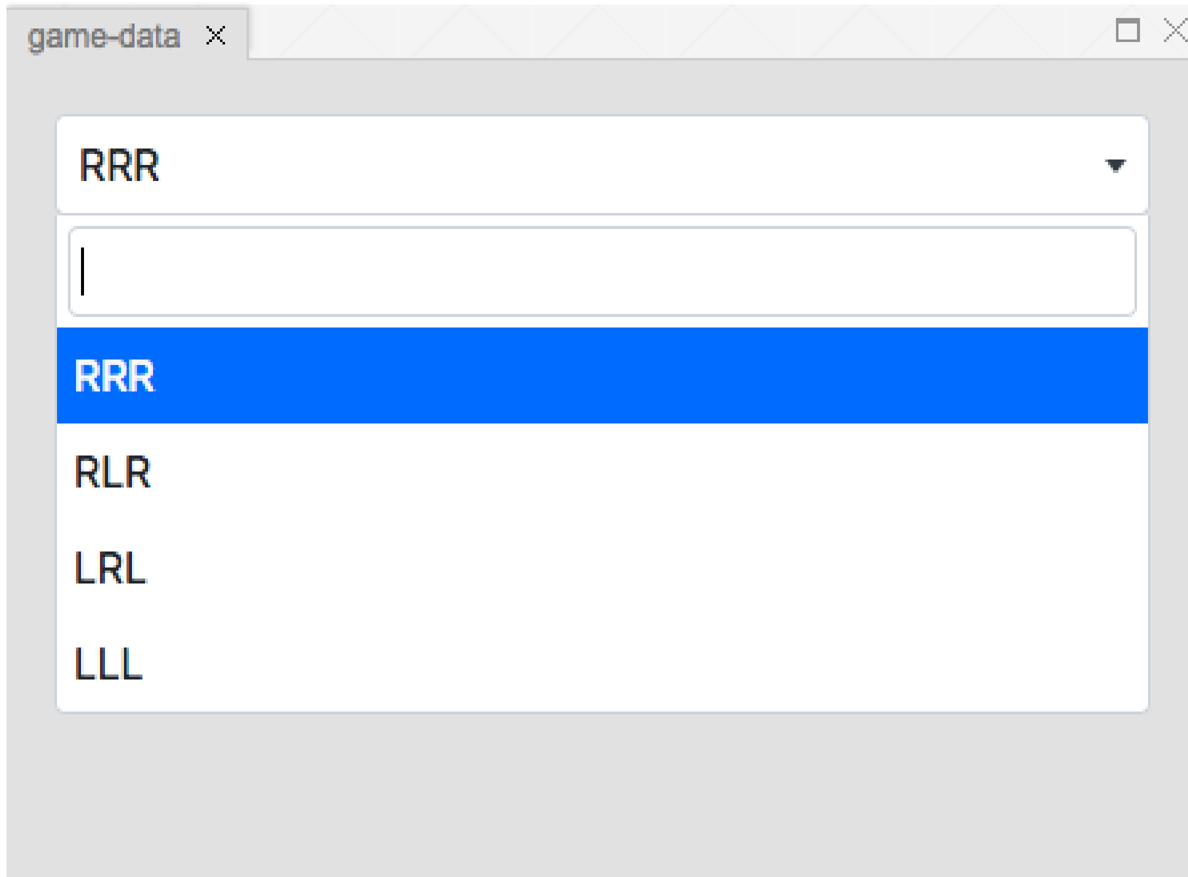
4.11.7 Field





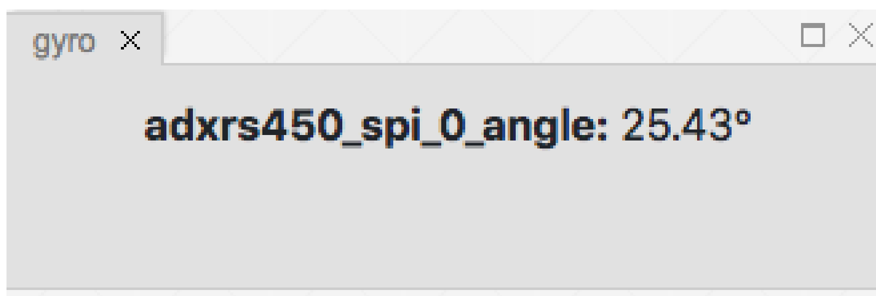
The field module is where the simulated robot physics happens. You can reset the field by opening the context menu and clicking the `Reset` menu item.

4.11.8 Game Data



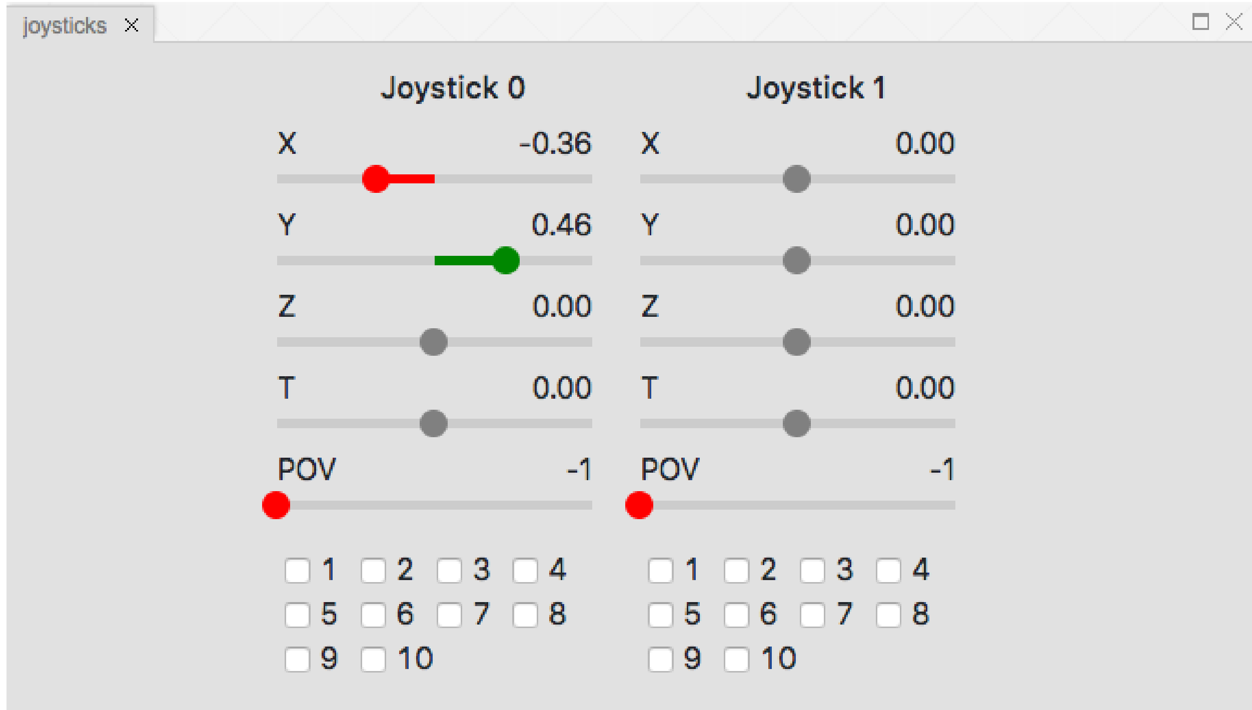
The game data module is used to set the game specific message.

4.11.9 Gyro



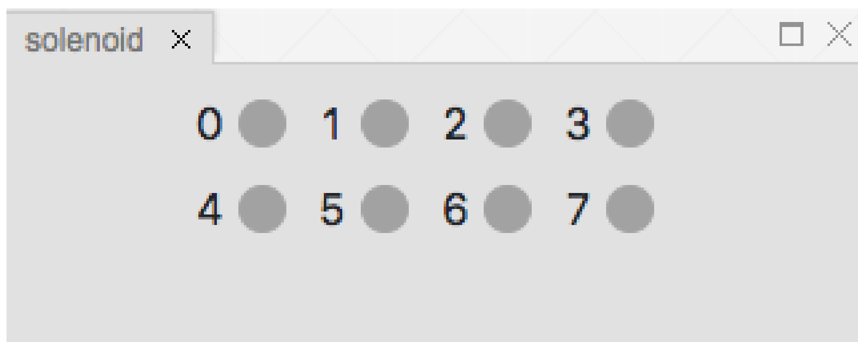
The gyro module is used to display the simulated gyro values. The gyro value is set using the simulated robot in the field module.

4.11.10 Joystick



The joystick module is used to simulate joystick values. Axes are set using sliders and buttons are set using checkboxes. Joystick axes and buttons can also be set using plugged in gamepads.

4.11.11 Solenoid



The solenoid module is used to get simulated solenoid values.

4.11.12 Tableviewer

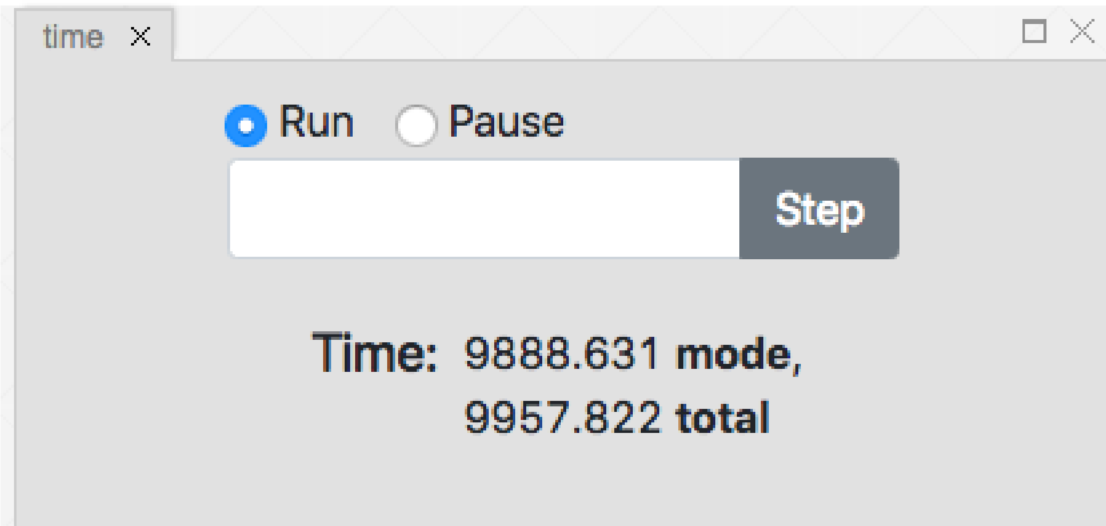
tableviewer

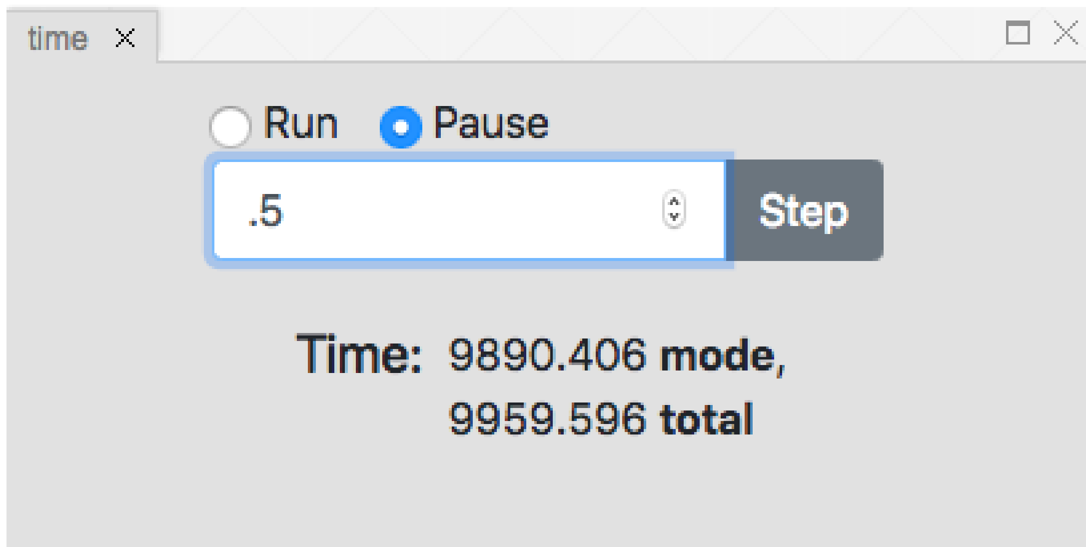
Key	Type	Value
▾ root		
▾ FMSInfo		
.type	string	FMSInfo
GameSpecificMessage	string	
EventName	string	sim-event
MatchNumber	number	0
ReplayNumber	number	0
MatchType	number	0
IsRedAlliance	boolean	☑ true
StationNumber	number	1

MatchType	number	0
sRedAlliance	boolean	☑ true
StationNumber	number	1
FMSControlData	number	35
iveWindow		

Tableviewer is used to display and set Networktable values. Carets can be clicked to expand subtables. Values can be edited by clicked on the value column and added from modals which can be opened from the context menu.

4.11.13 Time





The time module can be used to view the time spent in total and in the current mode, pause the sim, and step forward a certain amount of time in the simulation.

4.12 Custom Modules

The websim API gives you the ability to create your own modules and add them to the interface.

4.12.1 Module Anatomy

Custom modules can be added by creating a folder with an `index.js` file under the `sim` directory parellel to your `robot.py` file.

4.12.2 Riotjs

TODO

4.12.3 Redux

TODO

4.12.4 Tag Anatomy

TODO

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

F

FourMotorDrivetrain() (class), 13
FourMotorDrivetrain.getVector() (FourMotorDrivetrain method), 14
FourMotorSwerveDrivetrain() (class), 14
FourMotorSwerveDrivetrain.getVector() (FourMotorSwerveDrivetrain method), 15

L

linearDeadzone() (built-in function), 12

M

MecanumDrivetrain() (class), 14
MecanumDrivetrain.getVector() (MecanumDrivetrain method), 14
MOTOR_CFG_775_125 (None attribute), 16
MOTOR_CFG_775PRO (None attribute), 16
MOTOR_CFG_AM_9015 (None attribute), 16
MOTOR_CFG_BAG (None attribute), 16
MOTOR_CFG_BB_RS550 (None attribute), 17
MOTOR_CFG_BB_RS775 (None attribute), 16
MOTOR_CFG_CIM (None attribute), 15
MOTOR_CFG_MINI_CIM (None attribute), 15
MotorModel() (class), 17
MotorModel.compute() (MotorModel method), 17

T

TankModel() (class), 17
TankModel.getVector() (TankModel method), 19
TankModel.inertia (TankModel attribute), 19
TankModel.lPosition (TankModel attribute), 19
TankModel.lVelocity (TankModel attribute), 19
TankModel.rPosition (TankModel attribute), 20
TankModel.rVelocity (TankModel attribute), 20
TankModel.theory() (TankModel method), 20
TwoMotorDrivetrain() (class), 13
TwoMotorDrivetrain.getVector() (TwoMotorDrivetrain method), 13

U

UserPhysics() (class), 10
UserPhysics.addDeviceGyroChannel() (UserPhysics method), 10
UserPhysics.createField() (UserPhysics method), 10
UserPhysics.createRobot() (UserPhysics method), 10
UserPhysics.createRobotModel() (UserPhysics method), 10
UserPhysics.disableRobot() (UserPhysics method), 11
UserPhysics.init() (UserPhysics method), 11
UserPhysics.reset() (UserPhysics method), 11
UserPhysics.updateGyros() (UserPhysics method), 11
UserPhysics.updateHalDataIn() (UserPhysics method), 11
UserPhysics.updateSim() (UserPhysics method), 11