Gridsim

Release 0.1.3

Julia Ebert

Apr 20, 2020

CONTENTS

CHAPTER

ONE

INDEX

1.1 Getting Started

1.1.1 Installation

Note: This assumes that you're already familiar with virtual environments and pip.

Virtual Environment Setup

Create a Python 3 virtual environment in the current location in subfolder called `venv`, then set it as the Python source.

```
$ python3 -m venv venv
$ source venv/bin/activate
```

You can deactivate the virtual environment with `deactivate`.

Quick Install

This package is available through pip, so it's easy to install. With your virual environment active, run:

```
$ pip install gridsim
```

Within your own code, you can now import the Gridsim library components, such as:

```
import gridsim as gs

# Create an empty World of 100 x 100 grid cells
my_world = gs.World(100, 100)
```

1.1.2 Basic Usage

This will walk you through setting up your first robot and complete simulation.

On this page

Test using built in examples

The examples are in the examples directory of the source code. In the near future, I'll set up a way to run the examples directly when you install the package.

Creating a simple robot

For more detailed information about developing custom robots, see *Make your own Robot*.

To start, we will only need to make a simple robot based on the `GridRobot`. This needs to implement three methods:

- `receive_msg()`: Code that is run when a robot receives a message
- `init()`: Code that is run once when the robot is created
- `loop()`: Code that is run in every step of the simulation

Create a file for your robot class. Let's call it `random_robot.py`. Below is a simple Robot that moves randomly and changes direction every 10 seconds. You can copy this or directly download `random_robot.py`

```
1  import random
2
3  from gridsim.grid_robot import GridRobot
4  import gridsim as gs
5
6
7  class RandomRobot(GridRobot):
8      # Change direction every 10 ticks
9      DIR_DURATION = 10
10
11     def init(self):
12         self.set_color(255, 0, 0)
13         self._msg_sent = False
14
15         # Next tick when Robot will change direction
16         self._next_dir_change = self.get_tick()
17
```

(continues on next page)

(continued from previous page)

```
    def receive_msg(self, msg: gs.Message, dist: float):
        # This robot got a message from another robot
        self._msg_sent = True

    def loop(self):

        # Change direction every DIR_DURATION ticks
        tick = self.get_tick()
        if tick >= self._next_dir_change:
            new_dir = random.choice(GridRobot.DIRS)
            self.set_direction(new_dir)
            self._next_dir_change = tick + RandomRobot.DIR_DURATION

        # Broadcast a test message to any robots nearby
        msg = gs.Message(self.id, {'test': 'hello'})
        self.set_tx_message(msg)

        # Sample the environment at the current location
        c = self.sample()

        # Change color depending on whether messages have been sent or received
        # Robot will be white when it has successfully sent & received a message
        blue = 255 * self._msg_sent
        # self.set_color(255, green, 0)
        self.set_color(255-c[0], 255-c[1], blue)
```

A minimal simulation example

To run a simulation, you need to create a couple of robots, place them in a *World*. Then you call the *step()* method to execute you simulation step-by-step. *step()* will handle running all of the robots' code, as well as communication and movement.

We also want give our Robots something to sense by adding en environment to the World. An environment here is represented with an image. (You'll see what this looks like in the next step.) In each cell, the Robots can sense the color of the cell (i.e., the RGB pixel value) at that location with the *sample()* method. If you set up the environment with an image whose resolution doesn't match the grid dimensions, it will be rescaled, possibly stretching the image. To avoid any surprises, you should use an image whose resolution matches your grid dimensions (e.g., for a 50 × 50 grid, use a 50px × 50px image).

Use the code below or download `minimal_simulation.py` and the example environment `ex_env.png`.

```
import gridsim as gs

from random_robot import RandomRobot


def main():
    grid_width = 50  # Number of cells for the width & height of the world
    num_robots = 5
    num_steps = 100  # simulation steps to run

    # Create a few robots to place in your world
    robots = []
    for n in range(num_robots):
        robots.append(RandomRobot(grid_width/2 - n*2,
```

(continues on next page)

(continued from previous page)

```
                                    grid_width/2 - n*2))

    # Create a 50 x 50 World with the Robots
    world = gs.World(grid_width, grid_width,
                     robots=robots,
                     environment="ex_env.png")

    # Run the simulation
    for n in range(num_steps):
        # Execute a simulation step
        world.step()
        # To make sure it works, print the tick (world time)
        print('Time:', world.get_time())

    print('SIMULATION FINISHED')


if __name__ == '__main__':
    # Run the simulation if this program is called directly
    main()
```

With these files and `random_robot.py` in the same directory, and `gridsim` installed, you should be able to run the code with:

```
$ python3 minimal_simulation.py
```

Adding the Viewer

With that simple example, you have no way to see what the robots are doing. For that, we add a *Viewer*. This requires adding only two lines of code to our minimal simulation above.

Use the code below or download `viewer_simulation.py`.

```
import gridsim as gs

from random_robot import RandomRobot


def main():
    grid_width = 50  # Number of cells for the width & height of the world
    num_robots = 5
    num_steps = 100  # simulation steps to run

    # Create a few robots to place in your world
    robots = []
    for n in range(num_robots):
        robots.append(RandomRobot(grid_width/2 - n*2,
                                  grid_width/2 - n*2))

    # Create a 50 x 50 World with the Robots
    world = gs.World(grid_width, grid_width,
                     robots=robots,
                     environment="ex_env.png")

    # Create a Viewer to display the World
```

(continues on next page)

(continued from previous page)

```
    viewer = gs.Viewer(world)

    # Run the simulation
    for n in range(num_steps):
        # Execute a simulation step
        world.step()

        # Draw the world
        viewer.draw()

        # To make sure it works, print the tick (world time)
        print('Time:', world.get_time())

    print('SIMULATION FINISHED')


if __name__ == '__main__':
    # Run the simulation if this program is called directly
    main()
```

Notice that adding the Viewer slows down the time to complete the simulation, because the display rate of the Viewer limits the simulation rate. If you want to run lots of simulations, turn off your Viewer.

Using configuration files

Gridsim also provides the *ConfigParser* for using YAML configuration files. This simplifies loading parameters and (as described in the next section) saving parameters with simulation results data.

The *ConfigParser* is un-opinionated; it doesn't place any restrictions on what your configuration files look like, as long as they're valid YAML files.

Compared to our `minimal_simulation.py`, we only need one line to create our *ConfigParser*, from which we can retrieve any parameter values.

Use the code below or download `config_simulation.py` and YAML configuration file `simple_config.yml`.

```
import gridsim as gs

from random_robot import RandomRobot


def main():
    config = gs.ConfigParser('simple_config.yml')
    print(config.get('name'))
    grid_width = config.get('grid_width')
    num_robots = config.get('num_robots')
    # You can specify a default value in case a parameter isn't in the
    # configuration file
    num_steps = config.get('num_steps', default=100)

    # Create a few robots to place in your world
    robots = []
    # Configuration values can also be lists, not just single values.
    x_pos = config.get('robot_x_pos')
    for n in range(num_robots):
```

(continues on next page)

(continued from previous page)

```
        robots.append(RandomRobot(x_pos[n],
                                  grid_width/2 - n*2))

    # Create a 50 x 50 World with the Robots
    world = gs.World(grid_width, grid_width, robots=robots)

    # Run the simulation
    for n in range(num_steps):
        # Execute a simulation step
        world.step()
        # To make sure it works, print the tick (world time)
        print('Time:', world.get_time())

    print('SIMULATION FINISHED')


if __name__ == '__main__':
    # Run the simulation if this program is called directly
    main()
```

Logging data

Gridsim has a built-in *Logger*, designed to easily save data from your simulations to HDF5 files. This allows you to store complex data and simulation configurations together in one place. HDF5 files are also easy to read and write in many different programming languages.

There are three main ways to save data to your log files:

- Save the parameters in your configuration with *log_config()*. (Note that not all data types can be saved with `log_config`. See its documentation for more details.)
- Save a single parameter (that's not in your configuration file) with *log_param()*
- Save the state of your simulation/robots with *log_state()*. (This requires some setup.)

In order to log the state of the World, you first need to tell the *Logger* *what* you want to save about the `log_state`, this function is called and the result is added to your dataset. You can add as many aggregators as you want, each with their own name.

We can extend our `config_simulation.py` to show the three types of logging described above. Use the code below or download `logger_simulation.py`.

```
import gridsim as gs
from typing import List
import numpy as np
from datetime import datetime

from random_robot import RandomRobot


def green_agg(robots: List[gs.Robot]) -> np.ndarray:
    """
    This is a dummy aggregator function (for demonstration) that just saves
    the value of each robot's green color channel
    """
    out_arr = np.zeros([len(robots)])
    for i, r in enumerate(robots):
```

(continues on next page)

(continued from previous page)

```
        out_arr[i] = r._color[1]

    return out_arr


def main():
    config = gs.ConfigParser('simple_config.yml')
    print(config.get('name'))
    grid_width = config.get('grid_width')
    num_robots = config.get('num_robots')
    # You can specify a default value in case a parameter isn't in the
    # configuration file
    num_steps = config.get('num_steps', default=100)

    # Create a few robots to place in your world
    robots = []
    # Configuration values can also be lists, not just single values.
    x_pos = config.get('robot_x_pos')
    for n in range(num_robots):
        robots.append(RandomRobot(x_pos[n],
                                  grid_width/2 - n*2))

    # Create a 50 x 50 World with the Robots
    world = gs.World(grid_width, grid_width, robots=robots)

    # Logger
    trial_num = config.get('trial_num', default=1)
    # Create a logger for this world that saves to the `test.h5` file
    logger = gs.Logger(world, 'test.h5', trial_num=trial_num,
                       overwrite_trials=True)
    # Tell the logger to run the `green_agg` function every time that
    # `log_state` is called
    logger.add_aggregator('green', green_agg)
    # Save the contents of the configuration, but leave out the 'name' parameter
    logger.log_config(config, exclude='name')
    # Save the date/time that the simulation was run
    logger.log_param('date', str(datetime.now()))

    # Run the simulation
    for n in range(num_steps):
        # Execute a simulation step
        world.step()

        # Log the state every step
        logger.log_state()

        # To make sure it works, print the tick (world time)
        print('Time:', world.get_time())

    print('SIMULATION FINISHED')


if __name__ == '__main__':
    # Run the simulation if this program is called directly
    main()
```

Complete example

Most simulations will involve all of these components, and multiple trials. You can download a complete, detailed example here: `complete_simulation.py`, as well as a corresponding YAML configuration file: `ex_config.yml`

Here, the configuration file is used as a command line argument, so it's easy to switch what configuration file you use. Run it like this:

```
$ python3 complete_simulation.py ex_config.yml
```

1.1.3 Make your own Robot

Note: This assumes familiarity with object-oriented programming (particularly inheritance and abstract classes).

The Gridsim library provides a *Robot* class that manages underlying behavior and drawing of robots, making it easy for you to quickly implement your own functionality and algorithms.

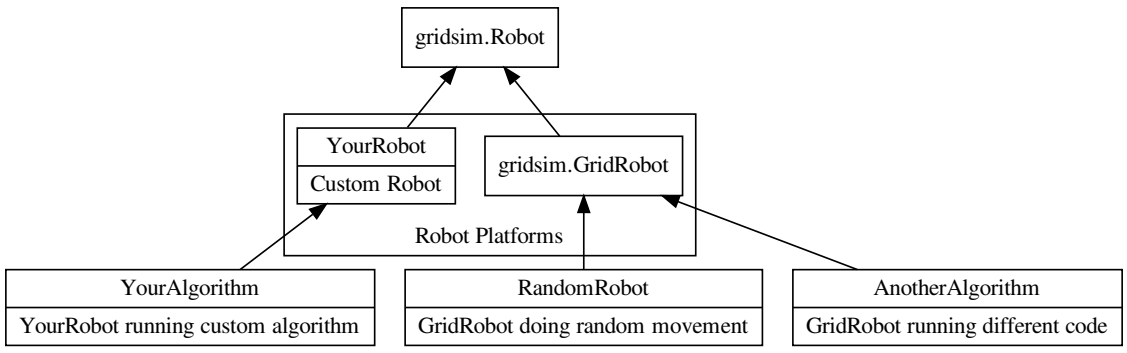
In fact, the default `Robot` class is an abstract class; you must *implement* your own `Robot` subclass. There are five abstract `Robot` methods that you must implement in your own class. (Inputs and outputs are not shown.)

- `move()`: Step-wise movement of the robot on the grid
- *comm_criteria()*: Distance-based criteria for whether or not another robot is within communication range of this robot.
- *receive_msg()*: Code that is run when a robot receives a message
- *init()*: Code that is run once when the robot is created
- *loop()*: Code that is run in every step of the simulation

It also includes an optional method you may want to implement in your subclass:

- *msg_received()*: Code that is run when a robot's successfully sends a message to another robot.

In general, you will likely want to implement your own robots with an additional *two* layers of subclasses, as seen in the graph below. This allows you to separate the physical robot platform you are representing from the algorithms/code you are running on that platform.



First, you create a subclass that represents the physical robot system you are representing (such as a Turtlebot or Kilobot). This is still an abstract class. It implements abstract methods that are properties of the physical system, such as the communication range (*comm_criteria()*) and movement restrictions (`move()`). Gridsim include the *GridRobot* as a simple robot platform. You can also create your down, as in the `YourRobot` above.

Second, you create a subclass of your new class for implementing specific algorithms or code on your new robot platform. Here you will implement message handling (*receive_msg()* and optionally *msg_received()*) and onboard code (*init()* and *loop()*). You can have multiple subclasses of your platform to run different code on the same platform, such as `RandomRobot` (created below as an example) and `AnotherAlgorithm`.

Custom robot example

Below is an example of the structure described above to create a simple robot that bounces around the arena.

First, we create , a robot with a circular communication radius of 5 grid cells that can move in the cardinal directions to any of four cells surrounding it. This robot is already provided in the library as *GridRobot*; you need not re-implement this robot platform if it meets your needs.

```
from typing import Tuple

from .robot import Robot
# If you are building your own Robot class, you would instead use:
# from gridsim import Robot


class GridRobot(Robot):
    STAY = 0
    UP = 1
    DOWN = 2
    LEFT = 3
    RIGHT = 4
    DIRS = [STAY, UP, DOWN, LEFT, RIGHT]

    def __init__(self, x: int, y: int, comm_range: float = 5):
        """
        Create a robot that moves along the cardinal directions. Optionally, you
        can specify a communication range for the robots.

        Parameters
        ----------
        x : int
            Starting x position (grid cell) of the robot
        y : int
            Starting y position (grid cell) of the robot
        comm_range : float, optional
            Communication radius (in grid cells) of the robot, by default 5
        """
        # Run all of the initialization for the default Robot class, including
        # setting the starting position
        super().__init__(x, y)

        self._comm_range = comm_range
        # Start with the robot stationary
        self._move_cmd = GridRobot.STAY
```

(continues on next page)

(continued from previous page)

```
    def set_direction(self, dir: int):
        """
        Helper function to set the direction the robot will move. Note that this
        will persist (the robot will keep moving) until the direction is
        changed.

        Parameters
        ----------
        dir : int
            Direction to move, one of UP, DOWN, LEFT, RIGHT, or STAY

        Raises
        ------
        ValueError
            If given direction is not one of UP, DOWN, LEFT, RIGHT, or STAY
        """
        if dir in GridRobot.DIRS:
            self._move_cmd = dir
        else:
            raise ValueError('Invalid movement direction "{}"'.format(dir))

    def move(self) -> Tuple[int, int]:
        """
        Determine the cell the Robot will move to, based on the direction set in
        by :meth:`~gridsim.grid_robot.GridRobot.set_motors`.

        Returns
        -------
        Tuple[int, int]
            (x,y) grid cell the robot will move to, if possible/allowed
        """
        x, y = self.get_pos()
        if self._move_cmd == GridRobot.UP:
            y -= 1
        elif self._move_cmd == GridRobot.DOWN:
            y += 1
        elif self._move_cmd == GridRobot.RIGHT:
            x += 1
        elif self._move_cmd == GridRobot.LEFT:
            x -= 1
        # else STAY, which keeps current position
        return x, y

    def comm_criteria(self, dist: int) -> bool:
        """
        Robots can communicate if their Euclidean distance is <= the radius
        specified at initialization (by default, 5 cells)

        Parameters
        ----------
        dist : int
            Euclidean distance of the other robot with which to communicate

        Returns
        -------
        bool
```

(continues on next page)

(continued from previous page)

```
            Whether distance is <= the communication radius
        """
        return dist <= self._comm_range
```

With our robot platform in place, we can now implement a Robot that implements whatever code we want the robot to run. In this case, it's a simple robot that chooses a random movement every 10 ticks. Its color is based on the color it samples at its current location, and whether it has communicated with another robot.

```
import random

from gridsim.grid_robot import GridRobot
import gridsim as gs


class RandomRobot(GridRobot):
    # Change direction every 10 ticks
    DIR_DURATION = 10

    def init(self):
        self.set_color(255, 0, 0)
        self._msg_sent = False

        # Next tick when Robot will change direction
        self._next_dir_change = self.get_tick()

    def receive_msg(self, msg: gs.Message, dist: float):
        # This robot got a message from another robot
        self._msg_sent = True

    def loop(self):

        # Change direction every DIR_DURATION ticks
        tick = self.get_tick()
        if tick >= self._next_dir_change:
            new_dir = random.choice(GridRobot.DIRS)
            self.set_direction(new_dir)
            self._next_dir_change = tick + RandomRobot.DIR_DURATION

        # Broadcast a test message to any robots nearby
        msg = gs.Message(self.id, {'test': 'hello'})
        self.set_tx_message(msg)

        # Sample the environment at the current location
        c = self.sample()

        # Change color depending on whether messages have been sent or received
        # Robot will be white when it has successfully sent & received a message
        blue = 255 * self._msg_sent
        # self.set_color(255, green, 0)
        self.set_color(255-c[0], 255-c[1], blue)
```

Notice that the abstraction layers mean that you have to write very little additional code to implement a new algorithm for your robot.

1.2 Class Reference

Each page contains details and full API reference for all the classes in the Gridsim library.

For an explanation of how to use all of it together, see *Basic Usage*.

1.2.1 World

The World is where all of the simulation happens. Robots are added to the World, and the Viewer and Logger refer to a World to draw the simulation and save data.

Once the World is created and you have added your robots, you will likely only need to call the `step()` method.

class gridsim.world.**World**(*width: int*, *height: int*, *robots: List[*gridsim.robot.Robot*] = []*, *environment: str = ''*, *allow_collisions: bool = True*)

__init__(*width: int*, *height: int*, *robots: List[*gridsim.robot.Robot*] = []*, *environment: str = ''*, *allow_collisions: bool = True*)

Create a World for simulating Robots in a grid world

Parameters

- **width** (*int*) – Width of the world (number of cells)
- **height** (*int*) – Height of the world (number of cells)
- **robots** (*List[Robot], optional*) – List of Robots to place in the World to start, by default []. Additional robots can be added after initialization with the *add_robot* method.
- **environment** (*str, optional*) – Filename of an image to use for a background in the World. Robots will be able to sense the color of this image. If the environment dimensions do not match the World dimensions, the image will be re-scaled (and possibly stretched). We recommend using an image with the same resolution as your grid size.
- **allow_collisions** (*bool, optional*) – Whether or not to allow Robots to exist in the same grid cell, by default True

add_environment(*img_filename: str*)

Add an image to the environment for the Robots to sense. This will also be shown by the Viewer.

Because sensing is cell-based, images will be scaled to the size of the World's grid. If the aspect ratio does not match, images will be stretched. To avoid any surprises from rescaling, we recommend using an image with the same resolution as your grid size. (e.g., if you have a 50x50 grid, use a 50px x 50px image.)

Parameters **img_filename** (*str*) – Filename of the RGB image to use as a background environment. Any transparency (alpha) is ignored by the robot sensing.

add_robot(*robot:* gridsim.robot.Robot)

Add a single robot to the World. Robots can also be added in bulk (as a list) when the `World` is created, using the `robots` keyword.

Parameters **robot** (*Robot*) – Robot to add to the World

get_dimensions() → Tuple[int, int]

Get the dimensions (in grid cells) of the World

Returns (width, height) of the World, in grid cells

Return type Tuple[int, int]

get_robots() → pygame.sprite.Group
Get a list of all the robots in the World

Returns All Robots currently in the World

Return type pygame.sprite.Group

get_time() → float
Get the current time of the World. At the moment, that's just the number of ticks (time steps) since the simulation started, since we're in a discrete world.

Returns Number of ticks (steps) since simulation started

Return type float

step()
Run a single step of the simulation. This moves the robots, manages the clock, and runs the robot controllers.

1.2.2 Robots

Gridsim provides two levels of abstract robot classes. The first, *Robot*, is designed to allow a user full control over their robot platform, specifying to communication criteria and allowed movements.

To get started faster, *GridRobot* implements a simple movement protocol and communication criterion, allowing the user to quickly start implementing their own code on the *GridRobot* platform.

For details on extending the Robot classes to create your own, see *Make your own Robot*.

class gridsim.robot.**Robot**(*x: int*, *y: int*)
Base class for all robot classes

__init__(*x: int*, *y: int*)
Abstract robot base class for all Robots

Parameters

- **x** (*int*) – Starting x position (grid cell) of the robot
- **y** (*int*) – Starting y position (grid cell) of the robot

abstract comm_criteria(*dist: int*) → bool
Criterion for whether message can be communicated (base on distance)

Parameters **dist** (*int*) – Distance between this robot and the other robot

Returns Whether or not the other robot is within communication range

Return type bool

distance(*pos: Tuple[int, int]*) → float
Get the Euclidean distance (in grid cells) between this robot and the specified (x, y) grid cell position.

If you want to change the distance metric (e.g., use Manhattan distance instead), you can override this method when you extend the Robot class.

Parameters **pos** (*Tuple[int, int]*) – (x, y) grid cell coordinate to get the distance to

Returns Euclidean distance of this robot from the given coordinate

Return type float

get_pos() → Tuple[int, int]
Get the position of the robot in the grid

Returns (x, y) grid position of the robot, from the top left

Return type Tuple[int, int]

get_tick() → int
Get the current tick of the robot (how many steps since the simulation started).

Returns Number of ticks since start of simulation

Return type int

get_tx_message() → Message
Get the message queued for transmission (broadcast).

The message is set by the *set_tx_message* function

Returns Message to continuously broadcast

Return type *Message*

get_world_dim() → Tuple[int, int]
Get the dimensions of the World that this Robot is in, so it can plan to avoid hitting the boundaries.

Returns (width, height) dimensions of the world, in grid cells

Return type Tuple[int, int]

Raises **ValueError** – Cannot get dimensions if Robot is not in a World. Add it during creation of a World or with *add_robot()*.

id: int = None
Unique ID of the Robot

abstract init()
Robot-specific initialization that will be run when the robot is set up

abstract loop()
User-implemented loop operation (code the robot runs every loop)

abstract move() → Tuple[int, int]
User-facing move command, essentially sending a request to move to a particular cell.

The robot will only make this move if it doesn't violate any movement conditions (such as edge of arena or, if enabled, collisions with other robots). Therefore, you do NOT need to implement any collision or edge-of-arena detection in this function.

Returns (x, y) grid cell position the robot intends to move to

Return type Tuple[int, int]

msg_received()
This is called when a robot successfully sent its message (i.e., when another robot received its message.)

By default, this does nothing. You can override it in your robot class to execute some operation or set a flag when a message is sent.

abstract receive_msg(*msg: Message*, *dist: float*)
Function called when the robot receives a message. This allows the specific robot implementation to choose how to process the messages that it receives, asynchronously.

Parameters

- **msg** (*Message*) – Received message from another robot
- **dist** (*float*) – Distance of the sending robot from this robot

sample(*pos: Optional[Tuple[int, int]] = None*) → Tuple[int, int, int]
Sample the RGB environment at the given cell location, or (if no `pos` given) and the robot's current position.

This allows you to sample *any* location in the World, but this is **probably cheating**. The robot platform you're modeling likely doesn't have such extensive sensing capabilities. This function is provided so that you can define any custom sensing capabilities (such as within a radius around your robot, or a line of sight sensor).

Parameters **pos** (*Optional[Tuple[int, int]]*) – (x, y) grid cell position of the World to sample. If not specified, the current robot position is sampled.

Returns (red, green, blue) color at the given coordinate in the range [0, 255]. If the world does not have an environment set, this will return (0, 0, 0)

Return type Tuple

set_color(*r: int*, *g: int*, *b: int*)
Set the color of the robot (as shown in Viewer) with 8-bit RGB values

Parameters

- **r** (*int*) – Red channel [0, 255]
- **g** (*int*) – Green channel [0, 255]
- **b** (*int*) – Blue channel [0, 255]

Raises **ValueError** – If all values are not in the range [0, 255]

set_tx_message(*msg: Message*)
Set the message that will be continuously broadcast

Parameters **msg** (Message) – Message to send to anyone within range

class gridsim.grid_robot.**GridRobot**(*x: int*, *y: int*, *comm_range: float = 5*)

__init__(*x: int*, *y: int*, *comm_range: float = 5*)
Create a robot that moves along the cardinal directions. Optionally, you can specify a communication range for the robots.

Parameters

- **x** (*int*) – Starting x position (grid cell) of the robot
- **y** (*int*) – Starting y position (grid cell) of the robot
- **comm_range** (*float, optional*) – Communication radius (in grid cells) of the robot, by default 5

comm_criteria(*dist: int*) → bool
Robots can communicate if their Euclidean distance is <= the radius specified at initialization (by default, 5 cells)

Parameters **dist** (*int*) – Euclidean distance of the other robot with which to communicate

Returns Whether distance is <= the communication radius

Return type bool

move() → Tuple[int, int]
Determine the cell the Robot will move to, based on the direction set in by `set_motors()`.

Returns (x,y) grid cell the robot will move to, if possible/allowed

Return type Tuple[int, int]

set_direction(*dir: int*)

Helper function to set the direction the robot will move. Note that this will persist (the robot will keep moving) until the direction is changed.

Parameters **dir** (*int*) – Direction to move, one of UP, DOWN, LEFT, RIGHT, or STAY

Raises **ValueError** – If given direction is not one of UP, DOWN, LEFT, RIGHT, or STAY

1.2.3 Viewer

The Viewer is a simple way to visualize your simulations. After creating the Viewer, just call *draw()* each step (or less frequently) to see the current state of the World.

Note: The maximum Viewer refresh rate (set at creation with the `display_rate` argument) also limits the simulation rate. If you want to run faster/higher-throughput simulations, don't use the Viewer.

---

class gridsim.viewer.**Viewer**(*world: gridsim.world.World*, *window_width: int = 1080*, *display_rate: int = 10*, *show_grid: bool = False*)

__init__(*world: gridsim.world.World*, *window_width: int = 1080*, *display_rate: int = 10*, *show_grid: bool = False*)

Create a Viewer to display the simulation of a World.

This is optional (for debugging and visualization); simulations can be run much faster if the Viewer is not used.

Parameters

- **world** (*World*) – World to display
- **window_width** (*int, optional*) – Width (in pixels) of the window to display the World, by default 1080
- **display_rate** (*int, optional*) – How fast to update the view (ticks/s), by default 10. In each tick, robots will move by one cell, so keep this low to be able to interpret what's going on.
- **show_grid** (*bool, optional*) – Whether to show the underlying grid in the World, by default False.

draw()

Draw all of the robots in the world into the World and its environment.

1.2.4 Configuration Parser

The `ConfigParser` is an optional class to help separate your code for experimental configurations by using YAML files for configuration. This imposes very few restrictions on the way you set up your configuration files; it mostly makes it easier to access their contents and save the configuration parameters with your data using the *Logger*.

This is useful for managing both values that are fixed through all experiments (e.g., dimensions of the arena) and experimental values that vary between conditions (e.g., number of robots). The latter may be saved as an array and a single value used for different conditions.

While the `ConfigParser` can load any valid YAML files, the largest restriction is what configuration parameter types can be saved to log files. For details, see the *log_config()* documentation.

class gridsim.config_parser.**ConfigParser**(*config_filename: str*)

Class to handle YAML configuration files.

This can be directly passed to the *log_config()* to save all configuration values with the trial data.

__init__(*config_filename: str*)

Create a configuration parser to manage all of the parameters in a YAML configuration file.

Parameters **config_filename** (*str*) – Location and filename of the YAML config file

get(*key: Optional[str] = None*, *default: Any = None*) → Any

Get a parameter value from the configuration, or get a dictionary of the parameters if no parameter name (key) is specified.

Parameters

- **key** (*Optional[str], optional*) – Name of the parameter to retrieve, by default None. If not specified, a dictionary of all parameters will be returned.
- **default** (*Any, optional*) – Default value to return if the key is not found in the configuration, by default None

Returns Parameter value for the given key, or the default value is the key is not found. If no key is given, a dictionary of all parameters is returned.

Return type Any

1.2.5 Logger

The logger provides an interface for easily saving time series data from many simulation trials, along with the parameters used for the simulation.

Data is logged in HDF5 (Hierarchical Data Format) files.

Data is stored by trial, in a hierarchy like a file structure, as shown below. Values in < > are determined by what you actually log, but the `params` group and `time` dataset are always created.

```
log_file.h5
├── trial_<1>
│   ├── params
│   │   ├── <param_1>
│   │   ├── <param_2>
│   │   ├──
│   │   └── <param_n>
│   ├── time
│   ├── <aggregator_1>
│   ├── <aggregator_2>
│   ├──
│   └── <aggregator_n>
├── trial_<2>
│   └──
├──
└── trial_<n>
```

All values logged with *log_param()* and *log_config()* are saved in `params`.

Time series data is stored in datasets directly under the `trial_<n>` group. They are created by *add_aggregator()*, and new values are added by *log_state()*. Calling this method also adds a value to the `time` dataset, which corresponds to the *World* time at which the state was saved.

class gridsim.logger.**Logger**(*world: gridsim.world.World*, *filename: str*, *trial_num: int*, *overwrite_trials: Optional[bool] = False*)

__init__(*world: gridsim.world.World*, *filename: str*, *trial_num: int*, *overwrite_trials: Optional[bool] = False*)

Create a Logger to save data to an HDF5 file, from a single simulation trial.

Note that this only creates the Logger with which you can save data. You must use the methods below to actually save anything to the file with the Logger.

Parameters

- **world** (*World*) – World whose simulation data you want to save.
- **filename** (*str*) – Name of the HDF5 file to save data to (`.hdf` extension). If the file does not exist, it will be created. If it does exist, it will be appended to (with the overwriting caveat specified below)
- **trial_num** (*int*) – Trial number under which to save the data.
- **overwrite_trials** (*Optional[bool], optional*) – Whether to overwrite a trial's data if it already exists, by default False

add_aggregator(*name: str*, *func: Callable[[List[gridsim.robot.Robot]], numpy.ndarray]*)

Add an aggregator function that will map from the list of all Robots in the world to a 1D array of floats. This will be used for logging the state of the World; the output of the aggregator is one row in the HDF5 Dataset named with the `name`.

The function reduces the state of the Robots to a single or multiple values. It could map to one float per robot (such as a state variable of each Robot) or a single value (length 1 array, such as an average value over all Robots).

Because of Python's dynamic typing, this does not validate whether the subclass of Robot has any parameters or functions that are called by the aggregator. The user is responsible for adding any necessary checks in the aggregator function.

Notes

The width of the aggregator table is set when this function is called, which is determined by the length of the output of `func`. If the length depends on the number of Robots, all Robots should be added to the `World` *before* adding any aggregators to the `Logger`.

The aggregator `func` will be applied to all robots in the world, regardless of type. However, if you have multiple types of Robots in your `World`, you can make an aggregator that applies to one type by filtering the robots by type within the `func`.

Parameters

- **name** (*str*) – Key that will be used to identify the aggregator results in the HDF5 log file.
- **func** (*Callable[[List[Robot]], np.ndarray]*) – Function that maps from a list of Robots to a 1D array to log some state of the Robots at the current time.

get_trial() → int

Get the trial number that this Logger is logging

Returns Number of the current trial being logged

Return type int

log_config(*config:* gridsim.config_parser.ConfigParser, *exclude: List[str] = []*)
Save all of the parameters in the configuration.

Notes

Due to HDF5 limitations (and my own laziness), only the following datatypes can be saved in the HDF5 parameters:

- string
- integer
- float
- boolean
- list of integers and/or floats

Parameters

- **config** (*ConfigParser*) – Configuration loaded from a YAML file.
- **exclude** (*List[str], optional*) – Names (keys) of any configuration parameters to exclude from the saved parameters. This can be useful for excluding an array of values that vary by condition, and you want to only include the single value used in this instance.

log_param(*name: str*, *val: Union[str, int, float, bool, list]*)
Save a single parameter value. This is useful for saving fixed parameters that are not part of your configuration file, and therefore not saved with *log_config()*.

This has the same type restrictions for values as *log_config()*.

Parameters

- **name** (*str*) – Name/key of the parameter value to save
- **val** (*Any*) – Value of the parameter to save. This can be a

log_state()
Save the output of all of the aggregator functions. If you have not added any aggregators with *log_state()*, nothing will be saved by this function.

The runs each previously-added aggregator function and appends the result to the respective HDF5 Dataset. It also saves the current time of the World to the `time` Dataset.

1.2.6 Messages

This provides a basic Message protocol for robot communication. Each message contains the ID of the sender and a dictionary of message contents. The values of the message contents may be any type, so the receiver must know how to process the data.

Additionally, Messages can optionally include a receiver type (`rx_type`). This is only needed if there are multiple types of robots in the World, and you only want certain types of robots to receive the message.

If no arguments are provided when a Message is created, it creates a null message, which signals that the robot is not broadcasting anything.

While it is possible to extend this class, the default Message class should meet most needs.

class gridsim.message.**Message**(*tx_id: Optional[int] = None*, *content: Optional[Dict[str, Any]] = None*, *rx_type: Type[gridsim.robot.Robot] = <class 'gridsim.robot.Robot'>*)

__init__(*tx_id: Optional[int] = None*, *content: Optional[Dict[str, Any]] = None*, *rx_type: Type[gridsim.robot.Robot] = <class 'gridsim.robot.Robot'>*)

A message sent by robots. Can be either a null (empty) message if no arguments are provided to the constructor. Or it contains the sender's ID, a dictionary of content, and (optionally) the type of robot that receives the message.

Parameters

- **tx_id** (*Optional[int], optional*) – ID of the sending (transmitting) robot, by default None
- **content** (*Optional[Dict[str, Any]], optional*) – Dictionary of message keys and values, by default None. Keys must be strings, but values can be of any type (incumbent on receiver for interpretation)
- **rx_type** (*Type[Robot], optional*) – Type of the receiving robot, by default Robot (i.e., message will be processed by any Robot.)

get() → Optional[Dict[str, Any]]

Get the contents of the message

Returns Dictionary of the message contents

Return type Optional[Dict[str, Any]]

tx_id() → Optional[int]

Get the ID (32-bit integer) of the robot that sent the message

Returns ID of the sending (transmitting) robot

Return type Optional[int]

1.3 Development

This is reference material for local development.

If you just want to use the library, you don't need any of this.

1.3.1 Build Documentation

from the `docs` directory, run:

```
make html
```

Then open the documentation:

```
open _build/html/index.html
```

1.3.2 Build the distributable for PyPi

(From the PyPi tutorial)

Make sure the necessary dependencies are installed.

```
pip3 install --upgrade setuptools wheel twine
```

Build the project. From the project root folder, run:

```
python3 setup.py sdist bdist_wheel
```

Upload it to the testing index:

```
python3 -m twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

Upload it to the actual index:

```
python3 -m twine upload dist/*
```

CHAPTER

TWO

ABOUT

Gridsim is a Python 3 library for simulating robots in a grid-based world. It has a simple, well-documented API, making it easy to implement your own algorithms with minimal overhead.

Key features include:

- Viewer for debugging and visualizing your simulations
- Built-in data logging to HDF5 files
- Support for YAML configuration files
- Extendable robot classes to avoid repeating your code
- Comprehensive documentation and examples

CHAPTER

THREE

QUICK INSTALL

```
$ pip install gridsim
```

For more information and instructions, check out the documentation.

CHAPTER

FOUR

LINKS

Documentation: Read the Docs

PyPi: gridsim

Source code: Github

CHAPTER

FIVE

CONTACT

If you have questions, or if you've done something interesting with this package, send me an email: julia@juliaebert.com.

If you find a problem or want something added to the library, open an issue on Github.

INDEX

Symbols

A

C

D

G

I

L

M

R

S

T

V

W