Grid Robot Simulator and Client

Robot Simulator and Application Programming Interface (API)

M L Walters, V1.0 Nov 2015

1 Introduction

The Robot Grid World Simulator is in two parts: A Simulator (RobotGridWorld.pyw) which simulates a simple grid square based world, and also a Client robot control programmer'slibrary (grobot.py), which enables the creation and control of one or more robots within the simple grid world simulator. Before trying to run or use either program file, they must first be extracted from the compressed .ZIP Archive! To do this, right click on the “RobotGridWorldSimulator” Zip Archive folder, as downloaded from Studynet, and select “Extract”. You can then follow the prompts to select and extract all files to your desired Python Program working folder (directory).

(Note: Both programs will run using either Python 2.6+ or 3.2+)

2 Grid Robot Simulator (RobotGridWorld.py)

To run the simulator, once extracted from the zip archive, simply double click on the program icon from your favourite file manager (Linux) or File Explorer (Windows or Mac).

The simulator program will initially display a blank map, which is a 30x30 grid of squares, with x and y co-ordinates marked along the lower and left sides respectively.
There are several buttons available categorised as:

Map Editing Controls:

"New Map" - Clears the current Map ready for editng a new one
"Load Map" - Loads a previously saved map
"Save Map" - Saves the current (edited) map, for later loading

Clicking anywhere on the map grid area will toggle the building and removal of walls on the map. When the pattern of walls is as desired, the map can be saved (click "Save Map" to a file (extension <name>.map). A previously “Saved” map can be recalled by selecting "Load Map". There should be several example maps extracted to your working folder (directory) along with the RobotGridWorld.pyw program file

Simulation Controls:

"Toggle Trails" - Toggles the display of appropriately coloured trails for each robot
"Speed" - A slider control to adjust the speed of the simulated robot(s) movements.

3 Client program (library) for the RobotGridWorld Simulator (grobot.py)

A client API for the RobotGridWorld Simulator - grobot.py
M L Walters. V0.5 Nov 2015
M L Walters, V1.0 Nov 2015
M L Walters, V1.2 Dec 2015

Note, this module (grobot.py – Python program file) must either be placed in the current working directory (folder) or in a suitable directory in the Python standard search path. When using IDLE, simply Open the program to set the correct Current Working Directory, even if you plan only to use the interactive command shell with grobot.py.

When imported into your own Python robot control program (or interactive Python Command Shell) grobot.py provides a simple Application Programming Interface (API) for the RobotGridWorld.py robot Simulator. This must be already running when you run your grobot program otherwise an error message will be displayed, and your grobot program will terminate.

****
Creating RobotGrid World Robot Programs
****

At the beginning of your robot program (or interactive Python Shell session) you must first import the grobot.py client into your own program. Usage:

```
import grobot
```

To create a new robot in the simulated world, use:

```
robotname=grobot.NewRobot(<name>, <xpos>, <ypos>, <colour>, <shape>)
```

Where:

<name> is a string which contains the name of the robot. If several robots are used, all must have unique names.

Integers <xpos> and <ypos> (both between 0 and 30) are the initial or starting world co-ordinate position of the robot. If left blank, the robot will be set to the default starting position x = 1, y = 1.

<colour> is a string which denotes which colour the robot will be. The common colours are ; "red", "blue", "green", "yellow", "black". If the colour parameter is not provided, the robot will be red.

<shape> is a string that denotes one of the standard turtle shapes. Normally, leave this blank if the standard robot shape is to be used. See the Python Turtle docs for more details.

The parameters for this function are all optional, but if several robots are to be simulated simultaneously, the <name> string parameter must be supplied, and the

initial positions (xpos, and ypos) for each robot must be different. E.g.:

```
fred = grobot.NewRobot("fred", 1, 2, "blue")
```

If the RobotGridWorld simulator is not running when you execute your grobot program, an error message will be displayed, and the program will exit. Once a robot is created (instantiated), the robot object has five methods available. For example, if the robot has been instantiated as fred:

`fred.forward()`
Moves robot orthogonally forward one grid square. If the way is blocked returns "Bang" and the robot will turn into a black circle (“Broken”), otherwise returns "OK". If the robot has already returned "Bang", subsequent calls to forward() will return "Broken".

`fred.right()`
The robot will turn 90 degrees right, returning "OK", or "Broken" if a collision has occurred.

`fred.left()`
The robot will turn 90 degrees left, returning "OK", or "Broken" if a collision has occurred.

`fred.look()`
Returns a list of 5 elements:
[viewLeft, viewDiagLeft, viewForward, viewDiagRight, viewRight]
Each element can be either "Wall" if the forward way is blocked by a Wall, a name of the another robot blocking the way, or None if the way is clear. If a collision has previously occured, this will return a list of ["Broken"]*5.

`fred.init(x,y)`
Resets and initialises the named robot to the start position (x, y). Note. x and y are optional and if not provided the robot will use the last provided values from a previous `init()` or `NewRobot()`.

There is one function in the module availalble: `grobot.demo()`
This provides a short demonstration of how to create a NewRobot() and use the methods described above. The demo is run automatically if the module is loaded and/or run directly (i.e. not imported):

```
def demo():
    # print used to show return value from method/function calls
    fred=NewRobot("fred", 1, 1)
    bill=NewRobot("bill", 1, 1, "green")
    print("Fred forward", fred.forward())
    print("Bill forward",bill.forward())
    print("Fred right", fred.right())
    print("Bill right", bill.right())
    #fred.init(7,7) # New start position
    count = 12
    while count > 0:
        #print(count)
        print("Fred looks at:", fred.look())
        print("Fred forward",fred.forward())
        print("Bill looks at:", bill.look())
        print ("Bill forward",bill.forward())
        count -= 1

    print("Fred looks forward at", fred.look()[2]) # element 2 is forward view
```

```
    print("Bill looks forward at", bill.look()[2])


Note, if you load either the "Demo.map" or Test.map world into the simulator,
the grobot.demo() will demonstrate the options and return values from the robot
methods (functions).

You can also import and use the grobot module (program)
from the interactive shell and try out the commands interactively, e.g.:

>>>import grobot
>>>grobot.demo()
>>>fred = grobot.NewRobot("fred", 5, 5, "red")
>>>fred.look()
"wall"
>>>fred.forward()
"Bang"
>>>fred.init()
"OK"
>>>

You can also display this guide by using Python's interactive help function:

>>>help("grobot")
```

4 GridRobot (grobot) Programming Challenges:

Here are some (grid world) robot programming challenges ordered in increasing difficulty which will incrementally develop your ability to write Python programs that control and receive information from the simulated Grid World Robot (grobot). These can all be written as console type programs (using print() and input() functions for user interaction etc.).

Advanced Programmers – may want to create GUI type programs using the Python tkinter library to create nice looking graphics, text boxes/labels to display messages/options etc., and also allow the user to select options using mouse and/or keyboard events.

4.1 Challenge: A simple user interface program for the grobot.

The grobot.py library API allows experienced programmers to create and control a simulated robot in the simulated GridWorld. However, for testing, and also non-programmers, it is useful to have a simple interface to allow the simulated robot to be remotely driven around the GridWorld. The Grobot only has three movements: forward, turn 90 degrees left, turn 90 degrees right, so your program should allow the user to easily select one of these movements, then once the movement has completed, display any (error) messages, then loop allowing the next robot command to be selected.

Hint: Look at the demo() function in grobot for examples of how to create a new robot, and control it using the robot methods available.

Advanced Programmers Only – Using the Python tkinter library, you can create a simple GUI, with direction buttons, and also trap key presses to present a user-friendly interface to the user.

4.2 Challenge: A program that keeps track of the location and heading of the grobot.

A useful feature for any robot is to know where it is – that is, its location or position in a given environment or working area. Bear in mind that any location will normally be relative to some base position or location. In our GridWorld simulation, all locations (individual grid squares) can be represented by two integer number coordinates along the horizontal and vertical axes (x and y), which are relative to the x=0, y=0 grid square. This is known as robot 'Localisation' (or Localization in US English!). There are two ways by which a robot can localise itself, and for most real robots a mixture of the two methods is used. The robots current location is usually specified as relative to a known "datum" position, which is usually with respect to a pre-stored or pre-learned map of the environment. One method to do this is for the robot to track internally of any movements it makes, and by counting the numbers of commanded turns and movements, infers the current position. This method is called 'Odometery' and literally means "route + measure".

Unfortunately, our simulated Grobot does not have a "built-in" method to keep track of its current position, so the next robot programming task problem is to implement a program that uses odometry to do this automatically. To test your program, you will also need to include (or modify) your first remote control program to drive your robot around in the GridWorld simulator with any of the supplied .map GridWorlds (or create your own?). Your program should display the updated x, y positions and current heading to the user at every move.

Hints: When you initialise a NewRobot, you can supply an initial known start position to the robot, which can be saved in suitable variables. To keep track of the robots position, you will need to create and update these two variables, and another for heading direction every time it makes a move. The heading variable should keep track of the new heading (North, South, East or West), and whenever the robot moves forward one square, the two x, y variables should be incremented or decremented appropriately, depending on the current heading direction. A neat way to do this is to write simple "wrapper" functions for the robots movements (robot.turnLeft(), robot.turnRight() and robot.Forward()) that also update global x, y and heading variables as required.

Advanced Programmers Only - can create a new class definition that inherits from the grobot.NewRobot class with new methods which wrap the existing movement methods and has attributes that keep track of x, y and heading.

[Note: Another method for robot localisation is to use sensor information from the environment (I.e the world) to recognise its location using a combination of visual recognition, distance sensing and/or 'Beacons' (light, infra-red, rfid etc.) with known positions. This is a difficult problem for real robots and the subject of much current robotics research. Where the robot "wakes up" in an unfamiliar environment, then the problem becomes greater as a map model must be constructed as well as keeping track of the robots position. This is known as SLAM (Simultaneous Localisation And Mapping).]

4.3 Challenge: A wandering or exploration program for the grobot

A useful behaviour for a robot is to exhibit wandering. Your next task is therefore to program a wandering behaviour for the grobot that allows the robot to explore as many of the unoccupied grid

squares of the GridWorld as possible. You can load any of the supplied .map GridWorlds (or create your own?) to test your program.

Hints: The robot should move forward autonomously, unless the way is blocked by an obstacle. If an obstacle is sensed in front, the robot should turn and move away in a free direction. For these types of “reactive” behaviours, it is common that the robot may get locked into a repetitive movement cycle, where it repeats the same sequence of moves over an over again. Therefore, it is usual to program random movements into the control strategy to break out of any repetitive cycles. For example: If an obstacle is sensed in front, and it is clear to both left and right of the robot, then a the turn direction may be selected at random. Also, even if the way ahead is clear, random turns (at random intervals?) may be generated to make sure that the robot eventually explores all reachable parts of the GridWorld. The Python random library has a number of useful functions for incorporating randomness in programs.

4.4 Challenge: A program for the grobot to navigate through a maze

A good test of a robot's (and the robot programer's) capabilities is to navigate the robot through a maze. Solving a maze problem typically involves either navigating successfully from an entry point to an exit point, or to navigate to a goal position within the maze, and returning back to the entry point. In the case of a GridWorld maze, for both cases, the robot needs to be able to recognise when it has reached a target grid square, which may be either the maze exit or a given goal. You should develop a maze navigation program that allows you to enter the desired exit grid square coordinates (x, y). Then the robot should navigate autonomously to the exit grid square. Use the supplied Maze.map world to test your program with entry point x=1, y=2 and exit point x=1, y=15.

Hints: A simple algorithm for solving mazes is the well known “Left hand rule” - keep following the wall on the robot's left (or right) side. For many maze types, this will eventually find a different exit from the maze, or if there is no second exit found will return to the entry point. Your program needs to keep track of the robots current (x, y) grid square position, so it recognises when the maze exit point is reached (i.e the program has competed successfully!).

Advanced Programmers Only – change your program to accept any (unblocked) starting and goal square x and y coordinates within a closed maze (MazeClosed.map), so that the robot can navigate from any given start square within the maze to any given goal square. A good starting point for information about maze solving algorithms can be found on Wikipedia: https://en.wikipedia.org/wiki/Maze_solving_algorithm

4.5 Challenge: Using the grobot to create its own internal map model of the current GridWorld

Although you can load and edit your own maps for the GridWorld simulator, your robot cannot access these directly, as it only has a “local” view of its immediate front and side adjacent grid squares. It is a useful capability for any robot to explore and create its own (internal) map or model of the wider world. You should develop a program that controls the robot to autonomously explore the GridWorld, simultaneously storing the location of any obstacles (walls) encountered in the internal map model representation. As the robot explores the GridWorld, your program should visually display the mapping progress at regular intervals.

Hints: You can use the wandering/exploration behaviour developed for the previous program to make the robot explore the supplied Maze.map world. Use a 2D List (30x columns, 30 x rows) to store the state of each grid square encountered. Use the current x and y position of the robot, in conjunction with the current heading to calculate the actual location of any sensed obstacles, then use these x,y values as indexes to the 2D list. To display the map to the console, use combinations of | and - to draw the grid squares, # to indicate a wall, space for clear and R for current robot location and heading. E.g.:

```
-------------------------------------------------
|#|#|#|#|#|#|#|#|#|#| |#|#|#|#|
-------------------------------------------------
|#|   |   |   |   |   |   |R|# |   |#|   |#|
–-----------------------------------------------
|#|#|#|#|#|#|#|   |#|# |   |#|#|#|
-------------------------------------------------  etc.
```

Advanced Programmers Only – use the graphics capabilities of the Python tkinter library to visually display the robots map building progress on a tk canvas. Also provide the option to save (and load) a grobot learned map to a file using comma or tab delimited formatting.