Problem Set 6: Simulating Robots

Handed out: Lec 12
Installation due: 11:59pm, Lec 13. *(No late days can be gained or used for this part of the assignment)*
Due: 11:59pm, Lec 14.

Introduction

In this problem set you will practice designing a simulation and implementing a program that uses classes.

Please don't be discouraged by the apparent length of this problem set. There is quite a bit to read and understand, but most of the problems do not involve writing much code.

Getting Started

Download Files: from ps6.zip

Installing pylab

To create plots for Problems #4 and #6 of this problem set, you will need these Python library packages: matplotlib and numpy.

OS X

- Download and install the following python modules: mac-install http://mit600.mit.edu/blog/wp-content/uploads/2011/03/mac-install.zip
- matplotlib 1.0.0
- numpy 1.5.0

Windows

- Follow the below links, download, and install the modules:
- Matplotlib 1.0.0 matplotlib 1.0.0
- numpy 1.5.0

Make sure the libraries that you download match the version of Python that you are using and the platform (i.e. Windows, Mac, Linux).

To test that you have successfully installed matplotlib and numpy, run the code provided in `ps6_pkgtest.py`, and upload the generated graph as `ps6_pylabtest.pdf` to your workspace by **11:59pm Lecture 13**. If you have problems with installation, be sure to go to office hours for help.

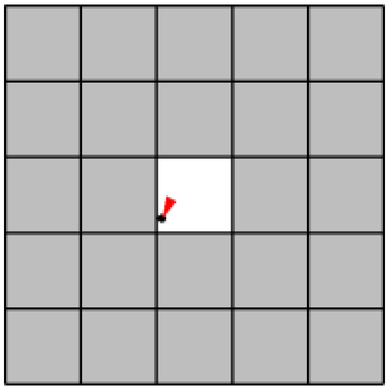
Hint: You can use the save button on the pylab figure to save as .pdf.

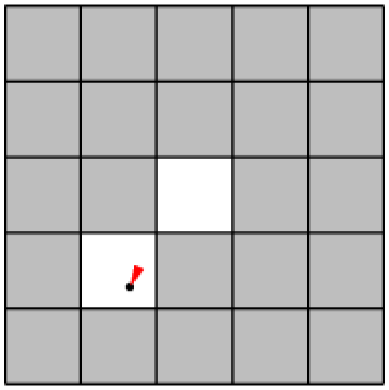
Simulation Overview

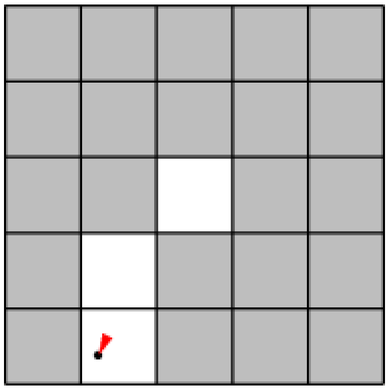
iRobot is a company (started by MIT alumni and faculty) that sells the Roomba vacuuming robot (watch one of the product videos to see these robots in action). Roomba robots move about a floor, cleaning the area they pass over. You will design a simulation to estimate how much time a group of Roomba-like robots will take to clean the floor of a room.

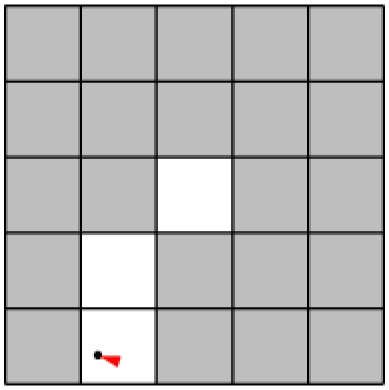
The following simplified model of a single robot moving in a square 5×5 room should give you some intuition about the system we are simulating.

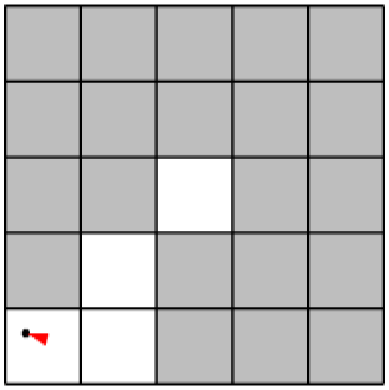
The robot starts out at some random position in the room, and with a random direction of motion. The illustrations below show the robot's position (indicated by a black dot) as well as its direction (indicated by the direction of the red arrowhead).


Time *t* = 0: The robot starts at the position (2.1, 2.2) with an angle of 205 degrees (measured clockwise from "north"). The tile that it is on is now clean.


***t* = 1**: The robot has moved 1 unit in the direction it was facing, to the position (1.7, 1.3), cleaning another tile.


***t* = 2**: The robot has moved 1 unit in the same direction (205 degrees from north), to the position (1.2, 0.4), cleaning another tile.


***t* = 3**: The robot could not have moved another unit in the same direction without hitting the wall, so instead it turns to face in a new, random direction, 287 degrees.


***t* = 4**: The robot moves along its new heading to the position (0.3, 0.7), cleaning another tile.

Simulation Details

Here are additional details about the simulation model. Read these carefully.

- **Multiple robots**. In general, there are $N > 0$ robots in the room, where N is given. For simplicity, assume that robots are points and can pass through each other or occupy the same point without interfering.
- **The room**. The room is rectangular with some integer width w and height h, which are given. Initially the entire floor is dirty. A robot cannot pass through the walls of the room. A robot may not move to a point outside the room.
- **Robot motion rules**:
    - Each robot has a position inside the room. We'll represent the position using coordinates (x, y) which are floats satisfying $0 \leq x < w$ and $0 \leq y < h$. In our program we'll use instances of the `Position` class to store these coordinates.
    - A robot has a direction of motion. We'll represent the direction using an integer d satisfying $0 \leq d < 360$, which gives an angle in degrees.
    - All robots move at the same speed s, which is given and is constant throughout the simulation. Every time-step, a robot moves in its direction of motion by s units.
    - If a robot would've ended up hitting the wall within the time-step, it instead picks a new direction at random. The robot continues in that direction until it reaches another wall.
- **Tiles**. You will need to keep track of which parts of the floor have been cleaned by the robot(s). We will divide the area of the room into 1×1 tiles (there will be $w * h$ such tiles). When a robot's location is anywhere in a tile, we will consider the entire tile to be cleaned (as in the pictures above). By convention, we will refer to the tiles using ordered pairs of integers: $(0, 0), (0, 1), \ldots, (0, h\text{-}1), (1, 0), (1, 1), \ldots, (w\text{-}1, h\text{-}1)$.
- **Termination**. The simulation ends when a specified fraction of the tiles in the room have been cleaned.

If you find any places above where the specification of the simulation dynamics seems ambiguous, it is up to you to make a reasonable decision about how your program/model will behave, and document that decision in your code.

Part I: The Rectangular Room and Robot classes

You will need to design two classes to keep track of which parts of the room have been cleaned as well as the position and direction of each robot.

In ps6.py, we've provided skeletons for the following two classes, which you will fill in in Problem #1:

RectangularRoom
Represents the space to be cleaned and keeps track of which tiles have been cleaned.

Robot
Stores the position and heading of a robot.

We've also provided a complete implementation of the following class:

Position

Stores the *x*- and *y*-coordinates of a robot in a room.

Read `ps6.py` carefully before starting, so that you understand the provided code and its capabilities.

Problem #1

In this problem you will implement two classes.

For the RectangularRoom class, decide what fields you will use and decide how the following operations are to be performed:

- Initializing the object
- Marking an appropriate tile as cleaned when a robot moves to a given position
- Determining if a given tile has been cleaned
- Determining how many tiles there are in the room
- Determining how many cleaned tiles there are in the room
- Getting a random position in the room
- Determining if a given position is in the room

For the Robot class, decide what fields you will use and decide how the following operations are to be performed:

- Initializing the object
- Accessing the robot's position
- Accessing the robot's direction
- Setting the robot's position
- Setting the robot's direction

Complete the RectangularRoom and Robot classes by implementing their methods in `ps6.py`.

(Although this problem has many parts, it should not take long once you have chosen how you wish to represent your data. For reasonable representations, *a majority of the methods will require only one line of code*.)

For your reference, here are **abbreviated** specifications for the methods of RectangularRoom and Robot. See ps6.py for complete details.

```
class RectangularRoom(object):
    """
    A RectangularRoom represents a rectangular region containing clean or
dirty
    tiles.
    A room has a width and a height and contains (width * height) tiles. At
any
    particular time, each of these tiles is either clean or dirty.
    """
    def __init__(self, width, height):
        """
        Initializes a rectangular room with the specified width and height.

        Initially, no tiles in the room have been cleaned.
        """

    def cleanTileAtPosition(self, pos):
        """Mark the tile under the position POS as cleaned."""

    def isTileCleaned(self, m, n):
        """Return True if the tile (m, n) has been cleaned."""

    def getNumTiles(self):
        """Return the total number of tiles in the room."""

    def getNumCleanedTiles(self):
        """Return the total number of clean tiles in the room."""

    def getRandomPosition(self):
        """Return a random position inside the room."""

    def isPositionInRoom(self, pos):
        """Return True if POS is inside the room."""

class Robot(object):
    """
    Represents a robot cleaning a particular room.

    At all times the robot has a particular position and direction in the
room.
    The robot also has a fixed speed.

    Subclasses of Robot should provide movement strategies by implementing
    updatePositionAndClean(), which simulates a single time-step.
    """
    def __init__(self, room, speed):
        """
        Initializes a Robot with the given speed in the specified room. The
        robot initially has a random direction and a random position in the
        room. The robot cleans the tile it is on.
        """

    def getRobotPosition(self):
        """Return the position of the robot."""

    def getRobotDirection(self):
        """Return the direction of the robot."""

    def setRobotPosition(self, position):
        """Set the position of the robot."""
    def setRobotDirection(self, direction):
        """Set the direction of the robot."""

    def updatePositionAndClean(self):
        """Simulate the passage of a single time-step."""
```

Part II: Creating and using the simulator

Problem #2

Each robot must also have some code that tells it how to move about a room, which will go in a method called `updatePositionAndClean`.

Ordinarily we would consider putting all the robot's methods in a single class. However, later in this problem set we'll consider robots with alternate movement strategies, to be implemented as different classes with the same interface. These classes will have a different implementation of `updatePositionAndClean` but are for the most part the same as the original robots. Therefore, we'd like to use inheritance to reduce the amount of duplicated code.

We have already refactored the robot code for you into two classes: the Robot class you completed above (which contains general robot code), and a StandardRobot class inheriting from it (which contains its own movement strategy).

Complete the `updatePositionAndClean` method of `StandardRobot` to simulate the motion of the robot after a single time-step (as described above in the simulation dynamics).

```
class StandardRobot(Robot):
    """
    A StandardRobot is a Robot with the standard movement strategy.

    At each time-step, a StandardRobot attempts to move in its current
direction; when
    it hits a wall, it chooses a new direction randomly.
    """
    def updatePositionAndClean(self):
        """
        Simulate the passage of a single time-step.

        Move the robot to a new position and mark the tile it is on as having
        been cleaned.
        """
```

Problem #3

In this problem you will write code that runs a complete robot simulation.

Recall that in each trial, the objective is to determine how many time-steps are on average needed before a specified fraction of the room has been cleaned. **Implement the following function:**

```
def runSimulation(num_robots, speed, width, height, min_coverage, num_trials,
                  robot_type):
    """
    Runs NUM_TRIALS trials of the simulation and returns the mean number of
    time-steps needed to clean the fraction MIN_COVERAGE of the room.

    The simulation is run with NUM_ROBOTS robots of type ROBOT_TYPE, each
with
    speed SPEED, in a room of dimensions WIDTH x HEIGHT.
    """
```

The first six parameters should be self-explanatory. For the time being, you should pass in `StandardRobot` for the `robot_type` parameter, like so:

```
avg = runSimulation(10, 1.0, 15, 20, 0.8, 30, StandardRobot)
```

Then, in `runSimulation` you should use `robot_type(...)` instead of `StandardRobot(...)` whenever you wish to instantiate a robot. (This will allow us to easily adapt the simulation to run with different robot implementations, which you'll encounter in Problem #5.)

Feel free to write whatever helper functions you wish.

We have provided the `getNewPosition` method of `Position`, which you may find helpful:

```
class Position(object):


    ⋮


    def getNewPosition(self, angle, speed):
        """
        Computes and returns the new Position after a single clock-tick has
        passed, with this object as the current position, and with the
        specified angle and speed.

        Does NOT test whether the returned position fits inside the room.

        angle: float representing angle in degrees, 0 <= angle < 360
        speed: positive float representing speed

        Returns: a Position object representing the new position.
        """
```

For your reference, here are some approximate room cleaning times. These times are with a robot speed of 1.0.

- One robot takes around 150 clock ticks to completely clean a 5×5 room.
- One robot takes around 190 clock ticks to clean 75% of a 10×10 room.
- One robot takes around 310 clock ticks to clean 90% of a 10×10 room.
- One robot takes around 3250 clock ticks to completely clean a 20×20 room.

(These are only intended as guidelines. Depending on the exact details of your implementation, you may get times different from ours.)

You should also check your simulation's output for speeds other than 1.0. One way to do this is to take the above test cases, change the speeds, and make sure the results are sensible.

Visualizing robots (Optional, but cool and very easy to do. May also be useful for debugging. Comment out before turning in.)

We've provided some code to generate animations of your robots as they go about cleaning a room. These animations can also help you debug your simulation by helping you to visually determine when things are going wrong.

Download ps6_visualize.py and save it in the same directory as your ps6.py. Add the following line to the top of your ps6.py:

```
import ps6_visualize
```

Here's how to run the visualization:

1. In your simulation, at the beginning of a trial, do the following to start an animation:

   ```
   anim = ps6_visualize.RobotVisualization(num_robots, width, height)
   ```

   (Pass in parameters appropriate to the trial, of course.) This will open a new window to display the animation and draw a picture of the room.

2. Then, on *each time-step*, do the following to draw a new frame of the animation:

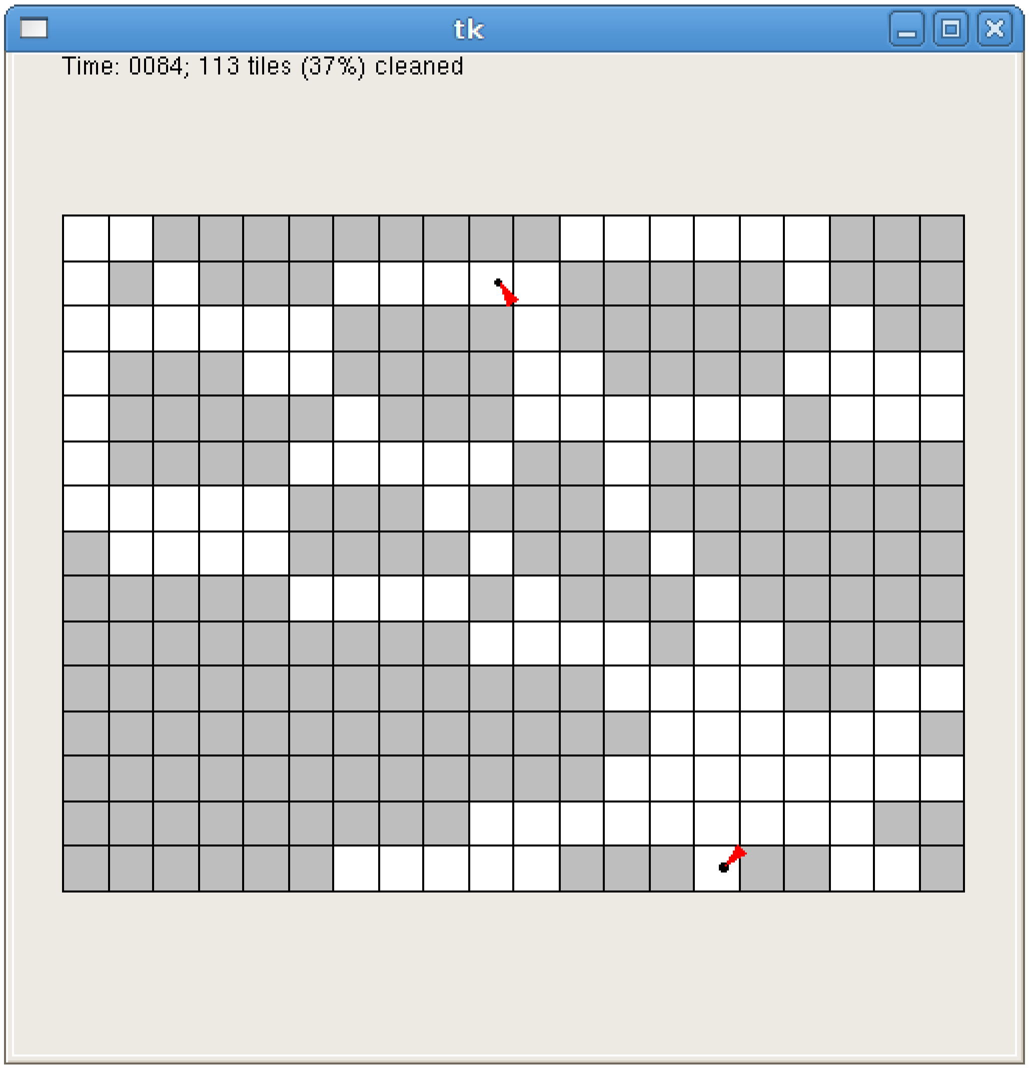
   ```
   anim.update(room, robots)
   ```

   Pass in a `RectangularRoom` object and a list of `Robot` objects representing the current state of the room and the robots in the room.

3. When the trial is over, call the following method:

   ```
   anim.done()
   ```

The resulting animation will look like this:



The visualization code slows down your simulation so that the animation doesn't zip by too fast (by default, it shows 5 time-steps every second). Naturally, you will want to avoid running the animation code if you are trying to run many trials at once (for example, when you are running the full simulation).

For purposes of debugging your simulation, you can slow down the animation even further. You can do this by changing the call to `RobotVisualization`, as follows:

```
anim = ps6_visualize.RobotVisualization(num_robots, width, height, delay)
```

The parameter `delay` specifies how many seconds the program should pause between frames. The default is 0.2 (that is, 5 frames per second). You can raise this value to make the animation slower.

For problems 4 and 6, you will want to make calls to `runSimulation()` to get simulation data and plot it. However, you don't want the visualization getting in the way. If you choose to do this visualization exercise, before you get started on problems 4 and 6 *and* before you turn your problem set in, **make sure to comment the visualization code out of `runSimulation()`.**

Problem #4

Now, use your simulation to answer some questions about the robots' performance.
Note: The instructions in the code for Problems 4-6 are slightly different than the instructions that follow here. *Follow the instructions in this write up!*

In order to do this problem, you will be using a Python tool called pylab. To learn more about pylab, please read this PyLab tutorial.

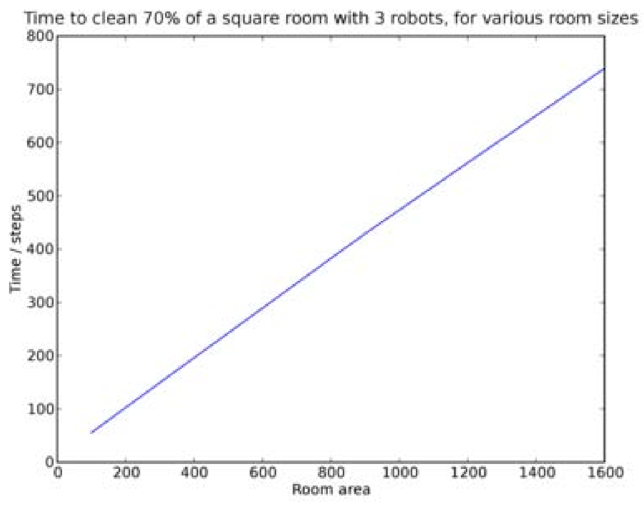
For both of the questions below, write code which will generate a plot using pylab. Put your code inside the corresponding skeleton functions in ps6.py (showPlot1 and showPlot2).

Each plot should have a title and descriptive labels on both axes.

1. How long does it take to clean 80% of a 20×20 room with each of 1-10 robots? Output a figure that plots the mean time (on the Y-axis) against the number of robots.
2. How long does it take two robots to clean 80% of rooms with dimensions 20×20, 25×16, 40×10, 50×8, 80×5, and 100×4? (Notice that the rooms have the same area.) Output a figure that plots the mean time (on the Y-axis) against the ratio of width to height.

Experiment with the number of trials. For your plots, use a number of trials which is large enough that you think the output is reliable.

Here is an example of a good plot:



As you can see, when keeping the number of robots fixed, the time it takes to clean a square room is basically proportional to the area of that room.

Problem #5

iRobot is testing out a new robot design. The proposed new robots differ in that they change direction randomly **after every time step**, rather than just when they run into walls. You have been asked to design a simulation to determine what effect, if any, this change has on room cleaning times.

Write a new class `RandomWalkRobot` that inherits from `Robot` (like `StandardRobot`) but implements the new movement strategy. `RandomWalkRobot` should have the same interface as `StandardRobot`.

Test out your new class. Perform a single trial with the new `RandomWalkRobot` implementation and watch the visualization to make sure it is doing the right thing. Once you are satisfied, you can call `runSimulation` again, passing `RandomWalkRobot` instead of `StandardRobot`.

Problem #6

Generate an appropriate plot (of your own design) that compares the performance of the two types of robots. Add your code to `showPlot3()`. As always, your plot should have an appropriate title, axis labels, and (if applicable) legend.

Within comments in `showPlot3`, comment briefly on how the two types of robots compare.

Hand-In Procedure

1. Save

Save your code in a single file, named **`ps6.py`**.

2. Test

Run your file to make sure it has no syntax errors. Run your plotting functions to make sure they produce plots when run. Test your `runSimulation` to make sure that it still works with **both** the `StandardRobot` and `RandomWalkRobot` classes. (It's common to accidentally break code while refactoring, which is one reason that testing is really important!)

3. Time and Collaboration Info

At the start of your file, in a comment, write down the number of hours (roughly) you spent on the problems, and the names of the people you collaborated with. For example:

```
# Problem Set 6
# Name: Jane Lee
# Collaborators: John Doe
# Time: 3:30
... your code goes here ...
```