

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.00 Introduction to Computer Science and Programming  
Fall 2008

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.

## 6.00: Introduction to Computer Science and Programming

# Problem Set 11: Simulating Robots

**Handed out:** Thursday, November 13, 2008.

**Due:** Thursday, November 20, 2008.

## Introduction

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

## Workload

Please let us know how long you spend on each problem. We want to be careful not to overload you by giving out problems that take longer than we anticipated.

In this pset, there is quite a bit to read and understand, but most of the problems do not involve writing much code.

## Collaboration

You may work with other students. However, each student should write up and hand in his or her assignment separately. *Be sure to indicate with whom you have worked.* For further detail, please review the collaboration policy as stated in the syllabus.

## Getting Started

Download the following files and save them in a single directory:

- ps11.py, the skeleton you will fill in
- ps11\_visualize.py, a nifty library we are providing (to be explained later)

## Installing pylab

You will need to install some libraries to create plots for Problems #4 and #6 of this problem set.

### Installing pylab on Windows

1. Check the version of Python on your machine (in IDLE, choose `Help->About IDLE`) and make sure you are running Python 2.5.
2. Download and install [numpy](#).
3. Download and install [Matplotlib](#).

On Windows Vista, you may get an error during Matplotlib installation that says something to the effect of "could not create key." You should run (or re-run) the installer by right-clicking on it and selecting "Run as administrator." (If you don't, Matplotlib will still work, but no entry will appear to allow you to uninstall it until you run the setup program as an administrator.)

### Installing pylab on Mac OS X

1. Check the version of Python on your machine (in IDLE, choose `Help->About IDLE`) and make sure you are running Python 2.5.
2. Download and install [python-dateutil](#), [pytz](#), and [numpy](#).
3. Download and install [Matplotlib](#).

## Using pylab on Athena

If you are working on Athena, find an machine that runs **Athena Linux** (that is, **not Solaris**— the Solaris machines are purple and say "Sun"). In a command prompt window, make sure that the shell you're using is a `bash` shell; you can check this by entering `echo $SHELL` and verifying you see a string

containing `bash` (and in particular not `csh`); if you don't, then run `bash`. Then run the following commands to launch IDLE:

```
$ . ~/6.00/arch/share/bin/setup
$ idle &
```

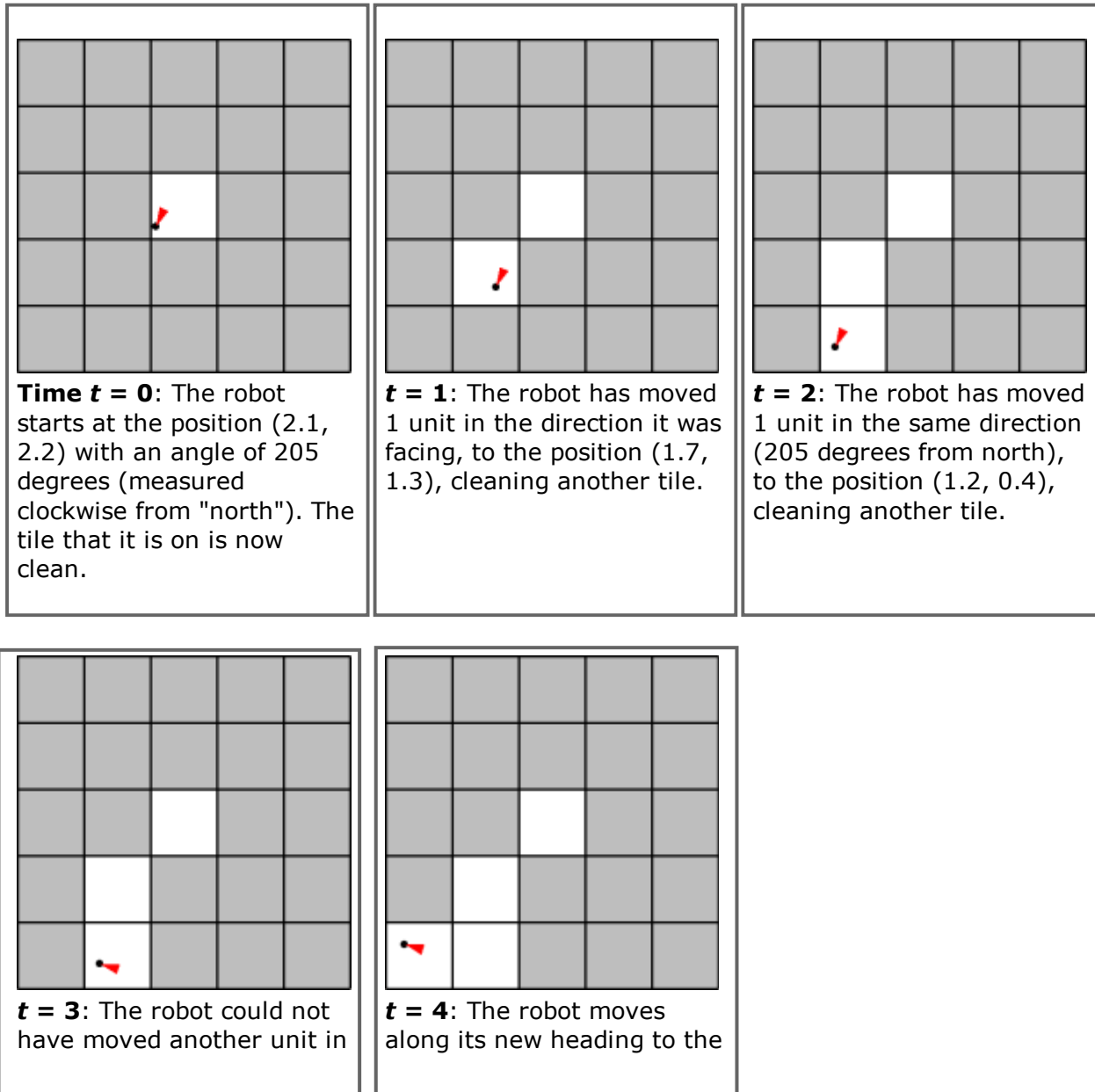
You must follow these steps **each time** you start working on an Athena machine.

## 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 5x5 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).



the same direction without hitting the wall, so instead it turns to face in a new, random direction, 287 degrees.

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 real numbers 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.
  - When a robot hits a wall, it 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  $1 \times 1$  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:  $(0, 0), (0, 1), \dots, (0, h-1), (1, 0), (1, 1), \dots, (w-1, h-1)$ .
- **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 RectangularRoom and BaseRobot 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 `ps11.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.

### BaseRobot

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 `ps11.py` carefully before starting, so that you understand the provided code and its capabilities.**

## Problem #1

**Complete the `RectangularRoom` and `BaseRobot` classes by implementing their methods in `ps11.py` according to the given specifications.**

For the `RectangularRoom` class, you will need to decide what fields you will use and 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 `BaseRobot` class, you will need to decide what fields you will use and 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

(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.*)

## 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 `BaseRobot` class you completed above (which contains general robot code), and a `Robot` class inheriting from it (which contains its own movement strategy).

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

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

    At each time-step, a Robot 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.  
    """
```

**Hint:** You may find the `getNewPosition` function in the `Position` class helpful.

### Problem #3

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

In each trial, the objective is to gather data on how many time-steps are 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, visualize):
    """
    Runs NUM_TRIALS trials of the simulation and returns a list of
    lists, one per trial. The list for a trial has an element for each
    timestep of that trial, the value of which is the percentage of
    the room that is clean after that timestep. Each trial stops when
    MIN_COVERAGE of the room is clean.

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

    Visualization is turned on when boolean VISUALIZE is set to True.
    """
```

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

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

Then, in `runSimulation` you should use `robot_type(...)` instead of `Robot(...)` 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.)

For now, set the `visualize` parameter to be `False`; you can change it to `True` when adding the visualization code in the latter part of this problem.

Feel free to write whatever helper functions you wish, but the file you submit should not have code outside these helper functions and the specified classes and functions.

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 5x5 room.
- One robot takes around 190 clock ticks to clean 75% of a 10x10 room.
- One robot takes around 310 clock ticks to clean 90% of a 10x10 room.
- One robot takes around 3250 clock ticks to completely clean a 20x20 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

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 `ps11_visualize.py` and save it in the same directory as your `ps11.py`. Add the following line to the top of your `ps11.py`:

```
import ps11_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 = psll_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)
```

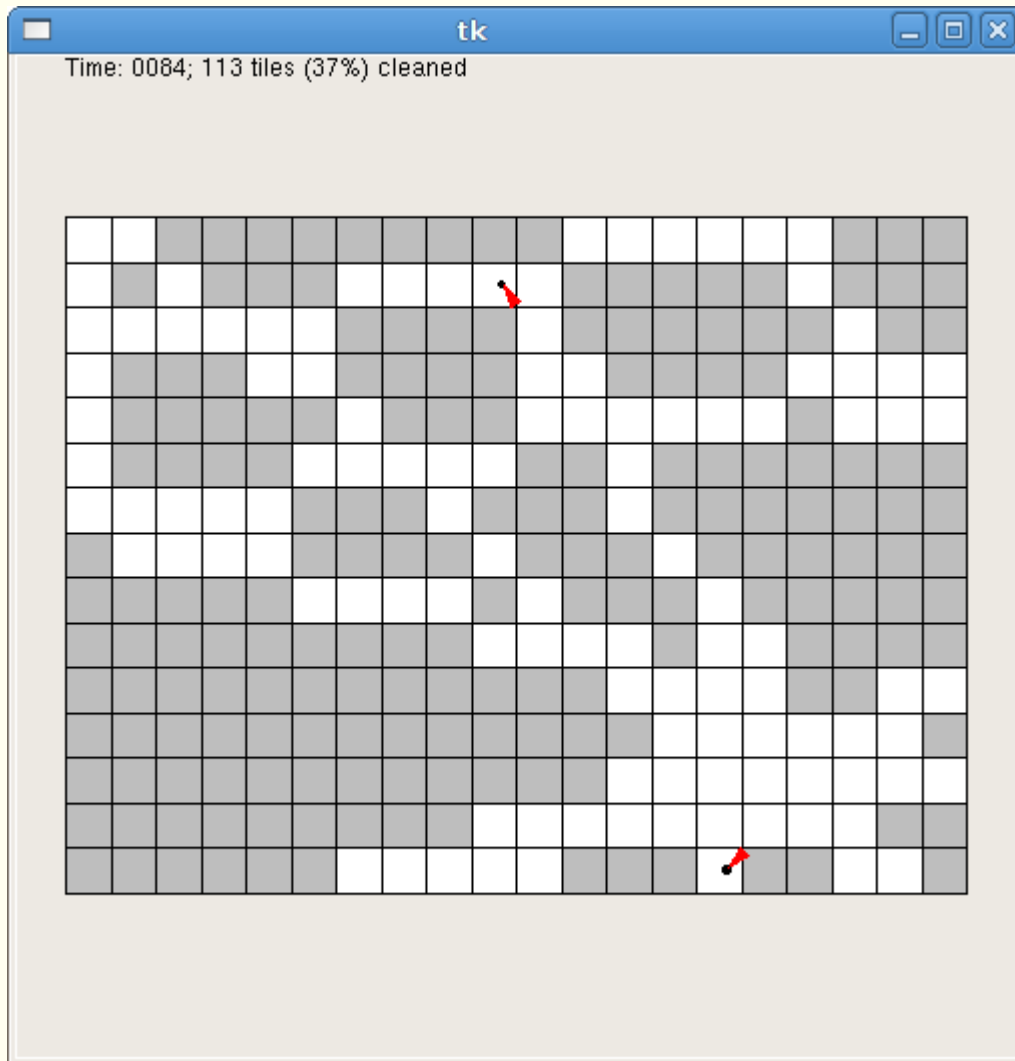
Pass in a `RectangularRoom` object, representing the current state of the room, and a list of the robots in the room.

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

```
anim.done()
```

Note that at this point, the program will wait for the user to close the animation window before continuing on to the next trial.

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 = ps11_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.

Don't forget to set the `visualize` parameter of `runSimulation` to `True` to see the animation.

## Problem #4

Now, use your simulation to answer some questions about the robots' performance.

**For the questions below, write code which will generate a plot using pylab. Put your code inside the corresponding skeleton functions in `ps11.py` (`showPlot1`, `showPlot2`, `showPlot3`, and `showPlot4`, respectively).**

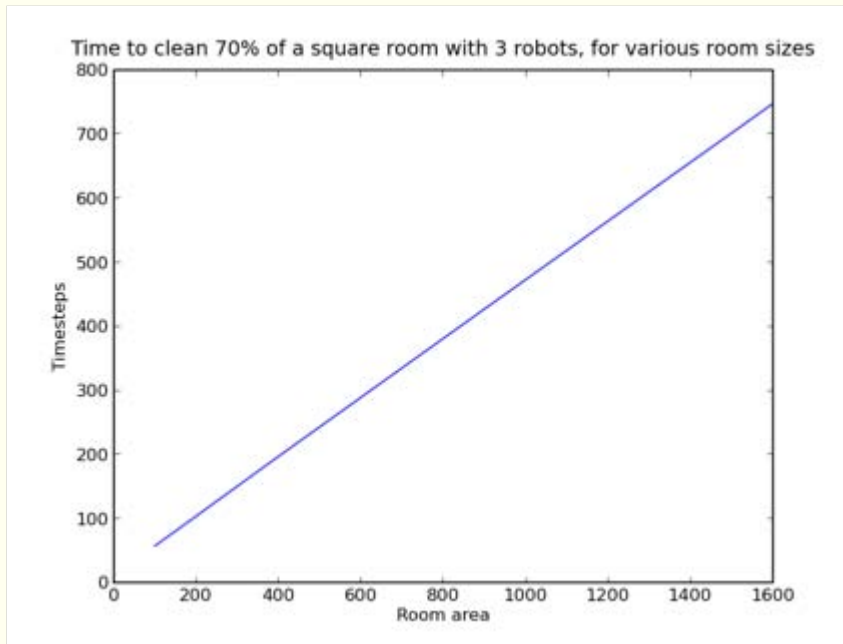
Each plot should have a title, descriptive labels on both axes, and a legend (if applicable). Assume the robots are moving at a speed of 1.0.

1. How long does it take a single robot to clean 75% of each of the following types of rooms: 5x5, 10x10, 15x15, 20x20, 25x25? Output a figure that plots the mean time (on the Y-axis) against the area of the room.
2. How long does it take to clean 75% of a 25x25 room with each of 1-10 robots? Output a figure that plots the mean time (on the Y-axis) against the number of robots.
3. How long does it take two robots to clean 75% of rooms with dimensions 20x20, 25x16, 40x10, 50x8, 80x5, and 100x4? (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.
4. How does the time it takes to clean a 25x25 room vary as `min_coverage` changes? Output a figure that plots mean time (on the Y-axis) against the percentage cleaned, for each of 1-5 robots. Your plot will have multiple curves.

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.

**Hint:** for plots 1-3, it may be useful to write a helper function that computes the average length of all the lists in the list of lists returned by `runSimulation`. For plot 4, you may find the function we have provided, `computeMeans`, to be helpful.

Here is an example of a good plot:



## 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` **which inherits from** `BaseRobot` (like `Robot`) **but implements the new movement strategy.** `RandomWalkRobot` should have the same interface as `Robot`.

**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 `Robot`.

## Problem #6

**Generate an appropriate plot (of your own design) that compares the performance of the two types of robots.** Add your code to `showPlot5()`. As always, your plot should have an appropriate title, axis labels, and (if applicable) legend.

Within comments in `showPlot5`, comment briefly on how the two types of robots compare.

## Hand-In Procedure

### 1. Save

Save your code in a single file, named `ps11.py`. **Remember, you should not have any code outside of the specified classes and functions, aside from any helper functions you may have written.**

### 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 `Robot` 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 11
# Name: Jane Lee
# Collaborators: John Doe
# Time: 3:30
#
... your code goes here ...
```