Name: Evan Leglar	**Lab Time** Tuesday 12:00 PM

Names of people you worked with:
• Kristina Owen • Declan Siewert • Andrea Oswalt

Websites you used:
• Scipy • stackexchange

Approximately how many hours did it take you to complete this assignment (to nearest whole number)?	10

The Rules: Everything you do for this lab should be your own work. Don't look up the answers on the web, or copy them from any other source. You can look up general information about Python on the web, but no copying code you find there. Read the code, close the browser, then write your own code.

By writing or typing your name below you affirm that all of the work contained herein is your own, and was not copied or copied and altered.

Evan Leglar

Note: Failure to sign this page will result in a 50 percent penalty. Failure to list people you worked with may result in no grade for this lab. Failure to fill out hours approximation will result in a 10-percent penalty.

Turn .zip files of Python code to Canvas or your assignment will not be graded

BEFORE YOU BEGIN

1. Grab a copy of the simulator from the Files area of Canvas. There are three versions, one for each major operating system
2. Check that it works:
3. You run this from the command line like this:

```
simulator waypoints 10
```

   where `simulator` is the name of the executable, `waypoints` is the name of waypoints file, and `10` is the problem instance. Make sure you can run this code with the supplied waypoints file on your computer.

Learning Objectives:

The goal of this homework is to give you some experience in dealing with external programs that calculate things for you, and in how to wrap them up in Python so that you don't have to think about them when you're using them.

Thoughts (come back and read these after you've read the problems):

1. The waypoint file has two numbers per line, corresponding to the x and y waypoint coordinates. Your first waypoint should be (-10, -10) and your final one should be (10, 10). There is no limit to the number of intermediate waypoints you can have.

2. Your evaluation function should take the list of waypoints (as a list of tuples), write it out to a waypoints file (in the correct format), call the external program (with the correct arguments), harvest the output, extract the cost, and return this value. You should do each of these things in turn, and verify that each does the right thing before moving on to the next.

3. For your `better_waypoints` file, it should look like this:

```
-10 -10
x y
10 10
```

   where `x` and `y` are the coordinates of the waypoint that you find with your code.

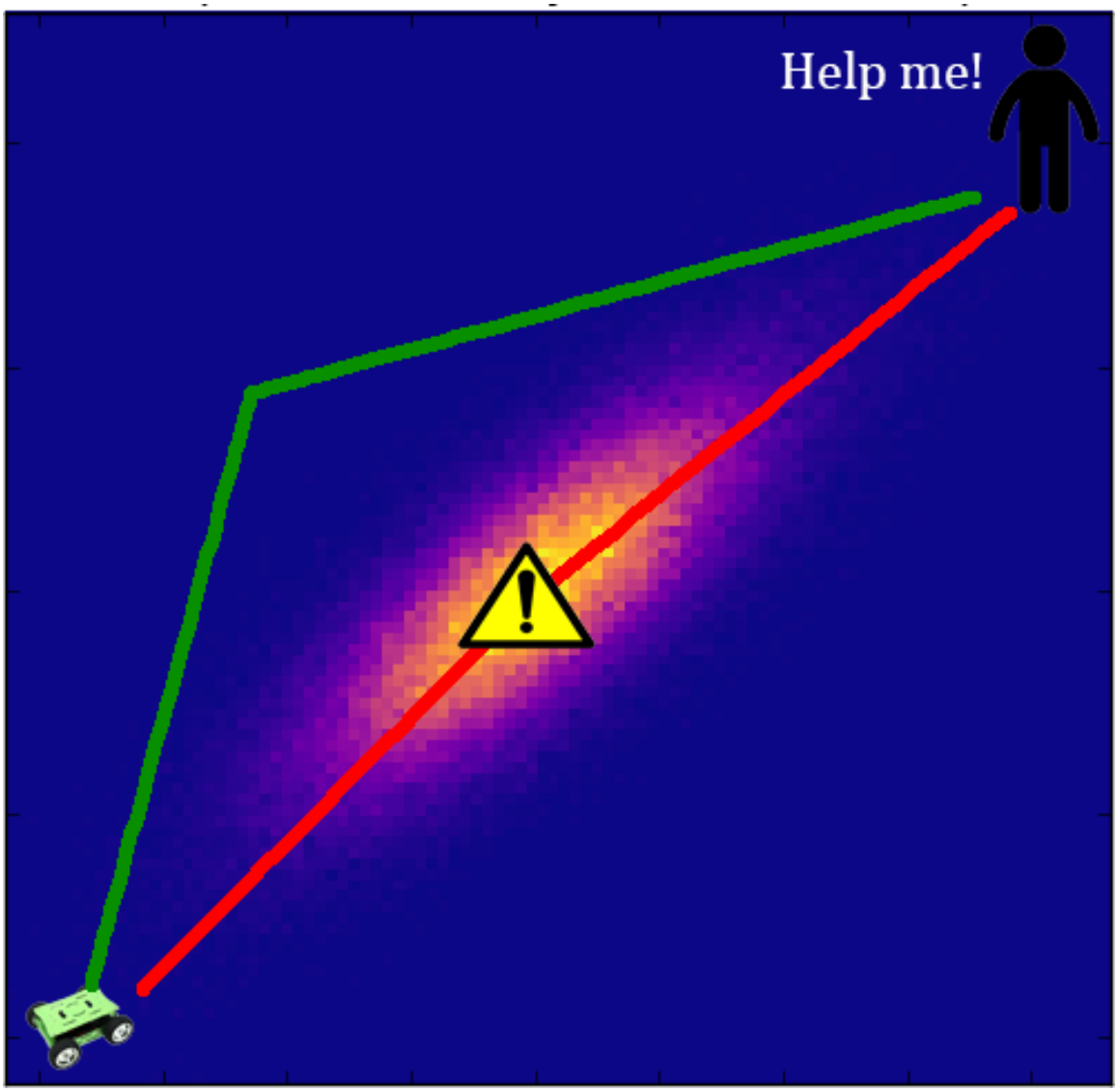
Grading Checkpoints (20 points + 3 extra credit)

1. Simulator class properly encapsulates data (1 point).
2. `evaluate()` writes out a file successfully (1 point)* and retrieves the output successfully as a float (3 points)*.
3. `search_random` gives us a different result each time (1 point)*, returns the desired values (2 points)*, and for a sufficiently large n returns a better path than the base path (1 point)*.
4. `search_grid` gives us a consistent result (1 point)* and returns the desired values (2 points)*.
5. `search_fmin` gives us a consistent result (1 point)*, is set up correctly (1 point)*, and returns the three desired values (2 points)*.
6. `compare_fmin_versus_random` runs correct set of tests (1 point). The resulting graph has a meaningful title and axes (1 point) and clearly distinguishes the results of using the random algorithm versus fmin (2 points).
7. **Extra Credit:** `search_fmin_multi` works for arbitrarily large n (2 points)* with a meaningful plot and discussion (1 point).

* The autograder will give you immediate feedback.
** The autograder will hide feedback until after grades are released.
No marker means it will be manually graded.

Hand in this document as a PDF, along with your simulator.py file to Gradescope. Do not include the executable files, we will be using our own.

Overview



In this assignment, you'll be running a simulator that drives a robot through a hostile, radiation-filled environment. Consider the above depiction; from the start, the robot wants to find a route to the goal at the top-right, but there's a big source of radiation in the middle. If the robot just drives straight to the goal, it will take on a big dose of radiation, which is bad. However, if it puts an intermediate waypoint which dodges the radiation, then it will reach the goal without incurring a large dose of radiation.

For this assignment, **you will not be able to actually see the environment**. You will run an external program which, given a problem number and a path, will tell you the amount of radiation the robot incurs. Our goal is, through repeated interactions with this program, to figure out a path which minimizes the robot's exposure to radiation.

Problem 1 Run the simulator

First, we're going to write a class which encapsulates running the simulator. **Create a new file called simulator.py.**

1. Write a class called `Simulator` whose `__init__()` method takes in a single argument, a problem instance number (i.e. an integer).

2. Write an instance method called `evaluate` which takes in one required argument, a list of 2-tuples representing waypoints, as well as an optional argument `file_name` argument which defaults to the string `'waypoints.txt'`. This function should then take care of running the simulator, writing out the list of passed-in points to the corresponding file. It should return the evaluation from the program as a floating point number.

   Your code should throw a ValueError if the list of passed in points does not start with (-10, -10) and end with (10, 10).

Example which runs a single-waypoint path on problem instance 10:

```
w = [(-10, -10), (0, 2), (10, 10)]
s = Simulator(10)
print(s.evaluate(w))
```

Warning to Windows users! For Windows users, when using `subprocess` module, when using `call` or `Popen`, **pass in the desired command as a list, not as a string!** For instance, if you want to run the command `my_script -h 3`, please structure your code to do `['my_script', '-h', '3']` instead of `'my_script -h 3'`. You may find the `shlex` module helpful.

The reason for this is that for some reason, Unix environments don't like the latter option, but Windows is OK with it. However, the autograder is on a Unix environment, and so it will complain if you try to do it. If you have any questions, ask a TA.

Comments for grader/additional information (if any)

Problem 2 Finding better waypoints

The basic path is from (-10, -10) to (10, 10). For this section, we're going to investigate various methods for finding a better path by inserting a single waypoint.

1. **Write an instance method `search_random()`** which takes in a single integer argument `n`. It should then generate `n` random pairs of float-valued waypoints inside of [-10, 10] x [-10, 10] and evaluate each of the corresponding single-waypoint paths.

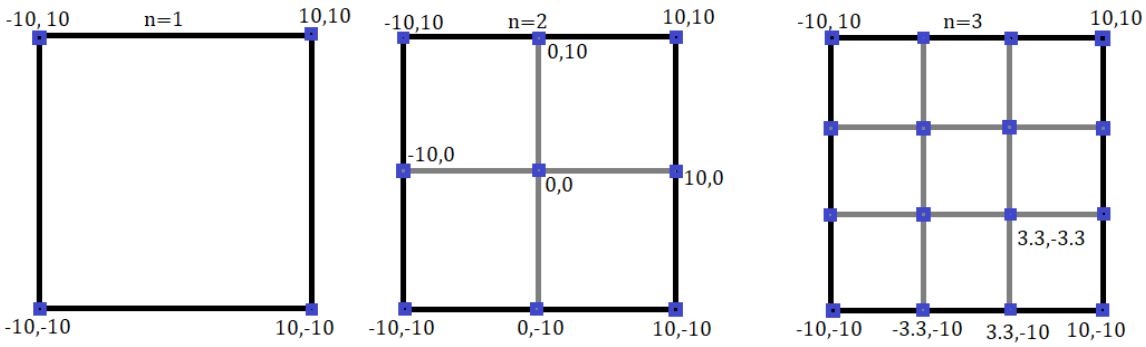
   It should return two values: The value for the path with the lowest cost, as well as the corresponding point.

   Ex:

   ```
   my_sim.search_random(100)  # Returns 3.917, (2.71, -1.98)
   ```

2. **Write an instance method `search_grid()`** which again takes in a single integer argument `n`. It should then search for a single waypoint by dividing up the square [-10, 10] x [-10, 10] into `n` x `n` cells, and evaluating the path at each of the lattice points. **It should return the best path cost and corresponding point like part 1.**

   See the below illustration for a visualization of the grid points to be searched for a given n:



   In general, for a given `n`, your function should be evaluating `(n+1)**2` waypoints.

3. **Write an instance method called `search_fmin`** which takes in no additional arguments. It should utilize scipy's `fmin` function to find a single waypoint. It should initialize its guess at (0, 0).

   It should return three values: The path cost, the corresponding point as a tuple, and *the number of times fmin evaluated your function*.

Hint: `fmin` works by passing in a reference to a function or instance method which takes in some number of arguments (say d of them), as well as an initial d-dimensional guess. Finding the best waypoint is essentially optimizing a 2-dimensional function.

Think about how you can represent running the simulator as a 2-dimensional function to `fmin`. (You may or may not wish to write a helper instance method for this purpose!)

Comments for grader/additional information (if any)

Problem 3: Analysis

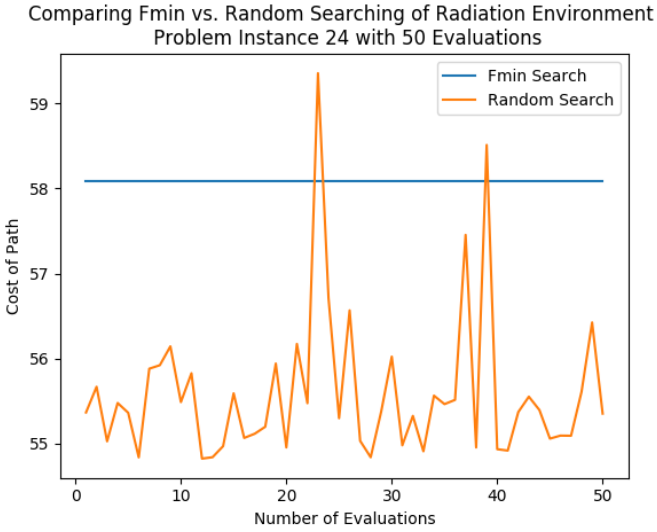
Write an instance method `compare_fmin_versus_random` which takes in an argument `n`. It should first use `search_fmin` to retrieve the path cost for the optimal path generated by `fmin`, as well as the number of evaluations took.

For each number of times fmin ran, say `num_fmin_evals`, it should then run `search_random(num_fmin_evals)`. It should repeat this process `n` times, recording the best value each time. It should then **use matplotlib to plot some sort of diagram** which compares how often random searching was able to "beat" `fmin`.

We leave the choice of diagram up to you, but it should have **a clear indication of the fmin value**, and should have **some sort of representation of the range of values that random searching produced**. **The title should also have the problem instance**, as well as **an indication of the number of times fmin evaluated the simulator**.

Have your code save it to the file `comparison_problem_{0}_n_{1}.png`, where the first argument is the problem instance and the second number is the number of comparisons `n`. For instance, if I was doing problem 10 and I did 100 comparisons, then the file would be `comparison_problem_10_n_100.png`.

Finally, run your code for problem instance 24 with n=50, and paste a resulting plot here.

comparison_problem_24_n_50.png


Comments for grader/additional information (if any)
Random Search appears to beat the fmin function consistently. Doesn't seem useful to use fmin.

Extra Credit

In this section, we'll see if we can generate some more sophisticated paths which contain more than one intermediate waypoint.

1. Write an instance method `search_fmin_multi` which takes in two arguments: A required argument `n` representing the number of waypoints in the path, and an optional integer argument `max_iter` which defaults to 500. It should then use `fmin` to find an `n`-waypoint path which minimizes the cost.

   The initial path should be a uniform linear interpolation between -10 and 10, e.g. if `n` is 3, then the initial guess for the initial waypoints should be (-5,-5), (0,0), (5,5).

   Your function should return 2 values: A list of the optimized waypoints, as well as the corresponding cost.

2. For some problem instance (put it in the plot title), generate a plot which shows the cost of the best path found as you increase n from 1 to 15, with max_iter set to 500. Comment briefly on the trend, as well as your thoughts on why you see the trend.

Comparison plot
`Copy and paste your code here`

Comments on trend

Other comments for grader/additional information (if any)