CISC 120: Lab 1

Contents

1 Introduction

This lab will be based on the textbook creative exercise 1.3.35: 2D Random Walk. This problem is stated as follows:

> A two dimensional random walk simulates the behavior of a particle moving in a grid of points. At each step, the random walker moves north, south, east, or west with probability $\frac{1}{4}$, independent of previous moves. Compose a program that takes a command-line argument n and estimates how long it will take a random walker to hit the boundary of a $2n$-by-$2n$ square centered at the starting point.

2 Problem Solving

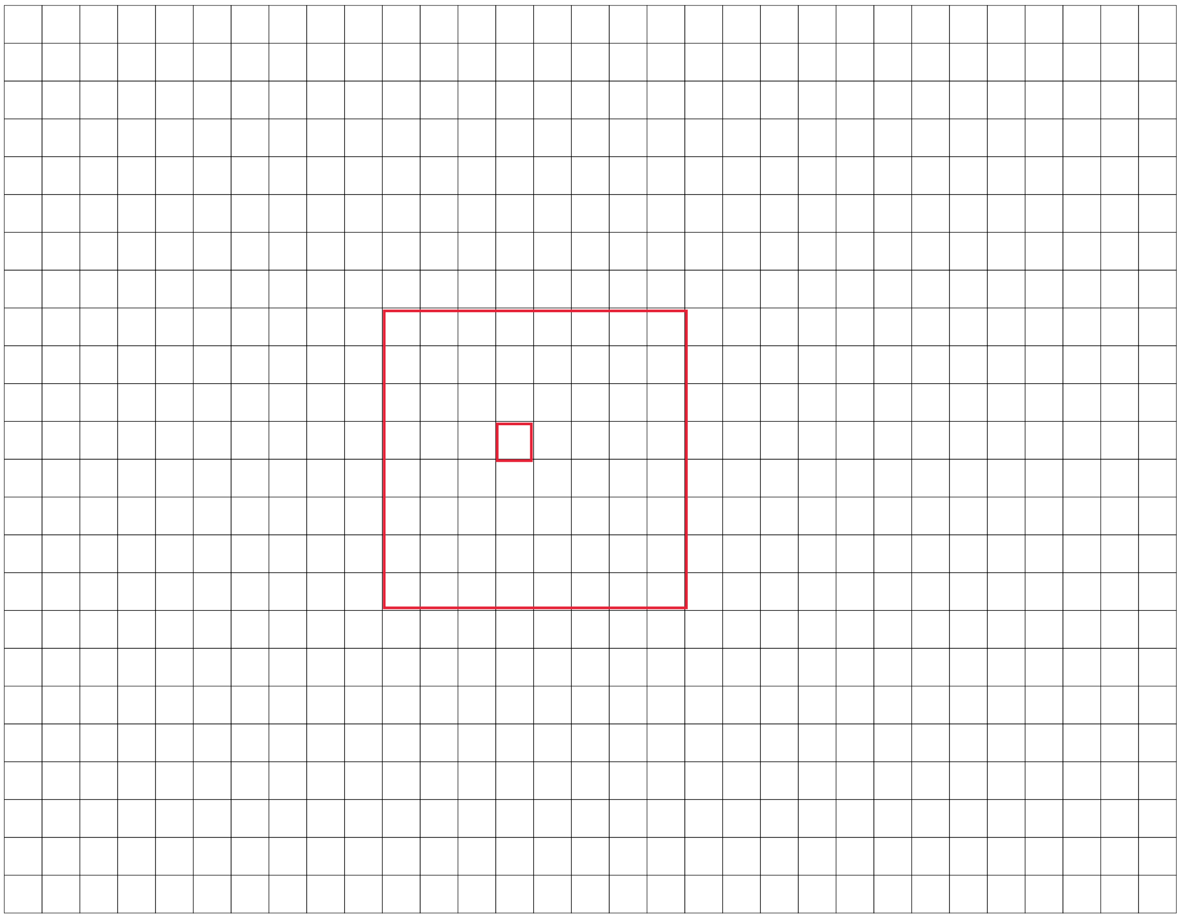
The first step to solving this problem will be to attempt to model the situation in question. Once we have a simple pencil-and-paper model of the problem, then we can attempt to replicate that model on a computer using Python.

We have two key elements to this problem that we need to model. The first element is the "walker" itself, and the second is the space in which the walker is moving. Let's start with the latter element.

Our space must consist of a $2n$-by-$2n$ grid of points. The fact that it is as high as it is wide means that we are looking at a square. Let's let $n = 4$ for the moment and consider this specific problem.

Modeling the Walker on Paper

Below you will find a grid. Using this grid, draw a $2n$-by-$2n$ square, letting $n = 4$. Indicate the center point in this square. **5pts.**



Now that you have the square represented, and its center point (which is where our walker must start), you can think about how you want to model the walker. Fundamentally, the walker itself has one key property as related to this problem. *This property is its position.*

Why Position?

Think about this for a moment, and write below why it is that the position of the walker is the property that we need to consider for our model (and program). **5pts.**

The position of the walker is fundamental because it allows us to quantify and track the walker's movement within the grid. Position is a measurable property that can change over time, and by tracking it, we can model the walker's path, predict future locations, or even backtrack to the origin. It's a discrete and objective property that is crucial for understanding the dynamics of the walker within the system.

Now that we have decide what we need to model in order to define our walker, we need to determine how we are going to model it. In what way can we track our walker's position?

Well, as you probably learned in GEND 112, positions are relative. You measure the position of an object based on its displacement from a particular point of reference. Thus, when we are modeling the position of something, we first need to define what we are measuring position relative to. In general, you will want to select a reference position that makes your life as easy as possible. In this case, let's use the starting point of the walker as our reference position.

So, then, we can model the position of the walker based on its displacement from the starting point. One common way of representing the position of objects is to use coordinates of some sort. You should be familiar with the Cartesian Coordinate Plane, in which the position of an object relative to a reference point (called the origin) is represented using a pair of numbers: an x and a y coordinate.

Return to your sketch of the square above, and draw a Cartesian Coordinate Plane, with the origin fixed at the starting point of the walker (the center of the square). This point will be called $(0, 0)$.

You can then "move" the walker by updating these coordinates. Let's assume that the walker can only move in unit intervals, and that each cardinal direction maps to increasing or decreasing a single coordinate of its position. Specifically, let's define its motions in the manner described by Table 1.

Let's come up with a random series of steps, and have our walker make them. On the

Table 1: Descriptions of the possible ways in which the walker can move

Movement Direction	**Coordinate Change**
North	Increase y by 1
South	Decrease y by 1
East	Increase x by 1
West	Decrease x by 1

Table 2: A random sequence of steps for the walker to take

Step No.	**Direction**	**Ending Position**
1	NORTH	(0, 1).
2	NORTH	(0, 2).
3	EAST	(1, 2).
4	SOUTH	(1, 1).
5	EAST	(2, 1).
6	SOUTH	(2, 0).
7	WEST	(1, 0).
8	NORTH	(1, 1).

grid, draw each step as an arrow pointing from the position at which the walker started, and terminating at the point at which it stopped. As all steps are of unit length, each arrow should start and terminate at an intersection of grid-lines.

Moving the Walker

Have the walker (which starts at the origin, $(0, 0)$) take the sequence of steps listed in Table 2. For each step, write the **ending** coordinates in the empty cell. That is, write the x and y coordinates describing the position of the walker *after* it has finished the step. Don't forget to draw the arrows on your diagram too! **5pts.**

Reaching the Boundary

Once you are finished with this process, determine whether the walker hit the boundary. Did it? And, how would you know?

If it did not, come up with a few more "random" steps that you could have the walker take, and take those steps. Record the direction and ending coordinates. Continue this until the walker has hit the boundary, then stop. Use the next page to record your steps, and your thoughts regarding reaching the boundary.

Of particular note, consider how you would determine *which* boundary was crossed. Can you come up with a check for each of the four boundaries? **5pts.**

Based on the 8x8 grid and the previous steps taken, the walker has not hit the boundary.
The furthest positions reached are (2,1) for the x-coordinate and (0,2) for the y-coordinate, which translates to (5,2) and (3,0) in grid coordinates, respectively.
None of these positions are at the grid's boundary (0 or 7).

Continuing with random steps:

Step 9: Move NORTH to (3,1) - No boundary hit.
Step 10: Move NORTH to (3,2) - No boundary hit.
Step 11: Move EAST to (4,2) - No boundary hit.
Step 12: Move EAST to (5,2) - No boundary hit.
Step 13: Move NORTH to (5,3) - No boundary hit.
Step 14: Move NORTH to (5,4) - No boundary hit.
Step 15: Move NORTH to (5,5) - No boundary hit.
Step 16: Move NORTH to (5,6) - Boundary hit at the North edge.
The walker hits the boundary at Step 16.

Okay, so at this point you will have "solved" the specific problem. We'll ignore the averaging stuff right now. Once we have a general solution for a single walk in place, we can find the average by running that same procedure many times. Solving the general case one time is basically the same as solving it 1000 times. Yay computers!

There are two things within our specific solution that we need to generalize. These are

1. Generalize the size of the region (i.e., change n)
2. Generalize the exact path taken (i.e., determine the logic necessary to make the walker take random steps, not just a specific sequence of steps that we define)

Generalizing n

Let's start by generalizing n.You noted the conditions under which the walker has hit the edge of the boundary for the specific case of $n = 4$ above. However, we need to come up with a rule that will work in general, for any box. Thus, we'll need to structure our rule in such a way that you can plug any positive integer into it, and determine if the walker has hit the boundary of a box with a size defined by that integer.

The basic idea is that, rather than using a specific number (4), when we create the rule, we should use a mathematical variable (n) in its place. This will allow us to replace n with whatever number we like, and run the calculation.

As an *unrelated* example, consider a test for the parity of a number (whether it is even or odd). We can test the number 8 for evenness by seeing if we can divide it by 2 and have no remainder (i.e., the resulting quotient has no fractional component). To generalize this test, we need to replace 8 in our description with a variable, say n, that can take on the value of any number to be tested. Thus, we would have the following rule:

> To determine if any integer, n, is even, divide it by 2. If there is any remainder, then n is not even. If there is no remainder, then it is.

Try applying this same process to your rules for determining if the walker has passed the boundary. Write your thoughts, and ultimately the rules themselves, on the next page. **5pts.**

General Rules for Determining Boundary Crossing in an n x n Grid:

1. Define the starting point:
   - For odd n, starting coordinates (x, y) are ((n-1)/2, (n-1)/2).
   - For even n, starting coordinates (x, y) are (n/2, n/2).

2. Define the boundary conditions:
   - The walker hits the north boundary if the y-coordinate is n-1.
   - The walker hits the south boundary if the y-coordinate is 0.
   - The walker hits the east boundary if the x-coordinate is n-1.
   - The walker hits the west boundary if the x-coordinate is 0.

To generalize the exact path taken by the walker:
   - Implement a random step generator that chooses between North, South, East, or West.
   - Update the walker's position according to the chosen direction,
increasing or decreasing the corresponding coordinate by 1.
   - After each step, check if the boundary condition is met. If so, the walk ends.
   - Repeat the process to simulate multiple walks and calculate averages or other statistics.

The odds are that you arrived at four different conditions, one for each boundary. This is adequate to solve the problem. However, there is a way to determine whether or not the walker has passed a boundary with only two conditions, one for both x boundaries, and one for both y boundaries. If you can't figure it out right now, that's fine. But give it a shot. It's an interesting problem, and will test your understanding of some concepts from algebra. The rules that you'll arrive at are actually a very commonly used form for boundary checks like this. Again, if you can't get it, don't worry. It won't hurt your grade to use four conditions instead of two.

Random Walks

Next up, we need to determine how to make our walker actually walk on its own. We will use the Python function `random.randint(a, b)` to complete this task.

This function will return a random *integer* on a specified interval, $[a, b]$. We need to somehow map the output of this function onto the specific steps our walker can take.

Per the problem specification, there are four possible actions that our walker can take: it can take a step NORTH, SOUTH, EAST, or WEST. The specification also makes our life a lot easier by saying that each step is independent of the previous one.[1]

Try coming up with a set of rules for determining, based on the output of this random number generator, in which direction the walker should move. Remember that each of the four directions should have an equal 25% chance of being selected. Make sure to also state what arguments you should use when you call the function for your solution to work. **5pts.**

Simplified Boundary Conditions:
- For x boundaries: |x - center_x| >= n/2
- For y boundaries: |y - center_y| >= n/2

where center_x and center_y are the coordinates of the grid's center.

Random Walk Logic Using random.randint(a, b):
- Call random.randint(1, 4) where each number corresponds to a direction:
  1: NORTH (Increase y by 1)
  2: SOUTH (Decrease y by 1)
  3: EAST (Increase x by 1)
  4: WEST (Decrease x by 1)
- This ensures each direction has a 25% chance of being selected.

[1]This means that the probability of the walker making a step in any direction is going to be the same no matter what the step that it just took was. Specifically, we know that the probability of our walker stepping in any one direction is $\frac{1}{4}$, otherwise known as 25%.

The General Process

Okay, so now we have each step solidified. Let's tie it all together in an organized process. The problem to be solved is as follows: **Given an input integer, n, determine the number of steps required for a walker that starts at the center of a region that is $2n$-by-$2n$ to reach the boundary of that region, where each step is 1 unit NORTH, SOUTH, EAST, or WEST, and the direction is randomly selected with a 25% chance of any one direction being taken.** (Remember, we're only considering a single walk at this time)

Draw a flowchart to describe the process for solving this. You've already done all of the steps for a specific n, and determined the appropriate changes to generalize it for any value of n.

HINT 1 *You will need to make use of a loop to complete this. While the walker is not at the boundary, take another step.*

Don't lose sight of the fact that we aren't necessarily interested in detecting when the walker hits the boundary in isolation. The output that we are *really* interested in is the number of steps that it took to reach that state. **10pts.**

1. Start Block: "Start Random Walk Simulation"

2. Initialization Block:
   - "Set 'n' to input integer."
   - "Initialize walker's position at center of a 2n x 2n grid."
   - "Set step count to 0."

3. Loop Start (Decision Diamond):
   - "Is walker at the boundary?"
     - If "No", proceed to Random Step Block.
     - If "Yes", move to Output Block.

4. Random Step Block (Process Rectangle):
   - "Generate random direction (N, S, E, W)."
   - "Move walker 1 unit in the chosen direction."
   - "Increment step count by 1."
   - Loop back to Loop Start.

5. Boundary Check Block (Process Rectangle):
   - "Check if absolute value of walker's position from center is >= 'n'."
   - This block loops back to Loop Start.

6. Output Block (Parallelogram):
   - "Output the total step count."

7. End Block: "End Simulation"

3 Writing the Code

Writing the Experiment

Now that we have modeled the problem and arrived at a solution, we can finally start writing code. Create a file called `walker.py` in a directory on your computer, and write the following code into it.

```python
import stdio
import random
import sys

n = int(sys.argv[1])

# Your code goes here

stdio.write('The walker took ')
stdio.write(c)
stdio.writeln(' steps.')
```

Now, we can begin to translate that flowchart you wrote, and your experience with solving this problem on paper, into a Python program.

Our high level goal right now (again, ignoring the averaging bit), is to write code that uses `n` to define the size of a box, sets the walker walking, and stores the number of steps taken to hit the boundary in `c`.

Naturally, this will require the use of a `while` loop, as well as several `if` statements. Using the flowchart and general process that you have designed above, come up with the necessary Python code and write it in place of the `# Your code goes here` comment. **30pts.**

WARNING 1 *This program must run to earn any points for this step.*

Writing the Simulation

Now that you have a program set up that can determine the number of steps required for a particular random walk, we can turn our attention to the actual problem–which is to determine the *average* number of steps. This means that we will need to simulate several walks, and calculate the average number of steps across all of them.

WARNING 2 *Make a backup copy of* `walker.py` *before starting this. We will be modifying it to complete this task, and you don't want to accidentally break it, and then not be able to fix it. Making a backup of the file ensures that, no matter what happens, we can roll our code back to our stable, working state.*

I'll give you most of this one–the trick will be to integrate your code into the framework that I give you. Using the code below, and your own code written above, write a program to determine the average number of steps required for a random walker to hit the boundary of a $2n$-by-$2n$ region.

You should name this file `simulation.py`. **15pts.**

WARNING 3 *Your code must run to earn any points for this step. Make sure that you update the indentation of your own code, so that it properly falls within the while loop. And remember that, if things get hairy and the loop doesn't stop, you can kill a program running in a command prompt by pressing ctrl-c.*

```
import stdio
import random
import sys

n = int(sys.argv[1])

m = int(sys.argv[2]) # the number of simulations to average over
i = 0   # iterator for our counting loop
total_steps = 0 # the total number of steps over all runs

while i < m: # run the simulation m times
    #
    # your walking simulation goes here
    # make sure that you call the number of steps taken
    # c, and that it is an integer.
    #

    stdio.write('The walker took ')
    stdio.write(c)
    stdio.writeln(' steps.')

    total_steps += c # add this simulation to the running total
    i += 1 # Increment the iterator

average_steps = total_steps / m # Calculate the average
stdio.write('The average number of steps was:\t')
stdio.writeln(average_steps)
```

Table 3: Analysis of $n = 4$

m	**Average Steps**
1	20.0
10	11.6
100	14.71
1000	13.456
10000	13.6282
100000	13.69421

4 Analysis

Now that we have a program written, we can use it to study the problem under examination. Let's vary `n` and `m` a bit and see how the averages change.

Analysis of m

To start with, let's fix `n` at 4, and vary `m`. If you wanted to, you could further modify the program to automatically test several values of `m` and report the results, but it is also fine (and a lot easier in this case) to just run the program repeatedly, altering the integer value referenced by the variable `m` manually.

Fill in Table 3 with the results for the specified `m` values. Some of these runs may take a bit of time to complete. **5pts.**

Analysis n

(a) Next, pick an `m` value that you like (try to balance runtime with the quality of the result) and calculate the average steps for a few different values of `n`. Record your results in Table 4. **2pts.**

(b) Draw a plot of the data in this table. **3pts.**

Conclusions

How does the choice of `n` affect the average number of steps. Specifically, what mathematical function (linear, quadratic, etc.) best fits the results that you obtained. Why do you think it is that this function in particular is the best fit–does it bear any relation to the way in which something else changes as you vary n? Additionally, does varying n at constant m have any effect on the runtime of the program? Especially compared to the effect of varying m at constant n? Record your thoughts and results on the next page. **5pts.**

Table 4: Analysis for variable n, with $m =$ 10000

n	**Average Steps**
4	13.9442
8	65.6173
16	281.2198
20	447.491

Analysis of m:
For n fixed at 4, as m increases, the average number of steps converges towards a value around 13.6, demonstrating the law of large numbers in action. The average steps stabilize with larger m values, indicating a reliable estimate of the true average for n = 4.

Analysis of n:
With m fixed at 10000, the average number of steps increases significantly as n increases. The data suggests a quadratic relationship between n and the average number of steps. The quadratic trend line fits well with the simulation data points.

Conclusions:
The quadratic relationship between n and the average number of steps implies that as the grid size increases, the average number of steps grows at a rate proportional to the square o f n. This is consistent with the fact that the area of the grid increases quadratically with n, which increases the walker's potential path length.

The runtime of the program is affected more significantly by increasing n than by increasing m. A larger grid size (larger n) likely leads to longer paths for the walker, thus increasing computation per simulation. In contrast, increasing m linearly increases the number of simulations, which linearly affects the runtime.

The plot of the average number of steps versus grid size (n) with a quadratic trend line shows that the simulation results align with the expected quadratic growth in path length as the grid size expands.

5 Lab Submission

To earn full credit for this lab, you must submit three documents by the assigned deadline. First, make a photocopy of **this lab packet**, with your answers to the questions written into it, and submit this as a .pdf file. Then, submit **two .py files**, `walker.py` and `simulation.py`. If you fail to turn in one of these files, you will forfeit any points associated with that file.

Both of these Python files should run. When grading `walker.py`, for example, I will attempt to execute the following on command prompt, with my working directory set to the folder containing all of your submitted documents,

```
python walker.py 10
```

If the program throws an error immediately, be it a syntax error, type error, name error, or any other error, then you won't receive any points for that particular program.

HINT 2 *If you cannot get the program to work, then implement as much of the required functionality as you can without running into any of these errors. This will maximize your opportunity to earn partial credit. It is better to turn in a half-functional program that runs, than an attempt to implement all of the functionality that does not.*

The assignment on Canvas will be configured to accept a few more than 3 files, should you want to upload any additional supporting documentation. For example, if you wanted to run more tests than required as part of your analysis (always a good idea!), you could record your data and make plots in a spreadsheet, and upload this too. Or perhaps some pictures of plots, if you don't like your plot sketching abilities. Just make sure to reference these documents by name within the lab packet, if you want me to refer to them instead.

6 Rubric

Question	Points	Score
Modeling the Walker on Paper	5	
Why Position?	5	
Moving the Walker	5	
Reaching the Boundary	5	
Generalizing n	5	
Random Walks	5	
The General Process	10	
Writing the Experiment	30	
Writing the Simulation	15	
Analysis of m	5	
Analysis n	5	
Conclusions	5	
Total:	100	