



Quest 13: Pokémon Go

Summary:

In this work, we will simulate a bunch of people playing Pokémon Go inside a city park. Along the way, we're going to learn about Brownian motion, which is a natural phenomenon with an incredible number of applications: the diffusion of atoms or molecules in a fluid; the dispersion of pollen through the air; the motion of people, birds, fish, and other animals in a swarm; the movement of dust and water droplets in a cloud; the thermal noise in a resistor; the behavior of stock markets and other economic systems with many buyers and sellers; and many, many others in all fields of science and engineering. Maybe you've heard of using "white noise" to help you sleep or drown out the sound of your surroundings. Brownian motion is closely related to so-called "brown noise," which has a more natural sound⁽¹⁾. In our simulation, we have a number of players wandering aimlessly around in search of Pokémon to capture. Let's model the motions of these random walkers so that we can estimate, among other things, how long it will take them to capture a Pokémon.

Skills to be Obtained:

- Simulate multiple independent particles acting over time
- Collect and analyze basic statistics from a large number of trial runs

Details:

For our simulation, we model the motion of one individual Pokémon Go player, then we model a second player, then a third and so on. Each player will move in a somewhat random way, but each player has a goal—you know what it is: they've Gotta Catch 'Em All⁽²⁾. After modeling all of the individual players, we will determine statistics about their collective behavior. This form of simulation, where randomness helps determine the outcome of each individual but a net system behavior emerges, is known as a Monte Carlo method, named after the famous casino⁽³⁾. We will make the assumption that the statistics we get from looking at one player playing over-and-over again is the same as the statistics we would determine from lots of players all playing at the same time⁽⁴⁾.

The behavior of an individual player over time will be determined as follows. Each player starts at the origin. At any point in time, the player is, of course, at one specific location. For simplicity, let's first imagine a player playing in a one-dimensional space. They start at position 0 on an x -axis that proceeds off to infinity both to the West (negative numbers) and East (positive numbers). Once every second, the player will randomly decide to walk either 1 foot to the West or 1 foot to the East. Each possibility occurs 50% of the time, and is chosen completely at random. If the walk is taken to the West, then the new position is equal to the old position minus 1. If the walk is taken to the East, then the new position is equal to the old position *plus* 1. After a long time, we might expect that the player has not moved far from position 0, but it also seems likely that they are not *exactly* back where they started.

If we want to expand the analysis to a two-dimensional space, then there are not just *two* possible motions that the player can take every second, but *four*: a 1 ft. walk to the East, West, North, or South. The x - and y -positions must both be updated accordingly after each step. We might store the x - and y -positions of the player as a function of time in two arrays. After the player is done playing, we could then plot the path of the player with either an animated view using `comet()` or a static view using `plot()`. If we re-run this simulation many times, then we could start determining statistics about the average player of this game, or perhaps the best or worst players. If we're the designers of this game,



we'd definitely want to model the time it takes for players to capture the Pokémon, so that we can know if we need to make it easier or harder.

As described above, the players will be moving somewhat randomly. Every second, a player will move exactly 1 foot, always directly North, South, East, or West, with the direction decided by a random number generator. If you were to add the directions Up and Down so that the players were moving in three dimensions, then you'd be modeling full 3-D Brownian motion. Molecules move randomly, so you could use your program to calculate how fast a drop of milk diffuses into your coffee and lots of other interesting things⁽⁵⁾. Your model could get extremely close to what you'd actually measure.

The players are in a park, which is not infinite in size. Players do not leave the park, since they know the Pokémon they are trying to catch is there. Before each player starts playing, a Pokémon is placed at a random (x, y) location in the park. The Pokémon stays at that location until it is captured, but it is then in a new location for the next player. Each player plays until they get close enough to capture the Pokémon and "win." To make this interesting, some details of your model will depend on your NUID number and your first name.

The last digit of your NUID number determines the boundaries of the park in which the players are playing the game. Each time after a player makes a move, they should check to see if they have stepped outside the boundary of the park. If so, they instantly run back to the origin to reset their position. If your NUID ends with a (all ranges include the endpoints):

- 1 or 2, then the park is a 24x24 square (i.e., both x and y span from -12 to +12)
- 3 or 4, then the park is a 36x36 rectangle (i.e., both x and y span from -18 to +18)
- 5 or 6, then the park is a 36x24 rectangle (i.e., x spans -18 to +18; y spans -12 to +12)
- 7 or 8, then the park is a 30x14 rectangle (i.e., x spans -15 to +15; y spans -7 to +7)
- 9 or 0, then the park is a 40x20 rectangle (i.e., x spans -20 to +20; y spans -10 to +10)

The first letter of your first name determines how close the player must be to the Pokémon in order to catch it. If your first name starts with (all ranges include the endpoints):

- A through G, then a player has to be 1 foot or less from the Pokémon (i.e., 1 spot directly to the East, West, North, or South).
- H through N, then a player has to be less than 2 feet from the Pokémon (i.e., any of the neighboring spots *including* the diagonals).
- O through Z, then a player has to be 2 feet or less from the Pokémon (i.e., any neighboring spot including the diagonals, or 2 feet directly to the East, West, North, or South).

Your MATLAB script(s) should create: a graphical depiction of the path that a few players took while playing, and statistics on the time it takes players to capture the Pokémon. First, create a MATLAB program that has 1 player play Pokémon Go as described above, saving in arrays the x - vs. y -positions at every second of time that they played. Ultimately, the code should output a plot of the x - vs. y -positions. This graph will be an overhead depiction of their path, starting at the origin and then moving randomly until they finally win. Sometimes, players will have a shorter path, while others will have a rough game wandering around for a while first. Each player should have a different path, because the model should have the Pokémon location be at a different random position inside the park each time, and each player chooses their path of movement randomly while playing.



Next, create a second MATLAB script (or second part of the same script), which will likely reuse a lot of the same code as the first part. The difference is that we now want to get a feel for how long it will take players to catch the Pokémon. Modify the code from part 1 so that it runs a game until the player wins and saves only the number of “seconds” it took for the player to win. Remember, in our model it takes the player 1 “second” each time they move. We don’t need to save the player’s path, just their final time to win. Let’s get a good, large dataset of at least a few thousand players. The result of this part of the code should be an array of the number of “seconds” it took for each of the players to win.

The output from your analysis should represent the physical world we are modeling, so all numbers should be described using the relevant physical units (including in plot axes). Your MATLAB script(s) should collect and present statistics including, at least:

- for three players from part 1, a graphical “overhead” plot of the path that the player took while playing, showing the boundary of the park as red dashed lines, the location of the Pokémon as a star or diamond, and the player’s meandering path with black line segments (3 graphs total, 1 for each player);
- for the thousands of players from part 2, the calculated minimum, maximum, median, and average times taken to complete the game; and
- a histogram plot (using the `hist()` function) of the time to game completion for the players from part 2. If you’re not sure what a histogram plot is or what it means, you could `doc hist` or you can read an explanation at [<https://goo.gl/bJimHq>].

Milestones Towards Completion:

Like any project that has some complexity, don’t try to sit down and solve this all at once. Solve a small problem, then make it grow by increasing the complexity one piece at a time.

You might first start by writing a model of 1 player travelling in 1-D for a fixed length of time—maybe 50 seconds—creating a single, 50-element array that stores the player’s position on the x -axis at each second. Display the array on screen after it has been fully calculated to verify that the x -position at time $t+1$ is always the position at time t plus a random jump of $+1$ or -1 . Once you’re convinced that this code is working, expand to movement in 2-D, where each time either the x - or y -position is adjusted by ± 1 (while the other remains the same value as before). Make sure the x - and y -position arrays are both of size 50 after the model runs. When you have confidence that this code is successfully implementing the random walk and saving the player’s path, add in the boundary condition so that the player never steps outside the park. Next, instead of always playing for exactly 50 seconds, determine a random location for the Pokémon at the start of the game, and have the simulation stop as soon as the player is close enough to catch the Pokémon. This should complete the first phase. The last thing to do is to implement all of this in a loop so that the code plays a few thousand times, and change the output so that the total play time for each player is saved instead of the x - and y -positions.

Submission Requirements:

Please submit your findings on Blackboard. You will submit a memo and all raw code files to the regular Blackboard location. The memo must be a pdf document with the usual format and content that describes your work, includes all requested numbers and graphs, and has all of your MATLAB code(s) transcribed as text at the end. Please also submit the MATLAB code(s) as raw .m files.



Notes and Further Reading:

1. Brown noise is named after its discoverer, Robert Brown. White noise is named by analogy to white light, because both contain a wide spectrum of frequencies. The fact that both brown noise and white noise, which are the two most common types of noise in nature, have “colorful” names has led to other types of noise to also be named by color. The difference between the colors comes from their frequency spectra, meaning roughly how much bass vs. mid-frequencies vs. high frequencies are present. Some people claim that pink noise creates the most soothing sound. Its name comes from it being halfway between white noise and brown noise, and it splits the difference between the tinny, high-pitched white noise and the more bass-heavy brown noise.
2. <https://bit.ly/2y4f8ap>
3. Monte Carlo methods were initially developed to model radioactive decay during the Manhattan Project. Naming this style of simulation after a casino is appropriate, since the outcome of each individual game is fairly random, but a net result emerges by the end of each day: the house wins!
4. So, we’re assuming the players don’t interact with each other. In theory, we could even average the behavior of just one single player playing just one time, but for a very, very long time. When the average-over-a-long-time of one thing is the same as the average-over-many-things at one point in time, the system is termed “ergodic.”
5. The path created by a particle moving with repeated, randomly chosen directions is called a random walk or, more evocatively, a drunkard’s walk. Albert Einstein showed in 1905 that a random walk is very closely related to Brownian motion, providing strong evidence that atoms and molecules were real, particle-like things.
For more: Wikipedia has decent articles on both random walks and Brownian motion; the most interesting part of the random walks article is perhaps the “Applications” section, which shows the many places where random walk processes show up.
[\[en.wikipedia.org/wiki/Random_walk\]](https://en.wikipedia.org/wiki/Random_walk) & [\[en.wikipedia.org/wiki/Brownian_motion\]](https://en.wikipedia.org/wiki/Brownian_motion)