1 MOD510 project feedback

Project number: 4

Group number: 4

Abstract Is the abstract accurate and informative? (3.3 / 5.0)

- Include some key results.
- Several sentences in your abstract are identical to sentences in your reference no. [1]. Even though you have cited it, this would be regarded as plagiarism.
- Note that it is not so common to provide citations in the abstract.
- Statements about sigmoid function and normal distribution are never mentioned again. If you include these statements, you should elaborate somewhere in the text.

Introduction Status of problem and the major objectives. (3.5 / 5.0)

- Next time, add some theory.
- Unclear / confusing wording: "a random walker simulation (which is modeled on an infected person)".
- "In each step, the visited site has a probability p of becoming infected": What does it mean for a site to be infected? In our model, it is the person moving to a site / location that can be infected.
- It seems a lot of the terminology you use is the same as in your reference no. [2]. However, the model in ref. [2] is not identical to our model. Also, and as mentioned already, you should be wary of copying too much text.

Reflections How did you work? What did you learn? What will you do differently next time? Critical comments on the exercise itself, what did work and what did not work, and why? (10.0 / 10.0)

- Very good feedback and personal reflections.

Conclusion What was learned about the methods used, and from the results obtained? Possible directions and future improvements? (9.5 / 10.0)

- Overall, good conclusion.
- Especially insightful remarks about the advantages of the "modular random walk framework.".
- You could try to include a few more specific results?

Overall presentation of results Clarity of figures, tables, algorithms and overall presentation. Too much or too little? (9.5 / 10.0)

- Consider adding axis labels also the walker plots.
- Some of the markdown appears to be crossed out.

References Relevant works cited accurately? (3.9 / 5.0)

- Good that you have included some references of your own and cited them.

Overall code quality Less coding, more done. Are the functions re-used across different exercises? Do parameters need to be changed more than one place? (8.8 / 10.0)

- Overall, good code with variable and function names that are easy to read and understand.
- You could vectorize even more of the code, e.g., the code inside `revert_illegal_move` could be greatly simplified.
- Recommend moving most of the plotting, statistical calculations etc. outside of the main simulator class (including the Monte Carlo iterations). That way, your code would become much more reusable for other purposes as well, since the only job of the simulator would then be to perform a single run of the random walk model. You could create extra functions and/or classes outside of the class to do the other tasks.
- Simplifying the class also means there will be less code to read for you or someone else in the future.
- Nice that you have been able to fruitfully use boolean masking etc. :).

Bonus points? (0.0 / 5.0)

Analysis Are the results well understood and discussed? (19.0 / 20.0)

- E1: Several insightful remarks.
- E2: Very good.
- E3: Extremely detailed and nice analysis on a large sample of data.

Code implementation & testing Does the code execute? Is the implementation correct? Testing and discussion of benchmarks. (23.8 / 25.0)

- E1: Seems correct, and the simulations run pretty fast too, good job!

Overall points: 91.3 / 100.0

2 Selected feedback to project 4

2.1 Wrong checks for illegal positions?

Many of you allowed 50 as a valid x- and $y-$coordinate when moving walkers. Since counting starts at 0 in Python, it means you were actually using a 51×51 grid instead of a 50×50 grid. This produced results which were slightly different from the reference solution shown in Figure 3 of the assignment.

Curiously, most of you still assumed a 50×50 grid when selecting initial walker positions.

2.2 Speed

In this project, speed was actually quite important. Below are a few guidelines that might help you improve the efficiency of your program:

- Don't do pointless checks. If you know already that something cannot happen, there is no need to check for it, especially not inside a time-critical for loop.
- Avoid raw loops in the parts of your code that are run many times. It can be a funny "game" to vectorize as much as possible.
- But be wary of premature optimization. Sometimes there is a trade-off between speed and readability. Also, if the code is very readable, you will probably be able to optimize it in a shorter amount of time.
- A good tip is to start by actually measuring the execution time of each of your functions. Then, you can find out which one is the slowest and concentrate your optimization efforts there.
- Limit the amount of data you copy around. Again, focus first on the parts of your program which are the most computationally intensive.
- To improve readability and avoid the risk of introducing bugs, try to split your code into several smaller functions, each of which does primarily one task.

In this project, it was the `collide()` function that were the main bottleneck, thus you should concentrate your efforts on improving the speed of this function. For the SI-model, some concrete suggestions are:

- If there are no infected people at a location, there cannot be any new infections, hence there is no need to loop over the people at that location.
- Similarly, if the whole population is already infected, there is no point in checking for new infections.
- Only check the locations where there is at least one infected person and at least one susceptible person, you can ignore the other locations.
- If you loop over each infected person, you will often end up checking the same location multiple times, this is not the most efficient method.
- Avoid for loops when drawing random directions.
- Avoid for loops when subsequently moving the walkers.
- Avoid for loops when reverting back illegal moves.

The point about drawing random directions may not be the most important, but it is interesting to note that the way you draw random numbers can actually have a measurable impact on speed[1]. An example of this is shown in Figure 1. The results show that if you are using the `np.random.randint` function, you should try to draw as many random numbers as possible in one go. Notice also that if you draw random integers one at a time, the NumPy version is slower than the corresponding function in the random module[2]. This is not really that surprising, because NumPy is optimized towards working with vectorized operations on large arrays of numbers.

It is further indicated that it may not be a good idea to use `np.append` repeatedly inside your program, the reason being that you make a copy of the entire array each time the function is called.

[1] `https://eli.thegreenplace.net/2018/slow-and-fast-methods-for-generating-random-integers-in-python/`
[2] `https://docs.python.org/3/library/random.html`

```python
def draw_random_numbers_v1(N=100):
    return np.random.randint(0, 1, N)

def draw_random_numbers_v2(N=100):
    # NB: Here, we use the random module and not NumPy
    return [random.randint(0, 1) for _ in range(N)]

def draw_random_numbers_v3(N=100):
    return [np.random.randint(0, 1) for _ in range(N)]

def draw_random_numbers_v4(N=100):
    random_numbers = []
    for _ in range(N):
        random_numbers.append(np.random.randint(0, 1))
    return random_numbers

def draw_random_numbers_v5(N=100):
    random_numbers = np.zeros(0)
    for _ in range(N):
        random_numbers = np.append(random_numbers, np.random.randint(0, 1))
    return random_numbers
```

```python
%timeit draw_random_numbers_v1()
%timeit draw_random_numbers_v2()
%timeit draw_random_numbers_v3()
%timeit draw_random_numbers_v4()
%timeit draw_random_numbers_v5()
```

```
10.9 µs ± 123 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
70.5 µs ± 431 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
359 µs ± 2.25 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
380 µs ± 1.91 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
997 µs ± 9.75 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Figure 1: Example showing that how you draw random numbers can measurably impact the speed of your program. This code was run on a computer with a 12th generation Intel Core i7 processor (3.60 GHz).

3 Some Python recommendations

If you are planning to continue programming in Python, below are some suggestions for websites and tools that can help you improve your code:

- realpython.com[3] has a lot of interesting tutorials and articles. While some require a membership, a lot of them can be read for free too.

[3] `https://realpython.com/`

- Apply static code analysis and formatting tools to improve your code and find bugs more quickly. Some notable examples are black[4], flake8[5], and pylint[6].
- If you are used to programming languages with static type-checking, you may also want to explore how you can add type annotations to your Python code[7]. You can then use tools like mypy[8] to check for errors or inconsistencies *before* running your programs.
- For larger programs, it may also be a good idea to write automated test functions, e.g., by using pytest[9].

[4] `https://black.readthedocs.io/en/stable/`
[5] `https://flake8.pycqa.org/en/latest/`
[6] `https://pylint.pycqa.org/en/latest/`
[7] `https://realpython.com/python-type-checking/`
[8] `http://mypy-lang.org/`
[9] `https://docs.pytest.org/`