Random walking dead

MOD510: Mandatory project #4

Deadline: 08.12.2019

Nov 22, 2019

Learning objectives. By completing this project, the student will:

- Model a zombie outbreak stochastically using Monte Carlo techniques.
- Study the movement and interacting random walkers on a 2D lattice grid.
- Investigate how model parameters in a continuum model can be predicted from random walk
- Quantify probabilities of different scenarios by considering an ensemble of model runs, and plotting histograms.

Contents

1 Introduction **2**

2 Exercise 1: Solving the SZ model using Monte Carlo **3**

3 Exercise 2: Random walking dead **4**

4 Exercise 3: Estimating β and the basic reproduction number **7**

5 Exercise 4: Old and young population - who survives? **8**

6 Exercise 5: Implement your own scenario **9**

7 Guidelines for project submission **9**

A Object-oriented programming? **10**

B The use of arrays to simulate a random walk **10**

C The combined use of classes and arrays **13**

1 Introduction

In this project we are going to gain further insight into zombie outbreaks by using Monte Carlo (MC) simulation. Previously, we studied the spreading of diseases with *compartment models*. Key parameters in this approach are the probability of infection β, and the recovery rate α (kill rate for zombies). Together these two parameters were used to estimate the basic reproduction number $\mathcal{R}_0$, which provides a measure of whether the disease will be able to spread in the population.

In project 3, we simply assumed that β declined exponentially as a function of time, eventually putting an end to the disease outbreak. However, we did not explain *why* such a rapid decline should occur, if at all. For the particular case of the 2014 Ebola virus outbreak in Liberia, researchers actually found that the value of $\mathcal{R}_0$ showed no signs of decreasing, even 6 months after the outbreak [1]. Understanding the mechanisms for how to decrease β is therefore important when faced with a potential epidemic.

To make the above statement more concrete, we will in this project model *interactions* between sick and healthy persons *directly*. In this way we can test various assumptions, and predict how β and α might vary as a function of time. To this end, we shall employ *random walk* [3, 2] simulations. We will start by solving the SZ model using a Monte Carlo approach. Here, we will not learn anything new regarding zombie-human interactions, because we are solving the same model as in project 3. However, you will discover a completely different way of solving the compartment model, and gain an appreciation for how random fluctuations can affect the final solution.

Next, we model detailed interactions between individuals in the population using a random walk method. Based on the output from these simulations, we predict the parameter β from the compartment model as a function of time, which we subsequently compare with the original SZ model. We also investigate the impact of various model assumptions regarding how humans learn from surviving zombie encounters.

Finally, we add the possibility that a subset of humans move more slowly than the rest, and explore some scenarios for how this impacts the predicted fate of the humans.

Models at different scales.

To gain a fundamental understanding of model parameters at a given scale, one needs models at a finer scale. During the process of *upscaling*, the output from the finer models are used to compute model parameters at the coarser scale.

2 Exercise 1: Solving the SZ model using Monte Carlo

The SZ model was described in Project 3, and is repeated here for completeness:

$$\frac{\mathrm{d}S(t)}{\mathrm{d}t} = -\beta \cdot \frac{S(t)Z(t)}{N} \tag{1}$$

$$\frac{\mathrm{d}Z(t)}{\mathrm{d}t} = \frac{\mathrm{d}(N-S)}{\mathrm{d}t} = -\frac{\mathrm{d}S(t)}{\mathrm{d}t} = \beta \cdot \frac{S(t)Z(t)}{N}\,. \tag{2}$$

The probability p_Z of a human (S) turning into a zombie (Z) was given as

$$p_Z = \beta \cdot \Delta t \cdot \frac{Z(t)}{N}\,, \tag{3}$$

where N is the total number of humans and zombies, and β was put equal to 0.06 1/hr.

This exercise.

- Let $N = 683$, and $Z(0) = 10$, and $\beta = 0.06$ 1/hr.
- Suppose that during *each time step*, *all* humans can be contacted by the zombie(s), and thus stand the chance of being infected.
- Specifically, for each human pick a random number u between (0,1). If $u < p_z$, convert the human to a zombie, i.e, update $Z = Z + 1$, and $S = S - 1$.
- Whenever a human is turned into a zombie update p_z according to equation (3)

Part 1.

- Make a code that uses the algorithm described above to predict the fate of humans and zombies

Part 2.

- Run the code 100-1000 times, and use the output to calculate the mean and standard deviation of the number of humans and zombies at each moment in time.
- Present the results in a figure, where you illustrate how the randomness in the simulations appear in the solution

1. A figure in which you plot the mean value plus/minus one standard deviation (you may use, e.g., `plt.fill_between`.)
2. Also include the analytical solution in the plot.
3. Make a histogram of the survivors after 100 hours

See figure 1 for how a possible solution might look like.

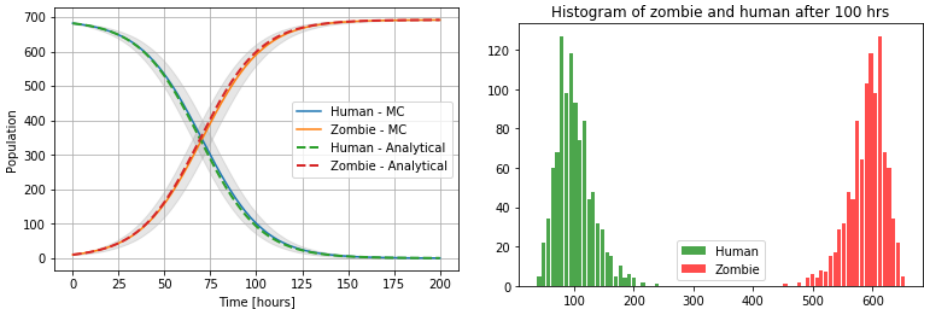

Figure 1: (left) Evolution of zombie and human as a function of time, $Z(0) = 10$ (right) A histogram showing the distribution of survivors and zombies at a specific moment in time. (1000 simulations was used)

Part 3.

- Repeat the simulation above, but this time start with $Z(0) = 1$. What is the difference, now compared to when $Z(0) = 10$? Would you say that the MC simulation is more "correct" than the continuum (analytical) solution?

3 Exercise 2: Random walking dead

Random walking dead and nuclear magnetic resonance (NMR).

The model you are going to develop is not that different from models describing NMR and magnetic resonance imaging (MRI). In MRI, all the magnetic spins are excited to a higher energy level. Then they move randomly (like random walk), whenever they bump into each other there is a probability that they fall into a lower energy level. The MRI machine measure the time it takes before all the spins returns to the original state. If the spins are confined in e.g. cells they will return to the lower„ original, energy state faster. Water contains more spins and will give a higher intensity signal, see figure 2, where the MRI image is from.

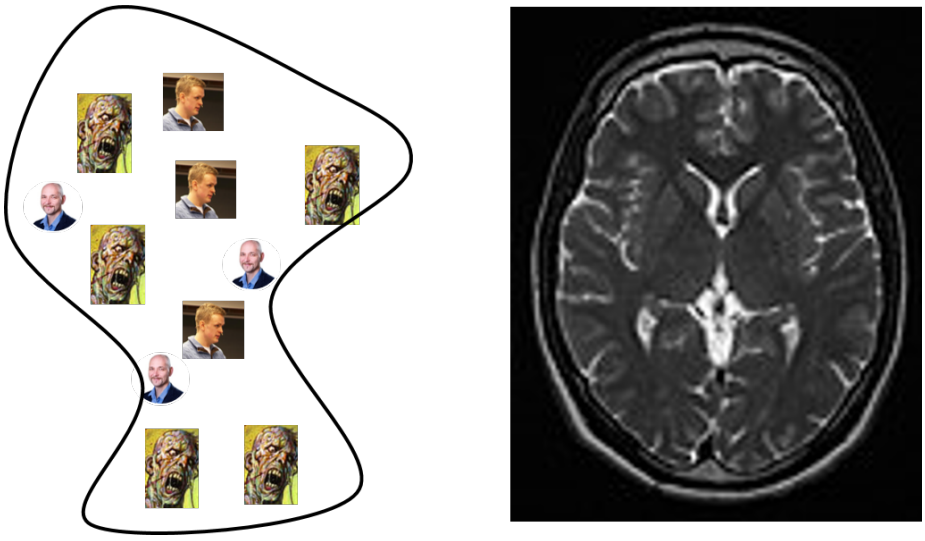

Figure 2: (left) Zombies and humans confined on an island, (right) MRI signal from a healthy brain.

We are going to investigate the interactions between humans and zombies in more detail to get a deeper understanding of the β value in the compartment model. You are going to implement the following algorithm:

- Humans and zombies are confined to a finite, square lattice with $L \times L$ nodes. This could for example represent an isolated island, or a closed off city.
- At $t = 0$, all individuals are placed out at random nodes of the lattice.
- For each subsequent time step, zombies and humans (walkers) move randomly. For simplicity, the walkers are assumed to only move in 2 dimensions. This is important: each individual moves in *either* in the $x-$ *or* $y-$direction, but not both at the same time (i.e., east, north, west, *or* south).
- After each move, find all nodes where both humans and zombies are present. Every human at such a site will face every zombie in an encounter.
- For a given human-zombie interaction, there is a probability q that the human is infected.
  - Example: If three humans meet two zombies at a given node, you have to account for six human-zombie pairs. In each encounter, draw a random number between 0 and 1; if it is less than q, the human becomes a zombie (it is possible for multiple zombies to infect the same human).

Read this first: Implementation guidelines.

To be able to implement a random walk algorithm in which walkers can have different properties and/or abilities, you should read through *all* the exercises before starting to code. It is important that your program can handle the different scenarios we want to study.

Several strategies for how to write your code are suggested in the appendix. It is recommended that you first read the appendix, even if you choose to adopt your own strategy.

Also, in this project it is extra important that you include sufficient documentation to your code.

Part 1. (THE MOST DIFFICULT AND COMPREHENSIVE PART) Write a function and/or class that can be used to conduct a single simulation of the random walk algorithm. It is important that you add every option that you will need in later exercises to the function / class! That is, you should not end up in a situation where you have to copy and paste large portions of the code repeatedly.

If you want, you can write your function/class in a separate .py file, and simply import it at the top of your Jupyter notebook.

Notice.

The final code should include the following steps:

1. Place walkers randomly on the grid
2. Move walkers
3. Check if they are at a legal position, move back to previous position if not
4. Save walker positions
5. Check nodes for possible human zombie interactions
6. At certain times, visualize walker positions (good for debugging).
7. Return to point 2

To show walkers at varying time steps can be achieved by e.g. `if i%1000==0: show_walkers()`. (Assuming that you have implemented a function called `show_walkers()`)

Part 2. Let $L = 50$ and $q = 0.9$. As before, set $N = 682$, $Z(0) = 1$. Run the (most basic) model repeatedly; at least $10 - 100$ times. For each time step,

calculate the mean and standard deviation of the number of the humans and zombies in the population.

Create a figure showing the expected time development of the two populations. Make sure to include the computed uncertainty in the plot.

4 Exercise 3: Estimating β and the basic reproduction number

The original SZR compartment model was (ignoring incubation time)

$$\frac{\mathrm{d}S(t)}{\mathrm{d}t} = -\beta(t) \cdot \frac{S(t)Z(t)}{N} \tag{4}$$

$$\frac{\mathrm{d}Z(t)}{\mathrm{d}t} = (\beta(t) - \alpha(t)) \cdot \frac{S(t)Z(t)}{N} \tag{5}$$

$$\frac{\mathrm{d}R(t)}{\mathrm{d}t} = \alpha(t)\frac{S(t)Z(t)}{N}\,, \tag{6}$$

where β was a constant model input parameter (or two, if using the exponential decline assumption). However, in the random walk model we can *estimate* values for β based on observed changes in the random walker population. To do this, we combine equation (4) with a first order approximation of the derivative to yield:

$$\beta \cdot \Delta t \approx -\frac{(S(t) - S(t - \Delta t))N}{S(t)Z(t)}, \tag{7}$$

Notice.

The time scale in the random walk model is dependent on the lattice size, i.e., how much time does it take for each encounter (depends on the speed of human and zombies and the size of the simulation domain)

Part 1.

1. Run the model once and estimate $\beta\Delta t$, from which you can estimate β as a function of time (Assume $\Delta t \simeq 1$ hour)

2. Lower the probability q of a zombie attack leading to infection, and again estimate β.

In both cases compare the outcome of the random walk simulation with the analytical solution of the SZ model (use the mean value of β).

Comment on the results.

Part 2. Again we will let the probability of being turned into a zombie be $q = 0.9$. However, suppose that whenever a human is attacked by a zombie *and survives*, the probability of surviving again at a later stage increases:

- Each time a human survives a zombie encounter, halve the probability of infection, i.e., update `q=q*0.5` for that person.
- What happens now? Plot β as a function of time. How does your prediction fit with the model we used in the project 3, i.e. $\beta(t) = \beta_0 e^{-\lambda t}$?

Part 3. For a slightly more hopeful scenario, assume that each time a human survives a zombie encounter, two things happen: a) the probability of surviving is increased as in the last exercise, and b) the human kills the zombie.

1. Estimate α using a similar approach as for β. Plot α as a function of time.
2. The basic reproduction number was $\mathcal{R}_0(t) = \beta(t)/\alpha(t)$. Comment on how the fate of humans and zombies depends on $\mathcal{R}_0(t)$.

5 Exercise 4: Old and young population - who survives?

For the remainder of the project, we shall divide the human population into two groups: young and old. For easy comparison we shall take the number of people in the two age categories to be the same, that is, equal to $(N-1)/2 = 341$.

Part 1. Young humans and zombies move as before, that is, with equal probability to go in each of the four directions (except when hitting the outskirts of the grid). On the other hand, old humans are slower to react: Suppose that the old humans have the same probability of not moving as they have of moving in one of the four directions (i.e., 20 %).

- Investigate the fate of the humans now, for the case in which humans never learn anything after meeting the zombies, and in the case they kill the zombie
- How are the results different from when all humans moved in the same way?

6 Exercise 5: Implement your own scenario

For the final part of the project, you are going to suggest your own scenario to explore. You are free to do what ever you like. However, below you will find examples of possible features to test:

- Whenever old humans survive a zombie attack, they only have a 30 % probability of killing the zombie afterwards (the probability of surviving future attacks still goes down by a factor 2)
- Add an incubation time, i.e., a certain time interval between when a human first becomes infected, and when the person actually becomes a zombie.
- Establish a *safe zone* in one part of the lattice, i.e., an area in which only humans are allowed to enter.
- Combine the previous two features: What happens if humans start exhibiting symptoms only *after* having entered the safe zone?
- Add the option that a zombie will have a higher probability to move in the direction where there are the most humans nearby.

Clearly, there are many different choices you may make for each of these model features, both individually and in combination. Therefore, it is very important that you state your assumptions carefully, and that you document your code accordingly.

Illustrate your findings in a figure, and discuss how your model scenario compare to the previously investigated cases.

7 Guidelines for project submission

The assignment is provided both as a PDF, and as a Jupyter notebook. However, the work done to answer the exercises only has to be handed in as a notebook, though you can submit an additional PDF if you want. You should bear the following points in mind when working on the project:

- Start your notebook by providing a short introduction in which you outline the nature of the problem(s) to be investigated.
- End your notebook with a brief summary of what you feel you learned from the project (if anything). Also, if you have any general comments or suggestions for what could be improved in future assignments, this is the place to do it.
- All code that you make use of should be present in the notebook, and it should ideally execute without any errors (especially run-time errors). If you are not able to fix everything before the deadline, you should give your best understanding of what is not working, and how you might go about fixing it.

- If you use an algorithm that is not fully described in the assignment text, you should try to explain it in your own words. This also applies if the method is described elsewhere in the course material.
- In some cases it may suffice to explain your work via comments in the code itself, but other times you might want to include a more elaborate explanation in terms of, e.g., mathematics and/or pseudocode.
- In general, it is a good habit to comment your code (though it can be overdone).
- When working with approximate solutions to equations, it is always useful to check your results against known exact (analytical) solutions, should they be available.
- It is also a good test of a model implementation to study what happens at known 'edge cases'.
- Any figures you include should be easily understandable. You should label axes appropriately, and depending on the problem, include other legends etc. Also, you should discuss your figures in the main text.
- It is always good if you can reflect a little bit around *why* you see what you see.

A Object-oriented programming?

When implementing the random walk algorithm, one approach would be to use an object-oriented programming style. For example, one could make a class called `Walker`, and have subclasses such as `Zombie`, `Human`, `Infected` etc. However, it can be challenging to decide on the right class hierarchy, as well as to make the program fast, because it takes more time to access a custom object than an array.

We will not pursue this approach further here.

B The use of arrays to simulate a random walk

An alternative method is to use a set of global arrays to record the state of the system. This option is more computationally efficient, but it requires more of you as a programmer; to code without classes, there is a good chance that your code will be hard to read. Below are some *suggestions* for how to proceed with this approach. It is important to follow some principles when structuring your code:

- The arrays will be accessed in various functions, and it is therefore important that they have have intuitive variable names.

- You should distinguish variables defined outside functions from those that are only used locally within a function. A common convention is to end global variable names with an underscore `_`.
- It is good practice to explicitly declare variables as global if you are going to change them inside a function: put the keyword `global` in front.
- You should do this even for lists/arrays, which can be modified in-place without the global keyword. This way you make it absolutely clear to people reading your code that it is going to be treated like a global variable.
- Very important: If you need to save, e.g., the positions of all walkers in order to use them at a later stage, you have to use `np.copy` (see further remarks below).

Below is one suggestion for a possible code structure:

```
global Walkers_     # (x,y) coordinate of each walker at time T
global N_           # Total population
global nxy_         # Lattice size (a square) nx=ny
```

Here, `Walkers` is a 2D array, with a (x, y) coordinate stored for each walker. To generate a random position for each walker, we can simply draw one $x-$coordinate, and one $y-$coordinate:

```
Walkers_=np.random.randint(nxy_+1,size=(N_,2))
```

Put, e.g., `nxy_=5`, and `N_=10`, `print(Walkers_)` and inspect the result. The next issue is how to move the walkers. It is important that they move at *random* and only *one step*, in *either* the $x-$ *or* the $y-$direction. There are many ways to achieve this. One method is to draw a random integer u between 1 and 4. If `u==1` move east, if `u==2` move north, if `u==3` move west, and if `u==4` move south. Assume for simplicity that we only have five walkers, and that for each of them we have randomly chosen the following new steps:

```
next_pos=[[0,1],[1,0],[1,0],[0,1],[-1,0]]
```

Then we can simply update the position of the walkers as follows:

```
Walkers_ += next_pos
```

In our case we need to make sure that none of the walkers move outside the grid. One way of doing this is to store all old (legal) positions

```
global Walkers_Old_ # old (x,y) position
Walkers_Old_ = Walkers_.copy()
```

Notice the use of `copy()` here. If you simply put `Walkers_Old_ = Walkers_`, it would not work, because arrays are mutable objects in Python. That is, if

`Walkers_` were assigned directly to `Walkers_Old`, both variable names would point to the same underlying object in memory. However, since the arrays contain objects of an immutable type (`int`), one way to avoid this problem is to create a shallow copy with `copy()`. In yet other applications, a `deepcopy()` operation might be needed.

Next, we find the index of all walkers outside the simulation domain, and put them back to the old position (this is often called a bounce back boundary condition)

```
# idx is an array of all walkers at illegal positions
Walkers_[idx]=Walkers_Old_[idx]
Walkers_Old_ = Walkers_.copy() # save the new positions
```

We still have no information about what type (zombie, infected, dead, etc.) the individual walkers are. Below is a suggestion on how to do this:

```
global State_       # Human = 0, Zombie = 1, Dead = 2, etc.
HUMAN_  = 0
ZOMBIE_ = 1
DEAD_   = 2

State_=np.zeros(N_,dtype=int) # Set all Walkers to Human
State_[0]=ZOMBIE_             # Set one of them to a Zombie
```

In this way it is easy to add more states, as the simulation is progressing the `State_` array will be updated. At any given time we can find the total number of humans, zombies, dead etc by using `np.bincount(State_)`, see documentation.

```
population = np.bincount(State_) # count number of 0, 1, 2 etc
population[ZOMBIE_] # returns number of zombies
population[HUMAN_]  # returns number of humans
```

A small warning: the above code does not work when, e.g., the humans disappear completely. This is because `np.bincount` counts the number of integers. If there are no zeros, the number 1 will be stored in `population[0]`, and the code will produce an error. To fix this, you could add an extra `if`-clause. Alternatively, you could use `np.count_nonzero instead`, e.g.:

```
num_humans = np.count_nonzero(State_==HUMAN_)
num_zombies = np.count_nonzero(State_==ZOMBIE_)
```

Important note!

During Monte Carlo simulation, you have to run the algorithm from start to finish many times. If you opt for the method outlined in this appendix, you will therefore have to include the whole algorithm inside some kind of function or loop, which are to be evaluated repeatedly.

C The combined use of classes and arrays

Yet another approach is to use a (single) class instead of a function to keep track of the state of a given simulation run. For example, the current state of the system can be stored in a set of class instance variables, while the integers flags denoting the status of a walker (human, zombie, etc.) can be class variables. The following code exemplifies the beginning of one such implementation:

```
class ZombieSimulator:
    """
    Class used to model the invasion of zombies in a closed off
    area with a 2-dimensional random walk.
    """

    # Class variables:
    HUMAN_ = 0
    ZOMBIE_ = 1
    KILLED_ZOMBIE_ = 2

    def __init__(self, num_humans0, num_zombies0, *, L=100):
        """
        Constructor used to initialize the model.

        :param num_humans0: Number of humans at t=0.
        :param num_zombies0: Number of zombies at t=0.
        :param L: Number of lattice nodes in each dimension.
        """

        self.L = L
        self.initial_no_humans = num_humans0
        self.initial_no_zombies = num_zombies0
        self.total_population = num_humans0 + num_zombies0

        # Various model options (to do: add more options here!!)
        self.infection_probability = 0.9

        # Current simulation state:
        self.current_time_step = 0
        self.pop_status = np.zeros(self.total_population, dtype='int')
        self.pos_x = np.zeros(self.total_population, dtype='int')
        self.pos_y = np.zeros(self.total_population, dtype='int')

        # Prepare for simulation:
        self.reset_initial_population_distribution()

    def reset_initial_population_distribution(self):
        """
        Reset model to time zero, including the initial random
        placement of humans and zombies on the lattice.
        """
        L = self.L

        # TO DO:
        # Select initial, random locations for humans and zombies (!)

        # Since locations were chosen randomly, we can let the first
        # ones in the array be the zombies:
        self.pop_status.fill(ZombieSimulator.HUMAN_)
        self.pop_status[:self.initial_no_zombies].fill(ZombieSimulator.ZOMBIE_)
```

```
        # Reset time counter:
        self.current_time = 0

    def plot_population_distribution(self):
        """
        Plot the spatial distribution of humans and zombies
        at the current moment in time.
        """
        pass # TO DO: implement code here (!)

    def simulate_random_walk(self, no_steps, *, print_steps=None):
        """
        Perform a single random walk simulation from start to finish.

        :param no_steps: Number of time steps to take.
        :param print_steps: If not None, a list of time steps at
                            which to plot the spatial population
                            distribution.
        :return: Dictionary containing the time steps, and the
                 corresponding solutions (number of humans,
                 zombies, etc.)
        """

        self.reset_initial_population_distribution()

        # Allocate arrays in which to store data for the current
        # simulation run:
        times = np.arange(no_steps+1)
        number_of_humans = np.zeros_like(times)
        number_of_zombies = np.zeros_like(times)
        number_of_killed_zombies = np.zeros_like(times)

        number_of_humans[0] = self.initial_no_humans
        number_of_zombies[0] = self.initial_no_zombies

        # Time loop:
        for i, curr_t in enumerate(times):

            self.current_time = curr_t
            if self.current_time > 0:
                # TO DO:
                # Perform random walk, and check for collisions.
                # Store no. of humans, zombies, etc. at current step.
                pass

            # Visualize where the humans and zombies are on the grid?
            if print_steps is not None and curr_t in print_steps:
                self.plot_population_distribution()

        solution_dict = {'t': times,
                         'S': number_of_humans,
                         'Z': number_of_zombies,
                         'R': number_of_killed_zombies}
        return solution_dict
```

Of course, the above code only contains the bare bones of the full implementation. To program the interesting parts of the algorithm, you can use many of the ideas that were outlined in appendix A.

An example showing how you can utilize the above class is also included:

```
N = 683
lattice_size = 10
simulator = ZombieSimulator(N-1, 1, L=lattice_size)

no_time_steps = 200
no_simulations = 100

sum_of_values_S_ = None
sum_of_squares_S_ = None
sum_of_values_Z_ = None
sum_of_squares_Z_ = None

for i_sim in range(no_simulations):

    sol = simulator.simulate_random_walk(no_time_steps)
    t_step = sol['t']  # note: this is the same for all runs
    S_i = sol['S']
    Z_i = sol['Z']

    if sum_of_values_S_ is None:
        sum_of_values_S_ = S_i
        sum_of_values_Z_ = Z_i
        sum_of_squares_S_ = S_i**2
        sum_of_squares_Z_ = Z_i**2
    else:
        sum_of_values_S_ += S_i
        sum_of_values_Z_ += Z_i
        sum_of_squares_S_ += S_i**2
        sum_of_squares_Z_ += Z_i**2

# Compute mean and standard deviation:
S_mean_ = sum_of_values_S_/no_simulations
Z_mean_ = sum_of_values_Z_/no_simulations
S_stdev_ = np.sqrt(sum_of_squares_S_/no_simulations-S_mean_**2)
Z_stdev_ = np.sqrt(sum_of_squares_Z_/no_simulations-Z_mean_**2)

# to do: plot results etc..
```

References

[1] Christian L. Althaus. Estimating the reproduction number of ebola virus (ebov) during the 2014 outbreak in west africa. *PLoS currents*, 6, 2014.

[2] Edward A. Codling, Michael J. Plank, and Simon Benhamou. Random walk models in biology. *Journal of the Royal society interface*, 5(25):813–834, 2008.

[3] Karl Pearson. The problem of the random walk. *Nature*, 72(1867):342, 1905.