Random walking dead

MOD300: Mandatory project #3

Deadline: 5. November (23:59)

Oct 18, 2023

Figure 1: Prepare for the worst (From the AMC show "The Walking Dead")

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

- Learn how to escape a zombie outbreak by modeling the outbreak as a random walk.
- Use Monte Carlo techniques to quantify model uncertainty.
- Investigate how model parameters in a continuum model can be predicted from random walk.
- Get insight into mechanisms important for spread of infectious diseases.

1 Introduction

In this project you are going to study the spread of a zombie outbreak in two Norwegian villages, Sokndal and Dirdal, using random walk simulations [10, 4, 3, 6]. You will also compare your results with classical compartment models [7].

The random-walk approach is an example of agent-based models [9, 2, 8], in which each individual (or a small group of individuals) is represented explicitly. Classical compartment models are much coarser, and do not seek to capture the behaviour of individuals. Typically, they use ordinary differential equations (ODEs) to capture the dynamics of how a disease spreads in the population. Regardless of the approach taken, the total population can, at any given time, be partitioned into compartments based on a set of possible "disease states", e.g., "Susceptible", "Infected (Zombie)", "Recovered", and "Dead". The part that is different is how we model the "flow" of people from one compartment to another.

Before we start:

In this project, more than any other, you have take care about numerical efficiency - your very future may depend on it!

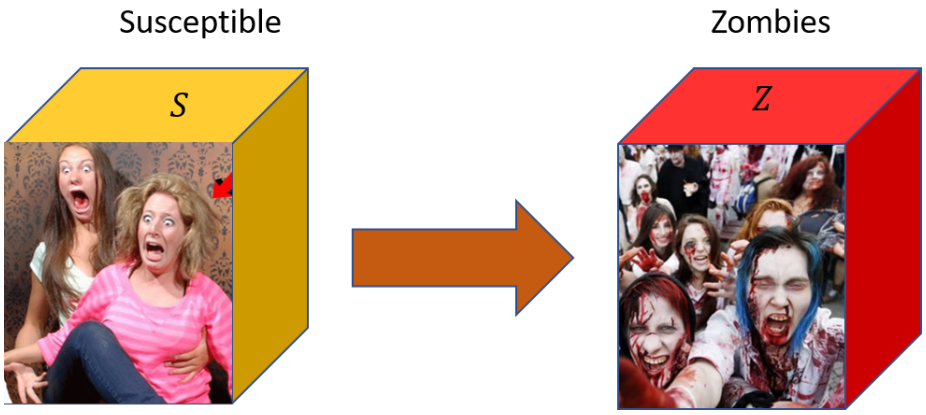

Figure 2: The SZ-model: All individuals are either "Susceptible" or "Zombies". Once you become infected, there is no recovery.

2 Exercise 1: Random walk SZ-model

To start out simple, we shall first consider the SZ-model depicted in figure 2. Later, we will extend the model by adding more compartments.

For the random walk approach, the scenario we shall consider is that of an isolated population in a restricted spatial area, e.g., an island, a boat [11], or a city surrounded by walls. Specifically, we model the local geography with a 2-dimensional, rectangular lattice composed of $n_x \times n_y$ equally spaced nodes (figure 3). Each node represents a specific location, and during a simulation people move randomly between neighbouring nodes.

Each time step we shall require that all walkers move completely randomly (i.e., with a 25 percent chance) in one of the following four directions: North, South, East, or West. That is, a walker goes *either* in the x-direction *or* in the y-direction, but not both at the same time (diagonal moves are prohibited). After walkers have moved, each human present at a location will "encounter" every zombie at the same location and risk being bitten, scratched and exposed to the zombie virus. All zombies have the same probability, q, of infecting a human.

Unless otherwise is noted explicitly, we will set $n_x = n_y = 50$ and assume there are $N = 683$ people in the population. Also, the probability of getting infected is by default assumed to be $q = 0.9$.

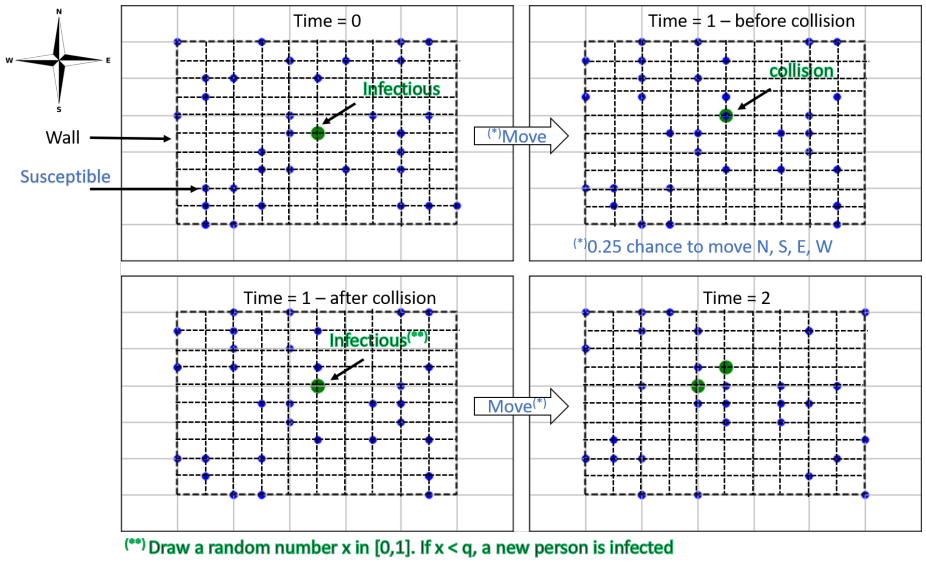

Figure 3: Example illustration of walkers moving on a lattice during one time step. In the plot, q is the probability that a walker is infected.

Task 1.(The most comprehensive task!)

- Write a class (or function) that can be used to conduct a single simulation of the random walk algorithm for the SZ-model.

Read this before you start coding.

We recommend that you look at *all* exercises before starting to code, because it is expected that the final simulator you hand in can be used for every scenario you model; that is, you will lose points if you make multiple solvers. Since your program could become quite large, it is crucial that you include sufficient documentation to your code. You can go a long way towards this goal by choosing intuitive names for classes, functions, and variables

For this particular exercise, you will get full marks as long as the SZ-model works as intended. At minimum, your implementation of the SZ-model has to perform the following steps:

1. At $t = 0$, place walkers randomly on the $n_x \times n_y$ rectangular lattice.
2. For each time step, update the position of all walkers, then:
   (a) Check if they are at a legal position; for those that are not, revert back to previous position.
   (b) For each S-Z encounter, check whether the healthy person was infected. If yes, update the disease state of the healthy individual from S to Z.
3. At any given time, calculate the total number of humans and zombies.
4. Keep track of the history of the number of walkers in each state since $t = 0$ (e.g., by storing these numbers in a set of arrays).

It is also a very good idea to create a function that plots the current location of all walkers on the "map" with a scatter plot. Use different colors to distinguish between walkers of different type (e.g., humans, zombies, dead). However, such a function should not be automatically invoked at each time step, because that will clutter your notebook with figures. A good default could be to plot every n-th timestep where n is "large enough", or to require the user to specifically input specific time steps at which to visualize the map.

If you want, you can place your simulation class (or function) into a separate .py file, and simply import it at the top of your Jupyter notebook.

Consult the Appendix for further help.

The appendices at the back contain suggestions for how you can structure your code so that it becomes easy to extend your model with additional compartments. In particular, Appendix B contains tips on how to make

an efficient implementation.The random walk simulations are very CPU-intensive, thus speed matters in this project!

Question 1: Assume that there is a single zombie initially, $Z(0) = 1$, what is the fate of the zombie and human population? (Figure 4 gives an indication of what we expect as an answer.)

Possible approach:

- Run the (default) random walk SZ-model forward in time repeatedly; at least 100 times. For each run, take 300 time steps. (If your code is fast, you should experiment with running the code even more times, and longer if necessary)
- For each time step, calculate 1) the sample mean and 2) the sample standard deviation of the number of humans (S) and zombies (Z) in the population.
- Create a figure showing the expected time-development of the two populations. Make sure to include the computed uncertainty in the figure.

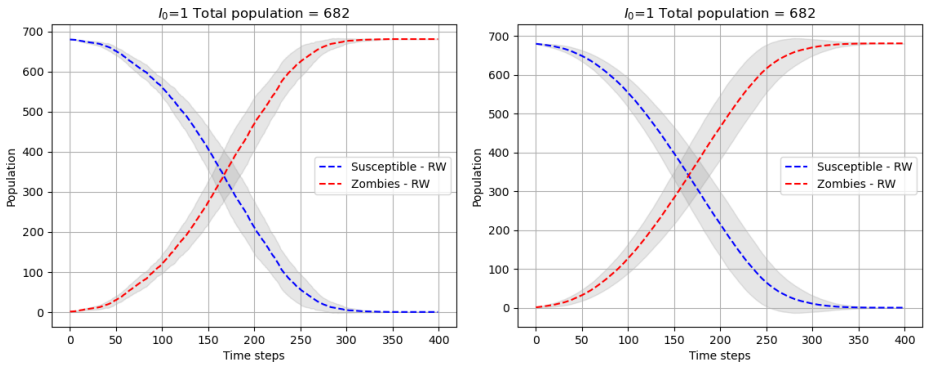

Figure 4: Left: average of 10 simulations of the random walk SZ-model. Right: average of 100 simulations. The gray shaded lines represent plus/minus one standard deviation.

Coding tip: To visualize the uncertainty, the above plots used the `fill_between`. function in `matplotlib`.

Question 2: If we instead started with $Z(0) = 10$, how does the time evolution change?

3 Exercise 2: Compare random walk and ODE-based models

The ODE-based, deterministic SZ-model is:

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

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

where $S(t)$ is the number of humans (susceptible) at time t, $Z(t)$ denotes the number of zombies and $N = S(t) + Z(t)$ is the total population size. β can be viewed an effective contact (disease transmission) rate; the total number of effective contacts made by any individual per unit time, multiplied with the probability of infection. The actual rate at which susceptible individuals become infected are given by the product $\beta(t)Z(t)/N$, sometimes referred to as the *force of infection*. See Appendix 9 for a detailed mathematical derivation of the SZ-model.

If we assume a constant β, the analytical solution is

$$Z(t) = \frac{N}{1 + \frac{S_0}{Z_0}\exp(-\beta t)}\,, \tag{3}$$

where $S_0 = S(0)$ is the number of healthy people at $t = 0$, and $Z_0 = Z(0)$.

How to interpret β?

If we assume that β is constant, we are making several *very strong* assumptions:

- Humans make the same number of contacts regardless of the population size.
- Humans make the same number of contacts independent of time.
- The probability of becoming infected never changes.

In reality, β is strongly time-dependent and accounts for a lot of biomedical, physical, and sociological factors. For example, in the beginning of an outbreak, β is likely to be large, because people might not yet understand the severity of the situation, or they may be in denial. As people start to realize the danger and fight back against the disease, β will most likely decrease. However, as we have seen with COVID-19, new strains of the virus may appear and cause transmission rates to go up again. Another complication is that people perceive the threat from the virus very

differently, and at least in some countries, this seems to be influenced by political factors.

Question 1: How does your Monte Carlo random walk prediction compare with the continuum (ODE) model in equation (3)? (To answer this question you need to estimate β from the random walk simulation and apply your estimate of β in equation (3) and compare the two simulations).

How to estimate β: Consider the random walk simulations conducted in Exercise 1, Part 1 with $Z(0) = 10$. We wish to find a representative β based on the results ("observations"). For a given simulation, we combine equation (1) with a first order approximation of the derivative to yield

$$\beta_n \cdot \Delta t \approx -\frac{(S(t_n) - S(t_n - \Delta t))N}{S(t_n)Z(t_n)} = -\frac{(S_n - S_{n-1})N}{S_n Z_n}\,, \tag{4}$$

where β_n is an estimate for $\beta(t)$ at the end of the n-th time step, $t_n = n\Delta t$, $n = 1, 2, \ldots$.

Possible approach to get representative β:

- For each run of the random walk model and every time step $n >= 1$, calculate $\beta_n \Delta t$ using equation (4).
- Use the resulting 2D array of values to find a single, representative (constant) β. You could to this, e.g., by calculating an appropriate mean or median value.
- Apply the analytical SZ-model, equation (1), with the β you just found. Compare it to the mean number of infected individuals from the random walk simulations.

Alternatively, you could try to construct a non-constant function $\beta = \beta(t)$ based on the random walk data. However, then you need to find another solution to the ODE-system (1)-(2), because equation (1) is no longer valid.

Remark on the time scale in the random walk algorithm.

The time scale is determined from the number of susceptible-infectious encounters per time unit. Thus, in the random walk simulations we can only determine the product $\beta\Delta t_{\mathrm{RW}}$. If we have data available on the spread of the disease, we can use the data to determine β after which we can estimate Δt_{RW}.

4 Exercise 3: Scenario what happens in Sokndal and Dirdal?


Figure 5: A view from Sokndal (left) and Dirdal (right).

Sokndal and Dirdal are two small villages in Rogaland. A group of 10 scientists from each of these places went to an international conference on numerical methods in Haiti[1]. During the conference, an excursion was arranged to a rural area, and a strange tomato salad consisting of, among other things, pufferfish venom, was served [5]. After arriving back in Norway, both groups of scientists got a fever, stopped eating, and subsequently started to behave suspiciously ...

Dirdal has about 683 inhabitants, and for sake of simplicity we will assume that the city center of Sokndal has a similar amount of 683 inhabitants [1].

At a certain point the inhabitants in the two villages can no longer ignore what is going on, and to avoid the coming zombie apocalypse, measures are taken. However, the time of realization and call for arms are different in the two villages.

Sokndal versus Dirdal?

In Sokndal, the average life expectancy of a zombie is about 24 hours. In Dirdal, it is about 48 hours. One explanation for this is that the inhabitants of Dirdal are more tolerant of unorthodox behavior, with the consequence that it takes more time for them to discover the zombies. However, anecdotal evidence suggests that people in Sokndal behave in quite strange ways also, as might be inferred from the lyrics of their local singer-songwriter Tønes.

[1] Maybe the most unlikely part of this project is that small villages in Norway have this many mathematicians ...

Question 1: Assume an average life expectancy of zombies of 48 (hours) and 24 (hours)[2] in Dirdal and Sokndal respectively. What happens in the two villages?

Question 2: For future zombie attacks: What is approximately the highest life expectancy of zombies, where humans still survive?

How to extend your code: Zombies have a certain life expectancy due to being attacked by planes, cars, guns or simply that they die due to stupidity. The key point is that they do not necessarily die because of human-zombie interaction, thus over the time of simulation a zombie can die without being close to another human.

- Extend your random walk implementation to handle the zombie deaths. By default, assume that no zombies dies, $p_{\text{death}} = 0$, thus recovering the SZ-model.

Coding tip: At each time iteration, and for each zombie, pick a random number, $x \in [0, 1]$, if $x < p_{\text{death}}$, update the state of the zombie to a dead state. (If you make plots at regular intervals, it looks cooler if the dead zombies do not move, and that they have a different color.)

Remember to take advantage of vectorization, you can draw a random number between zero and one for all zombies in one go.

```
#pseudo code
for i in range(Nstep): #Nstep: total time steps in one simulation
    move_walkers()
    set_back_illegal_moves()
    check_for_infection()
    check_if_zombies_dies()
```

Possible approach to answer the questions: Run the Sokndal case with $p_{\text{death}} = 1/24$, and Dirdal with $p_{\text{death}} = 1/48$.

- What happens in the two villages? You need to run multiple simulations and present the average and standard deviation to get a reasonable picture.
- Play around with other values of p_{death}, what is approximately the highest life expectancy of zombies, where humans still survive?

5 Exercise 4: Compare your results with SZR model (OPTIONAL!)

Question 1: How does your predictions in the previous exercise compare with a continuum (ODE) model?

[2]Note that we implicitly assume that one iteration in the Monte Carlo simulation is 1 hour.

To get you started: The corresponding ODE model is

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

$$\frac{\mathrm{d}Z(t)}{\mathrm{d}t} = \beta(t) \cdot \frac{S(t)Z(t)}{N} - \frac{1}{\tau_{\text{death}}} \cdot Z(t) \tag{6}$$

$$\frac{\mathrm{d}R(t)}{\mathrm{d}t} = \frac{1}{\tau_{\text{death}}} \cdot Z(t)\,, \tag{7}$$

where $R(t)$ is the number of "Removed" individuals or dead zombies at time t and τ_{death} represents the mean life time of zombies. If $\tau_{\text{death}} \to \infty$, a zombie lives forever, and equations (5)-(7) reduce to the SZ-model.

Possible approach:

- As before estimate β and τ_{death} from the random walk simulations.
- To get representative results, make sure you conduct many model runs. Plot the average number of susceptible, zombies and dead (removed) as a function of time. Visualize the uncertainty in your plots.

In figure 6 a typical result is shown, when values for β and τ are used in the ODE model.

The following code provides an implementation of the deterministic SIR-model (along with necessary package imports, which you should place at the top of your notebook):

```
import scipy as sp
import scipy.integrate

def SIR_model(t, *, beta, tau_s, N=683, I0=1, R0=0):

    def rhs(X, t):
        S, I, R = X
        return [-beta*S*I/N, +beta*S*I/N - I/tau_s, +I/tau_s]

    X0 = [N-I0-R0, I0, R0]
    sol = sp.integrate.odeint(rhs, X0, t)
    return sol
```

To estimate τ_{death} from the simulations you can do

$$\frac{\Delta t}{\tau_{\text{death}}} = \frac{R_n - R_{n-1}}{Z_n}, \tag{8}$$

where R_n is the number of dead zombies at time t_n.

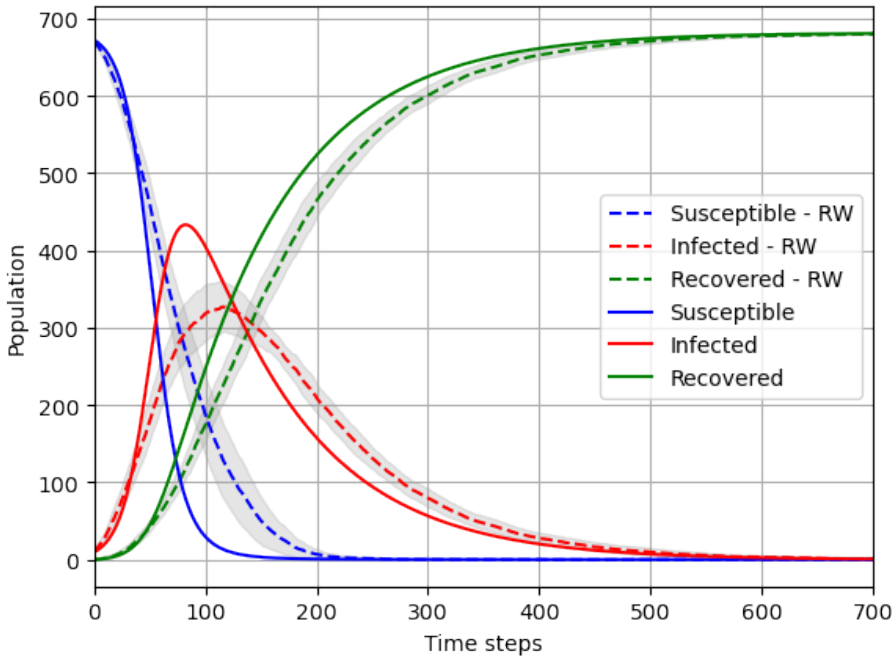

Figure 6: SZR Random walk model and comparison with ODE compartment model.

6 Exercise 5: Implement your own scenario

For the final part of the project, you are going to implement your own scenario. Below are some possible avenues for you to explore:

- Let $p_{\text{death}} = 0$ and the only way to remove zombies is by human zombie interactions. E.g. whenever a human survives an attack, the zombie die. Here you can also decrease the probability of that human to be infected at the next encounter (by e.g. reducing the probability of infection to 50%). To make this work you need an array describing infection probability for each human.
- Let some people be immune to the disease, and investigate how large part of the population should be immune for most to survive a zombie apocalypse. (Here you need to keep $p_{\text{death}} > 0$)
- Distinguish between old and young people. Let older people have a larger probability of dying. You can also let the old population have a probability of not moving at all during a time step, and/or to only move in a selected area (presumably, near "home").

- For those that have seen Walking Dead, you might try to add that those people that are good at killing zombies also get more dangerous and kill other humans.

Clearly, there are infinitely many choices you may make when it comes to model features, both individually and in combination. Therefore, it is very important that you state your assumptions carefully when presenting your scenario, and that you document your code accordingly.

It is also crucial that you illustrate your findings with one or more figures, and that you discuss how your model scenario compares to the previously investigated cases.

If you are able to relate your scenario to *the real world* (e.g., by comparing to relevant empirical data or movies), you can score some bonus points on this exercise!

7 Appendix A: How to implement the random walk?

As in project 2, we strongly recommend that you create a single simulator class responsible for running your model from start to finish. This allows different class functions to re-use the same variables, instead of having to pass them around all the time. At the same time, you avoid using global variables, which can be a source of hard-to-find bugs, and make your program hard to understand. Another advantage of using classes is that it becomes trivially simple to conduct parallell simulation runs, using either identical or variable model input parameters, which will be needed for this project.

Below is a suggestion for how you may start writing the `__init__` method of such a class:

```
class RandomWalkEpidemicSimulator:
    """
    Class used to model the spreading of a contagious disease in a
    population of individuals with a 2D random walk.

    Each walker has a disease state which is represented by an
    integer Enum. Also, a set of integer (x, y)-coordinates are
    stored for each walker. The possible coordinates are:

        {0, 1, ..., Lx-1} in the x-direction
        {0, 1, ..., Ly-1} in the y-direction

    It is only possible to move North, South, East, or West. If a
    walker attempts to move outside of the physical domain, nothing
    happens (i.e., a "bounce-back boundary condition" is enforced).
    """
    def __init__(self,
                 population_size,
                 no_init_infected=1,
                 nx=50,
                 ny=50,
                 q=0.9):
```

```
        """
        :param population_size: The total number of people (N).
        :param no_init_infected: The number of infected people at t=0.
        :param nx: The number of lattice nodes in the x-direction
        :param ny: The number of lattice nodes in the y-direction.
        :param q: The probability of infection (0 <= q <= 1).
        """
        self.N_ = population_size
        self.I0_= no_initially_infected
        self.nx_= nx
        self.ny_= ny
        self.infection_probability_ = q
```

As you work through the project, you will gradually add more content to the class; inside the `__init__` function, as well as in other functions.

Position of Walkers. At each time step, we need to know the positions of all walkers. This becomes simple if we store the walker (x, y)-coordinates in a 2D NumPy array. To generate random starting positions, we simply draw one x-coordinate and one y-coordinate for each walker:

```
self.Walkers_ = np.random.randint(0,
                                  [self.nx_, self.ny_],
                                  size=(self.N_, 2))
```

To understand what the code does, choose some small values for n_x, n_y, and N, and inspect the result.

Move walkers. It is important that the walkers move *randomly*, 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, and to say, e.g.:

- If `u==1`, move East: add $[1, 0]$ to the (x, y)-coordinates,
- If `u==2`, move North: add $[0, 1]$,
- If `u==3`, move West: add $[-1, 0]$,
- If `u==4`, move South: add $[0, -1]$.

As an example, suppose we have five walkers and moreover that we have already drawn the following (random) steps:

```
next_steps = np.array([[0, 1], [1, 0], [1, 0], [0, 1], [-1, 0]])
```

According to the scheme proposed above, these steps corresponds to the first and fourth walkers moving North, the second and third moving East, and the fifth moving South. Since all directions are stored in array of the same size as the population, we can update the walker positions simultaneously with a simple addition:

```
self.Walkers_ += next_steps
```

An alternative method is to update each of the four directions separately. A situation where this could be relevant is when we have drawn the random integers representing directions, but have not yet converted them into (x, y)-coordinate changes:

```
proposed_directions = np.array([2, 1, 1, 2, 3])
```

We can use boolean masking to find out which walkers move in which direction:

```
move_east = (proposed_directions == 1)
move_north = (proposed_directions == 2)
move_west = (proposed_directions == 3)
move_south = (proposed_directions == 4)
```

Finally, we update the coordinates:

```
Walkers_[move_east] += [1, 0]
Walkers_[move_west] -= [1, 0]
Walkers_[move_north] += [0, 1]
Walkers_[move_south] -= [0, 1]
```

Revert illegal moves (Bounce-back condition). Before checking for new infections, we need to make sure that none of the walkers are outside of the grid. We can achieve this by keeping track of walker coordinates from the previous time step: For each walker, we check whether that walker is at a legal position; if not, revert back to the old coordinates. This is often called a *bounce-back boundary condition*.

To extract the old (x, y)-coordinates, we can type:

```
self.Walkers_Old_ = self.Walkers_.copy()
```

Note the use of `copy()` here. If we had written

```
self.Walkers_Old_ = self.Walkers_  # Wrong code!!!
```

it would not work, because arrays are mutable objects in Python. This means that if `Walkers_` had been assigned directly to `Walkers_Old`, both variable names would point to the same underlying object in memory. 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 other situations, a `deepcopy()` operation might be needed; e.g., if we stored objects of a custom class in the `Walkers_` array (instead of integers).

To check for legal positions, the most obvious thing to do is to use a for loop:

```
# Note: Assumes we have already created a class (instance) function
#       "is_at_illegal_position" that checks whether the coordinates
#       of a given walker is valid.
for idx in range(self.N_):
    if self.is_at_illegal_position(idx):
        self.Walkers_[idx] = self.Walkers_Old_[idx]

# Remember to save the new positions for the next time step
self.Walkers_Old_ = self.Walkers_.copy()
```

However, this method is likely to be very slow. Instead you might want to use boolean masking to correct the position of each walker without using for loops. See Appendix B for more details.

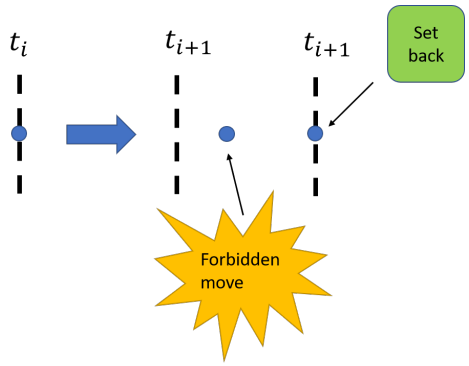

Figure 7: An illustration of the bounce-back boundary condition. The wall is located to the east, and a move in the east direction is illegal.

State of Walkers. We still have no information about the state (susceptible, infected, recovered, dead, etc.) of each individual walker. To handle this, we suggest introducing a set of *named integer constants*. These could for example be members of the simulator class (either class attributes or instance attributes), e.g.:

```
self.SUSCEPTIBLE_ = 0
self.INFECTIOUS_ = 1
self.RECOVERED_ = 2
```

Alternatively, they could be stored in a separate enumeration class:

```
from enum import Enum
class DiseaseState(Enum):

    SUSCEPTIBLE = 0
    INFECTIOUS = 1
    RECOVERED = 2
```

The following example shows how you can use this kind of approach in your code:

```
# First, let all walkers be in the susceptible compartment:
self.State_ = np.full(self.N_, self.SUSCEPTIBLE_)
# Next, change walker number 0, 1, 2, ..., I0-1 to be infectious:
self.State_[0:self.I0_] = self.INFECTED_
```

As the simulation is progressing, the `State_` array will be continually updated. At any given time, we can count the total number of infected, recovered etc. by

```
no_susceptible = np.sum(self.State_ == self.SUSCEPTIBLE_)
no_infectious = np.sum(self.State_ == self.INFECTIOUS_)
no_recovered = np.sum(self.State_ == self.RECOVERED_)
```

When using named integers (Enums), the code becomes much easier to read than if you work with hard-coded integers; it means you will never have to remember that `0=SUSCEPTIBLE` and `1=INFECTIOUS` etc.. Also, the approach can easily handle the addition of more compartments, or changing the numbering scheme later on. For instance, suppose you want to use to the following numbering instead:

```
self.SUSCEPTIBLE_ = 0
self.EXPOSED_ = 1
self.INFECTIOUS_ = 2
self.RECOVERED_ = 3
self.DEAD_ = 4
```

If you wrote your code referring directly to the integers, you would have to change each occurrence of `1` to `2`, and each occurence of `2` to `3`. Obviously, this is much more error-prone than the Enum-approach.

Checking for new infections. The "collision step" is crucial for performance. A naive implementation would be something like the following:

```
def collide_extremely_slowly(self): # note: class (instance) function

    # Get (x,y)-coordinates of susceptibles & infectious people
    S_coord = self.Walkers_[self.State_== self.SUSCEPTIBLE_]
    I_coord = self.Walkers_[self.State_ == self.INFECTIOUS_]

    no_infected = len(I_coord)
    for infected_idx in range(no_infected):
        for walker_idx in range(self.N_):
            walker_is_susceptible = (self.State_[walker_idx] == self.SUSCEPTIBLE_)
            infected_at_same_location = np.all(I_coord[infected_idx] == Walkers_[walker_idx])

            if walker_is_susceptible and infected_at_same_location:
                q = np.random.uniform(0, 1)
                if q < self.infection_probability_:
                    self.State_[walker_idx] = self.INFECTIOUS_
```

With the above approach, we are doing a lot of superfluous checks, every single time step of every simulation. In the best case scenario, we are checking each walker once every time step (this happens when there is a single infected person). In the worst case scenario, when everyone is infected, we are checking N^2 combinations of walkers. With $N = 683$, this means we are doing 466489 comparisons each and every time step! Obviously, this is not feasible.

The strategy of starting by looping over infected individuals is probably smart, however; at least initially, when there are few of them. However, you will need to avoid doing pointless checks.

8 Appendix B: Speeding up the code

To save time, you should only re-run simulations when you have to. Be careful that you do not perform unnecessary tasks inside the main simulation loop; for example, most plots and statistical calculations should be done after finishing the simulations, outside of the main solver. To be able to do that, you will need to store selected simulation results. It might be helpful to store some data into text files for later retrieval and post-processing.

That said, a good rule is to *never optimize code too early*; first, make sure the implementation works as expected, without thinking too much about performance. Later on, you might find out that the code runs too slowly and then you can try to fix that. Below are some tips on what could be improved.

- It is important to avoid loops whenever possible. For instance, drawing random numbers one at a time inside a loop tends to be very slow:

```
for k in range(0, N):
        x = np.random.uniform(0, d)
        y = np.random.uniform(0, d)
```

Instead, take advantage of built-in functionality in Numpy to draw all the numbers at once:

```
x = np.random.uniform(0, d, size=N)
y = np.random.uniform(0, d, size=N)
```

- When locating walkers at illegal positions, or finding positions where infectious and susceptible individuals meet (this step is crucial for speed!), consider using `numpy.where`.
- You can also use boolean masking, for example:

```
too_large_x = self.Walkers_[:, 0] >= self.nx_
too_small_x = self.Walkers_[:, 0] < 0
wrong_x = np.logical_or(too_large_x,too_small_x)
# alternatively you can use
wrong_x = too_large_x | too_small_x  # boolean "OR" applied elementwise
```

- Another example: Suppose we have created a boolean array of size `N_` called `at_illegal_pos`, where an entry is True iff the corresponding walker is outside the grid. Then, we can correct the illegal moves in a single line without a loop:

```
self.Walkers_[at_illegal_pos] = self.Walkers_Old_[at_illegal_pos]
```

- Numba translates python functions to optimized machine code, and might be something to look into.

A final tip (that might prove wrong)

- While it is often smart to use classes, avoid creating too many of them. For example, it might be tempting to represent individual walkers with a class, but our guess is that this will slow down the code considerably unless one is very careful. Accessing a class and its members adds extra overhead, which is why we prefer the approach of using arrays to hold information about the walkers.

9 Appendix C: Derivation of SZ-model

To derive equation (1) in the main text, we start by making some observations:

- During each time interval Δt, a certain number of individuals will come into contact with each other.
- We only care about susceptible (human) - zombie (infected) encounters, because that is the only scenario in which the number of infected people can increase.
- Whenever a healthy person meets an infected person, there is a certain probability that the healthy person becomes infected.

We shall take our imagined population to be *well mixed*, meaning that pairs of individuals interact with equal probability. Let $\mathcal{C}(N)$ denote the rate at which *any* individual in the population contacts *any* another individual, i.e., the average

number of contacts made per unit time. We calculate the change in the healthy population from time t to $t + \Delta t$ from

$$S(t + \Delta t) - S(t) = -\mathcal{C}(N) \cdot \Delta t \cdot p \cdot q \cdot S(t)\,, \tag{9}$$

where p denotes the conditional probability that a given contact is between a susceptible and infected individual, and q is the probability that such an encounter leads to disease transmission. Because of the well-mixed condition, we set $p = Z(t)/N$. The remaining challenge is to estimate $\mathcal{C}(N)$ and q. By merging the two factors into a single parameter, $\beta = \beta(t)$, we get

$$S(t + \Delta t) - S(t) = -\beta(t) \cdot \Delta t \cdot \frac{S(t)Z(t)}{N}\,, \tag{10}$$

Finally, by dividing by Δt and letting $\Delta t \to 0$, we obtain the following ordinary differential equation (ODE):

$$\frac{\mathrm{d}S(t)}{\mathrm{d}t} = -\beta(t) \cdot \frac{S(t)Z(t)}{N}\,. \tag{11}$$

Similarly, the evolution of the sick population is given by:

$$\frac{\mathrm{d}Z(t)}{\mathrm{d}t} = +\beta(t) \cdot \frac{S(t)Z(t)}{N}\,. \tag{12}$$

This last equation can also be derived at once from the relation $N = S(t) + Z(t)$.

10 Guidelines for project submission

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.
- Avoid duplicating code! If you find yourself copying and pasting a lot of code, it is a strong indication that you should define reuseable functions and/or classes.

- 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 very 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.

References

[1] Store Norske Leksikon. https://snl.no. Accessed: 2023-10-12.

[2] Jennifer Badham, Edmund Chattoe-Brown, Nigel Gilbert, Zaid Chalabi, Frank Kee, and Ruth F. Hunter. Developing agent-based models of complex health behaviour. *Health & Place*, 54:170–177, 2018.

[3] Norman TJ Bailey. The Simulation of Stochastic Epidemics in Two Dimensions. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 4, pages 237–257. University of California Press Berkeley and Los Angeles, 1967.

[4] 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.

[5] Wade Davis. The serpent and the rainbow, 2010.

[6] Douglas Kelker. A Random Walk Epidemic Simulation. *Journal of the American Statistical Association*, 68(344):821–823, 1973.

[7] William Ogilvy Kermack and Anderson G. McKendrick. A Contribution to the Mathematical Theory of Epidemics—i. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 115(772):700–721, 1927.

[8] Cliff C. Kerr, Robyn M. Stuart, Dina Mistry, Romesh G. Abeysuriya, Katherine Rosenfeld, Gregory R. Hart, Rafael C. Núñez, Jamie A. Cohen, Prashanth Selvaraj, Brittany Hagedorn, et al. Covasim: an agent-based model of COVID-19 dynamics and interventions. *PLOS Computational Biology*, 17(7):e1009149, 2021.

[9] Liliana Perez and Suzana Dragicevic. An agent-based approach for modeling dynamics of contagious disease spread. *International Journal of Health Geographics*, 8(1):1–17, 2009.

[10] S. Triambak and DP Mahapatra. A random walk Monte Carlo simulation study of COVID-19-like infection spread. *Physica A: Statistical Mechanics and its Applications*, 574:126014, 2021.

[11] Wikipedia. COVID-19 Pandemic on Diamond Princess. https://en.wikipedia.org/wiki/COVID-19_pandemic_on_Diamond_Princess, 2021. (Accessed on 19/11/2021).