6.S083 / 18.S190 Spring 2020: Problem set 4

Submission deadline: Tuesday April 28, 2020 at 11:59pm.

In this problem set, we will look at the simplest epidemic model in which we take **space** into account.

The simplest way to do this is to use **discrete space**: each agent lives in a cell of a square grid. For simplicity in the coding we will not allow more than one agent in any one grid cell, but this means that we need to carefully design the rules of the model to respect this.

The effect of space is thus to *restrict* the set of other agents with which a given agent can interact at any one time. (Compare the previous Problem Set, in which any agent could interact with any other.)

Exercise 1: Moving in 2D

In this exercise we will implement a random walk on a 2D lattice (grid).

1. Define an abstract type `AbstractWalker`.
2. Define an abstract type `Abstract2DWalker` that is a subtype of `AbstractWalker` (using `<:`).
3. Define an immutable type `Location` that contains integers `x` and `y`.
4. Define a `Walker2D` type that is a subtype of `AbstractWalker2D`. It should contain a field `position` that is a `Location` object.
5. Check that Julia automatically provides a constructor function `Walker2D(position)` that accepts an object of type `Location`.

   Construct a `Walker` located at the origin.
6. Write a new method of the constructor for `Walker2D` that accepts two integers, x and y, i.e. `Walker2D(x, y)`.
7. Write a function `make_tuple` that takes an object of type `Location` and returns the corresponding tuple `(x, y)`.

In the following questions, the functions should take an object of type `AbstractWalker2D` (or you can just leave them untyped).

8. Write a getter function `pos` that returns the position as a `Location` object.
9. Write a setter function `set_pos!` that sets the position to a given location `l`.

10. Write a function `jump` that returns a possible new position for a walker after a 2D jump as a `Location` object. This function should *not* modify its argument, and hence does not have a `!` in its name.

    Jumps are equally likely in the directions right, up, left and down.

    A nice way to implement this is to make a tuple `neighbours` of possible destinations.

11. Write a function `jump!` that moves a walker to a new position. What arguments does the function need? Use your `jump` function to write `jump!`.

12. Write a function `trajectory` that calculates a trajectory of a 2D walker of length N.

13. Plot 10 trajectories of length 10,000 on a single figure, all starting at the origin.

    Note that `Plots.jl` can accept a `Vector` of `Tuple`s, i.e. (x, y) pairs, as the coordinates to plot.

Exercise 2: Making agents move

In this exercise we will combine our `Agent` type from Problem Set 3 with the 2D random walker that we just created, by adding a position to the `Agent` type.

1. Define a mutable type `Agent` that is a subtype of `AbstractWalker2D` from Exercise 1, since it will behave like a random walker and lives in 2D.

   `Agent` should contain a `position` of type `Location`, as well as a `state` of type `InfectionStatus`.

   (For simplicity we will not require a `num_infected` field, but feel free to do so if you would like.)

2. Agents will live in a box of size `L` centered at the origin. We need to decide (i.e. model) what happens when they reach the walls of the box (boundaries), i.e. what kind of **boundary conditions** to use.

   One type of boundary condition that is relatively simple to implement are **reflecting boundary conditions**, as follows:

   Each side of the box is a reflective mirror. We can model this using "bounce-back": if the particle tries to jump beyond one of the boundaries, it hits a springy wall and bounces back to the *same* position that it started from. That is, it **proposes** to take a step, but "realises" that it is blocked in that direction, so just stays where it is instead for that step.

   Use the `jump` function from before (that proposes a new position) inside a new method of the `jump` function for an `Agent` that also accepts a size `L` and implements reflecting boundary conditions. It returns a `Location` object representing the new position inside the grid.

3. Check that this is working by drawing a trajectory of an `Agent` inside a square box of side length 20, using your function `trajectory` from Exercise 1.

   You should draw the boundaries of the box and also a trajectory that is sufficiently long to see what happens at the boundary, but not so long that it fills up the box.

Exercise 3: Spatial epidemic model – Initialization and visualization

We now have all of the technology in place to simulate an agent-based model in space!

For simplicity we will impose in the model that there is at most one agent on each site at all times, modelling the fact that two people cannot be in the same place as each other.

We thus begin by creating an initial condition for N agents that satisfies this. Later we must make sure that the dynamics also respects this.

1. Write a function `initialize` that takes parameters L, the side length of the square box where the agents live, and N, the number of agents.

   It should build, one by one, a collection of agents, by proposing a position for each one and checking if that position is occupied. If the position is occupied, it should generate another one, and so on until it finds a free spot.

   You may create additional functions to help with this if you find it useful to do so.

   The agents should all have initial status `S`, except for one of them, e.g. the first in the list, which has initial status `I` – i.e. it is the only source of infection.

   It should return the `Vector` of `Agent`s.

2. Run your initialization function for $L = 10$ and $N = 20$.

3. Write a function `visualize_agents` that takes in a collection of agents as argument. It should plot a point for each agent, coloured according to its status.

   You can use the keyword argument `c=cs` inside your call to the plotting function to set the colours of points to a vector of integers called `cs`. Don't forget to use `ratio=1`.

4. Run the function to visualize the initial condition you created.

Exercise 4: Spatial epidemic model – Dynamics

1. Write a function `step!` that does one step of the dynamics of the model. It takes as parameters L, p_I and p_R. This combines what we did in the last Problem Set with the 2D random walker above.

   The rules are as follows:

   - A single agent is chosen at random; call it agent i.
   - A new position is proposed for that agent.
   - If that new position is not occupied, the agent moves there.
   - If the new position *is* occupied, by agent j, then *neither* of them move, but they interact via the following rule:
   - If agent i is infected and agent j is susceptible then agent j becomes infected with probability p_I.
   - If agent i is infected, it recovers with probability p_R.

2. Make a small system and run the `step!` function a few times to check (by eye) that it's doing the right thing.

3. Make an interactive visualization to display the agents after each step, to again check visually that the implementation is correct.

4. Write a function `sweep!` that takes the relevant parameters and performs one sweep, i.e. N steps.

5. Write a function `dynamics!` that takes the same parameters as `step!`, together with a number of sweeps.

   Run the dynamics for the given number of sweeps.

   Save the state of the whole system, together with the total numbers of S, I and R individuals, after each sweep, for later use.

   You may need the function `deepcopy` to copy the state of the whole system.

6. Given one simulation run, write an interactive visualization that shows both the state at time n (using `visualize_agents`) and the history of S, I and R from time 0 up to time n. To do this, make two separate plot objects p_1 and p_2 and use the $hbox$ or $vbox$ function to put them together horizontally or vertically into a single plot.

7. Using $L = 20$ and $N = 100$, experiment with p_I and p_R until you find an epidemic outbreak. (Take p_R quite small.)

8. For the values of p_I and p_R that you found in [11], run 50 simulations. Plot S, I and R as a function of time for each of them (with transparency!).

9. Plot their means with error bars.

You should see a result that looks like all those plots that you've seen of the SIR model on the internet. (If you haven't seen any, then what are you waiting for – go and find some!) Except that they *never show you error bars*, even though error bars are clearly *of the utmost importance*.

Exercise 5 (Extra credit – or for summer vacation): Social distancing

We can use a variant of the above model to investigate the effect of the totally mis-named "social distancing"
(we want people to be socially *close*, but *physically* distant).

In this variant, we separate out the two effects "infection" and "movement": an infected agent chooses a neighbouring site, and if it finds a susceptible there then it infects it with probability p_I.

Separately, an agent chooses a neighbouring site to move to, moves there with probability p_M if the site is vacant. (Otherwise it stays where it is.)

1. When $p_M = 0$, the agents cannot move, and hence are completely quarantined in their original locations. Where can the disease spread in this case?

   Run the dynamics repeatedly, looking at which sites become infected. (Draw them!)

   How does this change as you increase the *density* $\rho = N/(L^2)$ of agents? Start with a small density.

   This is basically the **site percolation** model.

2. When we start to increase p_M, we allow some local motion via random walk. Investigate how this leaky quarantine affects the infection dynamics with different densities.