# 6.S083 Fall 2019: Problem set 3

Submission deadline: Friday November 15, 2019 at 11:59pm.

Submit a Jupyter notebook with your solutions to `sandersd@mit.edu`. Make sure the name of the file contains your first name and last name.

# Epidemic model

In this problem set, we will implement a stochastic epidemic model using interacting random walkers, which can be thought of as a type of Monte Carlo method. Hopefully you will agree that the end result is worth it!

Related models are common in physics, chemistry, biology, sociology, etc. It is often challenging to say anything mathematically about models like this.

## Exercise 1: 2D random walker

Firstly, let's use the ideas about objects from class to define a 2D discrete random walker that lives on the integer lattice in 2D.

1. Make an abstract type `AbstractWalker`. (This replaces the type `Walker` from class.)
2. Make an abstract type `Abstract2DWalker` that is a subtype of `AbstractWalker` using `<:`.
3. Make an immutable `Location` type that contains integers `x` and `y`.
4. Make a `Walker2D` type that is a subtype of `AbstractWalker2D`. It should contain a field `position` that is a `Location` object.
5. Make a new method of the constructor for `Walker2D` that accepts integers $x$ and $y$.
6. Make a method of `Base.Tuple` (i.e. overload the existing function `Tuple` from the `Base` module) that converts a `Location` into a tuple.
7. Make a function `pos` that returns the position of `AbstractWalker2D` as a `Location` object.
8. Make a function `set_pos!` that sets the position of an `AbstractWalker2D` to a given location `l`.
9. Make a function `jump` that takes a walker and returns a possible new position for it after a 2D jump. [This function should *not* modify its argument, and hence does not have a `!` in its name.] You can re-use (and modify if necessary) the method from PS2, Q.6

10. Make a function `jump!` that moves the walker to the new position. Use `jump` to write `jump!`.
11. Use the function `walk!` from class to calculate the trajectory of the 2D walker. 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 tuples of $(x, y)$ pairs.

## Exercise 2: Agent type

In this and the following exercises, we will construct a model to simulate an epidemic of a rumour or illness in a population of individuals, or **agents**, that move around in space and interact.

The agents move like 2D discrete random walkers, but also have an **internal state** that can be one of `S` (**susceptible**: able to be infected), `I` (**infected**) or R (**recovered**). There is no birth or death.

Two individuals that meet at the same location infect one another with probability $p_I$, and each infected individual recovers with probability $p_R$ at each step.

This is an example of an **agent-based model**.

1. Julia, like many languages, has a data type called an **enumerated type**. Variables of this type can take only one of a set of pre-defined set of states; we will use this to model the possible internal state of an agent.

   The code to define our enum is as follows:

   ```
   @enum InfectionStatus S I R
   ```

   This defines the type `InfectionStatus`, as well as names `S`, `I` and `R` that are the only possible values that a variable of this type can take.

   - Define a variable `x` equal to `S`. What is its type?
   - Convert `x` to an integer using the `Int` function. What value does it have?

2. Define a type `Agent` that is a subtype of `AbstractWalker2D`, 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`.

3. Create agents with different locations and infection statuses to see what happens.
4. Agents live in a box of size `L`; we will use bounce-back boundary conditions. Rename the `jump` function from before to `basic_jump` and use it inside a new `jump` function for an agent that also takes a size `L` and implements the boundary conditions. It returns a `Location` object representing the proposed new position.

5. Check that this is working by drawing a trajectory of an `Agent` inside the box using `walk!`.

**Exercise 3: Initialization**

We require that there is at most one agent on each site at all times. Firstly we create 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 size 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 sole source of infection.

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.

   Use the option `c=cs` to set the colours of points to a vector of integers called `cs`. Don't forget to use `aspect_ratio=1`.

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

**Exercise 4: Single step of dynamics**

1. Write a function `step!` that does one step of the dynamics of the model. It takes as parameters $L$, $p_I$ and $p_R$.

   The rules for the dynamics are as follows:

   - A single agent is chosen at random to move; 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 according to the following rule:
   - If one of them is susceptible and the other is infected, the susceptible one becomes infected with probability $p_I$.

- If the agent that moves 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.

**Exercise 5: Run the dynamics**

Dynamics in Monte Carlo simulations is often thought of in terms of **sweeps**. One Monte Carlo **sweep** corresponds to running $N$ steps.

Thus each agent moves on average once per sweep. In this way, times become comparable for runs with different numbers of particles.

1. 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.
2. 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 $R$ 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.
3. Using $L = 20$ and $N = 100$, experiment with $p_I$ and $p_R$ until you find an epidemic outbreak. (Take $p_R$ quite small.)

**Exercise 6: Performance**

Your code should run pretty fast for small values of $N$, but will get slower as $N$ increases.

1. Where is the main inefficiency in your code? i.e. What is the most inefficient operation?
2. What computational complexity (approximate number of operations) does it have as a function of $N$?
3. How could you reduce this significantly, at the expense of using more memory? Would you have to duplicate information, i.e. store the same information in a different way, in order to do so? What data structure could you use? (You do *not* need to write a full implementation, just suggest how you might be able to do it.)