The Computing Challenge for Academic Year 2021/ 2022

This year the Computing Challenge revolves around **writing a code to study a simplified model of how virus spread in the world**, and how this process can be affected by different choices. In fact, your final task is to use your model to assess the relative capability of different policies to contain the spread of the virus!

Please note that, because this is a simplified model, **its results have to be taken with more than a pinch of salt!** However, we will try to build a model with all the minimal ingredients necessary so that at least a qualitative evaluation of the situation can be made. I will leave you to consider how more realistic models can be implemented, and what information would be required in this case.

The main ingredients of our model (our "world") are Countries and People. People can move both through countries as well as within them, and do differen things depending on their status. In our model, actions will occur randomly, in the sense that events can occur, or not, according to a certain probability, but the process is not deterministic.

Countries

A country (is an object that) has the following characteristics:

- It has a certain name
- It is divided in different regions
- It has a certain population inside it. These people are, at least initially, randomly distributed among the regions of a country.

In practice, from a spatial point of view we represent each country as an C = NxN matrix (where N is external input variables). Each cell C[i,j] in this matrix is a region and each region can host any number of people. Regions for which the sum of their matrix indexes `i+j` differs by at most 1 are called contiguous.

For each country, the cell `C[0,0]` is the airport region (see more below)

People

People are the main element of our model and are described by the following attributes.

- Vaccination status (either `True` if vaccinated, or `False` otherwise)
- Infection status (either `True` if infected or `False` otherwise)
- Quarantine status (either `True` if quarantined or `False` otherwise)
- Age (a number between 0 and 100)
- Mobility (a number between 0 and 1, given by `G( x, 30, 10 )` , where `G( x, y, z)` is the value of a Gaussian function evaluated at point `x` , with mean `y=30` and variance `z=10` , `x` being the Age of a person)

Evolution of the system

Each day, **all** the people in our world must perform at least one of the following actions (and potentially additional ones, see later):

- A healthy person (not infected) can move, die, (potentially) get infected or do nothing, with the same probability.
- A sick person (not infected) can move, die, (potentially) get cured or do nothing, with the same probability.

In practice, this means that **for each day and each person, you have to randomly decide which action occurs, based on their probability.** When we say that a probability for an action is pX we mean that, if we choose N actions, for large N the number of times we took a certain action should be `N*pX` (we have seen how to do that in Python!).

IMPORTANT (but subtle) NOTE: as you are about to see, what occurs for different actions depends on the state of the system. In particular, the evolution each day occurs according to the initial state of the system for that specific day. This is to avoid that actions on the same day affect each others, in which case the order in which they are performed (i.e., which people are picked first) would matter.

The rules for different actions:

1) **Getting infected**: a non-vaccinated and healthy person that is NOT quarantined can get infected with probability `pI = max( 1, R * N_active / N_people )` , where `N_active` is the number of infected people in a region that are NOT quarantined, `N_people` the total number of people in that region and `R` a parameter that controls how infectious a virus is.

If the person is vaccinated, this probability is reduced by a factor `pV = ( 1 - vE )` , where `vE` is the vaccine efficacy. A person automatically enters quarantine after 3 day it is infected, when the symptoms start to develop. Note that you should be able to easily change vE and pQ and potentially run simulations with different values. In other words, these values should not be hard-coded, but left as an input value to be provided externally.

2) **Moving**: a person that is NOT quarantined can move to any contiguous region, chosen randomly, with a probabity given by the value of the mobility. If a person does not move, it just remains where it is. If a person is in the airport region, the move include not only its neighbouring regions, but also all airports of all countries. Note that in our model people can only get out of their country via the airport. In other words, there are no borders that can be crossed by a simple move via contiguous regions because different countries do not border. In practice, our model is better at describing what would happen if each country was actually an island, or if the borders were completely sealed except via plane.

3) **Dieing**: a person can die with probability given by `pD = max( 1, 1 - exp( -Age / 50 )*Xm )` where `max( x, y )` is the function that returns the maximum value between `x` and `y`. `Xm = 1` for a non infected person and is instead above 1 (see later) for an infected person. As for other parameters, this should be provided as an input and not be hard-coded.

4) **Recovering from infection**: an infected person will recover from an infection with probability pR = 1 - exp(- x / 2), where x is the number of days passed since the infection. Remember that even if a person is not infected anymore, it will remain in quarantine until `nQ` days have passed since infection. However, a person that is quarantined cannot become infected until it exits quarantine.

Modifying probabilities: Policies

Testing the effect of different policies means in practice we need to modify the rules for our previous actions in a certain way, or add some actions.

0) **Increase vaccination**

Simply increase the vaccination rate to its maximum value (100%).

1) **Test + quarantine only**

The vaccination rate does not change but we simply add an extra action to those above: a person (infected or not), after doing any of the previous moves, will also be tested with probability `pT = 0.5` (but `pT` could be more generally any input value between 0 and 1). If a person is tested **and** is infected, it will enter quarantine. A person exits quarantine automatically after `nQ` days.

2) **Banning flights**

When this policy is implemented, on top of implementing also the testing and quarantining described in 1), even if people are in the airport they can only move to neighbouring regions, but not travel through countries. The vaccination rate in this case does not increase.

3) **Hard ban on mobility**

When this policy is implemented, on top of implementing also the testing and quarantining described in 1), people cannot leave their region, nor fly to different countries. In other words, their mobility drops to zero. The vaccination rate in this case does not increase.

The starting point

To start your simulation, you will first need to provide an initial status for your system. To observe what happens due to different policies, you should also set a base case, where **none** of the policies are implemented.

To set a starting point, you should:

- Deciding how many countries are in your world
- Decide an initial population in each country. This means not only setting how many people there are for each country, but also the characteris

In practice, you should make the code flexible so that we can start with any value, but for our simulations we will do the following:

- Our world has 4 countries
- Each country has 16 regions, i.e. `N = 4`
- Each country has 200 people. Initially, 10 people per country, chosen randomly, will be infected, whereas initially none is quarantined. The initial location of each person is also chosen randomly.
- The number of people vaccinated is 50% for each country.
- In terms of age, 20% of the people are (randomly) between 0 and 20 years old, 60% between 21 and 60 and 20% above 60.
- You should do simulations spanning all possible combinations of the values for the following parameters: 1) `vE` (vaccine efficacy): 40% and 80% 2) `xM` (mortality factor): 1 and 3 3) `nQ` (number of quarantine days): 2,5 and 10 4) `R` (infectivity parameter): 2 and 4

In practice, notice that there are 24 combinations and for each you will have to run the 4 different policies. Also, note that because the process is effectively random (each time you run a simulation, even with the same parameters, you will obtain different values!), **for each initial state you should run 4 different simulations, and then average the data between them.** This will also allow you to provide a confidence interval for your infection / death curves.

What do you have to do in practice

Evolve the world for 1 year (365 days) using the rule above, gather the data and analyse it.

In particular, by analysing we mean:

1) **Every 10 days, for each country save into a file the number of people still alive and the number of people infected**. The file name should be `country_number.txt` , country is the name of a country (whatever you choose it to be) and `number` is number of the day of the snapshot saved. You want to format `number` in such a way that it always contains the same number of digits, padding 0s to the left if necessary. For example, the file name `UK_00030.txt` contains a snapshot of the UK after 30 days of evolution.

2) **Use the data in 2) to plot two graphs**: one with the **average number of people that died** and one with the **average number of people that are infected** as a function of time. The average is over 4 simulations with the same parameters (as stated before), and you need to have a graph for each country in your world.

3) **Use the data produced to answer the question**: given the rules and the values used, **what policy is better at minimising the number of deaths, and which one is the worst**? Note that you can summarise this in a table, where you report the result for each combination of the parameters considered.

How to do it: some tips

When you are about to solve a problem using a computational approach, there are a few steps typically involved. Some are related to how to face a complex problem in general, others to the fact that you should be doing that as a group and not alone.

The general approach I would suggest you to use is the so-called "Divide and Conquer" approach, the (somewhat wrong) English translation of the ancient Romans motto *Divide et Impera* (https://en.wikipedia.org/wiki/Divide_and_rule). In practice, the idea is to apply a centuries-old military strategy, whose philosophy can be simplified in this words: When a problem seems to big to be tackled, first decompose it in smaller, simpler parts. In this way, these smaller subparts can be more easily solved, one by one, and then re-assembled them together to find the solution of the initial problem. This is a general strategy that might be used in many situations and is very common when dealing with complex scientific problems.

Let us translate this strategy into practical steps:

1. First, read the whole text of the problem once all together.
2. Read it a second time, to identify the different subparts in which it can be split. Each subpart can correspond to a function or, maybe, an object/class, with specific characteristics / methods. Ideally, you want each person in the group to code the solution for one of these blocks.
3. Before each one starts to do code for his subproblem, it is important that you coordinate together within the group and decide first which are the inputs and which are the outputs needed for each block, and their format. This step is usually helped by starting to write the solution as pseudo-code, i.e., as a series of instructions, without worrying to use the correct synthax of the Python language (or for that matter any programming language you might wish to use).

   Example of Pseudo-code (for making a "Tiramisu", a delicious Italian dessert (https://en.wikipedia.org/wiki/Tiramisu)):

   i) Prepare the cream [INPUT: eggs whites, egg yolks, mascarpone, sugar; INSTRUMENT: mixer; OUTPUT: cream]

   ii) Prepare the Lady Fingers (a type of biscuits) [INPUT: biscuits, coffee, rum; INSTRUMENT: coffee machine; OUTPUT: basis for the tiramisu]

iii) Assemble the cream and Lady Fingers together and top with cocoa [INPUT: Cream, Lady Fingers, Cocoa powder; INSTRUMENT: Hands OUTPUT: Tiramisu]

iv) Eat & Enjoy!

4. To give you some potential hint closer to what you will effectively have to do, you could decide that one person could take care of designing a class "person", one a class "country" (and maybe also a class "world"...you should decide which are needed, and which not, **these are only examples**), maybe with their own methods and data attribute. Another person could write a function that takes as input the parameters describing the system and evolves it applying the algorithm defined above and a last person could be the "architect" that assembles all the previous functions together, or that takes the output and plots the data...and so on.
5. It is **good practice to test each sub-part of the code independently** to check that it works as an isolated unit before assembling them together. This will make sure that if there is a problem (because, for example, you obtain a nonsensical output), you know where it is, which makes it easier to correct it. You might want to submit these intermediate test also as part of the code that will be marked.

NOTE: *there is not a single recipe for doing this and you will have to experiment a bit*. However, if you do not coordinate initially, it will be much harder to combine all the various parts later, especially because the input for one part of the code is the output for another, and they have to blend together efficiently.

Marking criteria

1. Correct use of functions and their implementation, as well as correct use of the data types introduced in the lectures. **20 out of 100 Marks**
2. Correct use of the control flow constructs introduced in the lectures. **20 out of 100 Marks**
3. Use of the appropriate numpy functionalities. For example, you should avoid to re-code something that is already present in the library for you (at least if it is something we have seen in the course!). **20 out of 100 Marks**
4. The code implements the various steps in a way that traslates this problem into the correct algorithm to solve it **20 out of 100 Marks**
5. The output of the code is correct **10 out of 100 Marks**
6. The style of coding is neat and allow for external users to easily read and understand the code. For example, all functions (whether isolated functions or methods attribute of a class) have a proper documentation describing what is their input and output, the various algorithms are commented to describe the most salient steps they implement, and so on **10 out of 100 Marks**.

Submission

For organisational reasons, **each single student must** submit the code via Blackboard, within the **deadline provided in the Student Handbook** (this should be about 2 weeks after the final Computing lecture). Together with the code, the submission must include the requested graphs and the files containing the data to plot them, as well as the answer to the question on policies (in whichever form you want, you can write it in words or summarise as a table).

IMPORTANT: If you do not have the Student Handbook or you have forgotten where it is, **please contact the Student Office regarding the deadline, NOT me because I do not know them!**.

Note 1: Exactly the same code must be submitted by all the people in the same group. This is because without submission we cannot input the mark into Blackboard, even if it means submitting effectively multiple copies of the same stuff.

Note 2: This is a group effort and there will be one single mark for the whole group, because all members will be considered equally responsible for the whole final product.