An SIR grid model of disease transmission Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/sir/sir.pdf

---

---

1 The coughing passenger

Suppose you are a passenger on a plane, and you hear someone two rows ahead of you who is coughing repeatedly. You might begin to worry, because an airborne disease is most likely to spread to nearby neighbors of an infected person. If it was a really long flight, and the disease developed really rapidly, then it might spread first from the coughing passenger to the passenger in front of you, and then to you, taking two steps of infection.

During the world wide influenza epidemic of 1918-1920, the United States Army housed hundreds of sick soldiers on a grid of beds in huge hospital wards. We might wonder how such an arrangement would permit the gradual spread of a new disease, starting at one infected patient, and spreading to the neighbors.

We might suppose we have an $m \times n$ grid of people, who will not be changing position uring the extent of our experiment. We start with one infected person, at some random position (i, j). According to our simplified disease model, the other people are "susceptible", that is, they aren't sick, but they could become

so. However, a susceptible person can only catch the disease if they are next to an infected person (left, right, in front or in back). We might assume that a susceptible person next to an infected person for a day has a probability p of getting infected during that time.

Now given what we have said so far, it seems certain that if we wait long enough, everyone in the group must get infected; the one infected person we start with must eventually infect all their immediate neighbors, who must eventually infect all their neighbors, and so on.

This is not a very realistic model of disease. So let's add the idea that an infection only lasts for d days, after which the person is no longer sick, and in fact becomes immune. An immune person cannot get sick again.

So our model starts with three types of people: $m \times n - 1$ susceptibles (S), 1 infective (I), and 0 recovereds (R). Over time, we expect the value of I to grow, but then it must eventually decrease to zero as the disease "burns out". The value of interest, then, is the number of people who became sick versus those who never got the disease. So we are interested in the value, at the end of the experiment, for the fractions $\frac{S}{m*n}$ and $\frac{R}{m*n}$. If everyone got sick, then these values will be 0 and 1, respectively, so the disease swept through the entire population. Higher values of $\frac{S}{m*n}$ indicate that the disease was less effective in spreading.

Using this model, and the value of $\frac{S}{m*n}$, we could also experiment with small changes to the problem, such as the location of the initial infected person, or the use of a vaccine to reduce the value of p, the transmissibility of the disease, or a treatment that reduces d, the duration of the disease, or some means of isolating infected persons so they can't transmit the disease to others.

Here is an outline of an algorithm for the SIR model, assuming we are given values for m, n, p, d. The current status is stored in the array P, where $P(i, j) =$

- 1, if person (i,j) is susceptible;
- -k, if person (i,j) is infected for k more days;
- 0, if person (i,j) is recovered;

We continually updated the array P day by day until there are no infected persons:

```
P = ones ( m, n )
i, j = indices of a random location in P.
P(i,j) = -d   # negative value; person will recover in d dayss
day = 0

while any P < 0

  day = day + 1

  for (i,j) in the grid

    Pnew starts as a copy of P

    if ( P(i,j) is susceptible )
      for each infected neighbor of P(i,j)
        if random value < p
          Pnew(i,j) = -d
    else if ( P(i,j) is infected )
      Pnew(i,j) = P(i,j) + 1  # one less day to be infected
    end
```

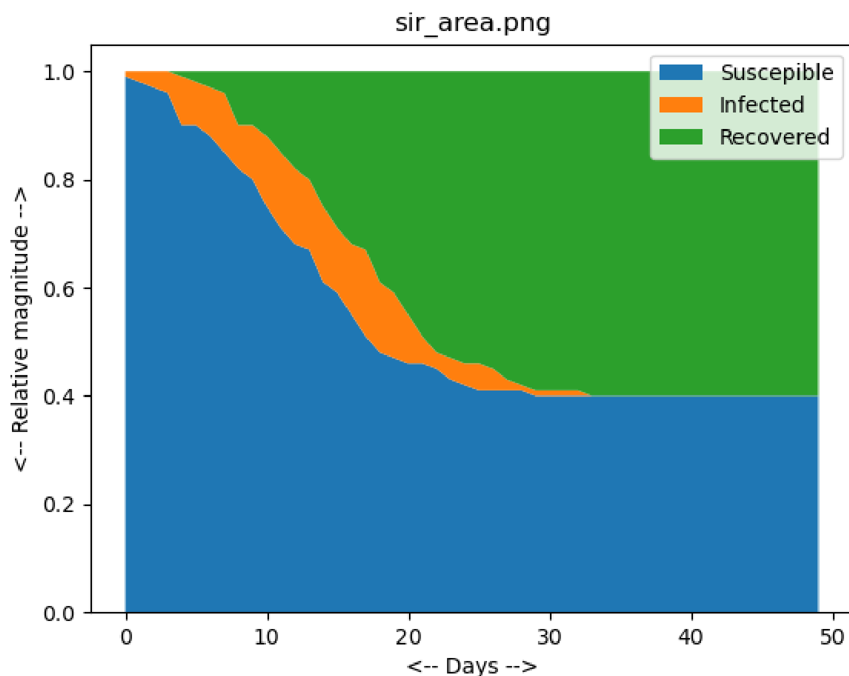
```
  end

  replace P by Pnew

end

print day, duration of epidemic
print sum ( P ) / m / n, proportion of population that did not get sick
```

Here is a plot of the variation in the values of S, I, R for a model in which at the end of the simulation, S = 0.14 and R = 0.86:



The behavior of S, I, and R for $m = n = 10, p = 0.2,$ $d = 4$.

What happens if immunity is not forever, but only lasts for a certain number of days? Are there situations in which a disease will come in waves?

2 Percolation

Oil drillers are familiar with peculiar structures of underground rock. Often, they are drilling through a material that is a jumbled combination of rock and hollowed-out cavities which might contain oil, gas, or water. Depending on the size, density, and frequency of these cavities, a particular underground "lake" of oil might extend for a few hundred feet, or for miles.

As another example, we might consider an object composed of two metals, one conducting and one not. We suppose the object involves many separate regions of each metal, which are touching, but not melted together. We know the approximate size of a typical "cell" or clump of each metal, and the proportion of the two metals. We ask for the probability that an electrical signal can pass from the top to the bottom

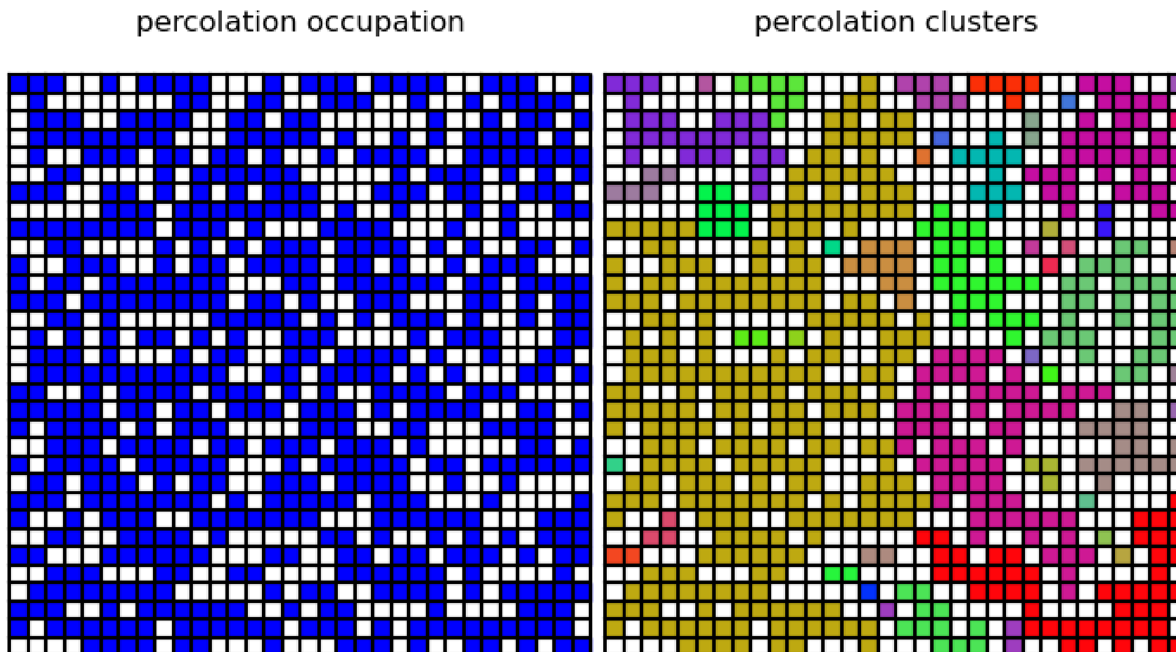
of the object. This will happen only if there is a some path, formed entirely of the conducting metal, that reaches from top to bottom.

John Hammersley proposed an approach for studying such problems. He imagined a porous stone such as pumice, which is made from volcanic activity. Suppose such a stone is submerged in a bucket of water. Would the center of the stone end up wet or dry? This depends on whether there is at least one microscopic channel between the surface and the center. If the water reaches the center, we say it has done so by *percolation*. This is the same word used to describe one method of making coffee, by having boiling water percolate through a layer of coffee grounds.

A simple 2d model of percolation can be constructed by setting up an $m \times n$ grid of square cells. We will allow some cells to be obstructions with status 0 (filled, or nonconducting, or solid) and others to be channels, with status 1 (open, conducting, or hollow). We will be looking for the existence of a path from top to bottom. To be a realistic model, the number and size of these cells would have to approximate those of a physical structure. For our learning purposes, though, we will be satisfied with rather coarse grids. We will look at several factors to vary in our problems:

- the total number of cells $m \times n$;
- the aspect ratio $\frac{m}{n}$;
- the probability p that a given cell is conducting;

Once we have chosen m, n, p, we can randomly set the status of each grid cell. The hard part now is to figure out whether the resulting pattern of conducting cells forms a path from top to bottom. If we plot the grid, we can usually spot a path if it exists, at least for our small example grids. We have to come up with a way for the computer to do the same thing, without the benefit of eyes.



The raw occupation grid, and the clustering pattern. $m = n = 32, p = 0.60.$

In the illustration on the left, we can see that a path exists. On the right, each connected "cluster" of cells is given its own color, and we can observe that there is an enormous group of cells which do indeed include a path from top to bottom. (I also believe that the clustering algorithm has made at least a couple mistakes!) The idea of clustering will be the key to helping the computer to identify such connecting paths.

An interesting feature of these mathematical percolation simulations is that, if we fix the values of m and n, then as we vary p we observe a fairly sudden transition in behavior. Below a critical value of p, we almost never get a connecting path, which above it, such a path almost always exists. By analogy with how ice changes to water, mathematicians describe this behavior as a *phase transition*. A variety of other numerical problems have this behavior, include something known as the *subset sum* problem, in which a set of integers must be divided into two sets of equal sum.

Given a sample simulation of the percolation problem, if it is small enough, we may be able to see whether there is a path from top to bottom. But if we need to be able to make this determination automatically, then we need an algorithm that can report whether such a path exists, no matter how big the problem.

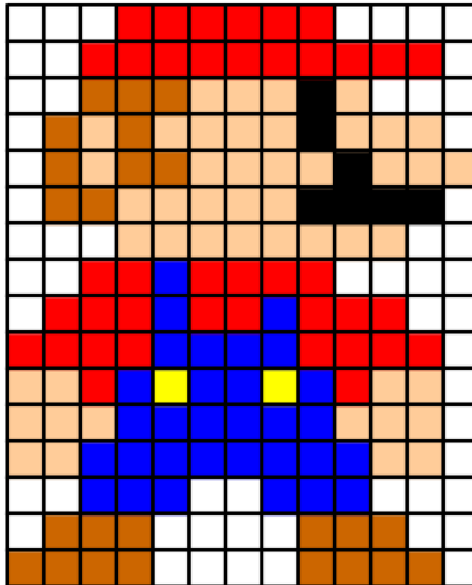
One possible approach is to regard the problem as an example of a maze. To search for a path, start at any open square on the top boundary. Now use the "right hand rule" to trace out a path towards the bottom. That is, always move in such a way that, on every step, you prefer to step in such a way that you stay in contact with what amounts to a wall on your right, formed by the blocked squares, or the right hand wall of the grid. If there is a path to the bottom, you can find it this way, even if you have to try every possible open square at the top.

The other approach organizes all the open squares into components. We are going to paint each open square, in such a way that neighboring open squares will all have the same color. Then, if there is a path from top to bottom, there will be at least one color that occurs in both the top and bottom rows of the grid. To do the component analysis, we start with an arbitrary open square and assign it a color. Then we consider all the neighboring squares which are open, give them the same color, and keep spreading this color until we run out of neighbors. Then we choose a new color, find a new open square that has not been colored, and repeat the process. If we keep doing this, we will have determined all the components, and then answering the path problem is easy.

Our example code for the percolation problem includes a function to determine the components. (And I think it has an error.) It would be worthwhile to try to think about how you would write a function that would do this task!

3 Plotting Mario

Needlepoint is a kind of sewing in which a design is created out of squares. The squares are filled in using threads or yarns of specific colors. We can achieve a similar effect using Python graphics to create a sort of needlepoint image of the Nintendo character known as Mario. Because the early Mario games had very low resolution, we can see that this image involves 16 rows and 13 columns of squares:



and involves 7 colors: white, black, red, blue, yellow, bisque, brown.

Here is a program to draw Mario, by creating a 16×13 grid of boxes, filling in each box with a color, and additionally outlining each box with a black boundary.

There are two new items you should pay attention to:

- Colors are specified by a `numpy` triple of [r,g,b] values between 0 and 1;
- The color of each box is listed in an array that uses (i,j) indexing; Drawing the boxes uses (x,y) coordinates. To convert to (x,y), we have to replace i by $m-1-i$ and swap the order of i and j. It's a headache to get right!

```
def mario ( ):

  import matplotlib.pyplot as plt
  import numpy as np

  m = 16
  n = 13
#
#  RGB color values are
#   'white', 'black', 'red', 'blue', 'yellow', 'bisque', 'brown'
#
  rgb = np.array ( [    \
    [ 1.0, 1.0, 1.0 ], \
    [ 0.0, 0.0, 0.0 ], \
    [ 1.0, 0.0, 0.0 ], \
    [ 0.0, 0.0, 1.0 ], \
    [ 1.0, 1.0, 0.0 ], \
    [ 1.0, 0.8, 0.6 ], \
    [ 0.8, 0.4, 0.0 ] ] )
#
#  For each cell of the grid, list a color index in the RGB array.
#
  color_index = np.array ( [ \
    [ 0, 0, 0, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0 ], \
    [ 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0 ], \
    [ 0, 0, 6, 6, 6, 5, 5, 5, 1, 5, 0, 0, 0 ], \
```

```
    [ 0, 6, 5, 6, 5, 5, 5, 5, 1, 5, 5, 5, 0 ], \
    [ 0, 6, 5, 6, 6, 5, 5, 5, 5, 1, 5, 5, 5 ], \
    [ 0, 6, 6, 5, 5, 5, 5, 5, 1, 1, 1, 1, 0 ], \
    [ 0, 0, 0, 5, 5, 5, 5, 5, 5, 5, 5, 0, 0 ], \
    [ 0, 0, 2, 2, 3, 2, 2, 2, 2, 0, 0, 0, 0 ], \
    [ 0, 2, 2, 2, 3, 2, 2, 3, 2, 2, 2, 0, 0 ], \
    [ 2, 2, 2, 2, 3, 3, 3, 3, 2, 2, 2, 2, 0 ], \
    [ 5, 5, 2, 3, 4, 3, 3, 4, 3, 2, 5, 5, 0 ], \
    [ 5, 5, 5, 3, 3, 3, 3, 3, 3, 5, 5, 5, 0 ], \
    [ 5, 5, 3, 3, 3, 3, 3, 3, 3, 3, 5, 5, 0 ], \
    [ 0, 0, 3, 3, 3, 0, 0, 3, 3, 3, 0, 0, 0 ], \
    [ 0, 6, 6, 6, 0, 0, 0, 0, 6, 6, 6, 0, 0 ], \
    [ 6, 6, 6, 6, 0, 0, 0, 0, 6, 6, 6, 6, 0 ] ] )
#
#  The "(i,j)" box is surrounded by matrix indices where
#  i represents a row (0 = top, m = bottom ),
#  j represents a column (0=left, n = right)
#
#  In (x,y) coordinates,
#  x represents the column, 0 = left, n = right
#  y represents the row, 0 = bottom, m = top)
#
#  So (i,j) ==> (x,y) where x = j and y = m - 1 - i
#     (x,y) ==> (i,j) where j = x and i = m - 1 - y
#
#    (i,j) <----------- (i,j+1)     (x,y+1) <-------- (x+1,y+1)
#      |                  ^           |                  ^
#      |                  |           |                  |
#      |    Color K       |           |    Color K       |
#      |                  |           |                  |
#      V                  |           V                  |
#  (i+1,j) ---------> (i+1,j+1)     (x,y) ---------> (x+1,y)
#
  plt.axis ( 'equal' )
  plt.axis ( 'off' )
#
#  Use Cartesian coordinates to describe the boxes.
#
  for y in range ( 0, m ):
    i = m - 1 - y
    for x in range ( 0, n ):
      j = x
      k = color_index[i,j]
      plt.fill ( [ x, x+1, x+1, x    ], [ y, y, y+1, y+1    ], color = rgb[k] )
      plt.plot ( [ x, x+1, x+1, x, x ], [ y, y, y+1, y+1, y ], 'k-' )

  plt.show ( )
  plt.close ( )
```

This example illustrates that you must be careful to keep in mind the difference between (i,j) and (x,y) coordinate systems when doing grid graphics!