Simulation on Spatial Grids
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/grid_simulation/grid_simulation.pdf

---

---

1 A Forest Fire

Let's consider a simple forest fire model.

We imagine a forest of trees laid out on a regular $m \times n$ grid. We imagine the trees have heights of 1, 2, 3 or 4 meters. The height of a tree will determine how long it can burn. To avoid having to worry about the boundaries of the region, we will assume the usual torus or video game or modular arithmetic conditions. So every tree has four neighbors, to the north, south, east and west.

We suppose that the fire is started by a single tree, and we will also assume this tree is tall (that is, has a heighf of 4 meters) so that it will burn for 4 time steps. Now we want to see if the fire spreads.

We will imagine that the probability that an unburnt tree catches fire is related to the number of burning neighbors. If there are 0 such neighbors, the tree will not ignite on this step. If just 1 neighbor is burning, there is a 25% chance that this tree will start to burn, and in general, if s neighbors are burning, there is a $s * 25\%$ chance. In particular, if all 4 neighbors are on fire, this tree must now ignite as well.

To keep track of the status of every tree, we create an array `forest[i,j]`. This array is randomly initialized with integer values between 1 and 4, representing the varying heights of the trees.

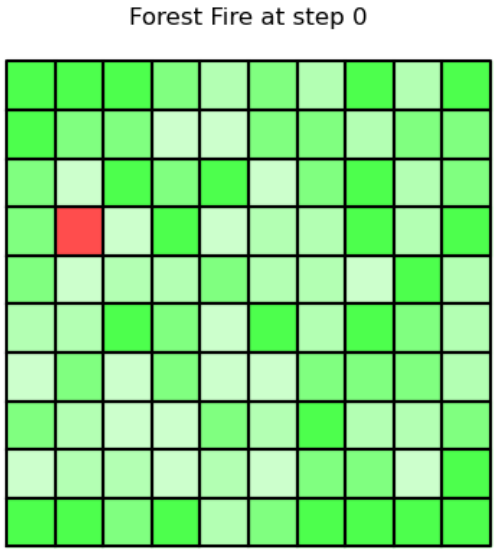
1.1 An outline of the simulation

To run the simulation, we start by picking at rando the location (i, j) where the fire will start. We set `forest[i,j]=-4`, that is, it's a tall tree (of height 4) and it's burning, because of the negative sign. We force it to be a tall tree so that it has 4 chances to catch its neighbors on fire.

Now we repeatedly advance one time step.

- Every tree that was burning now reduces its height by 1. In particular, the tree that started the fire goes down to -3, then -2, then -1, then 0. A tree that reaches height 0 is, of course, completely gone and no longer plays any role.
- Every tree that has neighbors that were burning now has to risk catching on fire, based on the number of such neighbors.
- Unburnt trees with no burning neighbors are left alone.

Here is a plot suggesting the initial situation in the forest. The darkest green spots are the tallest trees, and the red spot is where the fire starts, namely location [3,1].



1.2 Updating the status

Here is how we might implement the status of the forest over one step.

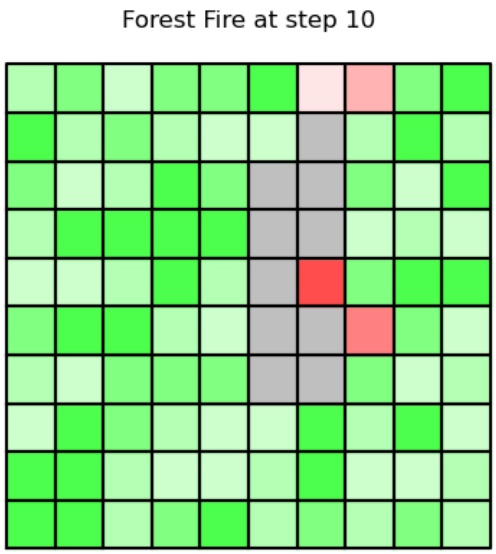
```
def forest_update ( forest ):

  import numpy as np

  m, n = forest.shape

  new_forest = forest.copy ( )

  for i in range ( 0, m ):
    for j in range ( 0, n ):
#
#  A negative value means the tree is burning.
#  Increase it by 1, so it burns towards 0.
#
      if ( forest[i,j] < 0 ):

        new_forest[i,j] = forest[i,j] + 1
#
#  A positive value means the tree is untouched.
```

```
#  Count the burning neighbors to see whether the tree should ignite.
#  This is done by making its index negative.
#
      elif ( 0 < forest[i,j] ):
#
#  Here, we locate the neighbors, but use modular arithmetic
#  so the grid wraps around at the edges.
#
        im1 = ( i - 1 ) % m
        ip1 = ( i + 1 ) % m
        jm1 = ( j - 1 ) % n
        jp1 = ( j + 1 ) % n
#
#  Count the number of neighboring trees that are burning.
#
        s = ( forest[im1,j] < 0 ) + ( forest[ip1,j] < 0 ) \
          + ( forest[i,jm1] < 0 ) + ( forest[i,jp1] < 0 )

        ignite = np.random.random ( )

        if ( ignite < 0.25 * s ):
          new_forest[i,j] = - new_forest[i,j]

  forest = new_forest.copy ( )

  return forest
```

If we keep following the updating rules, then the fire will eventually die out. We can ask questions such as how many trees are likely to survive, and what would happen if the direction of the wind influenced the fire spread.

1.3 Graphics considerations

Here is what the forest might look like after 10 steps of burning:



It's worth spending some time figuring out how to plot the forest fire.

Notice an interesting problem in geometry, the difference between array coordinates and Cartesian coordinates. We imagine the [0,0] entry of our array to be in the upper left; but if we are thinking in terms of (x, y) Cartesian coordinates, we would be tempted to imagine that tree is in the lower left corner instead. Either convention would work; we will choose to insist that the (0,0) tree is in the upper left, which means that when we do plotting, we will have to be careful to rearrange some data so that the plotting software gets things right!

Our plot can be thought of as an $m \times n$ visual array of boxes. Each box could be filled with a color to suggest the status of the tree there. Greens could be untouched trees, red shades suggest fire, and a gray box would represent a 0 value, where a tree has burnt to the ground. We can use a pair of `for()` loops to identify each box to be drawn. We can use a pair of `fill()` and `plot()` commands to fill a box with color, and then draw the outline of its boundary. For the `fill()` command, we want more choices than the simple one letter color codes. To get what we want, we will instead use the RGB color scheme.

```
if ( forest[i,j] == 0 ):
  rgb = [0.75,0.75,0.75]      # light gray
elif ( forest[i,j] == 1 ):
  rgb = [0.8,1.0,0.8]         # light green
elif ( forest[i,j] == 2 ):
  rgb = [0.7,1.0,0.7]         # medium green
elif ( forest[i,j] == 3 ):
  rgb = [0.5,1.0,0.5]         # strong green
elif ( forest[i,j] == 4 ):
  rgb = [0.3,1.0,0.3]         # dark green
elif ( forest[i,j] == -1 ):
  rgb = [1.0,0.9,0.9]         # light red
elif ( forest[i,j] == -2 ):
  rgb = [1.0,0.7,0.7]         # medium red
elif ( forest[i,j] == -3 ):
  rgb = [1.0,0.5,0.5]         # strong red
elif ( forest[i,j] == -4 ):
  rgb = [1.0,0.3,0.3]         # dark red
```

Once we have chosen the color for our box, we get the coordinates of its corners, fill the interior with color, and draw the outline:

```
x = [ j, j, j+1, j+1, j ]
ir = m - 1 - i
y = [ ir, ir+1, ir+1, ir, ir ]
plt.fill ( x, y, color = rgb )
plt.plot ( x, y, 'k-' )
```

After this, we can save the plot to a file, and display it using `plt.show()`.

1.4 Animation

Now let's consider the possibility of an animation. We already know that the plt.show() command will display the forest status at each time step. However, it would be much more dramatic to be able to watch the individual frames as a movie, without us having to close each frame in order to see the next one.

We may talk later about how to create an animation directly inside a Python program, using a peculiar feature known as `FuncAnimation()`. However, I find that feature awkward to use, and it involves some object-oriented coding I am uncomfortable with. As an alternative, since we have saved every frame as a `png` file, we can string them together as a movie, using one of several external programs.

On my system, I can use the ImageMagick `convert` function to take a collection of still frames and make a GIF animation. On my system, the movie can be created by

```
convert -delay 100 -loop 1 forest_fire*.png forest_fire.gif
```

where the `delay` option specifies the time delay between showing each frame, and the `loop` option specifies how many times the frames should be repeated.

The resulting GIF animation is available from the web page.

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.

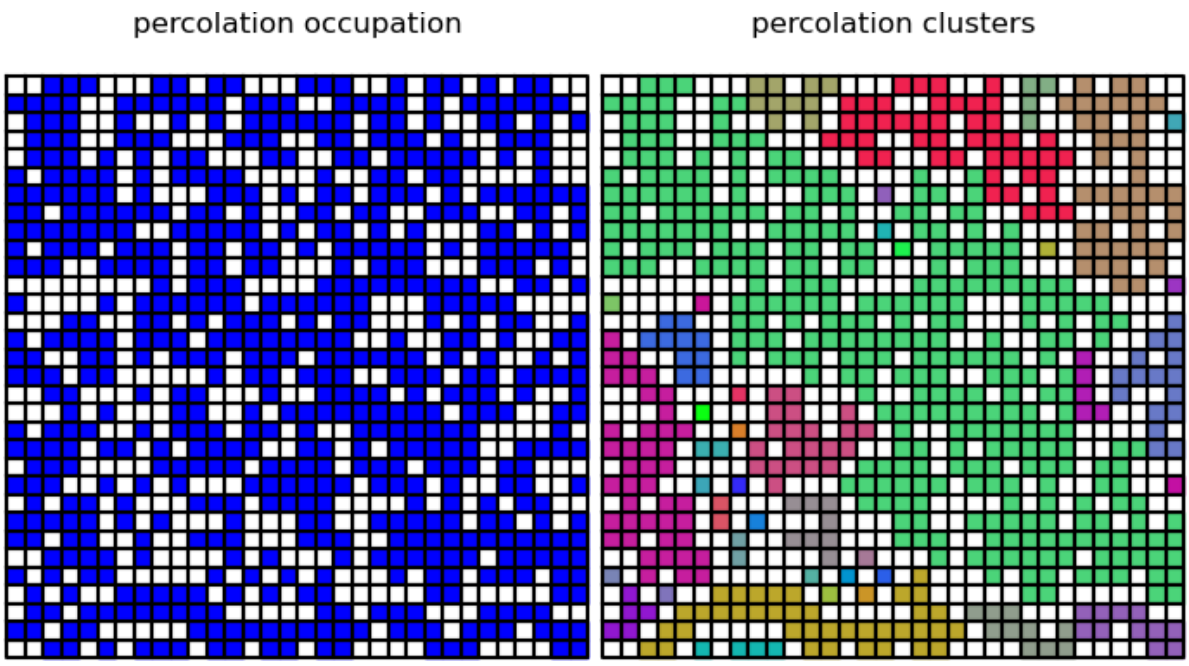
As another example, we might consider an object composed of two metals, one conducting and one not. We suppose the object involves many separate regions of each metal, which are touching, but not melted together. We know the approximate size of a typical "cell" or clump of each metal, and the proportion of the two metals. We ask for the probability that an electical 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 (filled, or nonconducting, or solid) and others to be channels (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 blue cells which do indeed include a path from top to bottom. 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 on 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.

For our example with $m = n = 32$, the phase transition seems to happen at about $p = 0.60$. That is, for densities below this value, there is usually a path, and for densities above this value, there is not. This density value is something like the freezing temperature of water. To estimate its value, you can run multiple experiments near the critical value and keep track of how often a path exists. The example program *percolation.py* could be used to investigate this phenomenon further.

3 A cellular automaton

Steven Wolfram, the developer of *Mathematica* has studied a class of 256 cellular automatons. We will think a cellular automaton as a rule for filling in values in an $m \times n$, given an initial row of 0's and 1's, and a specification for how to use the current row to fill in the next row.

The value to be assigned to cell (i,j) will only depend on three values in the previous row, in positions (i-1,j-1), (i-1,j) and (i-1,j+1). We also need a rule for what to do with the first and last cells in each row: (assume an extra 0? or perhaps use wrap-around?)

Wolfram found that some rules quickly produced an entire field of 0's or 1's, or made uninteresting patterns. But one rule, number 30, seemed to produce a wide range of results depending on the initial data. In fact, he found a way to turn the output of this rule into a random number generator.

The rule is summarized by listing the three values in the preceding row, and their result:

index	neighbors	result
0	0,0,0	0
1	0,0,1	1
2	0,1,0	1
3	0,1,1	1
4	1,0,0	1
5	1,0,1	0
6	1,1,0	0
7	1,1,1	0

Note that 00011110 is the binary representation for 30, which is how the rule got its name.

We can experiment with this rule by filling in a sample array using rule 30. We will assume that the cells on the left and right have an extra "ghost" zero when we are looking for neighbors.

```
0: 0 0 0 1 0 0 0
1: _ _ _ _ _ _ _
2: _ _ _ _ _ _ _
3: _ _ _ _ _ _ _
4: _ _ _ _ _ _ _
5: _ _ _ _ _ _ _
6: _ _ _ _ _ _ _
7: _ _ _ _ _ _ _
8: _ _ _ _ _ _ _
9: _ _ _ _ _ _ _
```

Now we can see how to write a program that will fill in the grid associated with a cellular automaton if we have the rules, and the values in the first row. Assuming we initialized the grid to zero, then we only have to decide whether to reset a cell value to 1.

```
Input:
  m, n: the number of rows and columns
  start[n]: initial value for row 0
Initialize
  c = np.zeros ( [ m, n ] )
  c[0,:] = start[:]
Compute:
  For 1 <= i < m
    for 0 <= j < n
      jm1 = ( j - 1 ) % n
      jp1 = ( j + 1 ) % n
    if ( c[i,jm1] == 0 and c[i,j] == 0 and c[i,jp1] = 1 ) or  # 001 -> 1
       ( c[i,jm1] == 0 and c[i,j] == 1 and c[i,jp1] = 0 ) or  # 010 -> 1
       ( c[i,jm1] == 0 and c[i,j] == 1 and c[i,jp1] = 1 ) or  # 011 -> 1
       ( c[i,jm1] == 1 and c[i,j] == 0 and c[i,jp1] = 0 )     # 100 -> 1
      c[i,j] = 1
Return c
```

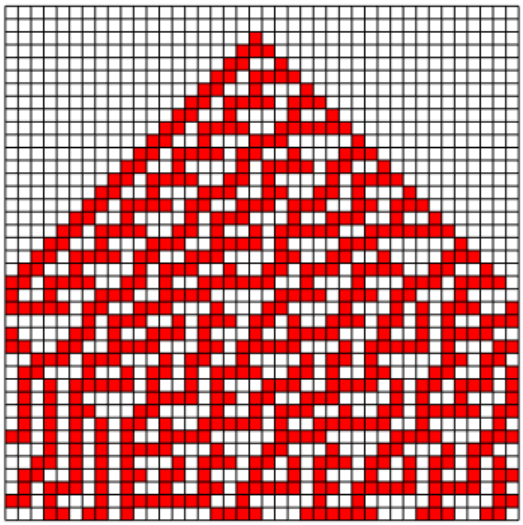
We can plot the cellular automaton easily. Here, plotting with i and j, I got an upside down plot. Defining `i2=step_num-1-i` fixed the discrepancy between matrix indices and Cartesian coordinates:

```
  plt.clf ( )

  for i in range ( 0, m ):
    for j in range ( 0, n  ):
      if ( c[i,j] == 1 ):
        col = 'r'
      else:
        col = 'w'
      i2 = step_num - 1 - i
      plt.fill ( [ j, j+1, j+1, j    ], [ i2, i2, i2+1, i2+1    ], color = col )
      plt.plot ( [ j, j+1, j+1, j, j ], [ i2, i2, i2+1, i2+1, i2 ], 'k-' )
```

Here, we handled the cells on the boundary by using wraparound. Let's see what the results would be for a simple starting arrangement with a single 1:

```
00000000000000000000110111100110100111111001000100011101111000000000000000000000
00000000000000000001100100011100111100000111101110110001000100000000000000000000
00000000000000000011011110110011100010001100001000101011101110000000000000000000
00000000000000000110010000101110010111011010011101101010001001000000000000000000
00000000000000001101111001101001110100010011110001001011011111000000000000000000
00000000000000011001000111001111000110111110001011111010010000100000000000000000
```

Some of the details may be more clear if we use red boxes for the 1 values, and show the outline of the boxes.



4 Random walks and robots

A random walk seems like a peculiar thing to study. However, it models a real physical situation, “Brownian motion”, noticed by Robert Brown in 1827, watching grains of pollen suspended in water, that randomly jiggled.

The puzzle wasn’t fully solved until Albert Einstein explained this by the collision of the pollen with individual water molecules whose velocities had a random variation.

From this, Einstein was able to show that a particle in Brownian motion would tend to drift away from its original position with a predictable variation.

Bellefield Robotics has designed a food truck robot named ”BumbleBee” which offers food for sale at the corners of street blocks along Fifth Avenue. Its home station is at Bellefield Avenue. BumbleBee simply wanders up and down Fifth Avenue, from block to block, looking for customers. As a safety feature, it shuts down if it reaches a distance of 10 blocks from home base.

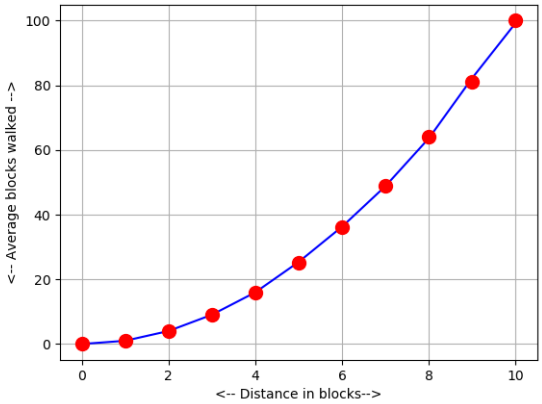
On Friday morning, BumbleBee starts at Bellefield, and chooses a random direction, east or west, to go to the next block. Today there are no customers, so after waiting at one block for five minutes, Bumblebee again randomly moves east or west to another block.

We may reasonably assume that at some point in the day, BumbleBee will randomly reach a distance of 10 blocks, and shut down. On average, when will this happen? Clearly, it takes a minimum of 10 * 5 minutes to get that far from home base, and obviously, there is theoretically no maximum amount of time that could be involved. But realistically, would we have to wait two hours, two days, or what?

We can easily simulate one instance of this walk:

```
def walk_1d ( ):

  directions = [ -1, +1 ]
  location = 0
  blocks = 0

  while ( -10 < location and location < 10 ):
    blocks = blocks + 1
    location = location + np.random.choice ( directions )

  return blocks
```

If we simulate BumbleBee's adventure five times, we see that walks of length 220, 42, 32, 118 or 48 blocks, to reach a destination that is only 10 blocks away. This is a little irregular and mystifying. The first thing we need to do is to generate lots of samples of this walk and find the average time it takes to reach a distance of 1 block, 2 blocks, ..., 10 blocks.

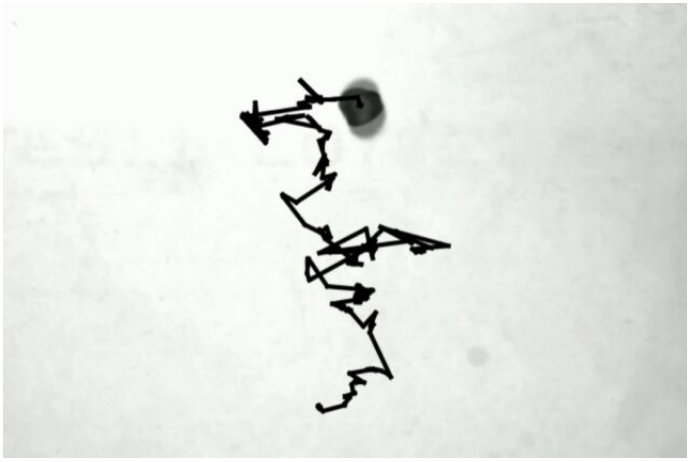


Surprisingly, the average data is very regular, and suggest that to reach d blocks in a random fashion, we actually typically have to walk d^2 blocks. Thus, on average, BumbleBee will shut down after 5×100 minutes, or a bit more than 8 hours after starting.

5 Mexican Jumping Beans

Random walks occur in real life. Consider the case of the Mexican jumping bean. A moth lays eggs inside a bean pod, the bean pod later falls to the ground, the egg hatches as a larva, but is still in the bean pod, eating and developing. Some of these pods are in direct sunlight, and the larva will die if it overheats. The larva can try to move the bean pod by making sudden jerking motions. The bean pod flips around in response. If it lands in a shaded spot, the larva can sense the cooler temperature, and stop the jumping.

See the report *Mexican jumping beans use random walk strategy to find shade* available at https://arstechnica.com/science/2023/02/taking-a-walk-on-the-random-side-helps-mexican-jumping-beans-find-shade/ which includes a link to a short video.



If you find this example interesting, you can try to simulate a very simple version of it. Suppose a jumping bean starts at the origin (0,0), and randomly chooses steps north, south, east or west of one unit. On average, how many steps must be taken before the bean is 0, 1, 2, ..., 10 units away from the origin? Measure distance by $d = \sqrt{i^2 + j^2}$. Do about 100 simulations for each distance measurement. Make a plot of the average number of steps as a function of d, something like the plot we made for the food delivery robot.

6 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.

In a simple model of disease propagation over time and space, we might have an $m \times n$ grid of people, who have agreed not to move during 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 sitting next to an infected person (left, right, in front or in back). A susceptible person who sits next to an infected person for an hour 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 h hours, 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 decreases 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 h, 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, h. The current status is stored in the array P, where $P(i, j) =$

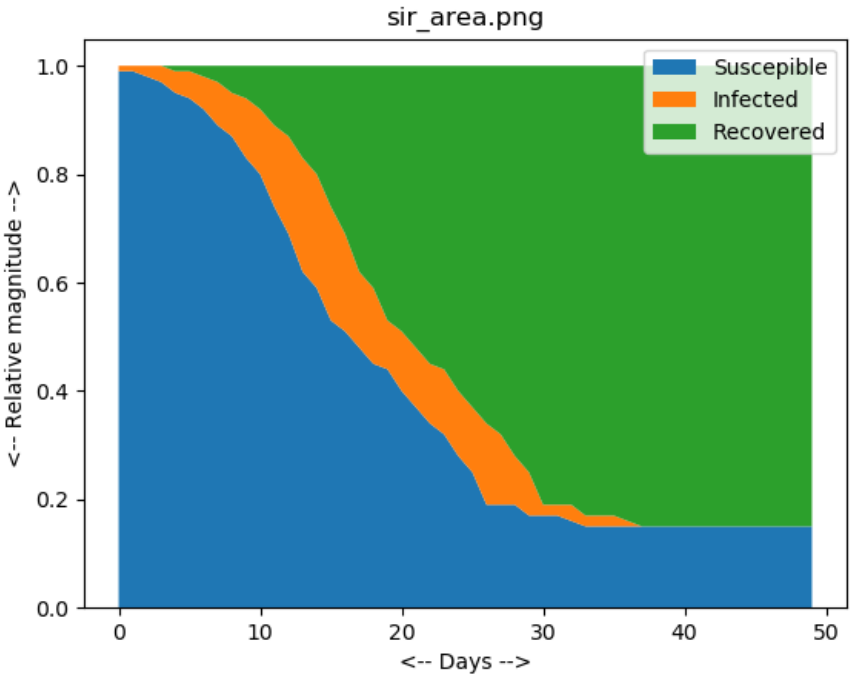
- 1, if person (i,j) is susceptible;
- -k, if person (i,j) is infected for k more hours;
- 0, if person (i,j) is recovered;

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

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

while any P < 0

  hour = hour + 1

  for (i,j) in the grid

    Q = zeros ( m, n )
    if ( P(i,j) == 1 )
      for north, south, east, west infected neighbor, infect P(i,j) with probability p
      if infected, Q(i,j) = -h
    else if ( P(i,j) < 0 )
      Q(i,j) = P(i,j) + 1
      infected person has one less hour to be infected
    end

  end

  replace P by value of Q

end
```

```
print hour, 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 $m = n = 10, p = 0.2,\ h = 4$.