Jack Sullivan

Monte Carlo Simulation of SARS-CoV-2 Aerosol Transport via Finite Difference Schemes

Vuorinen and an extensive team of researchers conducted several different types of simulations of SARS-CoV-2 aersol transport during the current pandemic. Many of these models were based on computational fluid dynamics softwares and too complex to replicate in a short timeframe. Here the Monte-Carlo modeling technique they used to study the spread of the virus based on parameters attained from CFD simulations were replicated in Python.

A series of dots, representing people, were made to move a room of certain dimensions around each second. At each time point, depending on whether or not they were infected, they either coughed or breathed normally (exhaled), or inhaled viral particles to accumulate a dose of the virus and eventually become infected. The infected spread the virus via a diffusion equation finite difference scheme that was central-difference in space and forward-difference in time. This simulation was represented in two dimensions, at a control height of 1m across the entire space as in [1].

```
height=100
width=100
# Number of points in each dimension
numX = 100
numY = 100

# Spatial Step
step = height / numX


### FIX gridX tomrrow!!!!! need in regular meters !

# Spatial coordinates
gridX = np.arange(0, 100, step)
gridY = np.arange(0, 100, step)

# Number of people
Np = 50
# per second
dt = 1
# Diffustion Coefficient
D = .05

# Average walking speed (m/s) chosen from [1]
avgWalkSpeed = .5
# Volume air inhaled per second
Vbreathe = .33
# Removal Timescale (ventilation)
tau = 50
# time from 0 to increment simulation (seconds)
time = 700
```

The spatial grid was constructed to be quantized, but that was eventually not used for the spatial coordinates of the “people” themselves, but to instead build the concentration field that the diffusion equation would manipulate.

The time and Np parameters were set arbitrarily, as they effected the outcome of the simulation but not the concentration of the viral particles in the air. Many of the other parameters were known from other simulations carried out by Vuorinen et al. The diffusion coefficient and timestep (dt), for example, had to be consistent with the known values. All values were measured in terms of meters and seconds, unlesss otherwise specified.

```
53 def setCoord():
54     hold = int((height-height/25) * random.random())
55     if (hold <= height/25):
56         hold += int(height/25)
57     return hold
58 
59 ### PEOPLE INSERTION
60 
61 # Format [currX, currY, Infected, targetArr, coughOffSet, Dose, velocity]
62 # coughOffSet chosen arbitrarily so all people do not cough on
63 # cue at 10 minute intervals
64 def makePpl(Np):
65     array = []
66     # Prevalence of Infected Person
67     # ***** May need to come back to this
68     infNum = int(Np*.05)
69 
70     for p in range(Np):
71         # infected people at infNum percent
72         hold = [setCoord(), setCoord(), False, [setCoord(), setCoord()],
    int(300*random.random()//1), 0]
73         if (p <= infNum):
74             hold[2] = True
75 
76         # Otherwise, they are non-infected
77 
78         distance = np.subtract(np.array(hold[3]), np.array(hold[:2]))
79         angle = np.arctan2(distance[1], distance[0])
80         velArr = [0, 0]
81         velArr[0] = .5 * np.cos(angle)
82         velArr[1] = .5 * np.sin(angle)
83         hold.append(velArr)
84         array.append(hold)
85     return array
86 
87 
88 people = makePpl(Np)
```

The setCoord function created random coordinates inside of the “room” area and is widely used in the rest of the code. The makePpl function creates data for each person to follow at the various steps of the simulation. It sets initial random coordinates of the person(currX, currY), whether or not they are infected (Infected), where they are going (targetArr), when they should start coughing (coughOffSet), how much of the virus they had

accumulated if healthy (Dose), and their velocity components in the X and Y directions (velocity). It does this for the chosen number of people (Np).

```
## DIFFUSION

# Square step for diffusion equation
step2 = step * step
# Initialize concentration field and "future" concentration
c0 = np.zeros((len(gridX), len(gridY)))
c = c0.copy()

def diffStep(c0, c):
# Diffusion equation, vectorized.
# Propagate with forward-difference in time, central-difference in space
# D must change depending on dt step size
########## Thus far drops off at the boundaries

    c[1:-1, 1:-1] = c0[1:-1, 1:-1] + dt * D* ((c0[2:, 1:-1] + c0[:-2, 1:-1]
+ c0[1:-1, 2:] - 4*c0[1:-1, 1:-1] + c0[1:-1, :-2])/step2 )

    c0 = c.copy()
    # c0 is shallow copied for next time, but is same as c
    return c0, c
```

As discussed, the diffusion equation is forward-difference in time and central difference in space. They appropriate time step size (1s) was crucial in working with the diffusion coefficient that was derived from Vuorinen et al.'s earlier simulations. Boundary conditions need to be applied to this in the future. The above code takes advantage of underlying C code for numpy, but accomplished the same as the following central-difference method for solving the diffusion equation [2]:

```
for i in range(1, nx-1):
    for j in range(1, ny-1):
        uxx = (u0[i+1,j] - 2*u0[i,j] + u0[i-1,j]) / dx2
        uyy = (u0[i,j+1] - 2*u0[i,j] + u0[i,j-1]) / dy2
        u[i,j] = u0[i,j] + dt * D * (uxx + uyy)
```

However [3] shows that the convergence of this method would seem to fail given the parameters used herin, where the scheme is only valid for:

$$\frac{\delta t}{(\delta x)^2} \leq \frac{1}{2}.$$

And dt = dx^2 = 1. It appears that Vuorinen et al. used the same position and timesteps described herin, and this simulation appeared to still function, but some future investigation is needed to determine what changes a smaller step, in one case, will produce.

```
135  # Defined for ANIMATION
136  fig = plt.figure()
137  ims = []
138  ### UPDATE EVERYTHING
139  # Loop through each second
140  for t in range(0, time):
141
142      # # run the diffusion finite difference step
143      c0, c = diffStep(c0, c)
144
145      # # Ventilation instantaneously removes some particles out of the air
     per second (1%)
146      # We want to be acting on c0 before next iteration
147      c0 = c0 - dt * c0/tau
148
149      # ANIMATION holders
150      healthX = []
151      healthY = []
152      infX = []
153      infY = []
```

The first for loop increments time, and the diffusion equation in time as well as in the particles in the diffusion field in two spatial dimensions. Certain other elements can be observed that store data along the way for an animation of the entire simulation.

The term on line 147 is a ventilation one, where tau is the removal timescale of a particle in inverse seconds. Essentially here, at every point of the concentration field, a small amount of particles were instantaneously removed. This term could possibly be adjusted or added to for particles to approximate more complex simulations that Vuorinen et al. conducted, where many of the particles expelled in a cough fell to the floor or were carried away.

```
156      for z in range(Np):
157
158          # Reference data elements for convenience of each person at each
     time
159          # Format [currX, currY, Infected, targetArr, coughOffSet, Dose, vel]
160          xCoord = people[z][0]
161          yCoord = people[z][1]
162          infBool = people[z][2]
163          targetArr = people[z][3]
164          coughSet = people[z][4]
165          dose = people[z][5]
166          xVel = people[z][6][0]
167          yVel = people[z][6][1]
168          distance = np.array([0, 0])
```

The second nested for loop (inside of the time loop) is to loop through each person at each time point, and to have them either add to the concentration of viral particles or inhale them, as well as to move closer to their respective targets before the next timestep. A future modification that needs to be implemented is the subtraction of the viral particles from the

concentration field.

```
166         ## TRANSMISSION
167         # If infected
168         if (infBool  == True):
169             # ANIMATION
170             infX.append(people[z][0])
171             infY.append(people[z][1])
172             # if the person is infected
173             # Different result if has just coughed
174             if ( (t + coughSet)  % 600 == 0):
175                 # If time to cough, change concentration at current
    coordinates
176                 # of infected person (Quantized nearest their location in
177                 # Concentration field)
178                 c0[int(xCoord), int(yCoord)] += (4*10**5 + 5)
179             else: c0[int(xCoord), int(yCoord)] += 5
180         else:
181             # Holding for Animation
182             healthX.append(people[z][0])
183             healthY.append(people[z][1])
184             # Accumulates dose at coordinate, at pre-determined rate of
185             # breathing at the concentration nearest the person's spatial
186             # coordinates (again, quantized for concentration field)
187             dose += c0[int(xCoord), int(yCoord)] * Vbreathe
188             if (dose >= 1000):
189                 # If healthy person accumulates critical dose of 100
    particles,
190                 # infect them with the virus
191                 people[z][2] = True
```

The above conditional logic increments the cough or breath if the infected Boolean shows that a person is a virus carrier, and if not, how they accumulate a dose and eventually become infected. The number of particles expelled in a cough is 40,000, which are simply “dropped” at the person’s current location, while they constantly exhale 5 per second, including when they cough [1]. The diffusion steps are solely responsible for spreading this in the room.

For a healthy person, if they breathe a volume of .33 cubic meters per second, the dose increment on line 187 represents the following from [1]:

$$N_b(t) = c_{ave}\dot{V}_b t.$$

Where the concentration of viral particles at the current point is used each second instead of the average across the entire room, which worked in two dimensions given the control height of 1m.

```
115 def setTarg(coord):
116
117     targ = [setCoord(), setCoord()]
118
119     # will have to loop through all v's eventually, would insert here
120     v = np.sqrt(np.abs(coord[0]-targ[0])**2+(coord[1]-targ[1])**2)//1
121     while(v < 1 ):
122         targ = [setCoord(), setCoord()]
123     return targ
124
125 def checkTarg(currCord, currTarg):
126
127     v = np.sqrt(np.abs(currCord[0]-currTarg[0])**2+(currCord[1]-
    currTarg[1])**2)//1
128     # if distance small enough, or close to a wall, give signal
129     # to make new target
130     if(v < 0.5 or currCord[0] >= 98 or currCord[1] >= 98 or currCord[0] <= 2
    or currCord[1] <= 2):
131         return 1
132     return 0
```

The setTarg and checkTarg helper functions for building target coordinates for each person that are a certain distance away (1m) and checking to see whether or not the person has made it there, or is close to one of the physical boundary regions.

```
199         ## WALKING
200         combCoord = people[z][:2] # for convenience
201         # check to see if hitting a wall or close to target
202         if (checkTarg(combCoord, targetArr) == 1):
203             # if yes, create new target
204             targetArr = setTarg(combCoord)
205             # set new velocity based on angle
206             distance = np.subtract(np.array(targetArr), np.array(combCoord))
207             angle = np.arctan2(distance[1], distance[0])
208             people[z][6][0] = .5 * np.cos(angle)
209             people[z][6][1] = .5 * np.sin(angle)
210
211
212         # References to velocity still work, if we've set above
213         # Updating the position coordinates of each person in the people
    array
214         # Update each person coordinate with forward Euler forward scheme
215         people[z][0] = xCoord + people[z][6][0] * dt
216         people[z][1] = yCoord + people[z][6][1] * dt
```

Depending on the checkTarg response, a person may get new “directions” via new velocity components and location coordinates of their target. The person’s position is then incremented by a forward-difference Euler step.

As discussed in class, this finite difference method is not the most stable for calculating actual derivatives in terms of convergence, but is ideal in this case of moving in a straight line to a target point.

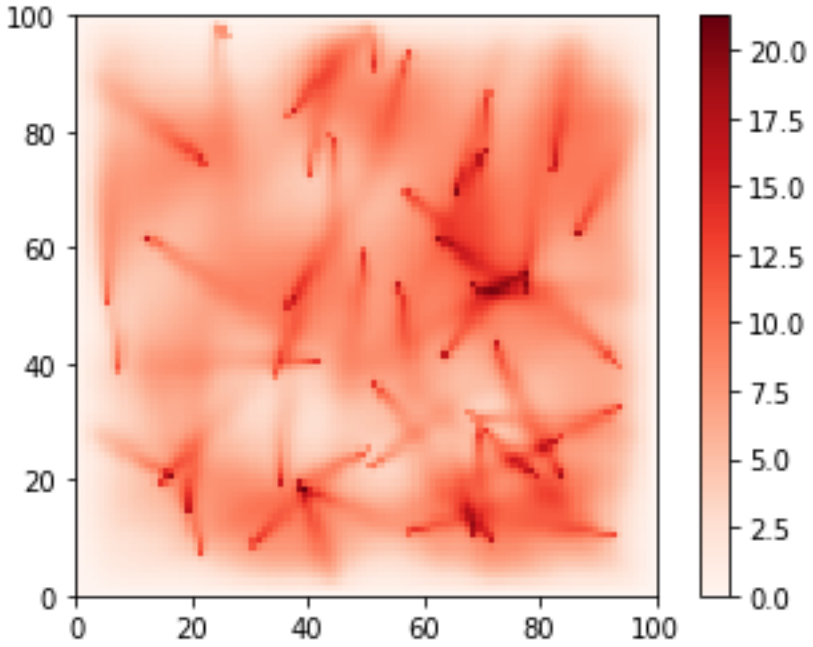
```
        ## WALKING
        combCoord = people[z][:2] # for convenience
        # check to see if hitting a wall or close to target
        if (checkTarg(combCoord, targetArr) == 1):
            # if yes, create new target
            targetArr = setTarg(combCoord)
            # set new velocity based on angle
            distance = np.subtract(np.array(targetArr), np.array(combCoord))
            angle = np.arctan2(distance[1], distance[0])
            people[z][6][0] = .5 * np.cos(angle)
            people[z][6][1] = .5 * np.sin(angle)


        # References to velocity still work, if we've set above
        # Updating the position coordinates of each person in the people array
        # Update each person coordinate with forward Euler forward scheme
        people[z][0] = xCoord + people[z][6][0] * dt
        people[z][1] = yCoord + people[z][6][1] * dt
```

```
    # ANIMATION- append to array of images
    im = plt.imshow(c0, extent=[0, width, 0, height], origin='lower', animated=True, cmap='Reds')
    # Store each image manually
    ims.append([im])

print(len(ims))
plt.colorbar()
plt.axis(aspect='image');
plt.figure(dpi=150)
ani = animation.ArtistAnimation(fig, ims, interval=1, blit=True, repeat=False)
ani.save('simulation.mp4', writer=writer)
```

Plots of concentration were generated at each second, using the above code, and compiled into a separate animation. As the infected people cough, only those intense, concentrated areas of viral particles are shown, until they disperse and more people have been infected, as indicated by the small trailing dots of red. More simulations need to be conducted on the outcome of no person coughing, which could be akin to completely asymptomatic virus carriers in a room.



Given the breadth of parameters in this simulation, and its derivative nature from other simulations, error is hard to quantify in terms of the parameters used. But mainly, the number of people infected, total number of people, and the removal timescale (tau) were changed to various effect. The critical dose was also changed to make it harder to infect a person, which could vaguely resemble giving everyone in the room a stronger immune system.

Sources

1. Ville Vuorinen, M. A. (2020). Modelling aerosol transport and virus exposure with numerical simulations in relation to SARS-CoV-2 transmission by inhalation indoors. *Safety Science, 130*. Retrieved from http://www.sciencedirect.com/science/article/pii/S0925753520302630

2. *The two-dimensional diffusion equation.* (n.d.). Retrieved from scipython.com: https://scipython.com/book/chapter-7-matplotlib/examples/the-two-dimensional-diffusion-equation/

3. Moehlis, J. M. (2001, October 24). *Solution of the Diffusion Equation by Finite Differences.* Retrieved from me.ucsb.edu: https://sites.me.ucsb.edu/~moehlis/APC591/tutorials/tutorial5/node3.html