Simulation 102: Agent-Based Disease Modeling

Modeling how disease spreads from person to person

Le Nguyen · Follow

11 min read · Jul 31, 2023

The Covid era has given all of us a lesson in how disease spreads. We know the basics: some people are infected and spread the disease to others until they recover. Our everyday experience is a good start point to understand disease spread, but here we will formalize our understanding as well as build a computational framework to simulate it.

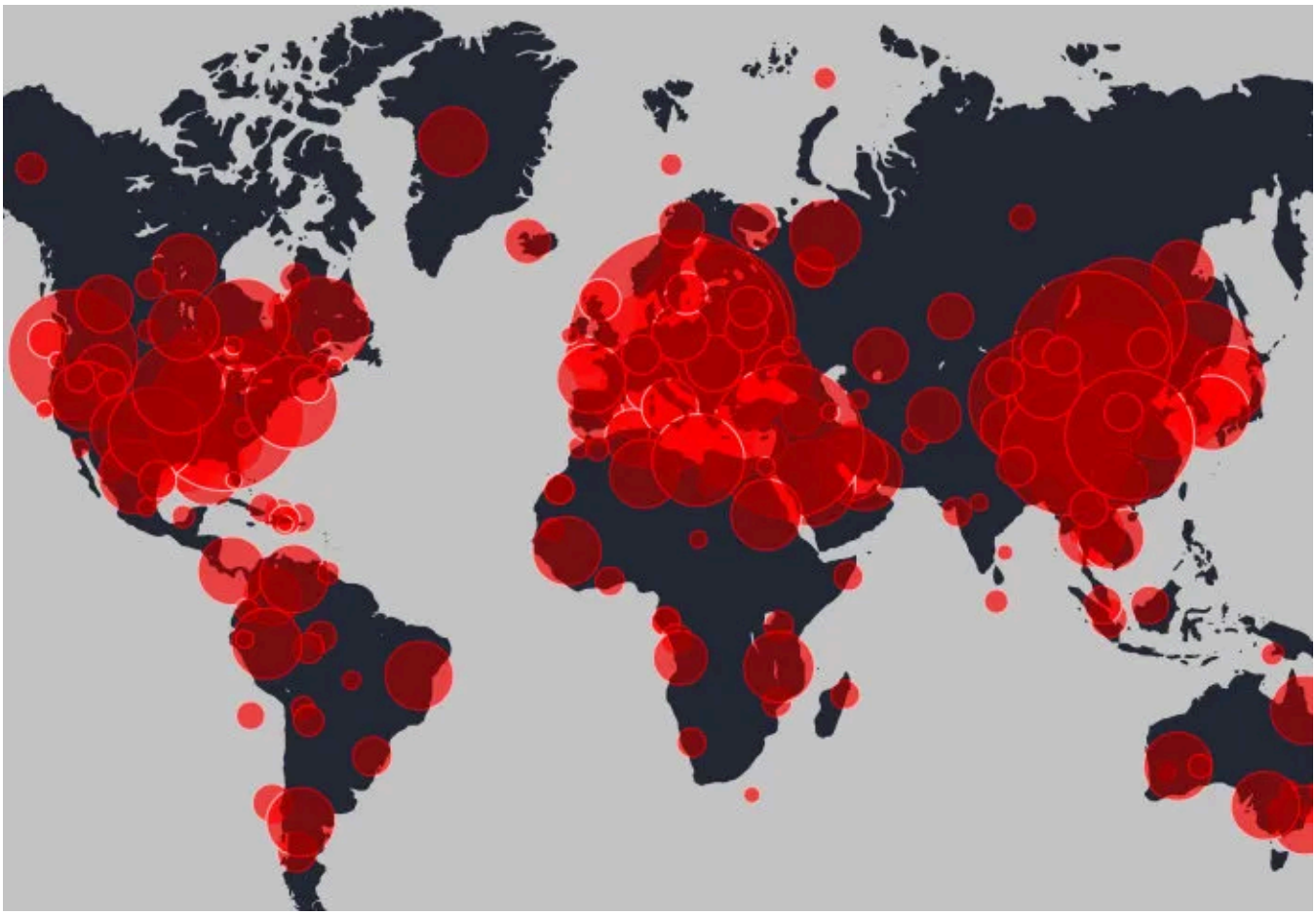


Figure 1: Pandemic effect map

In this article we will:

- Learn about agent-based models
- Learn the basic theory of disease spread
- Implement an agent-based model using classes
- Simulate disease spread with agents

Agent-Based Models

Agent-Based Models are models that focus on simulating individuals and their interactions. This is a fundamentally different approach to grid based simulation seen in our last article. In an agent-based model, a given number of individual agents are created and given certain properties. As the simulation evolves we see how the agents interact with each other and how that changes the properties of each agent.

In our case of simulating disease spread each agent will be a person in our simulation. We will keep track of the position of each person and whether or not

they are infected. Our agents are simple, the only properties they have are location and infection status, but even with simple agents we can make powerful tools.

With our agents, we need to define their interactions and how those interactions change their properties. We need to check the distance between our agents and if they are close to an infected agent. If an uninfected agent is close to an infected one we will let the disease transfer and change the uninfected agents status to infected. We iterate this process as our agents move around and that gives us everything we need to build our simulation.

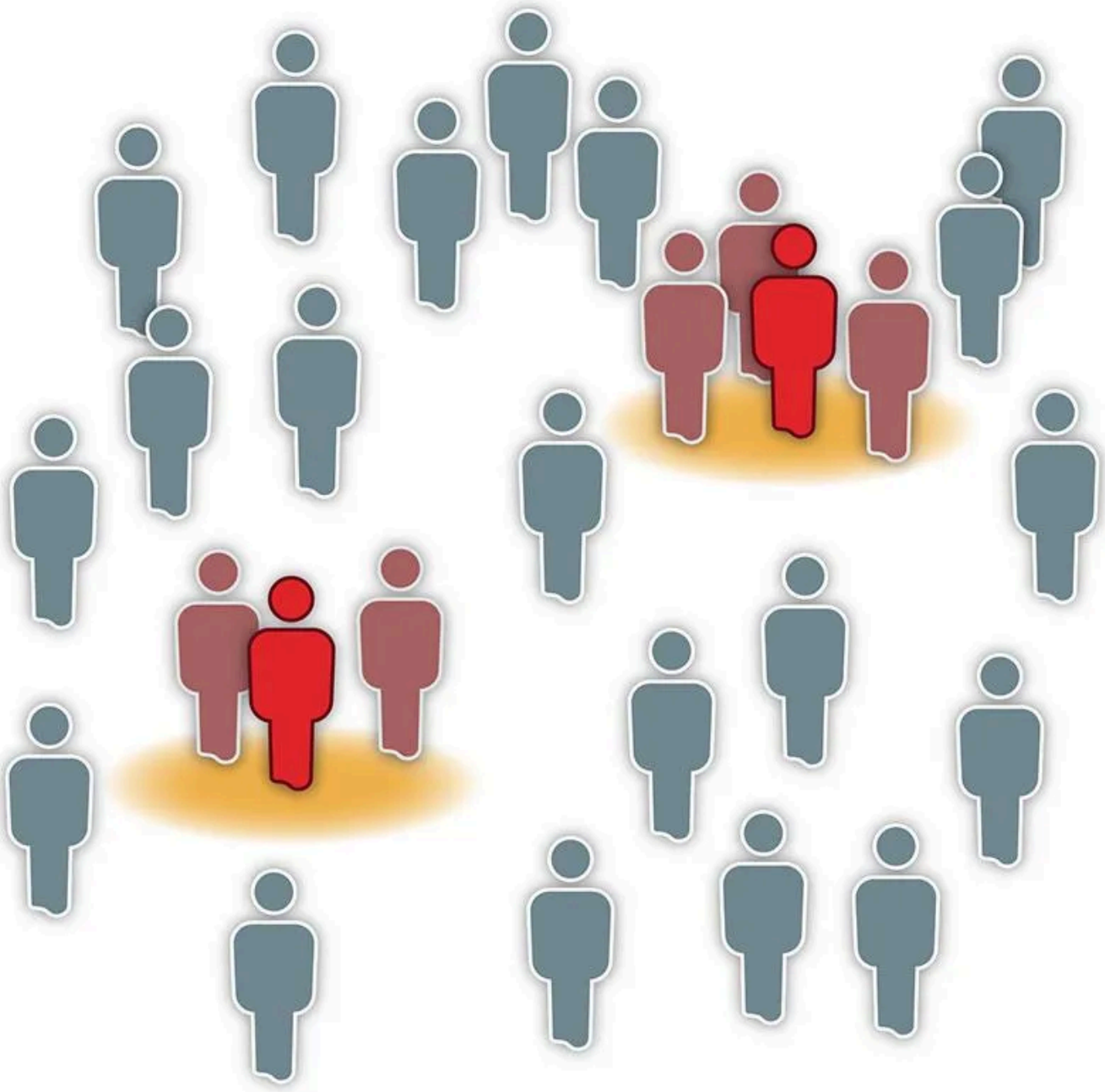


Figure 2: Representation of agents in a disease spread model.

Before going into simulation building, we will overview the theory behind disease spread.

Theory of Disease Spread

Mathematically, the spread of disease can be modeled with the SIR model, which stands for **S**usceptible, **I**nfectious, **R**ecovered. The model is a set of differential equations that keep track of the number of people who are susceptible to being infected, actively infectious and have recovered from previously been infected. The governing set of equations and a plot of their output is given below.

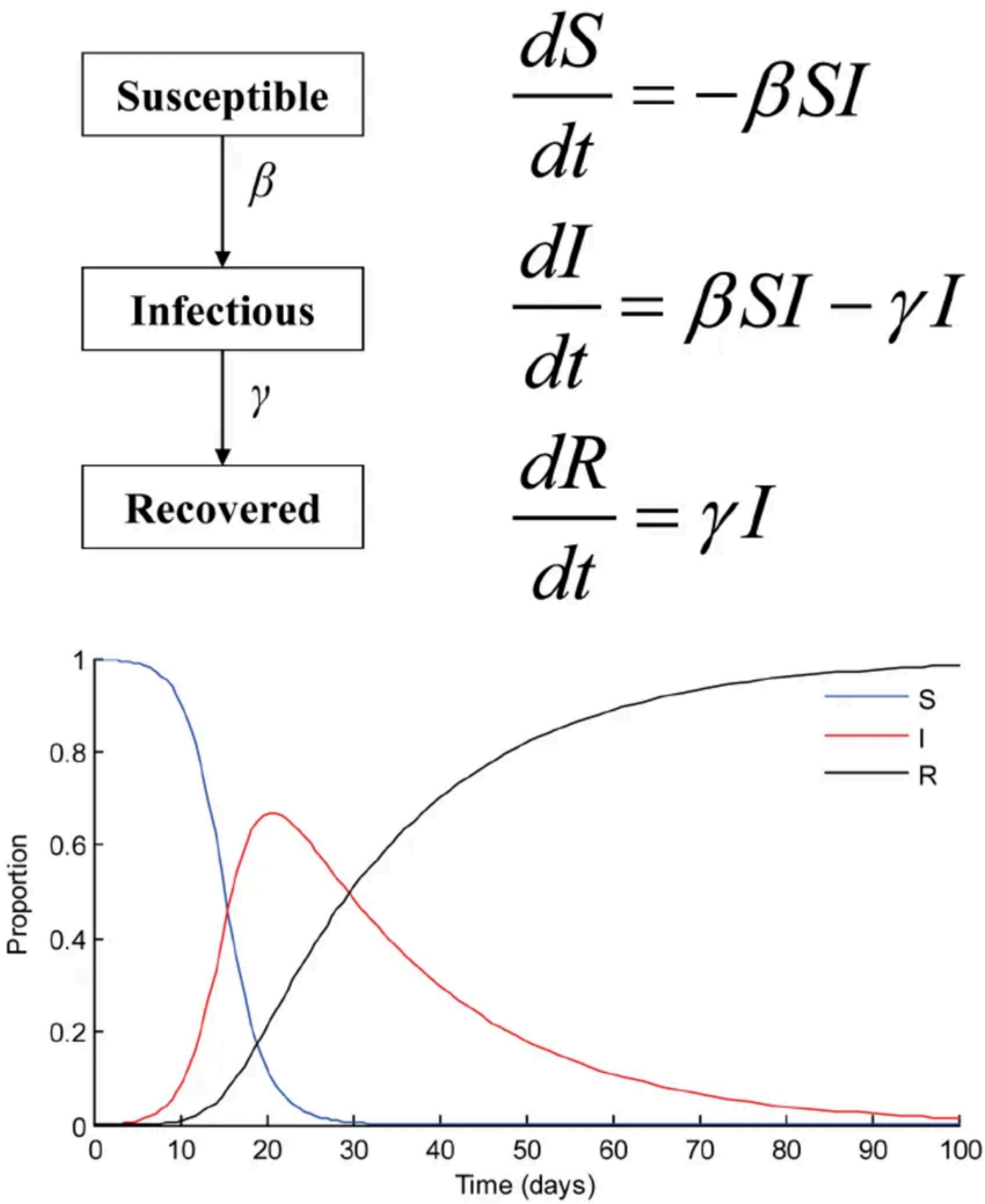


Figure 3: Equations and plot of SIR model.

We can see these equations calculate the change in the susceptible, infected, and recovered population in respect to time. Here, model parameters β and γ are the infection rate and recovery rate. We can understand the differential equations from their line plot. The susceptible population (blue line) decreases as more people get infected until all of the population has been infected. Conversely, the infected

population (red line) increases as the susceptible population decreases (get infected) but the infected population will decrease as people start to recover (black line).

This gives the infected population the skewed hump because it is controlled by how many people are infected vs how many people are recovered. At first no one in the infected population has had the time to recover, so it shoots up, but over time they will start to recover and there will be less and less susceptible population to infect because they have already been infected; thus the infected population tapers off.

With the general theory of disease spread in mind, we can get into the modeling.

Class Implementation

Classes are the ideal data object to create an agent-based model. A class is a templet to store values and methods that can easily be replicated. In our case, we will use classes to create our agents since classes allow us to create data structures that store the same kind of information over and over again.

The following subsections will detail how to create a class that represents our agents as well as outside functions to model their interactions. A lot of code is shown below but much of it is the same or similar code with an added feature onto it. By the end we will have a fully functioning model to simulate the spread of disease.

Initialize Agents

In our first snippet of code we will create our agent class and give all of our agents an x and y coordinate. Additionally, we create a utility function that will retrieve the positions of all of agents. All we need to do is create as many agents as we want in a loop and then plot where they are seen in figure 4.

```
#Agent class
class Agent():
    def __init__(self,x,y):
        self.x = x
        self.y = y

def getPosition(agents):
    positions = []

    for agent in agents:
        positions.append([agent.x,agent.y])

    return np.array(positions)
```

```
#Make Agents
agents = []
for i in range(0,100):
    x = np.random.uniform(0,100)
    y = np.random.uniform(0,100)
    agents.append(Agent(x,y))

#Get agent positions
positions = getPosition(agents)

#Plot Agents
plt.figure(figsize=(6,6))
plt.scatter(positions[:,0] , positions[:,1])
plt.tick_params(left = False, right = False , labelleft = False,labelbottom = F
```

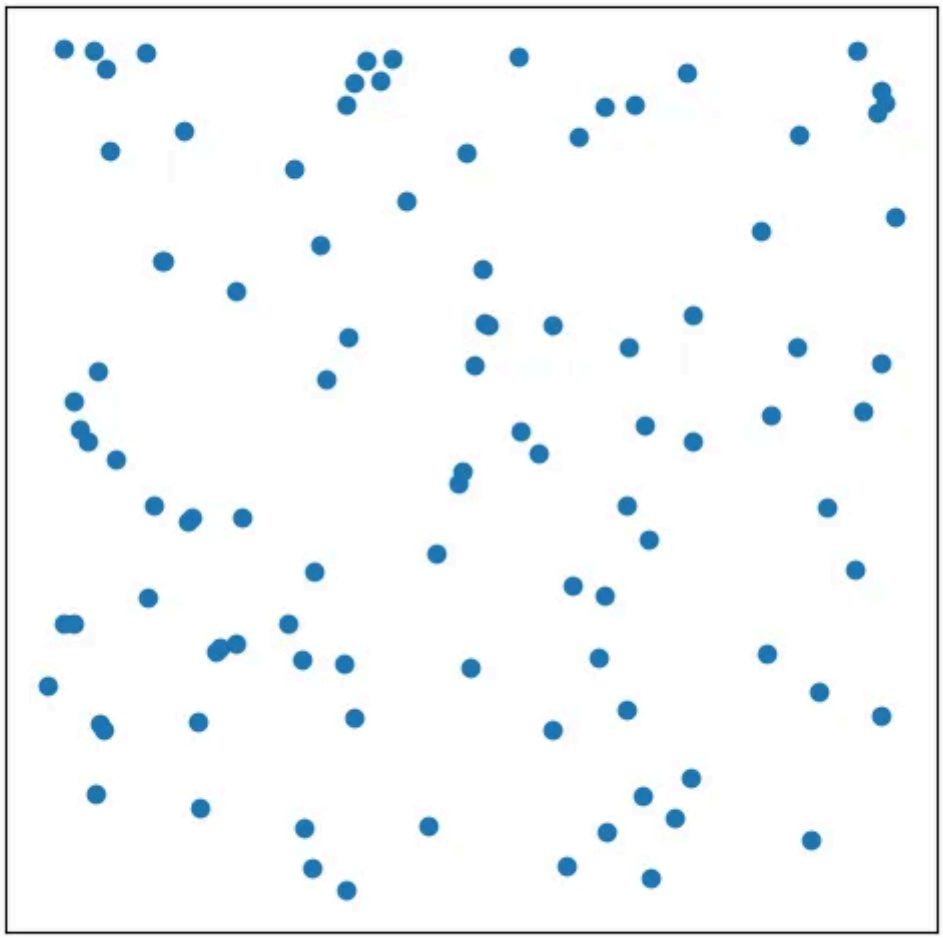


Figure 4: Initial position of all Agents.

Add Movement

Next, we add a movement function to our agents which will make them move around the environment randomly. Here we need to address our boundary conditions as well (what our agents do when they get to the edge of our simulation); we simply check to see if they are trying to move past the boundary and set them at the boundary for that movement step. We make another utility function that will go through all of our agents and have them move one step and then we call that function for our desired number of steps. We can see our agents moving in figure 5.

```
#Agent Class
class Agent():
    def __init__(self,x,y):
        #initalize position
        self.x = x
        self.y = y

    def movement(self,stepSize,xBounds,yBounds):
        #step
        self.x += stepSize*np.random.uniform(-1,1)
        self.y += stepSize*np.random.uniform(-1,1)

        #check boundaries
        if self.x < xBounds[0]:
            self.x = xBounds[0]
        elif self.x > xBounds[1]:
            self.x = xBounds[1]

        if self.y < yBounds[0]:
            self.y = yBounds[0]
        elif self.y > yBounds[1]:
            self.y = yBounds[1]

"""
Previous functions
"""

def moveAgents(agents,stepSize,xBounds,yBounds):
    for i in range(0,len(agents)):
        agents[i].movement(stepSize,xBounds,yBounds)

    return agents

#Make Agents
agents = []
for i in range(0,100):
    x = np.random.uniform(0,100)
    y = np.random.uniform(0,100)
    agents.append(Agent(x,y))
```

```
#Move Agents
stepSize = 1
xBounds = [0,100]
yBounds = [0,100]

frames = []
for i in range(0,200):

    agents = moveAgents(agents,stepSize,xBounds,yBounds)
    positions = getPosition(agents)
    frames.append([positions[:,0] , positions[:,1]])
```

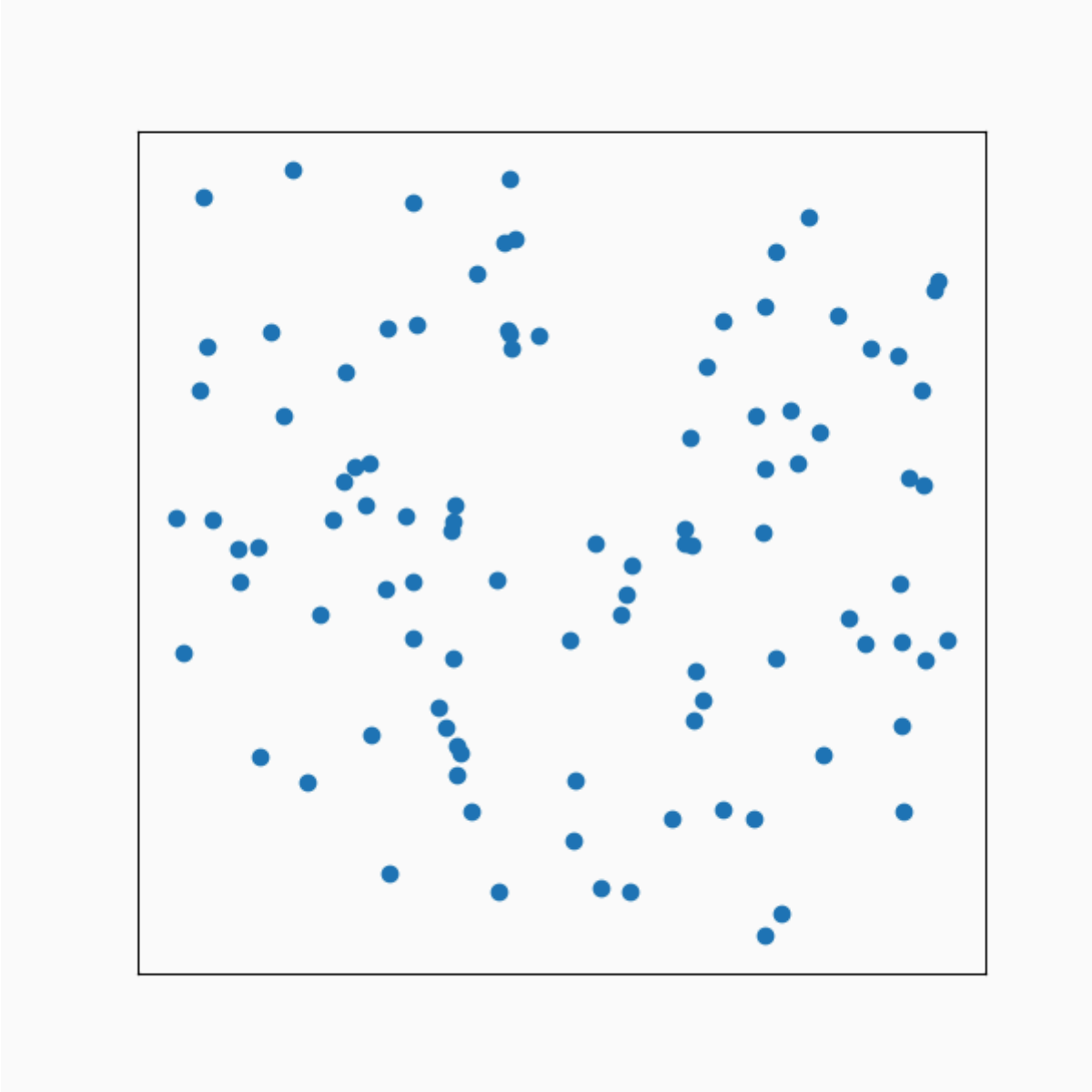


Figure 5: Gif of moving agents.

Add Infect Status

Since we are modeling the spread of disease, we need to add an infected status to our agents and write a utility function that will record the status of each of our

agents. Figure 6 shows a mix of infected (red) and healthy agents (green) initialized in our model.

```
#Agent Class
class Agent():
    def __init__(self,x,y,infected):
        #initalize position
        self.x = x
        self.y = y
        self.infected = infected
    """
    Movement function
    """
"""
Previous functions
"""

def getInfected(agents):
    infected  = []
    for agent in agents:
        infected.append(agent.infected)
    return infected
#Make Agents
agents = []
for i in range(0,100):
    x = np.random.uniform(0,100)
    y = np.random.uniform(0,100)
    agents.append(Agent(x,y,np.random.randint(0,2)))
#Get positions and infected status
positions = getPosition(agents)
infected = getInfected(agents)
#plot agents
plt.figure(figsize=(6,6))
plt.scatter(positions[:,0] , positions[:,1], c = infected, cmap = "RdYlGn_r")
plt.tick_params(left = False, right = False , labelleft = False,labelbottom = F
```

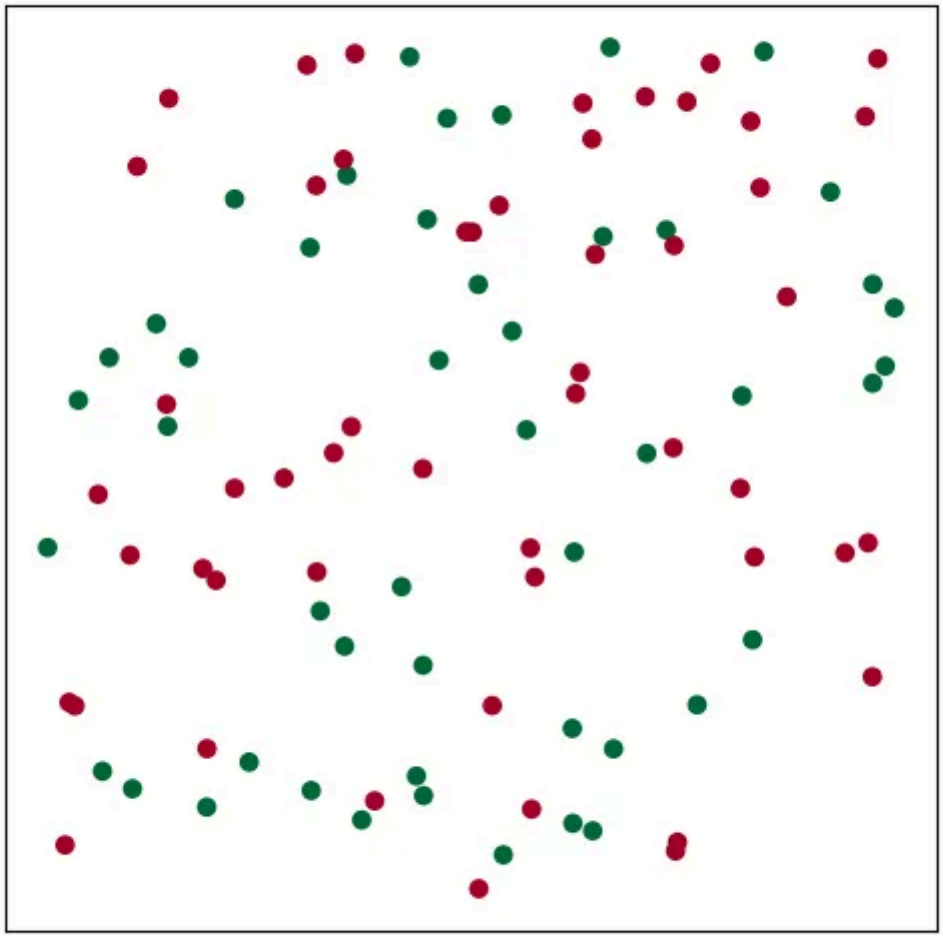


Figure 6: Mix of healthy and infected agents.

Add Disease spread

With our agents having an infected status, we have to be able to spread the infection. Every agent now gets a resistance and an infect function which will be used every time a healthy agent contacts an infected agent. The infect function rolls a random number and if that number is greater than the agents resistance, the agent will become infected. This allows us to play with difference resistance values in our population, say some agents are masked or vaccinated and we want to study the effects.

We need a couple new utility functions as well to make our agents spread disease. The first function we need is a function to find all agents who are close to one another to see which agents are at risked of being infected. This method calculates a distance matrix that contains the pair-wise distance for all agents in our simulation. We can then look at any agents that are close to each other (set at a distance

threshold of our choice) and see if infection can spread. The roll infect function does this for us by using the infect function on all agents close to an infected agent if they are healthy. We can see the spread of disease from agent to agent in figure 7.

```
#Agent Class
class Agent():
    def __init__(self,x,y,infected,resistance):
        #initalize position
        self.x = x
        self.y = y
        self.infected = infected
        self.resistance = resistance
    """
    Movement function
    """
    def infect(self):
        infectRoll = np.random.uniform()

        if self.infected == False and infectRoll > self.resistance:
            self.infected = True
"""
Previous functions
"""
#Check if any agents are within infecting distance
def getCloseAgents(distanceMatrix,agentNumber):
    sort = np.argsort(distanceMatrix[agentNumber])
    closeMask = distanceMatrix[agentNumber][sort] < 5
    closeAgents = np.argsort(distanceMatrix[agentNumber])[closeMask][1:]
    return closeAgents

#Go through close agents and roll to see if they get infected
def rollInfect(agents):
    positions = getPosition(agents)
    distanceMatrix = distance_matrix(positions,positions)

    for i in range(0,len(agents)):
        closeAgents = getCloseAgents(distanceMatrix,i)
        for j in closeAgents:
            if agents[j].infected == True:
                agents[i].infect()
    return agents

#Make agents
agents = []
for i in range(0,100):
    x = np.random.uniform(0,100)
    y = np.random.uniform(0,100)
    agents.append(Agent(x,y,0,.5))

for i in range(0,3):
```

```
    agents[i].infected = True
#Seed a few agents with infection
stepSize = 1
xBounds = [0,100]
yBounds = [0,100]
#Move Agents
frames = []
for i in range(0,200):
    agents = moveAgents(agents,stepSize,xBounds,yBounds)
    agents = rollInfect(agents)
    positions = getPosition(agents)
    infected = getInfected(agents)
    frames.append([positions[:,0] , positions[:,1], infected])
```

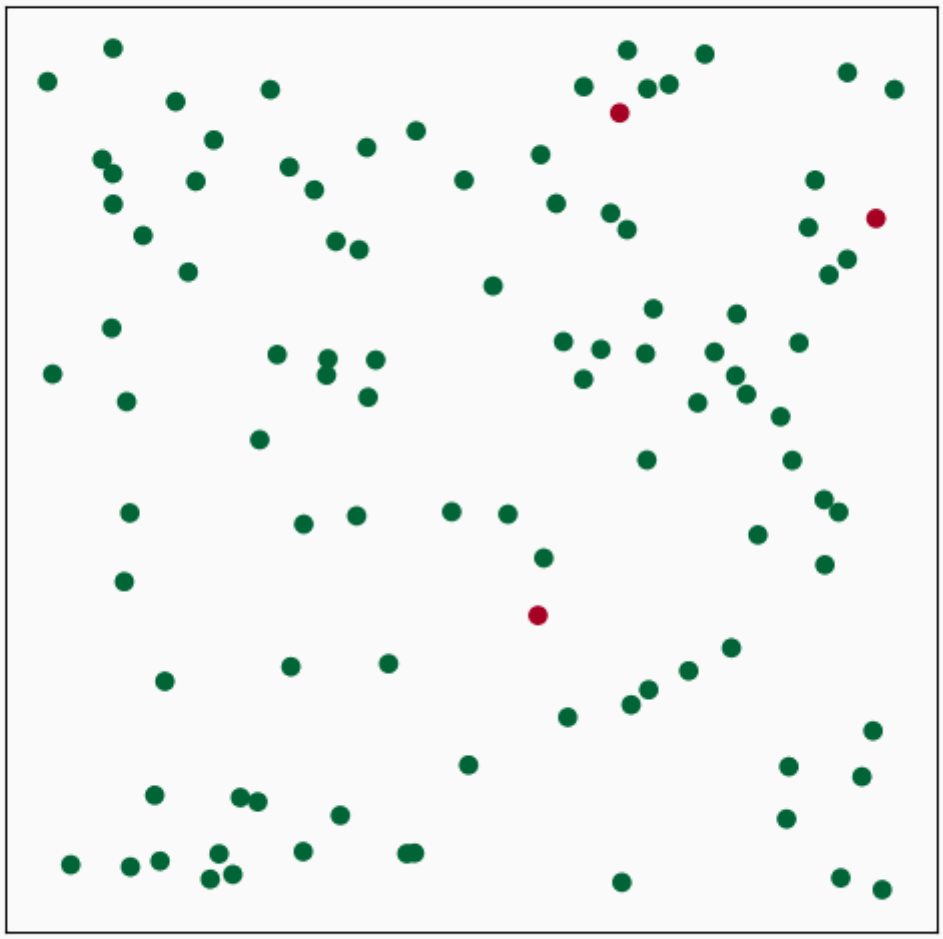


Figure 7: Infection spreading among agents.

Add Recovery (Final Full Agent)

To complete our agents to run a full SIR style simulation we need our agents to be able to recover from infection. To do this, we add an infected counter to the agent. When a agent is infected the counter is set at some value and every time step the counter start ticks down. Once the counter gets to zero the agent becomes healthy again. We can also change the resistance value after recovery so agents that have once been infected have less of a chance to be reinfected. Our fully implemented agents can be seen in figure 8.

```
#Agent Class
class Agent():
    def __init__(self,x,y,infected,resistance,infectedCounter = 0):
        #initalize position
        self.x = x
        self.y = y
        self.infected = infected
        self.resistance = resistance
        self.infectedCounter = infectedCounter

    def movement(self,stepSize,xBounds,yBounds):
        #step
        self.x += stepSize*np.random.uniform(-1,1)
        self.y += stepSize*np.random.uniform(-1,1)

        #check boundaries
        if self.x < xBounds[0]:
            self.x = xBounds[0]
        elif self.x > xBounds[1]:
            self.x = xBounds[1]

        if self.y < yBounds[0]:
            self.y = yBounds[0]
        elif self.y > yBounds[1]:
            self.y = yBounds[1]

        #Tick down infeced counter
        if self.infected == True:
            self.infectedCounter -= 1
            if self.infectedCounter <= 0:
                self.infected = False

    def infect(self):
        infectRoll = np.random.uniform()

        if self.infected == False and infectRoll > self.resistance:
            self.infected = True
            self.infectedCounter = 50
            self.resistance *= 1.5
"""
```

```
Previous functions
"""

#Make Agents
agents = []
for i in range(0,100):
    x = np.random.uniform(0,100)
    y = np.random.uniform(0,100)
    agents.append(Agent(x,y,0,.5))
#Seed infected
for i in range(0,3):
    agents[i].infected = 1
    agents[i].infectedCounter = 50
#Move Agents
stepSize = 1
xBounds = [0,100]
yBounds = [0,100]

frames = []
for i in range(0,200):
    agents = moveAgents(agents,stepSize,xBounds,yBounds)
    agents = rollInfect(agents)
    positions = getPosition(agents)
    infected = getInfected(agents)
    frames.append([positions[:,0] , positions[:,1], infected])
```

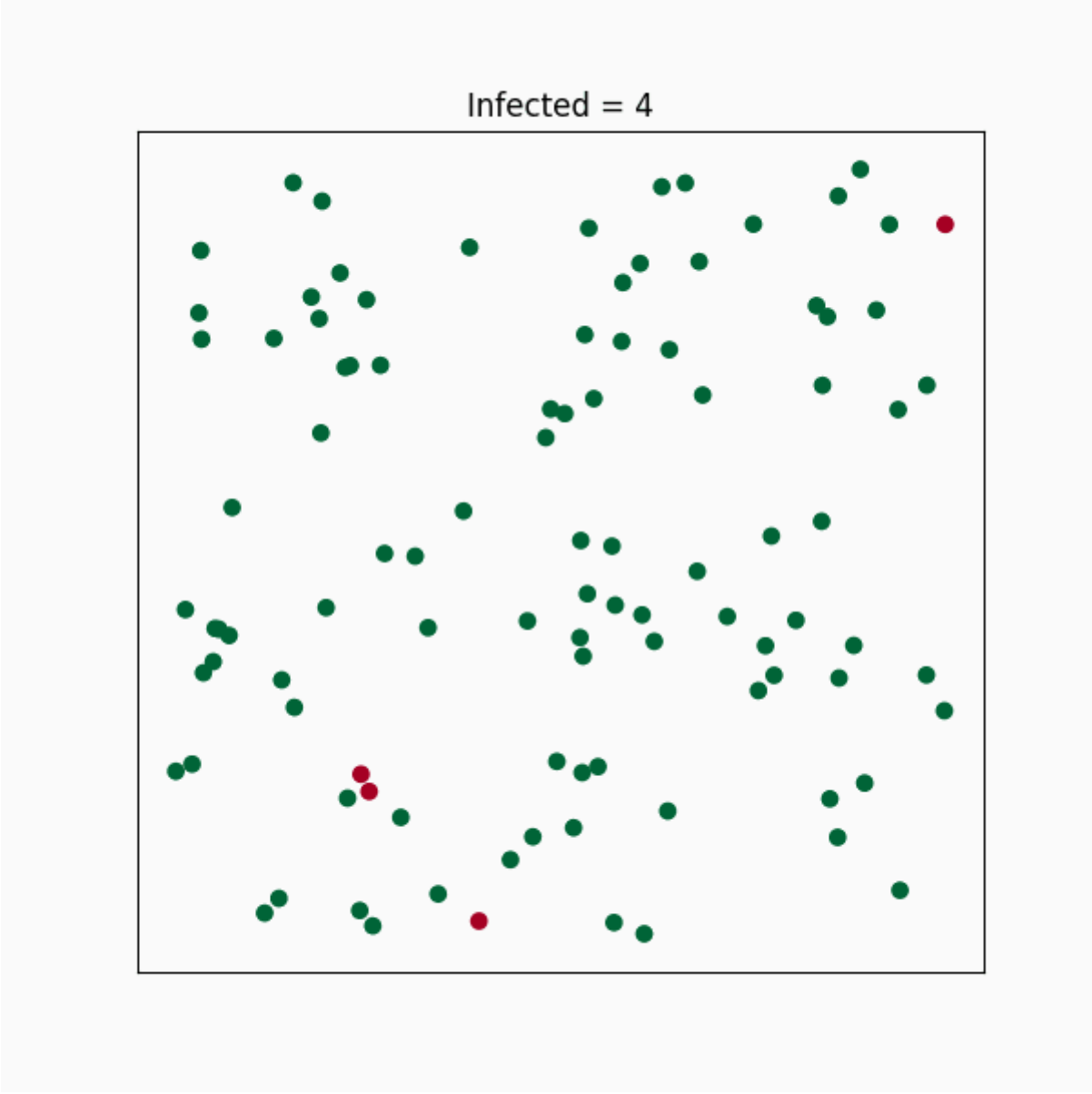


Figure 8: Full agents that implement the SIR model.

Our agents now have all the properties and functionality to run a simulation which we will do in the final section.

Simulation

We see our full simulation play out in figure 9. The simulation parameters are the following: 500 agents, 10 infected, uniform 30% resistance to infection, agents can infect those who are within 10 steps of them and they stay infected for 50 time steps with the entire simulation being 500 time steps.

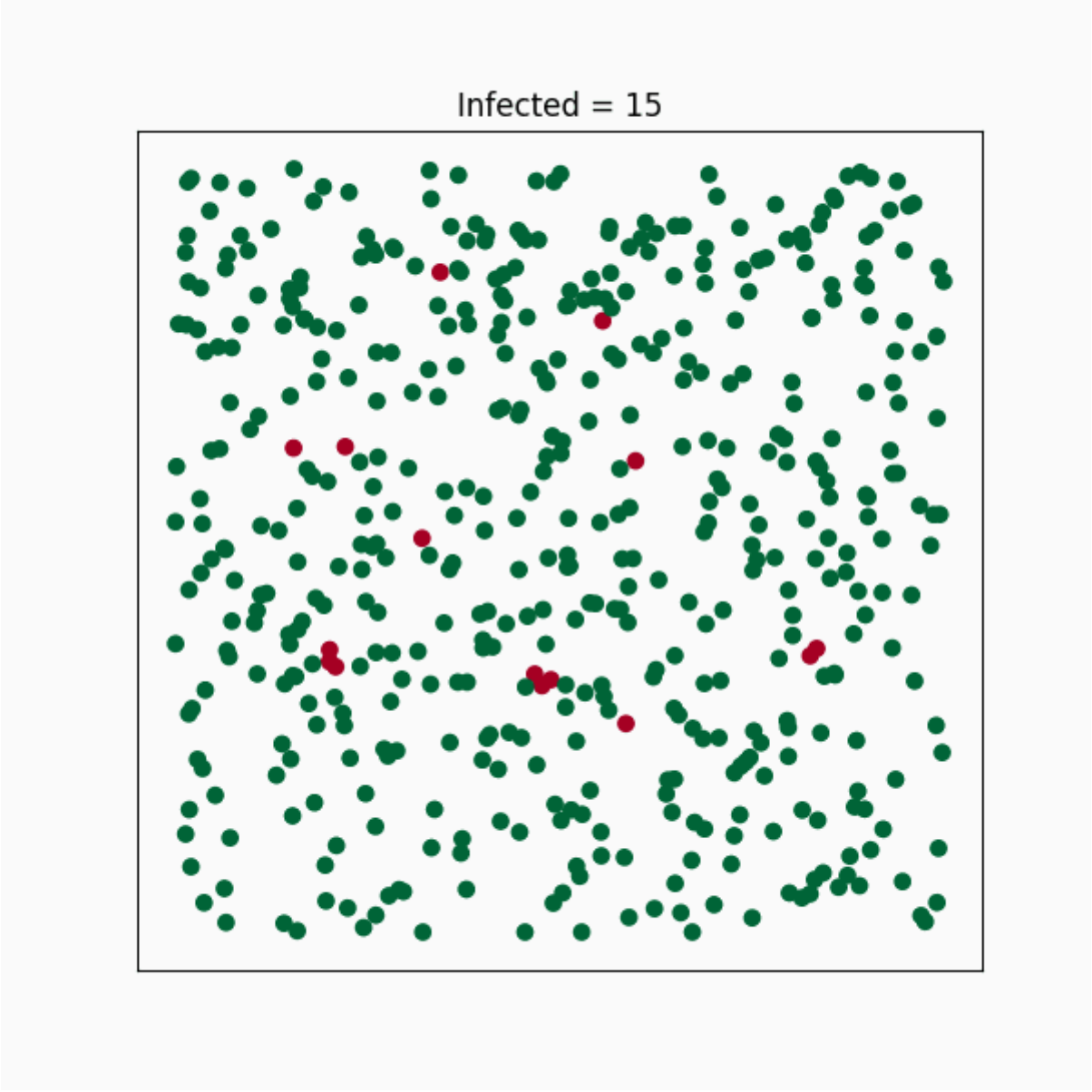


Figure 9: Full simulation

Keeping track of the number of infected agents over time we get the following curve seen in figure 10. As we can see, this curve fits well to the expected theoretical infected curve seen in figure 3 which proves we have constructed a valid model. We have also create a malleable model that can have its parameters changed to study many scenarios.

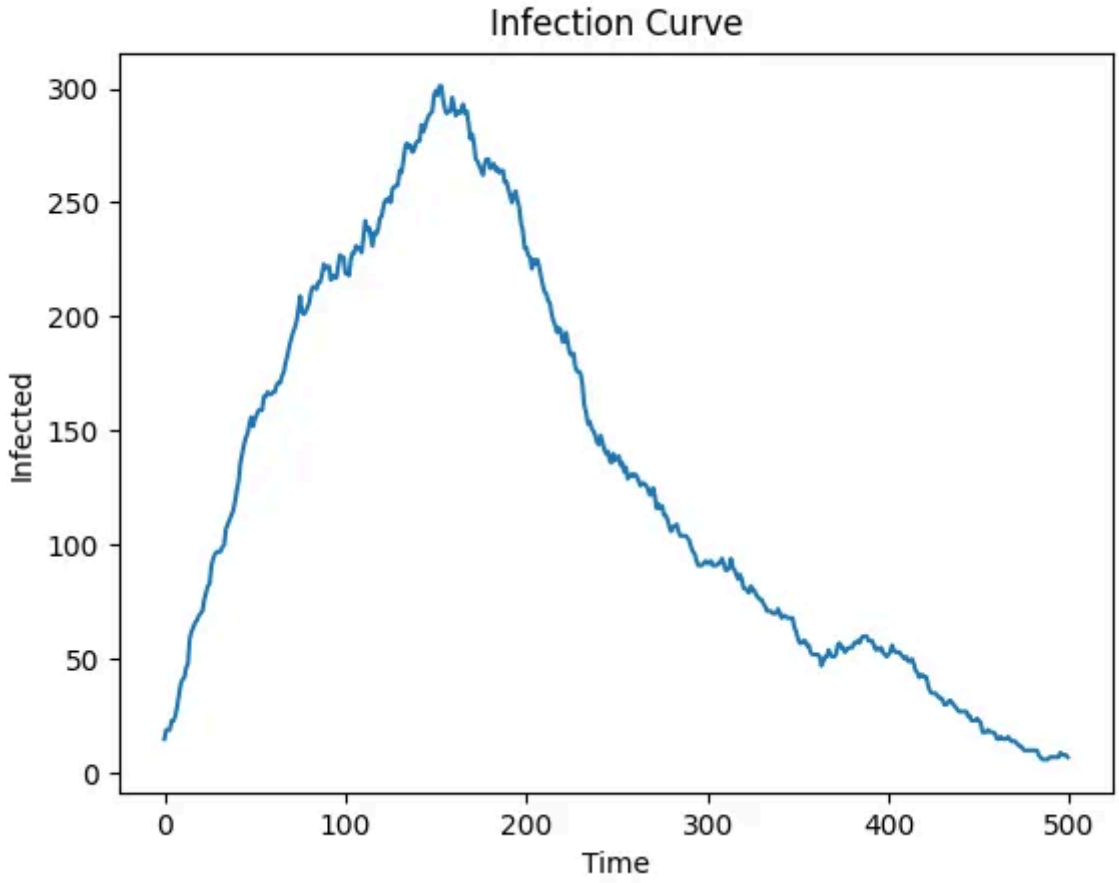


Figure 10: Infection cure.

Full Code

```
import numpy as np
import matplotlib.pyplot as plt
import imageio
from scipy.spatial import distance_matrix

class Agent():
    def __init__(self,x,y,infected,resistance,infectedCounter = 0):
        #initalize position
        self.x = x
        self.y = y
        self.infected = infected
        self.resistance = resistance
        self.infectedCounter = infectedCounter

    def movement(self,stepSize,xBounds,yBounds):
        #step
        self.x += stepSize*np.random.uniform(-1,1)
        self.y += stepSize*np.random.uniform(-1,1)

        #check boundaries
        if self.x < xBounds[0]:
```

```
            self.x = xBounds[0]
        elif self.x > xBounds[1]:
            self.x = xBounds[1]

        if self.y < yBounds[0]:
            self.y = yBounds[0]
        elif self.y > yBounds[1]:
            self.y = yBounds[1]

        #Tick down infeced counter
        if self.infected == True:
            self.infectedCounter -= 1
            if self.infectedCounter <= 0:
                self.infected = False

    def infect(self):
        infectRoll = np.random.uniform()

        if self.infected == False and infectRoll > self.resistance:
            self.infected = True
            self.infectedCounter = 50
            self.resistance *= 1.5

def getPosition(agents):
    positions = []
    for agent in agents:
        positions.append([agent.x,agent.y])
    positions = np.array(positions)
    return positions

def moveAgents(agents,stepSize,xBounds,yBounds):
    for i in range(0,len(agents)):
        agents[i].movement(stepSize,xBounds,yBounds)
    return agents

def getInfected(agents):
    infected  = []
    for agent in agents:
        infected.append(agent.infected)
    return infected

def getCloseAgents(distanceMatrix,agentNumber):
    sort = np.argsort(distanceMatrix[agentNumber])
    closeMask = distanceMatrix[agentNumber][sort] < 10
    closeAgents = np.argsort(distanceMatrix[agentNumber])[closeMask][1:]
    return closeAgents

def rollInfect(agents):
    positions = getPosition(agents)
    distanceMatrix = distance_matrix(positions,positions)
    for i in range(0,len(agents)):
        closeAgents = getCloseAgents(distanceMatrix,i)
        for j in closeAgents:
```

```
            if agents[j].infected == True:
                agents[i].infect()
    return agents

def makeGif(frames,name):
    !mkdir frames

    counter=0
    images = []
    for i in range(0,len(frames)):
        plt.figure(figsize = (6,6))
        plt.scatter(frames[i][0],frames[i][1], c = frames[i][2], cmap = "RdYlGr
        plt.title("Infected = " + str(np.sum(frames[i][2])))
        plt.tick_params(left = False, right = False , labelleft = False,labelbc
        plt.savefig("frames/" + str(counter)+ ".png")
        images.append(imageio.imread("frames/" + str(counter)+ ".png"))
        counter += 1
        plt.close()

    imageio.mimsave(name, images)

    !rm -r frames

agents = []
for i in range(0,500):
    x = np.random.uniform(0,500)
    y = np.random.uniform(0,500)
    agents.append(Agent(x,y,0,.3))

for i in range(0,10):
    agents[i].infected = 1
    agents[i].infectedCounter = 50

stepSize = 5
xBounds = [0,500]
yBounds = [0,500]

frames = []
for i in range(0,500):
    agents = moveAgents(agents,stepSize,xBounds,yBounds)
    agents = rollInfect(agents)
    positions = getPosition(agents)
    infected = getInfected(agents)
    frames.append([positions[:,0] , positions[:,1], infected])

makeGif(frames,"Simulation.gif")
```

References

[1] Modeling Disease Spread BioInteractive
https://www.biointeractive.org/classroom-resources/modeling-disease-spread

[2] The SIR Model for Spread of Disease https://www.maa.org/press/periodicals/loci/joma/the-sir-model-for-spread-of-disease-the-differential-equation-model

[3] SIR Model Schematic Representation https://www.researchgate.net/figure/SIR-model-Schematic-representation-differential-equations-and-plot-for-the-basic-SIR_fig1_47676805

Simulation | Epidemiology | Python | Object Oriented | Agent Based Modeling

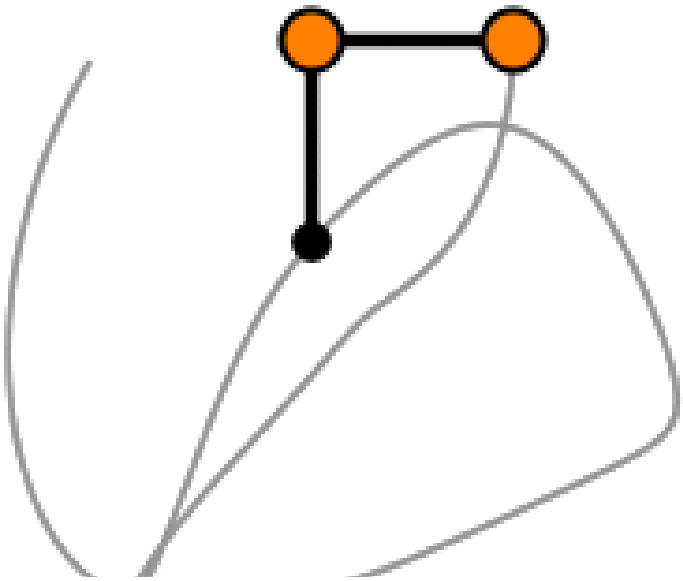
Follow

Written by Le Nguyen

100 Followers

Artificial Intelligence Research Scientist

More from Le Nguyen

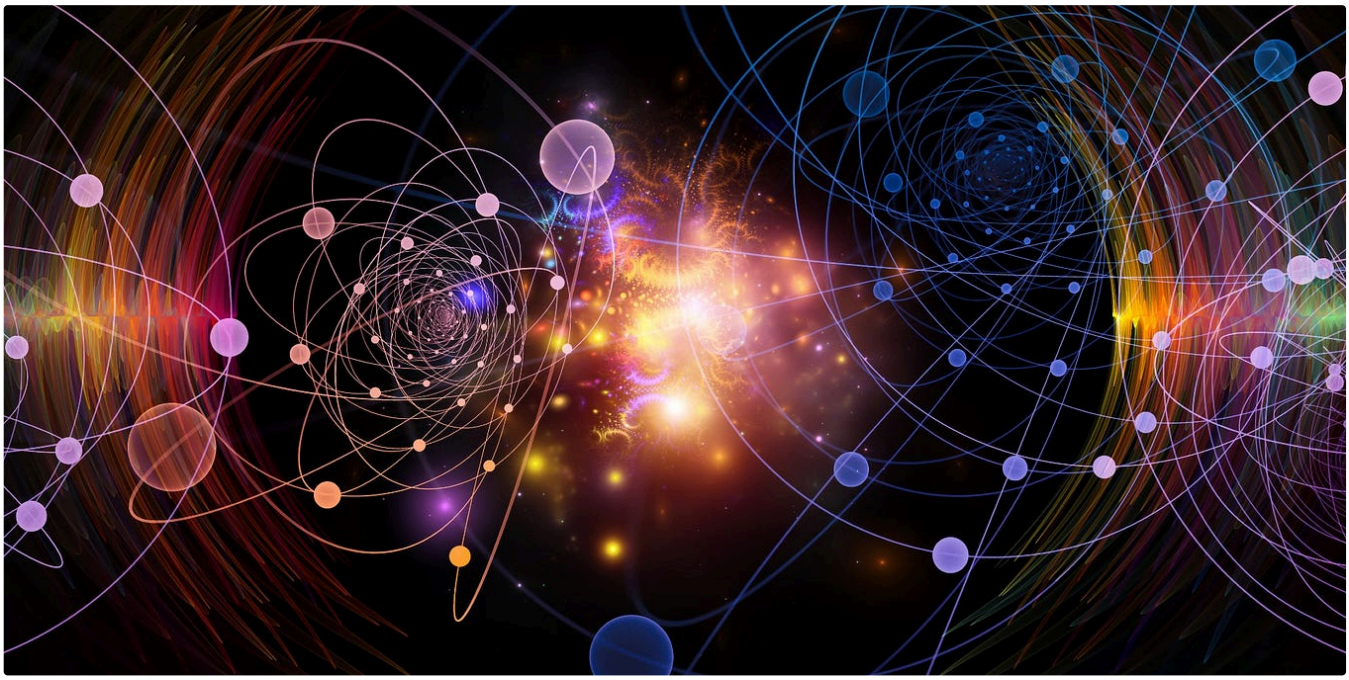


Le Nguyen in Towards Data Science

Simulation 105: Double Pendulum Modeling with Numerical Integration

Modeling a chaotic system

9 min read · Aug 14, 2023



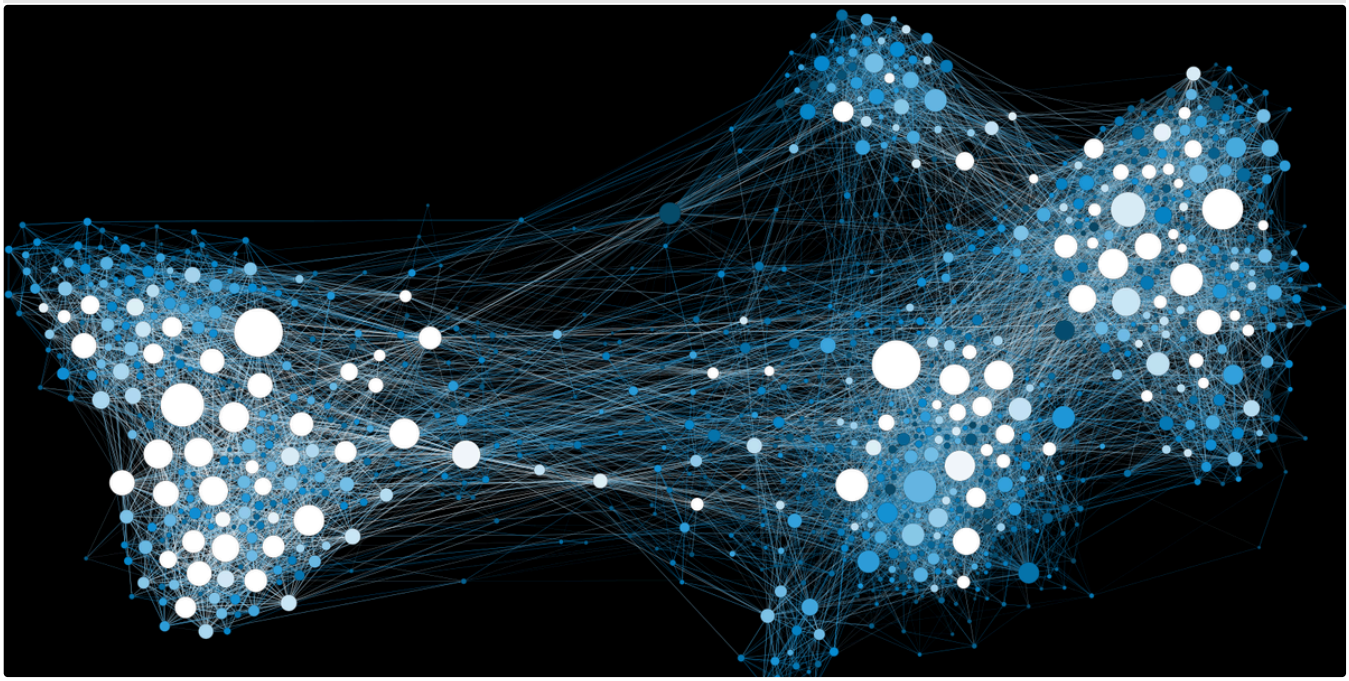
Le Nguyen

Simulation 103: Monte Carlo Modeling Quantum Mechanics

Quantum Mechanics is seen as an especially ominous field of physics but here we will develop the tools to model it

11 min read · Aug 3, 2023

15 2



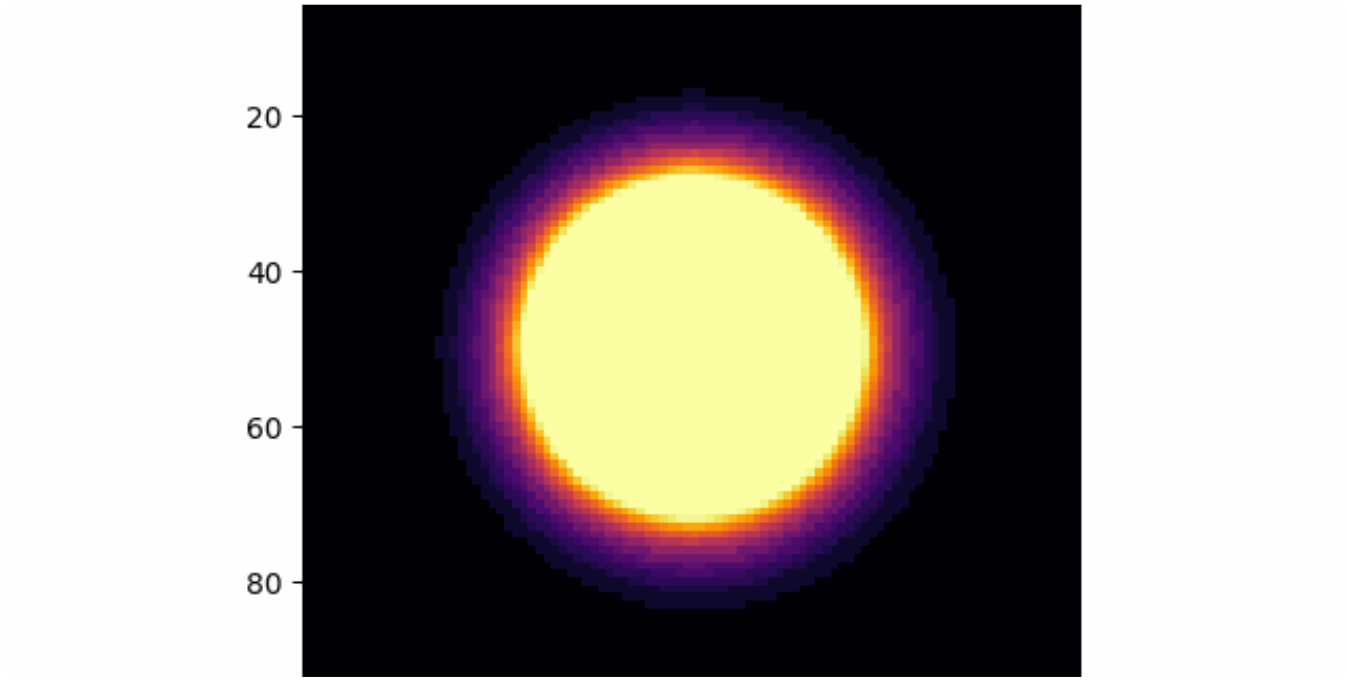
Le Nguyen in Towards Data Science

Simulation 106: Modeling Information Diffusion and Social Contagion with Networks

A graph-based approach to modeling the spread of information through social networks

· 12 min read · Sep 7, 2023

251



 Le Nguyen in Towards Data Science

Simulation 101: Conductive Heat Transfer

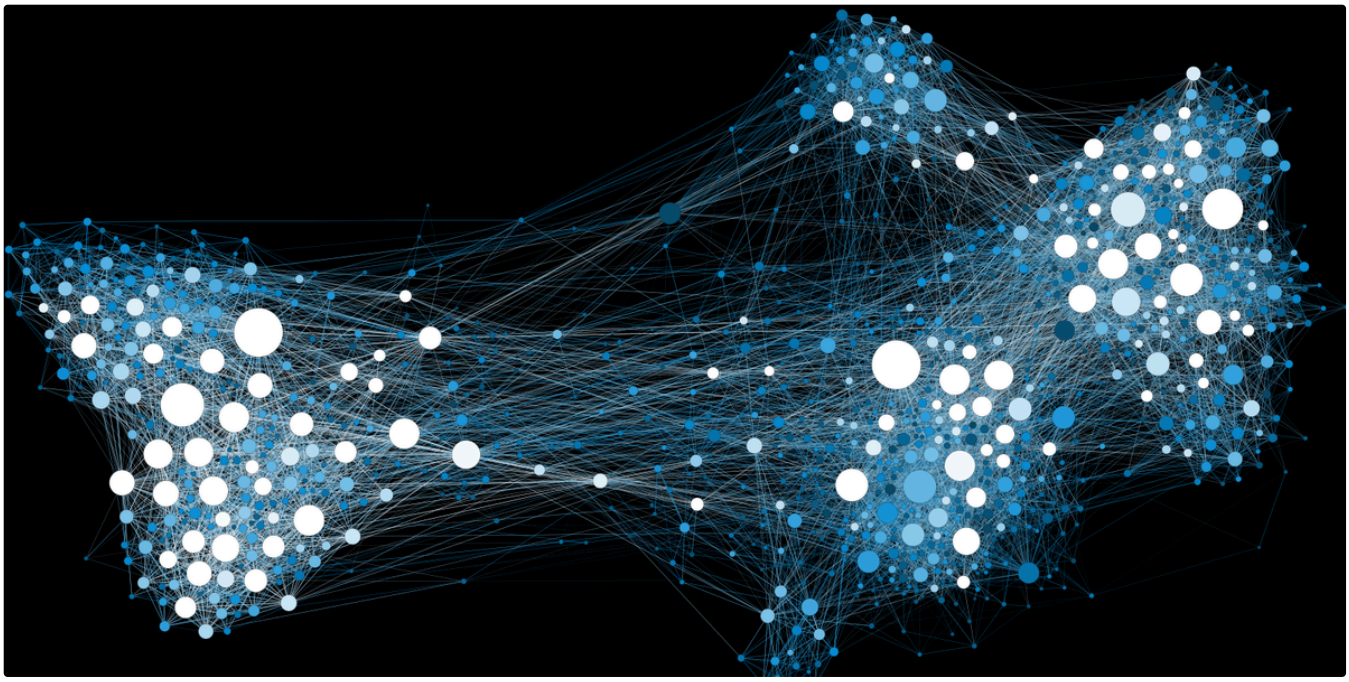
Conduction, or heat transfer between objects, is something we experience everyday. Here, we learn how to simulate the physics of...

11 min read · Jul 25, 2023

40

See all from Le Nguyen

Recommended from Medium

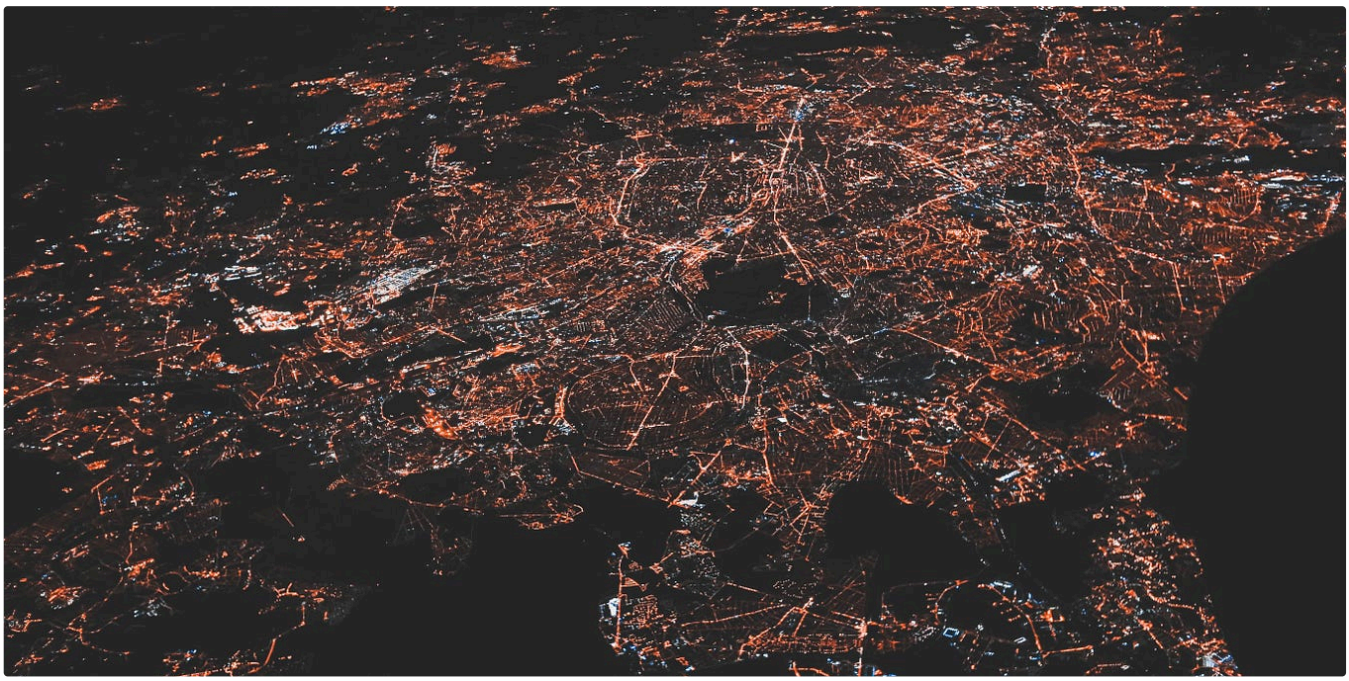


Le Nguyen in Towards Data Science

Simulation 106: Modeling Information Diffusion and Social Contagion with Networks

A graph-based approach to modeling the spread of information through social networks

· 12 min read · Sep 7, 2023



Ruby Abrams

Agent-Based Modeling

An overview

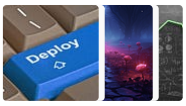
3 min read · Oct 4, 2023

16

Lists

Coding & Development

11 stories · 527 saves



Predictive Modeling w/ Python

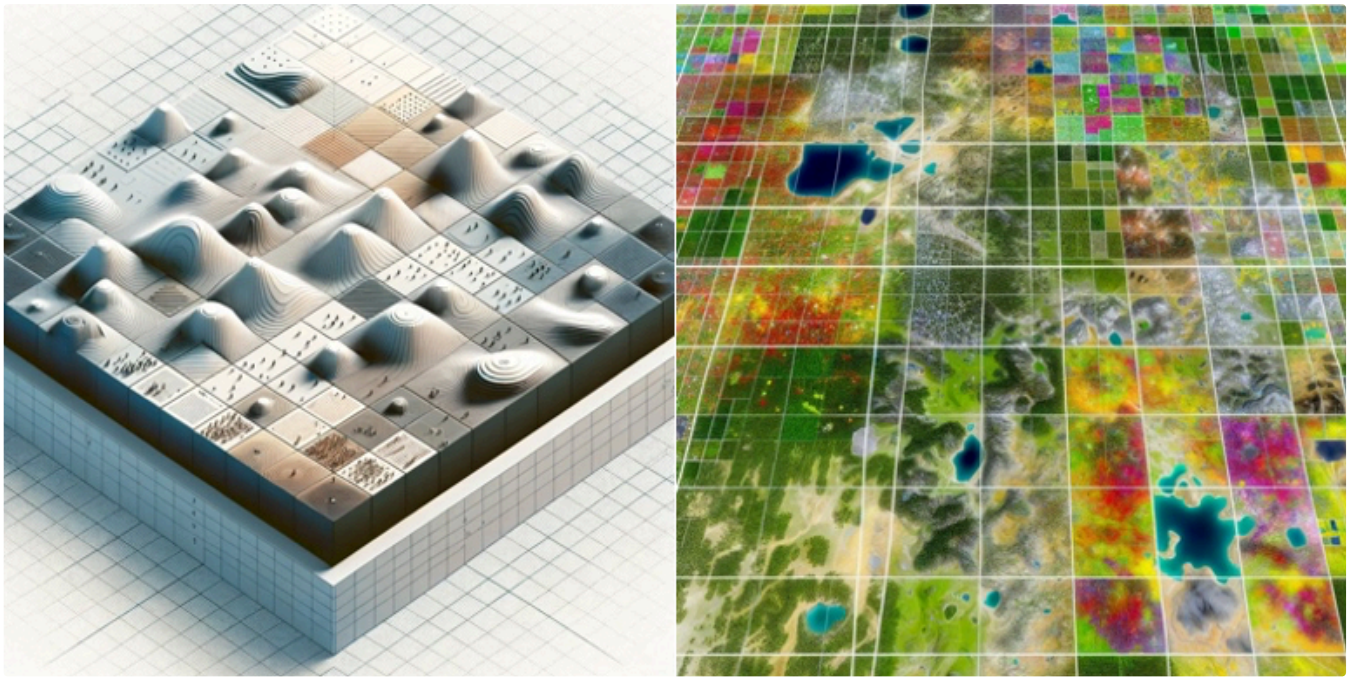
20 stories · 1042 saves

Practical Guides to Machine Learning

10 stories · 1249 saves

ChatGPT

21 stories · 544 saves



Leonardo Maldonado in Data And Beyond

Spatial Cross-Validation in Geographic Data Analysis

X: @ljmaldon | Personal website: www.leonardojmaldonado.com

· 4 min read · 6 days ago

54

Open	High	Low	Close	Volume	Dividends
745001	30.760000	30.740000	30.740000	662036	0.0
750000	30.760000	30.735001	30.750000	693719	0.0
750099	30.770000	30.740000	30.750000	476043	0.0
705000	30.760000	30.700001	30.750099	448260	0.0
684999	30.709999	30.670000	30.705000	567437	0.0

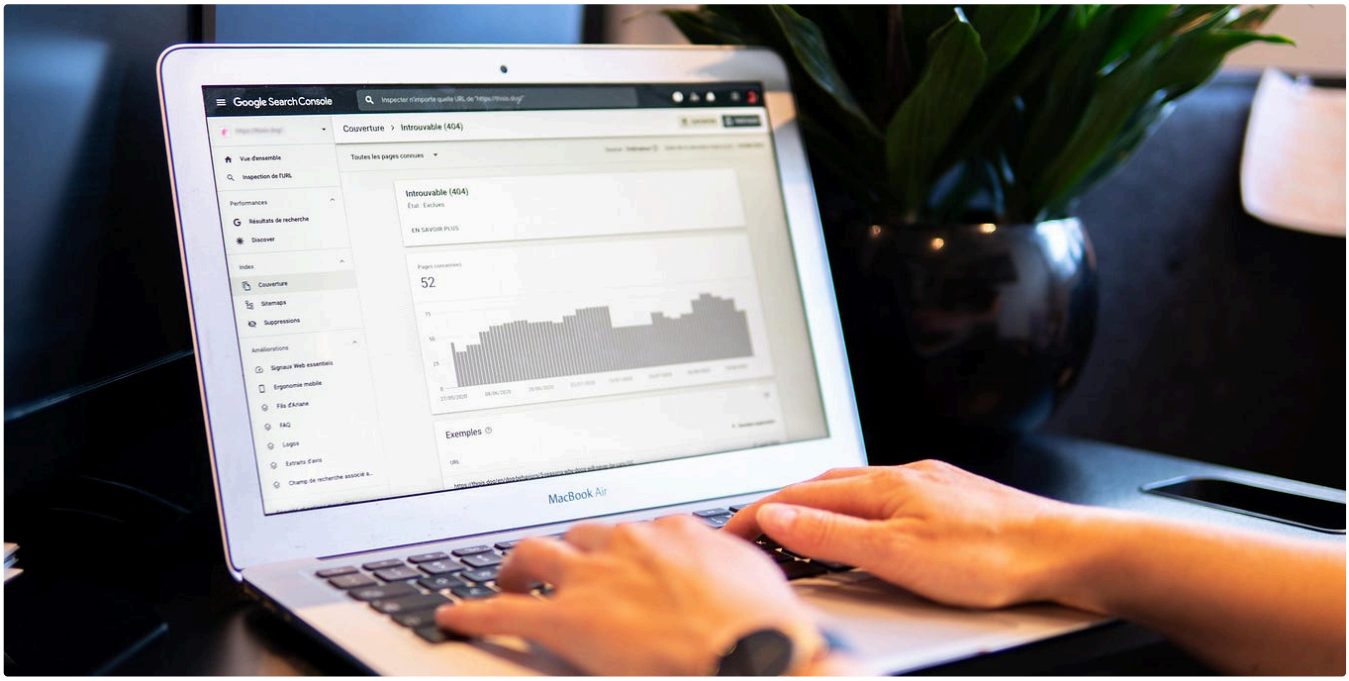
Nielsen Castelo Damasceno Dantas

Monte-Carlo Forecast using Python

Monte-Carlo simulation is an algorithmic method used to approximate a numerical value with random processes. Monte-Carlo is used where...

7 min read · Oct 12, 2023

106 1



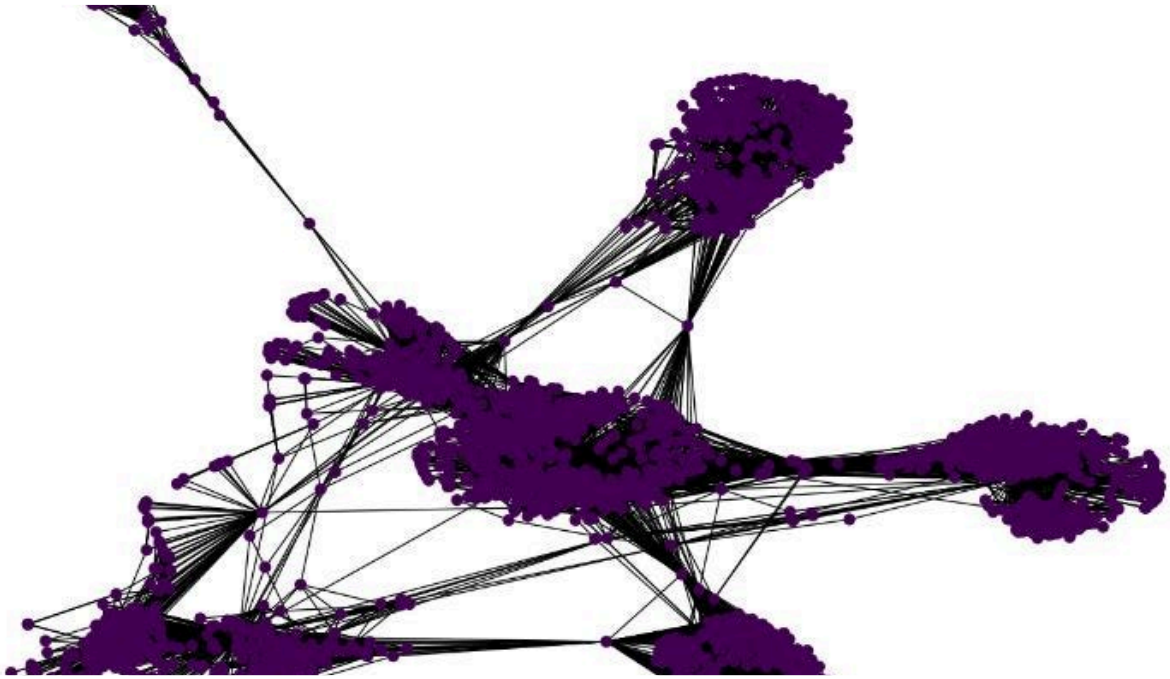
Data Overload

Exploring the Dynamics of Complex Systems: A Deep Dive into Agent-Based Modeling

In the realm of computational modeling, Agent-Based Modeling (ABM) has emerged as a powerful tool for understanding the intricate dynamics...

· 3 min read · Jan 9, 2024

36



Annapoornima S

Detecting Communities in large networks using networkx packages

objectives:

3 min read · Oct 6, 2023

See more recommendations