Lab1

Urlich Icimpaye, Jakob Häggström, Oscar Jacobson

April 2022

In order to see the gifs, please open the document with abdobe acrobat reader.

1 Firing brain

1.1 Rules

In this section a cellular automata with three simple rules was implemented and simulated on a 40 X 40 grid with periodic boundaries. Every cell has three possible states: "Ready", "Firing" and "Resting". The rules are:

- A *ready* cell with exactly two neighbours *firing* will fire at the next iteration.
- A *firing* cell will be *resting* at the next iteration.
- A *resting* cell will be *ready* at the next iteration.

1.2 Typical characteristics

The simulations typically shows the following characteristics after 10, 20, 100, and 1000 ticks:

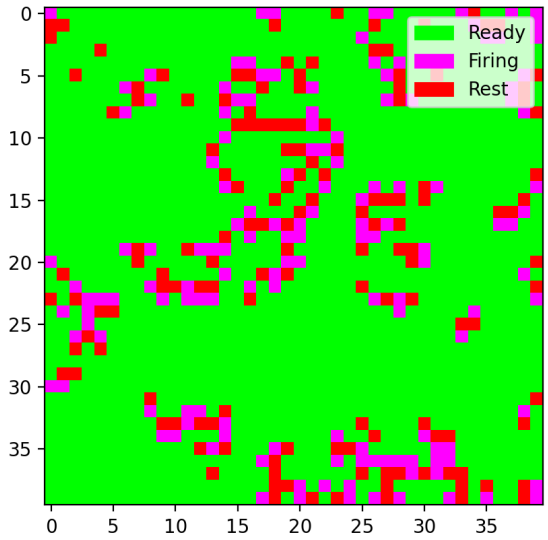


Figure 1: The mesh after 10 ticks.

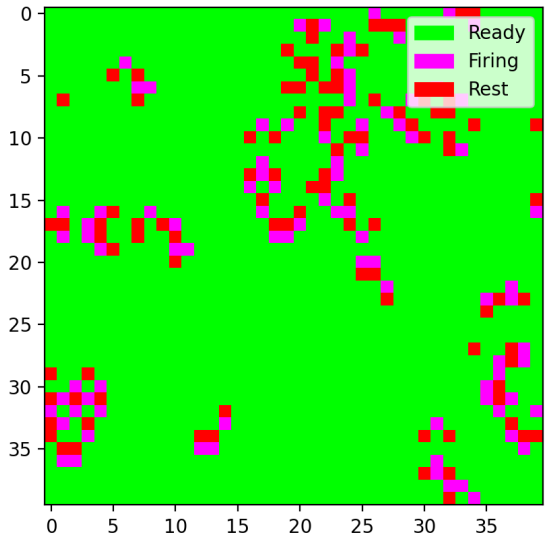


Figure 2: The mesh after 20 ticks.

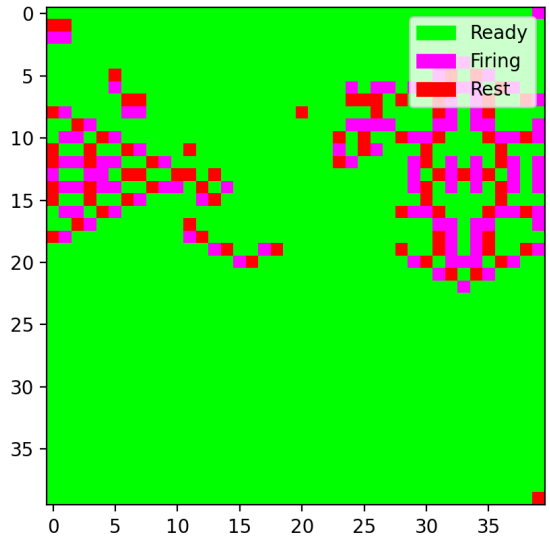


Figure 3: The mesh after 100 ticks.

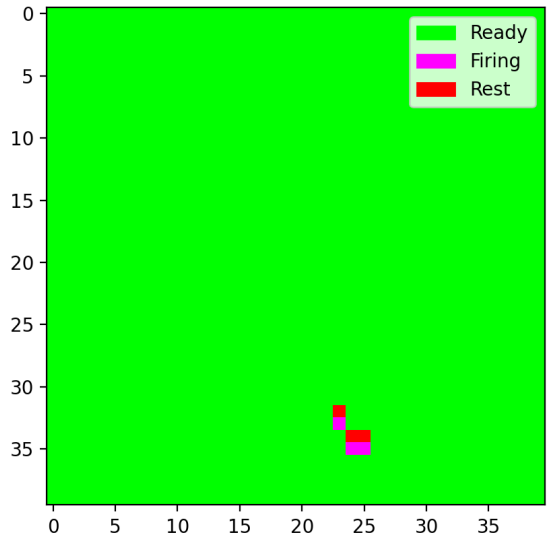


Figure 4: The mesh after 1000 ticks.

An average number of firing nodes was calculated for each time-tick for 100 different simulations with 0.1, 0.3 and 0.6 proability that a node is firing at the initial state. The average number of firing of nodes over time was also graphed when the initial state was chosen at random for each simulation a hundred times.

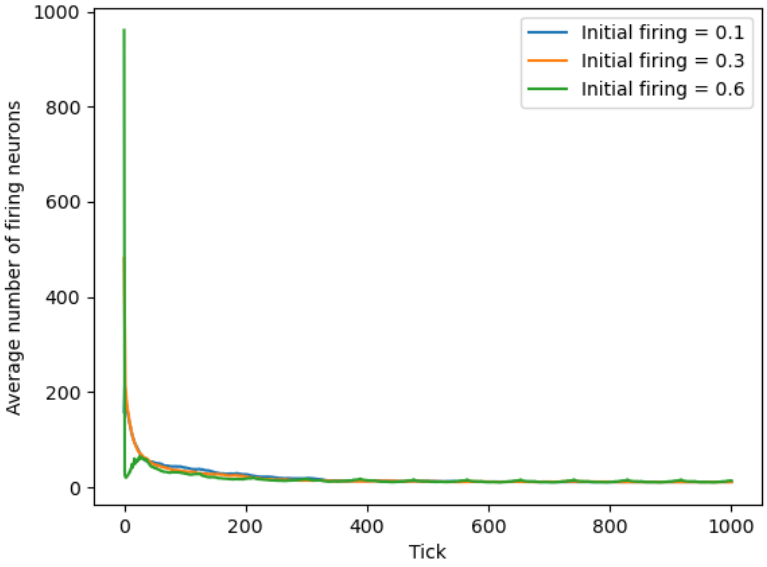


Figure 5: Average number of firing nodes at each tick.

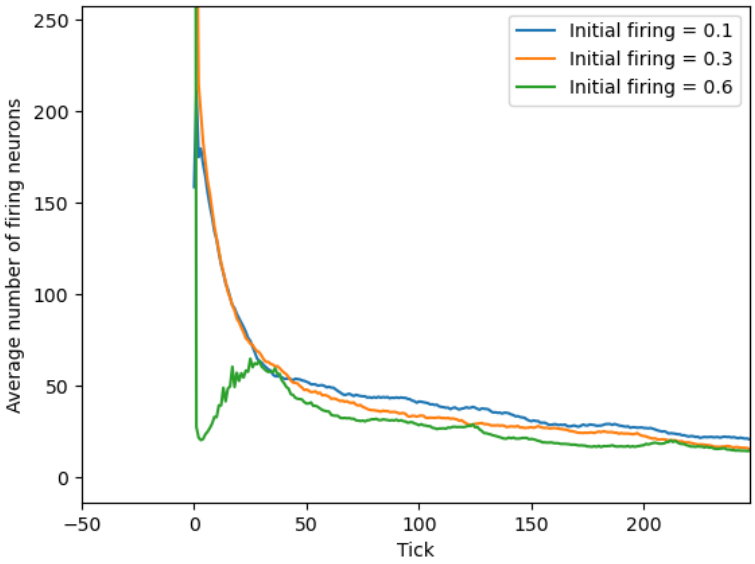


Figure 6: Zoom in of average number of firing nodes at each tick.

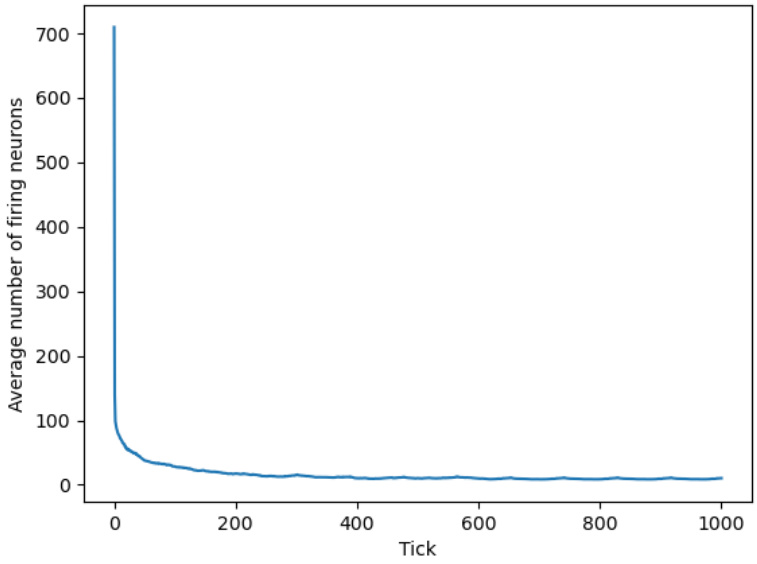


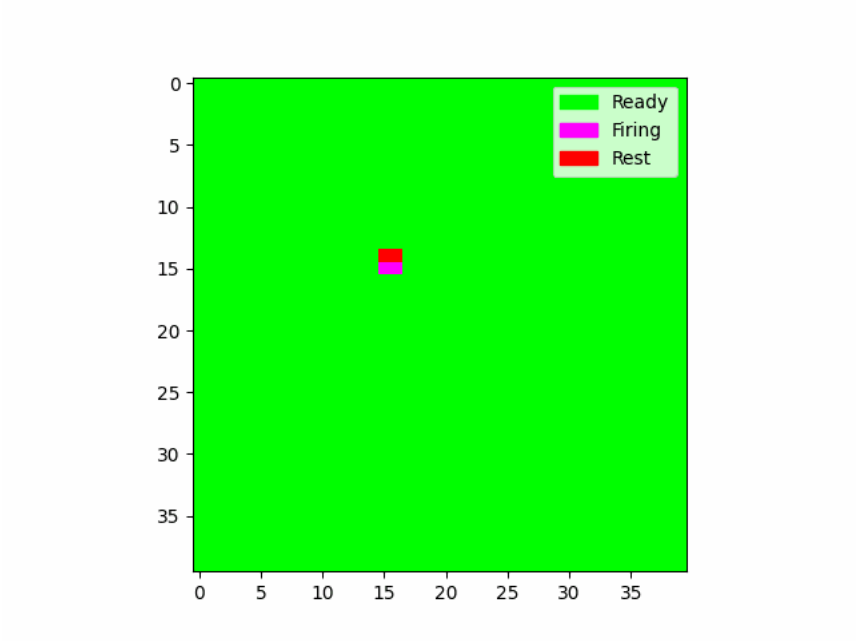
Figure 7: Zoom in of average number of firing nodes with randomized initial conditions.

As seen in figure 5-6, the initial states with lower probability of having firing cells in the initial state shows an exponential decay. Though the simulation with 0.6 probability decayed more rapidly the first timestep, and it stabilized to the same level of the other simulations. Moreover, the results when the initial condition is chosen at random seems to show similar behaviour as the simulations with 0.1 and 0.3 probability. This indicates that the most common behaviour of the decay is exponential. When the simulations reach a substantial amount of simulated steps, what typically happens is that larger structures collide with each other and vanishes. The ones that survives is smaller structures such as ships which rotate infinitely within the periodic boundaries. These ships continues to exist if they move in a trajectory that doesn't intercept with another structure, so most often there are only one or a few ships in the end of the simulation.

1.3 Interesting shapes

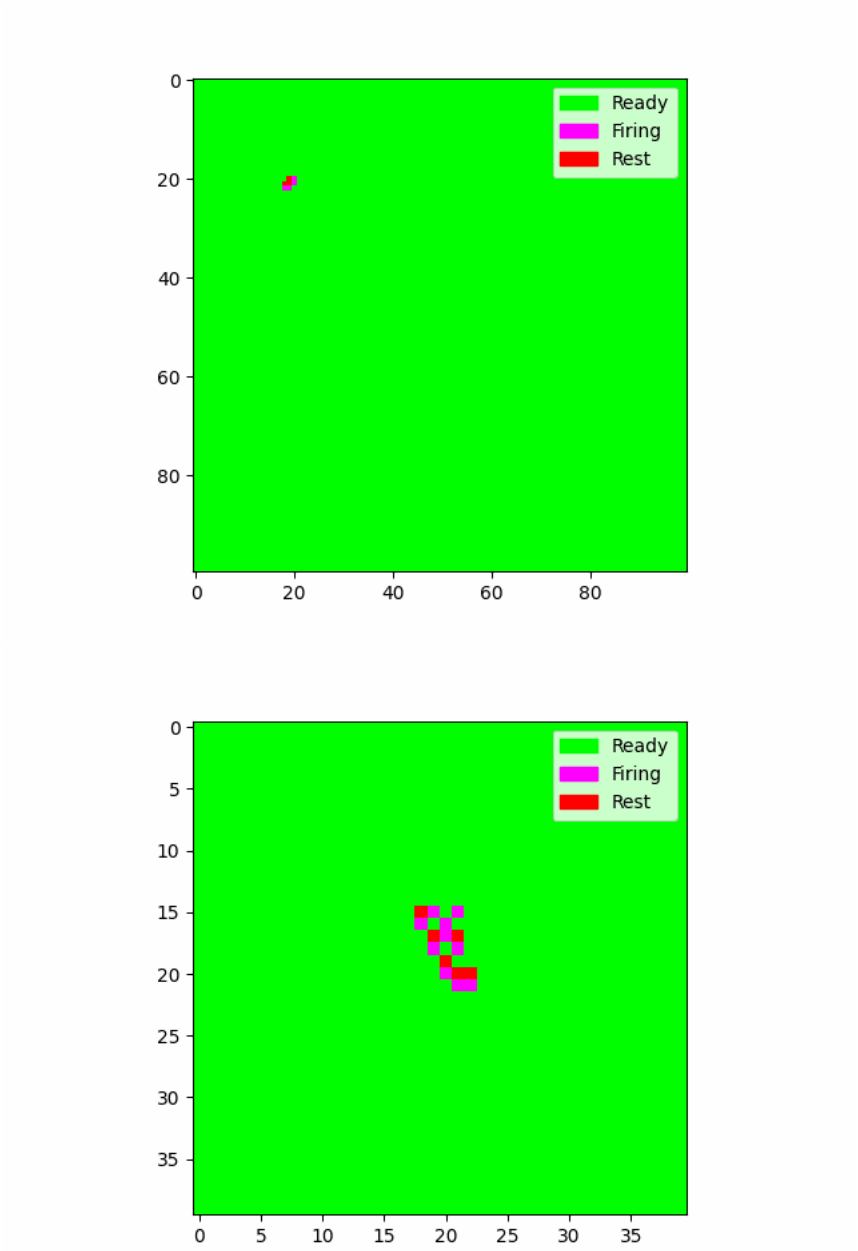
Some examples of equilibrium states or theoretically infinite and repeating patterns found within this set of rules are:
(Names coined by us)

1.3.1 The Glider



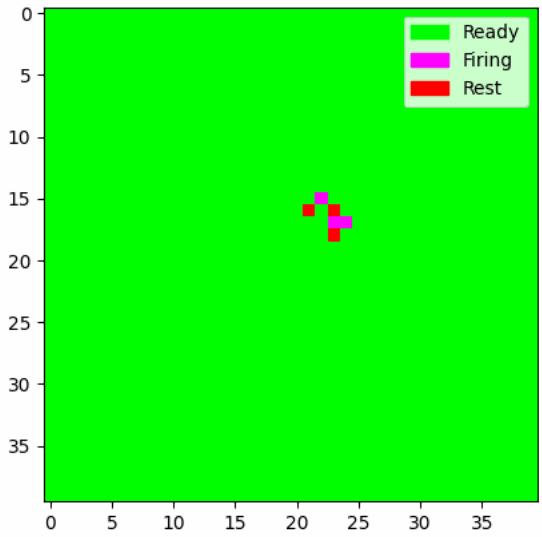
An example of shape 2a in the instructions.

1.3.2 The Bomber Ship or the Band



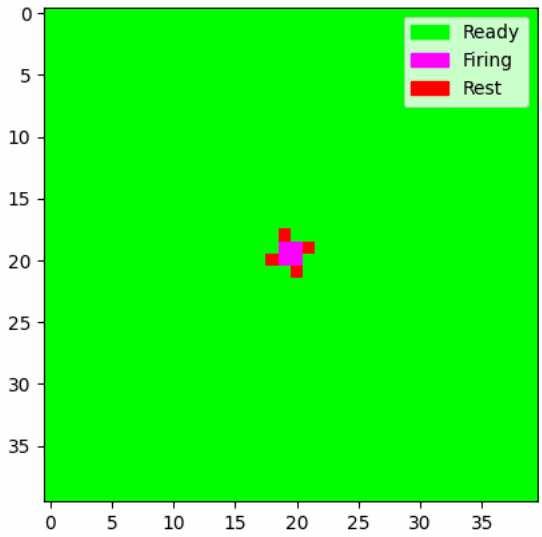
Examples of shape 2b in the instructions.

1.3.3 The Side Glider



An example of shape 2c in the instructions.

1.3.4 The Oscillator



An example of shape 2d in the instructions.

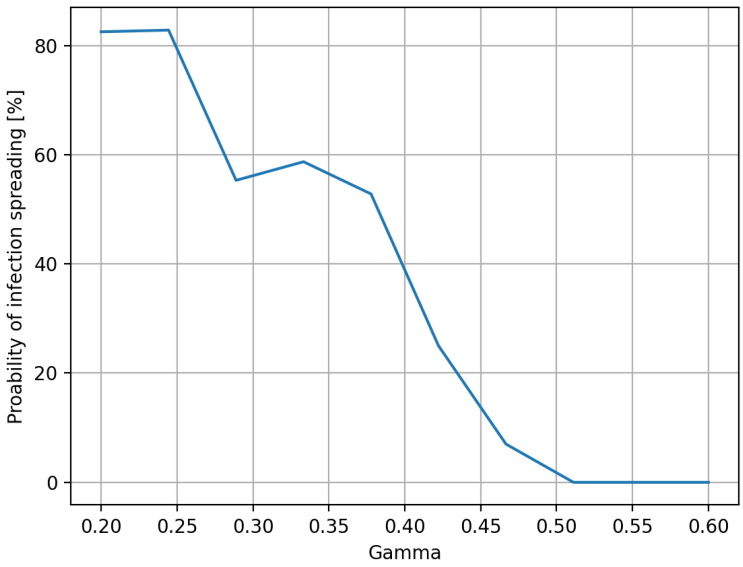


Figure 8: Probability of infection spreading vs gamma.

2 Spatial Epidemics

2.1 1D Rules

In this section a cellular automata with three different, but still simple, rules was implemented and simulated on a one dimensional grid with periodic boundaries. Every cell has two possible states: "Infected" or "Susceptible". The rules are:

- A *infected* cell is recovered with probability γ and therefore *susceptible* in the next iteration or still *infected* with probability $1 - \gamma$ in the next iteration.
- A *susceptible* cell with an *infected* neighbor becomes *infected* with probability $1 - \gamma$.
- A cell cannot be *infected* if already *infected* or *recover* if already *susceptible*.

2.2 1D Results

The one dimensional grid was implemented with N = 100 cells. All cells except one is set to be *susceptible* and one is set to be *infected* in the centre. The probability of a disease spreading was estimated by examining whether there is still infected cells after a certain timestep. This timestep was chosen to be at 50 ticks. If there is zero number of infected individuals before that, then the simulation is ended. This was repeated 100 times for different values of γ, and the probability was estimated by calculating the average amount of simulations that resulted in a surviving infection at timestep 50.

As seen in figure 8, the probability of the infections spreading seems to decrease linearly with increasing gamma until it reaches a threshold value, which seems to be around 0.5. After this

point, the probability of not getting infected and being recovered is too high in order for the infection to spread. By this logic this system should be increasing at γ below 0.5 and increasing for γ over 0.5.

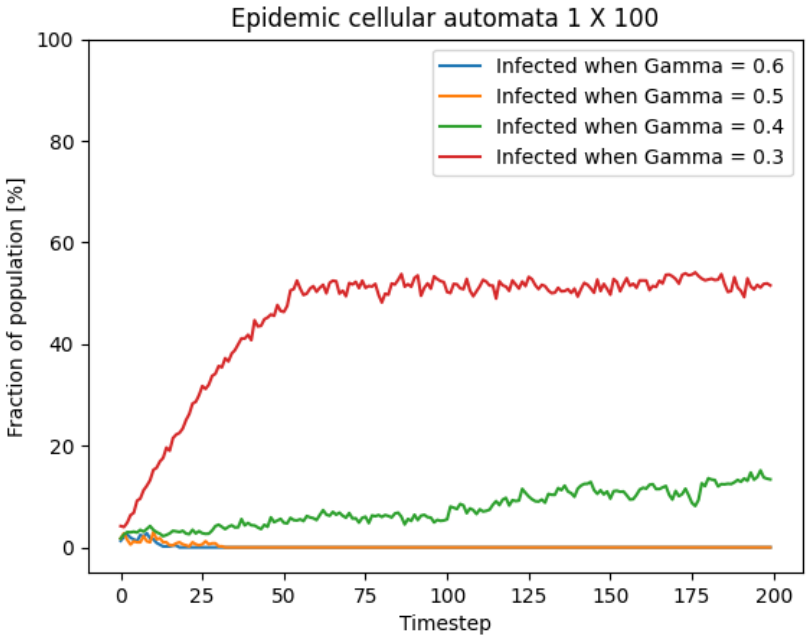


Figure 9: Simulations of epidemic caused by one initial infected cell for different gamma values.

Figure 9 shows the effect of placing one infected cell within in a line of 100 *susceptible* cells for different infection rates. From the figure we can tell that for low infection rates the epidemic dies out quickly. Interestingly when using high infection rates the spread does not increase to 100%, instead there appears to exist an equilibrium state where the spread of the disease is equal to the recovery rate. This is possibly the cause of the fact that the automata is implemented on a line of 100 cells restricting the amount of influence one *infected* cell can have to its two neighbours. This combined with the rule that an infected cell cannot become infected "twice" which guarantees that there will always be a fraction of the population that becomes *susceptible* in every timestep.

The simulation in figure 9 was only ran once. The epidemic seems to have died out for $\gamma = 0.5$ but only after about 30 timesteps As we ran the simulation once and only using a single infected cell to start nothing significant can be said about this observation but together with figure 8 this seems logical as $\gamma = 0.5$ is close to the point of never spreading.

An educated guess using both figures be that the green line in figure 9 is close to its final equilibrium state and will cease to increase at some maximum value. The slow increase is the cause of $\gamma = 0.4$ making the system very slightly increasing compared to highly increasing for $\gamma = 0.3$ and the system being decreasing for higher values of γ.

2.3 1D simulation with modified initial condition

The same cellular automata was then implemented and simulated with an initial state of every cell having a p percent chance of being infected at the first timestep. By changing the initial infection rate p and the transmission rate γ the long term chance of survival for the infection was plotted as a function of p and γ

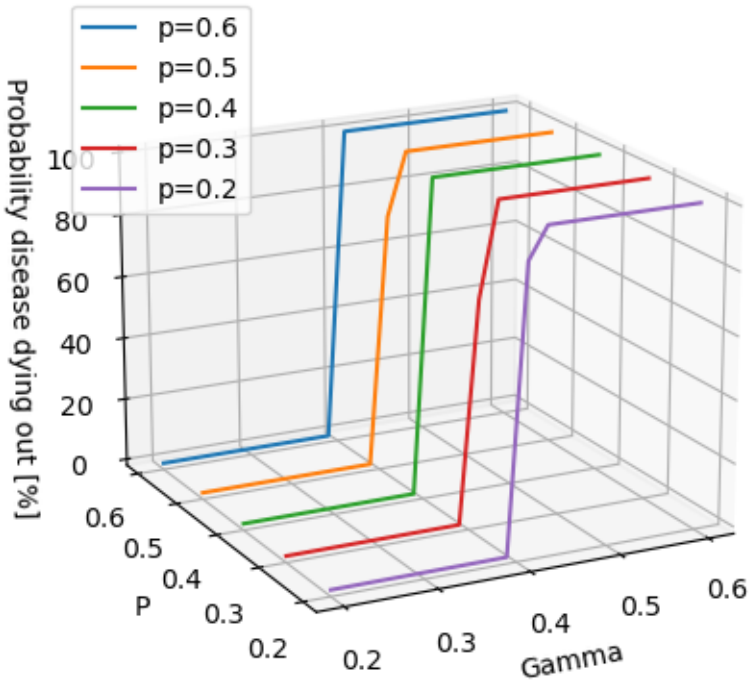


Figure 10: 3D plot of the survival chance of the infection vs γ for different initial conditions on p

The figure 10 seems to show some different characteristics compared to what was assumed in the previous section. When using initial conditions that is not only one single cell the chance of the epidemic dying out within the first 50 steps are significantly lower. Using these rules the only deciding factor for survival or death is the increasing or decreasing nature of the system, which is decided by the value of gamma.

The described system seems to have the characteristics of a classifier as the value of gamma almost entirely and or precisely decides the state of survival after a sufficiently large amount of timesteps. Given this information figure 10 points to that the cellular automata works as an eroder when γ is below 0.4 and as an constructor(?) when γ is larger than 0.4.

3 3.c N x N Epidemic simulation

In this section a cellular automata with five rules was implemented and simulated on a N x N dimensional grid with periodic boundaries. Every cell has three possible states: *"Infected"*, *"Susceptible"* or *"Recovered"*. The rules are:

- A *susceptible* cell with an *infected* neighbor becomes *infected* with probability γ.
- A *infected* cell is **recovered** with probability μ and therefore *susceptible* in the next iteration or still *infected* with probability $1 - \mu$ in the next iteration.
- A *infected* cell that **recovers** is immune with probability ν and therefore *recovered* in the next iteration or *susceptible* with probability $1 - \nu$ in the next iteration.
- A *recovered* cell is immune and cannot be *infected* again in any timestep.
- A cell cannot be *infected* if already *infected*, recover if already *susceptible* or immune if already *recovered*.

Examination of coefficients should find, $\frac{\gamma}{\gamma+\mu}$ will decide upper bound of equilibrium state. ν will set an upper bound on how long the epidemic is still around. ν will function like lowering the γ coefficient over time. When $\frac{\gamma(\nu * time)}{\gamma+\mu}$ gets too low the epidemic will die out. Having ν as anything but 0.0 will cause the automata to be an eroder as the epidemic eventually dies out. If we introduce something that causes immune cells to be susceptible again this might change depending on parameter values.

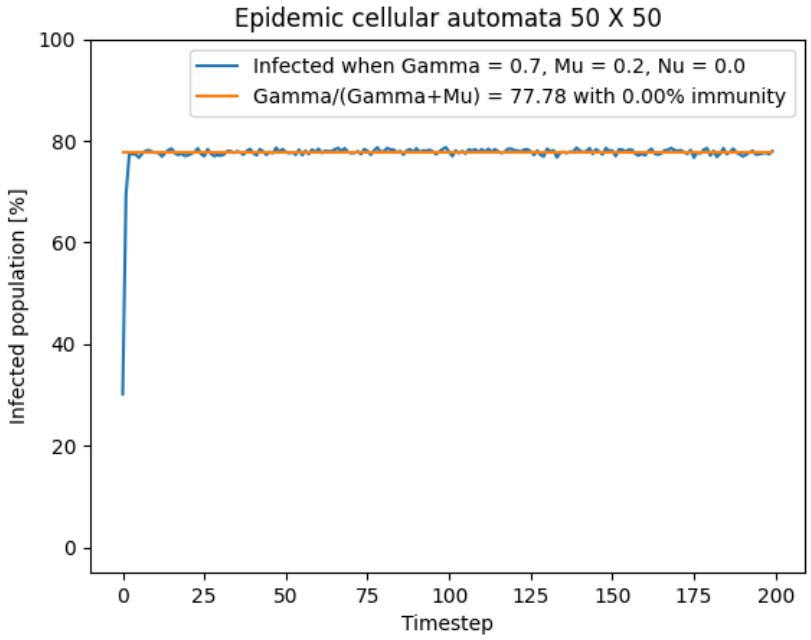


Figure 11: 50 x 50 automata with no immunity

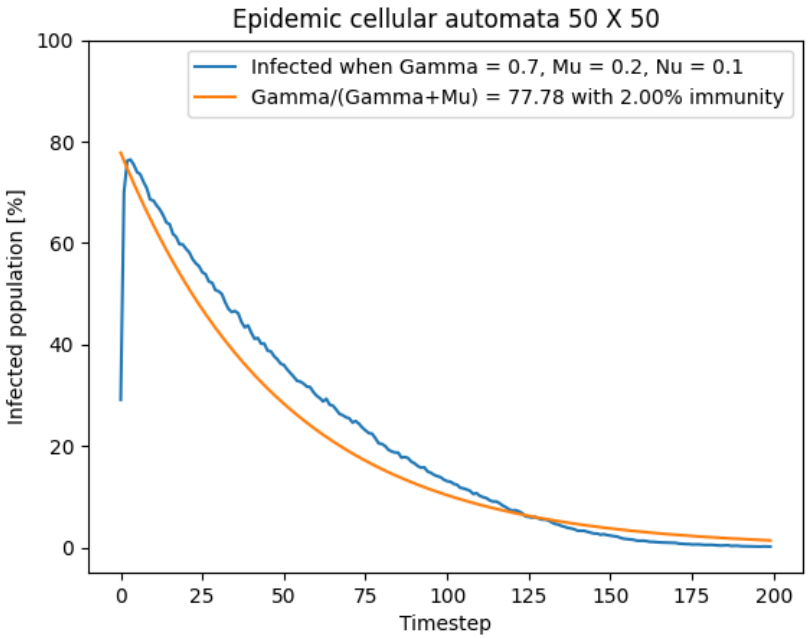


Figure 12: 50 x 50 automata with immunity

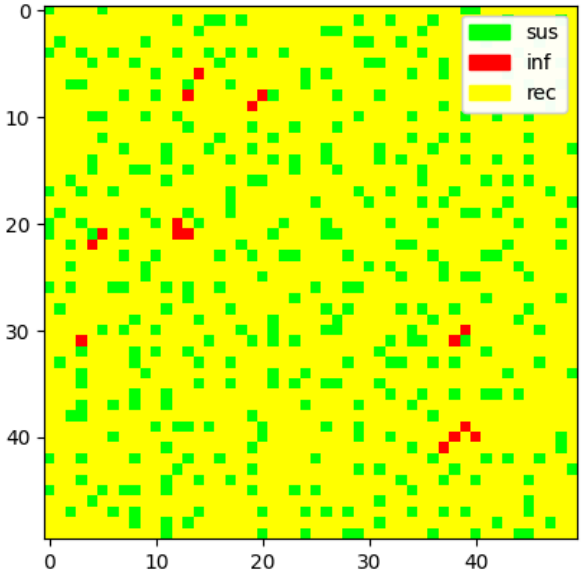


Figure 13: 50 x 50 automata with immunity at timestep 200

Graph 11 and 12 shows the average amount of infected cells over 3 runs of 200 time steps and initial infection probability = 0.3. The figures show the existence of an equilibrium state that is predictable at $\frac{\gamma}{\gamma+\mu}$ as well as the decreasing trend of introducing immunity into the automata. Immunity can be approximated as a percentage decrease over every timestep which

depends on the amount of infected and possibly recovering cells from the previous timestep.

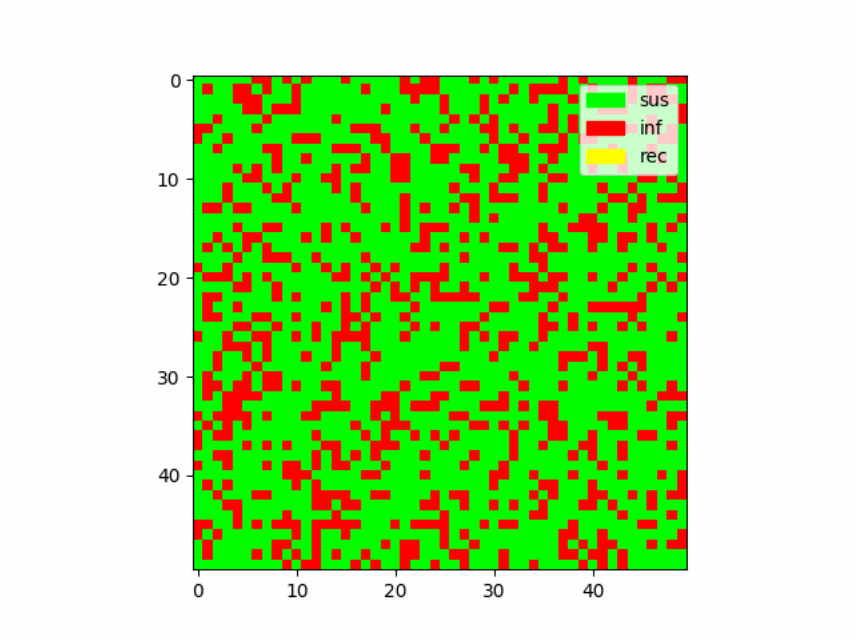


Figure 13 points to the final stages of isolating and dying outbreaks for an highly transmissible epidemic with a chance for immunity when recovering. The epidemic is being contained and not allowed to travel between susceptible cells. Immune people create immunity "walls" and protect the small minority that still isn't immune. This is a very useful concept in reality where a small minority of people often cannot get vaccinated because of inevitable medical reasons. The immune "herd" protects the *susceptible* minority.

Appendix

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import matplotlib.patches as mpatches
5  import datetime
6  from celluloid import Camera
7
8  color_dict = {'Ready': np.array([0, 255, 0]), 'Firing': np.array([255, 0, 255]), /
       'Rest': np.array([255, 0, 0])}
9
10
11
12 def generate_mesh(N, init = 0.3):
13     return np.random.choice(['Ready', 'Firing'], size = (N,N), p = [1 - init, init])
14
15 def get_number_firing(mesh):
16
17     return len(list(filter(lambda x: x == 'Firing', mesh.flatten())))
18
19 def is_firing(kernel):
20
21     temp_kernel = kernel.flatten().tolist()
```

```
22    temp_kernel.pop(4)
23
24    if len(list(filter(lambda x: x == 'Firing', temp_kernel))) == 2:
25       return True
26
27    return False
28
29 def step(mesh_old, mesh_new, N):
30    padded_old = np.pad(mesh_old,1, 'wrap')
31
32    for i in range(1,N+1):
33       for j in range(1,N+1):
34
35          if mesh_old[i-1,j-1] == 'Ready':
36
37             if is_firing(padded_old[i-1:i+2,j-1:j+2]):
38
39                mesh_new[i-1,j-1] = 'Firing'
40
41          elif mesh_old[i-1,j-1] == 'Firing':
42
43             mesh_new[i-1,j-1] = 'Rest'
44
45          elif mesh_old[i-1,j-1] == 'Rest':
46
47             mesh_new[i-1,j-1] = 'Ready'
48
49    return mesh_new
50
51 def print_mesh(mesh):
52    for line in mesh:
53       print('␣'.join(map(str,line)))
54    print()
55
56 def plot_mesh(mesh,N, camera = None):
57    res = []
58    for i in range(N):
59       temp = []
60       for j in range(N):
61          temp.append(color_dict[mesh[i,j]])
62       res.append(temp)
63
64    legend = [mpatches.Patch(color = color_dict[state] / 255, label = f"{state}") /
          for state in color_dict]
65
66
67
68    plt.imshow(res)
69    plt.legend(handles = legend)
70    if camera is not None:
71       camera.snap()
72
73
74 def run_sim(N, n_steps, plot_res = True,record = False, get_firing = False, init /
       = 0.3):
75    mesh = generate_mesh(N, init)
76    nr_firing = []
77    if get_firing:
78       nr_firing.append(get_number_firing(mesh))
79
```

```
80
81    if record:
82       fig = plt.figure()
83       camera = Camera(fig)
84       plot_mesh(mesh,N, camera)
85
86    if plot_res:
87       plot_mesh(mesh,N)
88
89    for i in range(n_steps):
90
91       mesh = step(mesh, mesh, N)
92
93       if record:
94          plot_mesh(mesh,N,camera)
95       if get_firing:
96          nr_firing.append(get_number_firing(mesh))
97
98    if record:
99       date_now = datetime.datetime.now()
100      date_string = date_now.strftime("%Y_%m_%d_%H_%M_%S")
101      animation = camera.animate()
102      animation.save(f"sim_{date_string}.gif", writer = 'imagemagick')
103
104   if plot_res:
105      plot_mesh(mesh,N,camera)
106
107   return np.array(nr_firing)
108
109 if __name__ == "__main__":
110
111   N = 40
112   n_steps = 1000
113   record = False
114   plot_res = False
115   get_firing = True
116   n_sim = 100
117   init_cond = np.random.random(n_sim)
118
119   #res = run_sim(N, n_steps,plot_res,record,get_firing)
120   plt.figure()
121
122   for i in range(n_sim):
123      if i == 0:
124         res = run_sim(N, n_steps,plot_res,record,get_firing, init_cond[i])
125
126      else:
127         temp = run_sim(N, n_steps,plot_res,record,get_firing, init_cond[i])
128         res = res + temp
129
130      print(f"{i}")
131
132   res = res / n_sim
133   plt.plot(list(range(1001)), res)
134   np.savetxt(f"init_rand.txt", res)
135
136
137   plt.xlabel("Tick")
138   plt.ylabel("Average number of firing neurons")
```

```
139    plt.show()
```

Listing 1: The basis of the code for all assignments in part one.

```
1
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import time
5
6  def generate_mesh(N, prob):
7     return np.random.choice(['sus','inf'],size = N , p = [1 - prob, prob])
8
9  def recover_or_infected(gamma):
10
11    return np.random.choice(['sus','inf'], p = [gamma, 1 - gamma])
12
13
14 def get_nr(mesh):
15
16    nr_inf = len(list(filter(lambda x: x == 'inf', mesh)))
17    nr_sus = len(mesh) - nr_inf
18
19    return [nr_inf, nr_sus]
20
21 def step(mesh_old, mesh_new, N, gamma):
22
23    for i in range(N):
24
25       if mesh_old[i] == 'inf':
26
27          mesh_new[i] = recover_or_infected(gamma)
28
29       elif any(np.take(mesh_old, [i-1, i+1], mode = 'wrap') == 'inf'):
30
31          mesh_new[i] = recover_or_infected(gamma)
32
33    return mesh_new
34
35 def plot_mesh(result,N, N_steps):
36
37    result = np.array(result)
38
39    plt.figure()
40    plt.plot(list(range(N_steps)), result[:,0]* 100 / N , label = 'Infected')
41    plt.plot(list(range(N_steps)), result[:,1]* 100 / N, label = 'Susceptible')
42    plt.legend()
43    plt.xlabel('Tick')
44    plt.ylabel('Fraction␣of␣population␣[%]')
45    plt.show()
46
47
48
49 def run_sim(N, n_steps, gamma, p):
50
51    mesh = generate_mesh(N, p)
52    result = []
53
54    np.random.seed(seed=int(time.time()))
55
56    for i in range(n_steps):
57
```

```
        mesh = step(mesh, mesh, N, gamma)

        tempres = get_nr(mesh)

        result.append(tempres)

        if tempres[0] == 0:

            break

    #plot_mesh(result,N,n_steps)

    return result


if __name__ == "__main__":

    N = 100
    n_steps = 1500
    num_sim = 10

    gamma_list = np.linspace(0.6,0.2, 20)
    prob_list = np.linspace(0.6,0.2, 5)
    p_prob = []
    avg_prob = []
    for p in prob_list:
        avg_prob = []
        print(f"p = {p}")
        for gamma in gamma_list:
            tot_result = 0
            for i in range(num_sim):
                res = run_sim(N, n_steps, gamma, p)
                res = np.array(res)
                if res[-1,0] == 0:
                    tot_result += 1
            tot_result = tot_result / num_sim
            avg_prob.append(tot_result)
        p_prob.append(avg_prob)

    p_prob = np.array(p_prob)

    fig = plt.figure()
    ax = plt.axes(projection = '3d')

    for i,p in enumerate(prob_list):

        p_list = [p for i in range(len(gamma_list))]
        ax.plot3D(gamma_list,p_list, p_prob[i,:] * 100, label = f"p={p}")


    ax.set_xlabel('Gamma')
    ax.set_ylabel('P')
    ax.set_zlabel('Probability disease dying out')
    ax.legend()
    # plt.grid(True)
    # plt.show()
```

Listing 2: The basis of the code for the 1D epidemic simulation

```

import numpy as np
```

```
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import datetime
from celluloid import Camera

color_dict = {'sus': np.array([0, 255, 0]), 'inf': np.array([255, 0, 0]), 'rec': /
    np.array([255, 255, 0])}



def generate_mesh(N, prob):
   return np.random.choice(['sus','inf'], size = (N,N), p = [1 - prob, prob])

def recover_or_infected(mu, ny):

   res = np.random.choice(['rec','inf'], p = [mu, 1 - mu])

   if res == 'rec':
      return np.random.choice(['rec','sus'], p = [ny, 1 - ny])

   return res

def get_stats(mesh, N):
   inf = len(list(filter(lambda x: x == 'inf', mesh.flatten())))
   sus = len(list(filter(lambda x: x == 'sus', mesh.flatten())))
   return [inf,sus, N*N - inf - sus]

def is_infected(kernel, gamma):

   temp_kernel = kernel.flatten().tolist()
   temp_kernel.pop(4)

   if len(list(filter(lambda x: x == 'inf', temp_kernel))) >= 1:
      return np.random.choice(['sus','inf'], p = [1-gamma, gamma])

   return 'sus'

def step(mesh_old, mesh_new, N, gamma, mu, ny):
   padded_old = np.pad(mesh_old,1, 'wrap')

   for i in range(1,N+1):
      for j in range(1,N+1):

         if mesh_old[i-1,j-1] == 'inf':

            mesh_new[i-1,j-1] = recover_or_infected(mu, ny)

         elif mesh_old[i-1,j-1] == 'sus':

            mesh_new[i-1,j-1] = is_infected(padded_old[i-1:i+2,j-1:j+2], gamma)

         elif mesh_old[i-1,j-1] == 'rec':

            continue

   return mesh_new

def print_mesh(mesh):
   for line in mesh:
      print('␣'.join(map(str,line)))
```

```
62    print()
63
64 def plot_mesh(mesh,N, camera = None):
65    res = []
66    for i in range(N):
67       temp = []
68       for j in range(N):
69          temp.append(color_dict[mesh[i,j]])
70       res.append(temp)
71
72    legend = [mpatches.Patch(color = color_dict[state] / 255, label = f"{state}") /
          for state in color_dict]
73
74
75
76    plt.imshow(res)
77    plt.legend(handles = legend)
78    if camera is not None:
79       camera.snap()
80
81 def plot_meshx(result,N, N_steps, gamma,mu,nu):
82    gmn = []
83    result = np.array(result)
84    #print(result[:,0])
85    gmn.append(gamma/(gamma+mu)*100)
86    # sus * gamma = 1-sus * mu
87    # sus * gamma = mu - sus*mu
88    # sus*gamma + sus*mu = mu
89    # sus(gamma+mu)=mu
90    # sus = mu/(gamma+mu)
91    for x in range(N_steps-1):
92       gmn.append(gmn[x]-(gmn[x]*mu*nu))
93
94
95
96
97    plt.plot(list(range(N_steps)), result[:,0] , label = 'Infected␣when␣Gamma␣=␣' /
          + str(gamma) + ',␣Mu␣=␣' + str(mu) + ',␣Nu␣=␣' + str(nu))
98    plt.plot(list(range(N_steps)), gmn, label = 'Gamma/(Gamma+Mu)␣=␣' + /
          str("{:.2f}".format(gmn[0])) + '␣with␣' + str("{:.2f}".format(mu*nu*100)) + /
          '%␣' + 'immunity')
99    #plt.plot(list(range(N_steps)), result[:,1]* 100 / N, label = 'Susceptible')
100   plt.legend()
101   plt.xlabel('Timestep')
102   plt.ylim(-5,100)
103   plt.ylabel('Infected␣population␣[%]')
104   plt.title(f'Epidemic␣cellular␣automata␣{N}␣X␣{N}')
105
106
107 def run_sim(N, n_steps,prob,gamma,mu,ny, plot_res = True,record = False, /
       get_firing = False):
108   mesh = generate_mesh(N, prob)
109   stats = []
110   if get_firing:
111      stats.append(get_stats(mesh, N))
112
113
114   if record:
115      fig = plt.figure()
116      camera = Camera(fig)
```

```
117         plot_mesh(mesh,N, camera)
118 
119     if plot_res:
120         plot_mesh(mesh,N)
121 
122     for i in range(n_steps):
123 
124         mesh = step(mesh, mesh, N,gamma, mu, ny)
125 
126         if record:
127             plot_mesh(mesh,N,camera)
128         if get_stats:
129             stats.append(get_stats(mesh, N))
130 
131     if record:
132         date_now = datetime.datetime.now()
133         date_string = date_now.strftime("%Y_%m_%d_%H_%M_%S")
134         animation = camera.animate()
135         animation.save(f"sim_{date_string}.gif", writer = 'imagemagick')
136 
137     if plot_res:
138         plot_mesh(mesh,N,camera)
139 
140     return np.array(stats)
141 
142 if __name__ == "__main__":
143     gmn = 0
144     imm = 0
145     N = 50
146     n_steps = 200
147     num_sim = 3
148     record = True
149     plot_res = False
150     get_firing = True
151     prob = 0.3 #Initial spread
152     gamma = 0.3 #Spread coef
153     mu = 0.2 #Recover coef
154     ny = 0.1 #Chance for immune after recover
155 
156     #stats = run_sim(N, n_steps,prob,gamma,mu,ny,plot_res,record,get_firing)
157     #print(stats)
158 
159     plt.figure()
160     arr = np.zeros(n_steps)
161     arr2 = []
162     avg_res = []
163     for i in range(num_sim):
164         print(i)
165         res = run_sim(N, n_steps,prob,gamma,mu,ny,plot_res,record,get_firing)
166         avg_res.append(res)
167         #print(res)
168     for i in range(num_sim):
169         for s in range(n_steps):
170             arr[s] += avg_res[i][s][0]
171     #print(arr)
172     for i in (range(n_steps)):
173         arr2.append([arr[i]/num_sim/(N*N)*100,0])
174     #print(arr2)
175     #print(len(arr2))
176     plot_meshx(arr2,N,n_steps,gamma,mu,ny)
```

```
177     plt.savefig('Epidem.png')
178     plt.show()
179
180     # for i in range(100):
181     # if i == 0:
182     # res = run_sim(N, n_steps,plot_res,record,get_firing)
183
184     # else:
185     # temp = run_sim(N, n_steps,plot_res,record,get_firing)
186     # res = res + temp
187
188     # print(f"{i}")
189
190
191     # plt.figure()
192     # plt.plot(list(range(1001)), res / 100)
193     # plt.xlabel("Tick")
194     # plt.ylabel("Average number of firing neurons")
195     # plt.show()
```

Listing 3: The code for the 2D epidemic simulation