# MCS lab4

Urlich Icimpaye, Jakob Häggström, Oscar Jacobson

May 2022

## 1 Intro to the Viscek model

The Viseck model is a simulation model describing self propelled particles. The model is popular because of its simplicity while being able to describe global and real life phenomenons depending on chosen parameters. There are five basic parameters in the model.

1. N: number of particles
2. $\eta$: noise parameter
3. L: size of domain
4. R: radius of interaction
5. v: velocity

The model consists of moving particles bound to a set of rules which determines their future velocities. The first rule is called the *alignment rule* which in two dimensions looks like equation 1: [3]

$$\theta_i(t+1) = tan^{-1}\left(\frac{\sum_j sin(\theta_j(t))}{\sum_j cos(\theta_j(t))}\right) + e(t) \tag{1}$$

The formula calculates the average angle of neighbours to particle **i** (Chosen by the radius of interaction, parameter $R$) where *e(t)* is a random number with range $\eta * [-\pi/2, \pi/2]$ decided by chosen noise-parameter $\eta$.

When simulating the Viscek model the combined behaviour of all particles in the system is of great importance. One way to analyze the systemic behaviour of all particles using an alignment measure. The Alignment measure used in this rapport is equation 2: [3]

$$\Theta_i = \frac{1}{N}\sqrt{(\sum_j sin(\theta_j(t)))^2 + (\sum_j cos(\theta_j(t)))^2} \tag{2}$$

## 2 Polarisation of Vicsek model

To illustrate the effect of the noise parameter $\eta$ on the Viscek model, this section measures the mean polarisation of alignment on the model implemented using the script from Francesco Turci's website. [1] The model was ran using a domain length of 20 units and simulating 40 particles in the system with $R = 1.0$ and $v = 0.5$. The $\eta$ parameter was varied between 0 and 1 in 20 steps. For every iteration, 50 simulations, of length 300 timesteps, was ran for each $\eta$ in order to get average values. The results are shown in figure 1.

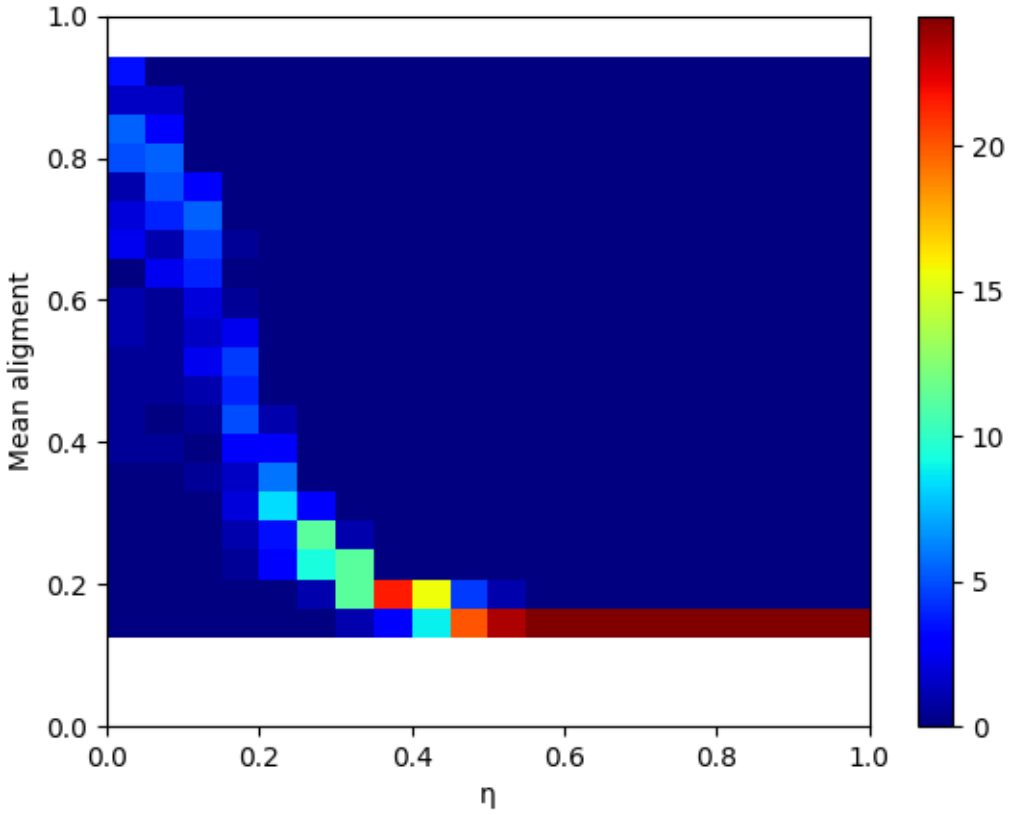


Figure 1: Phase diagram of alignment in the Viscek model with $\eta$ varying between 0 and 1.

As seen in figure 1, the mean alignment decreased as the noise parameter $\eta$ increased. Only reaching a maximum measure of alignment when $\eta$ is very low. When the noise parameter is large enough the graph flattens at the lowest possible measure, meaning there is no measurable alignment in the system. The noise quickly disorients the individuals making it impossible for the group to coordinate on a systemic level when $\eta$ is larger than approximately 0.4. Complete disorientation happens when eta is over 0.5 which coincides with when the random noise can start to cause particles to make more than $90^o$ turns. A turn that big will remove any previous alignment the particle had with others and the alignment-process will have to start over again.

## 3 Attraction and Aggregation

In this part the Viscek model was expanded upon to examine the effects of an added *attraction rule* in the model. We chose a simple rule where every particle gets attracted to the center of mass of all its neighbours with attraction strength decided by a gravitational constant $G$. The new and extended *update rule* in our Viscek model looks like equation 3:

$$\theta_i(t+1) = tan^{-1}\left(\frac{\sum_j sin(\theta_j(t))}{\sum_j cos(\theta_j(t))}\right) + G * tan^{-1}\left(\frac{\frac{\sum y_{j\in R}}{N_{j\in R}} - y_j}{\frac{\sum x_{j\in R}}{N_{j\in R}} - x_i}\right) + e(t) \tag{3}$$

The last part of the equation $e(t)$ is still a, uniformly random, added angle within the range $\eta * [-\pi, \pi]$. This model was simulated repeatedly for different combinations of gravitational strength $G$, noise $\eta$ and radius of influence $R$.

To examine the effects of this new rule a new performance measure was implemented as alignment doesn't prove anything of value when focusing on distances between particles. The new performance measure takes the average distances to the neighbours of every particle within their respective radius of influence and records the data in a phase diagram similar to the one in the previous section. This gives an overall view of how strongly the gravitational force attract the particles and how well groups are forming.

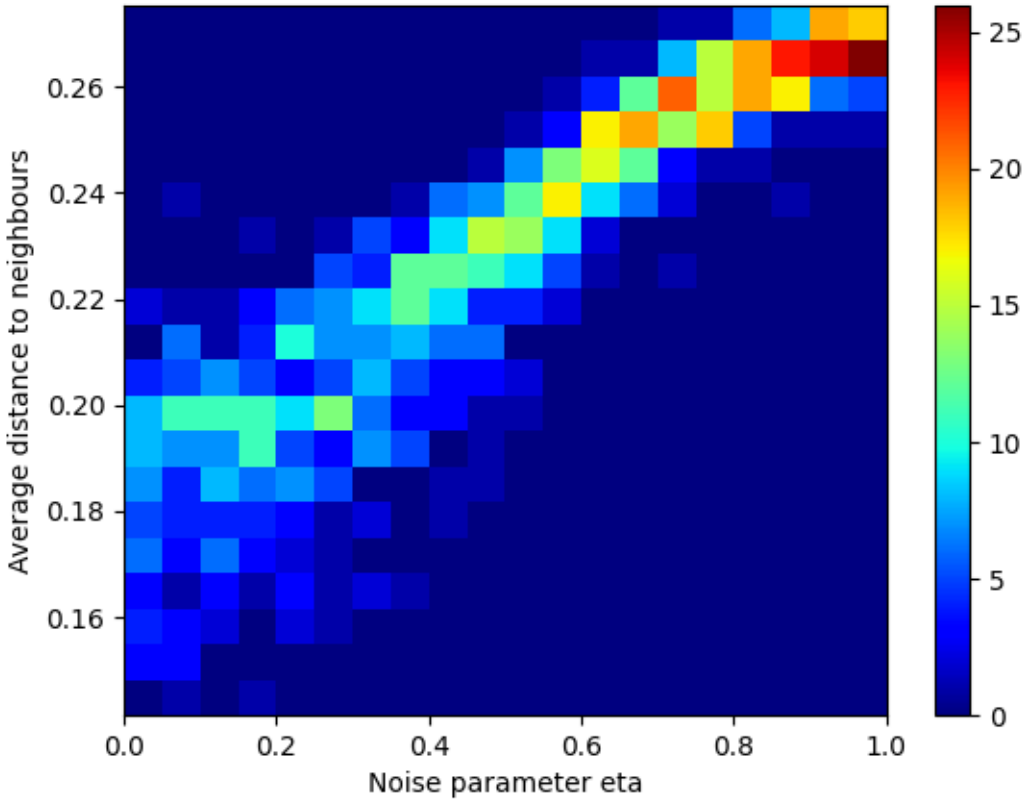


Figure 2: Phase diagram of distance to neighbors, varying $\eta$, with G = 1.0, r = 0,4.

In figure 2 the same setup as in the previous section was used on the new Viscek model. The noise parameter $\eta$ was varied from 0 to 1 in 20 steps for 50 simulations with 300 iterations each. The $G$ parameter was here set arbitrarily to 1.0 and the radius was 0.4. As expected the noise parameter causes the particles to be more spread apart during the simulation. The average distance to neighbors doesn't go all the way down to zero however. This is probably because of the fact that every particle is still trying to align itself with its neighbours which can be difficult in highly dense populations, especially if every particle is aligned differently to their collective center of mass (See Figure 4).

Interestingly a very similar graph to Figure 2 is produced when fixing $\eta$ and varying $G$. As seen in Figure 3 the patterns looks mirrored to Figure 2. This is to be expected though as many of the arguments for the former can also be made in reverse for the latter. A high noise parameter causes the particles to scatter whereas a high attraction parameter causes the particles to converge. There seems to be a lowest threshold here as well where the $\eta$ chaos or innate alignment causes the particles to spread out more even though the gravity constant is increasing.

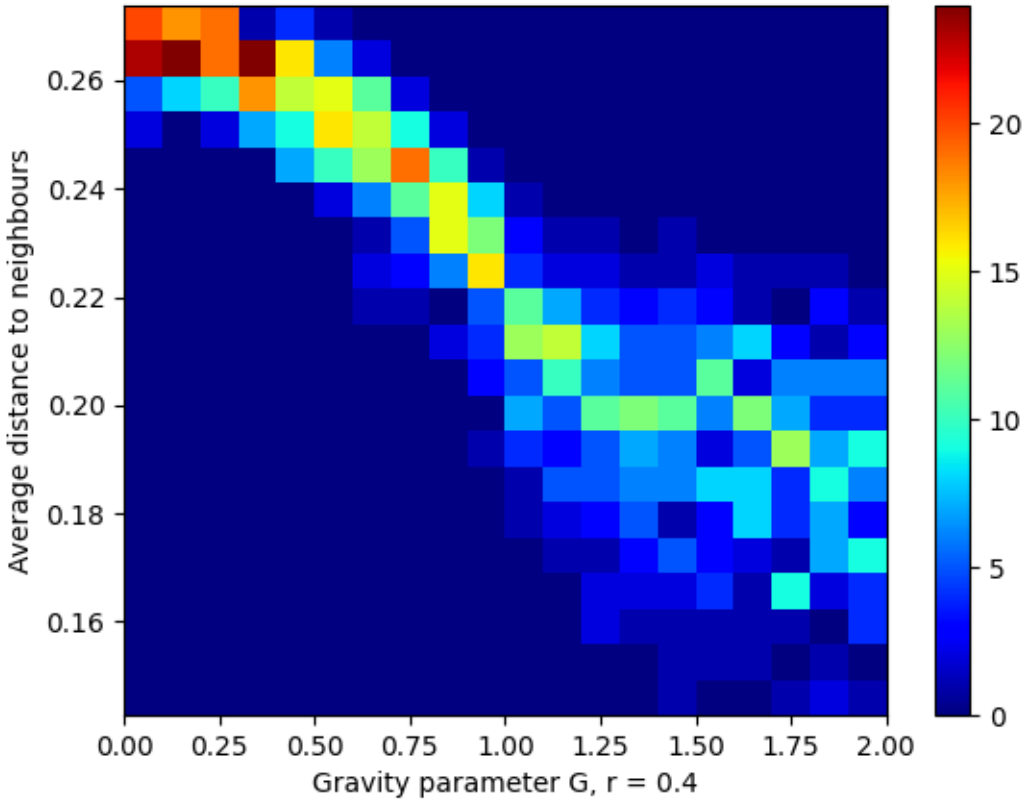


Figure 3: Phase diagram of distance to neighbors, varying $G$, with $\eta$ = 0,4, r = 0,4.

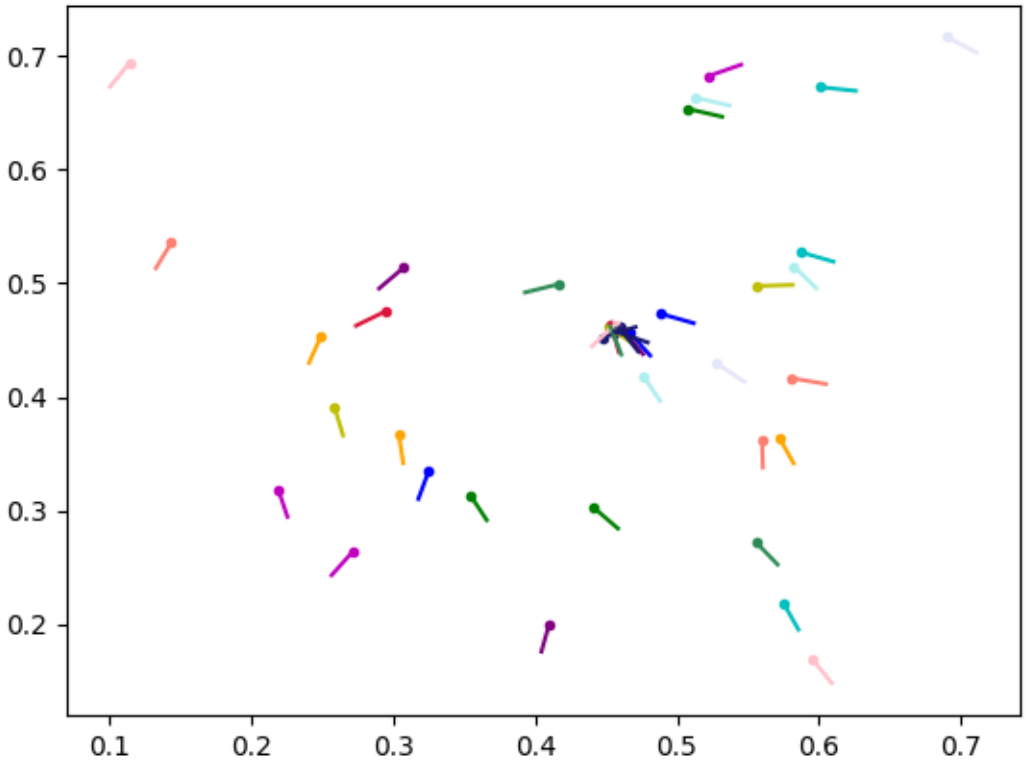


Figure 4: Example of gravity pulling particles together when G is 1,0.

To finalise the analysis an examination of the effect of the radius of influence on the model was plotted in Figure 5. The performance measure had to be changed as the measure itself was dependent on the radius that was varied. Therefore the average distance to all particles was considered in this part. Worth noting is that this will skew the results to show the entire system as a single large group. Small groups that consists of only a few individuals will therefore not score as well as a single large group of particles even though the individuals might be close to each other group wise.

As seen in Figure 5 the radius of influence has some influence over how the system functions. When the radius is very low the particles have a hard time finding each other and will therefore not be able to form a large group. When $R$ reaches a threshold of about 0.4 the behaviour seems to stop changing and is decided by the other unchanging parameters $G$ and $\eta$. A radius of 0.4 coincides with a diameter of almost the entire system if viewed from the centre which makes sense. When one particle can influence almost the entire system the other variables of the system will be the deciding factors in its behaviour.

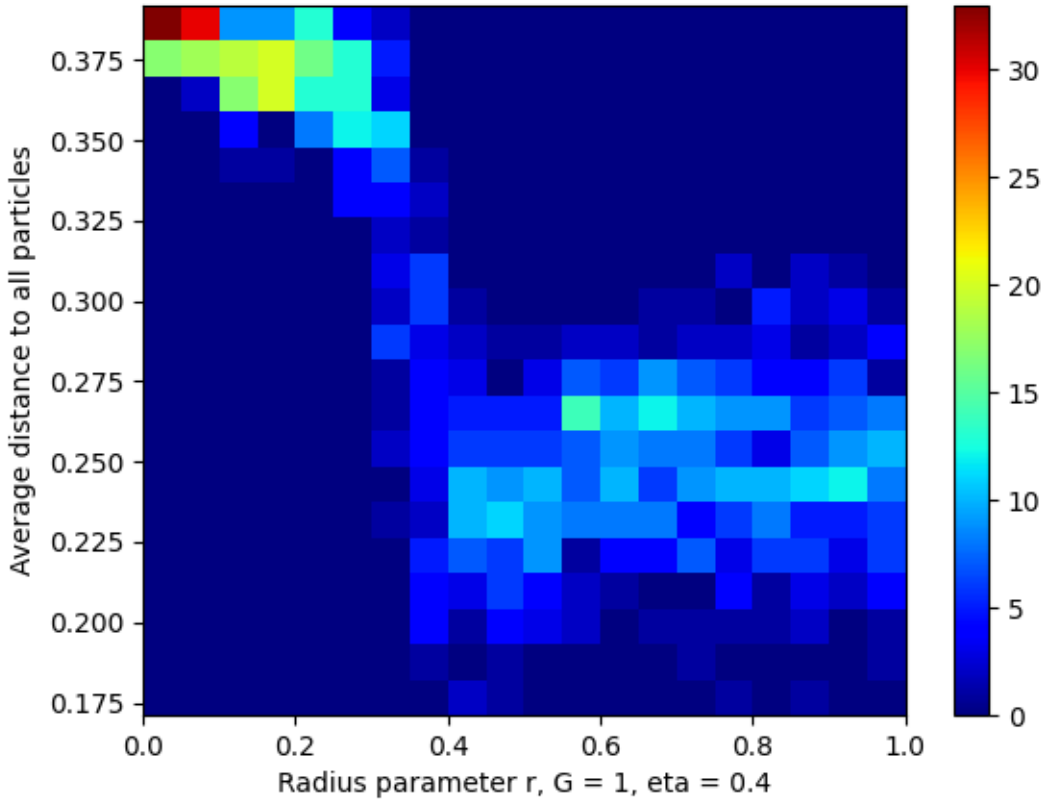


Figure 5: Phase diagram of average distance to all particles, varying $r$, with $\eta = 0{,}4$, G = 1,0.

## References

[1] Minimal Viscek model in python. Available from: https://francescoturci.net/2020/06/19/minimal-vicsek-model-in-python/

[2] vicsek_model.py. Available from: https://github.com/fskerman/vicsek_model/blob/master/vicsek.py

[3] Lecture: Self-propelled particles - Fiona Skerman, Uppsala University. Available from: https://fskerman.github.io/2022:L5andL6.pdf

## Appendix

```
import numpy as np
import scipy as sp
from scipy import sparse
from scipy.spatial import cKDTree
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from math import sqrt
```

```
8
9
10 L = 20.0
11 N = 40
12 print("␣N",N)
13
14 r0 = 1.0
15 deltat = 1.0
16 factor =0.5
17 v0 = r0/deltat*factor
18 iterations = 300
19
20
21 eta_vec = np.linspace(0,1,20)
22
23 result = []
24
25 totiter = len(eta_vec) * iterations * 50
26 iteri = 0
27
28 for eta in eta_vec:
29
30    pos = np.random.uniform(0,L,size=(N,2))
31    orient = np.random.uniform(-np.pi, np.pi,size=N)
32
33
34    for iter in range(50):
35       alignment = []
36       pos = np.random.uniform(0,L,size=(N,2))
37       orient = np.random.uniform(-np.pi, np.pi,size=N)
38       for i in range(iterations):
39
40          tree = cKDTree(pos,boxsize=[L,L])
41          dist = tree.sparse_distance_matrix(tree, /
                max_distance=r0,output_type='coo_matrix')
42
43          #important 3 lines: we evaluate a quantity for every column j
44          data = np.exp(orient[dist.col]*1j)
45          # construct a new sparse marix with entries in the same places ij of the /
                dist matrix
46          neigh = sparse.coo_matrix((data,(dist.row,dist.col)), /
                shape=dist.get_shape())
47          # and sum along the columns (sum over j)
48          S = np.squeeze(np.asarray(neigh.tocsr().sum(axis=1)))
49
50
51          orient = np.angle(S)+eta*np.random.uniform(-np.pi, np.pi, size=N)
52
53
54          cos, sin= np.cos(orient), np.sin(orient)
55          pos[:,0] += cos*v0
56          pos[:,1] += sin*v0
57
58          pos[pos>L] -= L
59          pos[pos<0] += L
60
61          temp_align = 1/N * sqrt(sum(cos)**2 + sum(sin)**2)
62
63          iteri+=1
64          alignment.append(temp_align)
```

```
65
66        print(f"{100*round(iteri/totiter,3)} done.")
67        result.append([eta, np.mean(alignment)])
68
69 result = np.array(result)
70
71 plt.hist2d(result[:,0], result[:,1], bins=(20,20),density = True, cmap=plt.cm.jet)
72 plt.ylim([0,1])
73
74 plt.ylabel("Mean aligment")
75 plt.xlabel("")
76 plt.colorbar()
77 plt.show()
78 plt.savefig('Question1')
```

Listing 1: The code for the first section.

```
1  import os, sys
2  from celluloid import Camera
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  from math import pi, sqrt, cos, sin, atan2
7
8  #for videos
9
10
11
12
13
14 #---------- Geometric functions ----------------------------
15
16 def vector_2_angle(v):
17     x = v[0]
18     y = v[1]
19     return atan2(y,x)
20
21
22 # generate random angle theta between -pi - pi
23 def rand_angle():
24     theta = np.random.uniform(-pi,pi)
25     return theta
26
27
28 # returns angle unit vector
29 def angle_2_vector(theta):
30     x = cos(theta)
31     y = sin(theta)
32
33     # transform to unit vector
34     v1 = np.array([x,y])
35     #v2 = np.array([0,0])
36     #uv = unit_vector(v1,v2)
37
38     uv = v1/ euclidean_distance(v1[0], v1[1], 0, 0)
39
40     return uv
41
42
43 # Euclidean distance between (x,y) coordinates
44 def euclidean_distance(x1, y1, x2, y2):
```

```
45    return sqrt((x1 - x2)**2 + (y1 - y2)**2)
46
47
48 # Euclidean distance between (x,y) coordinates on 1 x 1 torus
49 def torus_distance(x1, y1, x2, y2):
50    x_diff = min(abs(x1 - x2), 1 - abs(x1 - x2))
51    y_diff = min(abs(y1 - y2), 1 - abs(y1 - y2))
52    return sqrt(x_diff**2 + y_diff**2)
53
54
55 def unit_vector(v1, v2):
56    vector = v1 - v2
57    dist = euclidean_distance(v1[0], v1[1], v2[0],v2[1])
58    v1v2 = vector / dist
59    return v1v2
60
61
62 #-------------------------------------------------------------------------------------------
63
64 #-------------------------Functions of Neighbours /
       ---------------------------------------
65
66
67 # returns a list of indices for all neighbors
68 # includes itself as a neighor so it will be included in average
69 def get_neighbors(particles, r, x0, y0, selfind):
70
71    neighbors = []
72
73    for j,(x1,y1) in enumerate(particles):
74       dist = torus_distance(x0, y0, x1, y1)
75
76       if dist < r and j != selfind:
77          neighbors.append(j)
78
79    return neighbors
80
81
82 # average unit vectors for all angles
83 # return average angle by converting to vectors, using vector addition /
       top-to-tail, then taking arc tan to get angle of resulting vector.
84 def get_average(thetas, neighbors):
85
86    n_neighbors = len(neighbors)
87    avg_vector = np.zeros(2)
88
89    for index in neighbors:
90       theta = thetas[index,0]
91       theta_vec = angle_2_vector(theta)
92       avg_vector += theta_vec
93
94    avg_angle = vector_2_angle(avg_vector)
95    avg_angle = angle_2_vector(avg_angle)
96
97    return avg_angle
98
99
100 def get_avg_dist(neighbors, particles, x, y):
101
102    num_neighbours = len(neighbors)
```

```
103
104     if num_neighbours == 0:
105         return 0
106     avg_dist = 0
107
108     for ind in neighbors:
109         avg_dist += torus_distance(x, y, particles[ind,0], particles[ind,1])
110
111
112     return avg_dist / num_neighbours
113
114 def get_max_dist(neighbors, particles, x, y):
115
116     maxdist = 1E11
117     index = 1
118
119
120     for ind in neighbors:
121         dist = torus_distance(x, y, particles[ind,0], particles[ind,1])
122
123         if dist > maxdist:
124             maxdist = dist
125             index = ind
126
127     return max,index
128
129 def newton(G, particles, x, y, ind):
130
131     temp_part = particles.copy()
132     temp_part = np.delete(temp_part, [[ind, 0]], axis = 0)
133     pos = np.array([[x, y]])
134
135     epsilon = 1E-1
136
137     r_vec = np.zeros(temp_part.shape)
138     r_vec[:,0] = x - temp_part[:,0]
139     r_vec[:,1] = y - temp_part[:,1]
140     r_abs = np.sqrt(r_vec[:,0] ** 2 + r_vec[:,1] ** 2)
141
142     forces = np.zeros(temp_part.shape)
143     forces[:,0] = - G * r_vec[:,0] / (r_abs + epsilon) ** 3
144     forces[:,1] = - G * r_vec[:,1] / (r_abs + epsilon) ** 3
145
146     return np.sum(forces, axis = 0)
147
148
149 def center_of_mass(neighbors, particles, x, y):
150     x_temp = x
151     y_temp = y
152     num_neighbours = len(neighbors)
153
154     for ind in neighbors:
155         x_temp += particles[ind,0]
156         y_temp += particles[ind,1]
157
158     x_avg = x_temp/num_neighbours - x
159     y_avg = y_temp/num_neighbours - y
160     v = [x_avg, y_avg]
161     v = vector_2_angle(v)
162     v = angle_2_vector(v)
```

```
163     return v
164
165 #--------------------------------------------------------------------------
166
167
168 def plot_vectors(coords, thetas):
169
170   # generate random color for every particle
171   colors = ["b", "g", "y", "m", "c", "pink", "purple", "seagreen",
172       "salmon", "orange", "paleturquoise", "midnightblue",
173       "crimson", "lavender"]
174
175
176   for i, (x, y) in enumerate(coords):
177
178     c = colors[i % len(colors)]
179
180     # plot point
181     plt.scatter(x, y, color = c, marker = ".")
182
183     # plot tail
184     theta = thetas[i]
185     v = angle_2_vector(theta)
186     x1 = x - (0.025 * v[0])
187     y1 = y - (0.025 * v[1])
188     plt.plot([x, x1], [y, y1], color=c)
189
190
191
192   return
193
194
195
196 def save_plot(path, fname, eta):
197
198     # axes between 0 and 1
199     plt.axis([0, 1, 0, 1])
200
201     # remove tick marks
202     frame = plt.gca()
203     frame.axes.get_xaxis().set_ticks([])
204     frame.axes.get_yaxis().set_ticks([])
205
206     # title
207     plt.title("␣=␣%.2f" % eta)
208
209     # save plot
210     plt.savefig(os.path.join(path, fname[:-4]+".jpg"))
211     plt.close()
212
213     # clear for next plot
214     plt.cla()
215
216     return
217
218     # ------------------------------ RUNS FROM HERE ------------------------------
219
220 if __name__ == '__main__':
221
222
```

```
223
224 N =40 # num of particles
225 eta = 0.2 # noise in [0,1], add noise uniform in [-eta*pi, eta*pi]
226 r = 0.4 # radius
227 delta_t = 0.01 # time step
228
229 # Maximum time
230 t = 0.0
231 T = 0.2 #was 2.0
232 G = 1
233 #G_vec = np.linspace(2.5, 0.025, 10)
234 #G_vec = np.linspace(0, 200, 10)
235 #G_vec = np.linspace(1, 0.01, 10) [0.1]
236 #G_vec = np.linspace(0.1, 1, 10)
237
238 r_vec = np.linspace(0, 1, 20)
239 G_vec = np.linspace(0, 2, 20)
240 eta_vec = np.linspace(0, 1, 20)
241 iterations = 40
242 iteri = 0
243 # Generate random particle coordinates
244 # particles[i,0] = x
245 # particles[i,1] = y
246
247
248
249 totiter = totiter = len(eta_vec) * iterations * 50
250 tot_res = []
251
252
253 for r in r_vec:
254 #for G in G_vec:
255 #for eta in eta_vec:
256
257    for n in range(50):
258       particles = np.random.uniform(0, 1, size=(N, 2))
259
260       # initialize random angles
261       thetas = np.zeros((N, 1))
262       for i, theta in enumerate(thetas):
263          thetas[i, 0] = rand_angle()
264       result = []
265       for iter in range(iterations):
266
267          tempres = []
268          for i, (x, y) in enumerate(particles):
269             # get neighbor indices for current particle
270             #neighbors2 = get_neighbors(particles, 2, x, y, i)
271             neighbors = get_neighbors(particles, r, x, y, i)
272
273             avg = get_average(thetas, neighbors)
274             avgdist = get_avg_dist(neighbors, particles, x, y)
275             tempres.append(avgdist)
276
277             #force = newton(G, particles, x, y, i)
278
279             attraction = center_of_mass(neighbors, particles, x, y)
280             avg[0] = avg[0] + G*attraction[0]
281             avg[1] = avg[1] + G*attraction[1]
282             avg = vector_2_angle(avg)
```

```
283
284                 # get noise angle
285                 n_angle = rand_angle()
286
287                 noise = eta * n_angle
288
289                 # get new theta
290                 thetas[i] = avg + noise
291
292                 # move to new position
293                 particles[i,:] += delta_t * angle_2_vector(thetas[i])
294
295                 # assure correct boundaries (xmax, ymax) = (1,1)
296                 if particles[i, 0] < 0:
297                    particles[i, 0] = 1 + particles[i, 0]
298
299                 if particles[i, 0] > 1:
300                    particles[i, 0] = particles[i, 0] - 1
301
302                 if particles[i, 1] < 0:
303                    particles[i, 1] = 1 + particles[i, 1]
304
305                 if particles[i, 1] > 1:
306                    particles[i, 1] = particles[i, 1] - 1
307
308             iteri += 1
309             # plt.figure()
310             # plot_vectors(particles, thetas)
311             # plt.show()
312             result.append(np.mean(tempres))
313             # new time step
314             #t += delta_t
315         tot_res.append([r, result[-1]])
316         print(f"{100*round(iteri/totiter,3)} done.")
317
318
319 plt.figure()
320 plot_vectors(particles, thetas)
321 plt.show()
322 tot_res = np.array(tot_res)
323 plt.hist2d(tot_res[:,0], tot_res[:,1], bins=(20,20), cmap=plt.cm.jet)
324
325 plt.ylabel("Average distance to all particles")
326 plt.xlabel("Radius parameter r, G = 1, eta = 0.4")
327 plt.colorbar()
328 plt.savefig('Question2')
329 plt.show()
```

Listing 2: The code for the second section.