```diff
diff --git a/ca1_step1_solution.py b/ca1_step1_solution.py
index 17d9dc2..e511adf 100644
--- a/ca1_step1_solution.py
+++ b/ca1_step1_solution.py
@@ -1,131 +1,151 @@
+
+# Please put everything in the docstring also the comments
+
 #%%
 # ========================
 #        CA 1: STEP 1
 # ========================
 ''' Group 26:
     Anton Gusarov
     Martin Gustavsson
 '''
 import re
 import numpy as np
 import h5py
 import matplotlib
 from matplotlib import pyplot as plt
 from itertools import islice
 
 #%% Simply reading the first 2 lines:
 with open('ca1_step1_input_data.txt', 'r', errors='strict') as file: # opener is a custom callable
     lines = list()
     for i in range(2):
         lines = file.readline()
 
 # %% Parse the main header of the file (first 2 lines):
 with open('ca1_step1_input_data.txt', 'r', errors='strict') as file:
     # read the header only i.e. the first 2 lines
     # reg. expr. used where assumed that ',' or ';' are the separators:
     lines = [re.split(r'[,;]', line) for line in islice(file.readlines(), 2)]
 
 header_keys = tuple(i.strip() for i in lines[0]) # remove trailing spaces by .strip()
 header_keys = tuple(re.sub(r'[\s#)]','', i) for i in header_keys)
 header_keys = tuple(re.sub(r'[(]','_', i) for i in header_keys)
 
 # %% remove all space characters from header line 2 and tx to float:
 header_values = [float(i.strip()) for i in lines[1]]
 header_values[0] = int(header_values[0])  # time_steps to be int
 header_values[4] = int(header_values[4])  # N_particles to be int
 
 # %% Create a dict {header_key -> header_value}:
 header = {}
+
+# Please use pythonic looping with zip, i.e.
+# for key, value in zip(header_keys, header_values):
+
 for i in range(len(header_keys)):
     header[header_keys[i]] = header_values[i] # build dict incrementally
 
 locals().update(header)  # unzip the dict to a set of variables
 
 # %% You need to identify the start of each time step by looking for the
 # time-step header # time_step. And then parse the N_particles rows
 # of position and velocity data in each time-step block
 
 # First task: print headers for time_step:
 with open('ca1_step1_input_data.txt', 'r', errors='strict') as file:
     for line in islice(file.readlines(), 4):
         if '# time_step ' in line: print(line)
 
 # %%
 # Parse the data in each time-step block.
 # Store the data in a (three level) nested list called data.
 # 1. The first level should be the time step index,
 # 2. the second level the particle index,
 # 3. third level a list of the four values of the x, y position-
 #    and v_x, v_y velocitiy-components stored as floating point numbers (float).
 
 inner_list_particle = []
 data = []
 particle_count = 0
 
 # The parser:
 with open('ca1_step1_input_data.txt', 'r', errors='strict') as file:
     for line in islice(file.readlines(), 3, None, None):  # start from trird line - skip the header
         if (not line) or line.isspace() : # skip empty line: no symbols or only spaces
             #print('empty string')
             continue
 
         if '# time_step ' in line:
             time_step_index = int(''.join(re.findall(r'\d',line))) # parsed time-step index
             #print('time step:', time_step_index, '\n')
 
         if re.search(r'# ', line) is None: # find a line without '#' symbol i.e. with position and speeds numbers
             inner_list_particle.append([float(i.strip()) for i in re.split(r'[,;]', line)])
             particle_count += 1
```

```
+
+            # Please do not hard code the number of particles
+
             if particle_count == 400:
+
+                # This works, but is extra work for the computer
+                # Please do not copy the data and then remove the original
+
+                # Just pass the data along by appending it
+                # and set inner_list_particle to a new empty list []
+
                 data.append(inner_list_particle.copy())
                 inner_list_particle.clear()
                 particle_count = 0

 print('data list size:', len(data), len(data[0]), len(data[0][0])) # check data list dimensions

 # %% Convert a nested list to a multidimensional (NumPy) array
 data = np.array(data)
 print(type(data))
 print(data.dtype)
 print(data.shape)

 # %% Split position vectors and velocity vectors
 # To make the numerical analysis easier you want to store the final result in two separate ndarrays,
 # one for the positions called R and one for the velocities called V
 R = data[:,:,:2]
 V = data[:,:,2:]

 print(R.shape)
 print(V.shape)

 # %% To swap the order of the axes use the NumPy "transpose-like" manipulation routines:
 R = np.swapaxes(R,1,2)
 V = np.swapaxes(V,1,2)

 print(R.shape)
 print(V.shape)

 # %% Plot the first time step with Matplotlib
 fig, axs = plt.subplots()
 axs.set_aspect(aspect=1)
 axs.scatter(R[0, 0, :], R[0, 1, :],
             c='red', s=50.0, alpha=0.4, edgecolors='none')
 plt.xlim(-1, 1)
 plt.ylim(-1, 1)
 plt.show(fig)

 # %% Storing data using the hdf5 file format
 # use its h5py.File class together with the 'with'-statement to open the file ca1_step1_output_data.h5 file.
 # Use '.create_dataset' method to store the ndarrays R and V,
 # and store the other parameters as attributes using the .attrs.create method.

 with h5py.File('ca1_step1_output_data.h5', 'w') as hfile:
     dset1 = hfile.create_dataset('R', data=R)
     dset1 = hfile.create_dataset('V', data=V)

+    # Please store the attributes in the root of the hdf5 file not in dset1
+    # using the function pointed out in the description
+
     dset1.attrs['N_particles'] = N_particles
     dset1.attrs['time_steps'] = time_steps
     dset1.attrs['time_step_s'] = time_step_s
     dset1.attrs['radius'] = radius
     dset1.attrs['v_variance'] = v_variance

diff --git a/ca1_step3_solution.py b/ca1_step3_solution.py
index a8bfdff..502ecaa 100644
--- a/ca1_step3_solution.py
+++ b/ca1_step3_solution.py
@@ -1,216 +1,235 @@
+
+# use docstring here not comments
+
 #%%
 # ========================
 #        CA 1: STEP 3
 # ========================
 import os
 import itertools
 import h5py
 from scipy import stats
 from scipy.spatial import distance
 import numpy as np
 from numpy import random
 from matplotlib import pyplot as plt
 import matplotlib.animation as animation

 from scipy.spatial import distance
 from scipy.spatial import cKDTree
```

```
 from ca1_step_2_solution import animate_particles
 from ca1_step_2_solution import plot_statistical_analysis
 
+# Please move all functions here and put the other code after them
+
 #%%
 # 1. Particle system initialization
 #-----------------------------------
 N_particles = 400
 sigma_v = 0.1
 r0 = 2 * random.rand(2, N_particles) - 1
 v0 = sigma_v * random.randn(2, N_particles)
 
 print(r0.shape, v0.shape) # shape check
 
 # sanity check plot of initial positions:
 x0, y0 = r0[0, :], r0[1, :]
 fig, axs = plt.subplots()
 axs.set_aspect(aspect=1)
 axs.scatter(x0, y0, color='green', s=50.0, alpha=0.4, edgecolors='none')
 plt.xlim(-1.1, 1.1)
 plt.ylim(-1.1, 1.1)
 plt.show(fig)
 
 # %%
 # 2. Time propagation
 #-----------------------------------
 time_step = 0.02  # 50 steps/sec
 
 def time_propagation(r, v, time_step):
     r = r + v * time_step  # broadcasting was used to implement this in a single line
     return r, v
 
 
 r, v = time_propagation(r0, v0, time_step) # call the function
 
 # sanity check plot of time propagation:
 x, y = r[0, :], r[1, :]
 fig, axs = plt.subplots()
 axs.set_aspect(aspect=1)
 axs.scatter(x, y, color='blue', s=50.0, alpha=0.4, edgecolors='none')
 plt.xlim(-1.1, 1.1)
 plt.ylim(-1.1, 1.1)
 plt.show(fig)
 
 # %%
 # 3. Simulation function
 #-----------------------------------
 # The function uses the time_propagation function to compute
 # the time evolution of the particles for a given number of 'time_steps'
 # and returns the positions R and velocities V of all particles for all time steps.
 
 def simulate(r0, v0, time_step, time_steps, update_function, **kwargs):
     # Allocate R and V:
     R = np.empty((time_steps, r0.shape[0], r0.shape[1]))
     V = np.empty((time_steps, v0.shape[0], v0.shape[1]))
 
     # Initialize R[0] and V[0] using r0 and v0:
     R[0,:,:] = r0
     V[0,:,:] = v0
 
     # Loop over all time steps and call:
     for t in range(1, time_steps):
         R[t,:,:], V[t,:,:] = update_function(R[t-1,:,:], V[t-1,:,:], time_step, **kwargs)
     return R, V
 
 
 simulation1 = simulate(
     r0, v0, time_step, time_steps=400,
     update_function=time_propagation)
 
 #%%
 # call the animation:
 animate_particles(simulation1[0], time_step, filename='ca1_step3_movie1.mp4')
 
 # %%
 # 4. Hard wall boundary collisions
 #-----------------------------------
 # -- check if the position coordinates (x and  y) are out of bounds and
 # -- the particle at the same time has a velocity component going in the wrong direction (relative to the "wall").
 # -- In this case change the sign of the corresponding velocity component (v_x or v_y)
 
 # Use numpy boolean arrays for indexing so that each line of code treats all particles at the same time.
 # Modifiy the velocity vector v in-place so that the function does not have to return anything.
 
 def boundary_collisions(r, v):
     ''' Modifies the velocity vector according to the hard wall boundary conditions.
     '''
+
```

```
+    # Please add the check of the velocity components
+    # Without it particles tend to get stuck at the boundaries and even go outside it
+
     boundary = np.full(r.shape, 1)  # defines the abs value of coordinates |x| or |y|
     boundary_check = np.greater(np.absolute(r), boundary) # returns True if any module of coordinate > 1
     boundary_check = -2 * boundary_check.astype(int) + 1 # True/False to -1/1
     v = np.multiply(boundary_check, v)  # changes the sign of v components for where boundary_check is True
     return r, v


 def update_with_boundaries(r, v, time_step):
     ''' Time propagation with boundary conditions.
     '''
     r, v = time_propagation(r, v, time_step)
     r, v = boundary_collisions(r, v)
     return r, v


 simulation2 = simulate(
     r0, v0, time_step, time_steps=400,
     update_function=update_with_boundaries)

 animate_particles(simulation2[0], time_step, 'ca1_step3_movie2.mp4')

 # %%
 # 5. Particle-prticle collision
 #-----------------------------------
 # -- loops over all pairs of particles,
 # -- check if they are closer to each other than 2*radius and are travelling towards each other.
 # -- If yes then modifiy their velocity vectors according to a momentum conserving collision.

 radius = 0.05
 def particle_collisions(r, v, radius):
     dist = distance.pdist(np.swapaxes(r, 0,1), metric='euclidean')  # returns condensed vector of distances
     dist = distance.squareform(dist) # find pairs closer to each other than 2*radius

     check_dist = np.less(dist, 2*radius)
     np.fill_diagonal(check_dist, False) # fill diagonal
     check_dist = np.triu(check_dist) # remove duplicates due to symmetry
     indx = np.asarray(np.where(check_dist==True))  # return indices of pairs where distance is smaller than 2*r

     indx = np.split(indx, indx.shape[1], axis=1)
     for i in indx: # i - pair of indexes of the colliding particles
+
+        # Please add the check on the velocities that ensures that the
+        # particles are travelling towards each other.
+
+        # Please work with vector expressions instead of treating the x and y
+        # components separately
+
         # v_i = v[:, i[0]] # i-particle pair of v components (x, y)
         # v_j = v[:, i[1]] # j-particle pair of v components (x, y)

         v_i_x = v[:, i[0]][0] - (v[:, i[0]][0] - v[:, i[1]][0])
         v_i_y = v[:, i[0]][1] - (v[:, i[0]][1] - v[:, i[1]][1])

         v_j_x = v[:, i[1]][0] + (v[:, i[0]][0] - v[:, i[1]][0])
         v_j_y = v[:, i[1]][1] + (v[:, i[0]][1] - v[:, i[1]][1])


         v[:, i[0]] = [v_i_x, v_i_y]
         v[:, i[1]] = [v_j_x, v_j_y]
     return r, v


 def update_with_interactions_slow(r, v, time_step, radius):
     r, v = time_propagation(r, v, time_step)
     r, v = boundary_collisions(r, v)
     r, v = particle_collisions(r, v, radius)
     return r, v


 simulation3 = simulate(
     r0, v0, time_step, time_steps=100,
     update_function=update_with_interactions_slow, radius=radius)

 animate_particles(simulation3[0], time_step, 'ca1_step3_movie3.mp4')


 # %%
 # 6. Fast hard spheres
 #-----------------------------------
 # Use the KDTree class in scipy.spatial to rapidly find all pairs of particles
 # closer than two times the radius to each other.

 def particle_collisions_fast(r, v, radius):

     tree = cKDTree(np.swapaxes(r, 0,1))
     dist_set = tree.query_pairs(2*radius) # method; returns Python SET of pairs of points whose distance is at most r
```

```
     for i in list(dist_set): # i - pair of indexes of the colliding particles
+
+        # Please consider all the comments in the function `particle_collisions` above
+
         # v_i = v[:, i[0]] # i-particle pair of v components (x, y)
         # v_j = v[:, i[1]] # j-particle pair of v components (x, y)

         v_i_x = v[:, i[0]][0] - (v[:, i[0]][0] - v[:, i[1]][0])
         v_i_y = v[:, i[0]][1] - (v[:, i[0]][1] - v[:, i[1]][1])

         v_j_x = v[:, i[1]][0] + (v[:, i[0]][0] - v[:, i[1]][0])
         v_j_y = v[:, i[1]][1] + (v[:, i[0]][1] - v[:, i[1]][1])

         v[:, i[0]] = [v_i_x, v_i_y]
         v[:, i[1]] = [v_j_x, v_j_y]
     return r, v


 def update_with_interactions_fast(r, v, time_step, radius):
     r, v = time_propagation(r, v, time_step)
     r, v = boundary_collisions(r, v)
     r, v = particle_collisions_fast(r, v, radius)
     return r, v


 simulation4 = simulate(
     r0, v0, time_step, time_steps=1000,
     update_function=update_with_interactions_fast, radius=radius)

 animate_particles(simulation4[0], time_step, 'ca1_step3_movie4.mp4')

 # %%
 # 7. Statistical analysis of interacting particles
 #--------------------------------------------------
 plot_statistical_analysis(simulation4[0], simulation4[1], time_step, 'ca1_step3_figure_summary.svg')

diff --git a/ca1_step_2_solution.py b/ca1_step_2_solution.py
index 446ebe1..ebfaadd 100644
--- a/ca1_step_2_solution.py
+++ b/ca1_step_2_solution.py
@@ -1,219 +1,246 @@
+
+# Please use a docstring here not comments
+
 # %%
 # ========================
 #       CA 1: STEP 2
 # ========================
 import os
 import itertools
 import h5py
 from scipy import stats
 from scipy.spatial import distance
 import numpy as np
 from matplotlib import pyplot as plt
 import matplotlib.animation as animation

 def plot_statistical_analysis(R, V, time_step, filename):
     """
     Analyzes R and V and plots statistical analysis results
     """
     time = np.arange(0, R.shape[0]*time_step, time_step) # [sec]

+    # General comment on reshaping of arrays
+
+    # Please do not hard code the dimensions in the reshaping of arrays
+    # This makes your function only work for 1000 timesteps with 400 particles
+
+    # Instead of doing reshaping with given shapes please use the ndarray method .flatten()
+
+    # It is also convenient to refrain from defining flattened variables and instead
+    # flatten the array when calling .fit and .hist, i.e.
+    # plt.hist(R[:, 0, :].flatten(), ...)
+
     # Prepare data for x,y velocities and coordinates plots:
     #-------------------------------------------------------------
     # 1. x and y positions:
     R_reshaped_x = np.reshape(R[:,0,:], (400*1000,))
     R_reshaped_y = np.reshape(R[:,1,:], (400*1000,))
     V_reshaped_x = np.reshape(V[:,0,:], (400*1000,))
     V_reshaped_y = np.reshape(V[:,1,:], (400*1000,))

     v_x_loc, v_x_scale = stats.norm.fit(V_reshaped_x)  # fit model to the data
     v_x_pdf = stats.norm.pdf(np.arange(-0.5, 0.5, 0.01), loc=v_x_loc, scale=v_x_scale)  # modelled distribution

     v_y_loc, v_y_scale = stats.norm.fit(V_reshaped_y)
     v_y_pdf = stats.norm.pdf(np.arange(-0.5, 0.5, 0.01), loc=v_y_loc, scale=v_y_scale)

     x_loc, x_scale = stats.uniform.fit(R_reshaped_x)
     x_pdf = stats.uniform.pdf(np.arange(-1.5, 1.5, 0.01), loc=x_loc, scale=x_scale)
```

```
     y_loc, y_scale = stats.uniform.fit(R_reshaped_y)
     y_pdf = stats.uniform.pdf(np.arange(-1.5, 1.5, 0.01), loc=y_loc, scale=y_scale)

     plt.subplots_adjust(left=None, bottom=None, right=None, top=None,
                         wspace=0.5, hspace=0.7) # Tune the subplot layout

     # Prepare data for pair distance distribution function:
     #---------------------------------------------------------------
+
+    # Please use NumPy broadcasting to implement thie pair distances
+    # The goal is use the basic mechanism of broadcasting (not scipy.spatial.distance.pdist)
+
     R_split_time = np.split(R, R.shape[0], axis=0)
     def calculate_distances(R_split_time):
         dist = distance.pdist(np.swapaxes(np.squeeze(R_split_time), 0, 1),
                               metric='euclidean')  # returns condensed vector of distances
         return dist
     Dist = list(map(calculate_distances, R_split_time))  # produces the list of condensed distances
     Dist = np.asarray(Dist)  # (1000, 79800) ndarray

     # Prepare data for velocity norm pdf:
     #---------------------------------------------------------------
     V_norm = np.linalg.norm(V, ord=2, axis=1)
     V_reshaped = np.reshape(V_norm, (400*1000,))
     v_loc, v_scale = stats.rayleigh.fit(V_reshaped)
     v_pdf = stats.rayleigh.pdf(np.arange(0, 0.5, 0.01), loc=v_loc, scale=v_scale)

     # Prepare data for E_k and V_tot:
     #---------------------------------------------------------------
     V_sum = np.sum(V, 2)
     V_tot = np.linalg.norm(V_sum, ord=2, axis=1)

+    # This is the second time the function computes V_norm
+    # Please remove one of the calculations
+
     V_norm = np.linalg.norm(V, ord=2, axis=1)
-    V_norm.shape
+
+    # This row of code is not doing anything please clean up
+    V_norm.shape
+
     E_k = np.sum(np.square(V_norm)/2, 1)

     # PLOTS:
     #=======
     plt.figure(1, figsize=(20, 30))  # â\200\231figsize â\200\231 to make the figure larger
     plt.subplots_adjust(left=None, bottom=None, right=None, top=None,
                         wspace=0.5, hspace=1.5) # Tune the subplot layout
     # V_tot plot:
     #---------------------------------------------------------------
     plt.subplot(4,2,1)
     plt.plot(time, V_tot, color='blue', linewidth=1.5)
     plt.xlabel('Time')
     plt.ylabel('Total velocity')
     plt.grid(True)

     # E_k plot:
     #---------------------------------------------------------------
     plt.subplot(4,2,2)
     plt.plot(time, E_k, color='blue', linewidth=1.5)
     plt.ylim(0, 5) # can be removed to look at the noisy component
     plt.xlabel('Time')
     plt.ylabel('Kinetic energy')
     plt.grid(True)

     # pair distance distribution function plot:
     #---------------------------------------------------------------
     plt.subplot(4,2,3)
     plt.hist(Dist[500], 70, density=True, facecolor='blue', alpha=0.75)  # plot hist only at a single time-point
     plt.xlabel('Pair distance, $d_{ij}$')
     plt.ylabel('Probability')
     plt.xlim(0, 3.0)  # can be removed to look at the noisy component
     plt.grid(True)

     # Plot velocity norm pdf and hist:
     #---------------------------------------------------------------
     plt.subplot(4,2,4)
     plt.hist(V_reshaped, 70, density=True, facecolor='blue', alpha=0.75)
     plt.plot(np.arange(0, 0.5, 0.01), v_pdf, linewidth=2, color='red')
     plt.xlabel('Velocity norm')
     plt.ylabel('Probability')
     plt.xlim(0, 0.5) # can be removed to look at the noisy component
     plt.grid(True)

     # Plots for x,y velocities and coordinates:
     #---------------------------------------------------------------
     plt.subplot(4,2,5)  # Position number option
     plt.hist(V_reshaped_x, 40, density=True, facecolor='blue', alpha=0.75)
     plt.plot(np.arange(-0.5, 0.5, 0.01), v_x_pdf, linewidth=2, color='red')
```

```
    plt.xlabel('x velocities')
    plt.ylabel('Probability')
    plt.xlim(-0.4, 0.4) # can be removed to look at the noisy component
    plt.grid(True)

    plt.subplot(4,2,6)  # Position number option
    plt.hist(V_reshaped_x, 40, density=True, facecolor='blue', alpha=0.75)
    plt.plot(np.arange(-0.5, 0.5, 0.01), v_x_pdf, linewidth=2, color='red')
    plt.xlabel('y velocities')
    plt.ylabel('Probability')
    plt.xlim(-0.4, 0.4) # can be removed to look at the noisy component
    plt.grid(True)

    plt.subplot(4,2,7)
    plt.hist(R_reshaped_x, 70, density=True, facecolor='blue', alpha=0.75)
    plt.plot(np.arange(-1.5, 1.5, 0.01), x_pdf, linewidth=2, color='red')
    plt.xlabel('x coordinates')
    plt.ylabel('Probability')
    plt.xlim(-1.5, 1.5) # can be removed to look at the noisy component
    plt.grid(True)

    plt.subplot(4,2,8)
    plt.hist(R_reshaped_y, 70, density=True, facecolor='blue', alpha=0.75)
    plt.plot(np.arange(-1.5, 1.5, 0.01), y_pdf, linewidth=2, color='red')
    plt.xlabel('y coordinates')
    plt.ylabel('Probability')
    plt.xlim(-1.5, 1.5) # can be removed to look at the noisy component
    plt.grid(True)

    plt.savefig(filename)
    plt.show()


def animate_particles(R, time_step, filename='movie.mp4'):
    """Generates an mp4 movie with the particle trajectories
    using MatPlotLib. """

    if os.path.isfile(filename):
        print('WARNING (animate_particles): The output file', filename, 'exists. Skipping animation.')
        return

    frames = R.shape[0]
    frames_per_second = 1. / time_step

    fig = plt.figure(figsize=(6, 6))
    ax = plt.subplot()

    plt.xlabel('x')
    plt.ylabel('y')
    plt.axis('image')
    plt.xlim([-1.0, 1.0])
    plt.ylim([-1.0, 1.0])

    markers = ax.scatter([], [], facecolor='red', s=180, alpha=0.5)
    text = plt.text(-0.9, 0.9, 'frame =', ha='left')

    def update(frame, markers, text):
        print('frame =', frame)
        r = R[frame]
        markers.set_offsets(r.T)
        text.set_text('frame = {}'.format(frame))

    anim = animation.FuncAnimation(
        fig, update,
        frames=frames, interval=50,
        fargs=(markers, text),
        blit=False, repeat=False,
    )

    writer = animation.writers['ffmpeg'](fps=frames_per_second)
    anim.save(filename, writer=writer, dpi=100)
    plt.close()


if __name__ == '__main__':
    # Read h5:
    with h5py.File('ca1_step1_output_data.h5', 'r') as hfile:
        print(hfile.keys())

        dataset_R = hfile['R']
        R = dataset_R[:]
        dataset_V = hfile['V']
        V = dataset_V[:]

        time_step_s = dataset_V.attrs['time_step_s']
        radius = dataset_V.attrs['radius']
        v_variance = dataset_V.attrs['v_variance']
+
+    # Calling del on hfile is not required since you used the with statement
+
```

```
del dataset_R, dataset_V, hfile

# Plot the y-position of particle number 123 as a function of time t:
y_position_n123 = R[:, 1, 123]
time = np.arange(0, R.shape[0]*time_step_s, time_step_s) # [sec]

plt.plot(time, y_position_n123,
        color='red', linewidth=1.5)
plt.title('Particle 123 y-position')
plt.xlabel('Time')
plt.ylabel('y-position')
plt.grid(True)
plt.show()

# Call the statistical plots function:
plot_statistical_analysis(R, V, time_step_s, filename='ca1_step2_figure_summary.svg')
# Call the animation function:
animate_particles(R, time_step_s, filename='ca1_step1_movie.mp4')
```