Assignment 1- Group 4

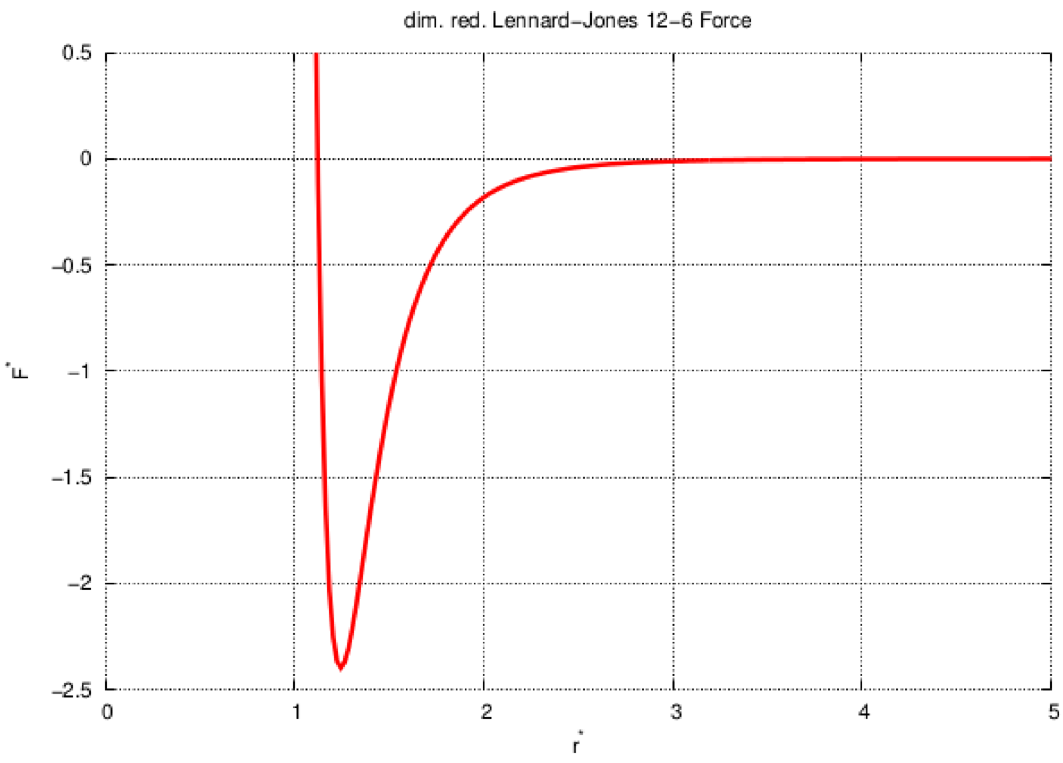
June 9, 2020

1 Linked Cell Algorithm

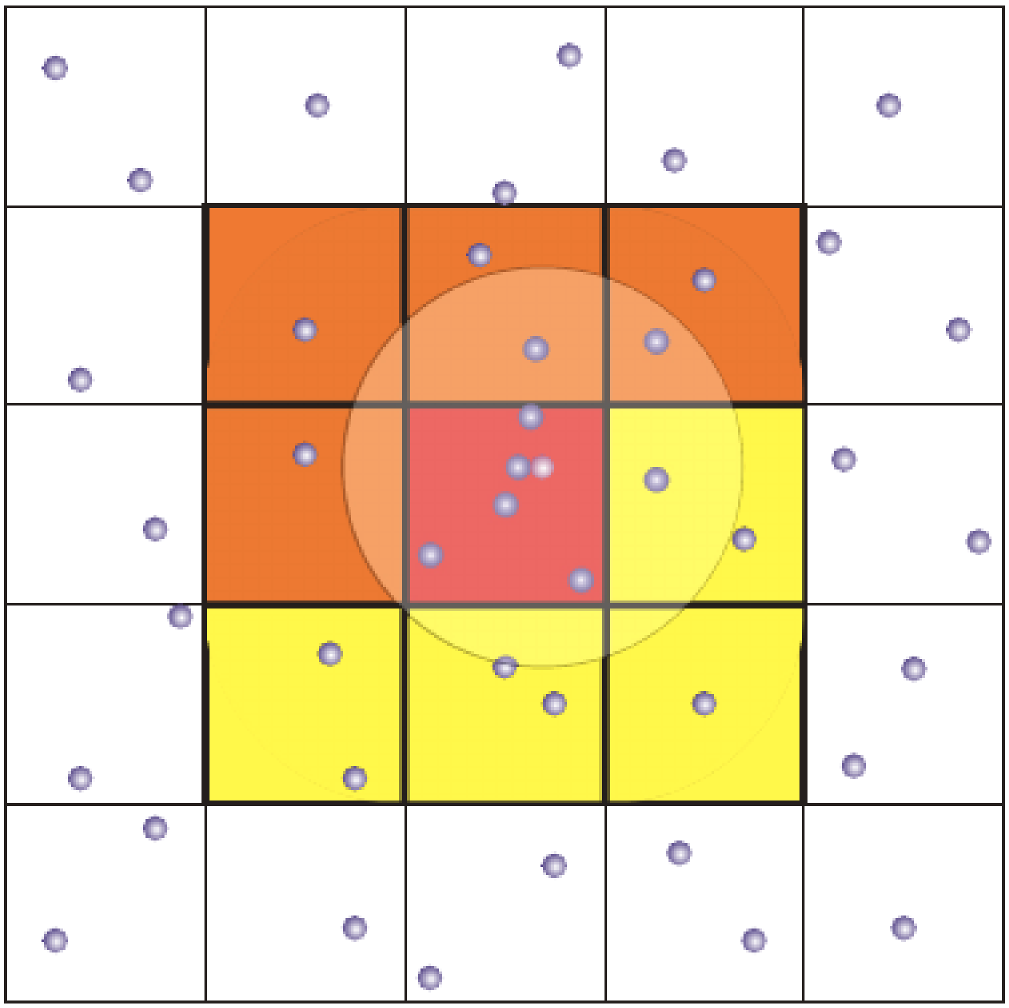
[Total : 34 points]

1.1 Classical Linked-Cell Algorithm

We discussed in the first worksheet that force calculation takes most of the time during molecular dynamics simulation. Therfore, in order to speed up the simulation, we need to improve the process of force calculation. The naive direct particle to particle interaction has complexity $\mathcal{O}(N^2)$. We will now have a look at methods that will reduce the complexity. In this assignment you will work with **Linked-Cell Algorithm**. There are many short range potentials like LJ potential (as shown in the figure below). For each molecule, an influence volume (closed sphere) with cut-off radius r_c can be assumed. Every molecule outside this influence volume is neglected.



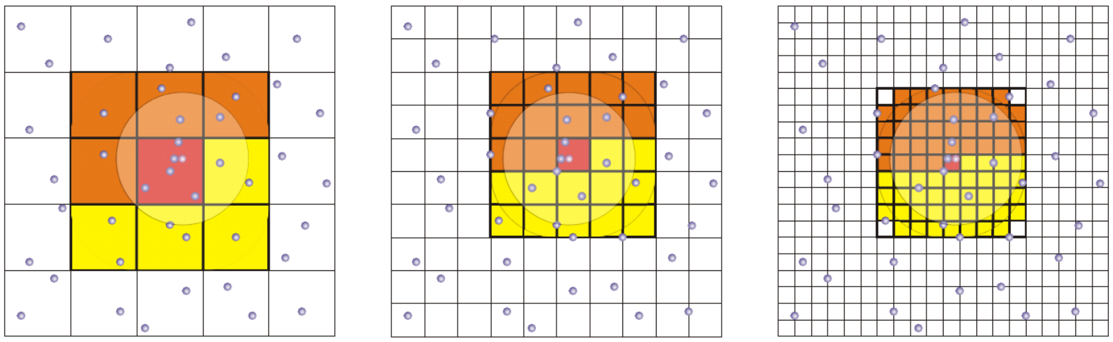
Therfore, during the calculation of force we need to calculate to consider the molecules within the sphere only. Since the total number of molecule within the sphere is small compared to the total number of molcules, the complexity of this algortihm is $\mathcal{O}(N)$. In the Linked-Cell Algorithm, you divide your domain into cells. The size of each cell is equal to the cut-off radius r_c. For every cell, there is a list of neighboring cells. The neighboring cells are those that are in direct contact with the current cell. In the figure below, the neighboring cells of the cell in red are shown in yellow and orange in color.



Let us try to implement this algorithm first.

1.2 Variable Linked-Cell

In linked-cell algorithm, we might interact with particles that are outside the cut-off radius. We can decrease these interactions using **Variable Linked-Cell Algorithm**. In variable linked-cell, the length of each cell is $\frac{r_c}{a}$, where a is generally an integer. We will call this as variable linked cell parameter. The neighboring cell of any cell will consist of those cells which might partially or completely fall within the cut-off sphere for any point inside the concerned cell. An example of variable linked-cell is shown below:



[1]:
```
%load_ext autoreload
%autoreload 2
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams, patches
import time
from itertools import product
import utils
import particle as pr
import cell_1
import cell_2
```

We define the function **lj_potential** to calculate the Lennard Jones Potential. Then we calculate the potential between two particles separated at a distance of 0.1, 0.2, 0.3 and 0.4 respectively. Notice the rapid decay of potential. **NOTE**: The function **lj_potential** is implemented in file *utils.py* which is inside the same directory as this iPython notebook.

[2]:
```
print("Potential when distance is 0.1:  ", utils.lj_potential(0.1))
print("Potential when distance is 0.2:  ", utils.lj_potential(0.2))
print("Potential when distance is 0.3:  ", utils.lj_potential(0.3))
print("Potential when distance is 0.4:  ", utils.lj_potential(0.4))
```

```
Potential when distance is 0.1:   -9.998999999999997
Potential when distance is 0.2:   -0.15624975585937495
Potential when distance is 0.3:   -0.013717419243152115
Potential when distance is 0.4:   -0.0024414061903953546
```

1.2.1 Definition of class Particle

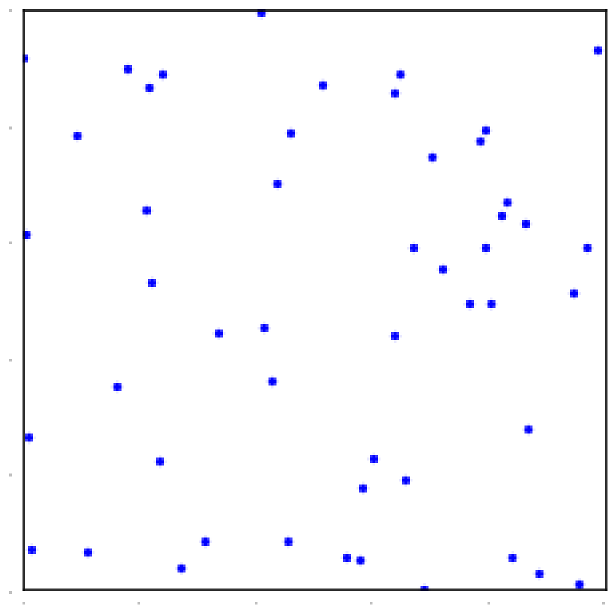
Now we define the class **Point** and **Particle** to define the properties of a particular particle. Open file *particle.py* and go through the implementation of both the classes and understand the implementation of both.

Let us create N particles and plot their position.

```
[3]: domain = 1.0
     N = 50
     def get_list_particles(N):
         list_particles = []
         for i in range(N):
             list_particles.append(pr.Particle(domain=domain))
         return list_particles

     list_particles = get_list_particles(N)

     # plotting
     ax = plt.gca()
     for particle in list_particles:
         particle.plot(color='b', s=5)
     ax.tick_params(axis='both',labelsize=0, length = 0)
     plt.xlim(left=0, right=domain)
     plt.ylim(bottom=0, top=domain)
     ax.set_aspect('equal', adjustable='box')
     plt.show()
```

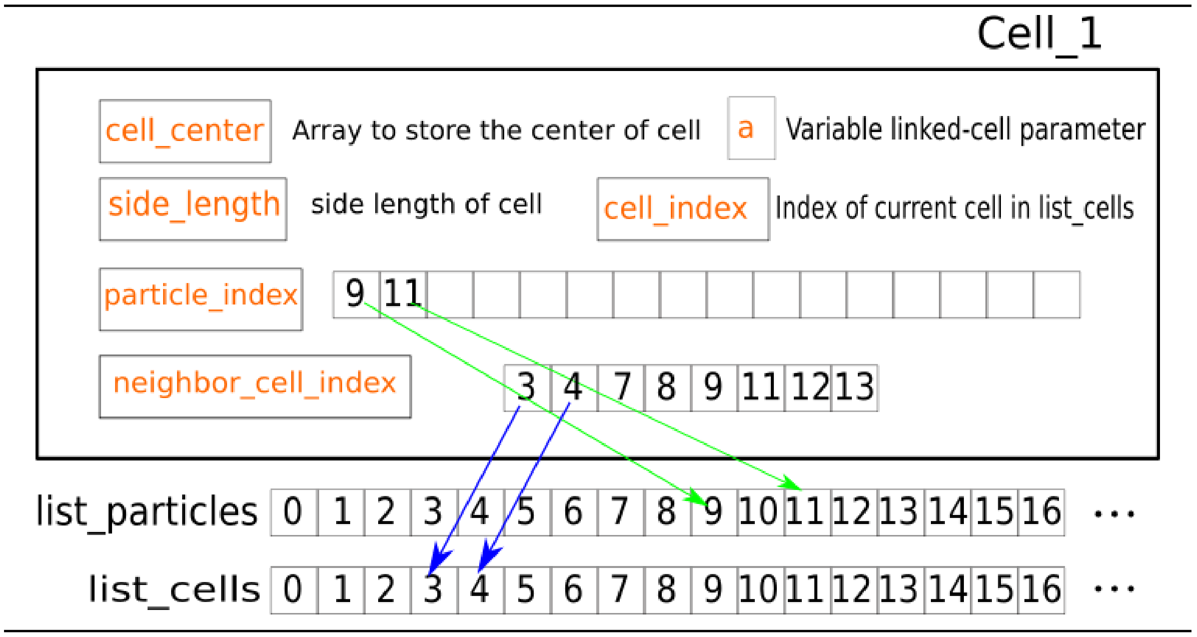


We will implement a class that defines **cell** in two different ways. We will call those two implementations as **Cell_1** and **Cell_2**. Both classes should be child classes of **Cell**. Let us discuss the first implementation(**Cell_1**).

1.3 Definition of class Cell_1

There are five member variables of class **Cell_1**

1. **cell__center**: Array of size two which contains the x and y coordinates of the center of the cell
2. **side__length**: The side length of the cell
3. **a**: Variable linked cell parameter
4. **particle__index**: List that contains the indices of particles (inside the list **list__particles**) which fall inside the cell
5. **neighbor__cell__index**: List that contain the indices of cells which are neighbors of current cell. These indices represents the location of cell in the list named **list__cells**. **list__cells** is list of all cells inside the domain. We will explain **list__cells** in detail later.
6. **cell__index**: Index of current cell in **list__cells**.



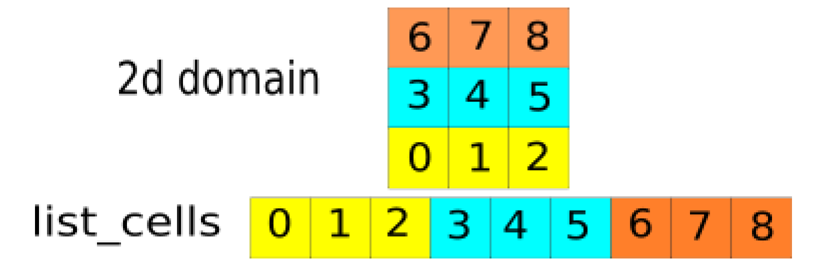
Cell_1
cell_center
Array to store the center of cell
a
Variable linked-cell parameter
side_length
side length of cell
cell_index
Index of current cell in list_cells
particle_index
9 11
neighbor_cell_index
3 4 7 8 9 11 12 13
list_particles 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ···
list_cells 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ···

For example: for the figure shown above, the cell has particle number 9 and 11. The neighboring cell indices are 3, 4, 7, 8, 9, 11, 12, 13.

Constructor of class: In order to create an object of class **Cell__1**, one need to provide following parameters:

1. lx : x-coordinate of bottom left corner of the cell
2. ly : y-coordinate of bottom left corner of the cell
3. r__c : Cut-off radius
4. cell__index : Index of cell in list__cells
5. neighbor__delta__coordinate : List of numpy array(of size 2). We will provide detailed explanation of this variable in the first task.
6. a : Variable linked-cell parameter (default value 1)
7. domain : Size of domain (default value 1.0)

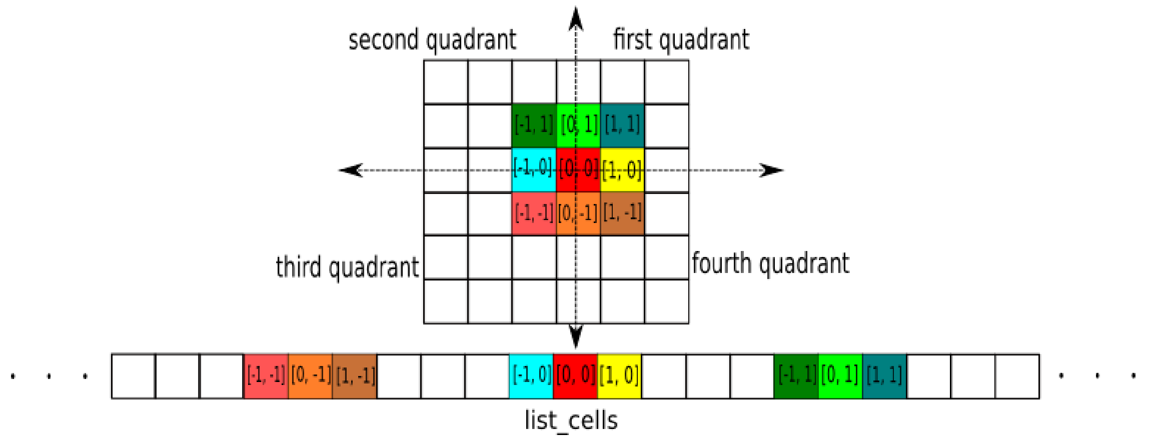
Before we move further and describe the member functions of class **Cell__1**, we take a short detour and define **list__cells**. **list__cells**: This is list of all the cells (object of type Cell__1) inside the domain. The cells are arranged in row major order. This means we store the cells in one row and then move to the next one. Let us take a simple example of domain which has 9 cells (index 0-8) with 3 rows and 3 columns. Then the arrangement is as shown below:



1.3.1 Task 1: Find neighbors

[8 points]

Our first task is to find the neighbors of a cell. In order to efficiently calculate the neighbors, we will take advantage of the arrangement of cells. Let us take a simple example where $a = 1$ as shown below:



Let us consider x-axis as horizontal axis and y-axis as vertical axis. We want to know the neighbors of the cell marked in red. Since $a = 1$, the colored cells are neighbor cells which interact with the red colored cell. The corresponding position of cell in **list__cells** is also shown. If we consider the red cell as $[0, 0]$, then the relative 2-d index of the neighboring interaction cell is also marked. The indices are symmetric about $[0, 0]$. This property is independent of the position of the cell or the value of a. We will make use of this property to find the neighbors. For any arbitrary integer value of a, we need to find the cells which partially or completely fall within the cut-off sphere for any point inside the concerned cell. The corner of cell is the outer most point of the cell which represent the maximum coverage of the cell. Imagine a circle of radius r_c at all four corners of cell. We need to find all cells which partially or completely fall within any of the four circles. The naive way of doing that is to calculate the distance between the four corners of the current cell to all four corners of the other cell. This gives us 16 combinations. We take the minimum distance amongst those 16 distance. If that distance is less than the cut-off radius(r_c) then the cell is the neighbor interaction cell. We can reduce this calculation by making use of the symmetry. Finding the neighbor cells along the x and y axis is a trivial problem (a cells along the x and y axis in both upward and downward directions) and does not require any calculations. Let us consider cells in first quadrant (upper right quadrant, see the figure above). The smallest distance between red cell and any cell in first quadrant is the distance between the top right corner of red cell and bottom left corner of the other cell. We can determine relative 2-d indices for the cells whose minimum distance is less that the cut-off radius in first quadrant. From this we can easily determine relative 2-d indices for other quadrants because of symmetry. So, we just store the relative 2-d indices for first quadrant. The list relative 2-d indices for first quadrant that falls inside the cut-off radius is the parameter **neighbor_delta_coordinate** that we pass to the constructor of **Cell_1**. For $a = 1$, **neighbor_delta_coordinate = [np.array(1,1)]**.

Task 1.1 [4 points]

Open the file *utils.py*. Implement the function **get_successor_neighbor_delta_coordinate**. It returns the **neighbor_delta_coordinate** as described above. We assume that all the cells are squares of same size. The parameters are: 1. a: Variable linked-cell parameter (default value is 1.0)

Task 1.2 [4 points]

Open the file *cell.py* and implement the function **create_neighbor_cell_index** inside the class **Cell**. This function creates the neighbor list for the current cell. Do not forget to take care of indices which point outside the domain.

1.3.2 Task 2: Create list of cells

[2 points]

Complete the implementation of the function named **get_list_cell** inside the file *cell_1.py*. It takes two parameter

1. r_c: Cut-off radius
2. neighbor_delta_coordinate
3. domain: This is an optional parameter during function call. Default value is 1.
4. a: Variable linked-cell parameter. Default value is 1 This function creates and returns the list of cells in order as described before.

1.3.3 Task 3 : Calculate Potential

[5 points]

Task 3.1 [2 points]

First member function of class **Cell_1** that we need to implemet is **p2p_self**. This function calculates the potential on all particle inside a cell due to other particles inside the same cell. The potential calculated is added in the the variable **phi** of particle object. This function takes list of particles **list_particles**(list of particles that we created above) as parameter. Your task is to implement this function in file *cell_1.py*.

Task 3.2 [2 points]

The second function is **p2p_neigbor_cells**. This function calculates the potential on all particle inside a cell due to particles in the neighor cells (cells index fiven by the variable neighbor_cell_index). The potential calculated is added in the the variable **phi** of particle object. Arguments are:

1. list_particles: List of all **Particle** objects
2. list_cells: List of all **Cell_1** objects Your task is to implement this function in file *cell_1.py*.

Task 3.3 [1 point]

Implement the function **calculate_potential** in file *cell_1.py*. This function calls the functions implemeted in **3.1** and **3.2** and calculates the total potential of particles inside the cell. Parameters are same as **p2p_neigbor_cells**.

1.3.4 Task 4: Assign particles to cells

[2 points]

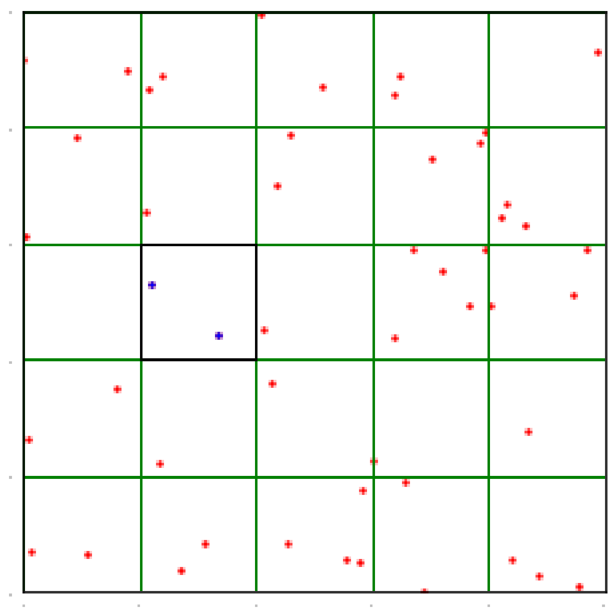
Complete the implementation of function **assign_particle_to_cell** in file *cell_1.py*. This function assigns a particle to the corresponding cell. The parameters are:

1. list_particles: List of all **Particle** objects
2. list_cells: List of all **Cell_1** objects
3. r_c: Cut-off radius
4. a: Variable linked-list parameter
5. domain: size of domain To assign particle to a cell use the function **add_particle**.

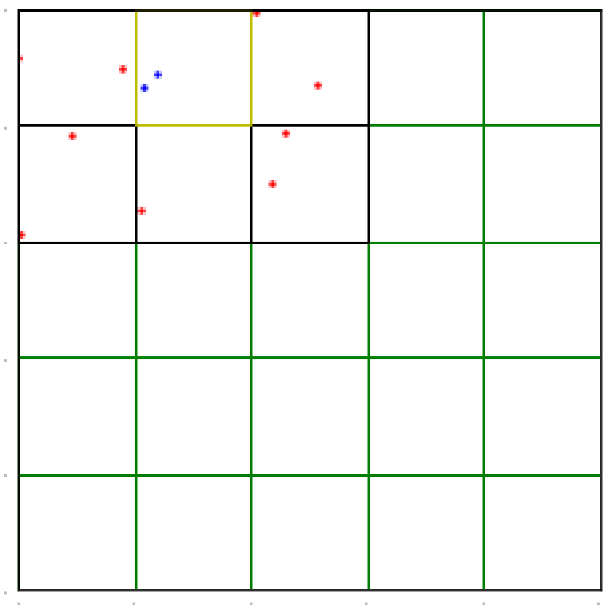
[4]:
```
r_c, a = 0.2, 1
delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
list_cells = cell_1.get_list_cell(r_c, delta_coordinate, domain=domain, a=a)
```

[5]:
```
cell_1.assign_particle_to_cell(list_particles, list_cells, r_c, a=a)
```

[6]:
```
# This block can be used for debugging
# In this block we plot all particles, choose a cell and plot it with different␣
 ↪cell
# Using this you can check if you are forming correct cells and assigning␣
 ↪correct particles to correct cell or not
idx = np.random.randint(low=0, high=len(list_cells)-1, size=1)[0]
a_cell = list_cells[idx]
ax = plt.gca()
for particle in list_particles:
    particle.plot()
utils.plot_all_cells(ax, list_cells, edgecolor='g', domain=domain)
a_cell.plot_cell(ax, edgecolor='k')
a_cell.plot_particles(list_particles, color='b')
ax.tick_params(axis='both',labelsize=0, length = 0)
plt.xlim(left=0, right=domain)
plt.ylim(bottom=0, top=domain)
ax.set_aspect('equal', adjustable='box')
plt.show()
```



[7]:
```
# This block can be used for debugging
# In this block we randomly choose a cell and plot its neighboring cells and␣
↪particles in different colors
# Using this you can check if you are assigning correct neighbors or not or not
idx = np.random.randint(low=0, high=len(list_cells)-1, size=1)[0]
a_cell = list_cells[idx]
ax = plt.gca()
utils.plot_all_cells(ax, list_cells, edgecolor='g', domain=domain)
a_cell.plot_particles(list_particles, color='b')
a_cell.plot_neighbor_cells(ax, list_cells, edgecolor='k')
a_cell.plot_cell(ax, edgecolor='y')
a_cell.plot_neighbor_cell_particles(list_cells, list_particles)
ax.tick_params(axis='both',labelsize=0, length = 0)
plt.xlim(left=0, right=domain)
plt.ylim(bottom=0, top=domain)
ax.set_aspect('equal', adjustable='box')
plt.show()
```



```
[8]: # Check if correct potential is calculated or not
     N = 100
     list_particles = get_list_particles(N)
     cell_1.set_potential_zero(list_particles)
     delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
     list_cells = cell_1.get_list_cell(r_c, delta_coordinate, domain=domain, a=a)
     cell_1.assign_particle_to_cell(list_particles, list_cells, r_c, a=a)
     cell_1.calculate_potential_linked_cell(list_cells, list_particles)
     direct_potential = cell_1.direct_potential_all_particles(list_particles)
     linked_cell_potential = cell_1.extract_linked_cell_potential(list_particles)
```

```
[9]: print("Mean relative error", utils.get_mean_relative_error(direct_potential,␣
      ↪linked_cell_potential))
```

```
Mean relative error 0.0012947311825780977
```

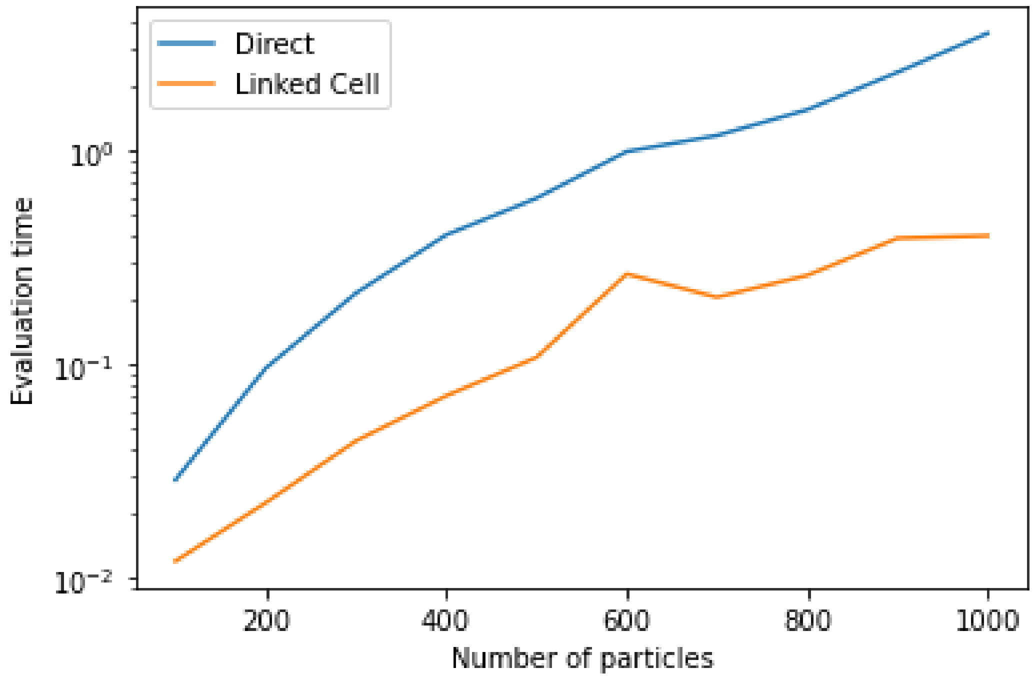
1.3.5 Time Scaling

Scaling with respect to number of particles for fixed cut-off radius ($r_c = 0.2$) and variable linked cell term $a = 8$

```
[10]: num_particles = [np.int(100 * i) for i in range(1,11)]
      n_instances = len(num_particles)
      time_linked_cell = np.zeros(n_instances, dtype=np.float)
      time_direct = np.zeros(n_instances, dtype=np.float)
      r_c, a, domain = 0.2, 8, 1.0
      delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
```

```python
for idx, N in enumerate(num_particles):

    list_particles = get_list_particles(N)
    cell_1.set_potential_zero(list_particles)
    list_cells = cell_1.get_list_cell(r_c, delta_coordinate, domain=domain, a=a)
    cell_1.assign_particle_to_cell(list_particles, list_cells, r_c, a=a)
    start_lc = time.time()
    cell_1.calculate_potential_linked_cell(list_cells, list_particles)
    end_lc = time.time()
    linked_cell_potential = cell_1.extract_linked_cell_potential(list_particles)
    time_linked_cell[idx] = end_lc - start_lc
    start_direct = time.time()
    direct_potential = cell_1.direct_potential_all_particles(list_particles)
    end_direct = time.time()
    time_direct[idx] = end_direct - start_direct
plt.plot(num_particles, time_direct, label='Direct')
plt.plot(num_particles, time_linked_cell, label='Linked Cell')
plt.legend()
plt.yscale('log')
plt.xlabel('Number of particles')
plt.ylabel('Evaluation time')
plt.show()
```



Now we find the ratio of approximate slope of the two curves (assuming it to be a straight line).

[11]:
```
slope_direct = (np.log(time_direct[-1]) - np.log(time_direct[0])) /␣
 ↪(num_particles[-1] - num_particles[0])
slope_lc = (np.log(time_linked_cell[-1]) - np.log(time_linked_cell[0])) /␣
 ↪(num_particles[-1] - num_particles[0])
print(slope_direct / slope_lc)
```

```
1.371226218314674
```

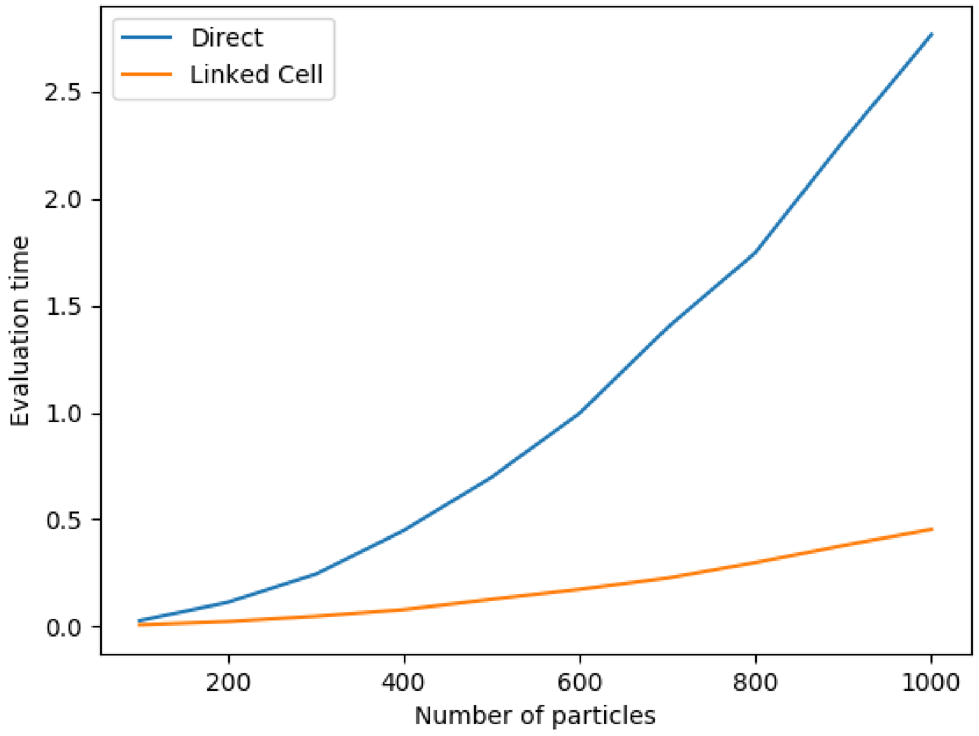
1.3.6 Question:

[2 points]

Explain with reasoning the behavior of the graph that we obtained above and the value of the ratio of the slope.

Solution From the graph, it is observed that the time required for the computation of potentials using the direct method is greater than that of the linked cell method. This is because, for computation of potential using linked cell method, we consider only the particles which are within the cutoff radius whereas for the computation of potential using the direct method, we consider all the particles in the domain. Hence the time required for the computation of potential using the direct method for each particle in the domain is greater than that of the linked cell method. The ratio of the slopes obtained is greater than 1. This also means that with an increase in the number of particles in the domain, time taken to calculate potential using direct method is greater than the linked cell method. Also, observing the linearly scaled y-axis graph in figure we can see that the time complexity of linked cell algorithm is linear meaning $\mathcal{O}(N)$ and for the direct method it's quadratic meaning $\mathcal{O}(N^2)$.

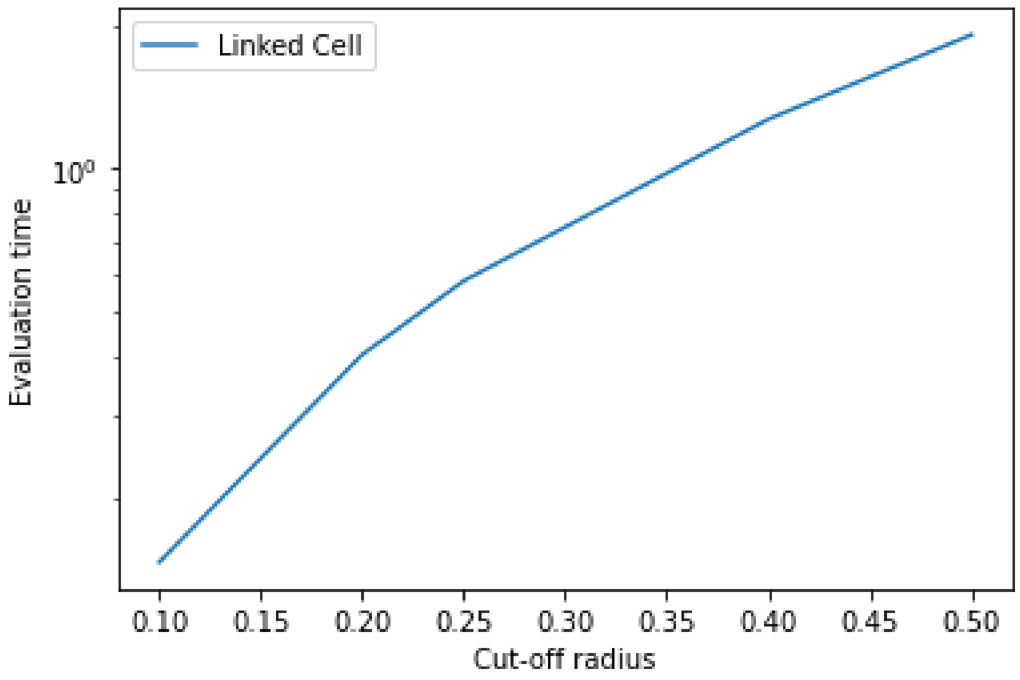
Fig 1: Linear scaled graph

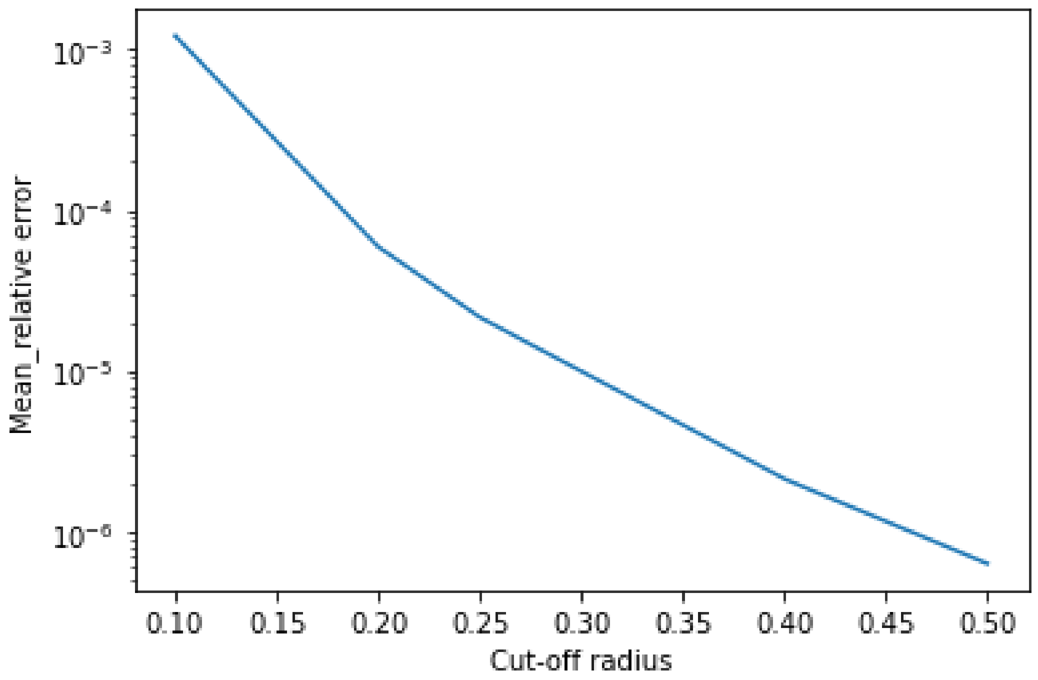


1.3.7 Evaluation time scaling

Let us fix the number of particles $N = 1000$ and variable linked cell term $a = 8$. We analyse the error with respect cut-off radius and evaluation time with respect to the cut-off radius.

[12]:
```
N = 1000
R = [0.1, 0.2,0.25, 0.4, 0.5]
n_instances = len(R)
time_linked_cell = np.zeros(n_instances, dtype=np.float)
mean_relative_error = np.zeros(n_instances, dtype=np.float)
a, domain = 8, 1.0
list_particles = get_list_particles(N)
direct_potential = cell_1.direct_potential_all_particles(list_particles)
delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
for idx, r_c in enumerate(R):
    cell_1.set_potential_zero(list_particles)
    list_cells = cell_1.get_list_cell(r_c, delta_coordinate, domain=domain, a=a)
    cell_1.assign_particle_to_cell(list_particles, list_cells, r_c, a=a)
    start_lc = time.time()
    cell_1.calculate_potential_linked_cell(list_cells, list_particles)
    end_lc = time.time()
    linked_cell_potential = cell_1.extract_linked_cell_potential(list_particles)
    time_linked_cell[idx] = end_lc - start_lc
    mean_relative_error[idx] = utils.get_mean_relative_error(direct_potential,␣
 ↪linked_cell_potential)
plt.plot(R, time_linked_cell, label='Linked Cell')
plt.legend()
plt.yscale('log')
plt.xlabel('Cut-off radius')
plt.ylabel('Evaluation time')
plt.show()
plt.plot(R, mean_relative_error)
plt.yscale('log')
plt.xlabel('Cut-off radius')
plt.ylabel('Mean_relative error')
plt.show()
```





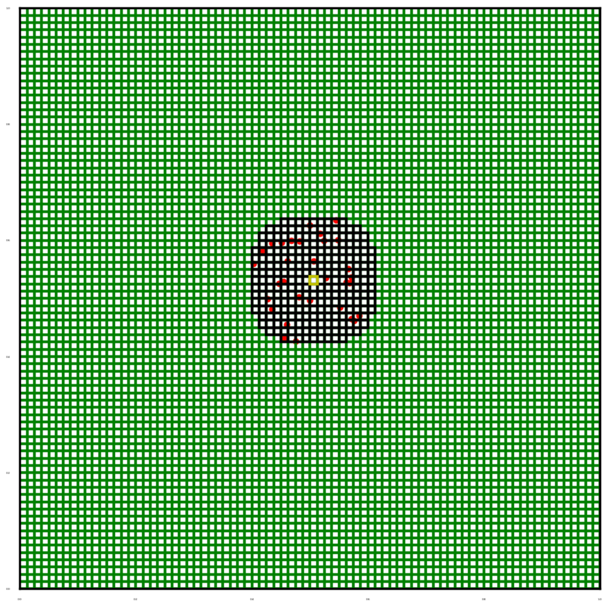
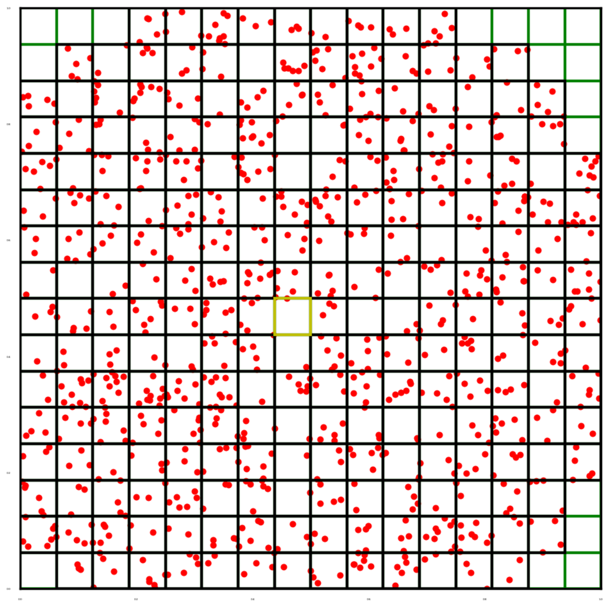
1.3.8 Question:

[2 points]

We observe that the evaluation time increases with increase of cut-off radius whereas the mean relative error decreases. Explain this behavior.

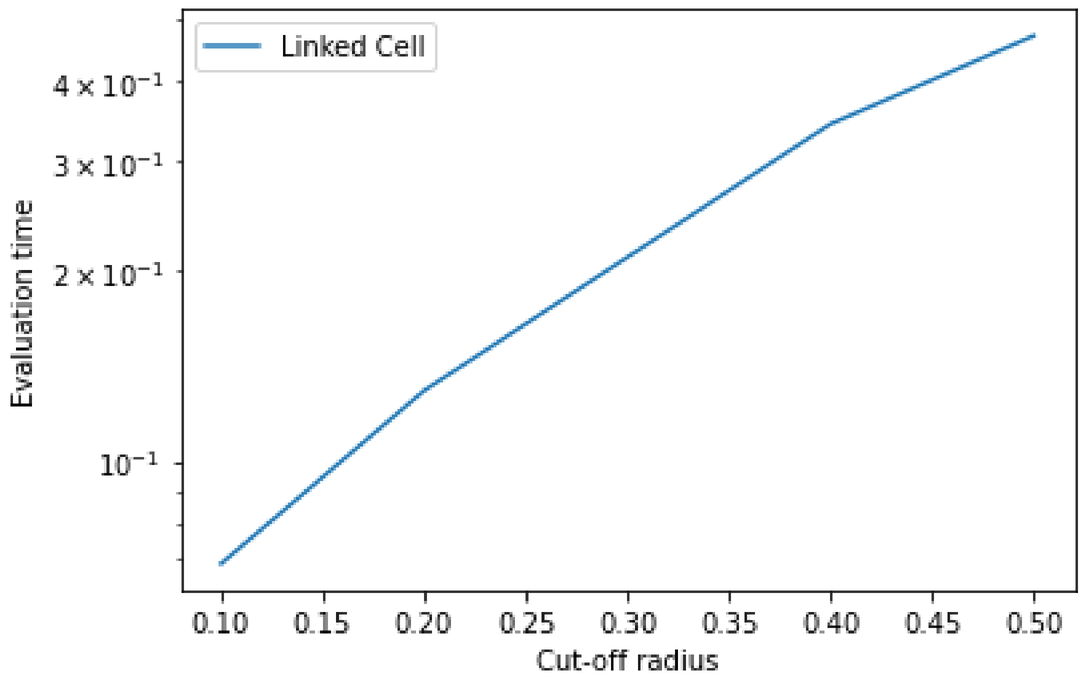
Solution With the increase in cut-off radius, we are considering more and more particles in the neighborhood of particle (of interest) for calculating potentials. Hence the time required for the computation of potential, and thus force, increases since more particles are taken into account. At cut-off radius 0.5 we are almost taking into account the whole domain and thus it would be similar to the direct solver method.

As we are computing the potential considering increased number of particles with increasing cut off radius, the number of neglected short-range potentials decreases and we obtain potential values using linked cell method close to direct potential i.e. if the cut off radius tends to domain size, the linked cell potential tends to direct potential. Hence the mean relative error decreases with an increase in cut-off radius.

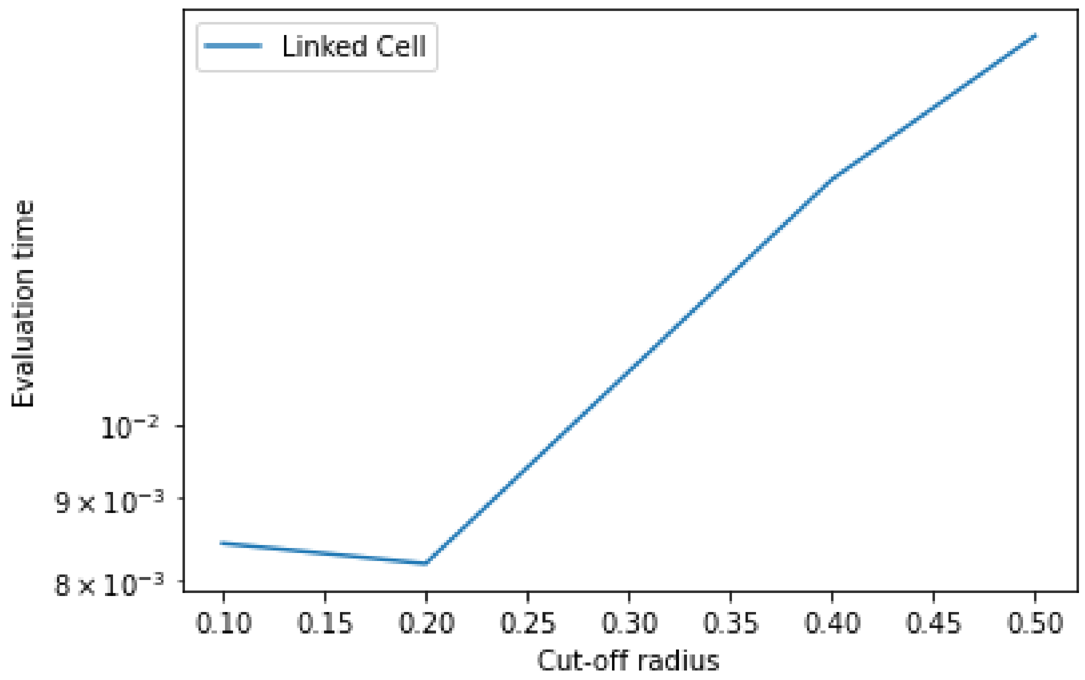
Fig 1: For Cutoff Radius = 0.1	**Fig 2: For Cutoff Radius = 0.5**
	

Let us plot the Evaluation time with respect to cut-off radius (same as before),but with $N = 500$ particles and $a = 8$. Then we do the same with $N = 100$ particles

[13]:
```
N = 500
R = [0.1, 0.2, 0.4, 0.5]
n_instances = len(R)
time_linked_cell = np.zeros(n_instances, dtype=np.float)
a, domain = 8, 1.0
list_particles = get_list_particles(N)
direct_potential = cell_1.direct_potential_all_particles(list_particles)
delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
for idx, r_c in enumerate(R):
    cell_1.set_potential_zero(list_particles)
    list_cells = cell_1.get_list_cell(r_c, delta_coordinate, domain=domain, a=a)
    cell_1.assign_particle_to_cell(list_particles, list_cells, r_c, a=a)
    start_lc = time.time()
    cell_1.calculate_potential_linked_cell(list_cells, list_particles)
    end_lc = time.time()
    linked_cell_potential = cell_1.extract_linked_cell_potential(list_particles)
    time_linked_cell[idx] = end_lc - start_lc
plt.plot(R, time_linked_cell, label='Linked Cell')
plt.legend()
plt.yscale('log')
plt.xlabel('Cut-off radius')
plt.ylabel('Evaluation time')
plt.show()
```



[14]:
```
N = 100
R = [0.1, 0.2, 0.4, 0.5]
n_instances = len(R)
time_linked_cell = np.zeros(n_instances, dtype=np.float)
a, domain = 8, 1.0
list_particles = get_list_particles(N)
direct_potential = cell_1.direct_potential_all_particles(list_particles)
delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
for idx, r_c in enumerate(R):
    cell_1.set_potential_zero(list_particles)
    list_cells = cell_1.get_list_cell(r_c, delta_coordinate, domain=domain, a=a)
    cell_1.assign_particle_to_cell(list_particles, list_cells, r_c, a=a)
    start_lc = time.time()
    cell_1.calculate_potential_linked_cell(list_cells, list_particles)
    end_lc = time.time()
    linked_cell_potential = cell_1.extract_linked_cell_potential(list_particles)
    time_linked_cell[idx] = end_lc - start_lc
plt.plot(R, time_linked_cell, label='Linked Cell')
plt.legend()
plt.yscale('log')
plt.xlabel('Cut-off radius')
plt.ylabel('Evaluation time')
plt.show()
```



1.3.9 Question:

[2 points]

We observe a different behavior of the evaluation time. Explain the results obtained. (Hint: There are two competing factors. What are those?)

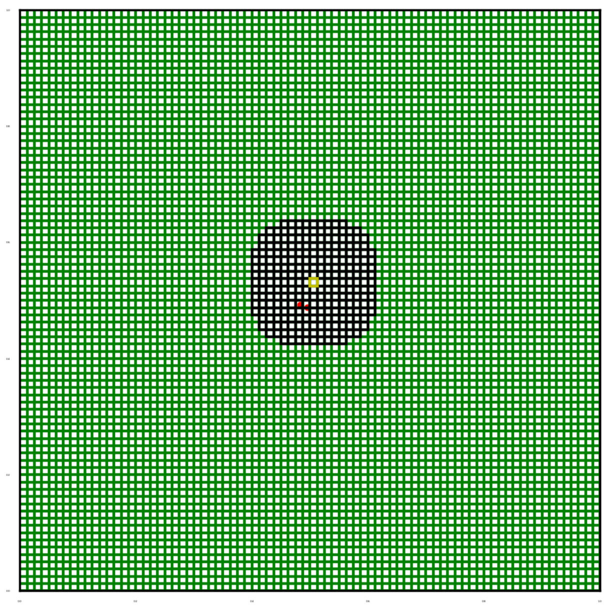
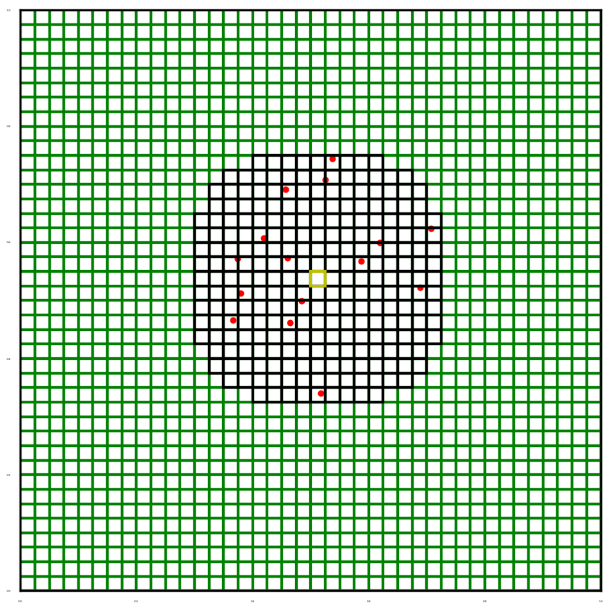
Solution The two competing factors are;

1. Number of potential (force) evaluations/ Particle density (The number of potential evaluations within the cut-off radius increase as particle density increases)
2. Cutoff radius/ Number of cell traversals (The number of cells that needs to be traversed increases when cut off radius decreases while keeping a constant)

In the first graph where the total number of particles is 500 with varying cutoff radius, the evaluation time increases with an increase in the cutoff radius as we are considering more particles in the neighborhood of particle (of interest) for the computation of potential. As the particle density is higher the number of potential evaluations are higher in every case and the number of force evaluations dominates at all instances.

However, in the second graph where the total number of particles is 100, with varying cutoff radius, we see an initial dip in the evaluation time followed by a gradual increase. This is because, with a reduced number of total particles, the particle density per cell also reduces. At a very small cut-off radius the number of cells will be very high and there would be more and more empty cells that we would have to traverse. So at 0.1 cut-off radius the cell traversal time will dominate compared to the potential calculations. As we increase the cutoff radius, the number of cells will be reduced (for every increase in cutoff radius by 0.1, the number of cells reduces by 1/4). After cut-off radius

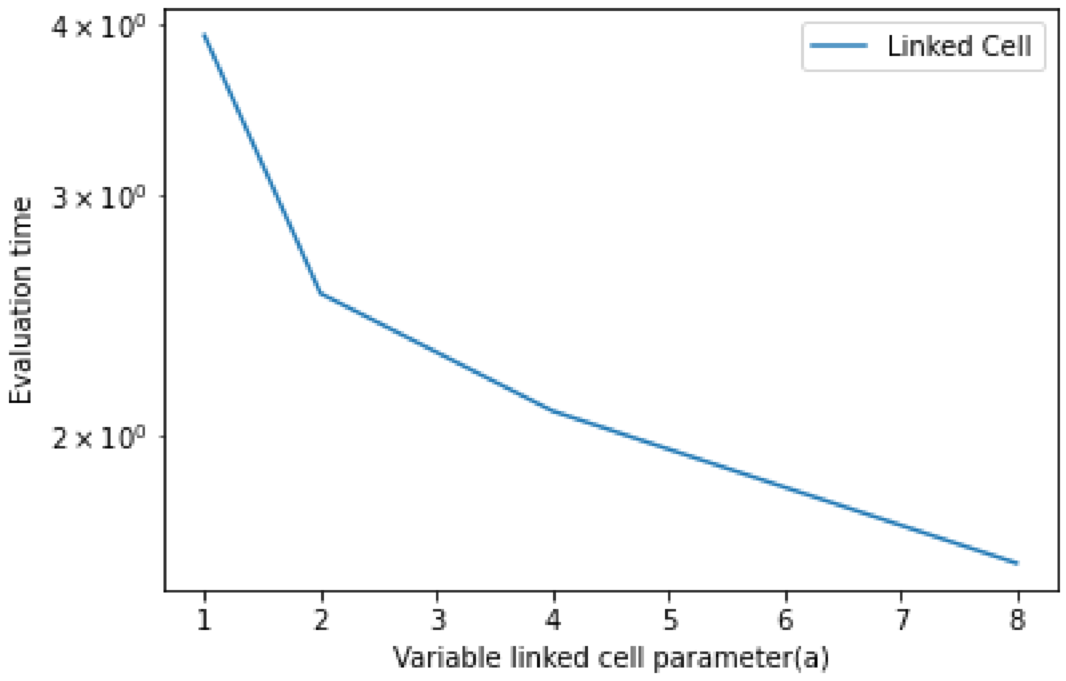
0.2, the number of potential calculations will start to dominate, and the evaluation time increases with increasing cut off radius since more particles in the neighborhood are taken into account for potential calculation.

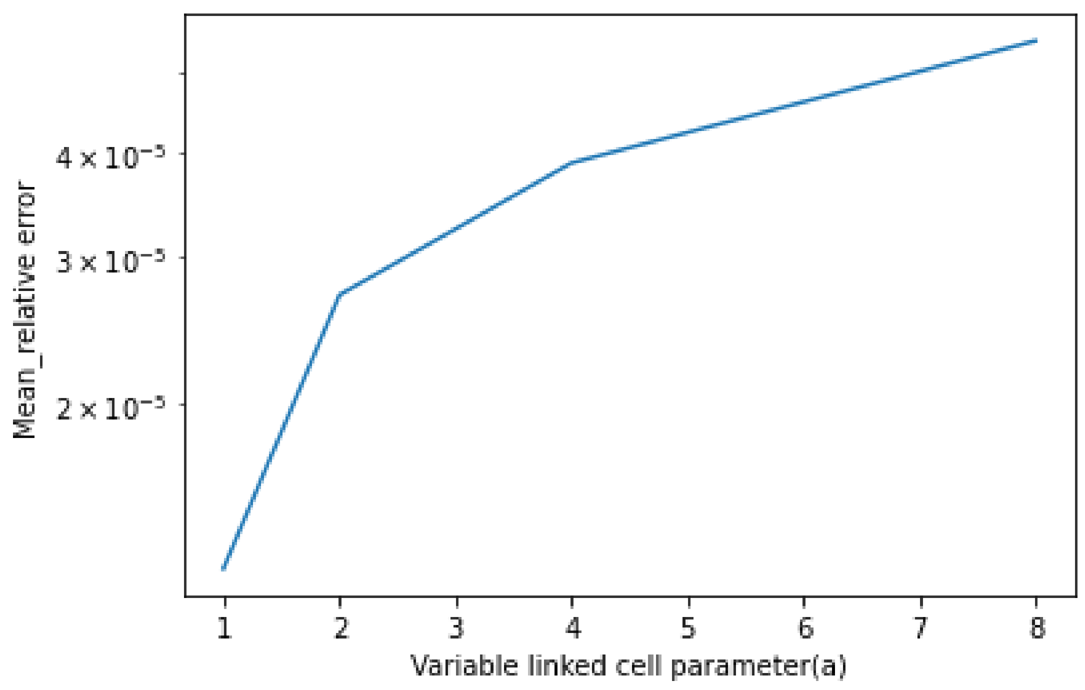
Fig 1: For Cutoff Radius = 0.1	**Fig 2: For Cutoff Radius = 0.2**
	

Let us plot the Evaluation time and mean relative error with respect to variable linked list parameter a. We fix number of particles $N = 2000$ paticles and cutoff radius $r_c = 0.2$.

```
[21]: N = 2000
      r_c = 0.2
      A = [1, 2, 4, 8]
      n_instances = len(A)
      time_linked_cell = np.zeros(n_instances, dtype=np.float)
      domain = 1.0
      list_particles = get_list_particles(N)
      direct_potential = cell_1.direct_potential_all_particles(list_particles)
      mean_relative_error = np.zeros(n_instances, dtype=np.float)
      for idx, a in enumerate(A):
          cell_1.set_potential_zero(list_particles)
          delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
          list_cells = cell_1.get_list_cell(r_c, delta_coordinate, domain=domain, a=a)
          cell_1.assign_particle_to_cell(list_particles, list_cells, r_c, a=a)
          start_lc = time.time()
          cell_1.calculate_potential_linked_cell(list_cells, list_particles)
          end_lc = time.time()
          linked_cell_potential = cell_1.extract_linked_cell_potential(list_particles)
          time_linked_cell[idx] = end_lc - start_lc
```

```
    mean_relative_error[idx] = utils.get_mean_relative_error(direct_potential,␣
 ↪linked_cell_potential)
plt.plot(A, time_linked_cell, label='Linked Cell')
plt.legend()
plt.yscale('log')
plt.xlabel('Variable linked cell parameter(a)')
plt.ylabel('Evaluation time')
plt.show()
plt.plot(A, mean_relative_error)
plt.yscale('log')
plt.xlabel('Variable linked cell parameter(a)')
plt.ylabel('Mean_relative error')
plt.show()
```





1.3.10 Question

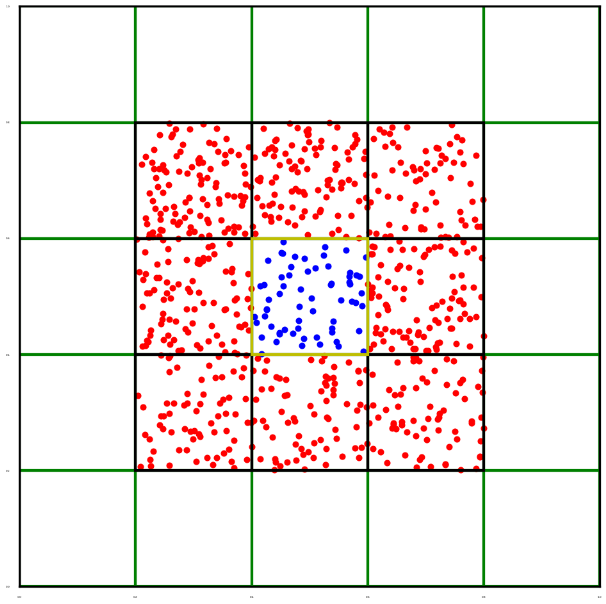
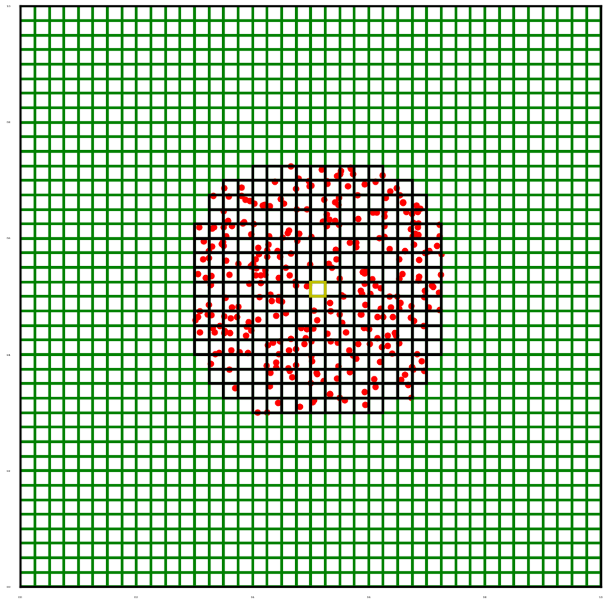
[2 points]

1. Why does the potential evalution time decreases with increase of a?
2. Why does the mean relative error increase with increase of a?

Solution

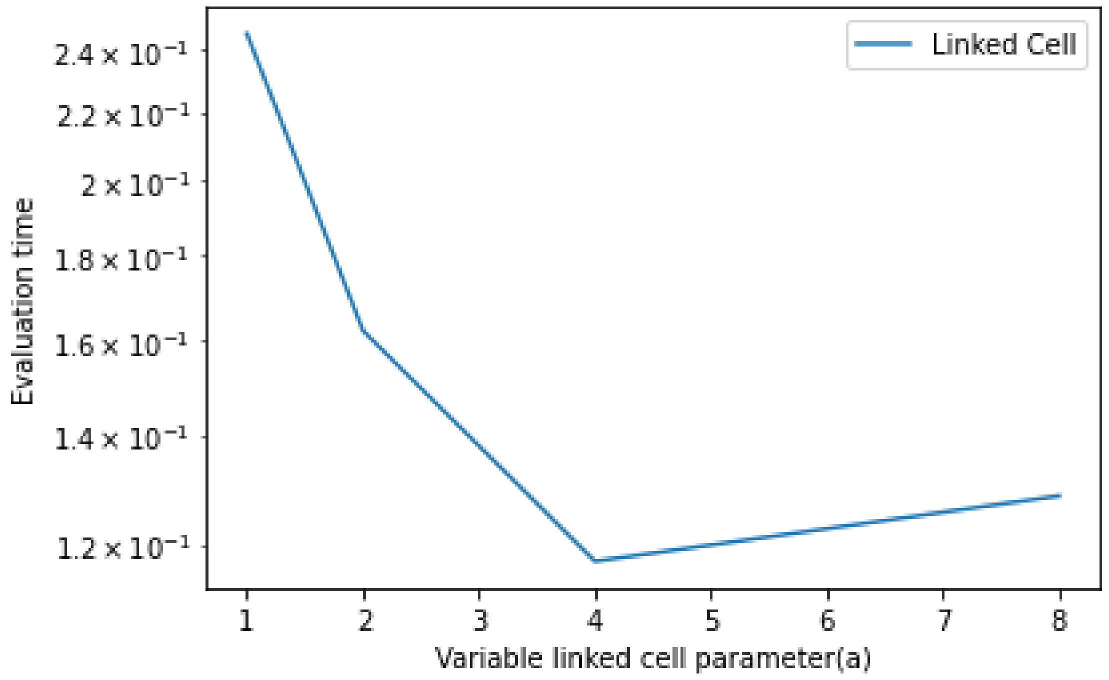
1. An increase in the value of the variable linked cell parameter a decreases the evaluation time because as variable linked cell parameter increases, the neighborhood of cells taken into account for potential calculation converges to a sphere in 3D and a circle in 2D.Thus by increasing the variable linked cell parameter a, we are considering the potentials of the particles that are more strictly within the cutoff radius and reducing the number of particles that are considered outside the cutoff radius of the particle (of interest), thereby reducing the evaluation time.

2. As stated above, with the increase in the value of the variable linked cell parameter, we are strictly considering the potentials of the particle that are within the cutoff radius. This decreases the total number of particles in consideration for calculating potential thus increasing the mean relative error.

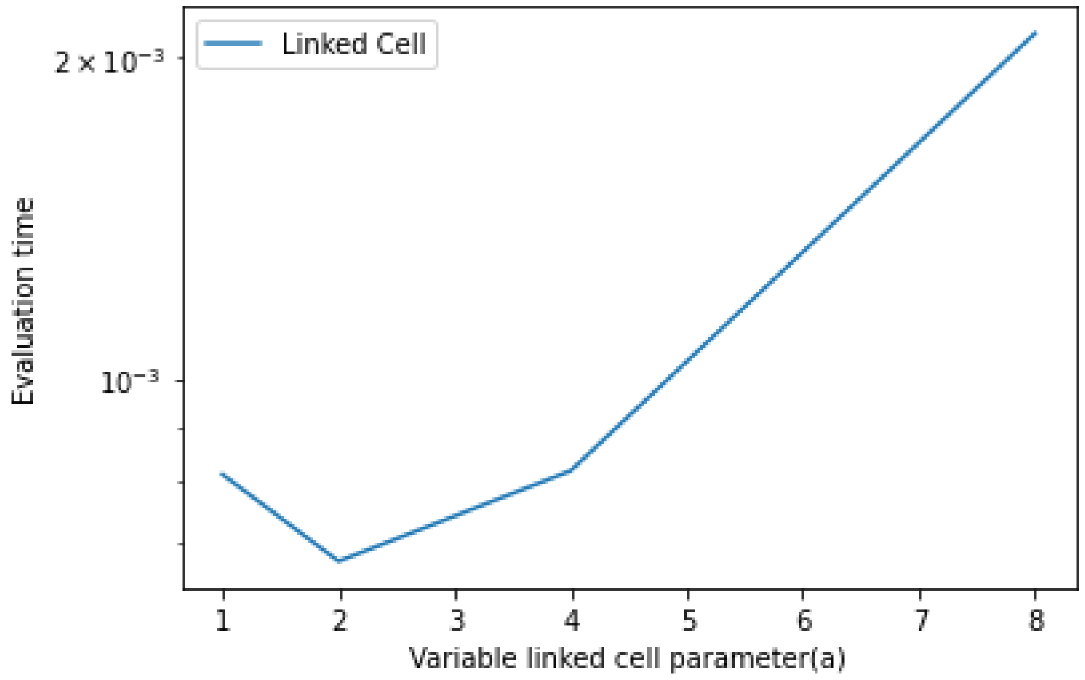
Fig 1: For a = 1	**Fig 2: For a = 8**
	

Let us plot the Evaluation time with respect to a (same as before), but with $N = 500$ particles and $r_c = 0.2$. Then we do the same with $N = 25$ particles.

[16]:
```
N = 500
r_c = 0.2
A = [1, 2, 4, 8]
n_instances = len(A)
time_linked_cell = np.zeros(n_instances, dtype=np.float)
domain = 1.0
list_particles = get_list_particles(N)
direct_potential = cell_1.direct_potential_all_particles(list_particles)
for idx, a in enumerate(A):
    cell_1.set_potential_zero(list_particles)
    delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
    list_cells = cell_1.get_list_cell(r_c, delta_coordinate, domain=domain, a=a)
    cell_1.assign_particle_to_cell(list_particles, list_cells, r_c, a=a)
    start_lc = time.time()
    cell_1.calculate_potential_linked_cell(list_cells, list_particles)
    end_lc = time.time()
    linked_cell_potential = cell_1.extract_linked_cell_potential(list_particles)
    time_linked_cell[idx] = end_lc - start_lc
plt.plot(A, time_linked_cell, label='Linked Cell')
plt.legend()
plt.yscale('log')
plt.xlabel('Variable linked cell parameter(a)')
plt.ylabel('Evaluation time')
plt.show()
```



[17]:
```
N = 25
r_c = 0.2
A = [1, 2, 4, 8]
n_instances = len(A)
time_linked_cell = np.zeros(n_instances, dtype=np.float)
domain = 1.0
list_particles = get_list_particles(N)
direct_potential = cell_1.direct_potential_all_particles(list_particles)
for idx, a in enumerate(A):
    cell_1.set_potential_zero(list_particles)
    delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
    list_cells = cell_1.get_list_cell(r_c, delta_coordinate, domain=domain, a=a)
    cell_1.assign_particle_to_cell(list_particles, list_cells, r_c, a=a)
    start_lc = time.time()
    cell_1.calculate_potential_linked_cell(list_cells, list_particles)
    end_lc = time.time()
    linked_cell_potential = cell_1.extract_linked_cell_potential(list_particles)
    time_linked_cell[idx] = end_lc - start_lc
plt.plot(A, time_linked_cell, label='Linked Cell')
plt.legend()
plt.yscale('log')
plt.xlabel('Variable linked cell parameter(a)')
plt.ylabel('Evaluation time')
plt.show()
```



1.3.11 Question:

[2 points]

We observe a different behavior of the evaluation time. Explain the results obtained. (Hint: There are two competing factors. What are those?)

Solution The two competing factors are;

1. Number of potential (force) evaluations/ Particle density (The 'number of potential/force' evaluations within the cut-off radius increase as particle density increases)
2. Number of cell traversals/ Linked cell parameter a (The 'number of cells' that needs to be traversed increases when a increases while keeping cutoff radius r_c constant.)

In the first graph where the total number of particles is 500, with varying variable linked cell parameter a, the evaluation time decreases with an increase in the parameter as we are considering the potentials of the particles that are more strictly within the cutoff radius and reducing the number of particles that are considered outside the cutoff radius of the particle (of interest), thereby reducing the evaluation time. The dominating factor for evaluation time here is the number of potential (force) calculations we need to do. But, as we increase a, we start to see that cell traversal starts to dominate slowly after $a = 4$.

However, in the second graph where the total number of particles is 25, with increasing a, we see an initial dip in the evaluation time followed by a continuous increase. This is because, with a very low number of total particles, the number of particles per cell will be very low. At larger Linked cell parameter a the number of cells would be very high and there would be more and more empty cells that we would have to traverse, but the number of potential evaluations wouldn't decrease very much since the particle density is low. So in this case the cell traversal time would dominate and the evaluation time would increase with Linked cell parameter a, as opposed to the N=500 case.

Fig 1: For N = 25, a = 1

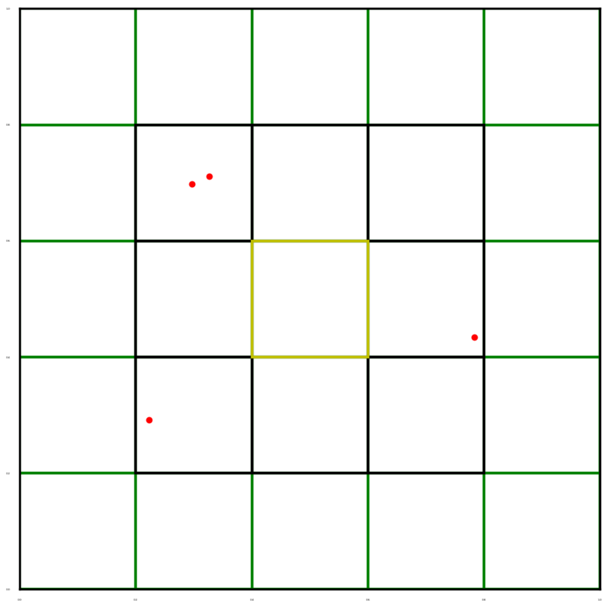


Fig 2: For N = 25, a = 2

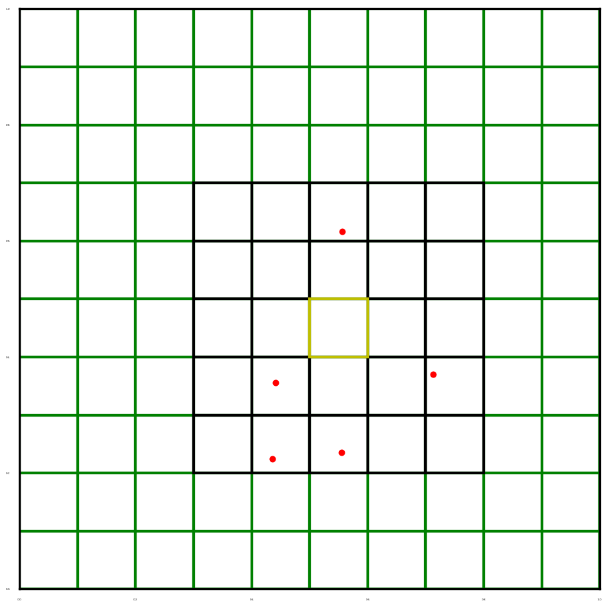


Fig 3: For N = 25, a = 8

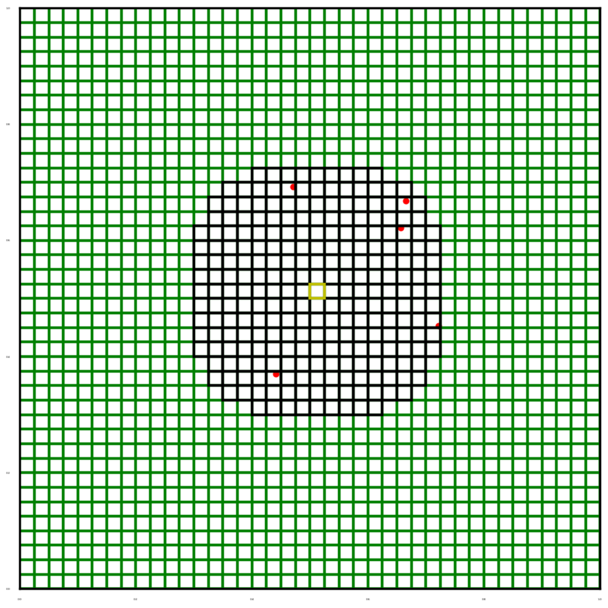


Fig 4: For N = 500, a = 1

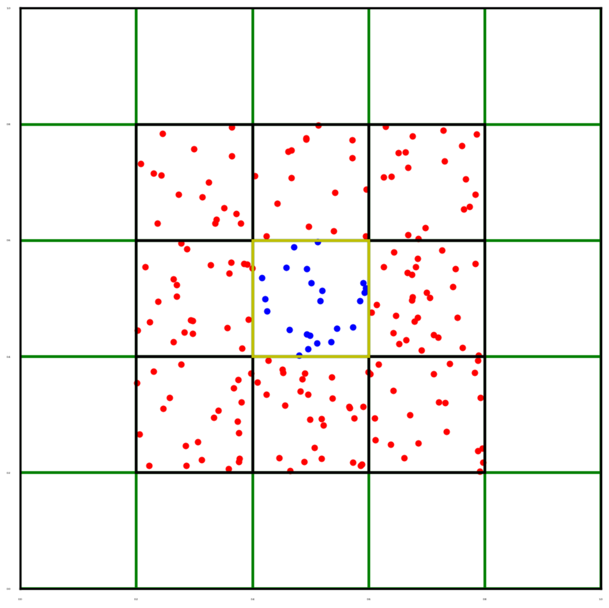


Fig 5: For N = 500, a = 4

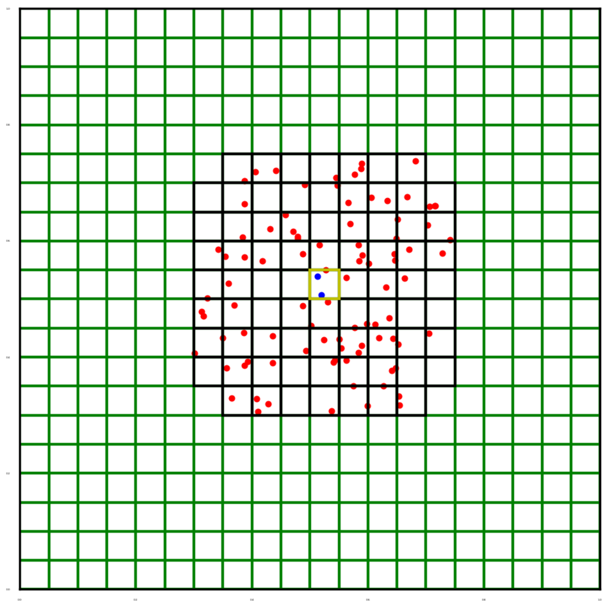
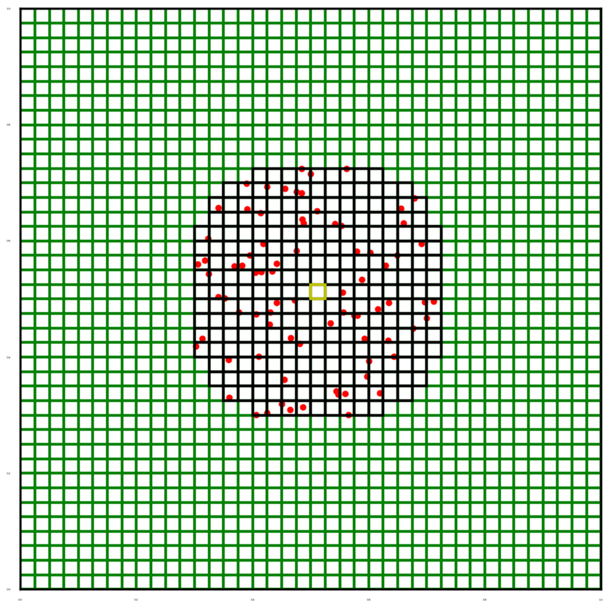


Fig 6: For N = 500, a = 8



1.4 Memory Access

In the implementation of class **Cell_1**, we stored the indices of the particles that are inside the cell. In the computer memory those particles may not be close to each other. While calculating the potential, we access particles inside a cell one at a time. With our current implementation, our access to memory is not continuous.

1.4.1 Question

[1 point]

What are the advantages of having contiguous memory access (avoiding jumps)?

Solution If the cell data is stored in a row major form, then when we access one cell, the contigious memory locations ie. the data corresponding to the same row will be loaded to the cache memory (provided cache size is appropriate) from the main memory. Then when we need the data of the next cell it will already be available in the cache memory instead of the main memory, and we can reduce the accesses to main memory during every cell traversal. Since cache reads are much faster than main memory reads, this reduces the overall evaluation time.

One of the solutions of this is to directly store the list of particles inside the object cell instead of storing particle indices inside the cell. So, we do not need a separate variable **list_particles**. Firstly, we will create **list_cells** in the same way as before, then create particles and directly add them to the required cell. The function that does that is already implemented inside file *cell_1.py*. Read and understand the function **create_assign_particle_to_cell**

1.4.2 Task 5: Calculate potential

[5 points]

Open the file *cell_2.py* and implement the functions to calculate the potential as described below:

5.1 [2 points]

First member function of class **Cell_2** that we need to implement is **p2p_self**. This function calculates the potential on all particle inside a cell due to other particles inside the same cell. The potential calculated is added in the variable **phi** of particle object.

5.2 [2 points]

The second function is **p2p_neigbor_cells**. This function calculates the potential on all particle inside a cell due to particles in the neighor cells (cells index given by the variable neighbor_cell_index). The potential calculated is added in the variable **phi** of particle object. This function takes **list_cells** as argument.

5.3 [1 point]

Implement the function **calculate_potential**. This function calls the functions implemented in **5.1** and **5.2** and calculates the total potential of particles inside the cell. Parameters are same as **p2p_neigbor_cells**.

[18]:
```
N, r_c, a = 100, 0.2, 5
delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
list_cells2 = cell_2.get_list_cell(r_c, delta_coordinate, domain=domain, a=a)
cell_2.create_assign_particle_to_cell(N, list_cells2, r_c, domain=1, a=a)
```

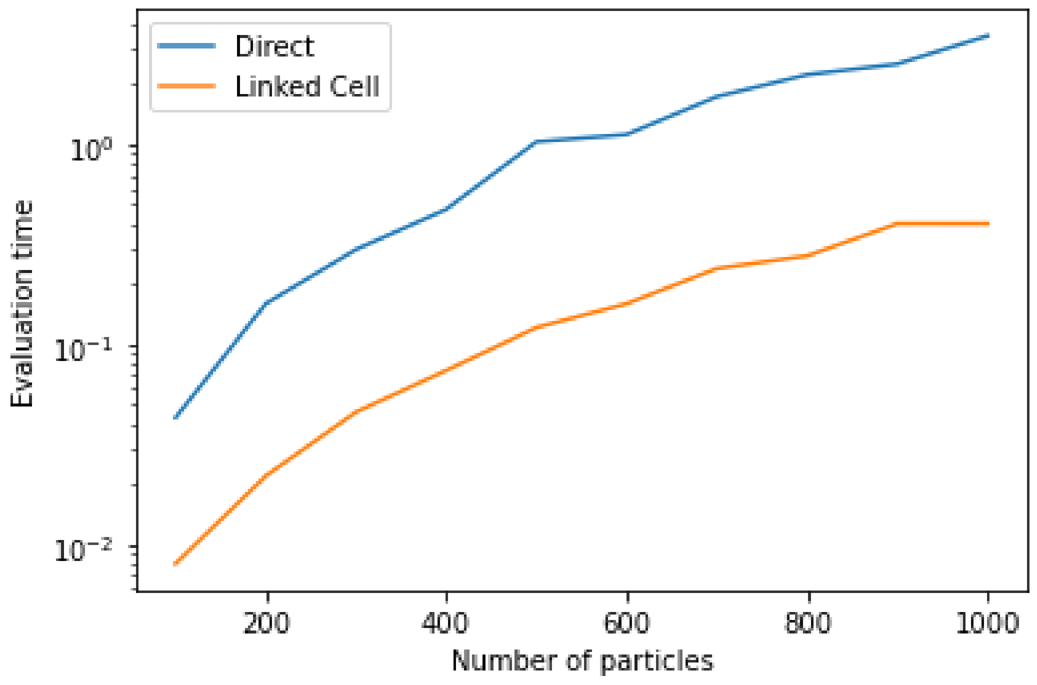
[19]:
```
direct_potential = cell_2.direct_interaction_v2(N, list_cells2)
cell_2.calculate_linked_cell_potential(list_cells2)
```

```
linked_cell_potential = cell_2.extract_linked_cell_potential(N, list_cells2)
```

1.4.3 Evaluation time scaling

Let us fix the number of particles $N = 1000$ and variable linked cell term $a = 8$. We analyse the error with respect cut-off radius and evaluation time with respect to the cut-off radius.

[20]:
```
num_particles = [np.int(100 * i) for i in range(1,11)]
n_instances = len(num_particles)
time_linked_cell = np.zeros(n_instances, dtype=np.float)
time_direct = np.zeros(n_instances, dtype=np.float)
r_c, a, domain = 0.2, 8, 1.0
for idx, N in enumerate(num_particles):
    delta_coordinate = utils.get_successor_neighbor_delta_coordinate(a=a)
    list_cells2 = cell_2.get_list_cell(r_c, delta_coordinate, domain=domain,␣
 ↪a=a)
    cell_2.create_assign_particle_to_cell(N, list_cells2, r_c, domain=1, a=a)
    start_lc = time.time()
    cell_2.calculate_linked_cell_potential(list_cells2)
    end_lc = time.time()
    linked_cell_potential = cell_2.extract_linked_cell_potential(N, list_cells2)
    time_linked_cell[idx] = end_lc - start_lc
    start_direct = time.time()
    direct_potential = cell_2.direct_interaction_v2(N, list_cells2)
    end_direct = time.time()
    time_direct[idx] = end_direct - start_direct
plt.plot(num_particles, time_direct, label='Direct')
plt.plot(num_particles, time_linked_cell, label='Linked Cell')
plt.legend()
plt.yscale('log')
plt.xlabel('Number of particles')
plt.ylabel('Evaluation time')
plt.show()
```



1.4.4 Question

[1 point]

What are the disadvantages of this **Cell_2** implementation? (Hint: Think what will happen if during simulation some particles move from one cell to another)

Solution In Cell_2 implementation, each cell holds the particle object lists instead of particle index list which is a list of integers. Therefore when a particle leaves the cell, the entire particle object has to be copied to the new cell. Since particle objects have larger memory size than integers, this leads to an increase in time spent in copying and deleting particle objects and would take up a lot of memory as well.