TEK5010

Fredrik Jaibeer Mahal Nordeng

September 2020

https://github.com/FredrikNM/Multi-Agent-Systems/tree/main/Random%20Agents

1 Abstract

In this paper I will make a environment where we implement a multi-agent system (MAS). We will see that, even tho we make very little hard coding in how they should move and interact, they still can be pretty efficient in trying to solve tasks we create.

2 Introduction

We are asked to create a square area where agents move around solving tasks they come over. The variables we consider here are :

Area of the environment (A) : How big/small (we have 1000*1000 in all our simulations)
Tasks (T) : How many of them
Task radius (Tr) : If agents inside this distance from task they will move towards it
Task capacity (Tc) : Agents needed to solve a task
Agents (R) : How many agents
Agents velocity (Rv) : Since we are iterating over time steps, this is the distance an agent can cover per iteration
Agents communtication radius (Rd) : Used for signaling task found
Time agents follows a signal (Rt) : In case task solved before an agent reaches signal, we use this to set agents back in to search modus
Call Off : Agents working can, instead of relying on Rt, send signal saying that the task is solved

3 One agent scenario

First we will implement task radius to 50 (as in meter/foot or what you want to think of it as), one agent moving around randomly, solving a task every time it is in the task radius. It moves around at a speed of 25 (meter/foot/etc) per iteration.

We are asked for a good model for moving around randomly. Lets consider the case of one agent. This is actually purely a definition of what moving randomly is. If we are to consider completely random, we are only left with a choice, a brownian motion at every time step. If we introduce memory, we can alternatively keep the direction until something is hit, task or a wall. Then our agent might only be steered by the shape of the walls, if we don't explicitly tell it to move randomly when hitting a wall, or it is done with a task. This is random in the sense that it depends on where you initialize your agent. As we see in the heatmap Fig. 15 it can actualy get stuck in a specific shape. We avoid this by telling the agent to move randomly after finishing a task or hitting a wall Fig. 15 .

The more you take away from a brownian motion the more systematically you can tell your agent to move. So lets rather see this from the perspective of how systematically can we actually move? This is very hard question. So lets imagine that our agent can not see, but it has memory of where it has been. It is also aware of it only being one task available to solve. If our agent do not have a map of the environment, we first realise that the agents vision in some sens is the radius of a task. Our agent should then ideally move in a bigger and bigger area around its origin, while the new vision is next to what it has seen before but never overlapping. One example would be a spiral, but with what shape it grows its area of seen space which is most efficient in an arbitrary shaped space is for the reader to look in to if they want.

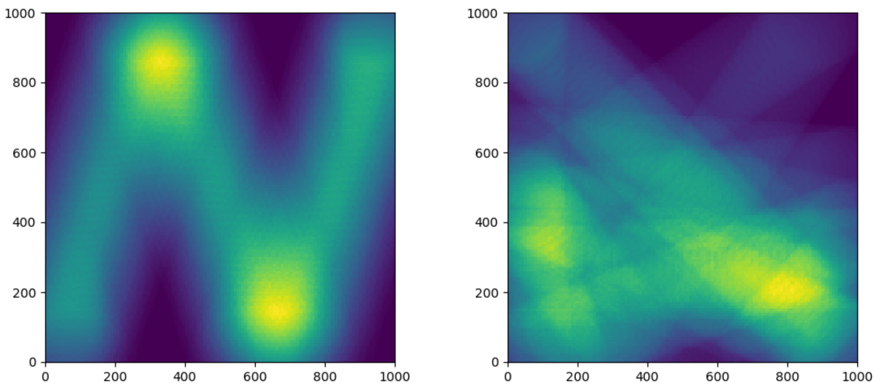

Figure 1: Left picture is agent steered by wall, getting stuck in a N shape. Right picture the agent move randomly after hitting a wall. In both pictures the area of heat is the radius of a task, since that is in a sense our agents vision area

Next thing we are asked is to plot how many tasks the agent finish on average at each time step. We also get the question if this a good measure in this search and task allocation STA problem.

To be fair, if our agent is moving around in a very large area, with only one task available do to at a time, this measure might not be very good. If we are moving around not as a brownian motion, but with some kind of memory or something else setting the direction, we might want to know how much time the agent spend in the same area. Because if it is going around in a little circle, it is very bad, but if it on the other hand spend very little time in the same places, it is actually doing a very good search job event tho it might not solve any task because the environment is to big for it to find it. So my proposal for assessing how good an agent is doing is time spent in the same area. Worst case this measure could lead to creating agents that avoid spots it has been in and at the same time avoid tasks. In our case this would never happen as they are obligated to work when a task is found. So lets then imagine a agent that knows where tasks are spawned, and by that manages to avoid the task radius. We have then created the inverse of a perfect agent. In multi-agent systems this can be used for shepherding if agents are programmed to avoid collisions, since the inverse of an perfect agent occupy the spaces where tasks are not spawned.

4 Multi-agent scenario

From figure we see that after around 1000 iterations we usually get into a steady-state where the performance stay almost the same. Even tho as we will se later we might not even reach it after 4000 iterations. If we end up in steady it could still be a possibility that our system have several steady-states it could end up in. From the N in Fig. 15 we see how this could happen. To get a good measure of performance, for different amount of agents and task available at each time, we then would have to simulate each scenario such that we can take advantage of the Central Limit Theorem, which has a rule of thumb of n > 30. ref modern mathematical statistics with applications. The cases we see on with no communication among the agents could be considered the "random" benchmark for the STA problem, where we have touched on what really random. We will in the rest of our test use agents that keep their direction, and gets a new random direction when hitting wall, other agents, are finished with a task, or a signal they followed has been called off.

Several different scenarios have been simulated, and to make them comparable I have chosen to see it as an average off task done compared to how many task that is available.

$$\frac{Average\,task\,finished}{Available\,tasks} \tag{1}$$

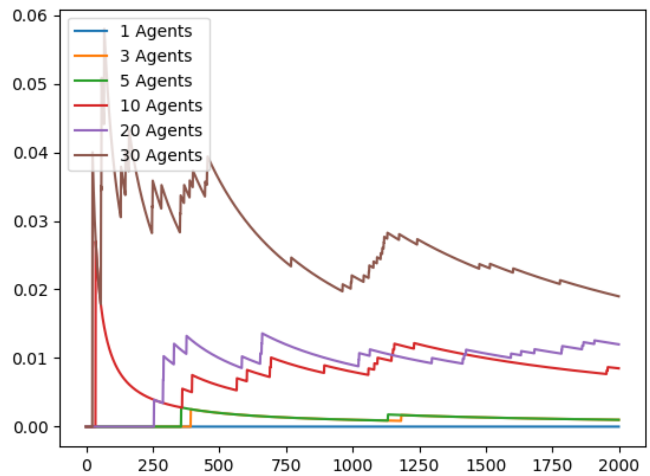

Figure 2: One agent needed to finish a task, and one task available at each time.

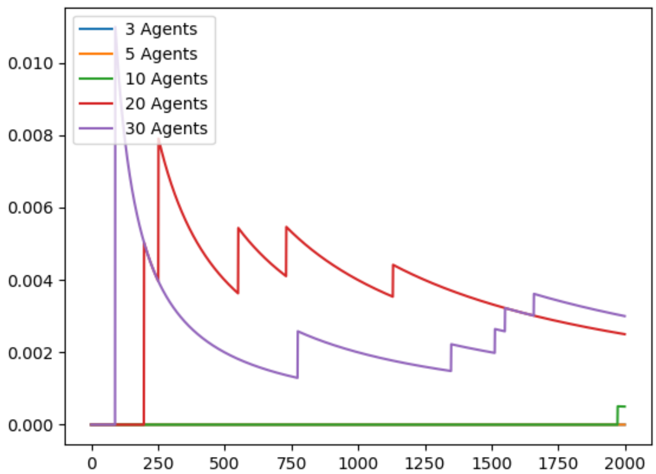

Figure 3: Three agent needed to finish a task, and one task available at each time. This will be our standard to check up against. Since more agents are need at each task we see the efficiency goes down from when only one agent was needed.

Lets see if adding more task will make it easier for the agents to find them.

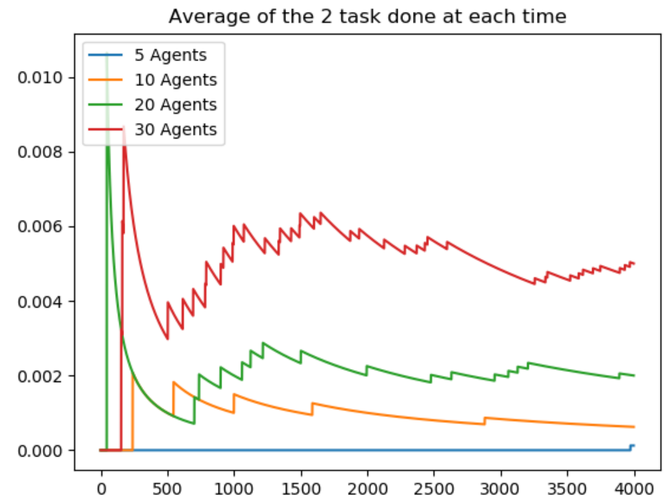

Figure 4: As we see efficiency goes a little bit up, but I also removed the scenario with three agents because it would be possible for 2 of the agents stuck at one of the task, with the last one waiting for help at the other task.

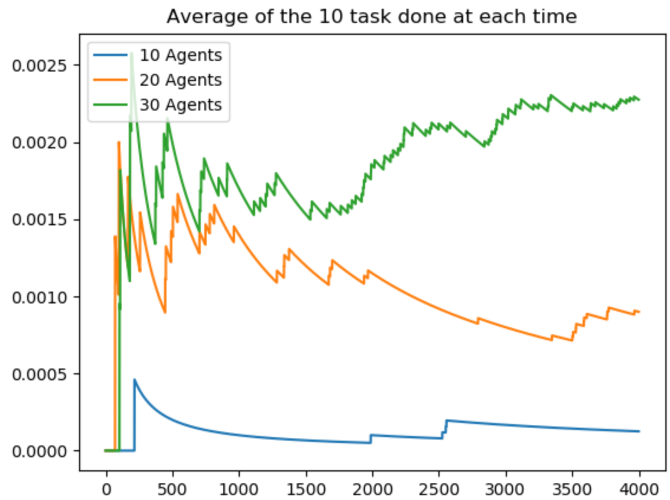

Figure 5: The efficiency is starting to fall as a consequence of agents waiting for help. Surprisingly the simulation with ten agents still have free agents at 2500 iterations.

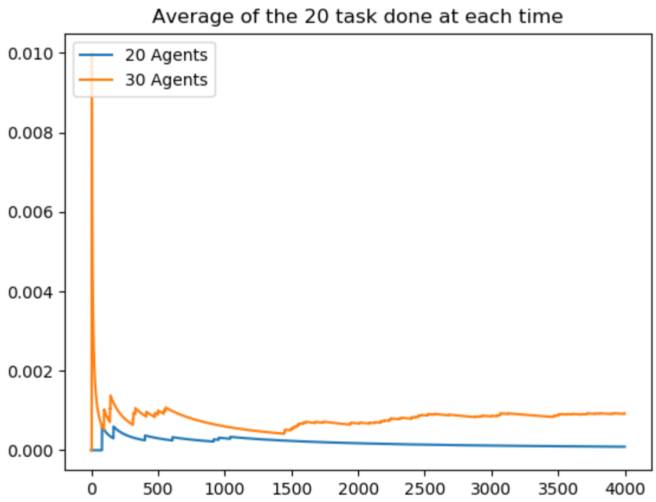

Figure 6: Here we see the waiting time for agents to come help each other become so large that even 20 and 30 agents almost gets nothing done. To many of them find different tasks.

For the last part of our assignment we implement communication between the agents, by sending a signal when they find a task. They send the signal only once and we simulate it with different radius which we then compare to simulation where they use call off. Call off is a new signal telling agents that has not reached the task before it is done that they can start searching for new tasks again. We have 30 agents,

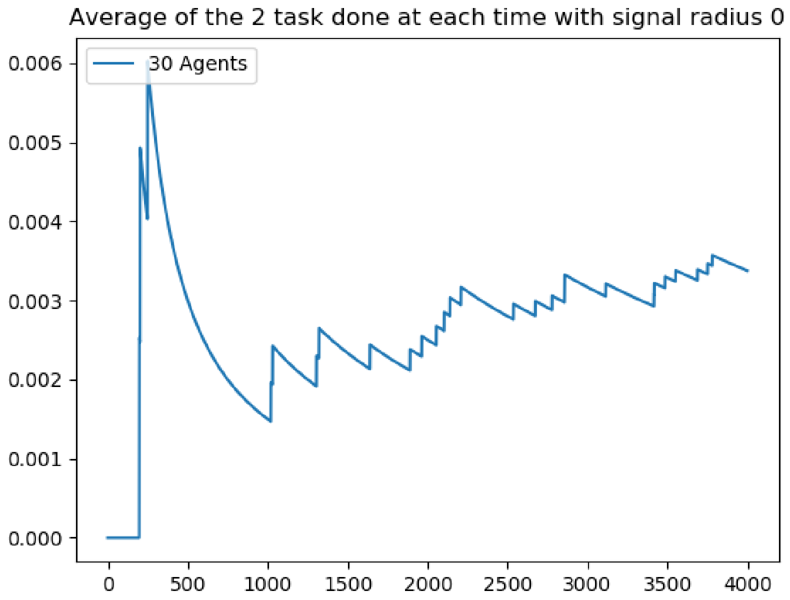

Figure 7: 30 agents with no signal and 2 tasks.

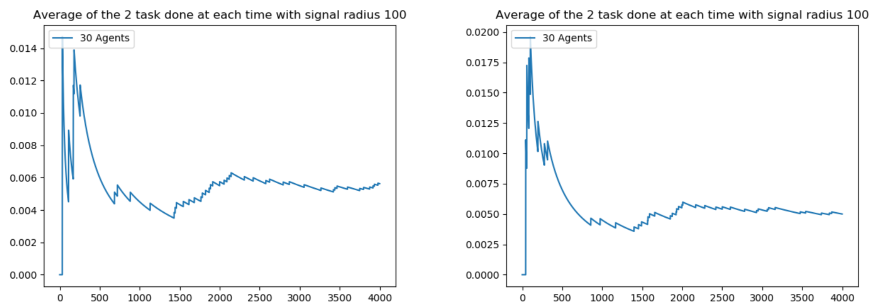

Figure 8: Left picture signal with out call off, right with call off.

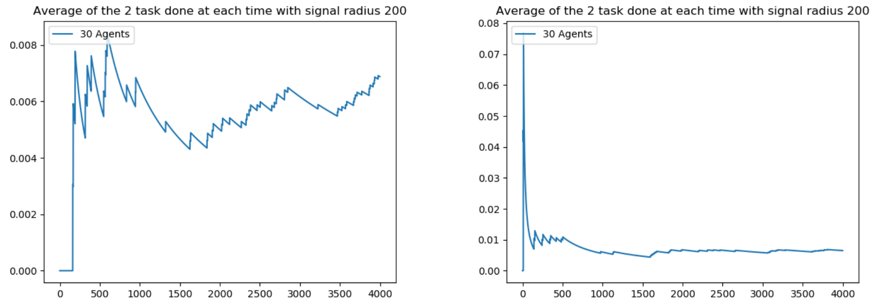

Figure 9: Left picture signal with out call off, right with call off.

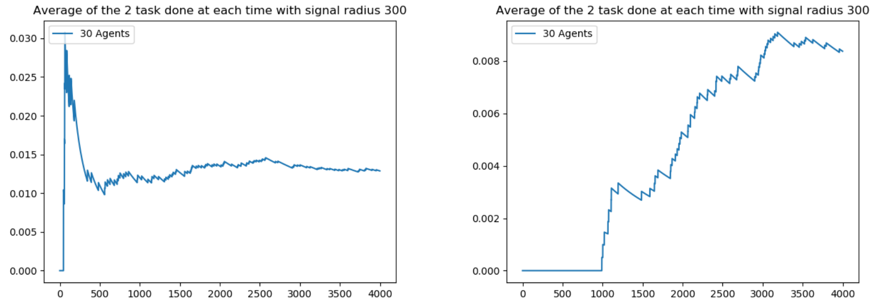

Figure 10: Left picture signal with out call off, right with call off. Here the call off simulation did not even reach steady state after 4000 iterations.

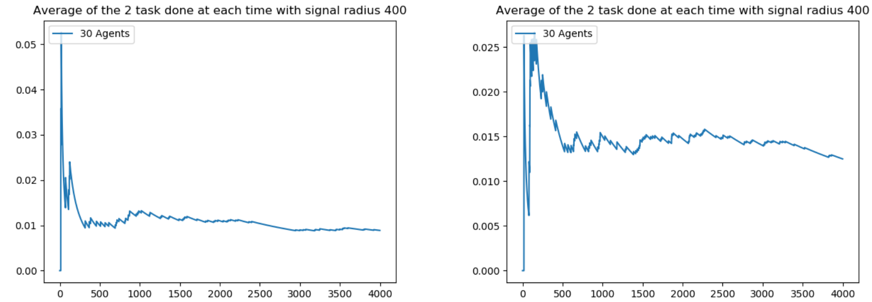

Figure 11: Left picture signal with out call off, right with call off. This is where we get our best result.

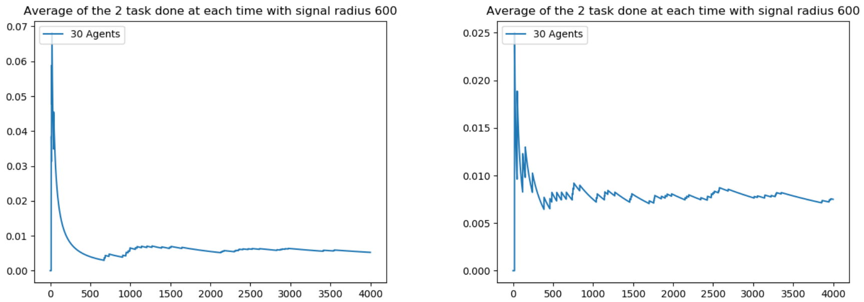

Figure 12: Left picture signal with out call off, right with call off. Our efficiency is starting to go down, due to the signal radius being so large that it is a nuisance and prevent agents from searching. The end up more clustered.

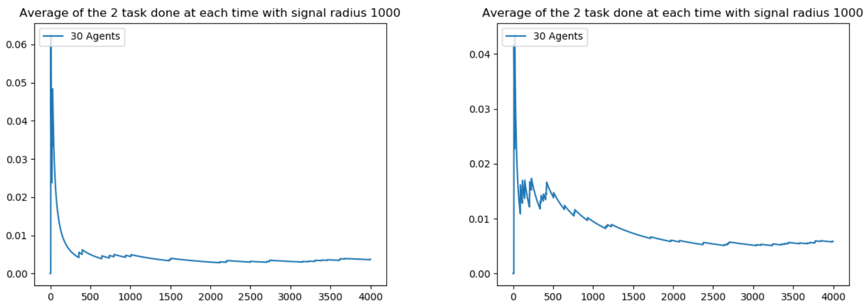

Figure 13: Left picture signal with out call off, right with call off.

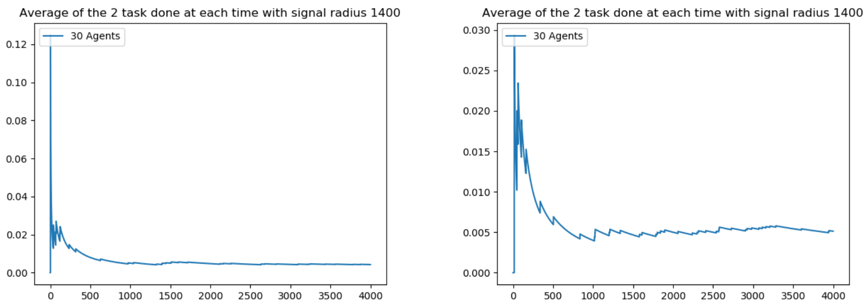

Figure 14: Left picture signal with out call off, right with call off. Here we see how bad it can get. I strongly suggest running the program to see it.

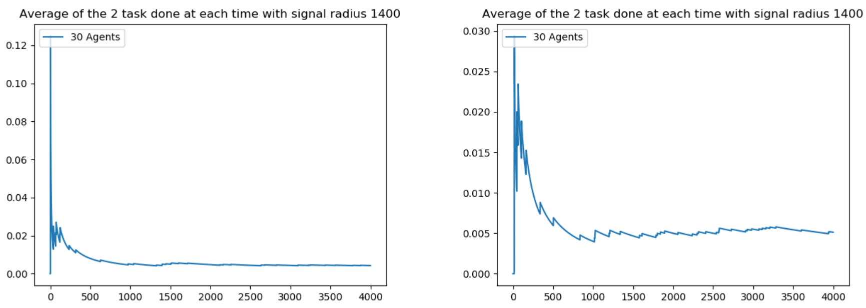

Figure 15: Left picture signal with out call off, right with call off. Here we see how bad it can get. I strongly suggest running the program to see it.

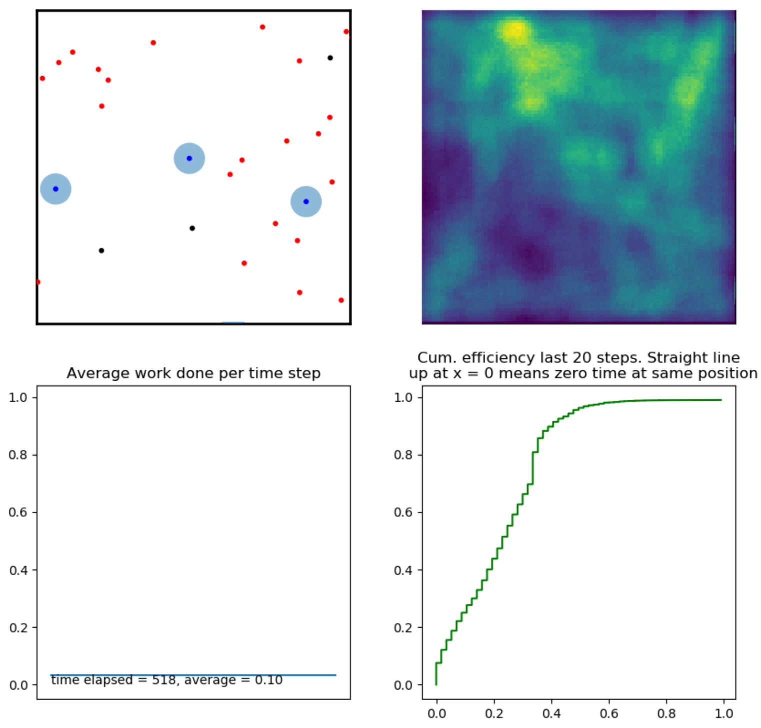

Figure 16: Example of a simulation and how it can look. The efficiency plot at bottom right is average time spent in same area last 20 time steps. Straight line up at 0 would mean that none of the agents have been at the same place again, individually.

A Code for simulating

```


'''
First I would just run it as it is, then try to change the VARIABLES TO CHANGE to get a grip
of how it is working. Some of the implementation could have been moved in to the agents class
and I could also created a task class, to make it a bit cleaner. Also some of the plotting part could
probably been made a lot better. As it stands, it should be easier to go trough and see what is
happening in the code, and it should be fairly easy to implement new stuff or take ides of how
this can be done. Like utility in an agent, maybe they get energy from working at a task,
new ways of moving, and probably other things!
Enjoy
'''


import copy
from agent import Agent
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation, writers
from scipy.spatial.distance import pdist, squareform, cdist
from heat_map_func import intensity_circle_plot
import matplotlib.gridspec as gridspec






########## VARIABLES TO CHANGE ##########

n_agents = 10   # How many agents to implement
agent_radius = 5  # Agents radius. The bigger the more likely they are to crash (Note that the points/agents in the animation is not adjusted correctly to their size)
steps_per_timeunit = 25   # Steps per counting time unit
x_y_walls = [[0.001, 1000], [0.001, 1000]]; X, Y = 0, 1   # [[x_min,x_max], [y_min, y_max]];  Coordinates index
task_radius = 50    # Task radius
task_numbers = 2  # Task at any timesteps
task_worktime = 35    # Time it takes for completing a task. If you want to scale it, you can say linear vs agents = True
task_worktime_linear_vs_agents = True   # Meaning if work time is divided by agents working on the task
agents_needed_for_a_task = 3  # How many agents that is need for a task to be completed
agents_max_waiting_time = 0 # How long will an agent wait to get help for a task found
signal_radius = 400   # Radius of signal sent out when a task is found
signal_search_time = 60   # Time steps agents follow signals
signal_call_off = True
random_bouncing_walls = True  # Bounce of the walls at a random direction for True, or False for bouncing according to angle agent hit the wall
movement = "Brownian"   # It is implemented two types of movements.
```

```
Random direction at each time step, or just a straight line
# movement = "Straight"



########## DIFFERENT PLOTS TO CHOOSE ##########

# Set the plots you want to see to True, and the rest to False
implement_agentplot = True
implement_average_workplot = True
# Cumulative density of average time spent in a radius of where agents been before. A straight line up at x = 0
# would be an agent always exploring new area of the map in a given timeframe which is defined by efficiency_last_n_steps
implement_efficiencyplot = True
if implement_efficiencyplot == True:
  efficiency_last_n_steps = 20
# Heatmap is currently a bit slow. Can be speeded up by making discretization smaller, which makes the plot uglier.
# Radius is the size of task radius
implement_heatmap = True
if implement_heatmap == True:
  discretization = 20
# Save as gif
save = True
if save == True:
  filename = "agent_c"
  frames = 200


########## NO ANIMATION, JUST SIMULATE TO SEE AVERAGE TASK COMPLETED ##########
# Set all above plots to False to see task completed as an function of how many agents we have
n_different_agents = [30]
simulation_time = 4000





def spawn_task(x_y_walls):
  # Creat new task at a random position
  return([np.random.uniform(x_y_walls[0][0], x_y_walls[0][1], 1)[0], np.random.uniform(x_y_walls[1][0], x_y_walls[1][1], 1)[0]])


def brownian_movement(n_agents):
  # Brownian motion in 2-dimensions
  real_imag = lambda x: np.array([x.real, x.imag])  # Helper function to extract real and imaginary part in next step
  direction = np.asarray([real_imag(np.exp(1j*(np.random.uniform(0, 2*np.pi) + 2*np.pi))) for i in range(n_agents)])
  return(direction)


def straight_line_movement(n_agents):
  # Moving in same direction as before
  direction = np.asarray([agents["A"+str(i)].velocity for i in range(n_agents)])
  return(direction)
```

```python



def simulate_movement(agents, agent_radius, x_y_walls, steps_per_timeunit, movement_function, \
  task_numbers, task_radius, agents_needed_for_a_task, random_bouncing_walls = False):
  # Simulate one step of movements for all agents, including bouncing of other agents and wall
  # Note that collision and bouncing in other agents/wall is check simultaneously for all agents, just preventing the most
  # imminent impact. This can lead to agents turning away from a collison, in to another that we dont check for,
  # and will not be check for. So for now it is just ignored
  global work_matrix, signal_matrix, signal_reciver, tasks_duration, task_completed, agents_position, tasks

  # Initializing tasks
  if 'tasks' not in globals():
    tasks = []
    while len(tasks) < task_numbers:
      tasks.append(spawn_task(x_y_walls))

  # Finding direction to move for each agent
  direction = movement_function(len(agents))



  # Checking for signals
  if len(np.where(signal_matrix == 1)[0]) > 0:
    # Finding distances for each agent to signals from working agents
    signaldist = cdist(agents_position, agents_position[np.unique(np.where(signal_matrix == 1)[0])])
    # If several signals, choose the one with closest distance
    signals_closest = np.argmin(signaldist, axis=1)
    # First now we are actually checking that distance of the agents is within the signals radius
    agent_in_signal_dist = np.where(signaldist[np.arange(len(signaldist)), signals_closest] < signal_radius)[0]
    # Removing the agent/agents that is already working and are sending the signal from the set
    agent_in_signal_dist = np.setdiff1d(agent_in_signal_dist, np.unique(np.where(signal_matrix == 1)[0]))
    # Saving agents that have recived a signal, and from whom
    agent_sending_signal = np.unique(np.where(signal_matrix == 1)[0])
    for n in range(len(agent_sending_signal)):
      signal_reciver[agent_sending_signal[n]] = list(set(np.where(signals_closest == n)[0]).intersection(set(agent_in_signal_dist)))

    if len(agent_in_signal_dist) > 0:
      # Storing signals that reached the agents
      temp_signal = [agents_position[np.unique(np.where(signal_matrix == 1)[0])][signals_closest[n]] for n in agent_in_signal_dist]
      # Calculating the vector from signal to agent
      direction_to_signal = temp_signal - agents_position[agent_in_signal_dist]
      # Normalizing (can cause owerflow) it so it becomes a direction for each step
```

```
      direction[agent_in_signal_dist] = direction_to_signal/np.absolute(direction_to_signal).sum(axis=1, keepdims=True)
      for n in agent_in_signal_dist:
        agents["A"+str(n)].recived_signal(signal_search_time*steps_per_timeunit)

  # Agents start working. Work matrix col_k corresponds to task_k, while row_i is agent_i
  for k in range(work_matrix.shape[1]):

    # Index of agents working at task k
    working_idx = np.where(work_matrix[:,k] != 0)[0]

    # Checking if enough agents is in the working at the task
    if len(working_idx) >= agents_needed_for_a_task:
      if tasks_duration[k] > 0:
        if task_worktime_linear_vs_agents == False:
          tasks_duration[k] -= 1
        # Task duratation as a linear function for how many working on it
        else:
          tasks_duration[k] -= len(working_idx)
      # If task duration is completed the agents are set free to roam again, and a new task is spawned
      else:
        # Setting new random direction
        direction[working_idx] = brownian_movement(len(working_idx))
        tasks[k] = spawn_task(x_y_walls)
        # Resetting variables
        tasks_duration[k] = task_worktime
        for n in np.where(work_matrix[:,k] == 1)[0]:
          agents["A"+str(n)].work = False
          # Calling off signal
          if signal_call_off == True:
            if n in signal_reciver:
              for i in signal_reciver[n]:
                agents["A"+str(i)].signal = False
              signal_reciver.pop(n)
        work_matrix[:,k] = 0
        signal_matrix[:,k] = 0
        task_completed += 1

    # Checking agents max waiting time at work station, if work is not started
    elif np.sum(work_matrix[:,k]) > 0 and agents_max_waiting_time != 0:
      # Index of agents waited max at task k
      working_and_max = list(set(working_idx).intersection(np.where(agents_max_waiting_time_array == 0)[0]))
      if len(working_and_max) > 0:
        # Removing agent from work station
        work_matrix[:,k][working_and_max] = 0
        # Setting direction away from work station
        direction_away_from_work = agents_position[working_and_max] - tasks[k]
        direction[working_and_max] = direction_away_from_work/np.absolute(direction_away_from_work).sum(axis = 1, keepdims=True)
        # The agents waiting time is restored
        agents_max_waiting_time_array[working_and_max] = agents_max_waiting_time
      else:
```

```
      agents_max_waiting_time_array[working_idx] -= 1


# Loop so we calculate collison at each step, instead of doing it for each unit of time.
for k in range(steps_per_timeunit):

  working_agents = np.where(work_matrix != 0)[0]
  # Calculate what would be the new position before actually moving, to avoid crashes
  possibly_new_pos = np.asarray([np.add(agents["A"+str(n)].position, direction[n]) for n in range(len(agents))])

  # Finding distances between agents, and which of them that will collide in their possible_new_pos
  dist = squareform(pdist(possibly_new_pos))
  iarr, jarr = np.where(dist < 2 * agent_radius)
  k = iarr < jarr
  iarr, jarr = iarr[k], jarr[k]

  # Choose to go with a random direction after a crash. Meaning they can still crash, so this is the easy solution.
  # If you want to do this correct, you would have to iterate over very possible collision, checking their new
  # possible postition after avoiding crash, then checking for new crashes, over and over, until there is no crashes.
  # If lots of agents, this would be very computational heavy, but it could be implemented in a smart way which I will not go in to.
  for i, j in zip(iarr, jarr):
    direction[i] = brownian_movement(1)
    direction[j] = brownian_movement(1)
    # Stopp following signal if agent crashes
    agents["A"+str(i)].signal = False
    agents["A"+str(j)].signal = False

  # Checking if the possibly new position would make the agent crash in the wall
  hit_left_wall = possibly_new_pos[:, X] < agent_radius
  hit_right_wall = possibly_new_pos[:, X] > x_y_walls[0][1] - agent_radius
  hit_bottom_wall = possibly_new_pos[:, Y] < agent_radius
  hit_top_wall = possibly_new_pos[:, Y] > x_y_walls[1][1] - agent_radius

  # Stopp following signal if agents hits the wall
  if len(np.where(hit_left_wall | hit_right_wall)[0]):
    for n in np.where(hit_left_wall | hit_right_wall)[0]:
      agents["A"+str(n)].signal = False
  if len(np.where(hit_bottom_wall | hit_top_wall)[0]):
    for n in np.where(hit_bottom_wall | hit_top_wall)[0]:
      agents["A"+str(n)].signal = False


  if random_bouncing_walls == True:
    # Agents turn in a random direction when reaching the wall # Careful. We dont check if new direction is a crash. Agents is built to stop if so.
    if len(direction[hit_left_wall | hit_right_wall, X]) > 0:
      direction[hit_left_wall | hit_right_wall] = brownian_movement(len(direction[hit_left_wall | hit_right_wall, X]))
```

```
      if len(direction[hit_bottom_wall | hit_top_wall, Y]) > 0:
        direction[hit_bottom_wall | hit_top_wall] = brownian_movement(len(direction[hit_bottom_wall | hit_top_wall, Y]))

    # Turning velocity_x or velcoity_y around depending on which wall agent hits
    direction[hit_left_wall | hit_right_wall, X] *= -1
    direction[hit_bottom_wall | hit_top_wall, Y] *= -1

    # Update position
    agents_position = np.asarray([agents["A"+str(n)].update_position(direction[n], x_y_walls) for n in range(len(agents))])
    # Check if agent is within distance of a task
    if len(tasks) > 0:
      task_dist = squareform(pdist(np.vstack([tasks, agents_position])))[task_numbers:, :task_numbers]
      agent_i, task_j = np.where(task_dist < task_radius)
      # Assigning working agents to the work matrix
      work_matrix[agent_i, task_j] = 1
      for n in agent_i:
        agents["A"+str(n)].working()


  signal_matrix += work_matrix
  cumulative_task.append(task_completed)

  if number_of_plots != 0:
    if len(np.where(signal_matrix == 1)[0]) > 0:
      return(agents_position, tasks, np.where(signal_matrix == 1)[0])
    if len(tasks) > 0:
      return(agents_position, tasks)
    else:
      return(agents_position)











########## ANIMATIONS ##########


# Figuring out how many plots, and how to set it up
number_of_plots = np.sum([implement_heatmap, implement_agentplot, implement_average_workplot, implement_efficiencyplot])

# Average task completion
if number_of_plots == 0:

  for n in range(len(n_different_agents)):

    # Initialize Agents with random position and velocity
    agents = {"A"+str(i) : Agent(position = np.array([np.random.randint(x_y_walls[0][0]+agent_radius, x_y_walls[0][1]-
```

```
agent_radius, 1)[0],\
                          np.random.randint(x_y_walls[1][0]+agent_radius, x_y_walls[1][1]-agent_radius, 1)[0]]), \
                     velocity = np.exp(1j*(np.random.uniform(0, 2*np.pi) + 2*np.pi)),\
                     radius = agent_radius,
                     mass = 1) for i in range(n_different_agents[n])}



    # Setting up some variables need
    tasks_duration = np.zeros(task_numbers) + task_worktime   # Array for keeping track of how much time it is left before a task is finished
    work_matrix = np.zeros((n_different_agents[n], task_numbers))  # Matrix describing if an agent is working on a specific task
    signal_matrix = np.zeros((n_different_agents[n], task_numbers))
    signal_reciver = {}
    agents_max_waiting_time_array = np.zeros(n_different_agents[n]) + agents_max_waiting_time # Array for checking how long an agent has been waiting to get help at a task
    task_completed = 0
    cumulative_task = []
    efficiency = []
    for k in range(simulation_time):
      print(k)
      simulate_movement(agents, agent_radius, x_y_walls, steps_per_timeunit, brownian_movement, task_numbers, task_radius, agents_needed_for_a_task)
    plt.plot(( (task_worktime*np.array(cumulative_task))/((np.arange(len(cumulative_task))+1)*task_numbers)), label=str(n_different_agents[n])+" Agents")
  plt.title("Average of the "+str(task_numbers)+" task done at each time with signal radius "+str(signal_radius))
  plt.legend(loc='upper left')
  plt.show()


if number_of_plots > 0:

  # Initialize Agents with random position and velocity
  agents = {"A"+str(i) : Agent(position = np.array([np.random.randint(x_y_walls[0][0]+agent_radius, x_y_walls[0][1]-agent_radius, 1)[0],\
                          np.random.randint(x_y_walls[1][0]+agent_radius, x_y_walls[1][1]-agent_radius, 1)[0]]), \
                     velocity = np.exp(1j*(np.random.uniform(0, 2*np.pi) + 2*np.pi)),\
                     radius = agent_radius,
                     mass = 1) for i in range(n_agents)}



  # Setting up some variables need
  tasks_duration = np.zeros(task_numbers) + task_worktime   # Array for keeping track of how much time it is left before a task is finished
  work_matrix = np.zeros((n_agents, task_numbers))  # Matrix describing if an agent is working on a specific task
  signal_matrix = np.zeros((n_agents, task_numbers))
```

```python
signal_reciver = {}
agents_max_waiting_time_array = np.zeros(n_agents) + agents_max_waiting_time  # Array for checking how long an agent has been waiting to get help at a task
task_completed = 0
cumulative_task = []
if save == True:
  efficiency = [1]
else:
  efficiency = []

if number_of_plots > 2:
  grid = [2, 2]
else:
  grid = [1, number_of_plots]
DPI = 100
width, height = 500*grid[1], 500*grid[0]
fig = plt.figure(figsize=(width/DPI, height/DPI), dpi=DPI)
plotted = 1


# Agentplot
if implement_agentplot == True:
  sim_ax = fig.add_subplot(grid[0], grid[1], plotted, aspect='equal', autoscale_on=False)
  plotted += 1
  sim_ax.set_xticks([]); sim_ax.set_yticks([])
  for spine in sim_ax.spines.values():
      spine.set_linewidth(2)
  sim_ax.set(xlim=(x_y_walls[0][0], x_y_walls[0][1]), ylim=(x_y_walls[1][0], x_y_walls[1][1]))
  # Color 3 agents black to make it easier to follow some of the movements
  c = np.array(['black']*n_agents); c[3:] = 'red'
  # simulate 1 step, to initialize positions of agents
  sim1 = simulate_movement(agents, agent_radius, x_y_walls, 1, brownian_movement, task_numbers, task_radius, agents_needed_for_a_task)
  if type(sim1) == tuple:
    sim1 = sim1[0]
  agentplot = sim_ax.scatter(sim1[:, X], sim1[:, Y], c=c, s=agent_radius*2, cmap="jet")
  # Adding circles that will be used for task radius
  circles = [plt.Circle([0,0], 0, alpha=0.5) for k in range(task_numbers)]
  # Adding circles that will be used for signal radius
  circles.append([plt.Circle([0,0], 0, color='g', fill=False) for k in range(n_agents)])
  circles = np.hstack(circles)

# Heatmap
if implement_heatmap == True:
  heat_obj = intensity_circle_plot(task_radius, discretization, x_y_walls)
  heatmap_fig = fig.add_subplot(grid[0], grid[1], plotted, aspect='equal')
  plotted +=1
  heatmap_fig.set_xticks([]); heatmap_fig.set_yticks([])
  heatmap = heatmap_fig.pcolormesh(heat_obj.x_mesh, heat_obj.y_mesh, heat_obj.intensity(agents))

# Average work
```

```
if implement_average_workplot == True:
    average_work = fig.add_subplot(grid[0], grid[1], plotted, aspect='equal')
    plotted += 1
    line, = average_work.plot(np.arange(0, 1, 0.01), np.arange(0, 1, 0.01))
    label = average_work.text(0,0,'time elapsed = {:d}, average = {:.2f}'.format(1, 0))
    average_work.set_title('Average work done per time step')
    average_work.set_xticks([]);

# Efficiency
if implement_efficiencyplot == True:
    efficiency_plot = fig.add_subplot(grid[0], grid[1], plotted, aspect='equal')
    line2, = efficiency_plot.plot(np.arange(0, 1, 0.01), np.arange(0, 1, 0.01))
    title = efficiency_plot.set_title('Cum. efficiency last '+str(efficiency_last_n_steps)+\
                                      ' steps. Straight line\n  up at x = 0 means zero time at same position')


def init_anim():
    """Initialize the animation"""

    return_values = []

    if implement_agentplot == True:
        agentplot.set_offsets([])
        for n in range(len(circles)):
            circles[n].center = [0,0]
            circles[n].radius = 0
        [sim_ax.add_patch(circle_i) for circle_i in circles]
        return_values.append([(*circles), agentplot])

    if implement_average_workplot == True:
        return_values.append([line,label,])

    if implement_efficiencyplot == True:
        if save == True:
            return_values.append([line2, title,])
        else:
            return_values.append([line2])

    if len(return_values) > 0:
        return (*np.hstack(return_values)),
    else:
        return(return_values)



def animate(i, agents, agent_radius, x_y_walls, steps_per_timeunit, movement_function, \
    task_numbers, task_radius, agents_needed_for_a_task, random_bouncing_walls):
    """Advance the animation by one step and update the frame."""
    global efficiency

    return_values = []
    sim = simulate_movement(agents, agent_radius, x_y_walls, steps_per_timeunit, \
```

```
    movement_function, task_numbers, task_radius, agents_needed_for_a_task, random_bouncing_walls)

  if implement_agentplot == True:
    if len(sim) == 2:
      sim_agent, tasks = sim[0], sim[1]
      c = np.array(['black']*(len(sim_agent)+len(tasks))); c[3:] = 'red'; c[len(sim_agent):] = 'blue'
    if len(sim) == 3:
      sim_agent, tasks, signals = sim[0], sim[1], sim[2]
      c = np.array(['black']*(len(sim_agent)+len(tasks))); c[3:] = 'red'; c[len(sim_agent):] = 'blue'

    # Circle center adjust
    if len(sim) == 2 or len(sim) == 3:
      agentplot.set_offsets(np.vstack([sim_agent, tasks]))
      agentplot.set_color(c)
    else:
      agentplot.set_offsets(sim)

    if len(sim) == 2:
      for k in range(len(tasks)):
        circles[k].center = tasks[k]
        circles[k].radius = task_radius
      for k in range(len(sim_agent)):
        circles[len(tasks)+k].radius = 0

    if len(sim) == 3:
      for k in range(len(tasks)):
        circles[k].center = tasks[k]
        circles[k].radius = task_radius
      for k in range(len(sim_agent)):
        if k in signals:
          circles[len(tasks)+k].center = sim_agent[k]
          circles[len(tasks)+k].radius = signal_radius
        else:
          circles[len(tasks)+k].radius = 0

    return_values.append([(*circles), agentplot])

  if implement_heatmap == True:
    heatmap_fig.cla()
    heatmap = heatmap_fig.pcolormesh(heat_obj.x_mesh, heat_obj.y_mesh, heat_obj.intensity(agents))
    return_values.append(heatmap)

  if implement_average_workplot == True:
    line.set_ydata(( (task_worktime*np.array(cumulative_task))/(task_numbers*len(cumulative_task)) )[-1])  # update the data.
    label.set_text('time elapsed = {:d}, average = {:.2f}'.format(i, (np.array(cumulative_task)/len(cumulative_task))[-1]))
    return_values.append([line,label,])

  if implement_efficiencyplot == True:
    efficiency_plot.cla()
    if i > efficiency_last_n_steps:
      efficiency.append([agents["A"+str(n)].calculate_efficiency(steps_per_timeunit, efficiency_last_n_steps) for n in range(n_agents)])
      X2 = np.sort(np.hstack(efficiency)[-10000:])
      F2 = np.array(range(len(X2)))/float(len(X2))
      line2, = plt.plot(X2, F2, 'g-')
```

```
      if save == True:
        title = efficiency_plot.set_title('Cum. efficiency last '+str(efficiency_last_n_steps)+\
                              ' steps. Straight line\n  up at x = 0 means zero time at same position')
        return_values.append([line2,title])
      else:
        return_values.append([line2])


    if save == True:
      print(i)

    return (*np.hstack(return_values)),


  # Number of frames; set to None to run until explicitly quit.
  if save == False:
    frames = None

  if movement == "Brownian":
    anim = FuncAnimation(fig, animate, frames=frames, interval=20, blit=True, init_func=init_anim, \
      fargs =(agents, agent_radius, x_y_walls, steps_per_timeunit, brownian_movement, task_numbers, \
        task_radius, agents_needed_for_a_task, random_bouncing_walls))

  if movement == "Straight":
    anim = FuncAnimation(fig, animate, frames=frames, interval=20, blit=True, init_func=init_anim, \
      fargs =(agents, agent_radius, x_y_walls, steps_per_timeunit, straight_line_movement, task_numbers, \
        task_radius, agents_needed_for_a_task, random_bouncing_walls))


  if save == True:
    anim.save(filename+'.gif', dpi=80, writer='imagemagick')
  else:
    plt.show()
```

B Heatmap class

```


import numpy as np

class intensity_circle_plot:

  # Class to fix how a circle is discretize on a grid (how many gridpoints is the radius equal to)
  # We set the grid so we can have a "round" circle, meaning the grid-radius relation
  # gives us a gridspace that is odd numbered such that the circle will have a center.
  # This is made for the specific case of a rectangular environment  for x between (0, some width),
  # y between (0, some hight). The objects we want a radius around is sent to the intensity function which
  # gives a the intensity as a matrix. If you want to use it for other cases that I have done,
```

```python
  # you probably must modify the intensity function, since now it loops over agents called A0, A1, ect,
  # that has the x, y coordinates

  def __init__(self, radius, discretization, walls):
    self.grid_size = int(radius/discretization) # Descretiziation of mesh for heatmap
    if self.grid_size < 1:
      self.grid_size = 1  # In case our cirle is discretizate to 0
    self.walls = walls  # Walls of our environment
    self.grid_x_max = walls[0][1]
    self.grid_y_max = walls[1][1]

    def radius_grid(radius, grid_size):

      # Radius grid relation
      return( int(radius*2/grid_size)+1 )

    def e_or_o(number):

      # Even or odd number
      return(number % 2 == 0)


    # If the relation is even we want to fix our grid such that the space have odd number length and height
    while e_or_o(radius_grid(radius, self.grid_size)):

      delta_grid_size = self.grid_size/100
      grid_temp = self.grid_size - delta_grid_size

      # Check that we dont move to much. Remember we only want to add an extra slot to our grid
      while abs(radius_grid(radius, grid_temp) - radius_grid(radius, self.grid_size)) > 1:
        delta_grid_size = delta_grid_size/10
        grid_temp = self.grid_size - delta_grid_size

      self.grid_size = grid_temp

    self.n_points = radius_grid(radius, self.grid_size)
    self.radius_n_points = int(self.n_points/2)

    def circular_mesh(radius, grid_size, n_points):

      ''' Circle made out of radius and descretizized  '''
      x_grid = np.linspace(0, (radius*2), int(radius*2/grid_size)+1 )
      base_circle = np.zeros(( len(x_grid), len(x_grid) ))
      [base_circle.__setitem__((i, j), 1) for i in range(len(base_circle)) for j in range(len(base_circle)) \
      if np.sqrt((i-int(len(base_circle)/2))**2 + (j-int(len(base_circle)/2))**2) <= int(len(base_circle)/2)]
      return(base_circle)

    self.base_circle = circular_mesh(radius, self.grid_size, self.n_points)
    antall_x_punkter = int(self.grid_x_max/self.grid_size) + 1; antall_y_punkter = int(self.grid_y_max/self.grid_size) + 1
    x_grid = np.linspace(0, self.grid_x_max, antall_x_punkter); y_grid = np.linspace(0, self.grid_y_max, antall_y_punkter)
    self.x_mesh, self.y_mesh = np.meshgrid(x_grid, y_grid)
```

```
    self.width_min = 0 ; self.width_max = len(self.x_mesh)
    self.height_min = 0 ; self.height_max = len(self.y_mesh)
    self.new_zeros = np.zeros((self.width_max, self.height_max))


  def idx_to_circle(self, width_min, width_max, height_min, height_max, x, y):
    # Returns the indexes to the circle in the gridspace

    idx_w = np.arange(self.n_points)
    if x < (width_min + self.radius_n_points):
      idx_w = np.arange( abs( self.radius_n_points - x ), self.n_points )

    if x >= (width_max - (self.radius_n_points+1)):
      idx_w = np.arange(0 , self.n_points - (x - (width_max - (self.radius_n_points+1)))  )

    idx_h = np.arange(self.n_points)
    if y < (height_min + self.radius_n_points):
      idx_h = np.arange( abs( self.radius_n_points - y ), self.n_points )

    if y >= (height_max - (self.radius_n_points+1)):
      idx_h = np.arange(0 , self.n_points - (y - (height_max - (self.radius_n_points+1)))  )

    return(idx_w, idx_h)


  def intensity(self, agents):

    for k in range(len(agents)):
      # Modify either your data to fit this structure, or this line to fit your need
      x, y = agents["A"+str(k)].position
      # Ensuring that the x, y coordinates are inside of the grid, else we would get errors
      x, y = np.array([ np.min([ np.max([x, self.walls[0][0]]), self.walls[0][1]  ]) , \
                  np.min([ np.max([y, self.walls[1][0]]), self.walls[1][1]  ])    ])

      x = int(x/self.grid_size)
      y = int(y/self.grid_size)

      idx_w, idx_h = self.idx_to_circle(self.width_min, self.width_max, self.height_min, self.height_max, x, y)
      self.new_zeros[np.max([self.width_min, x-self.radius_n_points]):np.min([self.width_max,x+self.radius_n_points+1]), \
            np.max([self.height_min, y-self.radius_n_points]):np.min([self.height_max, y+self.radius_n_points+1])] += self.base_circle[idx_w,:][:,idx_h]

    return(self.new_zeros.T)
```

C Agent class

```

import numpy as np
import copy
from scipy.spatial.distance import pdist, squareform
```

```
class Agent:
  ''' Create an agent '''

  def __init__(self, position, velocity, radius, mass):

    self.position = np.array(position)
    self.velocity = np.array([velocity.real, velocity.imag])
    self.radius = radius
    self.mass = mass
    self.memory = [copy.copy(position)]
    self.signal = False
    self.work = False


  def update_position(self, velocity, min_max):

    if self.work == False:

      if self.signal == False:
        self.velocity = velocity
      if self.signal == True:
        if self.count == 0:
          self.velocity = velocity
        self.count += 1
        if self.count == self.timer:
          self.signal = False
    else:
      self.velocity = np.array([0, 0])

    self.position = self.position + self.velocity
    # Extra security to keep the agent inside of the environment if
     something accidentliy push the agent out
    self.position = np.array([ np.min([ np.max([self.position[0],
    min_max[0][0]]), min_max[0][1]  ]) , \
              np.min([ np.max([self.position[1], min_max[1][0]]),
    min_max[1][1]  ])    ])
    self.memory.append(copy.copy(self.position))
    return(self.position)


  def working(self):
    self.work = True
    self.signal = False


  def recived_signal(self, timer):
    self.signal = True
    self.timer = timer
    self.count = 0


  def calculate_efficiency(self, steps_per_timeunit, timeframe):

    if len(self.memory) > timeframe*steps_per_timeunit:
      epsilon = 0.0001
      step_distance = squareform(pdist(np.vstack(self.memory[-
    timeframe*steps_per_timeunit + (steps_per_timeunit-1)::
    steps_per_timeunit])))
      step = step_distance[np.triu_indices(timeframe, k = 1)]
      if len(np.where(step < (steps_per_timeunit-epsilon))[0]) > 0:
        return(len(np.where(step < (steps_per_timeunit-epsilon))
```

```
        [0])/len(step))
    else:
        return(0)
```