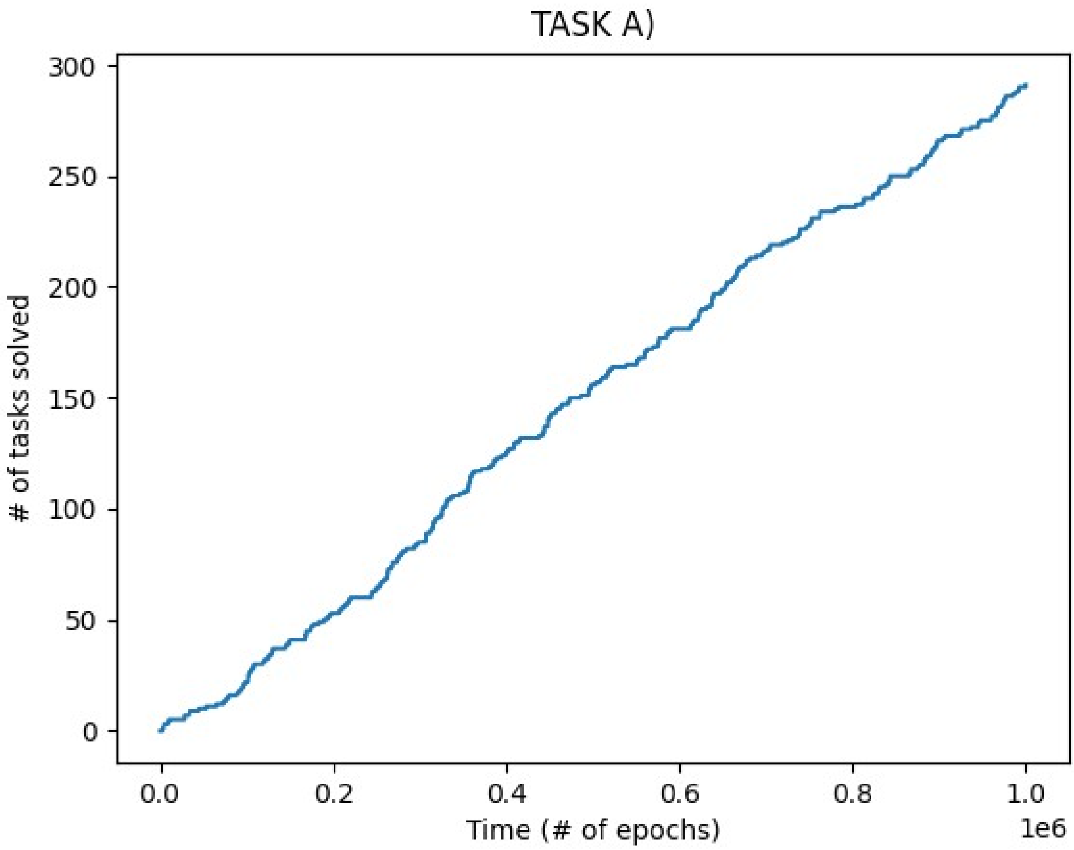
Results and Discussion

Task A

For modeling an agent moving randomly in the search area, I've implemented an agent that at each timestep moves in an entirely random direction with constant speed Rv=25, with the only limitation being that the agent is not allowed to move out of the grid. This should model an agent moving randomly well in this simulated environment; however, it is likely not a very accurate representation of real-world agent moving in physical environments, as they would likely not be able to change their moving direction at random at every timestep.

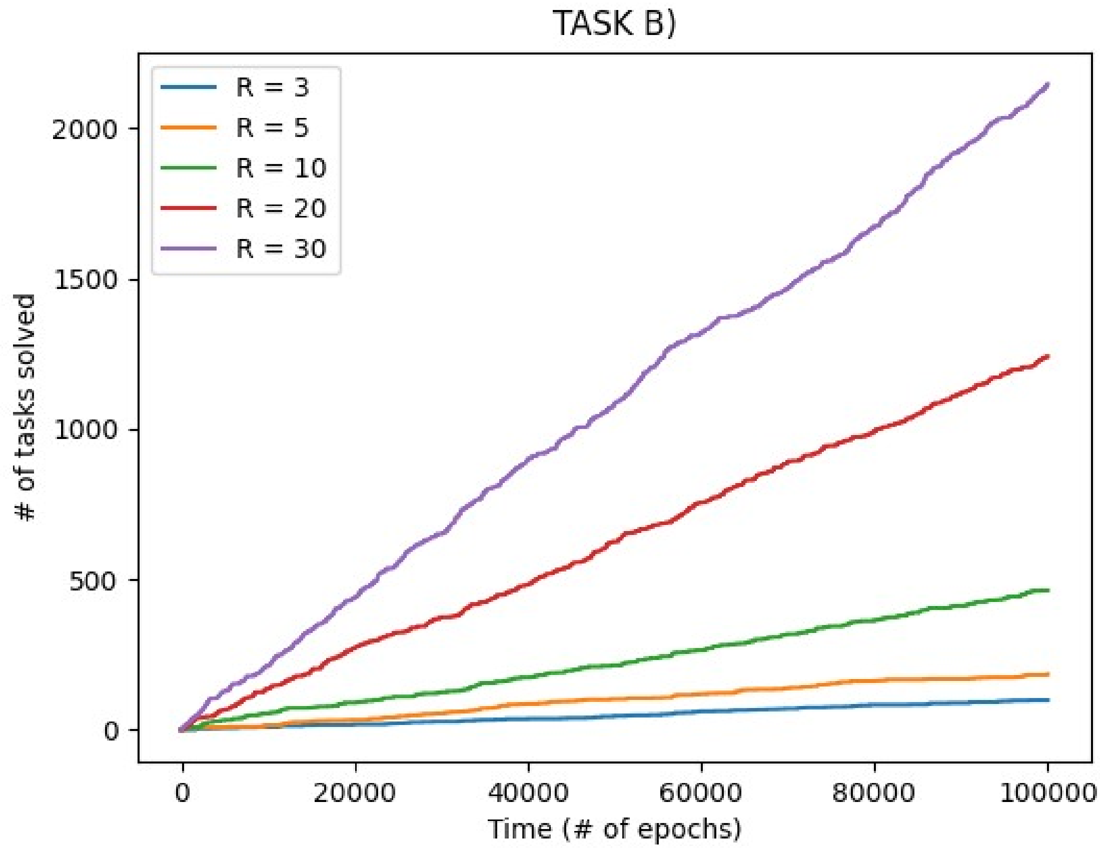


From the resulting graph of the simulation, we see that the number of tasks the agent solves per time is roughly linear, with some variance due to the randomness of the agent movement and the task position. In the case of one agent solving one task, this is a very sensible way of measuring performance in the STAC problem; specifically, it measures the Completion abilities of the agent well, as the graph measures the number of completed tasks per time.

As for Search, there may be other ways to assess whether good exploration is achieved, such as looking at grid coverage over time. However, this isn't necessary for this assignment, as we keep the exploration behavior constant for all agents in all exercises (they all move randomly until they find a task, or in later tasks, until they are called upon).

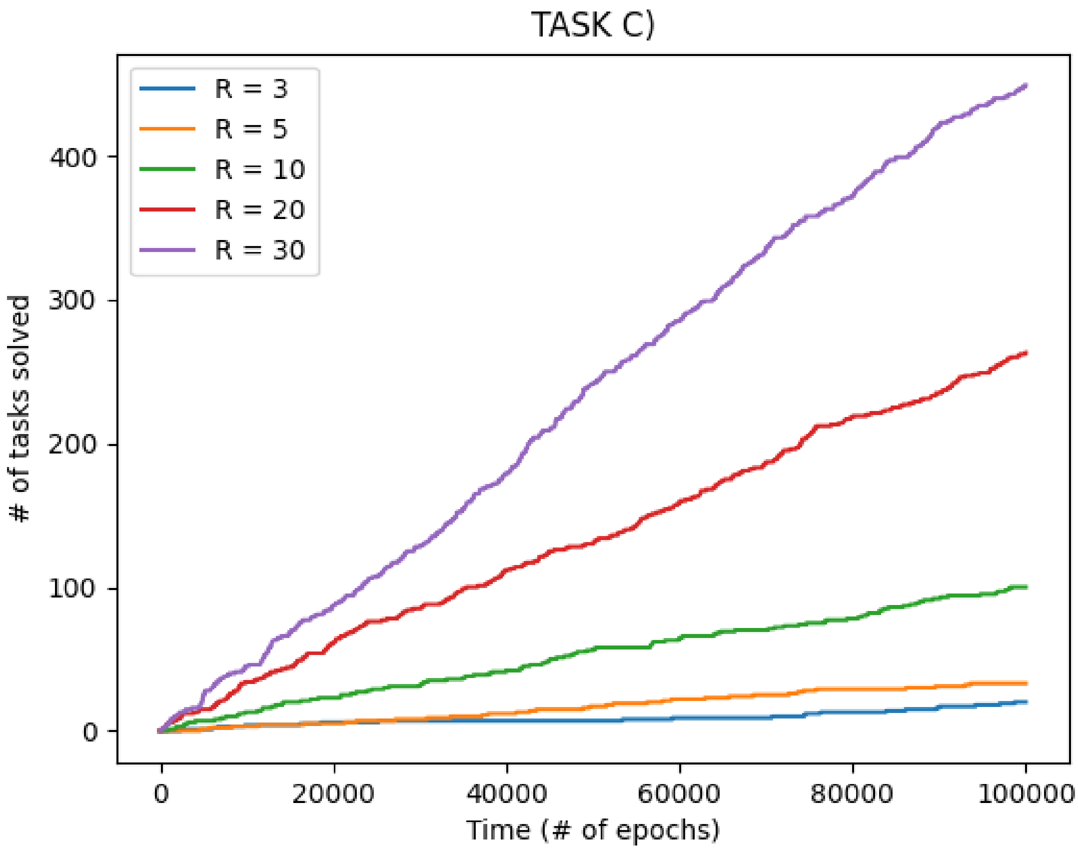
For the purpose of measuring Task Allocation, the number of tasks solved does not give us any direct assessment of whether efficient swarm behavior has been achieved; we can only assume that if the number of tasks solved should increase when task allocation efficiency increases, but we cannot observe this directly from such a graph.

Task B



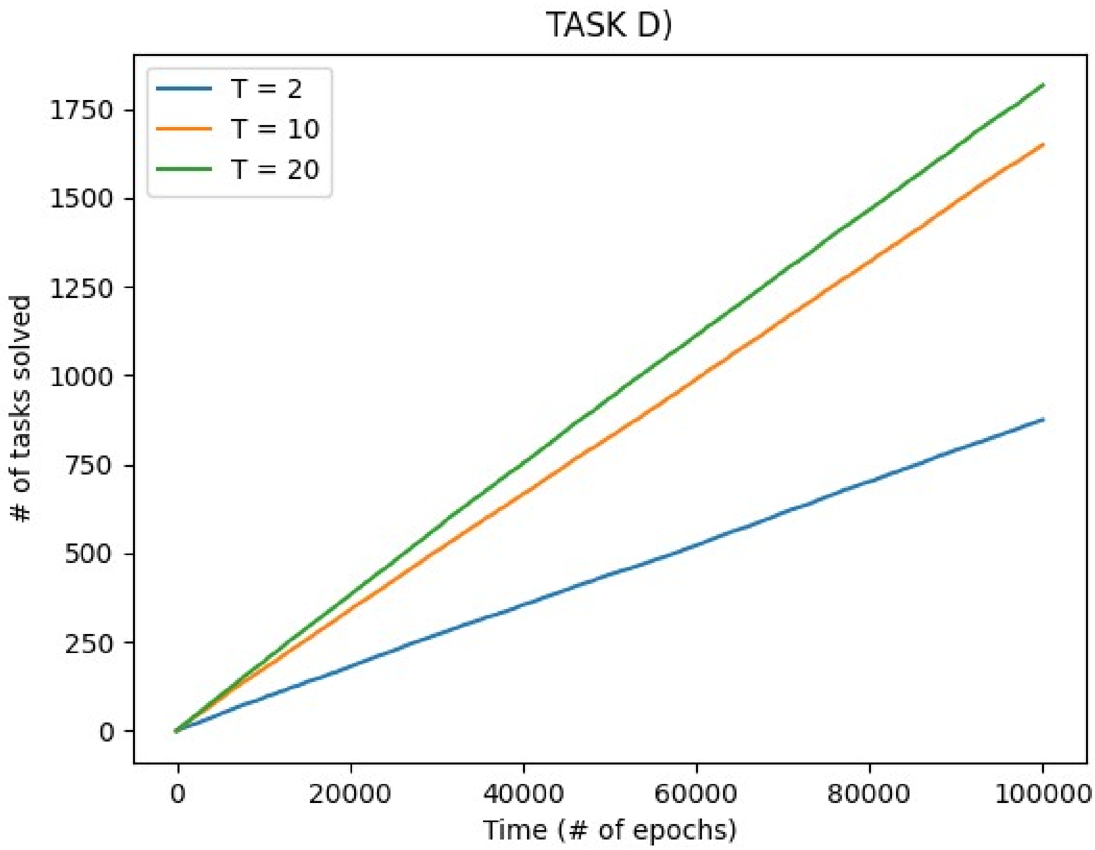
We observe that by adding agents, we still achieve a linear increase in tasks solved per time, but the rate of increase becomes higher as more agents are added. This makes sense, as more agents will cover more points in the grid at any given timestep, increasing the likelihood of one of them finding and completing the task.

Task C



This graph looks almost identical to the last one, but we observe that the number of tasks solved per time becomes much smaller than in task B.

Task D



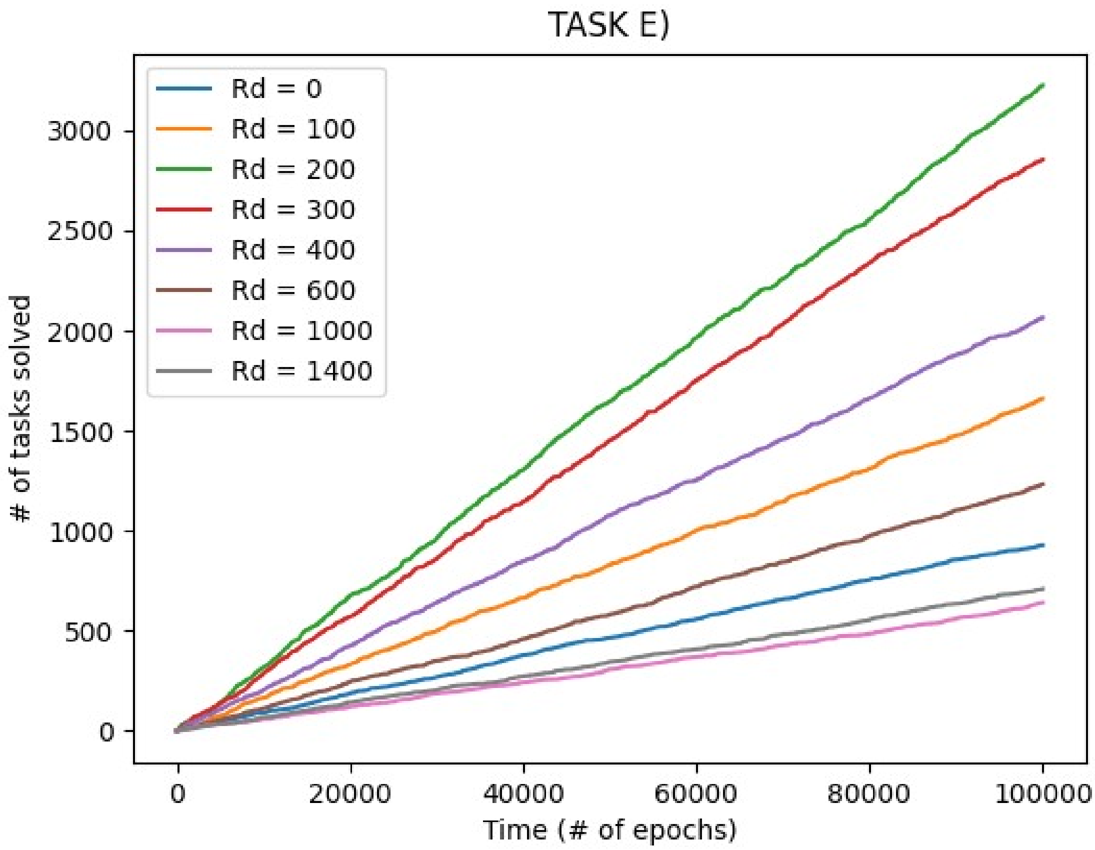
From the graph, we find that by increasing the number of tasks, we increase the average number of tasks that the agents are able to complete.

When it comes to the agents reading steady-state, I assume it refers to the possibility of all agents reaching a task in such a way that no single task has enough agents to complete the task. This only becomes possible when the number of agents is too low compared to the number of tasks and their task capacity. For instance, in my simulations, I have presumed that 30 agents are used (see code), and that tasks have a task capacity of 3. Then, each task may “trap” 2 agents (because they have reached inside a task radius, and remain there until the task is completed). In this case, whenever the number of tasks is 15 or higher, it becomes possible for all 30 agents to become trapped in the task radii, so that none of them move, and the environment has then converged to a steady state.

In theory, if the criteria above are fulfilled, any system with these criteria should at some point converge to a steady state by the randomness of movement in the agents. However, for the given numbers of tasks I have tested for, this seems very unlikely to happen, even with 20 tasks. We still observe the rate of increase growing as the number of tasks grows. I have also experimented with even larger numbers of tasks to see if it converges. At very large numbers of tasks (T = [40, 50, 60]), it seems random whether the environment converges or not from experiment to experiment, though it happens more often the higher the number of tasks.

I found that 10 episodes per number of tasks seems to give steady average performances with little variance as suggested by the very straight lines on the graph, meaning this should give a good statistical estimate of the average performance per number of tasks.

Task E

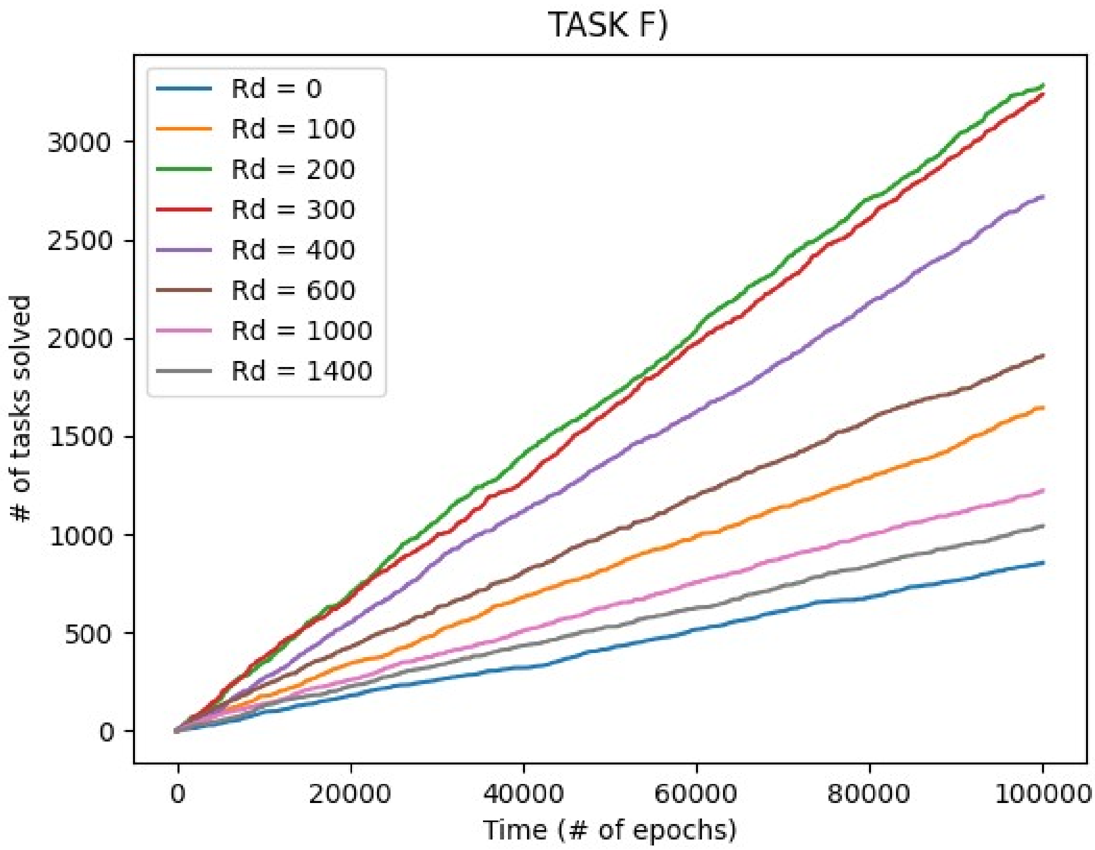


As in all the experiments so far, we again observe a linear increase in tasks solved per time. Interestingly, it is not always so that an increase in communication distance increases performance. In particular, notice how Rd=[1000, 1400] actually decreases overall performance when compared to the agents not being able to even communicate (Rd=0). At these communication distances, the agents can essentially “hear” each other across the entire grid. This means every agent can receive callouts from every agent within some task radius. A real-life analogy would be a lot of people shouting from different directions, telling you to come to them, in which case it would be hard to decide who to go to.

In the simulations, due to how I have programmed their behavior, this leads to all free agents going towards the last callout they heard, which is directly dependent on the order in which the program iterates through the agents. This can lead to unstable behavior where the agent may often change directions. While this can be changed to some other behavior like going towards the closest callout signal, the agent hearing multiple signals may still lead to unstable behavior.

Regardless of this specific instability in agent behavior, there is another likely reason that large communication distances turn out to be suboptimal. When we initialize agents randomly, this (hopefully) gives some spread of agents across the grid, leading to the agents exploring the grid better. If the agents can hear each other across the grid, the agents will tend to converge towards discovered tasks, increasing exploitation and decreasing exploration drastically, as this prevents the agents from covering the grid. By using a middle solution with a smaller communication distance, we find a middle ground where agents explore the grid, and exploit in small, local areas.

Task F



Overall, adding call-off leads to better performance in terms of tasks solved. In comparison to task E, even the largest communication distances now perform better than with no communication, though it is still somewhat short distances that give optimal performance in my simulations. It seems likely that implementing call-off combats the issues of instability and over-exploitation from task E.

Program Code Appendix

```python
from utils import *
from constants import *
import numpy as np

class Agent:
    def __init__(
            self,
            x: float = None,
            y: float = None,
            vx: float = 0.0,
            vy: float = 0.0,
            abs_velocity: float = AGENT_ABSOLUTE_VELOCITY,
            comm_dist: float = 0.0
            ):
        x = np.random.random()*1000 if x is None else x
        y = np.random.random()*1000 if y is None else y
        self.pos = np.array([x, y])
        self.velocity = np.array([vx, vy])
        self.abs_velocity = abs_velocity
        self.comm_dist = comm_dist
        self.target_pos = None
        self.inside_task_radius = False

    def update_pos(self):
        """
        Updates the current position of the agent by adding the self.velocity vector to the
        self.pos vector. When updating positions, the function disallows the agent to go out
        of bounds of the square grid spanning from (0, 0) to (1000, 1000). This means if the
        agent would go out of bounds by following its trajectory at its current absolute
        velocity, it instead moves in the same direction but at a lower absolute velocity, so
        that it stops at the border of the grid.
        """

        self.pos = self.pos + self.velocity
        self.pos = np.minimum(self.pos, 1000)
        self.pos = np.maximum(self.pos, 0)

    def update_velocity(self):
        """
        Updates velocity of agent according to following conditions:

        If it has reached a task (is inside task radius), it stops moving.

        If target_pos is specified (not None) and the agent has not reached it, sets agent
velocity
        towards that position.

        Otherwise, makes the agent's movement random by changing the velocity in each direction
        to some random number. Components vx and vy are set so that the absolute velocity sums
up to
        abs_velocity.
        """

        # Removes target_pos (should it be set) if agent is inside any task radius
        if self.inside_task_radius:
            self.target_pos = None
            self.velocity = np.zeros((2))
            return

        # Goes towards target_pos if specified and is not (almost) equal to pos
        if self.target_pos is not None and not np.allclose(self.pos, self.target_pos):
            self.velocity = self.target_pos - self.pos
            norm = np.linalg.norm(self.velocity)
            self.velocity = (self.velocity / norm) * np.minimum(self.abs_velocity, norm)
            return

        # If not, removes target_pos and initializes random movement
```

```
        self.target_pos = None

        # Sets velocity in each direction to random number in interval [-1, 1]
        self.velocity = (1 - (-1))*np.random.random(self.velocity.shape) - 1

        # Normalizing vector, making the norm of self.velocity equal to 1
        norm = np.linalg.norm(self.velocity)
        # To solve the unlikely case that both directional velicities are sampled as 0, we
        # randomly set vx = 1 or vy = 1 if that happens
        if norm == 0:
            self.velocity = np.array([1, 0]) if np.random.random() > 0.5 else np.array([0,
1])
            norm = np.linalg.norm(self.velocity)
        self.velocity = self.velocity / norm

        # Multiplying self.velocity (now a unit vector) with abs_velocity to achieve desired
        # (or random) velocity
        self.velocity = self.velocity * self.abs_velocity

    def callout(self, agents: list):
        """
        When the agent is within the task radius of any task, it emits a signal to other
agents
        within comm_dist to make them go towards that location, by setting their target_pos to
the
        position of the agent emitting the signal.

        The called upon agents will then go towards the coordinate from which the signal was
emitted until:
        a) they reach the signal location.
        b) they find themselves within a task radius themselves.
        The above conditions are checked for each agent when their velocities are updated.
        """

        # Send a signal to any agent within comm_dist
        for agent in agents:
            # Skips itself
            if agent == self:
                continue

            # Checking if the agent is within the comm_dist or is already within a task radius
            # (an agent which as already detected a nearby task will itself send out a signal,
            # and presumably would then be more interested in its own discovered task than the
            # signal of another agent)
            if (not agent.inside_task_radius) and distance_euclid(self.pos, agent.pos) <
self.comm_dist:
                agent.target_pos = self.pos

    def calloff(self, agents: list):
        """
        Performs the "opposite" action of callout: instead of giving agents within comm_dist a
target_pos,
        this function removes their target_pos if their previous target_pos was this agent's
current pos.

        This method is called from the task when it checks whether it should be completed.
        """

        # Send a signal to any agent within comm_dist
        for agent in agents:
            # Skips itself
            if agent == self or agent.target_pos is None:
                continue

            if np.allclose(self.pos, agent.target_pos) and distance_euclid(self.pos,
agent.pos) < self.comm_dist:
                agent.target_pos = None
```

```python
from utils import *
from constants import *
from agent import *
import numpy as np

class Task:
    def __init__(
            self,
            x: float = None,
            y: float = None,
            task_capacity: int = 1,
            task_radius: float = 100
            ):
        x = np.random.random()*1000 if x is None else x
        y = np.random.random()*1000 if y is None else y
        self.pos = np.array([x, y])
        self.task_capacity = task_capacity
        self.task_radius = task_radius

    def sufficient_agents_in_radius(self, agents: list, invoke_calloff: bool = False):
        """
        Checks whether there are enough agents within the task's radius for it to be complete.
        Also invokes calloff from agents within task radius if specified.
        """
        num_agents_in_radius = 0
        for agent in agents:
            # Checking if the agent is within the task radius, adding to num_agents_in_radius
            if distance_euclid(self.pos, agent.pos) < self.task_radius:
                num_agents_in_radius += 1
                agent.inside_task_radius = True

            # Checking if enough rgents are close enough to task to complete it
            if num_agents_in_radius >= self.task_capacity:
                for agent in agents:
                    if distance_euclid(self.pos, agent.pos) < self.task_radius:
                        agent.inside_task_radius = False
                        # Tells agents to perform calloff when this task is completed, if
specified
                        if invoke_calloff:
                            agent.calloff(agents)
                return True
        return False
```

```python
from utils import *
from constants import *
from agent import *
from task import *
import numpy as np
import matplotlib.pyplot as plt


def experiment_1():
    # Initializing task T at random position
    task = Task(task_capacity=1, task_radius=50)

    # Initializing agent R1 at random position
    r1 = Agent()

    # Starting simulation
    results = np.zeros((NUM_EPOCHS_A))
    completed_tasks = 0
    for i in range(NUM_EPOCHS_A):
        # Updating movement (velocity and position) of agent
        r1.update_velocity()
        r1.update_pos()

        # Checking if the agent is within the task radius, adding to
        # completed tasks and creating a new one if so
        if distance_euclid(task.pos, r1.pos) < task.task_radius:
            completed_tasks += 1
            task = Task(task_capacity=TASK_CAPACITY_A, task_radius=TASK_RADIUS_A)
        results[i] = completed_tasks

    # Plotting results
    x = np.linspace(1, NUM_EPOCHS_A, NUM_EPOCHS_A)
    y = results
    plt.plot(x, y)
    plt.title("TASK A)")
    plt.xlabel("Time (# of epochs)")
    plt.ylabel("# of tasks solved")
    plt.savefig(fname='figures/task_a')
    plt.close()


def experiment_2():

    # Performing experiment with different numbers of agents
    for agent_num in NUM_AGENTS_B:
        # Initializing agents at random positions
        agents = []
        for _ in range(agent_num):
            agent = Agent()
            agents.append(agent)

        # Initializing task T at random position
        task = Task(task_capacity=TASK_CAPACITY_B, task_radius=TASK_RADIUS_B)

        # Starting simulation
        results = np.zeros((NUM_EPOCHS_B))
        completed_tasks = 0
        for i in range(NUM_EPOCHS_B):
            # Updating movement (velocity and position) of agent
            for agent in agents:
                agent.update_velocity()
                agent.update_pos()

                # Checking if the agent is within the task radius, adding to
                # completed tasks and creating a new one if so
                if distance_euclid(task.pos, agent.pos) < task.task_radius:
                    completed_tasks += 1
```

```
                task = Task(task_capacity=TASK_CAPACITY_B, task_radius=TASK_RADIUS_B)
            results[i] = completed_tasks
        x = np.linspace(1, NUM_EPOCHS_B, NUM_EPOCHS_B)
        y = results
        plt.plot(x, y, label=f'R = {agent_num}')
        print(f"Simulations for R = {agent_num} complete.")

    # Plotting results
    plt.title("TASK B)")
    plt.xlabel("Time (# of epochs)")
    plt.ylabel("# of tasks solved")
    plt.legend()
    plt.savefig(fname='figures/task_b')
    plt.close()

def experiment_3():
    # Performing experiment with different numbers of agents
    for agent_num in NUM_AGENTS_C:
        # Initializing agents at random positions
        agents = []
        for _ in range(agent_num):
            agent = Agent()
            agents.append(agent)

        # Initializing task T at random position
        task = Task(task_capacity=TASK_CAPACITY_C, task_radius=TASK_RADIUS_C)

        # Starting simulation
        results = np.zeros((NUM_EPOCHS_C))
        completed_tasks = 0
        for i in range(NUM_EPOCHS_C):
            # Updating movement (velocity and position) of agent
            for agent in agents:
                agent.update_velocity()
                agent.update_pos()

            task_completed = task.sufficient_agents_in_radius(agents)
            if task_completed:
                completed_tasks += 1
                task = Task(task_capacity=TASK_CAPACITY_C, task_radius=TASK_RADIUS_C)
            results[i] = completed_tasks

        x = np.linspace(1, NUM_EPOCHS_C, NUM_EPOCHS_C)
        y = results
        plt.plot(x, y, label=f'R = {agent_num}')
        print(f"Simulations for R = {agent_num} complete.")

    # Plotting results
    plt.title("TASK C)")
    plt.xlabel("Time (# of epochs)")
    plt.ylabel("# of tasks solved")
    plt.legend()
    plt.savefig(fname='figures/task_c')
    plt.close()

def experiment_4():
    # Storing results across all episodes
    episodes = []

    # Performing experiment with different numbers of tasks
    for task_num in NUM_TASKS_D:
        for _ in range(NUM_EPISODES_D):
            # Initializing tasks at random positions
            tasks = []
            for _ in range(task_num):
                task = Task(task_capacity=TASK_CAPACITY_D, task_radius=TASK_RADIUS_D)
                tasks.append(task)
```

```
            # Initializing agents at random positions
            agents = []
            for _ in range(NUM_AGENTS_D): # For purpose of experiment, assuming R=30 agents
                agent = Agent()
                agents.append(agent)

            # Starting simulation
            results = np.zeros((NUM_EPOCHS_D))
            completed_tasks = 0
            for i in range(NUM_EPOCHS_D):
                # Updating movement (velocity and position) of agent
                for agent in agents:
                    agent.update_velocity()
                    agent.update_pos()

                for task_i in range(len(tasks)):
                    task = tasks[task_i]
                    task_completed = task.sufficient_agents_in_radius(agents)
                    if task_completed:
                        completed_tasks += 1
                        tasks[task_i] = Task(task_capacity=TASK_CAPACITY_D,
task_radius=TASK_RADIUS_D)
                results[i] = completed_tasks
            episodes.append(results)
        x = np.linspace(1, NUM_EPOCHS_D, NUM_EPOCHS_D)
        y = np.array(episodes).mean(axis=0)
        plt.plot(x, y, label=f'T = {task_num}')
        print(f"Simulations for T = {task_num} complete.")

    # Plotting results
    plt.title("TASK D)")
    plt.xlabel("Time (# of epochs)")
    plt.ylabel("# of tasks solved")
    plt.legend()
    plt.savefig(fname='figures/task_d')
    plt.close()

def experiment_5():
    # Performing experiment with different communication distances
    for comm_dist in COMM_DISTANCES_E:
        # Initializing tasks at random positions
        tasks = []
        for _ in range(NUM_TASKS_E): # Assuming T=2 tasks
            task = Task(task_capacity=TASK_CAPACITY_E, task_radius=TASK_RADIUS_E)
            tasks.append(task)

        # Initializing agents at random positions
        agents = []
        for _ in range(NUM_AGENTS_E): # Assuming R=30 agents
            agent = Agent(comm_dist=comm_dist)
            agents.append(agent)

        # Starting simulation
        results = np.zeros((NUM_EPOCHS_E))
        completed_tasks = 0
        for i in range(NUM_EPOCHS_E):
            for task_i in range(len(tasks)):
                task = tasks[task_i]
                task_completed = task.sufficient_agents_in_radius(agents)
                if task_completed:
                    completed_tasks += 1
                    tasks[task_i] = Task(task_capacity=TASK_CAPACITY_E,
task_radius=TASK_RADIUS_E)

            # Updating movement (velocity and position) of agent
            for agent in agents:
```

```python
                # Callout is performed before updating velocities, so that each
                # agent can evaluate whether conditions are met for it to follow
                # the target_pos it may receive from callout.
                if agent.inside_task_radius:
                    agent.callout(agents)
                agent.update_velocity()
                agent.update_pos()
            results[i] = completed_tasks
        x = np.linspace(1, NUM_EPOCHS_E, NUM_EPOCHS_E)
        y = results
        plt.plot(x, y, label=f'Rd = {comm_dist}')
        print(f"Simulations for Rd = {comm_dist} complete.")

    # Plotting results
    plt.title("TASK E)")
    plt.xlabel("Time (# of epochs)")
    plt.ylabel("# of tasks solved")
    plt.legend()
    plt.savefig(fname='figures/task_e')
    plt.close()

def experiment_6():
    # Performing experiment with different communication distances
    for comm_dist in COMM_DISTANCES_F:
        # Initializing tasks at random positions
        tasks = []
        for _ in range(NUM_TASKS_F): # Assuming T=2 tasks
            task = Task(task_capacity=TASK_CAPACITY_F, task_radius=TASK_RADIUS_F)
            tasks.append(task)

        # Initializing agents at random positions
        agents = []
        for _ in range(NUM_AGENTS_F): # Assuming R=30 agents
            agent = Agent(comm_dist=comm_dist)
            agents.append(agent)

        # Starting simulation
        results = np.zeros((NUM_EPOCHS_F))
        completed_tasks = 0
        for i in range(NUM_EPOCHS_F):
            for task_i in range(len(tasks)):
                task = tasks[task_i]
                task_completed = task.sufficient_agents_in_radius(agents, invoke_calloff=True)
                if task_completed:
                    completed_tasks += 1
                    tasks[task_i] = Task(task_capacity=TASK_CAPACITY_F,
task_radius=TASK_RADIUS_F)

            # Updating movement (velocity and position) of agent
            for agent in agents:
                # Callout is performed before updating velocities, so that each
                # agent can evaluate whether conditions are met for it to follow
                # the target_pos it may receive from callout.
                if agent.inside_task_radius:
                    agent.callout(agents)
                agent.update_velocity()
                agent.update_pos()
            results[i] = completed_tasks
        x = np.linspace(1, NUM_EPOCHS_F, NUM_EPOCHS_F)
        y = results
        plt.plot(x, y, label=f'Rd = {comm_dist}')
        print(f"Simulations for Rd = {comm_dist} complete.")

    # Plotting results
    plt.title("TASK F)")
    plt.xlabel("Time (# of epochs)")
    plt.ylabel("# of tasks solved")
```

```python
    plt.legend()
    plt.savefig(fname='figures/task_f')
    plt.close()


if __name__ == "__main__":
    print("\n################### TASK A) ###################")
    experiment_1()
    print("\n################### TASK B) ###################")
    experiment_2()
    print("\n################### TASK C) ###################")
    experiment_3()
    print("\n################### TASK D) ###################")
    experiment_4()
    print("\n################### TASK E) ###################")
    experiment_5()
    print("\n################### TASK F) ###################")
    experiment_6()
```

```
# Universal constants
AGENT_ABSOLUTE_VELOCITY = 25 # Rv as specified in assignment text

# Task a)
NUM_EPOCHS_A = int(1e6) # Number of iterations per episode
TASK_CAPACITY_A = 1 # Tc for all Task objects
TASK_RADIUS_A = 50 # Tr for all Task objects

# Task b)
NUM_EPOCHS_B = int(1e5)
TASK_CAPACITY_B = 1
TASK_RADIUS_B = 50
NUM_AGENTS_B = [3, 5, 10, 20, 30] # Number of agents

# Task c)
NUM_EPOCHS_C = int(1e5)
TASK_CAPACITY_C = 3
TASK_RADIUS_C = 50
NUM_AGENTS_C = [3, 5, 10, 20, 30]

# Task d)
NUM_EPISODES_D = 10 # Number of episodes
NUM_EPOCHS_D = int(1e5)
NUM_TASKS_D = [2, 10, 20] # Number of tasks
TASK_CAPACITY_D = 3
TASK_RADIUS_D = 50
NUM_AGENTS_D = 30

# Task e)
NUM_EPOCHS_E = int(1e5)
NUM_TASKS_E = 2
TASK_CAPACITY_E = 3
TASK_RADIUS_E = 50
NUM_AGENTS_E = 30
COMM_DISTANCES_E = [0, 100, 200, 300, 400, 600, 1000, 1400] # Communication distance

# Task f)
NUM_EPOCHS_F = int(1e5)
NUM_TASKS_F = 2
TASK_CAPACITY_F = 3
TASK_RADIUS_F = 50
NUM_AGENTS_F = 30
COMM_DISTANCES_F = [0, 100, 200, 300, 400, 600, 1000, 1400]
```

```python
import numpy as np

def distance_euclid(vec_a: np.array, vec_b: np.array):
    return np.linalg.norm(vec_b - vec_a)
```