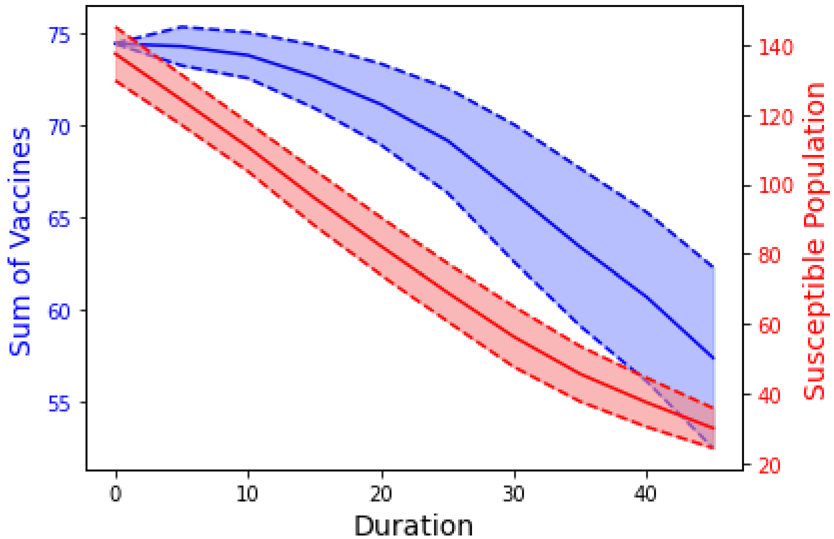
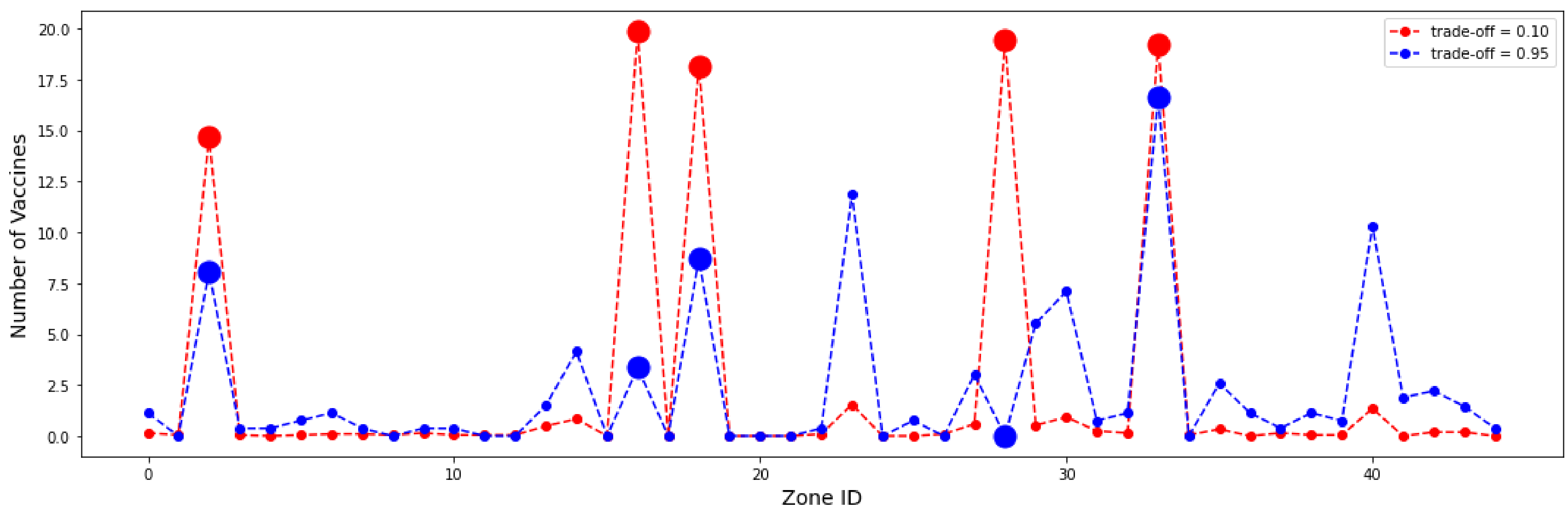
In [42]:
```
from matplotlib import pyplot as plt
import numpy as np

# experiment 2 rendition 2

import numpy as np
import matplotlib.pyplot as plt

# Create some mock data
t = np.arange(0.01, 10.0, 0.01)
data1 = [74.4044311142, 74.2625051138, 73.7763650102, 72.617034761, 71.12093181819999, 69.1502188716, 66.3084190
data2 = [137.52, 124.26, 110.74, 96.14, 82.34, 69.06, 56.36, 45.68, 37.52, 30.14]

vac = data1
sus = data2

vac_std = [0.01202053, 1.04794591, 1.24330532, 1.70601292, 2.19675224, 2.834807, 3.71498164, 4.25674477, 4.57118
sus_std = [7.7051671,  7.21612084, 6.99945712, 7.93727913, 8.2331282,  8.28832914, 8.65507943, 7.93080072, 7.032

fig, ax1 = plt.subplots()

ax1.set_xlabel('Duration', fontsize=14)
ax1.set_ylabel('Sum of Vaccines', color='blue', fontsize=14)
ax1.tick_params(axis='y', labelcolor='blue')
ax1.plot([i*5 for i in range(len(vac))], vac, color='blue')
ax1.plot([i*5 for i in range(len(vac))], [vac[i] + vac_std[i] for i in range(len(vac))], color='blue', linestyle
ax1.plot([i*5 for i in range(len(vac))], [vac[i] - vac_std[i] for i in range(len(vac))], color='blue', linestyle
ax1.fill_between([i*5 for i in range(len(vac))], vac, [vac[i] + vac_std[i] for i in range(len(vac))], color='#87
ax1.fill_between([i*5 for i in range(len(vac))], vac, [vac[i] - vac_std[i] for i in range(len(vac))], color='#87

ax2 = ax1.twinx()

ax2.set_ylabel('Susceptible Population', color='red', fontsize=14)
ax2.tick_params(axis='y', labelcolor='red')
ax2.plot([i*5 for i in range(len(sus))], sus, color='red')
ax2.plot([i*5 for i in range(len(sus))], [sus[i] + sus_std[i] for i in range(len(sus))], color='red', linestyle=
ax2.plot([i*5 for i in range(len(sus))], [sus[i] - sus_std[i] for i in range(len(sus))], color='red', linestyle=
ax2.fill_between([i*5 for i in range(len(sus))], sus, [sus[i] + sus_std[i] for i in range(len(sus))], color='#f7
ax2.fill_between([i*5 for i in range(len(sus))], sus, [sus[i] - sus_std[i] for i in range(len(sus))], color='#f7

fig.tight_layout() # otherwise the right y-label is slightly clipped
plt.savefig('experiment_condition_2_3_final.png', dpi=300)
```

```
plt.show()
```


75
70
65
60
55
Sum of Vaccines
140
120
100
80
60
40
20
Susceptible Population
0
10
20
30
40
Duration

In [41]:
```
# Experiment 1

p0_1 = [0.14747104, 0.049438067, 14.685921, 0.049310038, 0.0, 0.049428155, 0.09890848, 0.098838255, 0.049349479,
p0_95 = [1.1307041, 0.0, 8.0685203, 0.37807464, 0.37894179, 0.75796056, 1.1375408, 0.37891106, 0.0, 0.37909878,

LW = [18, 28, 16, 2, 33]

plt.figure(figsize=(15,5))
plt.plot([i for i in range(45)], p0_1, color='red', label='trade-off = 0.10', linestyle='--', marker='o')
plt.plot([i for i in range(45)], p0_95, color='blue', label='trade-off = 0.95', linestyle='--', marker='o')

plt.plot([i for i in LW], [p0_1[i] for i in LW], color='red', marker='o', markersize=15, linewidth=0)
plt.plot([i for i in LW], [p0_95[i] for i in LW], color='blue', marker='o', markersize=15, linewidth=0)

plt.ylabel('Number of Vaccines', fontsize=14)
plt.xlabel('Zone ID', fontsize=14)
plt.legend()
plt.tight_layout()
plt.savefig('experiment_condition_1_3_final.png', dpi=300)
plt.show()
```



In [ ]:
```
# Learning Rate Results

lr0_1 = [0.4, 0.40999999999375003, 0.3832857142780485, 0.35924285713391707, 0.36617571427154577, 0.343843857128
lr0_2 = [0.4, 0.32, 0.330999999990625, 0.31479999998625, 0.30183999998275, 0.316471999976825, 0.30317759997521,
lr0_002 = [0.4, 0.40019999999987504, 0.3993995999998753, 0.3996008007997505, 0.3990515991981198, 0.398503495999

plt.plot([i for i in range(len(lr0_1))], lr0_1, color='red', label='lr = 0.1')
plt.plot([i for i in range(len(lr0_2))], lr0_2, color='blue', label='lr = 0.2')
plt.plot([i for i in range(len(lr0_002))], lr0_002 , color='green', label='lr = 0.002')
plt.plot([i for i in range(len(lr0_002))], [0.2 for i in range(len(lr0_002))], linestyle='--', color='black', la
plt.ylabel('Learning Rate', fontsize=14)
plt.xlabel('Duration', fontsize=14)
plt.gca().invert_yaxis()
plt.legend()
plt.tight_layout()
plt.savefig('learning_rate_630.png', dpi=300)
plt.show()
```

In [ ]:
```
import pulp
import random
import networkx as nx
import simpy
import numpy as np
import pickle
import random
import math
import matplotlib.pyplot as plt
import operator
import itertools
import pandas as pd
import decimal
import matplotlib.pyplot as plt
from geopy import distance
from geopy.distance import geodesic
from scipy.spatial.distance import *
from scipy.optimize import minimize, differential_evolution
from scipy import optimize
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, calinski_harabasz_score

final_arr = [[] for i in range(50)]
learning_rate_change = []
double_arr = []
sus_arr = [[] for i in range(50)]

for alphabetagammaepsilon in range(50):

    class Node(object):

        def __init__(self, env, ID, coor, state, alpha, beta, gamma, delta, sigma, zone):

            global T, d

            self.ID = ID
            self.env = env
            self.alpha = alpha
            self.beta = beta
            self.gamma = gamma
            self.delta = delta
            self.sigma = sigma
```

```
        self.zone = zone

        # Neighbor list
        self.nlist = []

        self.old_coor = None
        self.new_coor = coor
        self.start = True

        self.ti = 3

        self.state = state

        if self.ID == 1:
            self.env.process(self.time_increment())
            self.env.process(self.optimizer())
            d = []

        self.env.process(self.move())
        self.env.process(self.influence())

    def move(self):

        global Xlim, Ylim, W, a, zone_coordinates, R

        while True:

            # if T % mho == 0 and self.state != 'D':
            if T % mho == 0:

                c = zone_coordinates[self.zone]
                r = R[self.zone]

                # Define a set of k random points (potential next positions) within the circle of my current
                k = 10
                P = []

                # Calculate the distance between current location (p) and each potential next hop
                D = []

                for i in range(k):
                    x = random.uniform(c[0] - r, c[0] + r)
                    y = random.uniform(c[1] - r, c[1] + r)
```

```
                    P.append((x, y))
                    D.append(euclidean(self.new_coor, (x, y)))

                # Select the next destination from P preferring short distances over long distances
                likelihood_of_selecting = [1.0/D[i] for i in range(k)]
                likelihood_of_selecting = [likelihood_of_selecting[i]/np.sum(likelihood_of_selecting) for i
                ind = np.random.choice([i for i in range(k)], p = likelihood_of_selecting, size = 1)[0]

                # New position of current agent
                self.new_coor = P[ind]

            yield self.env.timeout(minimumWaitingTime)

    def scan_neighbors(self):

        global eG, sensing_range, entities, Coor

        while True:

            if T % PT == 2:

                self.nlist = []

                if self.start:
                    for u in range(eG):
                        if euclidean(self.new_coor, Coor[u]) <= sensing_range:
                            self.nlist.append(u)
                    self.start = False

                else:
                    for u in range(eG):
                        if euclidean(self.new_coor, entities[u].new_coor) <= sensing_range:
                            self.nlist.append(u)

                self.nlist = [u for u in self.nlist if u != self.ID]

            yield self.env.timeout(minimumWaitingTime)

    def influence(self):

        global minimumWaitingTime

        while True:
```

```
            if T % PT == (self.ti + 1) % PT:

                state_change = False

                if self.state == 'S':
                    for u in self.nlist:
                        if entities[u].state == 'E' and random.uniform(0, 1) <= self.alpha:
                            self.state = 'E'
                            state_change = True
                            break
                '''
                if self.state == 'S' and state_change == False:
                    for u in self.nlist:
                        if entities[u].state == 'I' and random.uniform(0, 1) <= self.beta:
                            self.state = 'I'
                            state_change = True
                            break
                '''

                if self.state == 'E' and state_change == False:
                    if random.uniform(0, 1) <= self.gamma:
                        self.state = 'I'
                        state_change = True

                '''
                if self.state == 'E' and state_change == False:
                    if random.uniform(0, 1) <= pi:
                        self.state = 'R'
                        state_change = True
                '''

                if self.state == 'I' and state_change == False:
                    if random.uniform(0, 1) <= self.delta:
                        self.state = 'R'
                        state_change = True

                if self.state == 'I' and state_change == False:
                    if random.uniform(0, 1) <= self.sigma:
                        self.state = 'D'
                        state_change = True

            yield self.env.timeout(minimumWaitingTime)
```

```
    def time_increment(self):

        global Tracker, T, D, sus, exp, inf, rec, dth

        while True:

            T = T + 1
            sus = len([i for i in range(eG) if entities[i].state == 'S'])
            exp = len([i for i in range(eG) if entities[i].state == 'E'])
            inf = len([i for i in range(eG) if entities[i].state == 'I'])
            rec = len([i for i in range(eG) if entities[i].state == 'R'])
            dth = len([i for i in range(eG) if entities[i].state == 'D'])

            # print('sus: ' + str(sus) + ', exp: ' + str(exp) + ', inf: ' + str(inf) + ', rec: ' + str(rec)

            d.append((inf, rec, dth))

            # print (self.new_coor)
            if T % mho == 0 and self.old_coor != None:
                plt.scatter(self.new_coor[0], self.new_coor[1], s = 10, c = 'green')
                plt.plot([self.old_coor[   0], self.new_coor[0]], [self.old_coor[1], self.new_coor[1]], line

            yield self.env.timeout(minimumWaitingTime)

    def optimizer(self):

        global I, E, S, z, r, T, vaccines, f_interval, learning_rate_change

        while True:
            if T % vaccine_interval == 0:

                    vaccines_per_zone = resource_allocation(0.3, vaccines, T)
                    arr_infected, arr_suspected, arr_exposed = np.zeros(z), np.zeros(z), np.zeros(z)

                    for delta in range(z):
                        arr_infected[delta] = len([i for i in range(eG) if entities[i].state == 'I' and enti
                        arr_suspected[delta] = len([i for i in range(eG) if entities[i].state == 'S' and ent
                        arr_exposed[delta] = len([i for i in range(eG) if entities[i].state == 'E' and entit

                        available_vaccine = vaccines_per_zone[delta]
                        arr = [iota for iota in range(len(agent_zones)) if ((agent_zones[iota]==delta and er

                        immune, vaccinated = [], []
```

```
                        for phi in range(len(arr)):
                            while(available_vaccine > 0):
                                initial_state = entities[arr[phi]].state
                                entities[arr[phi]].state = np.random.choice([initial_state, 'R'], size=1, p=
                                vaccinated.append(arr[phi])
                                if(entities[arr[phi]].state == "R"):
                                    immune.append(arr[phi])
                                    if initial_state == 'S':
                                        arr_suspected[delta] -= 1
                                    elif initial_state == 'E':
                                        arr_exposed[delta] -= 1
                                available_vaccine -= 1
                                break

                        r[delta] = r[delta] + (((len(immune)/(len(vaccinated)+0.000000001)) - r[delta]) * le

                    learning_rate_change.append(r[2])

                    I = np.array(arr_infected)
                    S = np.array(arr_suspected)
                    E = np.array(arr_exposed)

            yield self.env.timeout(minimumWaitingTime)

def prospect(x, beta, lambdas):

    if x == 0:
        return 1

    return lambdas * math.pow(x, beta)


def count_extra(A):

    how_many_used = 0
    extra = 0

    for i in range(A.shape[0]):
        print (A[i])

        if np.sum(A[i]) > 0:
            how_many_used += 1
```

```
            if np.sum(A[i]) < 1.0:
                extra += 1.0 - np.sum(A[i])
    return how_many_used, extra

def latp(pt, W, a):

    # Available locations
    AL = [k for k in W.keys() if euclidean(W[k], pt) > 0]
    AL = cutoff(AL, W, pt)

    if len(AL) == 0:
        return pt

    den = np.sum([1.0 / math.pow(float(euclidean(W[k], pt)), a) for k in sorted(AL)])

    plist = [(1.0 / math.pow(float(euclidean(W[k], pt)), a) / den) for k in sorted(AL)]

    next_stop = np.random.choice([k for k in sorted(AL)], p = plist, size = 1)

    return W[next_stop[0]]

def least_distance_per_cluster(C, arr):

    array = np.zeros((len(arr), len(arr)))
    for row in range(len(arr)):
        for column in range(len(arr)):
            array[row][column] = euclidean(C[arr[row]], C[arr[column]])
    avg_distance = (np.mean(array, axis=1)).reshape(len(arr), )
    least = np.amin(avg_distance)
    result = 0
    for et in range(len(avg_distance)):
        if avg_distance[et] == least:
            result = arr[et]
            break
    return result

def def_zone_and_coor():

    global population_val, C

    population_val = np.array(file['Population'].values)
    pop = np.true_divide(population_val, np.sum(population_val))
```

```
    C = []
    coordinates = np.array(file['Location'].values)
    for a in range(0, len(coordinates)):
        arr = coordinates[a].split(', ')
        for b in range(0, len(arr)):
            arr[b] = float(arr[b])
        C.append((arr[1], arr[0]))
    agent_zones = np.random.choice(len(C), size=eG, p=pop)
    agent_initial_coordinates = [C[d] for d in agent_zones]

    return C, agent_zones, agent_initial_coordinates

def radius():

    global population_val, file

    area = np.true_divide(np.array(file['Population'].values), np.array(file['Population Density'].values))
    rad = [math.sqrt(area[i]/math.pi) for i in range(len(area))] / ((np.max(population_val))/eG)
    return rad

def initial_state():

    global infected_ratio, susceptible_ratio, exposed_ratio, population_val

    infected_ratio = np.true_divide(np.array(file['Total Infected'].values), np.array(file['Population'].val
    exposed_ratio = pe * (1 - infected_ratio)
    susceptible_ratio = 1 - (infected_ratio + exposed_ratio)
    initial_state = [(np.random.choice(['S', 'E', 'I'], size=1,
                               p=[susceptible_ratio[(agent_zones[c])],
                                  exposed_ratio[(agent_zones[c])],
                                  infected_ratio[(agent_zones[c])]])[0]) for c in range(len(agent_zones))]
    return initial_state

def initial_state():

    global infected_ratio, susceptible_ratio, exposed_ratio, population_val

    infected_ratio = np.true_divide(np.array(file['Total Infected'].values), np.array(file['Population'].val
    exposed_ratio = pe * (1 - infected_ratio)
    susceptible_ratio = 1 - (infected_ratio + exposed_ratio)
    initial_state = [(np.random.choice(['S', 'E', 'I'], size=1,
                               p=[susceptible_ratio[(agent_zones[c])],
                                  exposed_ratio[(agent_zones[c])],
```

```
                                          infected_ratio[(agent_zones[c])]])[0]) for c in range(len(agent_zones))]
    return initial_state

def resource_allocation(p, T, time):

    global trade_off, B, N, I, E, S, C, r

    # Low trade-off favor economic and high trade-off favors vaccine formulation
    trade_off = 0.95

    how_many = warehouse

    kmeans = KMeans(n_clusters=warehouse, random_state=0).fit(C)
    cluster_center = kmeans.cluster_centers_
    cluster_labels = kmeans.labels_

    label_arr = [[] for alpha in range(warehouse)]
    for beta in range(len(cluster_labels)):
        label_arr[cluster_labels[beta]].append(beta)

    # List of warehouses
    LW = [least_distance_per_cluster(C, label_arr[gamma]) for gamma in range(warehouse)]
    print('LW: ', LW)

    '''
    #Add the function to generate warehouses
    array = np.zeros((z, len(LW)))
    for row in range(z):
        for column in range(len(LW)):
            array[row][column] = euclidean(C[row], C[LW[column]])
    '''

    # Equally distributing vaccines across warehouse zones
    VW = {}
    current_warehouse = 0
    for f in range(1, T + 1):
        VW[f - 1] = LW[current_warehouse]
        if (f % (T / warehouse) == 0):
            current_warehouse += 1
    #print(VW.values())

    # Defining the parameters for the optimization
    B = np.array(file['Population Density'].values) * p  # rate of disease spread
```

```
N = np.array(num_agent_per_zone) # total population for each zone
if time <= vaccine_interval:
    r = np.array([0.4 for app in range(z)])
    I = np.array([infected_ratio[t]*num_agent_per_zone[t] for t in range(len(N))])
    E = (N-I) * pe
    S = N - (I + E)

'''
B = np.array([i + 1 for i in range(z)])
N = np.array([z - i for i in range(z)])
I = np.array([random.uniform(1, N[i]) for i in range(z)])
E = (N-I) * pe
S = N - (I + E)


B = np.array([2 for i in range(z)])
N = np.array([100 for i in range(z)])
I = np.array([25 for i in range(z)])
E = (N-I) * pe
S = N - (I + E)
'''

ir = I/(N+0.00001)

model = pulp.LpProblem("Vaccine problem", pulp.LpMinimize)
X = pulp.LpVariable.dicts("X", ((i, j) for i in range(warehouse) for j in range(len(B))), lowBound = 0.0

dist_array = [geodesic(C[VW[j]], C[b]).miles for j in range(T) for b in range(z)]
max_dist = np.max(dist_array)
den_economic = float(T * max_dist)
model += np.sum([X[LW.index(VW[j]), b] * geodesic(C[VW[j]], C[b]).miles for j in range(T) for b in range

# Constraint 1 ------------------------------------------------------------------------
for i in range(warehouse):
    # Condition 1: If you must assign all the vaccines generated by a warehouse
    #model += pulp.lpSum([X[(i, j)] for j in range(len(B))]) == int(T/warehouse)

    # Condition 2: If you want to minimize the number of vaccines
    model += pulp.lpSum([X[(i, j)] for j in range(len(B))]) <= int(T/warehouse)

# Constraint 2 ------------------------------------------------------------------------
s = 0.0
for i in range(warehouse):
```

```
        s += pulp.lpSum([X[(i, j)] for j in range(len(B))])

    # Condition 1: If you must assign all the vaccines generated by a warehouse
    #model += s == T

    # Condition 2: If you want to minimize the number of vaccines
    model += s <= T

    # Constraint 3 (fairness lower) ----------------------------------------------------------
    for i in range(len(B)):

        # Condition 3: If calculation is based on susceptible population
        c = (S[i] - r[i] * pulp.lpSum([X[(j, i)] for j in range(warehouse)]))/sum(S)

        # Condition 4: To include population density
        #c = c * B[i]/np.median(B)

        # Condition 5: To include infected population
        #c = c * ir[i]/np.median(ir)

        model += pulp.lpSum([X[(j, i)] for j in range(warehouse)]) >= trade_off * c * T

    # ---------------------------------------------------------------------------------------

    model.solve()
    print(pulp.LpStatus[model.status])

    # Transferred the pulp decision to the numpy array (A)
    A = np.zeros((T, len(B)))
    for i in range(warehouse):
        for j in range(len(B)):
            A[i, j] = X[(i, j)].varValue

    global double_arr
    double_arr.append(["Value: " + str(pulp.value(model.objective))])

    vaccines_per_zone = []
    for i in range(len(B)):
        vaccines_per_zone.append(np.sum(A[:, i]))

    print('Trade-off value: ' + str(trade_off))

    global final_arr, sus
```

```
        final_arr[alphabetagammaepsilon].append(np.sum(vaccines_per_zone))
        sus_arr[alphabetagammaepsilon].append(sus)

        plt.figure(figsize=(10,4))
        plt.bar([i for i in range(z)], vaccines_per_zone)
        print(vaccines_per_zone)
        #plt.tight_layout()
        #plt.savefig('plot.png', dpi=300)
        plt.show()
        print('var', alphabetagammaepsilon)
        return np.array(vaccines_per_zone)


    ''' Variables and Parameters for Simulation '''

    # Create Simpy environment and assign nodes to it.
    env = simpy.Environment()

    # Susceptible-> Exposed
    alpha = 0.2

    # Susceptible-> Infected
    beta = 0.2

    # Exposed-> Infected
    gamma = 0.2

    # Infected-> Recovered
    delta = 0.2

    # Infected-> Death
    sigma = 0.2

    PT = 10

    # Fraction of susceptible/exposed nodes
    pe = 0.3

    # Simulation area --> not relevant right now
    Xlim, Ylim = 100, 100

    # Number of agents, zones, warehouses, and vaccines (optimization parameters)
    eG = 200
```

```python
z = 45
warehouse = 5
vaccines = 100

# Simulation time-variable
T = 0

# Simulation duration
Duration = 50

# Move how often
mho = 3

# Time intervals for administering vaccines
vaccine_interval = 5

# Minimum waiting time
minimumWaitingTime = 1

# Variable used to increase proportion of infected (in case it's too low)
infected_bias = 0.0

sensing_range = 30

# File used for importing data
file = pd.read_csv("covid_confirmed_NY_july.csv")
file = file.iloc[0:z,:]

# Initial position and coordinates of node based on population density likelihood PW
zone_coordinates, agent_zones, Coor = def_zone_and_coor()

# Scaled radius of each zone
R = radius()

# Number of agents in each zone
num_agent_per_zone = [0 for i in range(z)]
for epsilon in range(len(agent_zones)):
    num_agent_per_zone[agent_zones[epsilon]] += 1

print("Number of agents per zone: ", num_agent_per_zone)

# Learning rate variables
expectedR = [0.2 for var in range(z)]
```

```
    learning_rate = 0.15
    learning_rate_change.append(0.4)

    # List of node initial states
    STATE = initial_state()

    entities = [Node(env, i, Coor[i], STATE[i], alpha, beta, gamma, delta, sigma, agent_zones[i]) for i in range
    env.run(until = Duration)

print('\n\n------------------------------------')
print('vac-avg', np.sum(final_arr, axis=0))
print('vac-std', np.std(final_arr, axis=0))
print('sus-avg', np.sum(sus_arr, axis=0))
print('sus-std', np.std(sus_arr, axis=0))
```