Project_Part_1

November 9, 2020

1 PSTAT 197A: Project Part 1

Peter Bayerle, Samantha Solomon, & Sophia Sternberg

2 Part 1 (a)

1. Using the model dynamics described in the previous section, and all the given and chosen parameters, simulate the behavior of the disease for 120 days for this single leaf node.

[1]:
```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import itertools
import time
import pandas as pd
from sklearn import datasets, linear_model,svm, metrics
```

[2]:
```
data = np.load('part1a.npz') # keys: N, Svc_0_pmf, Lc, Ic_0, gamma
```

[3]:
```
# initial conditions
N = data['N']
I0 = data['Ic_0']
S0 = data['Svc_0_pmf'] * (N-sum(I0))
R0 = 0
y0 = np.concatenate((S0.flatten(), I0.flatten(), [R0]))
t = list(range(0, 121))

print(f'total initial population: {np.sum(y0)}')
```

```
total initial population: 99999.99999999997
```

[4]:
```
# model params
gamma = data['gamma']
beta = np.concatenate([np.repeat(0.25, 4), np.repeat(0.5, 4), np.repeat(0.75,␣
 ↪4), np.repeat(1, 4)])
```

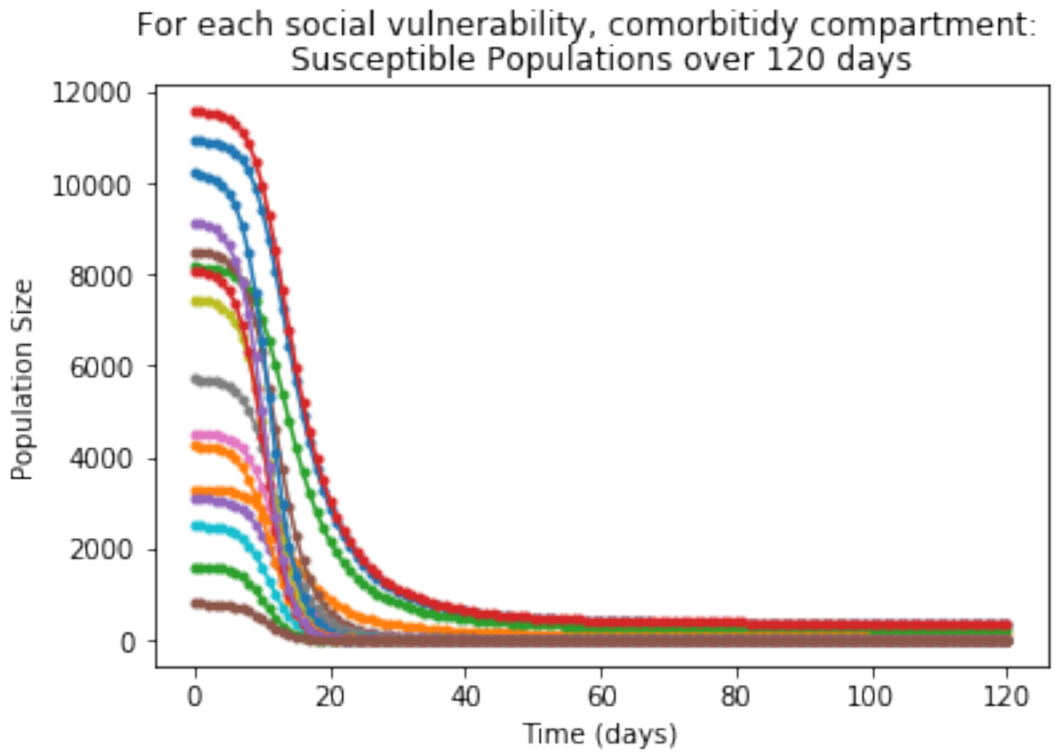
[5]:
```python
# define dy/dt = f(y,t)
def f(y, t):
    Si = y[:16]
    Ii = y[16:-1]
    Ri = y[-1]
    I = np.sum(Ii)

    f0 = beta * Si * (-I/N)
    f0_matrix = np.reshape(f0, newshape=(4,4)) # makes computation of f1 easier
    f1 = -1 * (np.sum(f0_matrix, axis=0) + gamma * Ii)
    f3 = gamma * I

    return np.concatenate((f0.flatten(), f1.flatten(), [f3]))
```

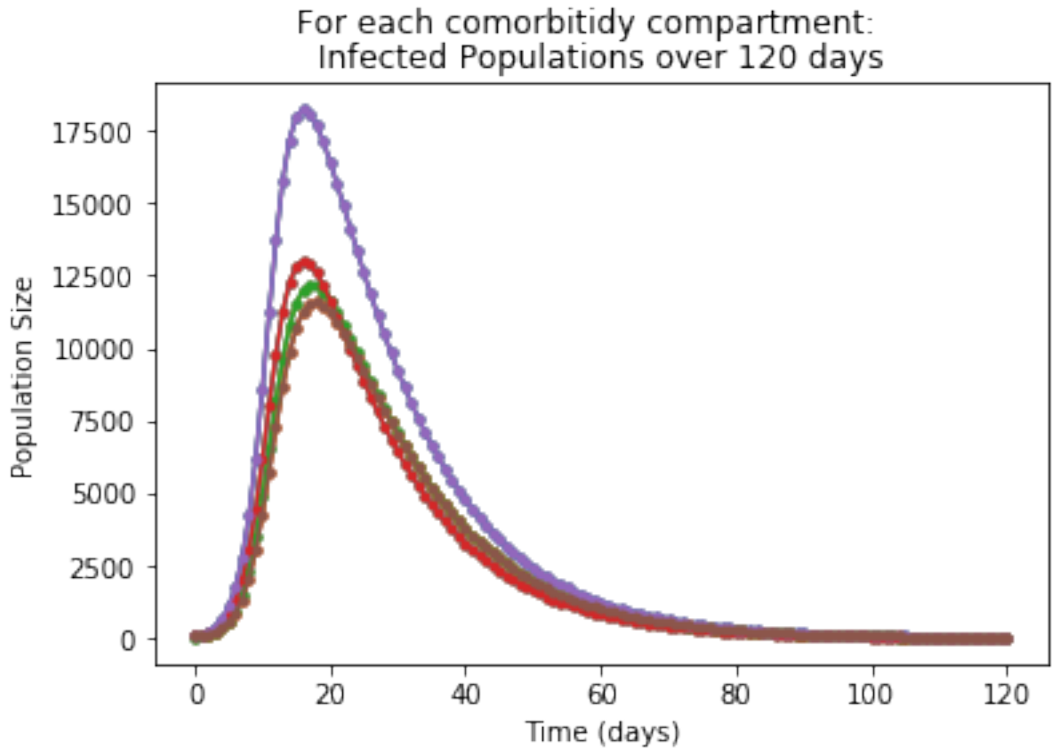
[6]:
```python
soln = odeint(f, y0, t)
# soln contains the predicted S,I,R values for each of the 120 days (so the␣
↪length of soln is 120).
# soln = [[s00, s01, ..., snn, i1, i2, i3, i4, r], ...]

S = soln[:,:16]
I = soln[:,16:-1]
R = soln[:,-1]

S_total = np.sum(S, axis=1)
I_total = np.sum(I, axis=1)
```

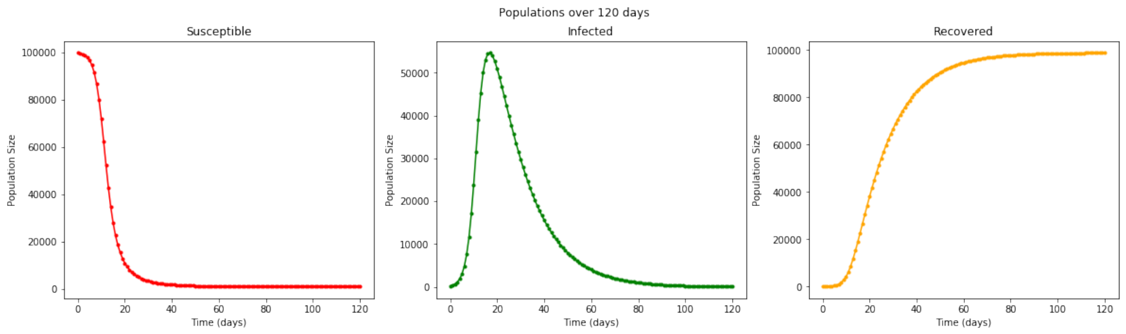
2. Plot $S_{v,c}$ and I_c values over time. (You can plot all $S_{v,c}$ on the same plot. Same for I_c.). Also plot the overall S, I, R and L values.

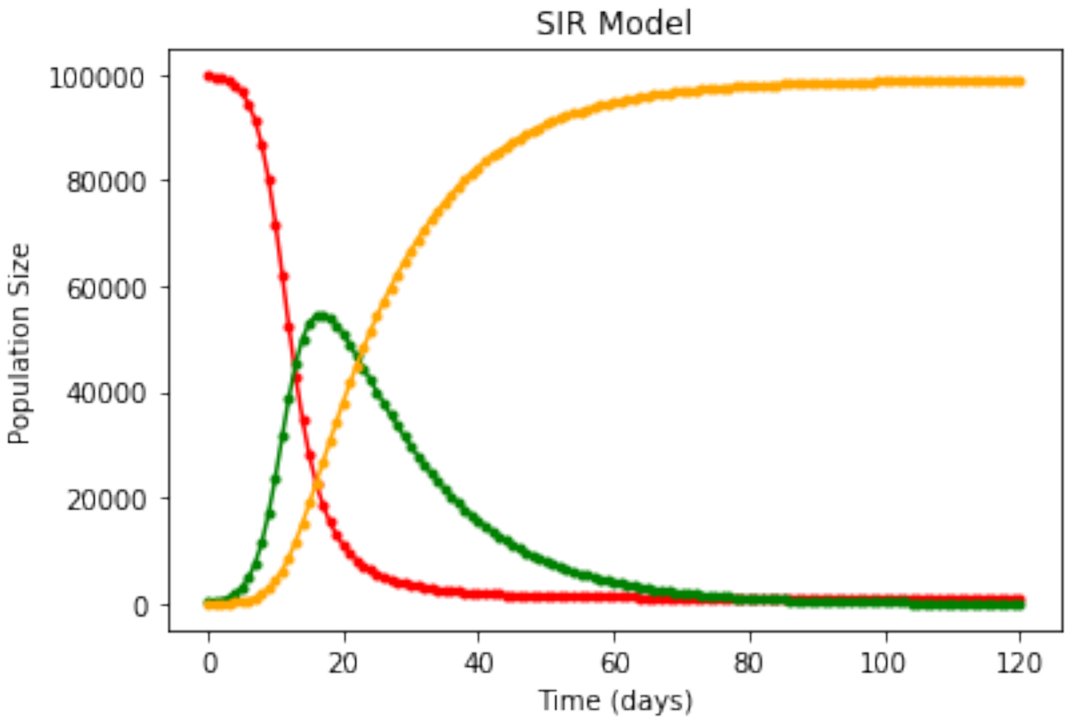
[7]:
```python
# Viral Load Density
Lc = data['Lc']

# initial viral load density
L0 = np.sum(Lc*I0)/N
L = np.array([L0])

for i in range(1,len(soln)):
  L = np.append(L,np.sum(soln[i,16:-1]*Lc)/N)
```

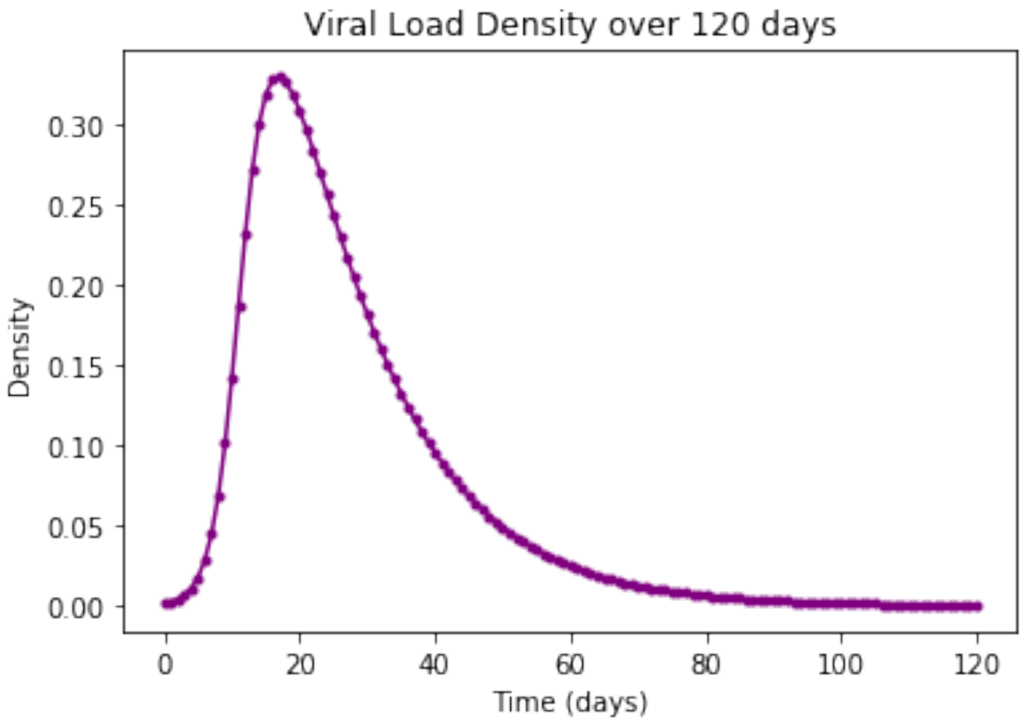
[8]:
```python
# helpers for plotting S, I, R, L values
def plt_SIR(S, I, R, t, title):
    fig, axs = plt.subplots(1, 3)
    fig.set_figheight(5)
    fig.set_figwidth(20)
    fig.suptitle(title)
    colors = ['red', 'green', 'orange']
    for i, data, title, color in zip(range(3), [S, I, R], ['Susceptible',␣
↪'Infected', 'Recovered'], colors):
        axs[i].set_title(title)
        axs[i].set_ylabel('Population Size')
        axs[i].set_xlabel('Time (days)')
        axs[i].plot(t, data, marker='.', color=color)
    plt.show()

def plt_L(L, t):
    plt.title('Viral Load Density over 120 days')
    plt.ylabel('Density')
    plt.xlabel('Time (days)')
    plt.plot(t, L,marker = ".", color = "purple");
    plt.show();


# Susceptible populations over time (S_v,c)
plt.title('Susceptible Populations over 120 days')
plt.suptitle('For each social vulnerability, comorbitidy compartment:')
plt.ylabel('Population Size')
plt.xlabel('Time (days)')
plt.plot(t,S, marker='.')
plt.show();

# Infected populations over time (I_v,c)
plt.title('Infected Populations over 120 days')
plt.suptitle('For each comorbitidy compartment:')
plt.ylabel('Population Size')
plt.xlabel('Time (days)')
for i in range(16,20):
  plt.plot(t,I, marker='.')
plt.show();

# Overall S,I,R
plt_SIR(S_total, I_total, R, t, 'Populations over 120 days')

# Complete SIR
plt.title("SIR Model")
plt.ylabel('Population Size')
plt.xlabel('Time (days)')
plt.plot(t,S_total,marker = ".", color = 'red');
plt.plot(t,I_total,marker = ".", color = "green");
plt.plot(t,R,marker = ".", color = "orange");
plt.show();

# Viral Load Density over time (L)
plt_L(L,t)
```


For each social vulnerability, comorbitidy compartment:
Susceptible Populations over 120 days
Population Size
Time (days)
0
2000
4000
6000
8000
10000
12000
0
20
40
60
80
100
120


For each comorbitidy compartment:
Infected Populations over 120 days
Population Size
Time (days)
0
2500
5000
7500
10000
12500
15000
17500
0
20
40
60
80
100
120


Populations over 120 days
Susceptible
Infected
Recovered
Population Size
Time (days)


SIR Model
Population Size
Time (days)



3. Do the shapes of S, I, R look similar to what you expected? Which of these plots is the "curve" people refer to when they say "flatten the curve"?

Answer: Yes, the shapes of the S, I, R look similar to what we expected. We would expect the Susceptible curve to start at N at time 0, and converge to 0 at the end of the 120 days, given our selected β parameters. The compartmentalized populations seem to be converging to 0, or very close to 0. The Infection curve seems to closely follow the Viral Load Density curve. The maximums of both the Infection curve and the Viral Load Density curve are close to time 20. Conversely from the Susceptible curve, we see that the Recovered curve converges to N by the end of the 120 days. When people say "flatten the curve", they are referring to the Infection curve, as a way to slow the transmission time. We are trying to flatten the curve due to our scarcity of resources and our efforts to minimize the population of infected people.

4. Do you observe that some $S_{v,c}$ compartments converged to zero while others converged to a positive value? Why do you think that is?

Answer: Yes, some $S_{v,c}$ compartments converged to zero while others converged to a positive value. This may have to do with the different compartmentalizations of the susceptible population. Some of these compartmentalized populations may never be infected by the end of the 120 days based on the interaction between social vulnerability and comorbidity.

5. Print the percentages of population that never got infected for all compartments (i.e. all values of v and c).

(defined as $\frac{S_{v,c}^{120}}{S_{v,c}^{0}}$ where $S_{v,c}^{0}$ is the initial value of $S_{v,c}$ and $S_{v,c}^{120}$ is the value of $S_{v,c}$ after 120 days).

[9]:
```
S120 = np.reshape(S[-1], (4,4))
percentages = S120 / S0 * 100

table = []
for i in range(4):
  for j in range(4):
    table.append([f'\033[0m S_{i}{j}', S0[i,j], S120[i,j], '{:.5f}%'.
 ↪format(percentages[i,j])])

df = pd.DataFrame(table)
df.columns = ['\033[1m S_vc', 'inital val', 'final val', 'uninfected %']
print(df.to_string(index=False))
```

```
 S_vc    inital val   final val uninfected %
 S_00  10925.597360  343.542915     3.14439%
 S_01   3292.424989  103.526539     3.14439%
 S_02   8158.351367  256.530029     3.14439%
 S_03  11564.660954  363.637539     3.14439%
 S_10   3081.935031    3.047159     0.09887%
 S_11   8490.458354    8.394653     0.09887%
 S_12   4517.306326    4.466333     0.09887%
 S_13   5703.815288    5.639454     0.09887%
 S_20   7452.562088    0.231693     0.00311%
 S_21   2489.328606    0.077391     0.00311%
 S_22  10210.700310    0.317441     0.00311%
 S_23   4245.926117    0.132002     0.00311%
 S_30   1601.800658    0.001566     0.00010%
 S_31   8089.091093    0.007908     0.00010%
 S_32   9135.041267    0.008930     0.00010%
 S_33    788.000191    0.000770     0.00010%
```

6. Multiply all the $\beta_{v,c}$ values by 1/4. What happened to the S, I, R plots? Did the "curve" flatten compared to the previous case? Print the percentages of population that never got infected with these $\beta_{v,c}$ values.

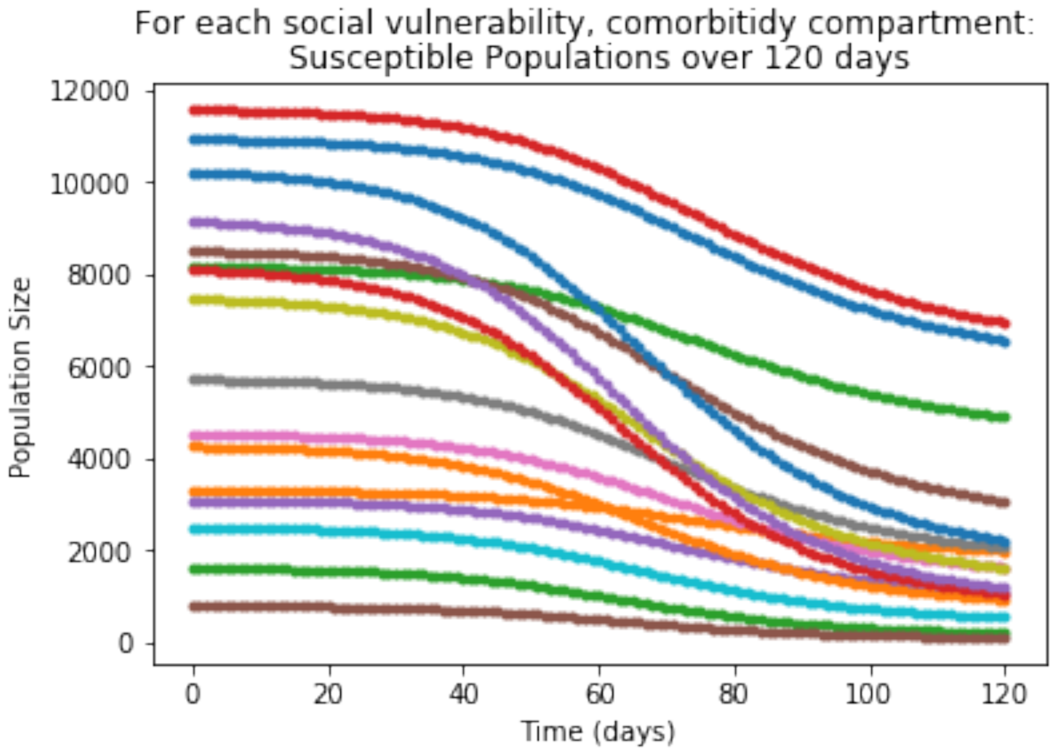
Answer: Yes, the infection "curve" flattened because the infection rate of the pathogen decreased by a factor of 4. The infection rate, β, is the reciprocal of transmission time, meaning a decreased β value indicates a higher transission time. We can see that this is shown through our plots. As a result, this drastically increases the percentage of compartmentalized populations that never got infected. Now, fewer compartments are converging to zero.

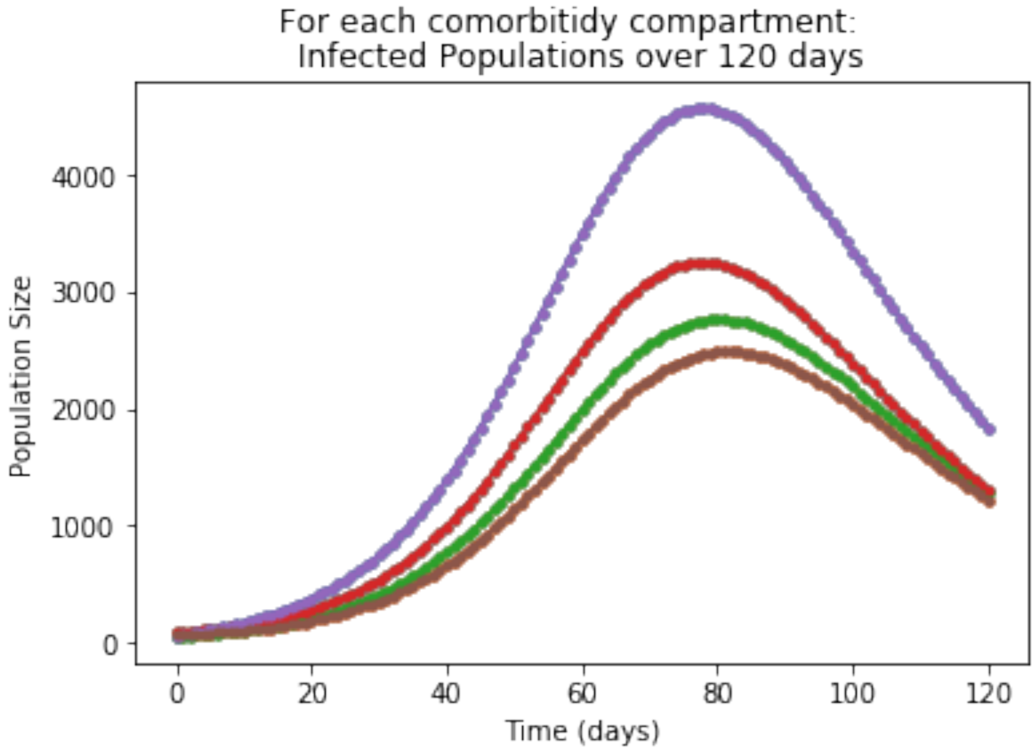
[10]:
```
# multiply betas by 1/4
beta = (1/4)*np.concatenate([np.repeat(0.25, 4), np.repeat(0.5, 4), np.repeat(0.
 ↪75, 4), np.repeat(1, 4)])
```

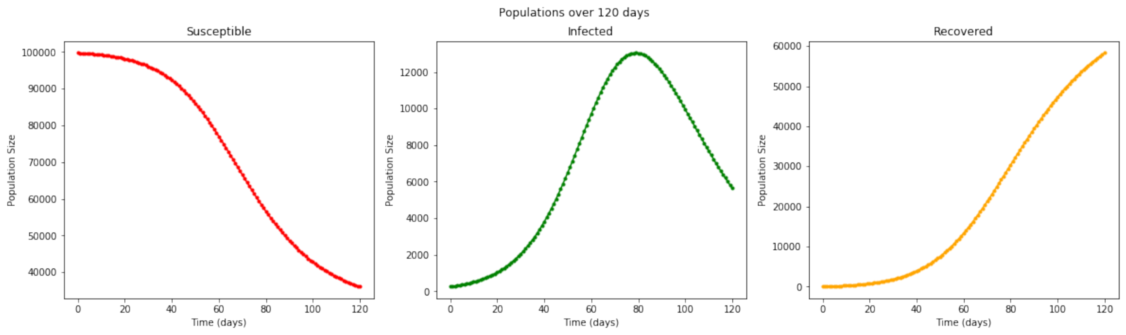
[11]:
```
# New beta values
soln = odeint(f, y0, t)
# soln contains the predicted S,I,R values for each of the 120 days (so the␣
 ↪length of soln is 120).
```

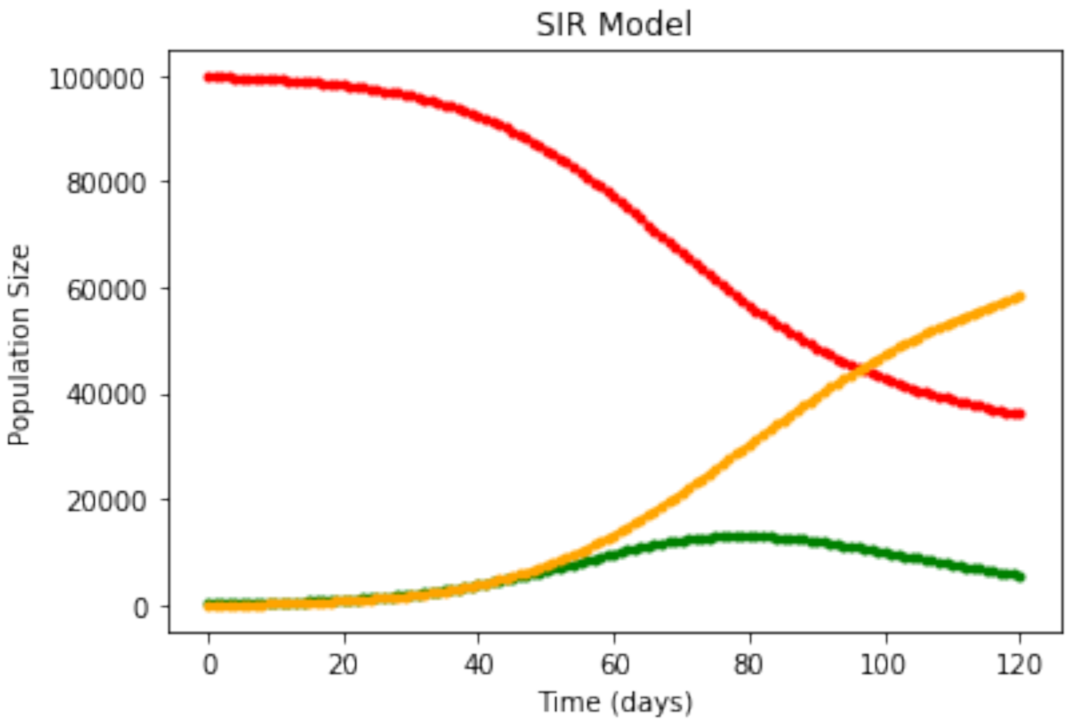
```
# soln = [[s00, s01, ..., snn, i1, i2, i3, i4, r], ...]

S = soln[:,:16]
I = soln[:,16:-1]
R = soln[:,-1]

S_total = np.sum(S, axis=1)
I_total = np.sum(I, axis=1)

# Viral Load Density
Lc = data['Lc']

# initial viral load density
L0 = np.sum(Lc*I0)/N
L = np.array([L0])

for i in range(1,len(soln)):
  L = np.append(L,np.sum(soln[i,16:-1]*Lc)/N)
```

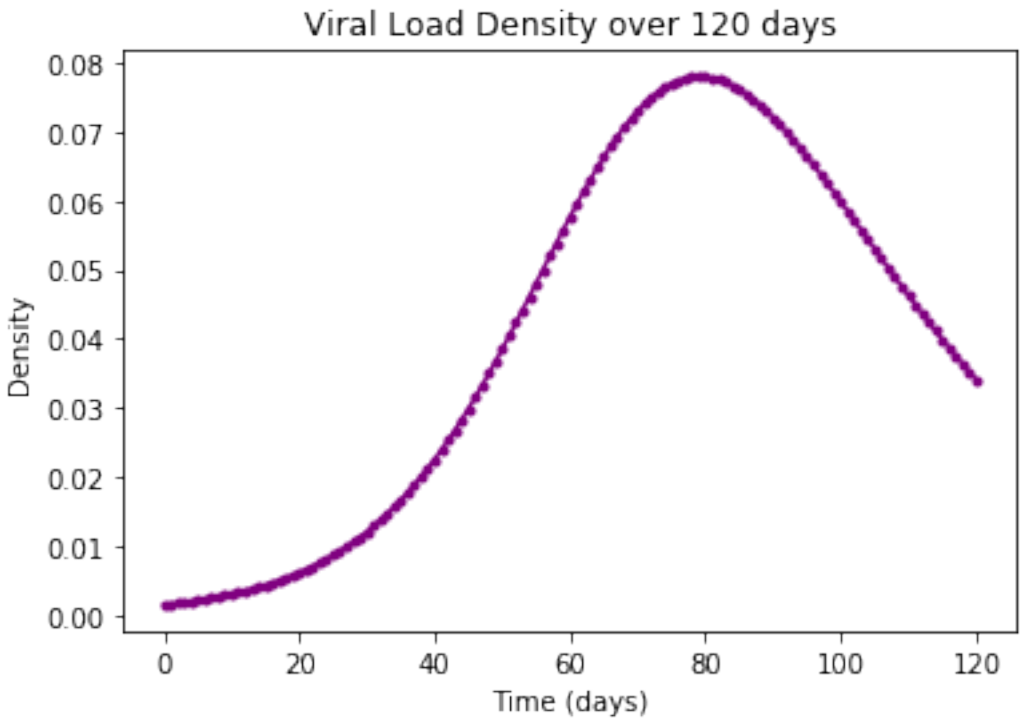
[12]:
```
# New beta values
# Susceptible populations over time (S_v,c)
plt.title('Susceptible Populations over 120 days')
plt.suptitle('For each social vulnerability, comorbitidy compartment:')
plt.ylabel('Population Size')
plt.xlabel('Time (days)')
plt.plot(t,S, marker='.')
plt.show();

# Infected populations over time (I_v,c)
plt.title('Infected Populations over 120 days')
plt.suptitle('For each comorbitidy compartment:')
plt.ylabel('Population Size')
plt.xlabel('Time (days)')
for i in range(16,20):
  plt.plot(t,I, marker='.')
plt.show();

# Overall S,I,R
plt_SIR(S_total, I_total, R, t, 'Populations over 120 days')

# Complete SIR
plt.title("SIR Model")
plt.ylabel('Population Size')
plt.xlabel('Time (days)')
plt.plot(t,S_total,marker = ".", color = 'red');
plt.plot(t,I_total,marker = ".", color = "green");
plt.plot(t,R,marker = ".", color = "orange");
plt.show();

# Viral Load Density over time (L)
plt_L(L,t)
```




For each comorbitidy compartment:
Infected Populations over 120 days
Population Size
Time (days)
0
1000
2000
3000
4000
0
20
40
60
80
100
120


Populations over 120 days
Susceptible
Infected
Recovered
Population Size
Time (days)


SIR Model
Population Size
Time (days)
100000
80000
60000
40000
20000
0
0
20
40
60
80
100
120


Viral Load Density over 120 days
Density
Time (days)
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
20
40
60
80
100
120

[13]:
```python
# New beta values
S120 = np.reshape(S[-1], (4,4))
percentages = S120 / S0 * 100

table = []
for i in range(4):
  for j in range(4):
    table.append([f'\033[0m S_{i}{j}', S0[i,j], S120[i,j], '{:.5f}%'.
 ↪format(percentages[i,j])])

df = pd.DataFrame(table)
df.columns = ['\033[1m S_vc', 'inital val', 'final val', 'uninfected %']
print(df.to_string(index=False))
```

```
 S_vc     inital val    final val uninfected %
 S_00   10925.597360  6560.288157    60.04512%
 S_01    3292.424989  1976.940569    60.04512%
 S_02    8158.351367  4898.691952    60.04512%
 S_03   11564.660954  6944.014666    60.04512%
 S_10    3081.935031  1111.165963    36.05417%
 S_11    8490.458354  3061.163925    36.05417%
 S_12    4517.306326  1628.677108    36.05417%
 S_13    5703.815288  2056.463016    36.05417%
 S_20    7452.562088  1613.387850    21.64877%
 S_21    2489.328606   538.908966    21.64877%
 S_22   10210.700310  2210.490785    21.64877%
 S_23    4245.926117   919.190680    21.64877%
 S_30    1601.800658   208.218532    12.99903%
 S_31    8089.091093  1051.503296    12.99903%
 S_32    9135.041267  1187.466663    12.99903%
 S_33     788.000191   102.432373    12.99903%
```

3 Part 1 (b)

Now suppose we don't know the model parameters $\beta_{v,c}$ and we are trying to estimate them from observed data—as would happen in the real world. To make the estimation of parameters easier, from all the leaf nodes the ones with uniform single social vulnerability are picked. So you are given 5 leaf nodes each for the four different values of social vulnerability in the part1b.npy file. The PMF of susceptible population $S_{v,c}^{(0)}$ in each comorbidity compartment areas follows: 5 Leaf nodes with vulnerability = 0.2: PMF of comoborbidity: (0.5, 0.3, 0.1, 0.1) for all nodes. 5 Leaf nodes with vulnerability = 0.4: PMF of comoborbidity: (0.4, 0.3, 0.2, 0.1) for all nodes. 5 Leaf nodes with vulnerability = 0.6: PMF of comoborbidity: (0.3, 0.3, 0.3, 0.2) for all nodes. 5 Leaf nodes with vulnerability = 0.8: PMF of comoborbidity: (0.1, 0.2, 0.3, 0.4) for all nodes.

[14]:
```python
data_b = np.load('part1b.npz') # keys: N, Lc, Ic_0, gamma, L_validation,␣
 ↪L_test, betas_validation
```

1. Using these distributions, total population sizes and observations of viral load densities L for 20 consecutive days (day 0 through day 19) for the 20 leaf nodes that are given to you, estimate the 16 parameters $\beta_{v,c}$ using grid search and MMSE on the "validation" data. In other words try to minimize MSE between Lvalidation and Lestimated. Compare the $\beta_{v,c}$ you estimated with the ground truth given in the file. This step is to make sure your code and logic works correctly.

[15]:
```python
L_val = data_b['L_validation']
N = data_b['N']
```

[16]:
```python
class SIRModel(object):
    def __init__(self, beta=None):
        # ODE parameters
        self.gamma = data_b['gamma']
        self.current_beta = beta # beta currently being considered in␣
 ↪simulation
        self.Lc = data_b['Lc']

        # training helpers
        self.beta_grid_4 = [np.linspace(0,1,21)] * 4
        self.min_mse = None
        self.optimal_beta = None # beta that minimizes mse
        self.total = 0

    def is_valid_beta_row(self, beta):
        # checks if a row [b0, b1, b3, b3] is valid
        return False not in (np.diff(beta) >= 0)

    def f(self, y, t):
        # dy/dt = f(y, t)
        Si = y[:4]
        Ii = y[4:-1]
        Ri = y[-1]
        I = np.sum(Ii)

        f0 = self.current_beta * Si * (-I/N)
        f1 = -1 * (f0 + self.gamma * Ii)
        f3 = self.gamma * I

        return np.concatenate((f0.flatten(), f1.flatten(), [f3]))

    def simulate(self, S_0, I_0, R_0, t):
        # run simulation given initial S,I,R values
        # returns a np.array of size (#nodes, #days, 9)
        # Example: solns[0] represents the 0th node.
        # Each row of solns[0] is a day. For each row, the first 4 values are␣
 ↪S, the next 4 are I, and the last is R.
        solns = []
        for i in range(5):
            y_ni = np.concatenate((S_0[i], I_0[i], [R_0[i]])) # initial value␣
↪of y for node i in SV 2
            soln = odeint(self.f, y_ni, t)
            solns.append(soln)

        return np.array(solns)

    def train(self, S_0, I_0, R_0, training_t, L_val):
        # calls self.simulate method on initial values for each point in the␣
↪beta grid
        # returns nothing but sets self.min_mse and self.optimal_beta after␣
↪training
        for beta in itertools.product(*self.beta_grid_4):
            self.current_beta = np.array(beta)
            if self.is_valid_beta_row(beta):
                solns = self.simulate(S_0, I_0, R_0, training_t)
                I = [soln[:,4:-1] for soln in solns]
                L_pred = np.array([np.sum(i * self.Lc, axis=1) for i in I])
                total = L_pred.shape[0] * L_pred.shape[1]
                mse = np.sum(np.square(L_pred - L_val)) / total

                if not self.min_mse or mse < self.min_mse:
                    self.min_mse = mse
                    self.optimal_beta = self.current_beta

                self.total += 1
```

[17]:
```
training_t = list(range(0, 20))
```

[18]:
```
### Social vulneratbility = 0.2
sv0 =  SIRModel()
I_0_0 = data_b['Ic_0'][:5]
S_0_0 = np.outer((N-I_0_0.sum(axis=1)),[0.5, 0.3, 0.1, 0.1]).round()
R_0_0 = np.zeros(5)
sv0.train(S_0_0, I_0_0, R_0_0, training_t, L_val[:5,:])
print('SV=0.2')
print(f'optimal beta is {sv0.optimal_beta}')
print(f'validation beta is {data_b["betas_validation"][0]}')
```

```
SV=0.2
optimal beta is [0.1  0.15 0.2  0.25]
validation beta is [0.1  0.15 0.2  0.25]
```

[19]:
```
### Social vulnerability = 0.4
sv1 =  SIRModel()
I_1_0 = data_b['Ic_0'][5:10,:]
S_1_0 = np.outer((N-I_1_0.sum(axis=1)),[0.4, 0.3, 0.2, 0.1]).round()
```

```
R_1_0 = np.zeros(5)
sv1.train(S_1_0, I_1_0, R_1_0, training_t, L_val[5:10,:])
print('SV=0.4')
print(f'optimal beta is {sv1.optimal_beta}')
print(f'validation beta is {data_b["betas_validation"][1]}')
```

```
SV=0.4
optimal beta is [0.2  0.25 0.3  0.4 ]
validation beta is [0.2  0.25 0.3  0.4 ]
```

[20]:
```
### Social vulnerability = 0.6
sv2 =  SIRModel()
I_2_0 = data_b['Ic_0'][10:15,:]
S_2_0 = np.outer((N-I_2_0.sum(axis=1)),[0.3, 0.3, 0.2, 0.2]).round()
R_2_0 = np.zeros(5)
sv2.train(S_2_0, I_2_0, R_2_0, training_t, L_val[10:15,:])
print('SV=0.6')
print(f'optimal beta is {sv2.optimal_beta}')
print(f'validation beta is {data_b["betas_validation"][2]}')
```

```
SV=0.6
optimal beta is [0.35 0.45 0.5  0.6 ]
validation beta is [0.35 0.45 0.5  0.6 ]
```

[21]:
```
### Social vulnerability = 0.8
sv3 =  SIRModel()
I_3_0 = data_b['Ic_0'][15:,:]
S_3_0 = np.outer((N-I_3_0.sum(axis=1)),[0.1, 0.2, 0.3, 0.4]).round()
R_3_0 = np.zeros(5)
sv3.train(S_3_0, I_3_0, R_3_0, training_t, L_val[15:,:])
print('SV=0.8')
print(f'optimal beta is {sv3.optimal_beta}')
print(f'validation beta is {data_b["betas_validation"][3]}')
```

```
SV=0.8
optimal beta is [0.4 0.5 0.6 0.8]
validation beta is [0.4 0.5 0.6 0.8]
```

2. Now use the “test” data (20 L values for 20 days) from the file and estimate the $\beta_{v,c}$. This time you won’t have access to the ground truth $\beta_{v,c}$values. Print the $\beta_{v,c}$ values you estimated.

[22]:
```
testing_t = list(range(0, 20))
L_test = data_b['L_test']
```

[23]:
```
### Social vulneratbility = 0.2
sv0 =  SIRModel()
sv0.train(S_0_0, I_0_0, R_0_0, testing_t, L_test[:5,:])
```

```python
print('SV=0.2')
print(f'optimal beta is {sv0.optimal_beta}')
```

```
SV=0.2
optimal beta is [0.05 0.1  0.15 0.25]
```

[24]:
```python
### Social vulnerability = 0.4
sv1 =  SIRModel()
sv1.train(S_1_0, I_1_0, R_1_0, testing_t, L_test[5:10,:])
print('SV=0.4')
print(f'optimal beta is {sv1.optimal_beta}')
```

```
SV=0.4
optimal beta is [0.2  0.25 0.3  0.4 ]
```

[25]:
```python
### Social vulnerability = 0.6
sv2 =  SIRModel()
sv2.train(S_2_0, I_2_0, R_2_0, testing_t, L_test[10:15,:])
print('SV=0.6')
print(f'optimal beta is {sv2.optimal_beta}')
```

```
SV=0.6
optimal beta is [0.4  0.45 0.5  0.6 ]
```

[26]:
```python
### Social vulnerability = 0.8
sv3 =  SIRModel()
sv3.train(S_3_0, I_3_0, R_3_0, testing_t, L_test[15:,:])
print('SV=0.8')
print(f'optimal beta is {sv3.optimal_beta}')
```

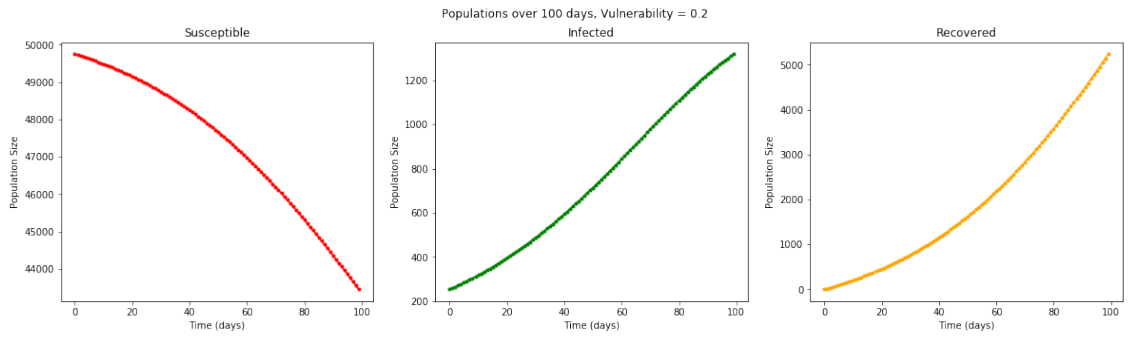
```
SV=0.8
optimal beta is [0.45 0.65 0.75 0.85]
```

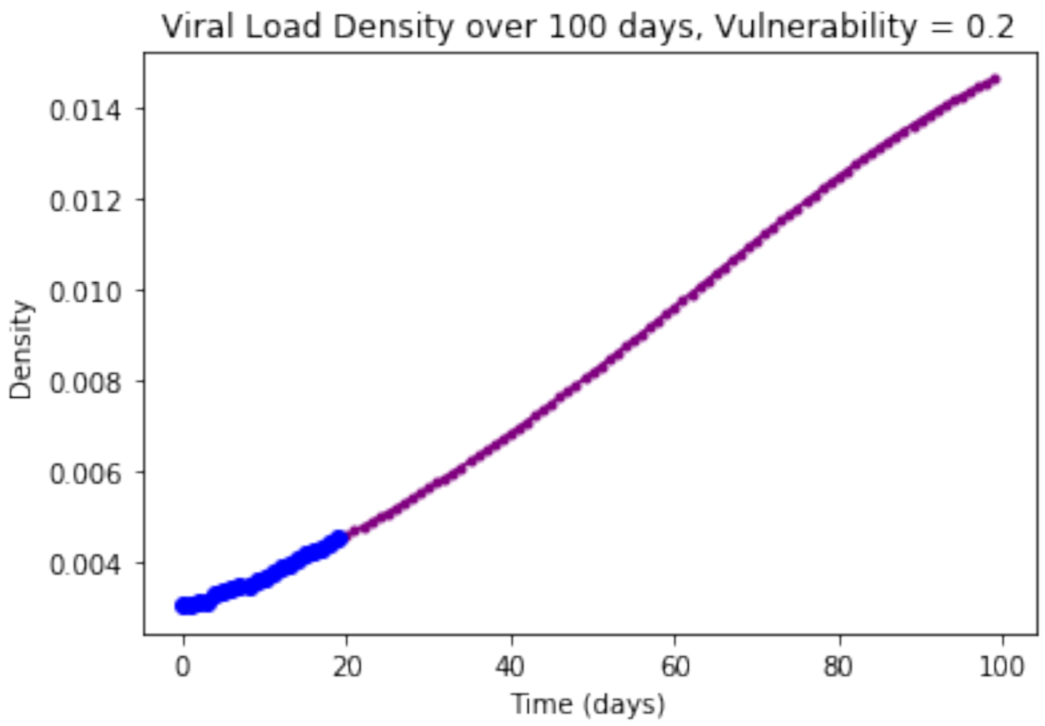
3. Use the $\beta_{v,c}$ you found to predict the disease behaviour for the future. Solve the equations for 100 days and plot S, I, R, L values for the first node in each social vulnerability case. On L graphs, also plot the corresponding observed values for the first 20 days (with a circle marker).

[27]:
```python
optimal_beta = sv0.optimal_beta
sv0_mod =  SIRModel(beta=optimal_beta)
soln0 = sv0_mod.simulate(S_0_0, I_0_0, R_0_0, list(range(0, 100)))
```

[28]:
```python
# helpers for plotting S, I, R, L values
def plot_SIR(S, I, R, t, title):
    fig, axs = plt.subplots(1, 3)
    fig.set_figheight(5)
    fig.set_figwidth(20)
    fig.suptitle(title)
    colors = ['red', 'green', 'orange']
    for i, data, title, color in zip(range(3), [S, I, R], ['Susceptible',␣
 ↪'Infected', 'Recovered'], colors):
        axs[i].set_title(title)
        axs[i].set_ylabel('Population Size')
        axs[i].set_xlabel('Time (days)')
        axs[i].plot(t, data, marker='.', color=color)

    plt.show()

def plot_L(L, t, L_test, t_test):
    plt.title('Viral Load Density over 100 days, Vulnerability = 0.2')
    plt.ylabel('Density')
    plt.xlabel('Time (days)')
    plt.plot(t, L,marker = ".", color = "purple");
    plt.plot(t_test, L_test, marker = "o", color = "blue")
    plt.show();
```

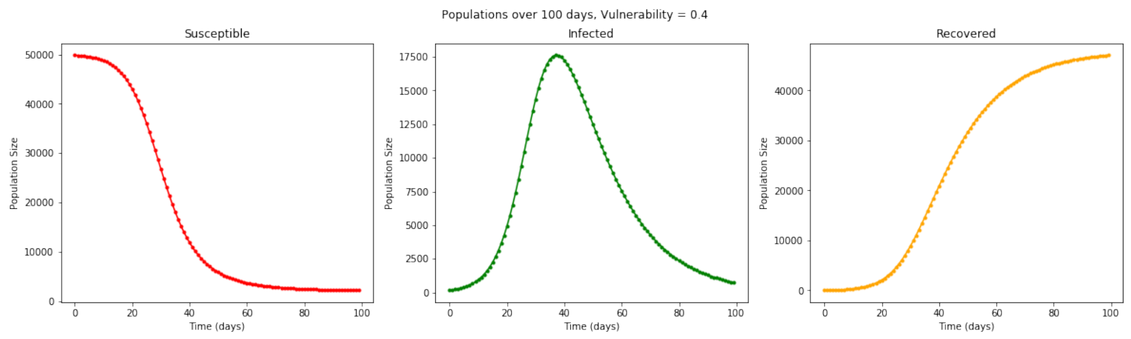
[29]:
```
# SV = 0.2
S_c = soln0[0][:,0:4]
I_c = soln0[0][:,4:8]
R = soln0[0][:,-1]
t = list(range(0,100))
S = np.sum(S_c, axis = 1)
I = np.sum(I_c, axis = 1)
L = np.array([])
for i in range(0,len(soln0[0])):
  L = np.append(L,np.sum(I_c[i]*data_b['Lc']/N))

plot_SIR(S, I, R, t, 'Populations over 100 days, Vulnerability = 0.2')
plot_L(L, t, L_test[0,:]/N, list(range(0,20)))
```

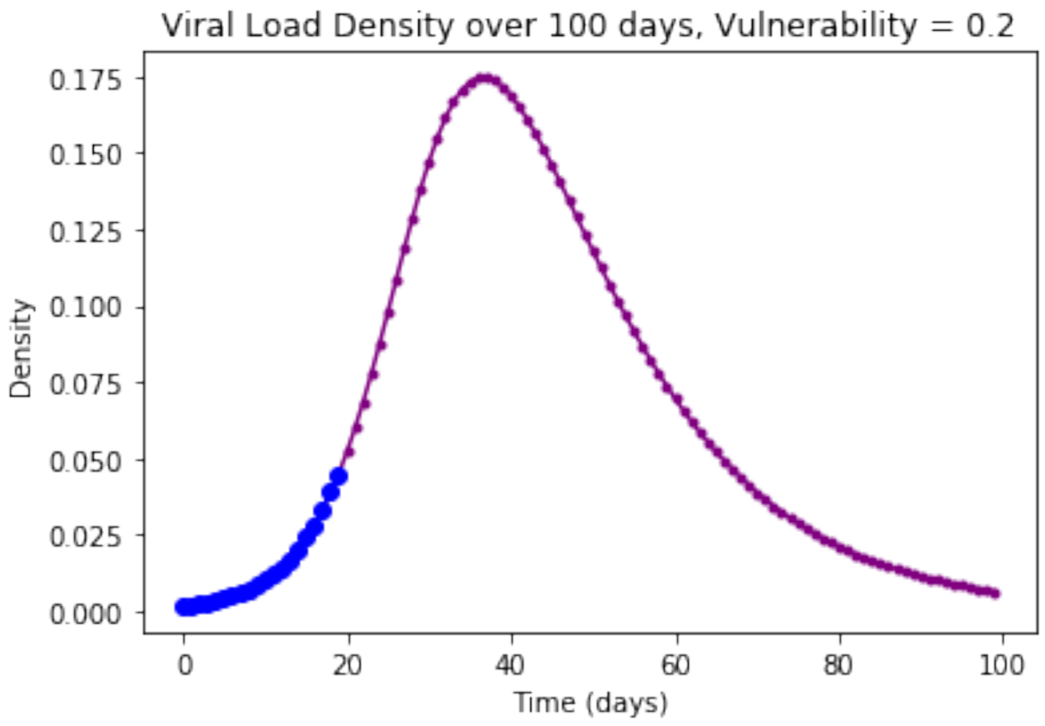




[30]:
```
optimal_beta = sv1.optimal_beta
sv1_mod =  SIRModel(beta=optimal_beta)
soln1 = sv1_mod.simulate(S_1_0, I_1_0, R_1_0, list(range(0, 100)))
```

[31]:
```
# SV = 0.4
S_c = soln1[0][:,0:4]
I_c = soln1[0][:,4:8]
R = soln1[0][:,-1]
t = list(range(0,100))
S = np.sum(S_c, axis = 1)
I = np.sum(I_c, axis = 1)
L = np.array([])
for i in range(0,len(soln0[0])):
  L = np.append(L,np.sum(I_c[i]*data_b['Lc'])/N)

plot_SIR(S, I, R, t, 'Populations over 100 days, Vulnerability = 0.4')
plot_L(L, t, L_test[5,:]/N, list(range(0,20)))
```

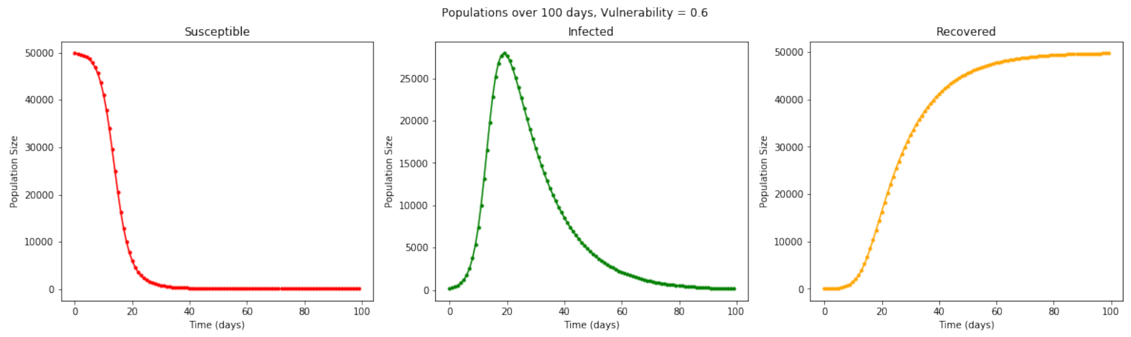


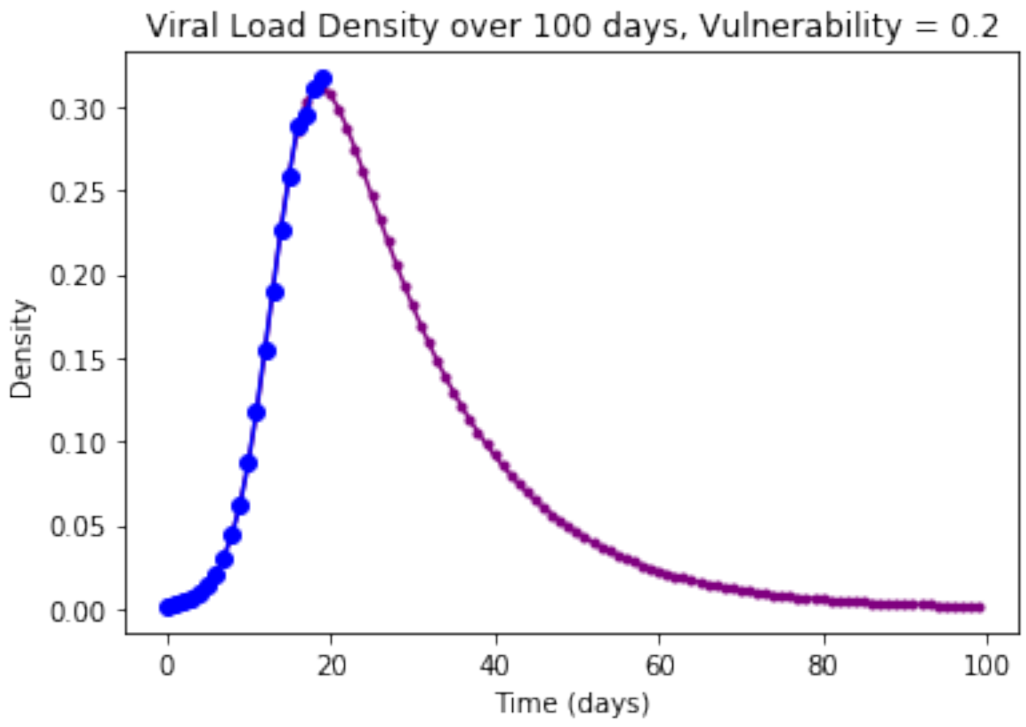


[32]:
```
optimal_beta = sv2.optimal_beta
sv2_mod =  SIRModel(beta=optimal_beta)
soln2 = sv2_mod.simulate(S_2_0, I_2_0, R_2_0, list(range(0, 100)))
```

[33]:
```
# SV = 0.6
S_c = soln2[0][:,0:4]
I_c = soln2[0][:,4:8]
R = soln2[0][:,-1]
t = list(range(0,100))
S = np.sum(S_c, axis = 1)
I = np.sum(I_c, axis = 1)
L = np.array([])
```

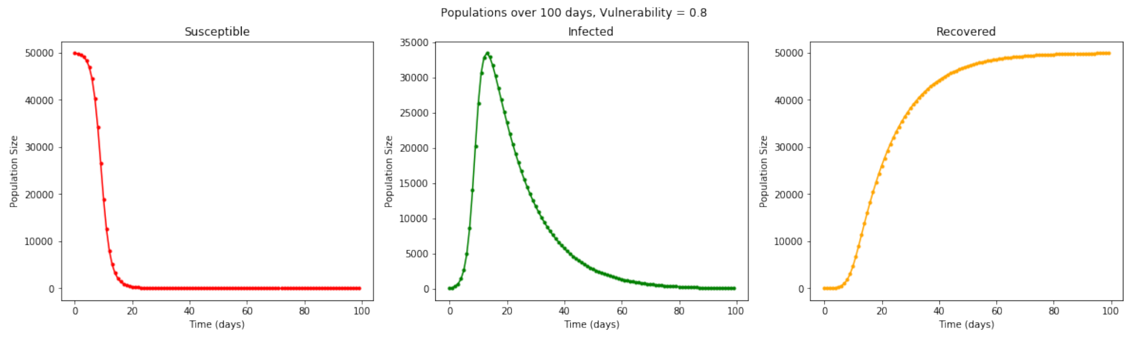
```
for i in range(0,len(soln0[0])):
  L = np.append(L,np.sum(I_c[i]*data_b['Lc'])/N)

plot_SIR(S, I, R, t, 'Populations over 100 days, Vulnerability = 0.6')
plot_L(L, t, L_test[10,:]/N, list(range(0,20)))
```

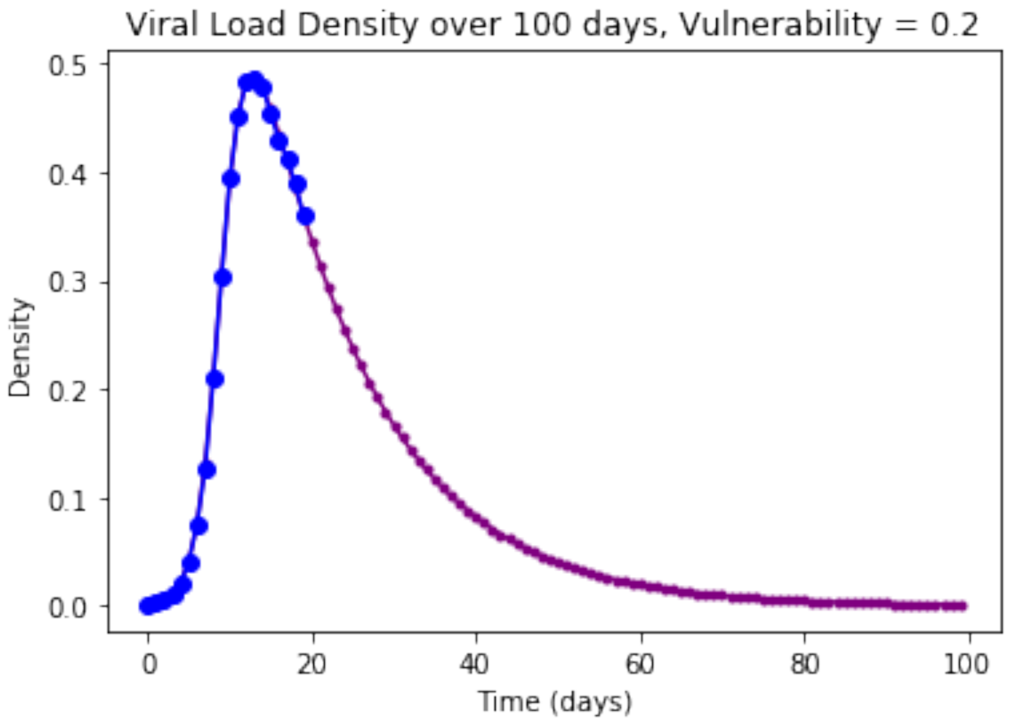




[34]:
```
optimal_beta = sv3.optimal_beta
sv3_mod =  SIRModel(beta=optimal_beta)
soln3 = sv3_mod.simulate(S_3_0, I_3_0, R_3_0, list(range(0, 100)))
```

[35]:
```
# SV = 0.8
S_c = soln3[0][:,0:4]
I_c = soln3[0][:,4:8]
R = soln3[0][:,-1]
t = list(range(0,100))
S = np.sum(S_c, axis = 1)
I = np.sum(I_c, axis = 1)
L = np.array([])
for i in range(0,len(soln0[0])):
  L = np.append(L,np.sum(I_c[i]*data_b['Lc'])/N)

plot_SIR(S, I, R, t, 'Populations over 100 days, Vulnerability = 0.8')
plot_L(L, t, L_test[15,:]/N, list(range(0,20)))
```


Populations over 100 days, Vulnerability = 0.8
Susceptible
Infected
Recovered
Population Size
Time (days)


Viral Load Density over 100 days, Vulnerability = 0.2
Density
Time (days)