4MB3 Asg 2

Zachary Levine and Jason Pekos

12/02/2021

Question One

Part A)

Let the total infectious period be the time between when an individual enters I_1 and leaves I_n. Assume that at time $t = 0$, we have I_{1_0} infectives in the first serially linked infectious compartment, I_1. Then, if we prevent contact with any susceptible individuals, the differential equation for I_1 becomes

$$\frac{dI}{dt} = n\gamma I_1$$

If we solve this differential equation using separation of variables, we obtain:

$$I_1(t) = I_{1_0} e^{-n\gamma t}$$

If at time t, $I_1(t) = I_{1_0} e^{-n\gamma t}$ individuals are in I_1, then after time t, the proportion of individuals in I_1 is reduced by a factor of $e^{-n\gamma t}$, so that the proportion of individuals who have an infectious period shorter than t is $1 - e^{-n\gamma t}$. Since this is the cumulative density function of the time an individual spends in I_1, the probability density function is the derivative of this function, or $n\gamma e^{-n\gamma t}$. The mean of this distribution is.

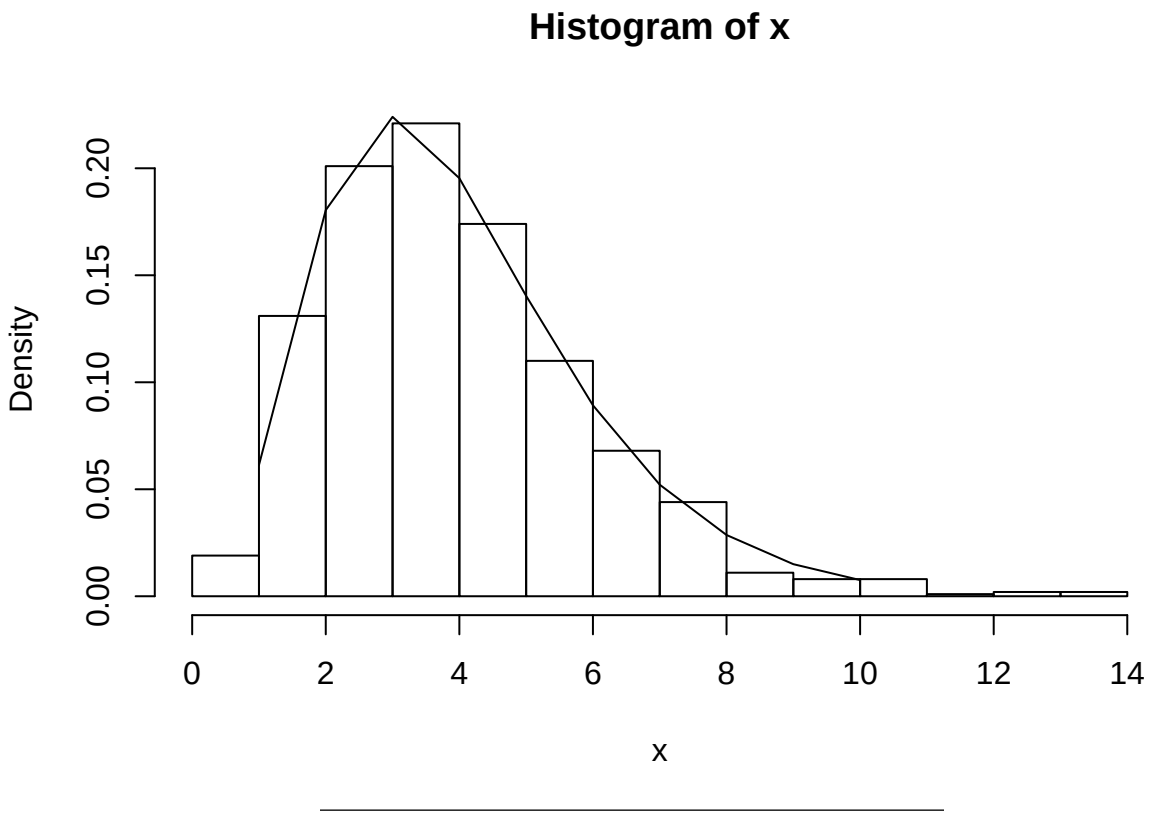
$$\int_0^\infty t n\gamma e^{-n\gamma t} dt = \frac{1}{\gamma n}$$

Part B)

So the mean time spent in I_1 is $\frac{1}{\gamma n}$ Since the removal rate for every infective compartment is the same, the average person should spend the same amount of time in each compartment. Thus, the total infectious period is $n\frac{1}{\gamma n} = \frac{1}{\gamma}$, which is unchanged from the standard SIR model with one infectious compartment.

Part C)

```
x <- numeric(1000)
for (i in 1:length(x)){
  x[i] <- sum(rexp(rate = 1, 4))
}


hist(x, freq =  FALSE)
lines(dgamma(1:10, shape = 4, rate = 1))
```



Question Two

Par A)

Reproduce Figure 2 from Shaw and Kennedy (2021) by coding the ODEs in R or Python and integrating them numerically. You don't need to get the colours exactly right.

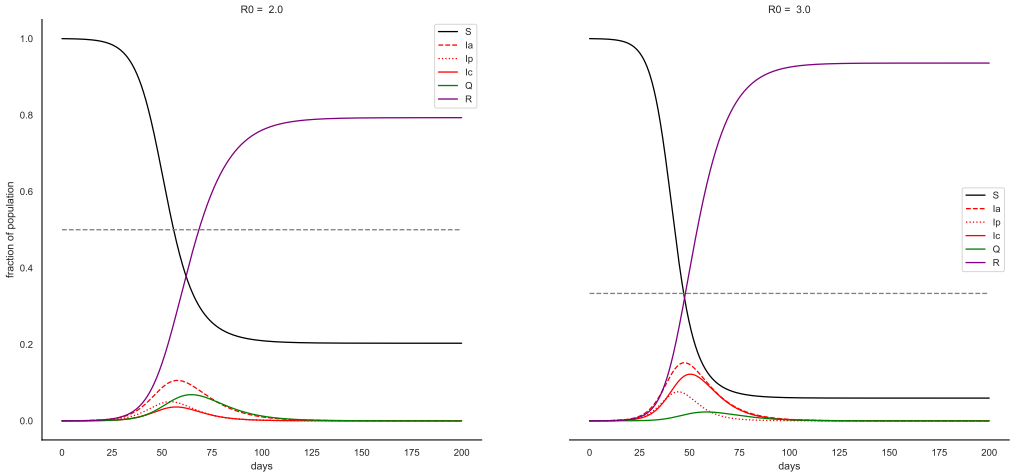
Code below:

```
import numpy as np
import pandas as pd
import scipy
import matplotlib.pyplot as plt
from scipy.integrate import odeint
import seaborn as sns
rc={'lines.linewidth': 2, 'axes.labelsize': 18, 'axes.titlesize': 18} #Graph formatting
sns.set(rc=rc) #update dict parameters


#Define gradient function for ode from Shaw Kennedy Paper
def ShawOde(y,t, betaA,betaP,betaC,phi,v,gammaA,gammaC,gammaQ,alphaC,alphaQ, lambdaVal):
    S, Ia, Ip, Ic, Q, R = y #Unpack state variables from initial condition

    #Gradients
    dydt = [-1*(betaA * Ia + betaP * Ip + betaC * Ic)* S,
            phi * (betaA * Ia + betaP * Ip + betaC * Ic)*S - (gammaA* Ia),
            (1-phi)*(betaA * Ia + betaP * Ip + betaC * Ic)*S - v*Ip,
            v*Ip - (alphaC + gammaC + lambdaVal) * Ic,
```

```
            lambdaVal * Ic - (alphaQ + gammaQ)*Q,
            gammaA * Ia +  gammaC* Ic + gammaQ * Q ]
    return dydt


#Calculate R0 for the Shaw-Kennedy system using the
#analytic formula derived from the survival function in the paper
def CalcR0(params):
    S = 1
    betaA,betaP,betaC,phi,v,gammaA,gammaC,gammaQ,alphaC,alphaQ, lambdaVal = params
    term1 = (phi*betaA*S)/(gammaA)
    term2 = ((1-phi)*(betaP)*S)/(v)
    term3 = ((1-phi)*(betaC)*S)/(lambdaVal + alphaC + gammaC)

    return(term1+term2+term3)



###PARAMS###
betaA = 0.12
betaP = 0.4
betaC = 0.4

phi = 0.5
v = 0.25

gammaA = 0.1
gammaC = 0.1
gammaQ = 0.1

alphaC = 0.001
alphaQ = 0.001

lambdaVal = 0.024

y0 = [1,0.0001,0.0001,0,0,0]
dt = 0.1

###Pack Params###
params = [betaA,betaP,betaC,phi,v,gammaA,gammaC,
gammaQ,alphaC,alphaQ, lambdaVal]

###Get R0 calculation for this set of parameters
R01 = CalcR0(params)

#create time arrays
t  = np.arange(0,200 + dt, dt )
t2 = np.arange(0,200 + dt, dt )


#solve with odeint /// using lsoda
solODEint   = odeint(ShawOde, y0, t, args =
(betaA,betaP,betaC,phi,v,gammaA,gammaC,gammaQ,alphaC,alphaQ, lambdaVal))
```

```
#Change Value of Lambda
lambdaVal = 0.23232323232323232323232323232

#Repack params
params = [betaA,betaP,betaC,phi,v,gammaA,
gammaC,gammaQ,alphaC,alphaQ, lambdaVal]

#Calculate R0 for new parameter values (Changed Lambda)
R02 = CalcR0(params)

#Re-solve system with new parameters
solODEint2   = odeint(ShawOde, y0, t, args =
(betaA,betaP,betaC,phi,v,gammaA,gammaC,gammaQ,alphaC,alphaQ, lambdaVal))


#Plotting
sns.set_style("whitegrid", {'axes.grid' : False}) #remove grey grid
sns.set_context("paper") #changes the theme (bunch of style parameters)
plt.rcParams['figure.figsize'] = [16, 7] #changes the size



#Put both plots next to eachother
fig, ax =plt.subplots(1,2)

#Plot all the groups in the SECOND model
ax[0].plot(t,solODEint2[:,0], label = "S",  color = "black")
ax[0].plot(t,solODEint2[:,1], label = "Ia", linestyle = "--", color = "red")
ax[0].plot(t,solODEint2[:,2], label = "Ip", linestyle = ":", color = "red")
ax[0].plot(t,solODEint2[:,3], label = "Ic", color = "red")
ax[0].plot(t,solODEint2[:,4], label = "Q", color = "green")
ax[0].plot(t,solODEint2[:,5], label = "R",  color = "purple")

#This adds the grey line from the Shaw Kennedy paper to the graph
ax[0].plot(t,(((1/R02))*np.ones(len(t))), color = "grey", linestyle = "--")


#plot all the groups in the FIRST model
plt.plot(t,solODEint[:,0], label = "S",  color = "black")
plt.plot(t,solODEint[:,1], label = "Ia", linestyle = "--", color = "red")
plt.plot(t,solODEint[:,2], label = "Ip", linestyle = ":", color = "red")
plt.plot(t,solODEint[:,3], label = "Ic", color = "red")
plt.plot(t,solODEint[:,4], label = "Q", color = "green")
plt.plot(t,solODEint[:,5], label = "R",  color = "purple")

#This adds the grey line from the Shaw Kennedy paper to the graph
plt.plot(t,(((1/R01))*np.ones(len(t))), color = "grey", linestyle = "--")

#Add legend and time axis to both
plt.legend(loc='best')
plt.xlabel('days')
plt.ylabel('fraction of population')
ax[0].legend(loc='best')
```

```
ax[0].set_xlabel('days')
ax[0].set_ylabel('fraction of population')

#Change some graph aesthetic to match the Shaw Kennedy paper:
ax[1].get_yaxis().set_visible(False)
sns.despine() #remove top and right spines for both
sns.despine(left=True, ax = ax[1]) #remove left spine for right grid
ax[0].spines['left'].set_color('black') #make spines black
ax[0].spines['bottom'].set_color('black')
ax[1].spines['bottom'].set_color('black')

#Set up title string to incorporate variable R_0
Title1 = f"R0 =  {round(R01,2)}"
Title2 = f"R0 =  {round(R02,2)}"

#Add title
ax[0].set_title(Title2)
ax[1].set_title(Title1)

###Final note: When testing, run the whole notebook! Otherwise, the change to
###lambda in this code
### section will make both graphs identical.
```



Part B

B) Change the model in some interesting way. See what changes about the model. Explain whether (and why) R0 does or doesn't change for your modified model, and derive the new value of R0 if it does. A few ideas: add vaccination

Our modification is adding a vaccine modification — simply a flow from the susceptable category straight to the recovered category – as a function of how many people remain susceptable.

Our new gradients:

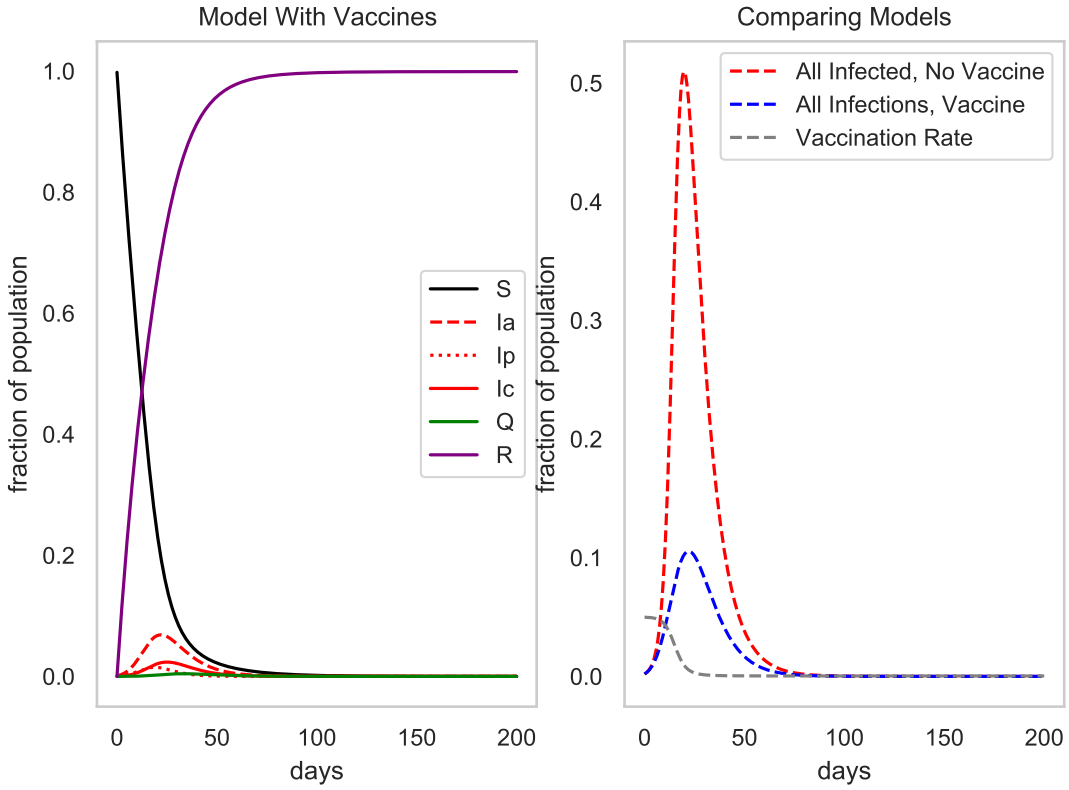
$$
\begin{aligned}
\frac{dS}{dt} &= -\left(\beta_A I_A + \beta_P I_P + \beta_C I_C\right) S - SVper \\
\frac{dI_A}{dt} &= \varphi\left(\beta_A I_A + \beta_P I_P + \beta_C I_C\right) S - \gamma_A I_A \\
\frac{dI_P}{dt} &= (1-\varphi)\left(\beta_A I_A + \beta_P I_P + \beta_C I_C\right) S - \nu I_P \\
\frac{dI_C}{dt} &= \nu I_P - \left(\alpha_C + \gamma_C + \lambda\right) I_C \\
\frac{dQ}{dt} &= \lambda I_C - \left(\alpha_Q + \gamma_Q\right) Q \\
\frac{dR}{dt} &= \gamma_A I_A + \gamma_C I_C + \gamma_Q Q + SVper
\end{aligned}
$$

The rate of vaccination is a function of the amount of susceptible people to maintain biological well-posedness.

```
def ShawOdeVaccineSimple(y,t, betaA,betaP,betaC,phi,v, VPer,
gammaA,gammaC,gammaQ,alphaC,alphaQ, lambdaVal):
    #This function defines the gradients for a version of the new model
    #where you vaccinate as a function of susceptable people remaining
    S, Ia, Ip, Ic, Q, R = y #unpack state variables from initial condition

    dydt = [-1*(betaA * Ia + betaP * Ip + betaC * Ic)* S - S*VPer,
            phi * (betaA * Ia + betaP * Ip + betaC * Ic)*S - (gammaA* Ia),
            (1-phi)*(betaA * Ia + betaP * Ip + betaC * Ic)*S - v*Ip,
            v*Ip - (alphaC + gammaC + lambdaVal) * Ic,
            lambdaVal * Ic - (alphaQ + gammaQ)*Q,
            gammaA * Ia +  gammaC* Ic + gammaQ * Q + S*VPer ]
    return dydt

def ShawOdeVaccineSimple2(y,t, betaA,betaP,betaC,phi,v, VPer,
gammaA,gammaC,gammaQ,alphaC,alphaQ, lambdaVal):
    #This function defines the gradients for a version of the new model
    #where you vaccinate as a function of total infected people in the population
    S, Ia, Ip, Ic, Q, R = y #unpack state variables from initial condition

    dydt = [-1*(betaA * Ia + betaP * Ip + betaC * Ic)* S - (Ic+Ia+Ip)*VPer,
            phi * (betaA * Ia + betaP * Ip + betaC * Ic)*S - (gammaA* Ia),
            (1-phi)*(betaA * Ia + betaP * Ip + betaC * Ic)*S - v*Ip,
            v*Ip - (alphaC + gammaC + lambdaVal) * Ic,
            lambdaVal * Ic - (alphaQ + gammaQ)*Q,
            gammaA * Ia +  gammaC* Ic + gammaQ * Q + (Ic+Ia+Ip)*VPer ]
    return dydt


###PARAMS###
#Transmisibility Params
betaA = 0.3
betaP = 0.8
betaC = 0.8

phi = 0.7

v = 0.25
```

```
VPer = 0.05 #Percent of population vaccinated at each
            #timestep assuming everyone is infected

gammaA = 0.1
gammaC = 0.1
gammaQ = 0.1

alphaC = 0.001
alphaQ = 0.001

lambdaVal = 0.024

y0 = [0.998,0.001,0.001,0,0,0]
dt = 0.01

#create time array
t  = np.arange(0,200 + dt, dt )
t2 = np.arange(0,200 + dt, dt )

#solve with odeint /// using lsoda
solODEintVaccine   = odeint(ShawOdeVaccineSimple, y0, t,
args = (betaA,betaP,betaC,phi,v, VPer,
gammaA,gammaC,gammaQ,alphaC,alphaQ, lambdaVal))

#Get solution with initial model also, with these same params, so we can compare
solODEint   = odeint(ShawOde, y0, t, args =
(betaA,betaP,betaC,phi,v,gammaA,gammaC,gammaQ,alphaC,alphaQ, lambdaVal))

#Plot
fig, ax =plt.subplots(1,2)
figsize=(10,20)

#Plot on right; this is simply the total infected people in the case
#of a vaccine and in the
#case of no vaccine
plt.plot(t,solODEint[:,1] + solODEint[:,2] + solODEint[:,3],
label = "All Infected, No Vaccine", linestyle = "--", color = "red")
plt.plot(t,solODEintVaccine[:,1] + solODEintVaccine[:,2] + solODEintVaccine[:,3],
label = "All Infections, Vaccine", linestyle = "--", color = "blue")
plt.plot(t,(solODEint[:,0] )*VPer, label = "Vaccination Rate",
linestyle = "--", color = "grey")

#Plot on left; this is simply the dynamics of the vaccine model
ax[0].plot(t,solODEintVaccine[:,0], label = "S",  color = "black")
ax[0].plot(t,solODEintVaccine[:,1], label = "Ia",
linestyle = "--", color = "red") #style is to match the paper
ax[0].plot(t,solODEintVaccine[:,2], label = "Ip", linestyle = ":", color = "red")
ax[0].plot(t,solODEintVaccine[:,3], label = "Ic", color = "red")
ax[0].plot(t,solODEintVaccine[:,4], label = "Q", color = "green")
ax[0].plot(t,solODEintVaccine[:,5], label = "R",  color = "purple")

#Adding legends and axis labels
plt.legend(loc='best')
```

```
plt.xlabel('days')
plt.ylabel('fraction of population')
ax[0].set_title("Model With Vaccines")
ax[1].set_title("Comparing Models")
ax[0].legend(loc='best')
ax[0].set_xlabel('days')
ax[0].set_ylabel('fraction of population')
plt.show()
```



Calculating R0 using the Next Generation Matrix method — as outlined in a 2017 paper by Pauline van den Driessche — first we look at the Shaw Kennedy model.

The gradients for the infected compartments are given by:

$$\begin{aligned}
\frac{dI_A}{dt} &= \varphi\left(\beta_A I_A + \beta_P I_P + \beta_C I_C\right) S - \gamma_A I_A \\
\frac{dI_P}{dt} &= (1-\varphi)\left(\beta_A I_A + \beta_P I_P + \beta_C I_C\right) S - \nu I_P \\
\frac{dI_C}{dt} &= \nu I_P - \left(\alpha_C + \gamma_C + \lambda\right) I_C \\
\frac{dQ}{dt} &= \lambda I_C - \left(\alpha_Q + \gamma_Q\right) Q
\end{aligned}$$

We split this into two operators, F and V, which are column vectors representing flow into and out of the compartments (rows) respectively, such that $F - V$ recovers the original system. Next we take the derivative matrix of each, recovering:

$$dF = \begin{bmatrix} \phi\beta_a S & \phi\beta_p S & \phi\beta_c S & 0 \\ (1-\phi)(\beta_a)S & (1-\phi)(\beta_p)S & (1-\phi)(\beta_c)S & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$dV = \begin{bmatrix} \gamma_a & 0 & 0 & 0 \\ 0 & v & 0 & 0 \\ 0 & -v & \alpha_c + \gamma_c + \lambda & 0 \\ 0 & 0 & -\lambda & \alpha_q + \gamma_q \end{bmatrix}$$

Computing FV^{-1} and looking at the largest eigenvalue, we find R_0 for our system.

This is implemented numerically below, showing this matches the values in the paper.

```
def NGRShaw(params, y):
    betaA,betaP,betaC,phi,v,gammaA,gammaC,gammaQ,alphaC,alphaQ, lambdaVal = params
    S, Ia, Ip, Ic, Q, R = y

    ###Calculate Flow in Mat (F)
    dIaIa = phi * (betaA)*S
    dIaIp = phi * (betaP)*S
    dIaIc = phi * (betaC)*S
    dIaQ  = 0

    dIpIa = (1 - phi) * (betaA)* S
    dIpIp = (1 - phi) * (betaP)* S
    dIpIc = (1 - phi) * (betaC)* S
    dIpQ  = 0

    dIcIa = 0
    dIcIp = 0
    dIcIc = 0
    dIcQ  = 0

    dQIa = 0
    dQIp = 0
    dQIc = 0
    dQQ  = 0

    matIn = np.matrix([
           [ dIaIa, dIaIp,dIaIc,dIaQ ],
           [ dIpIa, dIpIp,dIpIc,dIpQ ],
           [ dIcIa, dIcIp,dIcIc,dIcQ ],
           [ dQIa, dQIp,dQIc,dQQ ]])


    ###V (flow out) Mat
    dIaIa = (gammaA)
    dIaIp = 0
    dIaIc = 0
    dIaQ  = 0

    dIpIa = 0
    dIpIp = (v)
```

```
    dIpIc = 0
    dIpQ  = 0

    dIcIa = 0
    dIcIp = -v
    dIcIc = alphaC + gammaC + lambdaVal
    dIcQ  = 0

    dQIa = 0
    dQIp = 0
    dQIc = -lambdaVal
    dQQ  = +alphaQ + gammaQ

    matOut = np.matrix([
           [ dIaIa, dIaIp,dIaIc,dIaQ ],
           [ dIpIa, dIpIp,dIpIc,dIpQ ],
           [ dIcIa, dIcIp,dIcIc,dIcQ ],
           [ dQIa, dQIp,dQIc,dQQ ]])

    return(matIn, matOut)
```

Testing this, we see:

```
#Restate parameters for the Shaw Kennedy model
betaA,betaP,betaC,phi,v,gammaA,gammaC,gammaQ,alphaC,alphaQ = [0.12,0.4,0.4,0.5,0.25,
0.1,0.1,0.1,0.001,0.001]
lambdaVal = 0.024
y0 = [1,0.0001,0.0001,0,0,0]
dt = 0.1
params = [betaA,betaP,betaC,phi,v,gammaA,gammaC,
gammaQ,alphaC,alphaQ, lambdaVal]#Pack Parameters

###R0 using the intial formula given in the paper derived from the survival function
R01 = CalcR0(params)

###Calculate
GetNGRop = NGRShaw(params,y0)
print(max(abs(np.linalg.eigvals(np.matmul(GetNGRop[0],np.linalg.inv(GetNGRop[1]))))))
#R0 from next gen matrix
```

```
## 3.0000000000000004
```

```
print(R01) #R0 from formula in the Shaw Kennedy paper
```

```
## 3.0
```

Where the small difference is due to NLA issues.

Now, consider the modified system:

$$
\begin{aligned}
\frac{dS}{dt} &= -\left(\beta_A I_A + \beta_P I_P + \beta_C I_C\right) S - SVper \\
\frac{dI_A}{dt} &= \varphi\left(\beta_A I_A + \beta_P I_P + \beta_C I_C\right) S - \gamma_A I_A \\
\frac{dI_P}{dt} &= (1-\varphi)\left(\beta_A I_A + \beta_P I_P + \beta_C I_C\right) S - \nu I_P \\
\frac{dI_C}{dt} &= \nu I_P - \left(\alpha_C + \gamma_C + \lambda\right) I_C \\
\frac{dQ}{dt} &= \lambda I_C - \left(\alpha_Q + \gamma_Q\right) Q \\
\frac{dR}{dt} &= \gamma_A I_A + \gamma_C I_C + \gamma_Q Q + SVper
\end{aligned}
$$

Note that the vaccine flow does not change the gradients for the infected categories, and so the two models have identical Next Generation Matrices, and therefore they have identical R0 values. This is confirmed by numerical simulations of both models, showing that initial rate of growth is equivalent on a timestep of 0.1 by over five significant figures:

```
print("Slope, Vaccine, Same Params:",((solODEintVaccine[:,1]
+ solODEintVaccine[:,2] +
solODEintVaccine[:,3])[1] - (solODEintVaccine[:,1] + solODEintVaccine[:,2] +
solODEintVaccine[:,3])[0])/dt)
```

```
## Slope, Vaccine, Same Params: 9.993466563598761e-05
```

```
print("Slope, No Vaccine, Same Params:",((solODEint[:,1] + solODEint[:,2]
+ solODEint[:,3])[1]
- (solODEint[:,1] + solODEint[:,2] + solODEint[:,3])[0])/dt)
```

```
## Slope, No Vaccine, Same Params: 9.996230736975496e-05
```

```
print("Slope, Slightly Different Params :",((solODEint2[:,1] + solODEint2[:,2] +
solODEint2[:,3])[1] - (solODEint2[:,1] + solODEint2[:,2] +
solODEint2[:,3])[0])/dt)
```

```
## Slope, Slightly Different Params : 4.212497788830656e-05
```

Question Three

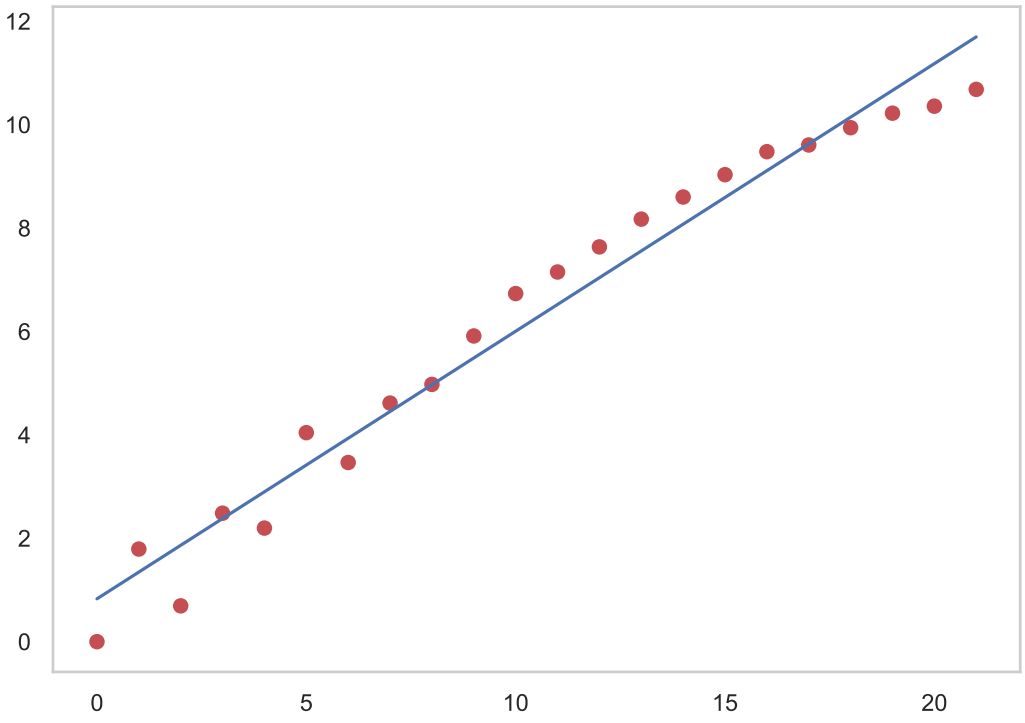
Fit a linear model to the log-death series as a function of time during the exponentially increasing phase of the epidemic to estimate r. Choose a subset of data that seems reasonable to you for this fit. Make sure to include the units of r in your answer! What is the doubling time implied by your estimate?

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm
from sklearn.linear_model import LinearRegression
from scipy import stats
from scipy.integrate import odeint
from sklearn.metrics import mean_squared_error
import seaborn as sns
```

```
rc={'lines.linewidth': 2, 'axes.labelsize': 18, 'axes.titlesize': 18}
sns.set(rc=rc)
sns.set_style("whitegrid", {'axes.grid' : False}) #remove grey grid
sns.set_context("paper")
#changes the theme or the size or something ... not sure why I added this.
plt.rcParams['figure.figsize'] = [16, 7] #changes the size


plague = pd.read_csv("great_plague.csv") #read in data
plague['log deaths'] = np.log(plague['plague_deaths']) #add log deaths column to datafram



cutoff = 21 #what week do we analyze up to?
X = plague.iloc[:cutoff+1, 1].values.reshape(-1, 1)   #Get weeks column
Y = plague.iloc[:cutoff+1, 3].values.reshape(-1, 1)   #Get log-deaths column
model = LinearRegression()   #Instantiate object from class
model.fit(X, Y)   #Fit model
```

```
## LinearRegression()
```

```
Y_pred = model.predict(X)   #Get trendline
plt.plot(Y, 'ro') #plot data
plt.plot(Y_pred)  #plot trendline
plt.show()
```



```
r  = model.coef_[0][0] #r = growth coefficient


print("r is given by",r, "1/weeks")
```

```
## r is given by 0.5175972425439678 1/weeks
```

```
print("Doubles every:",np.log(2) / r, "weeks")
```

```
## Doubles every: 1.3391632017843782 weeks
```

Implied doubling time is given by the output above.

B) The generation time/infectious period for plague is approximately 2.5 weeks. Combine this information with your r estimate to get a rough estimate of contact rate β, removal rate γ, and R_0 for plague in London in 1665.

```
k = 2.5
gamma = 1/k
beta = r + gamma
R0 = beta/gamma

print("Gamma:", gamma)
```

```
## Gamma: 0.4
```

```
print("Beta:", beta )
```

```
## Beta: 0.9175972425439678
```

```
print("R0:", R0)
```

```
## R0: 2.2939931063599195
```

ϕ, the death rate, is 0.4, and the population is 130000 people.

```
phi = 0.4
N = 130000
dt = 1
```

parameterize the incidence term in your model as β/NSI, set initial conditions:

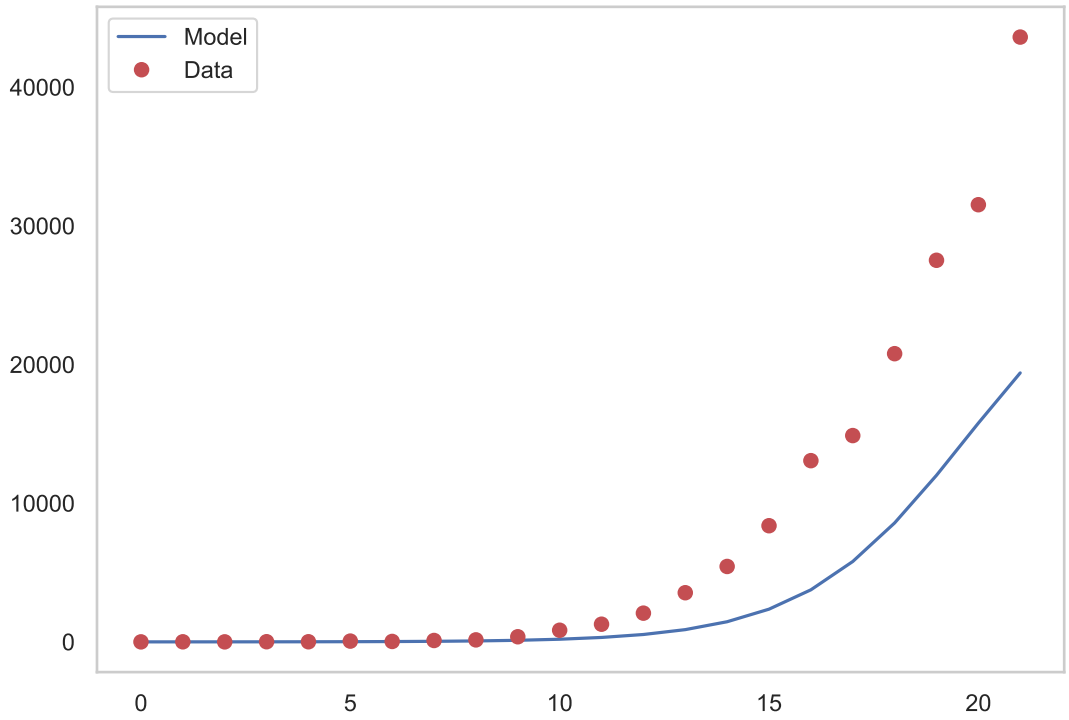
```
S0 =N
I0 = 1/(phi*beta)
y0 = [N,1/(phi*beta),0]

def SIRode(y,t, beta, gamma):
    S,I,R = y[0],y[1],y[2]
    dydt = [ (-beta/N*(N-(I))*(I)) , beta/N*(N-(I))*(I) - gamma*I, gamma*I ]
    return dydt
```

Numerically solve the ODE for your initial guess. Plot the time series of deaths and superimpose the curve of $\phi\frac{\beta}{NSI}$ from your solution. Do not expect them to match exactly; the point of all the work above was to get a solution that was at least on the right order of magnitude.d fitting

Plotting predicted deaths vs realized deaths

```
###Test###
t = np.arange(0,cutoff + dt, dt )
sol = odeint(SIRode, y0, t, args = (beta,gamma))
```

```
plt.plot(t,phi*sol[:,1], label = "Model")
plt.plot(plague['plague_deaths'][:cutoff+1], 'ro',label = "Data");
plt.legend(loc='best');
plt.show()
```

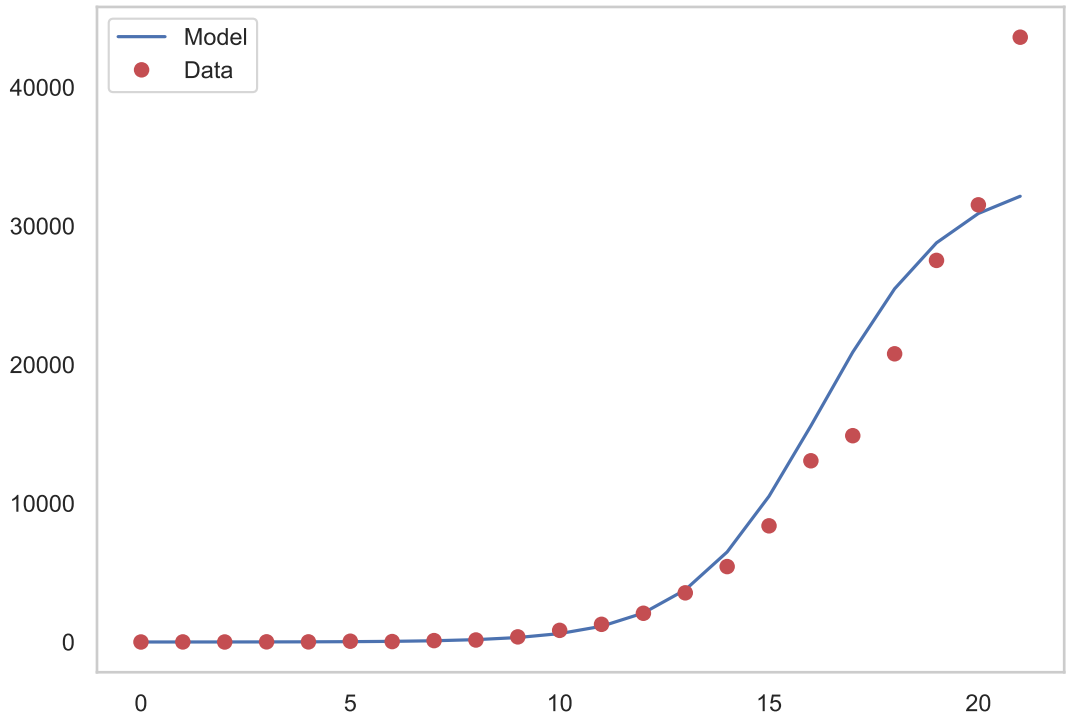


Adjust the values:

It undershoots because we estimated r from the log linear model on deaths — estimated from only deaths with no fractional scalar representing mortaility rate — and so we predict fit on deaths, predict cases from deaths without changing any numbers, and then multiply cases by $\phi = 0.4$. This was what was asked of us, but it will undershoot every single time. Increasing our estimate of r to account for this, a second guess is:

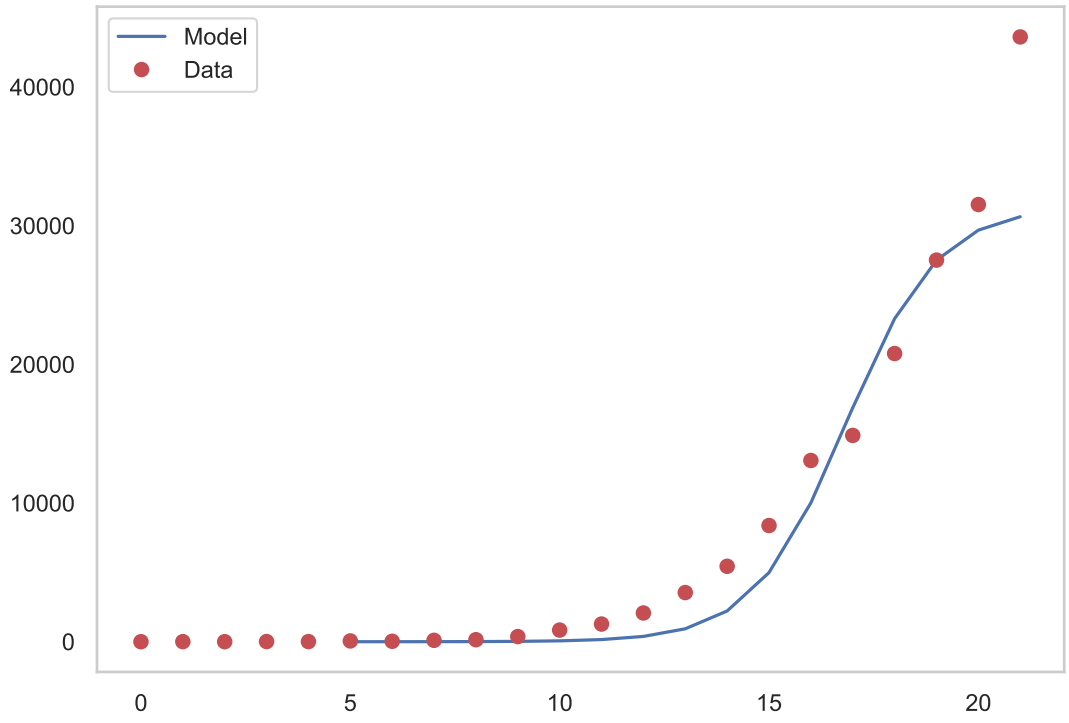
```
###PARAMS###
N = 130000
k = 2.5
gamma = 0.35
beta = 0.99175972425439678

R0 = beta/gamma
y0 = [N,1/(0.4*beta),0]
dt = 1


###Test###
t = np.arange(0,cutoff + dt, dt )
sol = odeint(SIRode, y0, t, args = (beta,gamma))
plt.plot(t,phi*sol[:,1], label = "Model")
```

```
plt.plot(plague['plague_deaths'][:cutoff+1], 'ro',label = "Data")
plt.legend(loc='best')
plt.show()
```



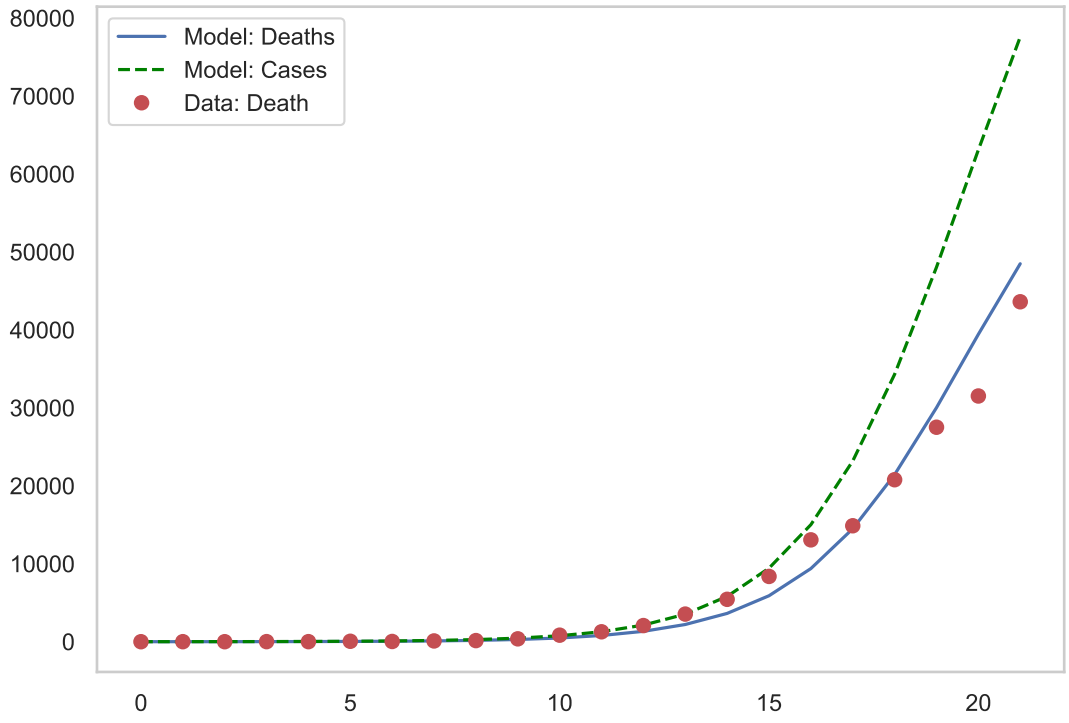
Starting the model `lag` days late and again increasing r:

```
###PARAMS###
N = 130000
k = 2.5
gamma = 0.6
beta = 1.5099175972425439678

R0 = beta/gamma
y0 = [N,1/(0.4*beta),0]
dt = 1


###Test###
lag = 5
t = np.arange(lag,cutoff  + dt, dt )
sol = odeint(SIRode, y0, t, args = (beta,gamma))
plt.plot(t,phi*sol[:,1], label = "Model")
plt.plot(plague['plague_deaths'][:cutoff+1], 'ro',label = "Data")
plt.legend(loc='best')
plt.show()
```



Switching instead to match fit on deaths and recover cases by multiplying by $2 - \phi$, removing time lag:

```
###PARAMS###
N = 130000
k = 2.5
gamma = 0.4
beta =  0.9175972425439678

R0 = beta/gamma
y0 = [N,1/(0.4*beta),0]
dt = 1


###Test###
lag = 0
t = np.arange(0,cutoff  + dt, dt )
sol = odeint(SIRode, y0, t, args = (beta,gamma))
plt.plot(t,sol[:,1], label = "Model: Deaths")
plt.plot(t,(2-phi)*sol[:,1],'--', label = "Model: Cases", color = "green")

plt.plot(plague['plague_deaths'][:cutoff+1], 'ro',label = "Data: Death")
plt.legend(loc='best')
plt.show()
```



Build a very simple optimizer to look over the space of parameters close to our choices, and to choose the ones that minimize MSE.

```
#Simple Optimizer
Iter = 100

bestBeta = beta
bestGamma = gamma
betaInit = beta
gammaInit = gamma
mintolBeta = -0.1
maxtolBeta =  0.1
mintolGamma = - 0.1
maxtolGamma = 0.1
mseArr = []
TotalArray = []
MSEInit = mean_squared_error(np.array(
plague['plague_deaths'][lag:cutoff+1]),sol[:,1])

for i in range(0,Iter):
    for j in range(0,Iter):

        beta  = betaInit  * (1+np.random.uniform(mintolBeta,maxtolBeta,1)[0])
        gamma = gammaInit * (1+np.random.uniform(mintolGamma,maxtolGamma,1)[0])

        y0 = [N,1/(0.4*beta),0]
```

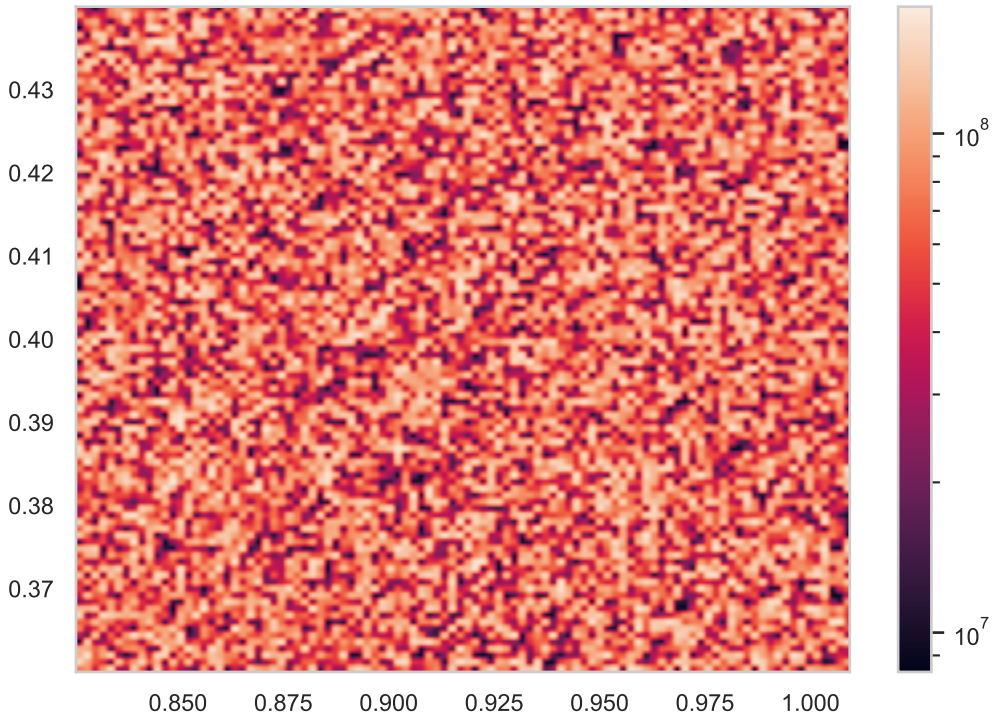
```
        sol = odeint(SIRode, y0, t, args = (beta,gamma))
        MSE = mean_squared_error(np.array(plague['plague_deaths'][lag:cutoff+1]),
        phi*sol[:,1])
        TotalArray.append([beta, gamma, MSE])

        mseArr.append(MSEInit)

        if MSE < MSEInit:
            MSEInit = MSE
            bestBeta = beta
            bestGamma = gamma
```

Is there a pattern we can use to quickly build a better optimizer? Looking at the MSE over the space of possible parameter combinations for Beta(x-axis) and Gamma(y-axis) that stay close enough that they don't totally lose our r value.

```
arr = np.array(TotalArray).T
zl = arr[2]
yl = arr[1]
xl = arr[0]

N = int(len(zl)**.5)
z = zl.reshape(N, N)
plt.imshow(z, extent=(np.amin(xl), np.amax(xl), np.amin(yl),
np.amax(yl)), norm=LogNorm(), aspect = 'auto')
plt.colorbar()
```

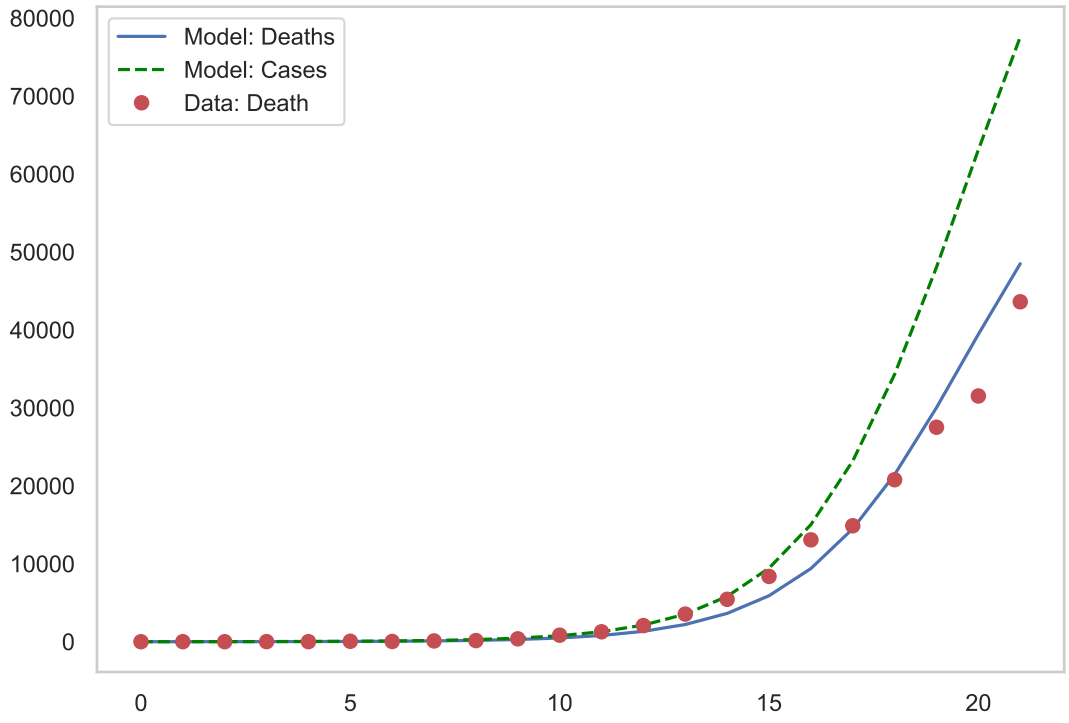
```
## <matplotlib.colorbar.Colorbar object at 0x7fc2a6c98438>
```

```
plt.show()
```



There isn't, so we'll stick with the parameters recovered here. Modelling it, we see:

```
###PARAMS###
N = 130000
k = 2.5
gamma = bestGamma
beta = bestBeta
R0 = beta/gamma
y0 = [N,1/(0.4*beta),0]
dt = 1


###Test###
lag = 0
t = np.arange(0,cutoff  + dt, dt )
sol = odeint(SIRode, y0, t, args = (beta,gamma))
q = (2-phi)*sol[:,1]
plt.plot(t,sol[:,1], label = "Model: Deaths")
plt.plot(t,(2-phi)*sol[:,1],'--', label = "Model: Cases", color = "green")

plt.plot(plague['plague_deaths'][:cutoff+1], 'ro',label = "Data: Death")
plt.legend(loc='best')
plt.show()



#Predict
```




```
print(bestBeta)
```

```
## 0.9175972425439678
```

```
print(bestGamma)
```

```
## 0.4
```

Model with final parameters:

```
###PARAMS###
N = 130000
k = 2.5
y0 = [N,1/(0.4*beta),0]
dt = 1


###Test###
lag = 0
t = np.arange(0,cutoff  + dt, dt )
sol = odeint(SIRode, y0, t, args = (beta,gamma))
#plt.plot(t,sol[:,1], label = "Model: Deaths")
plt.plot(t,sol[:,1],'--', label = "Model: Deaths", color = "Blue")
plt.plot(t,(2-phi)*sol[:,1],'--', label = "Model: Cases", color = "green")


plt.plot(plague['plague_deaths'][:cutoff+1], 'ro',label = "Data: Death")
plt.legend(loc='best')
```

```
#Parameters
print("Gamma:", gamma)
```

```
## Gamma: 0.4
```

```
print("Beta:", beta )
```

```
## Beta: 0.9175972425439678
```

```
print("R0:", R0)
```

```
## R0: 2.2939931063599195
```

```
print("r:", r)
```

```
## r: 0.5175972425439678
```

```
plt.show()
```