Lab: Parameter Estimation and Likehihood

Jessica Conrad, MSPH

MSRI Summer School on Algebraic Geometry July 2022 (Part 1)

Based off of the Parameter Estimation Lab by Dr. Marisa Eisenberg found here

Part 1: Deterministic SIR Model Review

The SIR model as we know it today was first formalized by Kermack and McKendrick in 1927. This mass-action compartmental model aims to predict the transmission of disease through a susceptible population over time. While the original model was a partial differential equaiton system tracking age-of-infection, we will explore here the simplified model that is only a function of time.

The basic model is as follows:

$$\frac{dS}{dt} = -\beta SI \quad (1)$$
$$\frac{dI}{dt} = \beta SI - \gamma I \quad (2)$$
$$\frac{dR}{dt} = \gamma I, \quad (3)$$

where *t* is time, *S(t)* is susceptible people, *I(t)* is infected people, and *R(t)* is removed (aka recovered and/or dead) people. β is the infection rate, and γ is the recovery rate.

Note: Usually, S + I + R = N implies a constant population. Our model assumes no birth or death explicitly.

To summarize, our important variables and definitions are:

- **S(t)** : susceptible population at time *t*
- **I(t)** : infected population at time *t*
- **R(t)** : removed population at time *t*
- β : infection rate; expected number of secondary infections per time *t*
- γ : recovery rate; $\frac{1}{\gamma}$ is average time a person is infectious/infected

Optional Resources

If you have never coded in Python before, the following short tutorials are recommended:

- 10 minute Python Tutorial Short tutorial with the basics recommended if you have done coding before in Python, but maybe not in a while
- Official Python Tutorial Official Python tutorial with details and references. Good if you have never coded in Python before.
- Think Python In depth introduction to Python coding

Coding the deterministic SIR Model

Now let's review how to code the deterministic SIR Model in Python. We will write this code together, step by step, then put it together! After we have learned how to write the model in Python, we will then build our likelihood function code.

First, we will import the libraries we will need for this code.

Python Hint: What are libraries and how do we use them? Libraries contain bundles of code created by other users, including useful functions, object classes, and data presets. Using the *import* command, we can access libraries instead of reinventing the wheel. Using the *from* command, we can access specific functions from a library. Some libraries are commonly referred to by a name other than their official name. For example, *scipy.stats* is often just called *stats*. Below you will see this name reference specified with the **as** command.

Python Hint: When using a Jupyter notebook, you need to run coding blocks in the order they are presented. If you run the blocks out of order, often errors will be generated. Click on the block below such that it is highlighted by a green outline. Then in the upper left corner of this notebook, click "Run."

In [8]:

```
from math import *                    # useful math functions
import numpy as np                    # useful array objects
                                      # (also a core scientific computing library)

import scipy.stats as stats           #useful statistics functions
import copy                           #allows for deep copies of objects

print("The libraries have been downloaded.")
```

```
The libraries have been downloaded.
```

Python Hint: What is a comment? You might have noticed some lines in the code above that are all in green. When writing in Python, you can comment code by putting a "#" before the character string. This tells the compiler to ignore the characters that follow *until you reach the next line of code*. Please comment your code always with notes for what a block of code is supposed to do! This is excellent coding practice called "Annotating".

Now let's try defining functions for movement between compartments! The functions have been written for you.

Python Hint: What is a function? Generally speaking a function is a block of code that only runs when called. In Python, we indicate that a function is about to be defined using the *def* command. The general form for defining a function in Python is: **def** myfuncname(inputs): ... Once a command is defined, you can call it using something like: myfuncname(inputs). We will show how to use this soon!

Fill in the state equations below:

In [9]:

```
# model equations for the scaled SIR model for python 2.7
# Original authors of this code, before edits by J. Conrad:
# Marisa Eisenberg (marisae@umich.edu)
# Yu-Han Kao (kaoyh@umich.edu) -7-9-17

def model(ini, time_step, params):
    #Define our ODE model

    ###########################
    #Input:
    #   ini           initial inputs for the state variables S, I, R
    #   time_step     time step definition; used by the ODE wrapper
    #   params        parameters for this model, in this case beta and gamma
    #
    #Output:
    #   Y             vector of state variable equations for S, I, and R
    ###########################
        Y = np.zeros(3)       #column vector for the state variables S, I, and R
        X = ini               #our initial conditions as defined from the input, also a column vector of our state vars
        beta = params[0]
        gamma = params[1]


### FILL IN THIS SECTION with your equations as defined above! Note that in python, S = X[0], I = X[1], and R = X[2]
        Y[0] = beta*X[0]*X[1]                #S (****remove for sandbox version of code****)
        Y[1] = beta*X[0]*X[1] - gamma*X[1] #I (****remove for sandbox version of code****)
        Y[2] = gamma*X[1]                    #R (****remove for sandbox version of code****)

        return Y
```

Part 2: Some Real Life Drama

Our basic model might be a little too basic. The researchers at the State Health Department have requested that you include birth and death of the population in your model. How do we reframe our original model to include birth and death, while maintaining the rule that S + I + R = N?

Write your model in the markdown section that follows here:

The basic model was:

$$\frac{dS}{dt} = -\beta SI \quad (4)$$
$$\frac{dI}{dt} = \beta SI - \gamma I \quad (5)$$
$$\frac{dR}{dt} = \gamma I, \quad (6)$$

My new model is:

$$\frac{dS}{dt} = \mu N - \beta SI - \mu S \quad (7)$$
$$\frac{dI}{dt} = \beta SI - \gamma I - \mu I \quad (8)$$
$$\frac{dR}{dt} = \gamma I - \mu R, \quad (9)$$

Oh no. The official from the department has notified you that even though they have been doing their best to identify all infectious cases, reporting isn't perfect. The data they recieve from hospitals and clinics represents only a fraction of all true cases.

As such, we will define now an output variable $y(t)$ which represents the number of reported infectious cases:

$$y = kI$$

where k is the reporting rate. This is also known as our measurement equation.

Oh wow! We have some new parameters, and now a defined output. Can you solve for the structural identifiability by hand?

Write the input-output equation for your new model below and discuss the structural identifiability of the new model.

What happens if we rescale the model in terms of "fraction of the population"?

How does this change the identifiability of the model? Make sure to include in the markdown section below the definition of all your new equations, including how to redefine the output equation. For consistency, let $\kappa = k * N$.

Ok, so how can we redefine the model function to include our new parameter(s)?

Redefine *model*() by adding your new terms

In [51]:

```
def model(ini, time_step, params):
        Y = np.zeros(3) #column vector for the state variables
        X = ini
        mu = 0
        beta = params[0]
        gamma = params[1]

        Y[0] = mu - beta*X[0]*X[1] - mu*X[0]          #S (****remove for sandbox version of code****)
        Y[1] = beta*X[0]*X[1] - gamma*X[1] - mu*X[1] #I (****remove for sandbox version of code****)
        Y[2] = gamma*X[1] - mu*X[2]                   #R (****remove for sandbox version of code****)

        return Y
```

The health department doesn't want to give us their HIPPA protected patient data right away. We need to prove to them that our model will work as expected. So first things first: if we had data with errors, would it return back to us a known parameter set that we decided on? How do we test this?

Part 3: Data Simulation

We are at a cross roads, where now we need to generate some fake data. For this section, we will begin by assuming the birth and death rates are slow enough to assume $\mu = 0$. This assumption is only valid for fast acting diseases. As such, let's define our "true" parameter set such that $\beta = 0.4$, $\gamma = 0.25$, and $\kappa = 80000$.

Note: Why is the assumption $\mu = 0$ valid here? Consider the timescale of our flowrates. If γ is the recovery rate, then the average time spent infected is $1/\gamma \approx 4$ days. Generally, we consider birth and death rates when the scale of the epidemic extends across many months or years. For the purpose of this simulation and simplicity, we do not need to consider birth/death at this time.

Note: I thought κ was the reporting rate! Why is it greater than 1? Recall that due to our rescaling, we defined $\kappa = k * N$ where k is the reporting rate given total population N.

Next, I'm going to give us a break and define some pre-generated test data with noise. Run the following block of code to generate our "true" data and new parameter definition!

In [52]:

```
#### Load Data ####
times = [0, 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]
data = [97, 271, 860, 1995, 4419, 6549, 6321, 4763, 2571, 1385, 615, 302, 159, 72, 34]

#DEFINE THE PARAMETER SET
#(****remove for sandbox version of code****)
params = [0.4, 0.25, 80000.0]        #make sure all the params and inition states are float
paramnames = ['beta', 'gamma', 'k']
```

For the initial conditions of our ODE, we are going to use:

$$S(0) = 1 - I(0) \quad (10)$$
$$I(0) = data(0)/\kappa \quad (11)$$
$$R(0) = 0, \quad (12)$$

where $data(0)$ is the first data value recorded. Note that the initial conditions depend on both the data and our parameter values! I often like to make the initial conditions a function, so that if we ever want to change the initial condition set up, we can just change the function definition and everything else will update automatically. As such, we define the following:

In [53]:

```
def x0fcn(params, data):
    #Set initial conditions

    ###########################
    #Input:
    #   data          true data to be fit
    #   params        parameters for this mode, in this case beta, gamma, and kappainv
    #
    #Output:
    #   X0            initial conditions for the SIR ODE
    ###########################
        S0 = 1.0 - (data[0]/params[2]) #(****remove for sandbox version of code****)
        I0 = data[0]/params[2]         #(****remove for sandbox version of code****)
        R0 = 0.0                       #(****remove for sandbox version of code****)
        X0 = [S0, I0, R0]

        return X0
```

Ok let's try using this function now!

Write the Python code below that will generate and print a new object called "ini", our initial conditions.

In [54]:

```
ini = x0fcn(params,data)   #(****remove for sandbox version of code****)
print(ini)                 #(****remove for sandbox version of code****)
```

```
[0.9987875, 0.0012125, 0.0]
```

We have the model equations in our function above, but the last thing we need to simulate the model is the measurement equation -- this defines what variables of the model we are observing. We'll make this a function too, so that way it's easy to update the code if we decide to measure something else. We'll make our measurement equation yfun take two inputs: the model simulation results (call this *res*) and the parameters (we'll call this *params*). Recall that we defined our measurement (or output) equation as $y = \kappa I$.

Python Hint: We know that our output data from the ODE should be returned as an object called "res" with 3 columns. In Python, when referencing a row or column, the indexing starts at 0. So if we want to recall the second column of the "res" object, we need to write res[:,1].

In [55]:

```
def yfcn(res, params):
    #Define measurement equation

    ###########################
    #Input:
    #   res           simulated data results
    #   params        parameters for this mode, in this case beta, gamma, and kappainv
    #
    #Output:
    #   simulated reported data
    ###########################
        return res[:,1]*params[2]
```

Now it's time to put all the pieces together and plot simulated data from our model against the true data I gave you. We need an ODE solver to take our model definition and parameters and generate "res". Instead of writing our own ODE solver, we can call one from a library! See how this is done below using the `from` command.

Python Hint: How did she know what inputs to give the ODE command? Either it's witchcraft...or it's Google! Try googling "scipy library odeint function". You should see a link for "SciPy v1.8.1 Manual" pop up! If not, see the documentation here . The documentation in this link will tell you in what order and what type of input objects can be used. The type of object you input into a function is *very* important. Often, when a function throws an error, the user input the wrong object type. For documentation inline while coding, you can alternatively run the line **help(ode)** instead of Googling.

In [56]:

```
from scipy.integrate import odeint as ode  # ode solver

#### Simulate the model ####
res = ode(model, ini, times, args=(params,))
print(res)
```

```
[[9.98787500e-01 1.21250000e-03 0.00000000e+00]
 [9.92834121e-01 3.42935111e-03 3.73652746e-03]
 [9.76425512e-01 9.42223912e-03 1.41522490e-02]
 [9.34348113e-01 2.39687527e-02 4.16831342e-02]
 [8.43538255e-01 5.08762600e-02 1.05585485e-01]
 [7.02321239e-01 7.75843282e-02 2.20094432e-01]
 [5.60356082e-01 7.84128832e-02 3.61231035e-01]
 [4.62754877e-01 5.64047466e-02 4.80840377e-01]
 [4.08915764e-01 3.29386418e-02 5.58145594e-01]
 [3.81909932e-01 1.72417257e-02 6.00848342e-01]
 [3.68845789e-01 8.55202325e-03 6.22602187e-01]
 [3.62613413e-01 4.13356186e-03 6.33253025e-01]
 [3.59657103e-01 1.97349811e-03 6.38369398e-01]
 [3.58258259e-01 9.36738200e-04 6.40805003e-01]
 [3.57597107e-01 4.43409290e-04 6.41959484e-01]]
```

So we have our model output data, and the true data! Except our true data was subject to a reporting bias :(

...good thing we made that measurement functin yfcn() to correct for this!

Define a new object called "sim_measure", our simulation predicted output data.

In [57]:

```
sim_measure = yfcn(res, params)   #(****remove for sandbox version of code****)
print(sim_measure)                #(****remove for sandbox version of code****)
```

```
[  97.          274.34808883  753.77912973 1917.50021899 4070.10080347
 6206.74625843 6273.03065253 4512.37972884 2635.09134782 1379.33805897
  684.1618596   330.68494879  157.87984873   74.93905598   35.47274323]
```

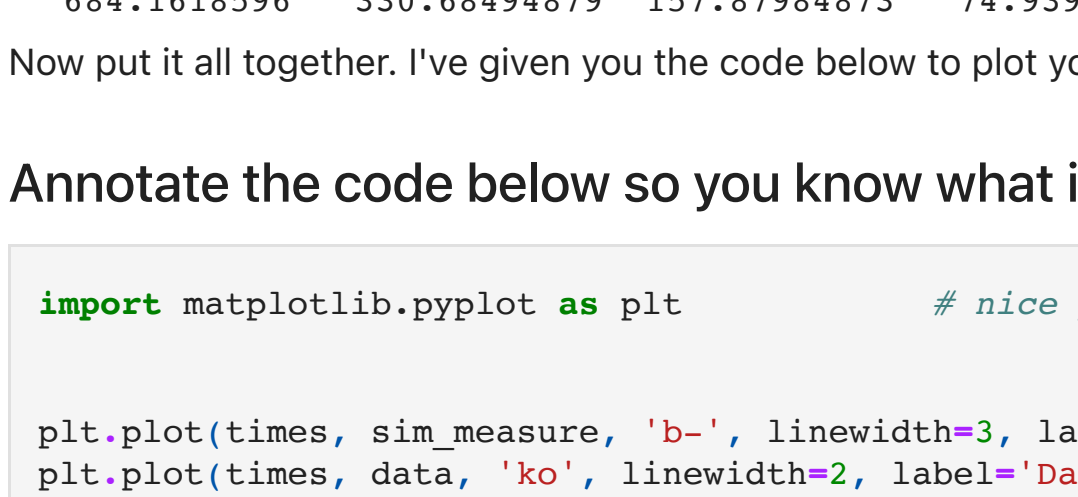
Now put it all together. I've given you the code below to plot your simulation and true data together. Just click run and see what you get!

Annotate the code below so you know what it does!

In [59]:

```
import matplotlib.pyplot as plt          # nice plotting commands, very similar to Matlab commands


plt.plot(times, sim_measure, 'b-', linewidth=3, label='Model simulation')
plt.plot(times, data, 'ko', linewidth=2, label='Data')
plt.xlabel('Time')
plt.ylabel('Individuals')
plt.legend()
plt.show()
```



Part 4: Numerical Identifiability Checks using Profile Likelihood

Wow we made it pretty far into our analysis. But now we want to know if the proposed parameter set is unique, or rather, identifiable from the data. We have already investigated the structural identifiability of this model, and now we want to explore the practical identifiability. That is, **can we recover the original parameter set when given ouput data with noise?**

To do this, we are going to use the Poisson maximum likelihood function that we derived together earlier! Remember that we re-framed the MLE as instead minimizing the negative log likelihood to simplify our algorithm.

Next, we will write code to estimate β, γ, and κ from the given data using Poisson maximum likelihood. Use the parameter values given above as starting parameter values, and you can use the initial conditions from above as well (note though that they depend on κ, which is a fitted parameter—so while we aren't fitting the initial conditions, they will need to change/update as we fit the parameters!). This means you will need to update your initial conditions inside the cost function, so Python uses the updated initial conditions when it tries new parameter values.

For note taking purposes, in the box below use the `sum` function to fill in the missing line.

Record here the Poisson likelihood function in Markdown, then fill in the Python block.

The Poisson maximum likelihood function, reframed for minimizing the negative log likelihood is:

$$min_p(-LL) = min_p\left(-\sum_{i=1}^{n} z_i ln(y(t_i)) + \sum_{i=1}^{n} y(t_i)\right)$$

In [61]:

```
from scipy.stats import poisson
from scipy.stats import norm

def NLL(params, data, times):
    #Define the negative log likelihood

    ###########################
    #Input:
    #   params        parameters for this mode, in this case beta, gamma, and kappainv
    #   data          true data to be fit
    #   times         time points when the true data is recorded
    #
    #Output:
    #   nll           negative log likelihood estimate
    ###########################
        params = np.abs(params)
        data = np.array(data)

    #Simulate the model with current parameters
        res = ode(model, x0fcn(params,data), times, args=(params,))

    #Apply the measurement equation
        y = yfcn(res, params)

    #Calculate the NLL for Poisson distribution
        nll = sum(y) - sum(data*np.log(y))   #(****remove for sandbox version of code****)

        # note this is a slightly shortened version--there's an additive constant term missing but it
        # makes calculation faster and won't alter the threshold. Alternatively, can do:
        # nll = -sum(np.log(poisson.pmf(np.round(data),np.round(y)))) # the round is b/c Poisson is for (integer) count
        # data this can also barf if data and y are too far apart because the dpois will be ~0, which makes the log
    # angry

        # ML using normally distributed measurement error (least squares)
        # nll = -sum(np.log(norm.pdf(data,y,0.1*np.mean(data)))) # example WLS assuming sigma = 0.1*mean(data)
        # nll = sum((y - data)**2)  # alternatively can do OLS but note this will mess with the thresholds
        #                             for the profile! This version of OLS is off by a scaling factor from
        #                             actual LL units.
        return nll
```

Estimate the parameters using `optimize`.

Armed with our likelihood function, we can now estimate the model parameters from the data. We will define a new object called "paramests" to save our parameter estimates in.

In [62]:

```
import scipy.optimize as optimize          #optimizer function package

optimizer = optimize.minimize(NLL, params, args=(data, times), method='Nelder-Mead')
paramests = np.abs(optimizer.x)
```

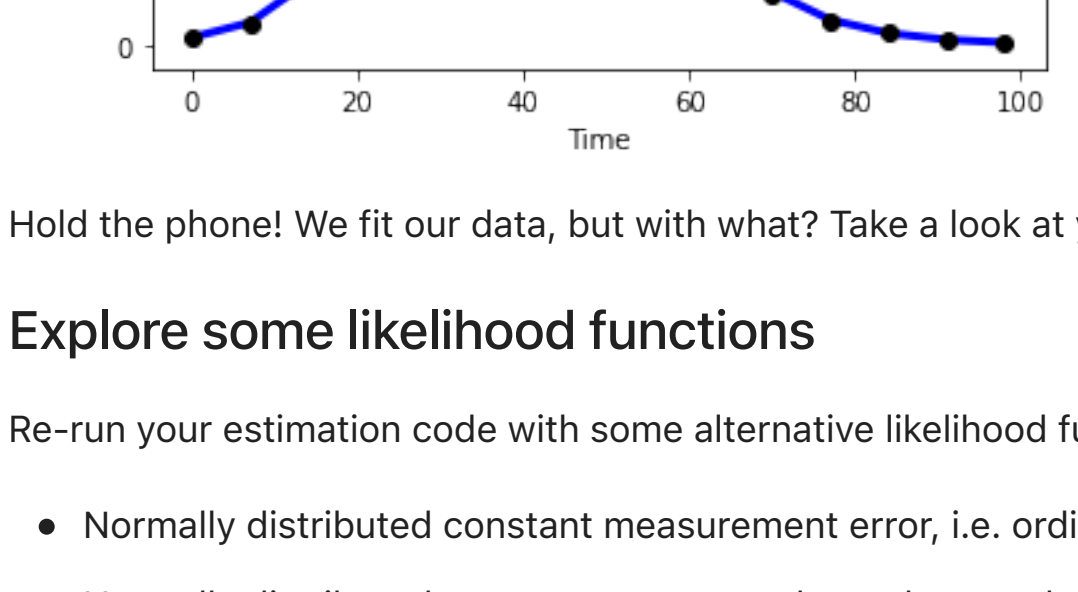
Finally, let's put the data together with our model using the parameter estimates we found. First we have to re-simulate the model using the parameter estimate values, otherwise we just have parameter estimates but no way to see what the fit looks like!

In [64]:

```
#### Re-generate initial case data based on new parameter estimates
iniests = x0fcn(paramests, data)      #(****remove for sandbox version of code****)

#### Re-simulate and plot the model with the final parameter estimates ####
#Simulate data
xest = ode(model, iniests, times, args=(paramests,))    #(****remove for sandbox version of code****)
#Apply the measurement equation
est_measure = yfcn(xest, paramests)                     #(****remove for sandbox version of code****)

#Plot
plt.plot(times, est_measure, 'b-', linewidth=3, label='Model simulation')
plt.plot(times, data, 'ko', linewidth=2, label='Data')
plt.xlabel('Time')
plt.ylabel('Individuals')
plt.legend()
plt.show()
```



Hold the phone! We fit our data, but with what? Take a look at your parameter estimates -- do they seem reasonable? How close to the initial values are they?

Explore some likelihood functions

Re-run your estimation code with some alternative likelihood functions, such as:

- Normally distributed constant measurement error, i.e. ordinary least squares, $Cost = \sum_i (data_i - y_i)^2$
- Normally distributed measurement error dependent on the data, weighted least squares, e.g. using Poisson-style variance, $Cost = \sum_i \frac{(data_i - y_i)^2}{data_i}$. This assumes the variance at any given data point is equal to the data ($\sigma^2 = data$), but you can also try other weightings! (Such as letting $\sigma^2 = data^2$.)
- Extended/weighted least squares, e.g. also using Poisson-style variance, $Cost = \sum_i \frac{(data_i - y_i)^2}{y_i}$. This assumes the variance at the i^{th} data point is equal to y_i, the model prediction at that time.
- Maximum likelihood assuming other distributions for the observation error, e.g. negative binomial

How do the parameter estimates and/or uncertainty differ from the estimates you got earlier? What are the underlying assumptions for on the model/measurement equation/likelihood that generate the different least squares cost functions given above?

In [ ]: