

# Lab: Fisher Information Matrix and Profile Likelihood

Jessica Conrad, MSPH

MSRI Summer School on Algebraic Geometry July 2022 (Part 2)

Based off of the Parameter Estimation Lab by Dr. Marisa Eisenberg found [here](#) and [here](#)

## Part 1: Recall our model

Here we resume our exploration of the following model:

$$\frac{dS}{dt} = \mu - \beta SI - \mu S \tag{1}$$

$$\frac{dI}{dt} = \beta SI - \gamma I - \mu I \tag{2}$$

$$\frac{dR}{dt} = \gamma I - \mu R \tag{3}$$

with measurement equation  $y = kI$ . In this case we are using the reparameterized model where we use the following definition:

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

We assume the births 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  $k = 80000$ .

Ok then we will provide for you the code we generated last session:

```
In [15]: ##### Import relevant libraries #####
from math import exp # useful math functions
import numpy as np # useful array objects
import scipy.integrate import odeint as ode # ode solver
import matplotlib.pyplot as plt # nice plotting commands, very similar to Matlab commands
from scipy.stats import norm # also a core scientific computing library
import scipy.optimize as optimize # Optimizer function package

#### Redefine the functions we wrote ####
def model(tini, time_step, params):
    """Define our ODE model"""
    #####
    #Input:
    # tini: 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
    X = tini
    beta = params[0]
    gamma = params[1]
    Y[0] = mu - beta*X[0]*X[1] - mu*X[0]
    Y[1] = beta*X[0]*X[1] - gamma*X[1] - mu*X[1]
    Y[2] = gamma*X[1] - mu*X[2]
    return Y

def xOfcn(params, data):
    """Define measurement equation"""
    #Input:
    # data: true data to be fit
    # params: parameters for this model, in this case beta, gamma, and kappaiv
    #Output:
    # x0: initial conditions for the SIR ODE
    #####
    beta = 1.0 # beta = data[0]/params[2]
    gamma = data[1]/params[2]
    kappa = data[2]/params[2]
    x0 = [1.0, 0.0, 0.0]
    return x0

def yfcn(res, params):
    """Define measurement equation"""
    #Input:
    # res: simulated data results
    # params: parameters for this model, in this case beta, gamma, and kappaiv
    #Output:
    # simulated reported data
    #####
    return res[:,1]/params[2]

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

    #Simulate the model with current parameters
    res = ode(model, xOfcn(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

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

    return nll
```

```
In [16]: ##### 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, 8321, 4763, 2571, 1385, 615, 302, 159, 72, 34]

##### Define our parameter set #####
params = [0.4, 0.25, 80000.0] # make sure all the params and initn states are float

##### Simulate the model #####
ini_xOfcn(params, data)
res = ode(model, ini_xOfcn(params, data), times, args=(params,))
sim_measure = yfcn(res, params)

##### Plot the data and simulation #####
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()
```

Python Hint: Why do I separate the coding blocks that contain libraries and function definitions versus parameter settings and function calls? Generally it is good coding practice to separate your functions in a file separate from variables and parameters that you the user will change. This will help prevent user errors. We first call our libraries and functions, and only then do we run our code experiments.

Let's also take a moment to remember how the likelihood function  $NLL$  could be used to estimate our parameters by wrapping it in an optimizer!

```
In [17]: ##### Choose an optimizer and estimate parameter values #####
optimizer = optimize.minimize(NLL, params, args=(data, times), method='Nelder-Mead')
parameters = np.abc(optimizer.x)

##### Re-generate initial case data based on new parameter estimates
inits = xOfcn(parameters, data)

##### Re-simulate and plot the model with the final parameter estimates #####
#Simulate data
x0 = ode(model, inits, times, args=(parameters,))
#Apply the measurement equation
est_measure = yfcn(x0, parameters)

#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()

print("The parameter estimates are beta = {params[0]:.4}, gamma = {params[1]:.4}, and kappa = {params[2]:.4}.format(parameters))
```

The parameter estimates are beta = 0.3999, gamma = 0.2468, and kappa = 80256.57055016875

## Part 2: Fisher Information Matrix

Ok we are all caught up on where we left off. Let's set up a new function to calculate the Fisher Information Matrix. Recall from the lecture, the exact definition of the Fisher Information Matrix is:

$$I(p)_{ij} = \mathbb{E} \left[ \left( \frac{\partial}{\partial p_i} \log \mathcal{L}(z, p) \right) \left( \frac{\partial}{\partial p_j} \log \mathcal{L}(z, p) \right) \right]$$

How do we translate this into code? The expectation function is going to be dependent on our model and likelihood function  $\mathcal{L}(z, p)$ , so we need to write code specific to the SIR model. We are not going to walk through this particular derivation and instead we will spend time learning how to make sense of the results.

Note: The FIM is a statistical method dependent on the quality of your data  $z$ , where  $z$  is defined as  $z_i = y_i + \epsilon_i$ , where  $y_i$  is the measurement equation and  $\epsilon_i$  is the error in each measurement of that output. If you are unable to characterize or limit the noise in your output data  $z$ , then all you will get is nonsense. As the age old saying goes: Garbage in, garbage out.

Here is the FIM function for the SIR model:

```
In [19]: # Simplified FIM (Fisher Information Matrix) function for the SIR model
# Marisa Eisenberg (marisa@umich.edu)
# Yu-Han Kao (kao@umich.edu) -7-9-17

def minifisher(times, params, data, delta = 0.001):
    """Calculate the FIM for the SIR model."""
    #####
    #Input:
    # times: time points when the true data is collected
    # params: parameters for this model, in this case beta, gamma, and kappaiv
    # data: true data to be fit
    # delta: fit parameter for FIM; preset to 0.001, but can be set by user
    #Output:
    # simulated reported data
    #####
    params = np.array(params)
    listX = []
    params_1 = np.array(params)
    params_2 = np.array(params)
    for i in range(len(params)):
        params_1[i] = params[i] * (1+delta)
        params_2[i] = params[i] * (1-delta)
        res_1 = ode(model, xOfcn(params_1, data), times, args=(params_1,))
        res_2 = ode(model, xOfcn(params_2, data), times, args=(params_2,))
        subX = (yfcn(res_1, params_1) - yfcn(res_2, params_2)) / (2 * delta * params[i])
        listX.append(subX.tolist())
    X = np.matrix(listX)
    FIM = np.dot(X, X.transpose())
    return FIM
```

Note: This FIM code is a numerical approximation method. Specifically,  $\text{minifisher}$  is using a numerical approximation for the derivative. Ariel Citrón Arias's slides found here give a nice introduction to estimation and sensitivity equations using the forward sensitivity equations instead.

Ok so now that we have our function, model, and data, let's try running this example and see what the FIM looks like and explore some of its features!

Write a line of code to generate the FIM for our model by looking at how the function call was defined above.

```
In [13]: ##### Calculate the simplified Fisher Information Matrix (FIM) #####
FIM = minifisher(times, params, data, delta = 0.001) #****remove for sandbox version of code****
print(FIM) #****remove for sandbox version of code****

[[[1.9806229e+10 1.10016690e+10 3.7053476e+04]
 [1.10016690e+10 1.10016690e+10 2.3922680e+04]
 [3.7053476e+04 2.3922680e+04 7.6847320e-02]]]
```

Python Hint: Are you wondering how to see what the objects you created actually look like? Try the `print()` function. For example, after you generate the object called "FIM", in the next line write `print(FIM)`. Click "Run" in the upper left corner and see what happens!

Using `np.linalg.matrix_rank()`, calculate the rank of your new matrix.

```
In [14]: ##### Calculate rank of FIM #####
print(np.linalg.matrix_rank(FIM)) #****remove for sandbox version of code****

3
```

Let's Ponder: What do we expect the structure of the FIM to look like? What does this tell us about the identifiability of the model? Try it out with the other un-scaled SIR model given in Lab 19 (or you can use the un-scaled SIR from the parameter estimation lab) -- how does that change things?

As a note to yourself, write down some of your findings and thoughts in the Markdown box below OR add new coding boxes to explore the suggested problem!

## Part 3: Generating Profile Likelihoods

The recovery rate  $\gamma$  is often approximately known, so let's fix the value of  $\gamma = 0.25$ . Now we have only two unknown parameters,  $\beta$  and  $k$ . We want to plot the likelihood as a surface or heat map as a function of  $\beta$  and  $k$  (i.e. so that color is the likelihood value, and your  $x$  and  $y$  axes are the  $\beta$  and  $k$  values respectively).

Note: Recall that  $\gamma$  is the recovery rate, and  $\frac{1}{\gamma}$  is the time spent in state  $I$ , or "time spent infected/infectious". Generally, a person is considered "infected" in our model when they present symptoms of disease. Symptoms of disease are noticeable, so we can often accurately approximate the "time spent infected/infectious" -- the inverse value of  $\gamma$ . Therefore,  $\gamma$  is often known with some bound of uncertainty.

As an example, here's some code to plot the likelihood for the Poisson case we used earlier. First, choose your own  $\beta$  and  $\gamma$  range of value to explore and their interval.

Use `np.arange()` to define a sequence of values for  $\beta$  and  $\gamma$  that you want to iterate through. Use `np.zeros()` to generate a blank matrix called "likevals" to store the likelihood values in.

```
In [44]: # Define the ranges for each parameter, and make an empty matrix for the likelihood values
beta_range = np.arange(0.35, 0.45, 0.01) #****remove for sandbox version of code****
gamma_range = np.arange(0.2, 0.3, 0.01) #****remove for sandbox version of code****
likevals = np.zeros((len(beta_range), len(gamma_range))) #****remove for sandbox version of code****

print(beta_range)
print(gamma_range)

[0.35 0.36 0.37 0.38 0.39 0.4 0.41 0.42 0.43 0.44 0.45]
[0.2 0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29 0.3]
[0.35 0.36 0.37 0.38 0.39 0.4 0.41 0.42 0.43 0.44 0.45]
[0.2 0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29 0.3]
[0.35 0.36 0.37 0.38 0.39 0.4 0.41 0.42 0.43 0.44 0.45]
[0.2 0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29 0.3]
[0.35 0.36 0.37 0.38 0.39 0.4 0.41 0.42 0.43 0.44 0.45]
[0.2 0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29 0.3]
[0.35 0.36 0.37 0.38 0.39 0.4 0.41 0.42 0.43 0.44 0.45]
[0.2 0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29 0.3]
```

I have sketched out some "for" loops for you.

How do we populate each value of the "likevals" matrix using the  $NLL$  functions?

```
In [52]: # Go through each point on the contour plot and calculate the likelihood value at those coordinates
for i in range(len(beta_range)):
    for j in range(len(gamma_range)):
        likevals[i,j] = NLL([beta_range[i], 0.25, kappaiv_range[j]], data, times) #NLL(params, data, times)

print(np.min(likevals))
print(np.max(likevals))

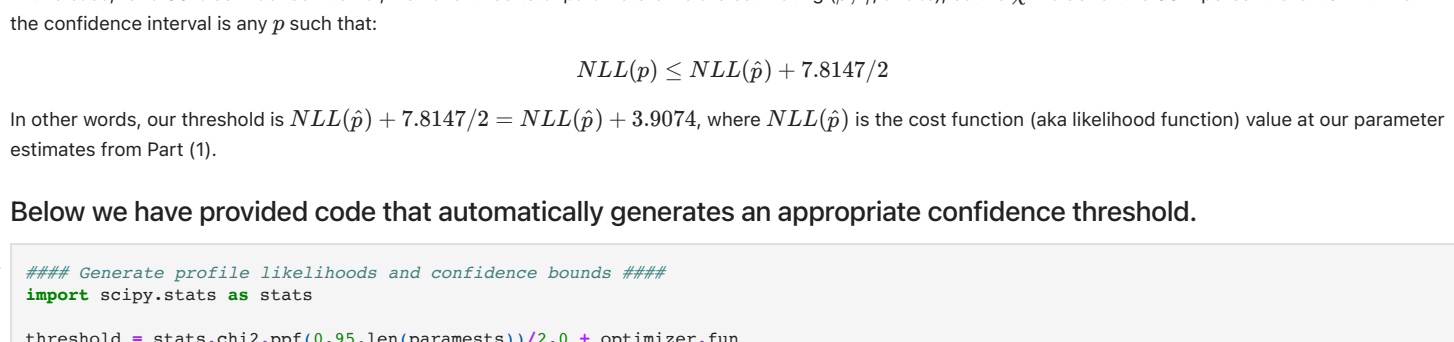
-219872.5894041744
-207845.1730756317
```

Ok now that we have the Likelihood values for different  $\beta$  and  $\gamma$  combinations, let's plot this matrix as a contour to visually assess what our results are. You can try different ranges for beta and kappa depending on how far out you want to look at the plot!

How does the shape of the profile likelihood change as you switch likelihood functions?

It may not change much, but you can often notice small differences between likelihood choices. What does the likelihood landscape tell us about the parameter identifiability of this model, assuming  $\gamma$  is known?

```
In [53]: ##### Make a contour plot: #####
plt.contour(beta_range, kappa_range, likevals)
plt.xlabel('Beta Range')
plt.ylabel('Kappa Range')
plt.colorbar()
```



Please write a discussion of your findings in the Markdown box that follows.

## Part 4: Profile Likelihood to Confidence Bounds on Parameters

Now we have looked at the likelihood but single parameter might look like we fix one of our parameters and explore the combinations of other parameters. But in the lecture, we discussed looking at the likelihood plots of space might look when letting all parameters vary in value.

In this section, we will explore **Profile Likelihood Plots** and **Likelihood-based Confidence Intervals**.

To make our lives a little easier, I am not going to make you write the profile likelihood generator code. Instead, let's walk through the profile likelihood generator code and make sure that we have an understanding of what it does, and more importantly does what it claims to do!

```
In [47]: # Profile Likelihood Generator
# Marisa Eisenberg (marisa@umich.edu)
# Yu-Han Kao (kao@umich.edu) -7-9-17

def profcofct (fit_params, profparam, profindex, data, times, cost_func):
    """Generate profile likelihoods and confidence bounds"""
    parameter = fit_params.tolist()
    parameter[profindex] = profparam
    return cost_func(parameter, data, times)

# Input definitions
# params = starting parameters (all, including the one to be profiled)
# profparam = index within params for the parameter to be profiled
# costfunc = cost function for the model -- should include params, times, and data as arguments.
# Note: costfunc doesn't need to be specially set up for fixing the profiled parameter
# it's just the regular function you would use to estimate all the parameters
# (it will get reworked to fix one of them inside profcofct)
# data, times = data set (times x values, or whatever makes sense)
# perange = the percent/fraction range to profile the parameter over (default is 0.5)
# numpoints = number of points to profile at in each direction (default is 10)

# Output
# A list with:
# - profparvals: the values of the profiled parameter that were used
# - nvals: the cost function value at each profiled parameter value
# - confvals: the confidence value at each profiled parameter value
# - paramvals: the estimates of the other parameters at each profiled parameter value

def proflike (params, profindex, cost_func, times, data, perange = 0.5, numpoints = 10):
    profrangedown = np.linspace(params[profindex], params[profindex] * (1 - perange), numpoints).tolist()
    profrangeup = np.linspace(params[profindex], params[profindex] * (1 + perange), numpoints).tolist()[1:] #skip the duplicated value
    curparams = []
    curflags = []

    fit_params = params.tolist() #make a copy of params so we won't change the original list
    fit_params.pop(profindex)
    print("Starting profile...")
    for i in range(len(profrangedown)):
        for j in range(len(profrangeup)):
            optimize.minimize(profcofct, fit_params, args=(j, profindex, data, times, cost_func), method='Nelder-Mead')
            fit_params = np.abc(optimizer.x).tolist() #save current fitted params as starting values for next round
            curparams.append(optimizer.fun)
            curflags.append(optimizer.success)
            curparams.append(np.abc(optimizer.x).tolist())

    #structure the return output
    profrangedown.reverse()
    out_profrange = profrangedown+profrangeup
    temp_ind = range(len(profrangedown))
    out_params = [curparams[i] for i in reversed(temp_ind)]+curparams[len(profrangedown):]
    out_vals = [curvals[i] for i in reversed(temp_ind)]+curvals[len(profrangedown):]
    out_flags = [curflags[i] for i in reversed(temp_ind)]+curflags[len(profrangedown):]
    output = (profrangeup, out_profrange, fit_params, out_params, out_vals, out_flags, 'covergence': out_flags)
    return output
```

Hopefully you wrote yourself notes in the code above. Now let's start by defining our confidence interval.

For the threshold to use in determining your confidence intervals, we note that  $2(NLL(\hat{p}) - NLL(\hat{p}))$  (where  $NLL(\hat{p})$  is the negative log likelihood) is approximately  $\chi^2$ -distributed with degrees of freedom equal to the number of parameters fitted (including the profiled parameter). Then an approximate 95% (for example) confidence interval for  $p$  can be found by taking all values of  $p$  that lie within the 95th percentile range of the  $\chi^2$ -distribution for the given degrees of freedom.

In this case, for a 95% confidence interval, we have three total parameters we are fitting ( $\beta$ ,  $\gamma$ , and  $k$ ), so the  $\chi^2$  value for the 95th percentile is 7.8147. Then the confidence interval is any  $p$  such that:

$$NLL(p) \leq NLL(\hat{p}) + 7.8147/2$$

In other words, our threshold is  $NLL(\hat{p}) + 7.8147/2 = NLL(\hat{p}) + 3.9074$ , where  $NLL(\hat{p})$  is the cost function (aka likelihood function) value at our parameter estimates from Part 1).

Below we have provided code that automatically generates an appropriate confidence threshold.

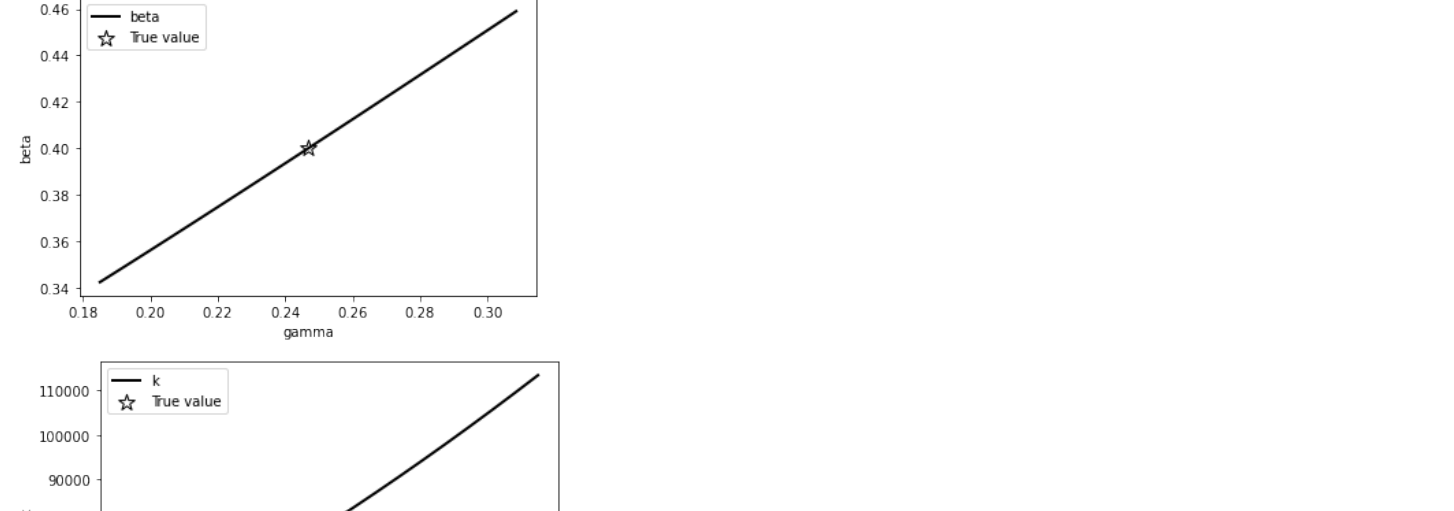
```
In [48]: ##### Generate profile likelihoods and confidence bounds #####
import stats.stats as stats

threshold = stats.ch2.ppf(0.95, len(parameters)) / 2.0 + optimizer.fun
perange = 0.25 #percent range for profile to run across
```

Now that we can determine if our parameter estimate is reasonable using our threshold, let's try profiling the parameters with a profile likelihood plot, as seen in our earlier lectures.

Are the parameters practically identifiable? Why or why not?

```
In [50]: ##### Plot the individual parameter profiles #####
profiles = {}
for i in range(len(parameters)):
    profiles[parameters[i]] = proflike(parameters, i, NLL, times, data, perange=perange)
    print("Profiles have been generated.")
    plt.figure()
    plt.scatter(parameters[i], optimizer.fun, marker='*', label='True value', color='k', s=150, facecolors='w', edgecolors='k')
    plt.plot(profiles[parameters[i]]['profparam'], profiles[parameters[i]]['confvals'], 'k-', linewidth=2, label='Profile Likelihood')
    plt.xlabel(parameters[i])
    plt.ylabel('Negative log likelihood')
    plt.legend(scatterpoints = 1)
    parameters_fit = [v for v in parameters if v not in [parameters[i]]]
    parameters_fit = [v for v in parameters if v not in [parameters[i]]]
    print(parameters_fit)
    print(parameters_fit)
```



What do we learn from the 2 parameter relationship analysis?

## Part 5: Bonus: Back to that real life drama

Lastly, let us consider the case where you are attempting to fit and forecast an ongoing epidemic (i.e. with incomplete data). Truncate your data to only include the first seven data points (i.e. just past the peak), then re-fit the model parameters and generate the profile likelihoods with the truncated data (you can also see if truncating the data affects the FIM rank).

- How do your parameter estimates change?
- Does the practical identifiability of the parameters change? How so?
- If any of the parameters were unidentifiable, examine the relationships between parameters that are generated in the profile likelihoods. Can you see any interesting relationships between parameters? What do you think might be going on--why has the identifiability changed?

Please add coding and markdown blocks below this section as you explore and answer the above questions! Remember to document and annotate your code so you can look back at this later.

```
In [ ]:
```