Lab: Fisher Information Matrix and Profile Likelihood

Part 1: Recall our model

```python
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])
        I0 = data[0]/params[2]
        R0 = 0.0
        X0 = [S0, I0, R0]

        return X0


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]


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
```

In [16]:

```python
#### 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 our parameter set ####
params = [0.4, 0.25, 80000.0]        #make sure all the params and inition states are float
paramnames = ['beta', 'gamma', 'k']

#### Simulate the model ####
ini = x0fcn(params,data)
res = ode(model, ini, 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 did I separate the coding blocks that contain *libraries and function definitions* versus *parameter settings and function calls*? Generally it is good coding practice to write 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 [67]:

```python
#### Choose an optimizer and estimate parameter values ####
optimizer = optimize.minimize(NLL, params, args=(data, times), method='Nelder-Mead')
paramests = np.abs(optimizer.x)

#### Re-generate initial case data based on new parameter estimates
iniests = x0fcn(paramests, data)

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

#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]}'.format(params=paramests))
```

```
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:

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

How do we translate this into code? The expectation function is going to be dependent on your 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 $z_i = y_i + e_i$, where y is the measurement equation and e is the error in each measurement of that output. If you are unable to characterize or limit the noise in your ouput 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 [9]:

```python
# Simplified FIM (Fisher information matirx) function for the SIR model
# Marisa Eisenberg (marisae@umich.edu)
# Yu-Han Kao (kaoyh@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 mode, in this case beta, gamma, and kappainv
    #   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, x0fcn(params_1,data), times, args=(params_1,))
                res_2 = ode(model, x0fcn(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, $minifisher$ 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]:

```python
#### 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****)
```

```
[[1.98066299e+10 1.10016690e+10 3.77053476e+04]
 [1.10016690e+10 1.10680375e+10 2.59296580e+04]
 [3.77053476e+04 2.59296580e+04 7.68473201e-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]:

```python
#### 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 (1) (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 problems!