Technology Enhanced Problem Based Learning with Applications to Real-World Problems

Padmanabhan Seshaiyer, Byong Kwon and Thomas Stephens
pseshaiy@gmu.edu, bkwon1@masonlive.gmu.edu, tstephe3@gmu.edu
Department of Mathematical Sciences
George Mason University, Fairfax, VA 20151
USA

Abstract: In this paper we consider the application of two diverse software tools to simulate differential equation models developed for disease dynamic models. Specifically, we motivate the SIR disease epidemic model that consists of a system of differential equations that can be solved via numerical algorithms including Euler's method and Runge-Kutta method. We implement these using a MATLAB literate programming approach and show how one can obtain best fit parameters for given data from the real-world and also visualize it using a graphical user interface (GUI). We also present an alternative approach using the AMPL Optimization software to do the same problem. The purpose of introducing the latter software is its capability to handle real-world problems with big data. The results obtained from both methods are comparable yielding similar best-fit parameters for a given set of data and they suggest that the methods proposed herein are reliable and robust for solving real-world applications.

1. Introduction

Research in computational mathematics, which comprises of modeling, analysis, simulation and computing is quickly becoming the foundation for solving most multidisciplinary problems in science and engineering. These real-world problems often involve complex dynamic interactions of multiple physical processes which presents a significant challenge, both in representing the physics involved and in handling the resulting coupled behavior. If the desire to control and design the system is added to the picture, then the complexity increases even further. Hence, to capture the complete nature of the solution to the problem, a coupled multidisciplinary approach is essential. Therefore, performing research and teaching in computational mathematics needs an in-depth understanding of the underlying mathematics and the fundamental principles that govern a physical phenomenon as well as understanding the underlying technology that can be used efficiently to simulate the physical phenomenon. It is well known that many physical systems can be described by differential equations. Thus, understanding the behavior of the numerical solution to such equations is important for elucidating the actual physical problem. Through our research in this field, we have come to appreciate that analyzing a numerical technique requires a combined theoretical and computational approach. Theory is needed to guide the performance and interpretation of the numerical technique while computation is necessary to synthesize the results. Therefore, the solution methodology involves formulating a mathematical model from a physical system and then being able to solve this model

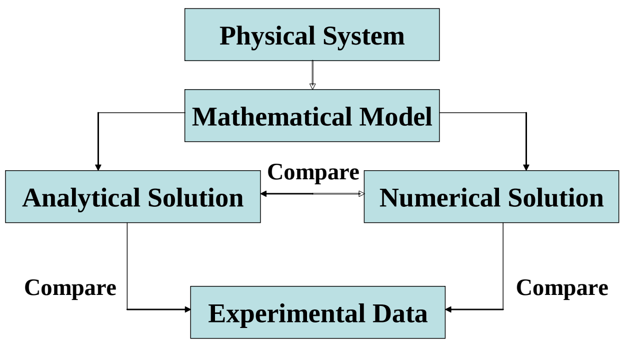


Figure 1: A solution methodology for research in Computational Mathematics

using an analytical (exact) or numerical (approximate) approaches. The mathematical models developed as a part of the research are then validated using real experimental data. This problem solving methodology is illustrated in Figure 1 and key ingredient that helps to perform the validations is technology [1].

Technology is important in both teaching and learning computational mathematics as it can not only influence the mathematics that is taught but also helps to enhance student learning. The word “enhances” is what characterizes technology as a tool with high leveraging power because technology can enhance a learning task. By tapping into these powerful technology tools in mathematics, we can take advantage of the dynamic capability to graph, model, compute, visualize, simulate, and manipulate, and *amplify* the mathematical properties and concepts. This instructional technology focus has important implications in preparing teachers and students to teach and learn with technology. Not only must one understand the importance of enhancing content using technology but also promote the awareness of pitfalls if technology is not used with relevant information. One such example is the Patriot Missile failure, in Dharan, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths and was ultimately attributable to poor handling of inexact computer round-off arithmetic. Another example is the explosion of the Ariane 5 rocket just 40 seconds after lift-off on its maiden voyage off French Guiana, on June 4, 1996. The destroyed rocket and its cargo were valued at $500 million [2].

The focus of this paper will be on understanding the need to use technology tools efficiently in solving complex real-world problems that can be modelled via differential equations. Specifically, we will consider a classic disease dynamics problem arising from a real-world application that are solved numerically and implemented via two different technology tools. The first one involves implementing the models developed in MATLAB and creating a front-end graphical user interface (GUI) tool for the user and the second one involves using an optimization software AMPL that naturally admits solving problems that can be posed in a least-squares formulation with differential equations as constraints and allows computations with big data quite easily. In section 2, we will develop the proposed disease dynamics model. In section 3, we will describe the MATLAB program to develop the GUI related to the application. Section 4 introduces readers to AMPL (a modelling language for mathematical programming) and discusses the implementation of the same problem that was introduced in the MATLAB GUI development. The advantages of using both softwares will be discussed in the final section.

2. Background, Models and Methods

Students are first exposed to differential equations in Calculus when they learn about the notion of a derivative followed by the anti-derivative and finally connecting these two big concepts. The traditional differential equations taught to students in Calculus are often separable where they learn to separate the respective variables and integrate to yield the solution. One of the standard applications students are exposed to in this process includes a function $y(t)$ whose rate of change $\frac{dy}{dt}$ is directly proportional to the function value $y(t)$. This leads to the underlying exponential growth and decay models $\frac{dy}{dt} = k\,y$ where k is a constant and the equation is a separable differential equation. With a prescribed initial condition $y(0) = y_0$ we obtain the familiar exponential solution $y(t) = y_0\, e^{k\,t}$. Variations of this equations arise in applications such as

Newton's Law of Cooling/Heating or mixing problems where the differential equations takes the form $\frac{dy}{dt} = k\,y - c$ where both *k* and *c* are constants. Note that one can rewrite the differential equation to $\frac{dy}{dt} = k\left(y - \frac{c}{k}\right)$ and with a simple transformation $Y(t) = y(t) - \frac{c}{k}$ we get $\frac{dY}{dt} = k\,Y$. The transformation also helps to modify the initial condition $Y(0) = y_0 - \frac{c}{k}$ which helps to compute an exponential solution.

While these separable differential equations admit an exact solution as indicated in Figure 1, as students transition to a traditional differential equation class they are exposed to more general linear differential equations $\frac{dy}{dt} + p(x)\,y = q(x)$. One can see the relation of the variable coefficients $p(x)$ and $q(x)$ which are continuous functions on some interval I, to be constants *k* and *c* respectively. Students are then introduced to the notion of an *integrating factor* that enables one to write the left side $\frac{dy}{dt} + p(x)\,y$ of the equation as a derivative of a product which helps to compute the exact solution once again. As problems become more complicated, it necessitates the introduction of multiple dependent variables and forming systems of differential equations. Next, let us consider the development of one such system for modeling disease dynamics.

2.1. Modeling Epidemics

Over the last few decades, the theory of epidemics introduced by Kermack and McKendrick [3] employed mathematical ideas to describe how to determine the threshold size for a susceptible population in comparison to those infected that will help predict the dynamics of the epidemic. In fact, the theoretical epidemic threshold introduced through these early models has been observed in practice and helps to measure the extent to which a real population is vulnerable to spread of an epidemic. Next, we present a simple coupled system that describes the disease dynamics when a fixed population is assumed to consist of three types of individuals defined by disease as: susceptibles (S), infectives (I) and removals (R). The model will assume that susceptibles can become infected upon effective contact with an infective, infectives will be assumed to have the disease and are capable of transmitting it, and removals are those who have recovered from being infected. Assuming the total population is fixed $S(t) + I(t) + R(t) = N$ and that the population is thoroughly mixed (so that each individual has equal probability to make effective contact with any other individual in the population), we can derive the following popular SIR model:

$$\frac{dS}{dt} = -\beta\, I\, S \qquad \frac{dI}{dt} = \beta\, I\, S - \gamma\, I \qquad \frac{dR}{dt} = \gamma\, I \tag{1}$$

where β is the *contact rate* of infectives (β > 0) and γ is the *recovery rate* (γ > 0). Note that this system is easy to understand. The last of the coupled equation system (1) refer to the rate at which infected are removed (die or recover) is proportional to the number of infectives $I(t)$. The second term in the second of the coupled system (1) relates to the same individuals that are removed from the infective class. The first term in that equation refers to the new number of infectives per unit time being proportional to both the number of potential susceptibles and to the number of existing infectives that are capable of infecting others. The first equation in system (1) can be similarly interpreted as the number of susceptibles becoming infected and leaving.

2.2. Numerical Solution

One way to help realize the solution to differential equations is to introduce the concept of slope fields that provides a graphical tool for visualization based on the principle of *local linearity*. The later promotes the necessity of thinking of the graphs of these differentiable functions to be locally "linear". A collection of such local line segments of various slopes obtained from the differential equation, forms the slope field which is a pictorial evolution of the family of solutions to the differential equation. While this notion of linearity can be easily verified by using a graphing calculator by zooming into a function graph, this local linearity concept has given rise to powerful computational algorithms such as the Euler's method for solving differential equations numerically by producing a numerical table of approximations to the associated initial value problem. For instance, the Euler's method for $\frac{dy}{dt} = f(t, y),\ y(0) = y_0$ where $f(t, y)$ is any general function, maybe implemented by a recursive sequence of approximate solutions for $i \geq 1$ as: $y_{i+1} = y_i + \Delta t\ f(t_i, y_i)$ where Δt is the stepsize in time. The form of the recursive equation indicates the *local linearity* as it resembles the slope intercept form of a line, namely, $y = m\,x + b$. Extending this to the system (4) we can obtain the following approximations for the susceptibles, infectives and removals for $i \geq 1$:

$$S_{i+1} = S_i - \Delta t\ \beta I_i S_i \qquad I_{i+1} = I_i + \Delta t\ \beta I_i S_i - \gamma I_i \qquad R_{i+1} = \Delta t\ \gamma I_i \qquad (2)$$

While the Euler's method is very popular for its ease of implementation, it does have limitations in terms of accuracy. In a senior level numerical analysis class, students are often exposed to such limitations which helps them to look for more advanced methods that are high-order accurate. One such method is the Runge-Kutta method. Next, we will explain how we can implement system (1) using this high order method in MATLAB using a Literate Programming style [4].

3. MATLAB in Literate Programming Style

We will examine how MATLAB was used to implement the above numerical approximations, directly within this paper to allow the reader to understand the implementation step-by-step and reproduce similar results. First, to execute the SIR system, a function script file called **sir_system.m** was created as follows:

```
function dydt = sir_system(t, y, r)
```

This function outputs the result of the SIR-system. The inputs to this function subroutine are the time span, previous solution and parameter values in the same order. As the inputs come in, the values are stored into the respective variables in system (1).

```
beta = r(1);
gamma = r(2);
S = y(1);
I =  y(2);
R = y(3);
```

Finally to efficiently enter the approximate differential equation system (1), we take advantage of the vector form $\frac{d\vec{y}}{dt} = \left[\frac{dS}{dt}; \frac{dI}{dt}; \frac{dR}{dt}\right]$ and its corresponding statement becomes:

```
dydt = [-beta*I*S; beta*I*S - gamma*I; gamma*I];
```

This function subroutine is then called from the following main program called **sir_main.m.** First we enter the true experimental data for infectives collected over a time span of 14 days.

```
data = [3 6 25 73 222 294 258 237 191 125 69 27 11 4];
time = 1:14;  tspan = time;
```

Next, we enter the initial conditions. In this case we will consider the initial susceptible population to be 760, infectives to be 3 and none for the removal. We also enter the parameters β and γ in the form of a vector and solve the system (1) using the built-in MATLAB differential equation solver ode45. Note that ode45 employs a high-order Runge-Kutta method which is much more accurate than the Euler's method.

```
y0 = [760 3 0];
r0 = [.01 .1];
[t, y] = ode45(@sir_system,tspan,y0,[ ],r0);
```

Finally, we plot the approximate solutions for the susceptibles, infectives and removals on the same graph along with the data collected. This is shown in Figure 2.

```
figure(1), clf; plot(t, y, 'linewidth', 2); hold on;
plot(time, data, 'k*', 'markersize', 10);
legend('susceptible','infected','recovered','data')
ylabel('Number of people')
xlabel('time')
```

As indicated in Figure 1, it is often required to numerical solve problems that do not admit analytical solutions. Figure 2 shows the approximate solutions produced by attempting to solve the system (4) numerically using a high-order accurate Runge-Kutta method. As figure 2 indicates, we are able to get a trend that we seem to expect such as the number of susceptibles going down as the number of infectives increasing and the number of removals increasing as the number of infectives decrease. The surprising fact is that the number of infectives computed from the numerical scheme does not really match the data. One way to efficiently solve the system as well as match the given data is to employ a process called parameter estimation which simply refers to the process of using sample data (data for infectives) to estimate the value of a population parameter (for example, the removal and infection rates). This can be done by doing a least-squares regression between the computed data from the MATLAB program described above and the true data collected to help estimate the best-fit parameters that describe the trend in the data. We write this as a subroutine **sir_lsq.m** that finds the least-squares error.

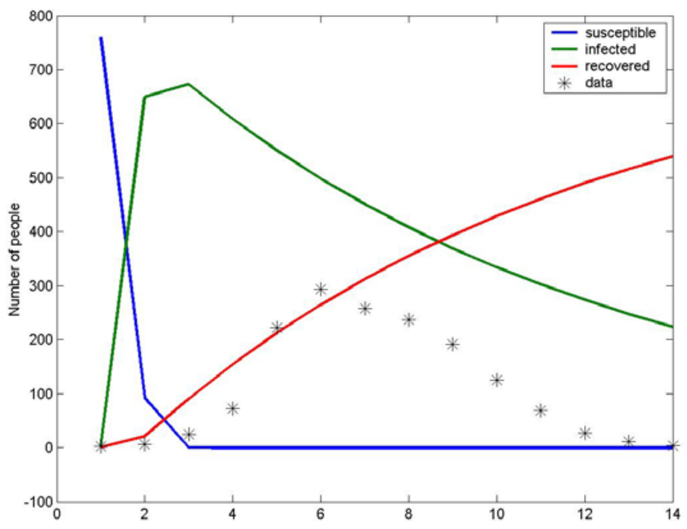


Figure 2: Evolution of the susceptibles (S), infectives (I) and removals (R) and data

```
function err = sir_lsq(r, data, tspan, y0)
    [t,y] = ode45(@sir_system,tspan,y0,[ ],r);
    Yc = y(:,2);
    err = sum((Yc-data').^2);
```

We will use this in conjunction with the MATLAB built-in fminsearch to obtain these best-fit parameters that represents the data better which is built into the GUI discussed next.

3.1. Graphical User Interface

One of the great visualization features available in MATLAB is called **GUIDE** [5] which is a MATLAB's graphical user interface (GUI) design environment. Guide can be started by typing guide on the command window or by selecting a new GUI. Once selected, the user gets access to a default guide layout editor that provides options for the user to design the layout of the GUI. These options available include push buttons, sliders, radio buttons, check boxes, Editable Texts, Static Texts, Pop-up Menu, Listbox, Toggle button and Table. We can also add Axes to create MATLAB plots. One also has the option to create panels to group controls and form a systematic hierarchy. Once the user has visualized how the screen should look, then the appropriate controls form the palette on the left can be chosen and placed in the position that the user wants. After placing the required controls, one can change the properties of the GUI by opening the property inspector for each of those controls. This inspector allows one to both view and set object properties for each of the controls selected. The typical properties that are changed include the "Name" property of the control which refers to the title that the user wants for the control, "String" property for the text the user wants to appear for the pushbutton controls as well as the pop-up menus. One of the most useful properties is the "Tag" property of the control that provides a unique identifier to help the user remember each of the controls. GUIDE will automatically use these tag properties to generate the appropriate MATLAB functions. Once all the controls are specified, then one can run the GUI which generates a MATLAB application as a ".fig" file which saves the layout that was created. Figure 3 displays a sample GUI that we created for the SIR system.

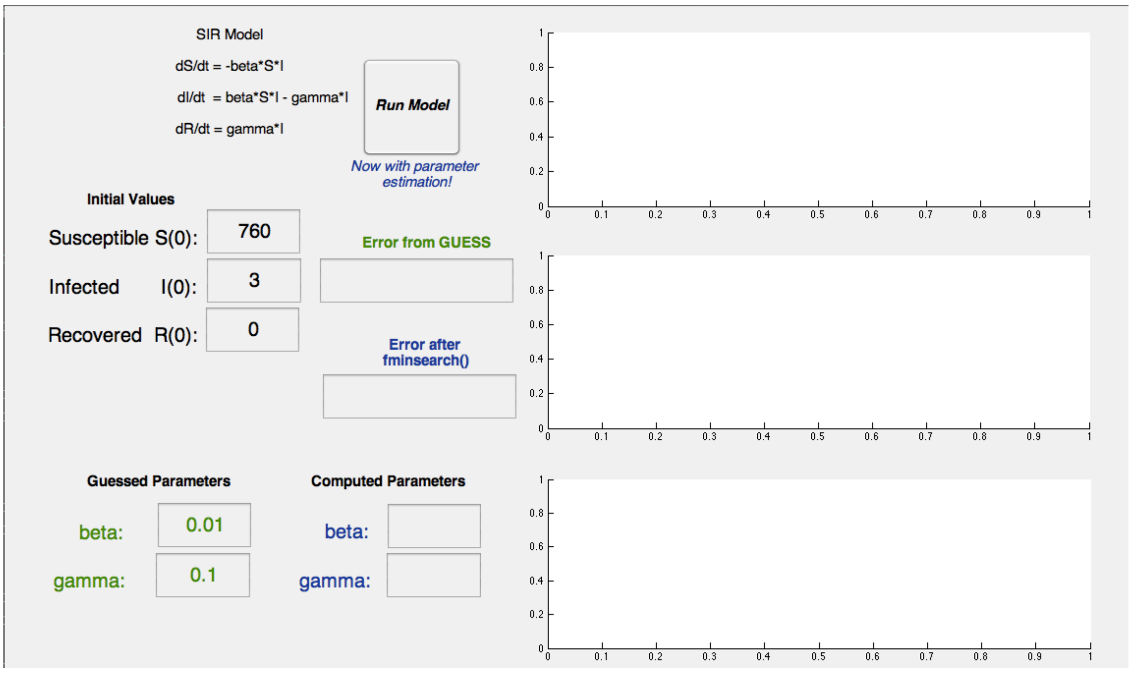


Figure 3: Blank GUI layout of the SIR system

As this was created, GUIDE also generated two files which includes a **.fig** file that contains the layout information and the **.m** file that contains the implementation code. The latter is an automatically generated with some space inside the code for the user to add any custom code that they want. These custom codes are referred to as **Callback** function. To implement the SIR model with parameter estimation to be able to obtain best fit parameters for the given data, we created a GUI shown in figure 4. The main Callback function that needed to be written was the "RunModel" pushbutton. When pressed, the pushbutton function not only plots the susceptibles, infectives and removals on their respective plots but also implements a non-linear regression that minimizes the error between the computed infectives and the true data of the infectives to obtain the best parameters.

```
function runModel_Callback(hObject, eventdata, handles)
```

The data is shared within the GUI using the "handles" structure. Also since the non-linear least-square regression requires an initial guess, the values of the removal and infection rates are entered as the initial guesses. Note that the values have to be converted from string to a double format. The following part shows how to obtain the initial values and parameters from user interface.

```
S0    = str2double(get(handles.S0,'String')); I0    = str2double(get(handles.I0,'String'));
R0    = str2double(get(handles.R0,'String'));
beta_guess  = str2double(get(handles.beta_guess,'String'));
gamma_guess = str2double(get(handles.gamma_guess,'String'));
```

We now repeat the main part of the code that solves the system (1) using Runge Kutta.

```
data = [3 6 25 73 222 294 258 237 191 125 69 27 11 4];
time = 1:length(data);
y0 = [S0 I0 R0];
tspan = time;
r0_guess = [beta_guess gamma_guess];
[t, y_guess] = ode45(@sir_system,tspan,y0,[ ],r0_guess);
```

In order to minimize the error between the computed values and the true data of the infectives collected we perform a non-linear least-squares algorithm to extract the best-fit parameters.

```
myFunc = @(r) sir_lsq(r, data, tspan, y0);
[params,fval,exitflag,output] = fminsearch(myFunc,r0_guess);
beta_comp = params(1);
gamma_comp = params(2);
set(handles.beta_comp,'String',beta_comp);
set(handles.gamma_comp,'String',gamma_comp);
```

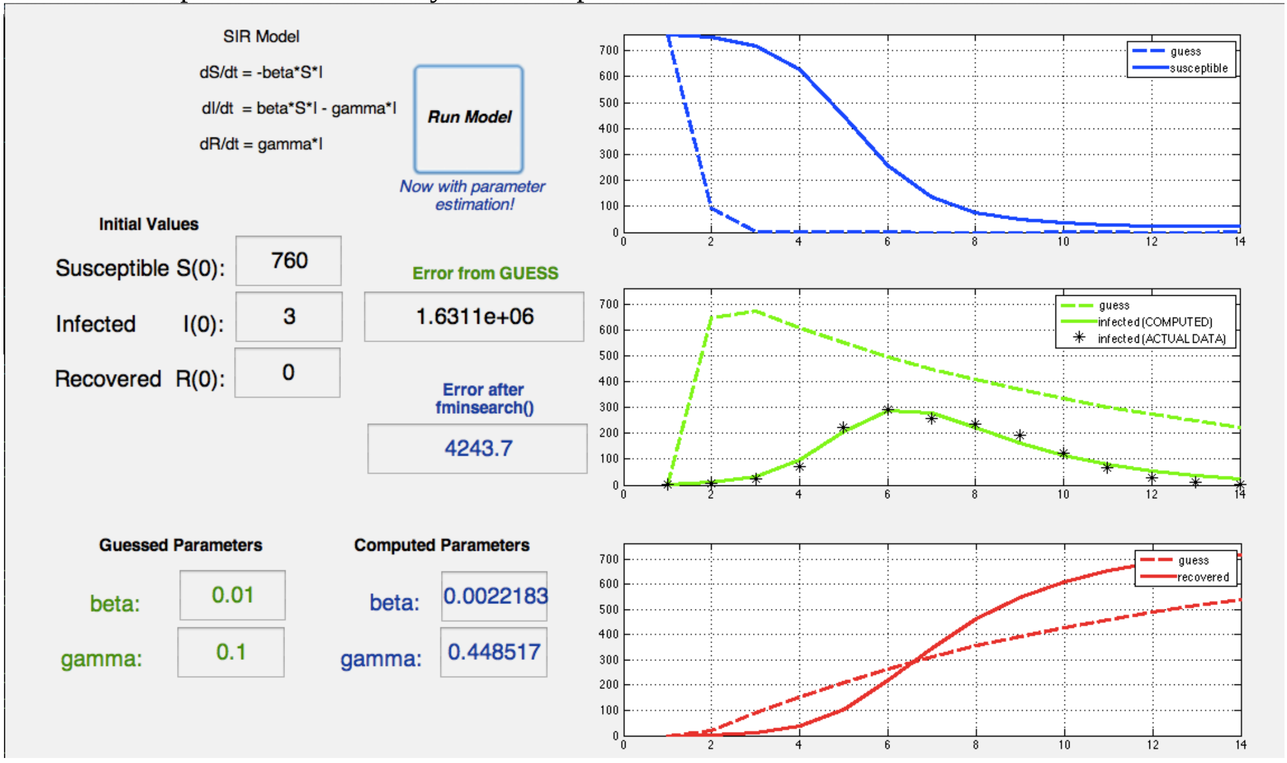
The values of the best-fit parameters are now stored in beta_comp and gamm_comp which described the trend the data of the infectives. We now use these newly computed best fit parameters and recompute the corresponding solution for the susceptibles, infectives and removals.In order to see the performance of the parameter estimation part of the code we compute the least-squares error

```
r0_comp = [beta_comp gamma_comp];
[t, y_comp] = ode45(@sir_system,tspan,y0,[ ],r0_comp);
```

from both the guessed as well as the computed parameters. This is reflected in the following piece of the code. Note that the respective errors are connected to the respective "tags" through the handles.

```
err_guess = sir_lsq(r0_guess,data,tspan,y0); set(handles.err_guess,'String',err_guess);
err_comp = sir_lsq(r0_comp,data,tspan,y0); set(handles.err_comp,'String',err_comp);
```

Finally we plot the solutions corresponding to both the guesses as well as the ones that correspond to the best fit parameters. We only show the part related to the infectives but the others are similar.



SIR Model

dS/dt = -beta*S*I

dI/dt = beta*S*I - gamma*I

dR/dt = gamma*I

Now with parameter estimation!

Initial Values

Susceptible S(0):	760
Infected I(0):	3
Recovered R(0):	0

Error from GUESS: 1.6311e+06

Error after fminsearch(): 4243.7

Guessed Parameters		Computed Parameters	
beta:	0.01	beta:	0.0022183
gamma:	0.1	gamma:	0.448517

Figure 4: SIR Model with parameter estimation showing the susceptibles, infectives and removals

One can see from Figure 4, that the best fit values for the given population was $\beta = 0.00222183$ and $\gamma = 0.448517$. The error between the computed and the experimental also drops significantly and the graphs are better aligned.

4. AMPL: The Optimization Approach

Besides the MATLAB ODE solver, we present the SIR parameter estimation problem from the optimization perspective using the software package AMPL (A Modeling Language for Mathematical Programming) [6]. Developed at Bell Laboratories, AMPL is a freeware optimization package that can solve large-scale optimization problems. The AMPL website (www.ampl.com) provides free downloads of the AMPL book, a student version of the software and supporting materials. In addition to the student software, one can upload and execute their AMPL code on the NEOS (Network-Enabled Optimization System) Server [7, 8, 9]. The NEOS Server (www.neos-server.org) is a free internet based server project at the University of Wisconsin-Madison that allows users to submit and solve optimization problems using a variety of state-of-the-art optimization solvers. The main benefit of NEOS is the ability to submit and execute quickly large-scale optimization problems. The AMPL student version is limited to solving problems of 300 variables and 300 constraints and objectives.

Although the SIR problem that we presented is a small-scale problem, AMPL and the NEOS Server are essential if one encounters large-scale problems that cannot be solved practically on desktop PCs using MATLAB. For example, one maybe interested in solving a large-scale optimization problem in management of water and energy resources or perhaps to conduct combustion research that typically involves a large-scale optimization problem. In the following section, we present and describe the AMPL code for the SIR parameter estimation problem.

4.1. AMPL Least-Squares Model

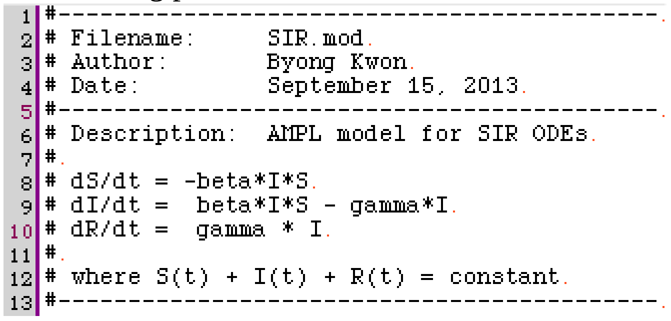
Using a generic text editor, we created the following three files:

(i) **SIR.mod** is the AMPL model file containing the optimization model;

(ii) **SIR.dat** is the AMPL data file containing data for the optimization model; and

(iii) **SIR.run** is a AMPL command file containing print commands for the NEOS Server.

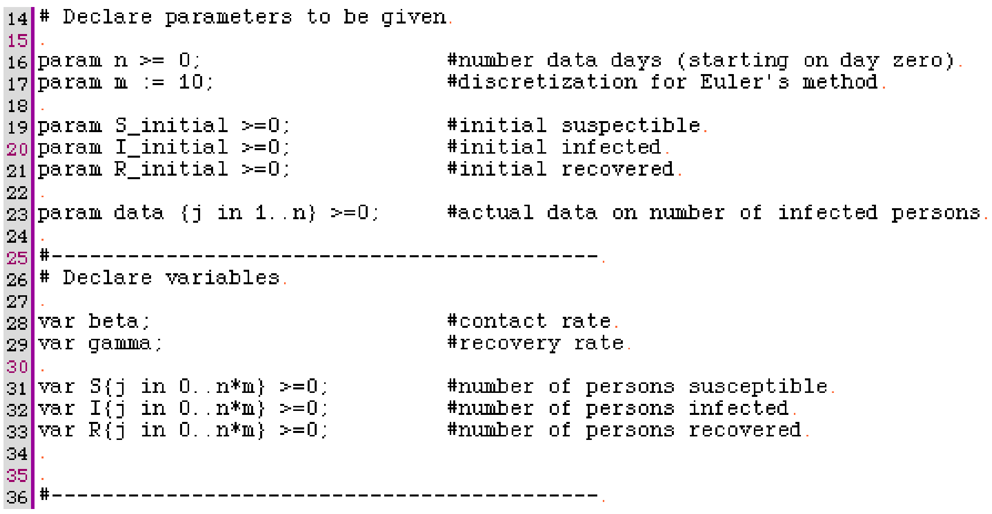
The file **SIR.mod** declares the objective function to minimize (or maximize), the constraints and variables for the problem. In Figure 5, we show lines 1-13 of the file **SIR.mod**. Please note that code lines beginning with the symbol # are comment lines not executed by the the program. Our SIR model assumes a fixed population rate and uses Euler's method to solve the IVPs.



```
 1 #-------------------------------------------
 2 # Filename:     SIR.mod
 3 # Author:       Byong Kwon
 4 # Date:         September 15, 2013
 5 #-------------------------------------------
 6 # Description:  AMPL model for SIR ODEs
 7 #
 8 # dS/dt = -beta*I*S
 9 # dI/dt =  beta*I*S - gamma*I
10 # dR/dt =  gamma * I
11 #
12 # where S(t) + I(t) + R(t) = constant
13 #-------------------------------------------
```

Figure 5: Excerpt from AMPL model file SIR.mod with introductory comments

In Figure 6, we present lines 14-36 of the file **SIR.mod**, which declare AMPL parameters and variables. AMPL parameters are data values provided by the user. The parameters must be declared in the AMPL model file, but the actual parameter data values can be provided in the model file (e.g. **SIR.mod**) or a separate data file (e.g. **SIR.dat**). For example, line 17 declares and provides a parameter data value to define the number of discretization steps (per day) in Euler's method. Alternative



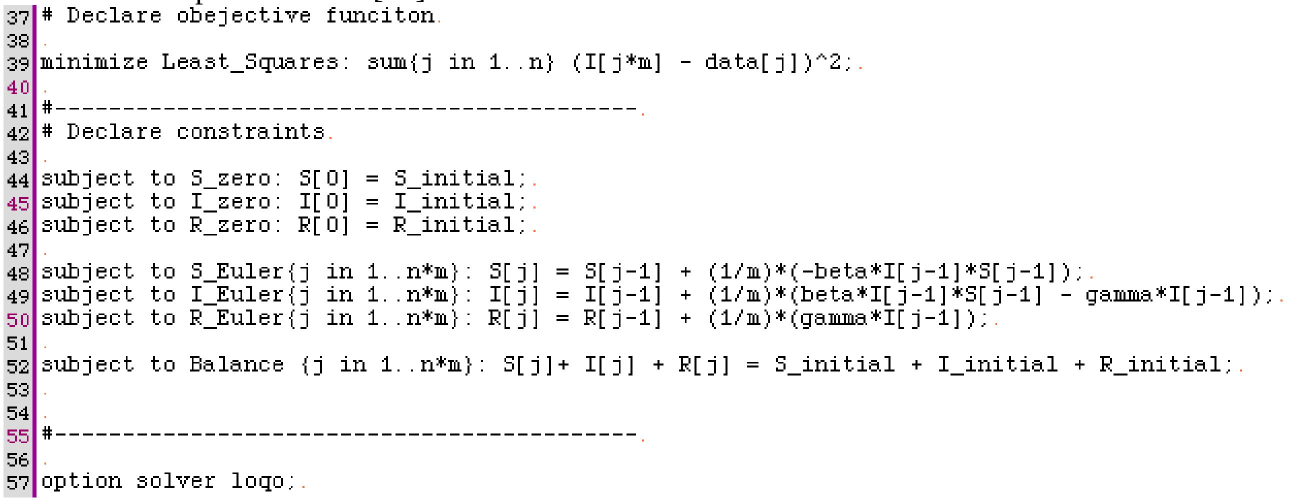
```
14 # Declare parameters to be given
15
16 param n >= 0;                 #number data days (starting on day zero)
17 param m := 10;                #discretization for Euler's method
18
19 param S_initial >=0;          #initial suspectible
20 param I_initial >=0;          #initial infected
21 param R_initial >=0;          #initial recovered
22
23 param data {j in 1..n} >=0;   #actual data on number of infected persons
24
25 #-------------------------------------------
26 # Declare variables
27
28 var beta;                     #contact rate
29 var gamma;                    #recovery rate
30
31 var S{j in 0..n*m} >=0;       #number of persons susceptible
32 var I{j in 0..n*m} >=0;       #number of persons infected
33 var R{j in 0..n*m} >=0;       #number of persons recovered
34
35
36 #-------------------------------------------
```

Figure 6: Excerpt from AMPL model file SIR.mod declaring parameters and variables

the model, initial conditions and the actual number of persons infected over the time period, respectively) but these parameter data values are provided in a separate data file, **SIR.dat** (see Figure 7). AMPL variables are those values that the user desires AMPL to find when solving the optimization problem. In Figure 6, lines 28-29, 31-33 declare variables that AMPL will find when solving the least squares problem.

In Figure 7, line 39 of the file **SIR.mod** declares the objective function that we desire to minimize, the least squares function concerning the number of infected persons,

$\sum_{j=1}^{n} [\mathrm{I}(\mathrm{j} * \mathrm{m}) - \mathrm{data}\,(\mathrm{j})]^2$ where n=14 days, data(j) is the actual number of persons infected at day j and I(j*m) is the estimated number of persons infected at day j. Function I(x) is evaluated at time j*m because I(x) is determined by Euler's method, which discretizes over m points per day. The optimization constraints are listed in lines 44-46 (the initial conditions), 48-50 (Euler's method for ODEs) and 52 (SIR model assumption that population is constant). The last line of code in file SIR.mod, line 57, commands AMPL to use the optimization solver LOQO, a nonlinear solver based on the interior-point method [10].



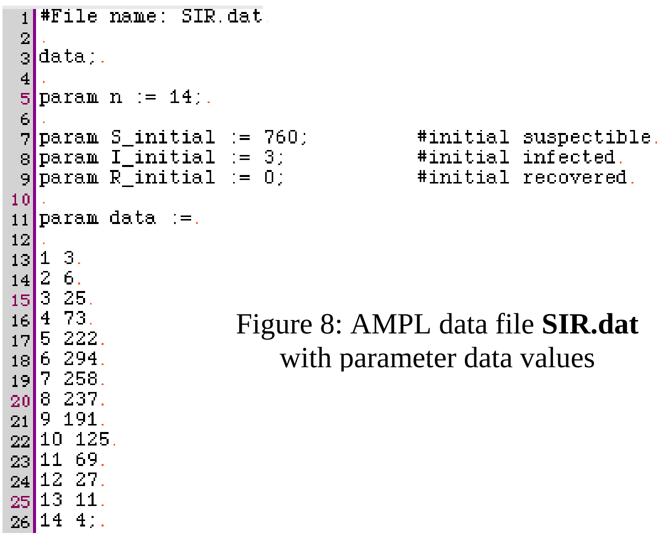
```
37 # Declare obejective funciton
38 
39 minimize Least_Squares: sum{j in 1..n} (I[j*m] - data[j])^2;
40 
41 #--------------------------------------------
42 # Declare constraints
43 
44 subject to S_zero: S[0] = S_initial;
45 subject to I_zero: I[0] = I_initial;
46 subject to R_zero: R[0] = R_initial;
47 
48 subject to S_Euler{j in 1..n*m}: S[j] = S[j-1] + (1/m)*(-beta*I[j-1]*S[j-1]);
49 subject to I_Euler{j in 1..n*m}: I[j] = I[j-1] + (1/m)*(beta*I[j-1]*S[j-1] - gamma*I[j-1]);
50 subject to R_Euler{j in 1..n*m}: R[j] = R[j-1] + (1/m)*(gamma*I[j-1]);
51 
52 subject to Balance {j in 1..n*m}: S[j]+ I[j] + R[j] = S_initial + I_initial + R_initial;
53 
54 
55 #--------------------------------------------
56 
57 option solver loqo;
```

Figure 7: Excerpt from AMPL model file **SIR.mod** declaring objective function to minimize, constraints and optimization solver

In the next section, we present the AMPL data file SIR.dat and the command file SIR.run.

4.2. AMPL data and command files

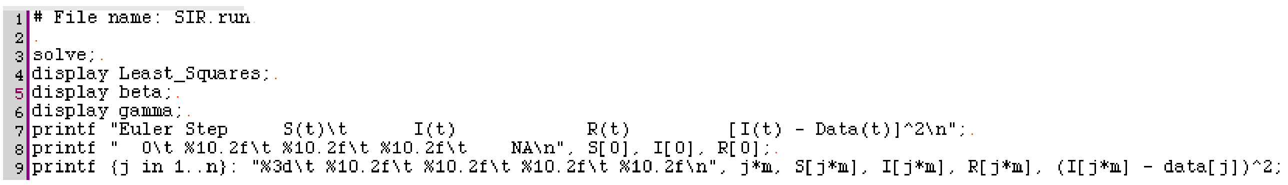
In Figure 8, we present the AMPL data file SIR.dat, which contains parameter data values defined in the AMPL model file SIR.mod. Line 3 in Figure 8 tells AMPL that file SIR.dat is a data file. Line 5 provides the number of days in the SIR model (n = 14) and lines 13-27 list the actual number of infected persons on days 1 through 14.



```
1  #File name: SIR.dat
2  
3  data;
4  
5  param n := 14;
6  
7  param S_initial := 760;          #initial suspectible
8  param I_initial := 3;            #initial infected
9  param R_initial := 0;            #initial recovered
10 
11 param data :=
12 
13 1 3
14 2 6
15 3 25
16 4 73
17 5 222
18 6 294
19 7 258
20 8 237
21 9 191
22 10 125
23 11 69
24 12 27
25 13 11
26 14 4;
```

Figure 8: AMPL data file **SIR.dat** with parameter data values

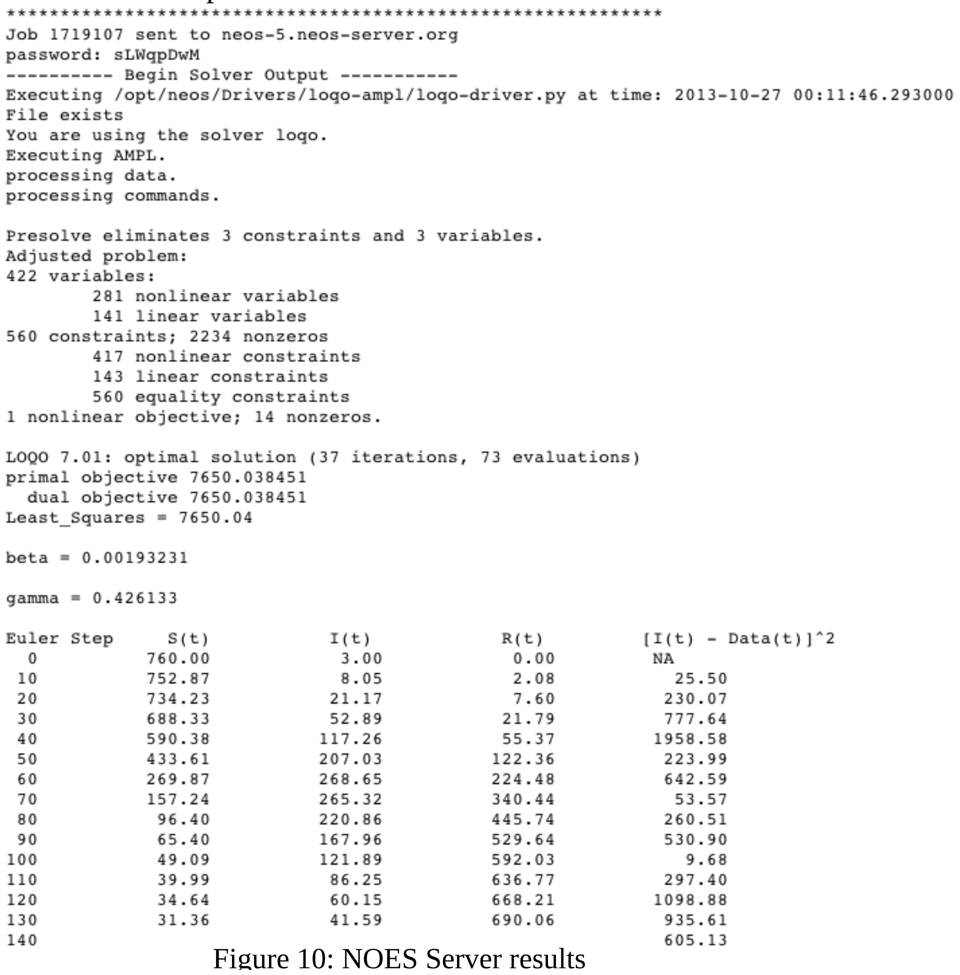
In Figure 9, we present the AMPL command file that instructs the NEOS server to solve the optimization problem (line 3), display the final value of the minimized objective function (the least squares solution, line 4), display the beta and gamma variables (lines 5-6) and execute Unix print commands to display other data values (lines 7-9).



```
1 # File name: SIR.run
2 
3 solve;
4 display Least_Squares;
5 display beta;
6 display gamma;
7 printf "Euler Step     S(t)\t      I(t)            R(t)         [I(t) - Data(t)]^2\n";
8 printf "  0\t %10.2f\t %10.2f\t %10.2f\t    NA\n", S[0], I[0], R[0];
9 printf {j in 1..n}: "%3d\t %10.2f\t %10.2f\t %10.2f\t %10.2f\n", j*m, S[j*m], I[j*m], R[j*m], (I[j*m] - data[j])^2;
```

Figure 9: AMPL command file SIR.run with print commands

The last step is to submit the three AMPL files, SIR.mod, SIR.dat, SIR.run, to the NEOS Server. On the NEOS Server page listing optimization solvers (http://www.neosserver.org/neos/solvers/index.html), we scroll toward the bottom of the page and click ``LOQO [AMPL input]" under the category Nonlinearly Constrained Optimization. This link will lead to the page where we upload the three AMPL files above. Provided that there are no errors in your AMPL files, the NEOS Server will email you the results. See Figure 10 for NEOS Server results for the SIR parameter estimation problem.



```
*************************************************************
Job 1719107 sent to neos-5.neos-server.org
password: sLWqpDwM
---------- Begin Solver Output -----------
Executing /opt/neos/Drivers/loqo-ampl/loqo-driver.py at time: 2013-10-27 00:11:46.293000
File exists
You are using the solver loqo.
Executing AMPL.
processing data.
processing commands.

Presolve eliminates 3 constraints and 3 variables.
Adjusted problem:
422 variables:
        281 nonlinear variables
        141 linear variables
560 constraints; 2234 nonzeros
        417 nonlinear constraints
        143 linear constraints
        560 equality constraints
1 nonlinear objective; 14 nonzeros.

LOQO 7.01: optimal solution (37 iterations, 73 evaluations)
primal objective 7650.038451
  dual objective 7650.038451
Least_Squares = 7650.04

beta = 0.00193231

gamma = 0.426133

Euler Step     S(t)           I(t)            R(t)         [I(t) - Data(t)]^2
  0          760.00           3.00            0.00          NA
 10          752.87           8.05            2.08            25.50
 20          734.23          21.17            7.60           230.07
 30          688.33          52.89           21.79           777.64
 40          590.38         117.26           55.37          1958.58
 50          433.61         207.03          122.36           223.99
 60          269.87         268.65          224.48           642.59
 70          157.24         265.32          340.44            53.57
 80           96.40         220.86          445.74           260.51
 90           65.40         167.96          529.64           530.90
100           49.09         121.89          592.03             9.68
110           39.99          86.25          636.77           297.40
120           34.64          60.15          668.21          1098.88
130           31.36          41.59          690.06           935.61
140                                                        605.13
```

Figure 10: NOES Server results

Since the underlying optimization problem is a quadratic problem (and thereby convex), we expect the least squares solution to be the global solution to the problem. The AMPL least squares solution and values for beta and gamma in Figure 7 are close to those values using MATLAB's ODE solver. The slight discrepancies between the MATLAB and the AMPL solutions are due to the fact that AMPL used Euler's method to solve the ODEs and Runge-Kutta was used in MATLAB. Nevertheless the values of the best fit parameters agree very well.

5. Conclusions

In this work, a framework for solving real-world problems using mathematical modelling and technology enhanced simulation approach was presented. A benchmark application in disease epidemics was considered and the numerical implementation was presented in two different software platforms including MATLAB and AMPL. The major contributions of the paper include the development of the MATLAB Graphical User Interface (GUI) and comparing the answers against another powerful optimization software AMPL that has advantages over MATLAB when it comes to solving systems with big data. The numerical results agreed very well between the different software platforms and the presentation of the programs were done via literate programming. We hope that this work will be useful for many educators who may be interested in employing these software tools in their research and education.

References

[1] Seshaiyer, P. (2012) Transforming practice through undergraduate researchers, *Council on Undergraduate Research Focus*, 33(1) 8-13.

[2] Paige, R., Seshaiyer, P. and Toda, M. (2007) Student misconceptions caused by misuse of technology, *International Journal for Technology in Mathematics Education*, 14(4), 189-196.

[3] Kermack, W.O. and McKendrick, A.G. (1927) A Contribution to the Mathematical Theory of Epidemics, *Proceedings of the Royal Society of London A*, 115, 700-721.

[4] Knuth, D.E. (1984), Literate programming. *The Computer Journal* 27(2), 97-111.

[5] Mathworks (2013), Creating Graphical User Interfaces. Online http://www.mathworks.com/help/pdf_doc/matlab/buildgui.pdf.

[6] Fourer, R., Gay, D. and Kernighan, B. (2003), *AMPL: A Modeling Language for Mathematical Programming*. Brooks/Cole, 2nd ed.

[7] Czyzyk, J., Mesnier, M. and More, J. (1998), The NEOS Server, *IEEE Journal on Computational Science and Engineering*,5 (3), 68-75.

[8] Gropp, W and More, J. (1997), *Optimization Environments and the NEOS Server*, pp. 167-182. Cambridge University Press.

[9] Dolan, E. (2001) *The NEOS Server 4.0 Administrative Guide*. Mathematics and Computer Science Division, Argonne National Laboratory.

[10] Vanderbei, R., LOQO. Online at http://www.princeton.edu/ rvdb/loqo/LOQO.html.