Differential Equations (Aggregate) Models with MATLAB and Octave
A Predator-Prey Example

Differential equations in biology are most commonly associated with aggregate models. Aggregate models consider a population as a collective group, and capture the change in the size of a population over time. Consider for example, the classic Lotka-Volterra predator prey equations:

$$\frac{dA}{dt} = rA - \alpha AB \tag{1}$$
$$\frac{dB}{dt} = -\delta B + \beta AB \tag{2}$$

where $A(t)$ represents the size of the prey population at time t, and $B(t)$ represents the size of the predator population at time t. Without the presence of predators (i.e. when $B(t) = 0$), the prey population is assumed to experience exponential growth $\left(\frac{dA}{dt} = rA\right)$. Similarly, if no prey are present (when $A(t) = 0$), the predator population will decrease exponentially $\left(\frac{dB}{dt} = -\delta B\right)$. The interaction of predators and prey, represented by the AB terms, have a negative impact on the prey and a positive impact on the predators. This system of differential equations models the change in the size of the prey and predator populations, collectively, over time.

MATLAB is a technical computing environment for high-performance ***numeric*** (and not typically symbolic) computation and visualization. It is a proprietary software used by researchers, educators, and students in industry and academia. GNU Octave is an open source high-level language, primarily intended for numerical computations that is mostly compatible with MATLAB. For the purpose of these examples, all of the code and commands can be used in MATLAB or Octave and nearly identical results would be produced.

1 Numerical Solutions to Differential Equations

Numerical computing environments such as MATLAB and Octave are not intended to solve differential equations models symbolically. In other words, given a differential equation such as $\frac{dx}{dt} = f(t, x)$, one should not expect to get a solution that is an equation of the form $x = g(t)$. Computer algebra systems like Mathematica and Maple were intended to be used to provide these types of answers, in examples where closed form solutions can be determined. Numerical computing environments can approximate the solution, $x = g(t)$, by generating a sequence of points $(t_0, x_0), (t_1, x_1), \cdots, (t_N, x_N)$ that are reasonably close to points (t, x) on the curve of the true solution. These solution sequences are generated using one of a set of numerical differential equation solution techniques, such as a Runge-Kutta method or the Adams-Bashforth-Moulton method.

Consider the initial value problem (a system of differential equations together with the appropriate initial conditions) below, for example:

$$\frac{dx}{dt} = 2tx \tag{3}$$
$$x(0) = 1 \tag{4}$$

This inital value problem has a closed form solution, $x(t) = e^{t^2}$. If we use the Runge-Kutta (4,5) method in MATLAB or Octave, we generate the approximation values found in Table 1 (as well as the true solution values and the associated error) and can visualize the accuracy of our result in Figure 1.

t-value	**Approximation**	**True Value** (e^{t^2})	**Error (|Approx - True|)**
0.000000	1.0000000000	1.0000000000	0.0000000000
0.012500	1.0001562623	1.0001562622	0.0000000001
0.025000	1.0006251954	1.0006251954	0.0000000000
0.037500	1.0014072392	1.0014072392	0.0000000000
0.050000	1.0025031276	1.0025031276	0.0000000000
0.062500	1.0039138896	1.0039138893	0.0000000002
⋮	⋮	⋮	⋮
0.462500	1.2385065419	1.2385065394	0.0000000026
0.475000	1.2531056633	1.2531056625	0.0000000008
0.487500	1.2682731473	1.2682731490	0.0000000017
0.500000	1.2840254167	1.2840254167	0.0000000000

Table 1: Comparing numerical result to the true solution

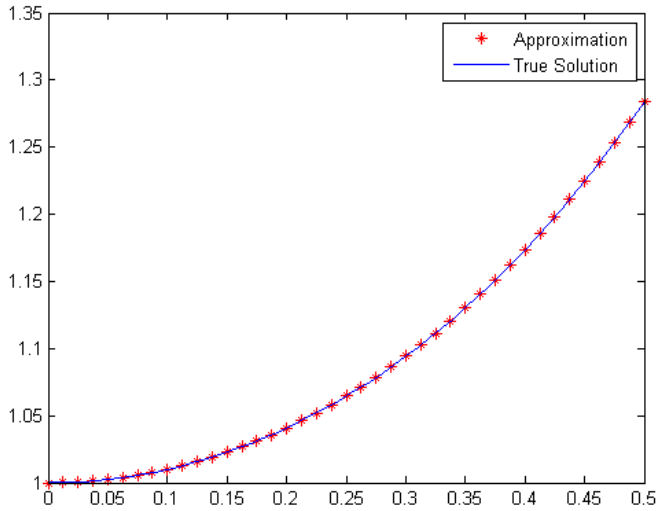


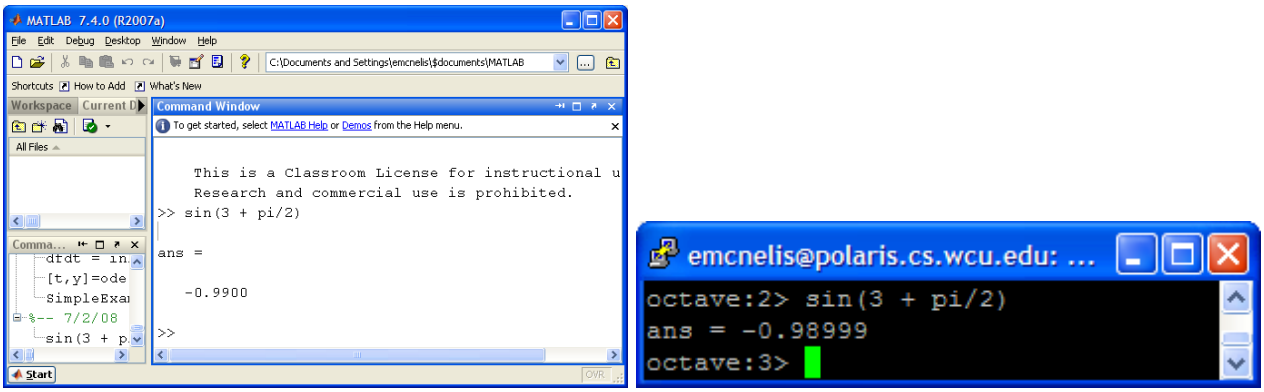
Figure 1: Comparing numerical results to the true solution

MATLAB and Octave have built-in initial value problem solvers, so it is not the responsibility of the user to implement these methods. In order to solve an initial value problem, the user must create code that specifies the system of differential equations and a set of procedures that tell MATLAB or Octave what initial values and files to use.

2 Communicating with MATLAB and Octave ...

2.1 Through the Command Line

The primary way of giving MATLAB or Octave a command is by typing it on its command line. MATLAB has a graphical user interface that includes a panel for the command window. The GNU Octave interface is command line only. In either case, the user simply needs to type the commands at the command line prompt and hit enter. For example, we can calculate $\sin\left(3+\frac{\pi}{2}\right)$ by typing `sin(3 + pi/2)` in either MATLAB or Octave, as Figure 2 illustrates.



(a) Command Line MATLAB (b) Command Line Octave

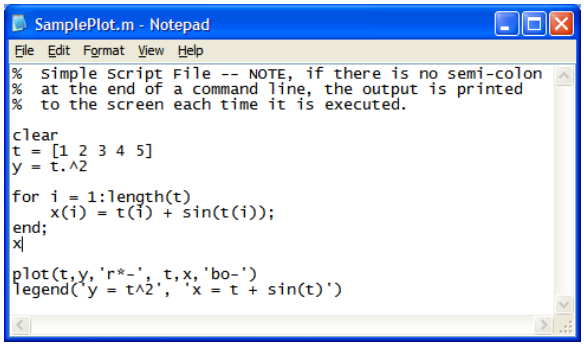
Figure 2: Using the command line in MATLAB and Octave

2.2 Through the Script Files

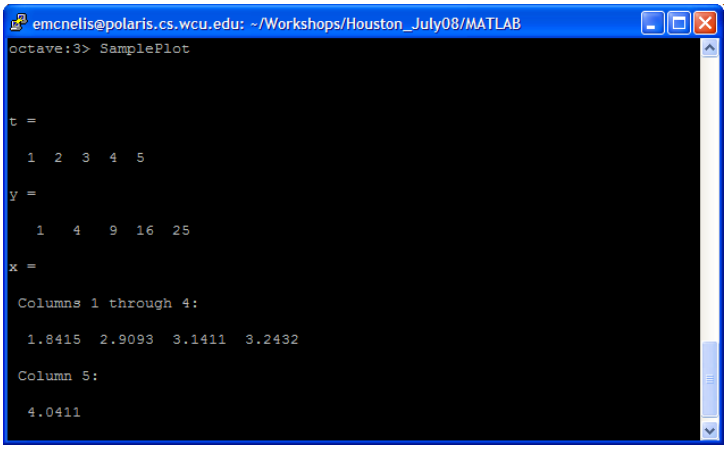
Commands associated with solving an initial value problem are simple enough to be entered line-by-line at the command prompt, but are more easily presented in a *script file*.

Definition 1 (Script File) *A* ***script file*** *in MATLAB or Octave is simply a text file that contains the commands that one would type at the prompt. It must be saved with a* `.m` *extension (e.g. saving the file as* SamplePlot.m*).*

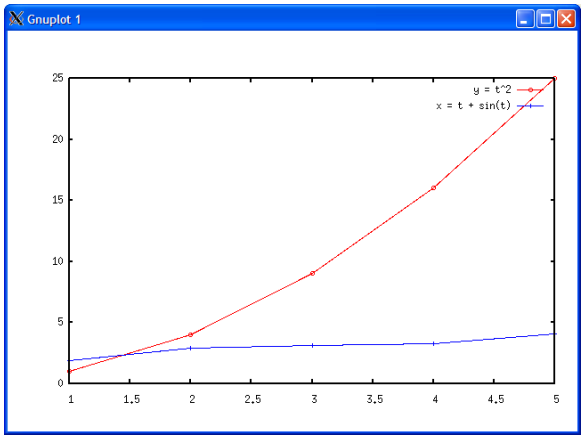
When generating a final product or an answer that requires entering a sequence of commands, it is often advantageous to put these commands in a script file. When this script file is saved with a `.m` extension, the user simply needs to type the name of the script file (without the `.m` extension) at the command prompt and MATLAB or Octave looks for that file in its current directory and executes the command lines in the file one at a time. When changing parameter values and repeating actions or executing a fairly long list of commands, this comes in very handy. The illustration in Figure 3 shows the use of a simple script file and its output.



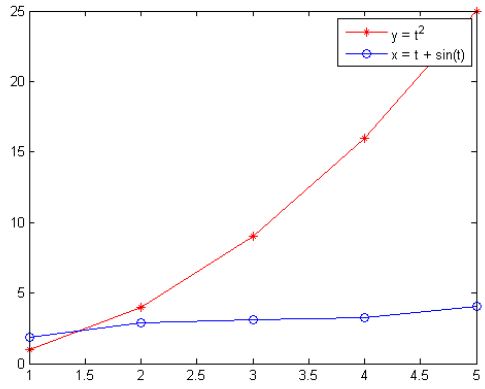
(a) Script File `SamplePlot.m` in Notepad



(b) Invoking Script File from Octave Command Line



(c) Graphical Results from Octave



(d) Graphical Results from MATLAB

Figure 3: Using a Script File

2.3 Through the Function Files

In addition to writing script files, we can create user-defined functions using m-files (also text files with the `.m` extension). MATLAB and Octave have an extensive library of mathematical functions built in, but there is often a need for a user to create their own functions. New functions may be added to the software "vocabulary" as function files.

Definition 2 (Function Files) ***Function files*** *are simply text files saved with a* `.m` *extension that designate a set of required input, the names of things to be output, and the means in which to calculate said output.*

The syntax for starting a function file is very specific. The first line of text in the function file must be presented in the following manner:

$$\texttt{function}\ [list\ of\ outputs] = function\_name(list\ of\ input\ variables)$$

For instance, if we wanted to create a function that calculates the mean and standard deviation of a set of data specified in an input vector, we can create a function file that looks like this (it would be saved as a text file named `stat.m`):

```
function [mean,stdev] = stat(x)
% STAT Interesting statistics.
% The vector x contains the observations
n = length(x);
mean = sum(x) / n;
stdev = sqrt(sum((x - mean).^2)/(n-1));
```

Note that this function requires a single input, called `x`, which is understood to contain a list of values of single variate data. The function returns two pieces of output, `mean` and `stdev`. The code following the comments expresses how to calculate the `mean` and `stdev` values the function returns. In order to use this function, we can type the following at the prompt:

```
>> [moe, larry] = stat([1 3 8 14 27])
```

and end up with the results:

```
moe =

   10.6000


larry =

   10.4547
```

Note that we specify two names to store the returned output from the `stat` function, but we do not have to use the same names stated in the function file (those are local variable names).

3 Solving Initial Value Problems in MATLAB or Octave

3.1 A Simple Single-Dependent-Variable Example

The most commonly used initial value problem solver for ordinary differential equations is `ode45`. It is based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair, and is the is the best function to apply as a first try for most problems. Typical syntax for calling the `ode45` function is as follows:

`[t,y] = ode45(@`*`diff_eqn_file`*`, [`*`start_time`*`, `*`end_time`*`], `*`initial_values`*`)`

where `t` is the output vector of independent variable values, `y` is the output vector of dependent values, *`diff_eqn_file`*`.m` is the function file that specifies the system of differential equations to be solved, *`start_time`* and *`end_time`* are the start and end value for the independent variable (typically time), and *`initial_values`* gives the values of the solution at *`start_time`*.

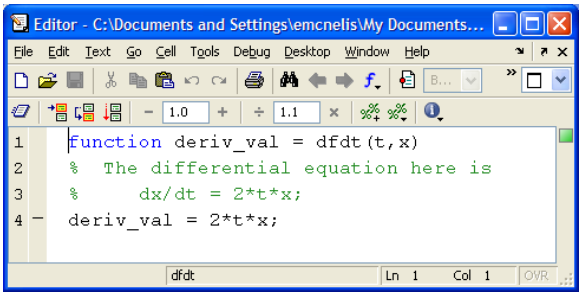


Figure 4: The differential equations function file, `dfdt.m`, in the MATLAB editor.

For instance, in our initial example:

$$\frac{dx}{dt} = 2tx \tag{5}$$

$$x(0) = 1 \tag{6}$$

our *diff_eqn_file* could be called `dfdt.m` (see Figure 4) and the script file used to generate the solution vectors and graphs could be saved as `SimpleExample.m` and would contain the following commands:

```
%  Simple Example
%  Initial Value Problem:
%    dx/dt = 2*x*t,  x(0) = 1  (Differential equation stored in file:
%    dfdt.m)
%  True solution is x(t) = exp(t^2).

%  Use ode45 to generate the vector of t and x-values in the solution:
%  [t,x]=ode45(@function_file_name, [start_time, end_time], initial_value)
%  Note the "@" needed in front of the function file name (and lack of ".m")
[t,x] = ode45(@dfdt,[0,0.5],1);

%  Plot our numerical approximation in red using *'s to mark the points:
plot(t,x,'r*')

hold on %  Hold that plot

%  Create a vector of independent variable (t) values from 0 to 0.5, spaced
%  0.01 apart.  Then calculate their associated x(t) values, exp(t^2).
tvals = [0:0.01:0.5];
true_solution = exp(tvals.^2);

%  Plot the true solution in blue
plot(tvals,true_solution,'b');

%  Add a legend to the plot
legend('Approximation','True Solution')
```

Note: typically we would not know the equation of the true solution to make the comparisons we did here, and we would have stopped after plotting our numerical approximation to the solution.

3.2 A Multiple-Dependent-Variables Example: Predator-Prey

The above example has only one dependent variable, x, and thus only one differential equation, $\frac{dx}{dt}$ to solve. In biological differential equations models, it is more common to have multiple dependent variables, and hence a system of two or more interrelated differential equations. Take for example the Lotka-Volterra predator prey equations

$$\frac{dA}{dt} = rA - \alpha AB \tag{7}$$

$$\frac{dB}{dt} = -\delta B + \beta AB \tag{8}$$

where $A(t)$ and $B(t)$ represent the number of prey and predators at time t, respectively. The rate of change in the prey population, $\frac{dA}{dt}$, depends not only on the amount of prey present (A), but on the number of predators present (B) too. The same can be said about the predators. Thus, we have a *coupled* system of differential equations. In this case, we must be more sophisticated in expressing our variables and differential equations.

The initial value problem solvers for ordinary differential equations require either (a) a single dependent variable, or (b) a single vector of dependent variables. In a similar fashion, the system of differential equations must be represented in vector notation. This is not as difficult as it sounds. If we have multiple dependent variables, we simply collect them into one vector of values. For instance, if our dependent variables are A and B, we can combine them into one vector, $\mathbf{x}$ as

$$\mathbf{x} = \begin{bmatrix} A \\ B \end{bmatrix} \text{ or indicating time dependence by } \mathbf{x}(t) = \begin{bmatrix} A(t) \\ B(t) \end{bmatrix} \tag{9}$$

We can then refer to A as $\mathbf{x}_1$, the first component of our vector $\mathbf{x}$, and to B as $\mathbf{x}_2$, the second component of our vector $\mathbf{x}$. Keeping this vector notation in mind, we can now express our system of differential equations, (7) and (8), as a single vector differential equation:

$$\begin{bmatrix} \frac{dA}{dt} \\ \frac{dB}{dt} \end{bmatrix} = \begin{bmatrix} rA - \alpha AB \\ -\delta B + \beta AB \end{bmatrix} \tag{10}$$

or in vector notation

$$\frac{d\mathbf{x}}{dt} = \begin{bmatrix} r\mathbf{x}_1 - \alpha \mathbf{x}_1\mathbf{x}_2 \\ -\delta \mathbf{x}_2 + \beta \mathbf{x}_1\mathbf{x}_2 \end{bmatrix} \tag{11}$$

Create the function file, `pred_prey_odes.m` which looks like:

```
function deriv_vals = pred_prey_odes(t,x)
%  We're calculating the values of the differential equations:
%     dA/dt = 2 A - 1.2 A*B   or deriv_vals = [ 2*x(1) - 1.2*x(1)*x(2) ]
%     dB/dt = - B + 0.9 A*B                   [ -x(2) + 0.9*x(1)*x(2)  ]
%  But we assume our input vector, x, is really:
%              [ A ]
```

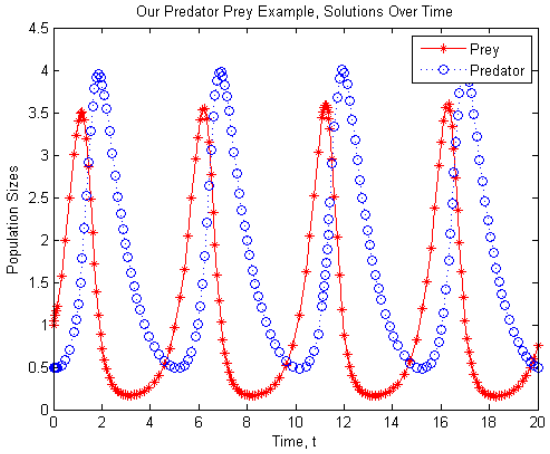
```
%           x =[   ]
%              [ B ]
%  so x(1) = A and x(2) = B.

%  Generate a vector of zeros the same size as the vector x.  The 0's are
%  just placeholders for our dA/dt and dB/dt values (or dx(1)/dt and
%  dx(2)/dt values)
deriv_vals = zeros(size(x))

%  Calculate dA/dt, the first value in the deriv_vals vector.  Remember A =
%  x(1) and B = x(2).
deriv_vals(1) = 2*x(1) - 1.2*x(1)*x(2);

%  Calculate dB/dt, the second value in the deriv_vals vector.  Remember A =
%  x(1) and B = x(2).
deriv_vals(2) = -1*x(1) + 0.9*x(1)*x(2);
```

This defines the Lotka-Volterra predator-prey system with $r = 2$, $\alpha = 1.2$, $\delta = -1$, and $\beta = 0.9$ in a vector fashion acceptable to MATLAB and Octave.

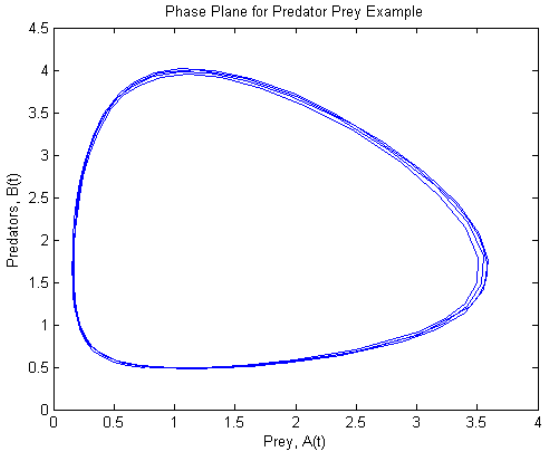
We can also create a script file, `PredPreyScript.m`, that looks like:

```
%  PredPreyScript.m:  The script file to run the predator-prey example:
%     dA/dt = 2 A - 1.2 A*B
%     dB/dt = - B + 0.9 A*B
%  The system of differential equations is specified in the file
%  pred_prey_odes.m

%  Define initial and final times
t0 = 0;
tf = 20;

%  Define initial values vector with A(0) = 1, and B(0) = 0.5
init_vals = [1; 0.5];

%  Use ode45 to generate approximations to the solution.  Keep in mind that
%  t is a vector of one dimension, just time, but x will be a matrix.  It's
%  first column will contain the A(t) values and it's second column will contain
%  the B(t) values.
[t,x] = ode45(@pred_prey_odes,[t0,tf],init_vals);

%  Peel off our A(t) and B(t) values for easier reading and plotting:
A = x(:,1);  %  A is the first row of the x matrix, collecting all columns.
B = x(:,2);  %  B is the second row of the x matrix, collecting all columns

%  Plot A(t) values in red with *'s, B(t) values in blue with circles.
plot(t,A,'r*-',t,B,'bo:')
xlabel('Time, t')
```

```
ylabel('Population Sizes')
title('Our Predator Prey Example, Solutions Over Time')
legend('Prey','Predator')

%  Plot the phase plane, A-values versus B-values
figure %  This opens a new window for an additional plot
plot(A,B)
xlabel('Prey, A(t)')
ylabel('Predators, B(t)')
title('Phase Plane for Predator Prey Example')
```



(a) Plot of $A(t)$ and $B(t)$ over Time



(b) Phase Plane, $A(t)$ versus $B(t)$

Figure 5: Results from our predator-prey example

By typing `PredPreyScript` at the Octave prompt we generate the graphs found in Figure 5. We can easily alter our parameter values, r, α, δ, and β and see how this affects our results by rerunning our script file. What situations can you create?