MATLAB REVIEW

The focus of this document is to review common, useful, higher-level Matlab operations that will be employed on your assignments, such as: ordinary differential equation time integrators (i.e. ode45 and ode15s, including event functions), curve fitting to a model, plotting options for x-y plots, and solving sets of nonlinear equations (fsolve). Additional examples will be made available for future topics. None of these exercises are required work, but are intended to be a great review and resource for future questions.

Task 1: Time integration of ODE's

Typically, a set of unsteady chemical engineering equations can be cast into the following form:

accumulation = input – output + generation

which leads to an unsteady set of equations of the form:

$$\frac{d\vec{f}}{dt} = \vec{g}(\vec{f}, t),$$

where time is the independent variable, $\vec{f}$ is the (unknown) vector of dependent variables, and $\vec{g}$ is the vector of functions giving the rate-of-change of each dependent variable. Given an initial condition for each dependent variable, $\vec{f}_0$, the equations can be integrated in time to find $\vec{f}(t)$. However, while an analytical expression may not be available for the solution, a numerical integration technique can always find a solution to this type of problem.

Open *odesintegrate.m*. This file solves for the concentrations of two chemical species in a solution a batch reactor (perfect mixing, isothermal, constant volume) using Matlab's ode solvers. The system is:

$$A \xrightarrow{r1} B \xrightarrow{r2} C$$

$$r_1 = k_1[A]$$

$$r_2 = k_2[B]$$

and applying the general chemical engineering balance equation on the reactor, we find that the ode's governing species are:

$$\frac{d[A]}{dt} = -k_1[A]$$

$$\frac{d[B]}{dt} = k_1[A] - k_2[B]$$

$$\frac{d[C]}{dt} = k_2[B]$$

Since [C] can be expressed in terms of [A] and [B] using mass balances, the third ode is redundant and need not be followed explicitly. Consider the following initial condition: [A0] = 1 M, [B0]=[C0] = 0 M. Through two reaction steps, the [A] is converted to [C].

Look at the example file *odesintegrate.m*. In particular, look at the calling sequence of the ode solver, which is generally:

```
[t,f]=ode_solver(@derivative_function_name,[t0,tf],f0,options,param);
```

where the ode solver can be ode45 or ode15s, for example. The returned values are a column vector t of time steps, and a matrix f, where each column corresponds to a variable.

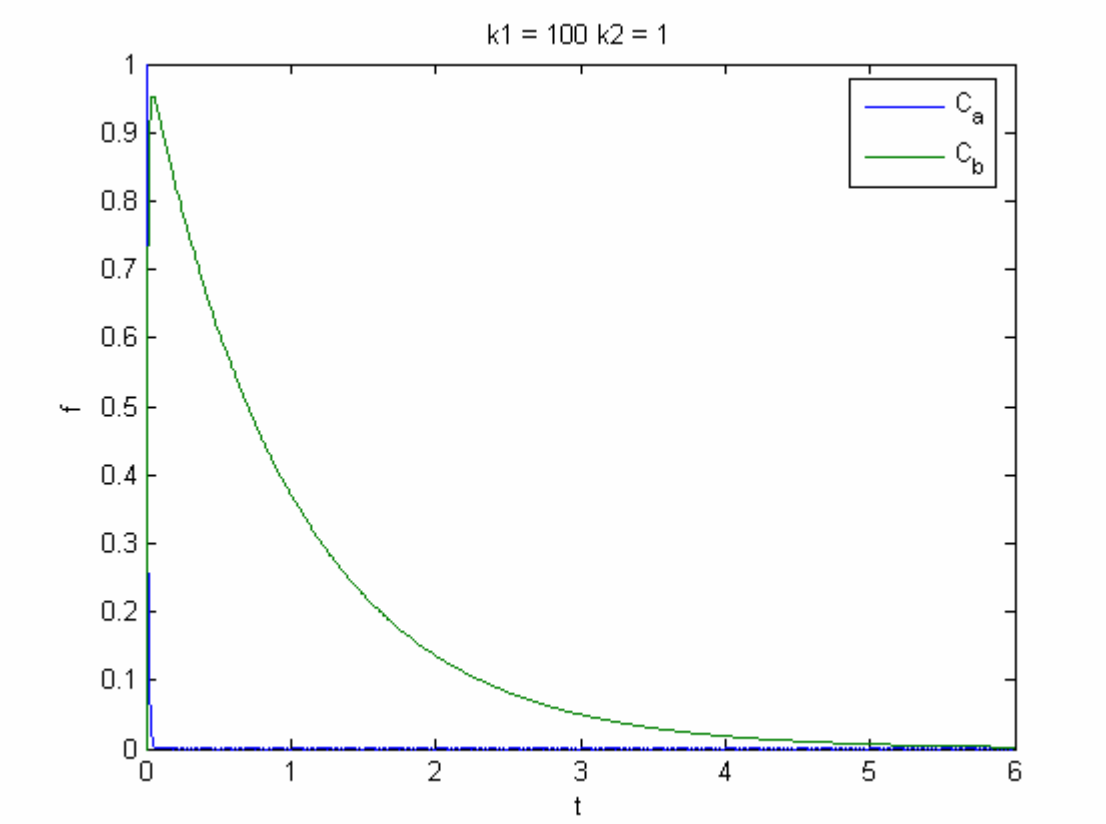
The derivative function has the form:

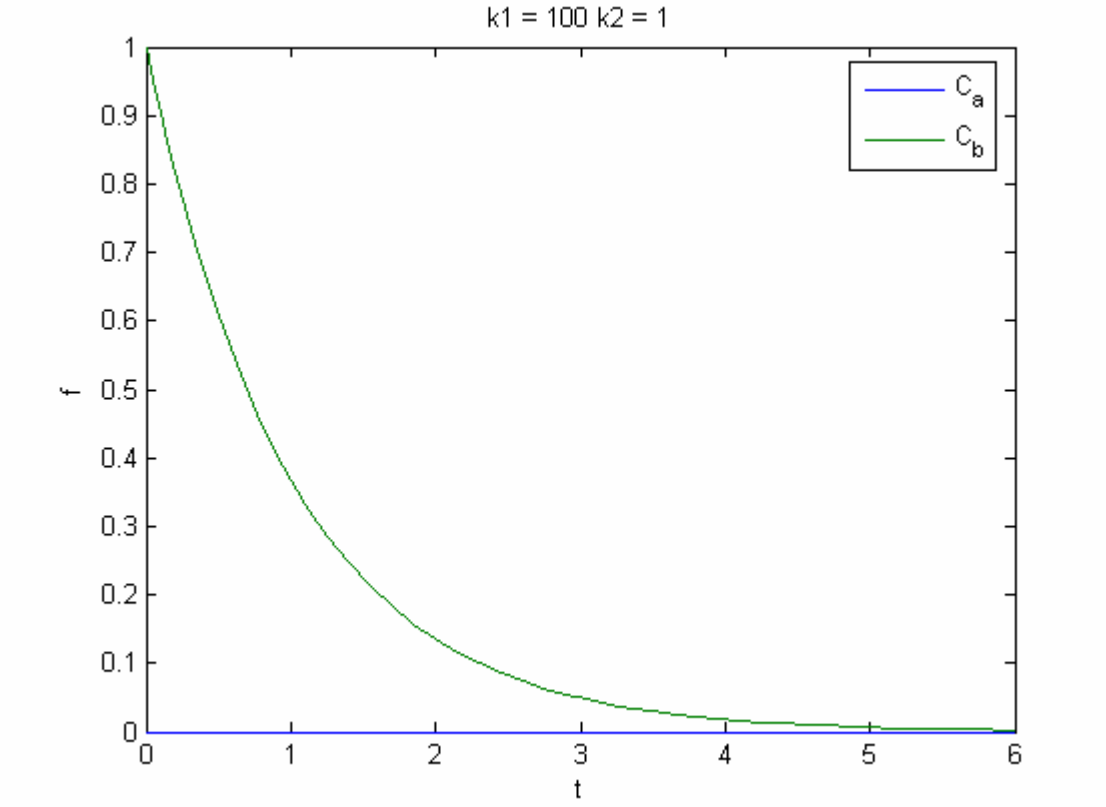
```
dfdt = derivative_function_name(t,f,param);
```

Basically, you need to pick a solver, set the function that calculates the time derivatives $\frac{d\vec{f}}{dt}$, set options (such as error tolerances), and pass additional necessary constants. Options and param may each be [] when using default error tolerances and no additional constants are needed in the derivative function. Note that only the current value of t (scalar) and the column vector of the current unknowns, *f* are passed to the derivative function.

IMPORTANT: always use column vectors for dfdt and f. This is the format Matlab expects and use of row vectors leads to errors that you will have to debug later.

Try running each set in the EXAMPLE sequence of *odesintegrate.m*. (Type "help odesintegrate" at the Matlab prompt and run each set of parameters in sequence.) This example sequence illustrates a difference between the performance of ode45 and ode15s. As k1 is ramped up with a constant k2, the [A] eventually disappears "instantaneously" relative to the rate of reaction of [B] into [C]. This essentially looks like [A0] =[C0]= 0 and [B0] = 1 case. See Figure 1. Note that the number of time steps that the ode solver requires to get an accurate result increases as k1 gets much larger than k2. Whenever a process has two or more vastly differing characteristic rates, the problem can be called "stiff".





dynamic results are very similar when viewed over the timescale of decay of [B]. Results obtained with integrator ode45.

While ode45 is a more accurate time integrator in general, and is a good overall tool, ode15s handles stiff problems much more quickly; ode15s has better stability properties and thus can take larger time steps. (This is because ode45 is explicit and ode15s is implicit. If you would like to more about the inner workings of the time integrators, feel free to ask.)

Programming Exercise 1:

Consider the following pair of coupled linear odes:

$$\begin{bmatrix} \frac{df_1}{dt} \\ \frac{df_2}{dt} \end{bmatrix} = \begin{bmatrix} -1 & 2 \\ 0 & -5 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

subject to the following initial condition:

$$\vec{f}_0 = \begin{bmatrix} 10 \\ 10 \end{bmatrix}$$

Your task is to solve this set of equations numerically in Matlab using ode45. Ensure you have the correct solution by plotting your numerical expressions for f1 and f2 along with the analytical expressions:

$$f_1(t) = 15\exp(-t) - 5\exp(-5t)$$

$$f_2(t) = 10\exp(-5t)\,.$$

Feel free to use the structure of the example file, *odesintegrate.m* as a template to build from.

Task 2: Using events to terminate ode solvers early:

A priori, you won't always know how long to run a time integration. Matlab incorporates "event function" functionality into the various ode solvers to allow for this capability. The best way to understand how this works is to see an example. Look at *learntouseevents.m*.

A set of event functions is attatched to an ode solver with the following options:

```
options = odeset('Events',@eventfunctionname);
```

Each event function in the set triggers an event when the function "value" returned is zero. The event function has the following format:

```
[value,isterminal,direction] = eventfunctionname(t,f,param);
```

The event function takes the current value of time, the unknown vector f, and any additional problem parameters and returns any number of event function values. Each of the three returned variables, value, isterminal, and direction are column vectors with one entry per event function. For example, if you wanted to have a single event function terminate integration as soon as f(1) became zero, you would write the code:

```
function [value,isterminal,direction] = eventfunctionname(t,f,param);

      value = f(1); %every zero in the value vector is an event

      isterminal = 1; %isterminal =1/0: stop/keep integrating

      direction = 0; %0/+1 or -1: direction doesn't matter/does matter

return;
```

See *learntouseevents.m* for a more sophisticated example of the use of event functions, including recovery of the log of events.

Programming exercise 2:

Using the same solver and code as exercise 1, add an event function that terminates integration when f2 = 1.

Task 3: Plot Management

Trying to put lots of things on the same graph can be a pain, so here is a summary of the figure creation process.

A plotting command replaces whatever is currently plotted on the active figure by default. A new figure can be created with the command:

```
figure;
```

or

`figure(n);` (n is a positive integer, the figure number)

In the second case, the figure will always have the number n and if figure n exists it will be replaced unless the hold is on (more on that later). Figure n becomes the "active figure" on which all plot commands will be placed.

X-Y plots are made by calling the plot command, which may take a variable number of arguments:

plot(x1,y1,formatstring1,x2,y2,formatstring2,x3,y3,formatstring3,…);

See Matlab documentation under "LineSpec" for examples of format strings (or "help plot"). If several data sets are given in a single plot call, Matlab automatically cycles through different colors. If you want to have separate plots in separate statements on the same figure, the "hold on" command must be used, and formats SHOULD be explicitly specified so you can tell which line is which. For example, try the following series of commands (copy and paste):

```
x = linspace(0,10,101); y1 = sin(x); y2 = cos(x);

figure(1); plot(x,y1); plot(x,y2);

figure(2); plot(x,y1); hold on; plot(x,y2); hold off;

figure(3); plot(x,y1,'ob'); hold on; plot(x,y2,'or'); hold off;

figure(4); plot(x,y1,x,y2);
```

In figure 1, the second plot command over-wrote the first. In figure 2, the second dataset plotted is by default the same color as the first; in figure 3, this is explicitly corrected and open circles are used and no line is used. In figure 4, the Matlab defaults automatically cycle through colors. I recommend you explicitly give figure numbers, so that you don't have to close your old figures when doing several runs in a row.

Plotting ranges can be set with the following options:

```
xlim([xmin,xmax]);
```

```
ylim([ymin,ymax]);
```

The title can be specified with:

```
title(titlestring);
```

and the legend can be specified with:

legend(series1string,series2string…);

or

legend({series1string,series2string,series3string…});

Also, a grid can be overlaid on the plot with

```
grid on;
```

This should cover most plotting needs you will have. Advanced options are available. Look up the Matlab "help plot" for details.

Programming exercise 3:

Using the results of exercise 1 or 2, plot f1 with red upward facing triangles (no line) and f2 with a blue dashed line (no symbol). Ensure the minimum x and y values on the plot are zero, and all the data are visible in the plotted range.

Task 4: Model fitting

The Matlab "fit" function is a versatile tool for fitting parameters to user-specified models. This example will show you how to fit some noisy data to a sine function.

First, you need a noisy data set f(x) (a column vector);

```
x = linspace(0,2*pi,101)';

f = sin(x+0.1*randn(size(x)))+0.1*randn(size(x));

plot(x,f);
```

Next set the model type to be fit. The call:

```
model=fittype('A*sin(B*x+C)+D','ind','x');
```

declares a 4-parameter model based on the sine function and declares x to be the independent variable. Next you should guess the parameters in the order they appear from left to right:

```
initial_guess=[1.00,1.00,0,0];
```

Note that we guessed our values as those that you should get if there were no noise in the data. Finishing up, call:

```
options=fitoptions(model);
set(options,'StartPoint',initial_guess);
fresult = fit(x,f,model,options);
```

The values of A, B, C, and D are stored in the struct fresult. The resulting parameters can be viewed as a whole by typing:

```
fresult
```

and the two plots can be compared by using the model directly as if it were a function. Try typing:

```
plot(x,f,x,fresult(x));
```

Individual parameters from the fit can be extracted with struct "dot" notation. For example, to see parameter A type

```
fresult.A
```

Programming Exercise 4:

Collect and save the f1 output you get from exercise 1. Add some noise to it, using the command:

```
f1 = f1 + (0.1*randn(size(f1)));
```

Fit a 4-parameter model to fit f1(t) = Aexp(Bt) + Cexp(Dt). Do you recover the coefficients A, B, C, and D that you expect?

Task 5: Solving Nonlinear Systems of Equations

Matlab has the ability to solve nonlinear systems of equations using fsolve, where equations are of the form:

$0 = \vec{g}(\vec{x})$.

This could be interpreted as directly finding the steady state of the equation

$$\frac{d\vec{f}}{dt} = \vec{g}(\vec{f}, t).$$

The calling sequence of *fsolve* is very similar to the use of ode integrators. The main difference is that instead of supplying a function to calculate the derivatives, you supply a function to calculate the right-hand-side of the equation, g. When g is not zero, this is also called the residual.

The calling sequence and use of fsolve is

```
x = fsolve(@gfun,x0,options,param);
```

Where x0 is the initial guess, param are additional constants needed by the g function, and gfun is the gfunction:

```
[y,Jacobian] = gfun(x,param);
```

gfun returns the value of each of the g(x) functions in y. When all the entries in the column vector y are zero, the system is solved with the current iterate of x.

The Jacobian matrix is an expression that contains the rate of change of each of the g functions with respect to each of the parameters x (as in Newton iteration). Entries are given by:

$$J_{ij} = \frac{\partial g_i}{\partial x_j}.$$

The Jacobian matrix is an optional returned value in gfun. In general, Matlab will use a numerical approximation of the analytical Jacobian. The user specifically mentions that an specified analytical Jacobian will be given with the statement:

```
options = optimset('Jacobian','on');
```

The use of fsolve is demonstrated in the example file *multiplefsolve.m*. Given a perimeter and area, this function finds the length and width of an appropriate rectangle if one exists:

$$g_1(L,W) = 2L + 2W - P$$

and

$$g_2(L,W) = LW - A,$$

where L = length, W = width, P = perimeter, and A = area. Notice that the g functions must be rearranged so that they are zero when satisfied by appropriate L and W.

Programming exercise 5:

Following the example of multiplefsolve.m, make a program that finds the height and radius of a cylinder with a given area and volume. Multiple solutions may exist! Verify that your program gives you R = 1 and H = 1 when Area = 4*pi and Volume = pi with an appropriate initial guess.

Good programming practices:

Use meaningful names for variables. If you have a list of 10 different chemical species stored in x(1), x(2)…x(10), you shouldn't be using them in that form in your reaction equations. Consider renaming them as soon as you enter the function or declare descriptive names equal to your integer indices, e.g.

```
Cl = 1;

Na = 2;

NaCl = 3;
```

Then you can say:

```
x(Cl)*x(Na);
```

which is much easier to understand and debug than:

```
x(1) * x(2);
```

Go for *simplicity* and *clarity* before going for *elegance*. Colon notation is an effective way to make Matlab programs more compact, but for loops are much more intuitive. Use for loops if colons give you trouble. Furthermore, in general, several simple steps are easier to understand than one long, complicated step.

Watch out for the differences between matrix and element-wise operations. This leads to common mistakes. Most functions, such as sin() and exp() can accept a matrix of any size rather than a single value. They operate on each matrix element individually, returning the sin() or exp() of each element, respectively. However, whenever you use the operations (*,/) and the objects being multiplied/divided are vectors or matrices, be sure to know the difference between (*,/) and (.*,./). The first uses matrix multiplication and the second does element-wise operations, operating on corresponding entries. If this gives trouble, feel free to use for loops to operate on one value at a time.

Learn to use the Matlab debugger. Programs usually fail on the first try. Matlab has a useful debugger, where you can put breakpoints in the code, examine variable values during execution, and even execute any additional operations from the command line (such as make plots). As an alternative to placing a breakpoint, the "keyboard;" command will do the same thing. To exit debug mode at the command line, type "return;". On the command line, "who" is a useful command that displays the names of all the variables that are known at this point in the code. Each one can be inspected or operated upon.

Use semicolons. If semicolons are omitted at the end of a line, more output is sent to the screen. In the case of large matrices, it can be a startlingly large amount.

Use functions rather than scripts. A function takes input and gives output; this makes it easy to build a code as a group of working parts with well-defined interfaces. If a function can't change its input to mess up the calling program. A script works at the same level as the regular Matlab prompt. In a script, all variables are effectively global, and thus from run to run things might change depending on what is stored in all current values of the variables. Also, if you run two different scripts, they could use the same variable names and operations and results can get mixed up.