This tutorial is parallel to the IDL tutorial developed by Prof. Carl Heiles at UCB.

1 THE BASICS

MATLAB provides a powerful vectorized high level language with syntax very similar to C, specially useful for data analysis and display. MATLAB originated as an extension of the LINPACK and EISPACK libraries of routines for linear algebra done in the late 80s by Cleve Moler, and has evolved into a (proprietary) language that incorporates several object-oriented features as well as powerful graphics. While it has a very large user base in Engineering and Physics, its Astronomy user base is small and as a result does not have the wealth of legacy software ported that IDL has. So, on the one hand, be prepared to program. On the other hand, it is substantially more modern, consistent, cohesive, and overall less clunky (I find) than the alternative.

1.1 Basic Commands

All quantities default to double precision numbers, unless the variable that contains them is declared to be of a different type before hand. So to MATLAB 3, 3.0, and 3e0, are all identical double precision versions of the number 3. This feature is very handy since it allows one to ignore typing in most situations and just get down to business, with the assurance that the calculations will be carried out with the best precision available. On the other hand, double precision numbers take 8 bytes to store, so it is more memory costly. See help datatypes to get information on the different data types available in MATLAB as well as the functions that create variables of a given type and convert between different types.

If unassigned, the result of a MATLAB operation is stored in the variable ans (short for answer, it used to be the default in programable calculators). Unless the input line is terminated by a semicolon (;) the result will be printed in the screen. So 3*5 will produce a reply ans = 15, while 3*5; will produce no reply but still store the result of the operation in the variable
ans. To see the contents of the variable, just enter **ans**.

**A=3*5** will create the double-precision variable **A** and set it equal to 15. To see its contents, just enter **A**.

MATLAB is case-sensitive. Routines (or commands) with capitals or mixed case are different from the lower case version. The same happens to variable names.

bf who or **whos** will tell you which variables are in memory, and give you information about them. For just one variable specify its name (e.g., **whos**(′A′)).

**a=5; a=sqrt(a)** will redefine **a** to be the square root of the number 5. The semicolon allows you to separate statements within a line (just like in C), as well as supressing output. Note that by default MATLAB gives you four significant digits in the output. The calculations are carried out at much higher precision, of course. You can change the output format using the **format** command. The default output is equivalent to **format short**. To see more digits, specify **format long**. Other format examples are **format rat**, **format short eng**, and **format hex**.

**a=′Joe′** defines **a** as a string variable.

**a=[1,2,3,4,5,6]** or **a=[1 2 3 4 5 6]** creates a six-element array containing the values 1 through 6. MATLAB uses commas or spaces to separate columns in a matrix (our array is technically a 1 × 6 size matrix. To separate rows MATLAB uses the semicolon — more on this later). A shorter way of generating the same array is to use the colon (:) operator: **a=1:6**. This generates an array of numbers starting with 1, incrementing by one until reaching the number 6. A more general construction involves two colons: **start:inc:end**. So **b=1:2:6** is equivalent to **b=[1 3 5]**.

MATLAB is a vectorized language. That means it operates automatically over each member of an array without the need for an explicit loop (which would be necessary in C or FORTRAN). In fact, it is not only more compact, but more efficient and faster to avoid loops if possible.

**b=sqrt(a)** will use the square root operator over each element of the array **a**.

**c=a.^0.5** will apply the “raise to the 0.5 power” operation to each member of the array **a**. This should produce the same result as taking the square root.

MATLAB uses the dot-operator construction here. Dot-operators are meant to operate on the members of the array, while for MATLAB using the ^ operator not preceeded by a dot means to do the proper matrix operation
of raising to a power (this would fail in our case, since the matrix “raise to a power” only works for arrays with the same number of rows and columns). Other common dot operators are “.*” to multiply and “./” to divide. Note that addition and subtraction are identical for arrays and numbers, so they do not need a dot-operator.

\[ \text{max}(b-c), \text{min}(b-c) \] prints the maximum and minimum values of the difference array.

\[ b==c \] will produce an array of the same dimensions as b and c, containing a 1 for each element that is equal, and a 0 for each element that is not equal in both arrays. If the arrays have different dimensions, it will produce an error. To ask whether all the elements in a vector are equal, we use the \( \text{all}(\) function: \( \text{all}(b==c) \). Similarly, we could use the \( \text{any}(\) function to ask if there are any elements that are equal in both arrays.

\[ \text{sum}(b) \] calculates the sum of the elements of vector \( b \).

\[ a=\text{zeros}(1,100) \] defines \( a \) as an array of zeros of 1 row by 100 columns. Similarly, we could define an array of ones using \( a=\text{ones}(1,100) \). Note that these function can also generate 2D arrays (or in fact nD arrays).

\[ a=[1:100] \] or \( a=1:100 \) generates an array of 100 numbers from 1 to 100.

\[ a(1), a(100) \] prints the first and last elements of the array \( a \). More generally, we can refer to the last element of any array as \( a(\text{end}) \). MATLAB departs from C in the convention for the indexing of arrays: indices start from 1 (just like FORTRAN or BASIC) and not from zero.

\[ a(10:20) \] will print the array elements 10 to 20. For large arrays, MATLAB will automatically indicate the column range in each line.

\[ b=\text{sin}(a/5) ./ \exp(a/50) \] defines an array \( b \) in which each element is related to the corresponding element in \( a \) by the mathematical expression. Recall that all operations are carried out in double precision, and numbers are always assumed to be floating point.

\[ \text{plot}(b) \] makes a plot of \( b \) versus its index number, connecting the elements of the vector with lines.

\[ \text{plot}(a,b) \] plots \( a \) in the horizontal axis versus \( b \) in the vertical axis. You can annotate the plot using \( \text{xlabel()} \) and \( \text{ylabel()} \), as in \( \text{xlabel(‘A (nepers)’)}. \)

1.2 Matrix operations

As discussed above, MATLAB provides all the matrix operations that one may want to use. In fact, the standard will be to interpret operators as corresponding to the matrix operation unless they are preceded by a “dot”
whenever it is necessary to distinguish between the matrix version and the scalar vectorized version of an operation.

2 THINGS THAT YOU REALLY WANT TO KNOW

2.1 Fundamental mathematical constants

MATLAB does not come with many predefined mathematical constants. The exception is the number \( \pi \), which is simply \texttt{pi}. The base of the natural logarithms, \( e \), is simply obtained doing \texttt{exp(1)}. Similarly, converting from degrees to radians requires defining a conversion factor, for example \texttt{cf=pi/180}.

2.2 The hypertext HELP facility

The basic online documentation in MATLAB is provided by the command \texttt{help}, which will produce hypertext-linked information on the command tree and/or individual commands (e.g., \texttt{help exp}).

A handy feature of MATLAB is that it is effortless to add help information to your own programs. In any file containing a user-defined function, MATLAB will interpret the block of comments after the function declaration as the help corresponding to that function. More on this later.

MATLAB has a full-blown hypertext help facility that is launched using the command \texttt{helpdesk}. To look for help on the random number generator, for example, click on the “Search Results” tab on the upper-left side of the helpdesk and type “random” in the search box. Help for many useful random number functions will become available. Look for \texttt{randn} in the list, and click on it.

In the description you will see that \texttt{randn} essentially takes one parameter, the size of the array that you want to generate. Random number generators use a number called “seed” to start the pseudo random sequence. MATLAB always initializes the seed to the same number when it starts. To generate a different random sequence, we need to change the seed. A handy blind way of doing so is to use the computer clock. Enter \texttt{randn('state',sum(100*clock))}; to randomize the generator.
MATLAB uses different functions (and different generators) to generate the different distributions. The function `randn` will generate normally-distributed random numbers. The function `rand` will generate uniformly-distributed random numbers.

```
output=randn(1,230);
```
will generate 230 normally-distributed random numbers, and put them in the `output` array. To plot them, try `plot(output)`. To see them plotted with individual symbols not joined by lines, try `plot(output,'x')`.

### 2.3 Command-line editing

Using the command line requires typing, and typing introduces errors. To a very good approximation the MATLAB command-line interface implements the `emacs` commands that are also available in a UNIX terminal. Here are the most useful:

- **backspace** deletes the character behind the cursor
- **delete** deletes the character under the cursor
- **left and right arrow keys** move the cursor on the current line
- **Ctrl-E or End key** move the cursor to the end of the line
- **Ctrl-A or Home key** move the cursor to the beginning of the line
- **Ctrl-K** erase the line from the position of the cursor to the end, and store in the “paste” buffer
- **Ctrl-Y** paste the “paste” buffer

There are three very handy features to the command-line interface:

- **up and down arrow keys** allow you to move through the command history, so you can re-enter and/or edit old commands.
- **pattern matching search through history** if you start typing the first few characters of a line you want to recall and then press the **up arrow** key, MATLAB will complete it with the previous issued command that started with those characters. If you keep pressing the up arrow, it will cycle through the command history presenting you all the lines that match your initial characters, if any. Try typing `a=` and then pressing **up arrow**.
- **line completion** if you start typing a line, and press the **tab** key, the interpreter will present several options available for completion, according to files in the MATLAB path (more on this later).

In the MATLAB graphical interface there is also a command history and a workspace tab available, where one can simply select and click to repeat commands and obtain information about variables in memory. Since
I became a MATLAB user well before these features were available I tend to not use them and just prefer typing (a sign of my age).

2.4 Batch files

To avoid all that command-line typing when one is repeating series of commands, one creates a program. The simplest version of a program is just a batch file. To do so, create a text file (a.k.a. an ASCII file) using an external editor (such as vi or emacs), or even better the handy MATLAB editor. To start it, simply enter `edit` at the prompt.

Now we can create a program with the following command lines:

```matlab
original=sin(([1:200]/35).^2.5);
original=original+2;
time=3*[1:200];
plot(time,original);
axis([0 600 0.5 3.5]); xlabel('Time'); ylabel('Amplitude');
```

It is rare to need a continuation character, but if necessary MATLAB uses the ellipsis (...) as an indication that a line will continue into the next:

```matlab
a=[1 2 3 4 ...
 6 7 8 9]
```

will be interpreted as if it was all in one line.

Now, to conclude the program, we need to save the file. MATLAB uses the extension `.m` to identify its files. Click on the “File Save” tab or the floppy disk icon in the editor, and save your program with the name “test.m”. To invoke and execute the program, just enter `test` at the MATLAB prompt.

Documentation is good, and programs should be documented as they are written (or else you will find yourself scratching your head and wondering what this code was for in a couple of weeks). The comment symbol is the `%` sign. The remainder of the line after a `%` sign will be ignored by the interpreter. Try including those in the editor, and see how the color of the text changes to green. Use this comment as the first line of the file: `% This file corresponds to the MATLAB tutorial`. Now save the file and try `help test`. Handy, isn’t it?

MATLAB also provides symbols for blocks of comments. Any lines between a `%{` and a `%}` will be ignored by the interpreter. These need to be the first characters in a line.
2.5 I GOOFED!

Just like in IDL or the UNIX shell, you can stop the execution of a program by pressing Ctrl-C. This will place you back in the environment from which the program was launched and give you control.

 Sometimes, when one is writing a complex program with many nested functions, it is very handy to have some debugging functions available to figure out why something isn’t working as it should. MATLAB provides several levels of debugging possibilities:

*Printing values.* The simplest debugging is simply looking at how a program changes a value to figure out what is going wrong. Since MATLAB prints out the values assigned to variables unless a semicolon is used to end the line, then the easiest thing to do is to remove the semicolons from strategic statements to see the output information. More elaborate, formatted output can be produced sprinkling disp or fprintf commands throughout your program. fprintf(1,...) will write to the standard output (i.e., the screen) and has the same syntax as the C command of the same name. Check out the help information on these two commands.

*Giving the control back to the user.* Sometimes I want more than just seeing a value. I want to look at several values (or arrays) and do calculations with them to figure out whether something is correct or not. The program flow will be interrupted and the control given back to the user when the interpreter finds the instruction keyboard. Try inserting it in your test program. To continue with the program flow, just say return in the command line.

*Full blown debugger.* MATLAB includes a very handy debugger: say help debug to get all the relevant information. A very useful pair of instructions are dbstop on error and dbclear all. The first one tells MATLAB that, if and when an error occurs, the control should be instantly given back to the terminal. It’s just like inserting a keyboard statement the moment an error happens (although you cannot simply return to continue the execution). The editor will automatically load the offending file and a green arrow will indicate the position of the offending statement. You can also insert and remove break points using the dbstop command or the handy buttons on the editor window, step instruction by instruction, etc. The dbclear all statement will simply clear all break points, and stop the debugger.
3 MAking Plots

3.1 Specifying data ranges, titles, etc

MATLAB is superb at producing publication-quality graphics in many formats. We showed in §1.1 how to produce and annotate a basic plot, which you can recreate at will just running the test program we wrote above. There other types of plots that can be produced, just look at the MATLAB help for graph2d.

The properties of graphics are manipulated in MATLAB using object handles. MATLAB distinguishes between figures and axes. Each figure has a separate window, with the figure number as a title. If you want to create a new window, issue the command figure. If you want to change the focus back to the older figure, say figure(1). Do you want to look at the properties of Figure 1? Issue the command get(1) to get a (somewhat overwhelming) list of figure properties. The documentation for all of them can be found in the helpdesk. To change the size of the figure, for example, one needs to alter the last two elements of the 4-element Position array (the first two are the x-y position in the screen in pixels, the second two the x-y sizes in pixels). To alter the size of the current figure, try the following commands:

\[
s = \text{get(gcf,'Position')};
\text{set(gcf,'Position',[s(1:2) 400 400])};
\]

The first command puts into the variable s the value of the property Position for the current figure (gcf stands for “get current figure”, and it contains the value of the handle of the active figure, in our case just 1... verify that by issuing the command gcf). The second command changes the value of the property Position in the current figure to an array of 4 numbers. The first two are the first two elements of s (s(1:2)). The last two are the new size of the window, 400 × 400 pixels. The graphics are resized automatically.

A figure can have many axes. The plot produced by the test program is one of them. If you have it on the screen and the active figure is the one that contains it, you can see its properties issuing a get(gca). As with gcf, gca is a command that returns the handle of the current axes object. Unlike figures, which have integer handles, axes have floating point handles that are not displayed anywhere. So gca is really handy.

The default background color of a plot is white. To change it, try issuing a set(gca,'Color','y'). Basic colors in MATLAB are abbreviated with one letter (see the help for plot for color and symbol abbreviations). To specify
an arbitrary color, give a 3-element array with the RGB component values in the range 0 to 1, for example, \texttt{set(gca,'Color',[0.2 0.8 0.6])}. To change the line width of the box framing the plot, try \texttt{set(gca,'Linewidth',3)}.

Similar handles are available for most of the plotting commands. Change the test program to return a handle in the command \texttt{plot}: \texttt{h=plot(time,original)}. Then play with the properties.

By the way, the properties are case insensitive and minimum-pattern-matched. So \texttt{set(gca,'Linewidth',3)} and \texttt{set(gca,'linew',3)} are the same command.

To clean the figure, issue the clear-figure command \texttt{clf}. To close the window, issue \texttt{close(1)}. The next plot command will create a window, if necessary.

### 3.2 Overplotting

Often you want to plot two graphs in the same plot — comparing data and theory is an example. Usually new plots erase the old plots. To change that behavior, issue \texttt{hold on}. To return to the old behavior, issue \texttt{hold off}. Just issuing \texttt{hold} toggles the behavior.

So, to illustrate this, try the following commands:

\begin{verbatim}
hold on
original_2=2+sin(((1:200)/35).^2);
plot(time,original_2,'--');
\end{verbatim}

this will overplot the new graph on the older graph using a dashed line. To plot it using symbols, try \texttt{plot(time,original_2,'s')}.

A very handy command to annotate plots of this type is \texttt{legend}. Look at the help for that command. Here, we can create a simple legend by issuing the command \texttt{legend('original','original_2')}.

### 3.3 Making a hard copy on paper

This is usually done by creating and printing a Postscript file. MATLAB has the handy \texttt{print} command, which takes care of this as well as output in other formats. Usually, the most common format for a graph that will be used as a figure in a document is encapsulated postscript. To create an encapsulated postscript file out of the current figure, try \texttt{print -depsc2 mytest.eps}. This creates an encapsulated level-2 color postscript file containing the plot. Look at the help for the \texttt{print} command for other options.
MATLAB automatically resizes the figure to fit properly within the letter paper bounds. Sometimes this is undesirable, because it may shift around some of the annotations, resize the axis labels, etc. To obtain a paper copy that looks identical to the screen, one has to set a particularly obscure property of the figure before printing: \texttt{set(gcf,'PaperPositionMode','auto')}.

4 STARTUP FILE

Although MATLAB has a number of automatically executed startup files that can be edited to reset the configuration (e.g., \texttt{help startup}), this is hardly necessary. To see the current search path, type \texttt{matlabpath}. To add a directory to the search path use \texttt{addpath} once. MATLAB will remember.

My personal MATLAB code, which may be handy for the lab, is in “/n/fornax1/bolatto/matlab”. Feel free to add it to the path if it isn’t already there.

5 MATLAB SAVE FILES

You can save all the variables on memory (or just a few of them) and restore them later using the \texttt{save} and \texttt{load} commands. The first one will create a \texttt{matlab.mat} file in the current directory, if no parameters are issued. The second one will read it. They can do much more than that. Check their help.

6 READING AND WRITING FORMATTED DATA ONTO DISK

You can read in text files produced with an editor, and write similar files too. Here is how.

6.1 Writing a file

There is more than one way to save the arrays we created above onto a text file. The simplest (and least flexible) is to use an option of the command \texttt{save}:

\begin{verbatim}
  m=[original;original_2]’;
\end{verbatim}
save testdata.txt -ascii m

The first statement puts both arrays in a matrix with a desirable format. The semicolon tells MATLAB that each array is to be a row of the matrix \( m \) (of course, this only works because they have the same length). The ‘ operator transposes that matrix, so that rows become columns and vice versa. An equivalent statement would be \( m = [\text{original}', \text{original}_2'] \); if you prefer to think that way. Try looking at the variable \( m \) to see the resulting format. The second line tells MATLAB to save the contents of the variable \( m \) in text format in the file \( 	ext{testdata.txt} \).

The second option is to use the C-like mechanism of \texttt{fprintf}. This allows for arbitrarily output formats, so it is infinitely flexible. If you know the C programming language, the following lines will look awfully familiar:

\begin{verbatim}
fp=fopen('testdata.txt','w');
fprintf(fp,'%f %f \n',m');
fclose(fp)
\end{verbatim}

Here the first line opens the file \( 	ext{testdata.txt} \) for writing (the old contents are erased). The second instruction is a vectorized version of the C \texttt{fprintf} statement: the format string is applied to all values in the matrix \( m \) starting by column order (i.e., the row index is the fast-changing index. This is the MATLAB convention.) That is why \( m \) has to be transposed first. The last statement closes the file, after we finished writing. You can try it in the screen issuing the same command with \texttt{fp} set to 1: \texttt{fprintf(1,'%f %f \n',m')}.

One does not need to use the vectorized features of \texttt{fprintf}. A more clunky way of achieving the same goals is to write an explicit \texttt{for} loop. Replace the middle statement above with the following:

\begin{verbatim}
for i=1:length(original), fprintf(fp,'%f %f \n',original(i),original_2(i));
end
\end{verbatim}

Here the function \texttt{length()} returns the longest dimension of the array (the other dimension in our case is 1). The syntax of the \texttt{for} loop is straightforward.

\section*{6.2 Formatting print statements}

\texttt{fprintf} can be used to format the output in almost any conceivable manner. Please see the MATLAB help on this routine for a detailed explanation.
6.3 Reading a file in column format

The simplest way of reading a file in column format, such as the one we have just written, is to use the load command. This command will automatically recognize the file as a text file, read the columns and rows, and return a matrix.

\[ m_2 = \text{load('testdata.txt');} \]

Please verify that the file was correctly read.

To parse more complex file formats MATLAB provides the commands fscanf and textscan. Please look at their help.

7 OPERATORS

7.1 Relational operators

We introduced a number of operators in the previous sections, such as the colon operator and the transpose operator. MATLAB provides the traditional relational operators. == and ~= are used to represent “equal” and “nonequal”. See help relop for a description of the relational operators.

\[ c = (a < b) \]

will produce an array of ones and zeros, 1 wherever the comparison is true, and 0 wherever it is false. This is very different from the result of the same instruction in IDL.

7.2 Other operators and useful functions

To request the full list of operators implemented in MATLAB, try help ops. There are also a number of is functions that identify particular conditions. This is very similar to the approach taken in the language C. For example, in the IEEE convention of floating point numerical representation used in all modern computers there are particular codes to identify NaNs (not-a-number) or infinity. MATLAB provides functions that test for these. Some useful examples are:

- \( \text{isnan(a)} \) Returns 1 for every NaN in array \( a \).
- \( \text{isinf(a)} \) Returns 1 for every infinite in the input.
- \( \text{isfinite(a)} \) Returns 1 for every finite number in the input.
- \( \text{isreal(a)} \) Returns 1 for every non-complex number in the input.

Try \( a = 1/0 \) and \( \text{isinf(a)} \), for example.
7.3 The all-important FIND

Suppose you have an array \( a \) and you want to identify the indices of that array for which the elements fulfill some condition (strictly speaking, it returns the indices of all elements that are non-zero. Remember that all the comparisons return zero when they are false). For example, \( \text{ix} = \text{find}(a > 10) \) will return the indices of every element of \( a \) that is larger than 10. This is extremely handy. Now, we can set them all to 10, for example, by doing \( a(\text{ix}) = 10 \); No for loops necessary!

8 FOR LOOPS: USING THEM AND AVOIDING THEM

To repeat the message: for and while loops are very flexible but very slow. They also complicate the code, and make it less compact. They are to be avoided whenever it is possible to use the vectorized features of MATLAB. Loops are also much much much slower than the vectorized implicit versions. Try the following comparison of the vectorized execution time

\[
\begin{align*}
\text{u} &= \text{rand}(1,1e6); \\
\text{tic}; \text{ix} &= \text{find}(\text{u} > 0.5); \text{u}(\text{ix}) = 0; \text{toc}
\end{align*}
\]

versus the non vectorized one

\[
\begin{align*}
\text{u} &= \text{rand}(1,1e6); \\
\text{tic}; \text{for } i = 1:\text{length} (\text{u}), \text{if } (\text{u}(i) > 0.5), \text{u}(i) = 0; \text{ end}; \text{ end}; \text{toc}
\end{align*}
\]

In my machine, the vectorized version is about 23 times faster than the one with the loop and comparison.

It is true that one cannot live without loops, but minimizing their use leads to much more compact, cleaner, and faster code.

9 PROCEDURES, FUNCTIONS, AND MAIN PROGRAMS

9.1 Procedures and functions

MATLAB does not have the distinction that IDL has between procedures and functions. The main distinction between batch files (such as the one we wrote during this tutorial) and functions is the fact that batch files run in the
user environment, while functions run in their own environment. What do I mean by environment? The contents of the variables defined. So, all batch files share the same memory space, thus altering a variable in one has effects on how other execute. So variables can be used to pass information between batch files. By contrast, every time we execute a function its environment is created anew, and the only variables defined are the parameters passed to the function. So functions are fairly safe and separated boxes, while batch files share all the same sandbox. Often it is easier to debug a batch file, since the internal variables remain in the user environment and are not deleted after it finishes. Another (minor) distinction between batch files and functions is that functions can return one (or more) variables as results.

To define a function in MATLAB, the first instruction in the file has to be something like

\[
\text{function } y = \text{myfunction}(x)\]

where \( x \) is a parameter passed to the function, and \( y \) is a variable returned by the function (and hopefully assigned some value within the function). MATLAB functions do not need to return a value.

9.2 DOCUMENTING your procedures

We already highlighted the importance of documenting the code with comments. We also mentioned that the first few lines of comments in a batch file will be returned by the `help` command when exercised on that file. Similarly, for functions the first block of lines that start with the comment symbol `%` after the function declaration are the help for that function. So it is painless to add help information to your own functions, and it should be done every time.

10 ROW- vs. COLUMN-MAJOR IN MATLAB

MATLAB matrices follow the standard mathematical convention, which is also used in C. They are column major. So the indices of 2-dimensional arrays follow the usual convention of row first, column second.

MATLAB has also a very strong implementation of sparse matrices, if you ever encounter that problem. Sparse matrices are matrices where most element are zero. If the matrix is very large (millions of elements), it really
pays to store and track the location of only those element that are nonzero. MATLAB implements this seamlessly.