Professor Alberto Bolatto
Observational Astronomy
ASTR 310
Fall 2009

Introduction to MATLAB

1 Introduction

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 and was developed in the late 80s by Cleve Moler. It then evolved into a (proprietary) language that incorporates several object-oriented features as well as powerful graphics (see `http://www.mathworks.com` for more details). 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. Therefore, 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.

Vectorized languages like MATLAB and IDL allow operations over arrays of numbers using very simple syntax, essentially the same syntax one would use to operate over scalars. MATLAB has a very flexible implementation of n-dimensional arrays: 3 and 4 dimensional data is very common in Astronomy, where the 2 spatial dimensions are generally combined with a 3rd spectral dimension and maybe a 4th polarization dimension. The display capabilities are extremely good: graphics are automatically resized and the user has complete control over a large number of properties, including extensive annotation. The implementation of 3D graphics is also very good. Finally, MATLAB can be easily interfaced with C or FORTRAN routines to increase its execution speed, if need be.

2 Starting MATLAB

As for IDL, the computer should be running Linux. Once you have logged into your account and started the X-windows system using the `startx` command, open a terminal. The latest version of MATLAB in the system is v7.8, and can be started by typing `matlab2009` at the prompt. This will open the MATLAB console.

The main part of the MATLAB console is the Command Window. There you will see the prompt `>>`. Usually to the left of the Command Window there are a number of auxiliary windows that are configurable by the user and contain the Command History, information about variables in the Workspace, etc. Just over the Command Window there are several buttons and pull down menues. One of the most useful is the little window that is used to select or indicate the current directory. That is the default directory where MATLAB will look for or create files. The same information can be obtained by entering `pwd` at the MATLAB prompt.

In fact, `pwd` is a MATLAB function. It actually produces a string variable as the output, containing the current work directory. Most commands produce an output in MATLAB. If unassigned, the

result of a MATLAB operation is stored in the MATLAB 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 entering `3*5` will produce a reply `ans = 15`, while entering `3*5;` will produce no reply but still store the result of the operation in the variable `ans`. To see the contents of a variable, just enter its name (no `print` command necessary... in fact `print` is used to obtain hardcopies, more on that later). For example, to see the output of the last unassigned operation, enter `ans`. If you request to see the contents of a variable that is undefined, MATLAB will return an error saying just that. If the variable is empty, MATLAB will return the empty matrix symbol `[]` if it is numerical, or the empty string symbol `''` if it is a string. Those same symbols can be used to define empty variables of the appropriate type.

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

My personal MATLAB code, which may be handy for the lab, can be found in

`/n/fornax1/bolatto/matlab`

Add that directory to the path if it isn't already there using the `addpath` command

```
addpath /n/fornax1/bolatto/matlab
```

If you are taking ASTR310, there is instead a local directory that you can use. In the Fall 2009 semester, that directory is

```
addpath /n/ursa/A310/abolatto/matlab
```

3 Obtaining Help

The basic online documentation in MATLAB is provided by the command `help`, which will produce hypertext-linked information on the command tree and/or individual commands (e.g., `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 `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 `randn` in the list, and click on it.

In the description you will see that `randn` essentially takes one parameter, the size of the array that you want to generate. Random number generators use a number called "seed" to prime the pseudo random sequence. MATLAB keeps different seeds for different random number generators, and always initializes them 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, which is provided by the function `clock()` (try `help clock` for an explanation of the format). Enter

```
randn('state',sum(100*clock));
```

at the prompt 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. Try the instruction

```
output=randn(1,230);
```

It will generate 230 normally-distributed random numbers, and put them into the `output` array. Just as an example of visualization, try plotting them using

```
plot(output)
```

A window, called "Figure 1", should appear with the plot at a number of controls in the top bar. By default, `plot()` uses blue lines on a white background to join the values in the array, and plots against the index of the array. To see them plotted with individual symbols not joined by lines, try

```
plot(output,'x')
```

4 The Very Basics

In MATLAB, all numerical quantities default to double precision numbers unless the variable that contains them is declared to be of a different type. 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 variable typing in most situations and permits to 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 intensive. When working with very large arrays, it may be useful to carefully consider how to store the data (this will not be a limitation for us). 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 to convert between different types.

Let us start doing simple calculations in MATLAB. Try issuing the command

```
A=3*5
```

This will create the double-precision variable A and set it equal to 15. To see its contents, just enter `A`. Try now entering

```
a=2*4
a==A
```

That will compare the contents of `a` and `A`, and produce a 1 or a 0 if they are equal or not.

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. To obtain information about the variables in memory you can look at the Workspace window in the MATLAB console, or use the commands `who` or `whos`. They 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')`).

Try entering `a=5; a=sqrt(a)`. This will define `a` to contain the number 5, then redefine it to contain its square root. The semicolon allows you to separate statements within a line (just like in C), as well as supressing output. If you want to separate statements in a line without suppressing the output, use the comma (`,`) operator.

Note that by default MATLAB gives you four significant digits in the output. The calculations are carried out and kept internally 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 (more exotic) format examples are `format rat`, `format short eng`, and `format hex`.

The syntax for string variables is very straightforward. For example, `a='Joe'` redefines `a` as a string variable, which is really, deep down, an array of numerical character codes.

4.1 Vectors, Matrices, and Arrays

Now onto vectors and matrices. The instructions

```
a=[1,2,3,4,5,6]
```

or

```
a=[1 2 3 4 5 6]
```

create a six-element array containing the values 1 through 6. MATLAB uses commas or spaces to separate columns in a matrix (our vector is technically a 1×6 size matrix). To separate rows MATLAB uses the semicolon

```
b=[1 2 3;4 5 6]
```

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, with the syntax *start:increment:end*. So `c=1:2:6` is equivalent to `c=[1 3 5]`.

Although MATLAB will dynamically change the size of an array as new elements are assigned, it is considerably faster, less accident-prone, and much more convenient to predefine large arrays. The instruction

```
a=zeros(1,100)
```

defines `a` as an array of zeros of 1 row by 100 columns. It also accepts the alternate syntax

```
c=[1,100]; a=zeros(c)
```

Similarly, we could define an array of ones using `a=ones(1,100)`. Note that these function can also generate 2D arrays (or in fact nD arrays).

It is frequently useful to generate arrays of running indices. The instruction

```
a=[1:10]
```

or `a=1:10` generates an array of 10 numbers from 1 to 10.

Many times it is important to find out the dimensions of an array. MATLAB uses the function `size()`, which returns a vector of dimensions. Sometimes we are not interested in all the dimensions, but just the longest one (as with $1 \times N$ arrays): MATLAB provides the handy `length()` to obtain that number.

4.2 Addressing Arrays: Indices and Subscripts

It is straightforward to access individual elements of a vector. Try `a(1),a(10)` to print the first and last elements of the array `a`. More generally, we can refer to the last element of any array as `a(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. If you want to see a range of values, `a(3:7)` will print the array elements 3 to 7. For large arrays, MATLAB will automatically indicate the column range in each line.

There are two mechanisms for accessing data that has more than one dimension: subscripts and indices. Subscripts are the intuitive mathematical representation. In this picture, `b(1,2)` refers to the element in the first row, second column of array `b`. With `b` as defined in the first paragraph of the previous section, the result of that operation should be 2. This can be easily generalized to an arbitrary number of dimensions, although it can become cumbersome.

Indices, on the other hand, use just one number to address (or index) any element of an array. It actually corresponds to the physical order in which elements are stored in the computer memory. In MATLAB matrices are stored running through the rows first, then the columns, thus the fastest running subscript of a matrix is the first subscript, the row subscript. So with `b` defined above, `b(2)` will result in the number 4 (the same as `b(2,1)`), and `b(3)` is equivalent to `b(1,2)` and is the number 2.

Since keeping all of this math straight in one's head can become rapidly challenging, MATLAB has two handy functions to switch between these two different ways of addressing arrays: `sub2ind()` and `ind2sub()`. Frequently, the most computationally expedient way of addressing an array is using indices.

In MATLAB the colon operator, when used to address the contents of a matrix, stands for "everything". So one can obtain all the numbers in the first row of matrix `b` by typing `b(1,:)`. Similarly, all numbers in the 3rd column are obtained by issuing `b(:,3)`, and just plain all numbers corresponds to `b(:)`. As with the example above on the array `a`, we can also specify a range of indices or subscripts: `b(3:6)` or `b(:,2:3)` are all valid and (almost) equivalent. Note that the dimensions of the result are different.

4.3 Vectorized Operations

As we mentioned in the introduction, 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.

Try `b=sqrt(a)`. This will use the square root operator over each element of the array `a`. Similarly, `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. Verify that by issuing `b==c`. As you can see, the equality operator `==` is also vectorized.

MATLAB uses the dot-operator construction to distinguish between scalar-vectorized operations and matrix operations. Dot-operators are meant to repeat operations 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"

is only mathematically defined 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 matrices, so they do not need a dot-operator.

Most (if not all) MATLAB functions are vectorized. For example,

```
max(b-c),min(b-c)
```

prints the maximum and minimum values of the difference array. The usual mathematical functions are also vectorized. Try `log(a)` for the natural logarithm, `sin(a)` for the trigonometric sine, and `log10(a)` for the decimal logarithm. The instruction

```
d=sin(a/5)./exp(a/50)
```

for example, defines an array `d` in which each element is related to the corresponding element in `a` by the mathematical expresion. Recall that all operations are carried out in double precision, and numbers are always assumed to be floating point.

As you saw, the comparison `b==c` produces 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, MATLAB uses the `all()` function: `all(b==c)`. Similarly, we could use the `any()` function to ask if there are any elements that are equal in both arrays.

Onto more examples of vectorization, `sum(b)` calculates the sum of the elements of vector `b`. There is also an analogous `prod` function for the product, and the cumulative sum and product can be calculated using `cumsum(b)` and `cumprod(b)`. More relevant to us, there is a `median()` function. In arrays of two or more dimensions, by default all of these operators do the calculations along the first non-singleton dimension. So if `c` is a matrix, `sum(c)` will produce a vector of the same length as the number of columns in the matrix. Try

```
d=[1 2 3; 4 5 6]
sum(d)
```

If this is not the dimension along which you want to operate these functions take a second optional parameter that specifies which dimension you want to collapse. So

```
sum(d,2)
```

will sum the columns and produce a consistently-sized result. If you want to sum all the numbers in `d` irrespective of their position in the array, do

```
sum(d(:))
```

4.4 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. In order to do so, MATLAB provides the `find()` function, which is very similar to the `where` instruction in IDL. Strictly speaking, `find` returns the indices of all elements that are non-zero in an array. This turns out to be very useful, since all the comparisons return 1 when they are true. For example, `ix=find(a>10)` will return the indices of every element of `a` that is larger than 10. Now, we can set them all to 10, for example, by doing `a(ix)=10;`. No loops necessary!

4.5 Beyond Arrays: Structures

Structures are variables that contain other variables. Think of them as a neat way to organize data. The different fields of a structure, can contain variables of different types, so if one gives the fields a meaningful name this becomes a great way to keep track of the data.

In MATLAB one can define a structure as one goes. This is an example of a structure with 5 fields of different types:

```
image.data=[1 2 3; 4 5 6; 7 8 5];
image.date='13-Jan-2008';
image.blank=NaN;
image.ra=13.3212;
image.dec=43.3455;
```

You can operate on the fields as you would with any variable of that particular type. For example, to invert the data matrix, one would use `inv(image.data)`.

4.6 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

insert toggles between insert and overwrite modes

page up and down keys just as you expect, they scroll the Command Window

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

! escape to the shell (i.e., give an in-line shell command)

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 charaters 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 (this is also a sign of my age).

4.7 Operators

4.7.1 Relational operators

We introduced a number of operators in the previous sections, such as the *colon* operator. Another useful operator is the *transpose* operator (`'`), which switches columns and rows in matrices. MATLAB provides the traditional relational operators. `==` and `~=` are used to represent "equal" and "nonequal". See `help relop` for a description of the relational operators.

As we mentioned before, MATLAB's syntax is close to the traditional C syntax (which is fairly intuitive). Note that the AND operator is the ampersand (&), and the OR operator is the vertical bar (|), as in C. Negation is the tilde operator (`~`), since the bang (`!`) employed in C is used to escape to the shell in MATLAB. The command `c=(a<b)` will produce an array of ones and zeros, 1 wherever the comparison is true, and 0 wherever it is false (i.e., it is the vectorized version of the same operation in C). This is very different from the result of the same instruction in IDL.

4.7.2 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:

`isnan(a)` Returns 1 for every NaN in array `a`.

`isinf(a)` Returns 1 for every infinite in the input.

`isfinite(a)` Returns 1 for every finite number in the input.

`isreal(a)` Returns 1 for every non-complex number in the input.

Try `a=1/0` and `isinf(a)`, for example. Since `NaN`s propagate, it is somethings useful to throw them out of operations like taking the mean. So a function that identifies them can be very useful: `ix=find(~isnan(a)); m=mean(a(ix));` (i.e., find all values that are not `NaN`s and average them).

4.8 Loops: Hating and Loving Them

MATLAB provides the traditional loop functions: `for` and `while`, as well as the branching constructs `if-then-else`, `if-elseif`, and `switch`. To see a list of the flow control possibilities, try `help lang`.

To repeat the message about vectorized languages: 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. Programs using loops are also much much much slower than the vectorized versions. Try the following comparison of execution times of a vectorized program

```
u=rand(1,1e6);
tic; ix=find(u>0.5); u(ix)=0; toc
```

versus one that uses a `for` loop to do exactly the same

```
u=rand(1,1e6);
tic; for i=1:length(u), if (u(i)>0.5), u(i)=0; end; end; toc
```

Here the function `length()` returns the longest dimension of the array `u` (the other dimension in our case is 1). The syntax of the `for` loop is straightforward, and you can increment `i` by something different from 1 each iteration by using the double-colon construct *start:increment:end* that we have presented before.

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.

5 Input and Output

Without the ability to input and output data, even the most powerful calculation engine would be next to useless. The simplest format for data is text files. These files are frequently also called ASCII files (ASCII — pronounced *aski* — stands for American Standard Code for Information Interchange, a standard for numerically representing alphabets developed in the 1960s). You can read in text files produced with an editor, and write similar files too. Here is how.

5.1 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 `save`. Try the following sequence of commands:

```
u=1:100; v=rand(1,length(u));
m=[u;v]';
save testdata.txt -ascii m
type testdata.txt
```

The first statement creates a couple of vectors: `u` contains the integers from 1 to 100, `v` is 100 uniformly-distributed random numbers. The second statement puts both vectors 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 apostrophe (`'`) operator transposes that matrix, so that rows become columns and viceversa. Thus the end result is a matrix with 100 rows and two columns of numbers. An equivalent statement would be `m=[u',v']` if you prefer to think that way. Try looking at the variable `m` to see the resulting format. The third line tells MATLAB to save the contents of the variable `m` in text format in the file `testdata.txt`. The final statement simply shows you what is inside a text file (the MATLAB command `type` is

equivalent to the UNIX command `cat`).

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

```
fp=fopen('testdata2.txt','w');
fprintf(fp,'%f %f \n',m');
fclose(fp)
```

Here the first line opens the file `testdata.txt` for writing (the old contents are erased). The second instruction is a vectorized version of the C `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 to output the numbers to the screen by using a `fprintf` instruction with `fp` set to 1: `fprintf(1,'%f %f \n',m')`.

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

```
for i=1:length(original), fprintf(fp,'%f %f \n',u(i),v(i)); end
```

You can see again that vectorizing saves a lot of typing, and allows for more compact (thus more easy to debug) code.

5.2 Formatting print statements

The function `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.

5.3 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. Let us try

```
m_2=load('testdata.txt');
```

Please verify that the file was correctly read. A quick version of the same `load` command would be

```
load testdata.txt
```

which will create a variable `testdata` with the contents of the file.

To parse more complex file formats MATLAB provides the commands `fscanf` and `textscan`. The latter is particularly easy to use. Please look at their help.

Here we have used `load` to read text files created by MATLAB, but it can read text row and column formatted files created in a word processor, as long as they have been saved as *plain text* or *ASCII*. As a handy documenting feature, you can insert comment lines in the file (i.e., lines that begin with the MATLAB comment character, the percent % sign) and they will be ignored by `load`. We will

talk more about using comments to document files and the importance of documenting below.

5.4 Reading FITS Format Files

Most astronomical data is saved in the FITS (Flexible Image Transport System) format. FITS was developed in the 1980s and it is the standard in observatories all around the world. Basic FITS files have two parts: a header section (which has information about when and how the data was obtained, coordinate systems, units, etc), and a data section (the raw data itself, stored as binary numbers).

MATLAB has a native FITS reader. Request help on the functions `fitsread()` and `fitsinfo()`. The first one allows you to read in the data stored in a FITS file, while the second one will retrieve the information in the FITS header.

The MATLAB reader works fine, but I find it less than ideal. I wrote my own FITS reader and writer many years ago, before MATLAB implemented them, so I tend to prefer my own code. It is in my `matlab` directory, so if you paid attention to what we discussed in §2 you should already have it in your path.

To read a FITS file, use the `rfits()` function (you can request `help rfits` like for any other function). The `rfits` reader will return a structure with many fields, corresponding to the different header keyworkd present in the FITS file. To write a FITS file, you use the `wfits` command. See its help too.

After reading an image using `rfits`, the image data itself will be in the `data` field. Try the following commands:

```
r=rfits('n4254_pr.fits');
imagesc(r.data');
axis image
set(gca,'ydir','nor');
```

These instructions will read in a FITS image of NGC 4254 (M99) into MATLAB, display it as an image, format the axis so that it displays square pixels, and orient it properly so that right is West and up is North. We will come back to visualization in a moment.

6 Visualizing your Data

Enough background, and onto more interesting things. A very important part of the analysis of any dataset is its visualization. Humans are not very good at taking in arrays of numbers and making sense of them, but they excel at seeing patterns in images. Visualization allows us to put the data in terms that are more easily understandable to our senses.

Let us generate something to work with, by using some of the knowledge we acquired in the previous sections. We will generate an array of numbers using

```
t=[0:0.01:10];
```

We can now calculate a function $f(t)$, for example:

```
f=sin(2*pi*t)./exp(t/5);
```

Note that `pi` is a MATLAB function that returns the number π. What does $f(t)$ looks like?

That is easy to answer using MATLAB's `plot()` function. Try `plot(t,f)`. As we saw before, `plot()` produces a plot of the second parameter against the first, or if only one parameter is given it will plot it against the index number. It is easy to change several properties of the plot. See `help plot` for a list of colors, line types, and symbols.

Visualization can serve many purposes. Try

```
u=t+cos(2*pi*t);
plot(u,f)
```

Now we can ask the question, what is the value of u when $u(t) = f(t)$? Click on the zoom button in the control bar of "Figure 1" (the icon with the magnifying glass and the plus sign). The zoom into the first intersection of u and f, either by repeatedly clicking on it, or by clicking and dragging to define a box around it. The answer is $u = 0.847$.

Sometimes we want to specify the range of coordinates in a plot, instead of letting MATLAB choose it for us. We can do it using the `axis()` function. It takes a vector of 4 numbers, specifying the minimum and maximum of x and y. For example, try `axis([0 4 -0.2 0.2])`. The same function is used to specify some characteristics of the plot. For example, sometimes we want the scales of the axes to be such that circles appear as circles, not ellipses. We can do that by issuing the command `axis equal` in MATLAB. Look at `help axis` for other features.

Documenting plots is very important. Otherwise we would soon forget what they are about. You can annotate a plot using `xlabel()` and `ylabel()`. Let us zoom back out by typing `axis auto`, moving the scroll wheel in the mouse, or right-clicking on the figure and selecting the "Reset Zoom" option. Then issue `xlabel('u(t)'); ylabel('f(t)')`. MATLAB actually has a built-in TEX interpreter, so very fancy labels are possible (if you do not know what TEX or LATEX are, do not worry... they are fancy editors used in Physics and Math).

MATLAB is superb at producing publication-quality graphics in many formats. We showed above how to produce and annotate a basic plot. There other types of plots that can be produced, just look at the MATLAB help for *graph2d*. For amusement, try clearing producing a polar plot of one of our functions by saying `polar(t,f)`.

6.1 Being in Control

Let us erase the polar plot by issuing a "clear figure" command and recreating the previous plot

```
clf
plot(u,f)
```

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=get(gcf,'Position')
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`. The last two are the new size of the window, 400×400 pixels. The graphics are resized automatically.

A figure can have many axes, our plot is just 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.

Something really handy in Astronomy is to change the orientation of the x-axis (R.A. decreases to the right, remember?). To flip it in MATLAB, try

```
set(gca,'xdir','reverse')
```

As we have seen in the FITS example above, sometimes one needs to flip the y-axis too. The instruction `set(gca,'ydir','normal')`, for example, makes it so that it increases upwards (the normal mathematical sense).

The default background color of a plot is white. To change try something like

```
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, `set(gca,'Color',[0.2 0.8 0.6])`. To change the line width of the box framing the plot, for example, try `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 plot: `h=plot(u,f)`. Then play with the properties. By the way, the properties are case insensitive and minimum-pattern-matched. So `set(gca,'Linewidth',3)` and `set(gca,'linew',3)` are the same command.

To clean the figure, issue the "clear figure" command `clf`. To close the window, issue `close(1)`. The next plot command will create a window with all properties reset to the defaults, if necessary.

6.2 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 `hold on`. To return to the old behavior, issue `hold off`. Just issuing `hold` toggles the behavior.

So, to illustrate this, try the following commands:

```
clf
plot(t,u)
hold on
plot(t,f,'r--')
```

this will overplot f on u using a red dashed line.

A very handy command to annotate plots of this type is `legend`. Look at the help for that command. Here, we can create a simple legend by issuing the command `legend('u','f')`.

6.3 2D and 3D Visualization

MATLAB allows you equally gracefully to display 2D and 3D datasets. To begin, let us create a 2D function.

```
[x,y]=meshgrid(-100:2:100,-100:2:100);
g=exp(-(x-25).^2/800-(y-35).^2/500);
g=g-2*exp(-(x+10).^2/700-(y+40).^2/1500);
mesh(x,y,g)
```

The first line uses the `meshgrid()` function to define the coordinate grid. The second and third line add a couple of 2D Gaussians to define our surface. The fourth line displays it as a mesh plot in 3D. The color scale corresponds to the "height" of the function. Say `colorbar` to place the current color scale on the right side of the plot.

MATLAB allows you to explore the 3D surface in real time. Try clicking on the icon next to the hand, on the icon bar of the current figure (it is the one with the cube and the circular arrow around it). Now left-click on the mesh graphic and hold the button down while moving the cursor: you can change your viewpoint in real time! The current Azimuth and Elevation of your viewpoint are noted in the left bottom corner. Enter `view(3)` to go back to the default 3D viewpoint. For a smooth version of the `mesh` plot, try:

```
surf(x,y,g)
shading interp
```

In astronomy we do not use surface plots as these very commonly (our data is too noisy or too full of details for them to be useful). Many times, we use countour plots. Try

```
contour(x,y,g)
```

for a simple contour plot. Frequently we want to specify the contours and their color. For that just include an array with the contours you want to use as another parameter, and add a string parameter specifying the color of the contours. Try

```
contour(x,y,g,-2:0.1:2,'k')
```

You can also produce filled contours very easily

```
contourf(x,y,g,-2:0.1:2)
```

To have the same scale in the x and y axes, try using the `axis equal` command.

All the annotations we described previously can be used with 2D and 3D plots. Contourplots can also be annotated with `clabel()`. Search for its help if you want to use it.

6.4 Visualizing Images

Let us recreate the 2D image of a galaxy

```
r=rfits('n4254_pr.fits');
imagesc(r.data');
axis image
set(gca,'ydir','nor');
```

The `imagesc()` function creates color display of the data using the current color map and the full data range. It also flips the y-axis by default because that is handy for JPEG or PNG images which are defined row by row starting at the top, so the `set()` statement is there to unflip it.

We have been using the standard MATLAB colormap to display our images, but that is only one of many choices. Say `help graph3d` to get a list of the 3D graphic functions, including the available colormaps. The `jet` colormap is the default. Other popular choices are `hot`, `gray`, or `hsv`. To change the color map and add an intensity bar, try

```
colormap hot
colorbar
```

Sometimes one wants to display the negative image. A color map is just a $N \times 3$ matrix of numbers that specifies the RGB color codes for a normalized set of intensities. So, if we flip it, we invert the color scale and obtain a negative image. We could write our own code to flip the rows in a matrix, but MATLAB already provides the `flipud()` function. Try

```
colormap(flipud(hsv))
```

Another simple useful manipulation is to change the range of intensities in the display. To MATLAB, the intensity is just another axis, called the color axis. To define its range, it uses the function `caxis()`. So try

```
caxis([1900 3000])
```

now we can see the details of the outer spiral arms, although the nucleus is saturated.

We have already talked about how to zoom into a plot (or an image). We can also request information about points in the image. Click on the "Data Cursor" icon in the Figure Window. Now when you place the cursor over the image it should look like a cross. Click anywhere in the image. A small rectangle appears, showing you the image coordinates of the point selected, the index (i.e., the value of the image at that point), and the corresponding RGB triplet in the current color map.

To see the nucleus and the spiral arms at the same time we need to use an intensity scale that is non-linear. To do so we can try to display the logarithm of the image, instead of the image itself. Let us try the following commands

```
m=r.data;
ix=find(r.data<=0);
m(ix)=NaN;
hist(log(m(:)),100);
imagesc(log(m'));
caxis([7.5 8.9])
```

The first three lines look for pixels that are zero or negative, and replace them with harmless not-a-number values. The logarithm of 0 is $-\infty$, which MATLAB will deal with perfectly fine but would throw havoc into our data ranges. The logarithm of a negative number is a complex number. Again, MATLAB will just do the calculations, but trying to display a matrix of complex number would not be allowed by `imagesc()`. It turns out that this image does not have problematic values, so

we could have ignored that. Then we display the histogram of the logarithm of all pixels in the image using the function `hist()`. The second parameter there indicates the number of bins. This is useful to select the range in the color axis. Then we display the logarithm of the image using `imagesc()`, and we set the intensity range to a good value. Simple, isn't it?

Finally, let us overplot something on the image. For example, sometimes it is useful to mark the location of something special with a cross. Or sometimes we want to overplot contour lines. Just like in the case of the graph, we can use the `hold on` command to prevent new plots from erasing old plots. Therefore

```
hold on
plot(104,214,'+w','MarkerSize',15)
text(108,223,'Something important here','Color','w')
```

where the extra "property-value" pairs are simply a short hand for setting some of the properties of the symbol (its size) or the text (its color) without having to get a handle to their graphic properties (and use `set()`).

6.5 Eye Candy

There is much more that can be done with graphical visualization. Just to give a couple of examples, I have prepared two scripts that run on a spectral-spatial datacube. The script `visualize2d` plays a movie of the third dimension of the example datacube. The script `visualize3d` does a 3D isosurface rendering of the same datacube, using a partially transparent surface.

6.6 Making a Hard Copy

This is usually done by creating and printing a Postscript file. MATLAB has the `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 `print -depsc2 myplot.eps`. This creates an encapsulated level-2 color postcript file containing the plot. Look at the help for the `print` command for other options. If you are producing reports using Word, for example, you may want to create PNG figures instead using `print -dpng myplot.png`.

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: `set(gcf,'PaperPositionMode','auto')`.

7 Writing a Program

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, which is what MATLAB calls a *script*. It is essentially the same as if you were typing on the terminal, but all the typing can be repeated with very little effort. To write a script, 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:

```
w=sin(([1:200]/35).^2.5);
w=w+2;
t=3*[1:200];
plot(t,w)
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:

```
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, or give the keystroke sequence `<Ctrl-x><Ctrl-s>`, and save your program with the name "test.m". To invoke and execute the program, just enter `test` at the MATLAB prompt.

7.1 Documenting Code

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 percent (%) 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.

7.2 Scripts and functions

The main distinction between batch files (such as the one we wrote during this tutorial) and functions is the fact that scripts run in the user environment, while functions run in their own environment. What do I mean by environment? Environment refers to the contents of the variables defined. A scripts share the same memory space: your main workspace, the same one you use when typing at the prompt. Thus altering a variable in one script has effects on how others execute. This can be good or bad: variables can be used to pass information between scripts, but the end product depends on precisely what sequence of scripts was issued.

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 scripts share all the same sandbox. Often it is easier to debug a script, since the

internal variables remain in the user environment and are not deleted after it finishes (so they can be examined). 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

```
function y=myfunction(x)
```

where `x` is a parameter (or a list of parameters) passed to the function, and `y` is a variable (it can also be a list such as `[y,z]`) returned by the function (and hopefully assigned some value within the function). MATLAB functions do not need to return a value. Try transforming the script in §7 into a function.

7.3 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, as when one is writing a complex program with many nested routines, 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, since an error has happened). The editor will automatically load the offending file and a green arrow will indicate the position of the offending statement. To get back to the working environment, say `dbquit`. 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 from pestering you again.