Matlab Mini Manual

1997

Eleftherios Gkioulekas

Mathematical Sciences Computing Center
University of Washington
Washington, USA

and

Gjerrit Meinsma

Faculty of Applied Mathematics
University of Twente
Enschede, The Netherlands

This document is to a large extent the same as Eleftherios Gkioulekas' tutorial *Programming with Matlab* from 1996.
It is available at `http://www.amath.washington.edu/~elf/tutorials/`.
Gjerrit Meinsma made some modifications to the text and added an appendix.
This document is available at `http://www.math.utwente.nl/~meinsma/`.

Contents

1

Overview

Matlab is an interactive programming environment that facilities dealing with matrix computation, numerical analysis and graphics.

Matlab stands for *mat*rix *lab*oratory and was initially intended to provide interactive access to the LINPACK and EISPACK packages. These packages were written in Fortran and consist of some sophisticated code for matrix problems. With Matlab it became possible to experiment interactively with these packages without having to write lengthy Fortran code (or C code nowadays). Other features of Matlab are its extensive help facilities and the ease with which users can make their own library of functions.

Throughout this tutorial we will give you an overview of various things and refer you to Matlab's on-line help for more information. The best way to learn is by experimentation, and the best way this tutorial can help you is by telling you the basics and giving you directions towards which you can take off exploring on your own.

1.1 Matlab essentials and the on-line help facilities

To begin a Matlab session type at the prompt

```
Matlab
```

or from WINDOWS95 you can double-click the Matlab icon to begin the session. If properly installed, the system will show for split second a small welcoming window and then some general information and something like

```
Commands to get started: intro, demo, help help
Commands for more information: help, whatsnew, info, subscribe
```

If you are new to Matlab you might take their advice and type

```
>> intro
```

The `>>` should not be typed, it is Matlab's way of saying that it is ready for you to enter a command, like `intro`. Typing `>> demo` will bring up a flashy screen from which various demos can be run, but it is not the way to learn Matlab. To quit the Matlab session type

```
>> quit
```

or

```
>> exit
```

Now you have learned to enter Matlab and quit it again. These are the most important things to know about any program. Before going into the details in the following chapters, let us do some simple tests. Enter Matlab again and type

```
>> x=2
>> y=3;
>> z=x*y
```

After entering the first line, Matlab immediately responds that `x` is indeed equal to 2. The second line produces nothing on screen although it does assign the value 3 to the variable `y`. The semicolumn `;` tells Matlab to execute the command silently. The third line has no `;` and Matlab responds with

```
z =

  6
```

If you type `>> x*y` then there is no variable to assign the value of `x*y` to. In such cases Matlab assigns the outcome to the variable `ans` (from *ans*wer).

Typing the following three lines will produce a plot.

```
>> x=[0 1 2 3 4 5];
>> y=sin(x);
>> plot(x,y);
```

Here `x` is assigned the value of the array `[0 1 2 3 4 5]`. At this stage we do not want to go into the details of arrays and `sin` and `plot` yet (we come to it in later chapters), instead we want to convey the message to use Matlab's on-line help whenever you encounter something unfamiliar. Look at the second line. Apparently we can calculate the sine of an array! How is it defined? Type

```
>> help sin
```

and Matlab will tell that `sin(x)` is the sine of *the elements* of `x`. Okay, so `y=sin(x)` assigns `y` the array `[sin(0) sin(1) sin(2) sin(3) sin(4) sin(5)]`. With the command `plot(x,y)` the actual plot is made, see Figure. 1.1.

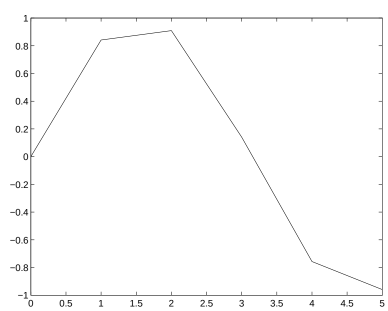

Figure 1.1: A coarse plot of $\sin(x)$.

To see how `plot` works type

```
>> help plot
```

There is a lot of help available on `plot`. If it scrolls off your screen you might want to enter `>> more on` and then `>> help plot` again. The help on `plot` is extensive and it should provide enough information to answer most questions you may have about plotting.

Another way of getting help is with the `lookfor` command. Suppose you wish to save the plot in a file on disk. Typing `help save` turns out to be of little use. In such cases try

```
>> lookfor save
```

This will print on screen a list of commands that have something to do with "save":

```
DIARY   Save the text of a MATLAB session.
SAVE    Save workspace variables on disk.
HARDCOPY Save figure window to file.
PRINT  Print graph or save graph to file.
WAVWRITE Saves Microsoft Windows 3.1 .WAV format sound files.
IDUISESS Handles load and save session as well as renaming of session.
```

Apparently `hardcopy` or `print` are the commands we need to look at. The help on `print` will tell you that

```
>> print -deps myfig.eps
```

saves the current plot in encapsulated postscript format in the file with name `myfig.eps`. Consult `help print` once again if you need a format other than postscript.

For more information on getting help, type

```
>> help help
```

which, you guessed it, gives help on using help. If you are on a Unix machine with a web browser, try also

```
>> doc
```

1.2 Loading, saving and m-files

The interactive nature of Matlab is useful, but at the end of the day you want your commands and results saved in a file. Bring up an editor[1] and type in the following Matlab program:

```
%
% These are comments.
% In Matlab you can add comments with a % character
% This is the standard hello world program

disp('Hello world!');
```

Save the program with the filename `hello.m`. Then start Matlab *under the same directory where you saved the file* and type `hello` at the Matlab prompt. Then you should see the following:

```
>> hello
Hello world!
>>
```

Notice that within the Matlab environment you don't have to type any funny commands to *load* your program. When you start Matlab from your shell, Matlab will load *all* the `*.m` files that happen to be under the present working directory at startup time. So, all you have to do when the prompt shows up is to type the name of the program (*without the `.m` extension*) and it will get executed. Note that Matlab will only look for the files with extension `.m` so you are forced to use it. There are no work-arounds

[1] On a Unix system you can use the following editors: `emacs`, `vi` and if available `pico`. On DOS there is `edit` on WINDOWS95 there are `notepad` and others. Warning: If you use a word-processor to type in Matlab programs, be sure to save the file as a plain text file or ASCII file. By default word processors save files in a format that Matlab cannot read.

for this. Still, Matlab has a provision for the situation where your files are scattered in more than one directory. You can use the Unix `cd`, `ls`, `pwd` commands to navigate in the file system and change your current working directory. Also, if you want to be able to have access to the files on two or more separate directories *simultaneously* type

```
>> help path
```

for more information. Matlab program files are often called *scripts*.

In the previous section you saw how to save a plot. It is also possible to save and load your variables on disk. Try

```
>> help save
```

and to load them the next session, try

```
>> help load
```

For simple problems there is no need to save and load the variables. Simply save the m-file that has all the commands and execute the m-file when needed. Yet another way to save results is with the `diary` command. From the moment you type

```
>> diary myoutput.txt
```

all subsequent commands that you type, and its output on screen, are also send to the file `myoutput.txt`. In fact *everything* that appears on screen will also appear in `myoutput.txt`. With

```
>> diary off
```

the file `myoutput.txt` is closed.

2

Scalars, arrays, matrices and strings

Matlab has three basic data types: strings, scalars and matrices. Arrays are just matrices that have only one row. Matlab has also lots of built-in functions to work with these things. You have already seen the `disp` function in our hello-world program.

2.1 Strings

Starting with strings, you can assign a string to a variable like this:

```
name = 'Indiana Jones';
```

Note that it is a syntax error to quote the string with anything other than the forward quote marks. So, the following are **wrong!**

```
name = "Indiana Jones";     % wrong!
name = `Indiana Jones`;     % wrong!
```

In a Matlab program you can prompt the user and ask him to enter in a string with the `input` command:

```
% This is a rather more social program, assuming you're a man
%
yourname = input('Hello! Who are you? ','s');
dadname  = input('What''s your daddy''s name? ','s');
fprintf(1,'Hail oh %s son of %s the Great! \n',yourname,dadname);
```

The `input` command takes two arguments. The first argument is the string that you want the user to be prompted with. You could stick in a variable instead of a fixed string if you wanted to. The second argument, `'s'`, tells Matlab to expect the user to enter a string. If you omit the second argument, then Matlab will be expecting a number, and upon you entering your name, Matlab will complain. Finally, it *returns* the value that the user enters, and that value is passed on through assignment to the variable `yourname`.

The `fprintf` command gives you more flexibility in producing output than the `disp` command. Briefly, `fprintf` takes two or three or more arguments. The first argument is a *file descriptor*. File descriptors are integers that reference places where you can send output and receive input from. In Matlab, file descriptor 1 is what you use when you want to send things to the screen. The terminology you may hear is that file descriptor 1 sends things to the *standard output*.

The rest of the arguments depend on what you want to print. If all you want to print is a fixed string, then you just put that in as your second argument. For example:

```
fprintf(1,'Hello world!\n');
```

The `\n` sequence will switch you over to the next line. `disp` will do this automatically, in `fprintf` you must explicitly state that you wish to go to a new line. This is a feature, since there may be situations where you do *not* want to go to a new line.

If you want to print the values of variables interspersed with your string then you need to put appropriate markers like `%s` to indicate where you want your variables to go. Then, in the subsequent arguments you list the variables in the appropriate order, making sure to match the markers. There are many markers and the Matlab online help will refer you to a C manual. The most commonly used markers are the following:

`%s`	Strings
`%d`	Integers (otherwise you get things like `5.0000`)
`%g`	Real numbers in scientific notation, like `1e-06`

In our example above, we just used `%s`. You will see further examples later on.

Note that if you merely want to print a variable, it is better to use `disp` since it will format it for you. `fprintf` is more useful for those occasions where you want to do the formatting yourself as well as for sending things to a file.

In Appendix A, Sections A.13 and A.14 you find listings of the standard string commands and the input/output commands like `fprintf`.

2.2 Scalars

Scalars can be assigned, inputed and printed in a similar fashion. Here is an example:

```
% yet another one of these happy programs
age = input('Pardon for asking but how old are you? ');
if (age < 75)
  ll = 365.25*24*(75 - age);
  fprintf(1,'You have %g hours left of average life expectancy.\n',ll);
else
  fprintf(1,'Geez! You are that old?!\n');
end
```

Note the following:

- String and numeric variables look the same. You don't have to declare the type of the variable anywhere. Matlab will make sure to do the right thing.
- When we use `input` to get the value of a numeric variable we omit the second `'s'` argument. This way, Matlab will do error-checking and complain if you entered something that's not a number.
- You can use `fprintf` to print numeric variables in a similar fashion, but you got to use the `%g` marker. If you are printing an integer you must use the `%d` marker, otherwise Matlab will stick in a few zeroes as decimal places to your integer. It is obvious that you can mix strings and numbers in an `fprintf` command, so long as you don't mix up the order of the variables listed afterwards.
- On the line

  ```
  life_left = 365.25*24*(75 - age);
  ```

  we see how you can do simple computations in Matlab. It's very similar to C and Fortran and to learn more about the operators you have available type

```
>> help ops
>> help relops
```

- Finally, we have an example of an `if` statement. We will talk of that more later. The meaning should be intuitively obvious.

In addition to ordinary numbers, you may also have complex numbers. The symbols `i` and `j` are reserved for such use. For example you can say:

```
z = 3 + 4*i;
```

or

```
z = 3 + 4*j;
```

where `i` and `j` represent $\sqrt{-1}$. If you are already using the symbols `i` and `j` as variables, then you can get a new complex unit and use it in the usual way by saying:

```
ii = sqrt(-1);
z = 3 + 4*ii;
```

Lists on what you can do with scalars and predefined constants, are in Appendix A, Sections A.2, A.4, A.6 and A.7.

2.3 Arrays

Next we talk about arrays. Arrays and matrices are Matlab's distinguishing feature. In Matlab arrays are dynamic and they are indexed from 1. You can assign them element by element with commands like:

```
a(1) = 23;
a(2) = input('Enter a(2)');
a(3) = a(1)+a(2);
```

It is a syntax error to assign or refer to `a(0)`. This is unfortunate since in some cases the 0-indexing is more convenient. Note that you don't have to initialize the array or state it's size at any point. The array will make sure to grow itself as you index higher and higher indices.

Suppose that you do this:

```
a(1) = 10;
a(3) = 20;
```

At this point, `a` has grown to size 3. But `a(2)` hasn't been assigned a value yet. In such situations, during growth any unset elements are set to zero. It is good programming practice however not to depend on this and always initialize all the elements to their proper values.

Notice that for the sake of efficiency you might not like the idea of growing arrays. This is because every time the array is grown, a new chunk of memory must be allocated for it, and contents have to be copied. In that case, you can set the size of the array by initializing it with the `zeros` command:

```
a = zeros(100);
```

This will set `a(1),a(2),...,a(100)` all equal to zero. Then, so long as you respect these boundaries, the array will not have to be grown at any point.

Here are some other ways to make assignments to arrays:

```
x = [3 4 5 6];
```

will set `x` equal to the array of 4 values with `x(1)=3`, `x(2)=4`, `x(3)=5` and `x(4)=6`. You can recursively add elements to your array `x` in various ways if you include `x` on the right hand side. For example, you can make assignments like

```
x = [x 1 2]    % append two elements at the end of the array
x = [1 2 x 3 ] % append two elements at the front, one at the back
```

How about making deletions? Well, first of all notice that we can access parts of the array with the following indexing scheme:

```
y = x(2:4);
```

will return the an array of `x(2)`, `x(3)`, `x(4)`. So, if you want to delete the last element of the array, you just have to find the size of the array, which you can do with the `size` or `length` command, and then

```
x=x(1:length(x)-1);
```

effectively deletes the last entry from `x`.

You can use multiple indices on the left hand side of the assignment:

```
x([1 5 10])=[0.1 0.2 0.3];
```

This assigns `x(1)=0.1`, `x(5)=0.2` and `x(10)=0.3`. By the way, this opens up another way to delete entries from an array:

```
x=[1 2 3 4 5];
x([3 5])=[];
```

Now `x=[1 2 4]`. In Matlab `[]` is an array or matrix with zero elements, so `x([3 5])=[]` instructs Matlab to remove `x(3)` and `x(5)` from the array `x`.

Yet another way to setup arrays is like this:

```
x = 3:1:6;
```

This will set `x` equal to an array of equidistant values that begin at 3, end at 6 and are separated from each other by steps of 1. You can even make backwards steps if you provide a negative stepsize, like this:

```
x = 6:-1:3;
```

It is common to set up arrays like these when you want to plot a function whose values are known at equidistant points.

2.3.1 Polynomials

A polynomial $p_n s^n + p_{n-1} s^{n-1} + \cdots + p_0$ is in Matlab represented as an array of its coefficients

$$\texttt{p = [}\ p_n \quad p_{n-1} \quad \cdots \quad p_0\ \texttt{]}$$

Formally, then, arrays and polynomials are the same thing in Matlab. Be aware of the reversed order of the coefficients: `p(1)` equals p_n and `p(length(p))` is the constant coefficient. Matlab has several commands for polynomials. For example the product of two polynomials `p` and `q` can be found with

```
r=conv(p,q)
```

And what about this one:

```
pz=roots(p)
```

It returns an array of the zeros of `p`. As you probably know, there is no general analytic expression for the zeros of a polynomial if its degree exceeds 4. Numerical computation of zeros is inherently an iterative process and it is not at all a trivial task.

See Appendix A, Sections A.2, A.4, A.6, A.7, A.9 and A.10 for listings of the standard array and polynomial commands. Several of the *toolboxes* have many more commands that deal with polynomials. See `help` for a list of the installed toolboxes.

2.3.2 Saving and loading arrays

Finally, to conclude, you may want to know how to load arrays from files. Suppose you have a file that contains a list of numbers separated with carriage returns. These numbers could be the values of a function you want to plot on known values of `x` (presumably equidistant). You want to load all of these numbers on a vector so you can do things to them. Here is a demo program for doing this:

```
filename = input('Please enter filename:','s');
fd = fopen(filename);
vector = fscanf(fd,'%g',inf);
fclose(fd);
disp(vector);
```

Here is how this works:

- The first line, prompts the user for a filename.
- The `fopen` command will open the file for reading and return a file descriptor which we store at variable `fd`.
- The `fscanf` command will read in the data. You really need to read the help page for `fscanf` as it is a very useful command. In principle it is a little similar to `fprintf`. The first argument is the file descriptor from which data is being read. The second argument tells Matlab, what kind of data is being read. The `%g` marker stands for real numbers in scientific notation. Finally the third argument tells Matlab to read in the entire file in one scoop. Alternatively you can stick in an integer there and tell Matlab to load only so many numbers from the file.
- The `fclose` command will close the file descriptor that was opened.
- Finally the `disp` command will show you what has been loaded. At this point you could substitute with somewhat more interesting code if you will.

Another common situation is data files that contain pairs of numbers separated by carriage returns. Suppose you want to load the first numbers onto one array, and the second numbers to another array. Here is how that is done:

```
filename = input('Please enter filename: ','s');
fd = fopen(filename);
A  = fscanf(fd,'%g %g\n',[2,inf]);
x  = A(1,:);
y  = A(2,:);
fclose(fd);
disp('Here comes x:'); disp(x);
disp('Here comes y:'); disp(y);
```

Again, you need to read the help page for `fscanf` to understand this example better. You can use it in your programs as a canned box until then. What we do in this code snippet essentially is to load the file into a two-column matrix, and then extract the columns into vectors. Of course, this example now leads us to the next item on the agenda: matrices.

2.4 Matrices

In Matlab, arrays are matrices that have only one row. Like arrays, matrices can be defined element by element like this:

```
a(1,1) = 1; a(1,2) = 0;
a(2,1) = 0; a(2,2) = 1;
```

Like arrays, matrices grow themselves dynamically as needed when you add elements in this fashion. Upon growth, any unset elements default to zero just like they do in arrays. If you don't want that, you can use the `zeros` command to initialize the matrix to a specific size and set it equal to zero, and then take it from there. For instance, the following example will create a zero matrix with 4 rows and 5 columns:

```
A = zeros(4,5);
```

To get the size of a matrix, we use the `size` command like this:

```
[rows,columns] = size(A);
```

When this command executes, the variable `rows` is set equal to the number of rows and `columns` is set equal to the number of columns. If you are only interested in the number of rows, or the number of columns then you can use the following variants of `size` to obtain them:

```
rows = size(A,1);
columns = size(A,2);
```

Since arrays are just matrices with one row, you can use the `size(array,2)` construct to get hold of the size of the array. Unfortunately, if you were to say:

```
s = size(array);
```

it would be wrong, because `size` returns both the number of rows and columns and since you only care to pick up one of the two numbers, you pick up the number of rows, which for arrays is always equal to 1. Not what you want!

Naturally, there are a few other ways to assign values to a matrix. One way is like this:

```
A = [ 1 0 0 ; 0 1 0 ; 0 0 1]
```

This will set `A` equal to the 3 by 3 identity matrix. In this notation you list the rows and separate them with semicolons.

In addition to that you can extract pieces of the matrix, just like earlier we showed you how to extract pieces of the arrays. Here are some examples of what you can do:

```
a = A(:,2);        % this is the 2nd column of A
b = A(3,:);        % this is the 3rd row of A
c = A(1:4,3);      % this is a 4 by 1 submatrix of A
d = A(:,[2 4 10]); % this is the 2nd, 4th and 10th columns of A stacked
```

In general, if `v` and `w` are arrays with integer components, then `A(v,w)` is the matrix obtained by taking the elements of A with row subscripts from `v` and column subscripts from `w`. So:

```
n = size(A,2);
A = A(:,n:-1:1);
```

will reverse the columns of A. Moreover, you can have **all** of these constructs appear on the left hand side of an assignment and Matlab will do the right thing. For instance

```
A(:,[3 5 10]) = B(:,1:3)
```

replaces the third, fifth and tenth columns of `A` with the first three columns of `B`.

In addition to getting submatrices of matrices, Matlab allows you to put together block matrices from smaller matrices. For example if `A,B,C,D` are a square matrices of the same size, then you can put together a block matrix like this:

```
M = [ A B ; C D ]
```

Finally, you can get the complex conjugate transpose of matrix by putting a `’` mark next to it. For example:

```
A = [ 2 4 1 ; 2 1 5 ; 4 2 6];
Atrans = A’;
```

Matlab provides functions that return many special matrices. These functions are listed in Appendix A, Section A.5 and we urge you to look these up with the `help` command and experiment.

To display the contents of matrices you can simply use the `disp` command. For example, to display the 5 by 5 Hilbert matrix you want to say:

```
>> disp(hilb(5))
    1.0000    0.5000    0.3333    0.2500    0.2000
    0.5000    0.3333    0.2500    0.2000    0.1667
    0.3333    0.2500    0.2000    0.1667    0.1429
    0.2500    0.2000    0.1667    0.1429    0.1250
    0.2000    0.1667    0.1429    0.1250    0.1111
```

The Hilbert matrix is a famous example of a badly conditioned matrix. It is also famous because the exact inverse of it is known analytically and can be computed with the `invhilb` command:

```
>> disp(invhilb(5))
          25        -300        1050       -1400         630
        -300        4800      -18900       26880      -12600
        1050      -18900       79380     -117600       56700
       -1400       26880     -117600      179200      -88200
         630      -12600       56700      -88200       44100
```

This way you can interactively use these famous matrices to study concepts such as ill-conditioning.

2.5 Matrix/Array Operations

- **Matrix addition/subtraction**: Matrices can be added and subtracted like this:

  ```
  A = B + C;
  A = B - C;
  ```

It is necessary for both matrices `B` and `C` to have the same size. The exception to this rule is when adding with a scalar:

```
A = B + 4;
```

In this example, all the elements of `B` are increased by 4 and the resulting matrix is stored in `A`.

- **Matrix multiplication**: Matrices `B` and `C` can be multiplied if they have sizes $n \times p$ and $p \times m$ correspondingly. The product then is evaluated by the well-known formula

$$A_{ij} = \sum_{k=1}^{p} B_{ik} C_{kj},\ \forall i \in \{1, \ldots, n\},\ \forall j \in \{1, \ldots, m\}$$

In Matlab, to do this you say:

```
A = B*C;
```

You can also multiply all elements of a matrix by a scalar like this:

```
A = 4*B;
A = B*4;  % both are equivalent
```

Since vectors are matrices that are $1 \times n$, you can use this mechanism to take the dot product of two vectors by transposing one of the vectors. For example:

```
x = [2 4 1 5 3];
y = [5 3 5 2 9];
p = x'*y;
```

This way, `x'` is now a $n \times 1$ "matrix" and `y` is $1 \times n$ and the two can be multiplied, which is the same as taking their dot product $\sum_{k=1}^{k=n} \bar{x}(k) y(k)$.

Another common application of this is to apply $n \times n$ matrices to vectors. There is a catch though: In Matlab, vectors are defined as matrices with one row, i.e. as $1 \times n$. If you are used to writing the matrix product as Ax, then you have to transpose the vector. For example:

```
A = [1 3 2; 3 2 5; 4 6 2];  % define a matrix
x = [2 5 1];                % define a vector
y = A*x;                    % This is WRONG
y = A*x';                   % This is correct
```

- **Array multiplication:** This is an alternative way to multiply *arrays*:

$$C_{ij} = A_{ij} B_{ij}, \quad \forall i \in \{1, \ldots, n\},\ \forall j \in \{1, \ldots, m\}$$

This is not the traditional matrix multiplication but it's something that shows up in many applications. You can do it in Matlab like this:

```
C = A.*B;
```

- **Array division:** Likewise you can divide arrays in Matlab according to the formula:

$$C_{ij} = \frac{A_{ij}}{B_{ij}}, \quad \forall i \in \{1, \ldots, n\},\ \forall j \in \{1, \ldots, m\}$$

by using the `./` operator like this:

```
C = A./B;
```

- **Matrix division:** There are two *matrix division* operators in Matlab: `/` and `\`. In general

  `X = A\B` is a solution to `A*X = B`
  `X = B/A` is a solution to `X*A = B`.

  This means that `A\B` is defined whenever `A` has full row rank and has as many rows as `B`. Likewise `B/A` is defined whenever `A` has full column rank and has as many columns as `B`. If `A` is a square matrix then it is factored using Gaussian elimination. Then the equations `A*X(:,j)=B(:,j)` are being solved for every column of `B`. The result is a matrix with the same dimensions as `B`. If `A` is not square, it is factored using the Householder orthogonalization with column pivoting. Then the corresponding equations will be solved in the least squares fit sense. Right division `B/A` in Matlab is computed in terms of left division by `B/A = (A'\B')'`. For more information type

  ```
  >> help slash
  ```

- **Matrix inverse:** Usually we are not interested in matrix inverses as much as applying them directly on vectors. In these cases, it's best to use the matrix division operators. Nevertheless, you can obtain the inverse of a matrix if you need it by using the `inv` function. If `A` is a matrix then `inv(A)` will return it's inverse. If the matrix is singular or close to singular a warning will be printed.

- **Matrix determinants:** Matrix determinants are defined by

  $$\det(A) = \sum_{\sigma \in S_n} \left\{ \operatorname{sign}(\sigma) \prod_{i=1}^{n} A_{i\sigma(i)} \right\}$$

  where S_n is the set of permutations of the ordered set $(1, 2, \ldots, n)$, and $\operatorname{sign}(\sigma)$ is equal to $+1$ if the permutation is even and -1 if the permutation is odd. Determinants make sense only for square matrices and can be computed with the `det` function:

  ```
  a = det(A);
  ```

- **Eigenvalues and singular values:** The eigenvalues of a square matrix A are the numbers λ for which $\lambda I - A$ is singular. In Matlab you can find the eigenvalues with

  ```
  E = eig(A)
  ```

  Perhaps even more important are the singular values and the singular value decomposition of a matrix A, with A not necessarily square.

  ```
  [U,S,V] = svd(A)
  ```

  produces a diagonal matrix `S`, of the same size as `A` and with nonnegative diagonal elements in decreasing order, and unitary matrices `U` and `V` such that `A=U*S*V'` (note the transpose sign). This is the singular value decomposition of `A` and the diagonal entries of `S` are called the singular values of `A`.

- **Matrix exponential function:** These are some very fascinating functions. The *matrix exponential* function is defined by

  $$\exp(A) = \sum_{k=0}^{+\infty} \frac{A^k}{k!}$$

where the power A^k is to be evaluated in the *matrix product* sense. Recall that your ordinary exponential function is defined by

$$e^x = \sum_{k=0}^{+\infty} \frac{x^k}{k!}$$

which converges for all x (even when they are complex). It is not that difficult to see that the corresponding matrix expression also converges, and the result is the *matrix exponential*. The matrix exponential can be computed in Matlab with the `expm` function. It's usage is as simple as:

```
Y = expm(A);
```

Matrix exponentials show up in the solution of systems of differential equations.

Matlab has a *plethora* of commands that do almost anything that you would ever want to do to a matrix. And we have only discussed a subset of the operations that are permitted with matrices. A list of matrix commands can be found in Appendix A, Sections A.2, A.4, A.6 and A.8 and A.5.

3

Flow Control

So far, we have spent most of our time discussing the data structures available in Matlab, and how they can be manipulated, as well as inputted and outputted. Now we continue this discussion by discussing how Matlab deals with *flow control*.

3.1 For loops

In Matlab, a for-loop has the following syntax:

```
for v = matrix
  statement1;
  statement2;
  ....
end
```

The columns of `matrix` are stored one at a time in the variable `v`, and then the statements up to the `end` statement are executed. If you wish to loop over the rows of the matrix then you simply transpose it and plug it in. It is recommended that the commands between the `for` statement and the `end` statement are indented by two spaces so that the reader of your code can visually see that these statements are enclosed in a loop.

In many cases, we use arrays for matrices, and then the for-loop reduces to the usual for-loop we know in languages like Fortran and C. In particular using expressions of the form `for v=1:10` will effectively make the loop variable be a scalar that goes from `1` to `10` in steps of 1. Using an expression of the form `start:step:stop` will allow you to loop a scalar from value `start` all the way to value `stop` with stepsize `step`. For instance the Hilbert matrix is defined by the equation:

$$A_{ij} = \frac{1}{i+j+1}$$

If we didn't have the `hilb` command, then we would use for-loops to initialize it, like this:

```
N = 10;
A = zeros(N,N);
for i = 1:N
  for j = 1:N
    A(i,j) = 1/(i+j-1);
  end
end
```

`a == b`	True when `a` equals `b`
`a > b`	True when `a` is greater than `b`
`a < b`	True when `a` is smaller than `b`
`a <= b`	True when `a` is smaller or equal to `b`
`a >= b`	True when `a` is greater or equal to `b`
`a ~= b`	True when `a` is not equal to `b`
`a & b`	True when both boolean expressions `a` and `b` are true
`a | b`	True when at least one of `a` or `b` is true.
`a xor b`	True only when only one of `a` or `b` is true.
`~a`	True when `a` is false.

Table 3.1: Some boolean expressions in Matlab.

3.2 While loops

In Matlab while loops follow the following format:

```
while variable
  statement1;
  statement2;
  ....
  statementn;
end
```

where `variable` is almost always a *boolean expression* of some sort. In Matlab you can compose boolean expressions as shown in Table 3.1.

Here is an example of a Matlab program that uses the while loop:

```
n = 1;
while prod(1:n) < 1.e100
  n = n + 1;
end
disp(n);
```

This program will display the first integer for which $n!$ is a 100-digit number. The `prod` function takes an array (or matrix) as argument and returns the product of it's elements. In this case, `prod(1:n)` is the factorial $n!$.

Matlab does not have repeat-until loops.

3.3 If and Break statements

The simplest way to set-up an `if` branch is like this:

```
if variable
  statement1;
  ....
  statementn;
end
```

The statements are executed only if the real part of the `variable` has all non-zero elements[1]. Otherwise, the program continues with executing the statements right after the `end` statement. The variable is usually the result of a boolean expression. The most general way to do if-branching is like this:

```
if variable
  statement1;
  ......
  statementn;
elseif variable2
  statement21;
  ......
  statement2n;
 [....as many elseif statements as you want...]
else
  statementm1;
  ......
  statementmn;
end
```

In this case, if `variable` is true, then the statements right after it will be executed until the first `else` or `elseif` (whichever comes first), and then control will be passed over to the statements after the `end`. If `variable` is not true, then we check `variable2`. Now, if that one is true, we do the statements following thereafter until the next `else` or `elseif` and when we get there again we jump to `end`. Recursively, upon consecutive failures we check the next `elseif`. If all the `elseif` variables turn out to be false, then we execute the statements after the `else`. Note that the `elseif`s and/or the `else` can be omitted all together.

Here is a general example that illustrates the last two methods of flow control.

```
% Classic 3n+1 problem from number theory:
%
while 1
  n = input('Enter n, negative n quits. ');
  if n <= 0
    break
  end
  while n > 1
    if rem(n,2) == 0    % if n is even
      n = n/2;          % then divide by two
    else
      n = 3*n+1;
    end
  end
end
```

This example involves a fascinating problem from number theory. Take any positive integer. If it is even, divide it by 2; if it is odd, multiply it by 3 and add 1. Repeat this process until your integer becomes a 1. The fascinating unsolved problem is: Is there any integer for which the process does not terminate? The conjecture is that such integers do not exist. However nobody has managed to prove it yet.

The `rem` command returns the remainder of a Euclidean division of two integers. (in this case `n` and `2`.)

[1]Note that in the most general case, `variable` could be a complex number. The `if` statement will only look into its real part.

4

Functions

Matlab has been written so that it can be extended by the users. The simplest way to do that is to write `functions` in Matlab code. We will illustrate this with an example: suppose you want to create a function called `stat` which will take an array and return it's mean and standard deviation. To do that you must create a file called `stat.m`. Then in that file, say the following:

```
function [mean, stdev] = stat(x)
% stat -- mean and standard deviation of an array
%   The stat command returns the mean and standard deviation of
%   the elements of an array. Typical syntax is like this:
%   [mean,dev] = stat(x);
m=length(x);
mean=sum(x)/m;
y=x-mean;
stdev=sqrt(sum(y.^2)/m);
```

The first line of the file should have the keyword `function` and then the syntax of the function that is being implemented. Notice the following things about the function syntax:

- The function name, `stat`, in the first line is the same as the name of the file, `stat.m`, but without the `.m`.
- Functions can have an arbitrary number of arguments. In this case there is only one such argument: `x`. The arguments are being passed by *value*: The variable that the calling code passes to the function is *copied* and the copy is being given to the function. This means that if the function internally changes the value of x, the change will **not** reflect on the variable that you use as argument on your main calling code. Only the copy will be changed, and that copy will be discarded as soon as the function completes it's call.
- Of course it is not desirable to only pass things by value. The function has to communicate some information back to the calling code. In Matlab, the variables on the right hand side, listed in brackets, are returned by the function to the user. This means that any changes made to those variables while inside the function will reflect on the corresponding variables on the calling code. So, if one were to call the function with:

  ```
  a = 1:20;
  m = 0;
  s = 0;
  [m,s] = stat(a);
  ```

  then the values of the variables `m` and `s` would change after the call to `stat`.

- All other variables used in the function, like `y` and `m`, are local variables and are discarded when the function completes it's call.
- The lines with a `%` following the function definition are comments. However, these comments are what will be spit out if you type `help stat`.

```
>> help stat
  stat -- mean and standard deviation of an array
    The stat command returns the mean and standard deviation of
    the elements of an array. Typical syntax is like this:
    [mean,dev] = stat(x);
>>
```

  You usually want to make the first comment line be a summary of what the function does because in some instances only the first line will get to be displayed, so it should be complete. Then in the lines afterwards, you can explain how the function is meant to be used.

A problem with Matlab function definitions is that you are forced to put every function in a separate file, and are even restricted in what you can call that file. Another thing that could cause you problems is *name collision*: What if the name you choose for one of your functions happens to be the name of an obscure built-in Matlab function? Then, your function will be completely ignored and Matlab will call up the built-in version instead. To find out if this is the case use the `which` command to see what Matlab has to say about your function.

Notice that function files and script files both have the `.m` extension. An m-file that does not begin with `function` is considered a script.

Many of the examples we saw earlier would be very useful if they were implemented as functions. For instance, if you commonly use Matlab to manipulate data that come out in (x, y) pairs you can make our earlier example into a function like this:

```
function [x,y] = load_xy(filename)
% load_xy  -- Will allow you to load data stored in (x,y) format
% Usage:
% Load your data by saying
%   [x,y] = load_xy(filename)
% where 'filename' is the name of the file where the data is stored
% and 'x' and 'y' are the vectors where you want the data loaded into
fd = fopen(filename);
A  = fscanf(fd,'%g %g\n',[2,inf]);
x  = A(1,:);
y  = A(2,:);
fclose(fd);
```

You would have to put this in a file called `load_xy.m` of course. Suppose that after making some manipulations you want Matlab to save your data on file again. One way to do it is like this:

```
function save_xy(filename,x,y)
% save_xy  -- Will let your save data in (x,y) format.
% Usage:
% If x and y are vectors of equal length, then save them in (x,y)
% format in a file called 'filename' by saying
% save_xy(filename,x,y)
fd = fopen(filename,'w');
A(1,:) = x;
A(2,:) = y;
```

```
fprintf(fd,'%g %g\n',A);
fclose(fd);
```

Notice that it is not necessary to use a `for` loop to `fprintf` or `fscanf` the data one by one. This is explained in detail in the on-line help pages for these two commands. In many other cases Matlab provides ways to eliminate the use of `for`-loops and when you make use of them, your programs will generally run faster. A typical example is *array promotion*. Take a very simple function

```
function y = f(x)
```

which takes a number `x` and returns another number `y`. Typical such functions are `sin, cos, tan` and you can always write your own. Now, if instead of a number you plug in an array or a matrix, then the function will be applied on every element of your array or matrix and an array of the same size will be returned. This is the reason why you don't have to specify in the function definition whether `x` and `y` are simple numbers, or arrays in the first place! To Matlab everything is a matrix as far as functions are concerned. Ordinary numbers are seen as 1×1 matrices rather than numbers. You should keep that in mind when writing functions: sometimes you may want to multiply your `x` and `y` with the `.*` operator instead of the `*` to handle array promotion properly. Likewise with division. Expect to be surprised and be careful with array promotion.

Let's look at an example more closely. Suppose you write a function like this:

```
function x = foo(y,z)
x = y+z;
```

Then, you can do the following on the Matlab prompt:

```
>> disp(foo(2,3))
     5
>> a = 1:1:10;
>> b = 1:2:20;
>> disp(foo(a,b))
     2     5     8    11    14    17    20    23    26    29
```

What you will *not* be allowed to do is this:

```
>> a = 1:1:10
>> b = 1:1:20
>> disp(foo(a,b))
??? Error using ==> +
Matrix dimensions must agree.

Error in ==> /home/lf/mscc/matlab/notes/foo.m
On line 2  ==> x = y+z;
```

The arguments `a` and `b` can not be added because they don't have the same size. Notice by the way that we used addition as our example on purpose. We challenge you to try `*` versus `.*` and see the effects!

One use of functions is to build complex algorithms progressively from simpler ones. Another use is to automate certain commonly-used tasks as we did in the example of loading and saving (x, y) pairs. Functions do not solve all of the worlds problems, but they can help you a lot and you should use them when you have the feeling that your Matlab program is getting too long and complicated and needs to be broken down to simpler components.

A good way to learn how to write functions is by copying from existing functions. To see the contents of an m-file, say, `hilb.m`, do `type hilb`.

```
>> type hilb

function H = hilb(n)
%HILB   Hilbert matrix.
%   HILB(N) is the N by N matrix with elements 1/(i+j-1),
%   which is a famous example of a badly conditioned matrix.
%   See INVHILB for the exact inverse.
%
%   This is also a good example of efficient Matlab programming
%   style where conventional FOR or DO loops are replaced by
%   vectorized statements.  This approach is faster, but uses
%   more storage.

%   C. Moler, 6-22-91.
%   Copyright (c) 1984-96 by The MathWorks, Inc.
%   $Revision: 5.4 $  $Date: 1996/08/15 21:52:09 $

%   I, J and E are matrices whose (i,j)-th element
%   is i, j and 1 respectively.

J = 1:n;
J = J(ones(n,1),:);
I = J';
E = ones(n,n);
H = E./(I+J-1);
```

To include this code in the file you're editing you can either cut-and-paste it from the Matlab-window or you can insert the file `hilb.m` directly. For the latter you need to locate the file first with the command

```
>> which hilb
```

Finally, we urge you to try to keep the function files readable. Functions written today may be difficult to understand in a few days if the layout and variable names are illogical. For reasons of readability it is sometimes useful to split commands like

```
A=myfunction(a,b,c,d,e,f,g,h,i,j,a2,b2,c2,d2,e2,f2,g2,h2,i2,j2)
```

into several parts. This can be done in Matlab with three or more dots at the end of a line:

```
A=myfunction(a, b, c, d, e, f, g, h, i, j,  ...
             a2,b2,c2,d2,e2,f2,g2,h2,i2,j2)
```

5

Examples

The idiosyncrasies of Matlab are often best clarified by examples. Here are several examples, some of which you may find useful. For interactive examples use `demo`.

2-D plotting and a root-finder. Enter Matlab and type at the prompt:

```
>> x=linspace(0,20,25);
>> y=sin(x)+exp(-x/10);
>> plot(x,y,'ro',x,y,'g');
>> title('A simple function');
```

This makes a plot of $\sin(x) + e^{-x/10}$ on the interval $[0, 25]$. Try also `plot(x,y,'ro')` and `plot(x,y,'g')`. A smoother plot can be made with `fplot('sin(x)+exp(-x/10)',[0 25])`. From the plot you see that $\sin(x) + e^{-x/10}$ has a zero near $x = 10$. Type

```
>> hold on;
>> plot(x,zeros(size(x)),'b');
>> zoom on;
```

Now you can use the mouse in the figure window to zoom in on this zero. When you grow tired of that, double click the left mouse button to reset zooming and type `zoom off` to switch zooming mode off. As long as `hold` is `on` all subsequent plots are added to the existing figure. You want to type `hold off`. Matlab has a few functions for minimization and finding zeros of functions. With

```
>> xz=fzero('sin(x)+exp(-x/10)',[9 10])

xz =

    9.8091
```

the zero is located in a flash. Note that `fzero`'s first argument is a string.

3-D plotting. The following is a rather straightforward implementation of a 3-D plot of the function $z(x, y) := x^2 - y^2$.

```
x=-3:.5:3;
y=-2:.5:2;
[X,Y] = meshgrid(x,y);
Z = X.^2 - Y.^2;
mesh(X,Y,Z);
axis([min(x) max(x) min(y) max(y) min(min(Z))-1 max(max(Z))+1])
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
```

3D-plotting and number of function arguments.

```
function  [xx,yy,zz] = torus(r,n,a)
%TORUS Generate a torus
%      torus(r,n,a) generates a plot of a torus with central
%      radius  a  and lateral radius  r.  n  controls the number
%      of facets on the surface.  These input variables are optional
%      with defaults  r = 0.5, n = 20, a = 1.
%
%      [x,y,z] = torus(r,n,a) generates three (n+1)-by-(2n+1)
%      matrices so that surf(x,y,z) will produce the torus.
%
%      See also SPHERE, CYLINDER

%%%  Kermit Sigmon, 11-22-93
if nargin < 3, a = 1; end
if nargin < 2, n = 20; end
if nargin < 1, r = 0.5; end
theta = pi*(0:2*n)/n;
phi   = 2*pi*(0:n)'/n;
x = (a + r*cos(phi))*cos(theta);
y = (a + r*cos(phi))*sin(theta);
z = r*sin(phi)*ones(size(theta));
if nargout == 0
   surf(x,y,z)
   ar = (a + r)/sqrt(2);
   axis([-ar,ar,-ar,ar,-ar,ar])
else
   xx = x; yy = y; zz = z;
end
```

This example illustrates that functions need not have a fixed number of input arguments and output arguments. All of the following are allowed:

```
[x,y,z]=torus;    % construct torus but make no plot
[x,y]=torus;      % same as previous, but z is discarded
torus;            % make a plot instead
torus(0.6);       % make a plot with r=0.6 and default n and a
x=torus(0.7, 25); % etcetera ....
```

Type at the prompt

```
>> torus
```

and you'll will be served a colorful 3-D plot of a torus. Properties of the plot may be modified afterwards. For example, type

```
>> shading interp
>> colormap(copper)
>> axis off
```

and the torus will look more and more like a donut.

The Routh-Hurwitz test. This is a famous result about stability of polynomials. A polynomial $p_n s^n + p_{n-1}s^{n-1} + \cdots + p_0$ is called *stable* if all its zeros have negative real part. Recall that in Matlab it is customary to represent a polynomial as an array of its coefficients

p = [$p_n \quad p_{n-1} \quad \cdots \quad p_0$].

Now to test if a polynomial `p` is stable you might be compelled to compute its zeros with

```
>> roots(p)
```

A simple glance at its zeros indeed tells if `p` is stable or not, but checking stability of polynomials through computation of its zeros is like killing a bug with a shotgun. About hundred years ago the mathematicians Routh and Hurwitz devised easier techniques to test for stability of polynomials. In hindsight their results essentially boil down to the following.

Lemma 5.0.1 (Routh-Hurwitz test). *A polynomial $p_n s^n + p_{n-1}s^{n-1} + \cdots + p_0$ (with $n > 0$, $p_n \neq 0$) is stable if and only if p_n and p_{n-1} are nonzero and have the same sign, and the polynomial*

$$q(s) := p(s) - \frac{p_n}{p_{n-1}}(p_{n-1}s^n + p_{n-3}s^{n-2} + s^{n-4}p_{n-5} + \cdots)$$

is stable.

The relevance of this result is that q has degree $n-1$, i.e., 1 less than that op p, and so stability is quickly established recursively. A Matlab implementation of this idea is:

```
function stabtrue=rhtest(p)
%RHTEST   The Routh-Hurwitz stability test for polynomials.
% Usage:  stabtrue=rhtest(p)
% Out:    stabtrue=1 if p is a Hurwitz stable polynomial,
%         otherwise stabtrue=0.
ind=find(abs(p) > 0);
p(1:ind(1)-1)=[];     % Get rid of leading zeros
degree=length(p)-1;
for n=degree:-1:1     % Reduce the degree to 1
  if p(1) == 0 | sign(p(1)) ~= sign(p(2))
    stabtrue=0;
    return
  end
  eta=p(1)/p(2);
  q=p;
  q(1:2:n)=p(1:2:n)-eta*p(2:2:n+1);
  % Now q(1)=0, i.e., q is of degree less than n
  q(1)=[]; % Remove q(1) from q
  p=q;
end
stabtrue=1;
```

This function is much faster than `roots`, on the negative side, however, `rhtest` may be ill conditioned if `p` has zeros close to the imaginary axis.

A

Appendix: index of functions, commands and operations

This appendix contains lists of the functions, commands and operations that come with any distributation of Matlab. The lists were extracted from the on-line help of Matlab, version 4.2 for Unix. The commands are grouped by subject.

An installation of Matlab has next to the standard set of functions and such, also several *toolboxes*. Toolboxes are packages of m-files that concentrate on a particular set of math or engineering problems. Including all the toolboxes in this appendix would have been too much. Type `help` to see what toolboxes there are on your machine. The toolboxes that at present can be acquired from The Mathworks[1] are listed

Chemometrics	Communications
Control System	Financial Toolbox
Frequency Domain System Identification	Fuzzy Logic
Higher-Order Spectral Analysis	Image Processing
LMI Control	Mapping
Model Predictive Control	μ-Analysis and Synthesis
NAG	Neural Network
Optimization	Partial Differential Equation
QFT Control Design	Robust Control
Signal Processing	Spline
Statistics	System Identification
Symbolic/Extended Symbolic Math	Wavelet

Table A.1: The toolboxes.

in Table A.1. The list continues to grow.

Matlab is still very much in development and even if the standard commands listed below are up to date "now", they need not be in future releases. This is usually not a big problem but be aware of it and always consult the on-line help.

A.1 General purpose commands

Extracted from `help general`.

[1] See `http://www.mathworks.com/`

Managing commands and functions	
`help`	On-line documentation
`doc`	Load hypertext documentation
`what`	Directory listing of M-, MAT- and MEX-files
`type`	List M-file
`lookfor`	Keyword search through the HELP entries
`which`	Locate functions and files
`demo`	Run demos
`path`	Control MATLAB's search path

Managing variables and the workspace	
`who`	List current variables
`whos`	List current variables, long form
`load`	Retrieve variables from disk
`save`	Save workspace variables to disk
`clear`	Clear variables and functions from memory
`pack`	Consolidate workspace memory
`size`	Size of matrix
`length`	Length of vector
`disp`	Display matrix or text

Working with files and the operating system	
`cd`	Change current working directory
`dir`	Directory listing
`delete`	Delete file
`getenv`	Get environment value
`!`	Execute operating system command
`unix`	Execute operating system command & return result
`diary`	Save text of MATLAB session

Controlling the command window	
`cedit`	Set command line edit/recall facility parameters
`clc`	Clear command window
`home`	Send cursor home
`format`	Set output format
`echo`	Echo commands inside script files
`more`	Control paged output in command window

Starting and quitting from MATLAB	
`quit`	Terminate MATLAB
`startup`	M-file executed when MATLAB is invoked
`matlabrc`	Master startup M-file

General information	
`info`	Information about MATLAB and The MathWorks, Inc
`subscribe`	Become subscribing user of MATLAB
`hostid`	MATLAB server host identification number
`whatsnew`	Information about new features not yet documented
`ver`	MATLAB, SIMULINK, and TOOLBOX version information

A.2 Operators and special characters

Extracted from `help ops`.

Operators and special characters		
`+`	Plus	see `help arith`
`-`	Minus	”
`*`	Matrix multiplication	”
`.*`	Array multiplication	”
`^`	Matrix power	”
`.^`	Array power	”
`\`	Backslash or left division	see `help slash`
`/`	Slash or right division	”
`./`	Array division	”
`kron`	Kronecker tensor product	see `help kron`
`:`	Colon	see `help colon`
`( )`	Parentheses	see `help paren`
`[ ]`	Brackets	”
`.`	Decimal point	see `help punct`
`..`	Parent directory	”
`...`	Continuation	”
`,`	Comma	”
`;`	Semicolon	”
`%`	Comment	”
`!`	Exclamation point	”
`'`	Transpose and quote	”
`=`	Assignment	”
`==`	Equality	see `help relop`
`< >`	Relational operators	”
`&`	Logical AND	”
`|`	Logical OR	”
`~`	Logical NOT	”
`xor`	Logical EXCLUSIVE OR	see `help xor`

LOGICAL CHARACTERISTICS	
`exist`	Check if variables or functions are defined
`any`	True if any element of vector is true
`all`	True if all elements of vector are true
`find`	Find indices of non-zero elements
`isnan`	True for Not-A-Number
`isinf`	True for infinite elements
`finite`	True for finite elements
`isempty`	True for empty matrix
`isreal`	True for real matrix
`issparse`	True for sparse matrix
`isstr`	True for text string
`isglobal`	True for global variables

A.3 Programming language constructs

Extracted from `help lang`.

MATLAB AS A PROGRAMMING LANGUAGE	
`script`	About MATLAB scripts and M-files
`function`	Add new function
`eval`	Execute string with MATLAB expression
`feval`	Execute function specified by string
`global`	Define global variable
`nargchk`	Validate number of input arguments
`lasterr`	Last error message

CONTROL FLOW	
`if`	Conditionally execute statements
`else`	Used with IF
`elseif`	Used with IF
`end`	Terminate the scope of FOR, WHILE and IF statements
`for`	Repeat statements a specific number of times
`while`	Repeat statements an indefinite number of times
`break`	Terminate execution of loop
`return`	Return to invoking function
`error`	Display message and abort function

INTERACTIVE INPUT	
`input`	Prompt for user input
`keyboard`	Invoke keyboard as if it were a Script-file
`menu`	Generate menu of choices for user input
`pause`	Wait for user response
`uimenu`	Create user interface menu
`uicontrol`	Create user interface control

DEBUGGING COMMANDS	
`dbstop`	Set breakpoint
`dbclear`	Remove breakpoint
`dbcont`	Resume execution
`dbdown`	Change local workspace context
`dbstack`	List who called whom
`dbstatus`	List all breakpoints
`dbstep`	Execute one or more lines
`dbtype`	List M-file with line numbers
`dbup`	Change local workspace context
`dbquit`	Quit debug mode
`mexdebug`	Debug MEX-files

A.4 Elementary matrices and matrix manipulation

Extracted from `help elmat`.

ELEMENTARY MATRICES	
`zeros`	Zeros matrix
`ones`	Ones matrix
`eye`	Identity matrix
`rand`	Uniformly distributed random numbers
`randn`	Normally distributed random numbers
`linspace`	Linearly spaced vector
`logspace`	Logarithmically spaced vector
`meshgrid`	X and Y arrays for 3-D plots
`:`	Regularly spaced vector

SPECIAL VARIABLES AND CONSTANTS	
`ans`	Most recent answer
`eps`	Floating point relative accuracy
`realmax`	Largest floating point number
`realmin`	Smallest positive floating point number
`pi`	3.1415926535897.....
`i, j`	Imaginary unit
`inf`	Infinity
`NaN`	Not-a-Number
`flops`	Count of floating point operations
`nargin`	Number of function input arguments
`nargout`	Number of function output arguments
`computer`	Computer type
`isieee`	True for computers with IEEE arithmetic
`isstudent`	True for the Student Edition
`why`	Succinct answer
`version`	MATLAB version number

TIME AND DATES	
`clock`	Wall clock
`cputime`	Elapsed CPU time
`date`	Calendar
`etime`	Elapsed time function
`tic, toc`	Stopwatch timer functions

MATRIX MANIPULATION	
`diag`	Create or extract diagonals
`fliplr`	Flip matrix in the left/right direction
`flipud`	Flip matrix in the up/down direction
`reshape`	Change size
`rot90`	Rotate matrix 90 degrees
`tril`	Extract lower triangular part
`triu`	Extract upper triangular part
`:`	Index into matrix, rearrange matrix

A.5 Specialized matrices

Extracted from `help specmat`.

SPECIALIZED MATRICES	
`compan`	Companion matrix
`gallery`	Several small test matrices
`hadamard`	Hadamard matrix
`hankel`	Hankel matrix
`hilb`	Hilbert matrix
`invhilb`	Inverse Hilbert matrix
`kron`	Kronecker tensor product
`magic`	Magic square
`pascal`	Pascal matrix
`rosser`	Classic symmetric eigenvalue test problem
`toeplitz`	Toeplitz matrix
`vander`	Vandermonde matrix
`wilkinson`	Wilkinson’s eigenvalue test matrix

A.6 Elementary math functions

Extracted from `help elfun`.

TRIGONOMETRIC FUNCTIONS	
`sin`	Sine
`sinh`	Hyperbolic sine
`asin`	Inverse sine
`asinh`	Inverse hyperbolic sine
`cos`	Cosine
`cosh`	Hyperbolic cosine
`acos`	Inverse cosine
`acosh`	Inverse hyperbolic cosine
`tan`	Tangent
`tanh`	Hyperbolic tangent
`atan`	Inverse tangent
`atan2`	Four quadrant inverse tangent
`atanh`	Inverse hyperbolic tangent
`sec`	Secant
`sech`	Hyperbolic secant
`asec`	Inverse secant
`asech`	Inverse hyperbolic secant
`csc`	Cosecant
`csch`	Hyperbolic cosecant
`acsc`	Inverse cosecant
`acsch`	Inverse hyperbolic cosecant
`cot`	Cotangent
`coth`	Hyperbolic cotangent
`acot`	Inverse cotangent
`acoth`	Inverse hyperbolic cotangent

EXPONENTIAL FUCNTIONS	
`exp`	Exponential
`log`	Natural logarithm
`log10`	Common logarithm
`sqrt`	Square root

COMPLEX FUNCTIONS	
`abs`	Absolute value
`angle`	Phase angle
`conj`	Complex conjugate
`imag`	Complex imaginary part
`real`	Complex real part

NUMERIC	
`fix`	Round towards zero
`floor`	Round towards minus infinity
`ceil`	Round towards plus infinity
`round`	Round towards nearest integer
`rem`	Remainder after division
`sign`	Signum function

A.7 Specialized math functions

Extracted from `help specfun`.

SPECIALIZED MATH FUNCTIONS	
`besselj`	Bessel function of the first kind
`bessely`	Bessel function of the second kind
`besseli`	Modified Bessel function of the first kind
`besselk`	Modified Bessel function of the second kind
`beta`	Beta function
`betainc`	Incomplete beta function
`betaln`	Logarithm of beta function
`ellipj`	Jacobi elliptic functions
`ellipke`	Complete elliptic integral
`erf`	Error function
`erfc`	Complementary error function
`erfcx`	Scaled complementary error function
`erfinv`	Inverse error function
`expint`	Exponential integral function
`gamma`	Gamma function
`gcd`	Greatest common divisor
`gammainc`	Incomplete gamma function
`lcm`	Least common multiple
`legendre`	Associated Legendre function
`gammaln`	Logarithm of gamma function
`log2`	Dissect floating point numbers
`pow2`	Scale floating point numbers
`rat`	Rational approximation
`rats`	Rational output
`cart2sph`	Transform from Cartesian to spherical coordinates
`cart2pol`	Transform from Cartesian to polar coordinates
`pol2cart`	Transform from polar to Cartesian coordinates
`sph2cart`	Transform from spherical to Cartesian coordinates

A.8 Matrix functions - numerical linear algebra

Extracted from `help matfun`.

MATRIX ANALYSIS	
`cond`	Matrix condition number
`norm`	Matrix or vector norm
`rcond`	LINPACK reciprocal condition estimator
`rank`	Number of linearly independent rows or columns
`det`	Determinant
`trace`	Sum of diagonal elements
`null`	Null space
`orth`	Orthogonalization
`rref`	Reduced row echelon form

LINEAR EQUATIONS	
`\ and /`	Linear equation solution; use `help slash`
`chol`	Cholesky factorization
`lu`	Factors from Gaussian elimination
`inv`	Matrix inverse
`qr`	Orthogonal-triangular decomposition
`qrdelete`	Delete a column from the QR factorization
`qrinsert`	Insert a column in the QR factorization
`nnls`	Non-negative least-squares
`pinv`	Pseudoinverse
`lscov`	Least squares in the presence of known covariance

EIGENVALUES AND SINGULAR VALUES	
`eig`	Eigenvalues and eigenvectors
`poly`	Characteristic polynomial
`polyeig`	Polynomial eigenvalue problem
`hess`	Hessenberg form
`qz`	Generalized eigenvalues
`rsf2csf`	Real block diagonal form to complex diagonal form
`cdf2rdf`	Complex diagonal form to real block diagonal form
`schur`	Schur decomposition
`balance`	Diagonal scaling to improve eigenvalue accuracy
`svd`	Singular value decomposition

MATRIX FUNCTIONS	
`expm`	Matrix exponential
`expm1`	M-file implementation of expm
`expm2`	Matrix exponential via Taylor series
`expm3`	Matrix exponential via eigenvalues and eigenvectors
`logm`	Matrix logarithm
`sqrtm`	Matrix square root
`funm`	Evaluate general matrix function

A.9 Data analysis and Fourier transforms

Extracted from `help datafun`.

Basic operations	
`max`	Largest component
`min`	Smallest component
`mean`	Average or mean value
`median`	Median value
`std`	Standard deviation
`sort`	Sort in ascending order
`sum`	Sum of elements
`prod`	Product of elements
`cumsum`	Cumulative sum of elements
`cumprod`	Cumulative product of elements
`trapz`	Numerical integration using trapezoidal method

Finite differences	
`diff`	Difference function and approximate derivative
`gradient`	Approximate gradient
`del2`	Five-point discrete Laplacian

Vector operations	
`cross`	Vector cross product
`dot`	Vector dot product

Correlation	
`corrcoef`	Correlation coefficients
`cov`	Covariance matrix
`subspace`	Angle between subspaces

Filtering and convolution	
`filter`	One-dimensional digital filter
`filter2`	Two-dimensional digital filter
`conv`	Convolution and polynomial multiplication
`conv2`	Two-dimensional convolution
`deconv`	Deconvolution and polynomial division

Fourier transforms	
`fft`	Discrete Fourier transform
`fft2`	Two-dimensional discrete Fourier transform
`ifft`	Inverse discrete Fourier transform
`ifft2`	Two-dimensional inverse discrete Fourier transform
`abs`	Magnitude
`angle`	Phase angle
`unwrap`	Remove phase angle jumps across 360 degree boundaries
`fftshift`	Move zeroth lag to center of spectrum
`cplxpair`	Sort numbers into complex conjugate pairs
`nextpow2`	Next higher power of 2

A.10 Polynomial and interpolation functions

Extracted from `help polyfun`.

POLYNOMIALS	
`roots`	Find polynomial roots
`poly`	Construct polynomial with specified roots
`polyval`	Evaluate polynomial
`polyvalm`	Evaluate polynomial with matrix argument
`residue`	Partial-fraction expansion (residues)
`polyfit`	Fit polynomial to data
`polyder`	Differentiate polynomial
`conv`	Multiply polynomials
`deconv`	Divide polynomials

DATA INTERPOLATION	
`interp1`	1-D interpolation (1-D table lookup)
`interp2`	2-D interpolation (2-D table lookup)
`interpft`	1-D interpolation using FFT method
`griddata`	Data gridding

SPLINE INTERPOLATION	
`spline`	Cubic spline data interpolation
`ppval`	Evaluate piecewise polynomial

A.11 Graphics

Extracted from `help plotxy`, `help plotxyz`, `help graphics` and `help color`.

ELEMENTARY X-Y GRAPHS.	
`plot`	Linear plot
`loglog`	Log-log scale plot
`semilogx`	Semi-log scale plot
`semilogy`	Semi-log scale plot
`fill`	Draw filled 2-D polygons

SPECIALIZED X-Y GRAPHS	
`polar`	Polar coordinate plot
`bar`	Bar graph
`stem`	Discrete sequence or ”stem” plot
`stairs`	Stairstep plot
`errorbar`	Error bar plot
`hist`	Histogram plot
`rose`	Angle histogram plot
`compass`	Compass plot
`feather`	Feather plot
`fplot`	Plot function
`comet`	Comet-like trajectory

Graph annotation	
`title`	Graph title
`xlabel`	X-axis label
`ylabel`	Y-axis label
`text`	Text annotation
`gtext`	Mouse placement of text
`grid`	Grid lines

A.11.1 Three dimensional graphics

Line and area fill commands	
`plot3`	Plot lines and points in 3-D space
`fill3`	Draw filled 3-D polygons in 3-D space
`comet3`	3-D comet-like trajectories

Contour and other 2-D plots of 3-D data	
`contour`	Contour plot
`contour3`	3-D contour plot
`clabel`	Contour plot elevation labels
`contourc`	Contour plot computation (used by contour)
`pcolor`	Pseudocolor (checkerboard) plot
`quiver`	Quiver plot

Surface and mesh plots	
`mesh`	3-D mesh surface
`meshc`	Combination mesh/contour plot
`meshz`	3-D Mesh with zero plane
`surf`	3-D shaded surface
`surfc`	Combination surf/contour plot
`surfl`	3-D shaded surface with lighting
`waterfall`	Waterfall plot

Volume visualization	
`slice`	Volumetric visualization plots

Graph appearance	
`view`	3-D graph viewpoint specification
`viewmtx`	View transformation matrices
`hidden`	Mesh hidden line removal mode
`shading`	Color shading mode
`axis`	Axis scaling and appearance
`caxis`	Pseudocolor axis scaling
`colormap`	Color look-up table

GRAPH ANNOTATION	
`title`	Graph title
`xlabel`	X-axis label
`ylabel`	Y-axis label
`zlabel`	Z-axis label for 3-D plots
`text`	Text annotation
`gtext`	Mouse placement of text
`grid`	Grid lines

3-D OBJECTS	
`cylinder`	Generate cylinder
`sphere`	Generate sphere

A.11.2 General purpose graphics functions

FIGURE WINDOW CREATION AND CONTROL	
`figure`	Create Figure (graph window)
`gcf`	Get handle to current figure
`clf`	Clear current figure
`close`	Close figure

AXIS CREATION AND CONTROL	
`subplot`	Create axes in tiled positions
`axes`	Create axes in arbitrary positions
`gca`	Get handle to current axes
`cla`	Clear current axes
`axis`	Control axis scaling and appearance
`caxis`	Control pseudocolor axis scaling
`hold`	Hold current graph

HANDLE GRAPHICS OBJECTS	
`figure`	Create figure window
`axes`	Create axes
`line`	Create line
`text`	Create text
`patch`	Create patch
`surface`	Create surface
`image`	Create image
`uicontrol`	Create user interface control
`uimenu`	Create user interface menu

HARDCOPY AND STORAGE	
`print`	Print graph or save graph to file
`printopt`	Configure local printer defaults
`orient`	Set paper orientation
`capture`	Screen capture of current figure

MOVIES AND ANIMATION	
`moviein`	Initialize movie frame memory
`getframe`	Get movie frame
`movie`	Play recorded movie frames

MISCELLANEOUS	
`ginput`	Graphical input from mouse
`ishold`	Return hold state
`graymon`	Set graphics window defaults for gray-scale monitors
`rbbox`	Rubberband box
`rotate`	Rotate an object about a specified direction
`terminal`	Set graphics terminal type
`uiputfile`	Put up dialog box for saving files
`uigetfile`	Put up dialog box which queries for file names
`whitebg`	Set graphics window defaults for white background
`zoom`	Zoom in and out on a 2-D plot
`waitforbuttonpress`	Wait for key/buttonpress over figure

A.11.3 Color control and lighting model functions

COLOR CONTROLS	
`colormap`	Color look-up table
`caxis`	Pseudocolor axis scaling
`shading`	Color shading mode

COLOR MAPS	
`hsv`	Hue-saturation-value color map
`gray`	Linear gray-scale color map
`hot`	Black-red-yellow-white color map
`cool`	Shades of cyan and magenta color map
`bone`	Gray-scale with a tinge of blue color map
`copper`	Linear copper-tone color map
`pink`	Pastel shades of pink color map
`prism`	Prism color map
`jet`	A variant of HSV
`flag`	Alternating red, white, blue, and black color map

COLOR MAP RELATED FUNCTIONS	
`colorbar`	Display color bar (color scale)
`hsv2rgb`	Hue-saturation-value to red-green-blue conversion
`rgb2hsv`	Red-green-blue to hue-saturation-value conversion
`contrast`	Gray scale color map to enhance image contrast
`brighten`	Brighten or darken color map
`spinmap`	Spin color map
`rgbplot`	Plot color map

Lighting models	
`surfl`	3-D shaded surface with lighting
`specular`	Specular reflectance
`diffuse`	Diffuse reflectance
`surfnorm`	Surface normals

A.12 Sound processing functions

Extracted from `help sounds`.

General sound functions	
`sound`	Convert vector into sound
`saxis`	Sound axis scaling

Computer-specific sound functions	
`auwrite`	Write mu-law encloded audio file
`auread`	Read mu-law encloded audio file
`wavwrite`	Write MS Windows .WAV audio file
`wavread`	Read MS Windows .WAV audio file
`mu2lin`	Mu-law to linear conversion
`lin2mu`	Linear to mu-law conversion

A.13 Character string functions

Extracted from `help strfun`.

General	
`strings`	About character strings in MATLAB
`abs`	Convert string to numeric values
`setstr`	Convert numeric values to string
`isstr`	True for string
`blanks`	String of blanks
`deblank`	Remove trailing blanks
`str2mat`	Form text matrix from individual strings
`eval`	Execute string with MATLAB expression

String comparison	
`strcmp`	Compare strings
`findstr`	Find one string within another
`upper`	Convert string to uppercase
`lower`	Convert string to lowercase
`isletter`	True for letters of the alphabet
`isspace`	True for white space characters
`strrep`	Replace a string with another
`strtok`	Find a token in a string

STRING TO NUMBER CONVERSION	
`num2str`	Convert number to string
`int2str`	Convert integer to string
`str2num`	Convert string to number
`mat2str`	Convert matrix to string
`sprintf`	Convert number to string under format control
`sscanf`	Convert string to number under format control

HEXADECIMAL TO NUMBER CONVERSION	
`hex2num`	Convert hex string to IEEE floating point number
`hex2dec`	Convert hex string to decimal integer
`dec2hex`	Convert decimal integer to hex string

A.14 Low-level file I/O functions

Extracted from `help iofun`.

FILE OPENING AND CLOSING	
`fopen`	Open file
`fclose`	Close file

UNFORMATTED I/O	
`fread`	Read binary data from file
`fwrite`	Write binary data to file

FORMATTED I/O	
`fscanf`	Read formatted data from file
`fprintf`	Write formatted data to file
`fgetl`	Read line from file, discard newline character
`fgets`	Read line from file, keep newline character

FILE POSITIONING	
`ferror`	Inquire file I/O error status
`feof`	Test for end-of-file
`fseek`	Set file position indicator
`ftell`	Get file position indicator
`frewind`	Rewind file

STRING CONVERSION	
`sprintf`	Write formatted data to string
`sscanf`	Read string under format control

WK1 FORMAT	
`wk1const`	WK1 record definitions
`wk1read`	Read WK1 file/range
`wk1write`	Write out matrix in a WK1 formatted file
`wk1wrec`	Write a WK1 record header

CSV Format	
`csvread`	Read Comma Separated Value formatted file into a matrix
`csvwrite`	Write out matrix in a CSV formatted file

ASCII Delimited Format	
`dlmread`	Read ASCII delimited file into a matrix
`dlmwrite`	Write out matrix in ASCII delimited file format

Bibliography

[1] E. Gkioulekas. Programming with matlab. Tutorial, available at `http://www.amath.washington.edu/~elf/tutorials/`, 1996.

[2] Tim Love. Using matlab 4. available at `http://www.math.washington.edu/~burke/crs/515/`, 1996.

[3] The MathWorks Inc. *The Student Edition of MATLAB*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[4] K. Sigmon. Matlab primer. available at `ftp://ftp.math.ufl.edu/pub/matlab/`, 1993.